Neural Networks in Cognitive Science

Jeff Yoshimi, Zoë Tosi, Scott Hotton, Chelsea Gordon, David C. Noelle

Version 2023.1.1

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 McLelland'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). 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, Zoë Tosi

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.

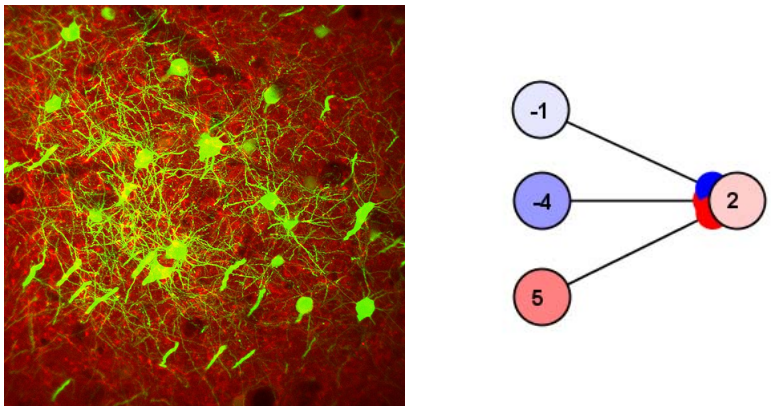


Figure 1.1: Left: A biological neural network. Right: A simple artificial neural network built in Simbrain, with three nodes connecting to another node via three weights.

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 13 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 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 way 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.[1]

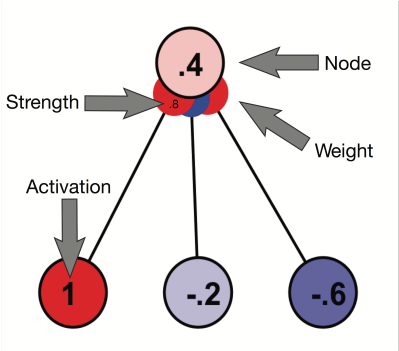


Figure 1.2: Nodes and their activations; weights and their strengths.

In the figure, a circle with a number in it is an artificial neuron or **node** (nodes are also referred to as "units").[2] 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. 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.[3] In neuroscience terms, these correspond to excitatory and inhibitory synapses. 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. 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.

Node activations change in accordance with *activation rules* (discussed in ch. 4), and weight strengths change in accordance with *learning rules* (discussed in several chapters, including chapters 7 and 11).

In some models a node can also produce a **spike**, which is a discrete event that corresponds to the action potential of a neuron.[4] Spiking neurons have their own rules and structures, which we discuss in ch. 14.

[1] Neural networks can be used for engineering, to model the brain, or to model the mind. These different uses are discussed in section 1.3. Depending on the way a network is being used, the way its parts are interpreted differs, as we'll see.

[2] 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).

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

[4] 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/docs/Pages/Network/neuron.html`.

Nodes and weights are the basic parts of a neural network. Together, they form a network structure or mathematically, a graph[5] w the nodes correspond to vertices, and the weights correspond to edges.[6] The graph-structure formed by a network's nodes and weights is the network's **topology**. 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).[7] Activity in this kind of network flows from an *input layer* through a (possibly empty) set of *hidden layers*, and then to an *output layer*.[8] In a feed-forward network activity simply passes through; when the input nodes are activated and the network is updated, the 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 **recurrent network**, the nodes are interconnected in such a way that activity can flow in repeating cycles.[9] 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.

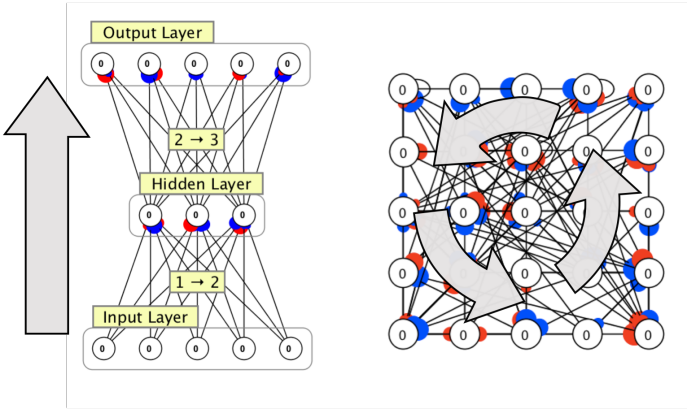


Figure 1.3: Feed-forward network (left) and recurrent network (right). Gray arrows give a sense of how activation flows through them.

There are other kinds of architecture beyond the ones just mentioned. An especially popular architecture since the 2010s has been the **deep network** architecture, which involves many layers of neurons and weights (some of them arranged into stacks or "tensors" at a given level), often using a special type of layer called a "convolutional" layer, which scans over its input to produce outputs. An example of a deep network is shown

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

[6] 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*).

[7] 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 *multipartite* because the vertices can be partitioned into *independent sets* (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.

[8] Sometimes a distinction is made between *node layers* and *weight layers*. 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"). To make matters even more confusing sometimes people don't count the input layer as a node layer. In Simbrain, node and weight layers are both represented as groups, indicated by the yellow interaction boxes: `http://www.simbrain.net/Documentation/docs/Pages/Network/groups.html`. When used without qualification, we use the term "layer" to mean "node layer".

[9] In graph-theoretic terms, this is a cyclic graph, which contains at least one cycle. Recall from footnote 7 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.

in figure 1.4. Deep networks are discussed in more detail in section 12. Note that networks like this are so large, that we can't represent each node and weight separately. Instead node layers containing thousands of nodes (or more) and weight layers containing tens of thousands of weights (or more) are shown as sheets or lines. This is something we have to get used to: there are different ways of drawing neural network diagrams.

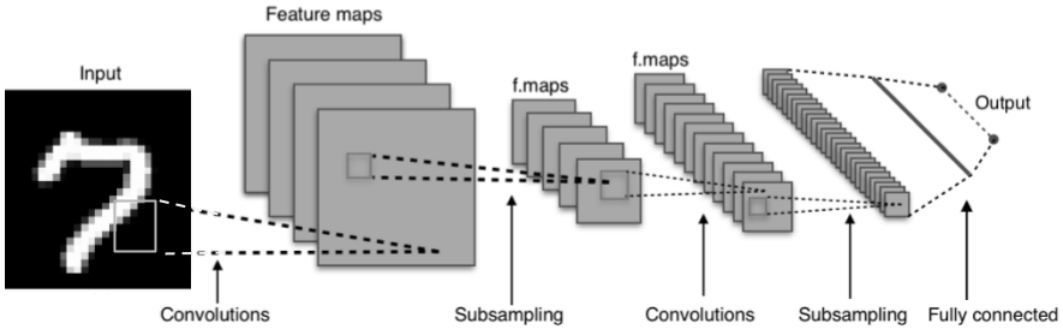


Figure 1.4: A deep neural network, trained to recognize images. The convolutional layers scan over the inputs they are linked to.

Networks almost always exist in some kind of **environment**, which gathers inputs for a network and receives its outputs. 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. However, 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 6. 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.[10]

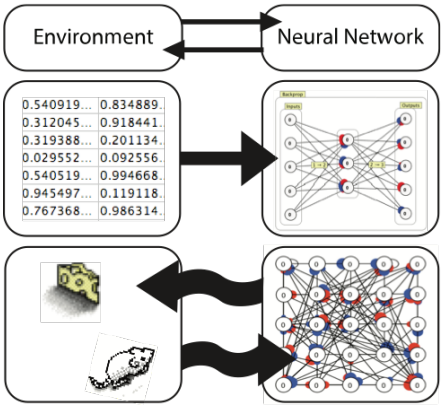


Figure 1.5: The relationship between a neural network and an environment. An "environment" is often something as simple as a table of values (middle row). However it can also be something more complex, like a virtual world (bottom row).

[10]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/docs/Pages/Workspace/Couplings.html`.

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: how they channel information, and how they can actually perform interesting tasks.

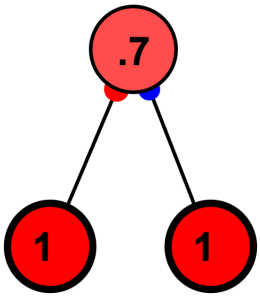


Figure 1.6: Simple feed-forward network with clamped inputs. By adjusting the weight strengths, we can make the output activation be whatever we want.

To begin with, how do neural networks "channel information"? The mathematical details are not too difficult, but for now we are developing basic intuitions.[11] Consider the network shown in figure 1.6: it's a simple feed-forward network with two node layers (an input and output layer) and one weight layer. The inputs are **clamped node**s, and are thus represented with a bold-faced outline. This means they will not change their value when we update the network. How is the activation of the output node determined? Let us briefly assume that all activations are positive, as they are in the figure.[12] Assuming all activations are positive, we can think of outputs as being computed in the following way:

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

2. Negative weights (blue disks) decrease 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 output. Suppose we want the output to be some other value besides .7, like .9. To do this, we could strengthen the positive weight a little bit so that it makes the output larger. We could also weaken the negative weight so that it decreases the output less. Or both. In a similar way, to reduce the output to .5, we could either weaken the positive weight or strengthen the negative weight. In doing so, we are tuning the weights up and down to get the output we want. Most of the theory of neural networks is about automatically tuning weights and other parameters to get useful flows of activation.

With this background in hand, let's move to an example that shows how computation in neural networks differs from computation in a standard digital computer. In both neural networks and digital computers, information is processed, but the *way* it is processed differs. Consider the simulation *threeObjectsDist.zip*. A screen shot of the network, which we call the "three object detector", is shown in Fig. 1.7.[13] 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. 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

[11] This is an example of a linear activation function. The output is obtained by multiplying each input activation times the strength of the intervening weight. See chapter 4. In this case the weight strengths are .8 and $-.1$, and so the output is $1 \times .8 + 1 \times (-.1) = .7$. Since the input activations are 1, the output in this case is just the sum of the two weights: $.8 - .1 = .7$.

[12] 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.

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

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).

In this example we also see how a network can be linked to a virtual environment. The mouse in the simulated world 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. The job of this network is to distinguish the three objects on the basis of those sensory inputs. Depending on which object the mouse is near, a different output node should be activated. This is called a "classification task".

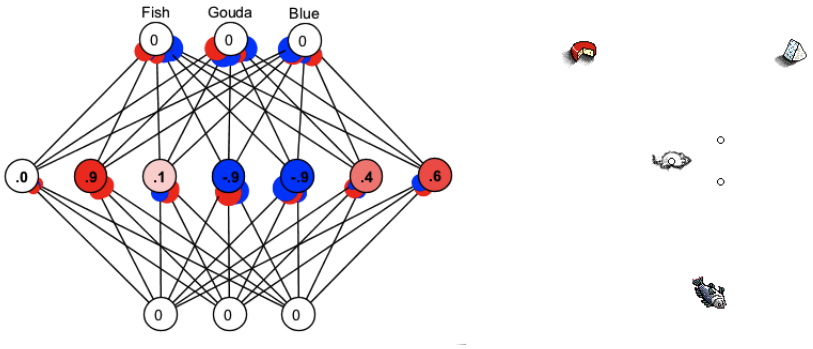


Figure 1.7: Simple feed-forward network that recognizes three objects.

We can use this example to illustrate certain general properties of computation in neural networks, which can be contrasted with classical computation in a digital computer.[14] 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.[15] Neural networks are also more tolerant of noisy signals and damage than digital computers are. And 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 10) 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 of cheese we have available, and on that basis it can recognize new pieces of cheese it's never seen before.

[14] 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.

[15] 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 5.

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**).[16]

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.

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."[17] 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.[18]

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).[19]

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 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. As we discuss linear algebra and dynamical systems how these patterns can be visualized as points in a space (chapters 5 and 8). In this way we can see how distributed representations cluster

[16]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.

[17]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.

[18]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.

[19]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.

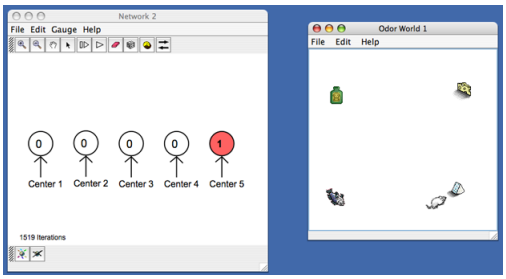


Figure 1.8: Localist representation

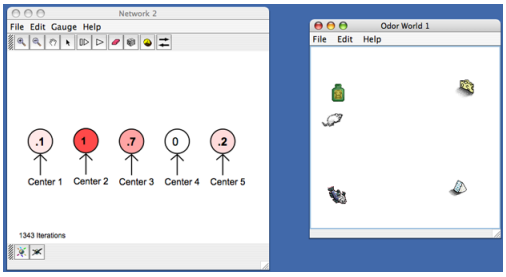


Figure 1.9: Distributed representation

into similar "smells in smell space", for example. We will describe algorithms for identifying these clusters, separating them, and fitting them to surfaces. We will develop ways for visualizing these structures even when the spaces are extremely high dimensional. It's extremely useful and works on localist representations as well.

1.3 Types of Neural Network Research

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

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.

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.

1.3.1 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. Neural networks in this area are used as statistical models (one kind of **machine learning** model). These models 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.

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

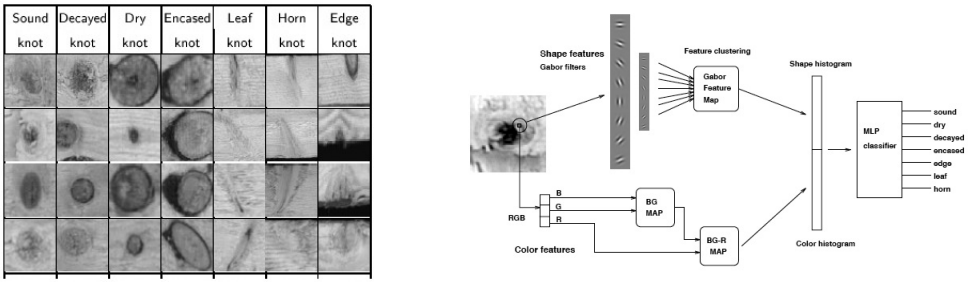


Figure 1.10: Left: Sample inputs to the knot classification network. Right: The knot classification system. The neural network is labelled "MLP."

figure 1.10. 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 [38].

1.3.2 Computational neuroscience

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. 1.11). It is a highly multidisciplinary field, encompassing biology, neuroscience, psychopharmacology, cognitive science, complexity science, psychology, and even physics, depending on the context.

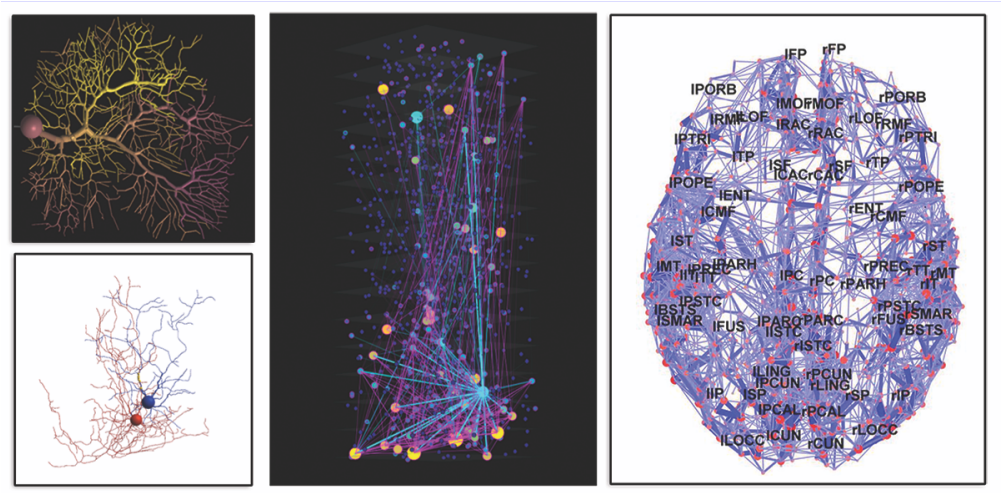


Figure 1.11: Micro, meso, and macro-level models in computational neuroscience. Left: micro-level models of individual neurons. Middle: meso-level model of a network of several thousand neurons. Right: macro-level model of neuronal connections spread out through the entire brain.

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 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.[20] 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 [12]. 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. 1.11).

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 14.[21]

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

1.3.3 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**.[22] In connectionist models, there is

[20] As an example, see `https://www.ncbi.nlm.nih.gov/pubmed/21511044` [14]

[21] 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.

[22] 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".

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 Sharks from *West Side Story* [57].[23]

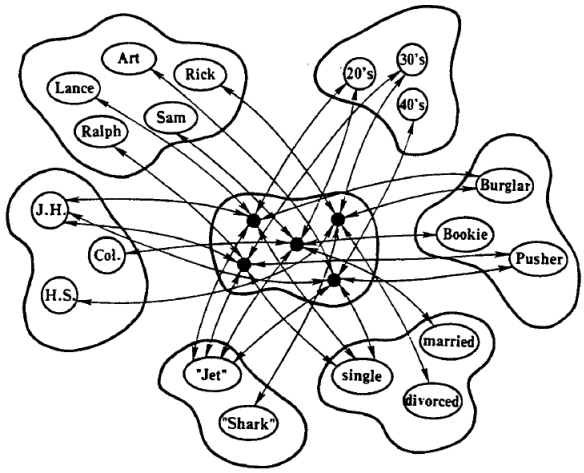


Figure 1.12: A fragment of the Jets and Sharks model. Nodes in this model don't represent neural activity, but activation of concepts in semantic memory.

An IAC model is organized into pools of nodes. In Fig. 1.12, 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 [57].

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

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

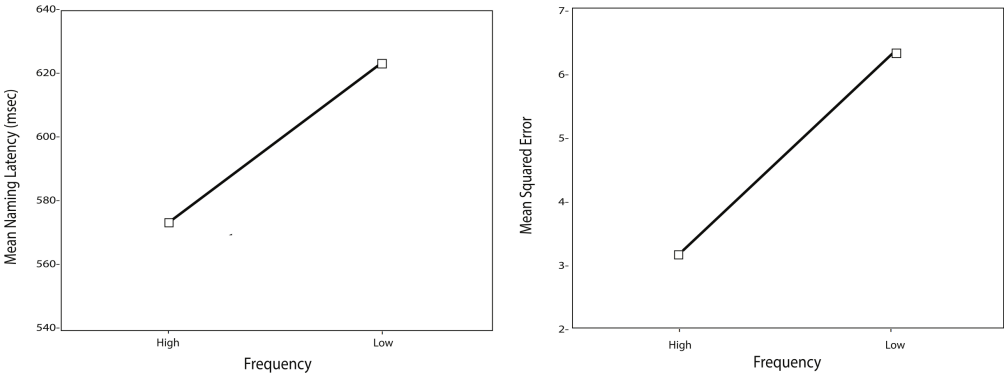


Figure 1.13: Data associated with Seidenberg and McClelland (1989)'s reading model. Human data are on the left, neural network data are on the right. Humans pronounce high frequency words more quickly than low-frequency words. The neural network makes fewer errors on high frequency than low-frequency words.

quantitative. For example, Seidenberg and McLelland 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 10). This simulation models the *word frequency effect*. Words that occur 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. 1.13. 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 [85]. 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.

1.3.4 Mixed and intermediate cases

Both distinctions discussed in this section can be difficult to apply. It can be difficult to know whether a neural network model is being used as an engineering or a scientific model. It can also be hard to say whether a scientific model is being used to simulate the brain (computational neuroscience), or cognition and behavior (connectionism). Some models do both at once.

The first distinction, between engineering and scientific uses of neural networks, can be confusing because the two usages have been historically intertwined. Some neural networks that originated as scientific models later got used as engineering tools. Sometimes the reverse happens: an engineering tool ends up being useful as a scientific model. Deep networks provide a striking example of these back and forths. As we discuss in chapters 2 and 3, deep neural networks were originally used as models of vision in the 1970s. These later turned out to be excellent tools for pattern recognition, when they were used to recognize digits on envelopes, leading to the deep learning revolution of the 2010s. 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 then got adapted back to science!

A good rule of thumb when considering how to classify a neural network is to ask: "What is the neural network being used for? As a tool, or as a scientific model?"[24] 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" [72]. At first, this sounds like it might be about a computational neuroscience or

[24] 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?

connectionist 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, *not* as a model of schizophrenia. So it's really an engineering usage of a neural network, rather than a scientific model.

As far as the distinction within scientific modeling between computational neuroscience and connectionist models, here too there are difficult cases, mainly because 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.

For example, models in **computational cognitive neuroscience** 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.[25] Some examples of this type of model are shown in figure 1.14.

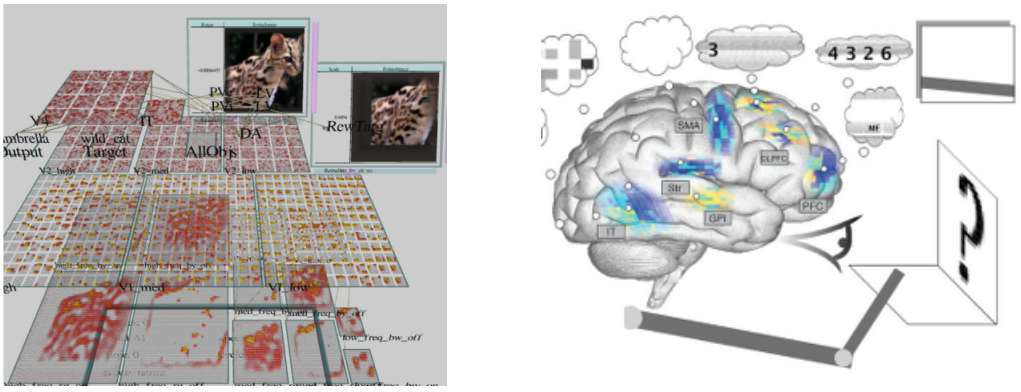


Figure 1.14: (Left) An Emergent simulation of visual processing, with labels indicating which brain areas each group of nodes represents. (Right) A Nengo simulation of the human ability to retrace a visually perceived number.

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.

[25] 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`).

Chapter 2

History of Neural Networks

Jeff Yoshimi

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. 2.1.

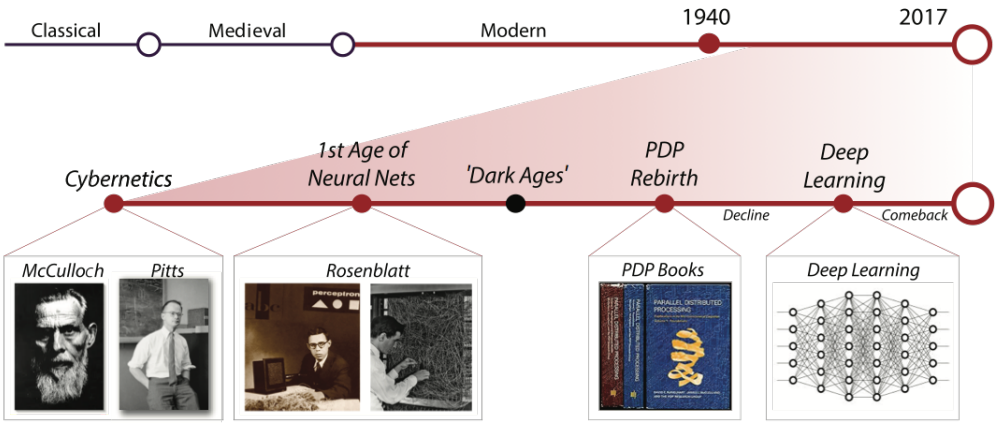


Figure 2.1: A timeline of the history of neural networks. The main history of neural networks runs from the mid 1940s to the present. We also consider some of the pre-history of neural networks, i.e. historical figures who linked the structure and dynamics of the mind with the structure and dynamics of the brain.

2.1 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. 2.2.

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 [61].

[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 [88]), but there is not (as of 2017) much scholarship on the history of neuroscience, cognitive science, or psychology in Africa, Asia, India, Meso-America, the Middle East, and other regions whose historical documents contain

Figure 2.2: 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 [23].

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 [23]. 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 2.3. 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.

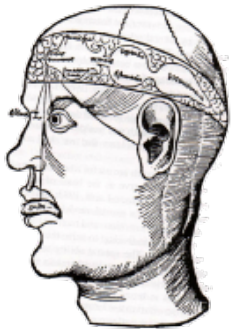


Figure 2.3: A medieval diagram which shows how spirits were thought to flow and combine through the ventricles of the brain. Different ventricles were associated with different faculties, such as sensation and the "common sense," integrating different senses, imagination, memory, and reason.

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) [3].

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

relevant information. There is however, some literature on Arabic and Islamic roots of neuroscience [65].

[3]For more on this period of history, see Sutton (1998) [90]. 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 [35, 3]. For example, Hartley [34] 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 7.

Later, in the 19th century, Bain illustrated these ideas with images that look strikingly like modern neural network diagrams, as in Fig. 2.4.

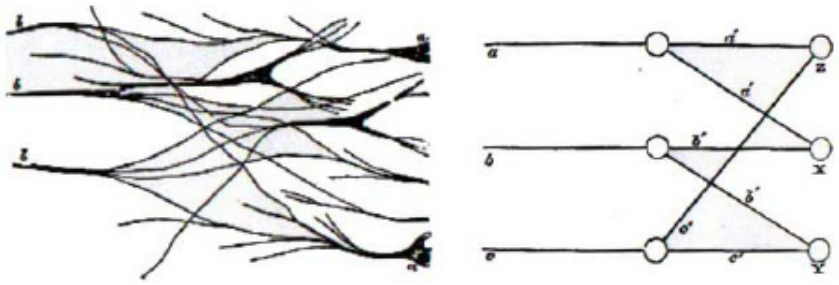


Figure 2.4: From Bain's 1873 book *Mind and Body*, which opens with the question "What has Mind to do with brain substance, white and grey"?

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" [25, 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. 2.5 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., [8]), 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 [82, p. 41].

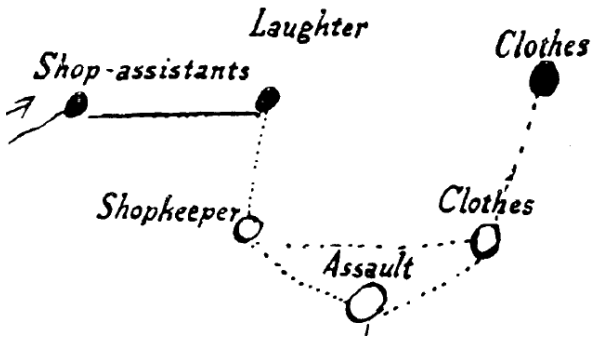


Figure 2.5: An image from Freud's early *Project for a Scientific Psychology*. The open circles are conscious ideas; the dark circles are unconscious ideas.

2.2 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 [1]). 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" [92]). 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" [98, 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 [60]. A diagram from McCulloch and Pitt's famous paper, *A Logical Calculus of Ideas Immanent in Nervous Activity*, is shown in figure 2.6.[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 4): 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: [21, pp. 22-26]. Levine, ch. 2 is especially detailed on McCulloch,s Pitts and Rosenblatt [54]. 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, [82], and Haykin (2nd Ed.) section 1.9 [36]. My favorite source is a series of interviews of leading figures in the history of neural networks collected in *Talking Nets*, [1].

[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 [1]. 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 [54].

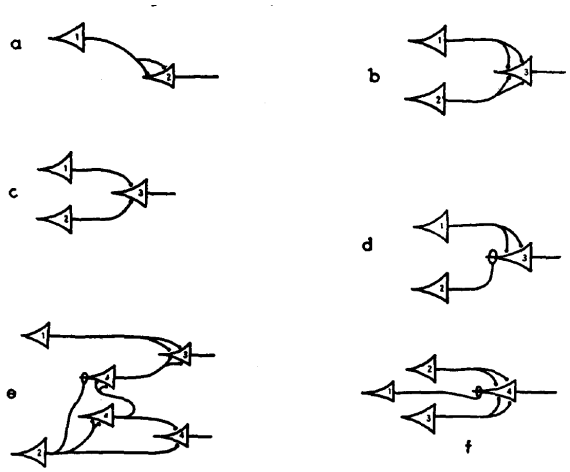


Figure 2.6: From the end of McCulloch and Pitt's famous article, in which they demonstrate that "for any logical expression satisfying certain conditions, one can find a net behaving in the fashion it describes." In these networks, what are today called "weight strengths" or "synaptic efficacy" correspond 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 [74].

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. 3). 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.

2.3 The Cognitive Revolution

The next major event in the standard history of neural networks was the development of the *Perceptron* in the late 1950s, which we discuss in the next section. However, during the 1950s-1970s many other developments took place that broadly supported a neural network approach to the study of cognition. Framing all these events was the emergence of cognitive science, via the "cognitive revolution."[9] 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. These *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, the 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.

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

[9] An excellent overview and history of the era is [2].

that fire together, wire together") [37] (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. 2.7 shows one of Hebb's own diagrams of a cell assembly) [37]. The concept of a specific, learned pattern of brain activity produced by a stimulus remains important today.[11]

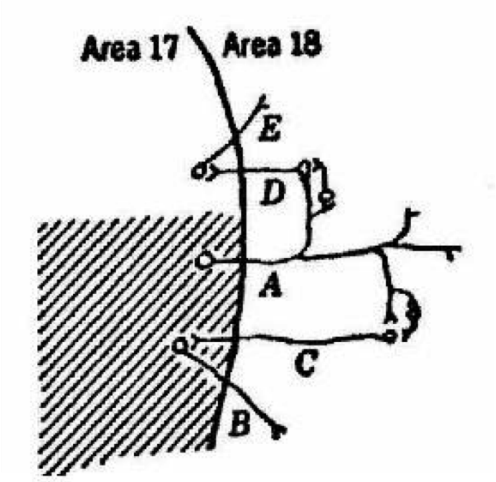


Figure 2.7: A Hebbian cell assembly. These neurons initially fired together, and then got wired together, and so they will tend to fire together in the future.

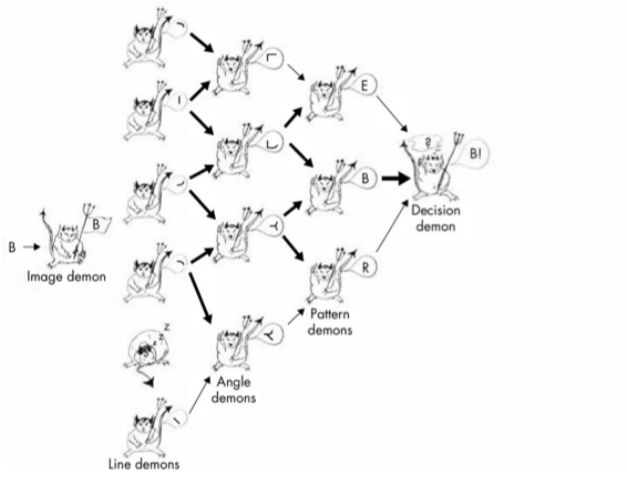


Figure 2.8: Selfridge's pandemonium model.

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 2.8 shows how Selfridge thought of the process of perceiving the letter "B". An image arrives at the eye, line demons detect lines in various orientations, and those demons send messages to angle demons who detect angles, and the process continues through a network of demons until a decision demon says "B!" [87]

[10] See `http://www.scholarpedia.org/article/Donald_Olding_Hebb`. Also see Werbos' interview in [1].

[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`.

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).

2.4 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 [97]).[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 (cf. Chap. 10). Thus both networks moved beyond McCulloch and Pitt's networks to networks that actually learned from experience.[13]

Figure 2.9: Rosenblatt with one of his hardware implementations of a perceptron (left) and another view of the perceptron (right).

Rosenblatt was a psychologist interested in human and animal behavior and its neural basis.[14] 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. 2.9). This was quite impressive at the time and got a considerable amount of press.[15] Rosenblatt also proved that perceptrons could find solutions to certain types of classification tasks in a finite time [81].[16] 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 [6]. Also see [97].

[13] There were differences as well. The perceptron had separate layers of input nodes and fixed weights designed to help with visual classification tasks. The perceptron learned from a discrete on/off signal, while the Adaline learned from a continuous weighted input signal (a more modern way of doing things; cf. chapter 10.) Widrow was an engineer who implemented the Adaline using a special electrical components called a "memistor" which Widrow designed himself.

[14] 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 [1].

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

[16] This is known as the "perceptron convergence theorem."

> 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.[17] In the hardware implementation shown in Fig. 2.10, the toggle switches control input node activations, the knobs control weight strengths, and the dial shows the activation of an output node.[18]

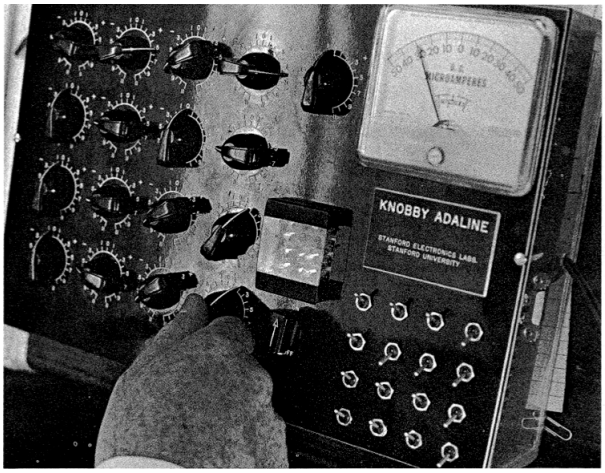


Figure 2.10: A hardware implementation of Widrow and Hoff's "Adaline" network, which Widrow called the "knobby Adaline" on account of the prominent grid of knobs on the left, which control weight strengths, and which were manually adjusted to implement their learning algorithm. Inputs were produced using 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`.

2.5 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.[19] The dropoff in interest has been attributed to several causes. The canonical story is that networks with a single layer of adjustable weights were shown by Minsky and Papert to suffer certain fundamental limitations [63] (cf. Chap. 10). 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.

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.[20]

[17] 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` which also shows him demonstrating his old hardware Adaline.

[18] For Widrow's personal recounting of the Adaline and its history, see his interview in *Talking Nets* [1]. Also see the videos referenced in the Figure caption.

[19] See PDP vol 1, ch. 1 [82]; Haykin p. 43 [36]; Fausett p. 24 [21]. 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 [1].

[20] Debates about the the status of neural networks in this period are covered in some detail in Talking Nets [1].

2.6 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, including the discovery of the backpropagation algorithm, which overcomes the limitations associated with perceptrons. While neural networks were being shown to be more powerful than had previously been thought, the competing program of symbolic AI was running into problems [18].

Another 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 aspects of cognition, and clarifying how connectionist networks differ from symbolic AI models [82]. John Hopfield's models of associative learning in recurrent networks (i.e. "Hopfield nets", discussed in chapter 7) were also influential in this period, in part because Hopfield presented his work in an especially clear, mathematically precise way.[40].[21]

2.7 Second Decline and Second Resurgence: The Deep Learning Revolution

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, Phillip Jannert clearly expressed this concept of a second decline: "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" [42, Ch. 18].

However, several things happened that have brought attention back to neural networks : (1) larger datasets for training neural networks have become available (hence current interest in "big data"), (2) higher performance hardware for parallel neural network computing has emerged, for example by 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] (3) new neural network architectures have emerged, and (4) more principled ways of training networks have been developed that provide the area with improved theoretical grounding, e.g. via "Bayesian hyperparameter optimization."[23]

Whereas most neural networks through the 1990s were just three layers, newer *deep learning* architectures can have many layers of units. These many-layered deep-learning networks existed as far back as the 1970s, but it is only in the 2010s that a variety of technical hurdles relating to this type of network were surmounted. Indeed many have described the period since the 2010s as a deep learning revolution, or as the decade of deep learning.[24]

[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 [1].

[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 5), 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] On moving beyond parameter "tweaking" or "fiddling" to more systematical methods, see the first 6 minutes of this video `https://www.youtube.com/watch?v=sq2gPzlrM0g`.

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

Chapter 3

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]

3.1 Neurons and synapses

3.1.1 Neurons

Neurons are brain cells, which have all the machinery any cell has: mitochondria, Golgi apparatus, a nucleus whose DNA is actively expressing hundreds of 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.

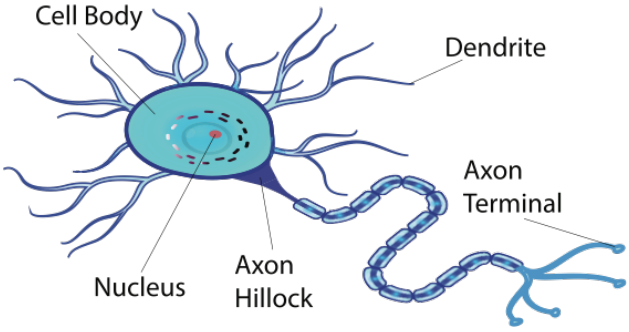


Figure 3.1: Main structures of a neuron.

Most neurons have dendrites, axons, and a cell body (see Fig. 3.1).[2] These are not really distinct

[1] Some useful general references include Kandel (2000) [43] and Gazzaniga (2002) [28]. An outstanding online source is `https://science.eyewire.org/home`. A detailed book length treatment of the topics outlined in this chapter is [70], 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]

3.1.2 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. 3.2.[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 signals. As different axons attaching to a neuron 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,

---

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.

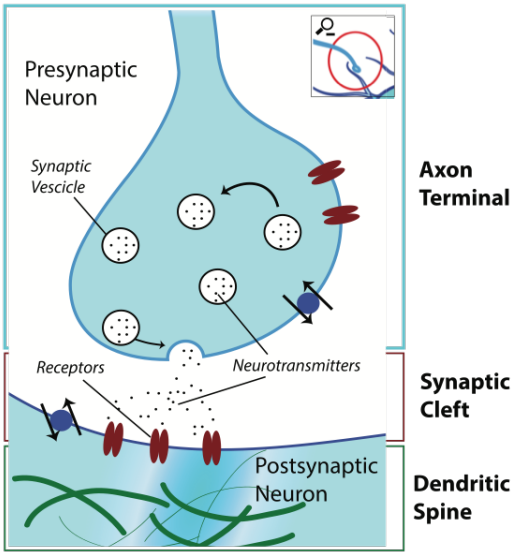


Figure 3.2: Some structures associated with a synapse.

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. 3.3. In the left panel, two inhibitory synapses are activated successively, and the membrane potential is reduced each time and then begins to approach the resting potential again.[6] In the right panel, two excitatory synapses are activated successively, which raises the membrane potential above threshold, and an action potential is fired.[7]

As more excitatory signals are received, 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.[8] 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). This is the basis of "rate-coding" models. In these models, the number in a node represents or is proportional to a neural firing rate. Many neural network models explicitly or implicitly use rate-coding.

The idea that synapses can be excitatory or inhibitory and that the neuron sums together these signals is the basis of many neural network models. The details vary. Some models describe the changing membrane potential directly. Some models simulate the action potential using discrete spiking events (see chapter 14). Sometimes membrane potentials and spikes are not mentioned at all, and the neuron model describes the *rate* at which the neuron fires action potentials ("rate based" models). In connectionist models, almost all 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. 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.[9] Long term potentiation is the basis of the Hebb rule, discussed in chapters 2

[6]Notice that though the inhibitory 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.*

[7]These processes can be explored in the Simbrain workspace *spikingNeuronTwoInputs.zip*.

[8]More accurately, it is average number of spikes per unit time, during a the time in which a neuron is measured.

[9]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.

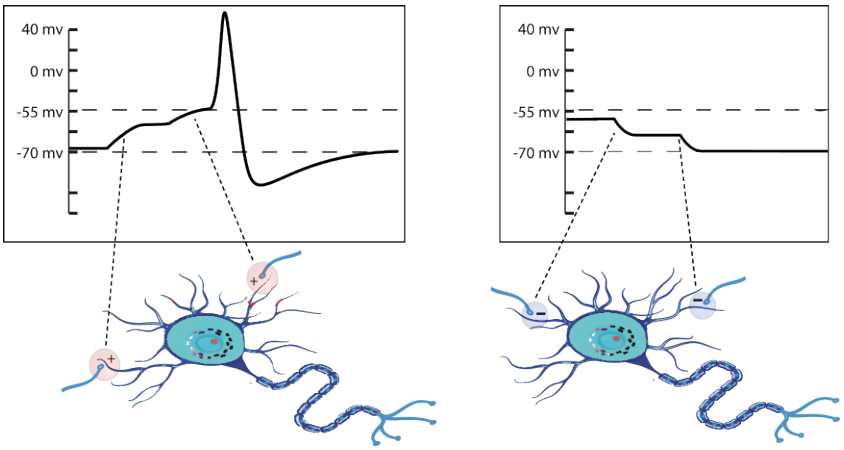


Figure 3.3: (Left) Two successive excitatory inputs increase the neuron's membrane potential beyond the threshold potential, after which it fires an action potential. (Right) Two successive inhibitory inputs reduce the neuron's membrane potential below an excited level of -58mv.

and 7. The basic idea of the Hebb rule is that "neurons that fire together, wire together."[10]

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 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.

3.1.3 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.[11] 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.

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

[11] 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." [13].

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.

3.2 The Brain and its Neural Networks

In this section we describe regions of the brain, functions associated with them (summarized in Fig. 3.4), 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, 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.

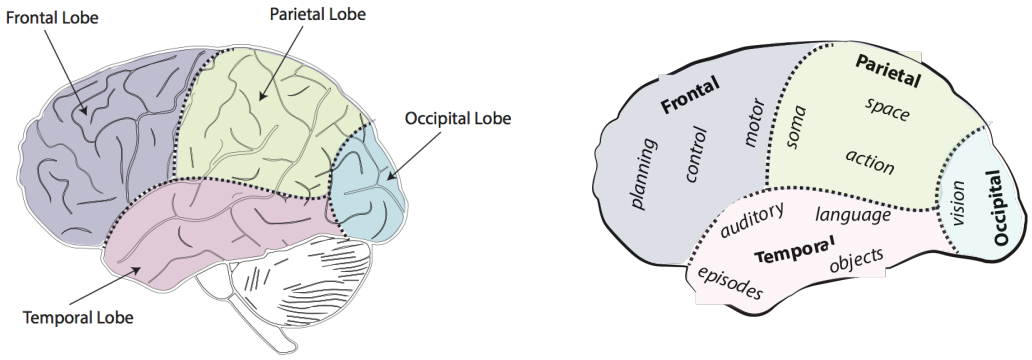


Figure 3.4: (Left) The lobes of the brain. (Right) Rough map of functions, abilities, and conceptual domains associated with major brain areas.

3.2.1 Cortex

Fig. 3.4 (Left) shows the major regions of the brain. Fig. 3.4 (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 7). 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 10). This view, known as the "predictive coding" or "predictive processing" view, originates in part in computational models of visual cortex [76], but it has since developed into a more general view about the structure of perception and cognition and their realization in the brain [15].

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.

3.2.2 The Occipital Lobe

The **occipital lobe** and some of its features are shown in Fig. 3.5. 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 blindness (**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.

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

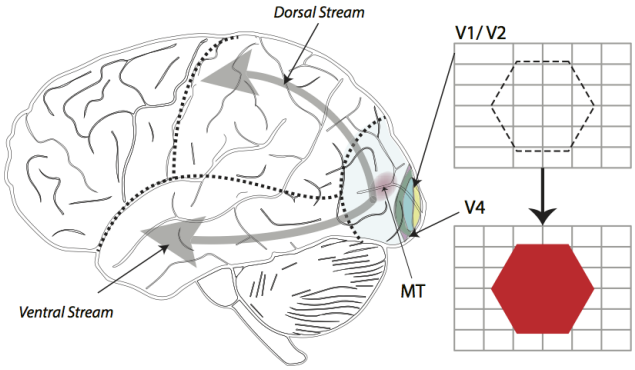


Figure 3.5: The visual cortex and associated structures.

is processed in a spatially organized manner. We will see that multiple sensory regions contain topographic maps of their respective sensory information. In chapter 7 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. 3.5). We will discuss these pathways in more detail below.

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 deep network known as "AlexNet" [49] is shown in the top panel of figure 3.6. 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. 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 2), while IT layers respond to dogs, cats, etc.[12]

3.2.3 The Parietal and Temporal Lobes

The **temporal lobe** is involved in auditory processing and semantic processing (see Fig. 3.7). 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**, where patients are unable to understand either spoken or written language.

The temporal lobe also receives connections from the visual processing centers of the occipital lobe, via the ventral stream (Fig. 3.5). 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

[12]There has also been some skepticism about deep network approaches to human vision [9].
[13]More specifically, the temporal lobe in the dominant hemisphere, which is usually the left hemisphere.

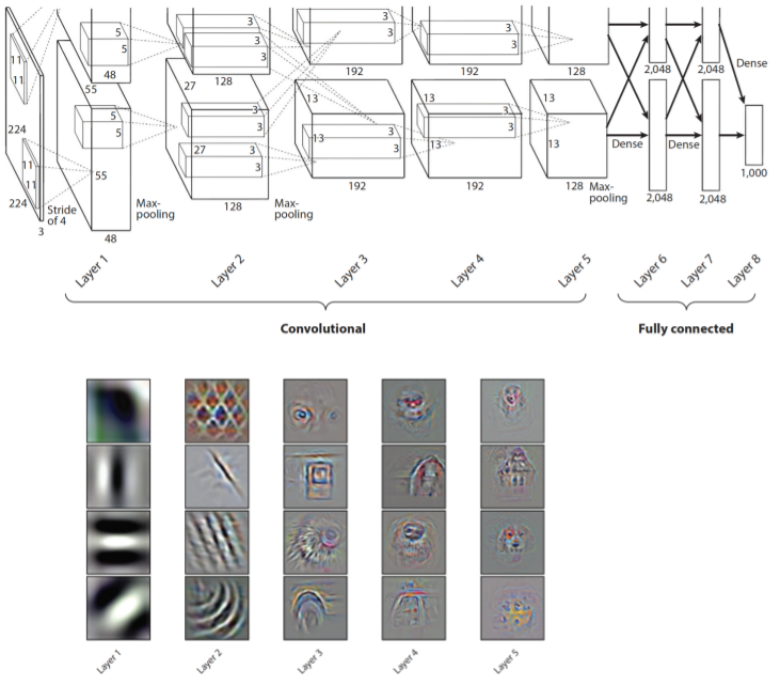


Figure 3.6: A well-known deep neural network architecture AlexNet (based on [49]) whose activations match those of actual brain areas. The top level shows the network architecture, and the bottom panel shows receptive fields (activations that maximally activate specific nodes) of a similar network [32]. 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.

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 [7]. 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. 3.7, 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.

The parietal lobe also receives tactile information from the body via the **somatosensory cortex** (Fig. 3.8), 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. 3.8, more sensitive body parts

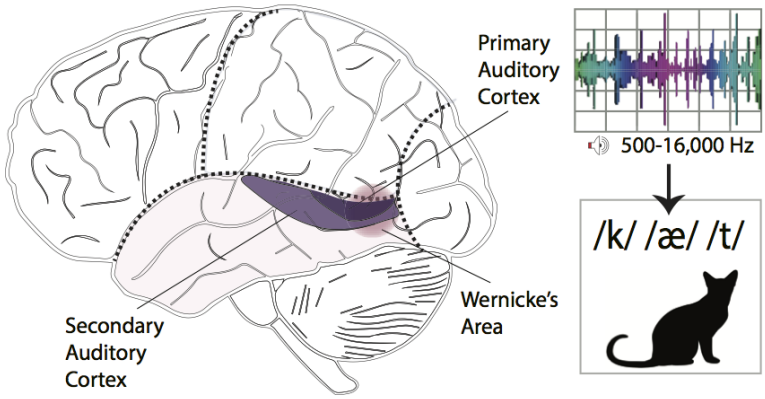


Figure 3.7: Auditory cortex and associated structures.

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.

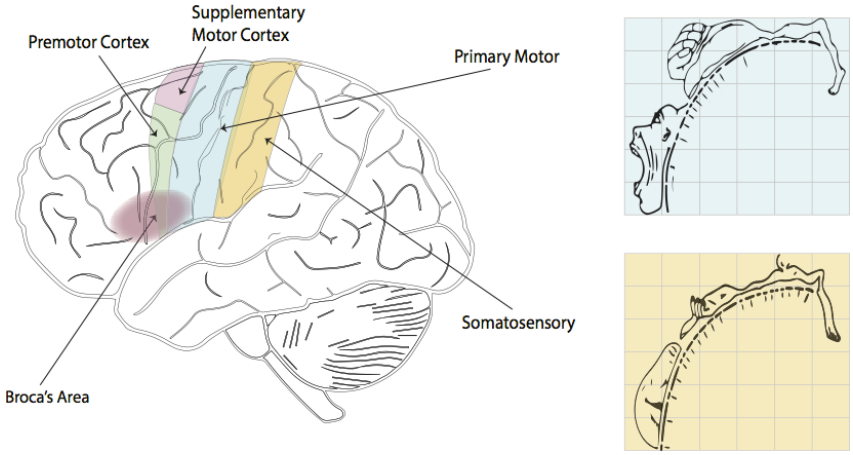


Figure 3.8: Somato-sensory and motor processing.

3.2.4 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.[14] 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. 3.8)

[14]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.

is responsible for many language functions, including gesture, understanding of action and action-language, and language production.

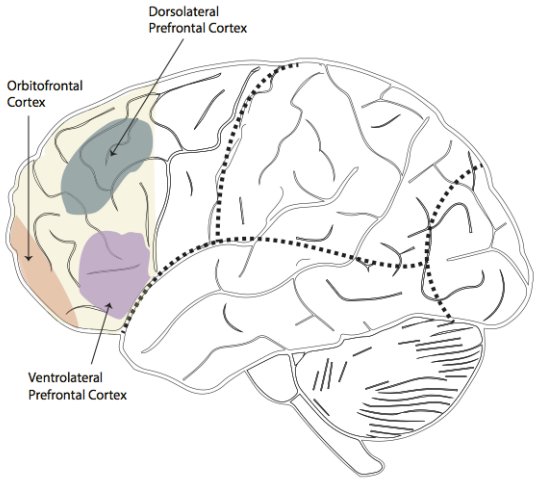


Figure 3.9: The prefrontal cortex.

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. 3.8). 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.

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 8), whereby activations tends to settle into stable patterns for a time, which are maintained in cortical "stripes" [48]. 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.[15]

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.

3.2.5 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

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

memory. However, there are also many other specialized circuits that have been modeled by distinctive forms of neural network model. See figure 3.10 for an overview of the structures we discuss here.

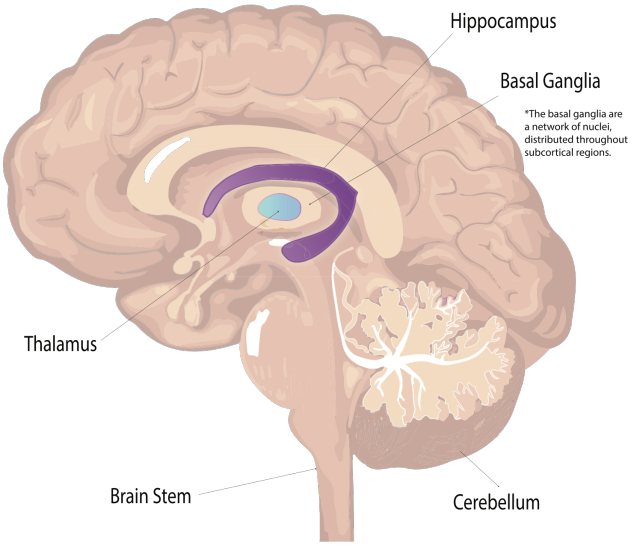


Figure 3.10: Some specific structures in the brain with specific neural network structures.

The **hippocampus** is a kind of short-to-medium-term memory system attached to the bottom of the cortex.[16] 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.[17] While learning in the cortex is slow, learning in the hippocampus is fast. 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 conditoning). 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.[18] In fact,

[16] 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.

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

[18] 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/`

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 [84, 68]. 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.[19]

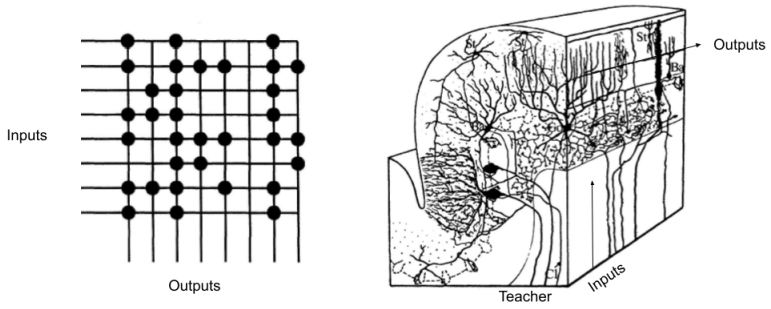


Figure 3.11: The cerebellum as a pattern associator, mapping sensory states to motor actions. (Left) An input-output architecture: synapses where input lines and output lines overlap can be turned on when a teacher signal turns on. (Right) Associated areas of the cerebellum thought to correspond to these 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 [56]. As can be seen in figure 3.11, 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.

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

Chapter 4

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 4.1 (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 3.

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]

4.1 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 4.1. 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 4.1 (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 5, 7, and 10, 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 14.

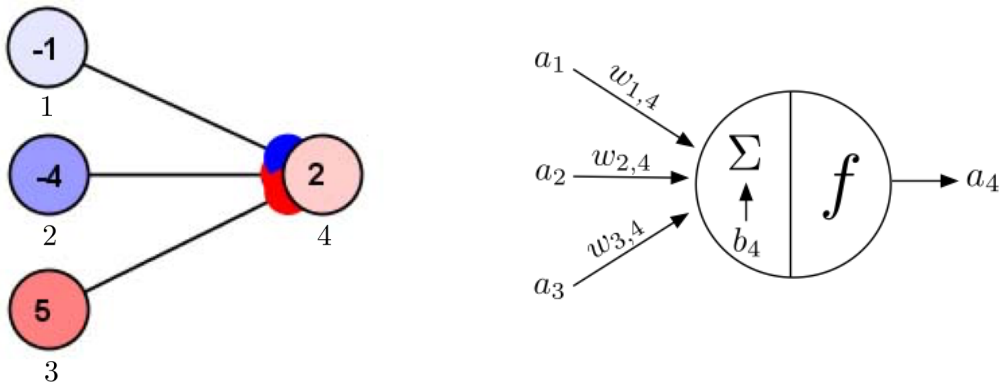


Figure 4.1: (Left) A simple neural network with three nodes attached to one node via three weights. (Right) Schematic of the same network to illustrate the notation being used here. Nodes $1, 2, 3$ are connected to node 4. In this example, $a_1 = -1$, $a_2 = -4$, $a_3 = 5$ and (though weight strengths 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 4.1, 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 4.5.

[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: $\sum_{j=1}^{N} i^3 = \left(\frac{N(N+1)}{2}\right)^2$.

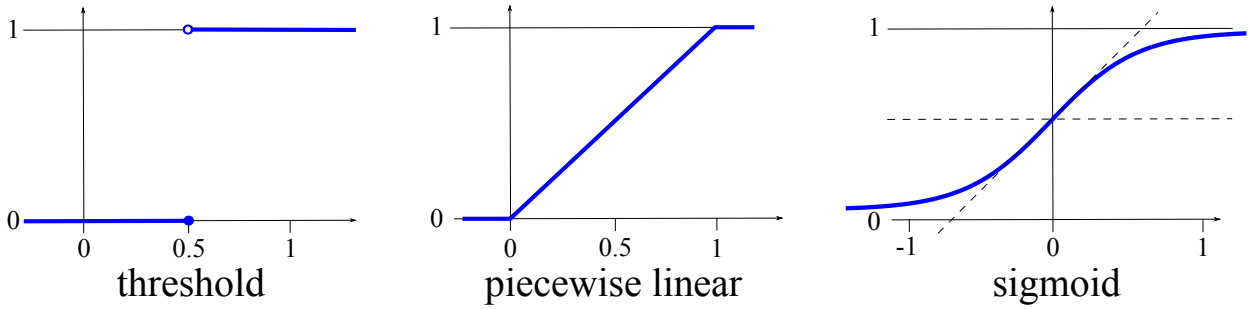


Figure 4.2: The graphs for three activation functions. In each of the graphs, the horizontal axis is the weighted input, n_k, and the vertical axis is the activation, a_k. Left: A threshold activation function with $(u, \ell, \theta) = (1, 0, 0.5)$. Middle: A piecewise linear activation function with $(u, \ell) = (1, 0)$. 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 4.1), 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 $\ell = -1$.

4.2 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 3.1.2).

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

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

[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 4.2). 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 4.2. 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.

4.3 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]

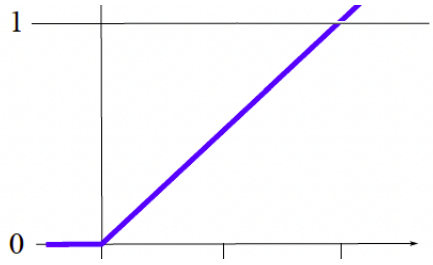


Figure 4.3: Graph for the rectified linear or "relu" activation function.

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 4.3).[7] Thus, they are a piecewise linear function with no upper bound:

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

This can be useful because it removes all negative activations, making overall activation in a large network more sparse. The function also has a simple derivative[8], and other mathematical properties that have made it

[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 in to a regular, non-piecewise linear node by turning off *clipping*.

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

[8] The derivative is 0 for $n_k <= 0$, and 1 otherwise, except at 0 (the point of discontinuity), where the derivative is not defined.

extremely popular, especially for deep networks. In fact, multiple varieties of relu function are now available, like "gelu". 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.

4.4 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 4.2. 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 in a neighborhood of the inflection point. As the population begins to fill the dish, 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 4.2 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 2).

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 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

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

bound ℓ. The larger m is, the more rapidly the sigmoid function converges to its bounds. Figure 4.2 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.

4.5 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.[10]

1. Consider the network shown in Fig. 4.1. 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. 4.1. 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:** $n_4 = 2$.

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 **??**). 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:** $a_k = 1$.

[10] 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.

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. 4.2. 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 5

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 4 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 8, and can be used to think about many features of neural network models in an intuitive way.[1]

5.1 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. 5.7.

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:

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

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 the vector space. For example, the set of all of ordered pairs of real numbers is a 2 dimensional real vector space. The set of all ordered triples is a 3 dimensional real vector space.

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 5.1).[5]

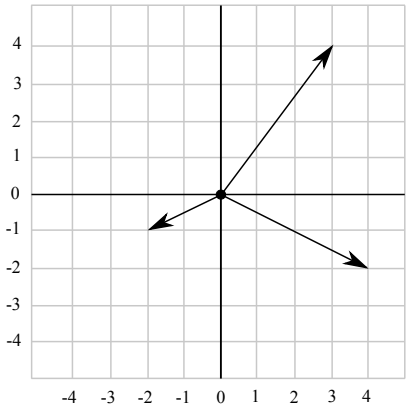


Figure 5.1: Each vector in a 2 dimensional vector space is associated to a point in the Euclidean plane by treating the components of the vector as the coordinates of the point. Try to find the vectors $(3,4)$, $(-2,-1)$, and $(4,-2)$.

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:

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

[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 5.1).

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. 5.2.[6]

5.2 Vectors and Vector Spaces in Neural Networks

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

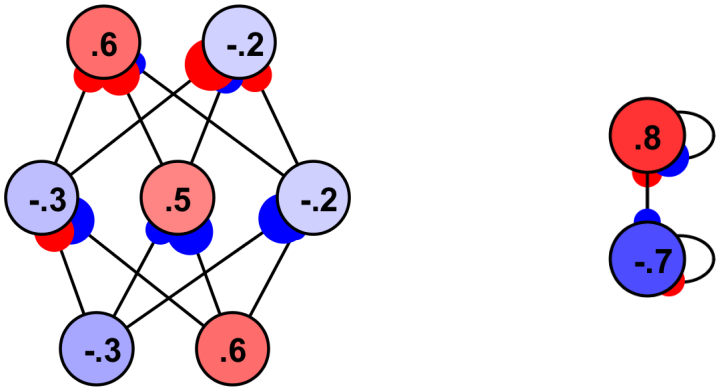


Figure 5.2: A feed forward and recurrent network in Simbrain. Try to identify the dimensionality of the activation space, input space, hidden unit space, output space, and weight spaces of each network. Left: A feed-forward neural network with activations showing. Right: A 2-node recurrent network with 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 5.2 (Left) from the bottom to the top and left to right (as in figure 5.6), 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 5.2 (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 5.2 (Right) from top to bottom (as in figure 5.6), 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 5.2 (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.

[6]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.

We can also talk about vectors of weights, or **weight vectors**, which exist in **weight spaces**. The feed-forward network in figure 5.2 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 5.6). 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 10 and 7), 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 5.2 with zeroed out activations, labeled node indices, and weight strengths is shown in figure 5.6 below. Some sample weight vectors for these networks are:

Feed forward network, neuron 3 fan-in $(1, -2)$
Feed forward network, neuron 3 fan-out $(-2, 0.9)$
Feed forward network, neuron 7 fan-in $(0.9, -1, -1.2)$
Recurrent network, neuron 2 fan-in $(1, -2)$

Some of these weight vectors live in 2 dimensional weight space, some live in a 3 dimensional weight space. Of course for larger networks, fan-in and fan-out vectors can be in higher dimensional weight spaces.[7]

5.3 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 5.4 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 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).[8] 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 5.3), Antarctica and Greenland look huge, and things are especially distorted at the two poles, farthest away from the equator.

[7] 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.

[8] 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.

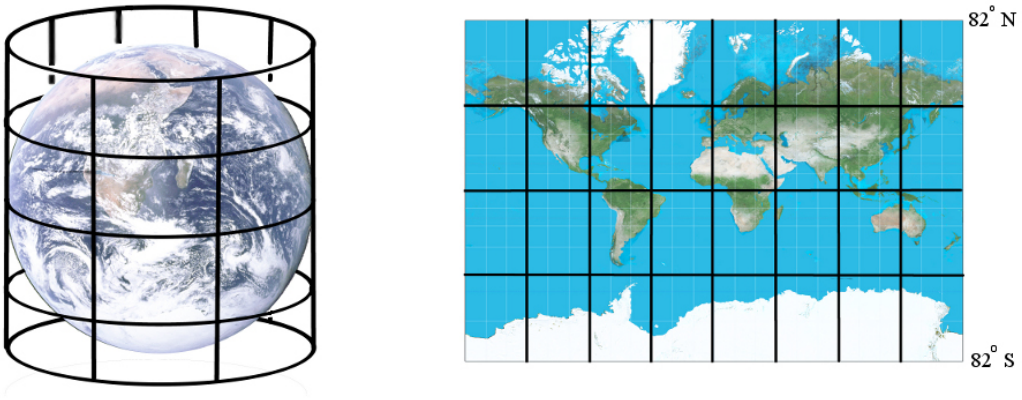


Figure 5.3: The Earth's surface in 3 dimensional space is rendered as a flat, 2 dimensional surface by the Mercator projection method. Most of the distortion produced by the projection occurs near the Earth's poles so small regions around the poles are cropped out from the maps. Most of the continents 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 5.4. 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 5.4. 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 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.

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.[9]

[9]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`.

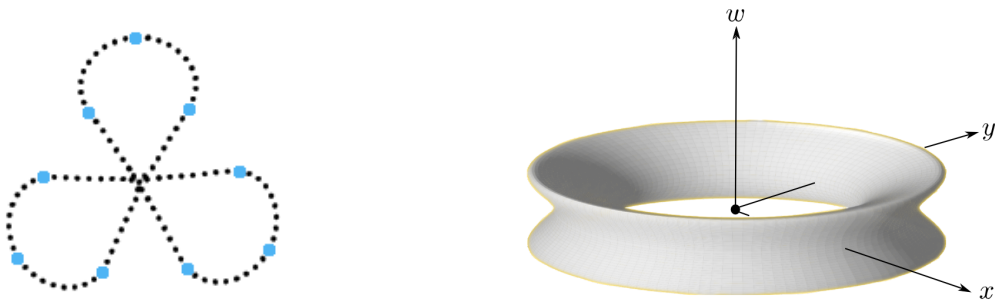


Figure 5.4: (Left) The projection of a symmetrical curve in a 6 dimensional space so that we can see its symmetry. The nine vectors listed at the beginning of section 5.3 are shown as nine large blue dots. (Right) A symmetrical surface in a 4 dimensional space is projected so that we can see its symmetry.

5.4 The Dot Product

The **dot product** is a simple but important function defined for pairs of vectors in a vector space.[10] The dot product is different from scalar multiplication (scalar multiplication and the scalar product are defined in section 5.7). 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.

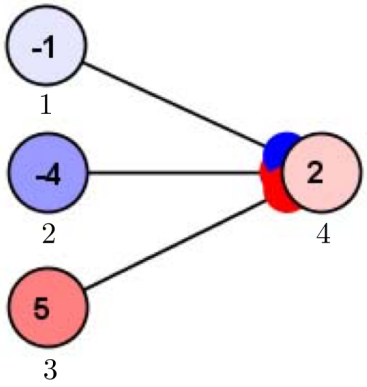


Figure 5.5: Simple feed-forward network with nodes labeled. The dot product can be used to compute the weighted inputs to node 4.

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

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

[10]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.

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

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

so they must be orthogonal.

Notice that we can concisely represent the weighted input (see chapter 4) 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 5.5 is $(-1, -1, 1)$, then the net-input is

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

5.5 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.[11] 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.[12]

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.[13]

Using this method, we can represent the weight matrix for the recurrent network in figure 5.6 as:

$$\begin{pmatrix} 2 & 1 \\ 1.1 & -2 \end{pmatrix}$$

For example, the strength of the weight connecting node 2 to node 1 is 1.1, so that's the value in the second row, first column of the matrix. The connections from a node directly back to itself are the diagonal components of this kind of weight matrix. If a weight does not exist, it is represented by a 0 in the corresponding matrix. In the recurrent network in figure 5.6 (Right), if there were no self-connections the

[11] 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.

[12] 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.

[13] 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 14.

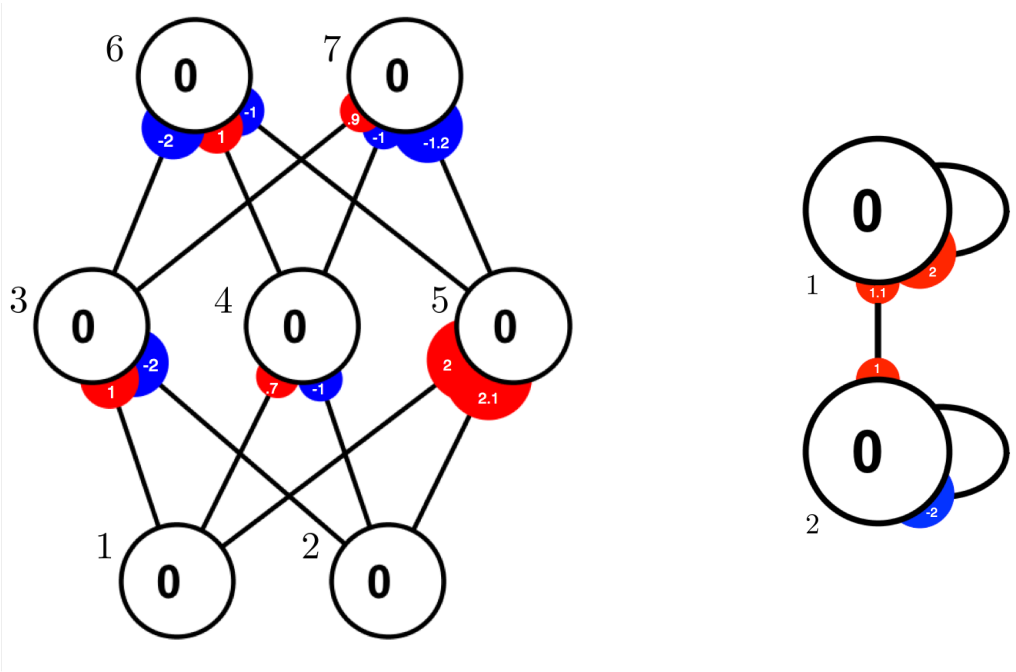


Figure 5.6: The feed forward and recurrent network shown in figure 5.2 with zeroed out activations, labeled nodes, and weight strengths shown. This makes it possible to see how weights in a network can be linked to values in a matrix.

diagonal of the matrix would have zeros in it: there would be no weight from node 1 to 1 or from node 2 to 2. There are a lot of zeros in the full weight matrix for the feed-forward network in figure 5.6. For example, there is no weight from node 1 to 2, so there is a 0 in the first row, second column, of that network's full weight matrix.

For the feed-forward network in figure 5.6, 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).

However, in practice it is natural to focus on the non-zero blocks, and to treat each set of connections between a pair of layers as its own matrix. That is, each weight layer of a feed-forward network can be represented by its own weight matrix. For example, for the weight layer connecting the input node layer to the hidden node layer in figure 5.6 (Left), we focus on this block of weights:

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

Given the way we are representing weight matrices, this can be thought of as a sequence of column vectors, one for each fan-in weight vector of the three hidden layer nodes. We can also think of it as a sequence of row vectors, one for each fan-out weight vector of the three input layer nodes.

5.6 Matrix Product

We can use matrix representations of weights to facilitate the computation of weighted inputs (see chapter 4). Consider the feed-forward network in figure 5.6, which uses linear activation functions without bias (we are also ignoring clipping) so that node activations just are weighted inputs. Given an input vector to that network, we can compute the hidden unit vector using *matrix multiplication*. We will not give a general definition of matrix multiplication 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.[14] That is, we multiply the input vector by the intervening weight matrix to obtain the hidden unit vector. 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 5.6 has linear activation functions and 0 bias, and its input activation vector is $(1, 2)$. 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.

We can do the same kind of thing with the recurrent network, but in this case we will be determining its activations at successive time steps (the output at a given time becomes the new input). Suppose the recurrent network has an initial activation vector of $(1, 2)$. What is its activation vector at the next time step? We can compute this as follows:

$$\begin{pmatrix}1 & 2\end{pmatrix}\begin{pmatrix}2 & 1\\ 1.1 & -2\end{pmatrix} = \big((1)(2)+(2)(1.1),\ (1)(1)+(2)(-2)\big) = \begin{pmatrix}4.2 & -3\end{pmatrix}$$

We can now repeat this process, using the network's previous output as its new input:

$$\begin{pmatrix}4.2 & -3\end{pmatrix}\begin{pmatrix}2 & 1\\ 1.1 & -2\end{pmatrix} = \big((4.2)(2)+(-3)(1.1),\ (4.2)(1)+(-3)(-2)\big) = \begin{pmatrix}5.1 & 10.2\end{pmatrix}$$

If we continue this process (and thereby simulate a "run" of this recurrent network) we will get activation vectors

$$(21.42, -15.3),\ (26.01, 52.02),\ (109.24, -78.03),\ (132.65, 265.30),\ (557.13, -397.95),\ \ldots$$

We will see in chapter 8 that this is one *orbit* in the network's activation space.

Clearly the activations of this network are blowing up! The orbit is heading off to infinity. If we were to use a piecewise linear, threshold, or sigmoid activation function, then these activations would be contained in the interval $[\ell, u]$ (see chapter 4).

[14] 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 13). 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.

5.7 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.[15] 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

[15] 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.

(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 8. 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 vector space. There are many different bases[16] 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.

5.8 Exercises

1. What is the dimensionality of the input space, hidden unit space, output space, weight space, and activation space in Fig. 5.6 (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. 5.6 (Right)? **Answer:** 4-dimensional and 2-dimensional.

3. What is the dimensionality of the weight space and activation space in Fig. 5.7? **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:** $(1 \cdot -1) + (-1 \cdot 1) + 5 = -1 - 1 + 5 = 3$.

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

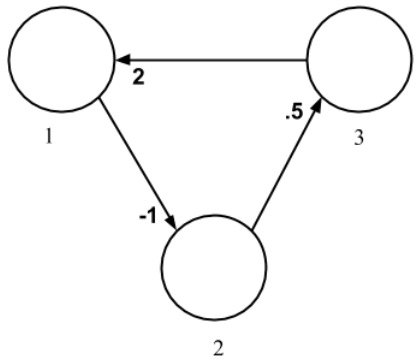


Figure 5.7: A recurrent network with three nodes labelled $1, 2, 3$ and weights $w_{1,2} = -1, w_{2,3} = 0.5, w_{3,1} = 2$.

9. What is the matrix representation of the weights in the network in Fig. 5.7? **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. 5.7? **Answer:** (-1).

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

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. 5.7 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. 5.7 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 6

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 section 1.3), 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.

6.1 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 6.3.

[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 validate the model on a separate set of data.

We will not discuss obtaining data or exploratory data analysis here. We discuss data wrangling in section 6.3. 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 8. In a neural network, these parameters are usually weights and biases (chapter 4). Parameters change the dynamics of a dynamical system. For a recurrent network, that means they change what activation patterns occur over time. They change a phase portrait. For a feed-forward network, they change 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 10, 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, which recognizes letters or faces in images. In chapter 13, 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]

6.2 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. 6.1, 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 6.1: 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 6.1, 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", 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 6.1, 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 6.2. 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.

6.3 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 5), 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`.

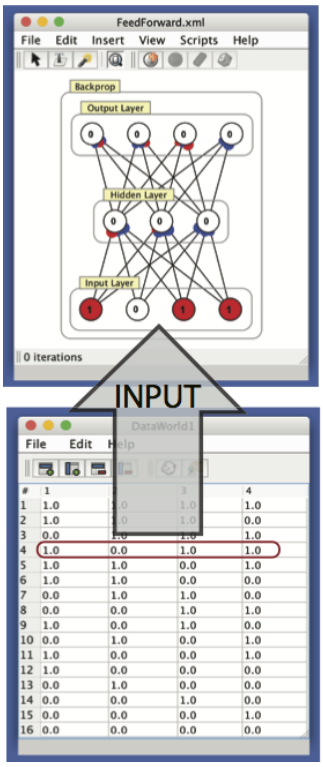


Figure 6.2: An example of an input dataset, which illustrates one standard way datasets are used with neural networks. Each row of the dataset is thought of as one input vector for the neural network.

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 other is 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. 6.3. Notice that the cylinder column in Fig. 6.1 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.[15] 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.[16]

- 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.[17]

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 6.4 shows the *mtcars* dataset of figure 6.3 after all columns have been rescaled to lie between 0 and 1. A simple and common 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*.[18] 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 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 this is sometimes called a z-score). Anything above 1 is unusually large, and similarly for values below -1. However, this method allows values above 1 and below -1.

[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] A tensor is like a generalized matrix, a multi-dimensional array. A vector is a rank 1 tensor, a matrix is a rank 2 tensor, a set of matrices is a rank 3 tensor, a rank 4 tensor is a set of these sets, etc. See `https://en.wikipedia.org/wiki/Tensor`.

[16] This is a part of the field "Distributional Semantics", where the semantic information of linguistic terms is related to their distributional properties in natural corpora. See `https://en.wikipedia.org/wiki/Distributional_semantics` for an overview. A famous algorithm for word embedding is word2vec. See `https://www.tensorflow.org/tutorials/word2vec`.

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

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

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

	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 6.4: Data from Fig. 6.1 rescaled to $(0, 1)$.

6.4 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.[19]

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. 6.2.
- **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. 6.8.

Examples of each type of dataset are shown in Fig. 6.5.

[19]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 **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.

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 10. 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. 6.5. 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 6.5: From left to right: an input dataset, output dataset, target dataset, and labeled dataset.

6.5 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. 6.6. The idea is that we first train the network using training data, and then validate it using testing data.[20]

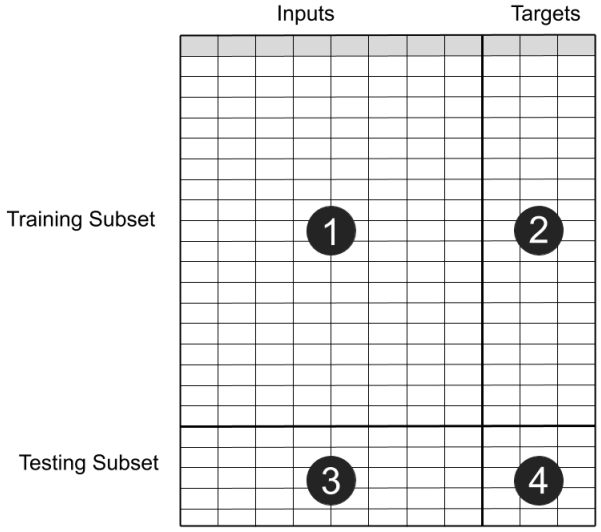


Figure 6.6: The rows of a labeled dataset (with inputs and targets) divided into a training and a testing subset. The training subset is used to train our model, and the testing subset is used to validate how well it generalizes. Thus we end up with four tables: (1) training inputs, (2) training targets 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.[21] 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.

[20] 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.

[21] 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`.

6.6 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.

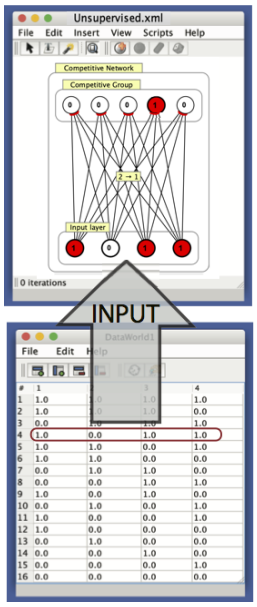


Figure 6.7: Illustration of how unsupervised learning relies only on an input dataset.

Unsupervised learning is learning without a teacher, which is covered in chapters 7 and 9. 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 6.7. 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 3. 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 [93]. It can also be useful in machine learning, since we oftentimes don't have training data available (hence the term "unlabeled data").

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 6.8. 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 10. 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 3, both of which are thought to learn via supervised learning.

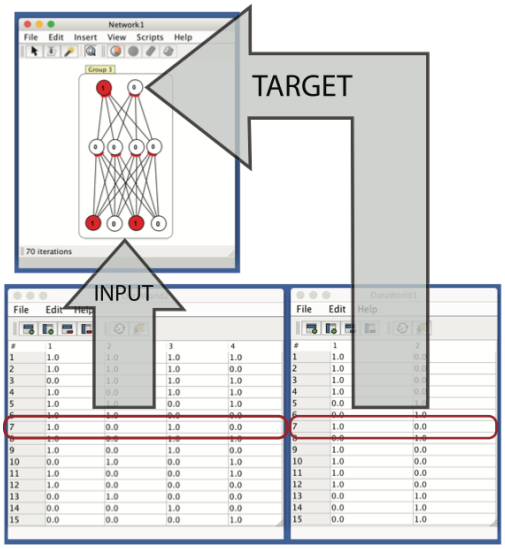


Figure 6.8: Illustration of how supervised learning uses both input and target datasets.

6.7 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 11 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 11), 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 13.[22]

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.

[22] 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.

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 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) [89]. Another famous example is a system that learned to play a bunch of old Atari video games [64]. The nice thing about reinforcement learning is that it is fairly realistic. As noted in chapter 3, 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 7

Unsupervised Learning

Jeff Yoshimi

7.1 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 6). 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 9, we consider how unsupervised methods can be used to train recurrent networks.

7.2 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 4), 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 3) 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 [37].

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: $w_{1,2} = -0.94$.

7.3 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 2 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 5) 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" [75]. 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 7.1. 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.

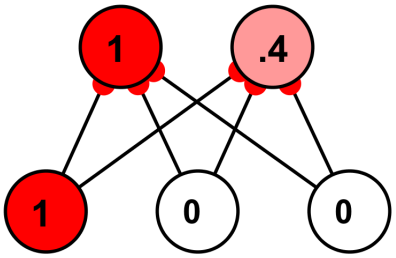


Figure 7.1: Learning one association in a feed-forward network using the Hebb rule.

(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 the 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 6).

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 10), 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 5 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 6), 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.

7.4 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 6); 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 5 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.

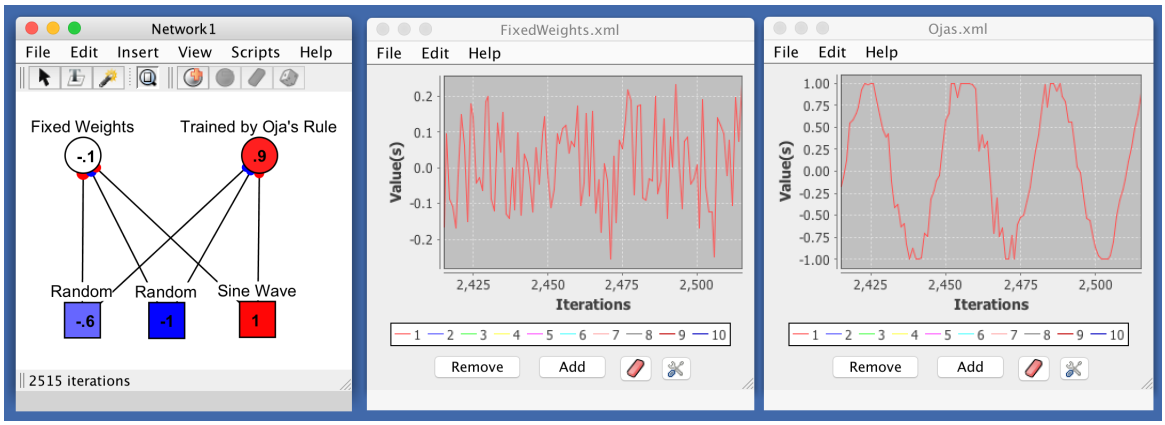


Figure 7.2: An illustration of Oja's rule for dimensionality reduction. Two of the input nodes produce random noise, and the third produces a sine wave. Think of each output node as performing a 3-to-1 dimensionality reduction. The output node on the left was initialized with fixed random weights that 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 [21], 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. 7.2. 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 3). 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.

7.5 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. 7.3). 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."

7.5.1 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 5 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 3.6 in chapter 3.

[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. 7.3, 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. 7.3 and Fig. 7.4. 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 7.4. 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 6) 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.

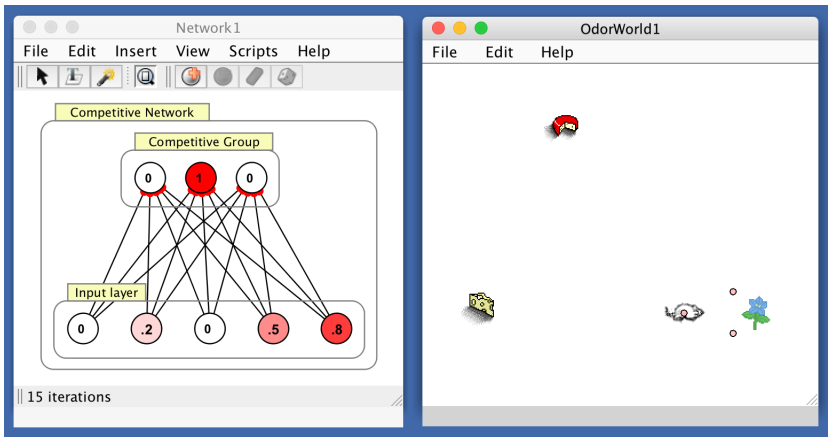


Figure 7.3: Competitive network with 3 output nodes, which can learn to detect up to 3 clusters in an 5-dimensional input space. Inputs correspond to smells of flowers. Once the network has been trained, the output nodes can be labelled. Which output node ends up classifying which type of smell will 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. 7.4, 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. 7.3 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.

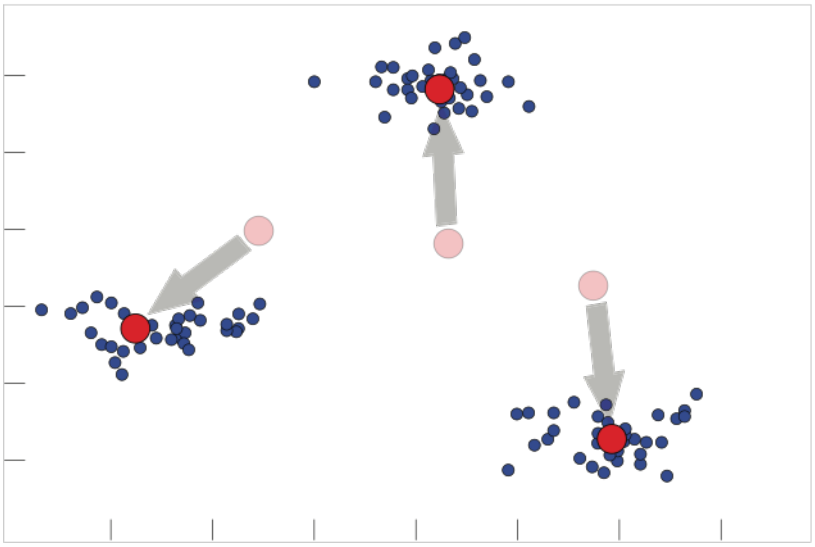


Figure 7.4: Geometrical illustration of how competitive networks learn. If we think of this as a representation of the smell inputs in Fig. 7.3, then each blue dot corresponds to one smell from one location in the virtual world, and the three clusters correspond to the three objects: Swiss cheese, Gouda, 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.

7.5.2 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 [46].

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 [46].

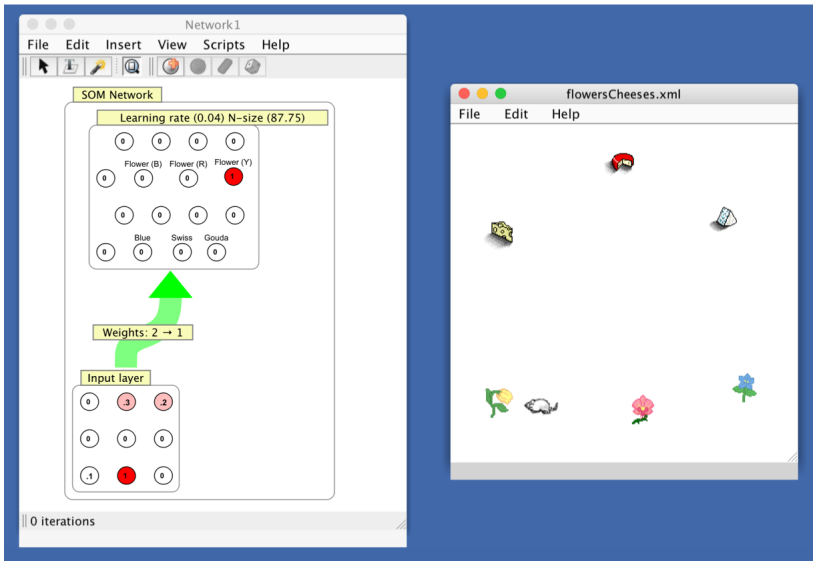


Figure 7.5: A self organizing map after it has been trained for several hundred iterations, with some of the categorizations it produces hand-labelled.

Fig. 7.5 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. 7.5. 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. 7.6 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. 7.7 we see a top view of 150 nodes in the output nodes of a SOM.

Recall from chapter 3 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. 7.6). 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) [46].

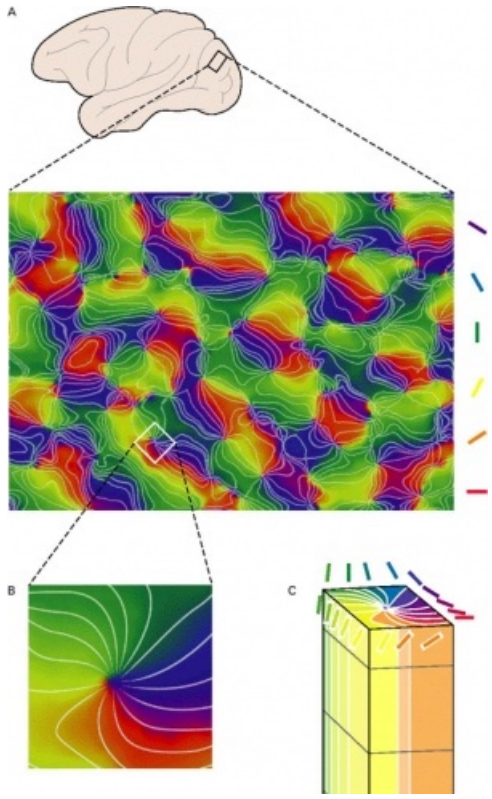


Figure 7.6: Topographically organized edge detectors in visual cortex. The main panel shows a top-down view on an area of cortex, with neurons colored according to what kind of edge they represent. To the right of the panel these edge orientations are shown. Notice that nearby neurons represent similar 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. 7.7 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)[46]. 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]

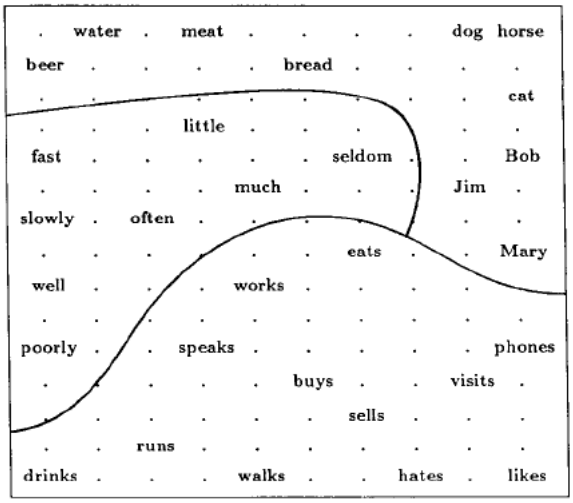


Figure 7.7: A self organizing map trained to represent semantic features of sentences. Each dot corresponds to an output node, and labels show what concepts these nodes have learned to represent. Notice that nouns, adjectives, and verbs are represented in specific parts of the network. From Kohonen (1990), 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 8

Dynamical Systems Theory

Jeff Yoshimi, Scott Hotton

In chapters 4, 5, and 10, 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. 8.1, 8.2, 8.3, 8.4, and 8.6 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 9). 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 14). 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 13). There is also a body of theoretical work showing that neural networks can approximate any continuous dynamical system with arbitrary precision [41]. This shows that the human brain has a great deal of flexibility in the kinds of behaviors and processes it could in principle produce.

8.1 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.

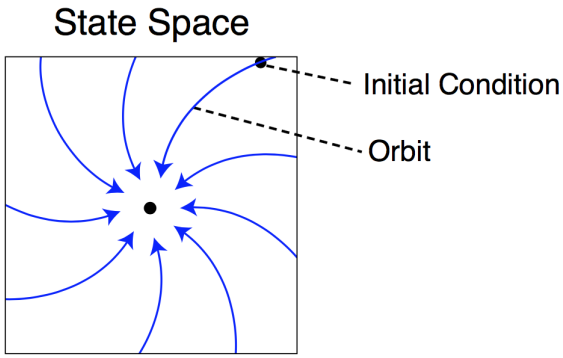


Figure 8.1: Some basic components of a dynamical system. The square region is the *state space* for a 2-dimensional system. Each point in that region is a *state*. Each state can be treated as an *initial condition*. When the system is run, an *orbit* unfolds from the initial condition. A picture like 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. 8.2 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 $(-1, -1)$.

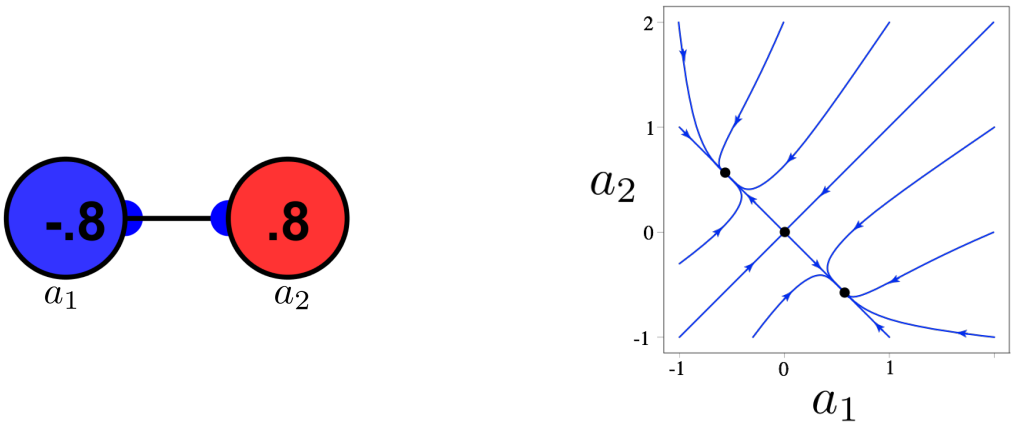


Figure 8.2: A 2 node recurrent network (left) and its phase portrait (right). The phase portrait has two fixed point attractors at $(-.8, .8)$ and $(.8, -.8)$, each with its own basin of attraction. There is a fixed point at the origin $(0, 0)$ which is attracting in one direction but repelling in the 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 5). 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 5 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. 8.1 and 8.2, 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.

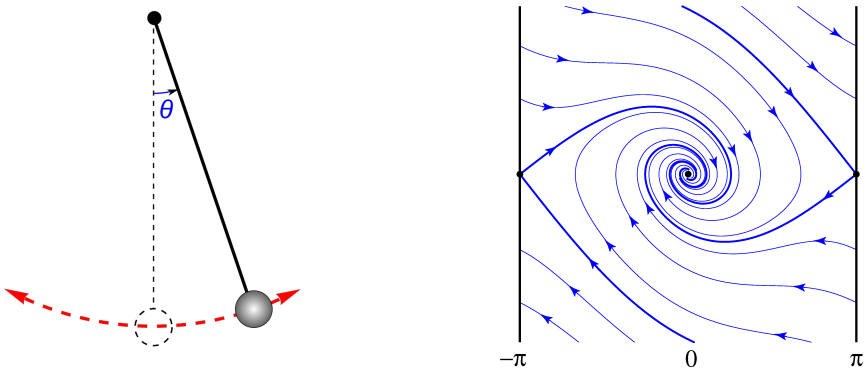


Figure 8.3: Penduluum (Left) and its state space with a phase portrait (Right). The vertical dimension of the state space corresponds to angular speed; the horizontal dimension corresponds to angular displacement away from hanging straight down.

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. 8.3 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. 8.3 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. 8.4. 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.

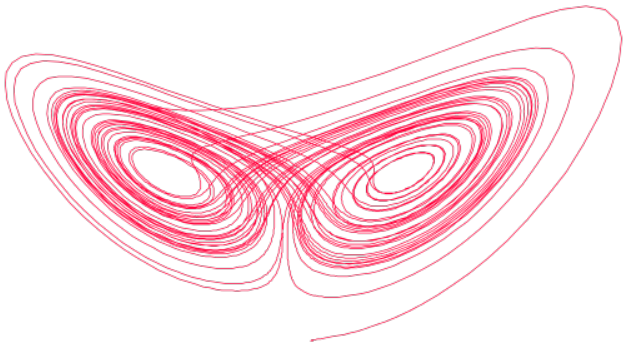


Figure 8.4: An orbit from a famous chaotic system called "The Lorenz Attractor." Nearby initial conditions can diverge arbitrarily far apart (within the attractor) over time.

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. 8.1 and Fig. 8.2 most of the orbits are curves that tend towards specific points. Similarly for the pendulum's state space. In the chaotic system in Fig. 8.4, 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.

8.2 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. 8.5. In that figure, a single point (left) gives rise to a circle (right) when the parameters are changed past a critical value.

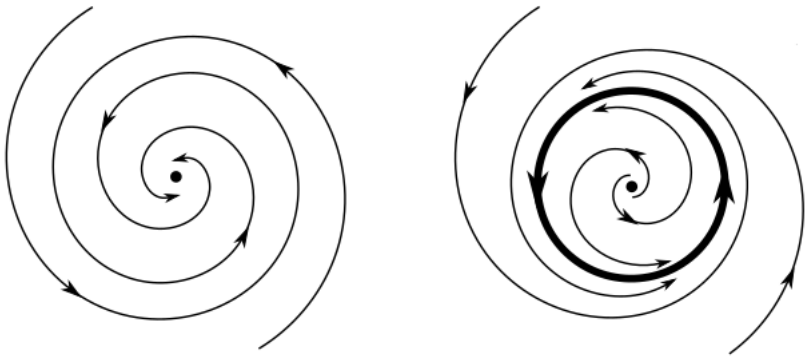


Figure 8.5: A bifurcation where a fixed point goes from being attracting to repelling, and in which an attracting periodic orbit appears. (This is called a "Hopf bifurcation").

8.3 Classifying 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. 8.1 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. 8.6.

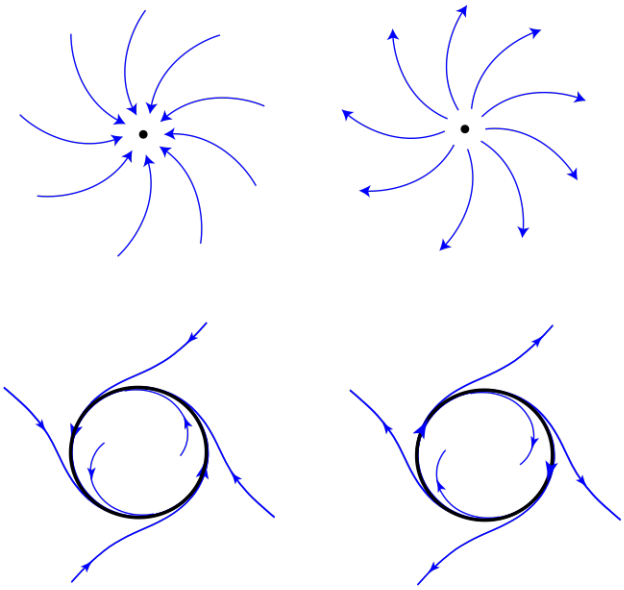


Figure 8.6: An attracting fixed point (upper left), repelling fixed point (upper right), attracting periodic orbit (lower left), and repelling periodic orbit (lower right). Attractors / repellers are shown in black; transient of orbits approaching or leaving attractors and repellers are shown in blue.

8.3.1 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. 8.6. 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. 8.6.[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.

There are other shapes besides points, lines and loops. Some orbits in higher dimensions are donut shaped "tori", for example. Others are even more complex, e.g. fractal shaped.

8.3.2 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. 8.6. 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. 8.6. 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 8.7). 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. 8.2 has two attractors with two basins of attraction.

As we discuss further in chapter 9, 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 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

[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.

[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.

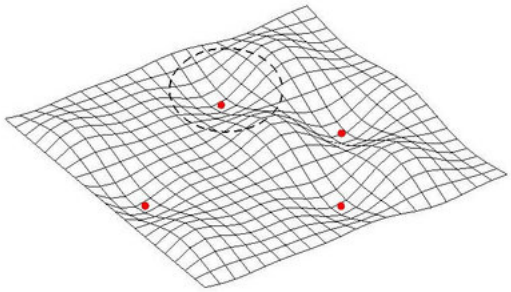


Figure 8.7: Attractors and basins of attraction pictured using a hill and valley metaphor.

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. 8.2 is attracting in one direction, and repelling in another.

8.3.3 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. 8.6. 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. 8.8 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]

[11] Notice that the orbits are not smooth. This emphasizes that a computer produces a discrete approximation of continuous time processes.

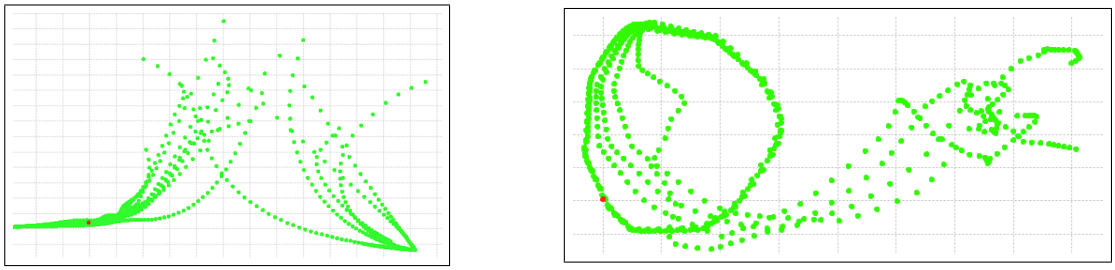


Figure 8.8: Phase portraits generated using Simbrain. A system with two attracting fixed points (Left) and a system with one attracting periodic orbit (Right). Both are projections from a 25 dimensional activation space to 2 dimensions. The red point is the current point in each simulation.

Chapter 9

Unsupervised Learning in Recurrent Networks

Jeff Yoshimi

9.1 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 7 with an analysis of unsupervised learning in recurrent networks. We will see that the tools of dynamical systems theory (chapter 8) are quite useful in this context. In chapter 13 we discuss more complex recurrent networks and their applications to psychology, neuroscience, and engineering.

9.2 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. 9.1, 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 7), 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. 9.2. It's the same idea as with the simple "L" pattern in Fig. 9.1, 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.

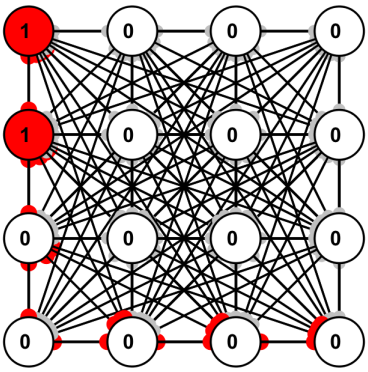


Figure 9.1: Cued recall of an "L" shaped pattern. Notice that the "L" pattern is visible in the red weights, which indicate how the pattern will be completed. In the past, those neurons fired together, so they were wired together by the Hebb rule.

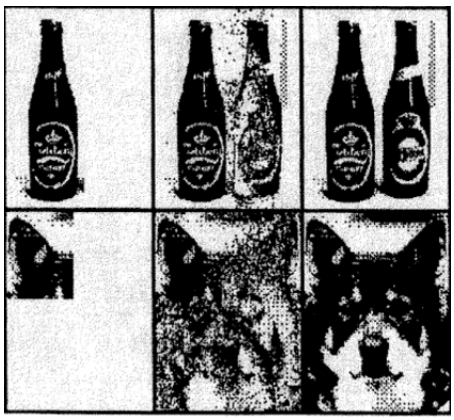


Figure 9.2: Pattern completion in a recurrent network with $130 \cdot 180 = 23,400$ nodes. The left-most image in each row shows the initial cue. The middle image shows the network part-way through the pattern completion process. The right image shows the final image. From Hertz et al. 1991. The network is a Hopfield network, which uses a variant on the Hebb rule.

Figure 9.3 shows how pattern completion in recurrent auto-associators can be understood in terms of dynamical systems theory (also see chapter 8). 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. 9.2 are fixed point attractors in the 23,400-dimensional activation space of that network. The "L" visible as a trace in Fig. 9.1 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. 9.2 are initial conditions, as is the upper part of the "L" in Fig. 9.1. 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]

9.3 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.

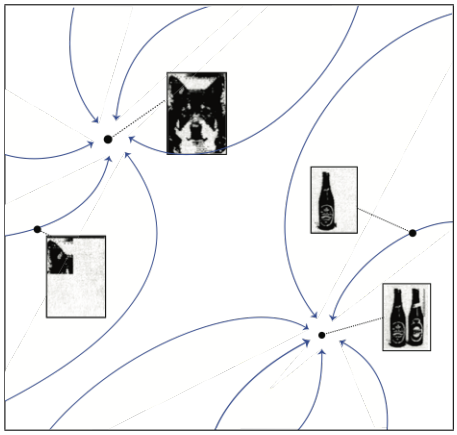


Figure 9.3: Schematic diagram of the attractors and basins of attraction for the images shown in Fig. 9.2. Fragments of images correspond to initial conditions, that evolve under the network dynamics to completed images, which correspond to attracting fixed points.

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 9.4) 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 9.4 and 9.5 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.

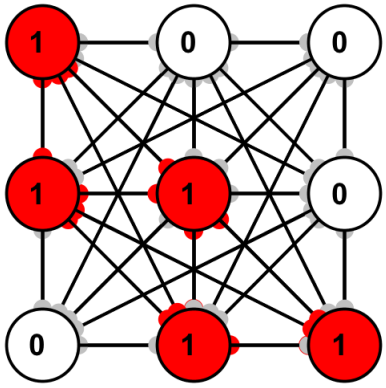


Figure 9.4: An auto associative network trained on a single binary pattern using Hebb's rule. Notice that only weights between co-active nodes have been strengthened. Since the other nodes have activations of 0, weights to and from them are not changed.

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

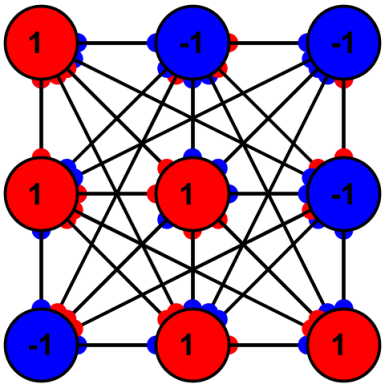


Figure 9.5: Bipolar version of pattern from Fig. 9.4, after training on a bipolar version of the same pattern. Notice how some of the weights have turned blue. Thus, when the pattern is recreated incompatible patterns will be inhibited. This version of the network can learn several patterns.

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 8 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.

9.4 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.

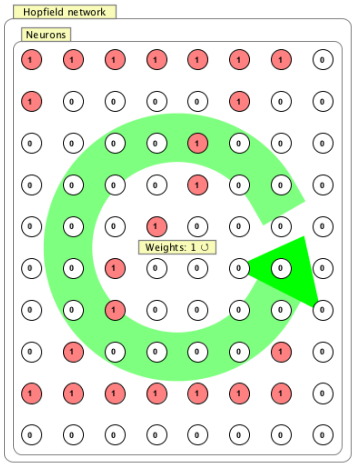


Figure 9.6: Hopfield network trained on the pattern for a "Z". Though a binary pattern is displayed behind the scenes this network uses bipolar patterns, with -1's where 0s are.

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 [40].

A Hopfield network with 80 nodes is shown in Fig. 9.6 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 [21], p. 140 and section 3.3.4. Also see Hopfield's original discussion at [40], p. 2556.

Chapter 10

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 6.4).[1] We say, "if you see this pattern, produce this other pattern." This is sometimes called "learning with a teacher." We saw in chapter 7 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 8 of finding attracting fixed points, but this time in weight space rather than activation space.

In chapter 11, 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 13, 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.

10.1 Labeled datasets

With supervised learning, we tell the network what we want it to do. There is a teacher or trainer. Recall from chapter 6 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.

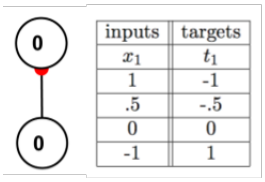
We will say that a labeled dataset is *compatible* with a feed-forward neural network if (1) the number of columns in the input dataset is the same as the number of input nodes in the network, and (2) the number of columns in the target dataset is the same as the number of output nodes in the network. 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 6: 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.

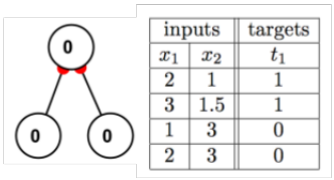
[2] This is sometimes called the 'Goldilocks principle'. See `https://en.wikipedia.org/wiki/Goldilocks_principle`.

used to train any network compatible with it. Examples of labeled datasets and compatible networks are shown in figure 10.1.

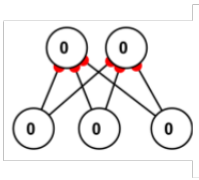
A labeled dataset can be thought of as a contract for a pattern association task: we'd like to train a network to come as close possible to implementing the input-output associations described by our labeled dataset. In the language of vector-valued functions, we are training a network to approximate 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 10.1: 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).

10.2 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. 8 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 11) 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 "row errors" and "overall error", which are discussed in greater detail in Sect. 10.6. Roughly speaking row errors says how far away the outputs produced by an input vector are from the target values for that input, and overall error combines the row errors.

To train a network, we select a subset of a compatible labeled dataset, a **training subset** (section 6.5). 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:
   (a) Set the input-layer activations of the network to $\mathbf{x}_r$.

[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.

(b) Compute the network's output vector $\mathbf{y}_r$.

(c) Compute row-errors by comparing the targets $\mathbf{t}_r$ with the outputs $\mathbf{y}_r$.

(d) Update $p_1, \ldots, p_n$ with the goal of reducing row 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."

10.3 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 6. 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 10.2 shows an example of a feed-forward network that classifies pixel patterns as one of 26 letters.

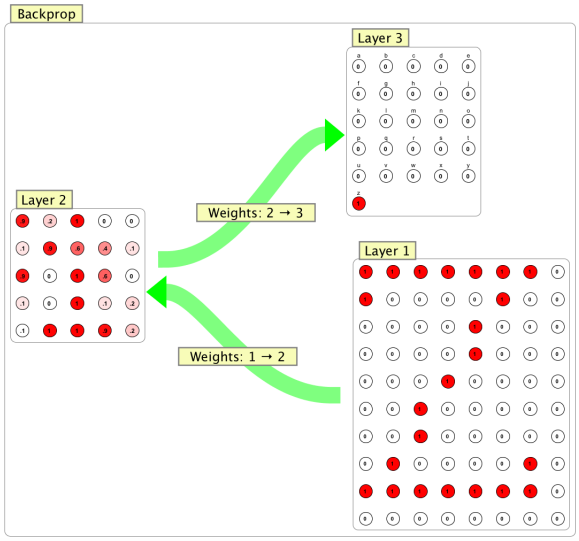
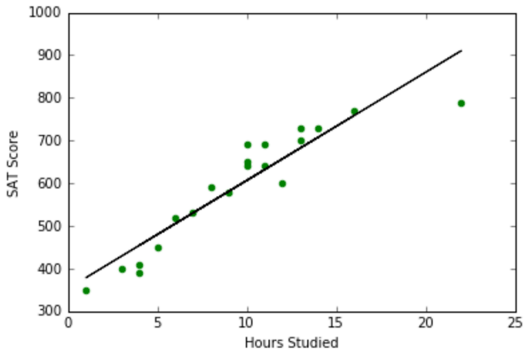


Figure 10.2: An example of classification. A feed-forward network trained via supervised learning to classify letter inputs as specific letters.

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 7 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



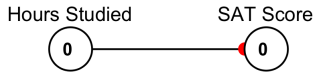


Figure 10.3: An example of regression. (Left) The data used to train the network. Hours of study vs. score on the SAT. (Middle) A plot of the data with a regression line. These are points in an input-target space, as discussed in chapter 10. This line can be used to predict how well someone will do based 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 10.3, 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.

10.4 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 6). 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 [6], chapter 3).

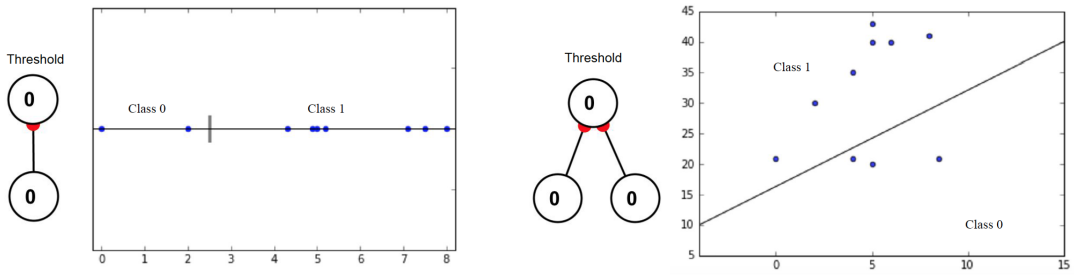


Figure 10.4: A classification task for a 1-1 network (Left) and a 2-1 network (Right). Both networks use threshold activation functions on the output nodes. Points in the input space are shown in blue. The decision boundaries between points classified as 0 or 1 are shown in gray. On the left, the decision 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 10.4 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. 10.4 and 10.5 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 10.4 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 10.4 the decision boundaries are $1 - 1 = 0$ dimensional (a point) and $2 - 1 = 1$-dimensional (a line).

10.5 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 10.5, 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.

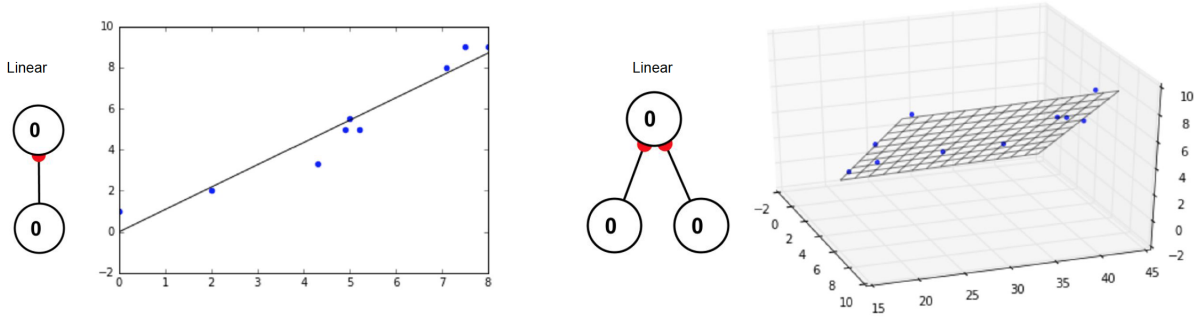


Figure 10.5: A regression task for a 1-1 (Left) and a 2-1 network (Right) network of linear nodes. Datapoints in the input-target space are shown in blue. The network implements a linear function from inputs to outputs, which is a line in the two dimensional input-target space on the left, and a plane 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 10.5, 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 10.5 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 10.5 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 10.4 (right) looks like the regression line in figure 10.5 (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 5.3), 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.

10.6 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 (cf. the intuitive overview in Section 10.2). 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 *row errors*, which are associated with specific rows of an output-target dataset). 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 the row 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 (cf. Chapter 6) 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 row errors $(1 - .9, 1 - .7) = (.01, .03)$.

Error is a pretty easy idea. If we want our network to produce the output vector $(1, 1, 1)$ but instead it produces $(-1, 0, .2)$, well, we have kind of bad error. But if we train it and then it starts to produce the output vector $(1, 1, .9)$, then we are doing much better! Intuitively, the first output is about 4 units "off", but the second output is about .01 off. That's all the basic idea involves. But saying this mathematically requires a bunch of symbols that will be intimidating to some of you. Just 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: mean squared error, cross-entropy, etc. Knowing which to use, and when, is a more advanced topic.[10] Here we are just introducing the idea. We focus on an error function called *sum of squared error* or SSE.

Suppose we are given our network and labeled dataset, and have computed the output dataset. To compute SSE we subtract each output vector $\mathbf{y}_r$ in the output dataset from its corresponding target vector $\mathbf{t}_r$, square the components of the resulting errors, and sum them:

$$SSE = \sum_{r=1}^{R} (\mathbf{t}_r - \mathbf{y}_r)^2 = \sum_{r=1}^{R} (\mathbf{t}_r - \mathbf{y}_r) \bullet (\mathbf{t}_r - \mathbf{y}_r)$$

The row errors $(\mathbf{t}_r - \mathbf{y}_r)^2$ are obtained by subtracting the output vector for row r from the target vector

[9] These are also known as "cost functions", "loss functions", or "objective functions".

[10] One main consideration is whether you are training a network on a classification task or a regression task. SSE, which we consider here, can be used on both.

for r using component-wise subtraction[11], then squaring the resulting vector, in the sense of taking its dot product with itself (refer to chapter 5 for a review of the dot product).

A sample computation for a network with two outputs is shown in figure 10.6.[12] Notice that SSE is low. The outputs are close to the targets. All we are really doing is subtracting a bunch of values to see how close they are, squaring them (so that the errors become positive), and then adding them up. It's an easy idea but there is a lot of notation to track.

$$\sum_R \left(\begin{array}{|cc|}\hline t_1 & t_2 \\ \hline 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ \hline \end{array} - \begin{array}{|cc|}\hline o_1 & o_2 \\ \hline .8 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & .9 \\ \hline \end{array} \right)^2 = \sum_R \left(\begin{array}{|cc|}\hline \text{errors} & \\ \hline 1\text{- }.8 & 0\text{-}0 \\ 0\text{-}0 & 1\text{-}1 \\ 1\text{-}1 & 0\text{-}0 \\ 0\text{-}0 & 1\text{-}.9 \\ \hline \end{array} \right)^2 = \sum_R \left(\begin{array}{|cc|}\hline \text{errors} & \\ \hline .2 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & .1 \\ \hline \end{array} \right)^2 = \sum_R \begin{pmatrix} (.2,0) \bullet (.2,0) \\ (0,0) \bullet (0,0) \\ (0,0) \bullet (0,0) \\ (.1,0) \bullet (.1,0) \end{pmatrix} = .04 + 0 + 0 + .01 = .05$$

Figure 10.6: Computing SSE for a network with two outputs. Error is fairly low.

$$\sum_R \left(\begin{array}{|cc|}\hline t_1 & t_2 \\ \hline 1 & 0 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ \hline \end{array} - \begin{array}{|cc|}\hline o_1 & o_2 \\ \hline 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ 1 & 1 \\ \hline \end{array} \right)^2 = \sum_R \left(\begin{array}{|cc|}\hline \text{errors} & \\ \hline 1\text{- }1 & 0\text{-}1 \\ 0\text{-}1 & 1\text{-}1 \\ 1\text{-}1 & 0\text{-}1 \\ 0\text{-}1 & 1\text{-}1 \\ \hline \end{array} \right)^2 = \sum_R \left(\begin{array}{|cc|}\hline \text{errors} & \\ \hline 0 & -1 \\ -1 & 0 \\ 0 & -1 \\ -1 & 0 \\ \hline \end{array} \right)^2 = \sum_R \begin{pmatrix} (0,-1) \bullet (0,-1) \\ (-1,0) \bullet (-1,0) \\ (0,-1) \bullet (0,-1) \\ (-1,0) \bullet (-1,0) \end{pmatrix} = 1 + 1 + 1 + 1 = 4$$

Figure 10.7: Computing SSE for a network with two outputs when error is higher, as might happen when we start with random weights, or as here, weights that initially cause the output to always be 1.

$$\sum_R \left(\begin{array}{|c|}\hline t_1 \\ \hline 1 \\ 2 \\ 3 \\ 4 \\ \hline \end{array} - \begin{array}{|c|}\hline o_1 \\ \hline .9 \\ 1.9 \\ 2.9 \\ 3.9 \\ \hline \end{array} \right)^2 = \sum_R \left(\begin{array}{|c|}\hline \text{errors} \\ \hline 1 - .9 \\ 2 - 1.9 \\ 3 - 2.9 \\ 4 - 3.9 \\ \hline \end{array} \right)^2 = \sum_R \left(\begin{array}{|c|}\hline \text{errors} \\ \hline .1 \\ .1 \\ .1 \\ .1 \\ \hline \end{array} \right)^2 = \sum_R \begin{array}{|c|}\hline .01 \\ \hline .01 \\ .01 \\ .01 \\ \hline \end{array} = .01 + .01 + .01 + .01 = .04$$

Figure 10.8: Computing SSE for a network with one output node.

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 10.7.

The same idea works for a network with 1 output node, and is much easier, since our vectors just have one component and are thus scalars. So we just subtract, square, and add! See 10.8.

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 6.5).

10.7 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 10.4 and 10.5. We will be talking about parameter spaces, or to make things easier, weight spaces (recall these have come

[11] This is easy to understand using an example, for example $(\mathbf{t}_1 - \mathbf{y}_1)$ where $\mathbf{t}_1 = (1,1)$ and $\mathbf{y}_1 = (2,3)$. Then $(\mathbf{t}_1 - \mathbf{y}_1) = (1,1) - (2,3) = (1-2, 1-3) = (-1,-2)$.

[12] In these figures, $\sum_R$ is shorthand for $\sum_{r=1}^{R}$

up in chapters 8 and 7). 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 10.9 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.

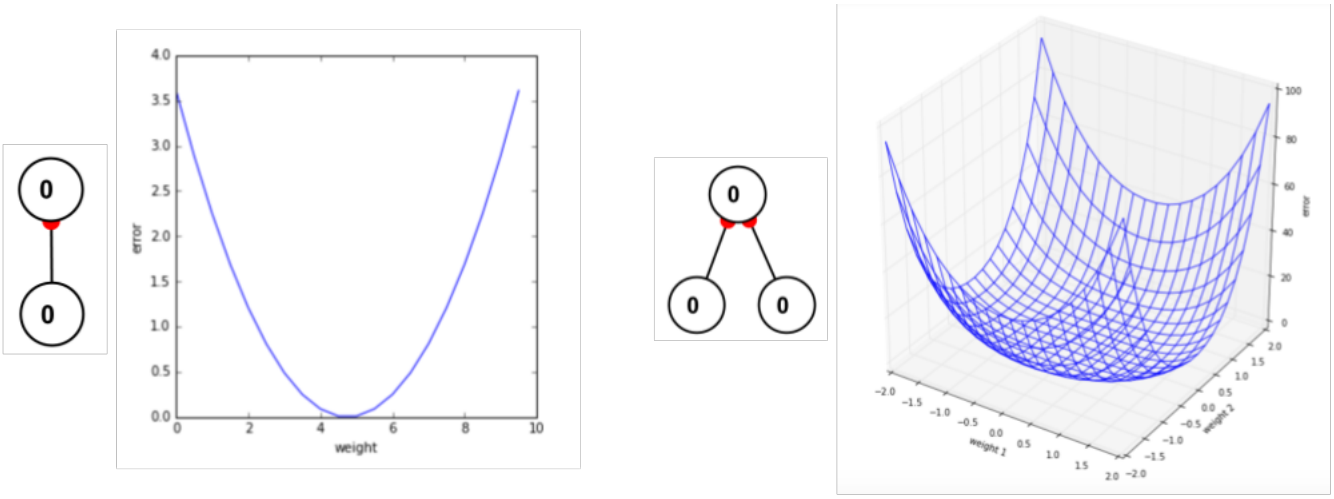


Figure 10.9: (Left) error surface for a 1-to-1 network where only the weight is adjusted. (Right) Error surface for a 2-to-1 network where the two weights are adjusted. In each case, the error surface also depends also the training data (the tables in figure 10.1).

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.

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.

[13] `https://en.wikipedia.org/wiki/Mathematical_optimization`.

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.

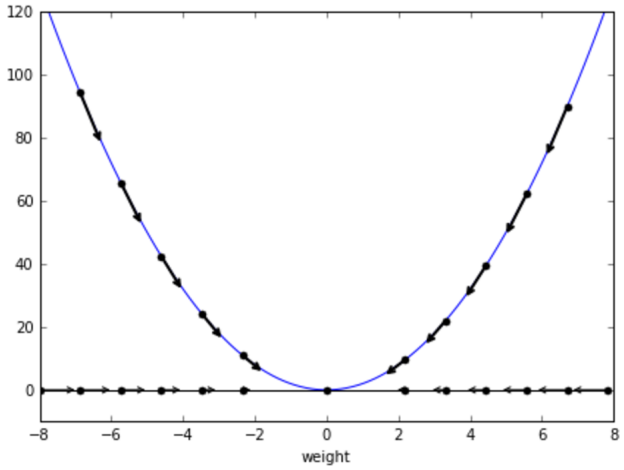


Figure 10.10: Error gradient on an error surface. The actual changes that happen in the weight space are shown on the horizontal axis.

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 10.9 (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. 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 10.10. 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 8 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 8, 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.[15] What we are doing is like what we did there when we looked for fixed points of a recurrent 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

[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]See the screenshot here: `http://www.simbrain.net/Documentation/docs/Pages/Network/training/trainingDialog.html`.

only gradually change. Both cases are shown in figure 10.11. 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.[16] 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.

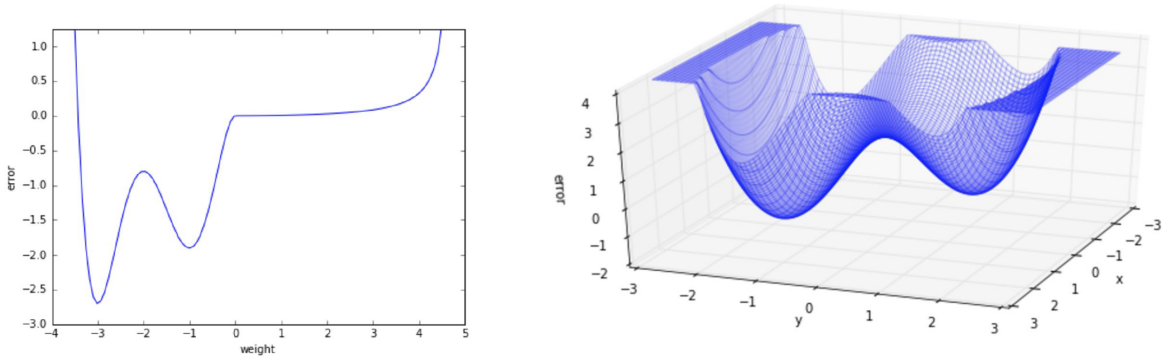


Figure 10.11: (Left) an error surface for one weight, with two local minima and a plateau. (Right) An error surface for two weights, with two local minima.

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.

[16] 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.

Chapter 11

Supervised Learning in Feed Forward Networks

Jeff Yoshimi

Having introduced supervised learning in chapter 10, 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". Second, backpropagation, which adds a powerful method of learning *internal representations* in a neural network. Third, we briefly discuss deep learning networks, which take backprop and apply it to networks with many layers off processing units. Finally, we show how even though these networks were mainly developed in engineering contexts, they have been used in connectionism and computational cognitive science to study the kinds of representations that humans develop to do things like recognize faces.

11.1 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. We have seen that repeated application of the Hebb rule tends to force weights to go to their maximum or minimum values. LMS is more stable: as we will see, repeated application of LMS shifts weights and biases incrementally up and down until they settle in on just the right values.[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 10.2: begin with random parameters (weights and biases), iterate through each of the input patterns, compute outputs (this is sometimes called a "forward pass"), and compare these outputs with desired or target outputs. That is, compute a set of row errors and then an overall error. Then, the weights and biases are adjusted in a way that reduces overall error, using gradient descent. Weights and biases are changed in such a way that overall error decreases. The next time the network sees the same input vectors again, it should produce outputs on each node that are closer to the desired values.

[1]This rule is also known as the "Widrow-Hoff" rule, or the "Delta Rule." It is a descendent of Rosenblatt's perceptron learning rule ([80]; also see chapter 2). Rosenblatt's rule computed error using the output of a node with a threshold activation function on classification tasks. Since both targets and outputs were either 0 or 1, this led to weight and bias updates that were "jittery" and sometimes unstable. Widrow and Hoff made a slight change to the rule, computing the error based on how far the target values were from the weighted input, which varied continuously. This made learning smoother and more reliable. It also allowed the rule to be used on a broader class of problems, including classification and regression problems. Widrow and Hoff built a device that used the rule called an "Adaline". The Delta rule can be shown to be equivalent to ordinary least squares regression, which is why the phrase "least mean squares" is used. Terminology in this area is not entirely consistent. For example, any network trained by LMS is often called a "Perceptron".

Intuitively, the algorithm works like this (for positive valued inputs). If the output is too high, the error is negative. For example, if I wanted 4 but got 5, row error is $4 - 5 = -1$. I have to make the weights and bias a little bit lower to make the output lower. On the other hand, if the output is too low, the error is positive. For example, if I wanted 5 but got 4, error is $5 - 4 = 1$. I have to make the weights and bias a little larger. (Notice that multiplying the weights by the row error would send us in the right direction, and that larger errors will lead to larger weight changes). If the output is just right, don't change anything. By incrementally changing the weights and biases in this way, the network slowly learns to produce the correct response to all inputs. Think of this as a kind of **Goldilocks principle**: just as someone might keep heating and cooling porridge until it is just right, we increase and decrease parameters until overall error is just right.[2]

11.1.1 The Algorithm

Now we can formally define the LMS rule, and verify that it causes the weights to change a such a way that the sum of squared errors (SSE) gets lower over time.[3]

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 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 row 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. This should make some intuitive sense. As we saw above in the goldilocks discussion, for positive inputs weights should be changed in the direction of row error. 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.

LMS also 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]

Applying the rule involves two stages at each iteration. First, a "forward pass", where we compute the output of the network for a given input pattern. Then a "backward pass" or "weight update" pass where we compute the error for each output node (how close was the node's output to the target value?) and then use this error to change its fan-in weights and its bias. Then we repeat the process for every output node on every input pattern, until we have worked our way through the whole training set.

11.1.2 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 10.1), 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 7), we use the prime symbol $'$ to indicate a variable after a delta term has been applied.

[2] A useful picture showing how this method leads to a decision boundary that correctly classifies all training examples is here: `https://commons.wikimedia.org/wiki/File:Perceptron_example.svg`.

[3] 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`.

[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).

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 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. 11.1.

(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. 11.1.

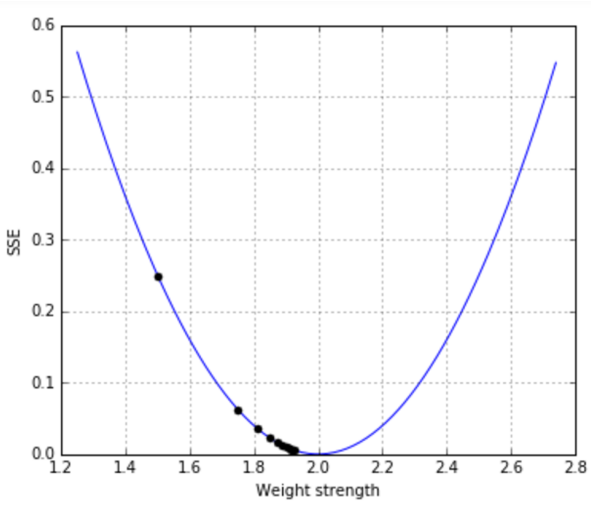


Figure 11.1: Gradient descent on the error surface for the LMS example discussed in Section. 11.1.2. As the LMS rule is applied the weight strength changes in a way that minimizes sum squared error.

(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. 11.1. 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 11.1.

11.1.3 Practice Questions

1. Given $a_1 = 2, w_{1,2} = 2, t = 3, \epsilon = .25$, what are a_2, SSE, $w'_{1,2}$, a'_2, and SSE′?
Answer:
(1) $a_2 = a_1 \cdot w_{1,2} = 2 \cdot 2 = 4$
(2) SSE $= (t - a_2)^2 = (3 - 4)^2 = (-1)^2 = 1$
(3) $w'_{1,2} = w_{1,2} + \Delta w_{1,2} = 2 + \epsilon a_1(t_2 - a_2) = 2 + (.25 \cdot 2 \cdot (3 - 4)) = 2 + (-.5) = 1.5$
(4) $a'_2 = a_1 \cdot w'_{1,2} = 2 \cdot 1.5 = 3$
(5) SSE′ $= (3 - 3)^2 = 0$

2. Given $a_1 = 2, w_{1,2} = 3, t = 5, \epsilon = .1$, what are a_2, SSE, $w'_{1,2}$, a'_2, and SSE′?
Answer:
(1) $a_2 = 2 \cdot 3 = 6$
(2) SSE $= (5 - 6)^2 = (-1)^2 = 1$
(3) $w'_{1,2} = 3 + (.1 \cdot 2 \cdot (5 - 6)) = 3 + (-.2) = 2.8$
(4) $a'_2 = 2 \cdot 2.8 = 5.6$
(5) SSE′ $= (5 - 5.6)^2 = .36$

3. Given $a_1 = -1, w_{1,2} = 1, t = -2, \epsilon = .25$, what are a_2, SSE, $w'_{1,2}$, a'_2, and SSE′?
Answer:
(1) $a_2 = -1$
(2) SSE $= 1$
(3) $w'_{1,2} = 1 + (.25 \cdot -1 \cdot (-2 - (-1))) = 1 + .25 = 1.25$
(4) $a'_2 = -1.25$
(5) SSE′ $= (-2 - (-1.25))^2 = .56$

4. Given $a_1 = 3, w_{1,2} = -.5, t = .8, \epsilon = .1$, what are a_2, SSE, $w'_{1,2}$, a'_2, and SSE′?
Answer:
(1) $a_2 = -1.5$
(2) SSE $= 5.29$
(3) $w'_{1,2} = .19$
(4) $a'_2 = .57$
(5) SSE′ $= .053$

11.2 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 10.4 (Right), but we can also directly label the points as 0 and 1, as in figure 11.2. 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 11.2 (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 11.2 (Middle). Such a line is, as we saw in section 10.4, 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 11.2 (Right) there is no way to use a line to separate the 0's and 1's perfectly.

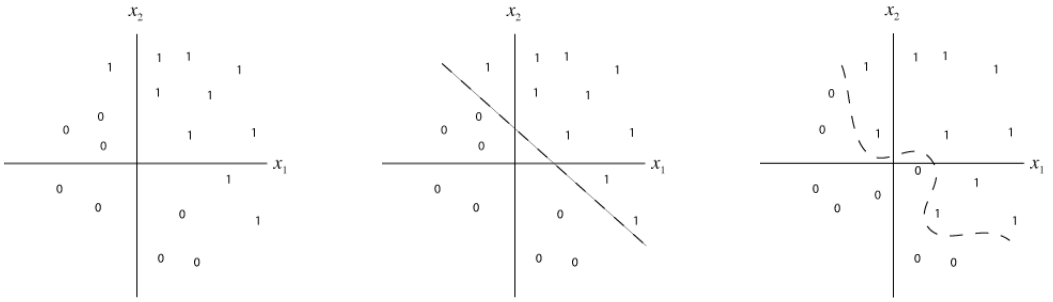


Figure 11.2: Three classification tasks. (Left) A linearly separable task. (Middle) A decision boundary that will solve the task. (Right) A linearly inseparable task and a non-linear decision boundary that can solve it.

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 11.2 (Middle) shows a linearly separable problem. When we cannot properly separate the categories with a line (or hyperplane), as in figure 11.2 (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).[5]

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.[6] 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 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. 11.3 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.

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

[5]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

[6]Doing this amounts to minimizing error. When the decision boundary properly separates the two classes, SSE will be 0. If not, as in Fig. 11.2 (Right), there will be some error. What will SEE be in that case?

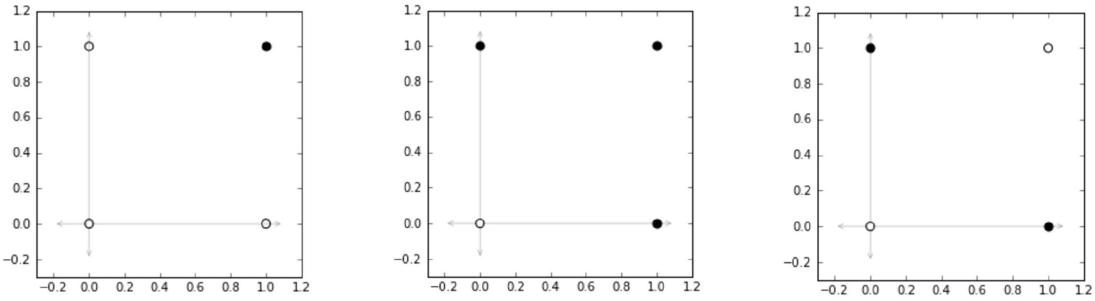


Figure 11.3: Input spaces for AND (left), OR (middle), and XOR (right). Open dots correspond to 0, filled dots to 1. Which tasks are linearly separable?

published a book called *Perceptrons* (perceptrons are a kind of 2-layer network trained by a variant of the LMS technique). In this book Minsky and Papert showed that such networks could not solve linearly inseparable problems [63]. This had a disastrous impact on neural network research in the following decade, during the "dark ages" of neural networks (see section 2.5). As Rumelhart and McClelland recall:

> Minsky and Papert's analysis of the limitations of the one-layer perceptron[7], 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) [82].

However, as Rumelhart and McClelland go on to point out, these results don't apply to neural networks with more than 2 layers [82]. 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" [41].[8]

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) [82]. In particular, Minsky and Papert pointed out that no one knew how such a network could be trained to solve specific pattern association tasks [63].

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.

11.3 Backprop

In this section we cover what was (at least until recently) 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 with linear activation functions. Backprop works for a much broader class of networks, in particular networks with non-linear (in particular, sigmoidal) activation functions and 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, and also psychologically interesting.

[7] 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

[8] 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`.

Backprop can be thought of as a generalization of the LMS technique or "Delta rule" described in Section 11.1. In fact backprop is sometimes called the "generalized delta rule." This rule had been proposed as early as the late 1960s / early 1970s [11, 94] and was independently discovered by several theorists in the 1980s [51, 73]. It was popularized by Rumelhart, McClelland, and Williams in the late 1980s [82]. 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 2).

11.3.1 An Informal Account of the Algorithm

We will not cover the details of the backpropagation algorithm in this chapter but will instead describe it in a qualitative way. As with LMS, the backprop algorithm conforms with the general template described in section 10.2. Roughly:

1. Initialize the parameters of the network (which now includes hidden unit weights and biases) to random values.
2. Present an input vector from the training set.
3. Update the network.
4. Compute the error at each output node for this input vector (by comparing the actual output with the target output).
5. Use this error to update the weights and biases of the network, in a way that reduces the error function (e.g. SSE).

The big innovation with backprop was figuring out how to do the last step on the hidden layer weights and biases, though we will not cover the details here.[9]

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 10.11). 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.[10] 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.[11]

11.3.2 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 11.3, right). Here is the labeled dataset we would use to train a network to implement XOR:

[9] Roughly speaking the output errors are "back propagated" to the hidden unit weights and biases. It is determined to what extent each hidden unit contributes to a given output nodes' error, "blame is assigned," and on this basis the weights to the hidden layer are updated. This is where the term "backpropagation" comes from.

[10] Compare the way we searched for fixed points in chapter 8, by starting at different random points in state space.

[11] Currently the industry standard seems to be the "Adam" method: `https://arxiv.org/pdf/1412.6980.pdf`.

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.

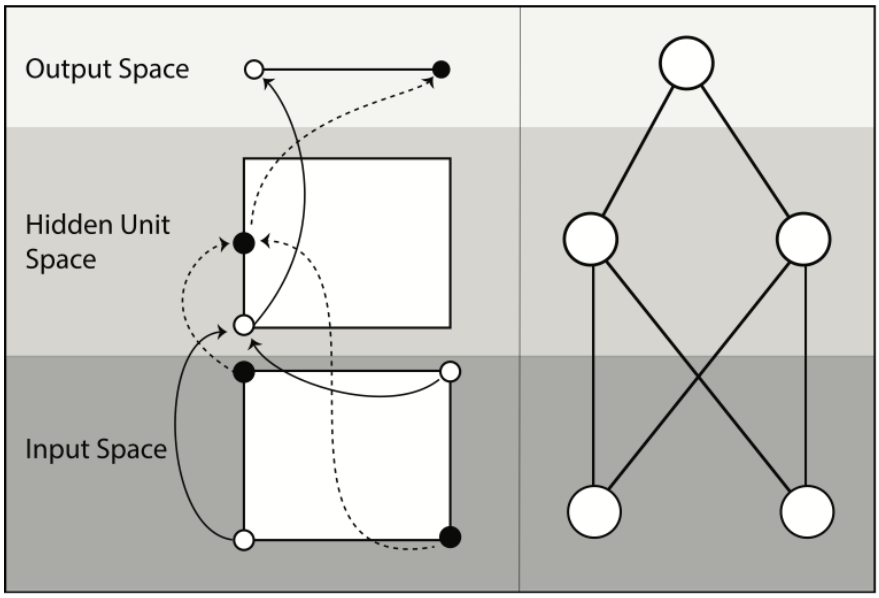


Figure 11.4: A remapping of the input space to the hidden unit space in the XOR problem. Note that the bottom panel shows the input space for XOR, and that it is linearly inseparable. The network then maps $(0, 0)$ and $(1, 1)$, to $(0, 0)$ in the hidden unit space. $(0, 1)$ and $(1, 0)$ are mapped to $(0, .5)$ 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.

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 11.4. 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.

11.4 Internal Representations and Psychological Applications

We have seen that multilayer feed-forward networks trained by supervised learning methods (*e.g.* backprop) will re-map the input space to reduce overall error on a learning task, for example by mapping linearly inseparable inputs to a separable set of points in the hidden units space. Even though these supervised learning

methods are not generally taken to be neurally realistic[12], they are relevant to psychology and cognitive science, since neural networks trained using methods like backprop often develop internal representations that are similar to representations humans use.[13]

The idea that feed-forward networks trained by supervised learning methods learn to remap an input space and develop psychologically realistic internal representations was the basis of much of the connectionists' work showing that neural networks could shed light on human perception and cognition. In this section, we focus on a few connectionist ideas that illustrate much of the work that came out of the first big wave of connectionism (the resurgence of interest in the 1980s and 1990s associated with the PDP group at UCSD).

An example is Gary Cottrell's emotion recognition model, EMPATH, shown in figure 11.5.[14] This network was trained using multiple images of multiple people, each of whom made different facial expressions. The network had several hidden layers, including a layer of edge detectors (outputs of Gabor filters), and a layer that used principle components analysis or PCA, which is similar to Oja's rule (see chapter 7) to do a dimensionality reduction of the previous layer, pulling out the most significant features of that layer. The second-to-last node layer of the network was called a "Gestalt layer". It produced responses similar to face-detectors in the ventral stream of the human visual system (see chapter 3). The output layer has a one-hot encoding of six basic emotions [16].

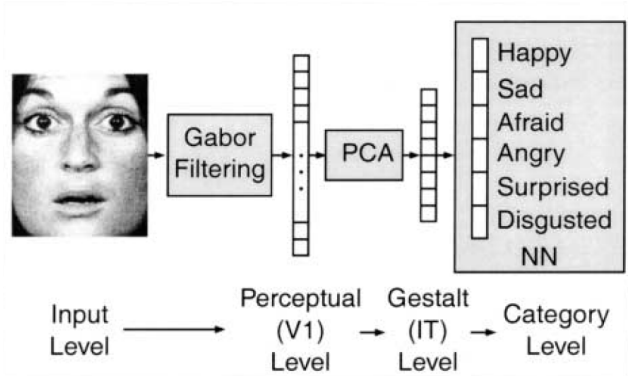


Figure 11.5: Structure of Cottrell's EMPATH emotion recognition network. On the left is a grid of nodes that respond directly to an image. This layer contains over 40,000 nodes. The image inputs are then processed using a layer of weights ("Gabor filtering") that produce responses in the "perceptual level" nodes similar to responses of edge detector neurons in the visual cortex. Another layer of weights performs PCA to reduce the perceptual level to a "gestalt" layer of 50 nodes, that produces responses similar to responses in the ventral steam of the human visual system. The final layer of 6 nodes does the actual classification into different emotion categories, using least mean squares.

Cottrell focused particularly on the second-to-last "Gestalt" layer of the network, which developed nodes that are responsive to particular faces. Cottrell has said that the face recognition nodes in his network act a lot like face recognition neurons in the brain.[15] Like real face-recognition neurons in the ventral stream of cortex (more specifically in area IT), these neurons respond to a picture of someone even if their eyes were occluded, and respond less strongly the more a face image is rotated. The network produced these representations of people *even though it was never told about individuals.* It was only trained to recognize emotions. It did this entirely on its own, as an artifact of the training process [16].

One thing you can do with this kind of network is visualize the internal representations it develops. If you focus on one of the face recognition nodes, and then find what visual input it is most tuned to, a "ghostly

[12]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 3). 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.

[13]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.

[14]See `http://authors.library.caltech.edu/6983/1/DAIjcn02.pdf` [16].

[15]`http://tdlc.ucsd.edu/events/boot_camp_2015/Cottrell_Backprop-representations.pdf`.

looking face" can be observed (see figure 11.6). This image is a kind of visualization of the network's internal template or representation for a particular person [16]. It might be similar to what gets activated in dreams and imagination.[16]

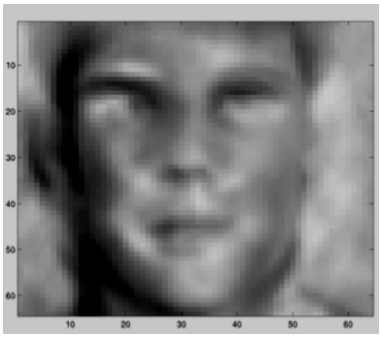


Figure 11.6: Optimal input pattern for a hidden ("Gestalt") layer face node. Shows what kinds of inputs the unit will respond to.

Another case where a feed-forward network trained using backprop finds psychologically interesting internal representations is NETtalk, a model of reading English words aloud, created by Terrence Sejnowski and Charles Rosenberg in 1987 [86]. It is one of the most famous early examples of a connectionist network, that is, a neural network used to shed light on human psychology, in this case the structure of language processing.

The network is a three layer feed-forward network with the structure shown in figure 11.7. The network was trained to speak aloud. It was presented with written letters in English and was trained to pronounce those letters. 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.[17] The output layer contains 26 units encoding different features of phonemes (components of spoken letters), including voicing and vowel height. The hidden layer has 80 hidden units [86].

There are about 18,000 weights and biases in the network, so that the error surface is 18,000 dimensional! Backprop was used to train these weights and biases to solve the problem. The network was trained on a corpus of 1000 common words. It took several days to get the error to a reasonable level using their circa 1987 computer [86]. 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 [86]. So it did quite well.

There are two things that made this example striking. First, the network again developed psychologically plausible internal representations, which it was not programmed to do. For example, the hidden units learned to produce a complete separation of consonants and vowels. Vowels produced hidden unit vectors in one region of hidden unit space, while consonants produced hidden unit vectors in another region of hidden unit space. 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 [86]. 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.

Second, the sounds the networks produced could be artificially synthesized and played back. This was done at several stages in training. Initially the network produced a babble of sounds, like a baby. But as it came to learn the task is got better and better, until it produced somewhat fluent speech [86]. 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: `https:`

[16]For a more recent example that takes this idea much further, see `https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html` [66]. Also see `https://distill.pub/2017/feature-visualization/`.

[17]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). The letters surrounding a letter in a word (3 before and 3 after) are coded to provide context which can be used to correctly pronounce a letter. Thus there were $29 \times 7 = 203$ input units. This kind of moving window is a way of approximating temporal processes in a static feed-forward network.

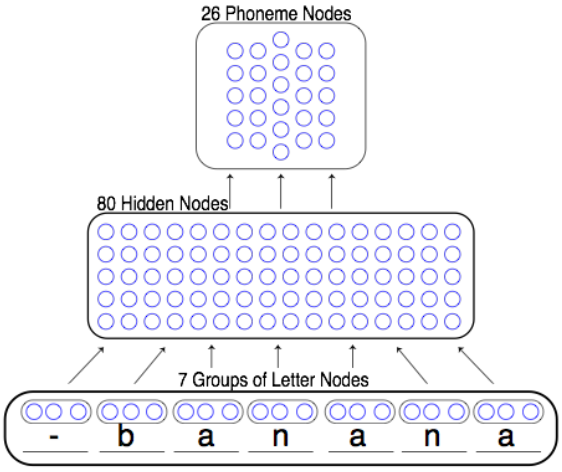


Figure 11.7: NETtalk models converting written words to sounds, *i.e.* reading aloud. Each letter group has 27 nodes (not the 3 shown). When trained using backprop, it developed internal representations of vowels and consonants, without having been told about them.

`//www.youtube.com/watch?v=gakJlr3GecE`. 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.[18]

[18]There are many other studies of learned representations and their psychological realism, in many domains of cognitive science and neuroscience. A book length study of semantic cognition, for example, is [79].

Chapter 12

Deep Networks

Jeff Yoshimi

The idea of re-mapping an input space to a hidden unit space in order to solve classification and regression problems is what drew neural networks out of its first "dark age" and back into the limelight, giving rise to an explosion of interest in engineering and connectionism in the 1980s and 1990s (chapter 2). The internal representations of these networks–usually 3 layer networks trained by backprop–allowed them to solve previously unsolvable problems like XOR (figure 11.4). These representations were often psychologically realistic (section 11.4). However, recall that a second winter lay in wait, as machine learning models took over in the late 1990s and 2000s. What got us out of that second winter was **deep network**s, that is, networks with more than 3 node layers, 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 4. These advances made it possible to use the same types network discussed in chapter 11 on much more difficult problems.[1] They also developed psychological and neurally realistic internal representations, and thus these networks are of considerable interest across the different domains of neural network research.

In terms of engineering, these many-layered networks have been associated with huge improvements in image recognition, speech recognition, language translation, and in many other areas [52, 29]. They do this by creating hierarchies of representations, corresponding to increasingly complex features of an input image. As we saw in chapter 3 (see figure 3.6) when such networks are trained to recognize images they develop internal representation 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 deep networks and deep learning are quite involved and the field is active and continues to grow (this is a preliminary chapter on the topic; it will be expanded in the future). Here we will describe some of the main concepts and some of their applications to neuroscience and psychology.

12.1 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.[2] Until now we've been dealing with weight layers that connect all the nodes in a source layer to all the nodes in a weight layer; these are sometimes called "dense layers" or "fully connected" layers to contrast them with convolutional layers.[3] 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. Convolutions are so important to deep networks that the term "convolutional neural network" or "CNN" is sometimes used (on analogy with ANN for artificial neural network and RNN for recursive neural network, etc.)

[1] 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."

[2] An outstanding visual discussion of the concept of a convolutions is at `https://youtu.be/KuXjwB4LzSA`

[3] There are other types as weight layer as well, for example sparse layers.

To understand how a convolutional layer works we can begin with the concept of a **filter** or **kernel**, which is the set of weights that is scanned over the source layer. The source layer of a convolutional layer is often itself a 2d array, prototypically an input image or a transformation of such an image, and so we can think of the source layer activations as pixels and the source layer as a pixel array (even when the input is not an image this language is useful). Filters are like pattern matchers that we slide across the pixel array, which "light up" most when they are on top of a similar patch. The filter is generally moved from left to right and top to bottom of the pixel array. At each stage of the scanning operation, it is multiplied by the patch of the pixel array it is on top of.[4] The multiplication is a dot product (see chapter 5), where each weight in the filter is multiplied by the corresponding activation of the pixel array. 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.[5]

The idea is illustrated in figures 12.1 and 12.2. A 3×3 filter is passed over a source 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). The code used to generate figure 12.2 is available online, and can be used to play with these layers to get a feel for how they work.[6]

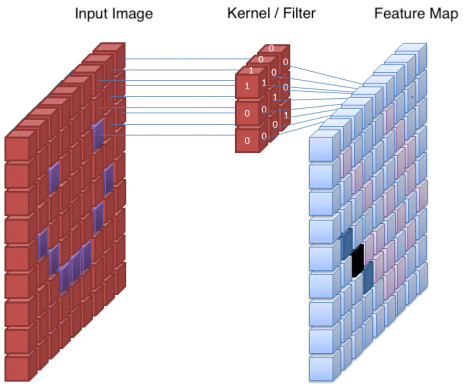


Figure 12.1: From left to right: an input image, a 3x3 convolutional filter (which detects edges with a $-45°$ angle), and the resulting feature map. The filter is scanned across the image. At each stage of this scanning process, the dot product of the filtter's receptive field in the input image is 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.

In figure 12.1 the general idea is evident, but in figure 12.2 all the numbers are included so you can see how the computations are done. Since the input image and the filter are both binary, the dot product simply counts how many places the filter overlaps the image as it is scanned. 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 or target layer 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

[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`.

[5] The number of pixels the filter moves at each step is called the *stride*. One issue that comes up is the edges, which the filter can't be passed over. To handle this *padding* can be added, in the form of extra zeros around the edges of the pixel array.

[6] See: `https://colab.research.google.com/drive/13kqHY0xs8VpLU6Sv7PYtgD6Ro7KdiqjY`.

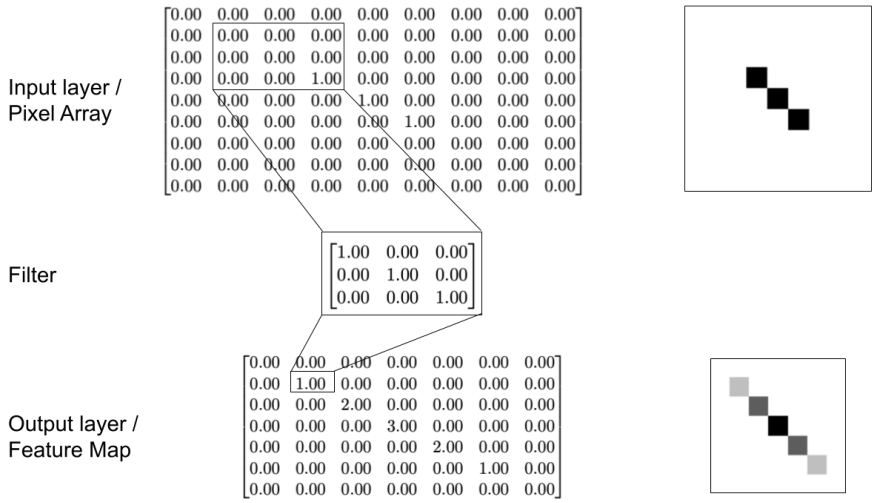


Figure 12.2: Worked example of a convolutional layer. Notice that the feature map has the highest values where the filter matches the pixel array. In the case shown, the filter matches the pixel array in one pixel only. Imagine the filter sliding over the pixel array, and computing dot products (here: numbers of matching 1's), which are used to populate the feature map.

resulting feature map, notice that the activation is highest when the filter is directly on top of such an edge. In figure 12.2 the output layer / feature map activations simply correspond to how many pixels the filter and image overlap on: as you can see, the values are 1, 2, or 3, and it is higher, the more edge like a given patch is. Thus, the feature map shows where this kind of edge occurs in the input image. It itself a new node layer, a new pixel array, a "convolution" of the previous one, that emphasizes the edges in the input image.

Again, note that this is totally different from the weights 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.

Also note that this edge detector is not programmed in. 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 10.7). There is a performance advantage to these convolutional layers. All that must be trained is (in the example shown in figure 12.1) $3 \times 3 = 9$ weights, rather than the $100 \times 100 = 10,000$ 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.

12.2 Feature Maps and Tensors

The magic really starts to happen when we train a *set of filters*, each of which produces a separate feature map. Figure 12.3 illustrates the idea. Note how several of the node layers say "Feature maps" plural, or "F.maps." Each feature map in one of these sets is a response to a different filter, for example, edges of different orientation. This is exactly 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.

A few things to note here.

First, a set of feature maps can be thought of as corresponding to a new kind of node layer. It is not just a single set of nodes, but rather is itself a whole array of matrices. It's like a stack of pancakes, if each pancake is a matrix. An array of matrices is a special kind of **tensor** (a generalization of the vectors and matrices we discussed in chapter 5 to more complex numerical structures), and so computations in deep

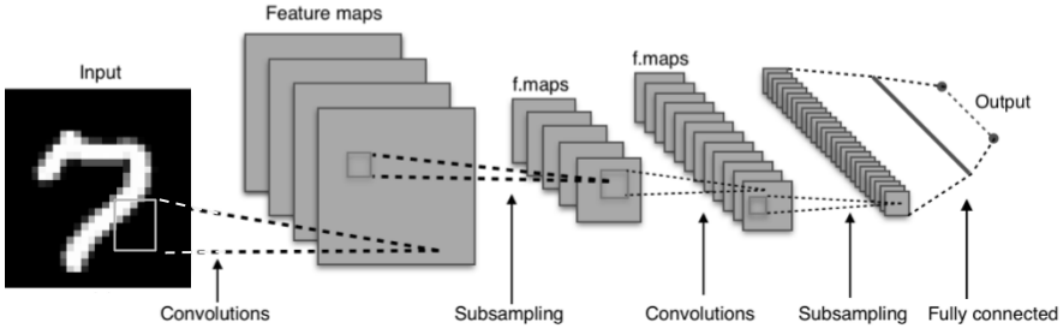


Figure 12.3: A deep neural network, trained to recognize images. The convolutional layers scan over the inputs they are linked to.

networks often involve the use of tensor mathematics.[7]

Second, these networks developing meaningful representations with training; the representations are not programmed in. This is kind of remarkable to ponder. 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. In the case shown in the figure, the training data 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 an old connectionist theme. In Nettalk (section 11.4), phonetic categories like consonant and vowel were not programmed in, but emerged with training. With simple recurrent networks (section 13.7), grammatical categories like verb and noun were not programmed in but emerged with training.

12.3 The many layers of a deep network

In a full deep network many convolutional layers and regular "dense" weight layers are combined, sometimes as many as 100 or more! 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.

As can be seen in figure 12.3, there are other kinds of layers besides convolutional layers. Subsampling refers to methods where the size of a representation is reduced, often by scanning over the source layer, and then averaging or finding the largest value in each window (*average pooling*, *max pooling*, etc.). This is also called downsampling.

The final layers of a deep network are often more conventional fully-connected dense layers. The 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 basically familiar backprop networks presented with the results of a whole lot of convolving and subsampling.

[7]Technically numbers and vectors and matrices are also tensors. The rank of a tensor is the number of indices it takes to specify an "entry" in the tensor. A number is rank 0 because it requires no indices. Recall that a vector is like a list of numbers. 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: one to specify a location in the array, and then a row and column index.

12.4 Applications of Deep Learning

As discussed in section 2.7, deep networks and deep learning led to a revolution in neural networks beginning in the 2010s. The revolution was in engineering initially, but the history of deep networks shows that they have applications across all the domains of neural network research: engineering, computational neuroscience, connectionism, and computational cognitive neuroscience.

Deep network models originate in computational neuroscience models of vision that were developed in the 1970s and 1980s [26]. These ideas were later used to engineer pattern recognition networks. A famous early application was recognizing zip codes written on envelopes [53]. As deep networks became mainstream based on technical improvements (big data, GPU and hardware acceleration, better architectures and training algorithms), scientists began using them, for example, to model the response profile of neurons in the visual system (recall the discussion of figure 3.6 in chapter 3).

The idea is also relevant to connectionism and computational cognitive neuroscience. You may recall from the history chapter that 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 2.8 in chapter 2). Deep networks instantiate this idea, in such a way that we can actually see what their receptive fields are.[8] The significance of these networks for psychology is still in its infancy, but early results are promising [100, 77].

[8]The receptive fields can be quite strange and even disturbing. See `https://distill.pub/2017/feature-visualization/` for some striking demonstrations.

Chapter 13

Supervised Recurrent Networks

Jeff Yoshimi

In chapter 1, we introduced the distinction between feed-forward and recurrent networks, and used that distinction to organize much of this book. 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 11). 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 8). 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 9, 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 8, 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, there has been an explosion of interest in recurrent networks based on the successful adaptation of deep learning methods (chapter 12) 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. As of this writing you can interact with these models using ChatGPT, and no doubt even more powerful models are on the way, which show just how powerful this kind of network can be.

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

[1] Other approaches to recurrent networks (reservoir networks like echo state machines), that are more geared towards computational neuroscience, are discussed in chapter 15.

in reaction to inputs. These sections give us a sense of how supervised learning can be applied to recurrent networks. We then show how these ideas can been pushed to amazing limits, focusing on the "transformer architecture", a kind of deep, many-layered network which uses a concept known as "attention", producing highly complex internal representations. We then show what models like these can do when used to model language, most recently with a massive model of human speech called "GPT-3." Finally, we discuss how these ideas have been used in connectionist contexts, to identify learned internal representations that are psychologically meaningful. These accounts show how (for example) human grammar could be learned rather than innate.

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 6.3). 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."

13.1 Types of Supervised Recurrent Networks

To start, let's think about ways these kinds of network can be useful. Figure 13.1 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.

These ideas have links to dynamical systems theory (chapter 8). 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 [27].[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.

[2] In these examples absent inputs are just zero vectors.

[3] Even more generally a recurrent network can reproduce any Turing Machine, and hence *any computational system.* They are "Turing Complete"; see [36] 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 11

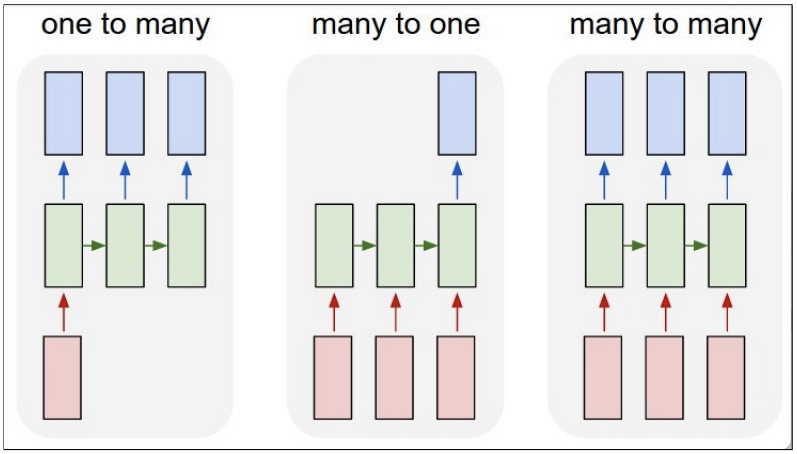


Figure 13.1: Different types of sequence learning possible with backprop through time.

13.2 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 [20], 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. 13.2. 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 10, 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 13.1 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.

[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.

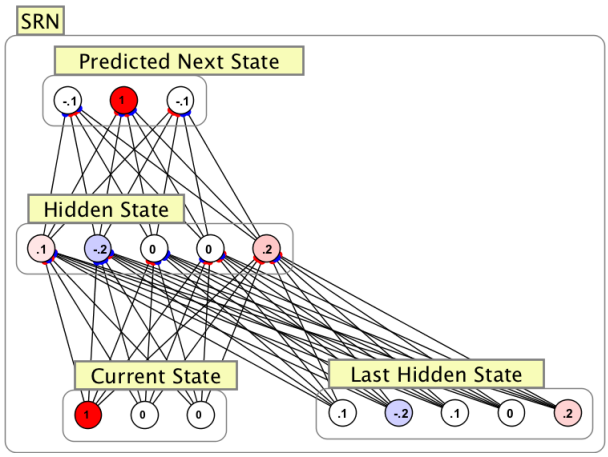


Figure 13.2: A simple recurrent network.

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 13.1: 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.

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 $(0, 1, 0)$ 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 13.2. 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.

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 13.2: 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.

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, or sounds have been converted into vectors (see the discussion of feature encoding in chapter 6). 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.

13.3 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" [95].[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 13.3).

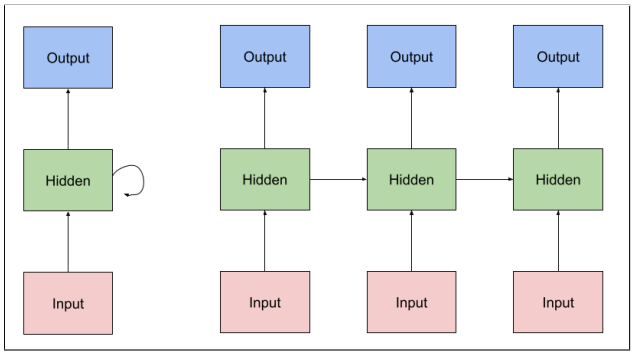


Figure 13.3: Schematic of backprop through time. The actual network that we are training is on the left: a feed-forward network with a recurrent weight layer in the middle. The unrolled network on the right. This network can learn a sequence of three input-target pairs. We train the unrolled network to produced 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

[7] There are other methods of training recurrent networks, eg. real time recurrent learning [99], and "dynamic reconstruction" (Haykin, 14.13) [36].

[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.

network, the one on the left side of figure 13.3. 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 generalize, producing similar sequences of outputs to those in the target set, given a similar sequence of inputs.

13.4 Language Models

Supervised recurrent networks and their successors have become incredibly powerful. As of this writing they are changing the way authorship and education will work, because they can write arguably professional-level papers and artworks. In the domain of language these are called "large language models", because they are based on such large datasets. Let's have a look at how these scarily powerful models emerged.

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 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 [44]. 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*

A neural network wrote that!

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 11.4). 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 [44]. A fragment from the artificially generated mathematics paper is shown in Fig. 13.4. These techniques have even been used to generate a fake script for a movie, which was then actually produced![11]

[9] `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`. Note that Karpathy makes use of LSTMs; see note 17.

[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 [67]

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 13.4: Fragment of "fake math" generated by a recurrent network trained on real math.

Karpathy's blog post was written in 2015. In 2020 the power of recurrent networks reached another milestone with the introduction of Open AI's model, GPT-3 [10, 24].[12] This model used all the latest ideas in recurrent neural networks, with a giant training dataset, which includes a web-scraped archive of the *entire internet*.[13] If you've every written anything online, it may be part of the training data that was used! The model has over 175 billion parameters; that is, 175 billion weights and biases! It is absolutely massive, and scary. It arguably passes the Turing Test, a long-standing test for artificial general intelligence, answering questions and producing convincing text in response to prompts.[14] For example, when asked to write an article with the title "United Methodists Agree to Historic Split: Those who oppose gay marriage will form their own denomination", it produced the following:

> After two days of intense debate, the United Methodist Church has agreed to a historic split - one that is expected to end in the creation of a new denomination, one that will be "theologically and socially conservative," according to The Washington Post. The majority of delegates attending the church's annual General Conference in May voted to strengthen a ban on the ordination of LGBTQ clergy and to write new rules that will "discipline" clergy who officiate at same-sex weddings. But those who opposed these measures have a new plan: They say they will form a separate denomination by 2020, calling their church the Christian Methodist denomination...

Most people can't tell that this was written by a computer. Also note that it refers to actual information from online sources, and that while the other examples from Karpathy feel fake when considered closely, this one does not. GPT-3 is currently being used in hundreds of applications "around the globe [to]... generate an average of 4.5 billion words per day".[15] Some philosophical analysis of the promise and potential dangers of this technology are in [24].

As of this writing it is late 2022, and progress remains rapid, with GPT-4 soon to be released and with more complex network architectures like stable-diffusion and Dall-E produce convincing images from text prompts.[16] So, this chapter should be changing on a regular basis for some time, and new chapters will be added as well to cover these advances. You can use ChatGPT for free online, and you should try it. It's strangely human watching questions and requests be answered in real-time like a really smart human who read the entire internet.

13.5 The Transformer Architecture

Recurrent networks trained using backprop through time (like Karpathy's) are impressive but not perfect. The main theoretical problem they run into is that they favor recent context over earlier context. But

[12] Open AI is a venture launched by Elon Musk and others, and currently funded in part by Microsoft.

[13] Common crawl: `https://en.wikipedia.org/wiki/Common_Crawl`.

[14] `https://plato.stanford.edu/entries/turing-test/`.

[15] From `https://openai.com/blog/gpt-3-apps/` retrieved in November 2021. The site also describes some of these applications.

[16] See `https://en.wikipedia.org/wiki/Text-to-image_model`.

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 11.3), 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 4).[17] 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.

A more powerful approach to training recurrent networks, which *has* had an impact on cognitive science, is the transformer architecture [91].[18] Transformer networks are not typically thought of as recurrent networks, but they build directly on the idea of a recurrent network, and they do have some recurrent features, and so we discuss them in this chapter. 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. Rather than relying on recurrent connections for temporal context, they provide an entire sequence of words all at once. Thus during training the network can learn to find important contextual dependencies in these sequences, including long-range dependencies.

Suppose we want to ask a network "hello how are you?" In a recurrent network, a vector representation (or "vector embedding") of each word in the sequence would be presented separately: "hello", "how", "are", and "you". In a transformer network, by contrast, we arrange these vector embeddings into a single matrix and present them all at once to the network. That is, the input to the network is the whole sentence {"hello", "how", "are", "you"}, in the form of a list of vectors, one for each word. There are a number of tricks used to represent this sentence in a useful way (recall the discussion of feature engineering in chapter 6)[19], but for our purposes it is enough to just think of a whole sentence as an input matrix.

This complex input encompassing an entire set of sentences is then sent through a specialized set of layers. Each layer combines a "self-attention" layer with a traditional linear layer and several other mechanisms. The self-attention layer is where the magic happens. One part of this layer compares each word in the sentence to every other word in the sentence. So "hello" is compared to "hello", "how", "are", and "you", and this comparison is used to create an output representation of the sentence that reflects dependencies between the words. All words are compared to each other, so no matter how far apart they are in a sentence, they can influence each other. The attention layer learns what relations between words in a sentence are important; in a sense it learns what parts of the sentence other parts of the sentence should 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.

Each self-attention layer of a transformer network can have multiple "heads"; it is a "multi-headed" attention mechanism. As a result, the network can learn *multiple* ways to compare words in the sentence to each other, a bit like how a convolutional network (section 12.1) 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.

[17] Two such functions are "long-short-term memories" (LSTMs) [83, 69] 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.

[18] The clearest discussions of transformer networks that I am aware of are this series of videos by Hedu AI, `https://www.youtube.com/watch?v=dichIcUZfOw&t=92s`, and this blog post by Jay Alammar: `https://jalammar.github.io/illustrated-transformer/`. Hedu AI is great to start with; she really brings out the intuitions behind the various components of the architecture, and makes it clear how information flows through the network.

[19] In particular positional encodings, a special form of vector that can be added to a word embedding that tags it for its location in a sequence. Since words are intrinsically tagged with their position, they can be processed in parallel, which allows the architecture to run on fast parallel hardware.

Now we take a lesson from deep networks, and stack many of these transformer layers on top of each other. When many of these attention layers are stacked together, we can get these extremely sophisticated representations. 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 2.3), 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. We return to this topic at the end of the chapter, but note that understanding the nature of these representations is still in its infancy. The transformer model was not built to help cognitive science, after all, but to support NLP engineering tasks. Nonetheless, the results are so compelling that they are of interest to cognitive scientists.

Ok, so we have this super complex structure, that can represent multiple forms of inter-word dependency at multiple layers of a deep net representation. How do we train it? Often using the basic auto-regressive "one step-ahead" prediction method we discussed at the start of the chapter. Just take a document—a big set of sentences—and send most of it as input to a transformer network, and ask it to predict what the very next word or token will be. This simple set up can be applied in a kind of recursive way to allow a network to learn to produce long responses to simple prompts.[20] However, they can also be trained in other ways. For example, we can feed a transformer network the first halves of hundreds of movie scripts and train it to produce the second halves of those scripts, or we can train it to predict what the first summary paragraph of a Wikipedia article is based on the main body of the article. Either way, we use gradient descent (section 10.7). When the network predicts the wrong next word (or the wrong second half of a script, or the wrong first paragraph of a Wikipedia article), all the many weights in all these transformer layers are adjusted, so that the next time it tries to predict the same word or words relative to the same prompt, it does a little better. Repeat this many times, and the network will become amazingly good at "talking" like a human. Of course, it is slow, but the structural features of transformer networks make them work well with modern parallel hardware.

13.6 Bert

Transformer networks have been extremely successful, as we saw in the discussion of GPT-3. They have also revolutionized natural language processing, via the BERT language model, which is making an impact in the cognitive sciences, especially in the computational study of language. In a paper co-authored by Jay McClelland [58] (a member of the original PDP group at UCSD; see section 2.6), 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 _____.

The reader expects the word "gone" next, but if "took the beer" is replaced with "took the ice" the reader expects something like "warm". But the first part of the sentence is too far apart from the last part for an unrolled backprop through time network to pick up the relationship. Transformer networks like BERT overcome this problem.

The big questions are: how exactly does BERT do this, and whatever it does, is it relevant to human language processing? This creates a fascinating twist on our earlier discussion of types of neural network research (section 1.3). Recall that neural networks are sometimes used for engineering, sometimes for science, but that there is a feedback between these categories of research. For example, deep convolutional networks originated in scientific studies of visual processing, got refined and used for pattern recognition tasks in engineering, and the resulting deep networks were so powerful that they were then taken back into science to describe (for example) the human visual system. Additionally, the training task for BERT differs from other contemporary language models; BERT uses "masked language modeling" (MLM) instead of traditional next

[20] This is easiest to see in a video; see `https://youtu.be/gJ9kaJsE78k?t=546`.

word prediction.[21] Instead of modeling language as a left-to-right stream of words, where the model predicts the next word based on the previous context, BERT instead masks a token in the middle of the sentence, and predicts it based on the surrounding sentential context. This is advantageous over unidirectional left-to-right and concatenated left-to-right and right-to-left models, since it creates a truly bidirectional representation where the left and right contexts are joined together.

Here we see a variant on that pattern. In this case, recurrent neural networks (with their own long history spanning cognitive science and engineering) weren't performing well enough, so engineers created these new transformer architectures, mainly to tackle engineering issues like vanishing gradients and parallelization. They used the intuitive concept of attention, but it was all engineering. BERT came out of Google and fit their engineering needs. Psychologists and linguists then realized BERT was doing better at analyzing language than other models in linguistics, so they started to treat it as an object of scientific interest in its own right. This gave rise to a new field called "BERTology". In a paper titled "A Primer in BERTology: What We Know About How BERT Works" [78], a "survey of over 150 studies of the popular BERT model", the authors explain:

> Although it is clear that BERT works remarkably well, it is less clear why, which limits further hypothesis-driven improvement of the architecture. Unlike CNNs, the Transformers have little cognitive motivation, and the size of these models limits our ability to experiment with pre-training and perform ablation studies. This explains a large number of studies over the past year that attempted to understand the reasons behind BERT's performance. In this paper, we provide an overview of what has been learned to date, highlighting the questions that are still unresolved.

This is a strange situation. Engineers built something and then scientists created a science to understand it!

Clearly there is a need for further study from connectionist and computational neuroscience standpoints (BERTologists are mainly computational linguists). These networks develop complex internal representations using neural networks, which are brain like, and perform at amazingly human levels, as we'll now see.

13.7 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 10 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 13.2 to predict the next words in a sentence.[22] 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. 13.5. 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 than others to follow one another, and so the predictions match these patterns. It also learns some general rules, like expecting verbs after nouns [20].[23]

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).

[21] See [17] for a detailed explanation.

[22] `http://psych.colorado.edu/~kimlab/Elman1990.pdf`.

[23] Simbrain simulations that partially replicate the results of these papers can be accessed in the script menu as `elmanPhonemes.bsh` and `elmanSentences.bsh`.

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 13.5: Some of the sentences used to train the word prediction network.

Figure 13.6 shows a schematic reconstruction of the structure of the SRN's hidden unit space.[24] 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 [20]. 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 [45].

This is by no means a settled area. As noted above, linguistic processing using transformers 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 [58], 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.

[24] 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.

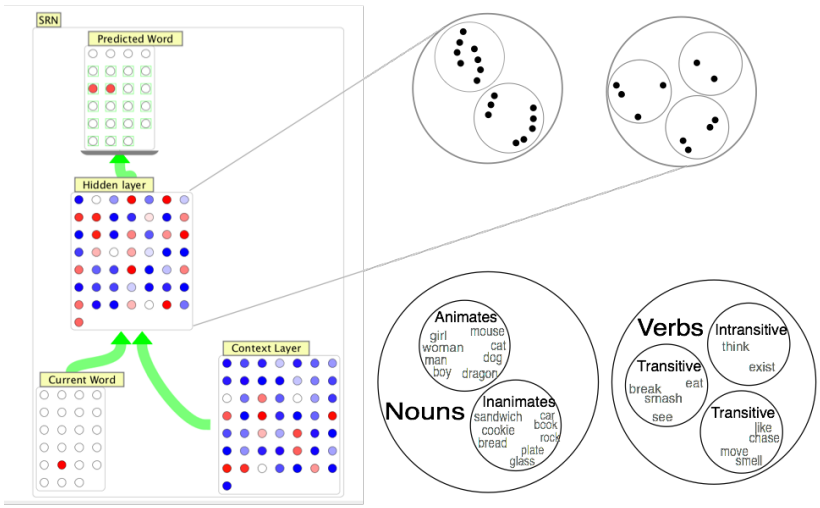


Figure 13.6: The SRN used in the word prediction task. The output layer predicts the next word (coded as a binary vector) in a sentence. It is currently undecided between which of two words might come next. The right top panel shows (schematically, this is not a projection of actual data) what points in the hidden unit space corresponding to different words. The right bottom panel shows how these points cluster in to grammatical and semantic categories.

Chapter 14

Spiking Models: Neurons & Synapses

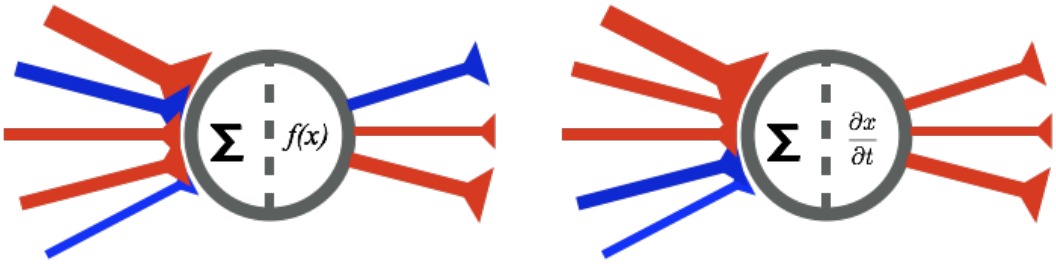
Zoë Tosi, Jeff Yoshimi

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 14.2). 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) [71]. 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 14.1: 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.

14.1 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:

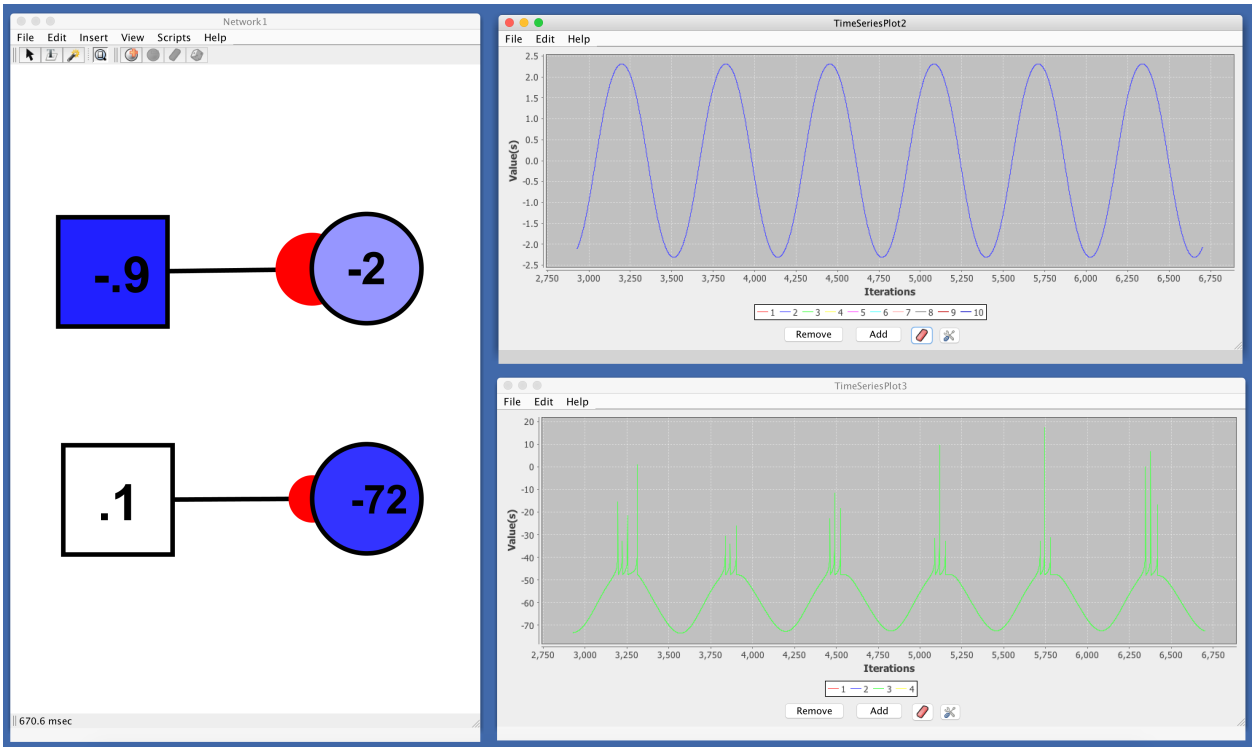


Figure 14.2: A Simbrain desktop with two cells receiving inputs from activity generators. Both are providing the same sinusoidal input. Above is a typical connectionist neuron using a logistic activation function (bounded on (-5, 5)) , and below is a spiking neuron model. Across from each is a time series 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 3.1). 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 3.1.3). 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. [55]. 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 [50, 62]. 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

14.2 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.

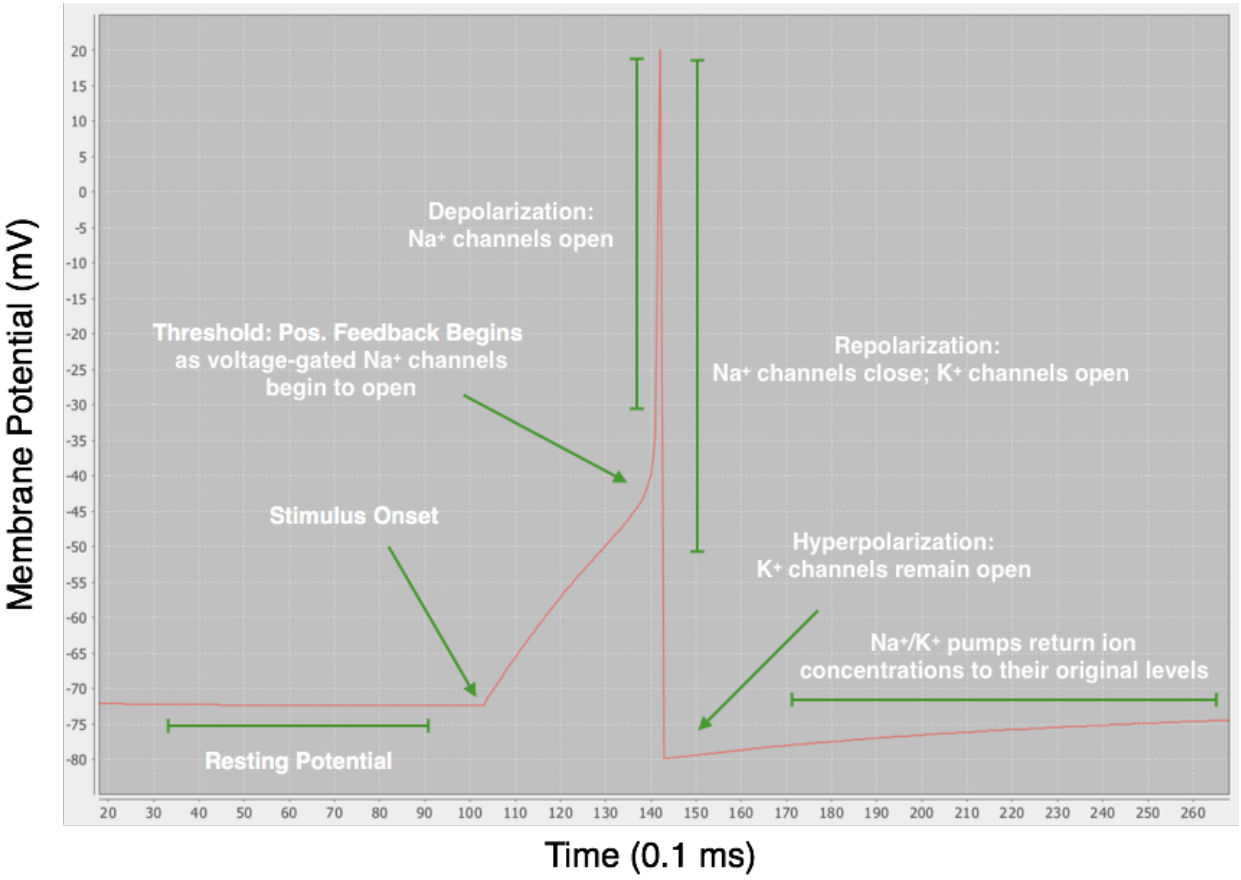


Figure 14.3: A canonical action potential.

14.3 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.

14.3.1 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. 14.1. 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.

$$r_j = \Theta\left(\sum_i r_i \,;\, \theta\right)$$
$$\text{where: } \Theta(x\,;\,\theta) = \begin{cases} 0 & x < \theta \\ 1 & x \geq \theta \end{cases}$$

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.

14.3.2 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.

14.4 Synapses with Spiking Neurons

14.4.1 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}.

14.5 Long-term plasticity

14.5.1 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.

14.5.2 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.

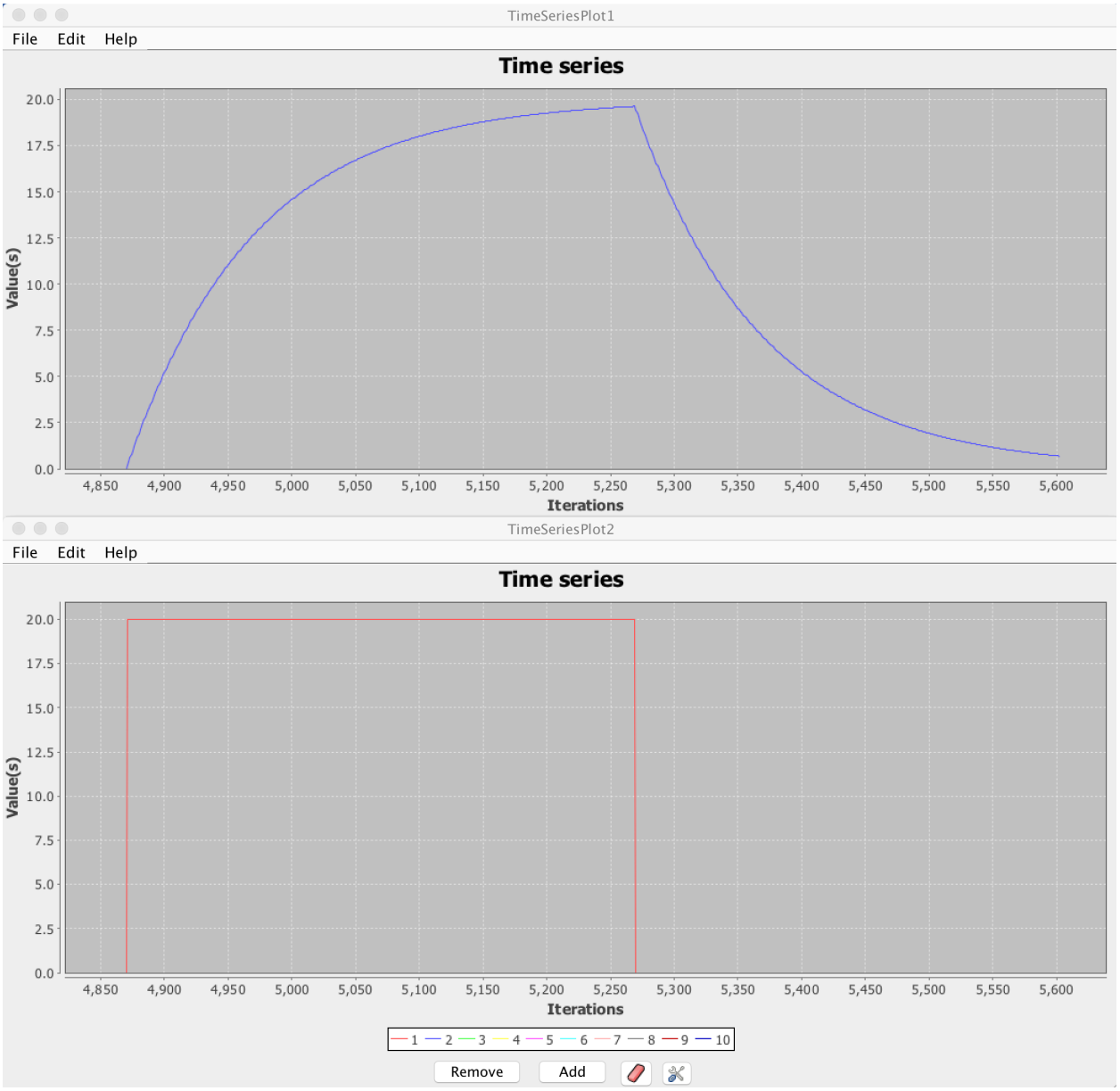


Figure 14.4: The response in the membrane potential of a linear integrate and fire neuron (blue/top) to a constant current injection (red/bottom). Notice the *nonlinear* dynamical response in the LinIF neuron's membrane potential to a constant step in current.

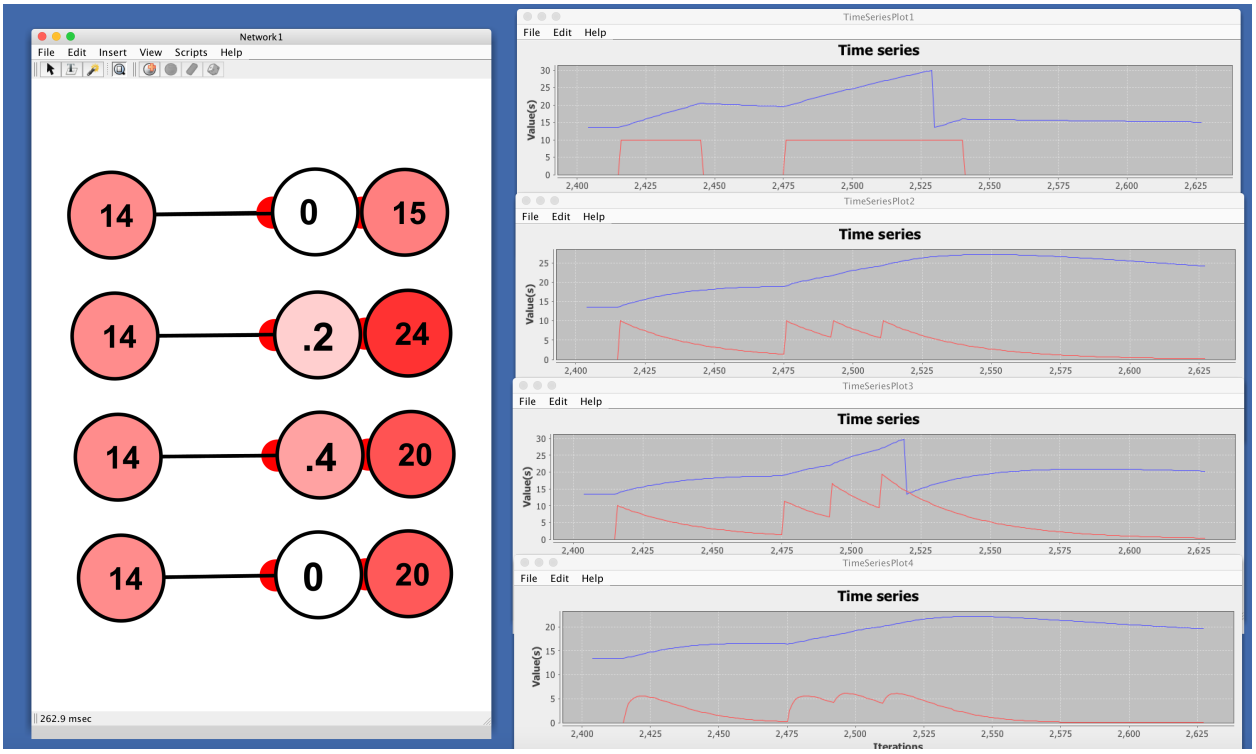


Figure 14.5: The post-synaptic responses for 4 different spike response mechanisms (red) plotted alongside the resulting change in post-synaptic membrane potential (blue). From top to bottom, we have a step-function, an instantaneous jump and decay with no convolution, a convolutional jump and decay, 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.

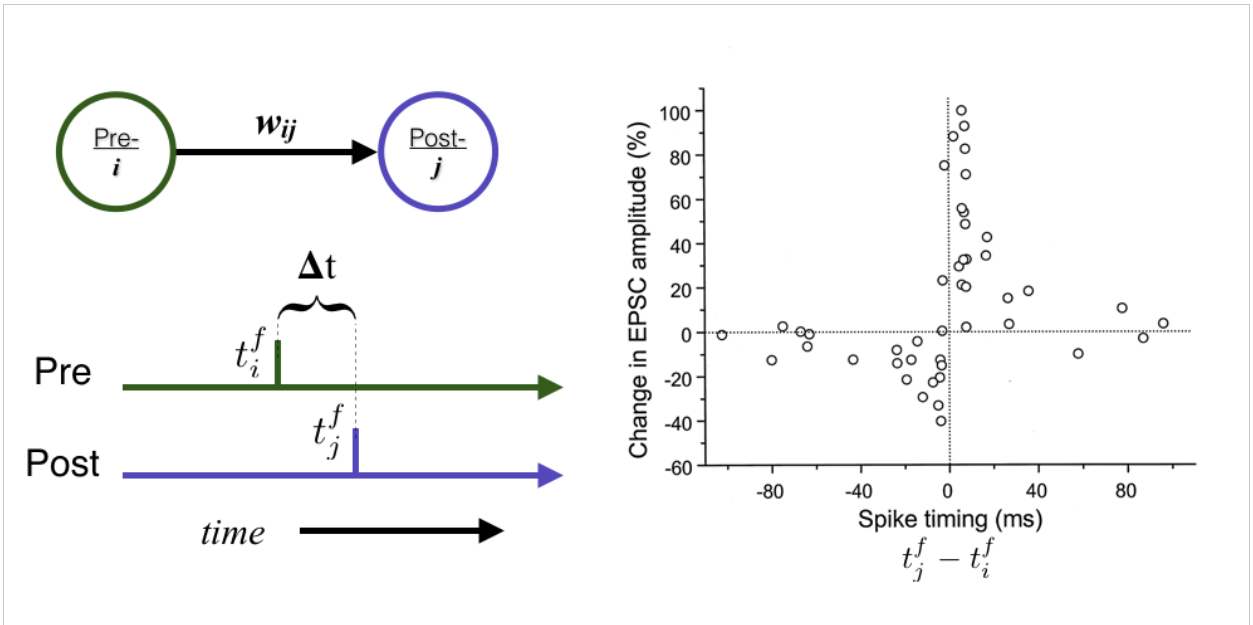


Figure 14.6: A diagram of STDP paired with empirical data showing the change in amplitude of the excitatory post-synaptic current (EPSC) produced by a synapse impinging on a neuron after repeated external stimulation of a pre- and post-synaptic neuron so as to force spikes in the two neurons with specific 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 [5]. 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 15

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 [22] where a literal bucket of water was used for nonlinear pattern separation 15.1.

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.

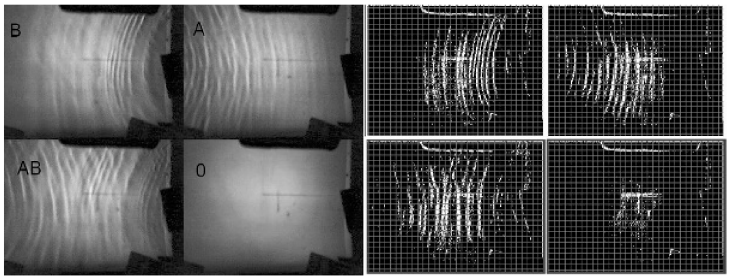


Figure 15.1: Pattern recognition has literally been done in a bucket of water, providing a concrete example of the water metaphor. Ripples on the surface were recorded in response to labelled inputs (perterbations emanating from the left or right side of the bucket, and when combined with a perceptron classifier was able to perform nonlinear pattern separation (XOR). Retrieved from [22].

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 15.2 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.

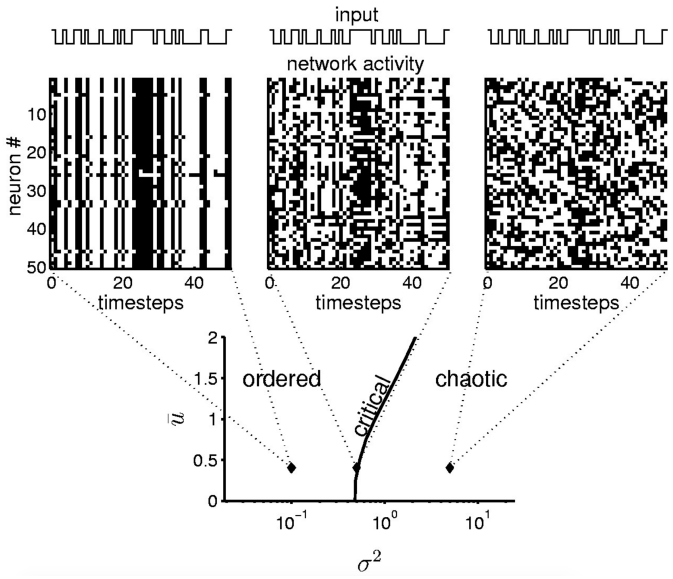


Figure 15.2: Here we have an example of ordered, chaotic and edge-of-chaos dynamical regimes in a recurrent neural network as well as what portions of the network's parameter space in terms of the variance of weights (σ^2) and the mean value of the input signal ($\bar{\mu}$) Notice the similarities in 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 [4] 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 [4].

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. 15.3. 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.

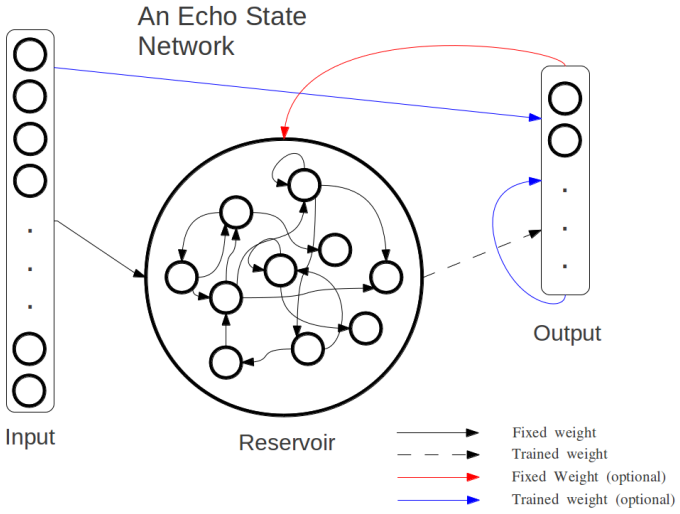


Figure 15.3: An echo state network, which is one kind of reservoir computing network.

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 8. 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 16

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.

Adaptive exponential integrate and fire model

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.

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.

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.

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.

Convolutional layer a special kind of weight layer where a set of weights (a "filter") is passed over a source node layer to produce activations in a target layer, in the sense of being multiplied by each of a moving sequence of subsets of the input activations. 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, 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.

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.

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 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.

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.

Learning In a neural network, a process of updating synaptic weights so that the network improves at producing some desired vector-valued function (for a feed-forward network) or dynamical system (for a recurrent network), or otherwise behaves in some way deemed useful by its designer.

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 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.

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.

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.

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.

Piecewise linear activation function a function that has the value ℓ for input below ℓ, the value u for input above u, and whose value varies linearly for input between ℓ and u.

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 activation function ("relu" is short for "rectified linear unit") 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

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.

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.

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.

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.

Tolerance of noisy inputs the ability of a network to perform well when the inputs presented to it have noise added.

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.

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.

Weight vector a list of values for the weights of a neural network or for one of its layers. A weight vector can be multiplied times an activation vector to produce a weighted sum of activations.

Weight space a vector space made up of all possible weight vectors for a neural network.

Weighted inputs (Synonym: net input) the sum of incoming activations to a node times intermediate weights, plus bias. Intuitively, it is the overall level of input a node is receiving.

Unsupervised learning a learning rule in which weights are adjusted without an explicit representation of desired outputs.

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: $(1, 2) - (3, 4) = (-2, -2)$.

Vectors ordered lists 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 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 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.

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.

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!) [60]. 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.

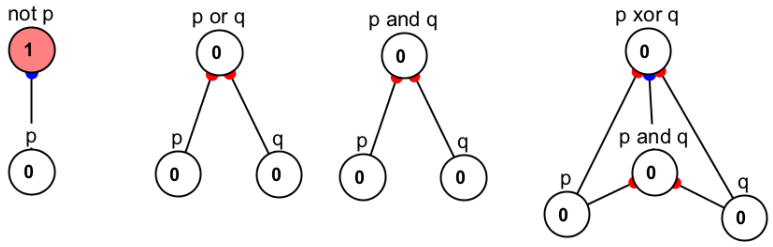


Figure A.1: Networks to implement basic logic gates.

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.

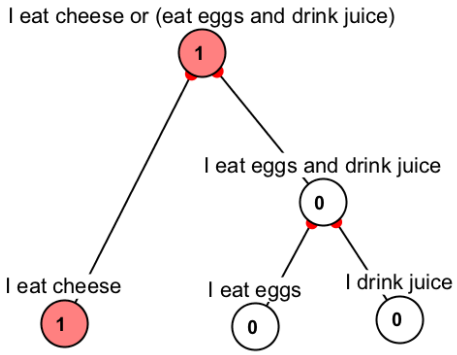


Figure A.2: Network to implement a more complex boolean function.

Figure Attributions

Bibliography

[1] James A Anderson and Edward Rosenfeld. *Talking nets: An oral history of neural networks*. MiT Press, 2000.

[2] Bernard J Baars. *The cognitive revolution in psychology*. The Guilford Press, 1986.

[3] Alexander Bain. *Mind and Body the Theories of Their Relation by Alexander Bain*. Henry S. King & Company, 1873.

[4] Nils Bertschinger and Thomas Natschläger. Real-time computation at the edge of chaos in recurrent neural networks. *Neural computation*, 16(7):1413–1436, 2004.

[5] 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.

[6] Christopher M Bishop. *Neural networks for pattern recognition*. Oxford university press, 1995.

[7] 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.

[8] Edwin Garrigues Boring. *History of experimental psychology*. Genesis Publishing Pvt Ltd, 1929.

[9] 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.

[10] 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.

[11] AE Bryson and YC Ho. Applied optimal control. *Optimization, Estimation and Control*, 1969.

[12] Ed Bullmore and Olaf Sporns. Complex brain networks: graph theoretical analysis of structural and functional systems. *Nature Reviews Neuroscience*, 10(3):186–198, 2009.

[13] Malcolm Burrows. *The neurobiology of an insect brain*. Oxford University Press Oxford, 1996.

[14] 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.

[15] Andy Clark. Whatever next? predictive brains, situated agents, and the future of cognitive science. *Behavioral and brain sciences*, 36(3):181–204, 2013.

[16] Matthew N Dailey, Garrison W Cottrell, Curtis Padgett, and Ralph Adolphs. Empath: A neural network that categorizes facial expressions. *Journal of cognitive neuroscience*, 14(8):1158–1173, 2002.

[17] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[18] Hubert L Dreyfus. *What computers still can't do: a critique of artificial reason*. MIT press, 1992.

[19] 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.

[20] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[21] Laurene V Fausett. *Fundamentals of neural networks*. Prentice-Hall, 1994.

[22] Chrisantha Fernando and Sampsa Sojakka. Pattern recognition in a bucket. In *European Conference on Artificial Life*, pages 588–597. Springer, 2003.

[23] Stanley Finger. *Origins of neuroscience: a history of explorations into brain function*. Oxford University Press, USA, 2001.

[24] Luciano Floridi and Massimo Chiriatti. Gpt-3: Its nature, scope, limits, and consequences. *Minds and Machines*, 30(4):681–694, 2020.

[25] 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.

[26] 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.

[27] Ken-ichi Funahashi and Yuichi Nakamura. Approximation of dynamical systems by continuous time recurrent neural networks. *Neural networks*, 6(6):801–806, 1993.

[28] Michael S Gazzaniga, RB Ivry, and GR Mangun. *Cognitive Neuroscience, New York: W. W.* Norton & Company, 2002.

[29] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[30] David Groome. *An introduction to cognitive psychology: Processes and disorders*. Psychology Press, 2013.

[31] 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.

[32] 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.

[33] 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.

[34] David Hartley. *Observations on Man: His Frame, His Duty and His Expectations...* Leake & Frederick, 1749.

[35] David Hartley. *Observations on man, his frame, his duty, and his expectations*. T. Tegg and son, 1834.

[36] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 2 edition, 1998.

[37] Donald Olding Hebb. *The organization of behavior: A neuropsychological theory*. Psychology Press, 2005.

[38] Jukka Heikkonen and Jouko Lampinen. Building industrial applications with neural networks. In *Proceedings of the European symposium on intelligent techniques*, pages 3–4, 1999.

[39] John A Hertz, Anders S Krogh, and Richard G Palmer. *Introduction to the theory of neural computation*, volume 1. Basic Books, 1991.

[40] 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.

[41] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

[42] Philipp K Janert. *Data Analysis with Open Source Tools: A Hands-on Guide for Programmers and Data Scientists.* " O'Reilly Media, Inc.", 2010.

[43] 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.

[44] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. *Andrej Karpathy blog*, 2015.

[45] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078*, 2015.

[46] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[47] 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.

[48] 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.

[49] 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.

[50] Andreea Lazar, Gordon Pipa, and Jochen Triesch. Sorn: a self-organizing recurrent neural network. *Frontiers in computational neuroscience*, 3:23, 2009.

[51] Yann Le Cun. Learning process in an asymmetric threshold network. In *Disordered systems and biological organization*, pages 233–240. Springer, 1986.

[52] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[53] 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.

[54] Daniel S Levine. *Introduction to neural and cognitive modeling.* Psychology Press, 2000.

[55] Wolfgang Maass. Computing with spikes. *Special Issue on Foundations of Information Processing of TELEMATIK*, 8(1):32–36, 2002.

[56] David Marr and W Thomas Thach. A theory of cerebellar cortex. In *From the Retina to the Neocortex*, pages 11–50. Springer, 1991.

[57] 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.

[58] 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.

[59] James L McClelland and David E Rumelhart. *Explorations in parallel distributed processing: A handbook of models, programs, and exercises.* MIT press, 1989.

[60] 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.

[61] Edmund S Meltzer and Gonzalo M Sanchez. *The Edwin Smith Papyrus: updated translation of the trauma treatise and modern medical commentaries*. ISD LLC, 2014.

[62] 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.

[63] Marvin Minsky and Seymour Papert. *Perceptrons.* MIT press, 1969.

[64] 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.

[65] Wael MY Mohamed. Arab and muslim contributions to modern neuroscience. *IBRO History of Neuroscience*, 169(3):255, 2008.

[66] Alexander Mordvintsev, Christopher Olah, and Mike Tyka. Inceptionism: Going deeper into neural networks. *Google Research Blog. Retrieved June*, 20:14, 2015.

[67] Annalee Newitz. Movie written by algorithm turns out to be hilarious and intense. *Ars Technica*, 2016.

[68] Yael Niv. Reinforcement learning in the brain. *Journal of Mathematical Psychology*, 53(3):139–154, 2009.

[69] Christopher Olah. Understanding lstm networks. *GITHUB blog, posted on August*, 27:2015, 2015.

[70] 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.

[71] 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.

[72] 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.

[73] DB Parker. Learning-logic (tr-47). *Center for Computational Research in Economics and Management Science. MIT-Press, Cambridge, Mass*, 8, 1985.

[74] 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.

[75] Marcel Proust. *Remembrance of things past*, volume 2. Wordsworth Editions, 2006.

[76] 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.

[77] 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.

[78] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. A primer in bertology: What we know about how bert works. *Transactions of the Association for Computational Linguistics*, 8:842–866, 2020.

[79] Timothy T Rogers, James L McClelland, et al. *Semantic cognition: A parallel distributed processing approach*. MIT press, 2004.

[80] Frank Rosenblatt. Perceptron simulation experiments. *Proceedings of the IRE*, 48(3):301–309, 1960.

[81] Frank Rosenblatt. A comparison of several perceptron models. *Self-Organizing Systems*, pages 463–484, 1962.

[82] David E Rumelhart, James L McClelland, PDP Research Group, et al. Parallel distributed processing, vol. 1, 1986.

[83] Jürgen Schmidhuber, Sepp Hochreiter, et al. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[84] Wolfram Schultz, Peter Dayan, and P Read Montague. A neural substrate of prediction and reward. *Science*, 275(5306):1593–1599, 1997.

[85] Mark S Seidenberg and James L McClelland. A distributed, developmental model of word recognition and naming. *Psychological review*, 96(4):523, 1989.

[86] Terrence J Sejnowski and Charles R Rosenberg. Parallel networks that learn to pronounce english text. *Complex systems*, 1(1):145–168, 1987.

[87] 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.

[88] Helaine Selin. *Encyclopaedia of the history of science, technology, and medicine in non-westen cultures*. Springer Science & Business Media, 2013.

[89] 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.

[90] John Sutton. *Philosophy and memory traces: Descartes to connectionism*. Cambridge University Press, 1998.

[91] 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.

[92] John Von Neumann. The principles of large-scale computing machines. *Annals of the History of Computing*, 3(3):263–273, 1981.

[93] Maya Zhe Wang and Benjamin Y Hayden. Latent learning, cognitive maps, and curiosity. *Current Opinion in Behavioral Sciences*, 38:1–7, 2021.

[94] Paul Werbos. Beyond regression: New tools for prediction and analysis in the behavioral sciences. *Ph. D. dissertation, Harvard University*, 1974.

[95] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[96] Bernard Widrow. Adaline: Smarter than sweet, 1963.

[97] 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.

[98] Norbert Wiener. *Cybernetics: Control and communication in the animal and the machine*. Wiley New York, 1948.

[99] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

[100] 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.