CHAPTER 2 Learning

We have now seen one example of a machine learning algorithm: the SGNS model. In this chapter we will learn how to think about machine learning more generally. We will then introduce the framework of neural networks and learn how neural networks are fit to (or, in neural network terminology, *trained on*) data.

IN-CLASS EXERCISE

Consider the following dataset.

Input	**Label**
1, 2, 4	TRUE
2, 4, 8	TRUE
16, 32, 64	TRUE
2, 1, 4	FALSE
3, 2, 1	FALSE

This dataset describes a simple binary classification problem. The *input* is a sequence of three numbers, and it is assigned a *label* of TRUE or FALSE.

(2.1) EXERCISE
Based on the data above, how would you classify the following inputs? Justify your reasoning.

- 4, 8, 16
- 16, 8, 4
- 3, 6, 12
- 1, 2, 3
- 0, 0, 0

(2.2) EXERCISE
Suppose I told you the correct labels for the above examples were TRUE, FALSE, TRUE, TRUE, *and* FALSE. *Now, try classifying the following examples.*

- 5, 6, 7
- 5, 10, 20
- 3, 6, 9

Would you have classified these differently had I not told you the correct labels for the previous exercise?

(2.3) EXERCISE
What are some patterns that are compatible with the original dataset? What are some patterns compatible with the original dataset and the new data? Give examples of data points that help a learner distinguish between your possible patterns.

2.1 What Is Learning?

When we learn from experience, we are using *inductive reasoning*: making generalizations based on a finite amount of evidence. The moral of the in-class exercise is that when we observe something, there are a lot of possibilities for how to generalize from that data. In order to choose a reasonable generalization, we need to rely on an *inductive bias* that favors some hypotheses over others.

There are two well-known thought experiments that investigate the nature of inductive bias. The first we will cover here is described in the book *Word and Object* by philosopher Willard Van Orman Quine (1960). Imagine you are trapped on a desert island where you do not speak the local language. The villagers point to a rabbit and say,

(2.4) Gavagai!

What do you think *gavagai* means? Your first guess might be to say that it means *rabbit*. But how do you know that the villagers are pointing to the rabbit, and not the rabbit's ears or its whiskers? After all, if a person is pointing to a rabbit's ears or whiskers, that person is necessarily also pointing to the rabbit itself. Suppose that later in the day, you witness the villagers saying *Gavagai!* while pointing to a rabbit that is missing its ears and whiskers. Now you know that *gavagai* does not mean *rabbit ears* or *rabbit whiskers* or *rabbit that has its ears and whiskers*. But there are still numerous possibilities for what *gavagai* could mean: *rabbit*, *rabbit eyes*, *undetached rabbit parts*, etc.

Now let's go to another thought experiment, offered by philosopher Nelson Goodman in the book *Fact, Fiction, & Forecast* (Goodman, 1955). Suppose that you are an expert on precious gems, and every emerald you have ever seen is green. It would be reasonable for you to tell your friends that

(2.5) All emeralds are green.

Your professional experience as a gem expert provides plenty of evidence that statement (2.5) is true. But now consider a slightly different statement.

(2.6) All emeralds are grue.

Roughly speaking, Goodman defines the word *grue* as follows: an object is *grue* if

- it is green, and the first person to observe the object does so before January 1, 2025, or
- it is blue, and nobody has observed or will observe it before January 1, 2025.

In order for (2.6) to be true, any emerald that has not been observed by January 1, 2025 must be blue. Since that's more than three years in the future, any fact that is consistent with (2.5) is also consistent with (2.6), so we have just as much evidence for (2.6) as we do for (2.5). Yet, these two claims are clearly very different in substance, and make different predictions regarding what the world is like.

(2.7) EXERCISE
Do you think that both statements are equally valid? Why or why not?

Let's now turn to an inductive bias that linguists believe is specific to language. Consider the following sentences, from Lasnik and Lidz (2016).

(2.8) a. While he$_{i,j}$ was dancing, the Ninja Turtle$_i$ ate pizza.
b. He$_{*i,j}$ ate pizza while the Ninja Turtle$_i$ was dancing.

Whom does *he* refer to? In (2.8a), *he* could either be the Ninja Turtle (subscript i) or some other person (subscript j). But in (2.8b), *he* cannot be the Ninja Turtle. (In linguistics, we communicate this fact using the notation $*i$; the symbol $*$ generally means that something is impossible in a language.) The difference between these sentences in terms of whether or not *he* could possibly refer to *the Ninja Turtle* is due to a rule known as *Principle C*,[1] which states that *referential expressions* like *the Ninja Turtle* or *Donatello* or *New Haven*, which identify specific objects in the world without an antecedent (as opposed to pronouns), cannot appear in certain syntactic environments.

As far as we know, every language in the world obeys Principle C: there is no known language where a sentence can have a structure similar to (2.8b) in which *he* refers to *the Ninja Turtle*. Moreover, Crain and McKee (1985) have shown experimentally that children as young as 3 years old can tell the difference between (2.8a) and (2.8b) just as well as adults, even though nobody specifically tells them that *he* in (2.8b) *can't* refer to *the Ninja Turtle*. The fact that everyone seems to obey Principle C, and that we seem to know it from a very early age without being told about it, has led linguists to believe that humans are *born* knowing Principle C. In learning terms, Principle C forms part of the inductive bias that children use as they are trying to learn their native language. The innate inductive biases that are specific to language are collectively known as *universal grammar*, or UG.

2.2 Machine Learning and NLP Basics

The textbook *Machine Learning* by Tom M. Mitchell defines machine learning as follows:

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E (?).

[1]The "C" doesn't stand for anything; a lot of linguistics terminology doesn't make sense.

WHEN A USER TAKES A PHOTO, THE APP SHOULD CHECK WHETHER THEY'RE IN A NATIONAL PARK...
SURE, EASY GIS LOOKUP. GIMME A FEW HOURS.
... AND CHECK WHETHER THE PHOTO IS OF A BIRD.
I'LL NEED A RESEARCH TEAM AND FIVE YEARS.
IN CS, IT CAN BE HARD TO EXPLAIN THE DIFFERENCE BETWEEN THE EASY AND THE VIRTUALLY IMPOSSIBLE.

FIGURE 2.1: Machine learning is often applied to tasks for which it's very difficult to come up with a set of instructions. Title text: *In the 60s, Marvin Minsky assigned a couple of undergrads to spend the summer programming a computer to use a camera to identify objects in a scene. He figured they'd have the problem solved by the end of the summer. Half a century later, we're still working on it.* Source: xkcd

In other words, a computer program uses machine learning whenever the program requires the computer to use inductive reasoning. Traditionally, computer programs are thought of as a set of very detailed instructions for the hardware to execute. However, some tasks, such as those involving image recognition or NLP, are too complex to be solved using a list of instructions. For those kinds of tasks, machine learning has proven to be an extraordinarily powerful tool: if a computer has the ability to use inductive reasoning, then we no longer need to explicitly tell it how to solve a task. Instead, we simply show the computer *examples* of how the task is solved, and hopefully the program will be able to generalize those examples at runtime to new use cases.

2.2.1 The Three Ingredients of Machine Learning

In machine learning, we wish to create a program that computes some *target function* $f : \mathbb{A} \to \mathbb{B}$. Given an input $x \in \mathbb{A}$, the program's desired behavior is to produce the output $y = f(a)$. We do not actually know what kind of algorithm could compute f, but what we do have is a dataset $\mathbb{D} \subseteq \mathbb{A} \times \mathbb{B}$ that contains a collection of sample inputs along with their corresponding outputs (i.e., $y = f(x)$ for each $(x, y) \in \mathbb{D}$). Our goal is to create a *learning algorithm* that takes the dataset $\mathbb{D}$ and produces a *model* $\hat{f}$ that approximates f as closely as possible.

(2.9) EXERCISE
Notice that we have assumed that our dataset contains the correct output for each input. What is the name for the type of machine learning that makes this assumption? (Hint: we learned this in the last chapter.)

A typical machine learning algorithm consists of the following ingredients.

(2.10) The Three Ingredients of Machine Learning
  a. A *model architecture* with *parameters*
  b. A *loss function* that measures how poorly the model fits the dataset
  c. An *optimization algorithm* that minimizes the value of the loss function

Let's learn about each of these ingredients in detail.

Model Architecture. A learner cannot produce a model out of thin air; it needs an inductive bias that will tell it what a model could look like. After all, there are infinitely many ways to generalize from any particular dataset, and the learning algorithm must have some idea of which generalizations should be considered and which should be ignored. Formally, we say that models are *parameterized functions*: they are functions of the form $\hat{f}(x; \boldsymbol{\theta})$, where $x \in \mathbb{A}$ is the input to the function and $\boldsymbol{\theta} \in \mathbb{R}^p$ is a *parameter vector*. Each parameter vector $\boldsymbol{\theta} \in \mathbb{R}^p$ serves as an identifier for the model $\hat{f}(\cdot; \boldsymbol{\theta})$, and the learning algorithm produces different models by coming up with different parameter vectors. The *parameter space* is the set of all possible parameter vectors, in this case $\mathbb{R}^p$; and the *hypothesis space* is the set of all possible models $\hat{f}(\cdot; \boldsymbol{\theta})$. The goal of our learning algorithm will be to search the parameter space for the parameter vector that results in the closest possible model to the target function.

Loss Function. But how should the learning algorithm search through the parameter space? Since the parameter space is (usually) infinite, it's impossible to simply loop through all the possible parameter vectors; and it seems silly to, say, sample parameter vectors at random.[2] Somehow, we need to guide our learning algorithm by giving it something to look for. This is accomplished through the use of a *loss function* $L : \mathbb{B} \times \mathbb{B} \to \mathbb{R}$.[3] Suppose that on some input $x \in \mathbb{A}$, the target function outputs $y = f(x)$, while a model predicts that the output should be $\hat{y} = \hat{f}(x; \boldsymbol{\theta})$. The value of the loss function $L(\hat{y}, y)$ measures how incorrect the model prediction $\hat{y}$ is, compared to the correct answer y: the higher the loss, the more incorrect the prediction is. Therefore, our learning algorithm's search through the parameter space can be viewed as an optimization problem in which the model searches for the parameter vector that results in the lowest average loss over the dataset.

$$\min_{\boldsymbol{\theta}} \frac{1}{|\mathbb{D}|} \sum_{(x,y)\in\mathbb{D}} L(\hat{f}(x; \boldsymbol{\theta}), y)$$

Since the contents of the dataset do not depend on the progress of the learning algorithm's search through the parameter space, the $1/|\mathbb{D}|$ multiplier can be treated as a constant. Therefore, we can drop it without changing the solution to the optimization problem.

$$\min_{\boldsymbol{\theta}} \sum_{(x,y)\in\mathbb{D}} L(\hat{f}(x; \boldsymbol{\theta}), y)$$

[2] Though there is a type of machine learning called *genetic learning* that does this.
[3] Sometimes the loss function is known as a *criterion* or a *cost function*.

The actual expression whose value is being minimized is known as the *objective*. We will sometimes denote the objective by $\mathcal{L}$:

$$\mathcal{L} = \sum_{(x,y)\in\mathbb{D}} L(\hat{f}(x;\boldsymbol{\theta}), y).$$

Thus, the optimization problem can be stated succintly as $\min_\theta \mathcal{L}$.

Optimization Algorithm. Finally, we need an algorithm that can actually solve the problem of minimizing the objective. Computationally, optimization is an extremely hard problem, so we cannot simply download an optimization library from the internet and plug in our objective. In this course, we will primarily use an optimization algorithm called *stochastic gradient descent*, as well as enhanced versions thereof.

(2.11) EXAMPLE
A very simple example of a machine learning algorithm is *linear regression*. Chances are, you have probably studied linear regression before, or used it for analysis in science or social science. What are the three ingredients of linear regression?

a. **Architecture:** Linear regression models are of the form

$$\hat{f}(x;\boldsymbol{\theta}) = \theta_1 x + \theta_2,$$

where the parameter space is $\mathbb{R}^2$. We usually use the alternative notation

$$\hat{f}(x;a,b) = ax + b,$$

where a is known as the *slope* of the model and b is known as the *intercept*.

b. **Loss Function:** The loss function for linear regression is the *mean squared error* loss function, given by

$$L_{\text{MSE}}(\hat{y}, y) = (\hat{y} - y)^2.$$

The mean squared error loss function simply measures the difference the model prediction $\hat{y}$ and the target output y; however, this difference is squared in order to make the function differentiable. Our objective is

$$\mathcal{L} = \sum_{(x,y)\in\mathbb{D}} L_{\text{MSE}}(\hat{f}(x;a,b), y) = \sum_{(x,y)\in\mathbb{D}} (ax + b - y)^2.$$

Notice that the "mean" in "mean squared error" has been dropped, since we have dropped the $1/|\mathbb{D}|$ multiplier from our objective.

c. **Optimization Algorithm:** Since our objective is quadratic in terms of our parameters, the parameter values that minimize our objective will satisfy $\partial\mathcal{L}/\partial\boldsymbol{\theta} = \mathbf{0}$. Using this, we obtain the optimal parameter values by solving the system of equations

$$0 = \frac{\partial \mathcal{L}}{\partial a} = \frac{\partial}{\partial a} \sum_{(x,y)\in\mathbb{D}} (ax + b - y)^2$$
$$0 = \frac{\partial \mathcal{L}}{\partial b} = \frac{\partial}{\partial b} \sum_{(x,y)\in\mathbb{D}} (ax + b - y)^2.$$

From the second equation we obtain

$$0 = 2 \sum_{(x,y)\in\mathbb{D}} (ax + b - y) = 2|\mathbb{D}|b + 2 \sum_{(x,y)\in\mathbb{D}} (ax - y),$$

thus

$$b = \frac{1}{|\mathbb{D}|} \sum_{(x,y)\in\mathbb{D}} (y - ax).$$

From the first equation we get

$$\begin{aligned} 0 &= 2 \sum_{(x,y)\in\mathbb{D}} x(ax + b - y) \\ &= 2 \sum_{(x,y)\in\mathbb{D}} \left(ax^2 + \left(\frac{x}{|\mathbb{D}|} \sum_{(x',y')\in\mathbb{D}} (y' - ax') \right) - yx \right) \\ &= 2a \left(\sum_{(x,y)\in\mathbb{D}} x^2 - \frac{x}{|\mathbb{D}|} \sum_{(x',y')\in\mathbb{D}} x' \right) + \sum_{(x,y)\in\mathbb{D}} \left(-yx + \frac{x}{|\mathbb{D}|} \sum_{(x',y')\in\mathbb{D}} y' \right), \end{aligned}$$

hence

$$a = \frac{\sum_{(x,y)\in\mathbb{D}} \left(yx - \frac{x}{|\mathbb{D}|} \sum_{(x',y')\in\mathbb{D}} y' \right)}{2 \left(\sum_{(x,y)\in\mathbb{D}} x^2 - \frac{x}{|\mathbb{D}|} \sum_{(x',y')\in\mathbb{D}} x' \right)}$$

and

$$b = \frac{1}{|\mathbb{D}|} \sum_{(x,y)\in\mathbb{D}} \left(y - x \frac{\sum_{(x'',y'')\in\mathbb{D}} \left(y''x'' - \frac{x''}{|\mathbb{D}|} \sum_{(x',y')\in\mathbb{D}} y' \right)}{2 \left(\sum_{(x'',y'')\in\mathbb{D}} x''^2 - \frac{x''}{|\mathbb{D}|} \sum_{(x',y')\in\mathbb{D}} x' \right)} \right).$$

Our optimization algorithm will simply plug the dataset into the expressions above in order to obtain the optimal parameter values.

2.2.2 Machine Learning and NLP Tasks

So far, our discussion of machine learning has been quite abstract. In the previous subsection, we have thought of programs simply as functions that take some input and produce an output. But what exactly do these functions *do*? The particular problems that machine learning is meant to solve are known as *tasks*. Because this is a computational linguistics course, we will be focusing exclusively on tasks that involve natural language in some way. In this chapter, we will be learning about two broad categories of NLP tasks: *regression tasks* and *classification tasks* (though most of our attention will be devoted to the latter). Later in the course, we will learn about other kinds of NLP tasks.

Regression. In a regression task, the model must take an input and produce an output in the form of a real-valued number. In other words, the target function must be of the form $f : \mathbb{A} \to \mathbb{R}$. Because natural language expressions are generally understood to be *discrete* objects, regression tasks are relatively rare in NLP. A simple example, however, would be to take the synopsis of a movie and predict the Rotten Tomatoes rating for that movie.

A typical dataset for a regression task consists of pairs (x, y), where x is a string and $y \in \mathbb{R}$ is a real number. Regression models are typically fit using the mean squared error loss function.

Classification. In a classification task, the model must take an input and produce an output drawn from a finite set of discrete categories, or *classes*. In other words, the target must be of the form $f : \mathbb{A} \to \mathbb{B}$, where $\mathbb{B}$ is a finite set that we often think of as being $\{1, 2, \dots, n\}$ for some n. Typically, models for classification tasks are of the form $\hat{f}(\cdot, \boldsymbol{\theta}) : \mathbb{A} \to \mathbb{R}^n$, where for an input $x \in \mathbb{A}$ and predicted output $\hat{\boldsymbol{y}} = \hat{f}(x; \boldsymbol{\theta})$, the quantity $\hat{y}_i$ is interpreted as the probability that x belongs to class i.

In the last chapter, we saw that the SGNS model solves a binary classification problem (i.e., a classification problem where $n = 2$). There, the model produced only one output $\hat{y}$, which was interpreted as the probability of class 1 ("the two words occur together"); because there are only two classes, the probability of class 0 ("the two words do not occur together") is $1 - \hat{y}$. We used the sigmoid function to ensure that the output of the model is always a valid probability. When there are more than two classes, however, the model must separately compute a probability for each class. To ensure this, we use the *softmax* function, which turns a vector of real numbers into a vector of valid probabilities.

(2.12) DEFINITION

The *softmax function* is the function softmax : $\mathbb{R}^n \to \mathbb{R}^n$ given by

$$\operatorname{softmax}(\boldsymbol{x}) = \frac{e^{\boldsymbol{x}}}{\mathbf{1}^\top e^{\boldsymbol{x}}}.$$

(2.13) EXERCISE

Let $\boldsymbol{x} \in \mathbb{R}^n$ be any vector, and let $\boldsymbol{p} = \operatorname{softmax}(\boldsymbol{x})$. Show that

- *for all $i \leq n$, $p_i \geq 0$ and*
- $\mathbf{1}^\top \boldsymbol{p} = p_1 + p_2 + \cdots + p_n = 1$.

Thus, $\boldsymbol{p}$ describes a probability distribution over n categories. We say that $\boldsymbol{p}$ is a probability vector *when the above two conditions are met.*

Hint: observe that for each $i \leq n$,

$$p_i = \frac{e^{p_i}}{\sum_{j=1}^{n} e^{p_j}} = \frac{e^{p_i}}{e^{p_1} + e^{p_2} + \cdots + e^{p_n}}.$$

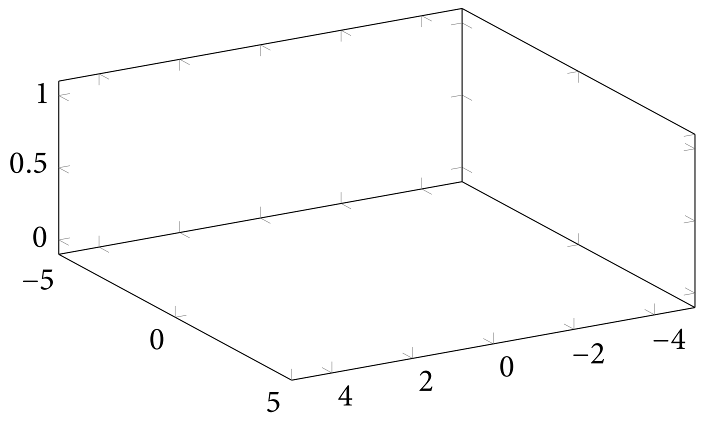


FIGURE 2.2: The softmax function, a multidimensional version of the sigmoid curve. This plot shows two-dimensional vectors $\boldsymbol{x} \in \mathbb{R}^2$ on the x- and y-axes, and $\mathrm{softmax}(\boldsymbol{x})_1$ on the z-axis.

When fitting our SGNS model, we used an objective based on the natural log of the probability of the target class predicted by the model. The loss function that produces this objective is called the *negative log-likelihood* or *cross-entropy loss function*.

(2.14) DEFINITION

The *cross-entropy loss function* is the function $L_{\mathrm{CE}} : \mathbb{R}^n \times \{1, 2, \dots, n\} \to \mathbb{R}$ defined by

$$L_{\mathrm{CE}}(\hat{\boldsymbol{y}}, y) = -\ln(\hat{y}_y).$$

In the previous chapter, we derived the cross-entropy loss function by using the method of maximum likelihood estimation—maximizing the probability assigned by our model to the data—and using the natural log to provide numerical stability. In Section 2.7, we give an alternate derivation of the cross-entropy loss function based on information theory. (This derivation will also explain where the name "cross-entropy" comes from.)

Here are some typical examples of classification tasks in NLP.

(2.15) Examples of classification tasks

a. **Spam filtering:** Classify an email as "spam" or "ham" (i.e., not spam).
b. **Fake news/hate speech detection:** Determine whether or not a text contains fake news or hate speech.
c. **Sentiment analysis:** Classify a text as having "positive sentiment" or "negative sentiment." For example, *This movie was amazing!* has positive sentiment, while *This movie was terrible!* has negative sentiment.
d. **Intent classification:** Take a natural-language command for a device or application, and figure out what the user wants the software to do. For example, in Amazon Alexa, the command *turn on the song thriller by michael jackson* is given the *domain* "MusicApp" and the *intent* "ListenMediaIntent," while the command *turn on the living room lights*

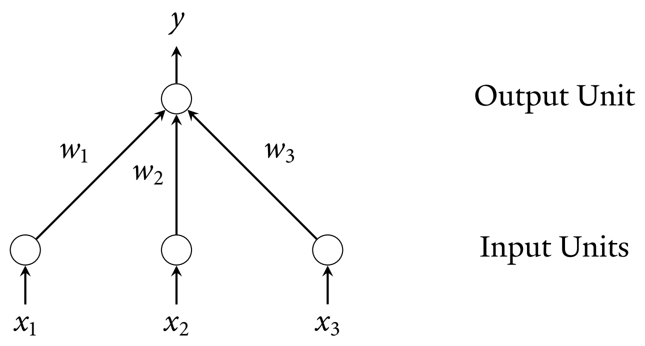


FIGURE 2.3: A perceptron.

belongs to the domain "HomeAutomation" and represents the intent "ActivateIntent" (Kollar et al., 2018).

Some of these tasks, particularly hate speech detection and sentiment analysis, may feel inherently unsolvable due to their subjective nature. It is hard to imagine, for example, that it would be possible for an algorithm to capture such intuitive notions as "sentiment" or "discrimination." This is not a problem for supervised learning, however: as long as the dataset contains the correct label for each example, the learning algorithm will try its best to generalize from the patterns in the dataset. Nonetheless, when you start preparing your paper presentation, it will be a good idea to think critically about what kinds of assumptions your chosen paper makes about the task it tries to solve and whether that task is even solvable in the first place.

2.3 Neural Networks

We now introduce *neural networks*, a conceptual framework for defining and understanding model architectures. We begin with a discussion of the biological origins of neural networks, and then introduce the *multi-layer perceptron* (MLP) architecture.

2.3.1 Perceptrons: A Model of the Nervous System

The *perceptron* is a model architecture invented by Rosenblatt (1957) in order to model neurons. Figure 2.3 shows a picture of a perceptron. Each of the circles in the picture is known as a *unit*, and each unit has some numerical value. The three circles on the bottom are called *input units*, and their values are the inputs to the perceptron (x_1, x_2, x_3). The circle on top is the *output unit*, and its value is the output of the perceptron (y). The units are *connected* to one another by arrows. In this picture, the input units are connected to the output unit, meaning that the output unit can access the values of the input units in order to determine its value. Each connection is associated with a *weight* (w_1, w_2, w_3), which represents the importance of each input for determining the model output.

The output of the perceptron is calculated as follows:

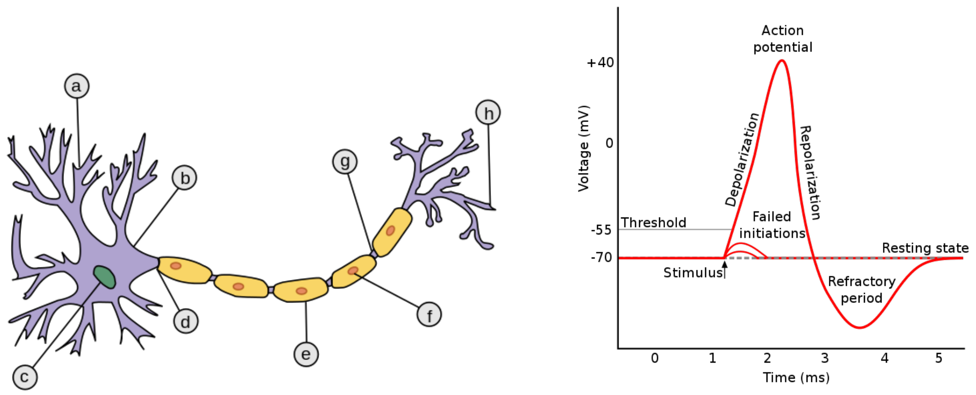


FIGURE 2.4: A typical neuron (left) receives input values from neighboring neurons through *dendrites* (a), and sends output values to neighboring neurons through the *axon* (d–h). Values are passed between neurons in the form of neurotransmitters. If neurotransmitters excite a neuron until its voltage rises above a certain threshold, the neuron fires an action potential (right), causing additional neurotransmitters to be released at its *axon terminal* (h). Image source: Wikimedia Commons user Lokal_Profil (CC BY-SA 3.0, left) and Wikipedia users Chris 73 and Diberri (CC BY-SA 3.0, right)

(2.16) Perceptron Formula

$$y = f(w_1x_1 + w_2x_2 + w_3x_3 + b)$$

where $f : \mathbb{R} \to \mathbb{R}$ is some function that is currently unspecified and b is a constant known as the *bias*. The value $w_1x_1 + w_2x_2 + w_3x_3 + b$ is known as the *activation*[4] of the output unit, and the function f is known as the *activation function*.

Perceptrons were originally developed as a simple model for a neuron. In Figure 2.3, the output unit represents a neuron, which receives input from the three input units. Biologically, the "activation" of a neuron is given by its *membrane potential*—the difference in voltage between the inside of the neuron and the outside. Neurons can communicate with one another by sending molecules known as *neurotransmitters* to connected neurons. Neurotransmitters have the ability to increase ("excite") or decrease ("inhibit") the membrane potential of neighboring cells by opening and closing *ion channels*, holes that allow ions to flow in and out of the cell. When one neuron increases the membrane potential of another, the connection between them is known as an *excitatory connection*; when a neuron decreases the membrane potential of another, the connection between them is known as an *inhibitory connection*. In a perceptron, excitatory connections are represented by positive weights ($w_i > 0$), while inhibitory connections are represented by negative weights ($w_i < 0$).

When the membrane potential of a neuron reaches a certain threshold, the neuron *fires an action potential*, causing neurotransmitters to be sent to neighboring neurons. In equation (2.16), the value of the activation function f represents the information that is sent to other neurons if an

[4]This terminology is used in Goodfellow et al.'s (2016) textbook, but it is not universally accepted. Whereas we use the word *activation* to refer to the input of the activation function, it is also often used to refer to the *output* of the activation function.

action potential is fired. One simple example of an activation function is the *step function*, defined below.

$$f(x) = \begin{cases} 0, & x \le 0 \\ 1, & x > 0 \end{cases}$$

In biological terms, the step-function activation says that when the activation is above the threshold of 0, the neuron fires an action potential and sends the value 1 to neighboring neurons. If the activation is not above the threshold, it sends the value 0 to neighboring neurons.

(2.17) EXERCISE
Suppose 0 *represents "false" and* 1 *represents "true." Using the step function as the activation function, construct a perceptron (i.e., give values for w_1, w_2, w_3, and b) such that y is the value of $x_1 \wedge x_2$. ($\wedge$ means "and," $\vee$ means "or," and $\neg$ means "not.")*

2.3.2 Activation Functions

When we use the step function as our activation function, our perceptron becomes a simple binary classifier that always assigns one class label a probability score of 1 and the other class label a probability score of 0. More typically, though, we use the sigmoid or softmax as an activation function, since non-continuous functions do not work well with optimization algorithms, which are typically based on calculus.

Depending on what we want our model to do, we might use a number of different activation functions. Here are two more examples of activation functions, other than sigmoid and softmax. The plots of these functions are shown in Figure 2.5.

(2.18) Examples of Activation Functions

a. **Rectifier:** The *rectifier* activation function is given by

$$\text{ReLU}(x) = \max(x, 0).$$

Like the step function, the rectifier activation represents a neuron firing an action potential after its activation crosses the threshold of 0. However, instead of sending the value 1 to its neighbors, the unit instead sends the value of its activation. A unit with rectifier activation is known as a *rectified linear unit* (hence the notation ReLU).

b. **Hyperbolic Tangent:** The *hyperbolic tangent* function is another s-shaped curve, which takes values between −1 and 1 instead of 0 and 1. It is defined as follows.

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

The hyperbolic tangent will come in handy in **??**, when we learn about long short-term memory networks. The rectifier will be used in the Transformer networks of **??**. We will learn more about the interpretations of these activation functions as we learn about more complex neural network architectures.

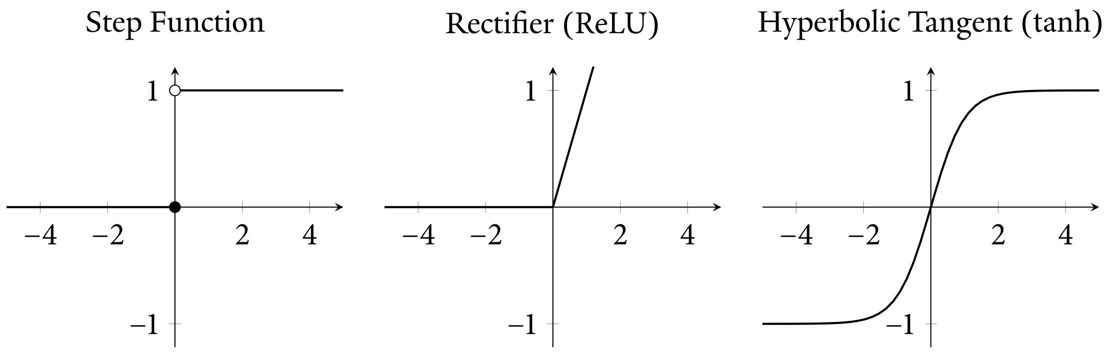


FIGURE 2.5: Various activation functions.

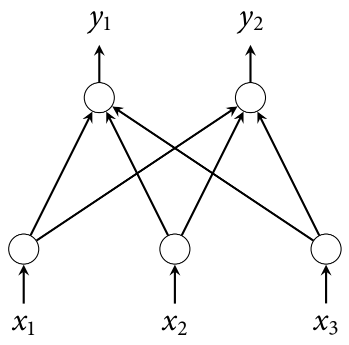


FIGURE 2.6: A fully-connected network.

(2.19) EXERCISE

Typically, we always use an activation function with a perceptron: very rarely do we use a perceptron without an activation function. Why is this?

Hint: Consider a neural network with two perceptrons linked together.

$$\begin{tikzcd} x \arrow[r] & \mathit{0} \arrow[r] & \mathit{1} \arrow[r] & \mathit{2} \arrow[r] & y \end{tikzcd}$$

Unit 0 is an input unit that simply transfers the value of x to unit 1. Unit 1 then calculates the value $u = f(w_1 x + b_1)$ and passes it to unit 2, which calculates the output $y = g(w_2 u + b_2)$. What happens if unit 1's activation function, f, is the identity function?

2.3.3 Multiple Outputs

The perceptron implements functions that take one or more numbers as input, but can only produce one output. Let's now consider a more complex neural network architecture that can produce more than one output. Figure 2.6 shows a *fully-connected network* with three input units and two output units. Each input unit has a connection to each of the output units. Let us denote the weight of the connection from input unit i to output unit j by $W_{j,i}$, and let us denote the bias of output unit j by b_j. Let us further assume that the two output units have the same activation function f. Then, the values of the two outputs are computed as follows.

(2.20) Fully-Connected Network Computation

$$y_1 = f(W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1)$$
$$y_2 = f(W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2)$$

We can write these equations more compactly if we represent the inputs, outputs, weights, and biases as matrices.

$$\boldsymbol{W} = \begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \end{bmatrix} \qquad \boldsymbol{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \qquad \boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \qquad \boldsymbol{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

Then, (2.20) can be rewritten as a linear map followed by an activation function.

(2.21) Equation (2.20), in Matrix Form

$$\boldsymbol{y} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

(2.22) EXERCISE
Expand (2.21) using matrix multiplication to find the values of y_1 and y_2, and verify that they are the same as the values given in equation (2.20).

Since our fully-connected network has multiple outputs, we can use the softmax activation function to classify an input into more than two classes.

(2.23) EXERCISE
One advantage of writing our equations using matrices is that it is more compact than writing them using scalars. What is another advantage offered by the matrix notation?
Hint: Consider a fully-connected network where the output units have a softmax activation function.

(2.24) EXERCISE
The sigmoid activation function represents binary classification problems, while softmax represents multiple classification problems. Given a sigmoid perceptron, construct a fully-connected network with softmax activation that implements the same binary classification problem.

2.3.4 Multi-Layer Perceptrons

Let's now think about a perceptron with two input units and step-function activation.

$$y = f(w_1x_1 + w_2x_2 + b)$$

Suppose 1 represents "true" and 0 represents "false." Since the output of our perceptron is always 0 or 1, it now implements a boolean function. The output is "true" ($y = 1$) when the activation value is positive:

$$w_1x_1 + w_2x_2 + b > 0.$$

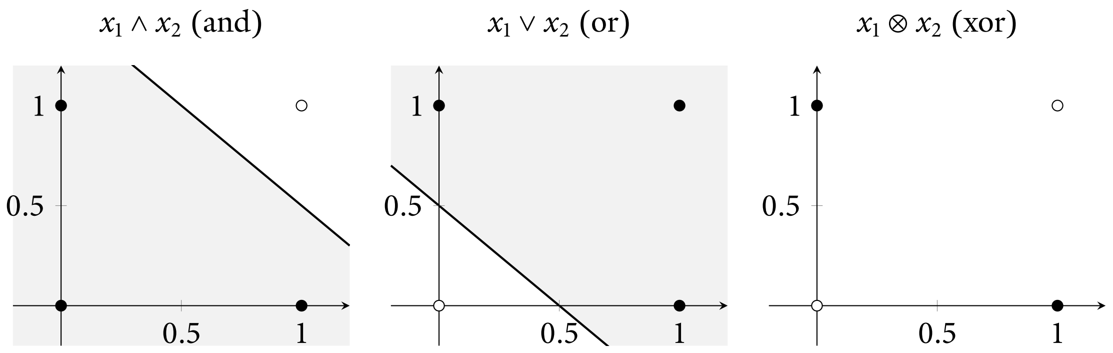


FIGURE 2.7: Boolean functions. • means $y = 1$ and ◦ means $y = 0$.

What this means is that if we plot the line $w_1x_1 + w_2x_2 + b = 0$ on the x_1–x_2 plane, then all the inputs $\boldsymbol{x}$ to which the perceptron assigns the value of "true" are on one side of the line, and all the inputs assigned the value of "false" are on the other side. (Since the step function maps 0 to 0, points on the line are assigned the value of "false.") Functions that can be described this way are called *linearly separable*.

(2.25) EXAMPLE
The left and center panels of Figure 2.7 show plots for $y = x_1 \wedge x_2$ and $y = x_1 \vee x_2$. Notice that both functions are linearly separable. The line separates the $y = 1$ points, which lie in the shaded region, from the $y = 0$ points, which lie in the unshaded region.

One function that is *not* linearly separable is $y = x_1 \otimes x_2$. The $\otimes$ symbol represents *exclusive or*, or xor. $x_1 \otimes x_2$ is true if x_1 is true or x_2 is true, but not both. To see why $y = x_1 \otimes x_2$ is not linearly separable, look at the right panel of Figure 2.7. Consider any line that you might draw on the plane. If both •s are above the line, then the ◦ at $(1, 1)$ must also be above the line. If both •s are below the line, then the ◦ at $(0, 0)$ must be below the line. This means that any perceptron with step-function activation that attempts to compute $y = x_1 \otimes x_2$ must get at least one input wrong.[5]

The fact that step-function perceptrons can only represent linearly separable functions severely limits their applicability to machine learning tasks. Most things we would like neural networks to learn cannot be described by linearly separable functions. Therefore, in order to solve complex tasks with neural networks, we need to use architectures more sophisticated than perceptrons.

Figure 2.8 shows a *multi-layer perceptron*. It consists of a fully-connected network attached to a perceptron. The units in the middle, which form the output of the fully-connected network and the

[5]This was first discovered in a book by Minsky and Papert (1969). The critics concluded that neural networks are useless, since perceptrons can't even compute simple functions like xor. This conclusion turned out to be incorrect.

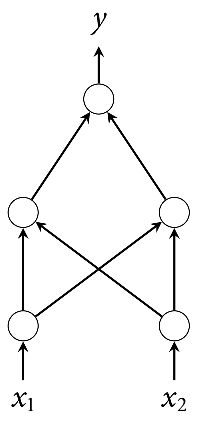


FIGURE 2.8: A multi-layer perceptron.

input of the perceptron, are known as *hidden units*. The computation of the network is as follows.

$$\boldsymbol{h} = f\left(\boldsymbol{W}^{(h)}\boldsymbol{x} + \boldsymbol{b}^{(h)}\right) \qquad \text{(Hidden Units)}$$
$$y = g\left(\boldsymbol{W}^{(o)}\boldsymbol{h} + b^{(o)}\right) \qquad \text{(Output Unit)}$$

Multi-layer perceptrons can easily implement the xor function.

(2.26) EXAMPLE

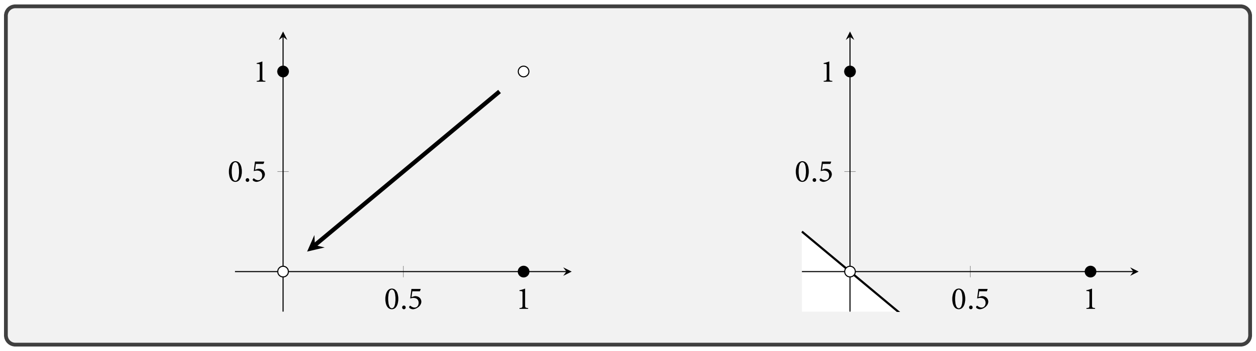
Let f be the step function, and let g be the identity function. Then, $x_1 \otimes x_2$ is implemented by the following multi-layer perceptron.[6]

$$\boldsymbol{W}^{(h)} = \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \qquad \boldsymbol{b}^{(h)} = \boldsymbol{0} \qquad \boldsymbol{W}^{(o)} = \boldsymbol{1}^\top \qquad b^{(o)} = 0$$

To see how this works, let's compute $\boldsymbol{h}$ for each of the four possible boolean inputs.

$$f\left(\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\begin{bmatrix} 0 \\ 0 \end{bmatrix}\right) = f\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$
$$f\left(\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ 0 \end{bmatrix}\right) = f\left(\begin{bmatrix} 1 \\ -1 \end{bmatrix}\right) = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$
$$f\left(\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\begin{bmatrix} 0 \\ 1 \end{bmatrix}\right) = f\left(\begin{bmatrix} -1 \\ 1 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$
$$f\left(\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ 1 \end{bmatrix}\right) = f\left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}\right) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

$\boldsymbol{h}$ is just the same thing as $\boldsymbol{x}$, unless $x_1 = x_2 = 1$, in which case $\boldsymbol{h} = \boldsymbol{0}$. Visually, the hidden units serve to shift the $\circ$ at $(1,1)$ to $(0,0)$, resulting in a linearly separable function.



The xor example demonstrates the power of having multiple layers of units in a neural network. Whereas a single perceptron cannot compute the xor function, the intermediate layer constructs a different representation of the inputs that makes it compatible with the output perceptron's architecture. In general, neural networks with many layers of units are known as *deep neural networks*, and this is where the name *deep learning* comes from.

(2.27) EXERCISE

The following multi-layer perceptron also computes $y = x_1 \otimes x_2$.

$$\begin{aligned} \boldsymbol{h} &= \mathrm{ReLU}\left(\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}\boldsymbol{x} + \begin{bmatrix} 0 \\ -1 \end{bmatrix}\right) \\ y &= \begin{bmatrix} 1 \\ -2 \end{bmatrix}\boldsymbol{x} \end{aligned}$$

What kind of intermediate representation is computed by the hidden units?

We can do multiple classification with a multi-layer perceptron by replacing the perceptron at the top with a second fully-connected network. A common architecture for this uses sigmoid activation in the hidden layer, softmax activation in the output layer, and no bias in the output layer.

$$\begin{aligned} \boldsymbol{h} &= \sigma\left(\boldsymbol{W}^{(h)}\boldsymbol{x} + \boldsymbol{b}^{(h)}\right) \\ \boldsymbol{y} &= \mathrm{softmax}\left(\boldsymbol{W}^{(o)}\boldsymbol{x}\right) \end{aligned}$$

2.3.5 The NLP Pipeline

So far, we have thought of perceptrons, multi-layer perceptrons, and fully-connected networks as functions that take real-numbered values for both their inputs and outputs. In order to neural network models of language, we will need to figure out a way to apply neural networks to textual data. We do this by converting texts into vectors—a process known as *preprocessing*.

In the first step of preprocessing, the data must be *cleaned* of any irrelevant irrelevant information. Often, this entails removing formatting information from webpages or other kinds of documents. Sometimes, developers may optionally choose to perform one or more of the following steps for data cleaning.

[6] **0** is the zero vector and **1** is the vector with all 1s.

```
'The cat said to the dog, "Hello world!"'
```

$$\downarrow$$

```
['The', 'cat', 'said', 'to', 'the', 'dog', ',',
      '"', 'Hello', 'world', '!', '"']
```

$$\downarrow$$

$$\begin{bmatrix}1109 & 5855 & 1163 & 1106 & 1103 & 3676 & 117 & 107 & 8667 & 1362 & 106 & 107\end{bmatrix}^\top$$

$$\downarrow$$

$$\begin{bmatrix}-.035 & -.073 & .013 & .038 & -.041 & .009 & -.042 & -.038 & -.023 & .020 & -.023 & -.038 \\ .018 & -.041 & .006 & .015 & .029 & -.089 & .029 & .039 & .022 & -.015 & .042 & .039 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ .005 & -.013 & .011 & -.033 & .024 & -.070 & -.016 & .044 & .012 & -.035 & .010 & .044\end{bmatrix}^\top$$

FIGURE 2.9: Converting a string to tokens, and then to indices, and then to word embeddings.

(2.28) Data Cleaning

a. **Special characters:** Collapse whitespace sequences into single spaces; replace uncommon punctuation marks with more common ones, or remove them altogether. Numbers might be replaced by a generic “number” symbol.

b. **Casing:** Convert all letters to lowercase.

c. **Stemming:** Remove affixes (prefixes and suffixes) from words, leaving only the stem of each word.

d. **Stopwords:** Remove common words that occur with high frequency but provide little semantic information, such as function words like *the* or *a*.

Cleaning is often accomplished using rule-based methods such as regular expression substitution.

After the data is cleaned, it must be *tokenized*. Tokenization is the process of dividing a string into *tokens*. A token usually represents one word, but may also represent a punctuation symbol or a part of a word. The set of all possible tokens is called the *vocabulary*. Vocabularies often include a special token called `[UNK]`, which represents an unknown or *out-of-vocabulary* (OOV) word; the `[UNK]` token is used whenever a text contains a token that is not found in the vocabulary. Some tokenization schemes, known as *subword tokenization* schemes, have tokens for each possible letter or punctuation mark; such tokenization schemes do not require an `[UNK]` token because unknown words can be decomposed into individual letters or other sub-word tokens.

Each item of the vocabulary is assigned an integer known as its *index*. After a text has been tokenized, each token is replaced with its index, resulting in a list of integers. These indices are then used to look up word embeddings from a *word embedding matrix* whose ith row is the word embedding for the token with index i. The final representation of our text is a matrix $\boldsymbol{X}$ where each row contains the word embedding for the corresponding token in the text.

In order to apply a perceptron, multi-layer perceptron, or fully-connected network to a text, we must somehow *flatten* our matrix of word embeddings into a vector. A common way of doing this is to simply add up all the word embeddings; i.e., to convert $\boldsymbol{X}$ into the vector $\boldsymbol{X}^\top\mathbf{1}$. Another way is to concatenate all the rows of $\boldsymbol{X}$ together, one after another. A disadvantage of the latter method is that the number of input units in the network now depends on the length of the input text. After $\boldsymbol{X}$ has been flattened into a vector, the vector is then fed into the input units of the network.