Neural Networks in Cognitive Science

Jeff Yoshimi, Zoë Tosi, Scott Hotton, Pierre Beckmann, Ellis Cain
Chelsea Gordon, David C. Noelle, Tim Meyer

Version 2024.1.1

Contents

Preface

This book was written by a group of researchers associated with UC Merced's Cognitive and Information Sciences Program (`http://cogsci.ucmerced.edu/`) to support learning about neural networks in a visual and interactive way. It is intended to be used in conjunction with Simbrain (`http://www.simbrain.net`), a free open source software package that makes it easy to build neural network simulations.[1] The philosophy behind the book is that it is possible to learn about neural networks even with minimal mathematical background, and that this is facilitated by the use of a visual simulation environment like Simbrain.

Though little mathematical background is assumed, there is a lot of mathematical detail in the book. Most of these details are included in lengthy footnotes. We have not shied away from heavy use of footnotes, since they provide a convenient way of providing additional layers of information independently from the main text.

Currently the book focuses on neural networks specifically in cognitive science, and to a lesser extent neuroscience. Of course today neural networks are best known as a tool in machine learning, and in particular *deep learning*. The book does provide some background relevant to machine learning uses of neural networks, but that is not its current focus. Given the modular nature of this book, it may develop in a way that encompasses these uses of neural networks, but it does not do so at present.

Several other sources written in the same spirit as this book should be mentioned: Randall O'Reilly and Yuko Munakata's *Computational Cognitive Neuroscience* book, and associated materials, which are based on the free, open source Emergent simulation platform[2], as well as several sources that provide more guidance on deep learning and machine learning, with the assistance of interactive tools and visualizations, some of which run directly in the browser.[3]

This book is meant to be improved, corrected, and expanded on a regular basis, and hopefully, remixed and remastered by others.[4] If you fix or improve something, please submit a pull request, and if you have suggestions, post an issue on the github repository, which is here: `https://github.com/simbrain/NeuralNetworksCogSciBook`. The plan is to have regular releases each year. Hence, the year-based versioning, e.g. version 2022.1, 2022.2, etc.

To support this flexibility, custom scripts and LaTeXcommands are provided. The most complete version of the book (the "master document") is hosted on the github repository. Those chapters can be combined and remixed in your own "container" documents that only contain the information you need for a particular use (*e.g.* for a particular class you are teaching). You can fork the repository and create your own container documents containing whichever chapters you like, including new material of your own, or adaptations of existing chapters. Guidelines for assembling your own container documents, and for producing new chapters (which we hope you will share with us!), are in the readme document of the github repo, which can be found just by scrolling to the bottom of `https://github.com/simbrain/NeuralNetworksCogSciBook`.

All glossary items are listed in **bold face**. For any bold faced glossary item there should be a corresponding entry in the glossary in the back of the book.

Chapter authors are listed in the order of their contribution to that chapter. Authors listed on the front cover of a container document are ordered by the weighted sum of their contributions to the chapters in that

[1] For more information see the online documentation at `http://www.simbrain.net/Documentation/v3/SimbrainDocs.html`, the Simbrain youtube channel or search #Simbrain on twitter (sort by "latest").

[2] `https://compcogneuro.org/`

[3] See `http://neuralnetworksanddeeplearning.com/` and the articles at `https://distill.pub/`, as well as `https://playground.tensorflow.org/`. Also see `https://www.tensorflow.org/tensorboard/get_started` and Jay McClelland's Matlab-based course: `https://web.stanford.edu/group/pdplab/pdphandbook/`

[4] Many issues and plans for improvement are included as comments in the latex documents, which you are welcome to peruse.

container document. Author orderings are produced using a python script included in the repository.

When references have a "*" symbol attached to them, it means that they refer to a chapter, section, or figure in the master document but not that container document. The master document is a kind of global container document hosted at the main github repository, that contains all chapters known to the original team.

The first version of this book was written by Jeff Yoshimi, as was the infrastructure to support it. Graphics support was provided by Pamela Payne, Elizabeth Reagh, and Soraya Boza (credits for individual figures are listed at the end of the document). David Cuesta, Eric Schwitzgebel, Eric Thomson reviewed several chapters in 2024, and additional feedback was provided by Shervin Nosrati, Kate Totter, Julia Ton, and Matthew Lloyd. Sergio Ponce de Leon reviewed several chapters in 2022. Liza Oh reviewed several chapters in Fall 2021. Tim Meyer helped review and edit the Spring 2017 and Fall 2017 versions of the manuscript. Sharai Wilson provided a great deal of help with the manuscript in Summer 2017. Every time the course is taught students and teaching assistants provide valuable feedback, going back to 2006 (Spring term of the year UC Merced opened, and the first time an earlier version of this text was used). Ricardo Velasco helped with many aspects of producing the first versions of this text in the 2000s.

As noted above, the book is closely tied to a separate open source project, Simbrain. Simbrain credits are here: `http://simbrain.net/SimbrainCredits.html`.

Chapter 1

Introduction

Jeff Yoshimi

The phrase "neural network" has several meanings. A **biological neural network** is an actual set of interconnected neurons in an animal brain. Fig. 1.1 (left) shows a biological neural network. "Neural network" can also mean **artificial neural network** (or "ANN"), that is, a computer model that has certain things in common with biological neural networks. Fig. 1.1 (right) shows an artificial neural network. It has "nodes" and "weights" that are analogous to the neurons and synapses of a biological neural network. We focus on artificial neural networks in this book, and when we refer to "neural networks" we usually mean artificial neural networks.

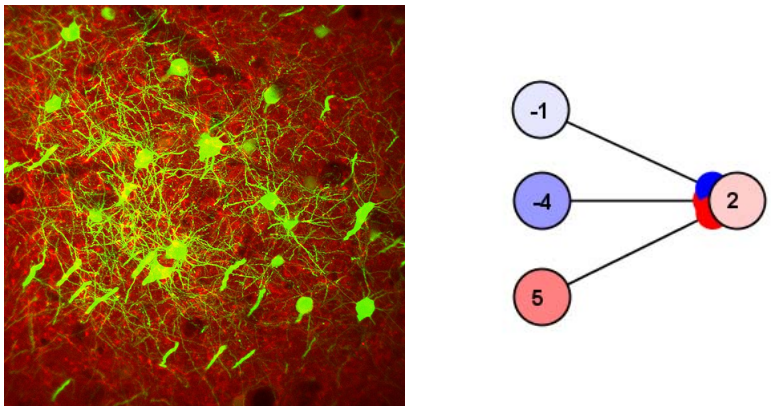


Figure 1.1: Left: A biological neural network. Right: A simple artificial neural network built in Simbrain, with three nodes connecting to another node via three weights.

Neural networks can be made to do many fascinating things. They can, for example, drive cars, forecast weather patterns, recognize faces in images, and play the game Go at championship levels. Recently, they have become eerily good at producing human-level written text and images (see the discussion of GPT-3 in chapter 16 or search the web for images produced by Dall-E). They have been used to model the brain at all of its levels, from individual neurons up to the entire brain. They have been used to model cognition in all of its forms, including memory, perception, categorization, language, and attention. In this chapter, we give a general introduction to neural networks, and survey some of these different ways they are used.[1]

[1]Some time in the 2010s or 2020s, it became standard to refer to neural networks as a form of "AI" or "Artificial Intelligence." From this standpoint, "AI" is a covering term that includes all the many forms of computer simulation that attempt to behave in an intelligent and often human-like manner. In the earlier literature "AI" was used to refer to more classical, symbolic forms of artificially intelligent system, and in that era there was a kind of battle between AI and neural networks. Some of this history is covered in section 3.3. The term "AI" is mostly avoided in this text.

1.1 Structure of Neural Networks

In figure 1.2 a simple neural network is shown with some of its parts labeled. In this section, we review the parts of neural networks (nodes and weights), the ways they can be structured (their topology), and the relationship between a network and its environment. In each case, bear in mind that what precisely the concepts mean depends on the type of model we are dealing with.[2]

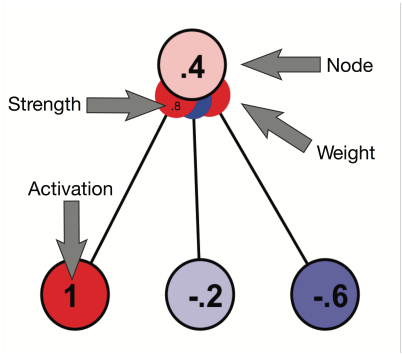


Figure 1.2: Nodes and their activations; weights and their strengths.

In the figure, a circle with a number in it is an artificial neuron or **node** (nodes are also referred to as "units").[3] The number inside a circle corresponds to that node's **activation**. In Simbrain, red corresponds to an "active" neuron (activation greater than 0), and how deep the red is corresponds to how close the activation is to its maximum value. Blue corresponds to an "inhibited" neuron (activation less than 0), and how deep the blue is corresponds to how close the activation is to its minimum value. White corresponds to an inactive neuron (activation equals 0). In a computational neuroscience model, these activations might represent the firing rate or membrane potential of a real neuron. In a psychological model, the number might represent the presence of an item in working memory, or the strength of an unconscious belief. As we will see, there is a great deal of variance in what concepts such as activation are taken to mean.

The lines with filled disks at the end of them are artificial synaptic connections or **weight**s. These correspond to connections between nodes, which control how activation flows through a network. The weights have a value, a **strength**. The larger the absolute value of a weight strength, the "stronger" it is. 2 is a stronger positive weight than 1 is, and -2 is a stronger negative weight than -1 is. Thus, to strengthen a weight is to increase its absolute value, and to weaken it is to reduce its absolute value. Stronger weights are shown as larger disks in Simbrain. The actual weight strength can be seen by hovering over the weights or double clicking on them. As activation flows through a network, the weights with a positive strength (the red weights in Simbrain) tend to enhance activation, and the weights with a negative strength (the blue weights) tend to reduce activation.[4] In neuroscience terms, these correspond to excitatory and inhibitory synapses (and the neurons at the source of these synapses called excitatory and inhibitory neurons). As you play with simulations in Simbrain and study the chapters to come, you will begin to get a feel for how different kinds of weights have different kinds of impacts on the flow of activation in a network.

What weight strength represents depends on what kind of model we are dealing with (see chapter 2). In a computational neuroscience model, it would represent **synaptic efficacy**–roughly speaking the impact a pre-synaptic neuron can have on a post-synaptic neuron after the pre-synaptic neuron fires an action potential. In a connectionist or psychological model, the strength might represent an association between concepts. In a machine learning model, there might be no direct interpretation of weight strengths at all: they are mere parameters in a statistical model that does something useful, like recognize faces in images.

[2]Neural networks can be used for engineering, to model the brain, or to model the mind. These different uses are discussed in chapter 2. Depending on the way a network is being used, the way its parts are interpreted differs, as we'll see.

[3]Sometimes, when the context makes it clear that we are talking about an artificial neural network, terms like "neuron" and "synapse" are used to mean artificial neuron (i.e. node) or artificial synapse (i.e. weight).

[4]For more details on the graphic representation see `http://www.simbrain.net/Documentation/v3/Pages/Network/visualConventions.html`.

Node activations change in accordance with *activation functions* (discussed in chapter 5), and weight strengths change in accordance with different types of learning algorithms discussed throughout the book. Indeed, how to train neural networks is one of the most fundamental topics in the field, and is the focus of several chapters, including chapters 9, 12 and 13.

In some models a node can also produce a **spike**, which is a discrete event that corresponds to the action potential of a neuron.[5] Spiking neurons have their own rules and structures, which we discuss in chapter 18.

Nodes and weights are the basic parts of a neural network. Together, they form a network structure or mathematically, a graph[6], where the nodes correspond to vertices, and the weights correspond to edges.[7] The graph-structure formed by a network's nodes and weights is the network's **topology**. At a first pass, neural network topologies fall into one of two rough types shown in Fig. 1.3: feed-forward and recurrent.

A **feed-forward network** is a sequence of *layers* of unconnected neurons stacked on top of each other such that each layer is fully connected to the next one in the sequence (each node in one layer sends a connection to every node in the next layer).[8] Activity in this kind of network flows from an *input layer* through a sequence of *hidden layers*, and then to an *output layer*. Sometimes there are no hidden layers and we have a feed-forward network that connects directly from an input layer to an output layer.

In a feed-forward network activity simply passes through the node layers; when activation is added to the input nodes and the network is updated, that activation flows from layer to layer and is then erased. Feed-forward networks are often classifiers: an input (which might represent an image, or a smell) passes through the layers of the network and the output activations then represent a way of classifying the input (saying who is in the image, or what object is being smelled).

In a feed-forward network we can distinguish between a **node layer** and a **weight layer**.[9] We will take node layers in a feedforward network to be collections of nodes that are treated as a group, and weight layers to be collections of weights that connect node layers. Often these are represented using vectors and matrices, respectively (see chapter 6). The network in Fig. 1.3 has three node layers (labelled "input layer", "hidden layer", and "output layer") and two weight layers (labelled "1-2" and "2-3").[10] When used without qualification, we use the term "layer" to mean "node layer".[11]

We can also distinguish between the "representational depth" and "representational width" of a feed-forward network. We take the **representational depth** of network to correspond to how many layers or layer-like structures the network has. Thus a "deep" network is one with many node and weight layers. The **representational width** of a given node layer corresponds to how many nodes it has.[12] We will see throughout the book that these two concepts provide distinct ways of understanding the representational capacities of neural networks. A wider layer can develop more sophisticated representations of its inputs. A network with more depth can develop representations that combine features of other representations, so that we get increasingly complex "representations of representations." The layers of a network trained to recognized images can go from representing lines, to sets of lines (that is, shapes), to sets of shapes, etc (see the discussion of Selfridge in chapter 3, and of convolutional networks in chapter 14). Analogues of these intuitive concepts of depth and width persist in more complex types of networks, like transformers (chapter

[5] A spike is represented in Simbrain by a node and all its outgoing connections turning yellow. For an illustration of a spiking node and how it looks in Simbrain, see `http://www.simbrain.net/Documentation/v3/Pages/Network/neuron.html`.

[6] `https://en.wikipedia.org/wiki/Graph_(discrete_mathematics)`.

[7] A neural network is a special type of graph: a vertex-labeled, edge-labeled, directed graph. This means that the edges between nodes have a direction (the graph is *directed*), and that numbers are associated with the vertices and edges (it is *vertex-labeled* and *edge-labeled*).

[8] In graph-theoretic terms, such a network is a directed, acyclic, multipartite graph. It is is *acyclic* because there are no *cycles*; there is is no way to "move" from one vertex back to itself along a sequence of vertices connected by edges. It is *multi-partite* because the vertices can be partitioned into *independent sets* (node layers), within which none of the vertices are connected. When such a network is not fully-connected from one layer to the next, it is still often referred to as "feed-forward". In some cases weights skip over a layer (e.g. go straight from input to output despite the presence of a hidden layer) and again, since activity will still flow "forward", this will be referred to as a feed-forward network.

[9] The distinctions introduced in this paragraph and the next one are stipulations made to organize the material in a coherent way. Terms like node layer, weight layer are not standard, but help organize the material in this book.

[10] To make matters even more confusing sometimes the input node layer is *not* counted as a layer.

[11] In Simbrain, node and weight layers are both represented as groups, indicated by the yellow interaction boxes: `http://www.simbrain.net/Documentation/v3/Pages/Network/groups.html`.

[12] While the concept of depth is fairly common, "width" as a named concept is not. Be aware that "width" and "depth" are also used to refer to the shapes of tensors (see section 6.9) but the meaning is different there, and we assume our meaning is clear in context (for example, we sometimes add "representational" to make it clear we mean these concepts, and not tensor shape concepts).

17), where we can contrast the number of heads in a given block (width) with the number of blocks that are stacked on top of each other (depth).

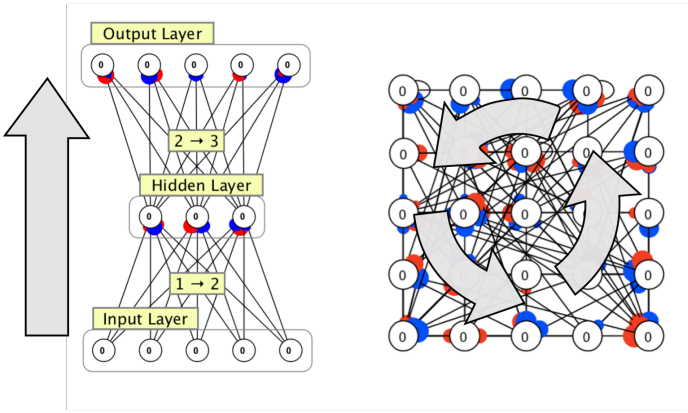


Figure 1.3: Feed-forward network (left) and recurrent network (right). Gray arrows give a sense of how activation flows through them. The feed–forward network has 3 node layers (a representational "depth" of 3, which is not deep; this is not a "deep network"), and the layers have representational "widths" of 5, 3, and 5.

In a **recurrent network**, the nodes are interconnected in such a way that activity can flow in repeating cycles.[13] Recurrent networks display complex dynamical behaviors that don't occur in feed-forward networks since activity in the network cannot always "leave" the network. Most biological neural networks are recurrent. In machine learning, recurrent networks can be used to simulate dynamical processes, for example, to mimic human speech, or create artificial music.

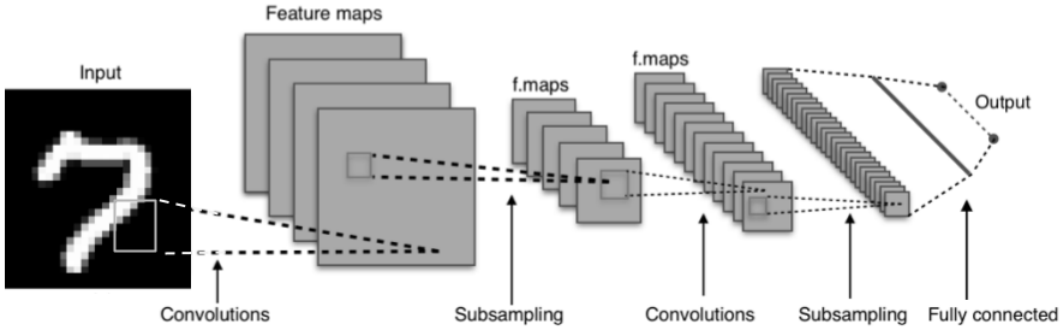


Figure 1.4: A deep neural network, trained to recognize images. The convolutional layers scan over the inputs they are linked to. Note that individual nodes and connections are not visible in this image.

The distinction between feedforward and recurrent networks is a useful first pass way to organize the field. However, more complex architectures are possible, and since the 2010s two specific architectures have been especially common: convolutional networks and transformers. There is not space to go into these structures here, but we discussion them in chapters 14 and 17. These networks operate on data structures more complex than arrays of numbers, for example on 3d arrays or volumes of numbers, which are a kind of tensor (tensors are discussion in section 6.9). An example of a convolutional network is shown in figure 1.4.

[13] In graph-theoretic terms, this is a cyclic graph, which contains at least one cycle. Recall from footnote 8 that a directed cycle is a sequence of directed edges that begin and end at the same vertex. That is, starting at one node of such a network, we can "travel" from one node to another via the connections and end up back where we started.

Networks like this are so large that we can't represent each node and weight separately, and the concept of a layer becomes more complex as more structure and processing is packed into each layer. But the basic concepts of feedforward and recurrent networks that structure the first part of this book still are useful for understanding these more complex forms of neural network.

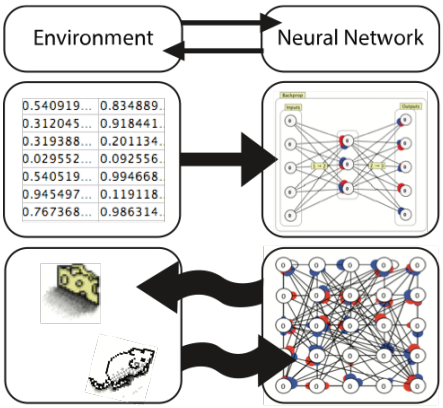


Figure 1.5: The relationship between a neural network and an environment. An "environment" is often something as simple as a table of values (middle row). However it can also be something more complex, like a virtual world (bottom row).

Networks almost always exist in some kind of **environment**, which gathers inputs for a network and receives its outputs. In cognitive science, the environment of our brain's neural network is the literal environment around us. In some simulations–for example, reinforcement learning simulations–simulated environments are used for neural networks. But if we take environment in a broader sense to simply be whatever it is that produces inputs to an artificial neural network and whatever deals with the outputs, then we capture a broader range of cases.[14] For example, a neural network that converts speech to text can be connected to audio sensors, like the microphone on your phone. It can take audio in, convert it to text, and send the result out via the speakers. By far the most common way a neural network is linked to inputs and outputs, especially when building and testing them, is via tables of data. Training and testing datasets are discussed at length in chapter 7. In Simbrain, we will also link neural networks to virtual environments. Figure 1.5 shows how some of these configurations might look. Couplings between a network and an environment occur at special nodes: an **input node** is influenced by the environment, while an **output node** exerts an influence on the environment. In figure 1.3 (left), for example, the input nodes are the nodes in the input layer, and the output nodes are the nodes in the output layer.[15]

1.2 Computation in Neural Networks

We've seen what the parts of a neural network are, and learned some basic concepts relating to their structure. We now turn to a few concrete examples in Simbrain that give a sense of how computation works in neural networks. We first develop a basic intuition for how they channel information, and then contrast this with computation in a classical computational system.

1.2.1 Performance vs. Learning

To develop an intuition for how neural networks "channel information" we can break the issue into two parts: performance and learning. Performance corresponds to how information flows through a network, given a fixed pattern of connectivity. Learning corresponds to how the pattern of connectivity is adjusted to make

[14]Here again we are using "environment" in a technical sense that does not track all standard usage, but that is useful for organizing the material and that also draws helpful conceptual connections.

[15]Information on how to couple nodes to an environment in a Simbrain simulation, and thus treat them as input or output nodes, is available here: `http://www.simbrain.net/Documentation/v3/Pages/Workspace/Couplings.html`.

the neural network do things more effectively. These are two different types of dynamics–a fast dynamics of performance or action and a slower dynamics of learning–and it is important to get a feel for them right from the start.

To understand **performance**, consider a network that has already been trained and ask how different inputs propagate through it. That is, we don't change its weights but only change its inputs to see how it responds, given its weights. When you use ChatGPT, you are using a network whose weights have already been set. You are using the final product of a long training process. You write text prompts, they propagate through a bunch of nodes, and new words are generated, one word at a time. When you talk to a grown person, something similar is happening. They are not learning (much) over the course of a few seconds, but their internal networks are reacting to what you say.

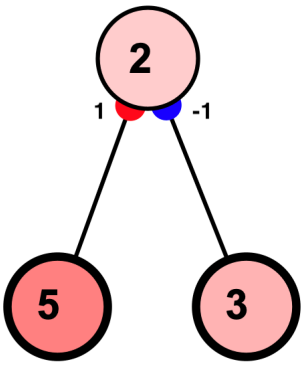


Figure 1.6: A simple feed-forward network with clamped inputs, that can be used to develop an intuition for how performance and learning work. The nodes are linear nodes that take a weighted sum of inputs. The weight strengths are 1 and -1 so the output is $5 \times 1 + 3 \times -1 = 5 - 3 = 2$. By adjusting the inputs up and down, we can make the output activation be whatever we want within the upper and lower bounds of a node (performance). Or, for a fixed pattern of input, we can make the output be whatever we want it to be by changing the weights.

How does this work? Consider the network shown in figure 1.6: it's a simple feed-forward network with two input nodes and one output node. The output node's activation is obtained by multiplying each input activation by the strength of the intervening weight and adding these products together.[16] The inputs are **clamped node**s, and are represented with a bold outline. This means they will not change their value when we update the network, unless we manually update them.[17] The weight strengths are 1 and -1, and so the output is $5 \times 1 + 3 \times -1 = 5 - 3 = 2$ By adjusting the two inputs up and down, we can get the output to be whatever we want. What if we want the output to be 5? We can do that by bumping the left "excitatory" input up to 8, so that the output is $8 \times 1 + 3 \times -1 = 8 - 3 = 5$ or leaving the left input 5 and reducing the right "inhibitory" input to 0, so that it stops inhibiting and the output is just $5 \times 1 = 5$.[18] Now try some more examples. How could you adjust the excitatory and inhibitory inputs to get an output of 0, 10, -4, or 2.5?

If you have built this network in Simbrain (which is quite easy to do), try pressing the play button and adjusting the inputs up and down to see how this works. It's just adding weighted inputs together. What could be easier? At bottom, the same thing is happening even in a giant neural network: each node is taking a weighted sum of its inputs and in this way activation propagates through the network. Or try this as an

[16]This is an example of a linear activation function. See chapter 5. Often the weighted sum is transformed or processed in further ways. Also note that we are using whole numbers just to ease into the idea, but neural networks generally use floating small point numbers between 0 and 1, like .84 or -.23.

[17]If we don't clamp them, then when we update the network they will immediately go to zero, because they have no environment; there are no inputs to them.

[18]In the brain we can speak of excitatory and inhibitory neurons (see chapter 4) because the axons extending out from a neuron lead either to all inhibitory synapses or all excitatory synapses. However, in an artificial neural network, a node can fan-out to a mix of positive and negative weights, so the terms 'excitatory' and 'inhibitory' nodes may not always apply. But in this case those terms can be used.

exercise: think of an activation you'd like to see in either the left or right network, and then start adjusting the inputs up and down until you get what are after.

We now consider **learning**, where a network's weights and other parameters are changed in a way that changes its performance. When a network fails to perform the way what we want it to, it must learn. And so we train neural networks. To get a feel for how this works, we can return to our simple network, and this time instead of adjusting the inputs, we can adjust the weights, to see how this changes the way information flows through it.[19] Given that both inputs are positive, we can think of positive weights as enhancing inputs and negative weights as diminishing inputs. That is, in this case:

1. Positive weights (red disks) increase or excite outputs. They "heat things up." As they are strengthened, the output gets larger.

2. Negative weights (blue disks) decrease or inhibit outputs. They "cool things down." As they are strengthened (as their absolute value is increased), the output gets smaller.

The two weights are like two knobs that we can turn up or down, that we can use to *tune* the network's response to a set of inputs. Suppose we want the output to be some other value besides 2, like 7, but we do *not* want to change the inputs. We want the current inputs to produce a 7 rather than 2. To do this, we could strengthen the positive weight so that it makes the output larger. In particular, we could increase it to be 2, so that we get $5 \times 2 + 3 \times -1 = 10 - 3 = 7$. Or what if we wanted to get an output of 1? In that case, we could leave the positive weight at 2 and strengthen the negative weight from -1 to -2 so that the output is now $5 \times 2 + 3 \times -3 = 10 - 9 = 1$.[20] With these two knobs we can get the network to do pretty much whatever we want. Most of the theory of neural networks is about automatically tuning the weights and other parameters of a network–often many thousands, millions, or more of them–to get them to channel information in a useful way.

1.2.2 The Three Object Detector

An example that illustrates both performance and learning is the simulation *threeObjectsDist.zip*. A screen shot of the network, which we call the "three object detector", is shown in Fig. 1.7.[21] The three object detector has a feed-forward topology with three nodes in the input layer, seven nodes in the hidden layer, and three nodes in the output layer.[22] In this example we also see how a network can be linked to a virtual environment. The mouse on the right of figure 1.7 is hooked up to this network. When the mouse is moved around, the activation in the input nodes changes. This simulates the way odor molecules impact the inner lining of the nose, causing sensory neurons to fire at different levels. So the input layer is a kind of simulated nose.

Let's first consider its performance. Press play and move the mouse around. (As you do, the weights are not changing). When we move the mouse close to an object, the corresponding output responds. When we move it away, it decays back to zero. This is called a **classification task**. It is performing well. It responds to each object by activating the appropriate node and no other.

However, this did not just happen by luck. We had to train the network. To see this, try pressing `w` and then `r`, to randomize all the weights. That more or less destroys what it has learned. Now press play and move the mouse around again. It can no longer classify inputs. But if you double click the interaction box labeled `Backprop` and hit the play button in the resulting diaolog, it quickly learns to perform the

[19] Note that we ignore negative activations for now because they can be difficult to reason about (when you change a weight from -1 to -2, are you strengthening it or turning it down?) In fact, in some contexts negative activations are taken to be unrealistic or problematic. Neuron spiking rates are always positive, for example. In recent years the "relu" activation function, which disallows negative activations, has become extremely popular in deep learning.

[20] In the first case, we could also weaken the negative weight so that it inhibits the output less. In the second case, we could also weaken the positive weight so it excites the output less.

[21] A video about the three object detector (including information about how to load it) is available at `https://youtu.be/yYzUmcPaurI?t=380`.

[22] The example is not meant to model the brain directly. It is more abstract: it classifies inputs in a brain-*like* way. It takes a pattern of inputs, and transforms those inputs through a network of connections. This is similar to the way information processing occurs in the brain. But it is not a realistic simulation of a brain circuit (as we will see, it is a "connectionist network" as opposed to a "computational neuroscience" model; it is also similar to how computations are done in engineering applications).

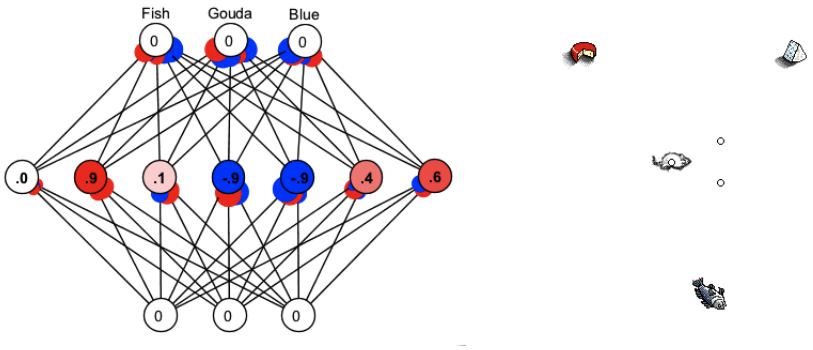


Figure 1.7: Simple feed-forward network that recognizes three objects.

classification task again. Error drops to zero, and it does what we want. This shows on a small scale how most neural networks are trained.

1.2.3 Neural vs. Symbolic Computation

In the history of cognitive science, one of the earliest debates was between those who thought human information processing was more like what occurs in a classical digital computer–that is, rule-governed manipulation of symbols–and those who thought it was more like what happens in the brain, where patterns of activation are transformed as they propagate through networks of synaptic connections (see section 3.3).[23] We won't go into the details of this debate here, but we will highlight a few prominent differences between the way a digital computer processes information and the way neural networks do, because they help to see what is distinctive about neural networks.

We can use the 3-object detector to illustrate these contrasts.In a classical computer discrete symbols (comprised of strings of 0s and 1s, or "bits") are operated on by rules, in a sequential manner. Bits of information are placed in registers on a computer's central processing unit (CPU), and logical rules in the CPU's instruction set are applied to these bits. Computers are hand-programmed to do useful things. The inside of a CPU and the memory systems on a computer are carefully controlled environments. They do not do well with noisy signals or damage. Computation in a neural network is different. Neural computation is not based on sequential, rule-based operations on bits, but on parallel operations where patterns of node activations are transformed by weight strengths.[24] Neural networks are also more tolerant of noisy signals and damage than digital computers are. Moreover, they are not programmed, in the way a computer is, but are trained, in something like the way a human child is.

Let's use the three object detector simulation to consider some of these contrasts in more detail.

Networks are *trained* via **learning**, not programmed. We show the network what we want it to do, and it learns to do it. In the three object detector, for example, here is (very roughly) what happened: we put the mouse near the fish, and said, "when you smell something like this, fire your first node." Then we did the same thing with the Gouda and blue cheese. Each time we exposed the network to an object, we used the "backprop" algorithm (discussed in a chapter 12) to adjust the network's weights. At first the network made mistakes, but with each exposure to an object the weights were changed a little, and over time it got better and better at recognizing cheese, in something like the way humans and animals gradually get better at doing things with training. This is called **supervised learning**, since we know the correct output for each input and can tell the network exactly what output it should produce for any given input. The great thing about this is that once we've trained the network on some data, **generalization** is possible, where it can deal with new data it's never been exposed to before. We can train a network to respond to a bunch

[23] Of course, neural network simulations are usually run on a traditional computer performing classical computations. But that is a convenient way of implementing the formal structure of a neural network. These implementation can still take advantage of all the special properties of neural networks. Moreover, it is in fact possible to implement neural networks directly on hardware.

[24] We can often represent this as a transformation of *activation vectors* (lists of activation values) by *weight matrices* (tables representing weight strengths). So, while the basic formalism of classical computation is logic, the basic formalism of neural networks is *linear algebra*, which we study in chapter 6.

of cheese we have available, and on that basis it can recognize new pieces of cheese it's never seen before. This is part of what makes neural networks–both the one's used in your cell phones but also the one that is inside your skull right now helping you read this–so valuable. After a bit of training and learning, they can be let loose in the world and deal with brand new situations.

This isn't the only way neural networks can be trained. For example, neural networks can also learn by a system of rewards and punishments (reinforcement learning). They can also learn without any kind of training signal or reinforcement, simply by picking up on the statistical structure of their environment (**unsupervised learning**).[25]

Second, neural networks emphasize **parallel processing**. Whereas digital computers normally do things one at a time, in a sequence, neural networks do a lot of things *at the same time*. To see the difference this can make, consider a simple problem: finding which of ten cups has a jelly bean under it. A serial approach would lift each cup up, one at a time, until the jelly bean was found. A parallel approach would lift all ten cups up at once. Neural networks operate like this, processing information in all the nodes, all at once, all the time. This is easy to see in the three object detector simulation: when you run the simulation and drag the mouse around, the activations of all the nodes will change based on the new inputs in parallel.[26]

Third, neural networks experience **graceful degradation** when they are damaged (this is also called "fault tolerance"). They are not brittle in the way a digital computer is. You can start deleting the weights of a neural network and it will still work reasonably well. You can try doing this on the three object detector! In a similar way, if you lose a few neurons and /or synapses, you will be just fine. Of course, if you lose enough neurons and synapses it will start to show, but it will happen gradually and proportionally to the damage. It is in this sense that neural networks degrade "gracefully."[27] A digital computer, by contrast, is not designed to continue functioning if its components are damaged. Pluck a micro-chip out of the motherboard, or snip a few wires, and there's a good chance your computer will stop working altogether.[28]

Fourth, neural networks are well-suited to using **distributed representation**s, rather than **localist representation**s. Mental representations (e.g. your knowledge of your grandmother) can be thought of in two ways: as being locally stored in one location in the brain, or as being distributed over many locations. A local representation scheme for the brain is sometimes called a "grandmother cell" doctrine, because it implies that there is just one neuron in your brain that represents your grandmother. In the context of neural networks, we can say that an object is locally represented by a neuron when activation of that neuron indicates the presence of that object. For example, in figure 1.8, blue cheese is locally represented by the neuron labelled "Center 5". When that neuron is activated, the blue cheese is present (here, "activation" means having a non-zero, positive activation value).[29]

In contrast, we can say that an object has a distributed representation in a neural network when a particular *pattern of activation* over a set of nodes indicates the presence of that object. In figure 1.9, the bottle of poison has a distributed representation. When the poison is present, a specific pattern of activation $(.1, 1, .7, 0, .2)$ occurs across the whole set of nodes.

For the most part, distributed representations are what one finds in the brain. Generally speaking, brain

[25]Much more rarely, the weights are hand-crafted, as in the IAC networks later in this chapter. But that is the exception that proves the rule. IAC networks are great at illustrating activation dynamics, but highly unusual in that the weight strengths are not learned from data, but are hand-set by a human.

[26]Of course, you run Simbrain on a conventional computer, so that in reality, the neural network is updated in serial. But this is just an artifact for convenience. In our brains, neurons fire in parallel, and large scale simulations of neural networks are also run in parallel (often on massive distributed computing clusters). In fact, all of the advances in the field since 2010 require parallel computation to achieve the required performance.

[27]This is related to the fact that they operate in parallel rather than serial. A neural network has lots of redundant wiring which can compensate for damage.

[28]A related point is that neural networks are good at *handling noisy inputs*. Digital computers don't like noisy input: they respond only to clean, precise inputs. Anyone who has worked with computers has some understanding of this. To get through a company's phone tree you have to enter just the right sequence of numbers–no mistakes allowed! But show me the same flower ten times, and I will see it as the same flower, even though the input to my brain is changing slightly (the lighting changes, things in my retina change, the whole process is noisy). You can see this in the Simbrain simulation by dragging the mouse around. Notice that even while the inputs change slightly the network continues to recognize which object it's looking at.

[29]Some older types of neural network use only local representations (e.g. the IAC networks discussed in this chapter), and we will see that it is sometimes useful to use local representations. However, the problem with local representations is that you lose some of the virtues above, in particular graceful degradation. If there is just one unit whose activation represents my grandmother, then if I lose that neuron I lose my whole memory of my grandmother. But the empirical evidence suggests that losing a single neuron will not lead to a person's losing an entire memory. So, even if some artificial neural networks use localist schemes to illustrate certain concepts, biological neural networks don't seem to.

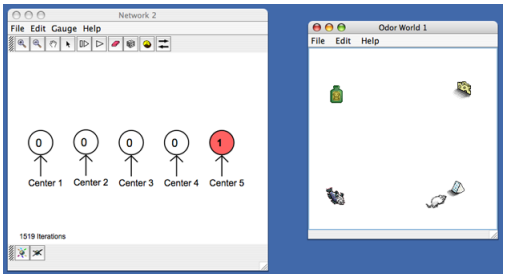


Figure 1.8: Localist representation

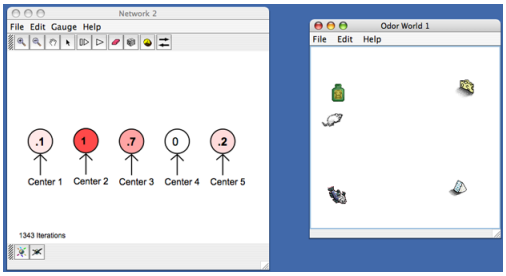


Figure 1.9: Distributed representation

functions are distributed over many neurons. Although it is harder to think directly about distributed representations than about local representations, a whole conceptual framework has developed which addresses the problem. In fact, much of the book is about developing a mindset that allows you to think about patterns of activation as points in a space (see in particularchapters 6 and 10). From that standpoint, the 3 object detector learns how to detect clusters of points, similar "smells in smell space."

Chapter 2

Applications of Neural Networks

Jeff Yoshimi, Zoë Tosi

In this chapter we consider how neural networks as models are applied in various areas of engineering and science, with an emphasis on applications to cognitive science, given the scope of the book. One fascinating feature of applications of neural networks, especially in recent years, is how a model that was initially designed for one purpose takes on a completely different meaning in another context. For example, neural networks that were initially used to study the visual system ended up being useful to engineers designing pattern recognition devices. Those pattern recognition devices then became interesting to scientists studying vision. ChatGPT and similar models were designed to provide a useful natural language interfaces (and to support other features like machine translation), but they perform so well that they have become objects of interest for scientists and theorists.[1]

2.1 Engineering vs. Scientific applications

In practice, neural networks are applied in two main ways: (1) as engineering tools, and (2) as scientific models.[2]

When neural networks are used as engineering tools, they are used to do useful things, like recognize faces in photographs or convert speech to text. Neural networks used for engineering do not have to be psychologically or neurally realistic, they just have to work well. In fact, it is preferable if they are *better* than humans, making fewer mistakes than we do.

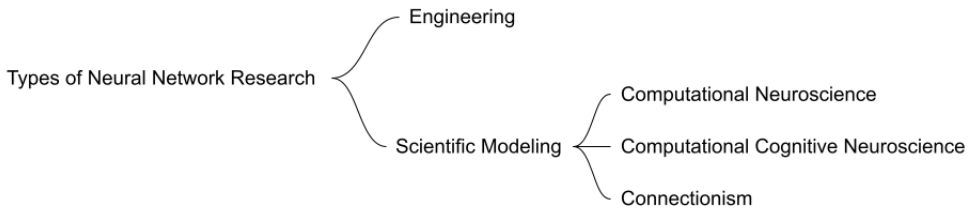


Figure 2.1: A taxonomy of types of neural network research, which serves as a guide to this section. Engineering uses (section 2.2) where the goal is to just make something useful, and scientific models where the goal is use neural networks to model the mind (connectionism, section 2.4), the brain (computational neuroscience, section 2.3), or both (computational cognitive neuroscience, section 2.5).

[1] It's a strange situation: systems designed by humans are not completely understood by the people who designed them, and so they have become topics of scientific and philosophical inquiry.

[2] See the discussion of "standards of intelligence" in [89]. Models of idealized intelligence are what we are calling "engineering" models here, while psychologically realistic cognitive models are what we are calling "neural networks as scientific models."

Neural networks used for scientific modeling should be neurally or psychologically realistic; they should accurately describe how the brain and the mind work. A neural network model of human memory, for example, should remember (and forget) things in the same way humans do in experiments. This second use of neural networks–as models of the mind and brain–itself subdivides into several subcategories, depending on what specifically is being simulated. Neural networks are sometimes used to understand the brain (in the field of "computational neuroscience"), sometimes to understand mind and behavior (this is sometimes called "connectionism"), and sometimes to understand both brain and mind simultaneously. A map of these types of research is in figure 2.1. As we will see, this taxonomy is not always so neat, and there are fascinating cases of, for example, engineering neural networks becoming objects of scientific interest.

A good rule of thumb when considering how to classify a neural network model is to ask: "What is the neural network being used for? As a tool that does something useful, or as a scientific model?"[3] Even then it can be tricky. For example, consider the following title of a journal article: "Use of Neural Networks in Brain SPECT to Diagnose Alzheimer's Disease" [93]. At first, this sounds like it might be scientific model, since it mentions the brain and Alzheimer's. However, the article is actually about how neural networks can be used to determine whether a person has Alzheimer's. The neural network is not being used as a model of the brain or any cognitive processes, but rather as an engineering tool to help diagnose Alzheimer's disease based on brain images. If we ask: "What is the neural network they made being used for?", the answer is to build a better diagnostic tool for Alzheimers. The network is *not* being used a model of Alzheimers. So it's really an engineering usage of a neural network, rather than a scientific model.

2.2 Engineering uses of neural networks

Neural networks in engineering are tools to solve problems. They are used as classifiers, controllers, signal processors and other components, alongside many other types of engineering tools. In fact, in the contemporary world, many things we take for granted are built on top of neural networks engineered to do useful things. They are at the heart of the current revolution in AI (as of 2023). They recognize voice and images, they generate speech, the generate images and movies, they drive cars, and of course, they can have human-like conversations with us (as with large language models like ChatGPT). They do many of the things humans do, often better than we can, because we can carefully engineer them and train them on such massive datasets.

We will see in later chapters how to systematically classify these different kinds of models, but for now we can focus on a very common case: the use of neural networks to classify objects, which is one application of **machine learning**. Classifiers are good at finding patterns in complex and noisy data. Remember, neural networks are trained, not programmed, making them well suited to tasks where there is no obvious way to mathematically determine the relationship between a set of inputs and a set of outputs.

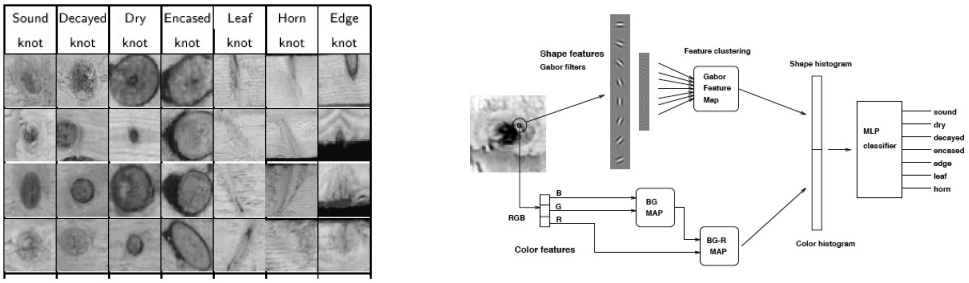


Figure 2.2: Left: Sample inputs to the knot classification network. Right: The knot classification system. The neural network is labelled "MLP."

Here is an example from the late 1990s. A lumber yard in Finland had to classify pieces of wood, identifying 30 different kinds of knot in images of lumber. Some examples are shown on the left side of

[3] A more advanced way to ask this question is to ask: how are the node activations and weight strengths being interpreted? Are they merely parameters in statistical models, or are they supposed to capture something real about neurons, or about concepts and their relations?

figure 2.2. A human can classify these knots, but it is time-consuming, expensive, and error-prone (look at how subtle some of the differences are between the different "dry knots"). It is also hard to program a computer to classify these knots according to explicit rules. Thus, neural networks were used, and they outperformed humans. A neural network trained on samples like the one shown have about 90% accuracy in this process, compared with 70-80% accuracy for humans. The neural network is shown in the figure. It is buried inside the system, the "MLP classifier" towards the right ("MLP" means "multi-layer-perceptron," which is a feed-forward network trained by backpropogation. It is similar to the 3-object detector above). This system takes a picture of a piece of wood, does some pre-processing on the resulting pixel image, and then summarizes features and colors of that image as a list of numbers, which is fed to the neural network as input. The neural network transforms these numbers into another list of numbers, which describe how decayed, burnt, dry, round, and so forth each sample is. This is a *feature vector*. This feature vector can then be used to classify the knot [52].[4]

2.3 Computational neuroscience

We now turn from neural networks used as engineering tools, to neural networks used as models of the mind and brain.

Computational neuroscience uses computational methods to answer questions related to neuroscience. Computational neuroscience spans many levels, from the micro-scale of cell membranes to the macro-scale of the human brain as a whole (see Fig. 2.3). It is a highly multidisciplinary field, encompassing biology, neuroscience, psychopharmacology, cognitive science, complexity science, psychology, and even physics, depending on the context.

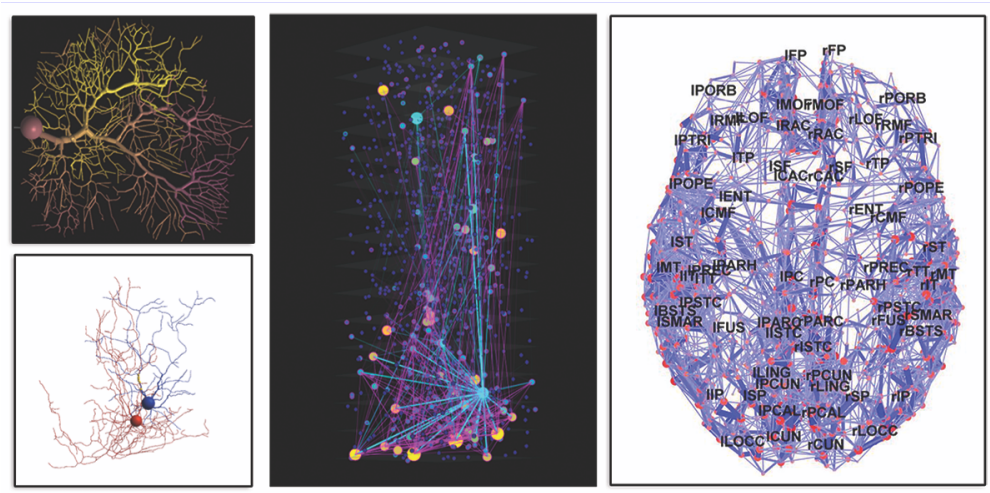


Figure 2.3: Micro, meso, and macro-level models in computational neuroscience. Left: micro-level models of individual neurons. Middle: meso-level model of a network of several thousand neurons. Right: macro-level model of neuronal connections spread out through the entire brain.

Micro-scale models in computational neuroscience study individual neurons or even individual parts of neurons, like the receptors that are studded in the cell membrane to let charged particles in and out of a cell (these are models of "receptor kinetics"). Models at this level often study the details of how charge flows through the tree-like structures of a neuron's dendrites and axons, and can accurately describe the behavior of individual neurons in a laboratory dish ("in vitro") when they are injected with current from small electrodes. Models at this level often attempt to answer questions that are physiological or pharmacological in nature like the effect of neuromodulators on the low level dynamics of a neuron, or how new receptors are

[4]Pre-processing is one aspect of data wrangling, which is discussed in chapter 7. Post-processing also occurs, for example all things that happen in ChatGPT after the network produces its raw output (for example filtering out inappropriate responses).

created or new dendritic spines are grown. These models are largely below the level of what is visible in a single Simbrain node.

Macro-scale models in computational neuroscience describe the behavior of large groups of thousands to millions of neurons and the connections between them. Oftentimes these models approximate the activity of thousands of neurons or even whole brain areas as the activity of a single higher level node.[5] These models can accurately describe the spatio-temporal organization of patterns of neural activity measured using brain imaging techniques like fMRI. In a Simbrain network simulation of this kind each node would represent the aggregate activity of thousands to millions of neurons and the whole network could represent the behavior of the entire brain. Models of this type are usually concerned with questions which are psychological or behavioral in nature. For instance the functional relationships between brain regions have been shown to be different in patients with schizophrenia resulting in a different overall graph structure of the functional connectivity between brain regions [18]. Often work at this level "bleeds over" into the realm of general neuroscience.

In this book we mostly focus on the *meso-scale* (or "middle"-scale) of computational neuroscience, between the micro and macro-levels. Whereas micro-scale models focus on individual neurons or their parts, meso-scale models focus on networks of *hundreds to thousands* (or more) of interconnected neurons. And whereas each node in a macro-level model *approximates* the activity of large group of real neurons, each of the simulated neurons in a meso-level simulation corresponds to a real neuron. Thus, a meso-level simulation containing 1000 artificial neurons is a direct simulation of a biological neural network containing 1000 real neurons. The emphasis is on discerning governing principles and dynamical phenomena associated with these networks. Meso-level models in computational neuroscience have been implemented in Simbrain (e.g. the middle image in Fig. 2.3).

The model neurons and synapses used in these simulations are more complex than the simple nodes and weights described above in Sect. 1.1, since they are designed to mimic the electrochemical properties of real nerve cells. Most neuron models in computational neuroscience are governed by equations acting on variables which represent specific electrochemical attributes of living neurons. Synapses have temporal delays and their signals have duration. Network models in computational neuroscience tend to be comprised of *spiking neurons* (neuron models which produce and propagate signals via action potentials) embedded in complex *recurrent* networks. We cover the special properties of these model neurons and networks in chapter 18.[6]

Very roughly, the focus of computational neuroscience at these three scales can be thought of as follows: Neuron Dynamics → Network Dynamics → Brain Dynamics

2.4 Connectionism

The use of neural networks as cognitive models, which behave in the same way humans and animals do, but without concern for neural realism, is sometimes called **connectionism**.[7] In connectionist models, there is no direct effort to understand the brain. The focus is on modeling some aspect of human or animal behavior using nodes and weights. Such models are usually meant to *suggest* how a given task is accomplished by the brain—they are "neurally plausible"—but they do not directly model the underlying neuroscience.

As an example, consider the "IAC" or "Interactive Activation and Competition" network. A famous example of an IAC network is McClelland's model of knowledge of two fictional 1950s gangs, the Jets and

[5] As an example, see `https://www.ncbi.nlm.nih.gov/pubmed/21511044` [20]

[6] In contrast to the micro-scale, which is (broadly) concerned with physiology, and the macro scale, which is often concerned with psychology, the meso-scale concerns itself with questions like: How do networks of interconnected neurons represent information? Can we replicate synaptic connectivity using plasticity rules? How does information processing emerge from the interactions of neurons embedded in a neural network?' Meso-scale models often attempt to understand formalisms that describe observations of groups of neurons (e.g. slices of brain tissue) with explanations of those observations using models of detailed low level processes at the micro-scale. It is known that when neurons fire in particular temporal sequences the synapses connecting them will become stronger or weaker depending upon that sequence. A micro-scale model might concern itself with how new receptors are created, or new spines are grown. A meso-scale model will only concern itself with the function translating that temporal sequence into a change in synaptic strength.

[7] Not everyone using the term "connectionism" in this way, but it is a fairly standard usage. A more precise phrase would be "connectionist model of a cognitive process".

Sharks from *West Side Story* [76].[8]

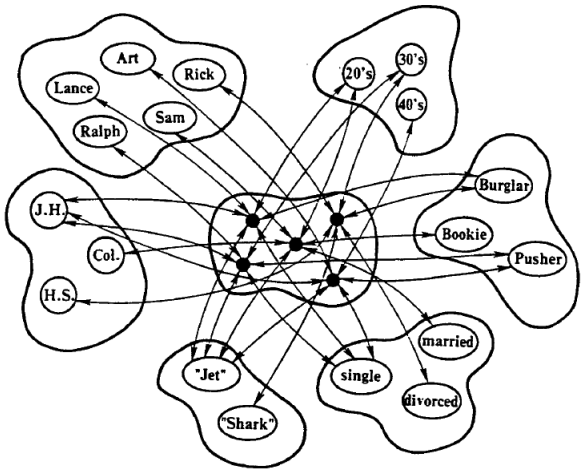


Figure 2.4: A fragment of the Jets and Sharks model. Nodes in this model don't represent neural activity, but activation of concepts in semantic memory.

An IAC model is organized into pools of nodes. In Fig. 2.4, there are 7 pools of nodes. These pools of nodes are used to model the internal concepts of a person who knows about these two gangs. The pools represent different traits: age, education level, marital status, job, etc. The central pool is a pool of "instance nodes" or "object nodes", shown as black disks, which correspond to individual people. Each person is associated with a node. The other pools correspond to properties of these people: their name, job, age, and gang affiliation. The nodes in each pool inhibit each other, which produces a *winner-take-all* structure. As the simulation runs, the activation of one node in each pool will tend to dominate the others. Notice that the topology of this network is recurrent, and as a result the network has dynamics: when we run it, activation starts to spread from one node to another over time. In fact, these have also been called *spreading activation* networks [76].

The IAC network models general features of human semantic memory, like the ability to retrieve attributes of a person based on their name. If the Lance node is activated and the simulation is run, activation will spread through the recurrent network, and after a while the Jets node, 20s node, Junior high education node, and burglar node will have the highest activations. This is like asking "Tell me about Lance?" and being told about him. The network can also model our ability to describe the properties of a group of people. If the Jets node is activated and the network is run, the standard characteristics of the Jets will light up: they tend to be in their 20s, with a junior high school education, and single. This is like asking "Tell me about the Jets?" and being told about that group. The network can also model our ability to identify people who match a specific description. If the 20s node and the junior-high education node are activated, then the name nodes for Lance, Jim, John, and George all light up. This is like asking "Who is in their 20s with a junior high education?" and being told "Well, that could be Lance, Jim, John, or George."

Though IAC networks are models of semantic memory, and are brain-like (spreading activation and winner-take-all types of dynamics do occur in the brain), they are *not* models of the brain, they do not capture any details of human neuroscience, or have nodes whose activation corresponds to activity in specific parts of the brain.

The IAC network is a qualitative model of human memory. Other connectionist simulations are more quantitative. For example, Seidenberg and McClelland modeled childrens' reaction times in reading words aloud. The network is a variant on a feed-forward network, similar to the network on the left side of figure 1.3. It has more nodes: 400 input units, which represent written words, and 460 output units, which represent spoken words. It was trained to pronounce all one-syllable words in English using a method called "backpropogation" (chapter 12). This simulation models the *word frequency effect*. Words that occur

[8]For a video overview of this network in Simbrain, see `https://www.youtube.com/watch?v=Nw3TEDfugLs`.

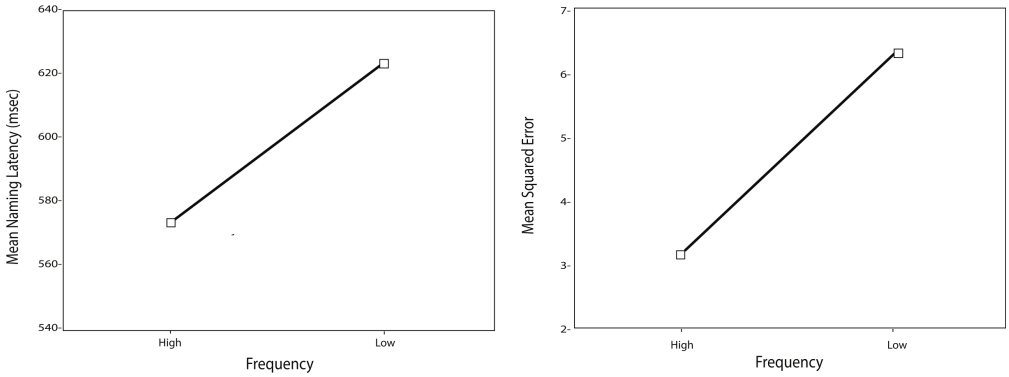


Figure 2.5: Data associated with Seidenberg and McClelland (1989)'s reading model. Human data are on the left, neural network data are on the right. Humans pronounce high frequency words more quickly than low-frequency words. The neural network makes fewer errors on high frequency than low-frequency words.

frequently in language (like "the") are pronounced more quickly than words that occur infrequently (like "rake").

Human data showing this effect are on the left side of Fig. 2.5. Humans pronounce high frequency words more quickly than low frequency words (the y-axis shows latency, or length of time to pronounce the word; lower values mean faster times). The neural network data on the right was generated by counting how many mistakes the network made for low and high frequency words [110]. When you line the two graphs up next to each other, they look the same. This suggests that the way the model reads is similar to the way humans read: both the neural network model and humans are better at reading more common words.

2.5 Computational Cognitive Neuroscience

For neural network used as scientific models, it can be hard to say whether it simulates the brain (computational neuroscience), or cognition (connectionism). Many models aim to do both at once. That is, many people who use neural network models are interested in *both* how the brain works *and* how cognition works, and of course, how the two are related. Thus, there are increasingly many models that attempt to capture both neural and psychological data, as was noted in the discussion of macro-level computational neuroscience above.

We will refer to these as **computational cognitive neuroscience** models, and define them as models that attempt to capture psychological and behavioral data while simultaneously paying attention to neural details. This type of model captures various aspects of cognition (e.g. visual attention, semantic and episodic memory, priming, familiarity, and cognitive control), using groups of neurons that are explicitly associated with specific brain circuits. Many researchers hope that over time computer models of brain and behavior will converge, and that future models will increasingly capture both neural and behavioral data, and thereby reveal how the dynamics of the brain give rise to the dynamics of cognition.[9] Some examples of this type of model are shown in figure 2.6.

From this standpoint, computational neuroscience and connectionism are two ends of a continuum or spectrum. We have computational neuroscience at one end, and connectionism at the other. All through the middle of this spectrum are models that try to model both biological data and psychological data at the same time. The goal is to understand how the circuits of the brain produce all the wealth and complexity of observable human and animal behavior.

[9]Examples of researchers and research groups working in this area include the work of Randy O'Reilly and his colleagues (`https://grey.colorado.edu/CompCogNeuro/index.php/CCNBook/Main`), Stephen Grossberg's work which began in the 1960s (`https://en.wikipedia.org/wiki/Stephen_Grossberg`), and the work of Chris Eliasmith and his colleagues (`https://uwaterloo.ca/centre-for-theoretical-neuroscience/people-profiles/chris-eliasmith`). There is also a burgeoning research community organized around the computational cognitive neuroscience conference (here is their 2024 program: `https://2024.ccneuro.org/`).

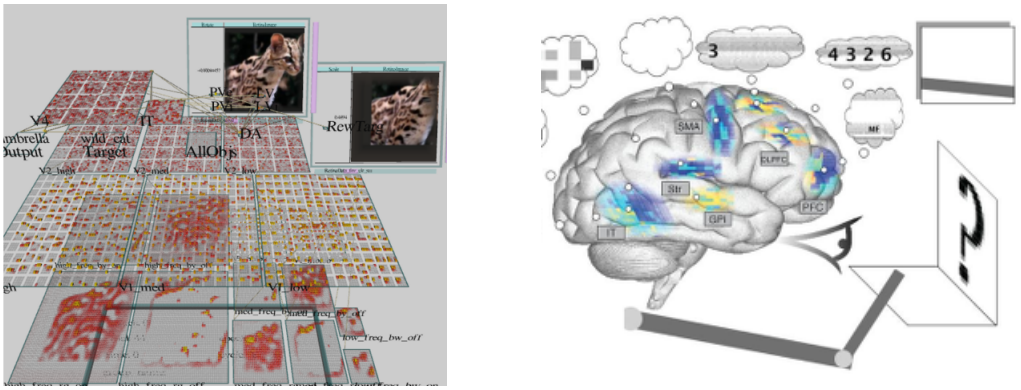


Figure 2.6: (Left) An Emergent simulation of visual processing, with labels indicating which brain areas each group of nodes represents. (Right) A Nengo simulation of the human ability to retrace a visually perceived number.

2.6 From science to engineering and from engineering to science

Some neural networks that originated as scientific models later informed the development of engineering tools. Conversely, sometimes an engineering tool ends up being useful as a scientific model. These are interesting historical dynamics. A prominent example are LLMs like ChatGPT. They originated as pure engineering, something useful. But now they have become an intense topic of scientific interest as linguists, cognitive scientists, philosophers and others ask to what extent they can tell us about human cognition (see section 17.5).

Deep networks provide a good example of these back and forths. Deep neural networks were originally used as scientific models of vision in the 1970s (see chapters 3 and 4). These early neural network models of vision turned out to be excellent tools for pattern recognition, a common engineering application, for example recognizing digits on an envelope. This is part of what led to the deep learning revolution of the 2010s. So we had a shift from science to engineering. But then it happened again, in the reverse direction. These new and improved deep networks turned out to be useful in computational neuroscience as a way to understand the human visual system. So, a neural network that started off in science, then got used for engineering, and then later that tool got adapted back to science!

Similar twists and turns from engineering to science are happening now with the emergence of large language models like ChatGPT. They originated as pure engineering models, that are meant to facilitate natural language processing. I think we can all agree that ChatGPT is useful, whether or not it "thinks like a human." But it's so compelling as a model, that linguists, psychologists, and philosophers are starting to study them as cognitive models. See Chapter 16. So in this case a model that started off in engineering subsequently became an object of scientific study.

Chapter 3

History of Neural Networks

Jeff Yoshimi, Pierre Beckmann

This chapter briefly outlines the history of neural networks, including the pre-history of neural networks and cognitive science extending back to ancient Egypt. The theory of neural networks has many historical precedents, but emerged as an explicit mathematical and computational formalism in the mid 1900s, via the work of McCulloch and Pitts. The main events developed in the chapter are shown in Fig. 3.1.

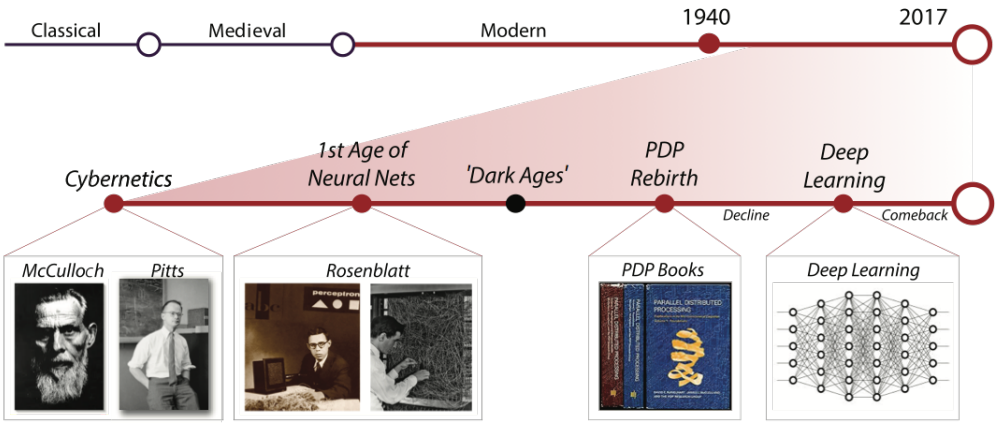


Figure 3.1: A timeline of the history of neural networks. The main history of neural networks runs from the mid 1940s to the present. We also consider some of the pre-history of neural networks, i.e. historical figures who linked the structure and dynamics of the mind with the structure and dynamics of the brain.

3.1 Pre-history

Cognitive science, the interdisciplinary study of mind, has ancient roots. Documentation of the idea that the brain plays some role in controlling behavior goes back to an Egyptian papyrus that is over 3000 years old.[1] Hieroglyphics from the papyrus describing the gyrations of the brain are shown in Fig. 3.2.

In Western philosophy and science, Plato, Aristotle, and other Greek philosophers had an interest in the structure of the human mind (or "soul") in relation to physical processes in the body.[2] Plato and Aristotle

[1] The papyrus can be viewed online; try searching for "Smith papyrus". Recent scholarship on the papyrus is collected in [80].

[2] I focus on Western roots of neural network theory, though there were precedents in other parts of the world, which I hope to add in future versions of this chapter. Currently, the literature is sparse. There is an expanding literature on the history of science globally (e.g [113]), but there is not (as of 2017) much scholarship on the history of neuroscience, cognitive science, or psychology in Africa, Asia, India, Mesoamerica, the Middle East, and other regions whose historical documents contain relevant

Figure 3.2: Hieroglyphics describing the sulci and gyri of the brain.

both described the soul as a set of interacting faculties (in Plato: reason, spirit, and appetite), and both speculated about its physical basis. They disagreed about whether the brain or heart is the physical basis of the soul (Aristotle thought the brain just cooled the blood), but by the end of the Classical period the dispute was resolved in favor of the brain [30].

In the Medieval period, priests, physicians, and natural philosophers throughout Europe and the Middle East discussed cognition in relation to the brain. Cognition was thought to be based on the play of "spirits" or vapors in the ventricles of the brain [30]. Spirits originating in the senses were combined in the "common sense" and then purified, and mixed in higher ventricles. A typical diagram from the period is shown in figure 3.3. The ventricles are now believed to be shock-absorbers and chemical reservoirs. They are not thought to play a central functional role in cognition. However, the idea that sensory inputs to the brain are combined and refined in various ways persists in connectionist models.

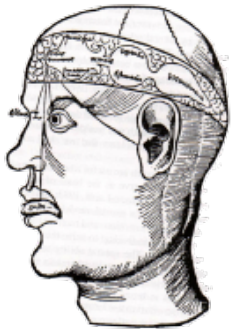


Figure 3.3: A medieval diagram which shows how spirits were thought to flow and combine through the ventricles of the brain. Different ventricles were associated with different faculties, such as sensation and the "common sense," integrating different senses, imagination, memory, and reason.

During the Enlightenment, many speculated that connections between ideas in the mind are based on connections between fibers in the brain (neurons had not yet been identified as distinct structures.)[3] In the 1700s, the empiricists Locke, Berkeley, and Hume famously claimed that ideas in the mind result from associations between simple sensory ideas: for example, a percept of an apple is composed out simple sensations corresponding to its color, shape, smell, and taste. One idea comes to mind, it calls another to mind, etc. Sometimes this happens instantaneously, as in the apple percept, but in other cases it might unfold in a temporal progression. Someone mentions apples, and that might make you think of fiber, which might in turn make you think of Raisin Bran. If someone mentions a person you know, associated thoughts about them–their age, where they live, their occupation, physical appearance, etc.–might also come to mind. One idea comes to mind, it calls another to mind, etc. Thus, the empiricists thought of the mind as something like an Interactive Activation and Competition (IAC) network (cf. Chapter 1) [5].

In this period, David Hartley argued that the empiricist theory of associations could be explained by

information. There is however, some literature on Arabic and Islamic roots of neuroscience [86].

[3]For more on this period of history, see Sutton (1998) [118]. Sutton's discussion of Descartes is especially interesting, since it shows how Descartes had a connectionist styled account of the brain, which on his view interacts with a non-physical soul via patterns of activity at the pineal gland. Other mechanist philosophers of the period such as Hobbes and La Mettrie had similar accounts but rejected the assumption of a non-physical soul.

laws governing connected neurons [49, 5]. For example, Hartley [48] proposed that sensations $A, B, C, ...$ which are associated with each other, are associated because of correlated associations between "vibrations" of brain fibers:

> PROPOSITION 10: Any sensations A, B, C, etc., by being associated with one another a sufficient number of times get such a power over the corresponding ideas a, b, c etc., that any one of the sensations A, when impressed alone, shall be able to excite in the mind b, c, etc., the ideas of the rest.
>
> PROPOSITION 11: Any vibrations A, B, C, etc., by being associated with one another a sufficient number of times get such a power over the corresponding miniature vibrations a, b, c etc., that any one of the vibrations A, when excited alone, shall be able to excite in the mind b, c, etc.

He proposed proposition 11 as a neural explanation of proposition 10, which is psychological. Note that proposition 11 is an early version of what would later became known as Hebb's rule or Hebbian learning ("neurons that fire together, wire together"), discussed below and in chapter 9.

Later, in the 19th century, Bain illustrated these ideas with images that look strikingly like modern neural network diagrams, as in Fig. 3.4.

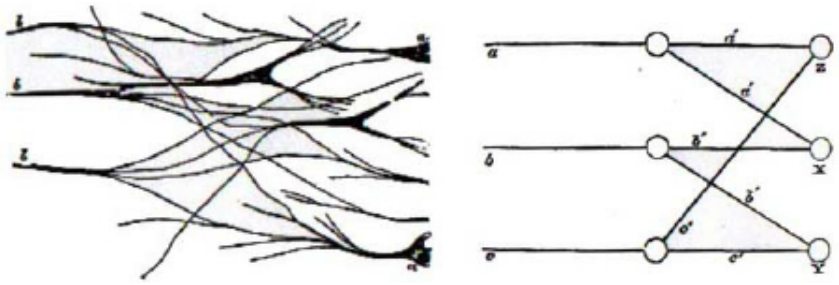


Figure 3.4: From Bain's 1873 book *Mind and Body*, which opens with the question "What has Mind to do with brain substance, white and grey"?

The notion that associations between thoughts and memories are based on neural connections in the brain was further developed in late 1800s by Sigmund Freud, who developed a psychodynamic theory, according to which psychical "energies" are based on the flow of activations in the neural networks of the brain. His goal was to show how psychology could become a natural science by representing "psychical processes as quantitatively determined states of specifiable material particles" [34, p. 355]. But whereas earlier theorists had simply speculated about associative processes, he based his on actual clinical observations, and in particular observations of (allegedly) neurotic patients experiencing "excessively intense ideas." He explained his clinical observations in terms of "neuronic excitation" understood as "quantities in a condition of flow" (p. 356). Fig. 3.5 shows part of an image from this early book, which describes a patient (Emma Eckstein, who went on to become a famous author) who avoided shops based on an earlier traumatic experience. The specifics of the account are dubious, and Freud himself gave up on the project of a direct neural account of psychological processes, but it does show that Freud was thinking about the mind in a broadly connectionist way.

Many other psychologists, neuroscientists, and philosophers in the late 19th and early 20th century contributed to the general idea that psychological processes are rooted in neural processes. Helmholtz, Mach, Ramón y Cajal, Golgi, and others advanced biological psychology in various ways (see, e.g., [13]), e.g. by establishing that neurons are individual cells, and by applying mathematical methods to psychology and neuroscience. In Russia, the psychologists Luria and Pavlov sought to understand the neural basis of associative learning, speech pathology, and other cognitive phenomena in a quantitative, experimentally tractable way.[4]

[4]On Luria in relation to the history of neural networks, see [106, p. 41].

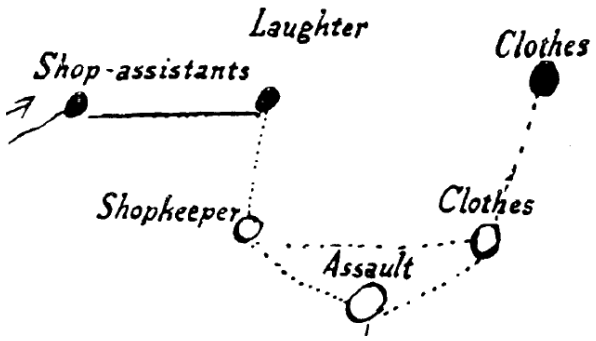


Figure 3.5: An image from Freud's early *Project for a Scientific Psychology.* The open circles are conscious ideas; the dark circles are unconscious ideas.

3.2 Birth of Neural Networks

We now turn to the history of neural networks proper, i.e. explicit formal descriptions of artificial neural networks, which could be implemented in computer programs.[5]

The first wave of research into neural networks occurred in the 1940s, via an array of neuroscientists, mathematicians, logicians, and engineers, many of them at MIT.[6] The history is complex, fascinating, and brimming with colorful personalities (see the early chapters of [3]). This was the period when digital computers were first being developed by people like John von Neumann, a child prodigy who later established a computer architecture still in use today (the "von Neumann architecture" [120]). The architecture involves a separation between memory and a central processing unit that retrieves data from memory and operates on it using logical rules.

Neuroscience had also been steadily advancing in this period, and the network structure of the brain and its relation to behavior were better understood. The field of control theory was emerging via the work of Norbert Wiener, another child prodigy. He developed the field of "cybernetics" (which is closely related to modern control theory), and defined it as "the science of control and communication in the animal and the machine" [127, p. 16]. In the late 1930s, the petroleum industry had developed central control systems to maintain refinery towers, and in WWII feedback systems were used to control anti-aircraft guns. A key idea in cybernetics was that these feedback circuits could coordinate complex movement, both in engineered systems and in the brain.

In this atmosphere, two scientists emerged as the "fathers of neural network theory": Warren McCulloch (a neurophysiologist affiliated with cybernetics) and Walter Pitts (a logician).[7] They wrote a famous paper showing how neuron-like elements could perform all the logical operations performed by computers. This in turn implies that whatever can be done on a computer can, in principle, be done using neurons [79]. A diagram from McCulloch and Pitt's famous paper, *A Logical Calculus of Ideas Immanent in Nervous Activity*, is shown in figure 3.6.[8] In appendix A, a demonstration of a similar approach to building logic gates using neural networks (in Simbrain) is developed.

McCulloch and Pitts used what are now called binary units or threshold units (see chapter 5): nodes that

[5]The history of neural network research from this point forward is covered in several places. Fausett has a 4 page overview that covers the main points nicely: [28, pp. 22-26]. Levine, ch. 2 is especially detailed on McCulloch,s Pitts and Rosenblatt [71]. A brief online history is at `http://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/`. A more recent history that extends to present day work in deep learning is at `https://www.skynettoday.com/overviews/neural-net-history`. Also see the end of chapter 1 of the first PDP chapter, [106], and Haykin (2nd Ed.) section 1.9 [50]. My favorite source is a series of interviews of leading figures in the history of neural networks collected in *Talking Nets*, [3].

[6]Important research relating to neural networks did occur earlier in the 20th century, e.g. work by Thorndike, Lashley, and Clark Hull. Hull's writings contain diagrams and formulas describing associative learning processes based on rat studies that look very much like connectionist networks (e.g. `http://psychclassics.yorku.ca/Hull/Hierarchy/part1.htm`).

[7]Both had vivid personalities. McCulloch had wild hair and charisma. Pitts was a quiet introvert who had trouble getting a regular job, but who was regarded by his associates as a genius and was supported by McCulloch for many years. For a fascinating first-hand account of their personalities see the interviews with Lettvin, Cowan, and Arbib in [3]. See in particular pages 9, 101, 104, 218, 223. Video interviews with McCulloch are available online.

[8]See Levine, p. 12, for a useful summary of how McCulloch / Pitts networks operate [71].

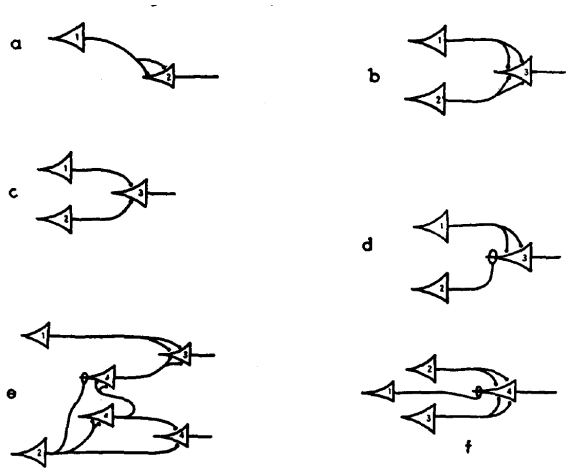


Figure 3.6: From the end of McCulloch and Pitt's famous article, in which they demonstrate that "for any logical expression satisfying certain conditions, one can find a net behaving in the fashion it describes." In these networks, what are today called "weight strengths" or "synaptic efficacy" correspond to number of connections. Nodes only fire if two or more incoming connections are activated. "Lasso" connections correspond to what are today called "inhibitory" connections. In these networks, a single activated lasso connection will disable the node it is connected to. A network computing *logical or* is shown in panel B (it will fire if either of the inputs nodes connected to it fires) and a network computing *logical and* is shown in panel C (it only fires if both input nodes connected to it fire). Panel E models the heat illusion (briefly held cold objects can feel hot). For an elaboration of this case see [97].

are only activated when their summed inputs are above a certain value. Nodes could only be active at a level of 0 or 1, based on the "all or none" property of neurons (cf. 4). They made some assumptions that are unusual by today's standards. For example they assumed that a single inhibitory input is sufficient to completely prevent a neuron from firing. More importantly, they did *not* describe connections between neurons using variable-strength weights. Their networks used fixed connections, which could not be adjusted using a learning rule. Learning rules would later become a primary focus of neural network theorists. Nonetheless, it was the first time an actual formal model of a neuron was presented, together with a serious effort to understand how networks of neurons could produce complex behaviors.

3.3 The Cognitive Revolution

In the 1950s advances in linguistics, early computer science, neuroscience, and psychology, among others, coalesced in a broad reaction to earlier approaches to psychology, which had focused on observable behavioral data. The earlier *behaviorists* had frowned upon discussions of internal processing between sensory inputs and motor outputs, and treated the mind as a kind of black box. The emerging cognitive scientists wanted to break open that black box and look inside: they wanted to understand what kind of processing occurs between sensory input and motor output in terms of *computation*. The big idea that got everyone excited was that inside the mind there is an information processing system, one similar to the computer systems that were just then beginning to be realized on a large scale. This was sometimes referred to as the "cognitive revolution" in psychology.[9]

In the early cognitive revolution many different kinds of cognitive model were considered, including early neural networks like the Perceptron, discussed next. However, from early on there were doubts about neural networks. There were, for example, concerns that statistical approaches were fundamentally limited, a concern most famously posed by Minsky and Papert (see chapter 13). Others, were concerned that neural networks could not store and manipulate context-free symbols to use in reasoning [33], and there was also

[9] An excellent overview and history of the era is [4].

a worry that human-like language and reasoning could not be learned from the input stream used to train a neural network ("poverty of the stimulus" arguments [9]). A competing approach to modeling cognition that avoided these problems was the symbolic approach—it goes by various names, including "Symbolic AI" and "Classical AI"–according to which cognition involves manipulating symbols in accordance with rules (see section 1.2.3).

Over time, a kind of war broke out between these groups (brewing in the 1950s and peaking in the 1980s). The connectionists had responded to the concerns raised by the symbolic AI camp, building networks that attempted to do all the things that symbolic AI advocates said they couldn't. The symbolic AI camp responded with doubts about these efforts, and for a time the "framework wars" were at the center of cognitive science. Today there are other camps as well, and some who mix ideas from both symbolic AI and connectionism. To jump ahead briefly, with the advent of large language models that produce convincing natural language, many feel that the Symbolic AI-connectionism war has been settled in connectionism's favor, and indeed as noted in the introduction today "AI" is often used to refer to neural networks. But the symbolic approach has fought back against this, and in some sense the old debate has been rekindled (see section 17.5.4).

Here are some themes in early cognitive science that prefigure connectionism. The Canadian psychologist and neuroscientist Donald Hebb formulated his famous learning rule for weights, the "Hebb rule" ("neurons that fire together, wire together") [51] (cf. the discussion of Hartley above).[10] Hebb also described the operation of the brain in terms of networks of connected neurons, formulating the concept of a "cell assembly", a group of neurons that becomes associated over time and thereafter tend to collectively reverberate in response to a stimulus (Fig. 3.7 shows one of Hebb's own diagrams of a cell assembly) [51]. The concept of a specific, learned pattern of brain activity produced by a stimulus remains important today.[11]

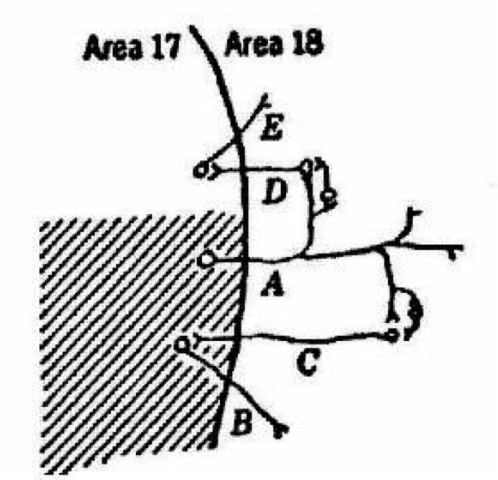


Figure 3.7: A Hebbian cell assembly. These neurons initially fired together, and then got wired together, and so they will tend to fire together in the future.

Another important figure in the period was the psychologist Oliver Selfridge, who pioneered the idea that psychological processes can be broken down into interacting sub-processes. His "Pandemonium" theory described the mind as a collection of "demons", each of which takes care of one specific aspect of a task. For example, figure 3.8 shows how Selfridge thought of the process of perceiving the letter "B". As can be seen the figure, the demons are basically nodes and the model is basically a feed forward network. An image arrives at the eye, line demons detect lines and curves in various orientations, those demons send messages to the next layer of demons who detect combinations of lines and curves, etc. The process continues through a network of demons until a decision demon says "B!" [112]. This is a striking anticipation of the concept of

[10] See `http://www.scholarpedia.org/article/Donald_Olding_Hebb`. Also see Werbos' interview in [3].

[11] See `http://www.scholarpedia.org/article/Cell_assemblies`. For a more up to date version of the idea cf. the concept of a polychronous neural group or PNG, `https://www.izhikevich.org/publications/spnet.htm`.

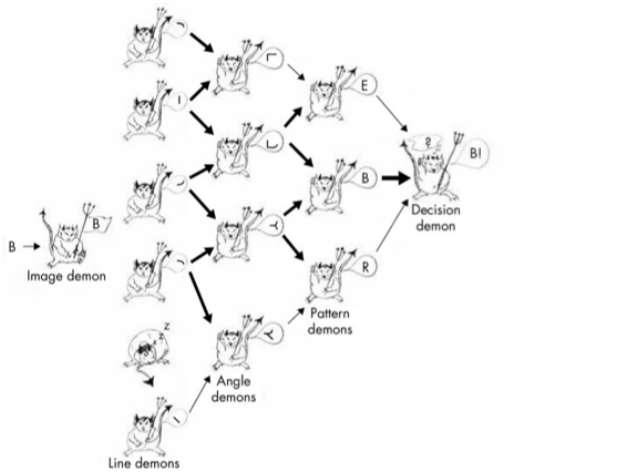


Figure 3.8: Selfridge's pandemonium model. Here the model is detecting the letter "B". The solid lines correspond to connections between active nodes, or "demons".

a deep network for vision with layers of increasingly complex feature detectors (see chapter 14).

Other important research in this period was carried out by the psychiatrist and cyberneticist William Ashby (who wrote *Design for a Brain* in 1952), Marvin Minsky (who wrote a dissertation on neural networks on 1954), and Dennis Gabor (a Nobel laureate who worked on holograms, and introduced a standard method for translating visual stimuli into a numeric form, that can be processed by neural networks).

3.4 The Age of the Perceptron

The types of layered feed-forward networks that are typically used today were first studied in detail in the 1950s and 1960s, primarily via the work of Frank Rosenblatt and Bernie Widrow (both published seminal papers in the late 1950s and early 1960s; see [126]).[12] Rosenblatt called his network the "Perceptron" and Widrow called his an "Adaline". Both had a single layer of adjustable weights, threshold output units, and learned using an error function (see chapter 12).Thus both networks moved beyond McCulloch and Pitt's networks—which did not involve any learning, just hand-crafted connections—to networks that actually learned from experience. This is, of course, what is distinctive about modern neural networks.

Rosenblatt was a psychologist interested in human and animal behavior and its neural basis.[13] He studied feed-forward networks with one layer of fixed weights and another layer of adjustable weights that could be trained to classify images on small displays as, for example, triangle vs. square, or male vs. female. He implemented his networks using huge tangles of wires for synaptic links (see Fig. 3.9). This was quite impressive at the time and got a considerable amount of press.[14] His rule was an early form of supervised learning rule based on a few if-then rules: if a sample is misclassified, then if output was too high, strengthen the weights, otherwise weaken the weights. Rosenblatt proved that perceptrons could find solutions to certain types of classification tasks in a finite time [104].[15] Haykin, who refers to this as the "classical period of the perceptron", summarizes Rosenblatt's importance as follows:

> The perceptron occupies a special place in the historical development of neural networks: It was the first algorithmically described neural network. Its invention by Rosenblatt, a psychologist, inspired engineers, physicists, and mathematicians alike to devote their research effort to different aspects of neural networks in the 1960s and the 1970s. Moreover, it is truly remarkable to find that

[12] A concise summary of this period of history is in Bishop p. 98 [11]. Also see [126].

[13] As with McCulloch and Pitts, Rosenblatt's personal history is fascinating, and in some ways tragic. See the Cowan and Hecht-Nielson interviews in Talking Nets [3].

[14] See `https://www.youtube.com/watch?v=cNxadbrN_aI`

[15] This is known as the "perceptron convergence theorem." For an intuitive discussion see `https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote03.html`.

Figure 3.9: Rosenblatt with one of his hardware implementations of a perceptron (left) and another view of the perceptron (right).

> the perceptron... is as valid today as it was in 1958 when Rosenblatt's paper on the perceptron was first published.

Whereas Rosenblatt focused on psychological implications of the perceptron, Widrow and his colleagues had engineering applications in mind, like adaptive noise cancelling in telephone wires.[16] In the hardware implementation shown in Fig. 3.10, the toggle switches control input node activations, the knobs control weight strengths, and the dial shows the activation of an output node.[17]

The least mean squares rule or LMS is discussed in section 13.1. Whereas Rosenblatt had derived his if-then rule from his understanding of biology, LMS was derived from calculus using first principles, which is how most modern learning rules are also derived. The essence of the rule is that we change each weight by a factor corresponding to the output error times the input node activation. This produces a kind of Goldilocks principle. When an output is too high (and the input is positive), we reduce the relevant weight so that the output will be lower, and when an output is too low, we increase the relevant weight so that the output will be higher. In this way the network zooms in on the right value, so that the output is just right. This is, in a nutshell, the essence of how most modern neural networks work! Widrow can be seen using this rule to train his machine to respond to T's and J's in variations positions in the video referenced in the figure caption.

3.5 The "Dark Ages"

Perceptrons and Adalines created a surge of interest in neural networks in the 1960s, but this was followed by a period of relative quiescence in the 1970s and 1980s, during what have been called the "dark ages", "quiet years", "drought", and "winter" of neural networks.[18] The dropoff in interest has been attributed to several causes. A key part of the story was that networks with a single layer of adjustable weights were shown by Minsky and Papert to suffer certain fundamental limitations [83] (see chapter 12). So it was thought that neural networks weren't powerful enough to do psychologically realistic things. Moreover, at precisely that time more symbolic AI models were flourishing.

[16] According to Widrow this technology is used in every modem in the world and is at the heart of the internet; see `https://www.youtube.com/watch?v=skfNlwEbqck`. Widrow also implemented his networks using a special electrical components called a "memistor" which Widrow designed himself, which allowed weight updates in hardware. ALl of this is demonstrated in the video.

[17] For Widrow's personal recounting of the Adaline and its history, see his interview in *Talking Nets* [3]. Also see the videos referenced in the figure caption, and the discussion in section 13.1

[18] See PDP vol 1, ch. 1 [106]; Haykin p. 43 [50]; Fausett p. 24 [28]. A variety of perspectives on the period are discussed in Talking Nets. 110, 155, 254, 305, 371. Grossberg, Carpenter, Kohonen, Anderson and others active in this period have their own interviews in Talking Nets [3].

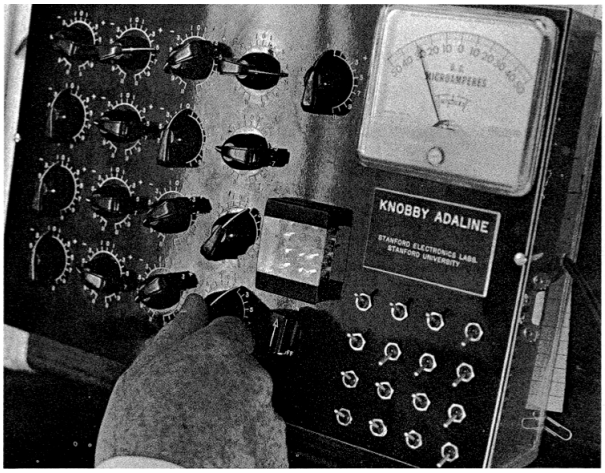


Figure 3.10: A hardware implementation of Widrow and Hoff's "Adaline" network, which Widrow called the "knobby Adaline" on account of the prominent grid of knobs on the left, which control weight strengths, and which were manually adjusted to implement their learning algorithm. Inputs were produced using the 12 toggle switches on the lower right (and displayed in the grid of small lights), and the resulting output activation is displayed in the meter on the upper right. Videos of Widrow demonstrating the knobby Adaline, back in the 60s and also more recently, are available at `https://www.youtube.com/watch?v=skfNlwEbqck` and `https://www.youtube.com/watch?v=IEFRtz68m-8&t=161s`

However, even if interest in neural networks waned for a time, especially in comparison to AI, neural network research was active in this period. Relevant researchers include Kohonen, Fukushima, Anderson, Sutton, Barto, Braitenberg, Grossberg, and Carpenter. These and others laid the foundations for many of the ideas described in this book. So the dark years really weren't that dark.[19]

3.6 First Resurgence: Backprop and The PDP Group

Connectionism came out of its (allegedly) dark decade and enjoyed a resurgence in the 1980s, for several reasons. Prominent among these was the discovery of the backpropagation algorithm, which overcomes the limitations associated with perceptrons. This makes it possible train networks with more than one weight layer, which in turn allows them to solve more complex problems (for example, "linearly inseparable" classification tasks; see chapter 12) by developing internal representations. A simple example of this is the ability to train a network to solve the exclusive or (XOR) logic gate. McCulloch and Pitts neurons could be hard-wired by hand to solve this problem, but no one could train a network automatically to solve it, because it requires an additional layer of processing. This is easy to see in Simbrain: you can easily build an XOR network by hand (appendix A), but try to use LMS to train a one weight layer network on the same problem and the error will never go to 0.

The internal representations discovered in many-layered networks using backprop were shown to have important properties both for engineering applications and for psychological theories, as we will see throughout the book.[20]

A related reason for this renewed interest—particularly among cognitive scientists—was the publication of a major two-volume work in the period, *Parallel Distributed Processing: Adventures in the Microstructure of Cognition*, in 1986, by David Rumelhart, James MClelland, and the "PDP research group" (a group of researchers, many of whom were at UC San Diego.) This publication brought connectionist networks back to the forefront, by clearly articulating the connectionist standpoint, showcasing a number of models of various

[19] Debates about the the status of neural networks in this period are covered in some detail in Talking Nets [3].

[20] In future iterations we hope to expand this section with a summary of some of the major work that began in this era, primarily in connectionism, using neural networks to model psychological phenomena. As a first note in this direction, we refer to the table at the end of this paper: `https://link.springer.com/article/10.1007/s42113-020-00081-z/tables/1`.

aspects of cognition, demonstrating how to interpret the internal states and representations of neural network models of cognition, and clarifying how connectionist networks differ from symbolic AI models [106]. John Hopfield's models of associative learning in recurrent networks (i.e. "Hopfield nets", discussed in chapter 9) were also influential in this period, in part because Hopfield presented his work in an especially clear, mathematically precise way.[55][21]

3.7 Second Decline and Second Resurgence: Convolutional Networks

For a time (roughly the late 1990s through about 2010), neural networks declined in interest as attention shifted to machine learning algorithms (cf. chapter 1). The problem was, in part, that tuning the parameters of a neural network seemed more an art than a science, especially when compared with machine learning, which is based on more tractable statistical principles. There was a sense that people just "twiddled" the knobs of a simulation as best they could until they got decent performance out of their network. In 2010 (in the midst of this decline), Phillip Jannert said: "Neural networks were very popular for a while but have recently fallen out of favor somewhat. One reason is that the calculations required are more complicated than for other classifiers; another is that the whole concept is very *ad hoc* and lacks a solid theoretical grounding" [57, Ch. 18]. There was a pervasive sense at the time that "...neural nets were janky and did not work very well. They were seen as a hassle to work with—the computers were not fast enough, the algorithms were not smart enough, and people were not happy" [65].

However, several things happened that brought attention back to neural networks. First, larger datasets for training neural networks became available (what is sometimes referred to as "big data"). Second, higher performance hardware for parallel neural network computing emerged, in particular using the graphical processing units or GPUs on graphics cards (the kinds used to play modern graphics intensive video games) and in proprietary hardware such as Google's tensor processing units (TMUs).[22] These breakthroughs suddenly allowed for the deployment of much larger **deep networks**. They not only had more layers, but relied on new, more complex, architectures.

Among these new architectures the most important were convolutional neural networks or CNNs (see chapter 14), which comprise sequences of convolutional layers, a special kind of weight layer in which a single set of shared weights is "scanned" over an input layer. Convolutional layers make it possible to effectively train deep networks with many more than 3 node layers. CNNs led to an explosion of interest in deep learning and deep networks. Rather than only dealing with vectors and matrices, these architectures began to use more complex tensors (see 6.9), such as "volumes" of activation describing images, videos, and other structures in rich detail. Whereas most neural networks through the 1990s had just three simple node layers, CNN's can have tens or hundreds of layers, many of them multi-dimensional arrays. Thus networks with a great deal of representational width and representational depth could be developed (see section 1.1 and chapter 14). These many-layered convolutional networks existed as far back as the 1970s (via Fukushima's neocognitron and related models), but it is only in the 2010s that a variety of technical hurdles relating to this type of network were surmounted. Indeed some have described the period beginning around the 2010s as a deep learning revolution, or as the decade of deep learning.[23]

3.8 The Age of Generative AI

In 2017 Google introduced the **transformer architecture**—discussed in chapter 17—which was subsequently adopted and extended by Open AI and many other groups in academia and industry [119]. The

[21] In Talking nets, on the PDP group, see pp. 180, 254, 277, and 281. On backprop and its history, see 286, 327, and 338. On Hopfield, see 113, 301 [3].

[22] It is interesting that these games require lots of parallel processors to render texture and shading in real-time graphics processing using linear algebra (cf. chapter 6), and that the same parallel processing circuits can be used to run neural networks. When graphics cards were first developed they did not have neural networks in mind!

[23] Andrey Kurenkov's history (`https://www.skynettoday.com/overviews/neural-net-history`) is excellent on these points. The achievements after 2010 are too numerous to survey here, but see `https://bmk.sh/2019/12/31/The-Decade-of-Deep-Learning/`.

power of this architecture became widely known with the public release of Open AI's ChatGPT in 2022. ChatGPT had the fastest adoption rate of any software in history, and marked another shift in the history of neural networks and AI broadly. In this period, it became common to train extremely large models on vast amounts of data, using innovative new architectures like the transformer architecture. The resulting neural networks could be used to generate convincing outputs in multiple modalities, but especially text, video, and audio, hence the term **Generative AI**.[24,25]

The transformer architecture builds on several streams of prior work. It is coming right off all the advances in training deep networks just discussed. They make use of huge datasets and benefit from innovations in hardware design.[26] They develop many layers of internal representations that have a great deal of context awareness, which they can use to (for example) represent relationships between different parts of a fairly long conversation. The details are discussed in chapter 17. The main initial use of transformer models was to generate natural language, and indeed ChatGPT is an example of a **large language model** (LLM), which is a language model trained on a large dataset–for example, a significant portion of text on the internet–to generate human-like text.[27] The text produced by these models is now so convincing that they (in some contexts) pass the **Turing Test**, producing outputs that are not distinguishable from what a human can produce. Related advances (e.g. diffusion models) produce convincing images, videos and audio.

These changes in the landscape of neural network research are significant enough that we are dubbing this the "age of generative AI" (the histories are just now being written, after all). This was when AI could really start generating new content, like news stories, essays, songs, images, and movies. In the future this will probably be seen as a landmark event, because this is when all the old doubts about neural networks were in a sense put to rest (of course, debate continues, but neural networks have clearly moved to the center of discussion), and when neural networks began to lead to fundamental changes in human existence, that we have not yet come to terms with.[28]

It has been a strange but exciting experience writing different versions of this chapter over the last few decades (the first version was written around 2005; see the Preface) as neural networks were out of vogue, then back in style, and then, arguably, completely transformed human society. No doubt more revisions to this chapter are coming, as the landscape continues to evolve.

[24] One mark of the shift is that it became extremely *expensive* to train state-of-the-art models, and so reseachers could not train their own but had to rely on models trained by large companies such as Google, Open AI, and Microsoft.

[25] Though the term generative "AI" is used, this usually means *neural networks* that have been used to generate these outputs, so this could more accurately be called "generative neural networks", but the term generative AI has stuck (see the first footnote in chapter 1).

[26] As evidence, consider the explosive growth of NVIDIA, which started off in the graphics card business but is now a major driver of generative AI.

[27] In fact, the terms "transformer" and "llm" have become conflated, though they are distinct (more on this in chapter 17).

[28] This was also when "artificial general intelligence" (AGI) started entering the public consciousness, that is, AI that is not just able to do specific things in specific domains, but could behave in an intelligent manner in multiple domains and contexts. It is not generally believed AGI has been achieved as of yet.

Chapter 4

Basic Neuroscience

Jeff Yoshimi, Chelsea Gordon, David C. Noelle

In this chapter, we review the basic physiology of neurons and synapses, which are the basis of equations describing how node activations and weight strengths change. We also provide an overview of the major circuits of the brain, reviewing their basic features, and giving a sense of how these circuits are understood from a neural networks standpoint.[1]

4.1 Neurons and synapses

4.1.1 Neurons

Neurons are brain cells, which have all the machinery any cell has: mitochondria, Golgi apparatus, a nucleus whose DNA is actively expressing genes, and a membrane studded with an array of proteins. Neurons communicate using a finely orchestrated pattern of electrical, chemical, and molecular processes. Neural network models typically abstract from most of these details. Classical neural network models only simulate certain high level features of the way information is transmitted from one neuron to another. Models in computational neuroscience (described in chapter 1) capture more of the biological details, but still abstract away from many features of real neurons. In the next two sections we give a rudimentary overview of the structure of neurons and synapses, focusing on features that are commonly referenced in neural network models.

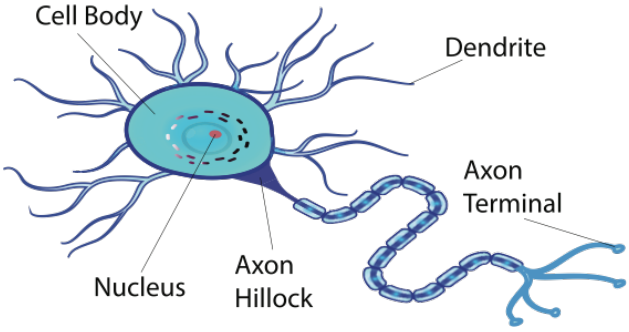


Figure 4.1: Main structures of a neuron.

Most neurons have dendrites, axons, and a cell body (see Fig. 4.1).[2] These are not really distinct

[1] Some useful general references include Kandel (2000) [58] and Gazzaniga (2002) [37]. An outstanding online source is `https://science.eyewire.org/home`. A detailed book length treatment of the topics outlined in this chapter is [91], which has also been developed into a free online text supported by open source software: `https://compcogneuro.org/`.

[2] There are many types of neurons in the brain, but in this section we focus on the *multipolar* neuron. This type of neuron

structures, but are parts of the cell. The neuron as a whole is a kind of container, whose overall charge is changing constantly over time like a fluctuating battery or capacitor.

Dendrites are extensions that grow out of the cell body like a tree ("dendrite" comes from a Latin word that means "tree"). This is where information-carrying chemicals are received from other neurons. Dendrites have small branches, which can receive signals from many other neurons. Some metaphors might help you remember this: You can think of a dendrite as the mail-box of a neuron, receiving messages from many nearby cells. You can also think of it as a catcher's mitt, since it receives or "catches" inputs from other neurons.

The cell body or **soma** is home to many organelles that work to produce and package proteins for the cell. These proteins have a variety of important functions, including the production of neurotransmitters that signal between cells. The soma sums together all the information gathered from the dendrites. If the voltage changes enough at a part of the soma called the *axon hillock*, the neuron will fire an **action potential** or spike along its axon. We expand on these ideas in the discussion of synapses next.

The **axon** is another extension growing out of the cell body. Axons can be different lengths, but they often have a long main extension terminating in a branched structure. When the neuron fires an action potential, electrical activity propagates down these extensions and triggers the stimulation of the dendrites of other neurons. Continuing our metaphors: the terminals at the end of the axons can be thought of as a neuron's post office, where the ionic messages transmitted down the axons are packaged into neurotransmitter chemicals and sent out on their route to the receiving neuron's dendrite. Or, continuing the baseball metaphor, the axon is like the arm of a pitcher, throwing signals to other dendrites, which catch them.[3]

4.1.2 Synapses and neural dynamics

Communication between neurons happens at a **synapse**, which is a junction where the axon of the *pre-synaptic* neuron almost touches the dendrite of the *post-synaptic neuron*, often at a protrusion called a *dendritic spine*. See Fig. 4.2.[4] When an action potential reaches the axon terminal of the pre-synaptic neuron, chemicals called **neurotransmitters** are released into the synaptic cleft. These neurotransmitters are housed in water-balloon-like containers called *vesicles*. The vesicles fuse into the pre-synaptic cell membrane when an action potential occurs, releasing their neurotransmitters into the synaptic cleft. The neurotransmitters then bind to **receptors** on the post-synaptic neuron. This is like a key being fit into a keyhole–the neurotransmitters are the keys and the receptors are the keyholes which, when opened, let ions (charged particles) flow in to the post-synaptic dendrite. These ions are negatively or positively charged, and the balance between the total charge of these ions on either side of the cell membrane is what is called the **membrane potential**.[5]

There are different kinds of synapses. The pre-synaptic neurons of **excitatory synapses** release neurotransmitters, which result in the post-synaptic voltage being raised, which makes it more likely that an action potential will occur post-synaptically. *Glutamate* is one of the most common excitatory neurotransmitters. **Inhibitory synapses** release neurotransmitters, which result in the voltage being lowered post-synaptically, and make it less likely that an action potential will occur post-synaptically. *GABA* is the most common inhibitory neurotransmitter.

A neuron can receive both excitatory and inhibitory inputs.[6] As different axons attaching to a neuron

has one axon and multiple dendrites, which allows it to receive information from many other neurons. Other types of neuron include unipolar and bipolar neurons.

[3]Fast communication between long-distance neurons is made more efficient by a fatty white substance, called *myelin sheath*, that wraps around the axons of neurons and insulates them, allowing better conduction of electrical signals

[4]There are two types of synapses, electrical synapses and chemical synapses. Electrical synapses, instead of having a cleft between the post- and pre-synaptic neurons, have a much smaller space called a *gap junction*, which connects the pre-synaptic neuron directly with the post-synaptic neuron, allowing for electrical communication. These synapses allow neurons to fire in synchrony and are important for quick communication between neurons. However, most communication happens via chemical synapses, which transmit much stronger signals and have more permanent effects. The main text focuses on chemical synapses.

[5]This charge is maintained by ion channels that selectively let some ions travel into and out of the cell. Some of these ion channels are called *passive ion channels*, which stay open and allow the constant light flow of Na^+ and K^+ ions through the cell membrane. There are also *gated ion channels*, which are those that open during an action potential and cause a much larger exchange of ions. When these open, the ions released cause changes in the membrane potential of the post-synaptic neuron.

[6]What we are calling excitatory and inhibitory inputs are actually synaptic events that allow ions to rush in or out of the cell, and thus produce excitatory or inhibitory currents. From this standpoint the neuron is like a battery or capacitor and its membrane dynamics can be understood in terms of circuits (the "Rall model" or "cable theory" of the neuron), which can

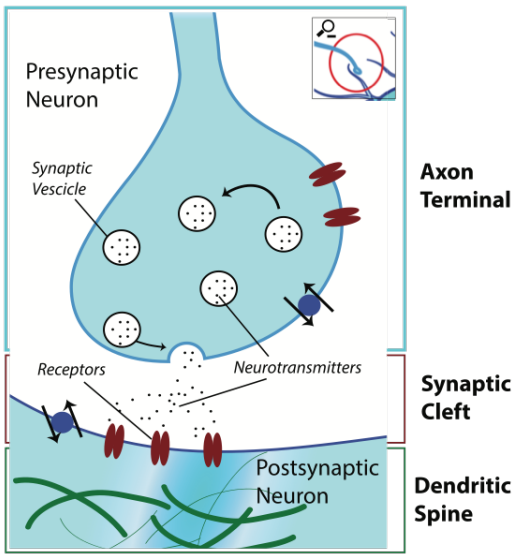


Figure 4.2: Some structures associated with a synapse.

release excitatory neurotransmitters (like glutamate) and inhibitory neurotransmitters (like GABA), the binding of neurotransmitters to receptors on the post-synaptic neuron causes ion channels to open and close, letting different ions in and out. As a result, the neuron's voltage goes up and down. When the voltage at the axon hillock passes a specific membrane potential called the **threshold potential**, an action potential is fired. The process is illustrated in Fig. 4.3. In the left panel, two excitatory synapses are activated successively, and the membrane potential is increased each time until it passes the threshold potential and an action potential is fired.[7] In the right panel, first an excitatory synapses is activated, which raises the membrane potential, and then an inhibitory synapse is activated, which reduces the membrane potential, in effect turning off or "shunting" the impact of the first excitatory input, so that no action potential occurs. Changes in membrane potential due to excitatory and inhibitory events that occur below threshold are sometimes referred to as sub-threshold dynamics.

As more excitatory signals are received, the membrane potential will exceed the threshold potential more often, and action potentials will begin to occur more frequently. Thus we can represent the overall activity of a cell in terms of its **firing rate**, which is measured in number of spikes per unit time, usually spikes per second. A highly "active" neuron is one that produces many spikes per second (e.g. 200 Hertz, which is 200 times per second), while a more dormant or quiescent neuron might only produce a few spikes per second (e.g. 2 Hertz). In a time series plot of such a neuron's membrane potential, spikes for a highly active neuron are close together; for a less active neuron they are farther apart. You can get a feel for this in the Simbrain simulation *spikingNeuronTwoInputs.zip*. In the simulation you can increase the firing rate of the neuron on the left by raising the excitatory input. You can then reduce its firing rate by raising the inhibitory input. See figure 4.4.

These observations are the basis of "rate-coding" models. In these models, the number in a node represents a neural firing rate. Weights in these models capture the idea the neuron sums together excitatory

be precisely described in computational neuroscience models. The membrane potential has an average resting state, usually around -70 mV, which is the difference between the electrical charge inside and outside of the membrane when a neuron is at rest. Excitatory and inhibitory inputs alter this resting state. One metaphor that helps here is that of a tank of water or a graduated cylinder, where the level of water represents the current voltage potential. There is a hole on the side of the tank that constantly lets some water out, and a tube above it that constantly lets some water in. These produce a "leak" current that balances out to a certain standing level in the tank, which is like the resting potential of a cell. That level is roughly where the hole on the side of the tank is. Excitatory inputs briefly open up additional tubes above the tank, letting more water in and raising the water level (the membrane potential), and inhibitory inputs briefly open up additional tubes below the tank, letting more water out and reducing the water level. These transient events lead to fluctuations in the water level which correspond to sub-threshold dynamics of the membrane potential.

[7]Notice the signals occur at spatially distinct dendrites and at different points in time, and that they have a cumulative effect on the membrane potential at the soma. This is known as *spatial and temporal summation*.

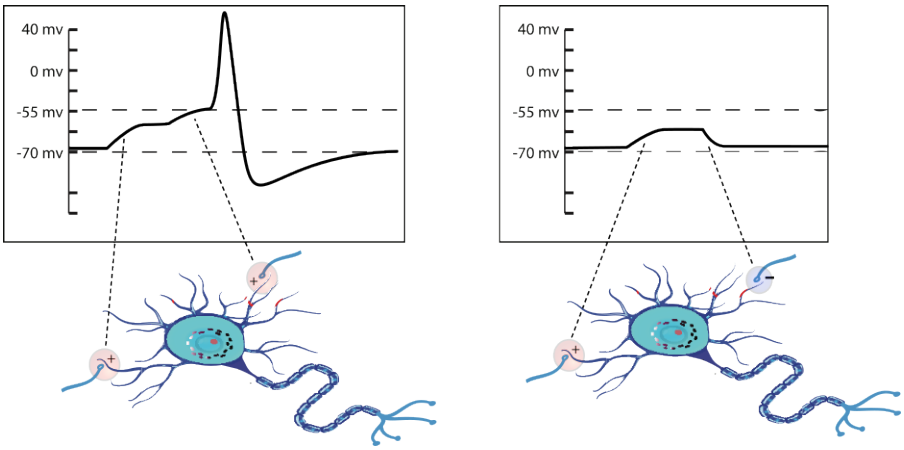


Figure 4.3: (Left) Two successive excitatory inputs increase the neuron's membrane potential beyond the threshold potential, after which it fires an action potential. (Right) First an excitatory input increases the neuron's membrane potential, then an inhibitory input reduces it.

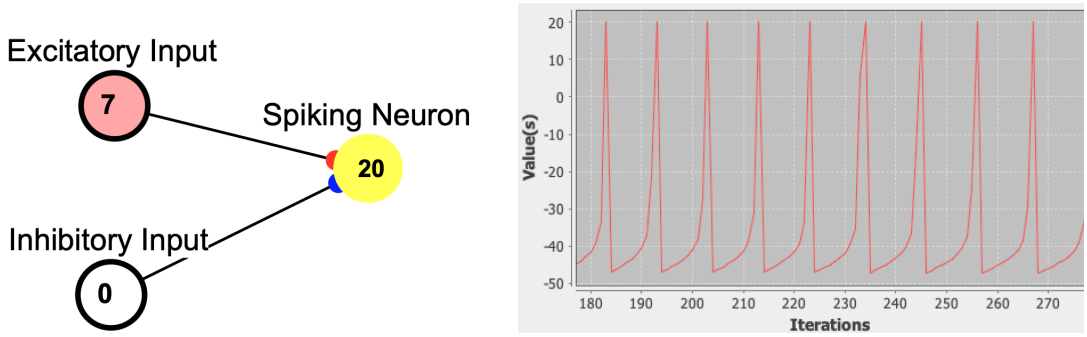


Figure 4.4: (Left) A Simbrain simulation that illustrates how different combinations of excitatory and inhibitory inputs produce different firing rates in an output neurons. As the excitatory input is increased, the firing rate increases, and as inhibitory input is increased, the firing rate decreases. The output neuron is in the middle of a spike at the moment shown.

and inhibitory signals. Rate coding is in the background of many connectionist models, where much of the biology is abstracted away and all that is maintained is the general idea that a weighted sum of inputs determines the output of a node. More complex computational neuroscience models describe the changing membrane potential directly, or simulate the action potential using discrete spiking events (see chapter 18). These models are discussed in the chapter on computational neuroscience and in the chapter on classical nodes and weights.

Synapses are modifiable. For example, **Long Term Potentiation** or LTP occurs when certain synapses transmit information repeatedly in a short time. When this happens the synapse is "strengthened": when an action potential reaches the same synapse after LTP has occurred, the post-synaptic response will be greater than it was before.[8] Long term potentiation is the basis of the Hebb rule, discussed in chapters 3 and 9. The basic idea of the Hebb rule is that "neurons that fire together, wire together."[9]

Synapses can be modified in other ways. Sometimes synapses are weakened via a process of *Long term depression* or LTD. Synapses can be modified in other ways as well, and the study of synaptic plasticity is

[8]The details of LTP are not well understood, but roughly what happens is this: the repeated stimulation of the post-synaptic neuron results in an influx of calcium ions, which has a number of effects. One is the recruitment of additional receptors to the post-synaptic dendrite, so that when neurotransmitters are subsequently released into the synaptic cleft more receptors open and more ions are allowed into the post-synaptic cell.

[9]This can be visualized in several Simbrain simulations, for example *autoassociator1.zip* and *autoassociator2.zip* in the courseMaterials directory.

a major area of research. These changes in synaptic efficacy are thought to be the basis of most forms of learning in humans and animals. The idea that changing connection strengths are the basis of learning is what gives "connectionism" its name, and is fundamental to neural network theory.

4.1.3 Neuromodulators

We mentioned GABA and glutamate above. These are neurotransmitters that support local communication from one neuron to another. Other neurotransmitters—which are sometimes called "neuromodulators"—are connected with circuits that project across larger regions of the brain and have longer-lasting impacts.[10] Some neural network simulations model the effects of these neurotransmitters.

Norepinephrine, or noradrenaline, is produced by neurons in the brainstem and broadcast throughout the brain. It regulates arousal: there is a greater amount of norepinephrine when awake and a decreased amount while asleep.

Serotonin is also produced by cells in the brainstem. Serotonin is involved in attention and complex cognitive function. Low serotonin has been linked to depression. A type of medicine referred to as an SSRI (selective serotonin reuptake inhibitor) causes less of the serotonin released by cells to be taken back into the pre-synaptic neuron, so that more serotonin stays in the synapse and gets used.

Acetylcholine, found in motor neurons in the spinal cord, is responsible for movement and also mediates certain forms of plasticity. Too little acetylcholine can inhibit movement, while too much acetylcholine can cause twitching. Black widows inject a chemical in their bite that promotes the release of acetylcholine, which leads to severe muscle twitching.

Dopamine is a neurotransmitter produced in the basal ganglia (in the "nigrostriatal pathway"), which plays an important role in controlling movement. Shortage of dopamine in the system can lead to *Parkinson's disease*, characterized by an inability to initiate movements. A drug called *L-Dopa* can be used to stimulate the production of dopamine, which helps to even out dopamine levels and alleviate some of the symptoms exhibited by Parkinson's patients. Dopamine is also important in regulating the reward-based learning that occurs in the basal ganglia, which is discussed further below. It is important to note that dopamine is *not* a direct signal of reward (that signal is carried by other opioid-based circuits in the brain), but rather a signal of how much more or less reward than expected was obtained; a kind of error signal. When things are going better than expected, dopamine neurons fire at an increased rate. When things are going worse, they fire at a reduced rate. When you are not expecting a donut (assuming you like donuts and are hungry), the arrival of a delicious donut will cause dopamine neurons to start firing. But as you eat, even if you are still hungry, dopamine neurons stop firing, because your expectations are no longer changing. On the other hand, if you were expecting those donuts when the door opened and you are disappointed to see that your friend forgot to bring them (oh the horror), your dopamine neurons will fire *less* than normal. Thus dopamine is an important signal which can be used to train animals, by encouraging them to do things that lead to unexpected rewards, and by discouraging them from doing things which lead to unexpected disappointment. This signal is key to computational models of the the basal ganglia, discussed below.

4.2 The Brain and its Neural Networks

In this section we describe regions of the brain, functions associated with them (summarized in Fig. 4.5), and give a sense of the computational role they serve in cognition and how they are modeled by neural networks.

It is worth noting at the outset that these associations between brain regions and cognitive functions are somewhat artificial. Most types of cognition are based on circuits that span multiple brain areas. Conversely,

[10] On the relationship between neurotransmitters, neuromodulators, and neurohormones "A neurotransmitter is a messenger released from a neuron at an anatomically specialised junction, which diffuses across a narrow cleft to affect one or sometimes two postsynaptic neurons, a muscle cell, or another effector cell. A neuromodulator is a messenger released from a neuron in the central nervous system, or in the periphery, that affects groups of neurons, or effector cells that have the appropriate receptors. It may not be released at synaptic sites, it often acts through second messengers and can produce long-lasting effects. The release may be local so that only nearby neurons or effectors are influenced, or may be more widespread, which means that the distinction with a neurohormone can become very blurred. A neurohormone is a messenger that is released by neurons into the haemolymph [or, in mammals, into the blood] and which may therefore exert its effects on distant peripheral targets." [19].

most areas of the brain are involved in many kinds of cognition and behavior. For instance, motor regions of the brain are known to be active in body movements, but also participate in movement planning, observation of the movements of others, and even the perception of objects that can be manipulated (i.e., grasped). Thus, when we talk about "language areas" or "decision-making regions", we are discussing regions that are active when the relevant behaviors occur, but these regions are not solely responsible for such tasks. In the same way that neurons work in groups to process information, higher brain areas work together to create complex thought and behavior.

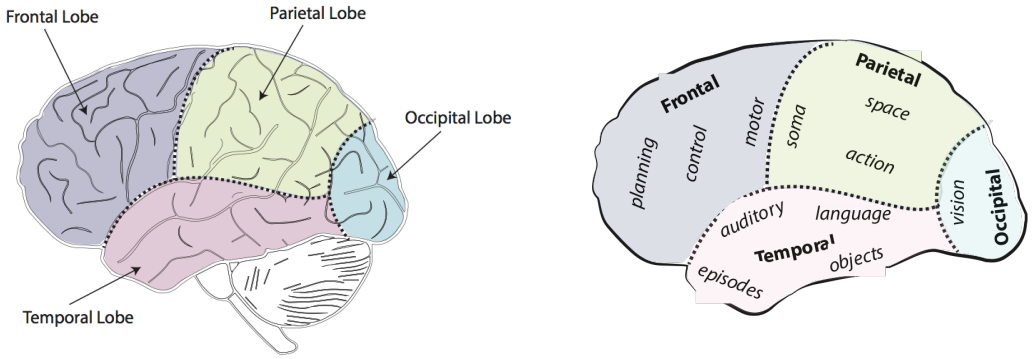


Figure 4.5: (Left) The lobes of the brain. (Right) Rough map of functions, abilities, and conceptual domains associated with major brain areas.

4.2.1 Cortex

Fig. 4.5 (Left) shows the major regions of the brain. Fig. 4.5 (Right) summarizes the functions associated with these areas. These are regions of the **cerebral cortex**, which is the wrinkled outer surface of the brain (cortex literally means "rind", like the outer skin of an orange or lemon). The wrinkling is caused by the cortex folding, like a crumpled piece of paper, in the limited volume of the skull. This folding results in bulges (called "gyri") and valleys (called "sulcuses" or "sulci") on the surface of cortex. The cortex is thought to be involved in the higher processing functions distinctive of complex behavior and intelligent animals. The size of an animal's cortex roughly correlates with the complexity of its behavior and overall intelligence: humans and dolphins have a relatively large cortex, chimps a smaller cortex, rats even smaller, and non-mammals like birds and insects have no cortex at all. The cortex is composed of two hemispheres, the right and the left hemispheres, connected by a structure made up of nerve fibers called the "corpus collosum". Between-hemisphere communication occurs through the corpus collosum. In patients with a neurological disorder called "epilepsy", where too much neuronal firing in the brain leads to seizures, the corpus collosum is often severed (this is called a "collosotomy") to reduce between-hemispheric communication and prevent future seizures. Both the right and left hemispheres are made up of the same lobes (occipital, temporal, parietal, and frontal).

From a computational standpoint, the cortex is the brain's primary long-term memory system, which stores all the many things we know about the world: how we classify objects, our concepts, our beliefs, our memories, our knowledge about our friends and family, our life goals and fundamental cares, almost everything is coded into this massive memory system. Many neural network models are directly or indirectly simulations of our long-term cortical memory system. They model pattern recognition via learning, memory storage and recall, pattern completion, spreading activation, and many other phenomena. In fact, unless otherwise noted, most neural networks are probably ultimately models of how cortex works.

The cortex has dense bi-directional recurrent connections that allow it to reverberate in sustained patterns. It also has long range connections between areas that allow it to produce complex brain-wide patterns or oscillations (though some circuits are also similar to feed-forward networks). In concert with the central thalamic relay station (more on this below), the posterior and parietal parts of cortex reverberate and coordinate sensory input and motor outputs when you engage in most behaviors. The frontal regions manage our plans and actions. Other circuits refine these signals, producing smoother movements (cerebellum),

coordinating sequences of activations to produce reward (basal ganglia), and managing recent memories (hippocampus). Thalamo-cortical oscillations are correlated with consciousness. When you see something and are aware of it, sustained processing in multiple cortical areas is associated with your experience: visual activations are associated with visual awareness, activation in somatic areas is associated with awareness of your body, more distributed activations are associated with inner thoughts, etc. These are sometimes referred to as the neural correlates of consciousness or NCCs.

Learning in this long-term memory store occurs via a mixture of unsupervised and supervised learning. Synapses are updated by LTP and other means, which can be modeled using unsupervised learning algorithms like Hebbian learning and unsupervised architectures such as self organizing maps (see chapter 9). One theory of cortex is that it is a giant collection of internal models in long-term memory: models of physical objects, the people you know, language, etc. These models learn from unsupervised methods, but they also come to have expectations about external inputs and inputs from other models. On this view, most processing in the cortex, and thus most of what we see and hear and understand, is based on what we *expect*, based on our internal models of situations. The signals that flow through cortex are actually error signals–a kind of training signal–which indicate how what we see differs from what we expect. Thus cortical networks incorporate elements of supervised learning (chapter 12). This view, known as the "predictive coding" or "predictive processing" view, originates in part in computational models of visual cortex [100], but it has since developed into a more general view about the structure of perception and cognition and their realization in the brain [22].

In many cortical areas there is a progression from areas that handle low-level sensory processing (e.g. edge detection in primary visual cortex, or tones in primary auditory cortex) to regions that handle more complex pattern recognition, like face recognition. The reverse direction is similar: high level plans are handled by more "interior" networks, while detailed motor movements are handled closer to the output layers of the cortex, that feed to thalamus and then to muscle systems. Thus, many cortical areas have a hierarchical structure, which is precisely what is modeled by deep networks.

4.2.2 The Occipital Lobe

The **occipital lobe** and some of its features are shown in Fig. 4.6. Its most dominant feature is the **visual cortex**, which supports visual processing, including edge detection, color detection, and simple motion detection. Damage to the visual cortex can produce **cortical blindness**, even if the eyes are intact. Processing begins in the eye (in the retina, which is itself a complex neural network), and is then passed along to several structures, most prominently the visual cortex. Processing within the visual cortex occurs in a series of stages, which are thought to correspond to the extraction of increasingly complex features of a visual scene. For example, V1 and V2 process information about edges and form, V4 is involved in processing of color, and area MT plays a role in motion processing.

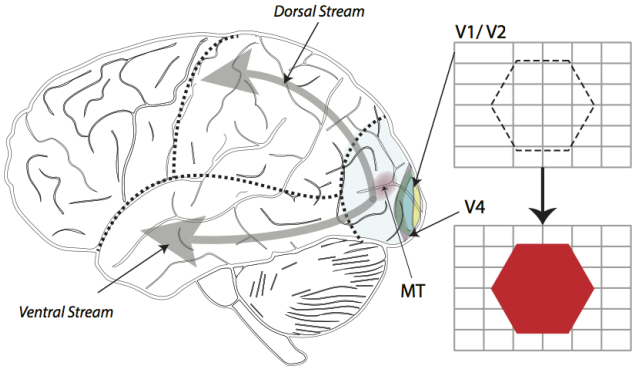


Figure 4.6: The visual cortex and associated structures.

Each of the regions of visual cortex contains something called a **retinotopic map**, which is a full neural map of locations in the retina, where groups of neurons nearest one another process information about nearby areas in visual space. More generally, a *topographic map* is an area of the cortex where sensory information

is processed in a spatially organized manner. We will see that multiple sensory regions contain topographic maps of their respective sensory information. In chapter 9 we will see that some neural network algorithms, like self organizing maps, can automatically produce banks of detectors that are topographically organized.

Information passes out of the visual cortex in two streams: a **dorsal stream** to the parietal lobe, which is involved in coordinating visual and spatial information, and a **ventral stream** to the temporal lobe, which is involved in processing complex visual features of objects and semantic knowledge, i.e. information about what things are (see Fig. 4.6). We will discuss these pathways in more detail below.

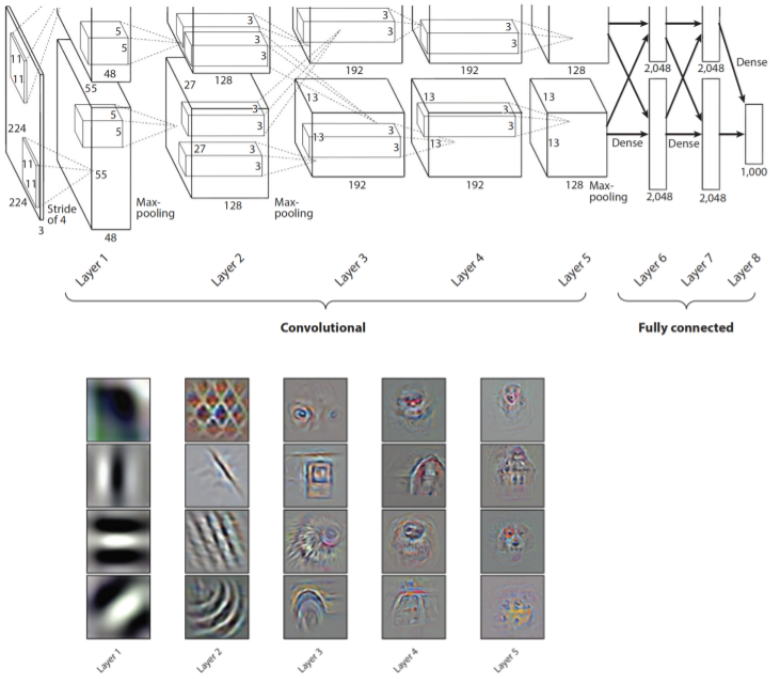


Figure 4.7: A well-known deep neural network architecture AlexNet (based on [64]) whose activations match those of actual brain areas. The top level shows the network architecture, and the bottom panel shows receptive fields (activations that maximally activate specific nodes) of a similar network [45]. Though these networks were developed as an engineering tool to classify images, they do a good job of describing neural activation in V1, V2, V4, and IT.

It has emerged in recent years that deep learning networks are particularly well suited to describing what these areas of the brain do. These networks are trained to recognize images, which is a useful engineering application, but it turns out they do a good job of describing neural activity in the brain. For example, a well known convolutional deep network known as "AlexNet" [64] is shown in the top panel of figure 4.7. It develops topographic maps similar to those in the brain. The activations it produces at its various layers in response to images match the activations of the brain in response to the same images quite well.The earlier layers of the model mimic the response properties and receptive fields of lower levels of processing, like V1, and later layers mimic properties V4 and ventral stream neurons in IT. The exciting thing about these models is that we can produce pictures of their receptive fields, showing precisely what kind of input each neuron learned to respond to.[11] As can be seen in the figure, lower level layers in this kind of network become edge detectors, further downstream layers respond to combinations of these features (compare Selfridge's demons from chapter 3), while IT layers respond to dogs, cats, etc.[12] The use of convolutional networks to model neural activations is further developed in chapter 14.5.

[11]For a visual sense of how these networks work, and what the activations are at different processing layers, see `https://cs.stanford.edu/people/karpathy/convnetjs/`.

[12]There has also been some skepticism about deep network approaches to human vision [14].

4.2.3 The Parietal and Temporal Lobes

The **temporal lobe** is involved in auditory processing and semantic processing (see Fig. 4.8). The **auditory cortex** is in the temporal lobes. Much of the sensory information from the ears is sent to auditory cortex. Primary auditory cortex (A1) contains a **tonotopic map** of the acoustic properties of sounds. That is, neurons in this region respond to preferred frequencies of sound in a similar way to the preferred spatial regions in retinotopic maps. Auditory information is also processed in a somewhat hierarchical fashion, similar to vision. After A1, information passes to the secondary auditory cortex (A2), where sound localization and processing of more complex sound features occurs. When the auditory cortex is damaged, people can experience hearing deficits (they may suffer from "central hearing loss" or cortical deafness) even if the ears are intact. Other parts of the temporal lobe are involved in language processing. **Wernicke's area**, located in the temporal lobe[13], plays an important role in speech understanding. This region is involved in assigning meaning to sounds. Damage to this area can produce Wernicke's aphasia (also receptive or fluent aphasia), where patients are unable to understand either spoken or written language, but where language production often remains intact.[14]

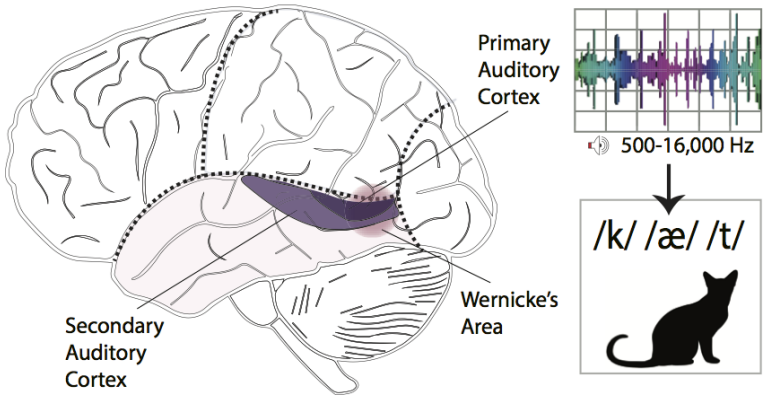


Figure 4.8: Auditory cortex and associated structures.

The temporal lobe also receives connections from the visual processing centers of the occipital lobe, via the ventral stream (Fig. 4.6). The ventral stream is involved in object recognition. For example, the fusiform face area (FFA) in the temporal lobes is connected with the recognition of faces. Damage to this region will cause **prosopagnosia**, an inability to recognize faces. It has since also been found that the FFA is active in bird experts while looking at birds and chess players while recognizing chess board configurations. This suggests that this region is involved in recognition of objects that one has expertise with [12]. Another form of damage to the ventral stream can cause *ideational apraxia*, where patients have difficulty interacting with objects because they can no longer understand what the object is used for.

The **parietal lobe**, shown in Fig. 4.8, is involved in integrating information from multiple regions of the brain, as well as processing information about space. The dorsal stream carries spatial information from the occipital to the parietal lobe. It is involved in spatial attention, reaching, grasping, using tools, and other activities that coordinate visual information with motor behavior. Damage to regions of the parietal lobe can produce a number of problems. Patients with **hemineglect** tend to only pay attention to certain parts of the visual field. Such a person might only eat food one one side of their plate, or draw images on only one side of a page. It is said that a director who had hemineglect produced movies in which the action only happened on one side of the screen. Another form of damage to the dorsal stream will cause *ideomotor apraxia*, which results in difficulty using objects in space. Someone with ideomotor apraxia will have difficulty converting the idea of an action into the action itself. For example, they might have difficulty combing their hair when asked to do so, even if they can identify the hair brush and understand the function of the brush. The difficulty is in the execution of the action. This disorder highlights the role of the dorsal stream in coordinating action in space.

[13] More specifically, the temporal lobe in the dominant hemisphere, which is usually the left hemisphere.
[14] An example of speech in this condition are here: `https://www.youtube.com/watch?v=3oef68YabD0`.

The parietal lobe also receives tactile information from the body via the **somatosensory cortex** (Fig. 4.9), which in turn receives touch and temperature information processed by specialized mechano-receptors on the skin. When the somatosensory cortex is stimulated, people report feelings in specific parts of the body. The somatosensory cortex is a **somatotopic map**, in which nearby regions of neural tissues respond to pressure or temperature on nearby regions of the body. As shown in Fig. 4.9, more sensitive body parts are allocated more space in somatosensory cortex. For instance, the fingers and lips have greater cortical representation than other regions, while the shoulders and trunk have much less. In the somatosensory cortex of a mouse, almost all of the space is allocated to the whiskers, with each whisker receiving a relatively large amount of neuronal space. The primary somatosensory cortex is located right next to the primary motor cortex (discussed below), allowing for quick communication between these regions.

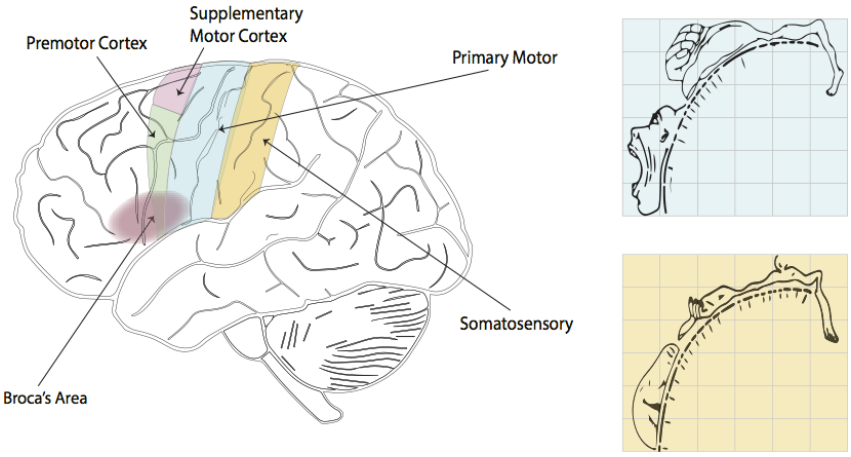


Figure 4.9: Somato-sensory and motor processing.

4.2.4 The Frontal Lobe

The **frontal lobe** is involved in higher-level cognitive functions, or *executive functions*, like the ability to pay attention, select strategies, solve problems, plan actions, make decisions, inhibit or suppress behaviors, and in general control one's behavior. These functional associations rely upon a distributed network of regions including the orbito-frontal, dorso-lateral prefrontal, and ventro-medial areas.[15] The rear-most parts of the frontal lobe, like the **primary motor cortex**, are directly involved in action. In fact, the frontal lobes can be thought of as controlling action on a spectrum from specific movement in the primary motor cortex to increasingly abstract planning and decision making in the front-most parts of the cortex, like the orbito-frontal cortex. Some language processing also takes place in the frontal lobe. **Broca's area** (Fig. 4.9) is responsible for many language functions, including gesture, understanding of action and action-language, and language production. Broca's aphasia (or expressive aphasia) impacts the ability to produce speech, though comprehension can remain intact. Here is an example of someone with Broca's aphasia explaining how they got to the hospital: "Yes... ah... Monday... er... Dad and Peter H... (his own name), and Dad.... er... hospital... and ah... Wednesday... Wednesday, nine o'clock... and oh... Thursday... ten o'clock, ah doctors... two... an' doctors... and er... teeth... yah."[39].[16]

The parts of the frontal lobe closer to the center of the brain are directly involved in controlling the body. Neural outputs to the body originate in the **primary motor cortex** (see Fig. 4.9). When parts of the primary motor cortex are stimulated people contract the relevant muscles of their body. When *premotor cortex* is stimulated, people will actually start to make complex movements, such as grasping. *Supplementary motor cortex* is less well understood and does not include a map of the body in humans, but this region is thought to play an important role in coordinating movement plans, in particular sequences of movements.

[15] The *Ventro-medial prefrontal cortex* has been shown to be involved in the representation of the internal state of the body and the relative value of decisions. *Orbito-frontal cortex* is also involved in decision-making and is thought to play a particular role in assessing reward.

[16] The example is cited in `https://en.wikipedia.org/wiki/Expressive_aphasia`.

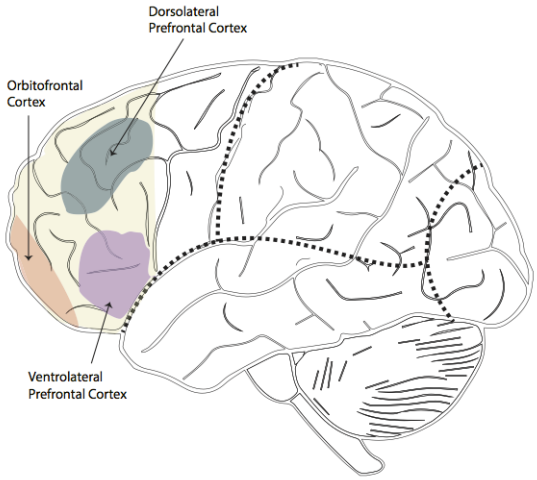


Figure 4.10: The prefrontal cortex.

The dorso-lateral **prefrontal cortex** or PFC is associated with working or short-term memory. It is part of the cortex but has evolved a distinctive structure that supports the unique demands of working memory. Its cells are relatively isolated from other areas, but with dense recurrent connections that facilitate a particular type of neural dynamics, an "attractor structure" (see chapter 10), whereby activations tends to settle into stable patterns for a time, which are maintained in cortical "stripes" [63]. These stripes are thought to encode task information in working memory. Your current plans and goals are maintained by active stripes in your PFC. As you go through your day doing one thing after another–making breakfast, driving to school, reading a book, etc.–different stripes corresponding to current goals are sequentially activated in PFC. Support for this idea is provided by experiments that show that while humans and monkeys maintain goals to look or reach in different directions, specific populations of neurons are active in the PFC. In some neural network models, actively maintained tasks are simulated simply by clamping certain nodes in the on or off position. For example, one node might correspond to reading letters, while another might correspond to saying what color the letters are written in, in a model of a task where you can either read letters or say their color.[17]

Damage to the frontal lobe results in a variety of deficits, including difficulties with impulse control, impaired judgment, personality abnormalities, and an inability to make any decisions at all. A famous case of damage to the frontal lobe is provided by Phineas Gage, a 19th century railroad worker whose skull was pierced by a large iron rod in an explosion. Once the iron rod was removed, Gage retained full cognitive function, and the only prominent change was in his behavior. After the surgery, he had a more difficult time inhibiting certain behaviors, became more hostile, drank excessively, and eventually became homeless.

4.2.5 Other Neural Networks in the Brain

We have been focusing on the cortex, which is by far the most dominant structure in the brain. Many neural network models are basically modeling cortex and how it extracts features from sensory inputs layer by layer, maintains task information in the frontal areas, etc. These are models of different aspects of long-term memory. However, there are also many other specialized circuits that have been modeled by distinctive forms of neural network model. See figure 4.11 for an overview of the structures we discuss here.

The **hippocampus** is a kind of short-to-medium-term memory system attached to the bottom of the cortex.[18] It is associated with memory consolidation, spatial memory, and episodic memory. It has a special neural network structure that allows it to watch what is happening in the cortex, and then build up special "sparse coded" representations.[19] While learning in the cortex is slow, learning in the hippocampus is fast.

[17] These are models of the Stroop effect; see `https://en.wikipedia.org/wiki/Stroop_effect`

[18] In fact it is directly attached to the temporal lobes, and is hard to see as being separate on visual inspection, but its neurons are arranged differently than the neurons in the cortex.

[19] For simulation-based tutorials on computational models of hippocampus see `https://compcogneuro.org/`.

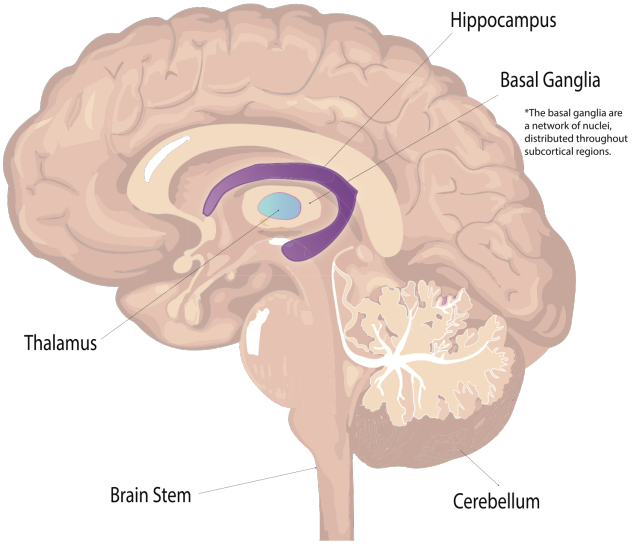


Figure 4.11: Some specific structures in the brain with specific neural network structures.

It can pick up all the things that happen in your day and remember them for a few weeks or even longer. These memories don't always last, and in fact new neurons are constantly being created in hippocampus (it's one of the few parts of the brain where neurogenesis continues into adulthood). Dreams are thought to be mediated by hippocampus, which is why dreaming often involves recent events. When things get repeated enough in hippocampus, they are consolidated into the cortex. It's like it learns a fast representation, and then that either evaporates or if repeated enough, gets transferred to cortex. Damage to the hippocampus can produce various forms of amnesia, e.g. **anterograde amnesia**, now familiar via movies like *Memento*, where characters live entirely in the present and cannot remember things that they learn after the date of their injury. Interestingly, patients with this form of amnesia are often able to create new *procedural memories*, such as a "memory" of how to ride a bike, but are unable to create any new episodic, semantic, or fact-based memories, such as memories of events in the news. Neural network models of hippocampus have been used to study how memories can be consolidated into long term memory. The models can, for example, be used to simulate amnesia.

The **basal ganglia** is an important collection of nuclei beneath the cortex which have a variety of functions, including (in concert with pre-frontal cortex) control of voluntary action, and learning how to take actions that are likely to lead an agent to rewarding stimuli. It is thought to implement a form of **reinforcement learning**, whereby actions that produce reward tend to be reinforced over time, and actions that produce costly outcomes are inhibited (this formalizes older ideas in psychology about operant conditioning). It's a bit like a task scheduler or sequencer, orchestrating extended sequences of activations in the cortex. It is also thought to be involved in deciding what tasks and goals should be loaded into the frontal areas of the brain, determining what tasks are maintained in PFC's stripes, and in what sequence. It implements reinforcement learning in part using the neuromodulator dopamine, discussed above. These same reinforcement learning techniques have also been shown to work well in machine learning.[20] In fact, there was a great deal of excitement in the 1990s when it was first discovered that dopamine neurons in the basal ganglia responded to rewards in the same way as certain variables in reinforcement learning models: more activity when reward is higher than expected; less activity when reward is less than expected [109, 88]. When things go better than we expect, dopamine is released, and synapses are strengthened, reinforcing whatever we have done recently, making us more likely to do the same thing in the same situation in the future. Thus the dopamine system and the basal ganglia "sequencer" learns to execute sequences of goals and actions that tend to produce reward in the long run and avoid punishment.[21]

[20] Reinforcement learning was, for example, used in Alpha Go (mentioned in chapter 1), the first neural network to beat a professional human Go player `https://deepmind.com/research/alphago/`

[21] Some of these ideas can be studied using the actor-critic model in Simbrain (available from the simulation menu). Links to operant conditioning can be studied using the Rescorla-Wagner and operant conditioning simulations). The relation to frontal

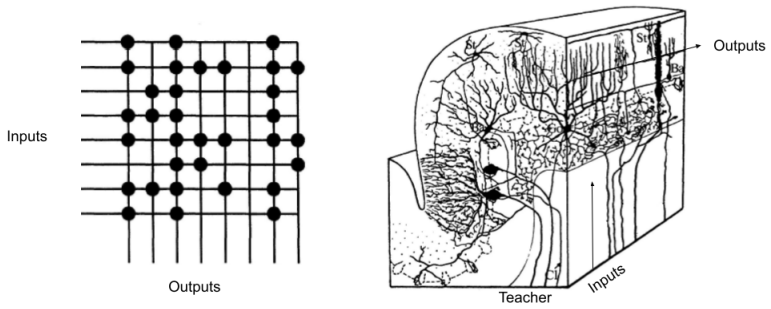


Figure 4.12: The cerebellum as a pattern associator, mapping sensory states to motor actions. (Left) An input-output architecture: synapses where input lines and output lines overlap can be turned on when a teacher signal turns on. (Right) Associated areas of the cerebellum thought to correspond to these functions.

The **cerebellum** is involved in fine motor control (it also has cognitive functions but these are less well understood). When it is damaged, movement becomes jerky and ballistic (**ataxia**). Neural network models treat the cerebellum as a massive pattern associator, or even as a neural "lookup table". It has a structure that made it an attractive target for computational neuroscientists in the late 1960s and early 1970s [75]. As can be seen in figure 4.12, it receives many inputs and produces many outputs, and there are also neurons that seem to climb up and surround certain neurons, suggesting that they carry an error signal used to update certain synapses. This led to the idea that it was a pattern associator trained by supervised learning, a theory which remains popular, though the issue is not settled. One thing it certainly does is learn to associate bodily and sensory inputs with motor outputs, which helps produce rapid and smooth action sequences. It's as if the coarse-grained motor plans produced by the cortex and basal ganglia are "smoothed" by this cerebellar associative map.

The **thalamus** is a subcortical region responsible for processing and relaying information between the cortex and sensory and motor structures on the body. Information from most sensory modalities passes through the thalamus on the way to the cortex. It is sometimes called the "gateway to the cerebral cortex." Recurrent loops between the thalamus and cortex (*thalamo-cortical loops*) produce wide-spread synchronized patterns of activity in the cerebral cortex which are, as noted above, associated with conscious experience.

Finally, more fundamental functions of the brain, like the control of the lungs, heart, and sleep, take place in the **brain stem**. Damage to the brain-stem often results in death.

lobes and task maintenance is explored in CECN models at `https://compcogneuro.org/`.

Chapter 5

Activation Functions

Jeff Yoshimi, Scott Hotton

As discussed in the introduction, artificial neural networks are comprised of nodes connected by weights. The nodes are usually pictured as circles and are associated with a number called an activation, while the weights are represented by lines connecting the nodes and are associated with a number called a strength. In Simbrain, node activations correspond to the colors of the nodes and to the number inside the nodes, and weight strengths correspond to the color and size of the filled disks at the end of the lines connecting nodes. In figure 5.1 (Left), three nodes are connected to one node via three weights.

When a neural network simulation is run, node activations and weight strengths change. Neural networks have *dynamics*, which describe a changing pattern of activation across the nodes and (in some cases) a changing pattern of strengths of across the weights. To see these dynamics in Simbrain, look for the triangular "play" button ▷. When you press it you usually see node activations change, and in some cases weight strengths. In this chapter we describe some of the rules that govern changing node activations.[1] These are based loosely on the physiology of neurons and action potentials, which was discussed in chapter 4.

Rules for updating node activations make use of an **activation function**, which sets the activation of a node based on the values of incoming node activations and the weights connecting them together. These are classical rules that have been used in many kinds of simulations since the early days of neural networks. There are many other rules for updating neural networks–some geared more towards computational neuroscience, some towards engineering–but even today these classical rules are frequently used.[2]

5.1 Weighted Inputs and Activation Functions

We will represent the activation of the j^{th} node of a network by a_j. The strength of the weight connecting the j^{th} node to the k^{th} node will be denoted by $w_{j,k}$. This notation is illustrated in figure 5.1. Activation a_j of node j is updated by first computing the **weighted input** to node j (roughly: the weighted sum of activations from other incoming nodes) and then passing that value through an activation function denoted by f. The basic flow of operations is shown in figure 5.1 (Right). In this section we discuss weighted inputs in more detail and then consider three of the most common forms for the activation function: threshold, linear, and sigmoid.

A basic feature of a node's activation is that it is a function of the activations of other nodes attached to it, and also the strengths of the intervening weights. This can be computed as a simple linear combination of incoming activations to a node, and intervening weights, which is called the **weighted input** (or "net

[1] When you open up a dialog to train a network, there is another play button that is used to modify the weights. When these buttons are pressed the dynamics of nodes and weights is simulated. In chapters 6, 9, and 12, we discuss the rules governing changes in weight strengths.

[2] To get a sense of the diversity of functions available, try editing a few nodes in Simbrain and changing the "update rule" drop down box. As you change the selection, you will notice that the parameters available to you change. You can also wire together a small network and just see what happens when you use these rules. Several neurally realistic "spiking" activation rules are included in Simbrain, which are discussed further in chapter 18.

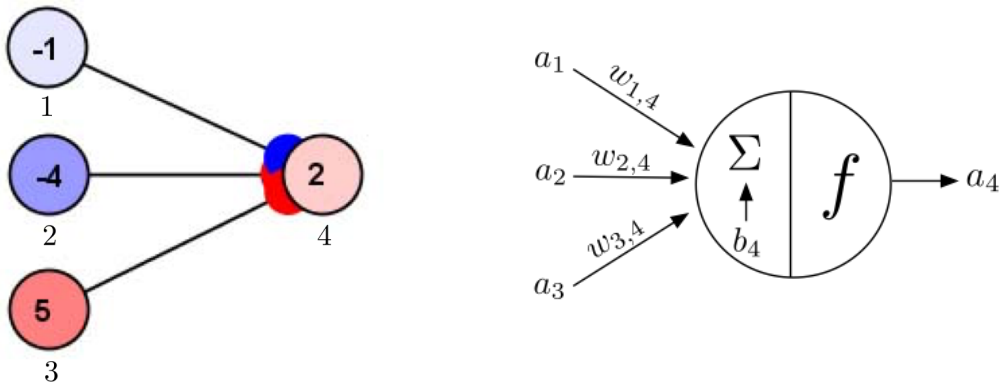


Figure 5.1: (Left) A simple neural network with three nodes attached to one node via three weights. (Right) Schematic of the same network to illustrate the notation being used here. Nodes $1, 2, 3$ are connected to node 4. In this example, $a_1 = -1$, $a_2 = -4$, $a_3 = 5$ and (though weight strengths are not visible) $w_{1,4} = -1$, $w_{2,4} = 1$, $w_{3,4} = 1$, $b_4 = 0$ and the activation function is linear with a slope of 1, so that $a_4 = 2$. Σ represents the weighted inputs, and f represents the activation function. A network like this is included with Simbrain as *simpleNet.zip*

input") to a node. That is, each incoming activation to a node is multiplied by the intervening weight strength, and these products are added together. We denote the weighted input to the k^{th} node as "n_k".

Nodes are also associated with a **bias**, which is a fixed and unweighted input to a node (it can also be treated as an input via a fixed weight whose strength is 1). It can be thought of as a property of the node itself (in Simbrain it is set by editing a node's properties), which determines the node's baseline activation.

The value of the weighted inputs n_k to a node is computed by multiplying the activations of incoming source nodes a_j by the intervening weights $w_{j,k}$, and adding any bias b_k. In the example shown in figure 5.1, n_4 can be expressed as:

$$n_4 = \sum_{j=1}^{3}(a_j w_{j,4}) + b_4 = (a_1 w_{1,4}) + (a_2 w_{2,4}) + (a_3 w_{3,4}) + b_4$$

If we have N inputs, then the value of n_k can be concisely expressed as:

$$n_k = \sum_{j=1}^{N} a_j w_{j,k} + b_k$$

If you are not familiar with the symbol "$\sum$", it is described in this footnote.[3]

Some examples of computations of weighted input are provided in section 5.6.

[3]We use "sigma" notation to represent the addition of several numbers. Sigma is the name of the Greek letter for 's', which is short for 'sum'. The letter has uppercase and lowercase forms. The uppercase sigma is used to denote summation. For instance, the sum of the cubes of the first 4 positive integers can be written as $\sum_{j=1}^{4} j^3 = 1^3 + 2^3 + 3^3 + 4^3 = 1 + 8 + 27 + 64 = 100$. The uppercase Greek letter 'Σ' tells us that we are to perform a summation. Beneath Σ it says '$j = 1$'. This tells us that we are going to increment j starting with the value of 1. Above Σ it says '4' which tells us to stop incrementing j at the value 4. The j^3 to the right of Σ tells us to cube each of the values for j. We start of with $j = 1$ and cube it. Next, increment j, cube it, and continue until $j = 4$. Finally, we sum all four of these cubed numbers. Often we use a letter for the final value of the incremented variable so that our formulas will work with sums with an arbitrary number of terms. For example: $\sum_{j=1}^{N} i^3 = \left(\frac{N(N+1)}{2}\right)^2$.

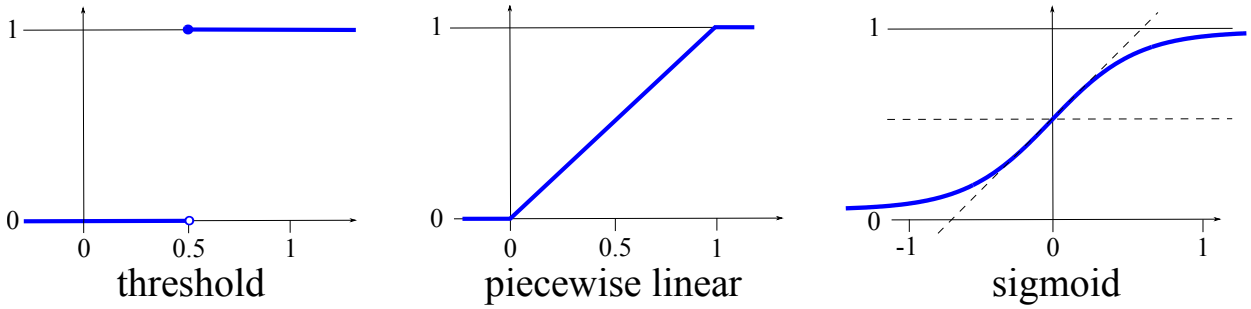


Figure 5.2: The graphs for three activation functions. In each of the graphs, the horizontal axis is the weighted input, n_k, and the vertical axis is the activation, a_k. Left: A threshold activation function with $(u, \ell, \theta) = (1, 0, 0.5)$. Middle: A piecewise linear activation function with $(u, \ell) = (1, 0)$. Right: A sigmoid activation function with $(u, \ell, m) = (1, 0, 1)$. The inflection point is located where the horizontal dotted line $a_k = (u + \ell)/2$ intersects the vertical axis. The tangent line to the graph at the inflection is shown by the dotted line with a slope of 1. The graph converges to 1 as the weighted input increases and to 0 as it decreases.

We now consider activation functions (labeled 'f' in figure 5.1), which associate weighted inputs with activation values. Activation functions are sometimes also called "transfer functions". Recall that in mathematics, a function f associates a unique output to each input. We say the input is mapped to the output. For example, if $f(x) = x^2$ then

The input 1 is mapped to the output 1.	$f: 1 \mapsto 1$	$f(1) = 1^2 = 1$
The input 2 is mapped to the output 4.	$f: 2 \mapsto 4$	$f(2) = 2^2 = 4$
The input 3 is mapped to the output 9.	$f: 3 \mapsto 9$	$f(3) = 3^2 = 9$

Two parameter values will be used repeatedly in this section: an upper value u, and a lower value ℓ (of course, we assume $\ell < u$). In Simbrain, the upper and lower values u and ℓ are set in the *upper bound* and *lower bound* fields of a neuron, respectively.[4] It will sometimes be useful to refer to these parameter values using vector notation. For example a statement like $(u, \ell) = (1, -1)$ means that $u = 1$ and $\ell = -1$.

5.2 Threshold Activation Functions

We begin with a simple activation function, the **threshold activation function** (it is also called a "binary" activation function, a "step function", or a "Heaviside function").[5] These nodes can take on one of two values, an upper value u and a lower value ℓ, and they are thus binary valued nodes. Which value the node takes on depends on whether weighted input is greater than or less than a threshold value θ. Threshold activation functions are inspired by real neurons, which operate in a discrete, on-off fashion, either firing or not firing an action potential depending on a summation of incoming excitatory and inhibitory currents (see section 4.1.2).

If the value of the weighted input to a threshold activation function is less than a threshold value θ, then the activation of a threshold node takes on a lower value ℓ. If the value of the weighted input is greater than or equal to θ then the activation of a threshold node takes on an upper value u.

$$a_k = f(n_k) = \begin{cases} \ell & \text{if } n_k < \theta \\ u & \text{if } n_k \geq \theta \end{cases}$$

[4]Except in the case of the binary threshold neuron, where the u is an "on value" and ℓ is an "off value." These values are sometimes also referred to as "ceiling" and "floor".

[5]The term "binary" refers to the fact that the node can only take on one of two values. The term "step function" refers to the way the function appears when plotted (see figure 5.2). The term "Heaviside" is a reference to Oliver Heaviside who used these functions to study electrical circuits.

The graph of a threshold activation function is shown in figure 5.2. When the weighted input increases from below the threshold of 0.5 to above the threshold, the function's output jumps from the lower bound, 0, to the upper bound, 1.

5.3 Linear Activation Functions

A **linear activation function** computes activation as a simple linear function of weighted input. To compute the activation we simply multiply the weighted input by the positive number, m. This number is the slope of the linear function.

$$a_k = f(n_k) = m \cdot n_k$$

m is usually set to 1 so that a linear activation function is the identity function (which takes every input to itself). This means that the activation of a node when it uses a linear activation with $m = 1$ just is the weighted input to the node.

A related type of activation function is a *piecewise linear* function. (For this discussion we assume that $m = 1$). For a piecewise linear function if the value of the weighted input is less than ℓ, then the activation is set to the lower value ℓ. If the value of weighted input is greater than u, then the activation is set to the upper value u. For values between the upper and lower bound, the activation is the weighted input:

$$a_k = f(n_k) = \begin{cases} \ell & \text{if} \quad n_k < \ell \\ n_k & \text{if} \quad \ell \leq n_k \leq u \\ u & \text{if} \quad n_k > u \end{cases}$$

A piecewise linear function is basically a clipped or truncated linear function. As the weighted inputs to a node get very large or small, the activation is truncated to the upper or lower value. This is biologically realistic (a membrane potential can't achieve arbitrarily high or low values; a neuron can't fire at arbitrarily high rates). Also, it is not uncommon for a neural network algorithm to produce uncontrolled growth or decay, which can be prevented by the simple act of truncating the signal for certain values.[6]

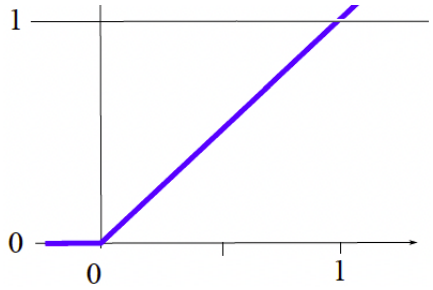


Figure 5.3: Graph for the rectified linear or "ReLU" activation function.

A special case of a piecewise linear activation function is a *rectified linear unit* or **ReLU** activation function. The terminology comes from electronics where rectifiers are often used to truncate the negative part of an alternating current to produce a direct current. They cut off all negative values, replacing them with a 0, and leave the weighted input unchanged otherwise (see figure 5.3).[7] Thus, they are a piecewise linear function with no upper bound:

$$a_k = f(n_k) = \begin{cases} 0 & \text{if} \quad n_k \leq 0 \\ n_k & \text{otherwise} \end{cases}$$

[6] In Simbrain, nodes are piecewise linear with $m = 1$ by default, so that by default a node simply displays weighted inputs, assuming weighted inputs fall within the upper and lower bounds of the neuron. A linear node can be converted into a regular, non-piecewise linear node by turning off *clipping*.

[7] The rule can be even more concisely stated as the maximum value between 0 and n_k, or $\max(0, n_k)$.

ReLU has a number of useful properties that have made it extremely popular since the deep learning revolution of the 2010s (section 3.7). To fully appreciate these advantages it helps to compare it to the piecewise linear activation function, which is in turn similar to the sigmoid activation function, discussed below. Piecewise linear and sigmoid activation functions are only really responsive to inputs between their upper and lower bounds. For large weighted inputs, they just max out at some number, like 1, limiting their expressive power. Whatever large input you throw at them, they always just output 1. But ReLU can take on *any* positive value, which gives it greater representational power. This in turns makes it more useful in training deep networks.[8] On the other hand, the clipping at 0 is nice, because it maintains a non-linearity (many layers of unclipped linear nodes are mathematically equivalent to one layer of linear nodes, so additional layers don't help). The clipping also removes all negative activations, making overall activation in a large network more sparse, so that different inputs produce nicely distinct activation patterns. In fact, multiple varieties of ReLU function are now available, in particular "GeLU", which allows some representation of negative values, is also easy to compute, and has well-defined derivative for every weighted input. In Simbrain, a ReLU unit can be approximated with a linear activation function whose *lower bound* is 0 and whose *upper bound* is a large number.

5.4 Sigmoid Activation Functions

A **sigmoid activation function** can be thought of as a smoothed version of a piecewise linear activation function. The sigmoid functions get their name from the Greek letter for "s", and their graphs are sometimes called "s-curves" because they resemble a stretched-out letter "s". This is directly visible in figure 5.2. As weighted inputs increase, the activation approaches the upper value u, which is usually 1. As weighted inputs decrease, the activation approaches the lower value ℓ, which is usually 0 or -1. When weighted inputs are near the inflection point of the function at 0, activation changes rapidly. Since outputs are always "squeezed" between u and l, it is sometimes called a "squashing function."

Sigmoid functions can be used to describe natural processes that involve a continuous increase from one value to another. Suppose a bacterium is placed in a Petri dish and we observe how quickly bacteria grow in the dish. At first, the population grows slowly. It then rapidly expands for a time, filling up the dish. Eventually, there is no more room in the dish, and the population size levels off, or plateaus. Something similar happens to the firing rate of a neuron as it receives more input currents. The firing rate rises slowly, then rapidly, and then approaches a maximum value.

Sigmoid activation functions are notable for being differentiable everywhere. If you have not taken calculus, this intuitively means that the function smoothly changes everywhere; there are no discontinuous breaks or hard edges. Notice that the threshold and piecewise linear activation functions in figure 5.2 are not differentiable everywhere. The threshold function has a discontinuity at the threshold value, and the piecewise linear function does not change smoothly at the truncation points (u, u) and (ℓ, ℓ). The ReLU function is discontinuous at $(0, 0)$. The differentiability of the function means that the derivative can be used, which in turn allows certain operations to be performed on sigmoidal nodes that would not otherwise be possible. This led to one of the major innovations in the history of neural networks: the move from linear networks (networks of nodes using linear activation functions) to networks using sigmoidal nodes, which could be trained using backpropagation. This in turn led to an increase in the use of neural networks in the 1980s (see chapter 3).

We will not focus on how to compute a sigmoid function here (the function can be computed in several different ways). We focus on the qualitative properties of sigmoid functions. However, to give a flavor of the idea, here is one common way by which some sigmoid functions can be computed:

$$a_k = f(n_k) = \frac{1}{1 + e^{-4\,m\,n_k}}$$

(Note that this version of the function incorporates a slope parameter m but not an adjustable upper or lower value. Adding u and ℓ parameters would make the function even more complex). This version of

[8] The derivative goes to 0 for large values of sigmoids, which prevents that node from contributing to learning due to the way errors are "back-propagated." See chapter 13 and the discussion of vanishing gradients. Also, the derivative is 0 for $n_k \leq 0$, and 1 otherwise, except at 0 (the point of discontinuity), where the derivative is not defined. A helpful discussion from Leo Dirac that outlines other virtues of ReLU is here: `https://youtu.be/S27pHKBEp30?t=1165`.

the sigmoid function is often called a "logistic function." There are other versions of the sigmoid function, for example, one based on the arctangent function from trigonometry, and another based on the hyperbolic tangent function.[9]

In general, we need three values to specify a sigmoid function. We need its upper bound, u, its lower bound, ℓ, and a positive slope, m. For the formula above, $(u, \ell) = (1, 0)$ and m can have any positive value. When the weighted input to a sigmoidal function equals 0, the resulting activation will be exactly half way between the upper and lower bounds, *i.e.* $(u + \ell)/2$, or in this case $(1 + 0)/2 = .5$. The point $(0, (u + \ell)/2)$ on the graph of the sigmoid function is called the *inflection point* of the function. Each sigmoid function is symmetrical about its inflection point. The value of m is the slope of the tangent line to the graph of the sigmoid function at the inflection point. In other words, m tells us how steeply the graph of a sigmoid function rises.

As the weighted input is increased indefinitely above 0, the value of a sigmoid function converges to its upper bound u. As the weighted input is decreased indefinitely below 0, its value converges to its lower bound ℓ. The larger m is, the more rapidly the sigmoid function converges to its bounds. Figure 5.2 shows the graph of a sigmoid function with $(u, \ell, m) = (1, 0, 1)$.

As the slope parameter is varied, the shape of the sigmoid function changes. When the slope is large or "steep", the sigmoid will begin to look like the threshold function. When the slope is near 0, it will begin to look more like a linear function.

5.5 Non-local activation functions

The activation functions we have discussed thus far are local in the sense of being computable based on information available at the neuron, by way of its connections to other neurons. However, in some cases a neuron's activation can only be computed based on the state of other neurons in a group.[10] These are often called "layers", as in a type of node layer. In Simbrain this is represented by drawing a border around the neurons (these structures are called neuron groups in Simbrain), as in figure 5.4.

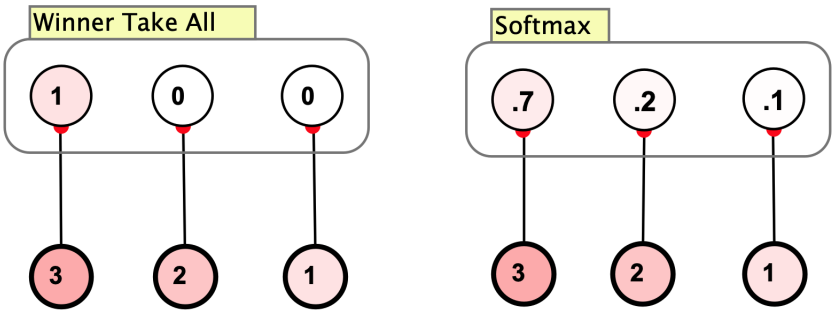


Figure 5.4: A winner-take-all and a softmax activation function. All weight strengths are 1 and the connections are one-to-one so the weighted inputs are just the input values visible in the figure. The boundary around the nodes in the "neuron group" indicates that they are all taken into consideration when computing the activations.

A simple example is a **winner-take-all** activation function like the one shown in the left panel of figure 5.4. In this case all the weighted inputs to a group of neurons are computed, and the activation for the node which received the greatest weighted input (in this example, the node on the left) is set to a prescribed winning number like 1, and activations for the rest of the nodes are set to a losing number, usually 0.

[9] In Simbrain this function is captured by the "Sigmoidal (Discrete)" update rule. Different implementations of the function can be selected within Simbrain.

[10] This is not biologically realistic, since a real neuron can only do things based on information available to it locally, but these functions often have useful formal and mathematical properties, and can in some cases be justified as valid simplifications or approximations of network activity.

$$a_k \quad = \quad f(n_k) = \begin{cases} \text{win\_value} & \text{if } n_k = \max(n_1, n_2, \ldots, n_m) \\ 0 & \text{otherwise} \end{cases}$$

This obviously requires considering every neuron in the group, even if they are not directly connected to each other. Although this is unrealistic, it is a reasonable approximation of a network of inhibitory and excitatory connections (compare the discussion of pools in an IAC networks in section 2.4).

Another well-known example of a non-local activation function is the **softmax** activation function, which is often used for classification tasks. It converts the weighted inputs to a group of nodes into a probability distribution over those nodes–that is, a set of values all of which are between 0 and 1 which sum to 1. These activations are often interpreted as probabilities associated with a group of class labels. For example if a softmax layer of a network trained to recognize visual inputs has nodes for cat, dog, and raccoon, then the softmax activations might be .05, .9, and .05, indicating a 5% probability that the input pattern represents a cat, a 90% probability that it represents a dog, and a 5% that it represents a raccoon.

The formula for softmax is this:

$$a_k = f(n_k) = \frac{\exp(n_k/T)}{\sum_{j=1}^{N} \exp(n_j/T)}$$

That is, for each neuron k in the softmax layer, its activation a_k is computed as follows. The weighted input n_k is exponentiated[11], and these exponentiated values are normalized (the summation on the bottom, which shows why the softmax activation function requires information about all nodes in a layer, and is thus part of a neuron group in Simbrain). T is a temperature parameter which scales the weighted inputs before they are exponentiated, and is discussed further below.

The exponentiation step increases the influence of larger input values, producing a more sharply peaked distribution. The exponential function is shown in figure 5.5. Think of the x-axis as weighted input. As weighted input gets bigger, the exponential gets much bigger, so that larger values are accentuated. An example is shown in the right panel of figure 5.4. Note how the inputs are 3, 2, and 1, but the softmax probabilities are .7, .2, and .1. The exponentiation step has accentuated the difference between the inputs.

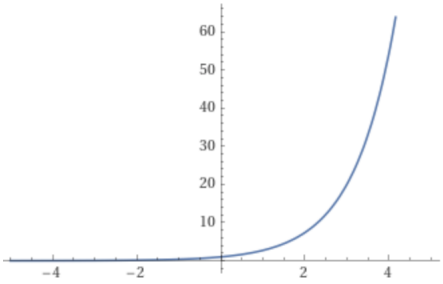


Figure 5.5: The exponential function from -5 to 5. As can be seen, negative and smaller values are more or less flattened to being near 0, but positive values grow rapidly and are in a sense accentuated.

The temperature T is a scaling parameter that controls how sharply peaked the softmax distribution is (see figure 5.6). When $T = 1$ it is a default softmax function. For high temperature values, larger than 1, the softmax distribution becomes more dispersed. Think of a "hotter" more active system, doing slightly more random or reckless things. In classification tasks, this can be interpreted as less confidence about the classification. For small temperature values between 0 and 1 the distribution becomes more sharply peaked. Think of a "cooler" more restrained system. In a classification task this can be interpreted as

[11] In the context of softmax layers weighted inputs–the raw inputs to the layer–are sometimes called "logits", in reference to the concept of a logit in statistics (`https://en.wikipedia.org/wiki/Logit`), which is a quantity that can can take on any value from negative to positive infinity, and which is then transformed into a probability.

more confidence about the predictions. One application of temperature is to large language models (chapter 17), where turning the temperature up can encourage the system to produce less predictable responses and turning it down can lead it to rely more on its training data.

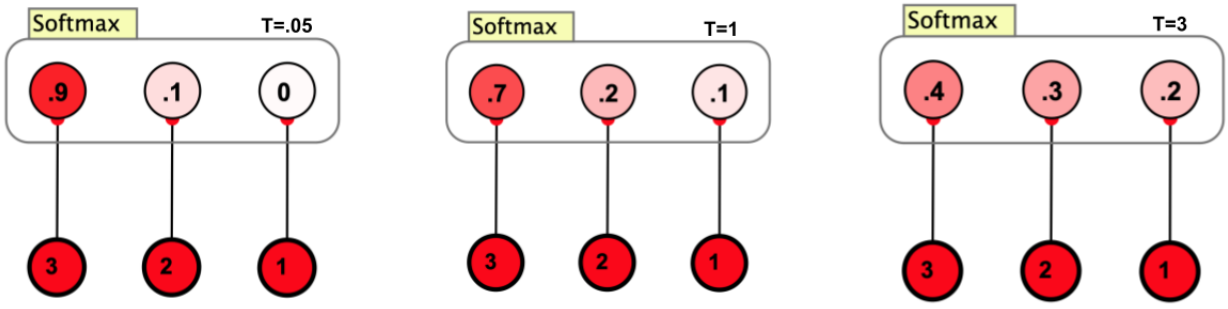


Figure 5.6: Softmax for relatively low, medium, and high temperatures of .05, 1, and 3, respectively. At lower temperatures the probability distributions over nodes is more sharply peaked and at higher temperatures it is more spread out.

5.6 Exercises

All of these exercises can be tested using a simple network of 3 nodes connected to one node, and adjusting the output node as appropriate. The 3 input nodes must be clamped.[12]

1. Consider the network shown in Fig. 5.1. We want to determine the weighted input to node 4 shown on the right-hand side of the network, *i.e.* we want the value of n_4. First, we identify the values for the activations of the input nodes, the weights, and the bias on node 4.

The activations on the input nodes: $(a_1, a_2, a_3) = (-1, -4, 5)$
The weights: $(w_{1,4}, w_{2,4}, w_{3,4}) = (-1, 1, 1)$
The bias: $b_4 = 0$

Next, we substitute these values into the formula for weighted input:

$$n_4 = \sum_{j=1}^{3} a_j w_{j,4} + b_4 = a_1 \cdot w_{1,4} + a_2 \cdot w_{2,4} + a_3 \cdot w_{3,4} + b_4 = (-1)(-1) + (-4)(1) + (5)(1) + 0 = 2$$

The weighted input to node 4 is 2, **Answer:** $n_4 = 2$.

2. Consider the network shown in Fig. 5.1. We want to determine the weighted input to node 4 shown on the right-hand side of the network, *i.e.* we want the value of n_4. First we identify the values for the activations of the input nodes, the weights, and the bias on node 4.

The activations on the input nodes: $(a_1, a_2, a_3) = (-1, -4, 5)$
The weights: $(w_{1,4}, w_{2,4}, w_{3,4}) = (-1, 1, 1)$
The bias: $b_4 = 0$

[12] A clamped node does not get updated, it just has a fixed activation; if the input nodes were themselves linear or had some other activation function, then they would, for example, immediately go to 0 at every update, because the weighted input to the input nodes is 0.

Next, we substitute these values into the formula for weighted input:

$$n_4 = \sum_{j=1}^{3} a_j w_{j,4} + b_4 = a_1 \cdot w_{1,4} + a_2 \cdot w_{2,4} + a_3 \cdot w_{3,4} + b_4 = (-1)(-1) + (-4)(1) + (5)(1) + 0 = 2$$

The weighted input to node 4 is 2, **Answer:** $n_4 = 2$.

3. Suppose we have the same network as in exercise 2, except $b_4 = 1$. What is the weighted input to node 4? **Answer:** $n_4 = 3$.

4. Suppose again that we have the same network as in exercise 2 except this time the activations are $(a_1, a_2, a_3) = (0, 0, 0)$. What is the weighted input to node 4? **Answer:** $n_k = 0$.

5. Suppose some node, call it node k, has a threshold activation function described by $(u, \ell, \theta) = (1, 0, 0.5)$ (the same as in figure 5.2). And suppose $n_k = 2$. What is the activation of node k? Since $n_k = 2$, and since $2 > 0.5$, the activation takes the upper value of 1. **Answer:** $a_k = 1$.

6. Suppose node k has a linear activation function with $m = 3$ and weighted input $n_k = 2$. What is the activation of node k? **Answer:** $a_k = 6$.

7. Suppose node k has a linear activation function with $m = 1$ and weighted input $n_k = -0.5$. What is the activation of node k? **Answer:** $a_k = -0.5$.

8. Suppose node k has a piecewise linear activation function with $(u, \ell, m) = (1, 0, 1)$ and weighted input $n_k = 10$. What is the activation of node k? **Answer:** $a_k = 1$.

9. Suppose node k has a ReLU activation function and weighted input $n_k = -10$. What is the activation of node k? **Answer:** $a_k = 0$.

10. Suppose node k has a ReLU activation function and weighted input $n_k = 19$. What is the activation of node k? **Answer:** $a_k = 19$.

11. Suppose we have the same network as in exercise 10, except $n_k = .8$. What is the activation of node k? **Answer:** $a_k = .8$.

12. Suppose node k has a sigmoid activation function with $(u, \ell, m) = (1, 0, 1)$. This is the sigmoid function shown in Fig. 5.2. Consult that graph. And suppose the weighted input is 0.75 ($n_k = 0.75$). What, approximately, is the activation of node k? Find 0.75 on the horizontal axis and find the vertical coordinate of the corresponding point on the graph. **Answer:** $n_k \approx 0.9$.

13. Suppose we have the same network as in exercise 12 except the weighted input is 0. What is the activation? **Answer:** $a_k = 0.5$.

14. This is a combined exercise that requires you to determine the weighted input and activation for a single node. Suppose we have a network with two input nodes (labeled 1 and 2) connected to a third node (labeled 3).

The activations on the input nodes: $(a_1, a_2) = (1, -1)$
The weights: $(w_{1,3}, w_{2,3}) = (-1, 1)$
The bias: $b_3 = 1$

And suppose node 3 has a linear activation function with $m = 3$. What is the activation of node 3? First, we compute the weighted input to node 3:

$$n_3 = a_1 \cdot w_{1,3} + a_2 \cdot w_{2,3} + b_3 = (1)(-1) + (-1)(1) + 1 = -1 - 1 + 1 = -1$$

Next, we compute the activation from the weighted input.

$$a_3 = m \cdot n_3 = (3)(-1) = -3$$

Answer: $a_3 = -3$.

15. Same as exercise 14 but:

The activations on the input nodes: $(a_1, a_2) = (-1, 1)$
The weights: $(w_{1,3}, w_{2,3}) = (0, .5)$
The bias: $b_3 = 0$

And suppose node 3 has a piecewise linear activation function with $(u, \ell, m) = (1, -1, 2)$ What is the activation of node 3? First, we compute the weighted input to node 3:

$$n_3 = (-1)(0) + (1)(.5) + 0 = .5$$

Then

$$a_3 = m \cdot n_3 = (2)(.5) = 1$$

Answer: $a_3 = 1$.

16. Compute activation for node 3:

The activations on the input nodes: $(a_1, a_2) = (1, 1)$
The weights: $(w_{1,3}, w_{2,3}) = (0, 4)$
The bias: $b_3 = 1$

And suppose node 3 has a linear activation function with slope 4. **Answer:** $a_3 = 20$.

17. Compute activation for node 5:

The activations on the input nodes: $(a_1, a_2, a_3, a_4) = (1, 1, -1, -1)$
The weights: $(w_{1,5}, w_{2,5}, w_{3,5}, w_{4,5}) = (0, 0, 1, 1)$
The bias: $b_5 = 0$

And suppose node 5 has a threshold activation function with $(u, l, \theta) = (1, -1, 0)$. **Answer:** $a_5 = -1$.

18. Suppose node k has a sigmoid activation function and consider three possible values for the weighted input, *i.e.* $n_k = 0$, $n_k = 2$, and $n_k = 2.5$. How can we design the sigmoid activation function so that the activation for $n_k = 0$ is 0.5 while the activation for $n_k = 2$ and $n_k = 2.5$ is far below 1? The key idea is to decrease the slope of the sigmoid function thereby "stretching out" the S shape. First, we let $u = 1$ and $\ell = 0$. This forces the activation to be 0.5 when $n_k = 0$ regardless of the slope m. Next, we set $m = 0.0001$. The activation in response to $n_k = 2$ is around 0.50020 and in response to $n_k = 2.5$ is around 0.50025, both of which are far below 1.

Chapter 6

Linear Algebra and Neural Networks

Jeff Yoshimi, Scott Hotton

In this chapter, we review some basic concepts of linear algebra with an emphasis on how they can be applied to the study of neural networks. This can be viewed as a transition from the formal structure of single nodes and weights, to the formal structure of *lists* and *tables* of nodes and weights. In particular, we consider vectors and matrices, which allow us to describe the behavior of groups of nodes and weights in a compact way.

Linear algebra also facilitates a powerful geometric framework for *visualizing* the structure and dynamics of a neural network. The properties of a set of inputs and whether they can be properly classified is an example of something that is more intuitively understandable when the input vectors are visualized as points in a space. Chapter 5 notes that whenever the "play" button ▷ is pressed in Simbrain, some dynamical process is simulated. The framework of linear algebra makes it possible to visualize the changing activations of a set of nodes, or the changing strengths of a set of weights, as a moving point in a space. This approach to thinking about neural networks uses *dynamical systems theory*, discussed in chapter 10, and can be used to think about many features of neural network models in an intuitive way.[1]

6.1 Vectors and Vector Spaces

Linear algebra is the study of vector spaces. Vector spaces are abstract mathematical systems that turn out to be extremely useful for describing the structure and dynamics of neural networks. A **vector space** is a collection of objects called **vectors** along with mathematical operations that we can perform with the vectors.[2] For instance, we can add two vectors together to get another vector. There is also a type of multiplication that can be performed between a vector and a **scalar**. The term "scalar" is used for those numbers that are allowed to be multiplied with a vector. We will focus on the case where scalars are real numbers, like 2, -1.2, or 5.9. If the scalars are the set of real numbers, then the vector space is called a *real vector space*. We will only work with real vector spaces. A more formal definition for a vector space is discussed in Sect. 6.11.

We will represent vectors as ordered lists of scalars.[3] Each of the scalars in the list is called a **component** of the vector. A list with 2 components is called an *ordered pair*. A list with 3 components is called an *ordered triple*. More generally, a list with n components is called an *n-tuple*. We can refer to the members of a vector in this sense as its "first component", "second component", etc. A vector in the sense of an n-tuple is often written out as a comma-separated list of numbers surrounded by parenthesis. For example, here are

[1] For a quick demonstration of this way of visualizing network dynamics, try running the simulation *highDimensionalProjection.bsh*. The dynamics of the network is visible in the projection component.

[2] You may have heard that vectors are geometric objects that have a magnitude and a direction. You may have seen them represented by an arrow or directed line segment. These different points of view on vectors supplement rather than contradict each other.

[3] Many classes of mathematical objects satisfy the formal definition of a vector space, and thus many objects can be vectors. The ordered lists we consider here are an especially convenient type of vector.

four vectors:

$$(0,0) \quad (0,1) \quad (1,0) \quad (1,1)$$

We sometimes adopt the convention of writing vectors using bold-faced lower-case letters, for example $\mathbf{a}_i$ for a vector of input values, $\mathbf{t}$ for a vector of target values, or $\mathbf{a}_4$ for the activation vector in the fourth node-layer of a feed-forward network. By contrast non bold-faced, italic lower-case letters are usually reserved for components of vectors, and the subscript indicates which component. For example, a_2 could designate the second component of an activation vector $\mathbf{a}$.

When the components of a vector are written horizontally, from left to right, it is called a *row vector*. The ordered pairs shown above are row vectors. The components can also be written out vertically, from top to bottom, in which case the vector is called a *column vector*.[4] Commas are usually not written in column vectors because it is clear what the components are. For example, here are four column vectors:

$$\begin{pmatrix}0\\0\end{pmatrix} \quad \begin{pmatrix}0\\1\end{pmatrix} \quad \begin{pmatrix}1\\0\end{pmatrix} \quad \begin{pmatrix}1\\1\end{pmatrix}$$

The number of components in a vector can be any positive integer. If there is only one component, then the vector is essentially just a scalar.

For any positive integer n, the set of all n-tuples forms a vector space. The integer n is called the **dimension of a vector space**. For example, the set of all of ordered pairs of real numbers is a 2 dimensional real vector space $\mathbf{R}^2$. The set of all ordered triples is a 3 dimensional real vector space $\mathbf{R}^3$. The set of all possible activations for a network with 172 nodes is a subset of $\mathbf{R}^{172}$.

The components of a vector can be thought of as the coordinates of a point in Euclidean geometry. The components of a vector can be used to locate a point by starting at the origin and moving parallel to each axis by the amount specified by the corresponding component. To locate the point corresponding to the vector $(3,4)$, for example, we move 3 units to the right along the horizontal axis and 4 units upwards along the vertical axis. In this way a 2 dimensional real vector space can be thought of as an Euclidean plane (see figure 6.1).[5]

For 2-dimensional spaces, finding your way around is a bit like using an "Etch a sketch" toy[6], where you move a tracing point around in a space using one knob for left-and-right (x-values) and one knob for up and down (y-values). If you plot the activation space of two nodes in Simbrain and adjust the activations of the two nodes, this analogy is useful for understanding what is happening.

The number of dimensions of the vector spaces that arise in the study of neural networks can be much larger than 3. We we can not geometrically visualize these higher-dimensional spaces directly. To work mathematically in these spaces, it is helpful to keep in mind that our starting point is the n-tuples, which are just lists of numbers. From this standpoint, the 4 dimensional real vector space we work with is just the set of all 4-tuples of real numbers, and the 5 dimensional real vector space we work with is just the set of all 5-tuples of real numbers. Here are some vectors in a 5 dimensional vector space:

$$(0,-1,1,0.4,9) \qquad (-1,2,4,-3,9) \qquad (0,0,0,-1,-1) \qquad (0,-1,0,-1,0)$$

We can keep going. The vectors that make up a 100 dimensional vector space are just lists of 100 numbers. The vectors that make up a billion dimensional vector space are just lists of a billion numbers.

Real vector spaces with more than 3 dimensions cannot be seen directly, but objects in them can be *projected* to lower dimensional real vector spaces where they can be visualized. We will discuss methods of projection from spaces with more than 3 dimensions in Sect. 6.2.[7]

[4] The choice of whether to use a row or a column vector to represent an abstract vector is primarily a matter of convenience or convention.

[5] There is a legend (probably fabricated, but pedagogically useful nonetheless) that the philosopher René Descartes came up with his proof that given a point in the plane there are unique coordinates for that point (and given a pair of coordinates there is a unique point in the plane) while observing flies on his ceiling. He noticed that the position of the flies on the ceiling could be described by superimposing a kind of grid on the ceiling—for example: there's a fly at $(3,4)$, 3 units to the right, and 4 units up; there's a fly at $(-2,-1)$, 2 units to the left, and 1 unit down; and there's another at $(4,-2)$, 4 units to the right, and two units down (see figure 6.1).

[6] `https://en.wikipedia.org/wiki/Etch_A_Sketch`.

[7] Here again the Simbrain simulation *highDimensionalProjection.bsh* is helpful. When you run the simulation, a sequence of points in a 25 dimensional space appears. Each point corresponds to a vector. If you hover the cursor over any one of the points, you will see the list of 25 numbers (the 25 activation levels for the network) that correspond to that point.

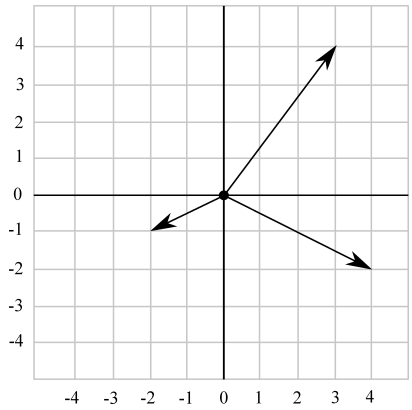


Figure 6.1: Each vector in a 2 dimensional vector space is associated to a point in the Euclidean plane by treating the components of the vector as the coordinates of the point. Try to find the vectors $(3, 4)$, $(-2, -1)$, and $(4, -2)$.

6.2 Vectors and Vector Spaces in Neural Networks

Vectors are frequently used to describe lists of activations, weights, and other quantities associated with neural networks.

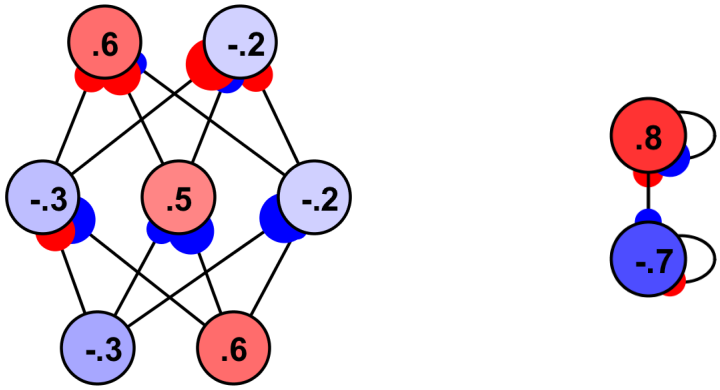


Figure 6.2: A feed forward and recurrent network in Simbrain. Try to identify the dimensionality of the activation space, input space, hidden unit space, output space, and weight spaces of each network. Left: A feed-forward neural network with activations showing. Right: A 2-node recurrent network with activations showing.

The activations of a neural network's n nodes can be described by an **activation vector** with n components, one for each activation value. For example, if we index the nodes of the feed-forward network in figure 6.2 (Left) from the bottom to the top and left to right (as in figure 6.6), then that networks' activation vector is $(-0.3, 0.6, -0.3, 0.5, -0.2, 0.6, 0.2)$. This is a vector in a 7 dimensional vector space. A vector space of activation vectors is called an **activation space**. The feed-forward network in figure 6.2 (Left) network has a 7 dimensional activation space.

Activation spaces are especially useful in studying recurrent networks. If we index the nodes of the recurrent network in figure 6.2 (Right) from top to bottom (as in figure 6.6), then its activation vector is $(0.8, -0.7)$. This is a vector in a 2 dimensional activation space. As the network changes, its activations change, and so we have a changing activation vector. We can picture this as a moving point in a 2 dimensional space.

In addition to describing the state of all of a network's nodes by an activation vector, we can describe

certain *subsets* of its nodes using activation vectors. In the feed-forward network in figure 6.2 (Left), for example, we can describe the activations of the input nodes as an *input vector* $(-0.3, 0.6)$ in a 2 dimensional **input space**. We can describe the activations of the hidden nodes as a vector $(-0.3, 0.5, -0.2)$ in a 3 dimensional *hidden unit space*. We can describe activations of the output nodes as an *output vector* $(-0.6, -0.2)$ in a 2 dimensional **output space**. Recall from chapter 1 that a table of data is a simple environment for a neural network. This table will sometimes contain a set of input vectors, which can be thought of as a set of points in the input space of a network. It can also contain a set of target vectors, which describes how we want the network to respond to input vectors by producing specific output vectors. Many problems in neural network theory can be understood in terms of properties of the input and output space.

Vectors and matrices are sometimes referred to in bold-faced letters, with a subscript indicating more information. So an input vector can be $\mathbf{a}_1$ for node layer 1 or just $\mathbf{a}_{input}$

We can also talk about vectors of weights, or **weight vectors**, which exist in **weight spaces**. The feed-forward network in figure 6.2 has 12 weights. The strengths of those weights is given by the vector

$$(-2, 1, -1, 0.9, -1, -1.2, 1, -2, 0.7, -1, 2, 2.1)$$

in a 12 dimensional weight space (see figure 6.6). The recurrent network has 4 weights whose current strengths is given by the vector $(1.1,\ 2,\ 1, -2)$ in a 4 dimensional weight space. In the chapters on supervised and unsupervised learning (chapters 12 and 9), we will see that it can be helpful to think of learning in terms of movement in a weight space. As the weights of a network are changed or "trained" we have a moving point in weight space. Points in weight space can be associated with an error value, which makes it possible to define an *error surface* over a weight space. Supervised learning can often be understood as finding low points on this error surface.

It can also be useful to talk about a **fan-in weight vector** (the list of weight strengths for the set of weights attaching to a node), and a **fan-out weight vector** (the list of weight strengths for the set of weights exiting a node). A version of the networks in figure 6.2 with zeroed out activations, labeled node indices, and weight strengths is shown in figure 6.6 below. Some sample weight vectors for these networks are:

$$\begin{aligned}
\text{Feed forward network, neuron 3 fan-in} \quad & (1, -2) \\
\text{Feed forward network, neuron 3 fan-out} \quad & (-2, 0.9) \\
\text{Feed forward network, neuron 7 fan-in} \quad & (0.9, -1, -1.2) \\
\text{Recurrent network, neuron 2 fan-in} \quad & (1, -2)
\end{aligned}$$

Some of these weight vectors have 2 components, some have 3 components. Of course for larger networks, fan-in and fan-out vectors can be in higher dimensional weight spaces.[8]

6.3 Dimensionality Reduction

How can we visualize sets of vectors that have more than three components? For example here are nine vectors in a 6 dimensional space:

$$\begin{array}{lll}
(2,\ 0,\ 0,\ 0,\ 0,\ 0), & (0,\ 0,\ 2,\ 0,\ 0,\ 0), & (0,\ 0,\ 0,\ 0,\ 2,\ 0) \\
(1,\ 1,\ 0,\ 0,\ 0,\ 0), & (0,\ 0,\ 1,\ 1,\ 0,\ 0), & (0,\ 0,\ 0,\ 0,\ 1,\ 1) \\
(1, -1,\ 0,\ 0,\ 0,\ 0), & (0,\ 0,\ 1, -1,\ 0,\ 0), & (0,\ 0,\ 0,\ 0,\ 1, -1)
\end{array}$$

We can't directly visualize these vectors since we only live in a 3 dimensional world but we can project them down to a lower dimension. Figure 6.4 shows the projection of these vectors down to 2 dimensions. Each vector above corresponds to one point in the figure. Notice that by visualizing the points we can immediately

[8]Notice that the fan-in weight vectors for the hidden units of the feed-forward network have the same number of dimensions as the input vectors. The input vectors and hidden layer fan-in weight vectors live in the same vector space. This fact is useful sometimes.

see a structure that is very hard if not impossible to see just by looking at the list of vectors. This is how we deal with unwieldy high dimensional data.

A projection is a mapping from a higher dimensional space (sometimes called the "upstairs" space or "total space") to a lower dimensional space (sometimes called the "downstairs" or "base" space).[9] A method for producing a projection is a **dimensionality reduction** technique. We are all familiar with projections insofar as we have seen globes, which are 3 dimensional objects, projected down to paper, which are 2 dimensional objects.

There are different ways of projecting globes to pages, each of which introduces distinct types of distortions. Even so, we still generally get a sense of what of the objects' shapes are. The geometric relationship between various regions in 3 dimensional space can be seen by just looking at a 2 dimensional map. For example, in a standard Mercator projection of the Earth (figure 6.3), Antarctica and Greenland look huge, and things are especially distorted at the two poles, farthest away from the equator.

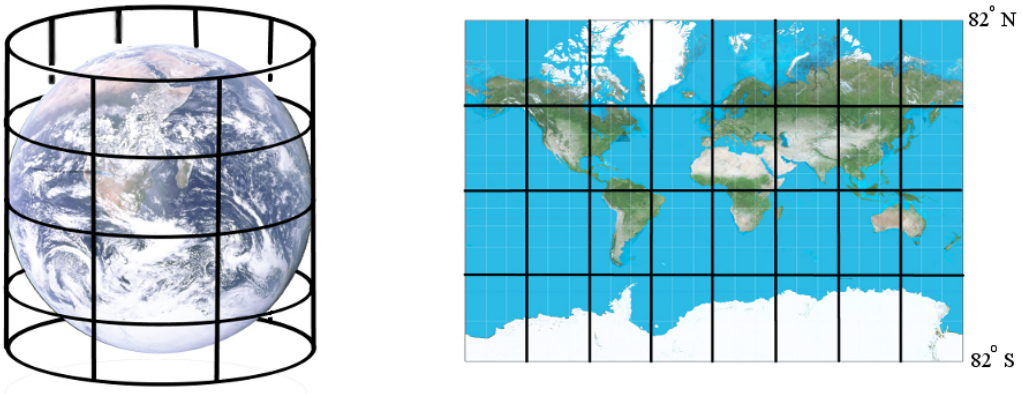


Figure 6.3: The Earth's surface in 3 dimensional space is rendered as a flat, 2 dimensional surface by the Mercator projection method. Most of the distortion produced by the projection occurs near the Earth's poles so small regions around the poles are cropped out from the maps. Most of the continents and oceans undergo little distortion by the projection which made it a popular projection method for making maps of the Earth.

We can still use the projection to get a sense of the layout of the Earth. How are we able to do this? One reason is that certain *topological* properties of the Earth's surface–that is, properties involving continuity–are preserved by the projection. For example, when we see on the map that Los Angeles and San Francisco are on the same coast, then we know we can sail along the coast to get from one city to the other. San Francisco is closer to Merced than it is to Los Angeles, but when we see on the map that Merced is not on the coast, then we know that we can not sail from San Francisco to Merced. Even though distances on the map have been distorted slightly we can still use it to make travel plans.

We can use the method of projection to visualize even higher dimensional spaces that can not be seen by human eyes. A somewhat exotic example is shown on the right of figure 6.4. This is the projection of a 2 dimensional surface in a 4 dimensional space down to a 3 dimensional space. The 4 dimensional space is the state space for a spherical pendulum, and the surface is the set of all states of the pendulum that have the same energy and angular momentum. We can see it resembles the surface of a donut with a groove along the side.

Another example is shown on the left of figure 6.4. It consists of three circles that intersect at a single point (called a "bouquet of three circles"). In this example, the three circles are perfectly round and lie in three mutually perpendicular planes, but we can not see this bouquet of circles directly with our eyes. We also can not see, directly with our eyes, that this bouquet of circles forms a symmetrical figure in a 6 dimensional space. Although the projection distorts the figure a little and we lose some of the roundness

[9] This is not a formal definition but it will suffice for our purposes. Also note that we focus on vector spaces, but the concept of a projection (and of a dimensionality reduction technique) applies to other types of spaces besides vector spaces.

of the circles in the projected image, we can still see the symmetry of the overall figure. The software that was used to do this projection is part of Simbrain (the "projection plot"). This plot can be used to visualize structures in the higher dimensional spaces associated with many neural networks.

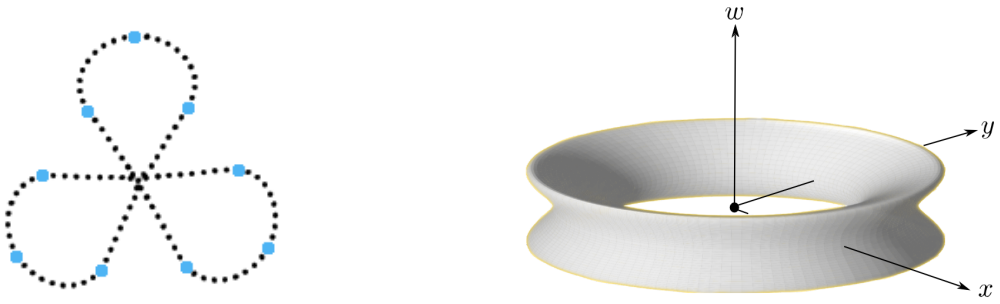


Figure 6.4: (Left) The projection of a symmetrical curve in a 6 dimensional space so that we can see its symmetry. The nine vectors listed at the beginning of section 6.3 are shown as nine large blue dots. (Right) A symmetrical surface in a 4 dimensional space is projected so that we can see its symmetry.

There are many different methods for projecting data from high dimensional spaces to lower dimensional spaces, and the field as a whole is called "dimensionality reduction". Each projection method has its pros and cons, and each one introduces different forms of distortion. But by using several such projections one can often get a good sense of the structure of some high dimensional data.[10]

6.4 The Dot Product

The **dot product** is a simple but important function defined for pairs of vectors in a vector space.[11] The dot product is different from scalar multiplication (scalar multiplication and the scalar product are defined in section 6.11). The dot product is a function of two vectors, whereas scalar multiplication is a function of a vector and a scalar. The dot product gets its name from the fact it is represented by a large dot: $\bullet$. It is also common to say that we are "dotting" one vector with another.

The dot product is computed by multiplying each of the corresponding components of a pair vectors, and summing the resulting products. For example

$$\begin{aligned}
(1,2,3) \bullet (4,5,6) &= 1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 32 \\
(0,0,0) \bullet (4,5,6) &= 0 \cdot 4 + 0 \cdot 5 + 0 \cdot 6 = 0 \\
(2,3,-1) \bullet (-1,1,1) &= 2 \cdot (-1) + 3 \cdot 1 + (-1) \cdot 1 = 0 \\
(1,1,1,1,1) \bullet (1,1,1,1,1) &= 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 = 5
\end{aligned}$$

Clearly the product of any vector with the zero vector (the vector whose components are all 0) is 0. However, the dot product of two non-zero vectors can also be 0. It turns out that if the dot product of two non-zero vectors is 0, then the vectors are *orthogonal* (perpendicular) to each other. It might be hard to tell right away that the vectors $(2,-1,1,-3,1,1)$, $(1,2,3,1,1,-1)$ are orthogonal to each other, but a quick calculation shows us

$$(2,-1,1,-3,1,1) \bullet (1,2,3,1,1,-1) = 0$$

so they must be orthogonal.

[10] The three methods used in Simbrain are described here: `http://hisee.sourceforge.net/about.html`. Other methods of projection are available in this free Matlab toolbox: `http://homepage.tudelft.nl/19j49/Matlab_Toolbox_for_Dimensionality_Reduction.html`.

[11] The dot product is more formally known as the "scalar product." The scalar product gets its name from the fact that its value is a scalar. The scalar product is a member of a general class of functions known as "inner products". Inner products are used to express geometric relationships between vectors.

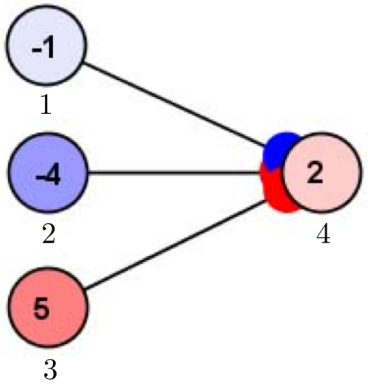


Figure 6.5: Simple feed-forward network with nodes labeled. The dot product can be used to compute the weighted inputs to node 4.

Notice that we can concisely represent the weighted input (see chapter 5) to a node using the dot product. The weighted input is just the dot product of the fan-in weight vector of the node with the input activation vector plus a bias term. For example, if the weight vector for node 4 in figure 6.5 is $(-1, -1, 1)$, then the net-input is

$$n_4 = (-1, -1, 1) \bullet (\ -1,\ -4,\ 5) = 1 + 4 + 5 = 10$$

6.5 Vector Spaces as Metric Spaces

A vector space can have a metric, which is a way to define the distance between any two points. The usual metric for a real vector space is the Euclidean metric, which can be expressed in terms of the dot product. The Euclidean metric uses the concept of a norm, denoted by double bars $||\cdot||$, which measures the "length" of a vector. Specifically, the norm of a vector $\mathbf{x}$ is the square root of the dot product of the vector with itself: $||\mathbf{x}|| = \sqrt{\mathbf{x} \bullet \mathbf{x}}$. With this metric the distance between any pair of vectors in $\mathbf{R}^n$:

$$\mathbf{x} = (x_1, x_2, \ldots, x_n) \qquad \mathbf{y} = (y_1, y_2, \ldots, y_n)$$

is:

$$||\mathbf{x} - \mathbf{y}|| = \sqrt{(\mathbf{x} - \mathbf{y}) \bullet (\mathbf{x} - \mathbf{y})} = \sqrt{\sum_{j=0}^{n} (x_j - y_j)^2}$$

There are many metrics for every vector space but usually we use the Euclidean metric.

The upshot of this is that we can interpret the vector spaces associated with neural networks–activation spaces, input spaces, weight spaces, etc.–as giving us a sense of how far apart relevant vectors are, and thus how similar the things they represent are. Points nearby one another in an input space correspond to similar inputs: similar smells, similar visual inputs, similar words relative to a word embedding (chapter 8), etc. Points nearby one another in a weight space are similar configurations of weights. These interpretations are often emphasized in analyses of neural networks (for example, see chapter 15), and in fact the spaces associated with neural networks are sometimes called "similarity spaces". An example which illustrates the importance of this way of thinking is figure 9.4.

6.6 Matrices

Another object studied in linear algebra is a **matrix**, which is a rectangular array of numbers arranged into rows and columns: basically a table of values. Here is an example of a matrix:

$$\begin{pmatrix} 1 & 9 & 7 \\ 5 & 3 & 2 \\ 0.3 & -1 & 0 \\ 0 & -0.4 & 0 \end{pmatrix}$$

It is conventional to describe matrices by stating the number of rows and columns they have, in that order. The example above is a 4×3 matrix because it has 4 rows and 3 columns.[12] Each row of a matrix is called a **row vector** and each column is called a **column vector**. The matrix above has four row vectors and three column vectors.[13]

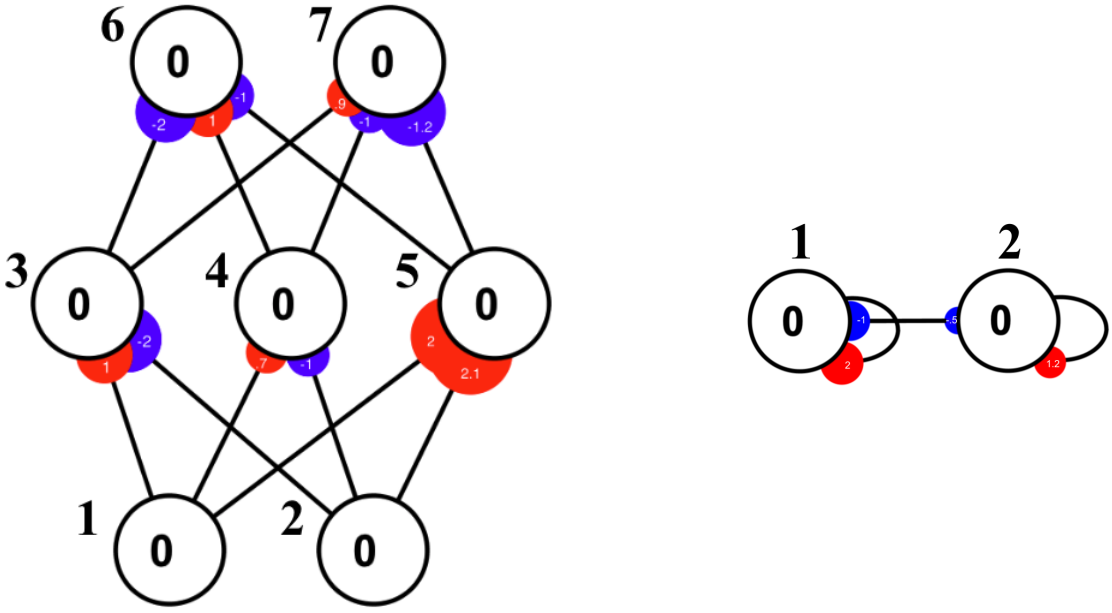


Figure 6.6: Feedforward and recurrent networks with nodes labelled, and weight strengths shown. Each weight layer of the feedforward network and the weights of the recurrent network can be associated with weight matrices.

We adopt the convention of writing matrices using bold-faced upper-case letters, for example $\mathbf{W}$ or $\mathbf{R}$. By contrast, non bold-faced, italic lower-case letters are reserved for entries in a matrix, and subscripts indicate which row and column. For example $w_{2,3}$ could be the scalar at the second row and third column of a 2×3 matrix $\mathbf{W}$.[14] In the matrix above, for example, $w_{1,3} = 7$

[12] The notation for vectors typically includes a comma-separated list of the vector's components. The notation for matrices typically does contain not commas. A matrix's components are only aligned into rows and columns without any extra characters to separate them. Otherwise we think of vectors as special cases of matrices. A vector with n components can be represented as either a $1 \times n$ matrix or as an $n \times 1$ matrix. So far we have been representing vectors as $1 \times n$ matrices.

[13] Note that vectors are matrices and matrices are vectors! As already noted, vectors are a special kind of a matrix, a matrix with one row or one column. Conversely, matrices are technically a kind of vector, since they satisfy the formal definition of a vector (they exist in vector spaces called "matrix spaces"). However, it will be easier for us to follow standard practice and treat these as separate kinds of mathematical objects.

[14] In physics and mathematics it would be more conventional to use upper-case letters like $W_{2,3}$ of the matrix $\mathbf{W}$ for this purpose, but we here follow more standard conventions in discussions of neural networks.

6.7 Weight Matrices

Matrices are often used to represent the weights of a network. This facilitates a compact way of describing many of the computations involved in updating a neural network. The weights of a neural network can be represented by a matrix by labeling the rows and columns of a neural network with indices $1, \ldots, n$ for rows, and $1, \ldots, m$ for columns. Then we can represent the strength of a weight from node j to node k as the value in the j^{th} row and k^{th} column of a weight matrix.[15] This implies that a source-target weight matrix representation of a set of weights will have as many rows as source neurons and as many columns as target neurons.

We first consider weight layers in feedforward networks. Each weight layer of a feedforward network can be represented by its own weight matrix $\mathbf{W}_{i,j}$ where i is the source node layer and j is the target layer. For example, $\mathbf{W}_{1,2}$ or $\mathbf{W}_{input,hidden}$ is the matrix connecting the input node layer to the hidden node layer on the left of figure 6.6. The column and row headings are shown in bold text and match the node labels in 6.6. Notice that there are as many rows as input neurons and as many columns as hidden layer neurons. You can see, for example, that the weight $w_{2,4}$ from node 2 in the input layer to node 4 in the weight layer is in the row labeled 2 and the column labeled 4: -1. On the right is a more standard matrix representation of $\mathbf{W}_{1,2}$.

	3	**4**	**5**
1	1	0.7	2
2	−2	−1	2.1

$$\begin{pmatrix} 1 & 0.7 & 2 \\ -2 & -1 & 2.1 \end{pmatrix}$$

Notice that columns of the weight matrix correspond to fan-in weight vectors for the hidden layer (rows correspond to fan-out). This implies that you can compute weighted inputs at the hidden layer by the dot product of an input vector and each column, a topic we discuss next. Regardless convince yourself you can see the link between fan-in weight vectors and columns.

As an exercise, see if you can produce the weight matrix representation for $\mathbf{W}_{2,3}$ or $\mathbf{W}_{hidden,output}$. Hint: it has 3 rows and 2 columns.

Now the recurrent case, starting with the recurrent network shown in figure 6.6. Here the source and target neurons are the same and so the weight matrix, which we can call $\mathbf{W}_{recurrent}$, is square: it has as many rows as columns. To fill it out we follow the same procedure, going from source label to target label and finding the corresponding entry. For example, $w_{1,2}$ is the weight from node 1 to 2, and is thus in the first row, second column of the matrix. Confirm the values match up:

	1	**2**
1	2	−.5
2	−1	1.2

$$\begin{pmatrix} 2 & -.5 \\ -1 & 1.2 \end{pmatrix}$$

In a weight matrix for a recurrent network diagonal entries correspond to connections from a node back to itself. Also note that fan-in weight vectors still correspond to columns. The first node has fan-in weights of -2 and 1, the second has fan-in weights of .5 and 1.2.

Figure 6.7 shows another example, that shows what we do when some links are missing. If a weight does not exist, it is represented by a 0 in the corresponding matrix. Here is its representation:

	1	**2**	**3**	**4**
1	0	0	0	1
2	−1	0	0	0
3	1	0	0	0
4	0	0	0	−1

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

[15] This is a somewhat unconventional way of representing weight matrices. It can be called a "source-target" representation, because rows are associated with source neurons and columns are associated with target neurons. It is more common in neural network texts to use a "target-source" representation, where rows are associated with target neurons and columns are associated with source neurons. This is related to the choice to represent the weight from node j to k with $w_{j,k}$, a "source-target" indexing scheme, rather than $w_{k,j}$. We made changes in convention in order to make the formalism easier to learn, but again they are non-standard. Further discussion is in note 16.

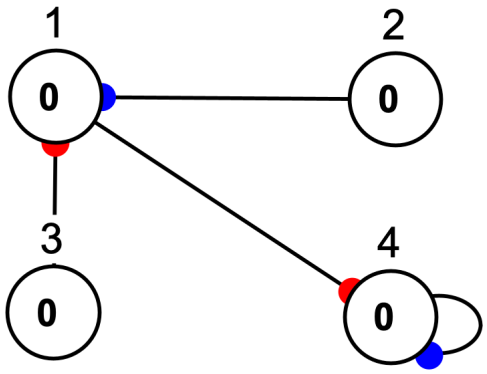


Figure 6.7: A sparse recurrent network, where most of the possible connections do not exist, so that in its matrix representation most entries will be 0. In this network red weights have a strength of 1 and blue weights have a strength of -1.

Now let's check your understanding: there are 4 nodes and thus the matrix is 4-by-4. There are two positive weights and two negative weights, and there are two positive entries and two negative entries in the table. Node 4 is self-connected and the fourth diagonal entry is non-zero. Nodes 1 and 4 have two weights in their fan-in and those two columns have two entries.

In the special case where most of the entries in a weight matrix are 0, the matrix is called a **sparse matrix**. The matrix representing the weights in figure 6.7 is sparse. A sparse matrix is contrasted with dense matrix, where most of the entries are non-zero. Really these are two ends of a continuum described by a number called **sparsity**, which ranges from 0 to 1, and where higher values are more sparse. The sparsity of a matrix is obtained by counting how many zero entries the matrix has and dividing by the total number of entries. If all of the matrix entries are 0 then the sparsity of the matrix is 1. If half of the entries are 0 then the sparsity is .5. If none of the matrix entries are 0 then the sparsity of the matrix is 0. The matrix in the table above has 16 entries, 12 of which are 0, so its sparsity is $12/16 = 3/4 = .75$. Sparse weight matrices represent sparsely connected networks, where only a few of the possible connections between relevant nodes exist (recall that a matrix representation of a network uses 0 to represent the absence of a connection). Dense matrices are used to represent densely connected networks. The word "dense connection" or "dense weight layer" in this context often implies a sparsity of 0, that is, an "all-to-all" or "fully connected" set of weights between relevant nodes.

6.8 Vector-Matrix Product

The power of matrices in neural networks is our ability to use them to represent weights and transformations of activation vectors as they propagate. We are splitting the topic into two parts. For the version of the course the book currently focuses on, we only consider one case: a vector-matrix product. We can understand this in a simplified way using mainly just the dot product. But this is just a specially curated case, and the full story is more complex and involves more cases.[16] But we're starting simple.

So the way we do things here, we focus on taking an input vector, and dotting it with columns of a

[16] We will not give a general definition of matrix multiplication or matrix product here. Instead, we will describe a special case of it: "vector-matrix" multiplication, which involves multiplying an activation vector on the left times a weight matrix on the right.We are presenting the computation as a row vector on the left times a weight matrix on the right, which is required by the "source-target" representation (see note 15). It is also easy to think about in terms of the operations involved in updating a neural network: an activation vector is "passed through" a weight matrix to produce another activation vector, in this case a hidden unit vector. However, this is non-standard. In linear algebra and most applications this operation would usually be represented with the matrix on the left and a column vector on the right. This is because a matrix is often thought of as a function that operates or acts on a vector, transforming inputs to outputs.

weight matrix, which (in the source-target format) represent fan-in weight vectors. In this way we produce a new output vector. Each of these dot products is a weighted input (see chapter 5). Assuming a default unbounded linear activation function, this can be used to represent weight propagation. We consider the feed-forward case first, then the recurrent case.

Feed-forward. Consider the feed-forward network in figure 6.6, which uses linear activation functions without bias (we are also ignoring clipping) so that node activations just are weighted inputs. That is, we multiply the input vector by the intervening weight matrix to obtain the hidden unit vector: $\mathbf{a}_{hidden} = \mathbf{a}_{input}\mathbf{W}_{input,hidden}$. We can then multiply the hidden unit vector by the hidden-to-output layer weight matrix to get the output vector. We can continue to do this for all the layers of a feed-forward network. So, for linear networks, pretty much all we do when updating the network is use matrix products (and even for non-linear networks, we use the matrix product to compute vectors of weighted inputs, which are then transformed by, for example, sigmoid functions).

Here is an informal description of how to multiply a vector on the left by a matrix on the right: take the dot product of the vector on the left and the first column vector in the matrix. The resulting number is the first component of a row vector, which will be the result of this operation. Then do this for each of the remaining columns of the matrix, adding these dot products to the row vector as you go. The resulting row vector is the matrix product of a row vector and a matrix. Intuitively, it is like you are writing out the matrix product, one number at a time, by dotting the vector on the right with each of the columns of the weight matrix.

Consider an example. Suppose the feed-forward network in figure 6.6 has linear activation functions and 0 bias, and its input activation vector is $\mathbf{a}_{input} = (1, 2)$. To compute the hidden layer activation vector, we compute the dot product between the input vector and each of the three column vectors:

$$\begin{pmatrix}1 & 2\end{pmatrix}\begin{pmatrix}1 & 0.7 & 2\\ -2 & -1 & 2.1\end{pmatrix} = \Big((1)(1)+(2)(-2),\ \ (1)(0.7)+(2)(-1),\ \ (1)(2)+(2)(2.1)\Big) = \begin{pmatrix}-3 & -1.3 & 6.2\end{pmatrix}$$

This can be visualized by imagining that an input activation vector is being combined ("dotted") with the fan-in weight vectors of each of the three nodes at the next layer, to produce the weighted input to each of them and thus the next layer's activation vector.

Now for the recurrent case. We can do the same kind of thing with the sparse recurrent network in figure 6.7, but in this case we will be determining its activations at successive time steps. This is because with a recurrent network, the output can always be fed right back into the network as input, which gives these networks their dynamic properties. Let the weight matrix be $\mathbf{W}_r$ then a sequence of activation vectors $\mathbf{a}(1), \mathbf{a}(2), \mathbf{a}(3), \ldots$ (with time in parentheses) is given by $\mathbf{a}(2) = \mathbf{a}(1)\mathbf{W}_r$, $\mathbf{a}(3) = \mathbf{a}(2)\mathbf{W}_r$, etc.

This is easy to see by exmaples. Suppose we have activated node 2 of the network in figure 6.7 and we start iterating. Since the activations and weights are all whole numbers, it's not too hard to compute this out for a few time steps. Again just dot the input vector by the columns to write out the first output, where we get the activation vector at time 2 by multiplying the initial activation vector by the weight matrix (that is, $\mathbf{a}(1)\mathbf{W}_r = \mathbf{a}(2)$):

$$\begin{pmatrix}0 & 1 & 0 & 0\end{pmatrix}\begin{pmatrix}0 & 0 & 0 & 1\\ -1 & 0 & 0 & 0\\ 1 & 0 & 0 & 0\\ 0 & 0 & 0 & -1\end{pmatrix} = \begin{pmatrix}-1 & 0 & 0 & 0\end{pmatrix}$$

Try to do it in your head, dotting the $(-1, 0, 0, 0)$ with the four columns of the matrix and thereby writing out the output vector. In the next iteration you will get $(0, 0, 0, -1)$, then $(0, 0, 0, 1)$. If we keep going we get the following sequence of states from the initial state $(0, 1, 0, 0)$:

$$(0,1,0,0),\ (-1,0,0,0),\ (0,0,0,-1),\ (0,0,0,1),\ (0,0,0,-1)\ \ldots$$

You can easily set this up in Simbrain and confirm this is what happens. We will see in the dynamical systems chapter 10 that this is one *orbit* in the network's activation space. In this case, the network has started to oscillate between two states, and it will do that forever, and that behavior is basically encoded in the weight matrix.

6.9 Tensors

A **tensor** is a generalization of the concept of a vector that encompasses numbers, arrays of numbers, matrices (2d arrays of numbers), arrays of matrices, arrays of these arrays, etc. These more complex structures are increasingly common since the time of the deep learning revolution (section 3.7). The basic idea is not just to work with vectors and matrices, but also sets of matrices and even sets of sets of matrices. These have special nomenclature like "volume". In this section we cover the basics.

The **rank** of a tensor is the number of indices required to specify an entry in it.These are also sometimes called "n-dimensional arrays" or "n-d arrays" (1d array, 2d array, etc.).[17] The **shape** of a tensor is the number of components it has along each of the array's dimensions. We have seen this with matrices, where the shape is stated in terms of rows and columns, e.g. a 5×2 matrix. For more complex tensors the shape is specified by a number of components for each dimension of the array, like a $4 \times 2 \times 5$ volume. Here are the main types of tensor:

- A **scalar** is a rank 0 tensor or 0d array because it requires no indices. The number 42 is a rank 0 tensor, a 0d array, but nobody talks about it that way.
- A **vector** is a rank 1 tensor or 1d array because it takes one index to specify an entry in a vector, and the result is spread out in one dimension. The vector $(0, 1, 0)$ is a rank 1 tensor, because it takes one index to specify entries, but people usually just call it a vector. Vectors were discussed in much of this chapter, starting in section 6.1.
- A **matrix** is a rank 2 tensor or 2d array, because it takes two numbers to specify an entry (a row and column index), and the result is spread out in two dimensions. We've seen lots of examples of matrices in this chapter, and they were discussed in section 6.6.
- A **volume** is an array of matrices. It is a rank 3 tensor or 3d array, because it takes 3 indices to specify an entry, and the result is spread out in three dimensions. The term "volume" is common but not completely standard, but is intuitive so we adopt it here. It can be visualized as a stack of matrices, or as a solid, something like a Rubik's cube or 3d chess board (see figure 6.9). A common use of volumes is to represent images, which requires several channels of 2d information, several "copies" of a pixel array. For example, an RGB color image that is 28 rows (height) by 28 columns (width) is represented by three matrices (red, green, and blue) of shape 28×28. Thus the whole image is represented by a tensor with shape $3 \times 28 \times 28$. These indices are referred to as channel, width, and height.
- A **batch of volumes** is an array of volumes. It is a rank 4 tensor or 4d array because it takes 4 indices to specify an entry. It is spread out in four dimensions, but since we can't visualize that we can instead visualize a set of volumes, as in the right panel of figure 6.8. The word "batch" is used here because we are often dealing with sets or batches of inputs in a training dataset (see chapter 7), in this case batches of RGB images, each of which is a volume with 3 channels. For example, if the images are $3 \times 28 \times 28$, then the batch of 100 images has shape $100 \times 3 \times 28 \times 28$.

Examples of each of these types of tensor are shown in figure 6.8. More examples are in chapter 14.

We can refer to the different indices for a tensor using standardized names in a standard order: batch size, depth (number of channels), height, and width.[18]

[17] Note that the dimensionalty of an array is the literal dimension of the object, like a vector is 1d and a matrix is 2d. This is not the same as the dimensionality of the space these objects live in. For example, $(1, 2, 1, 0)$ is a point in a 4d space, but it is a 1d tensor or 1d array of numbers.

[18] This is sometimes called NCHW format (number of samples, number of channels, height, width). However, usage varies. Some put height and width first in indexing; some put them last in indexing. Even though the rank of a tensor is the number of indices *required* to specify an entry, sometimes additional entries are included (e.g. specifying a vector as a column vector assumes it is being placed in a 2d space). Thus the presentation of the "shape" of a tensor can vary. In figure 6.8 the shapes might be given as 1×1, 6×1, 6×6, $3 \times 6 \times 6$, and $3 \times 3 \times 6 \times 6$.

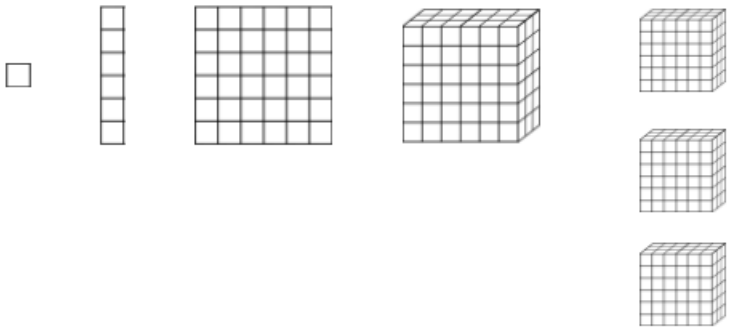


Figure 6.8: Schematic of different types of tensor. From left to right: a scalar, a vector, a matrix, a volume, and a batch of volumes. It can be seen that they are 0d, 1d, 2d, 3d and 4d arrays. Numbers are not drawn in; only the abstract shape of the tensors are shown.

Figure 6.9: Two ways a representing a 3d array: a stack of matrices (left) vs. a solid "Rubik's cube" or volume (right). Depending on the context one or the other representation is more useful.

6.10 Appendix: Block Matrix Representations

For the feed-forward network in figure 6.6, we can begin with a matrix for the full network, which illustrates some of its structure:

$$\left(\begin{array}{cc|ccc|cc} 0 & 0 & 1 & 0.7 & 2 & 0 & 0 \\ 0 & 0 & -2 & -1 & 2.1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & -2 & 0.9 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1.2 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

This is a "block matrix" containing two blocks of non-zero values (corresponding to the layers that are connected), and 7 blocks of zeros (corresponding to possible layer-to-layer weight matrices that don't exist for this network, e.g. a recurrent layer from the hidden layer to itself, or a direct layer from the input to the output layer).

6.11 Appendix: Vector Operations

Vectors are not just lists of numbers. They are members of *vector spaces*, which are abstract mathematical spaces that have an addition operation and a scalar multiplication operation, and other operations that can be defined on the basis of these. In this appendix we introduce these two basic operations and several others. We also develop the formal definition of a vector space.

The addition of two vectors with n components, or **vector addition**, is simply the component-wise addition of the two vectors. This is easiest to see by example. Here is an example of adding two vectors with 3 components:

$$(\ 0,\ -1,\ 9) + (\ 1,\ 2,\ 4) = (\ 0+1,\ -1+2,\ 9+4) = (1, 1, 13)$$

Here are a few more examples:

$$\begin{aligned}(1,1)+(2,3) &= (3,4)\\ (1,-1,1)+(0,0,0) &= (1,-1,1)\\ (2,3,5,8,13,21)+(3,5,8,13,21,34) &= (5,8,13,21,34,55)\\ (-1,.5,\sqrt{7})+(-1,-2,.8) &= (-2,-1.5,\sqrt{7}+.8)\end{aligned}$$

In a similar way, *vector subtraction* is the component-wise subtraction of the corresponding components of two vectors.[19] Here are some examples:

$$\begin{aligned}(1,1,1)-(0,1,0)=(1-0,1-1,1-0) &= (1,0,1)\\ (10,5)-(5,10)=(10-5,5-10) &= (5,-5)\\ (1,2,3,4,5,6,7)-(0,0,0,0,0,0,0) &= (1,2,3,4,5,6,7)\\ (2,6,1)-(.5,20,-100) &= (1.5,-14,101)\end{aligned}$$

If all of the components of a vector are 0 we call it the **zero vector**. Adding the zero vector to any vector leaves it unchanged.

Another operation that can be performed with vectors is "scalar multiplication". A **scalar** is a generic term for the type of numbers we choose to work with. These numbers are called scalars because we can "rescale" vectors using scalar multiplication. Usually we work with real numbers in which case we say our scalars are real numbers. Sometimes people use complex numbers or something even more exotic for their scalars.

The **scalar multiplication** of a scalar and a vector is obtained by multiplying each of the vectors component's by the scalar. Scalar multiplication is indicated by placing the scalar and vector next to each other without any intervening symbols. For example scalar multiplication of the scalar 3 with the vector $(1,2,4)$ can be written as

$$3(\ 1,\ 2,\ 4) = (\ 3\cdot 1,\ 3\cdot 2,\ 3\cdot 4) = (\ 3,\ 6,\ 12)$$

The operations of vector addition and scalar multiplication can be combined. The result is called a **linear combination** of vectors. For example

$$1(\ 0,\ 0)+2(\ 0,\ 1)+3(\ 1,\ 0)+4(\ 1,\ 1) = (\ 7,\ 6)$$

is a linear combination of the vectors $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$.

Scalar multiplication of any vector with the number 0 is the zero vector. The scalar multiple of a vector with the scalar -1 gives us the negative of the vector. We define **vector subtraction** of one vector from another as the addition of the vector's negative. Subtracting a vector from itself is the zero vector.

$$(\ 1,\ 2,\ 4)-(\ 1,\ 2,\ 4) = (\ 1,\ 2,\ 4)+(\ -1,\ -2,\ -4) = (0,0,0)$$

Now we can more formally define a vector space. A set of vectors that satisfies two conditions

(1) The sum of any two vectors in the set is also in the set.
(2) Every scalar multiple of a vector in the set is also in the set.

is called a **vector space**. We will apply vector spaces to neural networks in chapter 10. If a subset of a vector space satisfies these conditions, we say it is a **subspace** of the vector space. These definitions allow a vector space to be a subspace of itself.

The set of all linear combinations of a set of vectors is called the **span** of the vectors. The span of a set of vectors forms a subspace. If the span of a set of vectors is the whole vector space and any proper subset of that set of vectors does not span the whole vector space, then that set of vectors is a **basis** of the

[19] Vector subtraction can be defined in terms of vector addition and scalar multiplication. Thus vector subtraction is not fundamental to the definition of a vector space. It is nonetheless presented here because it is used in several other places in this book.

vector space. There are many different bases[20] for a vector space but all of them have the same number of members. This number is the dimension of the vector space.

For example

$$\{(1,0),(0,1)\} \quad \{(1,2),(1,1)\}$$

are both basis for the same 2 dimensional vector space. Every vector (x, y) can be written as

$$(x, y) = x(1, 0) + y(0, 1)$$

so $\{(1,0),(0,1)\}$ spans the plane. But every vector in the span of $(1, 0)$ has 0 for it second component and every vector in the span of $(0, 1)$ has 0 for its first component, so we cannot write every vector in the plane without both $(1, 0)$ and $(0, 1)$. The set $\{(1,0),(0,1)\}$ is a basis for the plane. It is called the standard basis.

Every vector (x, y) can be written as

$$(x, y) = (y - x)(1, 2) + (2x - y)(1, 1)$$

so the set $\{(1,2),(1,1)\}$ spans the plane. But the components of every vector in the span of $(1, 1)$ are equal to each other, so $(1, 2)$ is not in the span of $(1, 1)$. For every vector in the span of $(1, 2)$ the second component is twice the first, so (1,1) is not in the span of $(1, 2)$. Thus, $\{(1,2),(1,1)\}$ is also a basis for the plane.

6.12 Exercises

1. What is the dimensionality of the input space, hidden unit space, output space, weight space, and activation space in Fig. 6.6 (Left)? **Answer:** 2-dimensional, 3-dimensional, 2-dimensional, 12-dimensional, and 7-dimensional.

2. What is the dimensionality of the weight space and activation space in Fig. 6.6 (Right)? **Answer:** 4-dimensional and 2-dimensional.

3. What is the dimensionality of the weight space and activation space in Fig. 6.10? **Answer:** 3-dimensional and 3-dimensional.

4. What is $(1, 1, 1) \bullet (1, 1, 1)$? **Answer:** $(1 \cdot 1) + (1 \cdot 1) + (1 \cdot 1) = 1 + 1 + 1 = 3$.

5. What is $(-1, 0, 1) \bullet (-1, -1, 0)$? **Answer:** $(-1 \cdot -1) + (0 \cdot -1) + (1 \cdot 0) = 1 + 0 + 0 = 1$.

6. What is $(10, 2, -10) \bullet (0, 10, -10)$? **Answer:** $0 + 20 + 100 = 120$.

7. What is $(.5, -1, 1, -1) \bullet (10, -2, 1, 2)$? **Answer:** $5 + 2 + 1 - 2 = 6$.

8. Suppose we have $(a_1, a_2) = (1, -1)$, $(w_{1,3}, w_{2,3}) = (-1, 1)$ and $b_3 = 5$. What is n_3? **Answer:** $(1 \cdot -1) + (-1 \cdot 1) + 5 = -1 - 1 + 5 = 3$.

9. What is the matrix representation of the weights in the network in Fig. 6.10? **Answer:**

$$\begin{pmatrix} 0 & -1 & 0 \\ 0 & 0 & 0.5 \\ 2 & 0 & 0 \end{pmatrix}$$

10. What is the fan-in weight vector for node 2 in Fig. 6.10? **Answer:** (-1).

11. What is $\begin{pmatrix} 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$? **Answer:** $\Big((1)(1) + (1)(3), (1)(2) + (1)(4)\Big) = (4, 6)$.

[20] "Bases" is plural for "basis".

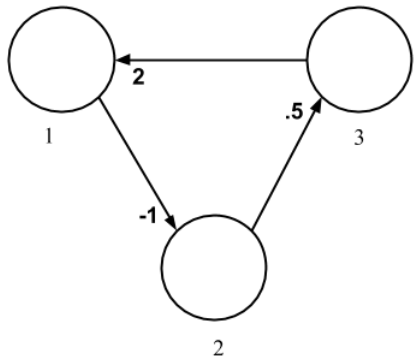


Figure 6.10: A recurrent network with three nodes labelled $1, 2, 3$ and weights $w_{1,2} = -1, w_{2,3} = 0.5, w_{3,1} = 2$.

12. What is $\begin{pmatrix}-1 & 1\end{pmatrix}\begin{pmatrix}1 & -2\\ 3 & 4\end{pmatrix}$? **Answer:** $\Big((-1)(1)+(1)(3), (-1)(-2)+(1)(4)\Big) = (2, 6)$.

13. What is $\begin{pmatrix}-10 & 10\end{pmatrix}\begin{pmatrix}0.5 & -0.5\\ -1 & 1\end{pmatrix}$? **Answer:** $(-15, 15)$.

14. If the network in Fig. 6.10 has linear nodes and is given the activation vector $(a_1, a_2, a_3) = (1, 1, 1)$, what will its activation be in the next time step? **Answer:**

$$\begin{pmatrix}1 & 1 & 1\end{pmatrix}\begin{pmatrix}0 & -1 & 0\\ 0 & 0 & 0.5\\ 2 & 0 & 0\end{pmatrix} = \Big((1)(0)+(1)(0)+(1)(2),\ \ (1)(-1)+(1)(0)+(1)(0),\ \ (1)(0)+(1)(.5)+(1)(0)\Big) = (2, -1, 0.5)$$

15. If the network in Fig. 6.10 has linear nodes and is given the activation vector $(a_1, a_2, a_3) = (-1, -1, 2)$, what will its activation be in the next time step? **Answer:**

$$\begin{pmatrix}-1 & -1 & 2\end{pmatrix}\begin{pmatrix}0 & -1 & 0\\ 0 & 0 & 0.5\\ 2 & 0 & 0\end{pmatrix} = (4, 1, -0.5)$$

16. If the network in question 14 is iterated four times, what will its activation be in those four time steps? We saw from question 14 that after one time step the activation vector is $(2, -1, 0.5)$. If we now use this as input to the network again we get:

$$\begin{pmatrix}2 & -1 & 0.5\end{pmatrix}\begin{pmatrix}0 & -1 & 0\\ 0 & 0 & 0.5\\ 2 & 0 & 0\end{pmatrix} = (1, -2, -0.5)$$

Repeating this process again with $(1, -2, -0.5)$ as input we get $(-1, -1, -1)$. Repeating one more time we get $(-2, 1, -0.5)$. **Answer:** $(2, -1, 0.5), (-1, -2, -0.5), (-1, -1, -1), (-2, 1, -0.5)$.

Chapter 7

Data Science and Learning Basics

Jeff Yoshimi

In this chapter, we cover fundamental concepts used when training neural networks, focusing in particular on the tables of data involved. Tables of data are the bread and butter of any neural network practitioner, and we must understand them well. Thus, in this chapter we begin with a brief introduction to **data science**, which is an area of practice focused on processing and analyzing datasets, often using machine learning models. Though data science is most connected with engineering uses of neural networks (see chapter 2), concepts from this field are equally applicable any time neural network models are used. Moreover, anyone who uses neural networks in practice must understand how to deal with data: how to clean it up, re-code certain features, produce exploratory visualizations, and so forth. In this chapter, we begin with an overview of basic concepts from data science. At the end of the chapter, we use these concepts to differentiate the main types of learning algorithm in neural networks.

7.1 Data Science Workflow

Here is a basic workflow that is common in data science:

1. Getting the data. Describing it. Understanding its basic features. Coming up with useful column names or "feature" names. You might obtain data from an experiment, download data from a website, or be given a large table or spreadsheet. To get a better sense of the nature of the data used in a neural network, and the kinds of work needed to wrangle it into a format that a network can process, several public repositories of machine learning and other kinds of data exist.[1] Special issues arise when using the very large datasets ("big data") required to effectively train some machine learning models[2], but we will focus on small datasets that are useful for illustrative purposes.

2. Visualizing the data / Exploratory Data Analysis (EDA). Developing an initial feel for data, often using visualizations.[3] Creating pictures that illustrate the main features of your data, which suggest how your machine learning task might be solved, e.g. finding data that show a correlation between some input data and the target data.

3. Preparing the data. Also called data wrangling. Creating useful features. Filling in missing data. Removing outliers. Discussed in more detail in section 7.3.

[1] See: `https://archive.ics.uci.edu/ml/index.php` and `https://en.wikipedia.org/wiki/List_of_datasets_for_machine_learning_research`. Many other sources of data exist, of course, including US Census data, World Health organization data, etc. The website Kaggle has a large repository of datasets and machine learning tasks that can be pursued in a game-like competitive framework. Many public tools, like R, sklearn, Pytorch, and Tensorflow have datasets included.

[2] Cf. `https://en.wikipedia.org/wiki/Big_data` and ETL `https://en.wikipedia.org/wiki/Extract,_transform,_load`.

[3] See `https://en.wikipedia.org/wiki/Exploratory_data_analysis`.

4. Create and train a model. Choose a type of model and then train it. This is where neural networks come in. In machine learning there are many kinds models, like multiple regression and decision trees and ensembles of models. But we focus on neural networks.

5. Assessing the model's performance on test data. We will see that it is often important to first train a model on one set of data, and then to see how well it generalizes to new data it's never seen before.

We will not discuss obtaining data or exploratory data analysis here. We discuss data wrangling in section 7.3. The rest of the chapter, and much of the rest of the book, is focused on creating and training neural network models. Assessing performance is briefly discussed in several places in this chapter and the next few chapters.

Of these steps, the main one in terms of learning is the step where we build and train a model. When the model is trained, we update its **parameter**s. Parameters are discussed further in the dynamical systems chapter, chapter 10. In a neural network, these parameters are usually weights and biases (chapter 5). For a feed-forward network, parameter changes impact the input-output function the network produces. Feed-forward networks take an input vector and produce an output vector. The process of training a feed-forward network is the process of modifying the parameters in such a way as to change the vector-valued function it implements. In chapter 12, we see how to update the parameters of a feed-forward network to achieve a desired input-output function, e.g. to make a network that recognizes letters or faces in images. For a recurrent network, parameter changes influence the dynamics of the system, they change its "phase portrait." From the same initial state different activation patterns will occur. In chapter 16, we see how the parameters of a recurrent network can be trained to achieve desired dynamics. This type of network can be used to produce realistic speech and convincing text, as we will see.

However, in this chapter we also see all the other work that is involved in actually building a neural network model. Data must be gathered, analyzed, and cleaned up. And then we must partition our data in a special way in order to test how well it works not just on the data we trained it on, but also on new data it has never seen before.[4]

7.2 Datasets

As discussed in chapter 1, neural networks are usually linked to an *environment*, where that environment is often a simple table or spreadsheet. Even though we are just dealing with tables here, there are a lot of concepts and terms to master. We will take a **dataset** to be a table of values to be used by a neural network. A row of a dataset is an **example**, *instance*, or *case*. A column of a dataset is a **feature** or *attribute*. The columns of a dataset often correspond to the nodes of a network. Rows often correspond to inputs that will be sent to the input layer of a network, or used to describe desired outputs. These rows and columns can be transformed, partitioned, and manipulated in various ways, as we will see.

As an example, consider the Motor Trend Cars dataset ("*mtcars*"), shown in Fig. 7.1, which is included with the R statistical computing environment. The data is based on a 1974 issue of the car magazine *Motor Trend*, which road-tested about 30 models of cars in the 1973-74 model year and measured a range of performance features.[5] The dataset contains 30 examples with 10 features each, 6 of which are shown (the row indices and the model names will not be sent to any neural network, so we don't count them as features). Notice that this table, as it stands, is not ready to be used by a neural network. The neural network can't deal with the names (which are strings, rather than numbers), and as we'll see, some of the values (like horsepower) are kind of big for a node that is only meant to deal with small numbers between -1 and 1. However, after a bit of processing, the data can be used to train a neural network to, for example, predict the fuel efficiency of a car based on its weight and number of cylinders.[6]

We can distinguish two main types of data, beginning with **categorical data**, also known as "nominal data". Categorical data can take one of a discrete list of values. For example, cards can have one of four

[4] A worked example illustrating many of the ideas in this chapter in tensor flow is here: `https://www.youtube.com/watch?v=-vHQub0NXI4`.

[5] See `https://stat.ethz.ch/R-manual/R-devel/library/datasets/html/mtcars.html`. A neural network that processes this dataset is included with Simbrain as a script called *backprop_cars.bsh*.

[6] A detailed discussion of this case is here: `https://www.youtube.com/watch?v=K4GZ51cozRs`.

	Model	MPG	Cylinders	Displacement	Horsepower	Weight	Quarter mile
0	Mazda RX4 Wag	21.0	6	160.0	110	2.875	17.02
1	Datsun 710	22.8	4	108.0	93	2.320	18.61
2	Hornet 4 Drive	21.4	6	258.0	110	3.215	19.44
...	...	...	...	...	...	...	...
28	Ferrari Dino	19.7	6	145.0	175	2.770	15.50
29	Maserati Bora	15.0	8	301.0	335	3.570	14.60
30	Volvo 142E	21.4	4	121.0	109	2.780	18.60

Figure 7.1: A fragment of the *mtcars* dataset showing some of its examples (rows) and some of its features (columns)

suits: hearts, diamonds, spades, or aces. The state one lives in can be one of 50 values. In the example in Figure 7.1, cylinders appears to be categorical, because there are three possible values for that feature: 4, 6, or 8 cylinders. For a neural network, these will be converted to numerical data using a **one-hot encoding** (where one node corresponding to the category is "hot" and the rest are not), as we will see.

Second, **numerical data** is data that is already in the form of numbers. These numbers can either be real-valued (represented by floating point values in a computer) or integer-valued. Examples: age, income, house prices, hours of study, GPA, length, width, weight, caloric intake. In Figure 7.1, most of the columns are numerical. A few seem to be integers (displacement, horsepower), and others are clearly real-valued (weight, quarter-mile).[7]

To get a general sense of how datasets are used with neural networks, see Fig 7.2. Each column, each feature, is generally associated with the nodes of a network. The figure shows an input dataset, but we will see there are other types of datasets used with neural networks as well.

7.3 Data Wrangling (or Preprocessing)

Raw data isn't usually ready to be fed to a neural network. Sometimes a dataset contains strings of text, images, sound files, and other structures that must be converted into a numerical format. Neural networks want *numbers*, and they often want those numbers to be a certain way. So we have to pre-process the data in various ways. Our ultimate goal is typically to have a table all of whose cells contain numbers that lie within a fairly small range, like between -1 and 1 or between 0 and 1.[8] That is, we want to end up with a dataset where each row is an input vector that can be fed to the input nodes of a neural network.

So we have work to do. We have to convert non-numerical data to single numbers. We have to fill in missing data. And even when all the data is numerical we must often do further things like rescaling the data. These operations correspond to **pre-processing** the data. This is sometimes called **data wrangling** or "data munging".[9] Here is how wikipedia defines it:

> Data munging or data wrangling is loosely defined as the process of manually converting or mapping data from one 'raw' form into another format that allows for more convenient consumption of the data with the help of semi-automated tools.[10]

[7] In more rigorous treatments (derived from the study of scale types in measurement theory), ordinal, interval, and ratio scales are distinguished. We collapse interval and rational scales into numerical. Ordinal data (e..g first, second, and third in line) can often be treated as integers or using a one-hot encoding.

[8] At that point our dataset has the form of a matrix (cf. Chapter 6), and mathematical operations of linear algebra can be applied to it.

[9] For a sense of some of the ways data can be wrangled, have a look at the *scikit-learn* pre-processing library: `http://scikit-learn.org/stable/modules/preprocessing.html`.

[10] `https://en.wikipedia.org/wiki/Data_wrangling`.

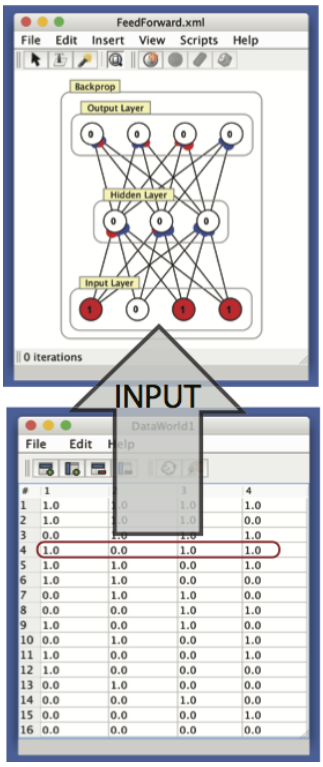


Figure 7.2: An example of an input dataset, which illustrates one standard way datasets are used with neural networks. Each row of the dataset is thought of as one input vector for the neural network.

The process of wrangling data is usually understood as a step-wise workflow or pipeline, where the data is obtained and transformed in stages until it is ready to be processed by a neural network. There are different ways of understanding this workflow. Here is a generic version of a data wrangling workflow:

- Data cleaning: remove, fix, or otherwise deal with bad data. Fill in missing data.
- Feature-extraction and feature-engineering: Transform data (e.g. text, images, audio files, DNA sequences) into a numerical format and more generally produce a set of numerical features to be used by the neural network.
- Rescaling: alter the numbers in the dataset to, for example, ensure that they are all in the range $(-1, 1)$

The first step is **data cleaning** or "data cleansing". There might be stray characters that make it hard to import the data, or columns that are irrelevant to what you are trying to do. Often it helps to simply focus on a subset of columns or rows (*subsetting*). A related cleansing step is dealing with missing data, using methods of *data imputation* to determine a policy for filling in missing data. Common techniques include filling in these cells with 0's, or with the mean value of the column they are in.[11]

The next step, **feature-extraction** involves converting non-numeric data into a numerical form suitable for a neural network. Images, movies, audio, DNA sequences, and of course, text, are all non-numeric data that must be converted to a numerical format.[12] This is often the most involved and most important step in building a working model. Many of the earliest connectionist models (e.g. Nettalk) relied on clever ways of representing written and spoken speech in a vectorized way that could be fed to a network.

We are construing this step quite broadly to include any steps involved in coming up with features (numerical columns) for a dataset, from flattening a matrix, to combining features into new features. The

[11] See `https://towardsdatascience.com/how-to-handle-missing-data-8646b18db0d4` and `http://www.stat.columbia.edu/~gelman/arm/missing.pdf`.

[12] Also see `http://scikit-learn.org/stable/modules/feature_extraction.html`

latter is sometimes also called *feature-engineering*, where a new feature is designed for use in training a model, e.g. deciding not to feed a neural network height and width information separately, but rather to feed it the ratio of the height to the width of an image, which might yield better results for some applications. Another example in the *mtcars* dataset would be to take the model of a car and then consult a database online to find new features of the cars.[13]

Here are some examples of feature extraction in this fairly broad sense.

- Taking a feature like the model of a car, state of residence, or gender, and converting it into a vector of binary values. There are several ways to do this, but the most common is using a one-hot or "one-of-k" encoding, which is a type of coding that converts categorical data to binary vectors.[14] If we have three categories–Fish, Swiss, and Gouda–then we can use a one-of-three encoding to represent Fish as $(1, 0, 0)$, Swiss as $(0, 1, 0)$, and Gouda as $(0, 0, 1)$. In a bank of nodes, this corresponds to one node being active ("hot") and the other nodes being inactive, hence "one-hot" encoding. One is hot, and the others are not. For example, in the cars dataset, we can represent 4-cylinder, 6-cylinder, and 8-cylinder by a one-hot (in this case one-of-3) encoding, as in Fig. 7.3. Notice that the cylinder column in Fig. 7.1 has been replaced by three columns. The column that has a "1" in it indicates whether the car is 4, 6, or 8 cylinders. These are also called dummy or indicator variables in psychology. They are also a form of localist as contrasted with distributed representation (see chapter 1).

- Taking a matrix and "flattening" it into a vector that can be treated as a row of a dataset. This is often done with images. Sometimes an even more complex object, a *tensor*, must be flattened (see section 6.9). Color images are often represented as three separate pixel images, corresponding to red, green, and blue channels. So we have three matrices that must be flattened and concatenated to produce one long vector, which is then a proper row of a dataset that can be fed to a network.

- Converting strings of texts to vectors. Thus, the word "red" might become the vector $(1, 0, 1, 0, 1, 1)$. Techniques for converting linguistic data to vectors are sometimes referred to as methods of *word embedding*. Word embedding is a major area of research in its own right (see chapter 8).

- Dividing a sound file into smaller time windows and converting those "clips" of audio into vectors, often using signal processing techniques like Fourier analysis.

- Hand coding video or audio data in some way, e.g. counting how many times a participant in a videotaped experiment hits a doll, or how many questions a participant asks. This kind of technique is often used in experimental settings, e.g. in psychology.[15]

	Model	4 Cyl	6 Cyl	8 Cyl	Displacement	Horsepower	Weight	Quarter mile
0	Mazda RX4 Wag	0.0	1.0	0.0	160.0	110	2.875	17.02
1	Datsun 710	1.0	0.0	0.0	108.0	93	2.320	18.61
2	Hornet 4 Drive	0.0	1.0	0.0	258.0	110	3.215	19.44
...	...	...	...	...	...	...	...	...
28	Ferrari Dino	0.0	1.0	0.0	145.0	175	2.770	15.50
29	Maserati Bora	0.0	0.0	1.0	301.0	335	3.570	14.60
30	Volvo 142E	1.0	0.0	0.0	121.0	109	2.780	18.60

Figure 7.3: Convert cylinders to a binary "one-hot" encoding.

[13] In competitive machine learning, as in Kaggle, often the best solutions are based on clever feature engineering, more so than anything in the machine learning model itself.

[14] See `https://en.wikipedia.org/wiki/One-hot`.

[15] See `https://en.wikipedia.org/wiki/Coding_(social_sciences)`.

Having coded all data as numerical, additional work often remains to be done, in particular, **rescaling** the data so that they fit in some standard range, e.g. $(0, 1)$ or $(-1, 1)$. Figure 7.4 shows the *mtcars* dataset of figure 7.3 after all columns have been rescaled to lie between 0 and 1. A simple way to do this for positive valued data is to divide each entry by the maximum value in that column. A similar method works on data that contains negative values. This is sometimes called *min-max scaling*.[16] This method ensures that all data are in the range $(0, 1)$. Another method is standardizing, where each value in a column is centered at the column mean and scaled by standard deviation. This makes it intuitive to interpret data. If we standardize a column, then the 0's correspond to average values, positive values are above average, and negative values are below average (in statistics these values are sometimes called z-scores). Anything above 1 is unusually large, and similarly for values below -1.

	Model	MPG	Cylinders	Displacement	Horsepower	Weight	Quarter mile
0	Mazda RX4 Wag	0.45	0.5	0.22	0.20	0.35	0.30
1	Datsun 710	0.53	0.0	0.09	0.14	0.21	0.49
2	Hornet 4 Drive	0.47	0.5	0.47	0.20	0.44	0.59
...	...	...	...	...	...	...	...
28	Ferrari Dino	0.40	0.5	0.18	0.43	0.32	0.12
29	Maserati Bora	0.20	1.0	0.57	1.00	0.53	0.01
30	Volvo 142E	0.47	0.0	0.12	0.20	0.32	0.49

Figure 7.4: Data from Fig. 7.1 rescaled to $(0, 1)$.

7.4 Datasets for Neural Networks

When we train a neural network, we update its parameters–its weights and biases–so that it can learn to do useful things. This is what our brains do when we learn, updating synaptic strengths in order to function more effectively. As we will see, for unsupervised learning, we take an input dataset and train it to pick up statistical features of the data. For supervised learning, we take "target data" or "labeled data" and use it to train a network to do some desired thing.[17]

To support these tasks, we must define several standard types of datasets:

- **Input dataset**: each row contains an input vector that can be sent to a neural network. This idea is illustrated in Fig. 7.2.
- **Output dataset**: each row contains an output vector that has been recorded from a neural network. These are also called "predictions".
- **Target dataset (labels)**: each row contains a target output vector we'd like a neural network to produce for a given input vector. The target dataset is a set of *desired outputs*, a set of labels.
- **Labeled dataset**: an input dataset and a corresponding target dataset. Note that the two datasets must have the same number of rows. This idea is illustrated in Fig. 7.8.

Examples of each type of dataset are shown in Fig. 7.5.

An **input dataset** contains input vectors to be sent to the input nodes of a neural network. Each row of an input dataset is a point in the input space of a neural network. Input datasets are used for all kinds of learning tasks, supervised and unsupervised.

[16] It can also be called "normalization" but that term is confusing because it is used in linear algebra in a different way.

[17] Note that the term "label" is associated specifically with classification tasks, where an input is sorted into one of a finite set of categories. Think of labeling images as cat vs. dog. But not all training tasks are like that; regression tasks for example associate inputs with real-valued targets. "Target data" is thus a more general term. However, the terminology of "labeled data" has become standard, and is snappier than "input-target dataset". We will use both terminologies interchangeably.

An **output dataset** is *generated* from an input dataset. We feed each row of the input dataset to a network and record the resulting output vector. Thus, an output dataset will have as many rows as the input dataset used to train it. The phrase "output dataset" is non-standard. Since these are often interpreted as predictions given a set of inputs, this table is sometimes referred to as a set of "predictions".

A **target dataset** contains the outputs we *want* the network to produce. These can be thought of as desired outputs. These targets are also called "labels", for classification tasks, described below. We compare an output dataset with a target dataset to produce an error, discussed in chapter 12. Like an output dataset, a target dataset will have as many rows as a corresponding input dataset.

A **labeled dataset** (also *labeled data* or *input-target dataset*) is a concatenation of two tables, an input and a target dataset. We can represent this by simply concatenating the two datasets and separating them with double vertical lines, as in the right-most panel of Fig. 7.5. This is perhaps the most common type of dataset to consider, since it is a specification of a supervised learning task. Labeled data is often difficult to obtain, because we can't simply gather it "from the world." If we take a bunch of pictures of people's faces and transform the data then we have our input dataset. But it is an extra step for a human to come in and label each face as male or female, so that we can confirm that a machine can also do the job. The contrast to labeled data is an input dataset by itself, or what is sometimes referred to as "unlabeled data".

Inputs		
1	0	0
0	1	0
0	0	1
1	0	1

Outputs	
.5	0
0	.5
-1	-.5
-1	-.5

Targets	
1	0
0	1
-1	-1
-1	-1

Inputs			Targets	
1	0	0	1	0
0	1	0	0	1
0	0	1	-1	-1
1	0	1	-1	-1

Figure 7.5: From left to right: an input dataset, output dataset, target dataset, and labeled dataset.

7.5 Generalization and Testing Data

One attractive feature of neural networks is that even if they have been trained on a specific dataset, they will tend to generalize well to new patterns they weren't trained on. This is easy to see with the 3-object detector in Simbrain, discussed in section 1.2. Try plugging in inputs it has not seen before, and it will still do well. This is a psychologically realistic property of neural networks. Suppose I have only ever seen two pineapples. My neural network was trained on only two pineapples. But I manage to correctly classify many other pineapples that I've never seen, even though they produce slightly different patterns on my eye. Our neural networks are good at **generalization**, at extrapolating from what they have seen to new things they have not seen.

On the other hand, sometimes the specific inputs a neural network is trained on are, in a sense, *too* specific. Ideally, we have diverse inputs that allow us to deal well with new situations. But sometimes people are exposed to data that is narrow and that leads to poor generalization. If you grow up in the forest you will be very good at classifying trees but not so good at classifying buildings. This is the origin of biases and stereotypes and linguistic accents.

This issue also comes up in neural networks. When you train a network on a labeled dataset, it can learn to be very good at predicting the target data you provide it. However, it might end up being *too* finely tuned on that data and thus fail to do well with new data. This is called *overfitting*. We want to build models that are not overfit to the data they were trained on. We want them to do well not just on the data we trained them on, but also on new data they have never seen. How well does the network generalize to new data? This is also referred to as "out of sample" performance (how well does a model do outside of the same data it was trained on). We train a network on 30 cars, and then test it on a new car it's never seen before. Or we train a network to classify 100 letters, but then we give it new letters it's never seen before. A good network can generalize from what it's been trained on, to new data.

To deal with this issue, we partition a labeled dataset into two subsets. We train the network on one subset of data and then test it on another set of data that we have "held out" to see how well the network generalizes. These two subsets are a training subset and a testing subset of a labeled dataset.

A **training subset** or *training dataset* or *training data* is a subset of a labeled dataset used for training your model.

A **testing subset** or *testing dataset* or *testing data* is a subset of a labeled dataset used for testing your model on new inputs. This is data that is held out to see how well a model generalizes.

The idea is illustrated in Fig. 7.6. The idea is that we first train the network using training data, and then validate it using testing data.[18]

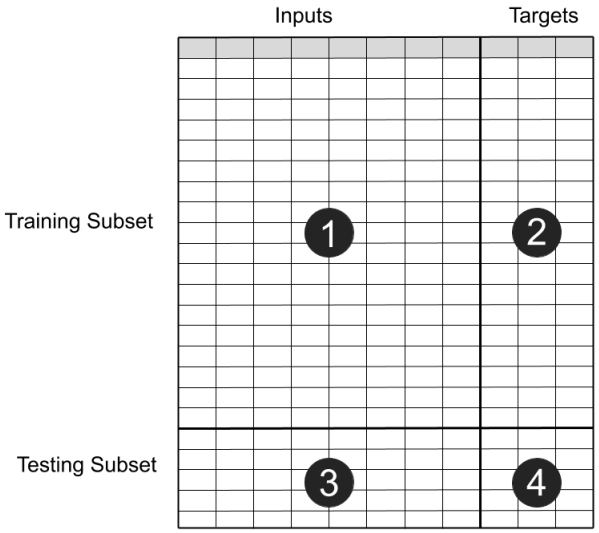


Figure 7.6: The rows of a labeled dataset (with inputs and targets) divided into a training and a testing subset. The training subset is used to train our model, and the testing subset is used to validate how well it generalizes. Thus we end up with four tables: (1) training inputs, (2) training targets or labels, (3) testing inputs, and (4) testing targets.

So what we actually often end up with, in supervised learning, is four tables. (1) Training inputs, (2) test inputs, (3) training targets, and (4) test targets. The training inputs and targets are used to train the model. The test inputs and targets are used to determine how well it performs on new data. In a model we might label these `train_inputs`, `train_targets`, `test_inputs`, and `test_targets`.

In practice, even more complex ways of partitioning labeled data into training and testing subset are used, for example splitting the data into training and testing sets different ways on different passes.[19] The particular training subset used in a given stage of training is often referred to as a "batch". We are keeping things simple here for illustrative purposes.

7.6 Supervised vs. Unsupervised Learning

We can distinguish two general ways of training a feed-forward neural network: supervised methods, where we tell the network what it should do with each input, and unsupervised methods, where we don't tell the network what we want it to do, but it figures out on its own (without a "supervisor") what to do. These concepts apply to recurrent networks as well, but we'll focus on feed-forward networks for now.

[18] In a machine learning context, we might also distinguish working from production data. Working data includes all the data mentioned above, used to train and test and validate a machine learning model. Production data is then data the machine learning model encounters in the "real world" when it has been deployed and is being used.

[19] The more general topic is cross validation, see `https://en.wikipedia.org/wiki/Cross-validation_(statistics)` and `http://scikit-learn.org/stable/modules/cross_validation.html`.

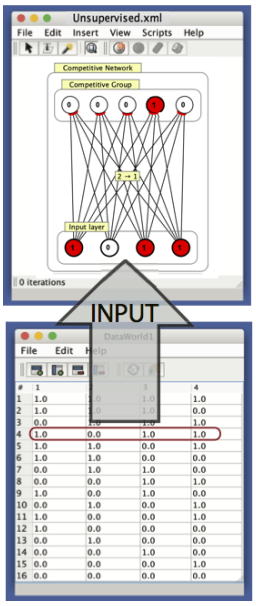


Figure 7.7: Illustration of how unsupervised learning relies only on an input dataset.

Unsupervised learning is learning without a teacher, which is covered in chapters 9 and 11. We don't tell the network what we want. It must adapt on its own, discovering statistical patterns in the inputs it is exposed to. There is just an input dataset, as shown in figure 7.7. There is no target data. In the example shown in the figure, we repeatedly expose the network to a set of inputs, and it will automatically develop feature detectors, which respond to specific clusters in the input dataset.

This is more neurally and psychologically realistic. After all, humans and animals don't constantly have a parent or teacher around telling them what's right or wrong. For this reason we saw that it was a general principle of learning in the neuroscience chapter 4. It is well known in psychology that a great deal of learning (e.g. "latent learning") occurs without explicit supervision; rats get to know their way around a maze even without explicit rewards [121]. It can also be useful in machine learning, since we oftentimes don't have training data available (hence the term "unlabeled data").

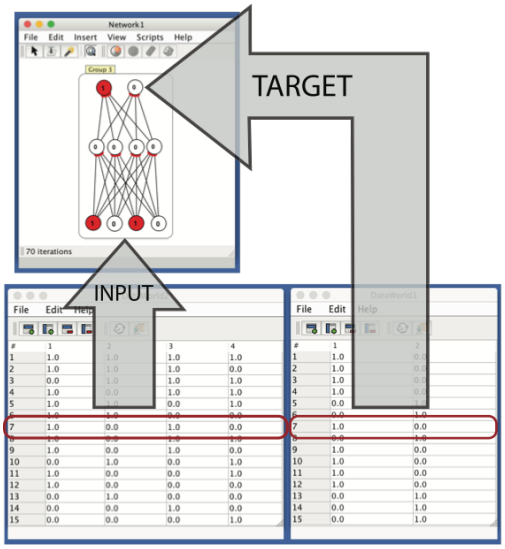


Figure 7.8: Illustration of how supervised learning uses both input and target datasets.

In the case of **supervised learning**, we tell the network what we want it to do. There is a teacher or trainer and so we have a labeled dataset. It's kind of like a parent telling a child, "No, that's wrong, this should be the answer!" For feed-forward networks, this means we give it a labeled dataset and say "implement that"! We train the network to perform a set of input-output associations. The general schema is illustrated in figure 7.8. We train a network using a supervised learning algorithm using a labeled dataset, which includes *two* tables, one for the inputs (the input dataset), and another for the outputs that we *should* get for each input (the target dataset). As each row of an input dataset is fed to the network, a corresponding row of a target dataset is used to determine how the network should respond.

Supervised learning is a huge topic that will be covered in chapter 12. Almost all of the major examples of things neural networks have done–drive cars, classify letters, translate languages or speech signals, etc. (see chapter 1)–were achieved using supervised learning. However these methods are not just useful in engineering. They have also been used to used in connectionism and computational cognitive neuroscience to study the kinds of representations the brain develops based on its exposure to inputs. Recall, for example, the discussion of the cerebellum and basal ganglia in chapter 4, both of which are thought to learn via supervised learning.

7.7 Other types of model and learning algorithm

Learning algorithms and the models they are used to train can be classified in other ways as well. For example, in chapter 13 we distinguish between supervised learning models that peform **classification task**s and **regression task**s, which is based on whether the target dataset contains categorical one-hot data (classification) or real-valued numerical data (regression).

Another distinction that is sometimes useful is that between a **generative model** and a **discriminative model**. A generative model is a model that can be used to generate prototypical features of a category with a given category label. They can be feed-forward networks that associate one-hot localist vectors with distributed feature vectors. For example, if you are asked to "describe a typical Golden Retriever", or "what are the height and weight of an average third grader", you can generate answers. These are contrasted with **discriminative model**s, where features are associated with categories. For example, if you are shown a picture of a dog and asked "is this a Golden retriever?", or a picture of a person and asked "is this a third grader?", you are simply discriminating a category based on inputs. A discriminative model is less demanding than a generative model, since you only must categorize items, rather than generating examples of items from a category (Compare multiple-choice questions with fill-in-blank questions on a test. Fill in the blank is harder, because you must generate and answer rather than just recognizing one answer as correct). Discriminative models, like classifiers (see chapter 13), are the focus of much of this book, and are well known in machine learning. Face and text recognition are usually based on discriminative models. But generative models are also important. For example, models that generate human speech or fake text are discussed in chapter 16.[20]

There are other types of learning algorithms and approaches to learning in neural networks as well.

An **evolutionary algorithm** (or genetic algorithm) is a class of algorithm that simulate evolutionary processes. You start by saying what counts as fitness and then set up an array of simulated genes and some kind of simulation. Then you run it! Millions of years of evolution can be compressed into minutes of time, as millions of simulations are run. When applied to neural networks, a batch of networks can be built, based on incrementally varying and mutated genes. The best are selected and further permuted, and the process continues. The script *evolveNetwork.bsh* in Simbrain evolves a network such that on average half of the nodes are active every few iterations. Run it a few times. You will see that it evolves a variety of solutions to the problem.

Another approach to training, **reinforcement learning**, is a variant of supervised learning where you don't just give the network a table of values to associate, but rather the action of an agent in a simulated environment, which tells the network when what it's doing is good or bad. This is a kind of virtual implementation of behaviorist psychology (Skinner famously thought all behavior could be explained as the result

[20] I am using non-standard and informal definitions of generative and discriminative models. A generative model is formally defined as a model of the joint probability distribution over inputs and outputs of a model, where the outputs are often categorical, which means that given a category you can estimate the associated features. A discriminative model is defined as a model of the probability of outputs given inputs.

of a history of reinforcement and punishment). So you take a virtual agent, put it in a virtual environment, and tell it what is good and bad in that environment. Getting the cheese is good. Getting attacked is bad. Now you simulate thousands or millions of explorations of the environment and it will learn to approach cheese and avoid predators. Some of the major recent developments in machine learning have been based on reinforcement learning (e.g. the success of AlphaGo) [114]. Another famous example is a system that learned to play a bunch of old Atari video games [85]. The nice thing about reinforcement learning is that it is fairly realistic. As noted in chapter 4, the basal ganglia are thought to mediate a form of reinforcement learning such that animals learn to maximize reward over time.

There are yet other methods, and machine learning is constantly evolving and adding more techniques and learning algorithms to its roster of approaches.

Chapter 8

Word Embeddings

Ellis Cain, Jeff Yoshimi

This chapter elaborates on the concept of a word embedding, which was briefly discussed in section 7.3. A word embedding associates each **token** in a set of tokens with a set of vectors (see chapter 6) in a way that that captures important characteristics of the set of tokens. Tokens can include words, word-parts, and other character sequences, but we focus on words here and it is standard to call these "word embeddings" rather than "token embeddings" (in fact we will often use "word" interchangeably with "token" simply as a matter of convenience).[1] In this way words can be "embedded" in a vector space, in that a word is associated with lists of numbers that can in turn be processed by a neural network, and associated with points in a space.

These techniques have a long history in linguistics and in the study of neural networks, but have become especially prominent in recent years with the advent of large language models (LLMs) like GPT (see chapter 17.4). While word embeddings may seem outdated and irrelevant due to the advent of LLMs, knowledge of the principles of word embeddings are extremely helpful for understanding LLMs due to the overlap, as these more advanced models build upon these principles and incorporate related techniques.

After giving some background on linguistics and natural language processing, we build up to the concept of a word embedding in stages. First we discuss document embeddings (associating whole documents with vectors of numbers), which are simpler to understand and serve as a useful basis for understanding word embeddings. Then we discuss word embeddings and their theoretical backing. Then we discuss the kind of pre-processing and workflow often involved in actually taking a text and creating an embedding. Throughout the chapter, we mention connections between text embeddings[2] and neural networks, whether the neural networks are used to create text embeddings or used with text embeddings as the input. Additionally, given the emphasis on converting words and other linguistic entities to vectors, this chapter overall also sheds further light on the concept of feature engineering (section 7.3) and "wrangling" data so it can be used in neural networks.

8.1 Background in Computational Linguistics

Computational linguistics is the study of language using computational methods. Analyzing large amounts of data and automating analysis for languages for under-documented language are two common applications of computational linguistics. Word embeddings originate in computational linguistics, and certain tools and concepts of the field will be useful in this chapter.

Linguistics is the study of language, which can be organized in terms of scale, going from smallest to largest unit of study:

[1] The concept of a "token" varies from one context to another. Tokens can also include punctuation, emojis, or word-parts and character sequences of various kinds. Tokens are usually extracted from a corpus using automated method, for example byte pair encodings. See `https://en.wikipedia.org/wiki/Byte_pair_encoding`.

[2] Formally a word embedding is a function from a set of tokens to a set of vectors, and a document embedding is a function from a set of documents to a set of vectors. We will use "text embedding" as a generic way to cover both cases.

1. Phonetics and phonology: the study of speech sounds.
2. Morphology: the study of words and word forms.
3. Syntax: the study of the structure or grammar, usually at the sentence level.
4. Semantics as the study of meaning, which can be at a variety of levels (words, phrases, sentences).
5. Pragmatics as the study of intentional meaning or implied meaning, such as implicit maxims and rules of conversations.

All of these levels have been studied using computational linguistics, and most of them have been studied in relation to neural networks. In some cases, features at these levels are used to generate vector representations of linguistic data (for example, word embeddings, the focus of this chapter). In other cases, neural networks have been used to analyze structures at these levels. Here is some more information on each level and their relevance to neural networks.

For spoken languages, the smallest unit would be the individual speech sounds that are used to create words. These are known as phonemes, such as the [b] in /bat/ or [p] in /pat/ and are often represented using specialized symbols of the International Phonetic Alphabet (IPA[3]). For applications such as speech recognition or text-to-speech, researchers may need to generate representations of these sounds in vector form. A simple way of doing this is to manually identify phonological features of phonemes and use them a binary feature vector, as in figure 8.1, which is based on Elman's early work [25]. Phonemes can also be associated with vectors using spectrograms. A spectrogram is a plot that shows time on the x-axis, frequency on the y-axis, and uses color to indicate the intensity or amplitude at a given frequency. Converting this data into vector representation can capture the frequency content of phonemes.

	Consonant	Vowel	Interrupted	High	Back	Voiced
b	[1	0	1	0	0	1]
d	[1	0	1	1	0	1]
g	[1	0	1	0	1	1]
a	[0	1	0	0	1	1]
i	[0	1	0	1	0	1]
u	[0	1	0	1	1	1]

Figure 8.1: Some of Elman's vector representations of phonemes, which involved hand-crafted feature vectors based on linguistic attributes. This illustrates the general idea that a linguistic item (here phonemes, later tokens) can be associated with a numeric vector.

The next level up is morphology, which focuses on the smallest units of meanings at the word level (morphemes). For example, "founded" can be divided into two constituent morphemes "found" and "-ed". Morphemes are used in tokenization, particularly with LLMs, such that GPT might have distinct vectors for "found" and "ed"[4]. Neural networks sensitive to morphological features have played other roles historically. For example, Rumelhart and McClelland created a network that took a (phonological representation) of a verb stem as input and produced the past-tense form as output, e.g. "look" to "looked" or "run" to "ran" [105].[5]

Syntax studies the order and hierarchical structures of words in sentences. This includes structures such as subject-verb-object (SVO) ordering of sentences or prepositional phrase attachment (i.e., "pet the

[3]See `https://en.wikipedia.org/wiki/International_Phonetic_Alphabet`

[4]These types of subtoken vectors assist LLMs with out-of-vocabulary issues because it allows them to recognize/represent subcomponents of an unknown word.

[5]Given that the network must learn a regular morphological pattern (verb stem + ed) and also exceptions (e.g. run to ran), questions surrounding this model and a competing dual process symbolic model were quite prominent for a time (recall the connectionist / symbolic debate discussed in section 3.3). Some of the debate pertaining specifically to the past tense model is summarized in [98].

frog [with the feather]"). Automated methods can be used to organize a representation of a sentence into grammatical categories (i.e., parts of speech) and to map out the hierarchical grammatical structure. For example, a treebank (such as the one shown in figure 8.2, associates a sentence with part-of-speech tags that reflects syntactic or grammatical structure and dependency relationships between parts of a sentence. The question of whether neural networks can deal with grammatical structures has been at the center of the connectionist / classicist debate (again, see section 3.3).[6]

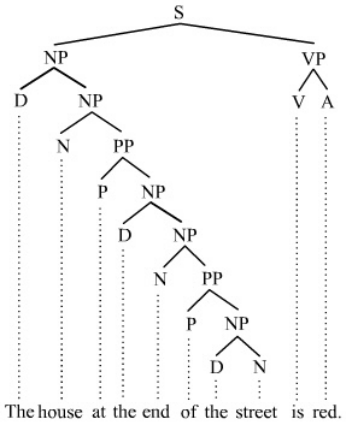


Figure 8.2: A treebank for the sentence "The house at the end of the street" which annotates each word with its syntactic or grammatical category and describes dependencies between parts of the sentence.

What about the meaning of words, or semantics? Since there isn't generally a clear unit of "meaning" for words, this meaning level is more broad and nebulous than the others. For example, the meaning of a word can be abstract or concrete ("justice" vs "cup") and is often context dependent ("bass" as the fish or instrument). Thankfully, clever researchers have developed ways to quantify and compare meanings. One way of modeling meanings that is friendly to neural networks is using vector space or "semantic space" representations of word meanings, where the meaning of a word is based on its position in a network of relations [27]. Word embeddings will be the focus of this entire chapter.

The last traditional level of linguistics is pragmatics, which can be understood as the implicit rules for language use. It is often not the literal meaning of words (i.e., semantics), but rather the meaning behind the usage. For example, if you asked someone if they studied for the exam, and they said that they "opened the textbook," you likely would understand that they mean that they didn't prepare well for the exam, even though their utterance does not literally mean that. Grice is well-known philosopher in this area [41, 42], and there has been a recent trend in using Bayesian statistics to model this type of pragmatics through Rational Speech Acts [40]. Pragmatics has not been studied much in neural networks, but has become important with LLMs since they can have actual conversations (see section 17.4), and it seems likely that in modern LLMs internal representation of pragmatic structures emerge.

8.2 Document embeddings

Though our focus in this chapter will mostly be on vector representations of words, historically vector representations of whole documents are important, because they are in some cases simpler and provide a background for understanding word embeddings.

One simple approach to document embedding is the **bag of words** approach, which associates documents with vectors of word frequencies. In these representations we don't care about the order in which tokens

[6] Briefly, classicists have argued that grammatical structure relies on a symbolic structure that associates constituents of sentences with symbols, where those symbols can be moved around and reorganized without changing their structure. It has been argued that neural networks either fail to have this ability or if they do, simply implement a symbolic structure [33]. It has also been argued that grammatical structure cannot be learned from the relatively limited stimuli available to children ("poverty of the stimulus" arguments, [9]). Connectionists have responded that grammatical structure can be learned [26] and that they can deal with compositional structure without being mere implementations of symbol systems [115].

occurs in a text, hence the term "bag". This approach ignores grammatical structure and just looks at how often different tokens occur in different documents. Simply put, we take each document and put the associated tokens into a bag, and count up how many tokens occur in each bag. Consider the following documents

Document 1 "The bass fish played the bass"

Document 2 "The fish played fish with the fish monger"

Each document will be associated with a bag of words, as shown in figure 8.1.

	the	bass	fish	played	with	monger
Document 1	2	2	1	1	0	0
Document 2	2	0	3	1	1	1

Table 8.1: Bag of words representation

These document-level bag of words embeddings have a number of problems. First, they ignore the syntactic structure of the sentences in the documents they encode. Second, they are influenced by the uneven distribution of certain terms in natural languages [96, 132]. To counteract this, we can use term frequency-inverse document frequency (TF-IDF) to calculate the importance of a specific term for a (set of) related document(s), which offsets how frequently a term appears (term frequency) by the number of other documents that also contain the term. This allows it to focus on meaningful terms and not those that generally appear frequently across many documents ('the', 'a', etc.).

Latent semantic analysis (LSA) is used with a set of documents to analyze semantic information and calculate document similarity. The set of documents is represented using a document-term matrix, where each row corresponds to a document, and each column corresponds to the frequency of a given term (see table 8.1). Then, singular value decomposition (SVD) is used for dimensionality reduction (see 6.3), resulting in a numeric vector for each document (based on term usage/frequency) which can be compared using cosine similarity (section 6.4) to get document similarity.

The above-mentioned methods are calculated at the document level, in that researchers calculate term frequencies in a given document or set of documents. We can then use these document embeddings to calculate document similarity, as with LSA. As we will see in the next section, these methods can also be adjusted and applied to the word level as well, allowing us to quantify word meaning as word embeddings. That is, the methods of document embeddings based on token occurrence and co-occurrence can be used to come up with word embedding techniques, and these word embeddings can be used to calculate how similar words are to each other.

8.3 Word embeddings

We now move from document embeddings to word embeddings. In this approach, instead of associating documents with vectors, we associate words or tokens with vectors. The general idea is that the words are embedded (or placed) into a vector space, in which the semantic information is captured and represented by the relative distance to other words in this space. Words that have similar semantic information would be near each other, and those which are unrelated would be far apart.[7]

For a sense of how the approach works, here are seven words embedded in a space, which illustrates that the relative relations between tokens are preserved (Fig. 8.3).[8] Notice that the creator of a medium (i.e., composer, author, painter) is closest to the medium or composition (i.e., music, book, painting). Interestingly, "critic" is closest to "author".

[7]We describe several common approaches to word embeddings here, but there are others. For example, LLMs use a random embedding (a random association between tokens and vectors) that is trained using gradient descent. In other cases LLMs themselves are used to produce embeddings.

[8]These were plotted using `http://vectors.nlpl.eu/explore/embeddings/en/`, which is a helpful interactive page for getting an intuitive feel for how word embeddings work.

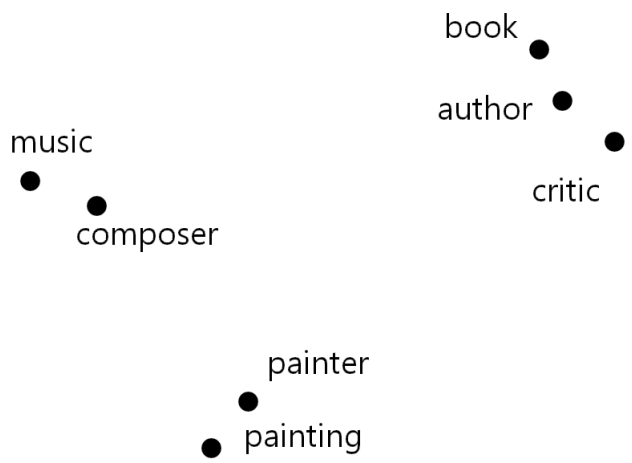


Figure 8.3: Example of a word embedding in a semantic space. Each word is associated with a point in a space and the distance between points corresponds to semantic similarity. Notice that intuitively similar words are near each other in this space.

The theoretical support for word embeddings is from distributional semantics (DS), an approach to semantics that emphasizes the information about a word's meaning is contained in linguistic context [47, 32] and the statistical properties of how that word is used (i.e., a distribution of how a word is used). Distributional semantics builds on earlier usage-based theories of language such as Wittgenstein's theory of "meaning as use" [129].[9] DS is often illustrated with the following quote, attributed to John Firth: "You shall know a word by the company it keeps." For example, when someone talks about a *river*, they may also mention *water* or *bank* (as in river bank), helping the listener correctly decode the intended meaning. Or, from the quote, you are able to interpret *company* as referring to other words in the sentence, and not a *business*, based on the earlier context.

8.3.1 Co-occurrence Based Word Embeddings

The high level idea with co-occurrence based embeddings is that two tokens will be close to each other in a vector space if they tend to appear near the same other words in the training corpus. Words like "nurse" and "doctor" will be near each other because they both tend to occur near words like "hospital", "patient", or "disease".[10] In other words, because the usage of these words overlap, their meaning should be similar or related to a certain extent.

Here is an example that illustrates the idea that words with similar meanings would be nearby each other in the vectors space (that is, have similar vector representations):

	Dimension 1	Dimension 2	Dimension 3	Dimension 4	...	Dimension n
Book	0.8	0.1	-0.2	0.5	...	0.7
Critic	0.6	0.3	-0.1	0.4	...	0.5
Painter	0.2	0.8	0.4	0.1	...	-0.3
Painting	0.3	0.7	0.5	0.2	...	-0.2
Music	-0.1	-0.4	0.9	0.3	...	0.5
Composer	-0.2	-0.3	0.8	0.4	...	0.6

Table 8.2: Vector space embedding representation. Note that similar words (top 2, middle 2, and bottom 2) correspond to vectors that are near each other.

[9] As Wittgenstein said, "a *large* class of cases of the employment of the word 'meaning'... can be explained in this way: the meaning of a word is its use in the language" (*Philosophical Investigations*, section 43).

[10] Thanks to Eric Schwitzgebel for this example.

Note that the columns themselves (the dimensions of the vector space) don't have any clear interpretable meaning[11]. Generally speaking, word embeddings are trained on a large text corpus, after which a type of dimensionality reduction is used. After dimensionality reduction, the columns can no longer be interpreted as co-occurrence with specific terms in the corpus. Rather, it is the pattern across them that matters, as the relative semantic relationships between tokens are encoded in this semantic space.

The nice thing, again, is that we can actually picture these by projecting from the high dimensional embedding space to two dimensions and then literally see the distance relationships between tokens, as in figure 8.3. The generated embedding space is much like our own semantic space [72], with the advantage that we can track and compare word meanings mathematically and visually.

8.3.2 Co-occurrence Matrices

Based on the distributional theory mentioned above, we can track co-occurrences and usage to patterns to capture the meaning of a set of words. These usages patterns are represented by a co-occurrence matrix. To construct a co-occurrence matrix, we must iterate across every sentence in a document or training corpus, where we iterate across each word and count the co-occurrences with the surrounding context. Each word is iterated as the 'label' / 'target', while the surrounding words serve as the 'context'. A window size is defined, which designates how many of the surrounding words to include in the 'context'. This context can be either unidirectional, using the preceding text, or bidirectional, using all surrounding context. Once the whole training corpora has been processed, the result is a co-occurrence matrix where each cell represents the raw co-occurrence counts for a given label-context pair.

Even at this point, if we compare across the rows of related tokens, their co-occurrences should generally be similar. Still, not every word is used with the same frequency and may negatively impact the quality of our word embeddings. Similar to our document embeddings, some words like determiners ('the', 'a') may be over-represented and skew the co-occurrence matrix.

Generally, there are two approaches: filtering and normalization. The simplest fix is to simply filter out these high-frequency, low-impact words. These words are called *stopwords*, and various NLP packages will have stopword lists for various languages.

Another approach (usually in addition to removing stopwords) is to normalize word embeddings so that common words don't drown out the vector representations. A common method is using a positive-pointwise mutual information (PPMI) transform to weight the matrix[12].

PPMI weights the co-occurrence values to avoid word-frequency-bias in embeddings. Words like "the" and "a" that should not be considered meaningful in terms of co-occurrence are down-weighted. Less frequent words on the other, like "shrubbery" or "herring", that are more meaningful in terms of co-occurrences, are up-weighted. The result is a set of n-dimensional vectors for a set of words, which are referred to as "word embeddings." For a more in-depth explanation and discussion of word embeddings and distributional semantics, see [70].

8.3.3 Neural Network Based Embeddings

Co-occurrence based word embeddings are not the only kind of word embedding. Besides these, neural networks can also be used to create word embeddings. A well-known example is Word2Vec [81], which trains shallow neural networks to predict targets based on the surrounding context.[13] After the network is trained, the weights of the hidden layer are extracted to serve as the word embeddings. In other words, sometimes co-occurrence based word embeddings are used as data for a neural network, whereas other times neural network based embeddings may be used for a different neural network.

[11] See the linear algebra chapter. These columns are sometimes called feature but feature implies something that can be easily interpreted.

[12] Lenci [70] explains PPMI as measuring "how much the probability of a target-context pair estimated in the training corpus is higher than the probability we should expect if the target and the context occurred independently of one another."

[13] A code-based walk through of the algorithm is here: `https://www.tensorflow.org/tutorials/word2vec`.

8.3.4 Evaluation of word embeddings

How do we know that a given word embedding accurately captures the meaning of a set of tokens? How can work embeddings be evaluated? Previous research on word similarity and relatedness has shown that directly asking for relatedness judgments can accurately capture word relations [31]. Therefore, one general method of evaluation is by collecting a set of human judgements and to use these to evaluate a given embedding relative to these judgments (though some have questioned this approach, e.g. [101]). These can be thought of as placing a ceiling on the performance of an embedding. Several "gold standard" usage databases exist, such as WordSim-353 [31, 2].

However, recent models (pre-LLM) had already reached or surpassed human performance on tasks such as similarity evaluation. Since the models have reached or surpassed theoretical performance ceilings based on human judgments, how can future improvements be evaluated? In response to this situation, Felix Hill and colleagues [54] set out to create a standard that separates similarity from association (previous standards had ignored the distinction) and therefore create a better standard for evaluation. Association refers to relatedness between two concepts, whereas similarity refers (almost) to synonymy. For example, "car" and "tire" would be considered associated but not similar, while "glasses" and "spectacles" are similar. While these issues are less prominent with LLMs, there still is active research exploring the shortcomings and common errors of models to create new benchmarks to aid in model evaluation [17].

While different algorithms may improve the model performance and similarity to our own semantic representations, the corpus quality also has an impact on model performance. Larger training corpora generally improve the quality of the derived embeddings, since the increased amount of data ideally adds more context to be processed. GloVe embeddings are trained on various training corpora, varying from 1 billion tokens to 42 billion tokens [95].

Besides the amount or size of the training corpora, the type of documents is also important; training solely on works of fiction would lead to different embeddings than a model trained on non-fiction, and so on. It is important to consider the meaning you are trying to capture (generalized or specific to a field, such as medical documents).

8.4 Workflow: Creating Word Embeddings

In practice there are many steps involved in applying these ideas. Here is a sample workflow or pipeline:

1. Sentence segmentation.
2. Word tokenization
3. Normalization and filtering
4. Creation of word embeddings

To get a sense for how this works, let's apply this pipeline to a sample document:

> My work is a matter of fundamental sounds, made as fully as possible. Even though they are fundamental, they bring rich aromatic hints of humor. If people want to have headaches among the overtones, let them. And provide their own aspirin. (Adapted from a letter from Samuel Beckett to Alan Schneider, 1957)

8.4.1 Sentence segmentation

For sentence segmentation, the paragraph or document is segmented into sentences. This step is particularly important when using text that has been scanned using optical character recognition (OCR), where errors might occur. Our sample document would be segmented into four sentences (given that we are segmenting just on periods):

1. *My work is a matter of fundamental sounds, made as fully as possible.*
2. *Even though they are fundamental, they bring rich aromatic hints of humor.*

3. *If people want to have headaches among the overtones, let them.*
4. *And provide their own aspirin.*

8.4.2 Word tokenization

Once the document has been segmented into sentences, a tokenizer is used to split each sentence into the comprising words. This yields a list of lists of tokens, as in

1. *my, work, is, a, matter, of, fundamental, sounds, made, as, fully, as, possible*
2. *even, though, they, are, fundamental, they, bring, rich, aromatic, hints, of, humor*
3. *if, people, want, to, have, headaches, among, the, overtones, let, them*
4. *and, provide, their, own, aspirin*

8.4.3 Normalization

Following tokenization, the words/tokens are generally normalized to remove capitalization or certain punctuation marks, such that the words are consistently in the same form. In this step, stopwords can be filtered out. In our case:

1. *work, matter, fundamental, sounds*
2. *fundamental, bring, rich, aromatic, hints, humor*
3. *people, headaches, overtones*
4. *provide, aspirin*

8.4.4 Create the word embeddings

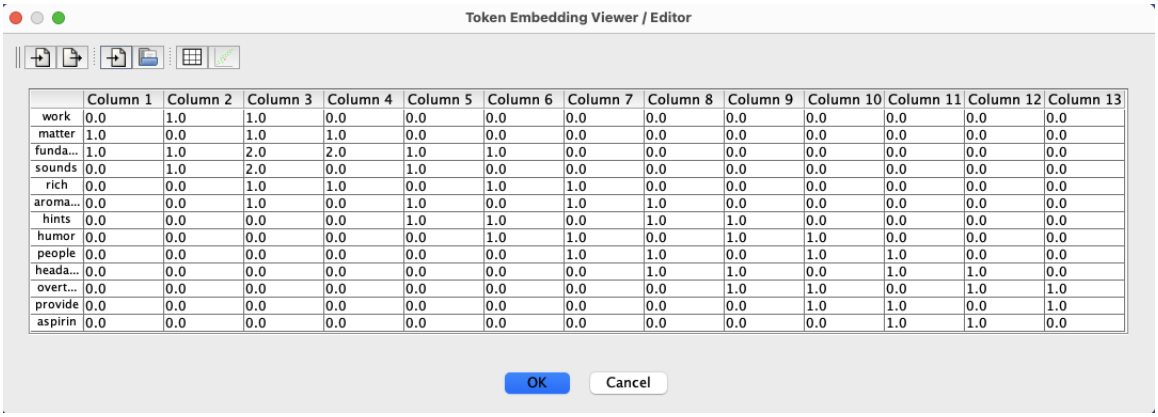
For a basic word embedding algorithm, the label-context pair co-occurrences are tracked in a co-occurrence matrix. For the first sentence, we would start with "work" as the first target. Then, given a bidirectional window size of 2, the surrounding context would be "matter" and "fundamental". Therefore, the co-occurrence pairs would be [work, matter] and [work, fundamental]. We would then create a co-occurrence matrix, with targets on the rows, and context on the columns. After processing the first sentence, the co-occurrence matrix can be seen in table 8.3:

Targets	work	matter	fundamental	sounds
work	0	1	1	0
matter	1	0	1	1
fundamental	1	1	0	1
sounds	0	1	1	0

Table 8.3: Co-occurrence matrix. Each word vector would be a row in the matrix. Note that this is *only* for the first sentence.

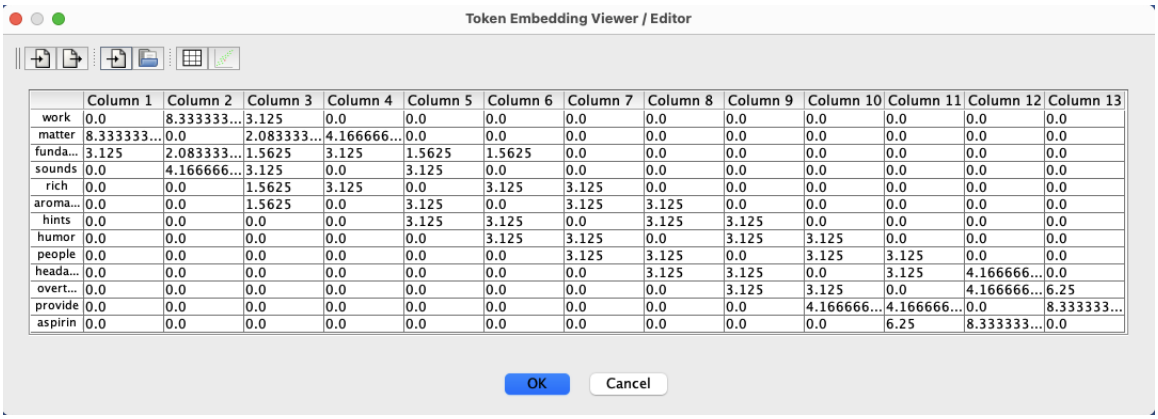
Once every sentence has been processed, the final co-occurrence matrix can be seen in figure 8.4.

After processing the whole document and calculating a co-occurrence matrix, we would use PPMI to weight the vectors (recall that PPMI is used to normalize word embeddings so that common words don't drown out the representations). The result can be seen in figure 8.5.



	Column 1	Column 2	Column 3	Column 4	Column 5	Column 6	Column 7	Column 8	Column 9	Column 10	Column 11	Column 12	Column 13
work	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
matter	1.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
funda...	1.0	1.0	2.0	2.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
sounds	0.0	1.0	2.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
rich	0.0	0.0	1.0	1.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0
aroma...	0.0	0.0	1.0	0.0	1.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0
hints	0.0	0.0	0.0	0.0	1.0	1.0	0.0	1.0	1.0	0.0	0.0	0.0	0.0
humor	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	1.0	1.0	0.0	0.0	0.0
people	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	1.0	1.0	0.0	0.0
heada...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	1.0	1.0	0.0
overt...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	1.0	1.0
provide	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0	1.0
aspirin	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	1.0	0.0

Figure 8.4: Co-occurrence matrix for the example sentences. Bidirectional window size of 2, without PPMI.



	Column 1	Column 2	Column 3	Column 4	Column 5	Column 6	Column 7	Column 8	Column 9	Column 10	Column 11	Column 12	Column 13
work	0.0	8.333333...	3.125	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
matter	8.333333...	0.0	2.083333...	4.166666...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
funda...	3.125	2.083333...	1.5625	3.125	1.5625	1.5625	0.0	0.0	0.0	0.0	0.0	0.0	0.0
sounds	0.0	4.166666...	3.125	0.0	3.125	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
rich	0.0	0.0	1.5625	3.125	0.0	3.125	3.125	0.0	0.0	0.0	0.0	0.0	0.0
aroma...	0.0	0.0	1.5625	0.0	3.125	0.0	3.125	3.125	0.0	0.0	0.0	0.0	0.0
hints	0.0	0.0	0.0	0.0	3.125	3.125	0.0	3.125	3.125	0.0	0.0	0.0	0.0
humor	0.0	0.0	0.0	0.0	0.0	3.125	3.125	0.0	3.125	3.125	0.0	0.0	0.0
people	0.0	0.0	0.0	0.0	0.0	0.0	3.125	3.125	0.0	3.125	3.125	0.0	0.0
heada...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.125	3.125	0.0	3.125	4.166666...	0.0
overt...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	3.125	3.125	0.0	4.166666...	6.25
provide	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	4.166666...	4.166666...	0.0	8.333333...
aspirin	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	6.25	8.333333...	0.0

Figure 8.5: Resulting co-occurrence matrix after PPMI has been applied. Bidirectional window size of 2.

8.4.5 Using a word embedding to make a document embedding

Of course word embedding can be used to generate document embeddings, simply by associating each token in a document with its vector embedding and concatenating the result. Thus we can associate a whole document with a matrix, where each row is the vector embedding for one word or token in the document. A prominent example where this idea is used is with LLMs like ChatGPT (see chapter 17), where a context window — a set of prompts and responses — is converted into a matrix using a vector embedding. Let's take a small, brutish context window: "dog chases cat and cat chases dog!". First we tokenize the input, then associate each token with an index, like this (notice that the punctuation mark is a token):

3	1	4	5	2	1	3	6
`dog`	`chases`	`cat`	`and`	`cat`	`chases`	`dog`	`!`

Now we can take each integer and associate it with a row of an embedding matrix, like this, shown here with integer labels on rows to make the idea clear.

$$\left[\begin{array}{c|ccc} 1 & 0.5 & 0.1 & 0.3 \\ 2 & 0.2 & 0.4 & 0.6 \\ 3 & 0.7 & 0.8 & 0.9 \\ 4 & 1.0 & 1.1 & 1.2 \\ 5 & 1.3 & 1.4 & 1.5 \\ 6 & 0.3 & 0.1 & 1.2 \end{array}\right]$$

If we take each token and the corresponding row for it, and stack the results vertically, we end up with a matrix representation of a set of words (that is, a document, or in an llm, a context window), like this:

$$\begin{bmatrix} 0.7 & 0.8 & 0.9 \\ 0.5 & 0.1 & 0.3 \\ 1.0 & 1.1 & 1.2 \\ 1.3 & 1.4 & 1.5 \\ 0.2 & 0.4 & 0.6 \\ 0.5 & 0.1 & 0.3 \\ 0.3 & 0.1 & 1.2 \end{bmatrix}$$

So there's our vector embedding for the document, what we can a "word embedding matrix" or a "token embedding matrix". This matrix is suitable for processing in a neural network, like an LLM.

You should confirm that the token embedding matrix above makes sense, that each row corresponds to the corresponding token in the sentence "dog chases cat and cat chases dog!"

Chapter 9

Unsupervised Learning

Jeff Yoshimi

9.1 Introduction

Unsupervised learning is learning without a teacher. It is a method for changing the weights of a network without telling a network what we want it to do (that is, without consulting target data or "labels"; see chapter 7). You set a neural network loose in an environment (which usually means exposing it to a bunch of samples from a table of input vectors) and see what it comes up with. How can a neural network adapt to an environment on its own, without being told what to do? How can it "self-organize"? Given how often animals find themselves in this situation—of not having a teacher around—it's an important topic.[1] It turns out that unsupervised learning algorithms are quite powerful, both as engineering tools in machine learning and also as a way of modeling the development of certain types of neural circuits and psychological capacities.

We begin the chapter with a discussion of Hebb's rule, a classic associative learning rule that allows us to associate paired stimuli. We consider how Hebb's rule and its variants can be used to create feed-forward pattern associators. We then consider how a Hebbian-type rule (Oja's rule) allows layered networks to achieve dimensionality reduction, where one layer extracts the most statistically important information from the previous layer. Finally, we review competitive learning algorithms that can be used to associate output neurons with clusters in an input space. This culminates in a discussion of self-organizing maps, which can be used to model certain features of the cerebral cortex and can also be used as a machine learning tool.[2] In chapter 11, we consider how unsupervised methods can be used to train recurrent networks.

9.2 Hebbian Learning

Learning in a neural network corresponds to adjustment of its weights by application of a **learning rule**. A learning rule is a method for updating the weights of a neural network over time. It is the counterpart to an activation function (chapter 5), but it acts on the weights between nodes rather than on the activation levels of the nodes. Some learning rules are visible in Simbrain by editing a weight and consulting the *update rule* drop-down box.

A learning rule for weight $w_{j,k}$ can be written as a delta value, $\Delta w_{j,k}$, meaning "change in strength of weight $w_{j,k}$" (the symbol Δ is often used to describe changes in a variable). At any time step, you simply add the current value of $\Delta w_{j,k}$ to the weight's current strength to get the weight's new strength at the next

[1] Of course, animals do experience unconditioned rewards and punishments and their learning is in that sense "supervised." Skinner famously thought this was enough to explain all animal behavior, and this is the basis of reinforcement learning (RL) approaches. So that form of supervised learning still has a chance, though RL researchers often also draw on unsupervised methods.

[2] There is a lot more to unsupervised learning than what is covered here. For an overview in the context of machine learning, see `http://scikit-learn.org/stable/unsupervised_learning.html`.

time step. If we let a prime symbol $'$ indicate the next time step, then we have this rule:

$$w'_{j,k} = w_{j,k} + \Delta w_{j,k}$$

It's quite simple. The strength of the weight $w_{j,k}$ at the next time step will be equal to its current value plus some delta value. It's just addition. For example, if $w_{1,4} = -1$ and $\Delta w_{1,4} = 4$, then $w'_{1,4} = -1 + 4 = 3$.

Hebbian learning is one of the oldest and simplest learning algorithms for neural networks. It is biologically plausible (it is based on Long Term Potentiation, discussed in chapter 4) but has limitations that prevent it from being widely used in its basic form (variants of the Hebb rule are, however, widely used).

The basic idea of Hebbian learning is that when connected neurons are both active, the weight connecting them is strengthened. You know the slogan: "neurons which fire together, wire together." Donald Hebb proposed this idea in the 1940s, before there was experimental support for it. As he put it:

> When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased [51].

Formally, the Hebb rule states that the change in a weight $w_{j,k}$ at a time is equal to the product of a **learning rate** ϵ, the source node's activation a_k, and the target node's activation a_k.

$$\Delta w_{j,k} = \epsilon a_j a_k$$

The learning rate ϵ controls the rate at which the weights change each time the rule is applied (each time we press "step" in Simbrain). If we set $\epsilon = 0.00002$, the weights will change very slowly. If we set $\epsilon = 10$, they will change very quickly. Note that we can *stop* learning by setting $\epsilon = 0$ (in which case $\Delta w_{j,k}$ will always be 0). Most of the learning algorithms we consider will have some sort of learning rate. Learning rates are usually set between .01 and 1.

When nodes j and k are clamped (so that their activations can't change), the rule is especially easy to apply. It is simply the product of the two nodes' activations times the learning rate. If both neurons have an activation of 1, then at each time step we simply add the learning rate to the weight. For example, if $a_j = 1, a_k = 1, \Delta w_{j,k} = 1$, then at time step the weight strength will increase by 1. If $a_j = 1, a_k = 1, \Delta w_{j,k} = .5$, then at time step the weight strength will increase by .5.

Notice that if both the source and target node activations of a Hebbian weight are positive, then the weight's value will increase (they "fire together" so they "wire together"). If one activation is positive and one is negative, the weight will decrease. If both activations are negative, they will also increase (since a negative number times a negative number is positive).[3] If either activation is 0, the weight will not change, which is an important baseline case: neurons that *don't fire*, don't wire! Most our neurons are quiet most of the time, and thus the synapses connected to them don't change.

Making a simple Simbrain network to test and explore these ideas is easy. Create two nodes and connect them with a weight. Double click on the weight, set its update rule to Hebbian, and set the learning rate to whatever you desire. Clamp both nodes. Now when you run the network, you will see the weight change: it gets larger when both nodes are negative or positive, and lower when one node is negative and the other is positive. The rate of change is set by the learning rate. The weight will generally just "explode", i.e. increase or decrease indefinitely. In Simbrain, the weight strengths are bounded, so they will often just race towards these bounds.

Example 1. Suppose we have two nodes with activations a_1 and a_2, and that the activations of the nodes are clamped. Further suppose we have

> The activations on the nodes: $(a_1, a_2) = (1, 2)$
> The weight: $w_{1,2} = -1$

and that the learning rate is $\epsilon = 0.5$. What will the value of the weight be after three time steps?

[3] The negative-positive and negative-negative cases are biologically implausible but still useful in many algorithms.

First, we compute $\Delta w_{1,2}$, the amount that the weight will change at each time step:

$$\Delta w_{1,2} = \epsilon \cdot a_1 \cdot a_2 = (0.5)(1)(2) = 1$$

The value of $\Delta w_{1,2}$ never changes because the activations are clamped. So the weight of $w_{1,2}$ will change by $\Delta w_{1,2} = 1$ for each of the time steps. Since $w_{1,2}$ begins at -1, it will be 0 after the first time step, it will be 1 after the second time step, and it will be 2 after the third time step. **Answer:** $w_{1,2} = 2$.

Example 2 You can see from the previous example that over time the weights will go towards extreme values with the Hebb rule. This is a problem with Hebbian learning: it tends to push weights towards positive or negative infinity. One way to slow this down is to use a smaller value for ϵ. Suppose in example 1 that $\epsilon = 0.01$. What would the values for the weight be at each time step? **Answer:** Time 1: $w_{1,2} = -0.98$, Time 2: $w_{1,2} = -0.96$. Time 3: $w_{1,2} = -0.94$.

9.3 Hebbian Pattern Association for Feed-Forward Networks

A classic use of the Hebb rule is to model pattern association. The underlying idea is simple: when an agent perceives multiple stimuli at the same time or nearly the same time (what the classical British empiricists discussed in chapter 3 called "contiguity in time and place"), we tend to associate them. If you often hear a song while being around some person, you may come to associate the two. The song may later remind you of that person.[4] If a dog hears a bell just before receiving food, it will associate the two. Thus classical conditioning can, to a first approximation, be explained by the Hebb rule.[5] But we will see that simple Hebbian pattern associators are problematic, so the rule must be supplemented in various ways.[6]

We begin by considering pattern association in feed-forward networks. Feed-forward networks can be thought of as functions that take a vector (a list of numbers; see chapter 6) as input and produce a vector as output. A multiple layered feed-forward network computes a series of vector-to-vector transformations based on the intervening weights (which, recall, can be represented by weight matrices).[7] As the weights in this network change, the way it associates input vectors with output vectors changes. That is, the function associated with a feed-forward neural network changes as its weights change.

The Hebb rule can be used to train a feed-forward network to learn a set of pattern associations, and thereby to approximate a function between a set of input vectors and a set of output vectors. The basic idea is simple. We simply set the (clamped) input and output nodes to the patterns we desire, and then apply the rule.

Suppose we want to use the Hebb rule to train a network to learn the following three associations:

- $(1, 0, 0) \rightarrow (1, .4)$
- $(0, 1, 0) \rightarrow (.8, .3)$
- $(0, 0, 1) \rightarrow (.5, .7)$

[4] Associations like these are a common literary theme. Marcel Proust's epic *Remembrance of Things Past* is a thousands-of-pages long novel that begins with memories inspired by the smell of a cookie, a "crumb of madeleine" [99]. Gabriel García Márquez's *Love in the Time of Cholera*, opens with "It was inevitable: the scent of bitter almonds always reminded him of the fate of unrequited love."

[5] An actual model of the conditioning is the Rescorla Wagner model, which influenced the development of similar models in reinforcement learning. Both can be explored in Simbrain. See the scripts *rescorlaWagner.bsh* and *actor_critic.bsh*.

[6] It should be noted that Hebbian pattern associators (especially feed-forward pattern associators) are really in a gray area between unsupervised and supervised learning. They are unsupervised in that there is no explicit teacher. However, in practice, we often clamp the nodes of these network using desired outputs. From an unsupervised perspective, we can think of this as values that were set "by nature" (e.g. the sound of a song in one neural population, and sight of a friend in another neural population), so it can still be thought as unsupervised, and it is still the case that there there is no explicit training signal. Also, the Hebb rule is a natural place to start in our study of unsupervised learning. So we cover Hebbian pattern associators here, even though they are on the cusp between the two types of learning.

[7] Recall from algebra that a function is a rule that associates objects (usually numbers) in a domain with unique objects (usually other numbers) in a range. For example $f(x) = x^2$ associates real numbers with positive real numbers: $f(2) = 4$, $f(-1) = 1$, and $f(0) = 0$. We can also think of feed-forward neural networks as computing functions, which associate input vectors with output vectors. For example, a network with 3 input nodes and 3 output nodes takes 3-dimensional input vectors with 3 dimensional output vectors. It can be thought of as a rule which associates vectors in a three-dimensional vector space with vectors in another three-dimensional vector space.

Something like this can occur in the brain. We can imagine that one population of neurons receives one kind of input (e.g. auditory signals from a song, or olfactory signals from a bowl of bitter almonds) and another part of the brain receives another kind of input (e.g. visual inputs corresponding to a person). The Hebb rule says that since these two populations of neurons are both firing at the same time, an association should form between the two patterns. I often smell bitter almonds when around that person, so later, when I am around bitter almonds again, it arouses a visual memory.

To build this kind of model in Simbrain, take the following steps:

(1) Create the network. Follow the template shown in figure 9.1. Make all the neurons clamped, the output neurons linear, and set all weights to Hebbian with a learning rate of 1. Also, initialize the weights to a value of 0 by pressing w then c.

(2) Train the network. Set the input and output nodes to their desired values. Now iterate Simbrain once. The weights will be updated according to the Hebb rule, and our first association has been formed. You will notice the synapses change color and size. Repeat this process exactly once for each association.

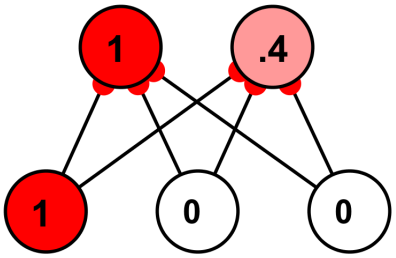


Figure 9.1: Learning one association in a feed-forward network using the Hebb rule.

(3) Test the network's ability to recall patterns. Now we want to test the network to see how well it can recall these associations. Will it recall the right target pattern given a source pattern? Has it properly implemented the vector-valued function above? To test the network, we need to do two things. First, we must unclamp the output neurons. Second, we must stop the weights from changing by clamping them (a **clamped weight** is the same as a clamped node; when weights are clamped, their value no longer changes). Now we are ready to test. For the first input pattern, set the input nodes to $(1, 0, 0)$, and iterate the workspace. The output neurons should produce the correct pattern, $(1, .4)$. Similarly for the other input-output pairs.

Notice that this particular Hebbian pattern associator associates localist category vectors (one-hot vectors) with distributed feature vectors. This is similar to what is sometimes called a **generative model**, that is, a model that can be used to generate prototypical features given a category label. But for the rest of this chapter, we will "reverse" the situation, focusing on **discriminative model**s that associate distributed feature vectors with one-hot categorical or localist vectors. (On generative vs. discriminative models, see chapter 7).

The simple Hebb rule, used as a pattern associator, is extremely brittle. Consider the following drawbacks of the method we used to train the associator.

First, we had to carefully clamp and unclamp the nodes and weights in a sequence. In fact, the method we used is almost like a form of supervised learning (chapter 12), because we carefully set the output nodes to target values. In a living network, no such clamping and unclamping occurs. The learning just happens automatically.

Second, we only updated once. If we had kept running the network, the weights would keep getting larger and the correct associations would have been wiped out.

Third, we used **orthogonal** input vectors. Recall from chapter 6 that orthogonal vectors have a dot product of 0. For example, $(1, 0)$ and $(0, 1)$ are orthogonal because $(1, 0) \cdot (0, 1) = 1 \times 0 + 0 \times 1 = 0$. One-hot encodings (see chapter 7), where just one node is on and the others are off, are an easy way to obtain orthogonal inputs. One-hot input vectors give rise to learning on *completely different weights*. The vector $(1, 0, 0)$ only changes the weights fanning out from the first input node. The vector $(0, 1, 0)$ only changes weights fanning out from the second node. On the other hand, if input vectors are *not* orthogonal, then

they interfere with one another and we are no longer guaranteed perfect recall. This is sometimes called **cross talk**. To see this, try repeating the example just given, but using input vectors $(1, 1, 0)$, $(0, 1, 1)$ and $(1, 1, 1)$. You will not get good results, because the input vectors overlap. In general, the degree of cross-talk in Hebbian pattern association is proportional to how similar the input vectors are.[8]

In practice, more robust variations on Hebb's rule are required to model conditioning and associative learning (see note 5). But this example still gives a flavor of how Hebbian learning works and what it can do.

9.4 Oja's Rule and Dimensionality Reduction Networks

We have seen that the Hebb rule tends to make weights explode to their maximum or minimum values. One solution to this problem is to update all the fan-in weights on a node together, and to update them using Hebb's rule and then rescale the weights so that they sum to 1 (see chapter 7); this is sometimes called "renormalizing" the weights). This prevents the weights from "blowing up" and means that what matters in a set of weights is just the *relative* size of each weight. This kind of computation can also be done locally. This is done using Oja's rule.[9]

One thing that Oja's rule can do is dimensionality reduction. Recall from chapter 6 that dimensionality reduction is a way of taking high dimensional data and representing it, usually with some distortion, in a lower dimensional space. We have seen how dimensionality reduction helps us to analyze data from neural networks. But it turns out that feed-forward neural networks can also *implement* dimensionality reduction. Suppose you have a feed-forward network with 3 input nodes and 1 output node. This is a form of dimensionality reduction! It's a way of taking vectors in a space (the input space) and projecting them to a lower dimensional space (the output space). We can call feed-forward networks where there are more input nodes than output nodes **dimensionality reduction networks**. A random 3-1 network might not do any significant type of dimensionality reduction, but it does take a set of points in a 3-dimensional input space, and for each of them produce some point in a 1-dimensional output space. So we have a 3-1 dimensionality reduction network.

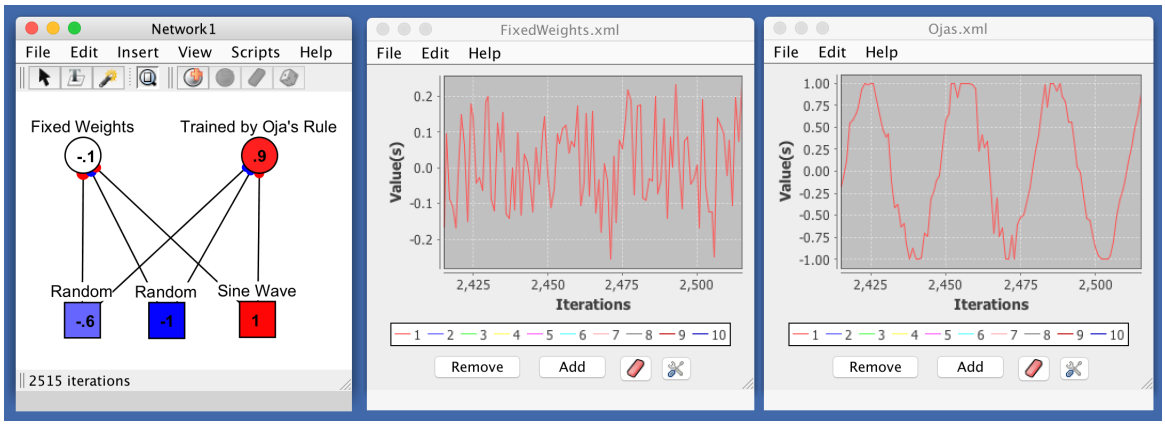


Figure 9.2: An illustration of Oja's rule for dimensionality reduction. Two of the input nodes produce random noise, and the third produces a sine wave. Think of each output node as performing a 3-to-1 dimensionality reduction. The output node on the left was initialized with fixed random weights that don't change. The output node on the right has weights trained by Oja's rule. Time series for two output nodes are shown in the middle and right panels. Notice that the node with weights trained by Oja's rule extracts the sine wave, which is the most informative part of the 3-dimensional input signal.

When Oja's rule is used on a set of fan-in weights, the resulting network performs a meaningful type

[8] For a more detailed discussion, see [28], p. 105.
[9] The formula is $\Delta w_{j,k} = \epsilon a_k (a_j - a_k w_{j,k})$.

of dimensionality reduction, specifically PCA (Principle Components Analysis).[10] This is the same method used by default in the Simbrain projection component, so it is something you may already have some intuitive familiarity with.

An example illustrating one use of Oja's rule is shown in Fig. 9.2. Two of the input activity generators produce random values, and the third produces a sine wave. In the 3-dimensional input signal the output nodes are receiving, the sine wave is the principal part. We'd like to be able to recover just the sine wave. That's what Oja's rule does to the extent that it implements PCA. The output node on the left has fixed random weights. The output node on the right has weights trained by Oja's rule. Think of these as two 3-to-1 dimensionality reduction networks. Notice that the time series plot of the activation of the node on the right shows the sine wave, but the time series plot on the left does not.

Something like this may be what's happening in the human brain. Hebb-like learning rules have been shown to operate in the brain (LTP, LTD, STDP, etc.; see chapter 4). Moreover, it is known that successive layers of cortical network extract increasingly refined and informative signals from preceding layers. Something like Oja's rule might be at work during this extraction, but it remains an open question.

9.5 Competitive learning

We now consider a second general type of unsupervised learning, **competitive learning**, where networks automatically detect statistical tendencies in an input environment. The Hebb rule was able to pick up associations between inputs and outputs automatically, but it was brittle. Competitive learning is much more robust, and it works in a purely unsupervised way (with the Hebb rule we imagined something else was setting the output values of the network).

Competitive networks are feed-forward networks where each output node "competes" with the others to represent a certain class or "cluster" of inputs in the input space. For example, in a world of cheese and flowers, a competitive network will automatically learn to represent cheeses and flowers with different nodes, without being told about the difference (see Fig. 9.3). It just develops a sense over time that cheese and flowers are different.

Something like this also happens in the brain. Neurons automatically come to represent features of an animal's sensory environment over time, even without a teacher. For example, neurons in the visual cortex learn to respond to specific types of edges, and neurons in auditory cortex learn to respond to specific frequencies of sound. They do this without targets, labels, or "desired outputs"; they learn to represent these features only based on the input dataset provided by nature.[11]

In machine learning, clustering algorithms do something similar to competitive networks, automatically detecting clusters of similar input vectors in an input space.[12] I like the example of a streaming movie service like Netflix. Netflix can look at a lot of movies, code the movies as vectors (which contain attributes about each movie and who tends to watch that movie). Then they can run an unsupervised clustering algorithm to automatically lump similar movies together. Then the people at Netflix can hand-label those clusters, with names like "Zombie Horror", "Quirky romance", and "Drama with a strong female lead."

9.5.1 Simple Competitive Networks

There are different approaches to competitive learning, both in machine learning and cognitive science. We begin with simple competitive networks, which can be easily created in Simbrain using *insert* > *network* > *competitive*, or by opening a workspace or script that begins with "competitive."

Recall from chapter 6 that the input nodes of a network define an input space, which corresponds to the set of all patterns (input vectors) that could occur on those nodes. Each pattern of activations over the input nodes of a network is a point in its input space. Often these sets of input vectors have some structure: in a smell network, for example, objects that produce similar odors will produce similar input vectors, which correspond to "clustered" points in the input space.

[10] Some background on PCA and how it works, and some of the other dimensionality reduction methods included with Simbrain, is here: `http://hisee.sourceforge.net/about.html`.

[11] Although it is worth noting that plausible feature detectors in the brain can also be developed using supervised learning, as with the deep network shown in figure 4.7 in chapter 4.

[12] This is how algorithms like k-means and dbscan work; see `http://scikit-learn.org/stable/modules/clustering.html`.

A competitive network will automatically learn to represent these clusters. The key idea that makes this possible is the fact that *the input space has the same number of dimensions as the fan-in weight space for each output node*. For example, in Fig. 9.3, the input space is 5-dimensional, and each of the three output nodes has a fan-in weight vector with 5 weights. Thus, each of the fan-in weight vectors can be represented in the same 5 dimensional input space. As the network learns, the fan-in weight vectors are updated in such a way that they become closer to specific clusters of inputs. In this way, different output nodes come to respond to different clusters of inputs. Thus, the output nodes are trained to be *cluster detectors*.

The basic idea is shown in Fig. 9.3 and Fig. 9.4. Each time the agent smells an object, a pattern of activity occurs over its 5 nodes, which is a point in a 5-dimensional input space. We can project these points to 2 dimensions, and then view them as in Figure 9.4. Each point corresponds to one smell.[13] At any time, only one output node in a competitive network is active. It is the winner of a winner-take-all competition (see chapter 1). The winning node relative to the current input is the one whose fan-in weight vector is closest to that input in the input space. The outputs are one-hot encoded, (see chapter 7) and the winning or "hot" output at a given time can be thought of as classifying inputs. The basic way a competitive network works, after it's been trained, is illustrated by the Simbrain 3-object-detector (discussed in section 1.2). The three nodes of that network respond to three different kinds of inputs in the input space.

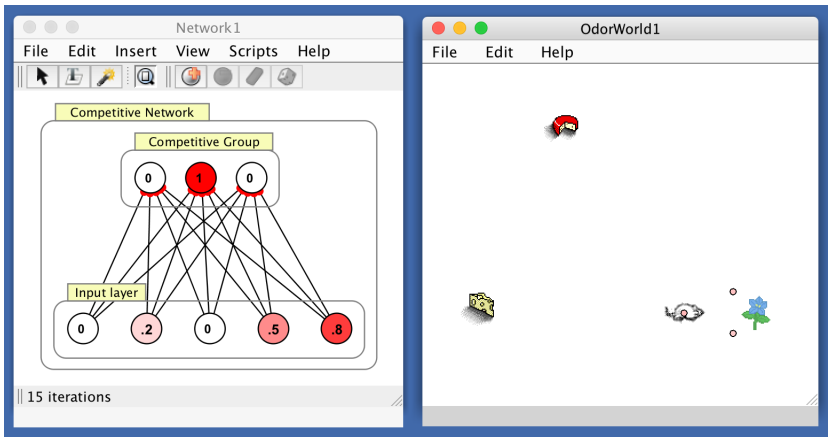


Figure 9.3: Competitive network with 3 output nodes, which can learn to detect up to 3 clusters in an 5-dimensional input space. Inputs correspond to smells of flowers. Once the network has been trained, the output nodes can be labelled. Which output node ends up classifying which type of smell will change from one run to another of this network.

The way a competitive network learns can be understood visually in terms of the input space of the network. In Fig. 9.4, each blue dot corresponds to an input, and each red dot corresponds to the fan-in weights to an output node. When an input (blue dot) is fed to the network, the nearest output node (red dot) is the node that will turn on. As the network learns, the red dots move to the centers of the clusters. In this way, the output nodes become cluster detectors.

Now we can say in more detail how competitive learning works. The competitive network is initialized with random weights. Thus, the red dots in Fig. 9.3 begin at random locations in the input space. When we start to apply the algorithm, input vectors (the blue points in the input space) are presented to the network in succession. At a given iteration, whichever fan-in weight vector is closest to that input vector "wins the competition" (hence the name "competitive learning"), and that weight vector is changed in such a way that it is moved closer to that input vector in the input space. Hence, that output node will be more likely to respond to that input in the future.

Here is the algorithm in more detail. For each row vector in the input dataset:

1. Use the row vector to set the input node activations.

[13] We could also use a table of inputs, in which case each row of the table, each sample, would be a point. This is how competitive learning is usually done in machine learning; in Simbrain, if you double click on a competitive network's interaction box, a training data tab appears that can be used to train the network in this way.

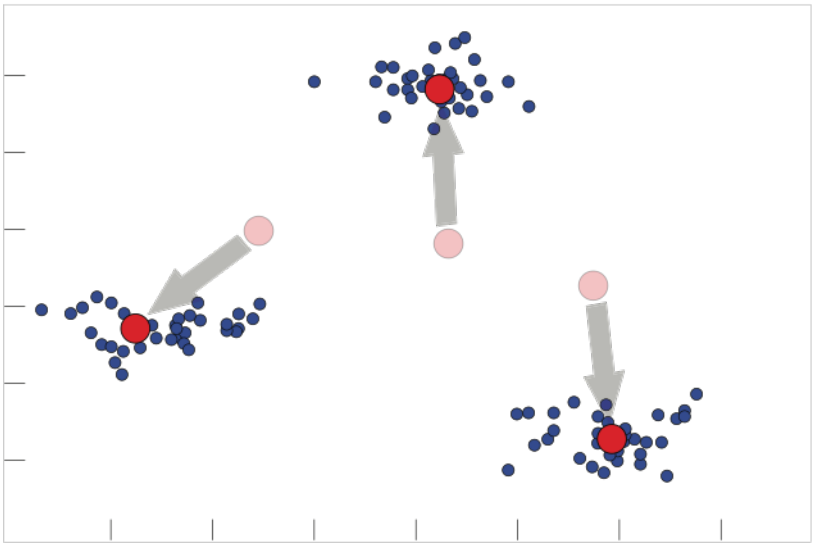


Figure 9.4: Geometrical illustration of how competitive networks learn. If we think of this as a representation of the smell inputs in Fig. 9.3, then each blue dot corresponds to one smell from one location in the virtual world, and the three clusters correspond to the three objects: Swiss cheese, Gouda, and the blue flower.

2. Determine the winning output node, which is the output whose fan-in weight vector is closest to the input vector in the input space.[14]

3. Assign the winner a value of 1 and the losers a value of 0.

4. Move the fan-in weight vector of the winning unit towards the current input in the input space. That is, update the weights attaching to the winning neuron, so that that neuron is more likely to fire in response to the same inputs in the future.

By repeated application of this algorithm, the fan-in weight vectors (the red dots) will incrementally move towards the input vectors closest to them. Over time, they will migrate towards the "centers" of the clusters in the input space. In this case, each red dot migrates towards the center of one of the three clusters of blue dots. After training, each of the three output neurons has come to represent one cluster of inputs. Each output neuron will now fire in response to any input in the cluster around it. This shows visually the sense in which the outputs have come to represent the statistics of the input space. Each output is tuned to respond to a given cluster of inputs.

The network will also generalize well: any new input near one of the clusters will activate the neuron for that cluster.

This process can be simulated in Simbrain using the workspace *competitiveNetSmells.zip*. When you open the workspace, you will see a competitive network with four outputs and a world with 6 objects. Each object is represented by a distributed pattern of activity on the input nodes. The 6 objects—3 cheeses and 3 flowers—correspond to 6 well-separated points in the input space. With training, 6 of the 9 output nodes should begin to respond to these inputs.

Now press play and just move the mouse from object to object. The mouse is simply smelling different things in its environment and not being told how to respond (so this is a nice case of unsupervised learning). As you move the mouse around, the network will start responding to the different inputs in different ways. Eventually, it should respond to each of the different inputs with a distinct output. The output nodes get "assigned" to these different inputs with learning. To make this clear, you can label the nodes appropriately and verify that the network has indeed learned to separately represent the different objects. So the mouse has learned something about the statistics of its environment without any training signal, assigning a distinct representation to each of four different distributed inputs.

[14]That is, the output node with the greatest weighted input.

9.5.2 Self Organizing Maps

A more sophisticated form of competitive learning is a **Self organizing map** (or SOM). The overall idea with this architecture is the same as with a simple competitive network: inputs are compared with fan-in weight vectors, a winner is chosen, and that winning neuron's fan-in weight vector is modified so that it "moves" closer to the input and thus comes to represent that input. Over time the output nodes come to represent specific regions of the input space [61].

The new idea with a SOM is to update the weights not just around the winning node, but also in a neighborhood around the winning node. A honeycomb pattern—a hexagonal array—is used on the output layer so that all nodes are equally distant from their nearest neighbors. A special algorithm is used whereby the size of this neighborhood starts is reduced over time. The result is that *nearby nodes come to represent similar inputs*. Thus, a bank of output nodes in a SOM network correspond to a kind of "map" of the input space. Output nodes that are near each other detect similar patterns in the input space [61].

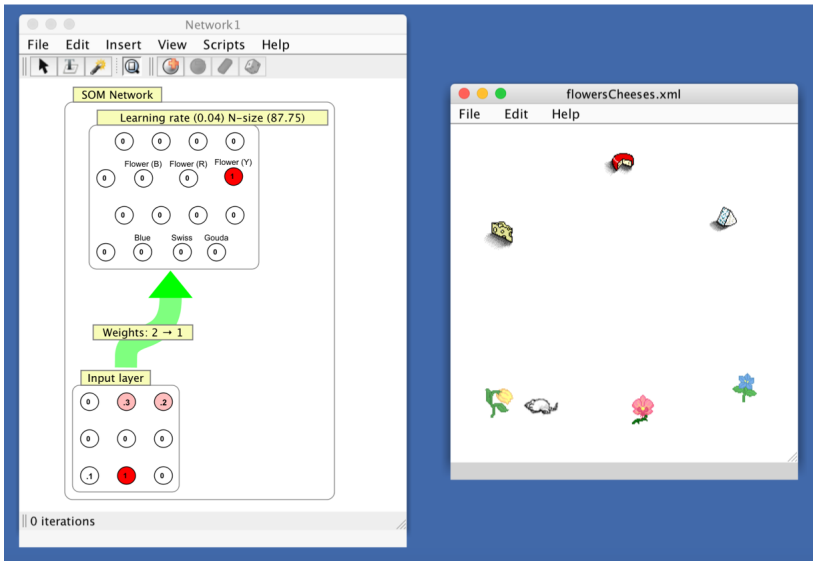


Figure 9.5: A self organizing map after it has been trained for several hundred iterations, with some of the categorizations it produces hand-labelled.

Fig. 9.5 shows an example of a self-organizing map in Simbrain, which is based on the workspace *somNetSmells.zip*. As with the simple competitive network, you can just run the network and drag the mouse around to the different objects. This simulates a sped-up process of human learning. As you run the simulation, notice that the neighborhood size (in pixels) and learning rate are being reduced. When the learning rate goes to 0, no more learning will occur, so be sure to expose the network to multiple inputs before that happens. If needed, you can right-click on the interaction box and select *Reset SOM Network*. After a while, the network will stabilize. At that point, you can move the agent around to the objects, see which nodes respond to them, and then label those nodes. I've done just that in Fig. 9.5. Notice that the three cheese and flower nodes are near each other in the hexagonal array of output nodes. Again, nearby nodes in the output layer correspond to nearby regions of the input space, which corresponds to objects that smell similar to each other.

SOMs are often represented only by their output nodes (that is, input nodes are omitted), since it is at the output nodes that the spatially organized maps take form. For example, in Fig. 9.6 we see a top view of a large sheet of millions of neurons in the brain that are thought to behave like the output layer of a SOM, and in Fig. 9.7 we see a top view of 150 nodes in the output nodes of a SOM.

Recall from chapter 4 that the brain is known to develop feature representations in a spatially organized way, via topographic maps. It is plausible to assume that some of these maps develop in an usupervised way over many years as a person interacts with their environment.[15] In visual cortex, for example, neighboring

[15]Though again we also saw that it can happen in a supervised way with deep networks.

neurons in retinotopic maps come to represent lines at similar angles (see Fig. 9.6). In somatosensory cortex, neighboring neurons in somotatopic maps represent nearby regions of the body. In fact, spatially organized feature maps have been identified in most sensory areas of the brain. Kohonen (1990), p. 1465, reviewing the literature at the time, says:

> Some of the maps, especially those in the primary sensory areas, are ordered according to some feature dimensions of the sensory signals; for instance, in the visual areas, there are line orientation and color maps, and in the auditory cortex there are the so-called tonotopic maps, which represent pitches of tones in terms of the cortical distance, or other auditory maps. One of the sensory maps is the somatotopic map, which contains a representation of the body, i.e., the skin surface. Adjacent to it is a motor map that is topographically almost identically organized. Its cells mediate voluntary control actions on muscles. Similar maps exist in other parts of the brain. Some maps represent quite abstract qualities of sensory and other experiences. For instance, in the word-processing areas, neural responses seem to be organized according to categories and semantic values of words. It thus seems as if the internal representations of information in the brain are generally organized spatially (p. 1465; numerous citations included in the original quote are omitted here) [61].

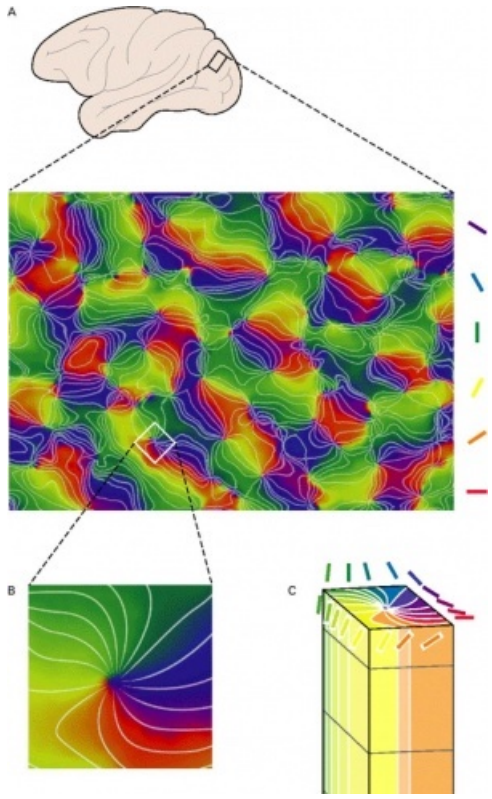


Figure 9.6: Topographically organized edge detectors in visual cortex. The main panel shows a top-down view on an area of cortex, with neurons colored according to what kind of edge they represent. To the right of the panel these edge orientations are shown. Notice that nearby neurons represent similar edges, that is, edges at similar angles. From `https://grey.colorado.edu/CompCogNeuro/index.php/CCNBook/Perception`.

In more abstract regions of the brain, nearby neurons may come to represent similar *concepts*. Fig. 9.7 shows a SOM trained to model relationships between words. A network with 150 output nodes was trained on semantic data. It was trained using "2000 presentations of word-context-pairs derived from 10,000 random sentences... Nouns, verbs, and adverbs are segregated into different domains" (p. 1476)[61]. Notice that nodes representing nouns, adjectives, and verbs occur in specific regions of the network, so that the network

represents grammatical categories. Also note that nearby nodes represent words with similar meanings, like "dog" and "horse" or "fast" and "slowly".[16]

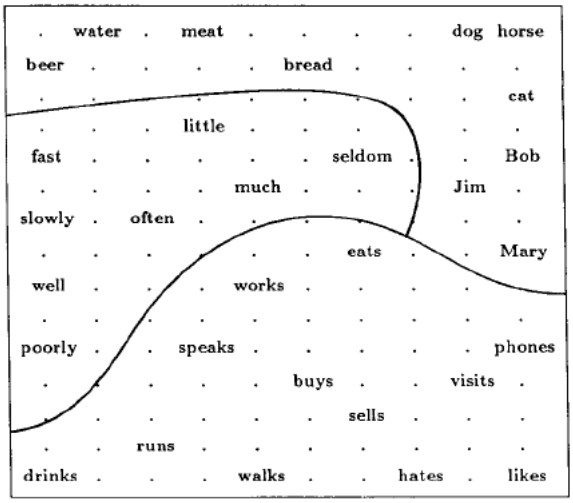


Figure 9.7: A self organizing map trained to represent semantic features of sentences. Each dot corresponds to an output node, and labels show what concepts these nodes have learned to represent. Notice that nouns, adjectives, and verbs are represented in specific parts of the network. From Kohonen (1990), p. 1476.

[16]Sergio Ponce de Leon refers me to this visually stunning video: `https://www.youtube.com/watch?v=k61nJkx5aDQ`. The point made is slightly different, and I can't speak to the merits of the study, but it is a striking way to see the general idea in action.

Chapter 10

Dynamical Systems Theory

Jeff Yoshimi, Scott Hotton

In chapters 5, 6, and 12, we have noted that when the "play" button ▷ is pressed in Simbrain a dynamical process is simulated; neurons start firing and changing, weights will sometimes change their size, etc. A **dynamical system** is a rule that says how a system changes its state in time. Neural networks are dynamical systems, which say how patterns of node activations, weight strengths, and other quantities change in time. When you press play in Simbrain, you run a dynamical system. Simbrain has special features, like the projection plot, which support dynamical systems analysis by allowing you to visualize network dynamics as they unfold in real-time.

Dynamical systems theory provides a formal, mathematical way to both analyze and visualize processes in neural networks (especially recurrent networks). Think of it this way: it's one thing to run a neural network in Simbrain and see a bunch of colors changing, or to look at a set of equations describing a neural network. But in these cases it's hard to say much about what exactly is happening in the network. However, when we use dynamical systems theory to describe and visualize that same neural network, suddenly we can see things that were previously invisible. We might find that no matter what state we initialize a network to, when we run it, it always ends up settling into just one of two possible states. Or we might find that it oscillates in one of three possible oscillatory patterns. These are things we can clearly see in a dynamical systems analysis, that would otherwise be hidden from us. As examples of this kind of visualization see Figs. 10.1, 10.2, 10.3, 10.4, and 10.6 below.

Dynamical systems theory is useful across all the domains of neural network theory. In connectionist models, memories can be thought of as stable states or attractors in a recurrent network (that is, states which the system tends to go to over time), which can be visualized as points in an activation space. Pattern completion—e.g. seeing part of a picture and then imagining the missing part—can be understood as an initial state of a system settling in to an attractor (see chapter 11). Learning in general can be understood as a dynamical process on the weight space of a network. More generally, connectionist theorists have thought of cognition as unfolding in a high-dimensional activation space, and learning as unfolding in an even higher-dimensional weight space. In computational neuroscience low level models of individual neurons are dynamical systems models, which describe how levels of calcium, sodium, spike rate adaptation, and other more abstract quantities change in time (see chapter 18). In machine learning, recurrent networks are trained to reproduce dynamical sequences of data, and can generalize from existing data to new data: this kind of network can be trained to generate paintings in the style of a particular painter, or speech in the style of a particular speaker (see chapter 16). There is also a body of theoretical work showing that neural networks can approximate any continuous dynamical system with arbitrary precision [56]. This shows that the human brain has a great deal of flexibility in the kinds of behaviors and processes it could in principle produce.

10.1 Dynamical Systems Theory

In this section, we introduce the basic concepts of dynamical systems theory and show how they can be used to study neural networks.

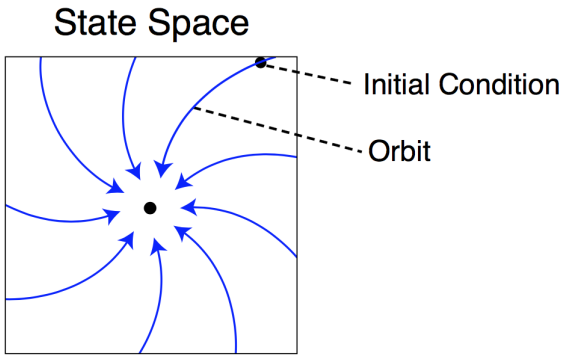


Figure 10.1: Some basic components of a dynamical system. The square region is the *state space* for a 2-dimensional system. Each point in that region is a *state*. Each state can be treated as an *initial condition*. When the system is run, an *orbit* unfolds from the initial condition. A picture like this that shows selected orbits in the state space is a *phase portrait*. The phase portrait shows in a concise, visually intuitive way what the dynamics of a system are. In this case we have a system with a single attracting fixed point. All orbits lead to that same state. Recurrent neural networks often display attractor dynamics.

We begin with the concept of a **state**. The word "state" is a general term to describe the condition of a system. For instance a bar of iron can be in a magnetic state or nonmagnetic state. The water in a jar can be in a frozen state but after being heated the water can change to a liquid state. A molecule can be in its ground state until it absorbs light, whereupon it enters an excited state. People can be in various emotional states. Oftentimes, states vary in a continuous way: the temperature of a pot of water, the sound level of a plucked guitar string, and the firing rate of a neuron all move up and down as internal and external conditions change. Note that the same object can have lots of states, depending on what we are interested in. A human has a temperature, a height, and a weight, and all of these are changing. Any of them can be the focus of a dynamical systems analysis.

Mathematically, the state of a system is represented by values for a collection of state variables. Each **state variable** describes a numerical value associated with a system at a time. If we have variables describing the temperature and pressure of a pot of water, then the state of that system at a given time is the value of those variables at that time. If we have three variables describing height, weight, and temperature of a person, then a state of that person at a time is the value of those three variables at that time. If we have a neural network with 1000 neurons, then we have 1000 state variables, one for each neuron in the network. But again, it's up to us what we consider a state of the network to be. We might just focus on a few of those neurons. Or we might shift attention from nodes to the weights. We could consider the full matrix of $1,000,000$ weights in that network, which correspond to a million state variables. Or we could consider the combined set of activations and weight strengths, which would involve $1,001,000$ state variables.

A **state** of a system is a specification of values for all of the state variables which describe that system. If we model water using temperature (Farenheit) and volume (Liters) as our state variables, a state for the pot of water might be $(89.8, 2)$. If we model a person using height (inches), weight (pounds), and blood sugar (mg/dl) as our state variables, a state for the person might be $(60, 150, 75)$. A state for the nodes or the weights of the network is a large vector or matrix that is too long to write out here. Dynamics then describe how these states–e.g. activation vectors, weight matrices, or others collections of state variables–change in time.

The state of the network in Fig. 10.2 is $(-.8, .8)$, which are the values of two state variables a_1 and a_2, corresponding to the activations of the two nodes. However, recall that what we take to be a "state" of a system is up to us. So instead of looking at node activations, we could have looked at weight strengths.

Then the state variables are $w_{1,1}, w_{1,2}$ and the current state is $(-1, -1)$.

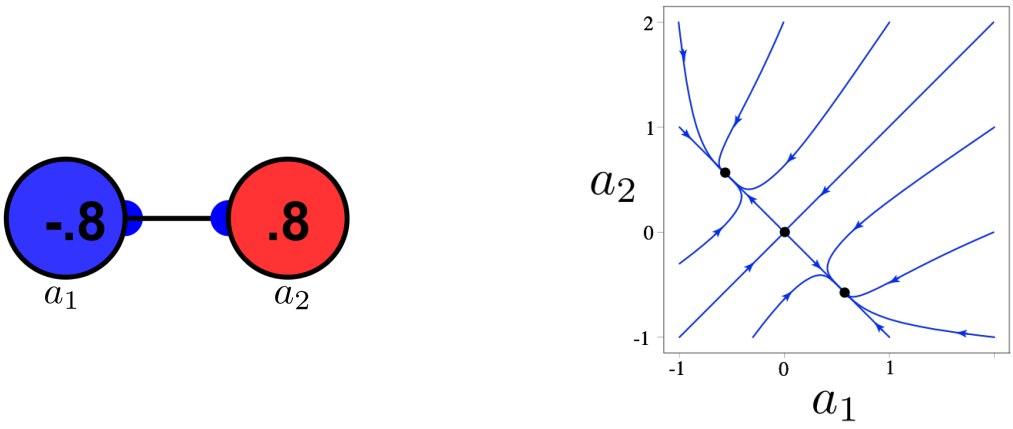


Figure 10.2: A 2 node recurrent network (left) and its phase portrait (right). The phase portrait has two fixed point attractors at $(-.8, .8)$ and $(.8, -.8)$, each with its own basin of attraction. There is a fixed point at the origin $(0, 0)$ which is attracting in one direction but repelling in the other (a "saddle-node"). This is a Hopfield network that was trained using a variant of the Hebb rule. The network stores two memories, corresponding to the two attractors.

Having specified what a state of a system is we can consider its **state space**, which is the set of *all possible* states of the system. The state space will, in the examples we consider, generally be a **vector space** (see chapter 6). For a neural network's activations, this means all possible activation vectors for that network, all possible patterns of activity that could occur over all of its nodes. So here, the state space of a network is its **activation space**. If we focus on a neural network's weights, the state space of a network is its **weight space**. A network with n-nodes has an n-dimensional activation space and a weight space that can be up to n^2-dimensional (there are n^2 possible weights in a network of n nodes; to see this recall that each weight can be represented as an entry in a matrix with n rows and n columns).

The dynamics of a network unfold in its state space. In a neural network, this is often a high dimensional space, e.g. the 20-dimensional space of a network with 20 neurons. Recall from chapter 6 that we can use dimensionality reduction techniques—like the projection plot in Simbrain—to visualize these dynamics in 2 or 3 dimensions.

An **initial condition** (or initial state) of a dynamical system is just the state the system begins in. For the case of a neural network's activation space, this is an initial specification of values for its nodes. Theoretically any point in the state space can be taken as an initial condition although sometimes it may be difficult or impossible to actually start a physical system in some particular state (e.g. setting the position and velocity of the earth or setting a person's age). In Figs. 10.1 and 10.2, any of the points shown could be taken as initial conditions. Each point corresponds to one pattern of activation over the nodes of the corresponding network. Often we just randomly choose initial conditions. In Simbrain, we do this for activation states by selecting all the nodes of a network and then pressing the randomize button. If we repeatedly press the randomize button, we end up putting the network in a whole bunch of different initial conditions, which is a useful way to explore the different ways the network can behave.

We are now in a position to give a definition of a dynamical system:

> **Dynamical system**: A rule that associates initial states of a system with (usually) future states of the system.

(We say "usually" because the system can also associate initial states with themselves at the present time, and can sometimes also associate initial states with past states; these are called "invertible" systems). A dynamical system can be thought of as a recipe for saying, given any initial point in state space (any initial condition), what states will *follow in time* for that system. If we know a system begins in state $(1, -1)$, the dynamical system will tell us exactly what states it will be in 4, 5, and 6 seconds (or iterations) from now. And we can do this no matter what initial condition we place our system in. Thus, dynamical systems are

deterministic: in theory they allow us to completely predict the future of a system based on its present state. A long-standing question in philosophy is whether the universe is deterministic, and thus describable by a dynamical system.[1]

Depending on whether "time" is taken to be continuous or discrete, we have a continuous time or discrete time dynamical system. In nature, time is thought to be continuous, and thus continuous time systems, described with the tools of calculus (for example, differential equations) are often used to model natural systems. In computer simulations, such as Simbrain, time is treated as something that occurs in discrete steps or iterations, and so dynamical systems as studied in computer simulations are discrete time systems.

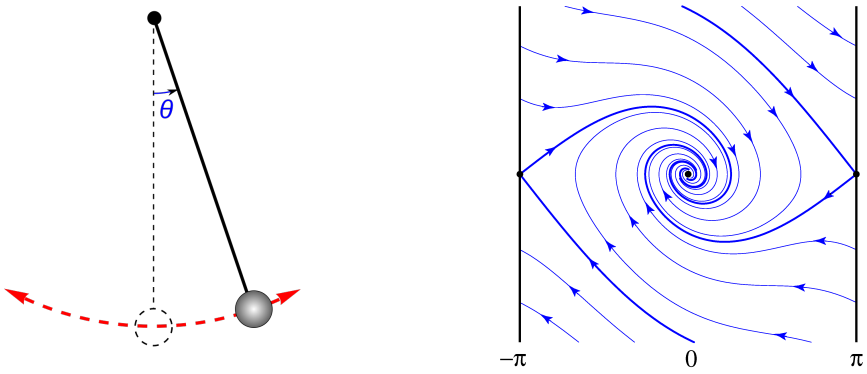


Figure 10.3: Penduluum (Left) and its state space with a phase portrait (Right). The vertical dimension of the state space corresponds to angular speed; the horizontal dimension corresponds to angular displacement away from hanging straight down.

The mathematical category of dynamical systems is abstract, and encompasses many more specific ideas in mathematics. For example, the solutions to differential equations are often dynamical systems, which allow us to predict future states of a system from its current state. An iterated function is another kind of dynamical system (think of entering 2×2 on a calculator and repeatedly clicking the = button).

A pendulum is a classical example of a dynamical system. Pendulums have been studied extensively ever since Leonardo da Vinci designed fairly accurate clocks based on them. The state of a pendulum is given by two variables, one variable θ for the angular displacement of the pendulum from verticality and another variable $\dot{\theta}$ for the angular speed.[2] If we start a pendulum with some chosen values for these two variables we can, at least in principle, say exactly how it will move forever in the future. Fig. 10.3 shows a pendulum on the left and its state space on the right. The horizontal axis shows the angular displacement, θ, and the vertical axis shows the rate of change of the angular velocity, $\dot{\theta}$. We can put the bob of the pendulum in any initial position and give it a push with any initial speed and the future of the pendulum will be determined. This behavior can be predicted by choosing the corresponding point in the phase portrait and following the orbit though that point. Eventually, because of friction, the orbit will close in on the point $(0, 0)$ which corresponds to the pendulum hanging straight down without moving.[3]

Neural networks are also like this. If we start a neural network off in some particular state–for example, if we specify values for all its nodes–then based on its update rules and the way it is wired together we can say just how it will behave for all future time. Thus, "running" a neural network, in Simbrain by pressing

[1] This is sometimes described in terms of "Laplace's Demon." As Laplace himself said: "We ought to regard the present state of the universe as the effect of its antecedent state and as the cause of the state that is to follow. An intelligence knowing all the forces acting in nature at a given instant, as well as the momentary positions of all things in the universe, would be able to comprehend in one single formula the motions of the largest bodies as well as the lightest atoms in the world, provided that its intellect were sufficiently powerful to subject all data to analysis; to it nothing would be uncertain, the future as well as the past would be present to its eyes. The perfection that the human mind has been able to give to astronomy affords but a feeble outline of such an intelligence. (Laplace 1820)". From Carl Hoefer's encyclopedia article: `https://plato.stanford.edu/entries/determinism-causal/`.

[2] Note the dot over $\dot{\theta}$, which indicates the first derivative of θ, i.e. rate of change of angular displacement, which is angular speed.

[3] The state space is actually an infinitely long cylinder. Imagine wrapping the left and right edges of the state space in Fig. 10.3 around and gluing them together.

the step or the play button, corresponds to applying the dynamical rule that describes it. A neural network which is predictable in this way is a dynamical system.

Can you think of ways to make a neural network *not* be a dynamical system? One way is to add some random noise to a node. When you do that, it is no longer possible to predict with complete accuracy what future states will follow from the present state.

Since dynamical systems are deterministic, they can be used to predict exactly what future states will follow from any initial condition. However, a **chaotic dynamical system** is a dynamical system whose future behaviors are difficult to predict. A chaotic system is still a dynamical system, so it's fully deterministic, but it's hard to predict how it will behave, especially moving farther in to the future. It's a kind of paradox: a chaotic system is fully determined by a set of equations, but it behaves in an unpredictable way.[4] To see chaos in Simbrain you can create a logistic activity generator. By default this rule produces chaotic dynamics (open its help page for more information). Many natural processes, like the weather, are thought to be chaotic. An example of chaotic behavior is shown in Fig. 10.4. Notice that given an initial condition it would be hard to predict where precisely that system would be at future times, even if we do know it would stay in that region of state space.

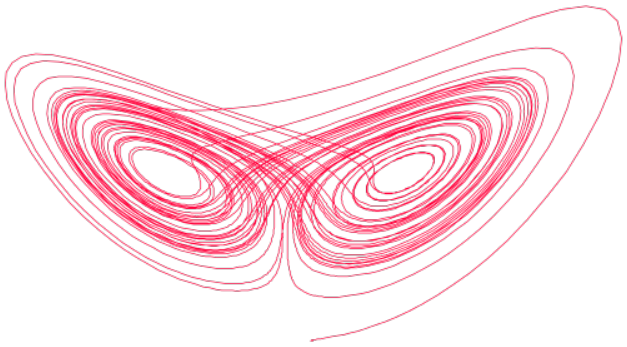


Figure 10.4: An orbit from a famous chaotic system called "The Lorenz Attractor." Nearby initial conditions can diverge arbitrarily far apart (within the attractor) over time.

An **orbit** of a dynamical system is the set of states that are visited by the system relative to a particular initial condition (orbits are also called "trajectories"). The idea is that you begin in some initial condition (you start at some point in the state space), then run the dynamical system, and the result is a time-ordered collection of states, one for each iteration or moment in time. This time oriented subset of states is an orbit. Orbits are drawn with arrows to show the direction in which the system moves with time. In Fig. 10.1 and Fig. 10.2 most of the orbits are curves that tend towards specific points. Similarly for the pendulum's state space. In the chaotic system in Fig. 10.4, the orbits are tangled and hard to follow. In some cases an orbit is a single state: start in that state, and you will stay there forever (keep this in mind! It sounds weird to call one point an "orbit", but sometimes a point is an orbit!). In terms of neural networks, we put the neural network in some initial state, we run the network, and we watch its activations or weight strengths (or other parameters) change. The resulting set of points is an orbit for the neural network. Orbits can be viewed using the projection plot.

We can visualize a dynamical system by drawing several of its orbits in state space. This gives us a sense of what a system tends to do relative to different initial conditions. This is a **phase portrait**, a picture of a state space with some important orbits drawn in it. Since every point in the state space is part of some orbit, if we drew all of the orbits they would fill the whole state space. So we only draw some of the more important ones, a selection of the orbits that are the most revealing. Most of the figures in this chapter show phase portraits. Of course, since most neural networks have more than 3 nodes, we will typically have

[4]The formal definition of chaos is a difficult and unresolved topic. One way to define chaos is in terms of "sensitive dependence on initial conditions." In this view, a chaotic system is such that initial conditions that are extremely close to each other in the state space, can end up being very far apart given enough time. This is sommtimes called the *butterfly effect*. Since weather is a chaotic phenomenon a small change in one part of the world, like the flapping of a butterfly's wings, can have a huge effect further in time. So for example, if you sneeze now (vs. not sneezing) it could influence food prices in Brazil a few months later.

to visualize a phase portrait using **dimensionality reduction** techniques, using the projection plots in Simbrain.

10.2 Parameters and State Variables

Above we noted that it is somewhat arbitrary what we take the state variables of a system to be. In a neural network, it can be the nodes, or the weights, or both, or something else! There is a related subtlety. Sometimes we treat some of the state variables associated with a system as being fixed and unchanging. For example, in a neural network we often *freeze* the weights of the network to study how the activations change. This is biologically unrealistic, since in the brain synapses are changing all the time. But we can justify this approach by noting that synaptic efficacies change *much more slowly* than neural activations do, and so as a simplification we can treat these efficacies as fixed. A variable like this is a **parameter**, that is, a variable that is treated as fixed, while other state variables are allowed to vary. In a neural network, weights and biases are often treated as parameters. The concept of a parameter also occurs in machine learning contexts, where the parameters of a model are adjusted during training, but then usually fixed when the model is being used.

The parameters of a dynamical system are part of what determines its phase portrait. Once we see this, we can start to *vary* the parameters of a system, and observe corresponding changes in its phase portrait. When we do this, we will see the phase portrait change. Think of each parameter as a knob, and a set of parameters as a set of knobs. When the knobs are changed, the dynamical system changes, and this is visible as a change in its phase portrait. Usually the result of changing a parameter is a mild change in the phase portrait. But sometimes changing a parameter can lead to a drastic change. A new stable state can emerge in the state space, or a set of nodes that were stuck in one state can start oscillating.[5]

These sudden shifts in the behavior of a system when parameters are changed are called bifurcations. A **bifurcation** is a radical (more precisely, "topological") change in the phase portrait of a dynamical system that occurs when the parameters are changed passed certain critical values (these values are sometimes called "critical points"). An example of a bifurcation is shown in Fig. 10.5. In that figure, a single point (left) gives rise to a circle (right) when the parameters are changed past a critical value.

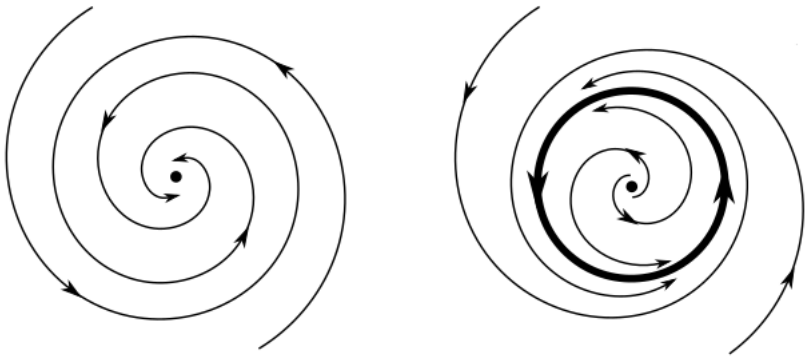


Figure 10.5: A bifurcation where a fixed point goes from being attracting to repelling, and in which an attracting periodic orbit appears. (This is called a "Hopf bifurcation").

10.3 Classification of orbits

A phase portrait is a complicated collection of orbits. How can we understand it? One way is to focus on a few prominent orbits in the phase portrait, and to extrapolate from these to get a sense of how the rest of the system behaves. Oftentimes a system has a few prominent orbits–e.g. certain kinds of stable states that

[5]This is similar to what happens in the graceful degradation lab: removing weights (which is like turning a weight knob to 0) usually doesn't make a big difference, but can sometimes lead to a massive change where the agent can no longer recognize something.

"pull in" nearby states–and the rest of the system can be understood relative to those prominent orbits.[6] For example, in Fig. 10.1 only 10 orbits are shown (one of which is a single point), but from these 10 orbits we can infer what the other orbits are like. The other orbits are "between" these 10. In fact, in this case we can get a pretty good sense of what the system will do just by focusing on the point in the center, and noting that it attracts all other states towards it.

In this section, we introduce some language for classifying orbits and using them to understand the overall behavior of a system. The basic categories we consider are shown in Fig. 10.6.

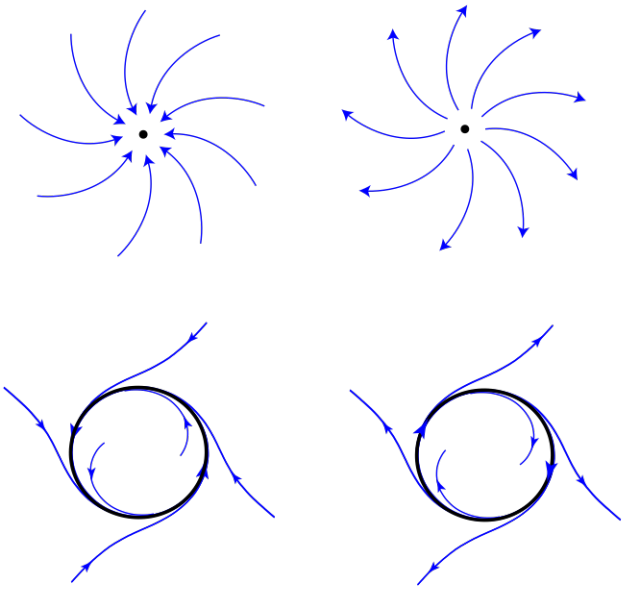


Figure 10.6: An attracting fixed point (upper left), repelling fixed point (upper right), attracting periodic orbit (lower left), and repelling periodic orbit (lower right). Attractors / repellers are shown in black; transient of orbits approaching or leaving attractors and repellers are shown in blue.

10.3.1 The Shapes of Orbits

One way to classify orbits is by their shape, or "topology."[7] Some prominent topologies for an orbit are a point, line and a loop. The analogues of these in a discrete time system are a point, a sequence of points, and a cycle of points.

The simplest shape for an orbit is a **fixed point** (also known as an equilibrium), a state that goes to itself under a dynamical system. See the top row of Fig. 10.6. This type of orbit is just a single point. When we start a dynamical system at a fixed point it remains there for all time. Once a system reaches a fixed point it stays in that state forever.

[6] In a more formal presentation, we would focus on *invariant sets* rather than orbits, which are subsets of a state space with the property that no orbit that enters it will ever leave. Such a set is "invariant" in that if a system begins somewhere in an invariant set, it will stay there for all time (Unless some external force pushes it out of the set, but then we no longer have a classical dynamical system.) Notice that a single orbit by itself is an invariant set. Other types of invariant sets contain many orbits.

[7] The topology of an orbit is its shape, in a special sense. Think of an orbit as a compressible / extendible string. This concept of shape allows arbitrary squashing and pulling of the orbits. Two orbits have the same topology if one can be squashed or squeezed in to the other shape without cutting the strings or gluing them together. In a discrete time system the topological properties of orbits work differently. Orbits are discrete chains and their topology is defined similarly to how the network diagrams in chapter 1 were defined.

Another shape for an orbit is a repeating loop or cycle. See the bottom row of Fig. 10.6.[8] This kind of orbit periodically visits the same points over and over again. This is a **periodic orbit**, a set of points that a dynamical system visits repeatedly in the same order. Periodic orbits repeatedly cycle back on themselves. This corresponds to an oscillation in a network, a repeating pattern of firings. For a continuous time dynamical system, a periodic orbit is loop-shaped (it has the topology of a circle), and its period is the amount of time takes to go around the loop.[9] For a discrete time system, a periodic orbit is a finite set of n states that the system cycles through (its period is n). This is also called an **n-cycle**. For example, a 2-cycle is a pair of states the system goes back and forth between. A 3-cycle is a set of 3 points that system visits in the same repeating sequence. Similarly for 4,5,100, and arbitrarily large n-cycles.

Figure 10.7: Two 5-cycles in a two dimensional state space. The points in the 5-cycles are indicated with open circles. The arrows on the dashed lines connecting the open circles indicate where each point in a 5-cycle goes under the dynamical system. (Left) A 5-cycle for a simple dynamical system that just rotates the points by 1/5 of a turn. (Right) A 5-cycle for a more complicated dynamical system.

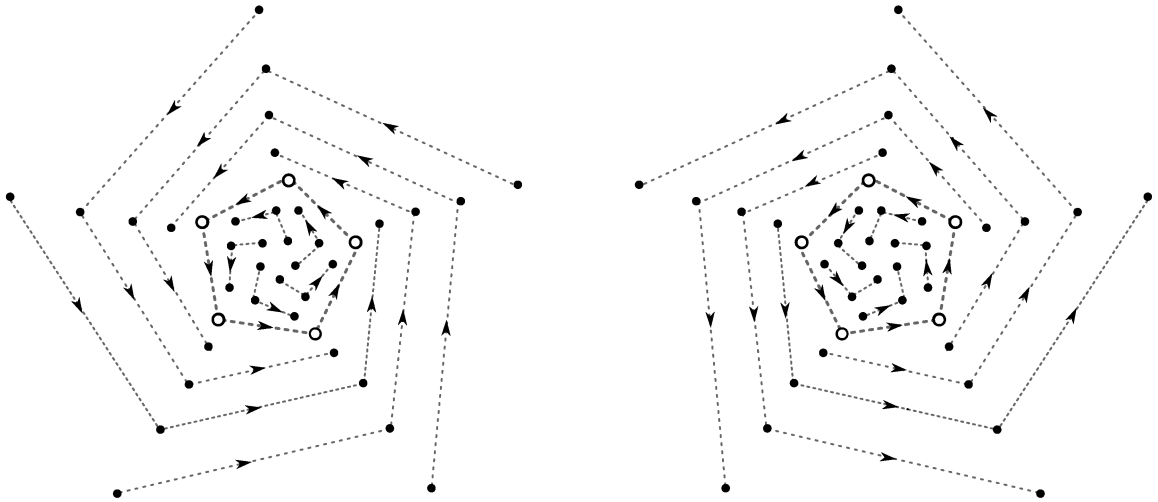


Figure 10.8: The 5-cycles are shown as in the left panel of Fig. 10.7. The dots indicate points that are not part of the 5-cycle. The arrows on the dashed lines connecting the dots indicate where the points go under the dynamical system. (Left) An attracting 5-cycle. Over time the points jump closer and closer to the 5-cycle without staying next any particular point in the 5-cycle. (Right) A repelling 5-cycle. Over time the points jump further and further from the 5-cycle.

[8]The Poincaré-Bendixson theorem tells us there must be at least one fixed point inside the periodic orbits shown on the bottom row of the figure. These fixed points have been omitted for pedagogical purposes.

[9]Do not confuse the period of a single periodic orbit with the number of periodic orbits in the state space. For example, a dynamical system can have one periodic orbit with period 2 and three other periodic orbits with period 5.

Joining orbits together produces an invariant set. States in an invariant set never leave it. For instance the disks inside the periodic orbits shown in the bottom row of Fig. 10.6 are invariant sets. They are the union of spiral shaped orbits (not all of which are shown) and a fixed point. Invariant sets can have many other shapes and extend into higher dimensional spaces. For example a two dimensional torus in a three dimensional state space can be an invariant set. Invariant sets can even be fractals like the one shown in Fig. 10.4.

10.3.2 Attractors and Repellers

Another way we can classify orbits is according to how states near them behave. Sometimes states near an orbit will tend to go toward the orbit. It's as though the orbit "pulls in" all nearby points. See the left side of Fig. 10.6. The fixed point and periodic orbit seem to draw other orbits to towards them. These are *attractors*. More formally, an **attractor** is an orbit such that all states sufficiently close to it will stay close to it. If you perturb a system slightly from an attracting state it will tend to go back to that state. Attracting fixe points are also called *stable states* or *stable equilibria*. These are states we are generally more likely to observe a system in. A chair or coin at rest, or a marble at the bottom of a bowl, are at attracting stable states. Move them a little and they will settle right back down.

In other cases states near an orbit will tend to go away from the orbit. It's as though the orbit "pushes away" all nearby points. See the right side of Fig. 10.6. Once a dynamical system starts running we are unlikely to see it near one of these orbits. More formally, a **repeller** is an orbit such that all states sufficiently close to it move away from it.[10] Perturb a system from a repelling state and it will begin to move away from that state. These are also known as "unstable" states. A chair or coin resting right on its edge, or a marble balanced precisely at the top of an upside-down bowl, is in a repelling state: move these systems a tiny bit away from their current state and they will go away from that state. Because of this, it is hard to observe systems in repelling states.

When a system has multiple attractors, we can associate each attractor with a **basin of attraction**, which is the set of all states that tend towards a given attractor. This is useful because we can then partition a state space in to basins, one for each attractor. If you start a system off anywhere in the basin of attraction for an attractor eventually it must end up on or very close to that attractor. We can think of basins of attraction using a "hill-and-valley" metaphor (see figure 10.9). We can think of the state space of a neural network like a wavy surface and we can imagine there is a marble rolling on the surface whose position marks the current state of the system. The marble rolls down along a path (orbit) in the valley that it starts out in. The attractor in this metaphor is the point at the very bottom of the valley that the marble comes to rest at and the whole valley is its basin of attraction. The state space in Fig. 10.2 has two attractors with two basins of attraction.

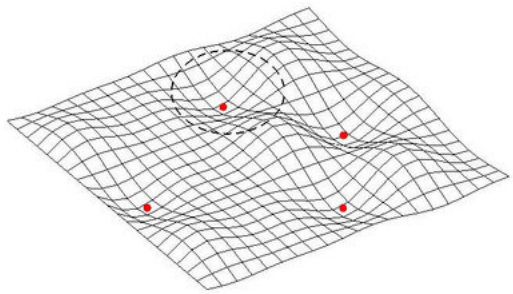


Figure 10.9: Attractors and basins of attraction pictured using a hill and valley metaphor.

As we discuss further in chapter 11, attractors of recurrent networks can be thought of as memories in some connectionist models. If a network of this kind has 20 attractors, we can think of it as having 20

[10]Technically these are definitions for "attracting sets" and "repelling sets". In order for an attracting set to be an attractor or a repelling set to be a repeller the set must satisfy a further property known as "topological transitivity" which is a concept we will not go into here. Fixed points and periodic orbits do have this property so this definition suffices for them.

memories. Recalling a memory corresponds to setting the system in an initial state and letting it settle in to the attractor of whatever basin of attraction it began in. Learning new memories corresponds to modifying the weights of the network to acquire new attractors. A related example is perceptual completion. Perceptual completion is when you see part of a picture and "fill in" the rest in your imagination. We can think of the state of seeing only part of the picture as being an initial condition in a neural network. And we can think of the process of "filling in" the rest of the picture as the network's dynamical processing that leads to the attractor which corresponds to the memory of the whole image.

There are orbits that are neither repellers nor attractors. For example, the central point in Fig. 10.2 is attracting in one direction, and repelling in another.

10.3.3 Combining these classifications

Combining the results of the last two subsections, we can classify the orbits of a dynamical system in terms of the topology of its orbits *together* with whether they are attracting or repelling. This classification is evident in Fig. 10.6. Here is the same classification in table form. In the table below, the columns correspond to different topologies or shapes that an orbit can have. The rows corresponds to the behavior of states nearby the orbit.

	Fixed Point	**Periodic orbit**
Attractor	attracting fixed point	attracting periodic orbit (e.g. an attracting n-cycle)
Repeller	repelling fixed point	repelling periodic orbit (e.g. a repelling n-cycle)

Recurrent networks displaying all four types of orbit can be created in Simbrain using the projection plot. However, as noted above, it's easier to find attractors than repellers. Fig. 10.10 shows a system with two attracting fixed points on the left, and a system with an attracting periodic orbit on the right. Both are projections of orbits of a 25 dimensional system to 2 dimensions. In both cases the image was generated by repeatedly randomizing network nodes (setting them in an initial condition), running the network, and observing the resulting orbits. About 17 initial conditions were used for the network on the left, and 5 on the right. It is possible that the systems contain more attractors than are shown, but that they were not found after that many attempts. Notice that attractors were found, but not repellers. To find a repeller you have to get lucky and land right on top of it, or find it using mathematical means. This emphasizes the mix of exploratory and more *a priori* or analytic modes of research involved in dynamical systems theory.[11]

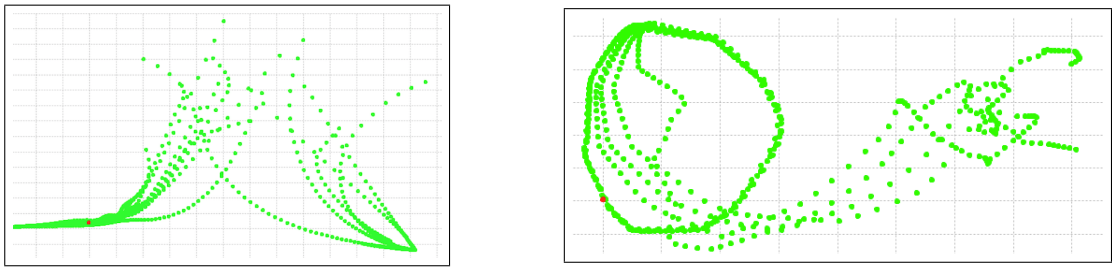


Figure 10.10: Phase portraits generated using Simbrain. A system with two attracting fixed points (Left) and a system with one attracting periodic orbit (Right). Both are projections from a 25 dimensional activation space to 2 dimensions. The red point is the current point in each simulation.

[11] Notice that the orbits are not smooth. This emphasizes that a computer produces a discrete approximation of continuous time processes.

Chapter 11

Unsupervised Learning in Recurrent Networks

Jeff Yoshimi

11.1 Introduction

In this chapter, we consider recurrent networks trained using the Hebb rule to complete patterns. This complements the discussion of unsupervised learning in feed-forward networks in chapter 9 with an analysis of unsupervised learning in recurrent networks. We will see that the tools of dynamical systems theory (chapter 10) are quite useful in this context. In chapter 16 we discuss more complex recurrent networks and their applications to psychology, neuroscience, and engineering.

11.2 Hebbian Pattern Association for Recurrent Networks

We now consider a *recurrent* network trained using the Hebb rule. When the Hebb rule is used in a recurrent network, connections between active nodes will be strengthened, and the result is a kind of trace of the pattern. If the weights are then clamped to prevent further learning (recall how sensitive Hebbian learning is), a fragment of the pattern, a "cue", can then be used to recreate the entire pattern. An example that makes the idea clear is in Fig. 11.1, where the residue of past training on an "L"-shaped pattern (a "memory trace") is evident. When the network is updated, it is obvious that the activation will fill in the L-shape.

Because they associate parts of a pattern with a whole pattern, these networks are sometimes thought of as auto-associators, or "self"-associators.

Just as it is fairly easy to train feed-forward pattern associators using the Hebb rule (see chapter 9), it is fairly easy to train recurrent pattern associators in this way. For practice, try using the self-contained Simbrain tutorials *autoAssociatorPart1.zip* and *autoAssociatorPart2.zip*, created by Alex Holcombe.[1]

These ideas can be used to understand conceptually how visual image completion might work. When we see a fragment of a familiar image or visual scene, we often "fill in the rest". This can be understood in terms of trained associations in a recurrent network where each node corresponds to a pixel. When a pattern is learned, all the correlations between pixels are encoded by strengthening corresponding connections using the Hebb rule. An example of a recurrent auto-associator for visual memories is shown in Fig. 11.2. It's the same idea as with the simple "L" pattern in Fig. 11.1, but with a much larger network, containing half a million rather a few hundred weights. In each case, learning a memory amounts to strengthening co-active nodes in a grid of nodes. In both cases, a partial cue triggers the completion of a stored pattern. The formation and recall of visual memories can be understood in these terms.

[1] In Simbrain press *open workspaces* and navigate to the *courseMaterials* folder.

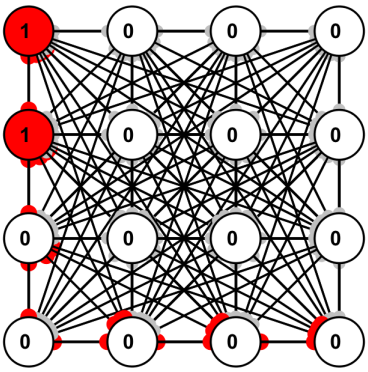


Figure 11.1: Cued recall of an "L" shaped pattern. Notice that the "L" pattern is visible in the red weights, which indicate how the pattern will be completed. In the past, those neurons fired together, so they were wired together by the Hebb rule.

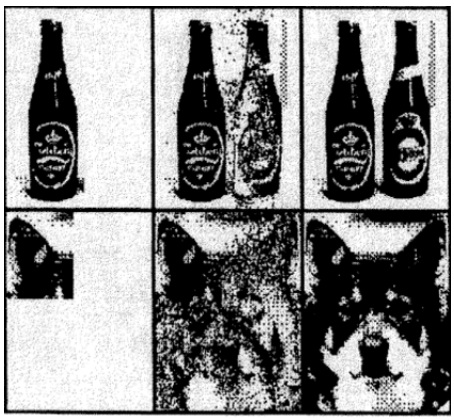


Figure 11.2: Pattern completion in a recurrent network with $130 \cdot 180 = 23,400$ nodes. The left-most image in each row shows the initial cue. The middle image shows the network part-way through the pattern completion process. The right image shows the final image. From Hertz et al. 1991. The network is a Hopfield network, which uses a variant on the Hebb rule.

Figure 11.3 shows how pattern completion in recurrent auto-associators can be understood in terms of dynamical systems theory (also see chapter 10). Each new pattern the network is trained on becomes a fixed point attractor in its activation space. The beer and dog images on the right of Fig. 11.2 are fixed point attractors in the 23,400-dimensional activation space of that network. The "L" visible as a trace in Fig. 11.1 is a fixed point attractor of the 16-dimensional activation space of that network. A cue corresponds to an initial condition. The single beer bottle and dog's ear in Fig. 11.2 are initial conditions, as is the upper part of the "L" in Fig. 11.1. Recall corresponds to following an orbit through the activation space. The final memory is the attractor corresponding to whatever basin of attraction the initial condition was in. Thus, on this model, learning a new pattern via Hebb's rule corresponds to adding a new attractor to the network's state space.[2]

11.3 Some features of recurrent auto-associators

First, they perform fairly well even if some synapses are removed (**graceful degradation**) or if you add noise to the inputs.

[2]Thus, learning in these cases also counts as a bifurcation, since this corresponds to a change of parameters (weights) that produces a change in the topological structure of the orbits in the state space.

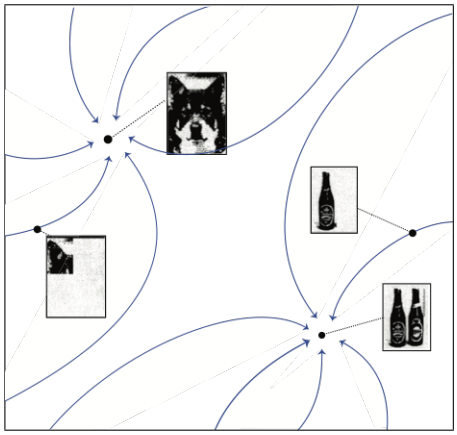


Figure 11.3: Schematic diagram of the attractors and basins of attraction for the images shown in Fig. 11.2. Fragments of images correspond to initial conditions, that evolve under the network dynamics to completed images, which correspond to attracting fixed points.

Second, the memories you train the recurrent network on can interfere with one another. Small networks trained using **binary vector** inputs (patterns of 0's and 1's, as in see Figure 11.4) cannot easily learn more than one memory. If a second pattern overlaps the first (in which case the two input vectors are not **orthogonal**), then during recall any partial version of either pattern will produce the *conjunction* of the two patterns. This is sometimes called **cross talk**. To address the problem, we can use **bipolar vector** patterns (see Figures 11.4 and 11.5 to see how binary and bipolar patterns compare), in which the "off" neurons are set to -1. The reason this helps is that the network is now learning not only to recreate a pattern of correlated activations, but also to *inhibit* activations inconsistent with the current pattern. The "on" nodes are connected to the "off" nodes with negative weights. Thus, during recall, activating one pattern will inhibit other patterns. This in turn makes it possible to store overlapping patterns. One pattern simultaneously represses the other.

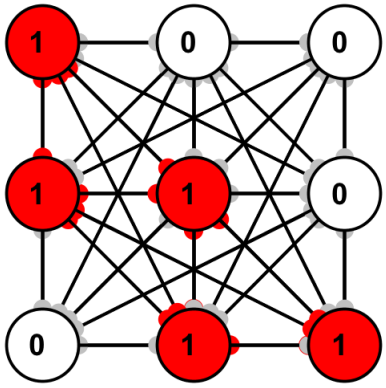


Figure 11.4: An auto associative network trained on a single binary pattern using Hebb's rule. Notice that only weights between co-active nodes have been strengthened. Since the other nodes have activations of 0, weights to and from them are not changed.

Third, when you train a recurrent network, you will sometimes notice new patterns, which are byproducts of other patterns: these are sometimes called *spurious memories*. In the case of a network trained on a single bipolar pattern, these are easy to predict: they correspond to the compliment of the trained pattern (that is, the pattern formed by -1's rather than 1's).[3]

[3]One way to get a feel for this is to use a projection to see all the stored and spurious patterns, which are attractors of the

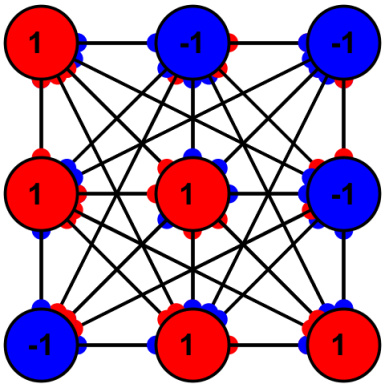


Figure 11.5: Bipolar version of pattern from Fig. 11.4, after training on a bipolar version of the same pattern. Notice how some of the weights have turned blue. Thus, when the pattern is recreated incompatible patterns will be inhibited. This version of the network can learn several patterns.

Finally, these networks tend to oscillate. If you try multiple initial conditions in a recurrent network trained using the Hebb rule, it may sometimes oscillate through an n-cycle rather than settling in to a fixed point attractor. In our study of dynamical systems in chapter 10 we saw that oscillations—i.e. attracting periodic orbits—often appear in recurrent networks. Using the Hebb rule can store a fixed point attractor memory, but un-desired n-cycles can come along for the ride. Hopfield networks, discussed next, avoid this problem.

11.4 Hopfield Networks

A special type of recurrent, auto associative, Hebbian network is a Hopfield network.[4] Hopfield networks have no self-connections and their weights are always symmetrical ($w_{i,j} = w_{j,i}$ for every weight in the network). They are also updated in a special way that gets rid of the unwanted oscillations.

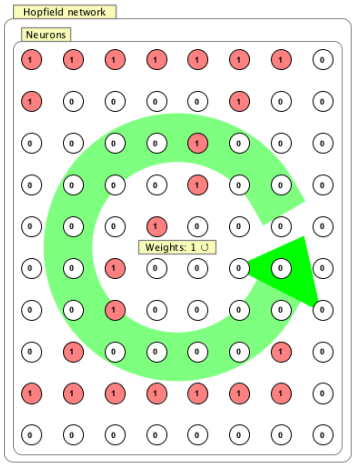


Figure 11.6: Hopfield network trained on the pattern for a "Z". Though a binary pattern is displayed behind the scenes this network uses bipolar patterns, with -1's where 0s are.

network. Often these will appear to be symmetrically positioned vertices of a hypercube.

[4]Hopfield networks are important historically. When John Hopfield, a physicist, introduced them in the 1970s it brought the existing engineering literature on neural networks and the formalisms in physics into greater contact with one another. Hopfield also pioneered the use of dynamical systems theory in neural networks [55].

A Hopfield network with 80 nodes is shown in Fig. 11.6 after it has retrieved one of the 4 memories it was trained on, a memory corresponding to the letter "Z". The theoretical memory capacity of Hopfield networks has been estimated to be 15% of its number of nodes.[5] Thus, a network with 20 nodes should be able to store about $3 = .15 \cdot 20$ memories. Hopfield networks have the advantage of not producing oscillations. However, they do produce spurious memories. To get a feel for how Hopfield networks work, you are encouraged to try the Simbrain simulation *hopfieldNet.zip* or to make and train a Hopfield network from scratch.

[5] A discussion of storage capacity for associative memories is in Fausett [28], p. 140 and section 3.3.4. Also see Hopfield's original discussion at [55], p. 2556.

Chapter 12

Supervised Learning

Jeff Yoshimi

With **supervised learning**, weights are changed using an explicit representation of how we want the network to behave. Input vectors in an **input dataset** are associated with targets or labels in **target dataset** (see section 7.4).[1] We say, "if you see this pattern, produce this other pattern." This is sometimes called "learning with a teacher." We saw in chapter 9 that Hebbian pattern associators can be trained by exposure to input / output pairs. Unfortunately, that method is unstable. The weights tend to explode to extreme values. So we need something more adaptive and robust: a way to get the weights to go up and down and settle in on just the right values, so that our network gets as close as possible to doing what we want. It's a bit like Goldilocks, trying to find the breakfast whose temperature is not too hot, not too cold, but just right.[2] Supervised learning algorithms provide a way to achieve this kind of zeroing in on just the right solution to a problem.

In this chapter we focus on general features of supervised learning in feed-forward networks, developing a toolkit of techniques and visualization methods. We will need to think clearly about labelled datasets, distinguish classification from regression tasks, learn how to visualize these two kinds of task, and discuss how to compute a metric of how well our network is doing at a given time ("error"). Finally, we will need to think about error *reduction* in a visual way, as downward motion on an error surface using the method of "gradient descent", which is basically the dynamical systems idea from chapter 10 of finding attracting fixed points, but this time in weight space rather than activation space.

In chapter 13, we cover some of the main classes of algorithm in supervised learning for feed-forward networks (including deep networks), and discuss their implications for cognitive science. In chapter 16, we discuss how supervised learning methods can be used to train recurrent networks, and how these trained recurrent networks have illustrated ideas in cognitive science. In both cases, we will see that internal representations are often learned by these networks, which seem to be similar to those humans use in processing language, recognizing faces, and in other tasks.

12.1 Labeled datasets

With supervised learning, we tell the network what we want it to do. There is a teacher or trainer. Recall from chapter 7 that a **labeled dataset** for a supervised learning task consists of a pair of datasets: an input dataset and a target dataset. Both datasets contain the same number of rows.

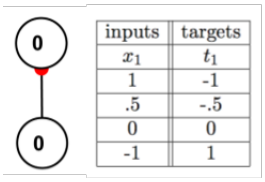
We will say that a labeled dataset is *compatible* with a feed-forward neural network if (1) the number of columns in the input dataset is the same as the number of input nodes in the network, and (2) the number of columns in the target dataset is the same as the number of output nodes in the network. The network can have any number of hidden layers and still be compatible with the dataset. A labeled dataset can be

[1] More review from chapter 7: recall that the input and target dataset together are a **labeled dataset** and that the part of the data we use to train a network is the **training subset** of the data. Also recall that the term 'label' is sometimes reserved just for classification tasks but is also (as here) sometimes used to refer to target data for regression tasks as well.

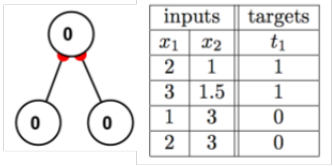
[2] This is sometimes called the 'Goldilocks principle'. See `https://en.wikipedia.org/wiki/Goldilocks_principle`.

used to train any network compatible with it. Examples of labeled datasets and compatible networks are shown in figure 12.1.

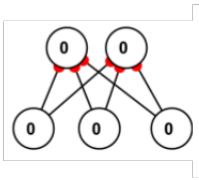
A labeled dataset can be thought of as a contract for a pattern association task: we'd like to train a network to come as close possible to implementing the input-output associations described by our labeled dataset. In the language of vector-valued functions, we are training a network to approximate a function that associates each vector in the input dataset with the corresponding target vector in the labeled dataset.



inputs	targets
x_1	t_1
1	-1
.5	-.5
0	0
-1	1



inputs		targets
x_1	x_2	t_1
2	1	1
3	1.5	1
1	3	0
2	3	0



inputs			targets	
x_1	x_2	x_3	t_1	t_2
1	0	0	.2	.4
0	1	0	.5	.7
1	0	1	.6	.7
0	1	1	.8	1

Figure 12.1: Some labeled datasets and the types of neural network topologies those training sets could be used on. Each dataset contains an input dataset and a target dataset with 4 rows. In each case, the input dataset has as many columns as its paired network has input nodes and the target dataset has as many columns as its paired network has output nodes. A classification task is shown in the middle (binary valued targets) and regression tasks are shown on the left and right (real-valued targets).

12.2 Supervised Learning: A First Intuitive Pass

In this chapter we focus on feed-forward networks, which can be thought of implementing vector valued functions. A labeled dataset is essentially a specification for a vector-valued function we'd like our compatible network to implement. Given an input vector $\mathbf{x}_r$ in a labeled dataset, we want the network to produce an output vector $\mathbf{y}_r$ as close as possible to the the corresponding target vector in that dataset.

The way we do this with supervised learning is by using algorithms that modify the **parameter**s $p_1, \ldots, p_n$ of the network, primarily the weight strengths and biases of the nodes. We start out with a network all of whose parameters p_i have been initialized to random values.[3] It's like making a network in Simbrain and pressing the `w` then `r` buttons, which selects all the weights and randomizes them. Recall from Chap. 10 that parameters are variables associated with a dynamical system that are fixed when the system is run but can be changed between runs. In a recurrent network, parameters determine the network's dynamics. In a feed-forward network, they determine the vector-valued function it approximates. Our goal is to set the parameters of the network so that it implements a vector-valued function that reproduces the associations in the labeled dataset as closely as possible.

Because we start with random values for the parameters of the network, it will generally not do well at first. Its outputs won't initially match target values. We then use a learning algorithm (several are covered in chapter 13) to incrementally update the parameters. If all goes well, the network should begin to behave in accordance with the specifications of the labeled dataset.

Below we refer to "errors" and "overall error", which are discussed in greater detail in Sect. 12.6. Roughly speaking individual errors says how far away the outputs produced by a network relative to an input are from the target values for that input, and overall error combines these errors together into a single number for all the nodes and all the rows of the dataset.

To train a network, we select a subset of a compatible labeled dataset, a **training subset** (section 7.5). This is our "training data": a set of input vectors and target vectors in a subset of a labeled dataset that we use to train our model. A schematic of the process that applies to most forms of supervised learning is as follows:

1. Randomize network's parameters $p_1, \ldots, p_n$.
2. For each row $\mathbf{x}_r$ of the input dataset:

[3]When I refer to"randomizing" a set of values, we mean setting them to values generated by a probability distribution, *e.g.* a uniform or a Gaussian distribution.

(a) Set the input-layer activations of the network to $\mathbf{x}_r$.

(b) Compute the network's output vector $\mathbf{y}_r$.

(c) Compute output errors by comparing the targets $\mathbf{t}_r$ with the outputs $\mathbf{y}_r$ (for each output, subtract the output from the target).

(d) Update $p_1, \ldots, p_n$ with the goal of reducing errors (so that outputs are closer to targets).

3. Repeat step 2 until overall error is sufficiently low.

We will see that this can be visualized in geometric way, as "gradient descent" to a low point on an "error hypersurface."

12.3 Classification and Regression

For feed-forward networks trained using supervised learning, an important distinction can be made between classification and regression tasks. At a first pass, classification tasks associate inputs with categories, and regression tasks associate inputs with numbers.[4]

In a **classification task**, the network sorts inputs into categories. The inputs might be the height and weight of a person, and the output will be a prediction about whether that person is a child or adult. Or, the network could take car data as input, and predict whether the car is a sports car or economy car. Notice that in both cases the outputs are categorical: they say which of n categories something falls in. This is one-of-k or one-hot encoding, discussed in chapter 7. One node for each category, and the one that's on corresponds to the category being classified. The three object detector (figure 1.7) is a classifier, which classifies smell inputs into one of three categories: Fish, Gouda, and Swiss. Figure 12.2 shows an example of a feed-forward network that classifies pixel patterns as one of 26 letters.

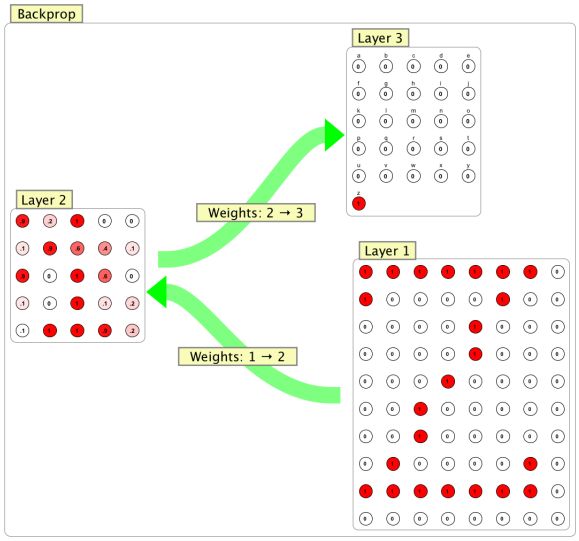
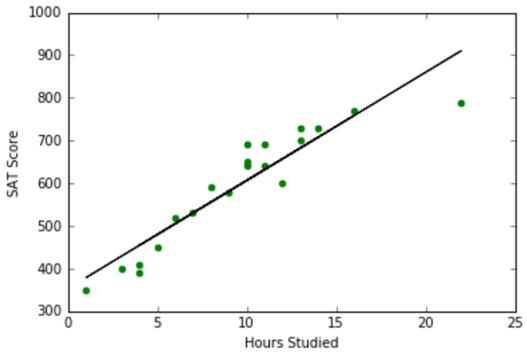


Figure 12.2: An example of classification. A feed-forward network trained via supervised learning to classify letter inputs as specific letters.

In a **regression task**, a network trained has real valued target values. This is simply the more general case of estimating a vector-valued function, where there are no constraints on how we interpret the outputs. If we train a network to predict the speed of a car (quarter-mile time) based on its fuel efficiency, engine size, and how many cylinders it has, we have a regression problem. We are not classifying cars into types, but predicting a numerical quantity about cars: their speed in a drag race.

[4]These concepts (especially classification) can also be applied to unsupervised learning. Recall from chapter 9 the discussion of competitive networks and self organizing maps, which learn to classify inputs into distinct categories without a teacher. The distinction also applies to recurrent networks, which can learn to classify dynamic inputs, for example, or to produce dynamic real valued outputs.

	Hours	SAT Score
1	1	350
2	3	400
3	4	410
4	4	390
5	5	450
...	...	...
16	13	730
17	13	700
18	14	730
19	16	770
20	22	790



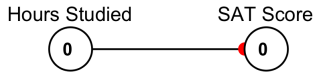


Figure 12.3: An example of regression. (Left) The data used to train the network. Hours of study vs. score on the SAT. (Middle) A plot of the data with a regression line. These are points in an input-target space, as discussed in chapter 12. This line can be used to predict how well someone will do based on how much they study. (Right) A simple 2-node network that could implement this regression solution. Enter hours studied in the input node, and it should display a predicted SAT score in the output node.

The term "regression" comes from the statistical technique of linear regression. In fact, neural networks provide a nice way to understand what linear regression is. To understand this, consider a classic example of linear regression: predicting how well someone will do on a test based on how many hours they study. As can be seen in figure 12.3, the more you study, the higher your score is likely to be. We can fit a line to this data using standard statistical techniques. But a neural network–like the one shown in the right panel of the figure–can also do it.[5] That network can be trained to predict SAT scores based on hours studied. Look how simple it is! That's all linear regression really does: it gives us a network that we can use to make predictions, in this case, simple predictions where a single input produces a single output. In fact, the details are also pretty straightforward. In this case, the slope of the line corresponds to the weight, and the intercept is the bias on the linear output node. Recall $y = mx + b$ from high school math class; here y is the output activation, x is the input activation, m is the weight, and b is the output node bias. Thus, in computing weighted inputs for the output node, when there is just one input, we are just computing a simple linear function.

Of course we can get more complex. We can have multiple inputs. We can predict how tall a tree will be based on its age, average rainfall where it is planted, and concentrations of chemicals in its soil. In that case we have multiple inputs predicting one output. In statistics this is called *multiple regression*, but you can see that it's just a matter of having a many-to-one network where we estimate the values of the weights and biases. When we have multiple outputs we just repeatedly use this technique on each output node. One output node predicts the height of the tree, another predicts how long it will live, etc. That is called *multi-variate multiple regression*. Thus networks with many inputs and many outputs can be understood as performing regression tasks.

As a simple procedure for deciding whether a task is a classification or regression task, look at the target data. If the target data represent categories, e.g. a one-hot encoding, it is probably a classification task. If they are real-valued or otherwise numerical data, then it is probably a regression task. Even more simply: classification tasks typically involve binary or discrete valued targets, while regression tasks typically involve real-valued targets.

12.4 Visualizing Classification as Partitioning an Input Region into Decision Regions

It is important in supervised learning to be able to conceptualize problems in terms of a set of graphical ideas. They are familiar ideas, and not too hard, but they confusingly overlap, so we must be careful and

[5]Note that in this example, the data have not been rescaled (chapter 7). Rescaling is often important, but not always necessary.

systematic about understanding them. You will see many diagrams that look quite similar. We will only be able to visualize what's going on directly for very small networks, but we can use these ideas to generalize to higher dimensions, which will give us a conceptual template for understanding more complex cases. This theme of visualizing ideas directly in small networks and then extending them to higher dimensions is often useful, as we will see.

The goal of a classification task is to create a model that correctly classifies inputs into a finite set of categories. Target values are binary; an input is either in a given category or not. Classification involves creating decision boundaries between inputs for the different categories. In this section we focus on decision regions produced by networks with a single weight layer and a single output node with a threshold activation function, but the ideas generalize in interesting ways to more complex networks (see Bishop [11], chapter 3).

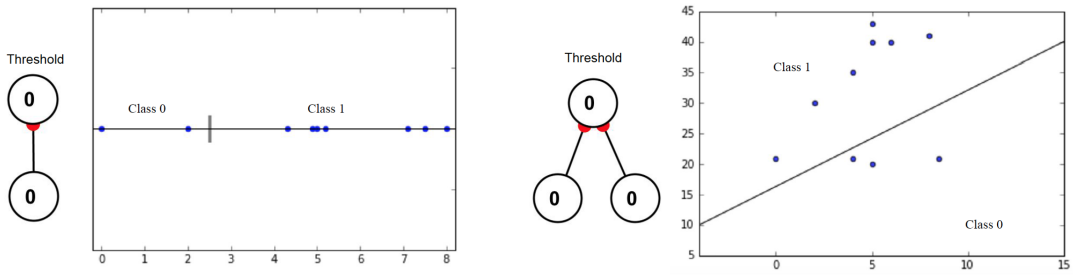


Figure 12.4: A classification task for a 1-1 network (Left) and a 2-1 network (Right). Both networks use threshold activation functions on the output nodes. Points in the input space are shown in blue. The decision boundaries between points classified as 0 or 1 are shown in gray. On the left, the decision boundary is a point shown as a small vertical hatchmark. On the right, the decision boundary is a diagonal line. The decision boundaries partition the input spaces into two decision regions, corresponding to outputs of 0 or 1.

Simple linear networks using threshold activation functions partition the input space into **decision regions** separated by lines, planes, and hyperplanes (a hyperplane is intuitively a plane in an high dimensional space). Figure 12.4 shows classification tasks for 1-1 and 2-1 networks. In each case, the output node has a threshold activation function and the network is being trained to classify inputs into one of two classes. When the output node turns on (weighted inputs above threshold), the input is in class 1. When the output node is off, the input is in class 0. On the left, the input space is 1-dimensional, since there is one input node. The threshold functions divide the 1-dimensional input space into decision regions labeled "class 0" and "class 1", via a **decision boundary**, in this case a 0-d point (represented by a hatchmark in the graph). On the right, the input space is 2-dimensional, and the decision boundary is 1-dimensional. The line again partitions the input space into two decision regions, for "class 0" and "class 1.'

These ideas generalize to higher dimensions. The *input space* will in general have as many dimensions as there are input nodes. The small networks in Figs. 12.4 and 12.5 have 1 and 2-d input spaces. A network with 5000 input nodes has a 5000-dimensional input space. The *decision boundary* in the input space of a network has as many dimensions as the number of input nodes minus 1. In the 5000-input node case, with one output node, the decision boundary is a 4999-dimensional hyperplane that divides the input space into two decision regions.

To summarize, for classification tasks we have:

Input space: the vector space corresponding to the input nodes of a network. It has as many dimensions as there are input nodes. In the cases shown in figure 12.4 the input spaces are 1 and 2 dimensional.

Decision boundary: a point, line, or surface separating the input space into decision regions corresponding to distinct categories. The boundary has as many dimensions as the number of input nodes minus one. In the cases shown in figure 12.4 the decision boundaries are $1 - 1 = 0$ dimensional (a point) and $2 - 1 = 1$-dimensional (a line).

12.5 Visualizing Regression as Fitting a Surface to a Cloud of Points

The goal of a regression task is to create a network that produces outputs as close as possible to a set of datapoints. Targets are real-valued. We can conceptualize regression as fitting a hypersurface (a generalization of lines, planes, and other surfaces to arbitrary dimensions)[6] as close to a cloud of data points as possible.[7] In the simple case shown in figure 12.5, left, we just have 1 input and 1 output. This is like graphing a function in high school algebra. The graph of the function is 1-dimensional, *i.e.* a line.[8] The graph of the function is one dimensional because there is one input node. However it is shown in a 2-dimensional input-target space, which is 2 dimensional because there is 1 input node plus 1 output node. Input/target pairs (*i.e.* rows of the labeled dataset) are plotted as points. Fitting the linear model, the neural network, can be thought of as turning two knobs: one for the weight (which sets the slope), and one for the bias (which sets the y-intercept). Think of trying to turn these knobs until the line (the "model") fits the data as best as possible. Training this kind of network is like fitting a linear regression model. Try to keep this easy-to-visualize example in mind even for much more complex networks, where there are many more knobs, and where the model being fit exists in many more dimensions.

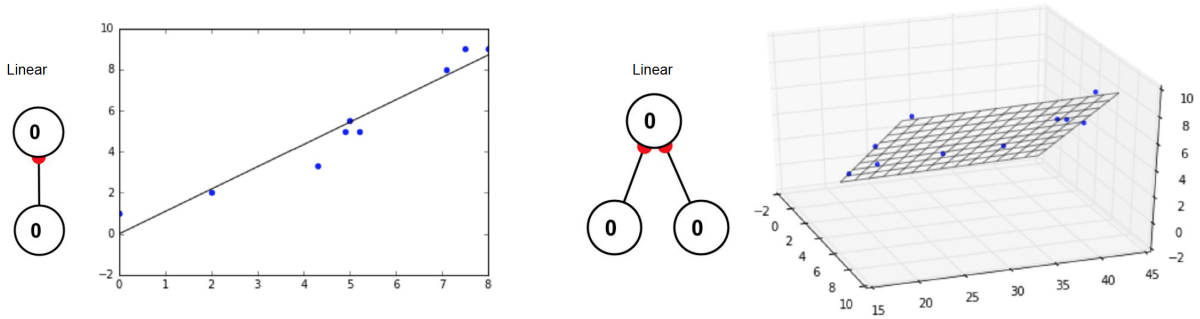


Figure 12.5: A regression task for a 1-1 (Left) and a 2-1 network (Right) network of linear nodes. Datapoints in the input-target space are shown in blue. The network implements a linear function from inputs to outputs, which is a line in the two dimensional input-target space on the left, and a plane in the three dimensional input-target space on the right.

Let's see how this works in a slightly larger network, a network with 2 input nodes and 1 output node, as in figure 12.5, right. Here the graph of the function computed by the network is a 2-dimensional surface, and the input-target space of the graph is 3 dimensional (2 input nodes and 1 output node). Visualize the algorithm fitting that surface so that it's as close to the datapoints as possible. It's like fitting the line in the 1-1 network case, but now we have more knobs (the two weights and the bias of the output node) for moving the surface around. Each of the vectors in the 2-d input space is associated with a target value in the 1-d output space. The surface has been fit to the points, so that any input will be associated with outputs as close as possible to the target values.

So here we have:

Regression hypersurface: a generalization of the concept of a regression line to arbitrarily many dimensions. Mathematically it is the graph of an n-dimensional hypersurface, where n is the number of input nodes. In the cases shown in figure 12.5 the hypersurfaces have 1 dimension (a line, for a neural network with 1 input node) and 2 dimensions (a plane, for a neural network with 2 input nodes). It is, in a sense, the "solution" a network learns for a regression problem.

[6] See `https://en.wikipedia.org/wiki/Hyperplane`. An even more general concept is a hypersurface: `https://en.wikipedia.org/wiki/Hypersurface`

[7] Regression models need not be "flat" like this. When sigmoidal node are used, for example, they will be more like wavy curves and surfaces. But we will not consider those cases here.

[8] In mathematics, even a curvy line is 1-dimensional, and even a wavy plane is 2-dimensional, even if they can only be visualized in a higher dimensional input-target space. Similarly for higher dimensions.

Input-target space: the space that the regression hypersurface lives in, which has as many dimensions as there are input nodes plus output nodes. The rows of the labeled dataset can be conceptualized as a cloud of datapoints in this space. The goal of regression is to make the hypersurface as close to that cloud of datapoints as possible. In the cases shown in figure 12.5 the input-target spaces have 2 dimensions (a neural network with 1 input node and 1 target node) and 3 dimensions (a neural network with 2 input nodes and 1 target node).

Notice that the decision boundary in figure 12.4 (right) looks like the regression line in figure 12.5 (left). Do not confuse these! In one case we have a decision boundary for a classification task computed by a 2-1 network; in the other case we have a regression line computed by a 1-1 network.

As soon as we have networks with more than 3 nodes, our ability to visualize things starts to break down. But we have experience thinking about shapes in higher dimensions (for example via our studies of dynamics in 4-dimensional state spaces, and recall our discussion of dimensionality reduction in section 6.3), so we should be able to do it. The regression hypersurface computed by a network has as many dimensions as there are input nodes. For example, if a linear network had 20 input nodes and 5 output nodes, its graph would be a 20-dimensional hyperplane fit to a cloud of points in a 25 dimensional space.

12.6 Error

To assess how well a supervised learning algorithm is training a network to approximate a vector valued function, we define an **error function**[9] and then modify the weights and other parameters of a neural network to get the value of this error function to be as small as possible (compare the intuitive overview in Section 12.2). An error function can be thought of as a method for producing a number which describes how well a network is doing at approximating the pattern-associations encoded by a labeled dataset. We call this the *overall error*, since it is associated with the entire labeled dataset (as contrasted with errors associated with particular outputs). In learning, the goal is to use mathematical techniques to change the parameters of the network so that overall error is as small as possible.

In practice, to compute overall error, we compute errors for each row of the training subset of our labeled dataset. We go through each in vector $\mathbf{x}_r$ and see what output vector $\mathbf{y}_r$ the network produces. The resulting output dataset (see Chapter 7) has the same number of rows and columns as the target dataset. We then go through each row $\mathbf{y}_r$ of the output dataset and compare it with the corresponding row $\mathbf{t}_r$ of the target dataset. We often end up comparing things like $\mathbf{t} = (1, 1)$ and $\mathbf{y} = (.9, .7)$ by subtracting the components of the two vectors, which gives us errors $(1 - .9, 1 - .7) = (.01, .03)$.[10]

The concept of error should be intuitive. If we want our network to produce the output vector $(1, 1, 1)$ but instead it produces $(-1, 0, .2)$, we have a few errors in the output. But if we train it and it starts to produce the output vector $(1, 1, .9)$, then we are doing much better! The first output is about 4 units "off", but the second output is about .01 off. That's all the basic idea involves. Saying this mathematically requires some notation, but remember all we're ultimately doing is finding a number that says how far "off" the outputs are from the target values.

There are many error functions that we can use to compute overall error.[11] Here we are just introducing the basic idea. We focus on an error function called *sum of squared error* or SSE.

Suppose we are given a network and labeled dataset, and have computed the output dataset. To compute SSE, we go through each row and subtract each output value from the corresponding target value. This gives an error value for each row. We then square these errors and add the squared errors together. That is the sum of squared errors. Figure 12.6 shows the computation for the case where there is a single output node.[12] In the case shown, outputs are very close to targets, and so the SSE is pretty low:

[9] These are also known as "cost functions", "loss functions", or "objective functions".

[10] At the level of an individual node, an error is just the difference between what we wanted and what we get, a target value minus an output value.

[11] One main consideration is whether you are training a network on a classification task or a regression task. A common error function used in classification tasks is cross-entropy. Sum squared error, which we consider here (or its close cousin mean squared error), is usually used for regression tasks, but it can also be used for classification, so it's a good example to start with.

[12] In these figures, $\sum_R$ is shorthand for $\sum_{r=1}^{R}$.

$$\sum_R \left(\begin{array}{|c|}\hline \text{targets} \\ \hline 1 \\ \hline 2 \\ \hline 3 \\ \hline 4 \\ \hline \end{array} - \begin{array}{|c|}\hline \text{outputs} \\ \hline .9 \\ \hline 1.9 \\ \hline 2.9 \\ \hline 3.9 \\ \hline \end{array} \right)^2 = \sum_R \left(\begin{array}{|c|}\hline \text{errors} \\ \hline 1 - .9 \\ \hline 2 - 1.9 \\ \hline 3 - 2.9 \\ \hline 4 - 3.9 \\ \hline \end{array} \right)^2 = \sum_R \left(\begin{array}{|c|}\hline \text{errors} \\ \hline .1 \\ \hline .1 \\ \hline .1 \\ \hline .1 \\ \hline \end{array} \right)^2 = \sum_R \begin{array}{|c|}\hline .01 \\ \hline .01 \\ \hline .01 \\ \hline .01 \\ \hline \end{array} = .01 + .01 + .01 + .01 = .04$$

Figure 12.6: Computing SSE for a network with one output node. As can be seen, each output is just .1 below the target, and so SSE is pretty low.

The same idea works for a network with more than one output node, but in that case the squaring operation must be defined for vectors. All this really amounts to is squaring target - output for each output value, and then adding the results together, and it turns out we can express this concisely using some of the vector operations we learned in the linear algebra chapter (chapter 6):

$$SSE = \sum_{r=1}^{R} (\mathbf{t}_r - \mathbf{y}_r) \bullet (\mathbf{t}_r - \mathbf{y}_r)$$

That is, for each row r, we subtract the output vector from the target vector using component-wise subtraction, and then take the dot product of the resulting vector with itself to get the squared error for that row. For example if $\mathbf{t}_1 = (1, 1)$ and $\mathbf{y}_1 = (2, 3)$, then $(\mathbf{t}_1 - \mathbf{y}_1) = (1, 1) - (2, 3) = (1 - 2, 1 - 3) = (-1, -2)$. We "square" this vector of errors by dotting it with itself: $(-1, -2) \bullet (-1, -2) = -1^2 + -2^2 = 5$.

Sample computations for a network with two outputs are shown in figures 12.7 and 12.8. Notice that SSE is low in figure 12.7, and higher in figure 12.8.

$$\sum_R \left(\begin{array}{|c|c|}\hline \text{targets} & \\ \hline 1 & 0 \\ \hline 0 & 1 \\ \hline 1 & 0 \\ \hline 0 & 1 \\ \hline \end{array} - \begin{array}{|c|c|}\hline \text{outputs} & \\ \hline .8 & 0 \\ \hline 0 & 1 \\ \hline 1 & 0 \\ \hline 0 & .9 \\ \hline \end{array} \right)^2 = \sum_R \left(\begin{array}{|c|c|}\hline \text{errors} & \\ \hline 1\text{-}.8 & 0\text{-}0 \\ \hline 0\text{-}0 & 1\text{-}1 \\ \hline 1\text{-}1 & 0\text{-}0 \\ \hline 0\text{-}0 & 1\text{-}.9 \\ \hline \end{array} \right)^2 = \sum_R \left(\begin{array}{|c|c|}\hline \text{errors} & \\ \hline .2 & 0 \\ \hline 0 & 0 \\ \hline 0 & 0 \\ \hline 0 & .1 \\ \hline \end{array} \right)^2 = \sum_R \begin{pmatrix} (.2, 0) \bullet (.2, 0) \\ (0, 0) \bullet (0, 0) \\ (0, 0) \bullet (0, 0) \\ (.1, 0) \bullet (.1, 0) \end{pmatrix} = .04 + 0 + 0 + .01 = .05$$

Figure 12.7: Computing SSE for a network with two outputs. Error is fairly low.

$$\sum_R \left(\begin{array}{|c|c|}\hline \text{targets} & \\ \hline 1 & 0 \\ \hline 0 & 1 \\ \hline 1 & 0 \\ \hline 0 & 1 \\ \hline \end{array} - \begin{array}{|c|c|}\hline \text{outputs} & \\ \hline 1 & 1 \\ \hline 1 & 1 \\ \hline 1 & 1 \\ \hline 1 & 1 \\ \hline \end{array} \right)^2 = \sum_R \left(\begin{array}{|c|c|}\hline \text{errors} & \\ \hline 1\text{-}1 & 0\text{-}1 \\ \hline 0\text{-}1 & 1\text{-}1 \\ \hline 1\text{-}1 & 0\text{-}1 \\ \hline 0\text{-}1 & 1\text{-}1 \\ \hline \end{array} \right)^2 = \sum_R \left(\begin{array}{|c|c|}\hline \text{errors} & \\ \hline 0 & -1 \\ \hline -1 & 0 \\ \hline 0 & -1 \\ \hline -1 & 0 \\ \hline \end{array} \right)^2 = \sum_R \begin{pmatrix} (0, -1) \bullet (0, -1) \\ (-1, 0) \bullet (-1, 0) \\ (0, -1) \bullet (0, -1) \\ (-1, 0) \bullet (-1, 0) \end{pmatrix} = 1 + 1 + 1 + 1 = 4$$

Figure 12.8: Computing SSE for a network with two outputs when error is higher, as might happen when we start with random weights, which, for example, cause the outputs to always be 1.

Low overall error (here low SSE) is something we generally see *after* training a network. When we start with an untrained network that has random weights, error will be higher, as in figure 12.8.

Try some examples of your own to get a feel for when SSE is large vs. small.

Learning algorithms minimize SSE and other overall error metrics relative to training subset of the labeled dataset. However, it is usually important to hold out some testing data as well, to see how well the network generalizes to data it was not trained on. Thus we often have two measures of error when we are done training a network: error for the training data, and error for the test data (see section 7.5).

12.7 Error Surfaces and Gradient Descent

Before we get to the main point of this section–gradient descent–we need to do a bit more visualizing. We are going to talk about *another* type of graph, separate from the ones discussed in sections 12.4 and 12.5. We will be talking about parameter spaces, or to make things easier, weight spaces (recall these have come up in chapters 10 and 9). On top of the weight space we will plot overall errors. That is, for each possible combination of weight values, we show what overall error (*e.g.* SSE) would result relative to our network and training set. This gives us an **error surface**.

Note that error surfaces depend on the training set, and in particular targets, because this determines SSE. If you change the training set, you change the error surface. Once you fix the training set, you can now look at the error surface, which shows all possible errors for that network *given* the training set.

Figure 12.9 shows errors surfaces for two cases that we can visualize. On the left we have an error surface for a 1-1 network where we are only adjusting a single weight. As we change that weight, SSE will change too. On the right we have a 2-1 network where we are adjusting two weights. Again, as the two weights are changed so will the SSE. Notice that in each case the error surface has a single minimum point, which turns out to be convenient: our goal, after all, will be to find that lowest point, the configuration of weights where overall error is lowest. But we are not always lucky enough to get such a surface, as we will see.

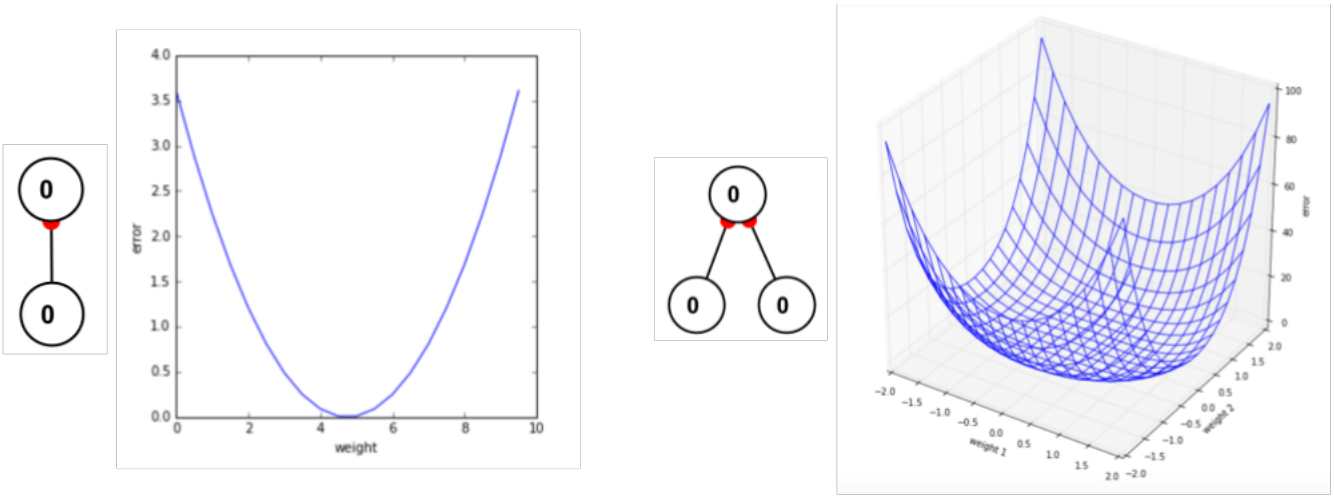


Figure 12.9: (Left) error surface for a 1-to-1 network where only the weight is adjusted. (Right) Error surface for a 2-to-1 network where the two weights are adjusted. In each case, the error surface also depends also the training data (the tables in figure 12.1).

These ideas generalize to higher dimensions, for networks with many parameters. As above, we can't visualize these cases directly, but it helps to have a visual template in mind. The error surface for a network with n adjustable parameters is a surface in an $n+1$ dimensional space (the n parameters plus the error term). For a network where we are adjusting three weights, we have the graph of a function from the three weights to the error, *i.e.* a surface in a 4-dimensional space. For each possible combination of three weights, we can generate an error, and thus we have an error surface in a 4-d space. For a network with 150 weights, we have an error surface in a 151 dimensional space.

For supervised learning tasks (regression or classification), our goal is usually going to be to *minimize the overall error function*. We want to find values for the parameters that make overall error, relative to the labeled dataset, as low as possible. It turns out there is a whole area of mathematics set up for problems like that. It's called **optimization**.[13] Optimization problems involve finding either the minimum value or maximum value of a function, relative to the set of all possible inputs to the function. Here the function is the error function and the inputs are adjustable parameters, usually weight strengths and node biases.

[13] `https://en.wikipedia.org/wiki/Mathematical_optimization`.

Optimization is useful. We often have to make decisions that involves many different variables and constraints. For example, suppose you want to buy a new laptop. You want the cost to be as low as possible, but the quality as high as possible. You need to buy it within 5 days and you really want a warranty. Often what you do is look at choices. When a tentative choice feels better, you go in that direction. If it feels worse you might tell yourself "no, I dislike that, look for something else". In this way you go back and forth–generally in the direction of "better"–and settle in on a solution. Once again, the Goldilocks principle.

Optimization automates this kind of process, providing an automatic way to solve this kind of problem. Optimization methods are used, for example, to determine the best ways to plant crops to maximize yield. Supervised learning methods also make use of optimization. A labeled dataset describes an optimization problem, a set of inputs that we want to associate with a set of targets. Mathematical optimization gives us a way to automatically update the parameters of a network so that it does the best job possible on a classification or regression task, producing outputs in response to inputs that are close as possible to their targets.

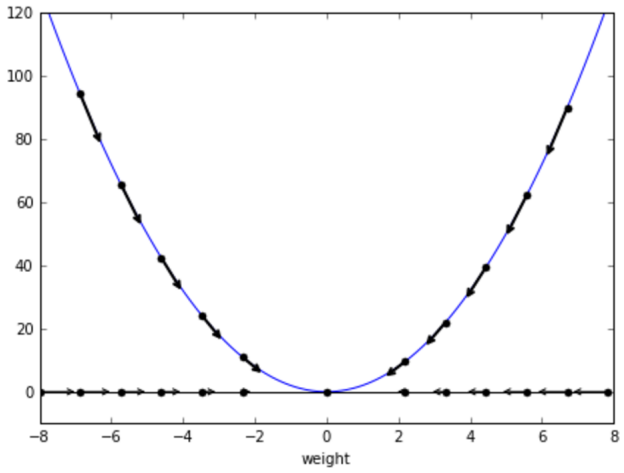


Figure 12.10: Error gradient on an error surface. The actual changes that happen in the weight space are shown on the horizontal axis.

The main method we will discuss for minimizing the error function is **gradient descent**.[14] In this method we start at some random point in parameter space. We begin with a network where the weights and biases and other adjustable parameters are set to random values. In the 1-to-1 network we just randomize that one weight. That puts it at a random place on the error surface over the weight space in figure 12.9 (left). Using the tools of calculus, we can then attach an arrow to any point on the error surface, which says in what direction the surface is decreasing most rapidly.[15] So we start at a random point, follow the arrow down from there (changing the weights to new values), and then repeat the process. By iterating the algorithm in this way we "descend the gradient."

The process is illustrated in figure 12.10. The error surface has a bowl shape. Wherever you start in the bowl, just follow the arrows down until you get to the low point. That's it! That's how it works.

We use the "arrows" on the error surface to derive a learning rule, which produces a *dynamical system on weight space*. In chapter 10 we mainly considered activation dynamics. Here we consider weight dynamics. The weight dynamics are shown in the bottom horizontal line of the figure; in that line a one-dimensional system describing how a single weight changes in order to implement a model of the training data. As noted in chapter 10, when a "play" button ▷ is pressed in Simbrain a dynamical process is simulated. In this case we will have a tools in Simbrain for running gradient descent using a play button, and observing error reduction.[16] What we are doing is like what we did there when we looked for fixed points of a recurrent

[14]The gradient of an error function is a vector that points in the direction of steepest increase on the error surface. The method of gradient descent involves changing a system in the direction of steepest decrease on the error surface, which is the negative of the gradient of the error function.

[15]An excellent discussion of the calculus relevant to neural networks is `https://explained.ai/matrix-calculus/`.

[16]See the screenshot here: `http://www.simbrain.net/Documentation/v3/Pages/Network/training/trainingDialog.html`.

network. Here the initial conditions are random weight values, which put us at a random point in the error surface, orbits are the paths we follow in weight space (which is our state space in this case), and we usually end up at an attracting fixed point, a weight state with low error.

Unfortunately, error surfaces don't always have just a single minimum point, as in the bowl example (if they did, training networks would always be easy). They sometimes have multiple fixed points in separate basins of attraction. These are "local minima". Error surfaces can also have plateaus where the error will only gradually change. Both cases are shown in figure 12.11. These can make finding the best solution difficult. Much of the mathematical theory of optimization (and research in neural networks) is focused on dealing with these "difficult" error surfaces.

We usually don't have a picture of an error surface. Still, we can use a program like Simbrain to get a feel for an error surface. To do so, we start at a random point in weight space, run the algorithm, and then see what the lowest error we get is. If it does not seem so low, we try again.[17] By repeatedly doing this we get a feel for how many minima they are and how long it takes to get to them. Of course, for very large networks, where training can take hours or days, we can't to this, but for small toy networks we can.

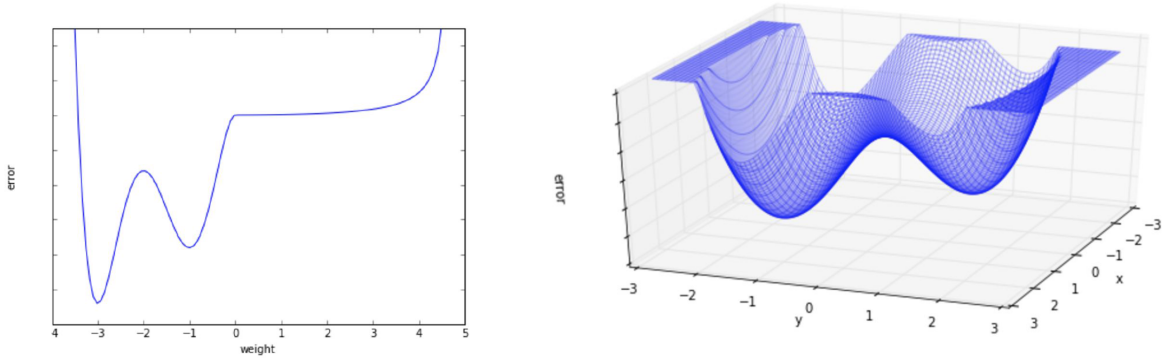


Figure 12.11: (Left) an error surface for one weight, with two local minima and a plateau. (Right) An error surface for two weights, with two local minima.

There is more to say here. A lot can go wrong, there are settings to adjust (like *how far* you go at each iteration), etc. These details are studied in the mathematical field of optimization.

12.8 Expansion of these methods

This chapter has described some very general features of supervised learning. The history and details of how these are applied in particular cases is spelled out in subsequent chapters. In chapter 13 we see details of this type of algorithm for the case of a simple one-weight-layer network, and then we see how backprop allowed the algorithm to be generalized to feedforward networks with more than weight layer. The key was calculus: by figuring out how to update parameters to reduce error using calculus, a systematic method for applying gradient descent to more complex networks was found.

One feature of the revolution in neural networks that began around 2010 (see section 3.7) is that these techniques became in a certain sense automated. Libraries were introduced that made it possible to define any kind of layer and link between layers, and as long as certain structures were defined, the calculus could be done automatically, allowing gradient descent to be used. This is sometimes called automatic differentiation, and it can be done on an arbitrary computational graph. That made it possible to really define all kinds of crazy networks and structures and layers and update rules and train them using tradient descent. Examples include convolutional layers (chapter 14) and transformers (chapter 17).

[17] It's easy to try this in Simbrain. Load up a backprop network with a training set, and train. Periodically press the random button and run again, and notice how the error changes each time. Again, this is almost exactly the same as searching for fixed points of a dynamical system, with the state space here being a weight space.

Because these methods are so powerful, it's often useful to find ways to apply them even when it seems they don't apply. For example, as we see in chapters 16 and 19, there are various ways it has been possible to use these same techniques–which work best with feedforward networks–on or with recurrent networks.

12.9 SSE Exercises

1. Given targets $(0, 1, 0)$ and output activations $(1, 1, 1)$ what is SSE?
Answer:
$(0-1)^2 + (1-1)^2 + (0-1)^2 = 1 + 0 + 1 = 2$

2. Given targets $(1, 1, 1)$ and output activations $(2, -1, -2)$ what is SSE?
Answer:
$(1-2)^2 + (1-(-1))^2 + (1-(-2))^2 = 1 + 4 + 9 = 14$

3. Given targets $(1, 1, 1)$ and output activations $(1, 1, 1)$ what is SSE?
Answer:
$(1-1)^2 + (1-1)^2 + (1-1)^2 = 0 + 0 + 0 = 0$

Chapter 13

Least Mean Squares and Backprop

Jeff Yoshimi

Having introduced supervised learning in chapter 12, in this chapter we consider several supervised learning algorithms for feedforward networks in depth, and consider their implications for cognitive science. First, an early form of supervised learning called the "Least Mean Squares rule" or "LMS rule". The remarkable thing about LMS is that almost all of the advancements in the area of neural networks in recent years arguably involve variations on this one simple algorithm, that allow it to solve increasingly complex problems. LMS itself is limited insofar as it can only be used to train networks with one weight layer. This takes us to a second algorithm, backpropagation or "backprop", which extends LMS in a way that allows it to be used to train many-layered networks. The hidden layers of a feedforward network develop *internal representations*, which remap the input space of a neural network in useful and often psychologically realistic ways.

13.1 Least Mean Squares Rule

We now consider our first supervised learning algorithm in detail: the **Least Mean Square**s rule or "LMS" or "Delta rule", which uses a form of gradient descent to minimize the error on a training set. It makes a nice contrast with Hebbian learning. Although the rule itself is similar in form, it is *adaptive*. While repeated application of the Hebb rule leads weights to explode to maximum or minimum values, LMS follows the Goldilocks principle, so that the weights zoom in on target values until the output is "just right."[1]

Note that LMS is an algorithm that only works with 2-layer networks. It cannot be directly applied to multi-layer networks. That is an important limitation that is overcome by backprop, as we will see.

LMS follows the template from section 12.2: begin with random parameters (weights and biases), iterate through each of the input patterns, compute errors, and use these errors to update weights and biases in such a way that errors are reduced. We will focus on the case of a single linear output node here. [2] When applying the rule, a change in a weight $w_{i,j}$ is equal to the product of a learning rate ϵ, the activation of the

[1] This rule is a descendent of Rosenblatt's perceptrons ([103]; also see chapter 3), which had one weight layer and a single binary output that was applied to classification tasks. His perceptron learning algorithm involved an if-then rule which would apply a form of gradient descent if an input was misclassified. Since both targets and outputs were thresholded binary units, this led to weight and bias updates that were "jittery" and sometimes unstable. The LMS rule, associated with Widrow and Hoff, was similar to the perceptron learning rule but worked with linear or sigmoidal outputs. (This is a natural progression insofar as a sigmoidal activation function can approximate a threshold activation function when the slope at the inflection point is steep enough). It was derived using calculus by finding the derivative of a point on an error surface over parameter space, and updating the parameters in a direction that led to error reduction (see section 12.7). It thus replaced the effort to create learning rules using intuition with a principled mathematical way of deriving learning rules mathematically. The same mathematical methods are still in use today and have enabled many of the incredible advances that have occurred in the field. Note that terminology in this area is not entirely consistent. For example, any network trained by LMS is often called a "Perceptron".

[2] We will not formally derive it, or state the algorithm in its full generality. However, the derivation is not too difficult: the key step involves taking the derivative of the error function with respect to a weight (how much is error changing as a function of that particular weight). A brief derivation is at `https://en.wikipedia.org/wiki/Delta_rule` and a more detailed discussion is at `http://uni-obuda.hu/users/fuller.robert/delta.pdf`. In subsequent planned sections we derive the rule, and extend it rule to handle many outputs, and more complex activation functions.

input node to that weight, a_i, and the difference between a desired and actual activation $(t_j - a_j)$ for that node:

$$\Delta w_{i,j} = \epsilon a_i (t_j - a_j)$$

Since $(t_j - a_j)$ is error (with a small "e"), the rule says that the change in a weight is equal to a learning rate times the activation of the input node for that weight, times the error (that is, $\Delta w_{i,j} = \epsilon a_i e_j$), This means that for positive inputs, weights are changed in the direction of error.[3] Notice that the form of the rule is identical to the Hebb rule, except with output activation replaced by error.

LMS changes the bias b_j of an output node j as follows:

$$\Delta b_j = \epsilon (t_j - a_j)$$

That is, the change in a bias for a node j is just the learning rate times the error on output node j.[4]

The LMS rule directly instantiates the Goldilocks principle. When heating up soup, or a bath, or a room, we adjust up and down based on errors.[5] Oh, it's too cold, let's heat it up. Oh wait, now it's too hot, cool it down. Ok now it's perfect, stop changing things, it's just right. In a similar way here we increase and decrease parameters until error is as low as possible. Consider how this works in a few cases where we assume a positive input:

- Target is 2 and output is 1. Output is too low. Error is $2 - 1 = 1$ and so we make the weight stronger.
- Target is 1 and output is 2. Output is too high. Error is $1 - 2 = -1$ and so we make the weight weaker.
- Target is 1 and output is 1. Output is just right. Error is $1 - 1 = 0$ and so we don't change a thing.

13.2 LMS Example

In this example and in subsequent practice questions, assume we have a simple 1-1 feed-forward network (as in the left panel of figure 12.1), and that the slope of the output node is 1 and bias is 0. So we have two nodes, with activations a_1 and a_2, and a weight $w_{1,2}$. We also assume a very simple labeled dataset with a single row: one input value and one corresponding target value. That is:

inputs	targets
1	2

We label the target value t. We will not consider updates to the bias term. As in the discussion of the Hebb rule (chapter 9), we use the prime symbol $'$ to indicate a variable after a delta term has been applied.

We can now work out a complete example, and in the process see how LMS implements gradient descent. Suppose we are given:

$$a_1 = 1$$
$$w_{1,2} = 1.5$$
$$t = 2$$
$$\epsilon = .5$$

With this information we can determine: (1) the output activation a_2 of the network (our "forward pass"), (2) SSE, (3) the updated weight value at the next time step, which we designate $w'_{1,2}$, (4) the output activation

[3] The input scales things so they work for positive or negative inputs and so that changes are bigger for bigger inputs, and the learning rate allows us to control how quickly learning happens. Even though we are giving interpretations for each factor in the rule, it can be derived purely from calculus.

[4] Note that this is really the same as the weight change rule, if we think of the bias in terms another input neuron, clamped at 1, and attached to this output neuron by a modifiable weight (which is in effect the bias).

[5] In fact, the LMS rule is comparable to a bunch of thermostats, one for each output node of a network, where the targets correspond to the settings on the thermostats.

a_2', and (5) error at the next time step, SSE′. We can then repeat these steps and check to see that SSE is reduced over time, which moves us down the error surface for this task, which is shown in Fig. 13.1.

(1) The network will produce a 1.5 in response to an input of 1, since the input activation is 1, the weight is 1.5, and $1 \cdot 1.5 = 1.5$ (we are, again, assuming output bias of 0 and slope of 1).

(2) SSE for these simple networks and labeled datasets is very easy, since there is just one row, one target value, and one output value. That is, SSE $= (t - a_2)^2$ or in this case $(2 - 1.5)^2 = .5^2 = .25$. Notice that this puts us at the point $(1.5, .25)$ in the graph in Fig. 13.1.

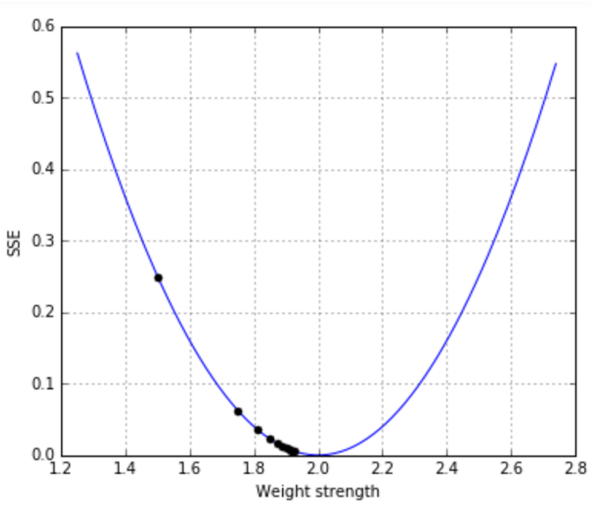


Figure 13.1: Gradient descent on the error surface for the LMS example discussed in Section 13.2. As the LMS rule is applied the weight strength changes in a way that minimizes sum squared error.

(3) Applying the formula above

$$\Delta w_{i,j} = \epsilon a_i (t_j - a_j)$$

we get

$$\Delta w_{1,2} = .5 \cdot 1 \cdot (2 - 1.5) = .25$$

We then use Δw to update the weight value from its old value of 1.5, so that

$$w_{1,2}' = w_{1,2} + \Delta w_{1,2} = 1.5 + .25 = 1.75$$

So our new weight value is 1.75.

(4) With this new weight, the network produces an output $a_2' = 1 \cdot 1.75 = 1.75$.

(5) The squared error is now $(2 - 1.75)^2 = .25^2 = .0625$, whereas before it was .25. So we have moved to the point $(1.75, .0625)$ in the graph of the error surface in Fig. 13.1. An improvement! We have moved lower on the error curve; we have descended the error gradient.

In subsequent time steps we get:

$$w_{1,2}'' = 1.75 + .5 \cdot 1 \cdot (2 - 1.75) = 1.875$$
$$w_{1,2}''' = 1.875 + .5 \cdot 1 \cdot (2 - 1.875) = 1.9375$$

As you can see, applying this rule leads to SSE getting lower and lower, and the output getting closer and closer to the desired output of 2. Ten successive points on the error curve are shown in figure 13.1.

13.3 Linearly Separable and Inseparable Problems

Two-layer feed-forward networks with linear output nodes, like LMS networks, are in a certain way limited. That limitation played an important role in the history of neural networks, paving the way for studies of *internal representations* in neural networks, which had lasting consequences both in machine learning and in connectionist applications of neural networks to psychology. The limitation concerns the **linearly separable** classification tasks. Thus, in this section, and in much of the rest of the chapter, we focus on classification rather than regression.

To understand what linear separability (and inseparability) are, recall that a classification task assigns each input to a different category. If we focus on networks with two input nodes and one output node, then we can plot a classification task as in figure 12.4 (Right), but we can also directly label the points as 0 and 1, as in figure 13.2. When we create this type of plot, it often becomes immediately clear what the relationship between the categories is, in the input space. In figure 13.2 (Left), for example, we can immediately see that the two classes are distinct in the input space. Notice that we can separate the two categories by drawing a line between them, as in figure 13.2 (Middle). Such a line is, as we saw in section 12.4, a *decision boundary*, which has the effect of separating the input space in to two *decision regions*, one for each possible classification. Input vectors in the region below the decision boundary will be classified as 0, while those in the region above the boundary will be classified as 1. However, note that for the task shown in figure 13.2 (Right) there is no way to use a line to separate the 0's and 1's perfectly.

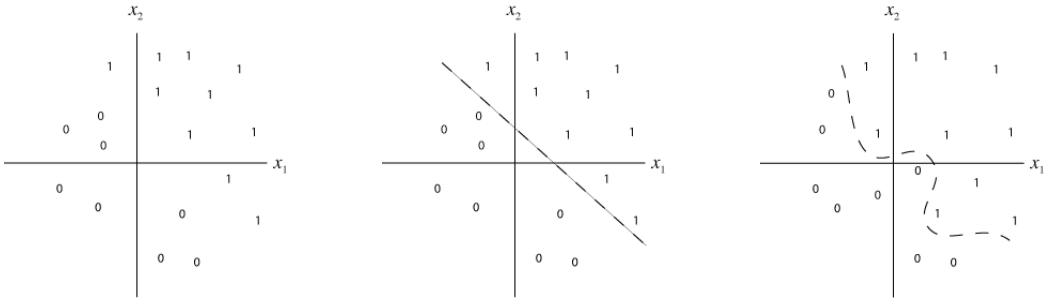


Figure 13.2: Three classification tasks. (Left) A linearly separable task. (Middle) A decision boundary that will solve the task. (Right) A linearly inseparable task and a non-linear decision boundary that can solve it.

If a classification task can be solved using a decision boundary which is a line (or, in more than 2-dimensions, a plane or hyperplane), the classification problem is called a **linearly separable** problem. Figure 13.2 (Middle) shows a linearly separable problem. When we cannot properly separate the categories with a line (or hyperplane), as in figure 13.2 (Right), the problem is **linearly inseparable**, there is no way to draw a line which separates the 0's and 1's in that example. There is no linear decision boundary for that problem (though there are non-linear decision boundaries that perfectly separate the classes, like the wavy curve shown in the figure).[6]

The goal of supervised learning of classification tasks is to set the weights of a network so that the decision boundary properly separates the two classes. The values of the weights and the output bias are like knobs that, when turned, will change where the decision boundary is: it can be rotated around and moved up and down. We want to turn the knobs so that they two classes are properly separated.[7] However, LMS only allows linear decision boundaries. Other algorithms have more knobs, and can be used to create more complex decision boundaries and decision regions.

Logic gates provide a convenient and historically important class of tasks that can be used to further illustrate these ideas. In appendix A it is shown that logic gates can be represented as 2-1 feed-forward neural networks. Pairs of input nodes corresponding to statements P and Q connect to output nodes representing boolean combinations of truth values (0 for false, and 1 for true): P AND Q (true when both are true), P

[6]First, note that in the case shown, we could still fit a line to the problem and we'd just have some error. Second, we will see that while LMS cannot solve this task, since it uses linear decision boundaries, other supervised classification algorithms like backprop exist that can solve these types of non-linearly separable classification task

[7]Doing this amounts to minimizing error. When the decision boundary properly separates the two classes, SSE will be 0. If not, as in Fig. 13.2 (Right), there will be some error. What will SEE be in that case?

OR Q (true when at least one is true), and P XOR Q (true only when one is true). These can be depicted using the same kinds of classification plots as above (here using open dots for 0, and filled dots for 1). Fig. 13.3 shows the input space for the AND, OR and XOR logic gates. Note that AND and OR are linearly separable, and that XOR is not.

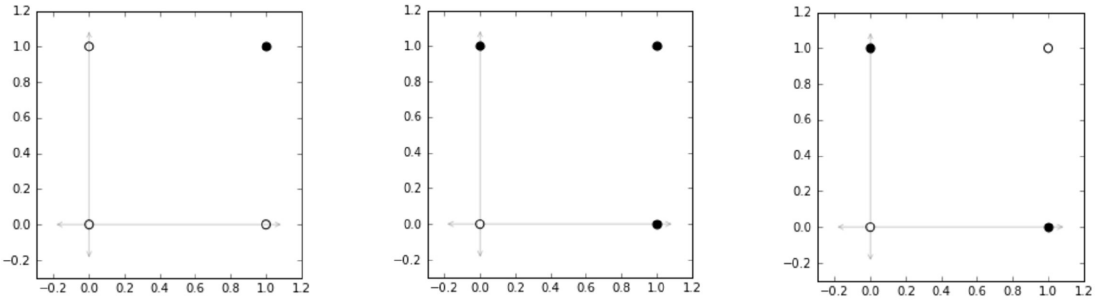


Figure 13.3: Input spaces for AND (left), OR (middle), and XOR (right). Open dots correspond to 0, filled dots to 1. Which tasks are linearly separable?

Now we get to the major problem affecting two layer networks trained using LMS: *they cannot solve linearly inseparable classification tasks*, like XOR. That two-layer linear networks cannot solve these problems was a major issue in the early history of connectionism. In 1969 Marvin Minsky and Seymour Papert published a book called *Perceptrons* (perceptrons were 2-weight-layer networks Rosenblatt trained; see section 3.4). In this book Minsky and Papert showed that such networks could not solve linearly inseparable problems [83]. This had a disastrous impact on neural network research in the following decade, during the "dark ages" of neural networks (see section 3.5). As Rumelhart and McClelland recall:

> Minsky and Papert's analysis of the limitations of the one-layer perceptron[8], coupled with some of the early successes of the symbolic processing approach in artificial intelligence, was enough to suggest to a large number of workers in the field that there was no future in perceptron-like computational devices for artificial intelligence and cognitive psychology (PDP 1, p. 112) [106].

However, as Rumelhart and McClelland go on to point out, these results don't apply to neural networks with more than 2 layers [106]. In fact, it has since been shown that multilayer neural networks with sigmoidal activation functions in the hidden layers are universal approximators in the sense that they can, in principle, approximate almost any vector-valued function (more specifically, any "Borel measurable function from one finite-dimensional space to another" [56].[9]

So multi-layer feed-forward networks can solve linearly inseparable problems. Great! But alas, there was another problem. Initially there was no way to train multi-layered networks. LMS only works on 2 layer networks. Minsky and Papert, who first clearly identified this problem, recognized that adding hidden layers could surmount the limitations they described. However, they thought that multi-layer networks were *too* powerful, describing them as "sufficiently unrestricted as to be vacuous" (Rumelhart and McClelland, p. 112) [106]. In particular, Minsky and Papert pointed out that no one knew how such a network could be trained to solve specific pattern association tasks [83].

Once algorithms were discovered that could be used to train multi-layer networks, it became possible to have networks learn solutions to linearly inseparable classification tasks (by finding non-linear decision boundaries) and to deal with much more complex problems than had previously been possible. The most famous algorithm of this type was backprop, which we turn to now.

[8] They are referring to a single weight layer connecting two layers of nodes. So what Rumelhart and McClelland call a "one-layer" network is what we have called a "2-layer" network

[9] This has since come to be known as the universal approximation theorem, and there is now a detailed Wikipedia page on the topic: `https://en.wikipedia.org/wiki/Universal_approximation_theorem`.

13.4 Backprop

In this section we cover what is probably the best known form of supervised learning: **backpropagation** (or "backprop"). Backpropagation is a powerful extension of the Least Mean Square technique. As we saw, LMS only works for two-layer networks. Backprop works for a much broader class of networks, in particular networks with non-linear activation functions containing one or more hidden layers. As we will see, these hidden layers allow a network to transform inputs into different types of representation, and in doing so makes them quite powerful (almost all modern neural networks use variants of backprop), and also psychologically interesting.

Backprop can be thought of as a generalization of the LMS technique or "Delta rule" described in the last few sections. In fact, backprop is sometimes called the "generalized delta rule." This rule had been proposed as early as the late 1960s / early 1970s [16, 122] and was independently discovered by several theorists in the 1980s [67, 94]. It was popularized by Rumelhart, McClelland, and Williams in the late 1980s [106]. The discovery and popularization of backprop led to a revival of interest in neural networks in the 1980s and 1990s, following the "dark ages" of the 1970s (again, see chapter 3).

We will not cover the details of the backpropagation algorithm here[10], but will instead describe it in a qualitative way. In fact, it is identical to LMS at the output layer, but adds a way to update the hidden weights. In fact, the weight updates are also almost the same, but the computation of the error is more complex, because we don't have direct access to the targets. That is, for a weight $w_{i,j}$ connecting an input node a_i to a hidden layer node a_j, the update rule is:

$$\Delta w_{i,j} = \epsilon a_i e_j$$

Where e_j is now just a different kind of error. This error is computed by a kind of reversal of how weighted inputs are computed in forward propagation. With weighted inputs or net inputs (see 5) to a node are computed, input activations are multiplied by intervening weights and added together. In linear algebra terms, the dot product of an input vector and fan-in weight vector is taken. This propagates activation forward. But now, we do the reverse. We take the errors on nodes at some level of a feedforward network, and multiply them by the intervening weights with the previous layer, and add the resulting products. That is, we backpropagate error. The idea is illustrated in figure 13.4. In the example shown, assume the weights from the hidden unit to the output are 1, so that the backpropped error is $2 * 1 + 0 * 1 = 2$. That error can then be used to update $w_{1,2}$ using $\Delta w_{1,2} = \epsilon a_1 e_2$. Notice that all that has changed from LMS is that we got the error via this backpropagation procedure.[11]

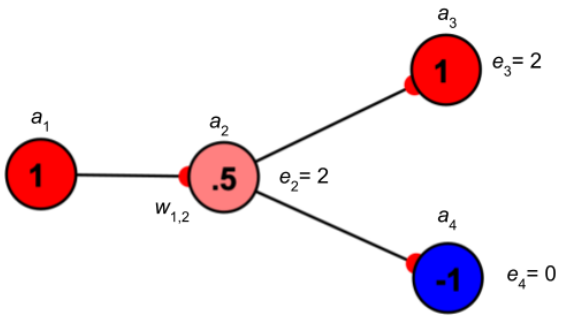


Figure 13.4: How error is backpropagated. Output errors are multiplied by intervening weights and added together (given the numbers, we can assume the target values are $t_3 = 3$ and $t_4 = -1$). In this case assume the weights are all 1. Thus the error e_2 at the hidden unit a_2 is $2 * 1 + 0 * 1 = 2$.

Intuitively, this corresponds to how much hidden unit contributes to a given output nodes' error It's as if "blame is assigned," and on this basis the weights to the hidden layer are updated.

[10] Though, as with the details LMS, a detailed discussion of backprop is being planned now.

[11] If $\epsilon = 1$, the weight $w_{1,2}$ will be updated from 1 to $1 + 1 \cdot 1 \cdot 2 = 3$. This will improve a_3 but make a_4 worse (however, repeated application of the rule with a lower learning rate ϵ would reduce error).

Since this idea works for any errors on the output of a weight layer, the rule can be applied recursively starting at the final output layer of a feedforward network and moving backwards through all previous layers. Thus it can be used to train many-layered networks.

As with LMS, backprop works by minimizing an error function with respect to a training set, so that we have gradient descent on an error surface. However, since multi-layer feed-forward networks are more complex than 2-layer networks, the error surface is more complicated. With two-layer linear networks, the error surface has a relatively simple bowl-like structure, which often has a single minimum value at the low-point of a "bowl" shape. With a multi-layer non-linear network, the error surface can be more complex and wavy, and there can be multiple local minima (cf. figure 12.11). These local minima can "trap" the gradient descent procedure, producing sub-optimal solutions.

One simple way to try to deal the issue is by re-initializing the parameters and re-running the algorithm. Each time this is done, the system moves to a different point on the error surface and tries finds the local minimum from that spot. By trying multiple times one can "search" for the lowest minimum possible.[12] This is like dropping a marble at different spots on the error surface and comparing how low the marble goes each time. In this way we can find the lowest of several local minima (which might turn out to be the global minimum) and in this way we can try to improve a network's performance on a task. In practice, however, one uses advanced optimization techniques that do things to automatically search for a global minimum.[13]

13.5 XOR and Internal Representations

We have not described in detail how the backprop algorithm works. However, we can get insight into what it does by considering what happens in the hidden layer of a network trained using backprop. In particular, we can begin to understand how multi-layer networks, like our brains, can solve problems by remapping input spaces to hidden unit spaces that contain useful internal representations.

The classic example to illustrate these ideas is the XOR problem. Recall that XOR, considered as a vector valued function, is not linearly separable (figure 13.3, right). Here is the labeled dataset we would use to train a network to implement XOR:

inputs		targets
x_1	x_2 s	t_1
0	0	0
1	0	1
0	1	1
1	1	0

As we saw, two-layer networks with linear units cannot solve this type of problem, but a three-layer network with non-linear units can solve it. This is easy to confirm in Simbrain: try training an LMS and Backprop network on this data, and notice the difference in the minimum error you can achieve in the two cases.

The key to backprop's superior performance is the way it *re-maps* the linearly inseparable problem in the input space to a linearly separable problem in the hidden unit space, as shown in figure 13.5. The crucial thing the hidden layer did was transform the input layer representation into a new internal representation, which includes a representation of "only one unit is on" and another representation of "both units are in the same state." These two states are now linearly separable, and the output layer can easily separate them. The solution shown in the figure was produced by training a 2-2-1 network using backprop. Other solutions (corresponding to other minima in the error surface) can also be found. You are encouraged to try the experiment yourself in Simbrain. Train a backprop network on XOR, get it to a minimum on the error surface, and then check to see what hidden layer activations occur for each input.

13.6 LMS Exercises

1. Given $a_1 = 1, t_2 = 4, a_2 = 2, \epsilon = 1$, what are e_2 and $\Delta w_{1,2}$?
Answer:

[12] Compare the way we searched for fixed points in chapter 10, by starting at different random points in state space.
[13] Currently the industry standard seems to be the "Adam" method: `https://arxiv.org/pdf/1412.6980.pdf`.

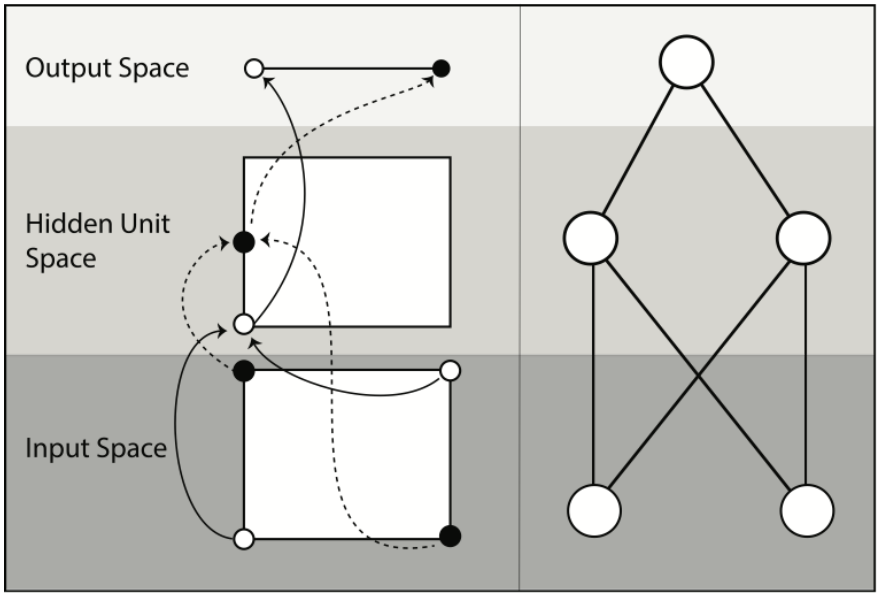


Figure 13.5: A remapping of the input space to the hidden unit space in the XOR problem. Note that the bottom panel shows the input space for XOR, and that it is linearly inseparable. The network then maps $(0,0)$ and $(1,1)$, to $(0,0)$ in the hidden unit space. $(0,1)$ and $(1,0)$ are mapped to $(0,.5)$ in the hidden unit space. Now notice that the hidden unit space is linearly separable! Also notice that the hidden unit space has developed an internal representation of the two main cases of interest: just one unit is one (represented by $(0,.5)$) and both units are in the same state (represented by $(0,0)$). Thus the separated hidden unit states can be mapped to the appropriate output states.

(1) $e_2 = t_2 - a_2 = 4 - 2 = 2$
(2) $\Delta w_{1,2} = \epsilon a_1 e_2 = 1 \cdot 1 \cdot 2 = 2$

2. Given $a_1 = 1, t_2 = 1, a_2 = 2, \epsilon = 1$, what are e_2 and $\Delta w_{1,2}$?
Answer:
(1) $e_2 = 1 - 2 = -1$
(2) $\Delta w_{1,2} = \epsilon a_1 e_2 = 1 \cdot 1 \cdot -1 = -1$

3. Given $a_1 = 1, t_2 = 2, a_2 = 2, \epsilon = 1$, what are e_2 and $\Delta w_{1,2}$?
Answer:
(1) $e_2 = 2 - 2 = 0$
(2) $\Delta w_{1,2} = \epsilon a_1 e_2 = 1 \cdot 1 \cdot 0 = 0$

4. Given $a_1 = 1, t_2 = 1, a_2 = 9, \epsilon = .25$, what are e_2 and $\Delta w_{1,2}$?
Answer:
(1) $e_2 = t_2 - a_2 = 1 - 9 = -8$
(2) $\Delta w_{1,2} = \epsilon a_1 e_2 = .25 \cdot 1 \cdot (-8) = -2$

5. Given $a_1 = 1, w_{1,2} = 1, t_2 = 2, \epsilon = 1$, what are the initial activation a_2, error e_2, and $\Delta w_{1,2}$, and after one weight update, the updated weight $w'_{1,2}$, updated activation a'_2 and updated error e'_2?
Answer:
(1) $a_2 = a_1 \cdot w_{1,2} = 1 \cdot 1 = 1$
(2) $e_2 = t_2 - a_2 = 2 - 1 = 1$
(3) $\Delta w_{1,2} = \epsilon a_1 e_2 = 1 \cdot 1 \cdot 1 = 1$
(4) $w'_{1,2} = w_{1,2} + \Delta w_{1,2} = 1 + 1 = 2$
(5) $a'_2 = a_1 \cdot w'_{1,2} = 1 \cdot 2 = 2$
(6) $e'_2 = (2 - 2) = 0$

6. Given $a_1 = 2, w_{1,2} = 1, t_2 = 3, \epsilon = .1$, what are the initial activation a_2, error e_2, and $\Delta w_{1,2}$, and after one weight update, the updated weight $w'_{1,2}$, updated activation a'_2 and updated error e'_2?
Answer:
(1) $a_2 = a_1 \cdot w_{1,2} = 2 \cdot 1 = 2$
(2) $e_2 = t_2 - a_2 = 3 - 2 = 1$
(3) $\Delta w_{1,2} = \epsilon a_1 e_2 = .1 \cdot 2 \cdot 1 = .2$
(4) $w'_{1,2} = w_{1,2} + \Delta w_{1,2} = 1 + .2 = 1.2$
(5) $a'_2 = a_1 \cdot w'_{1,2} = 2 \cdot 1.2 = 2.4$
(6) $e'_2 = (3 - 2.4) = .6$

7. Given $a_1 = 1, w_{1,2} = 3, t_2 = 1, \epsilon = 1$, what are the initial and final error e_2 and e'_2 after one weight update? **Answer:**
(1) $e_2 = 1 - 3 = -2$
(2) $e'_2 = 1 - 1 = 0$

8. Given $a_1 = 1, w_{1,2} = -1, t_2 = 1, \epsilon = 1$, what are the initial and final error e_2 and e'_2 after one weight update? **Answer:**
(1) $e_2 = 1 - (-1) = 2$
(2) $e'_2 = 1 - 1 = 0$

Chapter 14

Convolutional Neural Networks

Jeff Yoshimi, Pierre Beckmann

Convolutional neural networks are a prominent type of deep network. They are a kind of many-layered network that is often used for processing visual inputs, like recognizing patterns in images or movies, though they have broader applications. They make use of a special kind of mapping between node layers, a convolutional layer.

Convolutional neural networks were the first notable deep networks that got deep learning out of its second "dark age", by stacking a large number of hidden layers (following the first resurgence of interest in neural networks in the 1980s and 1990s; chapter 3).[1] They were trained using new tricks and techniques, like convolutional layers, the use of graphical processing units for fast parallel computation, and the Relu activation function discussed in chapter 5. These advances made it possible to use the same types network discussed in chapter 13 on much more difficult problems.[2]

In terms of engineering, these convolutional networks have been associated with improvements in image recognition, speech recognition, language translation, and in many other areas [68, 38]. They do this by creating hierarchies of representations, corresponding to increasingly complex features of an input image. As we saw in chapter 4 (see figure 4.7) when such networks are trained to recognize images they develop internal representations that are extremely similar to those developed by the human visual system. Thus they are relevant both to neuroscience (where they can describe the behavior of neurons in the visual system), and to psychology (where they can describe internal representations humans might rely on).

The topic of convolutional networks is quite involved and the field is active and continues to grow. Here we will describe some of the main concepts and some of their applications to neuroscience and psychology.

14.1 Convolutional Layers

The key idea with a deep network is to use a special type of weight layer called a **convolutional layer** to efficiently learn to recognize features in a previous layer.[3] Until now we've been dealing with weight layers that connect all the nodes in a source layer to all the nodes in a target layer; these are sometimes called "dense layers" or "fully connected" layers. By contrast, convolutional layers involve a set of weights that are "scanned" or "passed" over the source layer's activations to produce activations in a target layer. This can also be called a "convolution" of the source layer activations. Networks that feature convolutional layers are called "convolutional neural networks" or "CNNs".

The set of weights in a convolutional layer are called a **filter** or **kernel**. The weights in a filter are scanned over the source layer to produce outputs. This is a new idea. Rather than a fixed set of weights in

[1] The internal representations of these networks–usually 3 node-layer networks trained by backprop–allowed them to solve previously unsolvable problems like XOR (figure 13.5). These representations were often psychologically realistic (section 15).

[2] This history is well told by Kurenkov in section 3 of `https://www.skynettoday.com/overviews/neural-net-history`. As he summarizes, "Deep Learning = Lots of training data + Parallel Computation + Scalable, smart algorithms."

[3] An outstanding visual discussion of the concept of a convolutions is at `https://youtu.be/KuXjwB4LzSA`

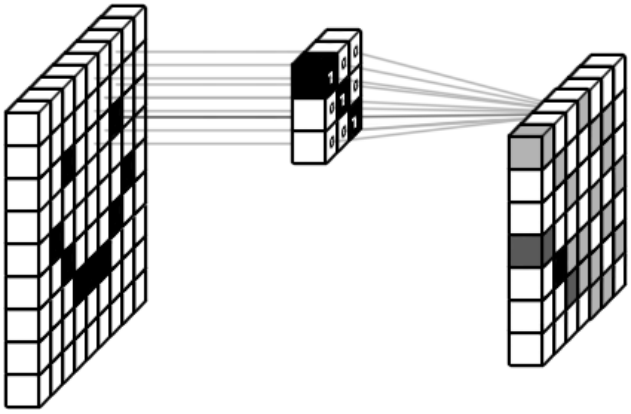


Figure 14.1: A convolutional layer and its components. From left to right: an input image, a 3x3 convolutional filter (which detects edges with a $-45°$ angle), and the resulting feature map. The filter is scanned across the image. At each stage of this scanning process, the dot product of the filter's receptive field in the input image is computed and used to populate the feature map. This whole process is known as a convolution and a layer like this is a convolutional layer.

a fully-connected weight layer, we have a small set of weights that are reused or shared during the scanning operation (this is also known as "weight sharing").

The source layer of a convolutional layer is often itself a 2d array, prototypically an input image, and so we can think of source layer activations as pixels and the source layer as a pixel array or input volume (even when the input is not an image this language is useful). Filters are like small pixel patterns that we slide across the whole pixel array, which "light up" most when they are on top of a similar pixel pattern. The filter can be thought of as is if it were moved from left to right and top to bottom of the pixel array. At each stage of the scanning process, it is multiplied by the patch or "receptive field" of the pixel array it is on top of.[4] The multiplication is a dot product (see chapter 6), where each weight in the filter is multiplied by the corresponding activation of the pixel array, and the results are added together. Recall that the dot product computes something like a similarity score: the more the filter and the pixels match, the greater the dot product will be. The resulting scalar is used to populate one activation in the target layer. Thus, the convolutional layer computes a kind of *sliding dot product* with the source activations, which highlights where the filter matches the pixel array.

The number of pixels the filter moves at each step is called the **stride**. Strides can be different in different directions, but we will assume they are the same in all directions. One issue that comes up is the edges, which the filter can't be passed over. To handle this, **padding** can be added to the input volume, in the form of extra zeros around the edges of the input pixel array. This can be done in such a way that the output volume ends up having the same shape as the input volume (more on this below).[5]

In practice, convolutional layers operate on a set of matrices (an input "volume") or a batch of these volumes (a 3d or 4d array; see section 6.9). The output is often also a volume or batch of volumes. How filters work on volumes is discussed below; we start with the simple case where the input and output of the convolutional layer are matrices, which, again, we refer to as a pixel array and feature map.

The idea is illustrated in figures 14.1 and 14.2. In this example all the numbers are included so you can easily check how the computations are done. A 3×3 filter is passed over a pixel array, from left to right and top to bottom. At each moment during this scanning process the dot product is computed between the filter and its receptive field in the source matrix (the part of the image the filter is on top of). Since the input image and the filter are both binary, the dot product simply counts how many places the filter overlaps its

[4]To get a better feel for how this works videos are helpful. A good place to start is with the first animated gif here: `https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks` This video is also great: `https://youtu.be/KuXjwB4LzSA`. Also note that in practice this operation is done in parallel: separate processors handle each multiplication.

[5]This page contains an interactive tool that can be used to understand these concepts: `https://distill.pub/2019/computing-receptive-fields/`.

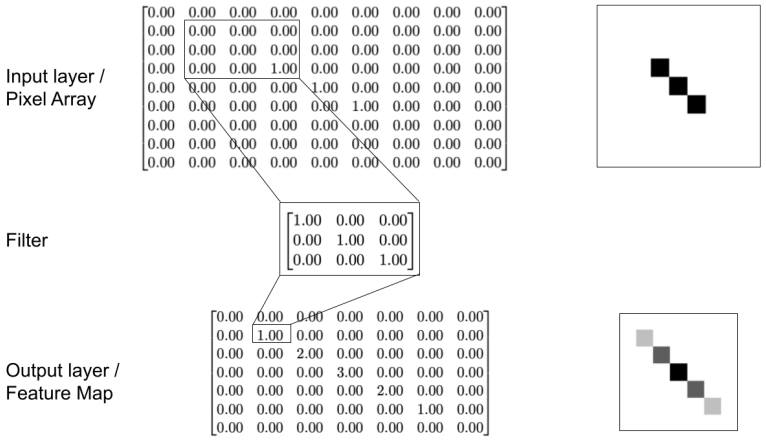


Figure 14.2: Worked example of a convolutional layer. Notice that the feature map has the highest values where the filter matches the pixel array. In the case shown, the filter matches the pixel array in one pixel only. Imagine the filter sliding over the pixel array, and computing dot products (here: numbers of matching 1's), which are used to populate the feature map.

receptive field as it is scanned over the image. In the example shown, it overlaps in one place, in the bottom right of the filter. So that entry in the target layer is populated with a 1. Notice that this filter produces the highest value of 3 only when it is directly on top of the line in the source layer. Try to understand how all the target layer activations are computed. You can also imagine what would happen if the filter or the image were changed.

The output of a convolutional layer is called a **feature map**. In these examples, the filter is an edge detector, that detects edges at a $-45°$ angle, that is, edges shaped like a backslash '\'. In the resulting feature map, notice that the activation is highest when the filter is directly on top of the backslash shape in the happy face, but also produces some activation wherever it is on top of any kind of active pixel. Thus, the feature map gives a sense of where this kind of edge occurs in the input image. [6]

Again, this is totally different from the weight layers we have been studying throughout the book: there are no fixed connections at all. Instead it is like there is a little floating scanner that gets passed over source layer activations to produce output activations.

There is a performance advantage to these convolutional layers, thanks to the weight sharing. All that must be trained is (in the example shown in figure 14.1) $3 \times 3 = 9$ weights, rather than the $81 \times 49 = 3969$ weights that would be required in a fully connected dense layer from the input layer to the feature map. This is a huge performance gain and part of what made it possible with deep learning to train such large networks.

Filters like this edge detector are not programmed. This is a neural network after all, and neural networks are trained, not programmed (section 1.2), usually using a form of gradient descent (section 12.7). Networks trained on vision tasks do not need to be told that edge detectors are useful in early stages of processing. They simply emerge as a structure that supports pattern classification when training a network.

14.2 Applying a Filter to a Volume

So far we have considered an artificial example where the input, filter, and feature map were matrices. However, the power of CNNs is that they can deal well with more complex tensors. In the context of image processing (and in most applications of convolutional networks), the input is a volume, a set of multiple stacked matrices or channels, a rank 3 tensor or 3d array.

[6] The code used to generate figure 14.2 is available online, and can be used to edit a small filter and kernel to get a feel for how they produce a feature map. See: `https://colab.research.google.com/drive/1ywr3z8HRXYNPK-vd34kAATUPVWwwFqQE?usp=sharing`.

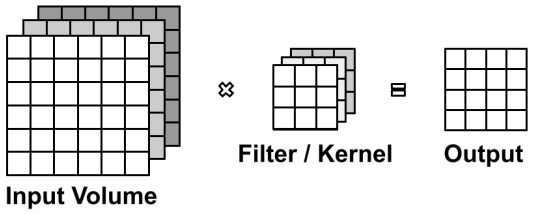


Figure 14.3: Applying a filter to an input volume. On the left is an input volume, in the middle is a filter (which includes three sub-filters) and on the right is the output volume that results from convolving or "combing" the filter over the input. Each sub-filter is convolved with each input channel, and the results are added together at each stage of the convolution.

To get an initial feel for this situation, think of the filter as a set of filters, one "sub-filter" (my term) for each input channel. This set of filters is "combed" across the input volume, in the sense that each sub-filter is convolved with a corresponding input channel, and the results are then added together and used to populate one entry in the output matrix. Figure 14.3 gives a sense of how it works. Convince yourself that the shapes make sense.

But the "combing" idea in figure 14.3 is just a way to ease you into thinking about how convolutions work in practice. Instead of thinking about sets of filters, we can think about filters as volumes which act on other volumes, as in figure 14.4. That is, we can collapse the "sub-filters" of a filter into a single volume, a little Rubik's cube type of object, that convolved with the input as follow. We imagine moving the filter "through" the input volume, top-to-bottom and left-to-right, and at each stage of this operation we dot the filter with the receptive field it is on top of. At each stage of this operation the 27 cuboids in the filter are dotted with 27 components of the input volume, and all 27 products are added together. That gives us one entry in the output, which in this simple example is a matrix.

In general, the depth of the filter (in the sense of tensor dimensions of depth, width, and height: see section 6.9) matches the depth of the input volume. That way the filter can capture information from across all the channels. (This is not always shown in figures and can be quite confusing!) In this example both are of depth 3.[7] Figure 14.5 makes the point with a variety of tensors. In each case the receptive field of a filter at one stage of a convolution is shown "inside" the input volume, and as can be seen in each case it spans the whole depth of the input.

Ignoring padding, the formula for width and height of the output tensor relative to input tensor and filter is:

$$\text{Output Width} = \left(\frac{\text{Input Width} - \text{Filter Width}}{\text{Stride}}\right) + 1$$

$$\text{Output Height} = \left(\frac{\text{Input Height} - \text{Filter Height}}{\text{Stride}}\right) + 1$$

In the example shown in figures 14.3 and 14.4 the input width is 6, the filter width is 3, and the stride (the amount it is moved for each step of a convolution) is 1 so the output width is $((6-3)/1+1) = 4$. Same for the height.

[7] This is not a necessary feature (filters with fewer channels than the input are possible and sometimes even useful), but it is pretty standard.

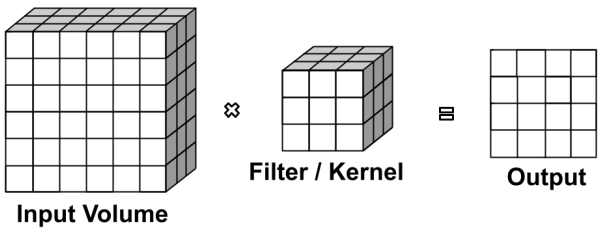


Figure 14.4: The same situation as in figure 14.3 but represented using volumes. This is a better way to think about convolutional layers, because it generalizes to the more complex situations discussed below. Try to image the filter being passed "through" the input volume, and being dotted with its receptive field at each stage of the operation, producing one number in the output.

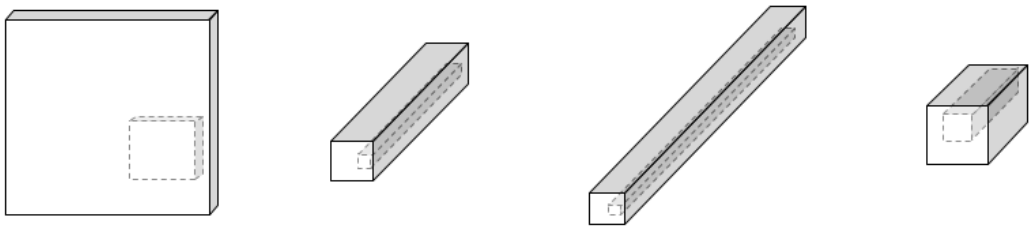


Figure 14.5: Illustration of how filters usually have a depth that matches the input volume they are passed over. Shown are the receptive fields of four filters, each of which is shown at one stage of a convolutional pass. Alternatively, think of these panels as showing filters placed "inside" their corresponding input volumes and moved through them.

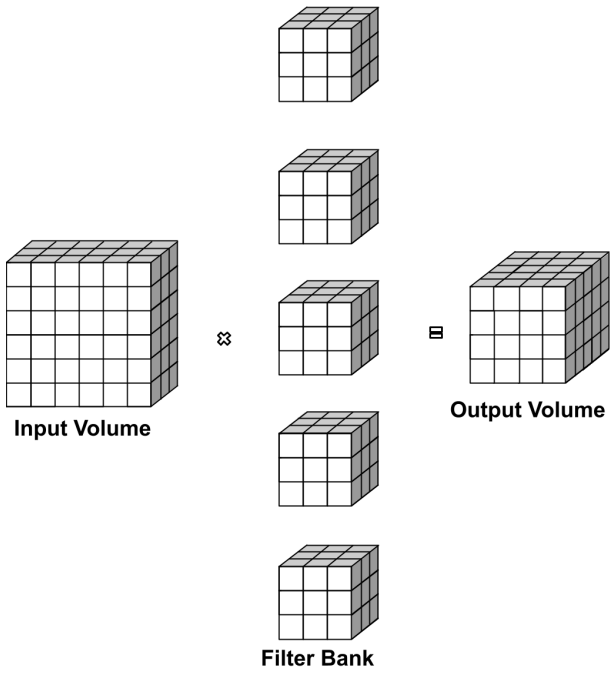


Figure 14.6: Result of applying a filter bank of 5 $3 \times 3 \times 3$ filters to a $6 \times 6 \times 3$ input volume with a stride of 1 to produce an output volume that is $5 \times 4 \times 4$. The output depth of 5 comes from the number of filters in the bank; the output height and width come from the formula in the main text.

The depth of the output shape corresponds to the number of filters we have (more on this soon), and so far we have just considered a single filter, a single "Rubik's cube". So the output shape so far in our example is 1.

So the overall shape of the output is 4×4, a four-by-four matrix.

14.3 Filter Banks (Representational Width)

In general a convolutional layer will involve multiple filters, a **filter bank**, each of which produces a separate feature map. Figure 14.6 illustrates the idea. Each filter produces its own output, and the results are concatenated into a new output volume. That is, we just repeat the process discussed in figure 14.4 but do it separately for each filter. In this example, we get a bunch of matrix outputs, and concatenate them into one output volume. That is why we said at the end of the last section that the depth of an output volume just corresponds to the number of filters we use.

Each feature map in a filter bank learns differently via gradient descent to represent the input. Thus we get different ways of representing the input. This generalizes the concept of the **representational width** of a layer (section 1.1) to CNNs. Just as more nodes makes a regular node layer of a feed-forward network more powerful, so too do more filters make a convolutional layer more powerful. Rather than learning just one way to represent inputs, a bank of filters can learn multiple complementary ways to represent inputs.

The representations are not programmed in but are learned via gradient descent. This is remarkable. We did not tell the network we want it to learn to respond to edges. All we focus on in training a network is inputs and outputs, using a labeled data set. Training data for these networks might involve images paired with numbers. The network is given nothing else but these training examples: if you see this picture, it's a 2; this picture is an 8, etc. Then the network adjusts all its parameters (all the weights in its convolutional layers), in such a way as to reduce error.

Edge detectors were learned by training, not programmed in. This is how primary visual cortex reacts to images, so this is a nice model of the brain, hence an example of computational neuroscience. However, these

networks are best known for the engineering benefits, since they are powerful pattern recognition systems. This is also an old connectionist theme. In Nettalk (section 15), phonetic categories like consonant and vowel were not programmed in, but emerged with training. With simple recurrent networks (section 16.6), grammatical categories like verb and noun were not programmed in but emerged with training.

14.4 Multiple Convolutional Layers (Representational Depth)

Now we stack these things on top of each other, add a new kind of layer, and also combine in old school neural networks too. A deep network assembles these various ideas together.

The idea here is to get back to representational structure, here **representational depth**. Recall from the history chapter that the concept of layered feature detection goes back to Oliver Selfridge and his "pandemonium" model, which at the time just speculated that in seeing letters a hierarchy of "demons" pass messages along: from edge demons to curve edges and finally to the output layer's "B demon" (see figure 3.8 in chapter 3). This allows the network to learn to identify not just simple features like edges or curves, but also *features of features*, like combinations of curves which make more complex shapes, and then combinations of these shapes. Deep networks max out this old idea, in such a way that we can actually see what their receptive fields are.[8]

There are several kinds of volume-to-volume layer in a convolutional network. One is a filter bank, as discussed above. The other is a **pooling layer**, which reduces the amount of information passing through the network without altering its basic structure. Finally, we can **flatten** a layer, which takes us from the convolutional layers back to a traditional feed-forward node and weight layers. Flattening a layer involves taking all the values in a tensor and simply arranging them into a single vector. A flattened 3x3 matrix, for example, would be a vector with 9 components. See figure 14.8. Pooling layers and flattening are discussed in this section.

14.4.1 Pooling

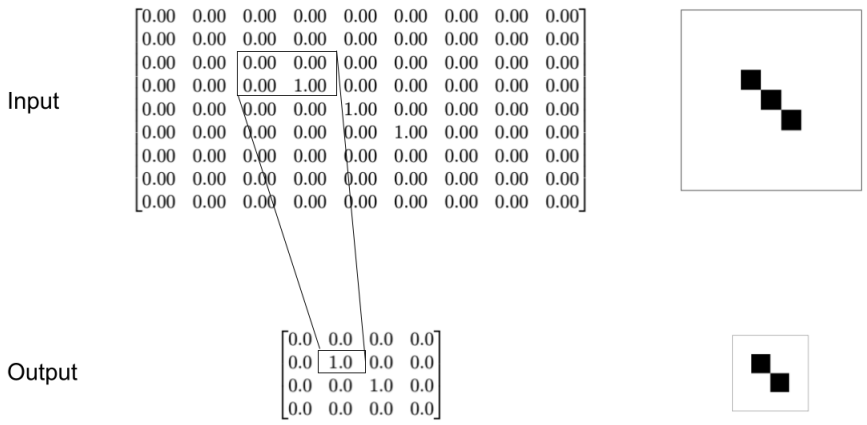


Figure 14.7: Max pooling operation being applied to a matrix input, producing a smaller output that retains the same core information. The operator is shown by a square which shows the pool size. In this example the stride is 2, so it is scanned across the input in increments of 2. The maximum value in the pool is used to populate the output as the window is scanned across. Note that there is no depth here. Pooling operations are generally applied to 2d arrays.

A **pooling layer** is useful when tensors start getting too big. The idea is to go from bigger to smaller tensors, to keep these more computationally manageable. But we must do it in a way that preserves important

[8]The receptive fields can be quite strange and even disturbing. See `https://distill.pub/2017/feature-visualization/` for some striking demonstrations.

information. Here again we pass a window over the input and slide it left-to-right and top-to-bottom, but instead of multiplying or dotting, we apply one of two common pooling operations:

Max pooling In each window, the maximum number is taken and written to an output matrix.

Average pooling The average value across a window is written to an output matrix.

The result of this operation is an output that is smaller than the input. These are sometimes called subsampling or downsampling methods. They are valuable in that they reduce the overall size of the network while preserving important structures.

The window passed over the input volume has a pool size, a width and height. It does not usually have depth because we do not want to combine the results of several feature maps. We pass the pooling operation over every feature map separately. The idea comes from the last section, where we saw that each filter learns to represent the inputs in a new way. When we downsample, we don't want to lose those differences. However, a new convolutional layer can learn to combine information.

However we can use a stride when we downsample. So the formula is the same as above when computing output shape from input shape.

An example is shown in figure 14.7. The pool is $2x2$ and the stride is 2, so the window is passed over the input 2 at a time (notice that the right-most column gets left out in this case; a stride of 1 would fix that, or some padding).

For these operations we use the same equations as above, but with pool width and height. There is no depth because pooling does not pool across channels.

$$\text{Output Width} = \left(\frac{\text{Input Width} - \text{Pool Window Width}}{\text{Pool Stride}}\right) + 1$$
$$\text{Output Height} = \left(\frac{\text{Input Height} - \text{Pool Window Height}}{\text{Pool Stride}}\right) + 1$$

14.4.2 Flattening and Dense Layers

Another thing we can do, usually in the final layers of a deep CNN, is to "flatten" a volume back into a vector. If we have a 5x5 matrix, for example, we can flatten it into an activation vector for a node layer with 25 nodes. If we have a 4x5x10 volume, this becomes the activation vector for a node layer with 200 nodes. Note that the shape of the flattened vector is just the product of the shape values for the larger tensor.

In this way we can convert whatever high-rank tensor we have into a vector, and thereby get back into the world of traditional neural networks focused on in much of the book. Then we just do things as we've done before. We can "couple" our complex convolutional layers into a standard feed-forward network. In this context, weight layers are sometimes called "fully-connected" or "dense" weight layers to contrast them with convolutional weight layers.

The overall idea is to start with layers that learn these complex features, then to compress these representations with subsampling, and finally to present the results to the final layers, which are the kinds of standard feedforward networks discussed in earlier chapters (for example, chapter 12).

14.5 Applications of Convolutional Networks

Convolutional neural networks originate in computational neuroscience models of vision that were developed in the 1970s and 1980s [35]. These ideas were later used to engineer pattern recognition networks. A famous early application was recognizing zip codes written on envelopes [69]. As convolutional networks became mainstream based on technical improvements (big data, GPU and hardware acceleration, better architectures and training algorithms; see section 3.7), scientists began using them, for example, to model the response profile of neurons in the visual system (recall the discussion of figure 4.7 in chapter 4).

A significant application of CNNs in cognitive science is illustrated by the pioneering work of Yamins and DiCarlo [130]. They used CNNs to study how the brain decodes sensory information through a cascade of simple neural operations. To address this problem, they trained a neural network to recognize objects

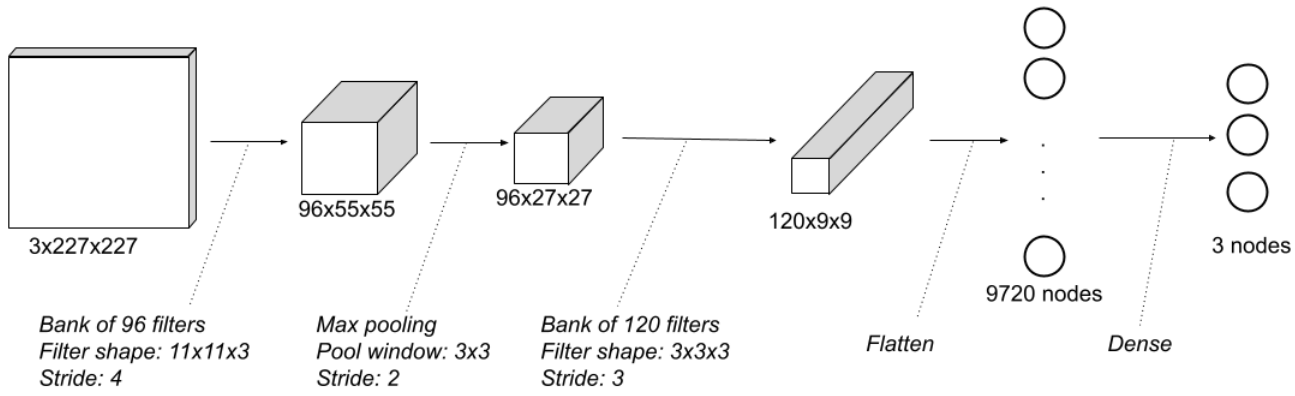


Figure 14.8: Sample deep network showing convolutional layers and pooling layers. You should try to apply the shape equations to convince yourself that the sizes make sense.

and living beings, and then tested whether the resulting network could predict neural responses in the visual cortex. They developed a hierarchical convolutional neural network (HCNN) with seven layers, training it to recognize objects in a 3D environment using different 2D views. They then presented the same images to monkeys and recorded the neural activities in various sub-regions of their visual cortex—specifically, in the order of signal processing: V1, V2, V3, V4, and the inferior temporal cortex (IT). The results were remarkable. The HCNN's layers achieved the best predictions of neural activity in terms of explained variance, outperforming models specifically trained for this purpose. Moreover, the initial layers of the HCNN more accurately predicted activations in the early visual cortex regions (V1-V4), while the later layers excelled in predicting the activity in the IT region.

The idea is also relevant to connectionism and computational cognitive neuroscience. The significance of these networks for psychology is still in its infancy, but early results are promising [134, 102].

Chapter 15

Internal Representations in Neural Networks

Jeff Yoshimi

In this chapter we discuss internal representations that develop in neural networks, which is a key theme when considering neural networks in *cognitive science*, which is the topic of this book, after all. We have seen that supervised learning methods are not generally taken to be neurally realistic in their mechanistic details, since there is no obvious way for error to be sent "backward" through the dendrite of neurons (section 13.4).[1] However, even if backprop is not realistic, it leads to the development of internal representations that are similar to representations humans use.[2]

A representation in a neural network is a state (in the sense of "state" discussed in 10, a pattern of activation over a set of nodes) that reliably occurs in response to some external input. For example, whatever activations occur in a neural network when an apple is present can be thought of as being a representation of apple. Note that a representation can be spread across many nodes, so that we have the input representation and hidden unit representations, for example. They can also have different levels of generality, and this can be unpacked in terms of relations of points in a state space. For example, different apples might all produce slightly different points in state space that are near each other, and different fruits might do the same, etc, producing a nested structure. The details of what a representation are–and whether they are even valuable to posit–are highly contested and controversial, but we simply assume this simple definition here.[3]

In this chapter a range of examples of representations are considered, all of which show how under the pressure of gradient descent, networks trained by backprop and its variants produce psychological realistic internal representations. Remember this was a theme in the introduction (chapter 1). Neural networks are trained, not programmed. They develop useful abilities and, as we will see, internal representations, without having anything programmed in. They develop these abilities and representations simply by learning to perform tasks from data. This is what got the connectionists excited in the 1980s (section 3.6). Remember

[1] However, some circuits have been identified in the brain that may implement error-based supervised learning, *e.g.* climbing fibers in the cerebellum (see chapter 4). Error based learning that is similar in some ways to backprop is also emphasized in predictive coding accounts of the brain and predictive processing accounts of cognition. For a recent discussion of neural realism of backprop, see [124].

[2] That is, it is assumed that, even if backprop does not happen in most parts of the brain, it can still be used as a device to discover the kinds of representations that the brain finds. How the brain develops these representations is still a mystery, but backprop let's us at least see what those representations might look like and what function they might serve. The idea that backprop could be used to identify valid neural representations even though it is not realistic was discussed as far back as Zipser's work on system identification, if not earlier. See [133].

[3] Doubts about representations are associated with embodied cognition, and examples, such as Braitenberg vehicles, which show that one can have intelligent behaviors in a body-world system without there being much value to interpreting internal states. Those who allow representations often add other components to the definition of a representation. For example, it should be possible to activate a representation agent state absent its normal cause (representations are "detachable" rather than "stimulus bound"), which allows for cases of mis-representation, planning in the absence of the object, and so on. It is generally assumed that representations are used by the agent to guide its behavior. Many assume that computational processes must transform representations into other representations, to support adaptive behaviors

(chapter 2), connectionism is a use of neural networks as cognitive models, which capture psychological and behavioral features of the mind, without concern for neural realism. After the deep learning revolution (section 3.7), the idea has come back, but with plenty of interest in neural realism as well (computational cognitive neuroscience). In general, the idea is that neural networks trained by algorithms like backprop are relevant to psychology and neuroscience, *even if backprop itself is not neurally realistic.*

This approach also builds on the visual language we have been developing for understanding the mind, in terms of points or vectors in high dimensional spaces. In section 13.5 we saw that multilayer feed-forward networks trained by supervised learning methods (*e.g.* backprop) will re-map an input space in order to make a linearly inseparable problem separable. That was an abstract example, but it also applies to more everyday cases. You should be able to interpret pictures like the one in figure 15.1. Points correspond to representations. In this example, black is one category, and white is another. It is similar to the XOR example. Note that the representations in the input space are not linearly separable but that they are in the hidden unit space.

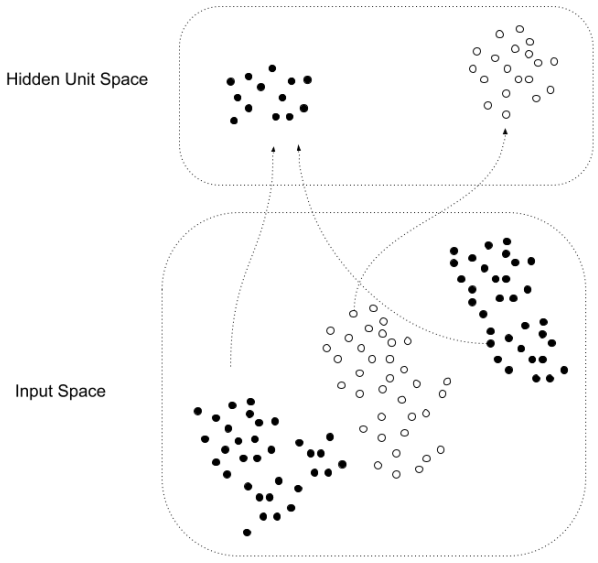


Figure 15.1: Remapping a linearly inseparable pair of clusters in the input space to a separable cluster in the hidden unit space. In this graph, colors distinguish different output values associated with states. Each dot corresponds to a a different state, a different representation. Input and hidden unit representations are shown.

One way to understand how these representations work is that they involve remapping and recombining the input space of a network in various ways. In some cases, as in the one in the figure, separate clusters get merged at another layer. We also saw this in the the discussion of Selfridge in chapter 3, how different layers will combine representations of earlier layers to produce complex features. The same idea also comes up in the "stacked" layers of a deep network (chapter 14).

In this chapter a series of examples are given, some of which are elaborations of discussions in other chapters.

15.1 Net Talk

NETtalk (1987) [111] was a model of reading English words aloud, created by Terrence Sejnowski and Charles Rosenberg in 1987. The network was trained to speak aloud. The network is a three layer feed-forward network with the structure shown in figure 15.2. The input layer codes for a moving window of letters, one of which is taken to be the current input and the rest of which provide the network with information about

neighboring letters.[4] The output layer contains 26 units encoding different features of phonemes including voicing and vowel height. The hidden layer has 80 hidden units. It was presented with written letters in English and was trained to pronounce those letters. There are about 18,000 weights and biases in the network. The network was trained on a corpus of 1000 common words using backprop. It took several days to get the error to a reasonable level using their circa 1987 computer. A slightly modified version of the network could generalize from these 1000 training samples to pronouncing 90 percent of the 20,000 words in a standard English dictionary correctly. The network was shown to perform well with noisy input and to gracefully degrade [111].

Here is the key point: the hidden units learned to produce a complete separation of consonants and vowels, even though the network was not told about consonants or vowels. That is, when the central input was a consonant, the hidden unit activation was in one part of the hidden unit space; when it was a vowel it was in another region. Different vowels in the context of different letters produced different points but as a group they were all near each other. Similarly for consonants. The network was not told about the difference between vowels and consonants–it simply learned these categories while it was trained on the pronunciation task [111]. This showed how in learning a mapping from sensory inputs to motor outputs, psychologically meaningful categories could take form in a network's hidden unit space.

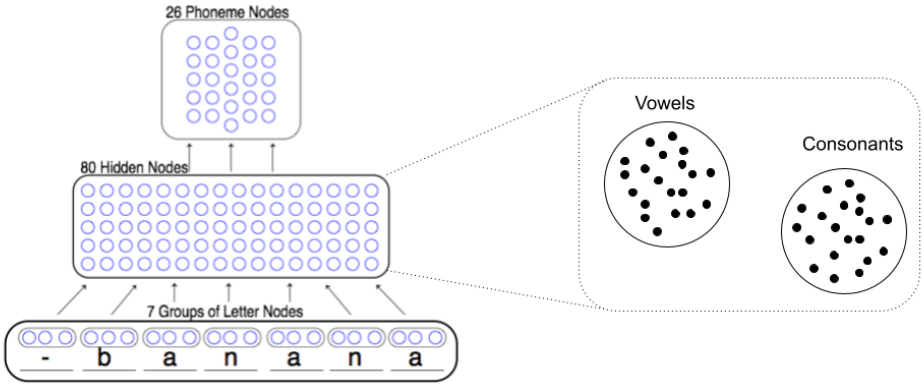


Figure 15.2: NETtalk models converting written words to sounds, *i.e.* reading aloud. Each letter group has 27 nodes (not the 3 shown). When trained using backprop, it developed internal representations of vowels and consonants, without having been told about them. The idea is schematically shown here. The current central input is an "n", a consonant, and so the hidden-unit state would correspond to one of the points in the consonant cluster.

On striking feature of this example is that the sounds the networks produced could be artificially synthesized and played back. I think this had quite an impact on the community at the time, since you could "hear" the network learn (listen for yourself at `https://www.youtube.com/watch?v=gakJlr3GecE`). Initially the network produced a babble of sounds, like a baby. But as it came to learn the task it got better and better, until it produced somewhat fluent speech [111]. This made it quite palpable what was going on: the network was slowly getting better, in a way that sounded vaguely like a child learning to read. I encourage you to listen to the (somewhat creepy) process. So again, even if backprop is not neurally plausible, it seems to be doing something *like* what the brain does, and it does so by developing psychologically realistic representations of vowels and constants.

15.2 Elman's Prediction Networks

Another early connectionist analysis of internal representations occurred in the context of Jeff Elman's simple recurrent networks. The architecture and training of these networks is discussed in detail in section 16.6. Here we simply reiterate some of his analysis of the internal representation they devloped. After being trained

[4]Each letter is coded by a 29-dimensional vector (involving one-hot, one-of-29 representations of particular letters, with three additional nodes representing punctuation and word boundaries).

to predict the next word in a simple text corpus (the same "auto-regressive" task used in transformers like GPT; see chapter 17), the network's internal states were studied. After training, the average response of the network to different words was tested, and it was found that similar words were near each other in the hidden unit space. This can be seen in figure 15.3.[5]

Words that had similar grammatical roles were near each other in hidden unit space, and words that were semantically similar were near each other. This was a big deal. They learned grammatical and semantic categories in its hidden layer without being told anything about grammar or semantics. Grammatical categories they developed include noun and verb. Within these categories, semantic sub-categories developed, including animal, human, food, and breakable. None of these categories were directly programmed in to the network: it simply learned these categories in the process of solving the input-output task it was given. The network developed meaningful representations of grammatical and semantic categories based on features of the input stream.

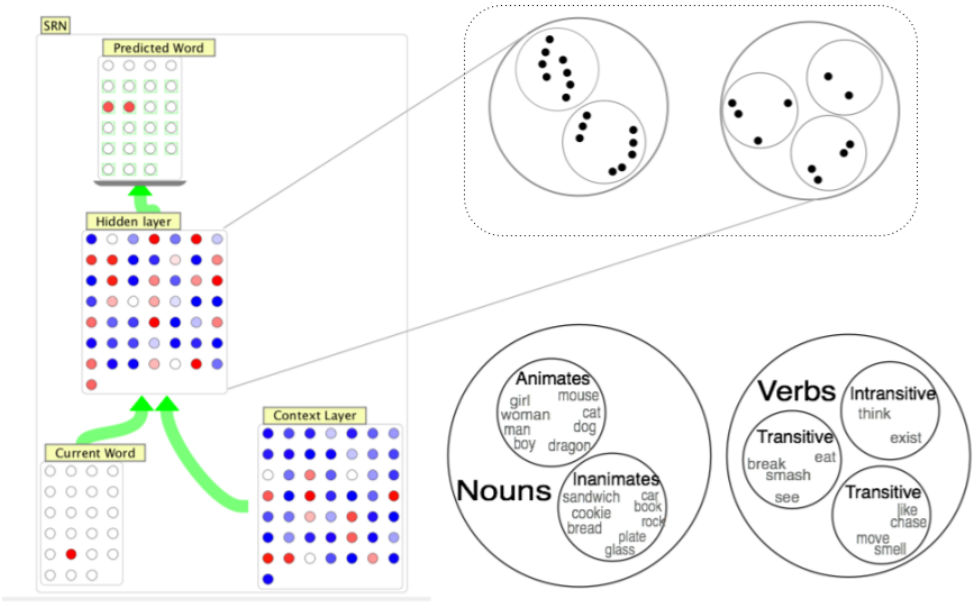


Figure 15.3: Categories in the input space of an Elman SRN. Points in the input space representing words are clustered nearby each other, as is shown. The image on the bottom-right shows what words the dots correspond to. The example shown is fabricated, but is inspired by the actual graphs in the original paper.

15.3 Deep Vision Networks

So far we have focused on cases where we could visualize clusters in a hidden unit space. But there are other ways to show that a network has developed realistic internal representations, which begin to take us from connectionism to computational neuroscience and computational cognitive neuroscience.

As discussed in chapter 3, after the deep learning revolution, neural networks came back in fashion, and scientists once again began turning to these as models of brain and cognition (for example, see [131, 130]). Deep networks (chapter 14) trained using backprop to recognize numbers and other objects in images produce activations that match neural responses in such brain areas as V1, V2, and IT, which are discussed in chapter 4. V1 and V2 are part of early visual processing, with V1 containing edge detectors. IT is part of the ventral stream (also discussed in chapter 4) which is involved in object recognition. The model contains separate layers corresponding to three parts of IT: an anterior, central, and posterior part. The correspondence

[5]There are two categories of transitive verbs because some verbs (like "chase") are always transitive in the data (that is, taking a direct object), while others (like "break") were sometimes transitive, sometimes not.

between layers of the deep net and parts of the brain are shown by green dashed lines in figure 15.4. The similarity of the response of one of the last layers of the deep network and IT neural responses are shown in figure 15.5. The model outperforms computational neuroscience models specially designed to model the visual systems of the brain. Simply under the pressure of the gradient descent algorithm, they develop realistic representations.

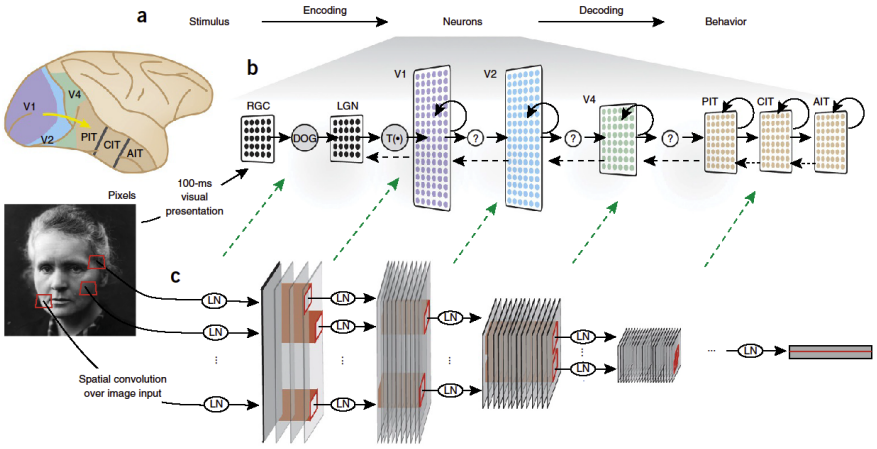


Figure 15.4: Architecture of a deep network whose neural responses were compared with neural responses in corresponding regions of the brain. These regions include primary and secondary visual processing areas and several areas of the ventral stream.

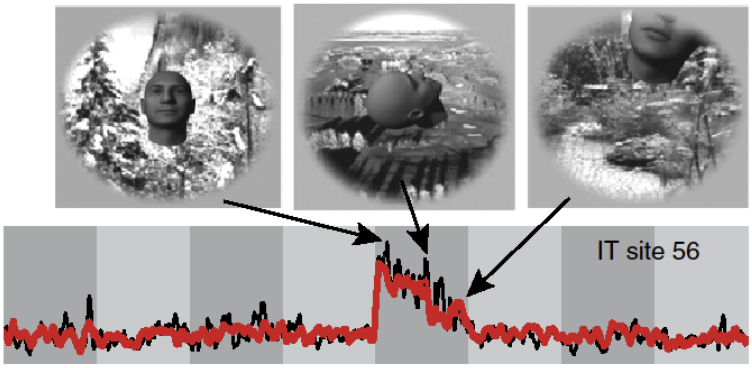


Figure 15.5: Comparison of node activations in a hidden layer of the deep net (black) shown in figure 15.4, with actual neural responses in brain area IT (red). Each point on the x-axis shows how the biological and artificial neural network respond to a particular image. Three sample images are shown above. As can be seen, images with a head in them produce a higher response both in the neural network and in the brain.

15.4 Other Examples

There are many other examples of psychologically and neurally realistic internal representations in artificial neural networks, including examples based on other more realistic learning algorithms. For example, the fixed point attractors of a Hopfield network discussed in chapter 11, the representations that develop in

competitive nets and SOM networks, and psychological elaborations of BERT (based on the same transformer architecture as GPT) discussed in chapter 17, all have properties that shed light on human cognition.

Chapter 16

Supervised Recurrent Networks

Jeff Yoshimi

In chapter 1, we introduced the distinction between feed-forward and recurrent networks. Feed-forward networks have historically been a focus of research activity and applications. They are easy to analyze as function approximators or pattern associators which associate vectors with vectors, and powerful methods like backprop (and the variants used to train deep networks) have emerged to allow them to approximate arbitrary functions (chapter 13). They also produce interesting representations in their hidden layers. As a result they have often dominated the conversation.

But recurrent networks have many advantages. Rather than just statically producing a single output for each input, recurrent networks *process* information, producing dynamically changing patterns of activity over time. Like the brain and mind, they are dynamical systems (chapter 10). Every network in the human brain is recurrently connected and will produce patterns of activity when stimulated. They are also psychologically plausible. In chapter 1, we saw that IAC networks like the Jets and Sharks network can respond to questions by a process of spreading activation in a recurrent connectionist network. In chapter 11, we saw that recurrent networks can be trained using unsupervised methods (like the Hebb rule) to produce fixed point attractors that correspond to memories. However, unlike the kinds of more purely recurrent networks discussed in chapter 10, where we would randomize a network and see what state it settled into, here we consider recurrent networks that retain features of feed-forward networks. They are generally layer-to-layer networks, where some of the layers are recurrently connected. Thus we retain the idea that there is an input layer and output layer where we want to train the network to produce certain outputs in response to certain inputs. We just add dynamics, so that the outputs can unfold automatically even if inputs are withheld.[1]

Recurrent neural networks or "RNNs" have a long history. Not long after backprop was discovered, people figured out how to apply these techniques to RNNs, often by converting an RNN into a feed-forward network using special tricks. The initial results were promising, but ran into various technical hurdles, as we will see. However, in recent years, techniques developed to train deep feed forward networks (chapter 14) have been adapted to recurrent networks. The results are both useful and amazing. Any time you "google" something, or type in a partial sentence on your cell phone, these RNNs (or closely related algorithms) are at work in the background suggesting text completions. They also automatically classify online movies, help convert speech to text, produce automated summaries of documents, translate between languages, and even create synthetic music or (more disturbingly) convincing fake news articles.

In 2022 Open AI introduced ChatGPT, which marked a major shift in the history of neural networks and AI. In earlier drafts of this book, the results covered in this chapter (like generated fake news articles) were somewhat remarkable. But after ChatGPT was released, these ideas became familiar to us all. The transformer architecture and other innovations used to propel these ideas forward (in a way that harnesses ideas associated with supervised recurrent networks in the context of complex feedforward networks) are discussed in chapter 17.

[1] Other approaches to recurrent networks (reservoir networks like echo state machines), that are more geared towards computational neuroscience, are discussed in chapter 19.

For the first part of this chapter we develop the basic theory of supervised recurrent networks, starting with an overview of types of applications, and then discussion of an important historical class of model: the simple recurrent network or SRN. This model basically uses some tricks which make it possible to apply classical backprop techniques to recurrent networks. We then describe backprop through time, which also uses tricks to make it possible to use backprop to train networks to produce specific dynamical behaviors in reaction to inputs. These sections give us a sense of how supervised learning can be applied to recurrent networks.

Supervised recurrent networks have been especially useful in the domain of natural language processing (NLP), where words and other linguistic items are represented as vectors, via "word embeddings" (this topic was briefly addressed in section 7.3). Note that for much of this chapter I refer to processing of words in a sentence, since that is a simple and easy case to think about, though sentences can be parsed into other types of linguistic units as well, such as parts of words. In machine learning these linguistic units are more generally referred to as "tokens" and the process of breaking a document up into these tokens is known as "tokenization."

16.1 Types of Supervised Recurrent Networks

To start, let's think about ways these kinds of network can be useful. Figure 16.1 shows some ways you can use a recurrent network trained using supervised methods like backprop through time.[2]:

Vector-to-sequence (one to many): Train a network to produce a sequence of desired input vectors from a single input vector. Example: train a network to produce a song or speech from an initial prompt.

Sequence-to-vector (many to one): Train a network to respond in a specific way after a sequence of input vectors has been presented. Example: train a network to classify a video clip. The video input runs for a while and at the end a classification is output.

Sequence-to-sequence (many to many): Train a network to respond to a sequence of inputs with a sequence of outputs. Example: train a network to translate a sequence of sentences in English with a sequence of sentences in German.

Even though this is framed in terms of single vectors and sequences of vectors, the boundaries between these cases can be fuzzy: a sequence-to-vector model, for example, might really involve a small sequence of vectors as input (like a brief text prompt) and a much longer sequence of output vectors. Thus, one might also think of these as involving: a long response to a short input; a short response to a long input; and equal-length responses.

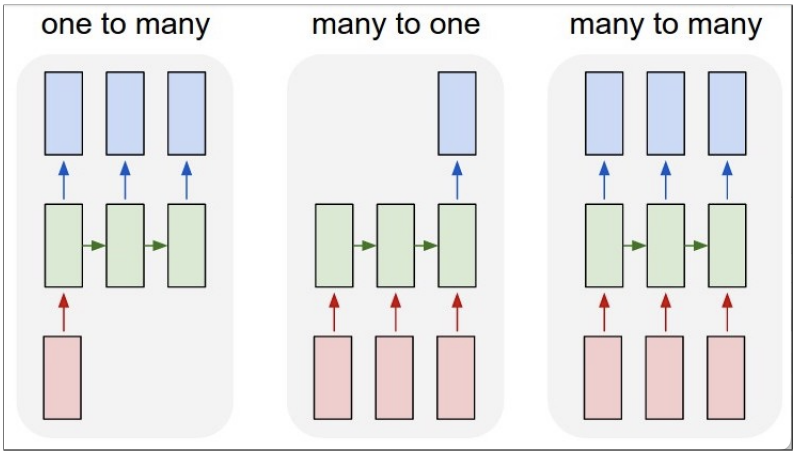


Figure 16.1: Different types of sequence learning possible with backprop through time.

[2]In these examples absent inputs are just zero vectors.

These ideas have links to dynamical systems theory (chapter 10). In the single input vector (one to many) case we are training the network to produce an orbit in the output space relative to an initial condition triggered by the input. Variations on this architecture and this method can be used to train a network to produce a whole phase portrait. In fact, there are theorems which show that recurrent networks can in principle produce any trajectory of any dynamical system [36].[3]

There are clearly many applications here. Again, many are in the domain of natural language: chat bots, sentiment analysis, text summarization, speech recognition, machine translations. But there are other applications: time series forecasting, video classification, video captioning, music generation, music recognition, video acton recognition (identifying objects in a video or determining what is being done), etc (for each case, ask which category it fits best in). As we will see, the technology is rapidly changing in this area, since it has so many applications, but cognitive science and neuroscience are paying close attention, and this material continues to be highly relevant to understanding the human mind and brain.

16.2 Simple Recurrent Networks

To begin to understand how these networks work, and how they were used in cognitive science, we can consider an old class of model, the **simple recurrent network** or SRN, which was developed by Jeff Elman [25], a member the original PDP research group (in fact, they are also sometimes called "Elman networks"). SRNs are important because (1) they show what the basic approach to training recurrent networks is, and (2) because they were used by connectionists to demonstrate how grammars could be learned by a network.

The structure of an SRN is shown in Fig. 16.2. It is basically a regular 3-layer feed-forward network trained using backpropagation, with some special machinery for processing temporal context.The special feature is the "last hidden state" portion of the input layer, which is always set to the hidden layer activation vector of last time step (in the first time step it is usually just set to the zero vector).[4] It is a "copy-back" of the hidden layer. Otherwise it is like another part of the input layer, fully connected to the hidden layer. Thus at any time the full input to the network is the current input, *plus some temporal context.* It's a bit like someone saying "Good times!". At the moment you hear them say "times!" you have some memory of them having just said "Good". You hear "times!" in the context of "Good". This allows you to distinguish "good times" from "bad times" from "crazy times", etc.[5,6]

SRNs are also trained in a special way. They use a training dataset, like the ones discussed in chapter 12, and shown below. But unlike a normal training dataset, the rows of an SRN's dataset must be presented in a specific temporal order. As each input vector is presented to the network, the output is computed based on that input vector, *and* on the hidden layer vector from the last time step. Then backprop is used in the usual way to update all the weights of the network. Table 16.1 shows an example of a training set for a recurrent network. To emphasize the importance of temporal order, a column for time has been added.

The dataset trains an SRN on a one-step prediction problem, which is often what SRNs are used for. In machine learning contexts this type of training is also common; there they are referred to as "auto-regression" tasks. The network learns to predict the next item in a sequence based on what items have occurred before. A nice thing about this kind of task is that there is no need for "labeled data." Any string of words or tokens is enough to train a network, since the target at any time is just the next item in a sequence.

In the toy example being considered here, the network is being trained on a "bouncing one" pattern. For example, if we enter $(1, 0, 0)$, the SRN should predict that $(0, 1, 0)$ comes next. But notice that the vector

[3] Even more generally a recurrent network can reproduce any Turing Machine, and hence *any computational system.* They are "Turing Complete"; see [50] section 15.5; also see `http://binds.cs.umass.edu/papers/1995_Siegelmann_Science.pdf`. These results are comparable to the universal approximation theorem for feed-forward networks noted in chapter 13

[4] As McClelland says,"The beauty of the SRN is its simplicity. In fact, it is really just a three-layer, feed-forward back propagation network. The only proviso is that one of the two parts of the input to the network is the pattern of activation over the network's own hidden units at the previous time step" `https://web.stanford.edu/group/pdplab/pdphandbook/handbookch8.html`.

[5] This idea occurs in philosophy in the work of Edmund Husserl and others who claimed that human experience essentially involves "time consciousness", which in turn includes an awareness of what has just-passed and what is about to come (and note that SRNs are usually trained to predict one step in the future). See `https://plato.stanford.edu/entries/consciousness-temporal/`.

[6] But note it's not the past input that is remembered, it's the past hidden state. That hidden state is influenced by the past input, *and* earlier hidden states. Thus there is a recursive relationship here that allows the temporal influence to extend arbitrarily far back in the past, though the influence is strongest in the recent past.

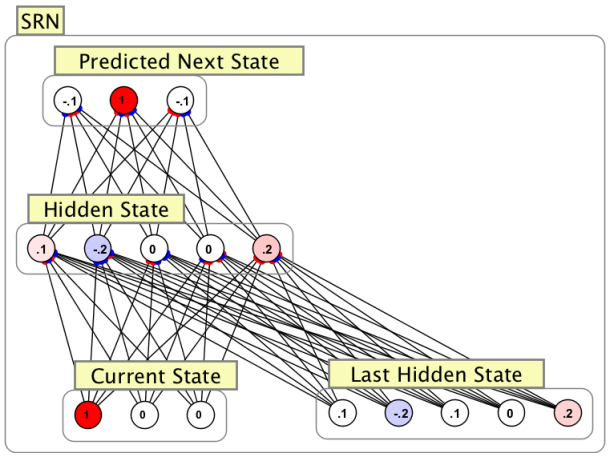


Figure 16.2: A simple recurrent network.

time	inputs			targets		
1	1	0	0	0	1	0
2	0	1	0	0	0	1
3	0	0	1	0	1	0
4	0	1	0	1	0	0
5	1	0	0	0	1	0

Table 16.1: The user provided training set for an SRN. We say what outputs we want to occur, in what order, given a time-ordered sequence of inputs. Note there is a puzzle: the state $(0, 1, 0)$ occurs twice, with two different outputs, and thus seems to pose a problem for training.

$(0, 1, 0)$ is ambiguous. It will predict *different* outputs depending on when we present it. It predicts $(1, 0, 0)$ after $(0, 0, 1)$, but $(0, 0, 1)$ after $(1, 0, 0)$. How can the network do this? The answer is: by using the context layer. The last hidden state after seeing $(1, 0, 0)$ is different then it is after seeing $(0, 0, 1)$. This allows the network to differentiate the same input, $(0, 1, 0)$, in different temporal contexts.

In fact, behind the scenes, what is happening is that the network uses regular backprop, but with a special training set. The real training set, "under the hood", is shown in table 16.2. The network learns to associate inputs with outputs, *in the context of specific hidden layer states*. Notice that the last hidden unit state at time 2 is different from the last hidden unit state at time 4. This allows the network to solve the problem.

We can train these networks on arbitrarily long sequences, like all the sentences in a document, or all the images in a movie, or all the sounds in a musical piece, assuming of course that the sentences, images,

time	inputs			last hidden					targets		
1	1	0	0	.5	.5	.5	.5	.5	0	1	0
2	0	1	0	0.9	0.9	0.9	0.2	0.9	0	0	1
3	0	0	1	0.4	0.8	0.8	0.9	-0.2	0	1	0
4	0	1	0	0.9	0.9	0.9	-0.2	0.9	1	0	0
5	1	0	0	0.9	0.3	0.3	0.9	-0.5	0	1	0

Table 16.2: The actual training set used "under the hood" by the SRN. The inputs are external inputs together with the last hidden state of the network, which reflects recurrent dynamic processing. This allows the network to disambiguate the $(0, 1, 0)$, which is different in its two temporal contexts, where the last hidden state is different.

or sounds have been converted into vectors (see the discussion of feature encoding in chapter 7). As we will see, the hidden layer can then be analyzed for the patterns it discovers and the sequences of patterns it goes through in time. In psychology, SRN's have been especially useful at studying the development of linguistic categories in networks that learn to predict the next sound in a speech stream, or the next word in a passage of text.

16.3 Backpropagation Through Time

A more general framework for training recurrent networks (which can be thought of as generalizing the SRN to include arbitrarily many time steps in the past) is by using "backpropagation through time" [123].[7] As with the SRN, we start with a simple three-layer feed-forward network, but instead of a "copy back" layer, we use a recurrent layer of weights from the hidden layer back to itself. Like the SRN, we have a training dataset that involves time ordered input-target pairs. In order to train the network, we "unroll" the network so that all the inputs can be put in the network at the same time. It's as if you take the original network, copy and paste it a bunch of times (once for each row of your training set), and then replace the recurrent weights from the hidden layer back to itself with lateral weights *between* the copy-pasted hidden layers (see figure 16.3).

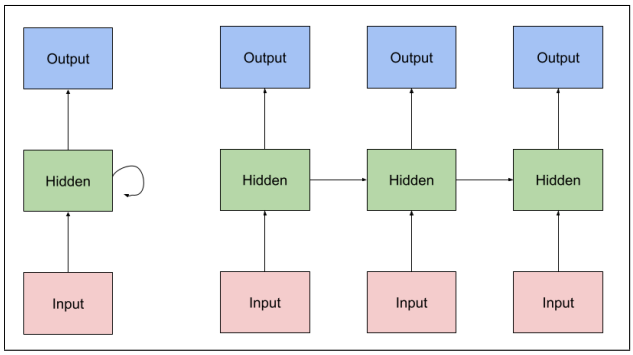


Figure 16.3: Schematic of backprop through time. The actual network that we are training is on the left: a feed-forward network with a recurrent weight layer in the middle. The unrolled network is on the right. This network can learn a sequence of three input-target pairs. We train the unrolled network to produce target values in response to current inputs *and* to previous hidden layer states. When training is done, the changes to the weight matrices (the arrows) are added together and so we have "rolled the network back up" to the network on the left.

Each of the unrolled networks is then responsible for a specific moment in time. To train the network, we put together all the inputs in the dataset into one long input, one for each of the unrolled networks. Then we train the whole unrolled network to produce the corresponding sequence of outputs.[8] But you are not just training the usual weights from input-to-hidden layer and from hidden-to-output layer. You are also training the hidden-to-hidden weights in the middle, that laterally connect the hidden layers of the unrolled network to each other. Note that the unrolled network is really just a feed-forward network, with some extra lateral weights. So this is ultimately a trick to use feed-forward methods on a recurrent network! When we are done training this big unrolled network, we add up all the input-to-hidden, hidden-to-hidden, and hidden-to-output weight matrices, which is like collapsing the unrolled network back to down the original network, the one on the left side of figure 16.3. If we present the network with a sequence of input vectors from the training set, in sequence, it should produce the same sequence of target vectors. And, it should

[7] There are other methods of training recurrent networks, eg. real time recurrent learning [128], and "dynamic reconstruction" (Haykin, 14.13) [50].

[8] This involves exposing the network to each of the inputs in a sequence from left to right, so that hidden layer activations are updated in a specified order. This hidden layer activations provide contextual disambiguation in the same way the copy-back later does in a SRN. The resulting sequence of outputs is compared to the target output sequence, producing an error, which is backpropogated.

generalize, producing similar sequences of outputs to those in the target set, given a similar sequence of inputs.

16.4 Recurrent Networks and Language Generation

Once various technical issues with training recurrent networks were resolved as part of the deep learning revolution, people started using them to do truly amazing things, in particular, build models of language that could generate text. This was a precursor to large language models, and was at the time quite amazing.

In a now-classic blog post, Andrej Karpathy describes some of these applications.[9] For example, he trained a neural network on a bunch of Shakespeare[10] (with words coded as vectors, of course) and then ran the RNN, which produced it's own version of Shakespeare [59]. Here is a sample:

> VIOLA:
>
> *Why, Salisbury must find his flesh and thought*
> *That which I am not aps, not a man and in fire,*
> *To show the reining of the raven and the wars*
> *To grace my hand reproach within, and not a fair are hand*

In another example Karpathy trained a network to speak like Tolstoy using an English Translation of *War and Peace*. Remember, these are just networks' trained using variations on backprop. So he would train the network to produce sequences of statements that were similar to sequences of statements in Tolstoy. As the network learns using gradient descent, the outputs seem more and more like Tolstoy. Recall how we could track Nettalk learning to speak at different stages of training, like a creepy baby (section 15). Here we can do the same thing with the network, observing get better and better at speaking in the voice of Tolstoy. Here is some network output after iteration 100:

> tyntd-iafhatawiaoihrdemot lytdws e ,tfti, astai f ogoh eoase rrranbyne 'nhthnee e plia tklrgd t o idoe ns,smtt h ne etie h,hregtrs nigtike,aoaenns ln

The network doesn't even really have the concept of a word yet. By iteration 700 it has words, kind of:

> Aftair fall unsuch that the hall for Prince Velzonski's that me of her hearly, and behs to so arwage fiving were to it beloge, pavu say falling misfort how, and Gogition is so overelical and ofter.

Finally, after a few thousand iterations, it's beginning to sound like a Russian novel:

> "Why do what that day," replied Natasha, and wishing to himself the fact the princess, Princess Mary was easier, fed in had oftened him. Pierre asking his soul came to the packs and drove up his father-in-law women.

Karpathy got these supervised recurrent networks to do other cool things: for example, he generated a fake wikipedia page, fake source code, and a fake math paper [59]. A fragment from the artificially generated mathematics paper is shown in Fig. 16.4. These techniques have even been used to generate a fake script for a movie, which was then actually produced![11]

16.5 Limitations of Supervised Recurrent Networks

Karpathy's blog post was written in 2015. In 2020, GPT was released, and in 2022, ChatGPT was publicly released, and thus began the next era in the history of neural networks, which we are currently living in.

[9] `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`. Note that Karpathy makes use of LSTMs; see note 12.

[10] The data is here: `http://cs.stanford.edu/people/karpathy/char-rnn/shakespear.txt`.

[11] The move is called "Sunspring", and can be viewed here: `https://www.youtube.com/watch?v=LY7x2Ihqjmc`. The opening shows a list of all the other movie scripts that were used to train the network to produce its script [87]

Lemma 0.1. *Assume (3) and (3) by the construction in the description.*

Suppose $X = \lim |X|$ *(by the formal open covering* X *and a single map* $\underline{Proj}_X(\mathcal{A}) = \mathrm{Spec}(B)$ *over* U *compatible with the complex*

$$Set(\mathcal{A}) = \Gamma(X, \mathcal{O}_{X,\mathcal{O}_X}).$$

When in this case of to show that $\mathcal{Q} \to \mathcal{C}_{Z/X}$ *is stable under the following result in the second conditions of (1), and (3). This finishes the proof. By Definition* **??** *(without element is when the closed subschemes are catenary. If* T *is surjective we may assume that* T *is connected with residue fields of* S*. Moreover there exists a closed subspace* $Z \subset X$ *of* X *where* U *in* X' *is proper (some defining as a closed subset of the uniqueness it suffices to check the fact that the following theorem*

(1) f *is locally of finite type. Since* $S = \mathrm{Spec}(R)$ *and* $Y = \mathrm{Spec}(R)$.

Figure 16.4: Fragment of "fake math" generated by a recurrent network trained on real math.

See chapter 17. In a way, those advances grew out of efforts to overcome limitations of supervised recurrent networks like the one's we've described.

We've seen that recurrent connections can provide a network with context information (for example with the copy-back layer of a SRN). What the network sees at time t is an external input and some trace of what happened in the past. Input plus context is enough for the network to learn to produce meaningful temporal sequences. Transformer networks improve on this basic idea. The main problem they run into is that they favor recent context over earlier context. But oftentimes it is important to be more sensitive to information much earlier in a sequence than more recent information, and as a result recurrent networks have a hard time capturing larger temporal structures, like the plot of a story or the overall theme of a musical performance.

There are also technical problems, like the the problem of vanishing gradients: as error is backpropagated through the weights of a network (section 13.4), the amount weights are changed further back in time gets smaller and smaller. Another problem is that this type of network does not lend itself to parallel computing hardware like GPUs on fancy graphics cards. This can be dealt with using some tricks, like truncating how far back the "window" of backprop goes ("truncated" backprop through time), but this only takes us so far.

One approach to this problem is to use nodes with more complex activations functions (see chapter 5).[12] However, these approaches have not been discussed much in connection with cognitive science, and are being supplanted by other techniques, so we pass over them for now.

The answer was to drop recurrent networks, in a sense, and move back to feed-forward.

16.6 Connectionist Applications

Having reviewed the latest and greatest in supervised recurrent networks, let us return to earlier and simpler models, to examine their relevance to cognitive science.

In Chapter 12 we saw that multilayer networks trained by supervised learning methods (e.g. backprop) can develop psychologically interesting internal representations that involve a re-mapping of inputs in a hidden layer. Thus backprop is relevant to psychology even if it is not neurally realistic. Similar work has been done with supervised recurrent networks.

In a famous paper, Jeff Elman trained a simple recurrent network like the one in figure 16.2 to predict the next words in a sentence.[13] The input data used to train the network consisted of thousands of sentences generated using a simple "caveman" grammar. Some sample sentences generated by this grammar are shown in Fig. 16.5. The network predicts the next word in a sentence at any time. For example, it predicts that the word "cat" will be followed by "chase" or "eat". With training, error can be reduced, but it never goes to zero, because a given word can be followed by more than one other word. But some words are more likely

[12] Two such functions are "long-short-term memories" (LSTMs) [107, 90] and "gated recurrent units" (GRUs). See `http://colah.github.io/posts/2015-08-Understanding-LSTMs`. These functions—which are like compound activation functions containing several others as parts—allow activations to be gated or turned off altogether. When placed in a recurrent network trained using gradient descent, they can learn to pass information far along a temporal sequence, "jumping" over intermediate nodes, in a way that avoids the problem of vanishing gradients.

[13] `http://psych.colorado.edu/~kimlab/Elman1990.pdf`.

than others to follow one another, and so the predictions match these patterns. It also learns some general rules, like expecting verbs after nouns [25].[14]

mouse move book	man move book
cat move	woman break
book break	cat eat cookie
dragon break glass	mouse see cat
dragon eat man	mouse move rock
dragon eat cat	cat break
monster eat cookie	rock move
woman see plate	monster smash plate
woman break plate	monster eat man
woman sleep	dragon eat mouse
woman see man	monster eat cookie
man move book	man sleep
cat eat bread	woman see woman
mouse see mouse	man move book
mouse break rock	woman break
cat break	mouse eat cookie
rock move	cat see cat

Figure 16.5: Some of the sentences used to train the word prediction network.

The striking thing about this network was that it learned a set of grammatical categories in its hidden layer, *without being told anything about grammar*. The internal representations the network learned corresponded to grammatical categories like noun and verb, as well as semantic categories like animal, human, food, and breakable. None of these categories were directly programmed in to the network: it simply learned these categories in the process of solving the input-output task it was given (compare the way Nettalk learned phonological categories when learning to read aloud, or the way deep networks develop realistic edge detectors and other feature detectors when trained to recognize objects).

Figure 16.6 shows a schematic reconstruction of the structure of the SRN's hidden unit space.[15] Points correspond to vectors in the hidden unit space. Distances between points are meaningful: points closer to each other in the diagram correspond to hidden unit vectors that are closer to each other in the hidden unit space. Notice that the internal representations form in to a kind of hierarchical structure, that mirrors the structure of English grammar and also some features of English semantics. Nouns are broadly separate from verbs. Within nouns we have animate vs. inanimate nouns.

The network was used to argue against the prevailing view in psycholinguistics, associated with Noam Chomsky, that grammars are innate, rather than learned. The rules of grammar were not given to the network. They were learned by the network when it was trained to predict the next word in grammatical sentences [25]. The grammar was in the environment–it was implicit in the sentences of the training dataset– and then learned by the network.

This is a good example of connectionism. Elman used backprop to show neurally plausible processes could capture psycho-linguistic phenomena, without claiming to show exactly what was happening in the brain.

A final, more recent example, shows that these ideas persist, even among those who are not concerned with cognitive science or psychology. Karpathy's amazing recurrent neural networks, which learned to generate fake math texts, encyclopedia pages, and source code, were also analyzed for internal representations. The results were fascinating. Some nodes only turned on when the network was producing text inside of quotations. Other nodes only turned on at the beginnings and ends of lines. Other nodes kept track of opening and closing brackets or parentheses [60].

This is by no means a settled area. As discussed in section 17.5.1, linguistic processing using transformers

[14]Simbrain simulations that partially replicate the results of these papers can be accessed in the script menu as `elmanPhonemes.bsh` and `elmanSentences.bsh`.

[15]The picture this is based on was generated by exposing the network to each word in the context of many sentences (*e.g.* "eat", "cat"), and then taking the average hidden unit activation across these exposures. The resulting vectors are like the centers of clusters in the hidden unit space.

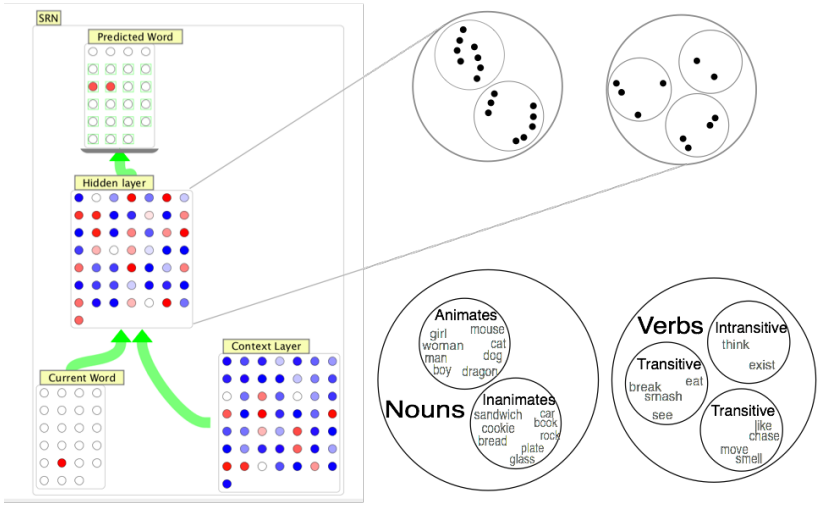


Figure 16.6: The SRN used in the word prediction task. The output layer predicts the next word (coded as a binary vector) in a sentence. It is currently undecided between which of two words might come next. The right top panel shows (schematically, this is not a projection of actual data) what points in the hidden unit space corresponding to different words. The right bottom panel shows how these points cluster in to grammatical and semantic categories.

has become a whole sub-field of computational linguistics, known as BERTology. The question of what kind of meaningful representations develop in other language models trained using transformers, such as GPT-3, is a natural direction for further research. To take just one example, in the paper referenced above [77], it is argued that the human cognitive system uses some variant on a transformer model, what they call "query based attention" or QBA. They note: "Some attention-based models (28) proceed sequentially, predicting each word using QBA over prior context, while BERT operates in parallel, using mutual QBA simultaneously on all of the words in an input text block. Humans appear to exploit past context and a limited window of subsequent context (29), suggesting a hybrid strategy." They argue that this context is not just linguistic, but encompasses a whole situation we are in, and that we use this situational context—what we see, hear, are doing etc.—to disambiguate words in context.

Chapter 17

Transformer Architecture and LLMs

Jeff Yoshimi, Pierre Beckmann, Tim Meyer

As discussed in the history section 3.8, we have entered a new stage in the history of neural networks, what we are calling the "age of generative AI", which should be familiar to you via such tools as ChatGPT. In their most familiar form, a **large language model** (LLM) based on the **transformer architecture** generates text responses to text inputs by repeatedly predicting the next word or token in a sequence.[1] They are trained on large datasets of everyday text, like text from the internet, which is easily available. As noted in section 3.8, it is common to equate "transformer" with "LLM", but the two concepts are distinct. The transformer is the neural network architecture, while an LLM is just any model of language generation that is based on a large dataset. An LLM can be built out of something besides a transformer, and transformers can be used on things besides language. For example, some state of the art image classification models are now transformer-based, and OpenAI has released an impressive video generation model – Sora, which also runs on a transformer architecture. However in this chapter we focus on transformer-based models of text generation like GPT.[2] We will sometimes refer simply to "LLMs" by which we mean transformer-based LLMs.[3]

Earlier efforts at text generation and natural language processing used supervised recurrent networks (chapter 16), which are, as we saw, in various ways limited. In particular, they can only process a small amount of context, and suffer the vanishing gradient problem. The transformer architecture is basically a complex feed-forward network that can be "aware" of multiple kinds of relationships between arbitrarily far-flung parts of an input stream. Because it is a feed-forward network, many of the older techniques covered in this book can be applied to the architecture. In particular, all the lessons of the deep learning revolution (section 3.7) apply here, and indeed, transformers are many-layered deep networks (chapter 14) that make good use of both **representational width** and **representational depth**. They can be trained on large datasets using highly optimized parallel hardware. Like all the other networks discussed in this book, they are not just useful as engineered tools, but are highly relevant both to neuroscience and cognitive science, and seem to develop meaningful internal representations.

We start with preliminary discussion of how transformer-based LLMs are trained using highly available text data, and how a special recursive trick can be used to make a feed-forward network that only predicts next words still produce meaningful conversational outputs. We then discuss how the transformer architecture itself works. Finally we consider how LLMs are utilized and evaluated and the relevance of these models to

[1] The concept of token is introduced in chapter 8. Following practice introduced there we will vacillate between "token", which is more accurate (since it encompasses punctuation, word parts, and other non-word entities) and "word", which is more intuitive.

[2] There are many other models in this class. As of this writing (June 2024), this includes the open Ai GPT series: GPT, GPT2, GPT3, GPT4, and GPT 4o. It also includes BERT (Google's first LLM, which is now out-dated), Gemini (Bard), several Claude models (Anthropic; semi open-source), Llama, LLama2 and LLama3 (Meta), and Alpaca (Stanford; open source). Most of these models can only be accessed online but some can be downloaded and run locally, further fine-tuned, etc. A list of LLMs ranked by how well they chat is here: `https://chat.lmsys.org/?leaderboard`.

[3] There are numerous high quality online resources for learning about LLMs. An excellent visual introduction is at this website: `https://poloclub.github.io/transformer-explainer/`. Three blue one brown is always excellent on visual intuition and he has a youtube video. For a more technical walk through on building an LLM from scratch see Karpathy's tutorial.

cognitive science, neuroscience and other areas.

Changes in this area are rapid, and the relevance of these areas to cognitive science is only now being studied, so updates to this chapter are expected.

17.1 Learning to speak Internetese

In section 16.4 we saw how recurrent neural networks trained on example text can learn to speak in a way that reflects the statistical properties of the training data. A network trained on Shakespeare will start to speak fake Shakespeare, a network trained on real math can generate fake math, etc. Large language models using transformers do the same thing, they just do it much better. The architecture is better suited to the task, as we will see, and they can use much larger datasets (hence "large" being added in front of "language model"). In fact, the training set for GPT-3 was not all of Shakespeare, or just a bunch of math papers, but rather a large subset of the *entire internet*, which included all of Wikipedia, a few compilations of books, and a web-scraped archive of the internet called "common crawl" (`https://en.wikipedia.org/wiki/Common_Crawl`). See figure 17.1. Similar datasets continue to be used on LLMs, so if you've ever written anything online, there is a decent chance it is part of the training data for one of these models.

Since all of Shakespeare is on the internet, and discussions of every topic of human endeavor from physics to history, and plenty of gossip and randomness about popular culture and everything else, these models can talk about all of these things. They can statistically generalize from their training data, which consists of a large part of the internet, which in turn encompasses many of the books and recorded knowledge of human history. In a sense, these models learn to speak "internetese".

Dataset	Quantity (tokens)	Weight in training mix
Common Crawl (filtered)	410 billion	60%
WebText2	19 billion	22%
Books1	12 billion	8%
Books2	55 billion	8%
Wikipedia	3 billion	3%

Figure 17.1: The datasets used to train GPT-3. The data mostly consists of data scraped from the internet, but lots of books and all of Wikipedia are also included.

The results are impressive. The texts these models produce are no longer obviously fake in the way the examples from section 16.4 were. In fact, in some cases they arguably pass the **Turing Test**, a long-standing test for artificial general intelligence, answering questions and producing convincing text in response to prompts. Whether LLMs really pass the test is a matter of ongoing controversy. This is discussed further in section 17.5.4.

17.2 Training Using Next-Word Prediction

Recall from chapter 7 that a training dataset consists of inputs and targets, and is also called a labeled dataset. As noted there, this kind of labeled data can be hard to obtain. We might have lots of pictures of people but not know the names or identities of the people in the pictures, or lots of pictures of cats and dogs but not whether a given picture is a cat or dog. Creating targets for a large dataset is labor-intensive, requiring humans to manually label each picture.

LLMs like GPT use a special method (sometimes known as auto-regression) to take *any piece of text* and convert it into a training data set. The trick is, roughly, to treat a given string of text tokens without the last token as an input, and then to treat the final token as a target or label. More specifically we use vector embeddings of the tokens. The amazing thing about this method is that it can be used to take *any text* as a training dataset. No longer do we have this difficulty of finding labeled data. Just take any old piece of written text, and you've already got multiple training examples, just by taking concatenated vector

embeddings of different sequences of tokens as input and vector embeddings of the next tokens after those sequences as targets.

For example, consider this block of text adapted from the Wikipedia page for UC Merced:

> The University of California, Merced is a public land-grant research university in Merced, California. It is one of the ten campuses in the University of California (UC) system. Established in 2005, Merced is the newest campus within the UC system.

From this, we can create a bunch of training examples, a list of input / target pairs. We might use "The University of" as an input, and then "California" as a target. We simply associate each token in the input with a vector using a word embedding (chapter 8). (The target is also vector encoded, but in a different way; we discuss this in section 17.4.2). Using the word we can build a table of input-target vector pairs, which we can use to train a feed-forward neural network. For a sense of the idea, see figure 17.2.[4]

Input Tokens									Targets
Hello	how	are	you	?					Good
Hello	how	are	you	?	Good				thank
Hello	how	are	you	?	Good	thank			you

Token Embeddings									Targets
(8, 8, 6, 1)	(2, 8, 7, 0)	(2, 1, 5, 2)	(6, 1, 3, 9)	(0, 1, 1, 5)	(0, 0, 0, 0)	(0, 0, 0, 0)	(0, 0, 0, 0)	(0, 0, 0, 0)	(0,0,0,0,1,0)
(8, 8, 6, 1)	(2, 8, 7, 0)	(2, 1, 5, 2)	(6, 1, 3, 9)	(0, 1, 1, 5)	(7, 3, 6, 1)	(0, 0, 0, 0)	(0, 0, 0, 0)	(0, 0, 0, 0)	(0,0,0,0,0,1)
(8, 8, 6, 1)	(2, 8, 7, 0)	(2, 1, 5, 2)	(6, 1, 3, 9)	(0, 1, 1, 5)	(7, 3, 6, 1)	(4, 3, 7, 8)	(0, 0, 0, 0)	(0, 0, 0, 0)	(0,0,0,1,0,0)

Figure 17.2: How text sequences can be converted into training datasets for large language models. A set of tokens is converted into vectors using a token embedding (the vectors shown are arbitrary, just to illustrate the idea). We can treat each concatenated partial sequence of token embeddings as an input and the next token as a target for the supervised learning algorithm.

Note that the word embedding in this example is 4-dimensional (each token is associated with an array of 4 numbers, a vector in a 4d space). In real LLMs this "embedding dimension" is quite large, for example over 12,000 for GPT-3 (see figure 17.8).

In practice, a sequence of tokens is converted into a matrix where each row corresponds to the token embedding for one token. This idea is illustrated in section 8.4.5, where it was described as a "token embedding matrix". This stack of vectors is ready to be processed by the LLM.[5]

One interesting feature of transformers is that the processing they do does not inherently care or know about the order of tokens in a sequence. Thus, the tokens in a token embedding matrix are in a sense unordered (their position in a stack is an order, but this information is not easily available to the network given how it processes information), which allows for super-fast parallel processing. Thus a **positional encoding** is used to modify the vectors in a token embedding matrix, often by using a trigonometric function that simply adds more or less to the numbers in the vectors depending on where they are in a sequence.[6] In figure 17.4 the encoding subtracts 1 from each component of the word embedding for "Hello" (which was $(8, 8, 6, 1)$ for the sample embedding in figure 17.2).

[4]Note that normally a word like "University" would be split into multiple tokens, but we are keeping things simple here. Some information tokenizers is in chapter 8.

[5]What is actually fed to the transformer is a batch of these token embedding matrixes, that is a rank-3 tensor (see section 6.9).

[6]This is a kind of feature engineering trick; see chapter 7.

17.3 How Text is Generated from a Feed-Forward Network

One confusing thing about figure 17.2 is that we go from a large input covering a whole set of tokens to a single token as output. So great, we can predict single words. But how do we go from single words to generating long text outputs, or having conversations? In fact, in hearing about generative AI as "next word prediction" machines, you may have sometimes wondered how such complicated things can happen when all they model does is predict next words.

The answer is by using what we can call the "recursion trick". This trick allows us to take a feed-forward network that only predicts next words and use it to produce streams of text output. In fact, it's remarkably simple. We feed a network a set of inputs corresponding to string of text, and it produces an output corresponding to the predicted next token. That output is then appended to the previous input, and this longer input is now fed back to the network. This process is repeated to produce a stream of text outputs. This technique can be used to generate unending sequences of text from any prompt.[7] The prompt is our input, and then answers are generated using the recursion trick. Text will continue to be generated until a special end of sequence token is reached.[8]

So this gets us one response. But then you type a new question. That *entire question* is appended to the *entire past conversation* including both what you and GPT have said so far.

Suppose, for example, we want to ask a network "hello how are you?" The input to the network is the whole sentence {"hello", "how", "are", "you", "?"}. Let's not worry about the vector embeddings, and just see the general idea, as shown in figure 17.3. Notice that the initial prompt is the initial input, but then the prompt *and* the first word of the response are used as the next input, and this process can be repeated until a response is written out.

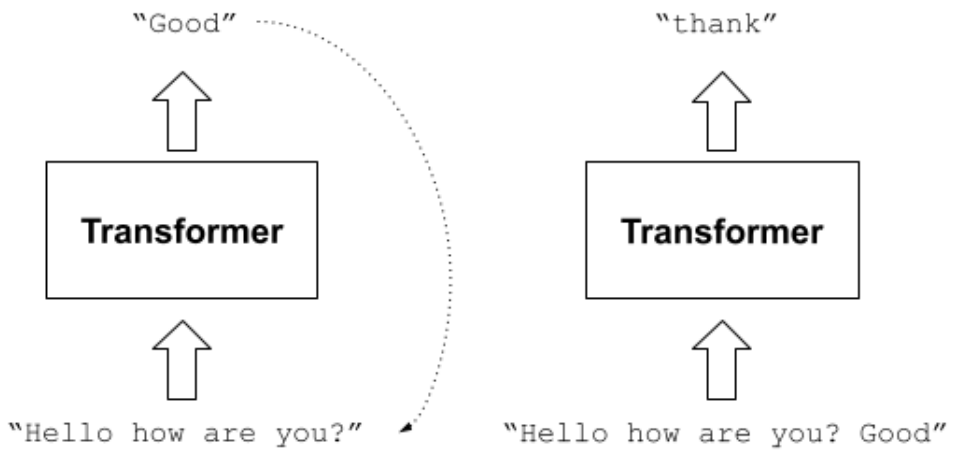


Figure 17.3: A schematic view of how "conversations" are generated from a feed-forward network in systems like GPT. The output from one moment is added to the end of the input, and the new input is then fed in. The process is repeated to generate a full response.

Of course, as we keep doing this, the inputs to the network get larger and larger, and there must be some limit to how far we can go, right? The answer is yes. Any LLM specifies a fixed length **context window**.

[7]Notice that this is a kind of recurrence, and arguably this makes LLMs used in this way a kind of recurrent network. Outputs are fed back in as part of inputs. However, the outputs are text which then must be converted back to text inputs which are then vector encoded. In fact, recurrent networks were originally used for text processing, as we saw in chapter 16,but it turns out fancy feedforward networks used in this way outperform them. (In those cases a vector representation of each token in the sequence would be presented separately: "hello", "how", "are", "you", and "?".).

[8]It was an important step forward with ChatGPT when the ability to produce end of sequence tokens in a way that mimic human conversation or "chat" was produced. This was done using special training techniques, in particular, RLHF, reinforcement learning with human feedback. RLHF uses reinforcement learning and policy gradient descent, with a human in the loop. This was introduced with GPT-3.

At the start of a session, this window is mostly zeros, except for the initial prompt. In a dialog, the prompts from a person and the responses from the LLM are both included until the context window is filled. Thus, if the context window is large enough, whole series of back and forth conversations can be processed. All the prompts and responses up until the current point are part of the input, and then the LLM uses the recursion trick to generate new responses that are sensitive to everything that's been discussed thus far. When the system runs out of slots in its context window, items are simply removed from the start of the context window (from a computer science standpoint, this is a queue). This is intuitive in figures 17.2 and 17.3. These context windows can be remarkably large. GPT-3 has a context window of about 2000 tokens or about 6 pages of text, and early versions of GPT-4 had context windows of 32,000 tokens or about 72 pages of text.

17.4 The Transformer Architecture

We have thus far covered high-level features of how LLMs work, but have treated the transformer as a black box. The time has come to open the box. How does the fancy feed-forward network at the heart of these models work? Its power rests on a few novel architectural innovations, which combine representational width and representational depth with a special form of context awareness. All the things we've seen about other neural networks apply here. It is a kind of deep feed-forward network, which uses a huge amount of training data. But the key innovation is that within each "layer" it can develop many forms of context representation, which relate all the tokens in a context window to each other.

17.4.1 Blocks

The transformer architecture [119] contains layers or "blocks" which are specialized to process the large context windows that are fed to the network as input. With training, they learn to find long-range dependencies between different parts of a context window. Recall that the context window includes an original prompt, its own response to that prompt, etc.; it includes the *entire exchange* you've had with GPT up to the current point, so long as it fits in the context window. Each block combines a "self-attention" layer with a traditional linear layer and several other mechanisms (see figure 17.4). The self-attention layer is where the magic happens. One part of this layer compares each token in the context window to every other token in the window. In our simple example, "hello" is compared to "how", "are", "you", and also to itself. These comparisons are used to create a representation of the sentence that reflects dependencies between the words. All words are compared to each other in the context window, so that no matter how far apart they are, they can still influence each other. The self-attention mechanism learns what relations between words in a context window are important; in a sense it learns what to focus on (hence "self attention"). This ability to find meaningful relationships within an input sequence is part of why transformers are relevant to cognitive science, as we will see.

The details of what occurs in a block are not developed in detail here though we are planning to expand the discussion in future versions of this chapter. However, here is a rough sketch of what happens. Each input token in a context window is first converted into a vector using a vector embedding (chapter 8). This vector is then simultaneously matrix multiplied by three weight matrices labeled $\mathbf{K}$, $\mathbf{Q}$, and $\mathbf{V}$ to produce three vectors: the key, query, and value vectors. This is what is shown in figure 17.4. Note that the matrices, in bold, are part of what is trained in this architecture.

The key and query vectors are then multiplied (using a dot product) in all possible combinations and then normalized to produce a context representation. This is shown in figure 17.5. These are also called self attention scores. These scores capture relationships between tokens in a context window. The value vectors (not shown in figure 17.5) are then matrix multiplied by the self-attention matrix to produce the outputs of one attention head of a transformer layer.

Within each attention block, there are multiple attention "heads". Each head carries out all the operations described above: it projects the input from the previous layer (or the embedding in the first layer) into separate spaces through unique projection matrices for $\mathbf{K}$, $\mathbf{Q}$, and $\mathbf{V}$. These projections result in smaller vectors for each head.[9] For example, if the embedding dimension were 9, it might be projected into three

[9]The dimensionality of each head is often the embedding dimension divided by the number of heads, so that when the

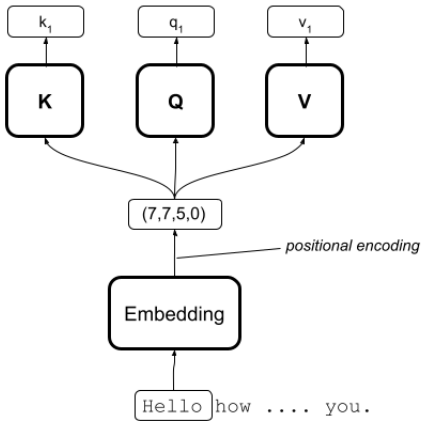


Figure 17.4: Part of a transformer block. Each token in the context window is converted into a vector using the word embedding, then transformed using the positional encoding. This vector is then multiplied by three matrices **K**, **Q**, and **V** to produce three vectors. These operations can be done concurrently and can thus run on fast parallel computing hardware. The resulting vectors will be used to produce a representation that captures relationships between items in the context window. Bolded items are matrices that are updated using gradient descent.

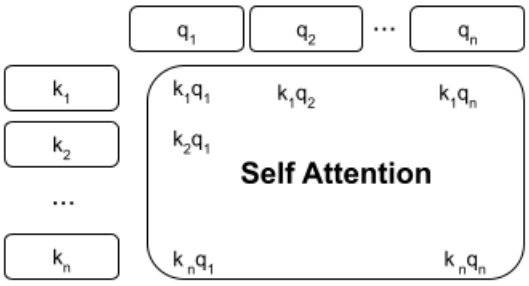


Figure 17.5: Scaled self attention matrix generated by multiplying key and query vectors (using the dot product). Value vectors (not shown) are then multiplied by this matrix to produce the output of one head of a block.

separate 3-dimensional heads.[10]

As a result of this multi-head attention, the network can learn *multiple* ways to compare words in the sentence to each other, a bit like how a convolutional network (section 14.1) develops *multiple* filters to analyze an image. The results of these different attention heads are combined and as a result each layer of a transformer network involves a sophisticated representation of the sentence that represents multiple types of inter-word dependency. The number of heads per block corresponds to a more complex form of representational width.

This entire process happens once for each head. The outputs of all the heads are then concatenated back together and normalized, and put through a standard feed-forward network of the kind we've been talking about throughout the book.[11] The output of the block is a set of vectors that has the same shape as the

outputs are concatenated, the original dimension is restored. In figure 17.8, the number of inputs to each head, d_{head}, is equal or almost equal to the embedding dimension d_{model} divided by the number of heads n_{heads}.

[10]Thus, for each head, the **K**, **Q**, and **V** matrices would typically have dimensions 3×9, where 9 corresponds to the input dimension from the previous layer, and 3 refers to the reduced dimensionality for each head (note that for efficiency, these matrices are often combined into a single larger matrix.)

[11]Residual connections are also used. That is, the input to the block is added directly to the output of the multi-head attention mechanism. This allows the gradient descent algorithm to have a direct route backwards through the blocks of the network, as a way to address the vanishing gradient problem (see chapter 16).

input vectors. See figure 17.6. For example, in the example in figure 17.7 the input would be 9 vectors with 4 components each, and the output would be too.[12]

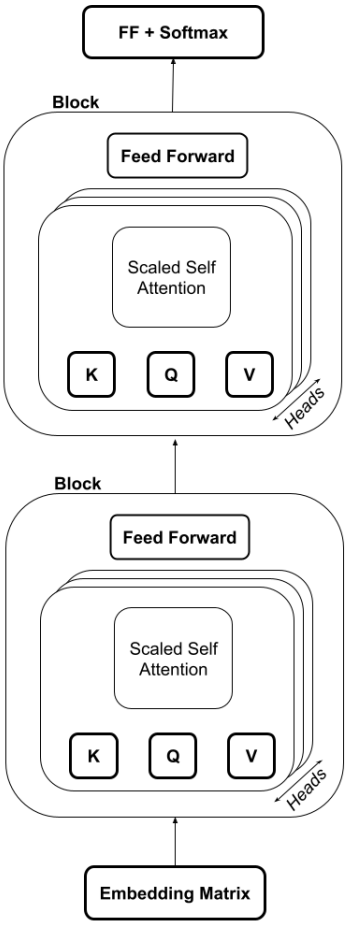


Figure 17.6: Schematic of the transformer architecture. Multiple blocks are stacked, capturing representational depth, as in a CNN. Each block contains a multi-head attention structure, where each head learns to represent inputs to the block in a different way. This captures representational width. The outputs of the multiple heads are combined in a standard feed-forward network. Other structures handling residual connections, and adding and norming activations, are not shown. As above bolded items contain trainable parameters that are updated via gradient descent.

Now we take a lesson from deep networks, and stack many of these transformer blocks on top of each other, to produce increasingly sophisticated representations. This is representational depth. Recall that with deep networks for vision, we get features, features of features, features of these features, etc. whose activations match neural response properties of different layers of the human visual system. This builds on the old idea of the *Pandemonium* model (section 3.3), which involved (at successive layers): edge detectors, detectors for combinations of edges, detectors for combinations of these combinations (e.g. fragments of letters), and ultimately letter detectors. In a similar way, the successive layers of a transformer model of language correspond to increasingly complex features of the input stream, including syntactic categories, semantic properties, and far more complex features as well.[13]

The whole thing is trained using gradient descent and supervised learning (chapter 12). It's the same

[12]Two excellent visualizations are `https://bbycroft.net/llm` and `https://poloclub.github.io/transformer-explainer/`.

[13]The extent to which activation patterns correspond to syntactic or semantic features is measured using post-hoc interpretation techniques such as probing. As with so many other neural network features, these were not "programmed in" by the engineers, but are emergent from the network after training, and are studied and described by scientists after the fact.

ideas as with a simple feed-forward network, but with more components trained. Items in bold in the figures above are trained: the word embedding, the key, query, and value matrices for each head in each block, and the normal weights and biases of the feed-forward networks. Gradient descent is being pushed back through a *lot* of stuff here!

17.4.2 Softmax Outputs

The first step in a transformer is to embed inputs, as we've seen. The last step is to un-embed them.

In figure 17.2 we saw that while inputs use a word embedding, outputs are probability distributions over the whole vocabulary, and targets are one-hot encodings consisting of all zeros and a single "hot" number one corresponding to the predicted next word (on one-hot encodings see section 7.3). When target data are binary one-hot encoded labels, the task given a network is technically a classification task (section 12.3). Thus transformers are technically classifiers, which classify input texts according to what word is likely to occur next. Classification can here be seen as serving as a *proxy task*. Our actual task is to predict a set of probabilities over next tokens. The output of an LLM is a set of probabilities over next tokens.[14]

The output of an LLM is often a **softmax** layer with around 50,000 outputs, which indicate how probable a token is given the input (all the prompts and responses in the context window so far). See figure 17.7 for how this might look for our simple example, where the output vocabulary just contains 7 tokens.

Input Tokens

									Targets
Hello	how	are	you	?					Good
Hello	how	are	you	?	Good				thank
Hello	how	are	you	?	Good	thank			you

Token Embeddings

									Targets
(8, 8, 6, 1)	(2, 8, 7, 0)	(2, 1, 5, 2)	(6, 1, 3, 9)	(0, 1, 1, 5)	(0, 0, 0, 0)	(0, 0, 0, 0)	(0, 0, 0, 0)	(0, 0, 0, 0)	(0,0,0,0,1,0)
(8, 8, 6, 1)	(2, 8, 7, 0)	(2, 1, 5, 2)	(6, 1, 3, 9)	(0, 1, 1, 5)	(7, 3, 6, 1)	(0, 0, 0, 0)	(0, 0, 0, 0)	(0, 0, 0, 0)	(0,0,0,0,0,1)
(8, 8, 6, 1)	(2, 8, 7, 0)	(2, 1, 5, 2)	(6, 1, 3, 9)	(0, 1, 1, 5)	(7, 3, 6, 1)	(4, 3, 7, 8)	(0, 0, 0, 0)	(0, 0, 0, 0)	(0,0,0,1,0,0)

Figure 17.7: What some training data might look like for the example in figure 17.3. The top panel shows the tokens in three training examples and the bottom row shows the corresponding token embeddings. The bottom panel shows inputs that can actually be fed into a neural network, in this case, a network with $9 4 = 36$ input nodes. Note that the targets are one-hot encoded, because these networks are actually classifiers (see section 17.4.2). Thus the output layer in this simple network would have 7 nodes for the 7 tokens: "Hello", "how", "are", "you", "?", "Good", "thank". The softmax outputs are not shown, but for a well-trained network the outputs would be probability distributions close to the targets, for example $(0, 0, .01, 0, .99, 0), (.01, .01, 0, 0, 0, .98), (0, .01, 0, .98, 0, .01)$. On softmax see section 5.5

Once we have a probability distribution over tokens, we select one of the most probable next tokens and that becomes the output. This is usually done by sampling from among the top n most probable next tokens. This can be controlled using the softmax temperature parameter, where higher temperatures make the outputs more random or "creative" (see section 5.5). Thus the final softmax layer does the opposite of what the embedding layer does. Rather than converting from tokens to activations, it converts from activations to tokens, and is thus a "de-embedding" layer.

[14]The interesting thing is that by *trying* to perfectly classify next tokens (an impossible task), we end up with good probabilities, which is exactly we're after. Here is a way to think of it. If the model was trained to 100% accuracy on the classification task, then it would always generate the same sentences from the same prompts (because it would assign one unique token to the current input). But then it could not generate new instances of text.

17.4.3 Parameters and hyperparameters

By way of summarizing, consider the parameters and **hyperparameters** associated with a transformer-based LLM. Recall that parameters are primarily weights and biases. Figure 17.8 shows this for a range of GPT-3 subvarieties as n_{params}. The released version of GPT-3 had 175 Billion weights and biases. By comparison, our xor 2-2-1 network had 3 biases and 6 weights, or 9 parameters. A standard convolutional neural network might have millions of parameters. So this is just massively larger. Of course things have gotten larger. GPT-4 has about 1.76 trillion parameters. It takes large server banks that consume huge amounts of energy to run these models, and so there is a thread attempting to achieve similar performance with smaller models, some of which can be run on a personal computer.[15]

Hyperparameters are values that describe the structure of a model. For a simple feed-forward network the main hyperparamters correspond to the number of layers and nodes per layer. For a transformer, they correspond to the number of layers n_{layers}, dimension of the word embedding d_{model}, number of attention heads n_{heads}[16], a batch size, and a learning rate. Even if the details of how transformers work is difficult, you should know enough to interpret this table. Certainly learning rate and batch size are values we've seen in other chapters.

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Figure 17.8: Number of parameters and values for some of the hyperparameters used in training different versions of GPT-3.

The size of the context window is another hyperparameter not shown in the figure. According to IBM research, "when ChatGPT made its debut nearly two years ago, its window maxed out at 4,000 tokens. If your conversation went over the 3,000-word chat-interface limit, the chatbot was likely to hallucinate and veer off-topic. Today, the standard is 32,000 tokens, with the industry shifting to 128,000 tokens, which is about the length of a 250-page book. IBM just open-sourced on Hugging Face two Granite models with a 128,000-token window, and more are on their way."[17]

17.5 Analysis of LLMs

As with all the architectures discussed in this book, transformer-based LLMs are not just used to engineer useful devices but are also major object of scientific interest (see chapter 2). They are studied in many academic disciplines and areas of cognitive science. They are in various ways being considered as models of cognition, linguistic processing, and neural processing.

These models have the remarkable property of being human engineered but not completely understood by the humans who engineered them (this is true of other models, but is especially the case here, given how complex LLMs are, and given how expensive they are to train).

Here are some representative examples of recent research into LLMs. Here again, the landscape is rapidly changing, and this section will have to be frequently updated.

[15] For example, Gemma2B achieves performance similar to GPT 3.5 with 2 billion parameter. See `https://developers.googleblog.com/en/smaller-safer-more-transparent-advancing-responsible-ai-with-gemma/`.

[16] Notice that the size of the heads, d_{head} is equal or almost equal to $d_{\text{model}} \times n_{\text{heads}}$ (some mismatches are allowed).

[17] `https://research.ibm.com/blog/larger-context-window`.

17.5.1 LLMs and Cognitive Science

A theme running through this book is that cognitive scientists are interested in neural networks, even when they originate in fields outside of cognitive science. That trend has continued with the emergence of large language models.

One obvious topic of interest to cognitive scientists is how transformers develop representations that are responsive to long-range dependencies in an input stream. A problem for earlier connectionist accounts of language processing was that they could not handle such dependencies. In an early paper on transformers co-authored by Jay McClelland [77] (a member of the original PDP group at UCSD; see section 3.6), the following example is used to illustrate the point:

> John put some beer in a cooler and went out with his friends to play volleyball. Soon after he left, someone **took the beer** out of the cooler. John and his friends were thirsty after the game, and went back to his place for some beers. When John opened the cooler, he discovered that the beer was _____.

What the reader expects in the blank spot depends on text earlier in the sentence. Here the reader expects the word "gone" next, but if "took the beer" is replaced with "took the ice" earlier in the sentence, the reader expects something like "warm". Transformers do quite well at this task. McClelland and colleagues describe transformers as using a "query-based attention" and have investigated the psychological and neural significance of the internal components of transformers when processing input streams.

Recent work has gone much further with the project of understanding the psychological, computational and neural significance of transformers relative to human perception and cognition. Among current interests are papers that show that LLM representations are aligned with neural activations, perceptual spaces, physical spaces and conceptual spaces. Abdou et al. show that language models can encode perceptual color relationships, even though they have never "seen" colors the way sighted humans have (but have only read about color relationship in texts) [1]. Liétard et al. find that LLMs can predict relative geographic distances between cities [73]. Christiansen et al. report that LLMs organize concepts in ways that correlate with human conceptual structures [21]. These findings collectively suggest that LLM representations are not "empty symbols" as in traditional computer programs (see the discussion in 1.2.3). Instead, they are content-rich and structured similarly to human mental representations.

Given these results, it seems likely that a lot of future research on LLMs will contribute to the field of cognitive science. Conversely, the acquired findings of cognitive science also contribute to research on LLMs. Notably, these findings are used to better measure the cognitive abilities of LLMs. One of the tasks of the main LLM benchmark BIG-BENCH, notably draws on cognitive science to test the ability to understand and combine concepts, including novel and made-up ones [117].

17.5.2 LLMs and Linguistics

Linguists have considered how these models handle syntax and semantics, revealing insights that bridge computational models and linguistic theory. One key study ("What Does BERT Look At? An Analysis of BERT's Attention") investigates the attention patterns in BERT and their correspondence to linguistic phenomena. This study found that BERT's attention heads often focus on specific linguistic roles, such as the direct objects of verbs or the determiners of nouns. Interestingly, some attention heads display broad attention across entire sentences, while others are highly focused, suggesting a complex, multi-faceted approach to language understanding within the model.

17.5.3 LLMs and Neuroscience

Just like Yamins and DiCarlo used convolutional neural networks (CNNs) to predict brain activity (section 14.5), researchers are now leveraging transformer-LLMs to understand how our brains process language.

Schrimpf and colleagues [108] explore how LLMs, like GPT and BERT, can predict neural responses during language comprehension. They proceeded by comparing brain activity of participants reading or listening to sentences with the activity patterns generated by these models. One of their key findings is that transformer models, especially GPT, can predict nearly 100% of the explainable variance in neural responses to sentences!

An interesting aspect of this study is the comparison between GPT and BERT. GPT, which processes language in a unidirectional manner (predicting the next word based on the previous words), was found to outperform BERT, which uses a bidirectional approach (considering the context from both directions). This suggests that our brain predicts unidirectionally, which makes sense given that we usually are confronted with language orally and so get "fed" sentences word by word.

17.5.4 LLMs and Philosophy

Philosophers are also interested in LLMs for the obvious reason that they have completely changed the game and even passed the Turing Test. But do they understanding language?[18] One line of argument is that a machine can't get the "meaning" of a word just by manipulating meaningless symbols. After an article by Emily Bender and colleagues [6], it has become common to say that LLMs are "stochastic parrots" that capture nothing more than mere co-occurrence probabilities.

These arguments have a long pre-history in what are sometimes called *unusual realization arguments*. These arguments

> ask us to imagine people carrying out simple operations on bit streams, connectionist-like computations on activations patterns, or the control operations of the central processing unit of a computer. For example, Dneprov asks us to imagine a stadium full of people who pass 0's and 1's to one another on the basis of instructions announced over a loudspeaker. We can further imagine that this stadium is linked to an input-output device that enables it to pass the Turing test. We can then ask ourselves: do we really think this stadium full of people has consciousness or genuine understanding? [89]

Bender and Koller have adapted these arguments to LLMs using an "Octopus test" [7], which goes like this: two English-speaking castaways communicate via telegraph across islands. A super-intelligent octopus intercepts their messages, learns the statistical patterns of the messages going back and forth, and, "feeling lonely", inserts itself into the conversation. The question is whether they would be able to convince the interlocutors they were intelligent– a version of the Turing Test. Bender and Koller argue that the octopus could pass the test for simple chatbot like interactions, but that it would fail in responding to strange situations involving words referring to objects the octopus has never perceived. This scenario is used to argue that, like the octopus, LLMs may produce seemingly meaningful outputs based solely on statistical patterns, without true understanding of language or the concepts they appear to discuss.

However, others argue that LLMs do understanding the meaning of words they produce. Relying on findings presented in section 17.5.1, Sogaard [116] argues that the representations LLMs rely on are grounded (that is, they are meaningful) in virtue of their alignment with human neural, cognitive, and perceptual spaces. These are not like the empty symbols passed along in Dneprov's stadium or in other "unusual realization arguments." Here, the LLM (or the octopus) manipulates representations that have internal structures aligned with those of the human mind. In any case, the impressive results of models such as chatGPT fundamentally question not only empirical insights about language but also very deep philosophical conceptions about language.

Another long-standing argument against machine intelligence due to Hubert Dreyfus [23] is that no computer or neural network could ever have truly human capacities, because they were not raised in the real world and can only parrot back what they were programmed or trained to say, at best with moderate abilities to generalize beyond that (Bender and Keller's argument develop this idea with respect to LLMs and argue these considerations are why the octopus would pass the test). However, humans with their embodied human existence develop an intuitive sense of the full context of the lived world. To show this, one of the authors (Yoshimi), who has regularly taught Dreyfus' argument for many years, would have the class ask strange or unusual questions to AI systems and chatbots of earlier years, like "What would happen if a penguin were to juggle alligators." The systems invariably struggled to say anything meaningful (though some did well by tricks, like pretending to be a teenager saying "lol who cares"). But chatGPT does just fine with the question, as you are welcome to see for yourself. Not everyone is convinced by this, and the

[18] This is a throwback to the old AI-connectionism debate (section 3.3), and one way to think about the current debate is as a revival of that one.

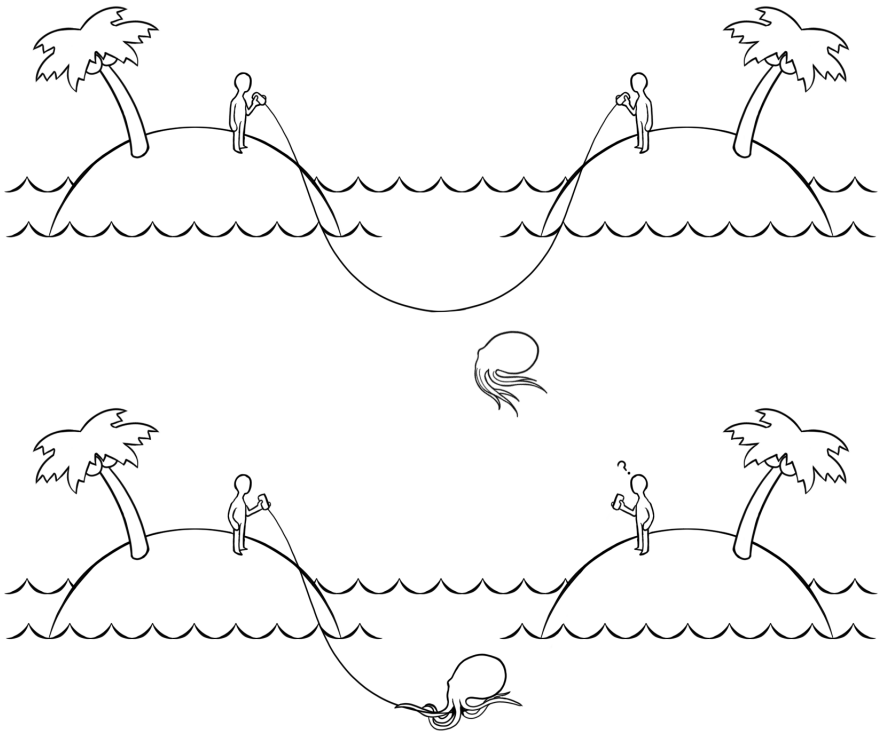


Figure 17.9: The octopus test. In the upper panel the octopus is observing the information being passed back and forth. At a certain point the octopus intercepts the wire and starts passing statistically similar messages to the person on the first island. This is shown in the lower panel. The communication is convincing, and the person on the left does not realize what has happened (the poor person on the right is just out of luck at this point). It has been argued that the octopus would not understand the information it passed along, even if it was convincing to the human. Moreover, it has been argued that the octopus would fail to produce convincing text in certain novel situations.

discussion remains active. Melanie Mitchell [84] has argued that despite appearances, these systems are not as intelligent as they appear (there are many reasoning tasks they fail on, for example).

Chapter 18

Spiking Models: Neurons & Synapses

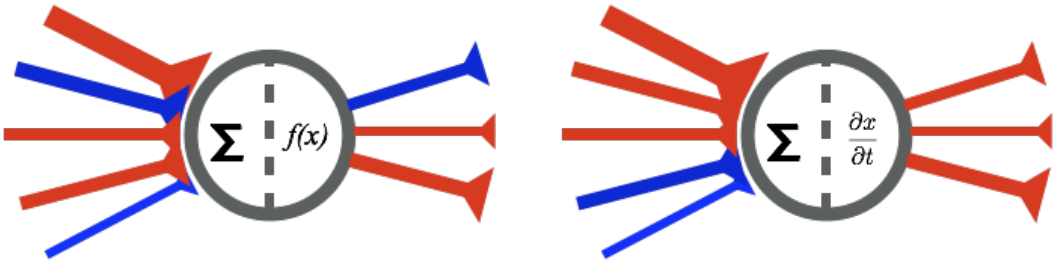
Zoë Tosi, Jeff Yoshimi

Neurons of higher animals (from *drosophila* to *homo sapiens*) are characterized by an all-or-nothing on/off response that propagates signals via discrete packets known as **action potentials (APs)**. APs are characterized by a sudden increase in the **membrane potential** of a cell, brought about by a positive feedback process. After a short period of time (typically < 1 ms) other mechanisms in the cell rein in this positive feedback, causing a rapid decrease in membrane potential. When voltage across the cell membrane is plotted over time we see a sudden sharp increase, then a rapid decrease in electrical potential taking the form of a **spike** (see 18.2). Canonically this spike in the membrane voltage travels down the axon of a neuron to its synaptic terminals where voltage sensitive processes cause neurotransmitters to be released into the synaptic cleft, where they are taken up by other cells.

Coming from an engineering or connectionist background this process may seem unfamiliar since "neurons" (perhaps better called "nodes") associated with canonical artificial neural networks take on continuous values. Explicitly stated or not these continuous values are typically understood to vaguely represent the firing rate (number of spikes produced in some time window) of neurons–if such things are even within the realm of consideration. It is the case that in simple organisms neurons do take on what are known as "graded potentials", which can be accurately described by a continuous variable. However these organisms tend to be extremely small and simple such as *C. Elegans*, a 1 mm flatworm for which all of the exactly 302 of its neurons and their connections are known. In higher animals the vast majority of neurons communicate via spikes and therefore any continuous variable assigned to represent the instantaneous activity of a neuron is in one way or another derived (using windowed or exponential averages, for example) or an indirect measure (e.g. a strong correlations between variables like excitatory conductance and firing rate) [92]. Collectively the set of techniques for converting the activity of a spiking neuron into a single continuous value representing that activity is known as **rate-coding**. As it stands there is no clear consensus as to the importance of precise spike timings versus generalized rate-coded activity when studying information processing in the brain.

What do we want out of a neuron model? The answer to this depends upon the question(s) we are interested in. Given that this chapter focuses on "spiking neurons" we will assume that at the very minimum the questions of interest require that APs not be abstracted away regardless of our use of rate-coding. From there we have a great many choices, which will be covered in detail later in the chapter. If this minimum criteria of "produces spikes" is our only criteria then highly simplified models scarcely different from the nodes of an ANN will fit the bill. Alternatively we may be interested in the particulars of some aspect of neural dynamics or require that our neurons' voltage waveforms during spikes correspond to those found in nature. If that is the case then our neuron model must itself be a complex entity with multiple variables governed by a set of coupled differential equations. Regardless of the complexity our biological models must replicate the basic functions of a neuron, which entails more than the neuron model itself. Contrary to what may be believed neurons and their functions can be understood in relatively simple terms. As a fundamental unit of the nervous system neurons 1) integrate inputs from neurons and/or some sensory stimulus 2) respond in some manner to that integrated input, and 3) propagate that output to other neurons or motor outputs.

The models discussed thusfar represent (2) in this formulation. In this chapter we will be discussing both spiking/biological neuron models and how those neurons interact with other neurons via synapses ((1) and (3)).



	Cannonical Neural Network Node and Weights	Canonical Biological Neuron Model and Synapses
Output	*f(x) = tanh(x), sigma(x), max(0, x), etc…* output is **continuous** value	*Sends an AP only if* $x > \theta$, where θ is some threshold or peak paramter; output is **discrete** "all or nothing" AP
Update	Fully determined by inputs and transfer function: *f(x)* where *x* is the net input; *updated in* ***discrete iterations***	Differential equation(s) operating on state variable(s) (represented by *x*), where inputs *affect* these dynamical variables; *numerically integrated* ***continuously in time***
Input Integration	Net activation **determined instantaneously** via a weighted sum from other neurons	Incoming currents or potentials **integrated over time**— *synapses and their effects on cells have their own dynamics*
Polarity Organization	**Synapses** are excitatory or inhibitory	**Neurons** are excitatory or inhibitory
Synapse Properties	Synaptic Signals **Instantaneous;** typically no dynamics	Synaptic Signals can have **time delays** and their responses have **duration**

Figure 18.1: A broad comparison of neurons typically found in artificial neural networks and those found in computational neuroscience network models. These characterizations are not completely definitive, but can be thought of as describing the prototypical or canonical instantiation of each type.

18.1 Level of abstraction

Neurons are intrinsically complex entities when high levels of detail are taken into account, and indeed many computational neuroscientists have made neuron models that capture most or all of the relevant features of the cell. This involves modeling membrane potentials and ion channel dynamics for many different parts of a neuron. Models that do this are known as *multi-compartment models*. This is further complicated by the large variety of neurons and their morphological and electrochemical differences. Depending on the questions one is interested in modeling with this level of detail may be required.

Here are focusing on networks, so we focus on model neurons with no specific morphology. Thus we focus on **point neurons** (sometimes "single-compartment" neurons). Similarly a neuron may connect to another neuron via many different synaptic terminals that have different impacts on the target cell based on their strengths and locations (cell dendrites, soma, and the axon hillock being common). We only consider the impact of a single synaptic connection between two neurons, which stands in for this more complex relationship. In the literature the total change in a post-synaptic's membrane potential from a single axonal fiber is referred to as a unitary post-synaptic potential.

Neurons are also associated with neurotransmitters, which have distinct functions. Synaptic receptors on the cell respond to specific neurotransmitters from other cells and can be divided into two classes:

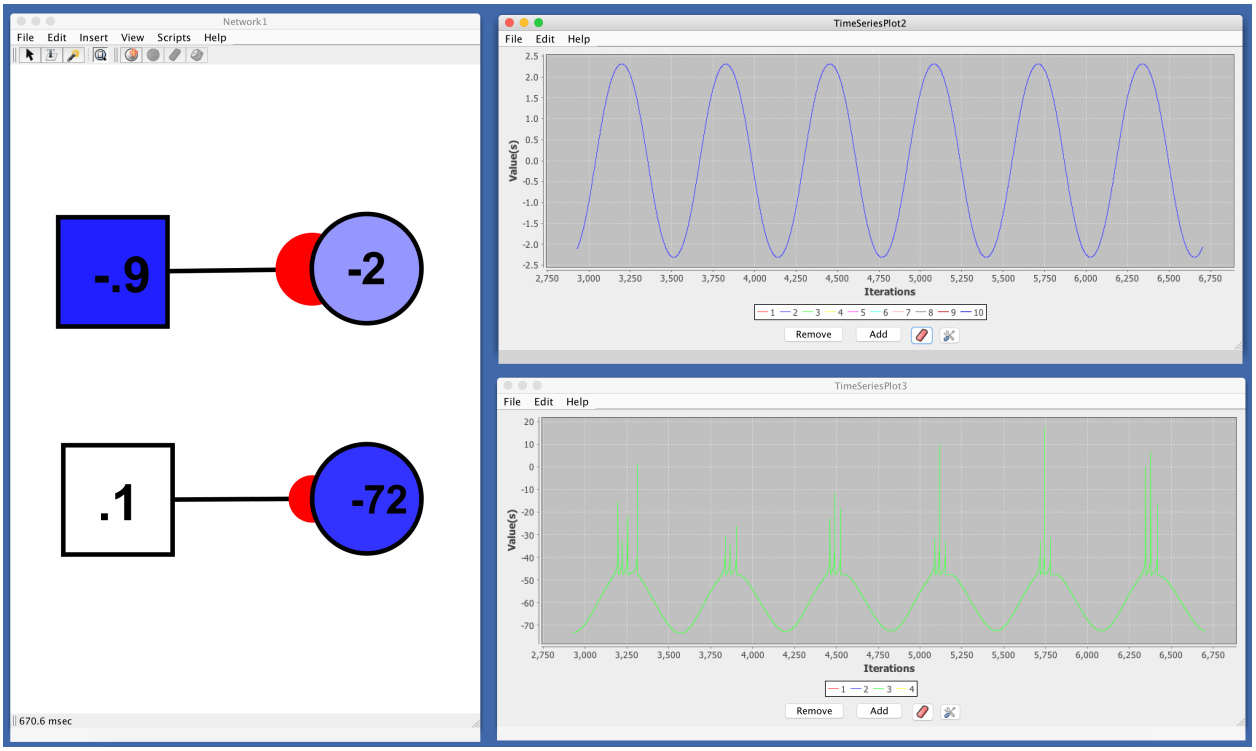


Figure 18.2: A Simbrain desktop with two cells receiving inputs from activity generators. Both are providing the same sinusoidal input. Above is a typical connectionist neuron using a logistic activation function (bounded on (-5, 5)) , and below is a spiking neuron model. Across from each is a time series of activation (blue) and membrane potential (mV; green) for the logistic and spiking neurons respectively. Notice the sharp increases and rapid decrease in membrane potential; these are "spikes" and if this neuron were connected to other neurons a signal would be propagating down the outgoing synapses. Also notice that the spikes are not the same in quantity or overall size and that there are small variations in their distance from one another. These are the result of the spiking neuron having its own internal dynamics unlike the logistic neuron.

Ionotropic and **Metabotropic**. Ionotropic receptors are the simplest and are essentially ion channels on the surface of a cell that respond to a neurotransmitter by either opening or closing, thus rapidly having a direct, clear-cut impact on the membrane potential of the cell (see 4.1). Most glutamate receptors (NMDA, AMPA, and kainate) and $GABA_A$ receptors are of this type and are excitatory and inhibitory, respectively. Metabotropic receptors on the other hand respond to neurotransmitters in complex long-term ways. When these receptors are activated a signal transduction pathway is set in motion that produces complex cellular and metabolic responses. Well-known neurotransmitters such as Dopamine, Serotonin, Acetylcholine, and Norepinephrine typically use receptors of this type (see 4.1.3). We focus on ionotropic neurotransmitters in this chapter.[1]

[1]Why abstract away so many potentially important features? The simple answer is that for many questions in neuroscience these details simply aren't necessary. This is particularly true of "higher level" phenomena. For instance a neuron's ability to store information about its past inputs can be explained without any reference to morphology, metabotropic neurotransmitter receptors, etc. [74]. Complex features of synaptic structure and firing patterns have also been explained via phenomenological models that make no reference or appeals to specific chemical interactions [66, 82]. Often it is assumed (with excellent outcomes) that lower level complexity in fact underlies the higher level mechanisms of a model such that direct simulation of those complexities is not required–only their effects or their phenomenological outcomes are of importance. As an analogy consider a car in a videogame about racing. Does the game simulate an internal combustion engine? Must it simulate the chemical combustion of octane for each cylinder, the friction between various components of the engine and so on? Or is it acceptable to merely simulate the motion of the car in response to inputs from the user? In this case the important aspect is the fact that the car moves not precisely how it does it and furthermore we have means of translating input from the user into the motion of the car in precise, reproducible, and realistic ways that make no appeal to the precise inner workings of the engine. Now in a sense this analogy could be construed as being in favor of the abstraction of all underlying mechanisms–indeed why use neural

18.2 Background: The Action Potential

An action potential begins with some sort of forcing, which can produced by sensory stimulation, input from other neurons, input from an experimenter, or the neuron's own complex dynamics. Regardless of case causes sodium ion channels to open.

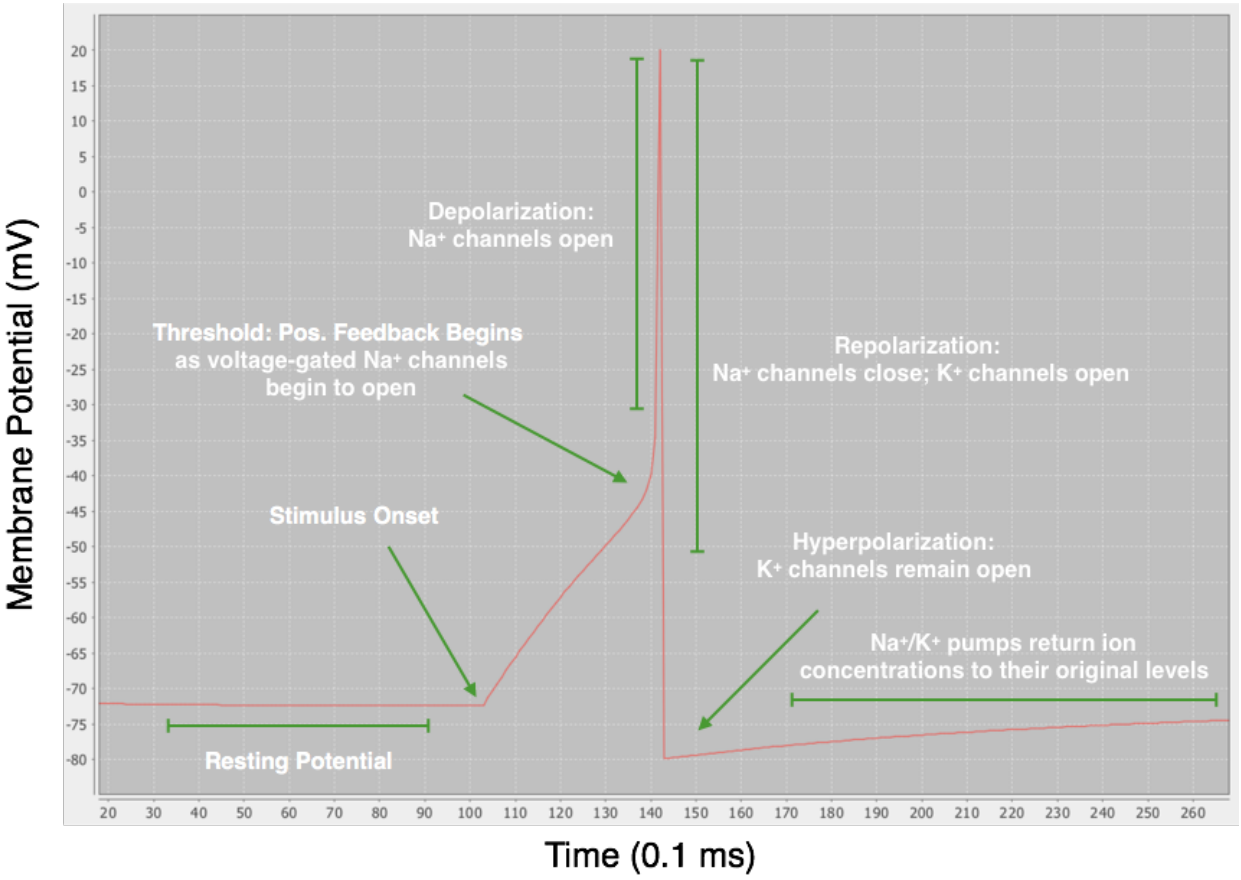


Figure 18.3: A canonical action potential.

18.3 Integrate and Fire Models

The action potential is an all or nothing phenomena that almost always has the same amplitude. Furthermore the conditions for release of neurotransmitter are also always roughly the same. Thus if we are unconcerned with modeling the behavior of the membrane potential we can turn our attention entirely to this aspect of the neuron: Neurons integrate some input which then causes a response. If this response exceeds some threshold then an action potential is triggered resulting in post-synaptic events being fired along the neuron's outgoing synapses. That is why these are referred to as "**Integrate and Fire (IF)**" neurons. These models are focused exclusively on the all or nothing action potential. They are usually tuned to make spike timings and **inter-spike intervals (ISIs)** more precise and to match these signals to biology. These terms are *not* concerned with matching the time-course of the membrane potential, or to ion channel behavior, or to known physiological values.[2]

networks at all! The lesson here is not that everything should be abstracted, but merely that certain things can *depending upon what we are interested in*. The "right" level of abstraction for neuroscience depends upon what we want to know, what we are using our models for, and so on. It is a question that is often hotly debated in nearly every context in which it occurs.

[2]A notable exception might be the Izhikevich model, which was tuned so as to be able to replicate the membrane voltage traces of different kinds of neurons in addition to matching spike timing behavior.

18.3.1 The Heaviside step function

The spiking threshold model is the simplest of all spiking models. In fact it is so simple that it doesn't even meet many of the criteria listed in Fig. 18.1. It focuses on only one aspect of a spiking neuron, namely the all or nothing response that occurs in response to exceeding a threshold. There are no internal dynamics.

$$r_j = \Theta\left(\sum_i r_i \,;\, \theta\right)$$
$$\text{where: } \Theta\left(x\,;\,\theta\right) = \begin{cases} 0 & x < \theta \\ 1 & x \geq \theta \end{cases}$$

Here r is the activation value of the neuron, which takes on a discrete value of 1 or 0 based on the output of the Heaviside function ($\Theta(x)$) for the net input of the neuron (the sum of the outputs of neurons i projecting onto j). Here we have parameterized the Heaviside function, which typically is 1 for values greater than or equal to 0 and 0 otherwise such that the argument passed to it (the net input in this case) must exceed a specified threshold for activation, θ. Models using these neurons are the simplest of all spiking models and typically are not simulated continuously in time–instead being simulated by discrete iterations. This implies that networks composed of these neurons typically do not have dynamic synapses and instead in many ways closely resemble a typical ANN architecture.

18.3.2 Linear Integrate and Fire

The linear integrate and fire model is one of the simplest of spiking neuron models that also has intrinsic dynamics. While we are not concerned here with fitting voltage traces for neurons, these models do attempt to capture precise spike timings, including timings that have interesting and varied behavior.

18.4 Synapses with Spiking Neurons

18.4.1 Spike Responses

In the world of spiking neurons we are typically working in a continuous-time scenario rather than with discrete iterations. This presents a problem for the transfer of synaptic current from one neuron to another, since spikes are instantaneous events and as instantaneous events have an integral of 0 for all finite values. Moreover, if we are in the business of simulating spiking neurons then we likely are concerned to some degree with the more realistic simulation of synapses as well, and in the real world synapses release neurotransmitters into the synaptic cleft over *some duration*. Therefore for biological, mathematical, and practical reasons, a number of functions have been used to describe the post-synaptic response a synapse delivers to the post-synaptic neuron after a spike occurs.These are **spike response functions** or in Simbrain, "spike responders".

Perhaps the most straightforward spike response function is an instantaneous jump and decay. Here it is assumed that upon the arrival of a spike a synapse instantly responds, proportional to its synaptic strength, and that response decays exponentially.

$$\dot{q_{ij}} = -q_{ij}\tau_{psr} + w_{ij}\delta(t - t_i^n - \tau_{dly})$$

Or in closed form:

$$q_{ij}(t) = \sum_{t_i^n < t} w_{ij} e^{-(t - t_i^n - \tau_{dly})/\tau_{psr}}$$

Here q is the post-synaptic response: the value at each synapse, which the post-synaptic neuron j sums over to determine the total current from synaptic responses. τ_{psr} is the time-constant of the exponential decay, w_{ij} is the weight of the synapse connecting neurons i and j. t is current time while t_i^n is the time of spike n at neuron i and τ_{dly} is the time-delay over the synapse. Lastly δ is the Dirac-delta function. This function is commonly used to represent spikes in continuous time systems, it is a function that is zero at all

points on $[-\infty, \infty]$ except 0 where it is infinite. Thus its integral $\int_{-\infty}^{\infty} \delta(x)dx = 1$. This corresponds to a convolutional instantaneous jump and decay since the result of the application of this differential equation on the entire spike train of neuron i, $(n = 1, ..., N; t_i^n)$ is equivalent to the convolution of that spike train with an exponential kernel with a time constant of τ_{psr}. This isn't always (nor does it have to be) treated as a convolution however and a maximum post synaptic response: q_{max} can be established if so desired.

The instantaneous rise in post-synaptic response is not particularly realistic, since it takes some time for neurotrasmitter release to fully activate (not every synaptic vesicle is filled and touching the cell membrane ready for release after all). Thus there exist formalisms for modeling both the rise and the decay of a post-synaptic response. The simplest of these is the *rise and decay* or *alpha function*, which models both rise and decay using the same time constant. Having the same same time constant for both rise and decay is not particularly realistic, but it is also not an unreasonable approximation, and provides more realism than the instantaneous jump and decay function.

$$\dot{q_{ij}} = -q_{ij}/\tau_{psr} + r$$
$$\dot{r} = -r/\tau_{psr} + w_{ij}\delta(t - t_i^n - \tau_{dly})$$

Or in closed form:

$$q_{ij}(t) = \sum_{t_i^n < t} w_{ij} \left(\frac{t - t_i^n - \tau_{dly}}{\tau_{psr}} \right) e^{1-(t-t_i^n-\tau_{dly})/\tau_{psr}}$$

Here all variables are the same as in the previous section. Notice that we will reach our maximum response when the difference between the current time t and the arrival of pre-synaptic spike n from neuron i $(t - t_i^n - \tau_{dly})$ is equal to our time constant τ_{psr}.

18.5 Long-term plasticity

18.5.1 Spike-Timing Dependent Plasticity (STDP)

Spike-Timing Dependent Plasticity (STDP) is to spiking neurons what Hebbian learning is to continuous-valued nodes in a traditional artificial neural network. It is a mechanism for altering the strength of synaptic connections in a (typically) temporally asymmetric way such that certain pairings of spike times result in a change of the strength of a synapse. STDP or some variation thereof is believed to be a key component of a group of mechanisms underly memory, learning, and information processing in the brain. It is believed to play a key role in the development and maintenance of the specific synaptic structure of neural circuits.

A Hebbian version of STDP is such that a spike in the pre-synaptic cell that temporally precedes a spike in the post-synaptic cell leads to a strengthening of the synapse while a spike in the post-synaptic cell which precedes a spike in the pre-synaptic cell leads to a weakening of the synapse. The former is known as **Long-Term Potentiation (LTP)** while the latter is referred to as **Long-Term Depression (LTD)**. This sort of relationship acts to create or reinforce positive temporal correlations between two neurons (at least in the case that the pre-synaptic cell is excitatory). However, STDP is not always Hebbian and can take on other forms. The **STDP Window** is a function that takes as input the difference in spike times between a pre- and post-synaptic cell and outputs the change in the strength of the synapse connecting them. The familiar Hebbian window is the most common as it is ubiquitous among connections between two excitatory cells (which make up 80% of any given neuronal population in cortex). More exotic windows (of which there are many) are typically found at synapses where one or both of the cells involved are inhibitory.

18.5.2 STDP

In its most simple forms STDP is performed additively giving rise to the name "Addititve STDP" (or "Add-STDP). Here we introduce the fundamental formalisms of STDP, which will appear in other variants and use add-STDP (sometimes referred to as "vanilla" STDP) as the particular instantiation.

Add-STDP refers to a broad set of models for STDP for which the change in the strength of a synapse is independent of the strength of that synapse. That is, weight changes occur in an *additive* fashion. Add-STDP takes on the following form:

$$\Delta w_{ij} = \eta A_{+/-} exp(-|u|\tau_{+/-})$$

where $u = t_j^f - t_i^f$ and t^f is the most recent time that either neuron i or j produced an action potential and η is a learning rate. The $A_{+/-}$ term is one of two different constants depending upon the sign of u (the difference in spike times between the pre- and post-synaptic cell). As the nomenclature implies this gives us A_-, which is typically negative reflecting LTD for negative u (pre- fired after post-) and A_+ (typically positive reflecting LTP) for positive u (post- fired after pre-). Similarly τ is a time-constant determining how quickly the effects of LTP and LTD decay over time in relation to the amount of time between pre- and post-synaptic spikes and as with A, τ_+ and τ_- correspond to LTP or LTD. Together, this creates a piecewise exponential function centered (usually) on zero. This discrete formulation is applied instantaneously whenever neurons i or j fire.

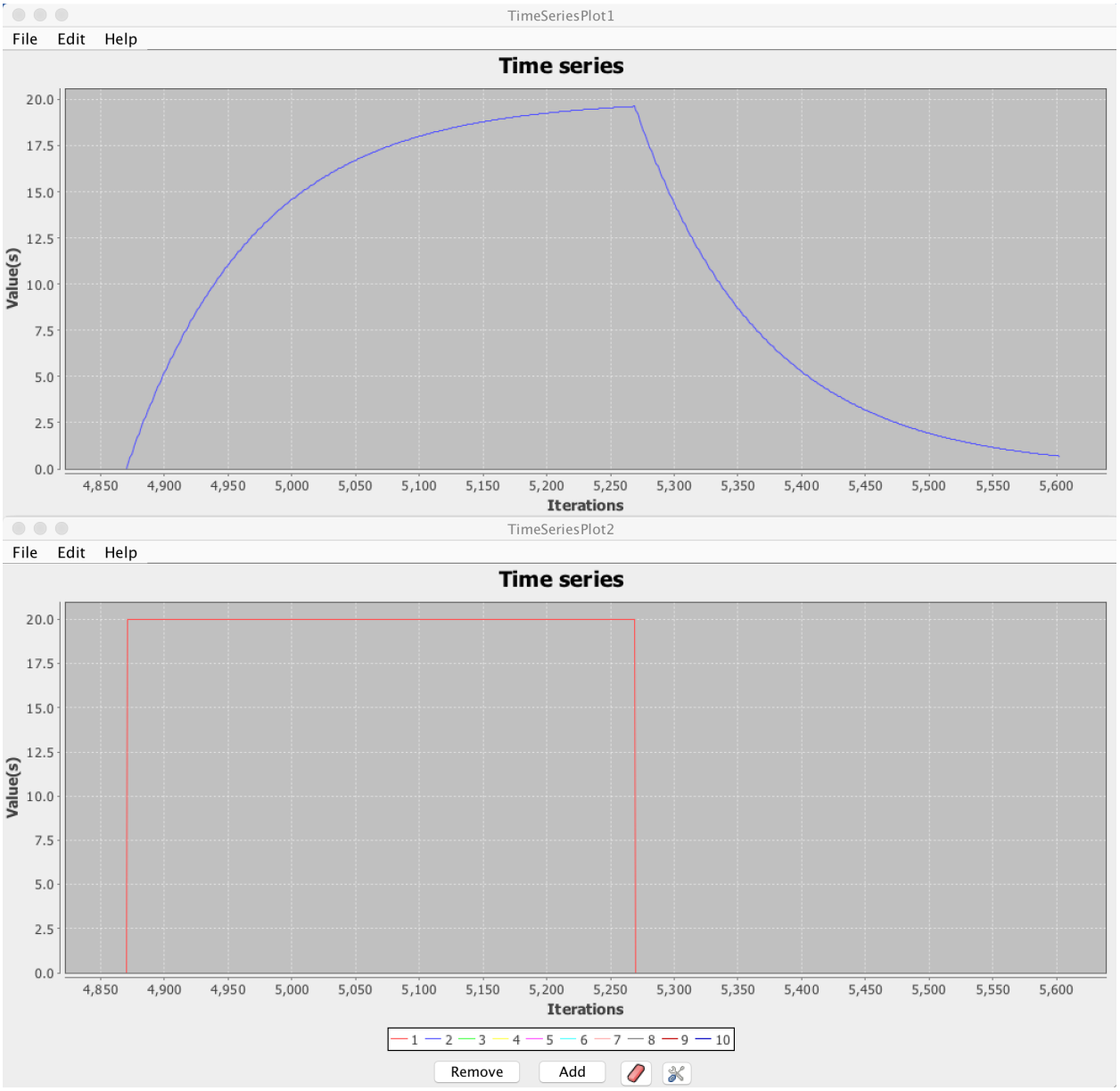


Figure 18.4: The response in the membrane potential of a linear integrate and fire neuron (blue/top) to a constant current injection (red/bottom). Notice the *nonlinear* dynamical response in the LinIF neuron's membrane potential to a constant step in current.

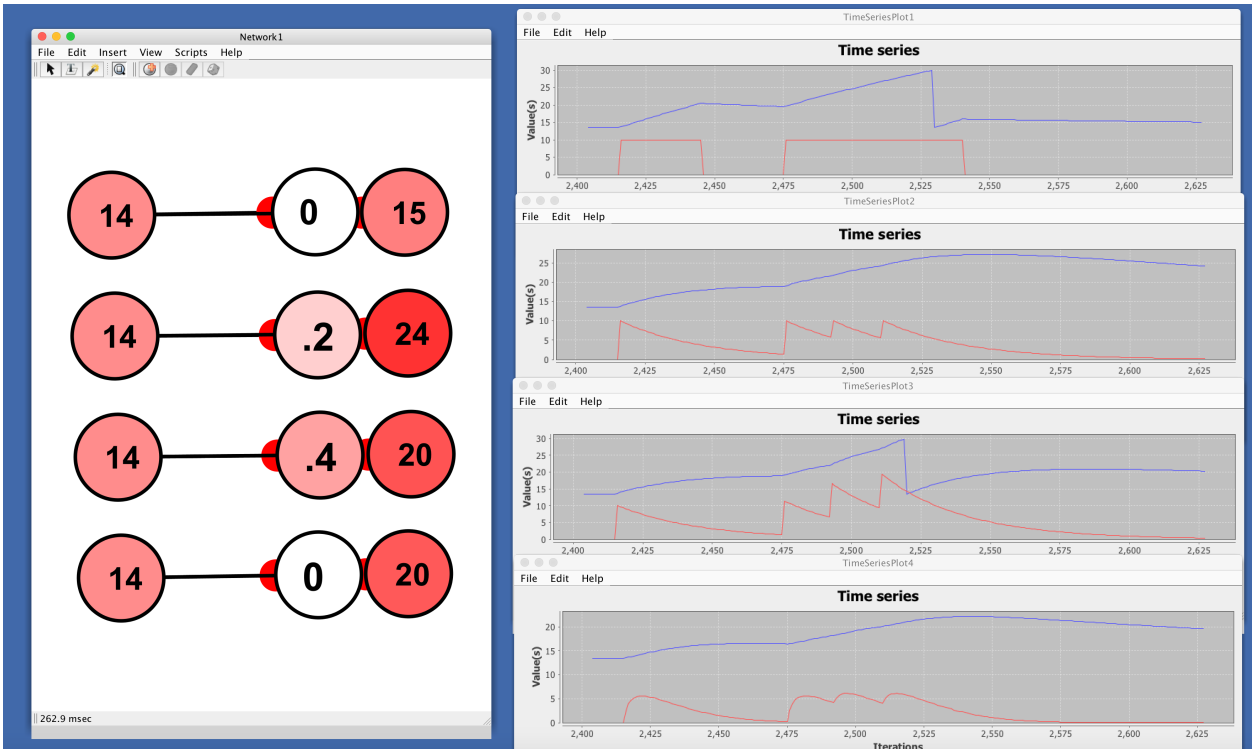


Figure 18.5: The post-synaptic responses for 4 different spike response mechanisms (red) plotted alongside the resulting change in post-synaptic membrane potential (blue). From top to bottom, we have a step-function, an instantaneous jump and decay with no convolution, a convolutional jump and decay, and a rise and decay function. Here post-synaptic neurons obeyed a linear integrate and fire rule. Notice the dynamical response from the post-synaptic IF neurons.

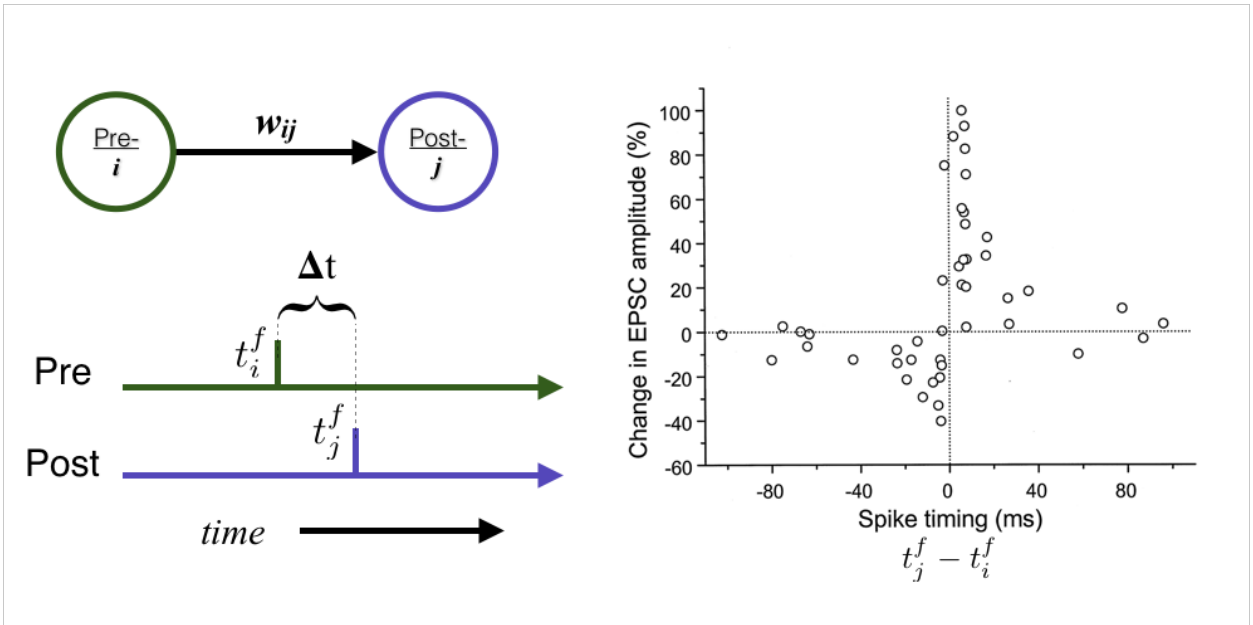


Figure 18.6: A diagram of STDP paired with empirical data showing the change in amplitude of the excitatory post-synaptic current (EPSC) produced by a synapse impinging on a neuron after repeated external stimulation of a pre- and post-synaptic neuron so as to force spikes in the two neurons with specific temporal differences. For each pair the stimulation protocol was repeated 60 times at a rate of 1 Hz and changes in synaptic efficacy were measured at 20 minutes after the protocol [10]. A distinct shape can be seen whereby synapses where the pre-synaptic neuron was forced to spike immediately before its post-synaptic partner experienced the greatest increase in strength with this increase dropping off exponentially with an increase in temporal difference between the pairs. Likewise pairs ordered such that the post-synaptic neuron fired first experienced a decrease in efficacy, which was most extreme if paired closely and also fell of exponentially with distance.

Chapter 19

Reservoir Networks

Zoë Tosi, Jeff Yoshimi

Another approach to training recurrent networks is to, in essence, *not train them at all*, using a technique known as **reservoir computing**. RNNs offer a suite of advantages over their feed-forward cousins owing to their potential ability to store information about past stimuli which is made possible by recurrent connectivity. Because feedback connections exist, the current state of the nodes in the recurrent portion of the network (and any information it contains about prior inputs or states) becomes part of the input in determining the next state of those nodes. They become ideal, then, for processing time-series data (as opposed to simple pattern association tasks).

However, the kind of internal dynamics which make RNNs desireable (they have a "life of their own") also makes them difficult to effectively use. Throughout the 1980s and 1990s the problem of how to properly train RNNs prevented them from being used on a wider scale. How exactly the recurrent weights should be altered in response to an error signal was largely unclear. In 2001 two separate researchers, Herbert Jager and Wolfgang Maass, a computer scientist and neuroscientist respectively, independently came to the conclusion that the reverberating internal dynamics within the recurrent portion of a network could be harnessed *without training the recurrent weights* so long as certain constraints were imposed on the networks in question. That is, instead of formally training the recurrent weights and trying to calculate what node activations should be in a likely chaotic system, why not bypass the issue entirely by instead simply focusing on putting the recurrent portion into a *helpful dynamical regime*. But what would such a regime look like? Does such a thing even exist? If so, how does one go about imbuing a RNN with it?

The answers to these questions can be found in a humble pond of water. Suppose you're standing in front of a pond on a cool autumn day. There is no wind, the multitude of water-dwelling insects of summer have gone into hiding for the winter, and the fish have no reason to breach the surface. It is in fact so placid that the pond could be mistaken for an oddly shaped mirror reflecting the sky above. You notice a few pebbles by your feet and, perhaps uncomfortable with just how still this scene is before you, you decide to pick one up and toss it into the water. It makes a small splash and ripples emanate in a ring from where the pebble plopped down. The ripples eventually reach the edge of the pond and are reflected back toward the center, the tiny waves interfere with one another creating a striking pattern given the former stillness of the pond. But as they continue to reverberate across the pond's surface they become smaller, less defined, and eventually the pond returns to its placid state–as if you'd never thrown the pebble at all.

You decide to toss in another pebble, but this time you throw a little harder. As before, the pebble plops into the pond, but this time significantly further away. Once again, ripples emanate from the place where the pebble dropped, but this time–being as it is, closer to the other side of the pond–some of the ripples reach the water's edge opposite you sooner, reflect back, and after a few moments you realize the patterns of ripples on the surface this time are *completely different* than before. You throw another stone... this time it lands mere centimeters from where the original pebble did. The ripples radiate outward once more and as far as you can tell are nearly indistinguishable from those that came from the first pebble you threw. The ripples reflect back, slightly differently from the first time, but to your surprise after mere moments the pattern of ripples on the water's surface is different from the ones produced by either of the two previous

stones, though much closer to the pattern produced by the first stone. This is because when you drop a stone into the pond the perturbation to the water's surface is *unique* to the location where the stone was dropped. Proximal drop locations produce similar patterns of ripples that remain more similar for longer amounts of time, while spatial distant ones are immediately completely different. Note the following: You did not have to do anything to this pond for this to be the case; it is simply *an intrinsic property* of fluids. It arises from the fact that the pond is 1) responsive to external perturbation, 2) all perturbations eventually die out, with the pond returning to its initial unperturbed state, and 3) that the dynamics of the system (water molecules and surface tension) respond consistently to the same perturbation yet are complex enough that different perturbations produce different states. This doesn't just apply to where a single stone has been dropped into the pond, for if one stone was thrown right after another the pattern of ripples would be unique to *where and when* the two stones were thrown relative to one another, with the effects of the more recently dropped stone taking longer to dissipate than those of its earlier counterpart. This means that the pattern of ripples on the surface of our pond at any given time actually represents (i.e. contains information concerning) the *history* of perturbations to the pond! An example of this representational capacity was concretely demonstrated in [29] where a literal bucket of water was used for nonlinear pattern separation 19.1.

Suppose you come back to the pond in the dead of winter. Now when you toss a stone at the pond they create tiny ablations on the hard surface, but unlike when the surface was liquid, the points of impact do not change over time. There is no way to tell how long ago you dropped the stones or in what order because the system doesn't respond meaningfully to external perturbations beyond the initial point of impact. Any pattern of marks on its surface could've arisen at any point in time after it froze and in any order. The ice simply sits there, solid, unchanging, and impregnable. The surface of the pond has lost its spatiotemporal representational capacity by being frozen–perfect for ice skating–though perhaps less interesting in and of itself. Dynamically this is referred to as an *ordered* regime.

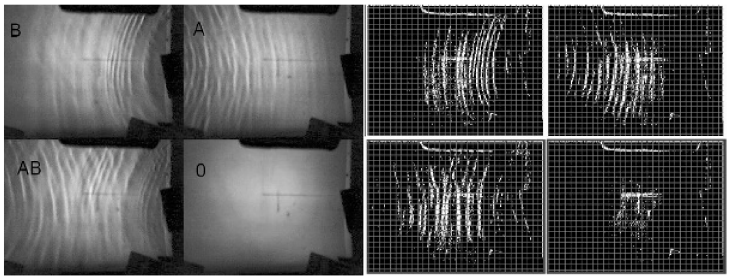


Figure 19.1: Pattern recognition has literally been done in a bucket of water, providing a concrete example of the water metaphor. Ripples on the surface were recorded in response to labelled inputs (perterbations emanating from the left or right side of the bucket, and when combined with a perceptron classifier was able to perform nonlinear pattern separation (XOR). Retrieved from [29].

After your afternoon of ice-skating, you once again brave the harsh, icy winds, and return home. Hungry, cold, and tired, nothing sounds better than a nice bowl of soup. You head to the kitchen, grab a pot, fill it with water, and then place it atop the stove. Setting the burner to high you stand in front of the pot, warming yourself while waiting patiently for the water to boil. Initially it's as still as the autumn pond. You tap the water's surface with a wooden spoon and watch as the ripples emanate from the point of contact, peacefully expanding out, hitting the walls, reflecting, interfering, and eventually dying out exactly as you remember. As time goes by, little bubbles form on the bottom of the pot, turning to larger bubbles, and before you know it the water in the pot has been brought to a rolling boil. Its surface constantly bubbles, always moving–a stark contrast to its prior stillness. You look at the complex and ever changing structure on the surface of the boiling water, and realize just how unlikely it is that at any previous time the water looked *exactly* as it does now. You forcefully, but carefully drop in the bottom of your spoon. You can see small waves from the point of contact, but only for a brief instant as they are fully consumed and incorporated into the surrounding disorder. Curious, you aim your spoon well and make contact with the surface just as forcefully before and as close as you could to the last place where you dropped the spoon. To no one's surprise once again small waves are created which quickly dissipate into the chaos of the churning, bubbling,

liquid in the pot. Consistent with your realization from before, it is apparent that despite dropping in your spoon with the exact same amount of force and in the exact same position, the state of the surface of the water before you has likely never looked exactly like this before and is *completely different* from its state in the same interval of time after your first spoon drop. The water, in its excited state, has taken on a life of its own–one which incorporates your perturbations, but which is unpredictable–assigning completely unique states to even the same perturbation. The water has lost it's spatiotemporal representational capacity by boiling. Dynamically this is referred to as the *chaotic* regime, and just as with the *ordered* regime it is largely useless from the perspective of reliably representing past inputs.

If the boiling pot was in a chaotic regime and the frozen pond was in an ordered regime, then how would we describe the liquid surface of the pond all those months ago? The answer is: the *edge of chaos*, a dynamical regime where the system is affected by perturbations, but the differences between the states produced by the perturbations and the perturbations themselves remain largely constant. Figure 19.2 gives us an example of the sorts of activity we expect to find as well as the conditions necessary to produce their respective dynamical regimes in an actual neural network as opposed to liquid (or solid) water.

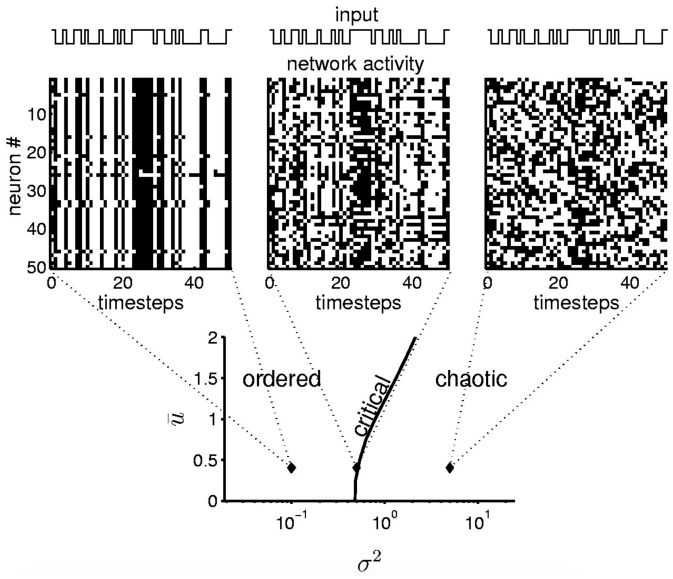


Figure 19.2: Here we have an example of ordered, chaotic and edge-of-chaos dynamical regimes in a recurrent neural network as well as what portions of the network's parameter space in terms of the variance of weights (σ^2) and the mean value of the input signal ($\bar{\mu}$) Notice the similarities in the network's responses to an input signal (top row) and the pond metaphor used previously. The left-hand side gives us a network that presents no or very little information on time-series and is only capable of representing the current input. For the toy network used in [8] the ordered regime occurs when the variance of the weights is low and/or the input signal is strong. The right-hand side gives us a network whose activity is dominated by its own internal dynamics, which is not consistently or obviously responsive to the input time-series. Likewise this dynamical regime is produced by high variance among the recurrent weights and typically weaker input signals.The middle plot represents the ideal dynamical regime for these networks because it responds to the input signal, but also has some autonomous internal dynamics which are capable of preserving information about previous entries in the time-series. Networks with this property occupy a specific manifold in the parameter space of variance on the weights and strength of the input signal. Image retrieved from [8].

This representational capacity comes "for free" as it were, given the nature of the material in question. What if we could imbue recurrent neural networks with those same properties, i.e. the intrinsic ability to represent perturbations as transients? Instead of peaks and troughs across the surface of water, could not the same sort of thing be accomplished with activity across a recurrent network? What sorts of constraints would need to be placed on the network for such a thing to be possible? What properties must it possess and how can we describe those properties? These are the questions that Wolfgang Maass and Herbert

Jaeger independently asked themselves. Instead of training recurrent weights to reduce error, the problem then was how to "tune" the network such that it would possess these properties, thus requiring no explicit training, beyond a linear classifier which would simply learn to distinguish different patterns of activity in the so-called dynamical reservoir. In fact Wolfgang Maass found that recurrent spiking neural networks held to some minimal biological constraints actually possessed this property intrinsically, while Herbert Jager was able to derive a set of theorems for constraints on the recurrent weight matrix in a connectionist-style network which guaranteed a fading memory and therefore this property (called "the echo-state property" by Jaeger).

There are two main kinds of reservoir networks: echo state networks and liquid state machines (the basic difference is that liquid state machines use spiking neurons). A picture of an echo state network is shown in Fig. 19.3. The idea is to take a recurrent network (the "reservoir") and see what we can get it to do by driving it with inputs. Reservoir computers are somewhat neurally realistic: the central reservoir is a recurrent network, like the kind of networks found in the brain, and the brain's recurrent networks are typically driven by inputs from other world or from the external world.

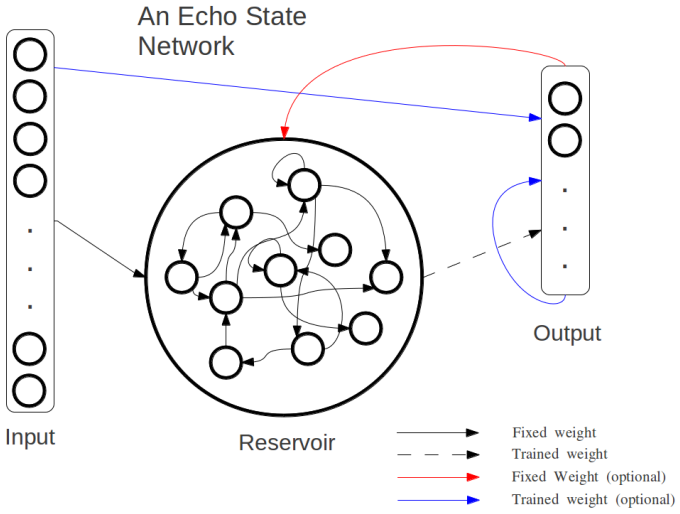


Figure 19.3: An echo state network, which is one kind of reservoir computing network.

The input nodes of an echo state network or liquid state machine drive a reservoir, which is an arbitrary recurrent network, like one of the networks discussed in chapter 10. Recall that those networks produce all kinds of interesting dynamical patterns, *e.g.* n-cycles of various lengths (2-cycles, 10-cycles, 100-cycles) and even chaotic behaviors. Recall that unfolding patterns in a recurrent network correspond to trajectories in an activation space. Here we have trajectories in the activation space of the reservoir. The inputs change which trajectory the system is following.[1] The trajectories induced by the input stream produce states that can then be classified using the output layer, a "readout", which is trained using LMS or a related algorithm.

Again we are just reusing existing ideas, combining a recurrent network with a few feed-forward networks and training some of the weights using the least mean squares algorithm. We are basically training a network to associate reservoir states with desired readout states, and thereby to associate trajectories in the reservoir activation space with trajectories in the output space.[2] Reservoir networks can be used to classify temporally

[1] The reservoir's activity at any moment can be thought of as a high dimensional spatial representation of a lower dimensional time-series over some interval which constitutes its effective memory capacity. Recall that we focused before on dimensionality *reduction*. Here we are going from lower to higher dimensions, which can be useful for classifying temporally extended input streams. States that are not linearly separable in a lower dimensional space can often be separated by a hyperplane through a higher dimensional one. That is the dynamical reservoir can be thought of as taking nonlinearly separable lower-dimensional time-series and projecting them into a higher dimension where they are linearly separable. Hence the only training that is required is the rather efficient training of output weights.

[2] Our training dataset $D = (I, T)$ has the same form as usual, but behind the scenes we can use it to create a second training

extended inputs (*e.g.* say what kind of music is being played) or to generate a particular type of input (*e.g.* the input is a frequency and the output is a sine wave at that frequency).

A primary goal of reservoir computing is to understand the computational power of the kinds of recurrent networks we see in brains. This is useful for the scientific project of understanding what the theoretical properties of brain networks are. It's also a research project in machine learning. It may be that reservoir computers end up having advantages over other trained recurrent networks, *e.g.* supervised recurrent networks, which we discuss next (which can be very computationally demanding). This is because they use very simple components. The output is just a linear classifier (trained using a variant of LMS) which is faster to use than supervised recurrent methods.[3].

dataset, $D' = (I', T')$, where $T = T'$, and where the ith input vectors in I' corresponds to the states produced in the reservoir when it is exposed to the corresponding ith input vector in I.

[3]The problem, however, is that it's not clear how one goes about making a good reservoir, which is an active research topic

Chapter 20

Glossary

Action potential The rapid change in the membrane potential of a neuron, caused by a rapid flow of charged particles or ions across the membrane that occurs during the excitation of that neuron.

Activation Value associated with a node. Has different interpretations depending on the context. It can, for example, represent the firing rate of a neuron, or the presence of an item in working memory.

Activation function Function that converts weighted inputs into an activation in some node updated rules.

Activation space The set of all possible activation vectors for a neural network.

Activation vector A vector describing the activation values for a set of nodes in a neural network.

Anterograde amnesia A type of memory loss or amnesia caused by brain damage in the hippocampus, where the ability to create new fact-based or declarative memories is compromised after the injury.

Artificial neural network (Acronym: ANN) a collection of interconnected units, which processes information in a brain-like way.

Attractor A state or set of states with the property that any sufficiently nearby state will go towards it. Fixed points and periodic orbits can be attractors.

Ataxia Impaired coordination or clumsiness caused by neurological damage in the cerebellum.

Auditory cortex Regions of the temporal lobes of the brain that process sound.

Auto-associator A pattern associator that learns to associate vectors with themselves. In a recurrent network this can be used to model pattern completion. In a feed-forward network this can be used to test whether an input can be represented in a compressed form in the hidden layer and then recreated at the output layer.

Automatic process A cognitive process that not require attention for its execution, and is relatively fast. Examples include riding a bike, driving a car, and brushing your teeth.

Axon The part of the neuron that carries outgoing signals to other neurons.

Bag of words : A representation method that associates a document with a vector of word frequencies, where each entry of the vector corresponds to a word and the value of the entry corresponds to the number of times that word occurs in the document. The underlying idea is that the word usage frequencies would capture the (semantic) content of a document. The order and grammatical information of the words in the document is disregarded, hence the term "bag". A form of document embeddings, as contrasted with token or word embeddings, are more common with neural networks.

Backpropagation (Synonyms Backprop): A supervised learning algorithm that can train the weights of a multi-layer network using gradient extent. Can be thought of an extension of Least Mean Square methods for multi-layer networks.

Basal ganglia A structure below the surface of the cortex (subcortical) that is involved in voluntary action and reward processing.

Basin of attraction The set of all states in a state space that tend towards a given attractor.

Basis A linearly independent set of vectors that span the whole vector space. Any two bases for a vector space have the same number of vectors.

Bias A fixed component of the weighted input to a node's activation. Determines its baseline activation when no inputs are received.

Bifurcation A topological change that occurs in a dynamical system as a parameter is varied.

Binary vector A vector all of whose components are 0 or 1.

Biological neural network A set of interconnected neurons in an animal brain.

Bipolar vector A vector all of whose components are -1 or 1.

Boolean functions Functions that take a list of 0's and 1's as input and produce a 0 or 1 as output. The 0 represents a "False" and the 1 represents a "True". Boolean functions can be realized by logic gates.

Brain stem The lowest part of the brain that connects to the spinal cord and is fundamental for breathing, heart rate, and sleep.

Broca's area A region of the frontal lobe associated with language production, gesture, and speech. Damage to this area can lead to Broca's aphasia, affecting speech fluency.

Categorical data Data that can take one of a discrete set of values. For example, the time of day can be treated as a categorical variable taking two values: day and night. Also called nominal data.

Cerebellum A structure below the surface of the cortex (subcortical) involved in balance and fine movements, as well as maintaining internal models of the world for motor control.

Cerebral cortex The outer layer of the brain characterized by its structural folding (gyri and sulci); underlies complex behavior and intelligence in higher animals.

Chaotic dynamical system A type of dynamical system in which the future behavior of the system is hard to predict. Such systems have sensitive dependence on initial conditions. Compare the "butterfly effect."

Clamped node A node that does not change during updating. Any activation function associated with the node is ignored, and its activation stays the same.

Clamped weight A weight that does not change during updating. Any local learning rule associated with the weight is ignored, and its strength stays the same.

Classification task A supervised learning task in which each input vector is associated with one or more discrete categories. An example would be classifying images of faces as male or female. When classification associates each input with one category only, a one-hot encoding is often used on the output layer.

Column vector A vector whose components are written in a column *e.g.* $\begin{pmatrix} 2 \\ 1 \\ 3 \end{pmatrix}$.

Competitive learning A form of unsupervised learning in which outputs nodes are trained to respond to clusters in the input space.

Component One entry of a tensor. For example, in the vector $(1, 2, 3)$ the second component is 2.

Computational neuroscience The study of the brain using computer models.

Computational cognitive neuroscience The use of neural networks to simultaneously model psychological and neural processes.

Connectionism The study of psychological phenomena using artificial neural networks.

Context window The set of tokens that can be converted to vectors using a word embedding and sent as input to a transformer-based LLM. The context window encompasses both user prompts and system responses; that is, both sides of a "conversation" with an LLM. Larger context windows allow LLMs to produce more coherent outputs and take account of more of a conversation.

Convolutional layer A special kind of weight layer where a set of weights (a "filter") is scanned over a previous node layer to produce activations in a target layer. This is related to the mathematical operation of convolution.

Controlled process A cognitive processes that requires attention for its execution, and is relatively slow. Examples include solving math problems, doing homework, or performing an unusual task that you have not practiced.

Cortical blindness Neurological impairment caused by damage to the occipital lobe that results in blindness or inability to see. This can occur without any damage to the eyes.

Cross talk A phenomenon where training patterns interfere with one another when training a neural network to perform some task.

Data cleaning Removing, fixing, replacing, or otherwise dealing with bad data. Includes subsetting data, i.e. extracting rows or columns or removing rows or columns. One stage of data wrangling.

Data science An area of science and practice concerned with managing and analyzing datasets, often using tools of machine learning, including neural networks.

Data wrangling (Synonyms Data munging, pre-processing): the process of transforming data into a form usable by a neural network. Encompasses obtaining, cleaning, imputing, coding, and rescaling data.

Dataset Any table of numerical values that is used by a neural network, or that will be used by a neural network after pre-processing. (This is not a standard definition, but one stipulated in this text). Input, output, target, and training datasets are specific types of tables used in specific ways by neural networks.

Decision boundary In the context of a classification task, a hypersurface (e.g., in 2 dimensions, a line) that divides an input space into decision regions. Each decision region is associated with one possible output.

Decision region In the context of a classification task, a region of the input space associated with a specific class label. Any input that is in that region produces an output corresponding to that regions class label.

Deep network A neural network with a large number of successive layers of nodes mediating between inputs and outputs. Deep networks are trained using *deep learning* techniques.

Dendrite The part of the neuron that receives signals from other neurons.

Dendritic spine Small outgrowths on the end of a dendritic branch where the receptors to which neurotransmitters attach can be found.

Dimension of a vector space The number of vectors in a basis for a vector space. This equals the number of components the vectors have. Examples: the line is a 1-dimensional vector space; the plane is a 2-dimensional vector space.

Dimensionality reduction A technique for transforming an n-dimensional vector space into another vector space with $m < n$ dimensions. A way of visualizing higher than 3-dimensional data that would otherwise be impossible to visualize.

Discriminative model A model that associates feature vectors, which are often distributed representations, with discrete categories (e.g. one-hot localist vectors). Categories can be discriminated from distributed feature vectors. This is a non-standard, informal definition. The formal definition is that a discriminative model is a model of the conditional probability of categorical outputs given inputs. Contrasted with generative models.

Distributed representation A representation scheme where patterns of activation across groups of neurons indicate the presence of an object.

Dorsal stream Pathway that extends from the occipital lobe into the parietal lobes, underlying visuospatial processing of visual objects in space. Damage to this pathway can cause impairment in reaching and grasping for objects.

Dot product The scalar obtained by multiplying corresponding components of two vectors then adding the resulting products together. Example: $(1, 2, 3) \bullet (0, 1, -1) = 0 + 2 - 3 = -1$.

Dynamical system A rule that says what state a system will be in at any future time, given any initial condition.

Environment A structure that influences the input nodes of a neural network or is influenced by the output nodes of a network, or both.

Error function A function that associates a network and a training dataset with a scalar error value. Many supervised learning techniques attempt to modify network parameters so as to reduce the error function. Also called a "loss function" or, in the context of mathematical optimization, an "objective function."

Error surface The graph of a function from parameters values of a network to error values of an associated error function. Each point on an error surface corresponds to different parameters (usually weight values and biases) of a network. Gradient descent finds minima on an error surface, where error is relatively low.

Example (Synonyms Instances, cases): rows of a dataset. Used in phrases like input example, training example, etc., depending on which dataset we are considering.

Excitatory synapses Synapses where an action potential in a presynaptic neuron triggers the release of neurotransmitters that then increase the likelihood of an action potential occurring in the postsynaptic neuron.

Evolutionary algorithm An algorithm that creates a model based on simulated evolution. In the context of neural networks, or "evolved neural networks", a set of randomly generated neural networks is created and they are used to perform some task. Those that perform best are kept and combined with other top performers, and the resulting networks are used to perform the same task. The process is repeated over many generations. Closely related to genetic algorithms.

Fan-in weight vector The weight vector for all of the inputs to a node in a neural network.

Fan-out weight vector The weight vector for all of the outputs from a node in a neural network.

Feature (Synonyms Attribute, property): a column of a dataset, often associated with a node of a neural network.

Feature Map A node layer that is the output of a convolutional layer, in which each activation is the result of a filter being multiplied (using the dot product) to one region of the input layer.

Feature-extraction (Synonym Coding): process of translating non-numerical data (e.g. text, images, audio files, DNA sequences) into a numerical format.

Feed-forward network A network comprised of a sequence of layers where all neurons in any layer (besides the last layer) are connected to all neurons in the next layer. Contains no recurrent connections.

Firing rate number of spikes (action potentials) A neuron fires per unit time. Usually measured in hertz, that is, number of spikes per second. A higher firing rate corresponds to a more "active" neuron.

Filter (Synonym Kernel): the set of weights in a convolutional layer. This is more or less the same thing as a convolutional layer, but it refers specifically to the weights, whereas "convolution" also refers the process of passing the filter over the input activations.

Filter Bank A set of filters. One kind of volume-to-volume layer in a convolutional neural network.

Fixed point A state that does not change under a dynamical system. The system "stays" in this state forever. An orbit consisting of a single point.

Flatten The process of converting a tensor with rank 3 or greater to a vector. Allows the output of a convolutional layer to be sent to a standard feed-forward node layer.

Frontal lobe Forward-most lobe of the brain whose many roles include decision-making, action planning, and executive control. Houses many important regions, including prefrontal cortex (PFC), orbitofrontal cortex (OFC), and motor cortex.

Generalization The ability of a neural network to perform tasks that were not included in its training dataset. An example would be a network that was trained to identify 10 faces as male, and 10 as female, being able to perform well on (or "generalize to") new faces it has not seen before.

Generative AI Artificially intelligent systems (often neural networks) that are capable of creating text, images, video, and other content, often at a level that is passably human.

Generative Model A model that can be used to generate prototypical features associated with some category, for example, associating a localist category label with a distributed feature vector. This is a non-standard definition. The formal definition is that it is a model of the joint probability distribution over a set of inputs and outputs. Contrasted with discriminative models.

Graceful degradation A property of systems whereby decrease in performance is proportional to damage sustained. The contrast is with brittle systems, in which a small amount of damage can lead to complete failure.

Gradient descent A technique for finding a local minimum of (in a neural network context) an error function. Network parameters are iteratively updated using the negative of the gradient of the error function, which can be thought of as an arrow pointing in the direction in which the error surface is dropping most rapidly.

Hemineglect Neurological impairment caused by damage to regions of the parietal lobes characterized by a lack of attending to anything in one half of the visual field.

Hippocampus A structure below the surface of the cortex (subcortical) that is involved in long-term memory consolidation and spatial maps. Damage to this structure can cause memory loss, or amnesia.

Hyperparameter In the context of neural networks, a parameter that is not updated while a learning algorithm is applied. A learning rate is a hyperparameter, as is the size of a hidden layer, because both are set prior to training and are not updated during the training process.

IAC network A neural network used to model human semantic memory by spreading activation between pools of mutually inhibitory nodes that implement a winner-take-all or competitive structure. Weights are set by hand.

Inhibitory synapses Synapses where an action potential in a presynaptic neuron triggers the release of neurotransmitters that then decrease the likelihood of an action potential occurring in the postsynaptic neuron.

Imputation The process of filling-in missing data in a data set. One stage of data wrangling.

Initial condition The state a dynamical system begins in.

Input dataset A dataset whose rows correspond to input vectors to be sent to the input nodes of a neural network.

Input node (Synonym Sensor): a node that takes in information from an external environment.

Input space The vector space associated with the input layer of a neural network. The set of all possible input vectors for a neural network.

Labeled dataset A conjunction of two datasets: an input dataset with input vectors, and a target dataset with target vectors or "labels". An input-target dataset. Used for supervised learning tasks.

Large language model (LLM) A model that generates language based on a large dataset. LLMs are typically implemented as using the transformer architecture.

Learning In a neural network, a process of updating synaptic weights so that the network improves at producing some desired behavior relative to an error function or other objective function.

Learning rate A value that controls how much parameters are updated each time a learning rule is applied. Lower values lead to slower learning; larger values to faster learning.

Learning rule (in neural networks) A rule for updating the weights of a neural network. Application of this rule is sometimes called "training."

Least mean square (Synonym Delta rule): a supervised learning algorithm that adjusts weights and biases of a 2-layer feed-forward network so that input vectors in a training dataset produces outputs as similar as possible to corresponding target vectors.

Linear activation function A function that is typically just the weighted input, sometimes scaled by a slope parameter. A piecewise linear activation function clips weighted input at upper and lower bounds. The ReLU function only clips at a lower bound of 0.

Linear combination To make a linear combination from a set of vectors we multiply each vector in the set by a scalar and then we add up the resulting vectors.

Linearly dependent A set of vectors is linearly dependent if there is at least one vector in the set can be expressed as a linear combination of the other vectors in the set.

Linearly independent A set of nonzero vectors that is not linearly dependent is linearly independent.

Linearly inseparable A classification task that is not linearly separable.

Linearly separable A classification task can be solved using a decision boundary that is a line (or, in more than 2-dimensions, a plane or hyperplane).

Logic gates Devices that compute Boolean functions. For example, an AND gate has two inputs and one output. When both inputs are set to "True" the gate produces a "True" as output; otherwise the gate produces a "False" as output. Simple neural networks can implement logic gates.

Localist representation A representation scheme where activation of individual neurons indicate the presence of an object. Example: activation of neuron 25 indicates the presence of my grandmother.

Long Term Depression (LTD) A process by which the efficacy of a synapse is decreased after repeated use.

Long Term Potentiation (LTP) A process by which the efficacy of a synapse is increased after repeated use. LTP is part of the basis of the Hebb rule.

Machine learning The use of statistical techniques to produce artificial intelligence. Uses of neural networks as engineering devices are a kind of machine learning.

Matrix A rectangular table of numbers. Often used to represents the weights of a neural network.

Membrane potential The voltage that results from the difference in the total sum charge of ions on either side of the cell membrane. A cell at rest typically has a resting membrane potential of -70 mV.

Motor cortex Region of cortex that resides in very rear-most part of the frontal lobes that is responsible for the planning and execution of movement.

n-cycle A finite set of n states that a discrete-time dynamical system visits in the same repeating sequence. For discrete-time dynamical systems periodic orbits are n-cycles.

Neuron A cell in the nervous system specialized to transmit information.

Neurotransmitters Small chemical packages that transmit signals from one neuron to another via synapses. These packages are released when an action potential in a pre-synaptic neuron stimulates their release from vesicles on the axon terminals into the synaptic cleft where they travel to receptors on the dendrites of a post-synaptic neuron.

Node (Synonyms Unit, artificial neuron): a simulated neuron or neuron-like element in an artificial neural network.

Node Layer A collection of nodes that are treated as a group. For example, in a feed-forward network every node in one layer can be connected to every node in another layer. The activations in a node layer can be represented with an activation vector. Without qualification, "layer" means node layer.

Numerical data Data that is integer or real-valued. Examples include age, weight, and height. Data for a neural network must usually be converted into a numerical form.

Occipital lobe The lobe located in the back of the cortex where visual processing primarily takes place.

One-hot encoding A one-of-k encoding technique in which a category with k values is represented by a binary vector with k components and the current value of the category corresponds to which nodes is on (or "hot"). Example: representing cheap, moderate, and expensive restaurants with vectors $(1, 0, 0)$,$(0, 1, 0)$ and $(0, 0, 1)$. One-hot encodings are orthogonal t each other.

Optimization The process of finding the maximum or minimum of a function. In neural networks, it is often used to find network parameters for which error is lowest.

Orbit (Synonym Trajectory): the set of states visited by a dynamical system from an initial condition.

Orthogonal Two vectors are orthogonal to each other if their dot product is zero. One-hot vectors are orthogonal. They are widely separated in input space and tend not to produce cross-talk in learning tasks.

Orbitofrontal cortex Front-most region of prefrontal cortex associated with decision-making.

Output dataset A dataset whose rows correspond to output vectors recorded from a neural network.

Output node (Synonyms Actuator, effector): a node that provides information to an external environment.

Output space The vector space associated with the output layer of a neural network. The set of all possible output vectors for a neural network.

Padding Entries added to an input in a convolutional layer in order to deal with issues relating to the edges of inputs. For example can be used to ensure width and height of the output remain the same.

Parallel processing Processing many items at once, concurrently. Contrasted with serial processing, where items are processed one at a time. Neural networks are known for processing items in parallel, whereas classical computer process items in serial.

Parameter A quantity for a dynamical system that is fixed as the system runs but can be adjusted and run again with a different value. Used in the description of bifurcations. In a neural network, the parameters we update are usually its weights and biases. This concept is also used in machine learning when treating a neural network as a trainable model, whose parameters (weights and biases) are updated using optimization techniques.

Parietal lobe The lobe located behind the frontal lobe and above the occipital lobe. This part of cortex plays a role in processing spatial information, integrating multisensory information, and is home to the somatosensory cortex, which processes information about touch sensation.

Pattern associator A neural network that associates each input vector in a set of input patterns with an output vector in a set of output (or "target") patterns. In most cases a pattern associator can be thought of as a vector-valued function.

Pattern classifier A pattern associator in which the output nodes are two-valued and are [interpreted as representing category membership. Example]: when the output node of a network is 1, this means it's seeing a male face, when it is 0 this means it's seeing a female face.

Performance Consideration of how a network responds to inputs, while keeping its weights fixed. Often the term is also used to consider how well it is doing relative to an error function or objective function.

Period of a periodic orbit For a continuous time dynamical system the period is the time it takes the dynamical system to cover the periodic orbit. For a discrete time dynamical system the period is the number of points in the periodic orbit.

Periodic orbit A set of points that a dynamical system visits repeatedly and in the same order. An n-cycle is a type of periodic orbit.

Phase portrait A picture of a state space with important orbits draw in it. A picture of the dynamics of a system.

Pooling layer A volume-to-volume layer in a convolutional layer which reduces the amount of information passing through the network, ideally without altering the essential structure of that information. Related to subsampling and downsampling.

Positional encoding : A method used in transformer models to add information about the position of tokens in a sequence to an activity pattern. This is important because transformers process information in parallel without knowledge of where a token occurs in a sequence.

Pre-processing The process of transforming data into a form usable by a neural network. Compare data-wrangling.

Prefrontal cortex The front-most part of the frontal cortex, which is involved in executive function, decision-making, and planning. It is also thought to have an attractor-based structure that supports the operation of working memory.

Primary motor cortex Strip in the motor cortex that houses a somatotopic map of the body and controls simple movement production.

Proposition (Synonyms Statement, sentence): an expression that can be true or false.

Projection A way of representing a group of points in a high-dimensional space in a lower dimensional space.

Prosopagnosia An impairment in recognizing faces that results from damage to particular regions of the ventral stream.

Recurrent network A network whose nodes are interconnected in such a way that activity can flow in repeating cycles.

Receptors Binding sites at the ends of dendrite branches of the post-synaptic neuron where neurotransmitters attach.

Regression task A supervised learning task in which the goal is to create a network that produces outputs as close as possible to a set of target values. Targets are real-valued rather than binary (as they often are in classification tasks). An example would be predicting the exact price of a house based on its features.

Reinforcement learning A form of learning in which a system learns to take actions that maximize reward in the long run. Actions that produce rewards, or action that lead to actions that produce reward, are reinforced in such a way that agents learn to obtain rewards and avoid costly situation. In humans, associated with circuits in the brain stem and basal ganglia. Sometimes treated as a third form of learning alongside supervised and unsupervised learning.

ReLU A linear activation function that is clipped at 0. It's activation is 0 for weighted inputs less than or equal to 0, and it is equal to weighted inputs otherwise. It is a popular activation function for deep networks. Note that "relu" is short for "rectified linear unit".

Repeller (Synonym Unstable state): a state or set of states R with the property that if the system is in a nearby state the system will always go away from R. Fixed points and periodic orbits can both be repellers.

Representational depth The number of node layers in a feed-forward network. More generally a description of the number of layer-like structures stacked in a neural network. Deeper networks can produce more complex representations which aggregate representations of earlier layers. (The term "depth" also refers to a way of describing one component of the size a tensor, as in depth-by-width-by-height. That is a separate, unrelated use of the term).

Representational width The number of nodes in a node layer of a feed-forward network. More generally a description of the representational capacity of a layer or layer-like structure in a neural network. Wider layers can capture more features of the previous layer. This is non-standard terminology adopted in this book as a useful organizing principle. (The term "width" also refers to a way of describing one component of the size a tensor, as in depth-by-width-by-height. That is a separate, unrelated use of the term).

Rescaling A mathematical transformation of a set of samples in a dataset that preserves their relations to one another but changes their values. Often values are rescaled to lie in the interval $(0, 1)$ or $(-1, 1)$. One stage of data wrangling.

Retinotopic map A topographic map of locations in the retina. Regions of the brain that are retinotopic maps have the property that neurons near one another process information about nearby areas in visual space.

Row vector A vector whose components are written in a row *e.g.* $(2, 1, 3)$.

Scalar Usually a real number but in some applications it can be a complex number.When we multiply a vector by a scalar we are "rescaling" the vector, *i.e.* changing the vector's length without changing its direction.

Scalar multiplication An operation used to "rescale" a vector. It takes a scalar and a vector and returns a vector with the same direction.

Self Organizing Map (Acronym SOM): A network trained by unsupervised competitive learning, in which the layout of the output nodes corresponds to the layout of the input space.

Sigmoid activation function An activation function that whose value increases monotoically between a lower and upper bound. As the input goes infinitely far in the positive direction the value converges to the upper bound. As the input goes infinitely far in the negative direction the value converges to the lower bound.

Soma (Synonym Cell body): the central part of a neuron, which the dendrites and axons connect to.

Softmax An activation function which normalizes inputs so that its activations in a layer (or in simbrain, a neuron group) can be interpreted as a probability distribution. Each activation is between 0 and 1 and the sum over all the activations in a softmax layer is 1.

Somatotopic map A topographic map in the somatosensory cortex that is organized by areas of the body. Nearby regions of this area represent nearby parts of the body.

Somatosensory cortex Front-most region of the parietal lobe that houses a somatotopic map of the body parts and processes tactile information from the body.

Span The set of all linear combinations of a set of vectors is called the span of that set of vectors.

Spike A discrete event that models the action potential for a neuron.

State A specification of values for all the variables describing a system. The state of a neural network is typically an activation vector.

State space The set of possible states of a system. The state spaces we consider are vector spaces. Two specific state spaces we focus on are activation spaces and weight spaces.

State variable A variable associated with a dynamical system that describes one number associated with a system at a time. Examples include a person's height and weight, a particle's position and momentum, and a neuron's activation. The *state* of a system is a vector each of whose components is the value of one state variable.

Strength A value associated with a weight. Has different interpretations depending on the context. It can, for example, represent the efficacy of a synapse, or an association between items in memory.

Stride In a filter bank or pooling layer, the number of pixels the filter or kernel or pooling window is moved when it is scanned across its input.

Sparse matrix A matrix in which most of the entries are 0. A sparse weight matrix represents a set of connections between nodes where most of the possible connections are missing. Contrasted with dense matrices and dense or all-to-all connectivity.

Sparsity Of a matrix is a number between 0 and 1 obtained by counting how many zero entries the matrix has and dividing by the total number of entries. A matrix with no zero entries has a sparsity of 0 and a matrix with all zero entries has a sparsity of 1.

Subspace Any subset of a vector space that also happens to satisfy the definition of a vector space. The sum of any two vectors in a subspace is in the subspace and any scalar multiple of a vector in a subspace is in the subspace.

Supervised learning A learning rule in which weights are adjusted using an explicit representation of desired outputs.

Synapse The junction between nerve cells where a information is transferred from one neuron to another.

Synaptic efficacy The degree to which a pre-synaptic spike increases the probability of a post-snyaptic spike at a synapse.

Target dataset A dataset whose rows correspond to target outputs we'd like a neural network to produce for corresponding input vectors. For classification tasks, a set of *class labels*.

Temporal lobe Lobe forward of the occipital lobe and below the parietal and frontal lobes. This region is involved primarily in processing semantic information about what things are and factual information, and also houses several important language areas.

Temporal lobe Lobe forward of the occipital lobe and below the parietal and frontal lobes. This region is involved primarily in processing semantic information about what things are and factual information, and also houses several important language areas.

Tensor A generalization of the concept of a vector that encopasses numbers, lists of numbers, matrices, sets of matrices, sets of these sets, etc. The rank of a tensor is the number of indices it takes to specify an "entry" in it. A number is rank 0 because it requires no indices. A vector is rank 1 because it takes one index to specify an entry in a vector. A matrix is rank 2, because it takes two numbers to specify an entry (a row and column). A set of matrices is rank 3, because it takes 3 indices to specify an entry. Etc.

Thalamus An internal brain structure that relays information from sensory and motor structures to the cortex.

Threshold potential The membrane potential of the cell above which an action potential is fired.

Threshold activation function A function that has one value for input at or below a fixed amount (the threshold) and another value for input above the threshold. Usually the value of the function is less below the threshold than it is above the threshold.

Token A word, word-part, or other sequence of characters. A generalization of the concept of a word to include other word-like entities. Tokens are associated with vectors in a word embedding.

Token A word, word-part, or other sequence of characters. A generalization of the concept of a word to include other word-like entities. Tokens are associated with vectors in a word embedding.

Token embedding (Near synonym: Word embedding). A way of associating each token or word in a document with a numerical vector. Useful in neural networks as a way to convert written text to network inputs. Modern LLMs utilize these token-level embeddings to deal with out-of-vocabulary terms, by breaking words down into sub-tokens. Many types of word embedding algorithm exist.

Topology The way the nodes and weights of a network are wired together. A network's "architecture."

Tonotopic map A topographic map in the auditory cortex that is organized by frequency of sounds. Similar sounds (in terms of frequency) are processed by neurons that are near one another.

Transformer architecture A complex feed-forward network structure which includes a "self-attention" mechanism that allows hidden layers to be "aware" of multiple kinds of relationships between different parts of a sequence of input activations. In the context of LLMs these networks can develop representations of words even when they are far apart in a document.

Testing subset A subset of a dataset used for testing a neural network model (or other machine learning model). This data has not been used in training (it has been "held out") and thus the testing data can be used to see how well a model generalizes from what it was trained on to new data.

Training subset A subset of a dataset used for training a neural network model (or other machine learning model). Contrasted with testing subset.

Truth table A table whose columns are the inputs and outputs of a Boolean functions.

Turing Test The Turing Test is a behavioral test that can be used to determine whether a system is intelligent or not. The standard form of the test is as follows: If a human judge communicating with a system via a text interface cannot determine whether the system is human or not, the system passes the test and can be deemed intelligent. The test derives from a similar test due to Alan Turing (the "imitation game") but is not identical with the imitation game. There is controversy surrounding whether the test is an adequate test of intelligence.

Unsupervised learning A learning rule in which weights are adjusted without an explicit representation of desired outputs.

Vector Ordered list of numbers (n-tuple of numbers). The numbers in a vector are its components. In many cases a vector represents a point. For example: $(2, 2)$ is a vector with two components, which represents a point in a plane.

Vector addition (Synonym Vector sum): two vectors with the same number of components can be added (or summed) by adding their corresponding components. Example: $(1, 2) + (3, 4) = (4, 6)$.

Vector subtraction Two vectors with the same number of components can be subtracted by subtracting their corresponding components.Example: $(1, 2) - (3, 4) = (-2, -2)$.

Vector space A collection of vectors, all of which have the same number of components. For example, the plane is a collection of vectors, all of which have two components. (Note that this is an informal definition; to be a vector space, a set of vectors must meet further requirements as well).

Vector-valued function A function that takes vectors as inputs and produces vectors as outputs. (A more precise designation would be "vector valued function of vector valued inputs").

Ventral stream Pathway that extends from the occipital lobe into the temporal lobes, underlying processing of visual object recognition.

Vesicles The parts at the end of axon terminals where the neurotransmitters are stored for release. Upon triggering caused by action potentials, these vesicles will open and release the neurotransmitters into the synaptic cleft.

Visual cortex Rear-most region in the occipital lobe involved in visual processing, where primary and secondary visual cortex are housed.

Weight (Synonyms Connection, artificial synapse): a simulated synapse or synapse-like element in a neural network.

Weighted Input (Synonym Net input): Dot product of an input vector and a fan-in weight vector, plus a bias term. Notated n_i for neuron i.

Weight layer A set of weights treated as a group. Often they are the collection of weights connecting one node layer to another, which can in turn be represented by a weight matrix.

Weight space The set of possible weight vectors for a given neural network.

Weight vector A vector describing the strengths of the weights in a neural network.

Wernicke's area A region of the temporal lobe associated with written and spoken language comprehension. Damage to this area can lead to Wernicke's aphasia, affecting the ability to understand language, even if language production remains intact.

Winner-Take-All A pool of nodes structured (often with mutually inhibitory connections) so that the node receiving the most inputs weighted inputs "wins" and becomes active while the other nodes become inactive. Also a type of non-local activation function where the node in the group receiving the most net input is active at a winning value and the rest take on a losing value (usually zero).

Zero vector A vector whose components are all 0. Adding the zero vector to any vector gives the same vector.

Appendix A

Logic Gates in Neural Networks

Jeff Yoshimi

We opened the book by noting that neural networks are very different from classical computers. They are trained, not programmed, they transform data using weight matrices rather than explicit rules, they gracefully degrade, and they are can easily handle noisy inputs. We made the contrast by comparing neural networks with Turing Machines, a simple type of machine that captures the kinds of things classical computers do. One thing classical computers do is simple logical computations. In fact, all the incredible things computers do can be boiled down to a bunch of simple logical computations. **Logic gates** are fundamental components of the circuits that make up digital computers. If a neural network could do the same thing, it would show that neural networks could, at least in principle, do anything a digital computer could do.

This problem occupied McCulloch and Pitts, the early pioneers of neural networks: how to make neural networks do logical computations, and implement logic gates. It turns out, they can do this (and thus, neural networks are as powerful as digital computers!) [79]. Understanding these logic functions and how neural networks can compute them is the goal of this section.

We will consider 4 kinds of logic gate: NOT, AND, OR, and XOR. We can think of these in logical terms, as **boolean functions**. These functions take truth values as inputs, and produce other truth values as outputs. Here is roughly how these four boolean functions work:

- NOT P is True if P is false.
- P AND Q is True if both P and Q are True.
- P OR Q is True if P is True, Q is True, or both P and Q are True.
- P XOR Q is true if one of P or Q is True, but not both.

To implement these on a neural network (and see in more detail how they work), we can map True to the number 1 and False to the number 0, and then think of these as vector-valued functions that take vectors of binary values as inputs and produce a single 0 or 1 as an output. It is standard in logic to represent boolean functions with **truth tables**, which are basically the tables we have been using to represent vector-valued functions in neural networks. Here is a truth table that represents the logic of AND:

P	Q	P AND Q
1 (T)	1 (T)	1 (T)
1 (T)	0 (F)	0 (F)
0 (F)	1(T)	0 (F)
0 (F)	0 (F)	0 (F)

In this table, P and Q can be interpreted as the truth values of **proposition**s. We can replace P and Q with any sentence. A proposition (also known as a statement or sentence) is an expression that can

be true or false. For example, P could stand for "I ate pizza" and Q could stand for "I drank soda." In response to a proposition you can say "I agree, that's true" or "I disagree, that's false". Questions, commands, exclamations, and fragments of sentences (like words by themselves) are not propositions. These are propositions: "2+2 = 4", "The moon is made of Swiss cheese", "The mind is distinct from the brain". These are not propositions: "What time is it?", "Swiss cheese","Ouch!", "Pass the salt." Notice that it's odd to say "I agree" or "I disagree" to the non-propositions, but that it is fine to say that to the propositions.

The four rows of the table then show all possible combinations of truth values for two propositions. It could be that I didn't eat pizza or drink soda (row 4), that I drank soda and ate pizza (row 1), that I only ate pizza (row 2), etc. The third column shows us the output of the boolean function for each of these combinations of truth values. For example, focusing on row 1: if I did eat pizza (P) and drink soda (Q), then it's also true that I ate pizza and drank soda (P AND Q).

NOT is also a boolean function, from one input to one output, that basically reverses the truth value of the input when it produces its output:

P	NOT P
1 (T)	0 (F)
0 (F)	1 (T)

Here is a combined truth table showing several other two-valued boolean functions as extra columns, including OR and XOR. XOR is "exclusive" or, which is only true if exactly one of P and Q is true. OR is "inclusive" or, which is true when one or both of P and Q is true.

P	Q	P AND Q	P OR Q	P XOR Q
1 (T)	1 (T)	1 (T)	1 (T)	0 (F)
1 (T)	0 (F)	0 (F)	1 (T)	1 (T)
0 (F)	1 (T)	0 (F)	1 (T)	1 (T)
0 (F)	0 (F)	0 (F)	0 (F)	0 (F)

We can implement any of these functions in a neural network. The input layers will have two nodes, and the output layer will have one node. For example, a network to compute AND would only produce an output of 1 if both input nodes were set to 1. In all other cases of binary inputs the output would be 0. Using binary output functions and appropriate weights and thresholds, it is not so hard to make AND and OR gates.

NOT would just have two nodes. A 0 would produce a 1, and a 1 would produce a 0. This can be done using a binary output node with a bias (so that it is on, by default, above threshold), and a negative weight.

XOR is harder because it requires a few layers of nodes (a point that is important when we talk about supervised learning), but basically it involves combining an OR gate to the output and an intermediate AND gate that inhibits the output. Examples of these are shown in Fig. A.1.

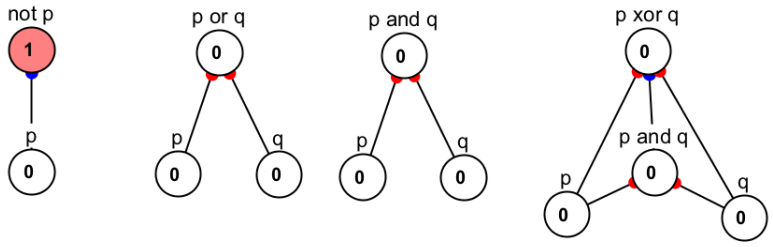


Figure A.1: Networks to implement basic logic gates.

Boolean functions can also be combined, to produce things like (P AND (Q OR NOT R)). To see what the truth table looks like for this type of network, you can try an online resource like `http://web.stanford.edu/class/cs103/tools/truth-table-tool/`. Neural networks to implement this type of function are also

not too difficult to make by combining together the other networks, e.g the output of an OR network is one of the two inputs to the AND network, to make (P AND (Q OR NOT R)).

A complex example, in which sample sentences have been used instead of the boring propositional variables P, Q, R etc., is shown in Fig. A.2.

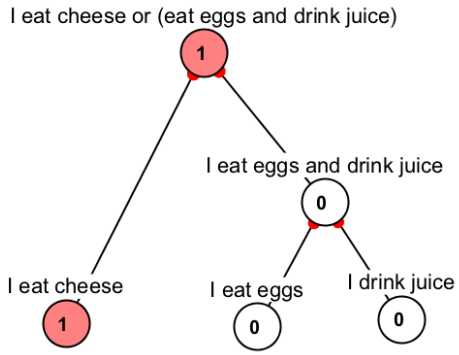


Figure A.2: Network to implement a more complex boolean function.

Figure Attributions

Bibliography

[1] Mostafa Abdou, Artur Kulmizev, Daniel Hershcovich, Stella Frank, Ellie Pavlick, and Anders Søgaard. Can language models encode perceptual structure without grounding? a case study in color. In *Proceedings of the 25th Conference on Computational Natural Language Learning*, pages 109–132, 2021.

[2] Eneko Agirre, Enrique Alfonseca, Keith Hall, Jana Kravalova, Marius Pasca, and Aitor Soroa. A study on similarity and relatedness using distributional and wordnet-based approaches. 2009.

[3] James A Anderson and Edward Rosenfeld. *Talking nets: An oral history of neural networks*. MiT Press, 2000.

[4] Bernard J Baars. *The cognitive revolution in psychology*. The Guilford Press, 1986.

[5] Alexander Bain. *Mind and Body the Theories of Their Relation by Alexander Bain*. Henry S. King & Company, 1873.

[6] Emily M Bender, Timnit Gebru, Angelina McMillan-Major, and Shmargaret Shmitchell. On the dangers of stochastic parrots: Can language models be too big? In *Proceedings of the 2021 ACM conference on fairness, accountability, and transparency*, pages 610–623, 2021.

[7] Emily M Bender and Alexander Koller. Climbing towards nlu: On meaning, form, and understanding in the age of data. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pages 5185–5198, 2020.

[8] Nils Bertschinger and Thomas Natschläger. Real-time computation at the edge of chaos in recurrent neural networks. *Neural computation*, 16(7):1413–1436, 2004.

[9] Robert C Berwick, Paul Pietroski, Beracah Yankama, and Noam Chomsky. Poverty of the stimulus revisited. *Cognitive science*, 35(7):1207–1242, 2011.

[10] Guo-qiang Bi and Mu-ming Poo. Synaptic modifications in cultured hippocampal neurons: dependence on spike timing, synaptic strength, and postsynaptic cell type. *Journal of neuroscience*, 18(24):10464–10472, 1998.

[11] Christopher M Bishop. *Neural networks for pattern recognition*. Oxford university press, 1995.

[12] Amy L Boggan and Chih-Mao Huang. Chess expertise and the fusiform face area: Why it matters. *Journal of Neuroscience*, 31(47):16895–16896, 2011.

[13] Edwin Garrigues Boring. *History of experimental psychology*. Genesis Publishing Pvt Ltd, 1929.

[14] Jeffrey S Bowers, Gaurav Malhotra, Marin Dujmović, Milton Llera Montero, Christian Tsvetkov, Valerio Biscione, Guillermo Puebla, Federico G Adolfi, John Hummel, Rachel Flood Heaton, et al. Deep problems with neural network models of human vision. *Preprint*, 2022.

[15] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[16] AE Bryson and YC Ho. Applied optimal control. *Optimization, Estimation and Control*, 1969.

[17] Emanuele Bugliarello, Laurent Sartran, Aishwarya Agrawal, Lisa Anne Hendricks, and Aida Nematzadeh. Measuring progress in fine-grained vision-and-language understanding. *arXiv preprint arXiv:2305.07558*, 2023.

[18] Ed Bullmore and Olaf Sporns. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience*, 10(3):186–198, 2009.

[19] Malcolm Burrows. *The neurobiology of an insect brain*. Oxford University Press Oxford, 1996.

[20] Joana Cabral, Etienne Hugues, Olaf Sporns, and Gustavo Deco. Role of local network oscillations in resting-state functional connectivity. *Neuroimage*, 57(1):130–139, 2011.

[21] Jonathan Gabel Christiansen, Mathias Gammelgaard, and Anders Søgaard. Large language models partially converge toward human-like concept organization. In *NeurIPS 2023 Workshop on Symmetry and Geometry in Neural Representations*.

[22] Andy Clark. Whatever next? predictive brains, situated agents, and the future of cognitive science. *Behavioral and brain sciences*, 36(3):181–204, 2013.

[23] Hubert L Dreyfus. *What computers still can't do: a critique of artificial reason*. MIT press, 1992.

[24] Chris Eliasmith, Terrence C Stewart, Xuan Choo, Trevor Bekolay, Travis DeWolf, Yichuan Tang, and Daniel Rasmussen. A large-scale model of the functioning brain. *science*, 338(6111):1202–1205, 2012.

[25] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[26] Jeffrey L Elman. *Rethinking innateness: A connectionist perspective on development*, volume 10. MIT press, 1996.

[27] Katrin Erk. Vector space models of word meaning and phrase meaning: A survey. *Language and Linguistics Compass*, 6(10):635–653, 2012.

[28] Laurene V Fausett. *Fundamentals of neural networks*. Prentice-Hall, 1994.

[29] Chrisantha Fernando and Sampsa Sojakka. Pattern recognition in a bucket. In *European Conference on Artificial Life*, pages 588–597. Springer, 2003.

[30] Stanley Finger. *Origins of neuroscience: a history of explorations into brain function*. Oxford University Press, USA, 2001.

[31] Lev Finkelstein, Evgeniy Gabrilovich, Yossi Matias, Ehud Rivlin, Zach Solan, Gadi Wolfman, and Eytan Ruppin. Placing search in context: The concept revisited. In *Proceedings of the 10th international conference on World Wide Web*, pages 406–414, 2001.

[32] John Firth. A synopsis of linguistic theory, 1930-1955. *Studies in linguistic analysis*, pages 10–32, 1957.

[33] Jerry A Fodor and Zenon W Pylyshyn. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28(1):3–71, 1988.

[34] Sigmund Freud, Marie Ed Bonaparte, Anna Ed Freud, Ernst Ed Kris, Eric Trans Mosbacher, and James Trans Strachey. *Project for a scientific psychology*. Basic Books, 1954.

[35] Kunihiko Fukushima and Sei Miyake. Neocognitron: A new algorithm for pattern recognition tolerant of deformations and shifts in position. *Pattern recognition*, 15(6):455–469, 1982.

[36] Ken-ichi Funahashi and Yuichi Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural networks*, 6(6):801–806, 1993.

[37] Michael S Gazzaniga, RB Ivry, and GR Mangun. *Cognitive Neuroscience, New York: W. W.* Norton & Company, 2002.

[38] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[39] Harold Goodglass and Norman Geschwind. Language disorders (aphasia). *Handbook of perception*, 7:389–428, 1976.

[40] Noah D Goodman and Michael C Frank. Pragmatic language interpretation as probabilistic inference. *Trends in cognitive sciences*, 20(11):818–829, 2016.

[41] H Paul Grice. Meaning. *The philosophical review*, 66(3):377–388, 1957.

[42] Herbert P Grice. Logic and conversation. In *Speech acts*, pages 41–58. Brill, 1975.

[43] David Groome. *An introduction to cognitive psychology: Processes and disorders*. Psychology Press, 2013.

[44] Otto-Joachim Grüsser. 'on the 'seat of the soul', cerebral localization theories in medieval times and later. In *Brain–perception, cognition: proceedings of the 18th Göttingen Neurobiology Conference*. Georg Thieme Verlag, 1990.

[45] Umut Güçlü and Marcel AJ van Gerven. Deep neural networks reveal a gradient in the complexity of neural representations across the ventral stream. *Journal of Neuroscience*, 35(27):10005–10014, 2015.

[46] Patric Hagmann, Leila Cammoun, Xavier Gigandet, Reto Meuli, Christopher J Honey, Van J Wedeen, and Olaf Sporns. Mapping the structural core of human cerebral cortex. *PLoS Biol*, 6(7):e159, 2008.

[47] Zellig S Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

[48] David Hartley. *Observations on Man: His Frame, His Duty and His Expectations...* Leake & Frederick, 1749.

[49] David Hartley. *Observations on man, his frame, his duty, and his expectations*. T. Tegg and son, 1834.

[50] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 2 edition, 1998.

[51] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[52] Jukka Heikkonen and Jouko Lampinen. Building industrial applications with neural networks. In *Proceedings of the European symposium on intelligent techniques*, pages 3–4, 1999.

[53] John A Hertz, Anders S Krogh, and Richard G Palmer. *Introduction to the theory of neural computation*, volume 1. Basic Books, 1991.

[54] Felix Hill, Roi Reichart, and Anna Korhonen. Simlex-999: Evaluating semantic models with (genuine) similarity estimation. *Computational Linguistics*, 41(4):665–695, 2015.

[55] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[56] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[57] Philipp K Janert. *Data Analysis with Open Source Tools: A Hands-on Guide for Programmers and Data Scientists*. " O'Reilly Media, Inc.", 2010.

[58] Eric R Kandel, James H Schwartz, Thomas M Jessell, Steven A Siegelbaum, and AJ Hudspeth. *Principles of neural science*, volume 4. McGraw-hill New York, 2000.

[59] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. *Andrej Karpathy blog*, 2015.

[60] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

[61] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[62] Nikolaus Kriegeskorte. Deep neural networks: a new framework for modeling biological vision and brain information processing. *Annual review of vision science*, 1:417–446, 2015.

[63] Trenton Kriete, David C Noelle, Jonathan D Cohen, and Randall C O'Reilly. Indirection and symbol-like processing in the prefrontal cortex and basal ganglia. *Proceedings of the National Academy of Sciences*, 110(41):16390–16395, 2013.

[64] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

[65] Andrey Kurenkov. A brief history of neural nets and deep learning. *Skynet Today*, 2020.

[66] Andreea Lazar, Gordon Pipa, and Jochen Triesch. Sorn: a self-organizing recurrent neural network. *Frontiers in computational neuroscience*, 3:23, 2009.

[67] Yann Le Cun. Learning process in an asymmetric threshold network. In *Disordered systems and biological organization*, pages 233–240. Springer, 1986.

[68] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[69] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[70] Alessandro Lenci. Distributional models of word meaning. *Annual review of Linguistics*, 4:151–171, 2018.

[71] Daniel S Levine. *Introduction to neural and cognitive modeling*. Psychology Press, 2000.

[72] Molly Lewis, Martin Zettersten, and Gary Lupyan. Distributional semantics as a source of visual knowledge. *Proceedings of the National Academy of Sciences*, 116(39):19237–19238, 2019.

[73] Bertrand Liétard, Mostafa Abdou, and Anders Søgaard. Do language models know the way to rome? In *Proceedings of the Fourth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, pages 510–517, 2021.

[74] Wolfgang Maass. Computing with spikes. *Special Issue on Foundations of Information Processing of TELEMATIK*, 8(1):32–36, 2002.

[75] David Marr and W Thomas Thach. A theory of cerebellar cortex. In *From the Retina to the Neocortex*, pages 11–50. Springer, 1991.

[76] James L McClelland. Retrieving general and specific information from stored knowledge of specifics. In *Proceedings of the third annual meeting of the cognitive science society*. Citeseer, 1981.

[77] James L McClelland, Felix Hill, Maja Rudolph, Jason Baldridge, and Hinrich Schütze. Placing language in an integrated understanding system: Next steps toward human-level performance in neural language models. *Proceedings of the National Academy of Sciences*, 117(42):25966–25974, 2020.

[78] James L McClelland and David E Rumelhart. *Explorations in parallel distributed processing: A handbook of models, programs, and exercises*. MIT press, 1989.

[79] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[80] Edmund S Meltzer and Gonzalo M Sanchez. *The Edwin Smith Papyrus: updated translation of the trauma treatise and modern medical commentaries*. ISD LLC, 2014.

[81] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[82] Daniel Miner and Jochen Triesch. Plasticity-driven self-organization under topological constraints accounts for non-random features of cortical synaptic wiring. *PLoS Comput Biol*, 12(2):e1004759, 2016.

[83] Marvin Minsky and Seymour Papert. *Perceptrons.* MIT press, 1969.

[84] Melanie Mitchell. Why ai is harder than we think. *arXiv preprint arXiv:2104.12871*, 2021.

[85] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[86] Wael MY Mohamed. Arab and muslim contributions to modern neuroscience. *IBRO History of Neuroscience*, 169(3):255, 2008.

[87] Annalee Newitz. Movie written by algorithm turns out to be hilarious and intense. *Ars Technica*, 2016.

[88] Yael Niv. Reinforcement learning in the brain. *Journal of Mathematical Psychology*, 53(3):139–154, 2009.

[89] David C Noelle and Jeffrey Yoshimi. Artificial intelligence and computational theories of mind. In *Mind, Cognition, and Neuroscience*, pages 127–148. Routledge, 2022.

[90] Christopher Olah. Understanding lstm networks. *GITHUB blog, posted on August*, 27:2015, 2015.

[91] Randall C. O'Reilly, Yuko Munakata, Michael J. Frank, Thomas E. Hazy, and Contributors. *Computational Cognitive Neuroscience*. Online Book, 4th Edition, URL: `https://CompCogNeuro.org`, 2012.

[92] Randall C O'Reilly, Thomas E Hazy, and Seth A Herd. The leabra cognitive architecture: How to play 20 principles with nature. *The Oxford Handbook of Cognitive Science*, page 91, 2016.

[93] Michael PA Page, Robert J Howard, John T O'Brien, Muriel S Buxton-Thomas, and Alan D Pickering. Use of neural networks in brain spect to diagnose alzheimer's disease. *The Journal of Nuclear Medicine*, 37(2):195, 1996.

[94] DB Parker. Learning-logic (tr-47). *Center for Computational Research in Economics and Management Science. MIT-Press, Cambridge, Mass*, 8, 1985.

[95] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014.

[96] Steven T Piantadosi. Zipf's word frequency law in natural language: A critical review and future directions. *Psychonomic bulletin & review*, 21:1112–1130, 2014.

[97] Gualtiero Piccinini. The first computational theory of mind and brain: a close look at mcculloch and pitts's "logical calculus of ideas immanent in nervous activity". *Synthese*, 141(2):175–215, 2004.

[98] Steven Pinker and Michael T Ullman. The past and future of the past tense. *Trends in cognitive sciences*, 6(11):456–463, 2002.

[99] Marcel Proust. *Remembrance of things past*, volume 2. Wordsworth Editions, 2006.

[100] Rajesh PN Rao and Dana H Ballard. Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects. *Nature neuroscience*, 2(1):79–87, 1999.

[101] Russell Richie, Sachin Grover, and Fuchiang Rich Tsui. Inter-annotator agreement is not the ceiling of machine learning performance: Evidence from a comprehensive set of simulations. In *Proceedings of the 21st Workshop on Biomedical Language Processing*, pages 275–284, 2022.

[102] Samuel Ritter, David GT Barrett, Adam Santoro, and Matt M Botvinick. Cognitive psychology for deep neural networks: A shape bias case study. In *International conference on machine learning*, pages 2940–2949. PMLR, 2017.

[103] Frank Rosenblatt. Perceptron simulation experiments. *Proceedings of the IRE*, 48(3):301–309, 1960.

[104] Frank Rosenblatt. A comparison of several perceptron models. *Self-Organizing Systems*, pages 463–484, 1962.

[105] David E Rumelhart and James L McClelland. On learning the past tenses of english verbs. *Psycholinguistics: Critical Concepts in Psychology*, 4:216–271, 1986.

[106] David E Rumelhart, James L McClelland, PDP Research Group, et al. Parallel distributed processing, vol. 1, 1986.

[107] Jürgen Schmidhuber, Sepp Hochreiter, et al. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[108] Martin Schrimpf, Idan Asher Blank, Greta Tuckute, Carina Kauf, Eghbal A Hosseini, Nancy Kanwisher, Joshua B Tenenbaum, and Evelina Fedorenko. The neural architecture of language: Integrative modeling converges on predictive processing. *Proceedings of the National Academy of Sciences*, 118(45):e2105646118, 2021.

[109] Wolfram Schultz, Peter Dayan, and P Read Montague. A neural substrate of prediction and reward. *Science*, 275(5306):1593–1599, 1997.

[110] Mark S Seidenberg and James L McClelland. A distributed, developmental model of word recognition and naming. *Psychological review*, 96(4):523, 1989.

[111] Terrence J Sejnowski and Charles R Rosenberg. Parallel networks that learn to pronounce english text. *Complex systems*, 1(1):145–168, 1987.

[112] O. G. Selfridge. Pandemonium: a paradigm for learning in Mechanisation of Thought Processes. In *Proceedings of a Symposium Held at the National Physical Laboratory*, pages 513–526, London, November 1958.

[113] Helaine Selin. *Encyclopaedia of the history of science, technology, and medicine in non-westen cultures*. Springer Science & Business Media, 2013.

[114] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[115] Paul Smolensky. On the proper treatment of connectionism. *Behavioral and brain sciences*, 11(1):1–23, 1988.

[116] Anders Søgaard. Grounding the vector space of an octopus: Word meaning from raw text. *Minds and Machines*, 33(1):33–54, 2023.

[117] Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022.

[118] John Sutton. *Philosophy and memory traces: Descartes to connectionism*. Cambridge University Press, 1998.

[119] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[120] John Von Neumann. The principles of large-scale computing machines. *Annals of the History of Computing*, 3(3):263–273, 1981.

[121] Maya Zhe Wang and Benjamin Y Hayden. Latent learning, cognitive maps, and curiosity. *Current Opinion in Behavioral Sciences*, 38:1–7, 2021.

[122] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*, 1974.

[123] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[124] James CR Whittington and Rafal Bogacz. Theories of error back-propagation in the brain. *Trends in cognitive sciences*, 23(3):235–250, 2019.

[125] Bernard Widrow. Adaline: Smarter than sweet, 1963.

[126] Bernard Widrow and Michael A Lehr. 30 years of adaptive neural networks: perceptron, madaline, and backpropagation. *Proceedings of the IEEE*, 78(9):1415–1442, 1990.

[127] Norbert Wiener. *Cybernetics: Control and communication in the animal and the machine*. Wiley New York, 1948.

[128] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

[129] Ludwig Wittgenstein. Philosophical investigation.(tr. gem anscombe) oxford. *UK: Blackwell*, 1953.

[130] Daniel LK Yamins and James J DiCarlo. Using goal-driven deep learning models to understand sensory cortex. *Nature neuroscience*, 19(3):356–365, 2016.

[131] Daniel LK Yamins, Ha Hong, Charles F Cadieu, Ethan A Solomon, Darren Seibert, and James J DiCarlo. Performance-optimized hierarchical models predict neural responses in higher visual cortex. *Proceedings of the national academy of sciences*, 111(23):8619–8624, 2014.

[132] George Kingsley Zipf. The meaning-frequency relationship of words. *The Journal of general psychology*, 33(2):251–256, 1945.

[133] David Zipser. Identification models of the nervous system. *Neuroscience*, 47(4):853–862, 1992.

[134] Marco Zorzi, Alberto Testolin, and Ivilin P Stoianov. Modeling language and cognition with deep unsupervised learning: a tutorial overview. *Frontiers in psychology*, 4:515, 2013.