# Lecture Notes – MIT 6.390 Spring 2025

# Contents

# CHAPTER 1

## Introduction

The main focus of machine learning (ML) is *making decisions or predictions based on data*. There are a number of other fields with significant overlap in technique, but difference in focus: in economics and psychology, the goal is to discover underlying causal processes and in statistics it is to find a model that fits a data set well. In those fields, the end product is a model. In machine learning, we often fit models, but as a means to the end of making good predictions or decisions.

This description paraphrased from a post on 9/4/12 at `andrewgelman.com`

As ML methods have improved in their capability and scope, ML has become arguably the best way–measured in terms of speed, human engineering time, and robustness–to approach many applications. Great examples are face detection, speech recognition, and many kinds of language-processing tasks. Almost any application that involves understanding data or signals that come from the real world can be nicely addressed using machine learning.

One crucial aspect of machine learning approaches to solving problems is that human engineering plays an important role. A human still has to *frame* the problem: acquire and organize data, design a space of possible solutions, select a learning algorithm and its parameters, apply the algorithm to the data, validate the resulting solution to decide whether it's good enough to use, try to understand the impact on the people who will be affected by its deployment, etc. These steps are of great importance.

and often undervalued

The conceptual basis of learning from data is the *problem of induction*: Why do we think that previously seen data will help us predict the future? This is a serious long standing philosophical problem. We will operationalize it by making assumptions, such as that all training data are so-called i.i.d.(independent and identically distributed), and that queries will be drawn from the same distribution as the training data, or that the answer comes from a set of possible answers known in advance.

This means that the elements in the set are related in the sense that they all come from the same underlying probability distribution, but not in any other ways.

In general, we need to solve these two problems:

- **estimation:** When we have data that are noisy reflections of some underlying quantity of interest, we have to aggregate the data and make estimates or predictions about the quantity. How do we deal with the fact that, for example, the same treatment may end up with different results on different trials? How can we predict how well an estimate may compare to future results?

- **generalization:** How can we predict results of a situation or experiment that we have never encountered before in our data set?

We can describe problems and their solutions using six characteristics, three of which characterize the problem and three of which characterize the solution:

1. **Problem class:** What is the nature of the training data and what kinds of queries will be made at testing time?
2. **Assumptions:** What do we know about the source of the data or the form of the solution?
3. **Evaluation criteria:** What is the goal of the prediction or estimation system? How will the answers to individual queries be evaluated? How will the overall performance of the system be measured?
4. **Model type:** Will an intermediate model of the world be made? What aspects of the data will be modeled in different variables/parameters? How will the model be used to make predictions?
5. **Model class:** What particular class of models will be used? What criterion will we use to pick a particular model from the model class?
6. **Algorithm:** What computational process will be used to fit the model to the data and/or to make predictions?

Without making some assumptions about the nature of the process generating the data, we cannot perform generalization. In the following sections, we elaborate on these ideas.

Don't feel you have to memorize all these kinds of learning, etc. We just want you to have a very high-level view of (part of) the breadth of the field.

## 1.1 Problem class

There are many different *problem classes* in machine learning. They vary according to what kind of data is provided and what kind of conclusions are to be drawn from it. Five standard problem classes are described below, to establish some notation and terminology.

In this course, we will focus on classification and regression (two examples of supervised learning), and we will touch on reinforcement learning, sequence learning, and clustering.

### 1.1.1 Supervised learning

The idea of *supervised* learning is that the learning system is given inputs and told which specific outputs should be associated with them. We divide up supervised learning based on whether the outputs are drawn from a small finite set (classification) or a large finite ordered set or continuous set (regression).

#### 1.1.1.1 Regression

For a regression problem, the training data $\mathcal{D}_n$ is in the form of a set of $n$ pairs:

$$\mathcal{D}_n = \{(x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)})\},$$

where $x^{(i)}$ represents an input, most typically a d-dimensional vector of real and/or discrete values, and $y^{(i)}$ is the output to be predicted, in this case a real-number. The $y$ values are sometimes called *target values*.

Many textbooks use $x_i$ and $t_i$ instead of $x^{(i)}$ and $y^{(i)}$. We find that notation somewhat difficult to manage when $x^{(i)}$ is itself a vector and we need to talk about its elements. The notation we are using is standard in some other parts of the ML literature.

The goal in a regression problem is ultimately, given a new input value $x^{(n+1)}$, to predict the value of $y^{(n+1)}$. Regression problems are a kind of *supervised learning*, because the desired output $y^{(i)}$ is specified for each of the training examples $x^{(i)}$.

#### 1.1.1.2 Classification

A classification problem is like regression, except that the values that $y^{(i)}$ can take do not have an order. The classification problem is *binary* or *two-class* if $y^{(i)}$ (also known as the *class*) is drawn from a set of two possible values; otherwise, it is called *multi-class*.

### 1.1.2 Unsupervised learning

*Unsupervised* learning doesn't involve learning a function from inputs to outputs based on a set of input-output pairs. Instead, one is given a data set and generally expected to find some patterns or structure inherent in it.

#### 1.1.2.1 Clustering

Given samples $x^{(1)}, \dots, x^{(n)} \in \mathbb{R}^d$, the goal is to find a partitioning (or "clustering") of the samples that groups together similar samples. There are many different objectives, depending on the definition of the similarity between samples and exactly what criterion is to be used (e.g., minimize the average distance between elements inside a cluster and maximize the average distance between elements across clusters). Other methods perform a "soft" clustering, in which samples may be assigned 0.9 membership in one cluster and 0.1 in another. Clustering is sometimes used as a step in the so-called density estimation (described below), and sometimes to find useful structure or influential features in data.

#### 1.1.2.2 Density estimation

Given samples $x^{(1)}, \dots, x^{(n)} \in \mathbb{R}^d$ drawn i.i.d. from some distribution $\Pr(X)$, the goal is to predict the probability $\Pr(x^{(n+1)})$ of an element drawn from the same distribution. Density estimation sometimes plays a role as a "subroutine" in the overall learning method for supervised learning, as well.

The capital $X$ is a typical practice to emphasize this is a so-called random variable. Small letters are often used in probability too; those are typically reserved to denote the realized values of random variables. It might help to concretely think of coin-tosses; there, the toss outcome is a random variable and it may be realized as a "head". This paragraph actually talks about both a random variable and a realization of it, can you spot that from the notation and do you feel the difference?

#### 1.1.2.3 Dimensionality reduction

Given samples $x^{(1)}, \dots, x^{(n)} \in \mathbb{R}^D$, the problem is to re-represent them as points in a $d$-dimensional space, where $d < D$. The goal is typically to retain information in the data set that will, e.g., allow elements of one class to be distinguished from another.

Dimensionality reduction is a standard technique that is particularly useful for visualizing or understanding high-dimensional data. If the goal is ultimately to perform regression or classification on the data after the dimensionality is reduced, it is usually best to articulate an objective for the overall prediction problem rather than to first do dimensionality reduction without knowing which dimensions will be important for the prediction task.

### 1.1.3 Sequence learning

In sequence learning, the goal is to learn a mapping from *input sequences* $x_0, \dots, x_n$ to *output sequences* $y_1, \dots, y_m$. The mapping is typically represented as a *state machine*, with one function $f_s$ used to compute the next hidden internal state given the input, and another function $f_o$ used to compute the output given the current hidden state.

It is supervised in the sense that we are told what output sequence to generate for which input sequence, but the internal functions have to be learned by some method other than direct supervision, because we don't know what the hidden state sequence is.

### 1.1.4 Reinforcement learning

In reinforcement learning, the goal is to learn a mapping from input values (typically assumed to be states of an agent or system; for now, think e.g. the velocity of a moving car) to output values (typically we want control actions; for now, think e.g. if to accelerate or hit the brake). However, we need to learn the mapping without a direct supervision signal to specify which output values are best for a particular input; instead, the learning problem is framed as an agent interacting with an environment, in the following setting:

- The agent observes the current state $s_t$.
- It selects an action $a_t$.
- It receives a reward, $r_t$, which typically depends on $s_t$ and possibly $a_t$.
- The environment transitions probabilistically to a new state, $s_{t+1}$, with a distribution that depends only on $s_t$ and $a_t$.
- The agent observes the current state, $s_{t+1}$.
- ...

Note it's standard practice in reinforcement learning to use $s$ and $a$ instead of $x$ and $y$ to denote the machine learning model's input and output. The subscript $t$ denotes the timestep, and captures the sequential nature of the problem.

The goal is to find a policy $\pi$, mapping $s$ to $a$, (that is, states to actions) such that some long-term sum or average of rewards $r$ is maximized.

This setting is very different from either supervised learning or unsupervised learning, because the agent's action choices affect both its reward and its ability to observe the environment. It requires careful consideration of the long-term effects of actions, as well as all of the other issues that pertain to supervised learning.

### 1.1.5 Other settings

There are many other problem settings. Here are a few.

In *semi-supervised* learning, we have a supervised-learning training set, but there may be an additional set of $x^{(i)}$ values with no known $y^{(i)}$. These values can still be used to improve learning performance (if they are drawn from $\Pr(X)$ that is the marginal of $\Pr(X, Y)$ that governs the rest of the data set).

In *active* learning, it is assumed to be expensive to acquire a label $y^{(i)}$ (imagine asking a human to read an x-ray image), so the learning algorithm can sequentially ask for particular inputs $x^{(i)}$ to be labeled, and must carefully select queries in order to learn as effectively as possible while minimizing the cost of labeling.

In *transfer* learning (also called *meta-learning*), there are multiple tasks, with data drawn from different, but related, distributions. The goal is for experience with previous tasks to apply to learning a current task in a way that requires decreased experience with the new task.

## 1.2 Assumptions

The kinds of assumptions that we can make about the data source or the solution include:

- The data are independent and identically distributed (i.i.d.).
- The data are generated by a Markov chain (i.e. outputs only depend only on the current *state*, with no additional *memory*).
- The process generating the data might be adversarial.

- The "true" model that is generating the data can be perfectly described by one of some particular set of hypotheses.

The effect of an assumption is often to reduce the "size" or "expressiveness" of the space of possible hypotheses and therefore reduce the amount of data required to reliably identify an appropriate hypothesis.

## 1.3 Evaluation criteria

Once we have specified a problem class, we need to say what makes an output or the answer to a query good, given the training data. We specify evaluation criteria at two levels: how an individual prediction is scored, and how the overall behavior of the prediction or estimation system is scored.

The quality of predictions from a learned model is often expressed in terms of a *loss function*. A loss function $\mathcal{L}(g, a)$ tells you how much you will be penalized for making a guess $g$ when the answer is actually $a$. There are many possible loss functions. Here are some frequently used examples:

- **0-1 Loss** applies to predictions drawn from finite domains.

$$\mathcal{L}(g, a) = \begin{cases} 0 & \text{if } g = a \\ 1 & \text{otherwise} \end{cases}$$

If the actual values are drawn from a continuous distribution, the probability they would ever be equal to some predicted $g$ is 0 (except for some weird cases).

- **Squared loss**

$$\mathcal{L}(g, a) = (g - a)^2$$

- **Absolute loss**

$$\mathcal{L}(g, a) = |g - a|$$

- **Asymmetric loss** Consider a situation in which you are trying to predict whether someone is having a heart attack. It might be much worse to predict "no" when the answer is really "yes", than the other way around.

$$\mathcal{L}(g, a) = \begin{cases} 1 & \text{if } g = 1 \text{ and } a = 0 \\ 10 & \text{if } g = 0 \text{ and } a = 1 \\ 0 & \text{otherwise} \end{cases}$$

Any given prediction rule will usually be evaluated based on multiple predictions and the loss of each one. At this level, we might be interested in:

- Minimizing expected loss over all the predictions (also known as *risk*)
- Minimizing maximum loss: the loss of the worst prediction
- Minimizing or bounding regret: how much worse this predictor performs than the best one drawn from some class
- Characterizing asymptotic behavior: how well the predictor will perform in the limit of infinite training data
- Finding algorithms that are probably approximately correct: they probably generate a hypothesis that is right most of the time.

There is a theory of rational agency that argues that you should always select the action that *minimizes the expected loss*. This strategy will, for example, make you the most money in the long run, in a gambling setting. As mentioned above, expected loss is also sometimes called risk in ML literature, but that term means other things in economics or other parts of decision theory, so be careful...it's risky to use it. We will, most of the time, concentrate on this criterion.

Of course, there are other models for action selection and it's clear that people do not always (or maybe even often) select actions that follow this rule.

## 1.4 Model type

Recall that the goal of a ML system is typically to estimate or generalize, based on data provided. Below, we examine the role of model-making in machine learning.

### 1.4.1 Non-parametric models

In some simple cases, in response to queries, we can generate predictions directly from the training data, without the construction of any intermediate model, or more precisely, without the learning of any parameters.

For example, in regression or classification, we might generate an answer to a new query by averaging answers to recent queries, as in the *nearest neighbor* method.

### 1.4.2 Parametric models

This two-step process is more typical:

1. "Fit" a model (with some a-prior chosen parameterization) to the training data
2. Use the model directly to make predictions

In the *parametric models* setting of regression or classification, the model will be some hypothesis or prediction rule $y = h(x;\Theta)$ for some functional form $h$. The term *hypothesis* has its roots in statistical learning and the scientific method, where models or hypotheses about the world are tested against real data, and refined with more evidence, observations, or insights. Note that the parameters themselves are only part of the assumptions that we're making about the world. The model itself is a hypothesis that will be refined with more evidence.

The idea is that $\Theta$ is a set of one or more parameter values that will be determined by fitting the model to the training data and then be held fixed during testing.

Given a new $x^{(n+1)}$, we would then make the prediction $h(x^{(n+1)};\Theta)$.

We write $f(a;b)$ to describe a function that is usually applied to a single argument $a$, but is a member of a parametric family of functions, with the particular function determined by parameter value $b$.

The fitting process is often articulated as an optimization problem: Find a value of $\Theta$ that minimizes some criterion involving $\Theta$ and the data. An optimal strategy, if we knew the actual underlying distribution on our data, $\Pr(X, Y)$ would be to predict the value of $y$ that minimizes the *expected loss*, which is also known as the *test error*. If we don't have that actual underlying distribution, or even an estimate of it, we can take the approach of minimizing the *training error*: that is, finding the prediction rule $h$ that minimizes the average loss on our training data set. So, we would seek $\Theta$ that minimizes

This notation describes a so-called "joint distribution"; roughly, as the name suggests, it captures how both random variables X and Y "contribute" to the chance of something happening.

$$\mathcal{E}_n(h;\Theta) = \frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(h(x^{(i)};\Theta), y^{(i)}) \ ,$$

where the loss function $\mathcal{L}(g,a)$ measures how bad it would be to make a guess of $g$ when the actual value is $a$.

We will find that minimizing training error alone is often not a good choice: it is possible to emphasize fitting the current data too strongly and end up with a hypothesis that does not generalize well when presented with new $x$ values.

## 1.5 Model class and parameter fitting

A model *class* $\mathcal{M}$ is a set of possible models, typically parameterized by a vector of parameters $\Theta$. What assumptions will we make about the form of the model? When solving a regression problem using a prediction-rule approach, we might try to find a linear function $h(x; \theta, \theta_0) = \theta^T x + \theta_0$ that fits our data well. In this example, the parameter vector $\Theta = (\theta, \theta_0)$.

For problem types such as classification, there are huge numbers of model classes that have been considered...we'll spend much of this course exploring these model classes, especially neural networks models. We will almost completely restrict our attention to model classes with a fixed, finite number of parameters. Models that relax this assumption are called "non-parametric" models.

How do we select a model class? In some cases, the ML practitioner will have a good idea of what an appropriate model class is, and will specify it directly. In other cases, we may consider several model classes and choose the best based on some objective function. In such situations, we are solving a *model selection* problem: model-selection is to pick a model class $\mathcal{M}$ from a (usually finite) set of possible model classes, whereas *model fitting* is to pick a particular model in that class, specified by (usually continuous) parameters $\Theta$.

## 1.6 Algorithm

Once we have described a class of models and a way of scoring a model given data, we have an algorithmic problem: what sequence of computational instructions should we run in order to find a good model from our class? For example, determining the parameter vector which minimizes the training error might be done using a familiar least-squares minimization algorithm, when the model $h$ is a function being fit to some data $x$.

Sometimes we can use software that was designed, generically, to perform optimization. In many other cases, we use algorithms that are specialized for ML problems, or for particular hypotheses classes. Some algorithms are not easily seen as trying to optimize a particular criterion. In fact, a historically important method for finding linear classifiers, the perceptron algorithm, has this character.

CHAPTER 2

# Regression

*Regression* is an important machine-learning problem that provides a good starting point for diving deeply into the field.

"Regression," in common parlance, means moving backwards. But this is forward progress!

## 2.1 Problem formulation

A *hypothesis* $h$ is employed as a model for solving the regression problem, in that it maps inputs $x$ to outputs $y$,

$$x \rightarrow \boxed{h} \rightarrow y \quad ,$$

where $x \in \mathbb{R}^d$ (i.e., a length $d$ column vector of real numbers), and $y \in \mathbb{R}$ (i.e., a real number). Real life rarely gives us vectors of real numbers; the $x$ we really want to take as input is usually something like a song, image, or person. In that case, we'll have to define a function $\varphi(x)$, whose range is $\mathbb{R}^d$, where $\varphi$ represents *features* of $x$, like a person's height or the amount of bass in a song, and then let the $h : \varphi(x) \rightarrow \mathbb{R}$. In much of the following, we'll omit explicit mention of $\varphi$ and assume that the $x^{(i)}$ are in $\mathbb{R}^d$, but you should always have in mind that some additional process was almost surely required to go from the actual input examples to their feature representation, and we'll talk a lot more about features later in the course.

Regression is a *supervised learning* problem, in which we are given a training dataset of the form

$$\mathcal{D}_n = \left\{\left(x^{(1)}, y^{(1)}\right), \dots, \left(x^{(n)}, y^{(n)}\right)\right\} \quad ,$$

which gives examples of input values $x^{(i)}$ and the output values $y^{(i)}$ that should be associated with them. Because $y$ values are real-valued, our hypotheses will have the form

$$h : \mathbb{R}^d \rightarrow \mathbb{R} \quad .$$

This is a good framework when we want to predict a numerical quantity, like height, stock value, etc., rather than to divide the inputs into discrete categories.

What makes a hypothesis useful? That it works well on *new* data; that is, that it makes good predictions on examples it hasn't seen. But we don't know exactly what data this hypothesis might be tested on when we use it in the real world. So, we have to *assume* a connection between the training data and testing data – typically, the assumption is that

My favorite analogy is to problem sets. We evaluate a student's ability to *generalize* by putting questions on the exam that were not on the homework (training set).

they (the training and testing data) are drawn independently from the same probability distribution.

To make this discussion more concrete, we have to provide a *loss function*, to say how unhappy we are when we guess an output $g$ given an input $x$ for which the desired output was $a$.

Given a training set $\mathcal{D}_n$ and a hypothesis $h$ with parameters $\Theta$, we can define the *training error* of $h$ to be the average loss on the training data:

$$\mathcal{E}_n(h;\Theta) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(h(x^{(i)};\Theta), y^{(i)}) \ , \tag{2.1}$$

The training error of $h$ gives us some idea of how well it characterizes the relationship between $x$ and $y$ values in our data, but it isn't the quantity that we *most* care about. What we most care about is *test error*:

$$\mathcal{E}(h) = \frac{1}{n'}\sum_{i=n+1}^{n+n'} \mathcal{L}(h(x^{(i)}), y^{(i)})$$

on $n'$ new examples that were not used in the process of finding the hypothesis.

For now, we will try to find a hypothesis with small training error (later, with some added criteria) and try to make some design choices so that it *generalizes well* to new data, meaning that it also has a small *test error*.

It might be worthwhile to stare at the two errors and think about what's the difference. For example, notice how $\Theta$ is no long a variable in the testing error? this is because in evaluating the testing error, the parameters will have been "picked"/"fixed" already.

## 2.2 Regression as an optimization problem

Given data, a loss function, and a hypothesis class, we need a method for finding a good hypothesis in the class. One of the most general ways to approach this problem is by framing the machine learning problem as an optimization problem. One reason for taking this approach is that there is a rich area of math and algorithms studying and developing efficient methods for solving optimization problems, and lots of very good software implementations of these methods. So, if we can turn our problem into one of these problems, then there will be a lot of work already done for us!

We begin by writing down an *objective function* $J(\Theta)$, where $\Theta$ stands for *all* the parameters in our model (i.e., all possible choices over parameters). We often write $J(\Theta;\mathcal{D})$ to make clear the dependence on the data $\mathcal{D}$.

Don't be too perturbed by the semicolon where you expected to see a comma! It's a math way of saying that we are mostly interested in this as a function of the arguments before the ";", but we should remember that there's a dependence on the stuff after it, as well.

The objective function describes how we feel about possible hypotheses $\Theta$: we will generally look for values for parameters $\Theta$ that minimize the objective function:

$$\Theta^* = \arg\min_{\Theta} J(\Theta) \ .$$

You can think about $\Theta^*$ here as "the theta that minimizes J": $\arg\min_x f(x)$ means the value of $x$ for which $f(x)$ is the smallest. Sometimes we write $\arg\min_{x\in\mathcal{X}} f(x)$ when we want to explicitly specify the set $\mathcal{X}$ of values of $x$ over which we want to minimize.

A very common form for a machine-learning objective is

$$J(\Theta) = \left(\frac{1}{n}\sum_{i=1}^{n} \underbrace{\mathcal{L}(h(x^{(i)};\Theta), y^{(i)})}_{\text{loss}}\right) + \underbrace{\lambda}_{\text{non-negative constant}} R(\Theta). \tag{2.2}$$

The *loss* tells us how unhappy we are about the prediction $h(x^{(i)};\Theta)$ that $\Theta$ makes for $(x^{(i)}, y^{(i)})$. Minimizing this loss makes the prediction better. The *regularizer* is an additional term that encourages the prediction to remain general, and the constant $\lambda$ adjusts the balance between reproducing seen examples, and being able to generalize to unseen examples. We will return to discuss this balance, and more about the idea of regularization, in Section 2.6.

## 2.3 Linear regression

To make this discussion more concrete, we have to provide a hypothesis class and a loss function.

We will begin by picking a class of hypotheses $\mathcal{H}$ that we think might provide a good set of possible models of the relationship between $x$ and $y$ in our data. We will start with a very simple class of *linear* hypotheses for regression. It is both simple to study and very powerful, and will serve as the basis for many other important techniques (even neural networks!).

In linear regression, the set $\mathcal{H}$ of hypotheses has the form

$$h(x; \theta, \theta_0) = \theta^T x + \theta_0 \ \ , \tag{2.3}$$

with model parameters $\Theta = (\theta, \theta_0)$. In one dimension ($d = 1$) this has the same familiar slope-intercept form as $y = mx+b$; in higher dimensions, this model describes the so-called hyperplanes.

We define a *loss function* to describe how to evaluate the quality of the predictions our hypothesis is making, when compared to the "target" $y$ values in the data set. The choice of loss function is part of modeling your domain. In the absence of additional information about a regression problem, we typically use *squared loss*:

$$\mathcal{L}(g, a) = (g - a)^2 \ \ .$$

where $g = h(x)$ is our "guess" from the hypothesis, and $a$ is the "actual" observation (in other words, here $a$ is being used equivalently as $y$). With this choice of squared loss, the average loss as generally defined in 2.1 will become the so-called *mean squared error (MSE),* which we'll study closely very soon.

The squared loss penalizes guesses that are too high the same amount as it penalizes guesses that are too low, and has a good mathematical justification in the case that your data are generated from an underlying linear hypothesis with the so-called Gaussian-distributed noise added to the $y$ values. But there are applications in which other losses would be better, and much of the framework we discuss can be applied to different loss functions, although this one has a form that also makes it particularly computationally convenient.

We won't get into the details of Gaussian distribution in our class; but it's one of the most important distributions and well-worth studying closely at some point. One obvious fact about Gaussian is that it's symmetric; this is in fact one of the reasons squared loss works well under Gaussian settings, as the loss is also symmetric.

Our objective in linear regression will be to find a hyperplane that goes as close as possible, on average, to all of our training data.

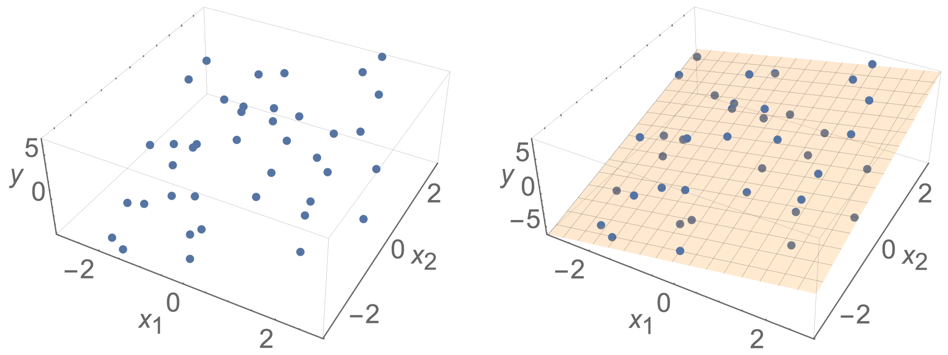
Applying the general optimization framework to the linear regression hypothesis class of Eq. 2.3 with squared loss and no regularization, our objective is to find values for $\Theta = (\theta, \theta_0)$ that minimize the MSE:

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^{n} \left( \theta^T x^{(i)} + \theta_0 - y^{(i)} \right)^2 \ \ , \tag{2.4}$$

resulting in the solution:

$$\theta^*, \theta_0^* = \arg\min_{\theta, \theta_0} J(\theta, \theta_0) \ \ . \tag{2.5}$$

For one-dimensional data ($d = 1$), this becomes the familiar problem of fitting a line to data. For $d > 1$, this hypothesis may be visualized as a $d$-dimensional hyperplane embedded in a $(d + 1)$-dimensional space (that consists of the input dimension and the $y$ dimension). For example, in the left plot below, we can see data points with labels $y$ and input dimensions $x_1$ and $x_2$. In the right plot below, we see the result of fitting these points with a two-dimensional plane that resides in three dimensions. We interpret the plane as representing a function that provides a $y$ value for any input $(x_1, x_2)$.



A richer class of hypotheses can be obtained by performing a non-linear feature transformation before doing the regression, as we will later see (in Chapter 5), but it will still end up that we have to solve a linear regression problem.

## 2.4 A gloriously simple linear regression algorithm

Okay! Given the objective in Eq. 2.4, how can we find good values of $\theta$ and $\theta_0$? We'll study several general-purpose, efficient, interesting algorithms. But before we do that, let's start with the simplest one we can think of: *guess a whole bunch* ($k$) *of different values of* $\theta$ *and* $\theta_0$, see which one has the smallest error on the training set, and return it.

```
RANDOM-REGRESSION($\mathcal{D}, k$)
For $i$ in $1 \dots k$: Randomly generate hypothesis $\theta^{(i)}, \theta_0^{(i)}$
Let $i = \arg\min_i J(\theta^{(i)}, \theta_0^{(i)}; \mathcal{D})$
Return $\theta^{(i)}, \theta_0^{(i)}$
```

This seems kind of silly, but it's a learning algorithm, and it's not completely useless.

**Study Question:** If your data set has $n$ data points, and the dimension of the $x$ values is $d$, what is the size of an individual $\theta^{(i)}$?

**Study Question:** How do you think increasing the number of guesses $k$ will change the training error of the resulting hypothesis?

## 2.5 Analytical solution: ordinary least squares

One very interesting aspect of the problem of finding a linear hypothesis that minimizes mean squared error is that we can find a closed-form formula for the answer! This general problem is often called the *ordinary least squares* (OLS)

What does "closed form" mean? Generally, that it involves direct evaluation of a mathematical expression using a fixed number of "typical" operations (like arithmetic operations, trig functions, powers, etc.). So equation 2.5 is not in closed form, because it's not at all clear what operations one needs to perform to find the solution.

Everything is easier to deal with if we assume that all of the the $x^{(i)}$ have been augmented with an extra input dimension (feature) that always has value 1, so that they are in $d+1$ dimensions, and rather than having an explicit $\theta_0$, we let it be the last element of our $\theta$ vector, so that we have, simply,

$$y = \theta^T x \ .$$

In this case, the objective becomes

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} - y^{(i)}\right)^2 \quad . \tag{2.6}$$

**Study Question:** Stop and prove to yourself that adding that extra feature with value 1 to every input vector and getting rid of the $\theta_0$ parameter is equivalent to our original model.

We approach this just like a minimization problem from calculus homework: take the derivative of J with respect to $\theta$, set it to zero, and solve for $\theta$. There are additional steps required, to check that the resulting $\theta$ is a minimum (rather than a maximum or an inflection point) but we won't work through that here. It is possible to approach this problem by:

- Finding $\partial J/\partial\theta_k$ for $k$ in $1, \dots, d$,
- Constructing a set of $k$ equations of the form $\partial J/\partial\theta_k = 0$, and
- Solving the system for values of $\theta_k$.

We will use $d$ here for the total number of features in each $x^{(i)}$, including the added 1.

That works just fine. To get practice for applying techniques like this to more complex problems, we will work through a more compact (and cool!) matrix view. Along the way, it will be helpful to collect all of the derivatives in one vector. In particular, the gradient of J with respect to $\theta$ is following column vector of length d:

$$\nabla_\theta J = \begin{bmatrix} \partial J/\partial\theta_1 \\ \vdots \\ \partial J/\partial\theta_d \end{bmatrix} .$$

**Study Question:** Work through the next steps and check your answer against ours below.

We can think of our training data in terms of matrices X and Y, where each column of X is an example, and each "column" of Y is the corresponding target output value:

$$X = \begin{bmatrix} x_1^{(1)} & \dots & x_1^{(n)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \dots & x_d^{(n)} \end{bmatrix} \quad Y = \begin{bmatrix} y^{(1)} & \dots & y^{(n)} \end{bmatrix} \quad .$$

**Study Question:** What are the dimensions of X and Y?

In most textbooks, they think of an individual example $x^{(i)}$ as a row, rather than a column. So that we get an answer that will be recognizable to you, we are going to define a new matrix and vector, $\tilde{X}$ and $\tilde{Y}$, which are just transposes of our X and Y, and then work with them:

$$\tilde{X} = X^T = \begin{bmatrix} x_1^{(1)} & \dots & x_d^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(n)} & \dots & x_d^{(n)} \end{bmatrix} \quad \tilde{Y} = Y^T = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{bmatrix} \quad .$$

**Study Question:** What are the dimensions of $\tilde{X}$ and $\tilde{Y}$?

Now we can write

$$J(\theta) = \frac{1}{n}\underbrace{(\tilde{X}\theta - \tilde{Y})^{T}}_{1\times n}\underbrace{(\tilde{X}\theta - \tilde{Y})}_{n\times 1} = \frac{1}{n}\sum_{i=1}^{n}\left(\left(\sum_{j=1}^{d}\tilde{X}_{ij}\theta_j\right) - \tilde{Y}_i\right)^2$$

and using facts about matrix/vector calculus, we get

$$\nabla_\theta J = \frac{2}{n}\underbrace{\tilde{X}^{T}}_{d\times n}\underbrace{(\tilde{X}\theta - \tilde{Y})}_{n\times 1} \quad . \tag{2.7}$$

You should be able to verify this by doing a simple (say, 2 × 2) example by hand.

See Appendix A for a nice way to think about finding this derivative.

Setting $\nabla_\theta J$ to 0 and solving, we get:

$$\begin{aligned}
\frac{2}{n}\tilde{X}^{T}(\tilde{X}\theta - \tilde{Y}) &= 0\\
\tilde{X}^{T}\tilde{X}\theta - \tilde{X}^{T}\tilde{Y} &= 0\\
\tilde{X}^{T}\tilde{X}\theta &= \tilde{X}^{T}\tilde{Y}\\
\theta &= (\tilde{X}^{T}\tilde{X})^{-1}\tilde{X}^{T}\tilde{Y}
\end{aligned}$$

And the dimensions work out!

$$\theta = \underbrace{\left(\tilde{X}^{T}\tilde{X}\right)^{-1}}_{d\times d}\underbrace{\tilde{X}^{T}}_{d\times n}\underbrace{\tilde{Y}}_{n\times 1}$$

So, given our data, we can directly compute the linear regression that minimizes mean squared error. That's pretty awesome!

## 2.6 Regularization

The objective function of Eq. 2.2 balances (training-data) memorization, induced by the *loss* term, with generalization, induced by the *regularization* term. Here, we address the need for regularization specifically for linear regression, and show how this can be realized using one popular regularization technique called *ridge regression*.

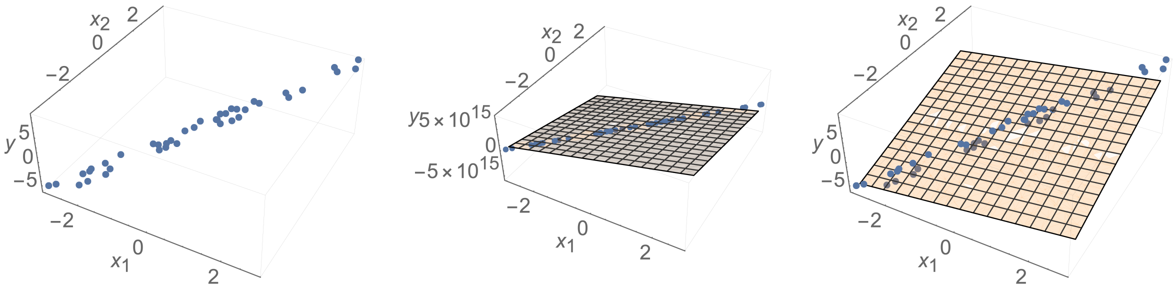
### 2.6.1 Regularization and linear regression

If all we cared about was finding a hypothesis with small loss on the training data, we would have no need for regularization, and could simply omit the second term in the objective. But remember that our ultimate goal is to *perform well on input values that we haven't trained on!* It may seem that this is an impossible task, but humans and machine-learning methods do this successfully all the time. What allows *generalization* to new input values is a belief that there is an underlying regularity that governs both the training and testing data. One way to describe an assumption about such a regularity is by choosing a limited class of possible hypotheses. Another way to do this is to provide smoother guidance, saying that, within a hypothesis class, we prefer some hypotheses to others. The regularizer articulates this preference and the constant $\lambda$ says how much we are willing to trade off loss on the training data versus preference over hypotheses.

For example, consider what happens when $d = 2$, and $x_2$ is highly correlated with $x_1$, meaning that the data look like a line, as shown in the left panel of the figure below. Thus, there isn't a unique best hyperplane. Such correlations happen often in real-life data,

Sometimes there's technically a unique best hyperplane, but just because of noise.

because of underlying common causes; for example, across a population, the height of people may depend on both age and amount of food intake in the same way. This is especially the case when there are many feature dimensions used in the regression. Mathematically, this leads to $\tilde{X}^T\tilde{X}$ close to singularity, such that $(\tilde{X}^T\tilde{X})^{-1}$ is undefined or has huge values, resulting in unstable models (see the middle panel of figure and note the range of the $y$ values—the slope is huge!):



A common strategy for specifying a *regularizer* is to use the form

$$R(\Theta) = \left\|\Theta - \Theta_{prior}\right\|^2$$

when we have some idea in advance that $\Theta$ ought to be near some value $\Theta_{prior}$.

Learn about Bayesian methods in machine learning to see the theory behind this and cool results!

Here, the notion of distance is quantified by squaring the $l_2$ *norm* of the parameter vector: for any d-dimensional vector $v \in \mathbb{R}^d$, the $l_2$ *norm* of $v$ is defined as,

$$\|v\| = \sqrt{\sum_{i=1}^{d} |v_i|^2} \ .$$

In the absence of such knowledge a default is to *regularize toward zero*:

$$R(\Theta) = \|\Theta\|^2 \ .$$

When this is done in the example depicted above, the regression model becomes stable, producing the result shown in the right-hand panel in the figure. Now the slope is much more sensible.

### 2.6.2 Ridge regression

There are some kinds of trouble we can get into in regression problems. What if $\left(\tilde{X}^T\tilde{X}\right)$ is not invertible?

**Study Question:** Consider, for example, a situation where the data-set is just the same point repeated twice: $x^{(1)} = x^{(2)} = [1 \ \ 2]^T$. What is $\tilde{X}$ in this case? What is $\tilde{X}^T\tilde{X}$? What is $(\tilde{X}^T\tilde{X})^{-1}$?

Another kind of problem is *overfitting*: we have formulated an objective that is just about fitting the data as well as possible, but we might also want to *regularize* to keep the hypothesis from getting *too* attached to the data.

We address both the problem of not being able to invert $(\tilde{X}^T\tilde{X})^{-1}$ and the problem of overfitting using a mechanism called *ridge regression*. We add a regularization term $\|\theta\|^2$ to the OLS objective, with a non-negative scalar value $\lambda$ to control the tradeoff between the training error and the regularization term.

**Study Question:** Why do we emphasize the non-negativity of the scalar $\lambda$? When we add a regularizer of the form $\|\theta\|^2$, what is our most "preferred" value of $\theta$, in the absence of any data?

Here is the ridge regression objective function:

$$J_{\text{ridge}}(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} + \theta_0 - y^{(i)}\right)^2 + \lambda\|\theta\|^2$$

Larger $\lambda$ values pressure $\theta$ values to be near zero. Note that we don't penalize $\theta_0$; intuitively, $\theta_0$ is what "floats" the regression surface to the right level for the data you have, and so you shouldn't make it harder to fit a data set where the $y$ values tend to be around one million than one where they tend to be around one. The other parameters control the orientation of the regression surface, and we prefer it to have a not-too-crazy orientation.

There is an analytical expression for the $\theta, \theta_0$ values that minimize $J_{\text{ridge}}$, but it's a little bit more complicated to derive than the solution for OLS because $\theta_0$ needs special treatment. If we decide not to treat $\theta_0$ specially (so we add a 1 feature to our input vectors as discussed above), then we get:

$$\nabla_\theta J_{\text{ridge}} = \frac{2}{n}\tilde{X}^T(\tilde{X}\theta - \tilde{Y}) + 2\lambda\theta \ .$$

Setting to 0 and solving, we get:

Remember that I stands for the identity matrix, a square matrix that has 1's along the diagonal and 0's everywhere else.

$$\begin{aligned}
\frac{2}{n}\tilde{X}^T(\tilde{X}\theta - \tilde{Y}) + 2\lambda\theta &= 0 \\
\frac{1}{n}\tilde{X}^T\tilde{X}\theta - \frac{1}{n}\tilde{X}^T\tilde{Y} + \lambda\theta &= 0 \\
\frac{1}{n}\tilde{X}^T\tilde{X}\theta + \lambda\theta &= \frac{1}{n}\tilde{X}^T\tilde{Y} \\
\tilde{X}^T\tilde{X}\theta + n\lambda\theta &= \tilde{X}^T\tilde{Y} \\
(\tilde{X}^T\tilde{X} + n\lambda I)\theta &= \tilde{X}^T\tilde{Y} \\
\theta &= (\tilde{X}^T\tilde{X} + n\lambda I)^{-1}\tilde{X}^T\tilde{Y}
\end{aligned}$$

Whew! So the solution is:

$$\theta_{\text{ridge}} = \left(\tilde{X}^T\tilde{X} + n\lambda I\right)^{-1}\tilde{X}^T\tilde{Y} \tag{2.8}$$

Now, why is the term $\left(\tilde{X}^T\tilde{X} + n\lambda I\right)$ invertible? Explaining this requires some linear algebra. The matrix $\tilde{X}^T\tilde{X}$ is positive semidefinite, which implies that its eigenvalues $\{\gamma_i\}_i$ are greater than or equal to 0. The matrix $\tilde{X}^T\tilde{X} + n\lambda I$ has eigenvalues $\{\gamma_i + n\lambda\}_i$ which are guaranteed to be strictly positive since $\lambda > 0$. Recalling that the determinant of a matrix is simply the product of its eigenvalues, we get that $\det(\tilde{X}^T\tilde{X} + n\lambda I) > 0$ and conclude that $\tilde{X}^T\tilde{X} + n\lambda I$ is invertible.

This is called "ridge" regression because we are adding a "ridge" of $n\lambda$ values along the diagonal of the matrix before inverting it.

**Study Question:** Why is $\tilde{X}^T\tilde{X}$ positive semidefinite?

**Study Question:** What is the dimension of I in the equation above?

## 2.7 Evaluating learning algorithms

In this section, we will explore how to evaluate supervised machine-learning algorithms. We will study the special case of applying them to regression problems, but the basic ideas of validation, hyper-parameter selection, and cross-validation apply much more broadly.

We have seen how linear regression is a well-formed optimization problem, which has an analytical solution when ridge regularization is applied. But how can one choose the best amount of regularization, as parameterized by $\lambda$? Two key ideas involve the evaluation of the performance of a hypothesis, and a separate evaluation of the algorithm used to produce hypotheses, as described below.

### 2.7.1 Evaluating hypotheses

The performance of a given hypothesis $h$ may be evaluated by measuring *test error* on data that was not used to train it. Given a training set $\mathcal{D}_n$, a regression hypothesis $h$, and if we choose squared loss, we can define the OLS *training error* of $h$ to be the mean square error between its predictions and the expected outputs:

$$\mathcal{E}_n(h) = \frac{1}{n}\sum_{i=1}^{n}\left[h(x^{(i)}) - y^{(i)}\right]^2 \quad .$$

It's a bit funny to interpret the analytical formulas given above for $\theta$ as "training," but later when we employ more statistical methods, "training" will be a meaningful concept.

Test error captures the performance of $h$ on unseen data, and is the mean square error on the test set, with a nearly identical expression as that above, differing only in the range of index $i$:

$$\mathcal{E}(h) = \frac{1}{n'}\sum_{i=n+1}^{n+n'}\left[h(x^{(i)}) - y^{(i)}\right]^2$$

on $n'$ new examples that were not used in the process of constructing $h$.

In machine learning in general, not just regression, it is useful to distinguish two ways in which a hypothesis $h \in \mathcal{H}$ might contribute to test error. Two are:

**Structural error**: This is error that arises because there is no hypothesis $h \in \mathcal{H}$ that will perform well on the data, for example because the data was really generated by a sine wave but we are trying to fit it with a line.

**Estimation error**: This is error that arises because we do not have enough data (or the data are in some way unhelpful) to allow us to choose a good $h \in \mathcal{H}$, or because we didn't solve the optimization problem well enough to find the best $h$ given the data that we had.

When we increase $\lambda$, we tend to increase structural error but decrease estimation error, and vice versa.

There are technical definitions of these concepts that are studied in more advanced treatments of machine learning. Structural error is referred to as *bias* and estimation error is referred to as *variance*.

### 2.7.2 Evaluating learning algorithms

*Note that this section is relevant to learning algorithms generally—we are just introducing the topic here since we now have an algorithm that can be evaluated!*

A *learning algorithm* is a procedure that takes a data set $\mathcal{D}_n$ as input and returns an hypothesis $h$ from a hypothesis class $\mathcal{H}$; it looks like

$$\mathcal{D}_n \longrightarrow \boxed{\text{learning alg } (\mathcal{H})} \longrightarrow h$$

Keep in mind that $h$ has parameters. The learning algorithm itself may have its own parameters, and such parameters are often called *hyperparameters*. The analytical solutions presented above for linear regression, e.g., Eq. 2.8, may be thought of as learning algorithms, where $\lambda$ is a hyperparameter that governs how the learning algorithm works and can strongly affect its performance.

How should we evaluate the performance of a learning algorithm? This can be tricky. There are many potential sources of variability in the possible result of computing test error on a learned hypothesis $h$:

- Which particular *training examples* occurred in $\mathcal{D}_n$
- Which particular *testing examples* occurred in $\mathcal{D}_{n'}$
- Randomization inside the learning *algorithm* itself

#### 2.7.2.1 Validation

Generally, to evaluate how well a learning *algorithm* works, given an unlimited data source, we would like to execute the following process multiple times:

- Train on a new training set (subset of our big data source)
- Evaluate resulting $h$ on a *validation set* that does not overlap the training set (but is still a subset of our same big data source)

Running the algorithm multiple times controls for possible poor choices of training set or unfortunate randomization inside the algorithm itself.

#### 2.7.2.2 Cross validation

One concern is that we might need a lot of data to do this, and in many applications data is expensive or difficult to acquire. We can re-use data with *cross validation* (but it's harder to do theoretical analysis).

```
CROSS-VALIDATE($\mathcal{D}, k$)
divide $\mathcal{D}$ into $k$ chunks $\mathcal{D}_1, \mathcal{D}_2, \dots \mathcal{D}_k$ (of roughly equal size)
for $i = 1$ to $k$
    train $h_i$ on $\mathcal{D} \setminus \mathcal{D}_i$ (withholding chunk $\mathcal{D}_i$ as the validation set)
    compute "test" error $\mathcal{E}_i(h_i)$ on withheld data $\mathcal{D}_i$
return $\frac{1}{k}\sum_{i=1}^{k} \mathcal{E}_i(h_i)$
```

It's very important to understand that (cross-)validation neither delivers nor evaluates a single particular hypothesis $h$. It evaluates the *learning algorithm* that produces hypotheses.

#### 2.7.2.3 Hyperparameter tuning

The hyper-parameters of a learning algorithm affect how the algorithm *works* but they are not part of the resulting hypothesis. So, for example, $\lambda$ in ridge regression affects *which* hypothesis will be returned, but $\lambda$ itself doesn't show up in the hypothesis (the hypothesis is specified using parameters $\theta$ and $\theta_0$).

You can think about each different setting of a hyper-parameter as specifying a different learning algorithm.

In order to pick a good value of the hyper-parameter, we often end up just trying a lot of values and seeing which one works best via validation or cross-validation.

**Study Question:** How could you use cross-validation to decide whether to use analytic ridge regression or our random-regression algorithm and to pick $K$ for random regression or $\lambda$ for ridge regression?

# CHAPTER 3

# Gradient Descent

In the previous chapter, we showed how to describe an interesting objective function for machine learning, but we need a way to find the optimal $\Theta^* = \arg\min_{\Theta} J(\Theta)$, particularly when the objective function is not amenable to analytical optimization. For example, this can be the case when $J(\Theta)$ involves a more complex loss function, or more general forms of regularization. It can also be the case when there are simply too many parameters to learn for it to be computationally feasible.

There is an enormous and fascinating literature on the mathematical and algorithmic foundations of optimization, but for this class, we will consider one of the simplest methods, called *gradient descent.*

Which you should consider studying some day!

Intuitively, in one or two dimensions, we can easily think of $J(\Theta)$ as defining a surface over $\Theta$; that same idea extends to higher dimensions. Now, our objective is to find the $\Theta$ value at the lowest point on that surface. One way to think about gradient descent is that you start at some arbitrary point on the surface, look to see in which direction the "hill" goes down most steeply, take a small step in that direction, determine the direction of steepest descent from where you are, take another small step, etc.

Below, we explicitly give gradient descent algorithms for one and multidimensional objective functions (Sections 3.1 and 3.2). We then illustrate the application of gradient descent to a loss function which is not merely mean squared loss (Section 3.3). And we present an important method known as *stochastic gradient descent* (Section 3.4), which is especially useful when datasets are too large for descent in a single batch, and has some important behaviors of its own.

## 3.1 Gradient descent in one dimension

We start by considering gradient descent in one dimension. Assume $\Theta \in \mathbb{R}$, and that we know both $J(\Theta)$ and its first derivative with respect to $\Theta$, $J'(\Theta)$. Here is pseudo-code for gradient descent on an arbitrary function $f$. Along with $f$ and its gradient $\nabla_{\Theta} f$ (which, in the case of a scalar $\Theta$, is the same as its derivative $f'$), we have to specify some hyper-parameters. These hyper-parameters include the initial value for parameter $\Theta$, a *step-size* hyper-parameter $\eta$, and an *accuracy* hyper-parameter $\epsilon$.

The hyper-parameter $\eta$ is often called *learning rate* when gradient descent is applied in machine learning. For simplicity, $\eta$ may be taken as a constant, as is the case in the pseudo-code below; and we'll see adaptive (non-constant) step-sizes soon. What's important to notice though, is that even when $\eta$ is constant, the actual magnitude of the change to $\Theta$ may not be constant, as that change depends on the magnitude of the gradient itself too.

```
1D-GRADIENT-DESCENT($\Theta_{init}, \eta, f, f', \epsilon$)
$\Theta^{(0)} = \Theta_{init}$
$t = 0$
repeat
    $t = t + 1$
    $\Theta^{(t)} = \Theta^{(t-1)} - \eta\, f'(\Theta^{(t-1)})$
until $|f'(\Theta^{(t)})| < \epsilon$
return $\Theta^{(t)}$
```

Note that this algorithm terminates when the derivative of the function $f$ is sufficiently small. There are many other reasonable ways to decide to terminate, including:

- Stop after a fixed number of iterations $T$, i.e., when $t = T$. Practically, this is the most common choice.
- Stop when the change in the value of the parameter $\Theta$ is sufficiently small, i.e., when $\left|\Theta^{(t)} - \Theta^{(t-1)}\right| < \epsilon$.

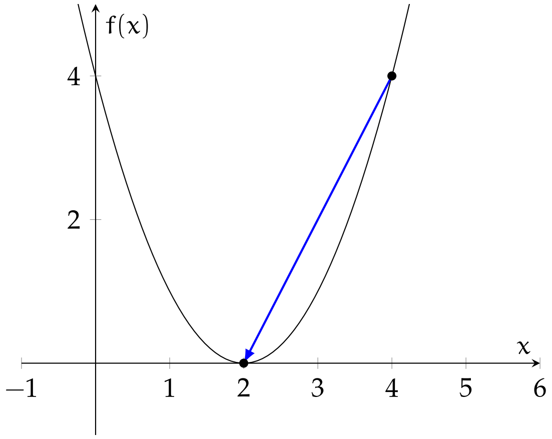
**Study Question:** Consider all of the potential stopping criteria for 1D-GRADIENT-DESCENT, both in the algorithm as it appears and listed separately later. Can you think of ways that any two of the criteria relate to each other?

**Theorem 3.1.1.** *Choose any small distance $\tilde{\epsilon} > 0$. If we assume that $f$ has a minimum, is sufficiently "smooth" and convex, and if the learning rate $\eta$ is sufficiently small, gradient descent will reach a point within $\tilde{\epsilon}$ of a global optimum point $\Theta$.*

A function is convex if the line segment between any two points on the graph of the function lies above or on the graph.

However, we must be careful when choosing the learning rate to prevent slow convergence, non-converging oscillation around the minimum, or divergence.

The following plot illustrates a convex function $f(x) = (x-2)^2$, starting gradient descent at $x_{\text{init}} = 4.0$ with a step-size of $1/2$. It is very well-behaved!



**Study Question:** What happens in this example with very small $\eta$? With very big $\eta$?

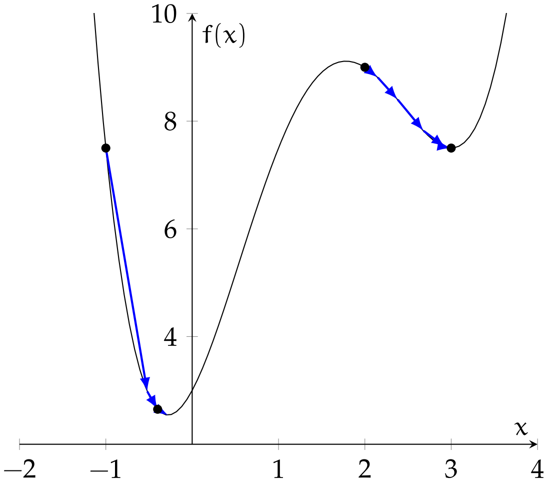
If $f$ is non-convex, where gradient descent converges to depends on $x_{init}$. First, let's establish some definitions. Let $f$ be a real-valued function defined over some domain $D$. A point $x_0 \in D$ is called a *global minimum point* of $f$ if $f(x_0) \leqslant f(x)$ for all other $x \in D$. A point

$x_0 \in D$ is instead called a *local minimum point* of a function $f$ if there exists some constant $\epsilon > 0$ such that for all $x$ within the interval defined by $d(x, x_0) < \epsilon$, $f(x_0) \leqslant f(x)$, where $d$ is some distance metric, e.g., $d(x, x_0) = \|x - x_0\|$. A global minimum point is also a local minimum point, but a local minimum point does not have to be a global minimum point.

If $f$ is non-convex (and sufficiently smooth), one expects that gradient descent (run long enough with small enough learning rate) will get very close to a point at which the gradient is zero, though we cannot guarantee that it will converge to a global minimum point.

There are two notable exceptions to this common sense expectation: First, gradient descent can get stagnated while approaching a point $x$ which is not a local minimum or maximum, but satisfies $f'(x) = 0$. For example, for $f(x) = x^3$, starting gradient descent from the initial guess $x_{init} = 1$, while using learning rate $\eta < 1/3$ will lead to $x^{(k)}$ converging to zero as $k \to \infty$. Second, there are functions (even convex ones) with no minimum points, like $f(x) = \exp(-x)$, for which gradient descent with a positive learning rate converges to $+\infty$.

The plot below shows two different $x_{init}$, and how gradient descent started from each point heads toward two different local optimum points.



## 3.2 Multiple dimensions

The extension to the case of multi-dimensional $\Theta$ is straightforward. Let's assume $\Theta \in \mathbb{R}^m$, so $f : \mathbb{R}^m \to \mathbb{R}$. The gradient of $f$ with respect to $\Theta$ is

$$\nabla_\Theta f = \begin{bmatrix} \partial f / \partial \Theta_1 \\ \vdots \\ \partial f / \partial \Theta_m \end{bmatrix}$$

The algorithm remains the same, except that the update step in line 5 becomes

$$\Theta^{(t)} = \Theta^{(t-1)} - \eta \nabla_\Theta f(\Theta^{(t-1)})$$

and any termination criteria that depended on the dimensionality of $\Theta$ would have to change. The easiest thing is to keep the test in line 6 as $\left|f(\Theta^{(t)}) - f(\Theta^{(t-1)})\right| < \epsilon$, which is sensible no matter the dimensionality of $\Theta$.

**Study Question:** Which termination criteria from the 1D case were defined in a way that assumes $\Theta$ is one dimensional?

## 3.3 Application to regression

Recall from the previous chapter that choosing a loss function is the first step in formulating a machine-learning problem as an optimization problem, and for regression we studied the mean square loss, which captures loss as $(\text{guess} - \text{actual})^2$. This leads to the *ordinary least squares* objective

$$J(\theta) = \frac{1}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} - y^{(i)}\right)^2 \quad . \tag{3.1}$$

We use the gradient of the objective with respect to the parameters,

$$\nabla_\theta J = \frac{2}{n}\underbrace{\tilde{X}^T}_{d\times n}\underbrace{(\tilde{X}\theta - \tilde{Y})}_{n\times 1} \ , \tag{3.2}$$

to obtain an analytical solution to the linear regression problem. Gradient descent could also be applied to numerically compute a solution, using the update rule

$$\theta^{(t)} = \theta^{(t-1)} - \eta\frac{2}{n}\sum_{i=1}^{n}\left(\left[\theta^{(t-1)}\right]^T x^{(i)} - y^{(i)}\right)x^{(i)} \,. \tag{3.3}$$

Beware double superscripts! $[\theta]^T$ is the transpose of the vector $\theta$.

### 3.3.1 Ridge regression

Now, let's add in the regularization term, to get the ridge-regression objective:

$$J_{\text{ridge}}(\theta,\theta_0) = \frac{1}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} + \theta_0 - y^{(i)}\right)^2 + \lambda\|\theta\|^2 \quad .$$

Recall that in ordinary least squares, we finessed handling $\theta_0$ by adding an extra dimension of all 1's. In ridge regression, we really do need to separate the parameter vector $\theta$ from the offset $\theta_0$, and so, from the perspective of our general-purpose gradient descent method, our whole parameter set $\Theta$ is defined to be $\Theta = (\theta, \theta_0)$. We will go ahead and find the gradients separately for each one:

Some passing familiarity with matrix derivatives is helpful here. A foolproof way of computing them is to compute partial derivative of J with respect to each component $\theta_i$ of $\theta$. See Appendix A on matrix derivatives!

$$\begin{aligned}\nabla_\theta J_{\text{ridge}}(\theta,\theta_0) &= \frac{2}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} + \theta_0 - y^{(i)}\right)x^{(i)} + 2\lambda\theta \\ \frac{\partial J_{\text{ridge}}(\theta,\theta_0)}{\partial\theta_0} &= \frac{2}{n}\sum_{i=1}^{n}\left(\theta^T x^{(i)} + \theta_0 - y^{(i)}\right) \quad .\end{aligned}$$

Note that $\nabla_\theta J_{\text{ridge}}$ will be of shape $d \times 1$ and $\partial J_{\text{ridge}}/\partial\theta_0$ will be a scalar since we have separated $\theta_0$ from $\theta$ here.

**Study Question:** Convince yourself that the dimensions of all these quantities are correct, under the assumption that $\theta$ is $d \times 1$. How does $d$ relate to $m$ as discussed for $\Theta$ in the previous section?

**Study Question:** Compute $\nabla_\theta \|\theta\|^2$ by finding the vector of partial derivatives $(\partial \|\theta\|^2/\partial\theta_1, \ldots, \partial \|\theta\|^2/\partial\theta_d)$. What is the shape of $\nabla_\theta \|\theta\|^2$?

**Study Question:** Compute $\nabla_\theta J_{\text{ridge}}(\theta^T x+\theta_0, y)$ by finding the vector of partial derivatives $(\partial J_{\text{ridge}}(\theta^T x + \theta_0, y)/\partial\theta_1, \ldots, \partial J_{\text{ridge}}(\theta^T x + \theta_0, y)/\partial\theta_d)$.

**Study Question:** Use these last two results to verify our derivation above.

Putting everything together, our gradient descent algorithm for ridge regression becomes

```
RR-GRADIENT-DESCENT($\theta_{init}, \theta_{0init}, \eta, \epsilon$)
1  $\theta^{(0)} = \theta_{init}$
2  $\theta_0^{(0)} = \theta_{0init}$
3  $t = 0$
4  repeat
5      $t = t + 1$
6      $\theta^{(t)} = \theta^{(t-1)} - \eta\left(\frac{1}{n}\sum_{i=1}^{n}\left(\theta^{(t-1)^T}x^{(i)} + \theta_0^{(t-1)} - y^{(i)}\right)x^{(i)} + \lambda\theta^{(t-1)}\right)$
7      $\theta_0^{(t)} = \theta_0^{(t-1)} - \eta\left(\frac{1}{n}\sum_{i=1}^{n}\left(\theta^{(t-1)^T}x^{(i)} + \theta_0^{(t-1)} - y^{(i)}\right)\right)$
8  until $\left|J_{\text{ridge}}(\theta^{(t)}, \theta_0^{(t)}) - J_{\text{ridge}}(\theta^{(t-1)}, \theta_0^{(t-1)})\right| < \epsilon$
9  return $\theta^{(t)}, \theta_0^{(t)}$
```

**Study Question:** Is it okay that $\lambda$ doesn't appear in line 7?

**Study Question:** Is it okay that the 2's from the gradient definitions don't appear in the algorithm?

## 3.4 Stochastic gradient descent

When the form of the gradient is a sum, rather than take one big(ish) step in the direction of the gradient, we can, instead, randomly select one term of the sum, and take a very small step in that direction. This seems sort of crazy, but remember that all the little steps would average out to the same direction as the big step if you were to stay in one place. Of course, you're not staying in that place, so you move, in expectation, in the direction of the gradient.

The word "stochastic" means probabilistic, or random; so does "aleatoric," which is a very cool word. Look up aleatoric music, sometime.

Most objective functions in machine learning can end up being written as a sum over data points, in which case, stochastic gradient descent (SGD) is implemented by picking a data point randomly out of the data set, computing the gradient as if there were only that one point in the data set, and taking a small step in the negative direction.

Let's assume our objective has the form

$$f(\Theta) = \sum_{i=1}^{n} f_i(\Theta) \ ,$$

where $n$ is the number of data points used in the objective (and this may be different from the number of points available in the whole data set). Here is pseudocode for applying SGD to such an objective $f$; it assumes we know the form of $\nabla_\Theta f_i$ for all $i$ in $1 \ldots n$:

```
STOCHASTIC-GRADIENT-DESCENT($\Theta_{init}, \eta, f, \nabla_\Theta f_1, \ldots, \nabla_\Theta f_n, T$)
$\Theta^{(0)} = \Theta_{init}$
for $t = 1$ to $T$
    randomly select $i \in \{1, 2, \ldots, n\}$
    $\Theta^{(t)} = \Theta^{(t-1)} - \eta(t)\,\nabla_\Theta f_i(\Theta^{(t-1)})$
return $\Theta^{(t)}$
```

Note that now instead of a fixed value of $\eta$, $\eta$ is indexed by the iteration of the algorithm, $t$. Choosing a good stopping criterion can be a little trickier for SGD than traditional gradient descent. Here we've just chosen to stop after a fixed number of iterations $T$.

For SGD to converge to a local optimum point as $t$ increases, the learning rate has to decrease as a function of time. The next result shows one learning rate sequence that works.

**Theorem 3.4.1.** *If $f$ is convex, and $\eta(t)$ is a sequence satisfying*

$$\sum_{t=1}^{\infty} \eta(t) = \infty \ \textit{and} \ \sum_{t=1}^{\infty} \eta(t)^2 < \infty \ ,$$

*then SGD converges* with probability one *to the optimal $\Theta$.*

We have left out some gnarly conditions in this theorem. Also, you can learn more about the subtle difference between "with probability one" and "always" by taking an advanced probability course.

Why these two conditions? The intuition is that the first condition, on $\sum \eta(t)$, is needed to allow for the possibility of an unbounded potential range of exploration, while the second condition, on $\sum \eta(t)^2$, ensures that the learning rates get smaller and smaller as $t$ increases.

One "legal" way of setting the learning rate is to make $\eta(t) = 1/t$ but people often use rules that decrease more slowly, and so don't strictly satisfy the criteria for convergence.

**Study Question:** If you start a long way from the optimum, would making $\eta(t)$ decrease more slowly tend to make you move more quickly or more slowly to the optimum?

There are multiple intuitions for why SGD might be a better choice algorithmically than regular GD (which is sometimes called *batch* GD (BGD)):

- BGD typically requires computing some quantity over every data point in a data set. SGD may perform well after visiting only some of the data. This behavior can be useful for very large data sets – in runtime and memory savings.

- If your $f$ is actually non-convex, but has many shallow local optimum points that might trap BGD, then taking *samples* from the gradient at some point $\Theta$ might "bounce" you around the landscape and away from the local optimum points.

- Sometimes, optimizing $f$ really well is not what we want to do, because it might overfit the training set; so, in fact, although SGD might not get lower training error than BGD, it might result in lower test error.

CHAPTER 4

# Classification

## 4.1 Classification

Classification is a machine learning problem seeking to map from inputs $\mathbb{R}^d$ to outputs in an unordered set. Examples of classification output sets could be $\{\text{apples}, \text{oranges}, \text{pears}\}$ if we're trying to figure out what type of fruit we have, or $\{\text{heartattack}, \text{noheartattack}\}$ if we're working in an emergency room and trying to give the best medical care to a new patient. We focus on an essential simple case, *binary classification*, where we aim to find a mapping from $\mathbb{R}^d$ to two outputs. While we should think of the outputs as not having an order, it's often convenient to encode them as $\{-1, +1\}$. As before, let the letter $h$ (for hypothesis) represent a classifier, so the classification process looks like:

in contrast to a continuous real-valued output, as we saw for linear regression

$$x \rightarrow \boxed{h} \rightarrow y \quad .$$

Like regression, classification is a *supervised learning* problem, in which we are given a training data set of the form

$$\mathcal{D}_n = \left\{ \left( x^{(1)}, y^{(1)} \right), \ldots, \left( x^{(n)}, y^{(n)} \right) \right\} \quad .$$

We will assume that each $x^{(i)}$ is a $d \times 1$ *column vector*. The intended use of this data is that, when given an input $x^{(i)}$, the learned hypothesis should generate output $y^{(i)}$.

What makes a classifier useful? As in regression, we want it to work well on new data, making good predictions on examples it hasn't seen. But we don't know exactly what data this classifier might be tested on when we use it in the real world. So, we have to *assume* a connection between the training data and testing data; typically, they are drawn independently from the same probability distribution.

In classification, we will often use 0-1 loss for evaluation (as discussed in Section 1.3). For that choice, we can write the training error and the testing error. In particular, given a training set $\mathcal{D}_n$ and a classifier $h$, we define the *training error* of $h$ to be

$$\mathcal{E}_n(h) = \frac{1}{n} \sum_{i=1}^{n} \begin{cases} 1 & h(x^{(i)}) \neq y^{(i)} \\ 0 & \text{otherwise} \end{cases} \quad .$$

For now, we will try to find a classifier with small training error (later, with some added criteria) and hope it *generalizes well* to new data, and has a small *test error*

$$\mathcal{E}(h) = \frac{1}{n'} \sum_{i=n+1}^{n+n'} \begin{cases} 1 & h(x^{(i)}) \neq y^{(i)} \\ 0 & \text{otherwise} \end{cases}$$

on $n'$ new examples that were not used in the process of finding the classifier.

We begin by introducing the hypothesis class of *linear classifiers* (Section 4.2) and then define an optimization framework to learn *linear logistic classifiers* (Section 4.3).

## 4.2 Linear classifiers

We start with the hypothesis class of *linear classifiers*. They are (relatively) easy to understand, simple in a mathematical sense, powerful on their own, and the basis for many other more sophisticated methods. Following their definition, we present a simple learning algorithm for classifiers.

### 4.2.1 Linear classifiers: definition

A linear classifier in $d$ dimensions is defined by a vector of parameters $\theta \in \mathbb{R}^d$ and scalar $\theta_0 \in \mathbb{R}$. So, the hypothesis class $\mathcal{H}$ of linear classifiers in $d$ dimensions is parameterized by the *set* of all vectors in $\mathbb{R}^{d+1}$. We'll assume that $\theta$ is a $d \times 1$ column vector.

Given particular values for $\theta$ and $\theta_0$, the classifier is defined by

$$h(x; \theta, \theta_0) = \text{sign}(\theta^T x + \theta_0) = \begin{cases} +1 & \text{if } \theta^T x + \theta_0 > 0 \\ -1 & \text{otherwise} \end{cases} .$$

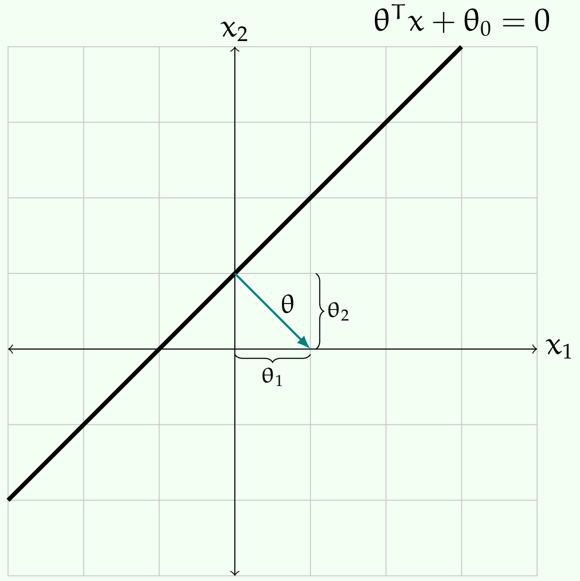
Remember that we can think of $\theta, \theta_0$ as specifying a $d$-dimensional hyperplane (compare the above with Eq. 2.3). But this time, rather than being interested in that hyperplane's values at particular points $x$, we will focus on the *separator* that it induces. The separator is the set of $x$ values such that $\theta^T x + \theta_0 = 0$. This is also a hyperplane, but in $d-1$ dimensions! We can interpret $\theta$ as a vector that is perpendicular to the separator. (We will also say that $\theta$ is *normal to* the separator.)

For example, in two dimensions ($d = 2$) the separator has dimension 1, which means it is a line, and the two components of $\theta = [\theta_1, \theta_2]^T$ give the orientation of the separator, as illustrated in the following example.

### 4.2.2 Linear classifiers: examples

**Example:** Let $h$ be the linear classifier defined by $\theta = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \theta_0 = 1$.

The diagram below shows the $\theta$ vector (in green) and the separator it defines:



What is $\theta_0$? We can solve for it by plugging a point on the line into the equation for the line. It is often convenient to choose a point on one of the axes, e.g., in this case, $x = [0, 1]^T$, for which $\theta^T \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \theta_0 = 0$, giving $\theta_0 = 1$.

In this example, the separator divides $\mathbb{R}^d$, the space our $x^{(i)}$ points live in, into two half-spaces. The one that is on the same side as the normal vector is the *positive* half-space, and we classify all points in that space as positive. The half-space on the other side is *negative* and all points in it are classified as negative.

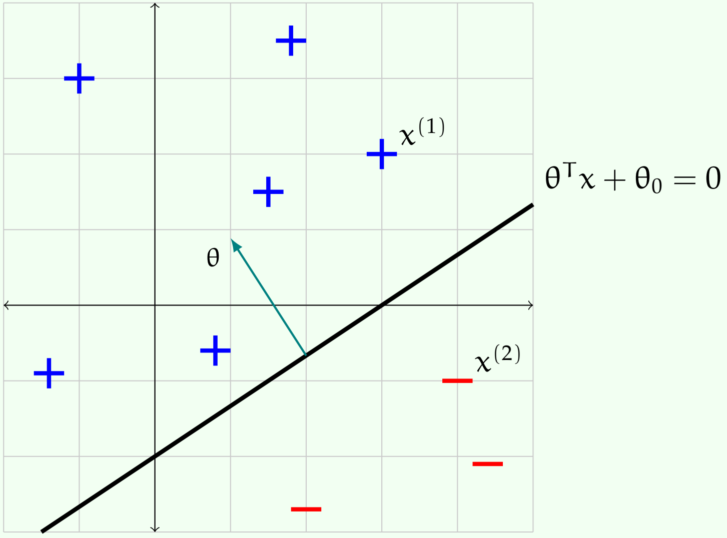
Note that we will call a separator a *linear separator* of a data set if all of the data with one label falls on one side of the separator and all of the data with the other label falls on the other side of the separator. For instance, the separator in the next example is a linear separator for the illustrated data. If there exists a linear separator on a dataset, we call this dataset *linearly separable*.

**Example:** Let $h$ be the linear classifier defined by $\theta = \begin{bmatrix} -1 \\ 1.5 \end{bmatrix}, \theta_0 = 3$.

The diagram below shows several points classified by $h$. In particular, let $x^{(1)} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$ and $x^{(2)} = \begin{bmatrix} 4 \\ -1 \end{bmatrix}$.

$$h(x^{(1)}; \theta, \theta_0) = \text{sign}\left(\begin{bmatrix} -1 & 1.5 \end{bmatrix} \begin{bmatrix} 3 \\ 2 \end{bmatrix} + 3\right) = \text{sign}(3) = +1$$
$$h(x^{(2)}; \theta, \theta_0) = \text{sign}\left(\begin{bmatrix} -1 & 1.5 \end{bmatrix} \begin{bmatrix} 4 \\ -1 \end{bmatrix} + 3\right) = \text{sign}(-2.5) = -1$$

Thus, $x^{(1)}$ and $x^{(2)}$ are given positive and negative classifications, respectively.



**Study Question:** What is the green vector normal to the separator? Specify it as a column vector.

**Study Question:** What change would you have to make to $\theta, \theta_0$ if you wanted to have the separating hyperplane in the same place, but to classify all the points labeled '+' in the diagram as negative and all the points labeled '-' in the diagram as positive?

## 4.3 Linear logistic classifiers

Given a data set and the hypothesis class of linear classifiers, our goal will be to find the linear classifier that optimizes an objective function relating its predictions to the training data. To make this problem computationally reasonable, we will need to take care in how we formulate the optimization problem to achieve this goal.

For classification, it is natural to make predictions in $\{+1, -1\}$ and use the 0-1 loss function, $\mathcal{L}_{01}$, as introduced in Chapter 1:

$$\mathcal{L}_{01}(g, a) = \begin{cases} 0 & \text{if } g = a \\ 1 & \text{otherwise} \end{cases} .$$

However, even for simple linear classifiers, it is very difficult to find values for $\theta, \theta_0$ that minimize simple 0-1 training error

$$J(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}_{01}(\text{sign}(\theta^T x^{(i)} + \theta_0), y^{(i)}) \ .$$

This problem is NP-hard, which probably implies that solving the most difficult instances of this problem would require computation time *exponential* in the number of training examples, $n$.

The "probably" here is not because we're too lazy to look it up, but actually because of a fundamental unsolved problem in computer-science theory, known as "P vs. NP."

What makes this a difficult optimization problem is its lack of "smoothness":

- There can be two hypotheses, $(\theta, \theta_0)$ and $(\theta', \theta_0')$, where one is closer in parameter space to the optimal parameter values $(\theta^*, \theta_0^*)$, but they make the same number of misclassifications so they have the same J value.
- All predictions are categorical: the classifier can't express a degree of certainty about whether a particular input $x$ should have an associated value $y$.

For these reasons, if we are considering a hypothesis $\theta, \theta_0$ that makes five incorrect predictions, it is difficult to see how we might change $\theta, \theta_0$ so that it will perform better, which makes it difficult to design an algorithm that searches in a sensible way through the space of hypotheses for a good one. For these reasons, we investigate another hypothesis class: *linear logistic classifiers*, providing their definition, then an approach for learning such classifiers using optimization.

### 4.3.1 Linear logistic classifiers: definition

The hypotheses in a linear logistic classifier (LLC) are parameterized by a $d$-dimensional vector $\theta$ and a scalar $\theta_0$, just as is the case for linear classifiers. However, instead of making predictions in $\{+1, -1\}$, LLC hypotheses generate real-valued outputs in the interval $(0, 1)$. An LLC has the form

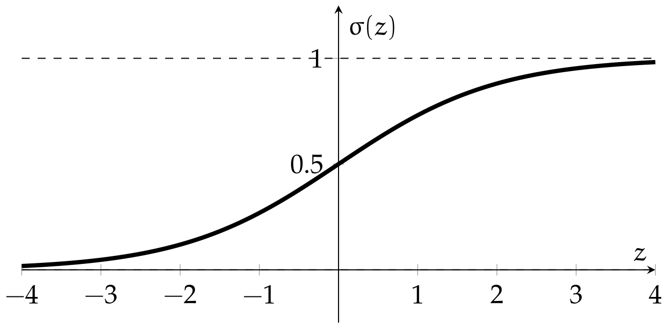
$$h(x; \theta, \theta_0) = \sigma(\theta^T x + \theta_0) \ .$$

This looks familiar! What's new?

The *logistic* function, also known as the *sigmoid* function, is defined as

$$\sigma(z) = \frac{1}{1 + e^{-z}} \ ,$$

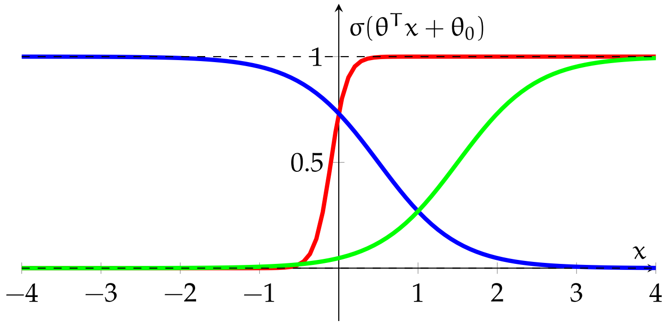
and is plotted below, as a function of its input $z$. Its output can be interpreted as a probability, because for any value of $z$ the output is in $(0, 1)$.



**Study Question:** Convince yourself the output of $\sigma$ is always in the interval $(0, 1)$. Why can't it equal 0 or equal 1? For what value of $z$ does $\sigma(z) = 0.5$?

### 4.3.2 Linear logistic classifier: examples

What does an LLC look like? Let's consider the simple case where $d = 1$, so our input points simply lie along the $x$ axis. Classifiers in this case have dimension 0, meaning that they are points. The plot below shows LLCs for three different parameter settings: $\sigma(10x + 1)$, $\sigma(-2x + 1)$, and $\sigma(2x - 3)$.



**Study Question:** Which plot is which? What governs the steepness of the curve? What governs the $x$ value where the output is equal to 0.5?

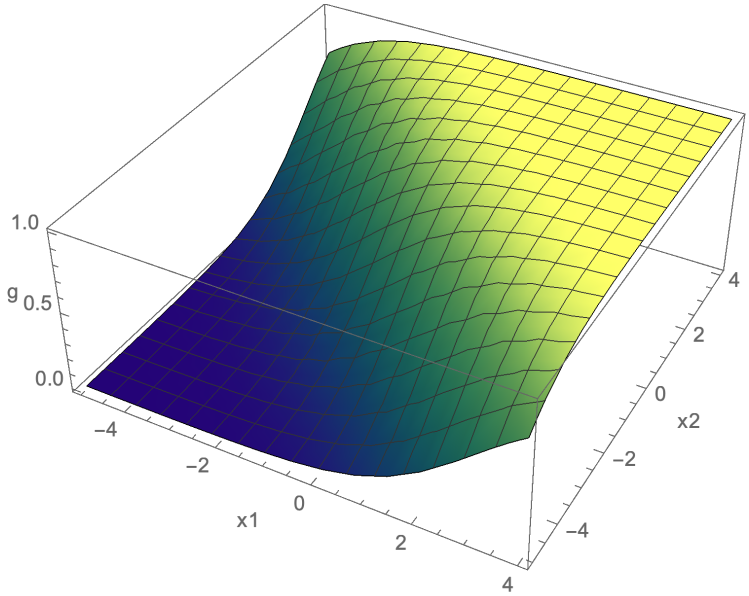
But wait! Remember that the definition of a classifier is that it's a mapping from $\mathbb{R}^d \rightarrow \{-1, +1\}$ or to some other discrete set. So, then, it seems like an LLC is actually not a classifier!

Given an LLC, with an output value in $(0, 1)$, what should we do if we are forced to make a prediction in $\{+1, -1\}$? A default answer is to predict $+1$ if $\sigma(\theta^T x + \theta_0) > 0.5$ and $-1$ otherwise. The value 0.5 is sometimes called a *prediction threshold*.

In fact, for different problem settings, we might prefer to pick a different prediction threshold. The field of *decision theory* considers how to make this choice. For example, if the consequences of predicting $+1$ when the answer should be $-1$ are much worse than the consequences of predicting $-1$ when the answer should be $+1$, then we might set the prediction threshold to be greater than 0.5.

**Study Question:** Using a prediction threshold of 0.5, for what values of $x$ do each of the LLCs shown in the figure above predict $+1$?

When $d = 2$, then our inputs $x$ lie in a two-dimensional space with axes $x_1$ and $x_2$, and the output of the LLC is a surface, as shown below, for $\theta = (1, 1), \theta_0 = 2$.



**Study Question:** Convince yourself that the set of points for which $\sigma(\theta^T x + \theta_0) = 0.5$, that is, the "boundary" between positive and negative predictions with prediction threshold 0.5, is a line in $(x_1, x_2)$ space. What particular line is it for the case in the figure above? How would the plot change for $\theta = (1, 1)$, but now with $\theta_0 = -2$? For $\theta = (-1, -1), \theta_0 = 2$?

### 4.3.3 Learning linear logistic classifiers

Optimization is a key approach to solving machine learning problems; this also applies to learning linear logistic classifiers (LLCs) by defining an appropriate loss function for optimization. A first attempt might be to use the simple 0-1 loss function $\mathcal{L}_{01}$ that gives a value of 0 for a correct prediction, and a 1 for an incorrect prediction. As noted earlier, however, this gives rise to an objective function that is very difficult to optimize, and so we pursue another strategy for defining our objective.

For learning LLCs, we'd have a class of hypotheses whose outputs are in $(0, 1)$, but for which we have training data with $y$ values in $\{+1, -1\}$. How can we define an appropriate loss function? We start by changing our interpretation of the output to be *the probability that the input should map to output value 1* (we might also say that this is the probability that the input is in class 1 or that the input is 'positive.')

**Study Question:** If $h(x)$ is the probability that $x$ belongs to class $+1$, what is the probability that $x$ belongs to the class $-1$, assuming there are only these two classes?

Intuitively, we would like to have *low loss if we assign a high probability to the correct class.* We'll define a loss function, called *negative log-likelihood* (NLL), that does just this. In addition, it has the cool property that it extends nicely to the case where we would like to classify our inputs into more than two classes.

In order to simplify the description, we assume that (or transform our data so that) the labels in the training data are $y \in \{0, 1\}$.

**Remember to be sure your $y$ values have this form if you try to learn an LLC using NLL!**

We would like to pick the parameters of our classifier to maximize the probability assigned by the LLC to the correct $y$ values, as specified in the training set. Letting guess $g^{(i)} = \sigma(\theta^T x^{(i)} + \theta_0)$, that probability is

That crazy huge $\Pi$ represents taking the product over a bunch of factors just as huge $\Sigma$ represents taking the sum over a bunch of terms.

$$\prod_{i=1}^{n} \begin{cases} g^{(i)} & \text{if } y^{(i)} = 1 \\ 1 - g^{(i)} & \text{otherwise} \end{cases} ,$$

under the assumption that our predictions are independent. This can be cleverly rewritten, when $y^{(i)} \in \{0, 1\}$, as

$$\prod_{i=1}^{n} {g^{(i)}}^{y^{(i)}} (1 - g^{(i)})^{1 - y^{(i)}} \quad .$$

**Study Question:** Be sure you can see why these two expressions are the same.

The big product above is kind of hard to deal with in practice, though. So what can we do? Because the log function is monotonic, the $\theta, \theta_0$ that maximize the quantity above will be the same as the $\theta, \theta_0$ that maximize its log, which is the following:

$$\sum_{i=1}^{n} \left( y^{(i)} \log g^{(i)} + (1 - y^{(i)}) \log(1 - g^{(i)}) \right) \quad .$$

Finally, we can turn the maximization problem above into a minimization problem by taking the negative of the above expression, and writing in terms of minimizing a loss

$$\sum_{i=1}^{n} \mathcal{L}_{\text{nll}}(g^{(i)}, y^{(i)})$$

where $\mathcal{L}_{\text{nll}}$ is the *negative log-likelihood* loss function:

$$\mathcal{L}_{\text{nll}}(\text{guess}, \text{actual}) = -\left(\text{actual} \cdot \log(\text{guess}) + (1 - \text{actual}) \cdot \log(1 - \text{guess})\right) \quad .$$

This loss function is also sometimes referred to as the *log loss* or *cross entropy*.and it won't make any real difference. If we ask you for numbers, use log base e.

**What is the objective function for linear logistic classification?** We can finally put all these pieces together and develop an objective function for optimizing regularized negative log-likelihood for a linear logistic classifier. In fact, this process is usually called "logistic regression," so we'll call our objective $J_{\text{lr}}$, and define it as

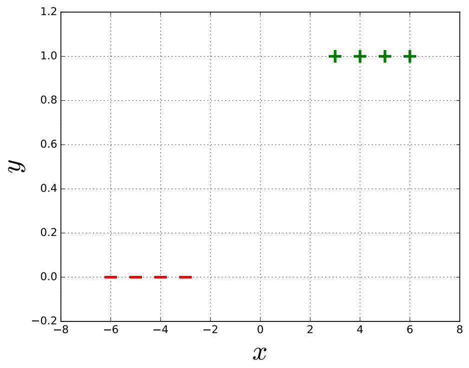
That's a lot of fancy words!

$$J_{\text{lr}}(\theta, \theta_0; \mathcal{D}) = \left( \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}_{\text{nll}}(\sigma(\theta^T x^{(i)} + \theta_0), y^{(i)}) \right) + \lambda \left\|\theta\right\|^2 \quad . \tag{4.1}$$

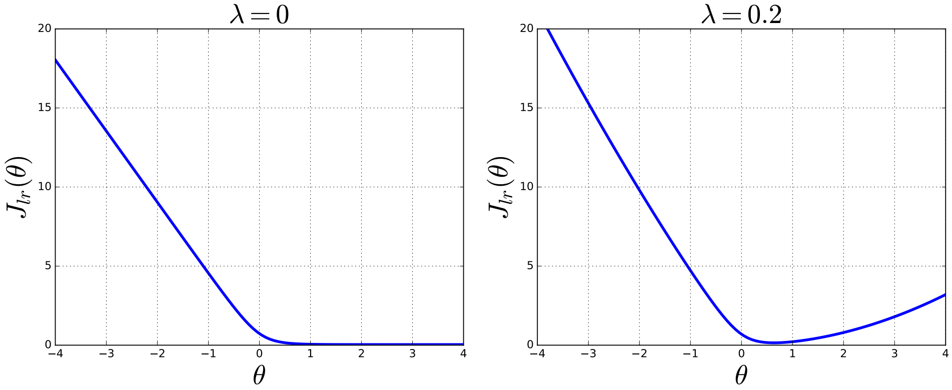
**Study Question:** Consider the case of linearly separable data. What will the $\theta$ values that optimize this objective be like if $\lambda = 0$? What will they be like if $\lambda$ is very big? Try to work out an example in one dimension with two data points.

**What role does regularization play for classifiers?** This objective function has the same structure as the one we used for regression, Eq. 2.2, where the first term (in parentheses) is the average loss, and the second term is for regularization. Regularization is needed for building classifiers that can generalize well (just as was the case for regression). The parameter $\lambda$ governs the trade-off between the two terms as illustrated in the following example.

Suppose we wish to obtain a linear logistic classifier for this one-dimensional dataset:



Clearly, this can be fit very nicely by a hypothesis $h(x) = \sigma(\theta x)$, but what is the best value for $\theta$? Evidently, when there is no regularization ($\lambda = 0$), the objective function $J_{lr}(\theta)$ will approach zero for large values of $\theta$, as shown in the plot on the left, below. However, would the best hypothesis really have an infinite (or very large) value for $\theta$? Such a hypothesis would suggest that the data indicate strong certainty that a sharp transition between $y = 0$ and $y = 1$ occurs exactly at $x = 0$, despite the actual data having a wide gap around $x = 0$.



**Study Question:** Be sure this makes sense. When the $\theta$ values are very large, what does the sigmoid curve look like? Why do we say that it has a strong certainty in that case?

In absence of other beliefs about the solution, we might prefer that our linear logistic classifier not be overly certain about its predictions, and so we might prefer a smaller $\theta$ over a large $\theta$. By not being overconfident, we might expect a somewhat smaller $\theta$ to perform better on future examples drawn from this same distribution. This preference can be realized using a nonzero value of the regularization trade-off parameter, as illustrated in the plot on the right, above, with $\lambda = 0.2$.

To refresh on some vocabulary, we say that in this example, a very large $\theta$ would be *overfit* to the training data.

Another nice way of thinking about regularization is that we would like to prevent our hypothesis from being too dependent on the particular training data that we were given: we would like for it to be the case that if the training data were changed slightly, the hypothesis would not change by much.

## 4.4 Gradient descent for logistic regression

Now that we have a hypothesis class (LLC) and a loss function (NLL), we need to take some data and find parameters! Sadly, there is no lovely analytical solution like the one we obtained for regression, in Section 2.6.2. Good thing we studied gradient descent! We can perform gradient descent on the $J_{lr}$ objective, as we'll see next. We can also apply stochastic gradient descent to this problem.

Luckily, $J_{lr}$ has enough nice properties that gradient descent and stochastic gradient descent should generally "work". We'll soon see some more challenging optimization problems though – in the context of neural networks, in Section 6.7.

First we need derivatives with respect to both $\theta_0$ (the scalar component) and $\theta$ (the vector component) of $\Theta$. Explicitly, they are:

Some passing familiarity with matrix derivatives is helpful here. A foolproof way of computing them is to compute partial derivative of $J$ with respect to each component $\theta_i$ of $\theta$.

$$\nabla_\theta J_{\text{lr}}(\theta, \theta_0) = \frac{1}{n}\sum_{i=1}^{n}\left(g^{(i)} - y^{(i)}\right)x^{(i)} + 2\lambda\theta$$

$$\frac{\partial J_{\text{lr}}(\theta, \theta_0)}{\partial \theta_0} = \frac{1}{n}\sum_{i=1}^{n}\left(g^{(i)} - y^{(i)}\right) \quad .$$

Note that $\nabla_\theta J_{\text{lr}}$ will be of shape $d \times 1$ and $\frac{\partial J_{\text{lr}}}{\partial \theta_0}$ will be a scalar since we have separated $\theta_0$ from $\theta$ here.

**Study Question:** Convince yourself that the dimensions of all these quantities are correct, under the assumption that $\theta$ is $d \times 1$.

**Study Question:** Compute $\nabla_\theta \|\theta\|^2$ by finding the vector of partial derivatives $(\partial \|\theta\|^2 / \partial\theta_1, \dots, \partial \|\theta\|^2 / \partial\theta_d)$. What is the shape of $\nabla_\theta \|\theta\|^2$?

**Study Question:** Compute $\nabla_\theta \mathcal{L}_{\text{nll}}(\sigma(\theta^T x + \theta_0), y)$ by finding the vector of partial derivatives $(\partial\mathcal{L}_{\text{nll}}(\sigma(\theta^T x + \theta_0), y)/\partial\theta_1, \dots, \partial\mathcal{L}_{\text{nll}}(\sigma(\theta^T x + \theta_0), y)/\partial\theta_d)$.

**Study Question:** Use these last two results to verify our derivation above.

Putting everything together, our gradient descent algorithm for logistic regression becomes:

```
LR-GRADIENT-DESCENT($\theta_{init}, \theta_{0init}, \eta, \epsilon$)
$\theta^{(0)} = \theta_{init}$
$\theta_0^{(0)} = \theta_{0init}$
$t = 0$
repeat
    $t = t + 1$
    $\theta^{(t)} = \theta^{(t-1)} - \eta\left(\frac{1}{n}\sum_{i=1}^{n}\left(\sigma\left({\theta^{(t-1)}}^T x^{(i)} + \theta_0^{(t-1)}\right) - y^{(i)}\right)x^{(i)} + 2\lambda\theta^{(t-1)}\right)$
    $\theta_0^{(t)} = \theta_0^{(t-1)} - \eta\left(\frac{1}{n}\sum_{i=1}^{n}\left(\sigma\left({\theta^{(t-1)}}^T x^{(i)} + \theta_0^{(t-1)}\right) - y^{(i)}\right)\right)$
until $\left|J_{\text{lr}}(\theta^{(t)}, \theta_0^{(t)}) - J_{\text{lr}}(\theta^{(t-1)}, \theta_0^{(t-1)})\right| < \epsilon$
return $\theta^{(t)}, \theta_0^{(t)}$
```

Logistic regression, implemented using batch or stochastic gradient descent, is a useful and fundamental machine learning technique. We will also see later that it corresponds to a one-layer neural network with a sigmoidal activation function, and so is an important step toward understanding neural networks.

### 4.4.1 Convexity of the NLL Loss Function

Much like the squared-error loss function that we saw for linear regression, the NLL loss function for linear logistic regression is a convex function. This means that running gradient descent with a reasonable set of hyperparameters will converge arbitrarily close to the minimum of the objective function.

We will use the following facts to demonstrate that the NLL loss function is a convex function:

- if the derivative of a function of a scalar argument is monotonically increasing, then it is a convex function,
- the sum of convex functions is also convex,
- a convex function of an affine function is a convex function.

Let $z = \theta^T x + \theta_0$; $z$ is an affine function of $\theta$ and $\theta_0$. It therefore suffices to show that the functions $f_1(z) = -\log(\sigma(z))$ and $f_2(z) = -\log(1 - \sigma(z))$ are convex with respect to $z$.

First, we can see that since,

$$\begin{aligned}
\frac{d}{dz} f_1(z) &= \frac{d}{dz}\left[-\log(1/(1+\exp(-z)))\right],\\
&= \frac{d}{dz}\left[\log(1+\exp(-z))\right],\\
&= -\exp(-z)/(1+\exp(-z)),\\
&= -1+\sigma(z),
\end{aligned}$$

the derivative of the function $f_1(z)$ is a monotonically increasing function and therefore $f_1$ is a convex function.

Second, we can see that since,

$$\begin{aligned}
\frac{d}{dz} f_2(z) &= \frac{d}{dz}\left[-\log(\exp(-z)/(1+\exp(-z)))\right],\\
&= \frac{d}{dz}\left[\log(1+\exp(-z))+z\right],\\
&= \sigma(z),
\end{aligned}$$

the derivative of the function $f_2(z)$ is also monotonically increasing and therefore $f_2$ is a convex function.

## 4.5 Handling multiple classes

So far, we have focused on the *binary* classification case, with only two possible classes. But what can we do if we have multiple possible classes (e.g., we want to predict the genre of a movie)? There are two basic strategies:

- Train multiple binary classifiers using different subsets of our data and combine their outputs to make a class prediction.
- Directly train a multi-class classifier using a hypothesis class that is a generalization of logistic regression, using a *one-hot* output encoding and NLL loss.

The method based on NLL is in wider use, especially in the context of neural networks, and is explored here. In the following, we will assume that we have a data set $\mathcal{D}$ in which the inputs $x^{(i)} \in R^d$ but the outputs $y^{(i)}$ are drawn from a set of K classes $\{c_1, \dots, c_K\}$. Next, we extend the idea of NLL directly to multi-class classification with K classes, where the training label is represented with what is called a *one-hot* vector $y = \begin{bmatrix} y_1, \dots, y_K \end{bmatrix}^T$, where $y_k = 1$ if the example is of class k and $y_k = 0$ otherwise. Now, we have a problem of mapping an input $x^{(i)}$ that is in $\mathbb{R}^d$ into a K-dimensional output. Furthermore, we would like this output to be interpretable as a discrete probability distribution over the possible classes, which means the elements of the output vector have to be *non-negative* (greater than or equal to 0) and sum to 1.

We will do this in two steps. First, we will map our input $x^{(i)}$ into a vector value $z^{(i)} \in \mathbb{R}^K$ by letting $\theta$ be a whole $d \times K$ *matrix* of parameters, and $\theta_0$ be a $K \times 1$ vector, so that

$$z = \theta^T x + \theta_0 \ .$$

Next, we have to extend our use of the sigmoid function to the multi-dimensional *softmax* function, that takes a whole vector $z \in \mathbb{R}^K$ and generates

Let's check dimensions! $\theta^T$ is $K \times d$ and $x$ is $d \times 1$, and $\theta_0$ is $K \times 1$, so $z$ is $K \times 1$ and we're good!

$$g = softmax(z) = \begin{bmatrix} \exp(z_1)/\sum_i \exp(z_i) \\ \vdots \\ \exp(z_K)/\sum_i \exp(z_i) \end{bmatrix} \ .$$

which can be interpreted as a probability distribution over K items. To make the final prediction of the class label, we can then look at $g$, find the most likely probability over these K entries in $g$, (i.e. find the largest entry in $g$,) and return the corresponding index as the "one-hot" element of 1 in our prediction.

**Study Question:** Convince yourself that the vector of $g$ values will be non-negative and sum to 1.

Putting these steps together, our hypotheses will be

$$h(x; \theta, \theta_0) = softmax(\theta^T x + \theta_0) \ .$$

Now, we retain the goal of maximizing the probability that our hypothesis assigns to the correct output $y_k$ for each input $x$. We can write this probability, letting $g$ stand for our "guess", $h(x)$, for a single example $(x, y)$ as $\prod_{k=1}^{K} g_k^{y_k}$.

**Study Question:** How many elements that are not equal to 1 will there be in this product?

The negative log of the probability that we are making a correct guess is, then, for *one-hot* vector $y$ and *probability distribution* vector $g$,

$$\mathcal{L}_{\text{nllm}}(g, y) = -\sum_{k=1}^{K} y_k \cdot \log(g_k) \ .$$

We'll call this NLLM for *negative log likelihood multiclass.* It is also worth noting that the NLLM loss function is also convex; however, we will omit the proof.

**Study Question:** Be sure you see that is $\mathcal{L}_{\text{nllm}}$ is minimized when the guess assigns high probability to the true class.

**Study Question:** Show that $\mathcal{L}_{\text{nllm}}$ for $K = 2$ is the same as $\mathcal{L}_{\text{nll}}$.

## 4.6 Prediction accuracy and validation

In order to formulate classification with a smooth objective function that we can optimize robustly using gradient descent, we changed the output from discrete classes to probability values and the loss function from 0-1 loss to NLL. However, when time comes to actually make a prediction we usually have to make a hard choice: buy stock in Acme or not? And, we get rewarded if we guessed right, independent of how sure or not we were when we made the guess.

The performance of a classifier is often characterized by its *accuracy*, which is the percentage of a data set that it predicts correctly in the case of 0-1 loss. We can see that accuracy of hypothesis $h$ on data $\mathcal{D}$ is the fraction of the data set that does not incur any loss:

$$A(h;\mathcal{D}) = 1 - \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}_{01}(g^{(i)}, y^{(i)}) \ ,$$

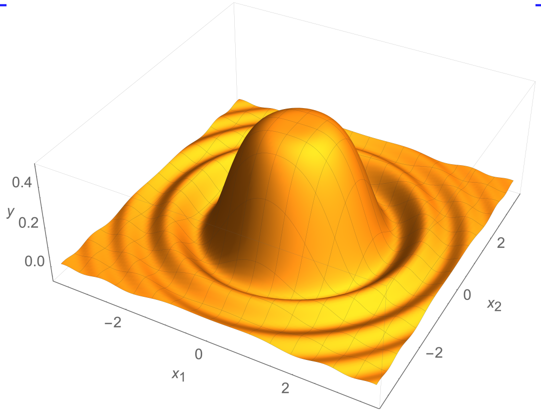
where $g^{(i)}$ is the final guess for one class or the other that we make from $h(x^{(i)})$, e.g., after thresholding. It's noteworthy here that we use a different loss function for optimization than for evaluation. This is a compromise we make for computational ease and efficiency.

# CHAPTER 5

# Feature representation

Linear regression and classification are powerful tools, but in the real world, data often exhibit *non-linear* behavior that cannot immediately be captured by the linear models which we have built so far. For example, suppose the true behavior of a system (with $d = 2$) looks like this wavelet:

This plot is of the so-called *jinc* function $J_1(\rho)/\rho$ for $\rho^2 = x_1^2 + x_2^2$



Such behavior is actually ubiquitous in physical systems, e.g., in the vibrations of the surface of a drum, or scattering of light through an aperture. However, no single hyperplane would be a very good fit to such peaked responses!

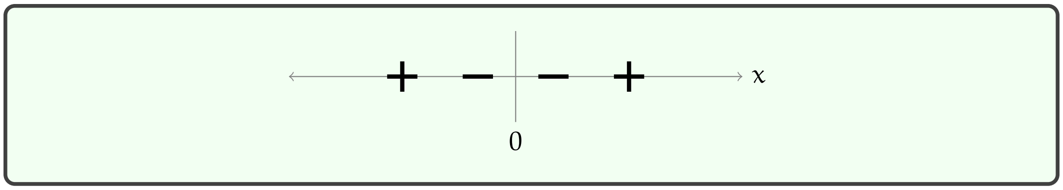
A richer class of hypotheses can be obtained by performing a non-linear feature transformation $\phi(x)$ before doing the regression. That is, $\theta^T x + \theta_0$ is a linear function of $x$, but $\theta^T \phi(x) + \theta_0$ is a non-linear function of $x$, if $\phi$ is a non-linear function of $x$.

There are many different ways to construct $\phi$. Some are relatively systematic and domain independent. Others are directly related to the semantics (meaning) of the original features, and we construct them deliberately with our application (goal) in mind.

## 5.1 Gaining intuition about feature transformations

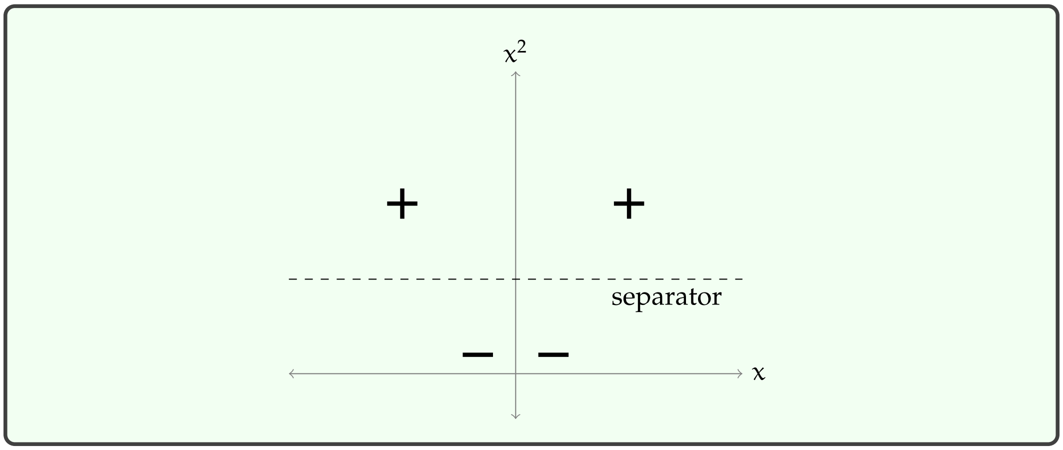
In this section, we explore the effects of non-linear feature transformations on simple classification problems, to gain intuition.

Let's look at an example data set that starts in 1-D:

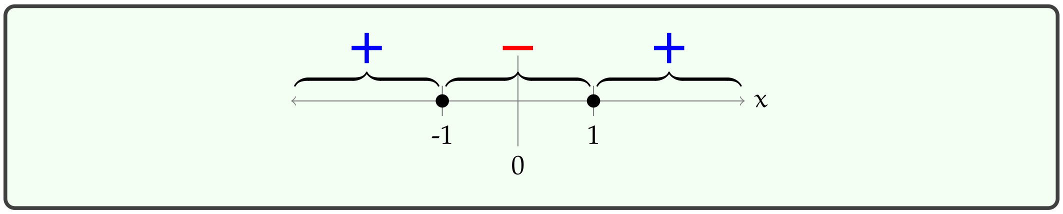


These points are not linearly separable, but consider the transformation $\phi(x) = [x, x^2]^T$. Putting the data in $\phi$ space, we see that it is now separable. There are lots of possible separators; we have just shown one of them here.

What's a linear separator for data in 1D? A point!



A linear separator in $\phi$ space is a nonlinear separator in the original space! Let's see how this plays out in our simple example. Consider the separator $x^2 - 1 = 0$, which labels the half-plane $x^2 - 1 > 0$ as positive. What separator does it correspond to in the original 1-D space? We have to ask the question: which $x$ values have the property that $x^2 - 1 = 0$. The answer is $+1$ and $-1$, so those two points constitute our separator, back in the original space. And we can use the same reasoning to find the region of 1D space that is labeled positive by this separator.



## 5.2 Systematic feature construction

Here are two different ways to systematically construct features in a *problem independent* way.

### 5.2.1 Polynomial basis

If the features in your problem are already naturally numerical, one systematic strategy for constructing a new feature space is to use a *polynomial basis*. The idea is that, if you are using the kth-order basis (where $k$ is a positive integer), you include a feature for every possible product of $k$ different dimensions in your original input.

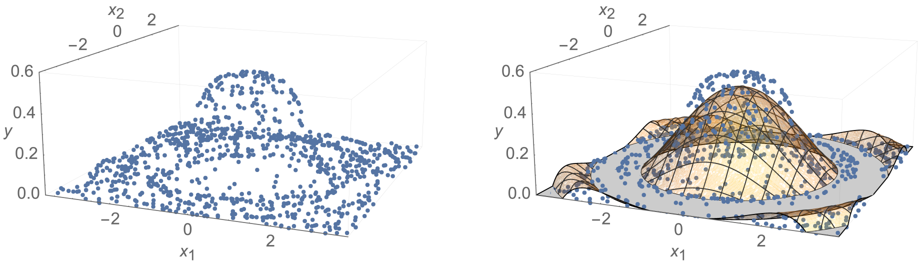
Here is a table illustrating the kth order polynomial basis for different values of $k$, calling out the cases when $d = 1$ and $d > 1$:

| Order | $d = 1$ | in general ($d > 1$) |
|---|---|---|
| 0 | $[1]$ | $[1]$ |
| 1 | $[1, x]^T$ | $[1, x_1, \dots, x_d]^T$ |
| 2 | $[1, x, x^2]^T$ | $[1, x_1, \dots, x_d, x_1^2, x_1x_2, \dots]^T$ |
| 3 | $[1, x, x^2, x^3]^T$ | $[1, x_1, \dots, x_1^2, x_1x_2, \dots, x_1x_2x_3, \dots]^T$ |
| $\vdots$ | $\vdots$ | $\vdots$ |

This transformation can be used in combination with linear regression or logistic regression (or any other regression or classification model). When we're using a linear regression or classification model, the key insight is that a linear regressor or separator in the *transformed space* is a non-linear regressor or separator in the original space.

For example, the wavelet pictured at the start of this chapter can be fit much better than with just a hyperplane, using linear regression with polynomials up to order $k = 8$:

Specifically, this example uses $[1, x_1, x_2, x_1^2 + x_2^2, (x_1^2 + x_2^2)^2, (x_1^2 + x_2^2)^4]^T$

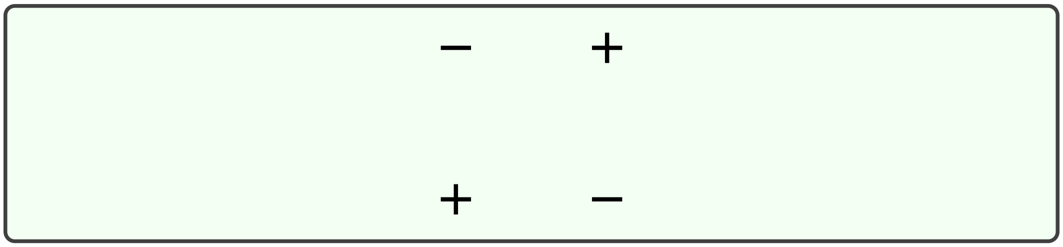


The raw data (with $n = 1000$ random samples) is plotted on the left, and the regression result (curved surface) is on the right.

Now let's look at a classification example and see how polynomial feature transformation may help us.

One well-known example is the "exclusive or" (XOR) data set, the drosophila of machine-learning data sets:

D. Melanogaster is a species of fruit fly, used as a simple system in which to study genetics, since 1910.



Clearly, this data set is not linearly separable. So, what if we try to solve the XOR classification problem using a polynomial basis as the feature transformation? We can just take our two-dimensional data and transform it into a higher-dimensional data set, by applying $\phi$. Now, we have a classification problem as usual.

Let's try it for $k = 2$ on our XOR problem. The feature transformation is

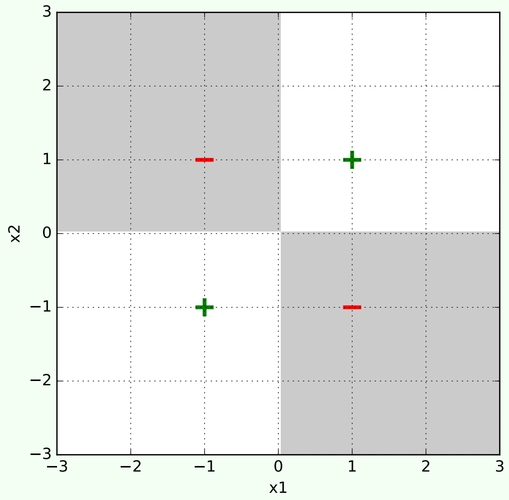
$$\phi([x_1, x_2]^T) = [1, x_1, x_2, x_1^2, x_1x_2, x_2^2]^T \quad .$$

**Study Question:** If we train a classifier after performing this feature transformation, would we lose any expressive power if we let $\theta_0 = 0$ (i.e., trained without offset instead of with offset)?

We might run a classification learning algorithm and find a separator with coefficients $\theta = [0, 0, 0, 0, 4, 0]^T$ and $\theta_0 = 0$. This corresponds to

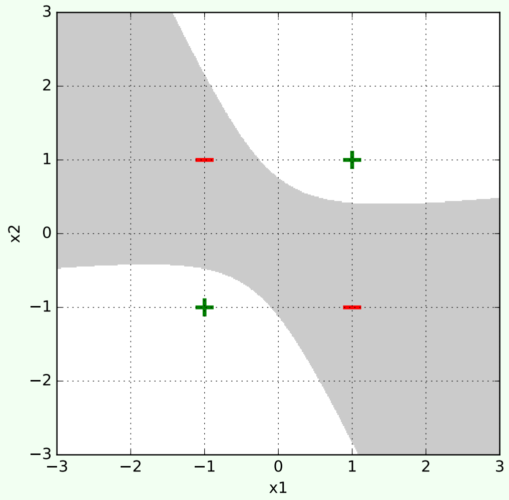
$$0 + 0x_1 + 0x_2 + 0x_1^2 + 4x_1x_2 + 0x_2^2 + 0 = 0$$

and is plotted below, with the gray shaded region classified as negative and the white region classified as positive:

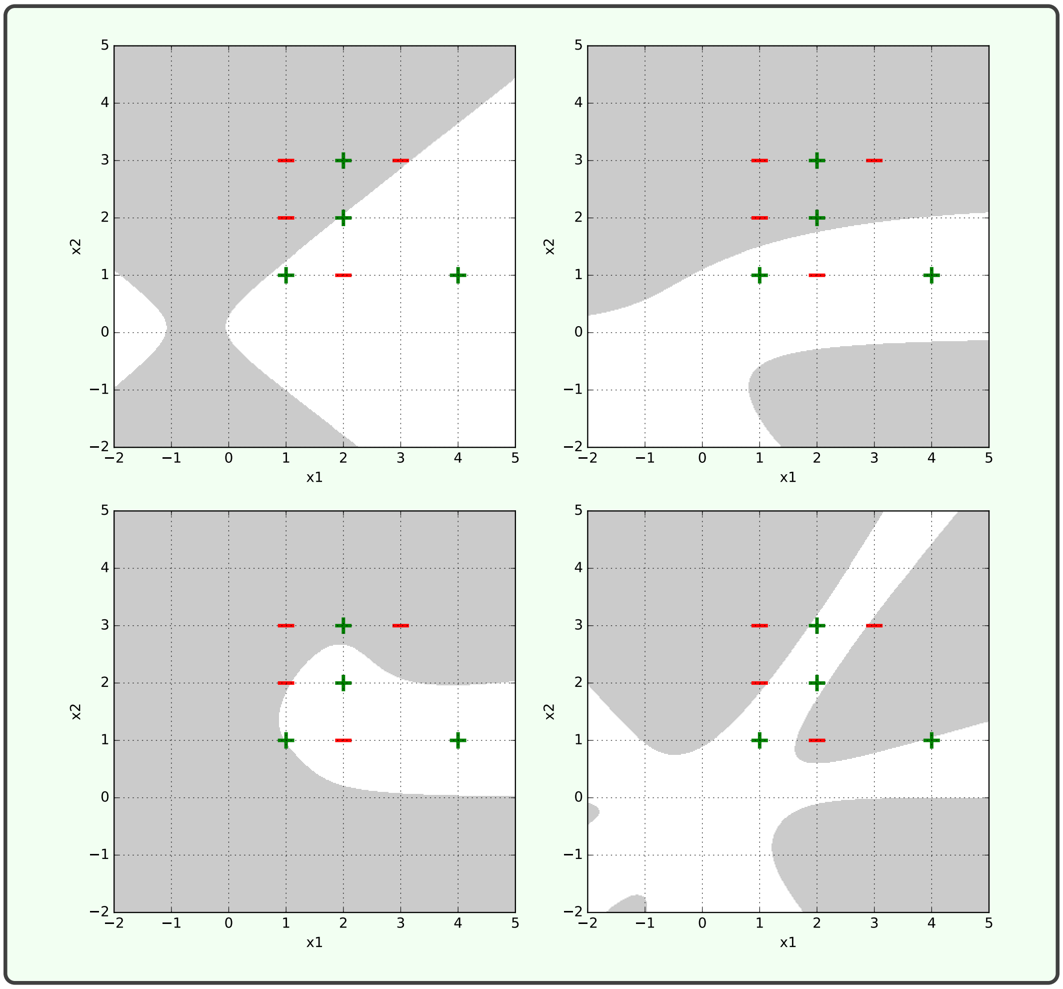


**Study Question:** Be sure you understand why this high-dimensional hyperplane is a separator, and how it corresponds to the figure.

For fun, we show some more plots below. Here is another result for a linear classifier on XOR generated with logistic regression and gradient descent, using a random initial starting point and second-order polynomial basis:



Here is a harder data set. Logistic regression with gradient descent failed to separate it with a second, third, or fourth-order basis feature representation, but succeeded with a fifth-order basis. Shown below are some results after ~ 1000 gradient descent iterations (from random starting points) for bases of order 2 (upper left), 3 (upper right), 4 (lower left), and 5 (lower right).



**Study Question:** Percy Eptron has a domain with four numeric input features, $(x_1, \ldots, x_4)$. He decides to use a representation of the form

$$\phi(x) = \text{PolyBasis}((x_1, x_2), 3)^\frown\text{PolyBasis}((x_3, x_4), 3)$$

where $a^\frown b$ means the vector $a$ concatenated with the vector $b$. What is the dimension of Percy's representation? Under what assumptions about the original features is this a reasonable choice?

### 5.2.2 Radial basis functions

Another cool idea is to use *the training data itself* to construct a feature space. The idea works as follows. For any particular point $p$ in the input space $\mathcal{X}$, we can construct a feature $f_p$ which takes any element $x \in \mathcal{X}$ and returns a scalar value that is related to how far $x$ is from the $p$ we started with.

Let's start with the basic case, in which $\mathcal{X} = \mathbb{R}^d$. Then we can define

$$f_p(x) = e^{-\beta\|p-x\|^2} \quad .$$

This function is maximized when $p = x$ and decreases exponentially as $x$ becomes more distant from $p$.

The parameter $\beta$ governs how quickly the feature value decays as we move away from the center point $p$. For large values of $\beta$, the $f_p$ values are nearly 0 almost everywhere except right near $p$; for small values of $\beta$, the features have a high value over a larger part of the space.

**Study Question:** What is $f_p(p)$?

Now, given a dataset $\mathcal{D}$ containing $n$ points, we can make a feature transformation $\phi$ that maps points in our original space, $\mathbb{R}^d$, into points in a new space, $\mathbb{R}^n$. It is defined as follows:

$$\phi(x) = [f_{x^{(1)}}(x), f_{x^{(2)}}(x), \ldots, f_{x^{(n)}}(x)]^T \quad .$$

So, we represent a new datapoint $x$ in terms of how far it is from each of the datapoints in our training set.

This idea can be generalized in several ways and is the fundamental concept underlying *kernel methods*, that you should read about some time. This idea of describing objects in terms of their similarity to a set of reference objects is very powerful and can be applied to cases where $\mathcal{X}$ is not a simple vector space, but where the inputs are graphs or strings or other types of objects, as long as there is a distance metric defined on it.

## 5.3 Hand-constructing features for real domains

In many machine-learning applications, we are given descriptions of the inputs with many different types of attributes, including numbers, words, and discrete features. An important factor in the success of an ML application is the way that the features are chosen to be encoded by the human who is framing the learning problem.

### 5.3.1 Discrete features

Getting a good encoding of discrete features is particularly important. You want to create "opportunities" for the ML system to find the underlying regularities. Although there are machine-learning methods that have special mechanisms for handling discrete inputs, most of the methods we consider in this class will assume the input vectors $x$ are in $\mathbb{R}^d$. So, we have to figure out some reasonable strategies for turning discrete values into (vectors of) real numbers.

We'll start by listing some encoding strategies, and then work through some examples. Let's assume we have some feature in our raw data that can take on one of $k$ discrete values.

- **Numeric:** Assign each of these values a number, say $1.0/k, 2.0/k, \ldots, 1.0$. We might want to then do some further processing, as described in Section 5.3.3. This is a sensible strategy *only* when the discrete values really do signify some sort of numeric quantity, so that these numerical values are meaningful.

- **Thermometer code:** If your discrete values have a natural ordering, from $1, \ldots, k$, but not a natural mapping into real numbers, a good strategy is to use a vector of length $k$ binary variables, where we convert discrete input value $0 < j \leqslant k$ into a vector in which the first $j$ values are 1.0 and the rest are 0.0. This does not necessarily imply anything about the spacing or numerical quantities of the inputs, but does convey something about ordering.

- **Factored code:** If your discrete values can sensibly be decomposed into two parts (say the "maker" and "model" of a car), then it's best to treat those as two separate features, and choose an appropriate encoding of each one from this list.

- **One-hot code:** If there is no obvious numeric, ordering, or factorial structure, then the best strategy is to use a vector of length $k$, where we convert discrete input value $0 < j \leqslant k$ into a vector in which all values are 0.0, except for the $j$th, which is 1.0.

- **Binary code:** It might be tempting for the computer scientists among us to use some binary code, which would let us represent $k$ values using a vector of length $\log k$. *This is a bad idea!* Decoding a binary code takes a lot of work, and by encoding your inputs this way, you'd be forcing your system to *learn* the decoding algorithm.

As an example, imagine that we want to encode blood types, that are drawn from the set $\{A+, A-, B+, B-, AB+, AB-, O+, O-\}$. There is no obvious linear numeric scaling or even ordering to this set. But there is a reasonable *factoring*, into two features: $\{A, B, AB, O\}$ and $\{+, -\}$. And, in fact, we can further reasonably factor the first group into $\{A, \text{not}A\}$, $\{B, \text{not}B\}$. So, here are two plausible encodings of the whole set:

It is sensible (according to Wikipedia!) to treat O as having neither feature A nor feature B.

- Use a 6-D vector, with two components of the vector each encoding the corresponding factor using a one-hot encoding.

- Use a 3-D vector, with one dimension for each factor, encoding its presence as 1.0 and absence as −1.0 (this is sometimes better than 0.0). In this case, $AB+$ would be $[1.0, 1.0, 1.0]^T$ and $O-$ would be $[-1.0, -1.0, -1.0]^T$.

**Study Question:** How would you encode $A+$ in both of these approaches?

### 5.3.2 Text

The problem of taking a text (such as a tweet or a product review, or even this document!) and encoding it as an input for a machine-learning algorithm is interesting and complicated. Much later in the class, we'll study sequential input models, where, rather than having to encode a text as a fixed-length feature vector, we feed it into a hypothesis word by word (or even character by character!).

There are some simple encodings that work well for basic applications. One of them is the *bag of words* (BOW) model. The idea is to let $d$ be the number of words in our vocabulary (either computed from the training set or some other body of text or dictionary). We will then make a binary vector (with values 1.0 and 0.0) of length $d$, where element $j$ has value 1.0 if word $j$ occurs in the document, and 0.0 otherwise.

### 5.3.3 Numeric values

If some feature is already encoded as a numeric value (heart rate, stock price, distance, etc.) then we should generally keep it as a numeric value. An exception might be a situation in which we know there are natural "breakpoints" in the semantics: for example, encoding someone's age in the US, we might make an explicit distinction between under and over 18 (or 21), depending on what kind of thing we are trying to predict. It might make sense to divide into discrete bins (possibly spacing them closer together for the very young) and to use a one-hot encoding for some sorts of medical situations in which we don't expect a linear (or even monotonic) relationship between age and some physiological features.

If we choose to leave a feature as numeric, it is typically useful to *scale* it, so that it tends to be in the range $[-1, +1]$. Without performing this transformation, if we have one feature with much larger values than another, it will take the learning algorithm a lot of work to find parameters that can put them on an equal basis. We could also perform a more involved scaling/transformation $\phi(x) = \dfrac{x - \overline{x}}{\sigma}$, where $\overline{x}$ is the average of the $x^{(i)}$, and $\sigma$ is the standard deviation of the $x^{(i)}$. The resulting feature values will have mean 0 and standard deviation 1. This transformation is sometimes called *standardizing* a variable.

Such standard variables are often known as "z-scores," for example, in the social sciences.

Then, of course, we might apply a higher-order polynomial-basis transformation to one or more groups of numeric features.

**Study Question:** Consider using a polynomial basis of order $k$ as a feature transformation $\phi$ on our data. Would increasing $k$ tend to increase or decrease structural error? What about estimation error?

# CHAPTER 6

# Neural Networks

You've probably been hearing a lot about "neural networks." Now that we have several useful machine-learning concepts (hypothesis classes, classification, regression, gradient descent, regularization, etc.) we are well equipped to understand neural networks in detail.

This is, in some sense, the "third wave" of neural nets. The basic idea is founded on the 1943 model of neurons of McCulloch and Pitts and the learning ideas of Hebb. There was a great deal of excitement, but not a lot of practical success: there were good training methods (e.g., perceptron) for linear functions, and interesting examples of non-linear functions, but no good way to train non-linear functions from data. Interest died out for a while, but was re-kindled in the 1980s when several people came up with a way to train neural networks with "back-propagation," which is a particular style of implementing gradient descent, that we will study here. By the mid-90s, the enthusiasm waned again, because although we could train non-linear networks, the training tended to be slow and was plagued by a problem of getting stuck in local optima. Support vector machines (SVMs) that use regularization of high-dimensional hypotheses by seeking to maximize the margin, and kernel methods that are an efficient and beautiful way of using feature transformations to non-linearly transform data into a higher-dimensional space, provided reliable learning methods with guaranteed convergence and no local optima.

As with many good ideas in science, the basic idea for how to train non-linear neural networks with gradient descent was independently developed by more than one researcher.

However, during the SVM enthusiasm, several groups kept working on neural networks, and their work, in combination with an increase in available data and computation, has made them rise again. They have become much more reliable and capable, and are now the method of choice in many applications. There are many, many variations of neural networks, which we can't even begin to survey. We will study the core "feed-forward" networks with "back-propagation" training, and then, in later chapters, address some of the major advances beyond this core.

The number increases daily, as may be seen on `arxiv.org`.

We can view neural networks from several different perspectives:

**View 1**: An application of stochastic gradient descent for classification and regression with a potentially very rich hypothesis class.

**View 2**: A brain-inspired network of neuron-like computing elements that learn distributed representations.

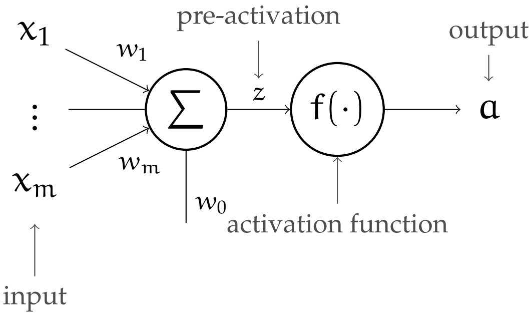
**View 3**: A method for building applications that make predictions based on huge amounts of data in very complex domains.

We will mostly take view 1, with the understanding that the techniques we develop will enable the applications in view 3. View 2 was a major motivation for the early development of neural networks, but the techniques we will study do not seem to actually account for the biological learning processes in brains.

Some prominent researchers are, in fact, working hard to find analogues of these methods in the brain.

## 6.1 Basic element

The basic element of a neural network is a "neuron," pictured schematically below. We will also sometimes refer to a neuron as a "unit" or "node."



It is a non-linear function of an input vector $x \in \mathbb{R}^m$ to a single output value $a \in \mathbb{R}$. It is parameterized by a vector of *weights* $(w_1, \ldots, w_m) \in \mathbb{R}^m$ and an *offset* or *threshold* $w_0 \in \mathbb{R}$. In order for the neuron to be non-linear, we also specify an *activation function* $f : \mathbb{R} \to \mathbb{R}$, which can be the identity ($f(x) = x$, in that case the neuron is a linear function of $x$), but can also be any other function, though we will only be able to work with it if it is differentiable.

Sorry for changing our notation here. We were using $d$ as the dimension of the input, but we are trying to be consistent here with many other accounts of neural networks. It is impossible to be consistent with all of them though—there are many different ways of telling this story.

This should remind you of our $\theta$ and $\theta_0$ for linear models.

The function represented by the neuron is expressed as:

$$a = f(z) = f\left(\left(\sum_{j=1}^{m} x_j w_j\right) + w_0\right) = f(w^T x + w_0) \ .$$

Before thinking about a whole network, we can consider how to train a single unit. Given a loss function $\mathcal{L}(\text{guess}, \text{actual})$ and a dataset $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(n)}, y^{(n)})\}$, we can do (stochastic) gradient descent, adjusting the weights $w, w_0$ to minimize

$$J(w, w_0) = \sum_i \mathcal{L}\left(\text{NN}(x^{(i)}; w, w_0), y^{(i)}\right) \ ,$$

where NN is the output of our single-unit neural net for a given input.

We have already studied two special cases of the neuron: linear logistic classifiers (LLCs) with NLL loss and regressors with quadratic loss! The activation function for the LLC is $f(x) = \sigma(x)$ and for linear regression it is simply $f(x) = x$.

**Study Question:** Just for a single neuron, imagine for some reason, that we decide to use activation function $f(z) = e^z$ and loss function $\mathcal{L}(\text{guess}, \text{actual}) = (\text{guess} - \text{actual})^2$. Derive a gradient descent update for $w$ and $w_0$.

## 6.2 Networks

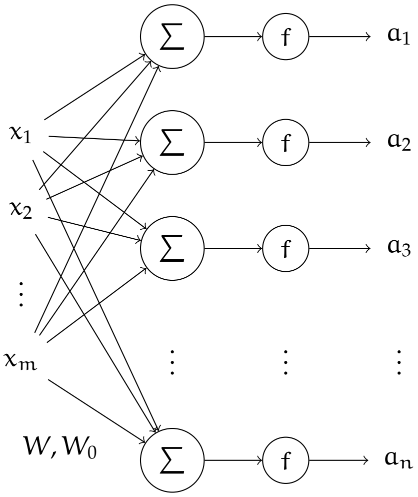
Now, we'll put multiple neurons together into a *network*. A neural network in general takes in an input $x \in \mathbb{R}^m$ and generates an output $a \in \mathbb{R}^n$. It is constructed out of multiple neurons; the inputs of each neuron might be elements of $x$ and/or outputs of other neurons. The outputs are generated by $n$ *output units*.

In this chapter, we will only consider *feed-forward* networks. In a feed-forward network, you can think of the network as defining a function-call graph that is *acyclic*: that is, the input to a neuron can never depend on that neuron's output. Data flows one way, from the inputs to the outputs, and the function computed by the network is just a composition of the functions computed by the individual neurons.

Although the graph structure of a feed-forward neural network can really be anything (as long as it satisfies the feed-forward constraint), for simplicity in software and analysis, we usually organize them into *layers*. A layer is a group of neurons that are essentially "in parallel": their inputs are outputs of neurons in the previous layer, and their outputs are the input to the neurons in the next layer. We'll start by describing a single layer, and then go on to the case of multiple layers.

### 6.2.1 Single layer

A *layer* is a set of units that, as we have just described, are not connected to each other. The layer is called *fully connected* if, as in the diagram below, all of the inputs (i.e., $x_1, x_2, \dots x_m$ in this case) are connected to every unit in the layer. A layer has input $x \in \mathbb{R}^m$ and output (also known as *activation*) $a \in \mathbb{R}^n$.



Since each unit has a vector of weights and a single offset, we can think of the weights of the whole layer as a matrix, $W$, and the collection of all the offsets as a vector $W_0$. If we have $m$ inputs, $n$ units, and $n$ outputs, then

- $W$ is an $m \times n$ matrix,
- $W_0$ is an $n \times 1$ column vector,
- $X$, the input, is an $m \times 1$ column vector,
- $Z = W^T X + W_0$, the *pre-activation*, is an $n \times 1$ column vector,
- $A$, the *activation*, is an $n \times 1$ column vector,

and the output vector is

$$A = f(Z) = f(W^T X + W_0) \ .$$

The activation function $f$ is applied element-wise to the pre-activation values $Z$.

### 6.2.2 Many layers

A single neural network generally combines multiple layers, most typically by feeding the outputs of one layer into the inputs of another layer.

We have to start by establishing some nomenclature. We will use $l$ to name a layer, and let $m^l$ be the number of inputs to the layer and $n^l$ be the number of outputs from the layer. Then, $W^l$ and $W_0^l$ are of shape $m^l \times n^l$ and $n^l \times 1$, respectively. Note that the input to layer $l$ is the output from layer $l-1$, so we have $m^l = n^{l-1}$, and as a result $A^{l-1}$ is of shape $m^l \times 1$, or equivalently $n^{l-1} \times 1$. Let $f^l$ be the activation function of layer $l$. Then, the pre-activation outputs are the $n^l \times 1$ vector

$$Z^l = {W^l}^T A^{l-1} + W_0^l$$

and the activation outputs are simply the $n^l \times 1$ vector

$$A^l = f^l(Z^l) \ .$$

It is technically possible to have different activation functions within the same layer, but, again, for convenience in specification and implementation, we generally have the same activation function within a layer.

Here's a diagram of a many-layered network, with two blocks for each layer, one representing the linear part of the operation and one representing the non-linear activation function. We will use this structural decomposition to organize our algorithmic thinking and implementation.

$$\xrightarrow{X = A^0}\underbrace{\boxed{\begin{matrix}W^1\\ W_0^1\end{matrix}}\xrightarrow{Z^1}\boxed{f^1}}_{\text{layer 1}}\xrightarrow{A^1}\underbrace{\boxed{\begin{matrix}W^2\\ W_0^2\end{matrix}}\xrightarrow{Z^2}\boxed{f^2}}_{\text{layer 2}}\xrightarrow{A^2}\cdots\xrightarrow{A^{L-1}}\underbrace{\boxed{\begin{matrix}W^L\\ W_0^L\end{matrix}}\xrightarrow{Z^L}\boxed{f^L}}_{\text{layer L}}\xrightarrow{A^L}$$

## 6.3 Choices of activation function

There are many possible choices for the activation function. We will start by thinking about whether it's really necessary to have an $f$ at all.

What happens if we let $f$ be the identity? Then, in a network with $L$ layers (we'll leave out $W_0$ for simplicity, but keeping it wouldn't change the form of this argument),

$$A^L = {W^L}^T A^{L-1} = {W^L}^T {W^{L-1}}^T \cdots {W^1}^T X \ .$$

So, multiplying out the weight matrices, we find that

$$A^L = W^{\text{total}} X \ ,$$

which is a *linear* function of $X$! Having all those layers did not change the representational capacity of the network: the non-linearity of the activation function is crucial.

**Study Question:** Convince yourself that any function representable by any number of linear layers (where $f$ is the identity function) can be represented by a single layer.

Now that we are convinced we need a non-linear activation, let's examine a few common choices. These are shown mathematically below, followed by plots of these functions.

**Step function:**

$$\text{step}(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{otherwise} \end{cases}$$

**Rectified linear unit (ReLU):**

$$\text{ReLU}(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases} = \max(0, z)$$

**Sigmoid function:** Also known as a *logistic* function. This can sometimes be interpreted as probability, because for any value of $z$ the output is in $(0, 1)$:

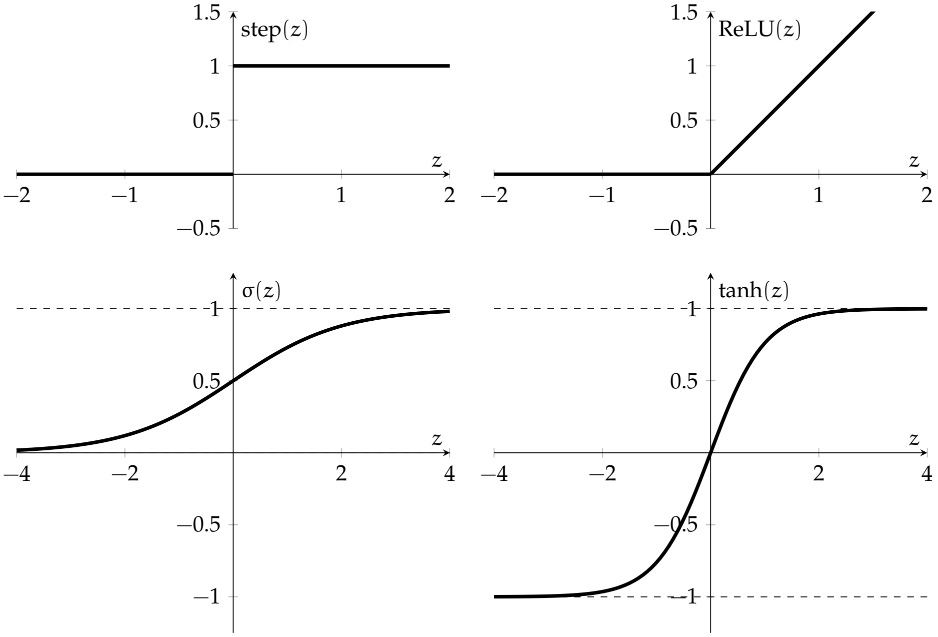
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

**Hyperbolic tangent:** Always in the range $(-1, 1)$:

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

**Softmax function:** Takes a whole vector $Z \in \mathbb{R}^n$ and generates as output a vector $A \in (0,1)^n$ with the property that $\sum_{i=1}^{n} A_i = 1$, which means we can interpret it as a probability distribution over $n$ items:

$$\text{softmax}(z) = \begin{bmatrix} \exp(z_1)/\sum_i \exp(z_i) \\ \vdots \\ \exp(z_n)/\sum_i \exp(z_i) \end{bmatrix}$$



The original idea for neural networks involved using the **step** function as an activation, but because the derivative of the step function is zero everywhere except at the discontinuity (and there it is undefined), gradient-descent methods won't be useful in finding a good setting of the weights, and so we won't consider them further. They have been replaced, in a sense, by the sigmoid, ReLU, and tanh activation functions.

**Study Question:** Consider sigmoid, ReLU, and tanh activations. Which one is most like a step function? Is there an additional parameter you could add to a sigmoid that would make it be more like a step function?

**Study Question:** What is the derivative of the ReLU function? Are there some values of the input for which the derivative vanishes?

ReLUs are especially common in internal ("hidden") layers, sigmoid activations are common for the output for binary classification, and softmax activations are common for the output for multi-class classification (see Section 4.3.3 for an explanation).

## 6.4 Loss functions and activation functions

Different loss functions make different assumptions about the range of values they will get as input and, as we have seen, different activation functions will produce output values in different ranges. When you are designing a neural network, it's important to make these things fit together well. In particular, we will think about matching loss functions with the activation function in the last layer, $f^L$. Here is a table of loss functions and activations that make sense for them:

| Loss | $f^L$ | task |
|---|---|---|
| squared | linear | regression |
| NLL | sigmoid | binary classification |
| NLLM | softmax | multi-class classification |

We explored squared loss in Chapter 2 and (NLL and NLLM) in Chapter 4.

## 6.5 Error back-propagation

We will train neural networks using gradient descent methods. It's possible to use *batch* gradient descent, in which we sum up the gradient over all the points (as in Section 3.2 of chapter 3) or stochastic gradient descent (SGD), in which we take a small step with respect to the gradient considering a single point at a time (as in Section 3.4 of Chapter 3).

Our notation is going to get pretty hairy pretty quickly. To keep it as simple as we can, we'll focus on computing the contribution of one data point $x^{(i)}$ to the gradient of the loss with respect to the weights, for SGD; you can simply sum up these gradients over all the data points if you wish to do batch descent.

So, to do SGD for a training example $(x, y)$, we need to compute $\nabla_W \mathcal{L}(\text{NN}(x; W), y)$, where $W$ represents all weights $W^l, W_0^l$ in all the layers $l = (1, \ldots, L)$. This seems terrifying, but is actually quite easy to do using the chain rule.

Remember that we are always computing the gradient of the loss function *with respect to the weights* for a particular value of $(x, y)$. That tells us how much we want to change the weights, in order to reduce the loss incurred on this particular training example.

Remember the chain rule! If $a = f(b)$ and $b = g(c)$, so that $a = f(g(c))$, then $\frac{da}{dc} = \frac{da}{db} \cdot \frac{db}{dc} = f'(b)g'(c) = f'(g(c))g'(c)$.

### 6.5.1 First, suppose everything is one-dimensional

To get some intuition for how these derivations work, we'll first suppose everything in our neural network is one-dimensional. In particular, we'll assume there are $m^l = 1$ inputs and $n^l = 1$ outputs at every layer. So layer $l$ looks like:

$$a^l = f^l(z^l), \quad z^l = w^l a^{l-1} + w_0^l.$$

In the equation above, we're using the lowercase letters $a^l, z^l, w^l, a^{l-1}, w_0^l$ to emphasize that all of these quantities are scalars just for the moment. We'll look at the more general matrix case below.

To use SGD, then, we want to compute $\partial \mathcal{L}(\text{NN}(x; W), y)/\partial w^l$ and $\partial \mathcal{L}(\text{NN}(x; W), y)/\partial w_0^l$

Check your understanding: why do we need exactly these quantities for SGD?

for each layer $l$ and each data point $(x, y)$. Below we'll write "loss" as an abbreviation for $\mathcal{L}(\text{NN}(x; W), y)$. Then our first quantity of interest is $\partial\text{loss}/\partial w^l$. The chain rule gives us the following. First, let's look at the case $l = L$:

$$\begin{aligned}\frac{\partial\text{loss}}{\partial w^L} &= \frac{\partial\text{loss}}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial w^L}\\ &= \frac{\partial\text{loss}}{\partial a^L} \cdot (f^L)'(z^L) \cdot a^{L-1}.\end{aligned}$$

Now we can look at the case of general $l$:

$$\begin{aligned}\frac{\partial\text{loss}}{\partial w^l} &= \frac{\partial\text{loss}}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdots \frac{\partial z^{l+1}}{\partial a^l} \cdot \frac{\partial a^l}{\partial z^l} \cdot \frac{\partial z^l}{\partial w^l}\\ &= \frac{\partial\text{loss}}{\partial a^L} \cdot (f^L)'(z^L) \cdot w^L \cdot (f^{L-1})'(z^{L-1}) \cdot \cdots \cdot w^{l+1} \cdot (f^l)'(z^l) \cdot a^{l-1}\\ &= \frac{\partial\text{loss}}{\partial z^l} \cdot a^{l-1}.\end{aligned}$$

Note that every multiplication above is scalar multiplication because every term in every product above is a scalar. And though we solved for all the other terms in the product, we haven't solved for $\partial\text{loss}/\partial a^L$ because the derivative will depend on which loss function you choose. Once you choose a loss function though, you should be able to compute this derivative.

**Study Question:** Suppose you choose squared loss. What is $\partial\text{loss}/\partial a^L$?

**Study Question:** Check the derivations above yourself. You should use the chain rule and also solve for the individual derivatives that arise in the chain rule.

**Study Question:** Check that the the final layer ($l = L$) case is a special case of the general layer $l$ case above.

**Study Question:** Derive $\partial\mathcal{L}(\text{NN}(x; W), y)/\partial w_0^l$ for yourself, for both the final layer ($l = L$) and general $l$.

**Study Question:** Does the $L = 1$ case remind you of anything from earlier in this course?

**Study Question:** Write out the full SGD algorithm for this neural network.

It's pretty typical to run the chain rule from left to right like we did above. But, for where we're going next, it will be useful to notice that it's completely equivalent to write it in the other direction. So we can rewrite our result from above as follows:

$$\frac{\partial\text{loss}}{\partial w^l} = a^{l-1} \cdot \frac{\partial\text{loss}}{\partial z^l} \tag{6.1}$$

$$\frac{\partial\text{loss}}{\partial z^l} = \frac{\partial a^l}{\partial z^l} \cdot \frac{\partial z^{l+1}}{\partial a^l} \cdots \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial\text{loss}}{\partial a^L} \tag{6.2}$$

$$= \frac{\partial a^l}{\partial z^l} \cdot w^{l+1} \cdots \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot w^L \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial\text{loss}}{\partial a^L}. \tag{6.3}$$

### 6.5.2 The general case

Next we're going to do everything that we did above, but this time we'll allow any number of inputs $m^l$ and outputs $n^l$ at every layer. First, we'll tell you the results that correspond to our derivations above. Then we'll talk about why they make sense. And finally we'll derive them carefully.

OK, let's start with the results! Again, below we'll be using "loss" as an abbreviation for $\mathcal{L}(\text{NN}(x; W), y)$. Then,

$$\underbrace{\frac{\partial \text{loss}}{\partial W^l}}_{m^l \times n^l} = \underbrace{A^{l-1}}_{m^l \times 1} \underbrace{\left(\frac{\partial \text{loss}}{\partial Z^l}\right)^T}_{1 \times n^l} \tag{6.4}$$

$$\frac{\partial \text{loss}}{\partial Z^l} = \frac{\partial A^l}{\partial Z^l} \cdot \frac{\partial Z^{l+1}}{\partial A^l} \cdot \dots \cdot \frac{\partial A^{L-1}}{\partial Z^{L-1}} \cdot \frac{\partial Z^L}{\partial A^{L-1}} \cdot \frac{\partial A^L}{\partial Z^L} \cdot \frac{\partial \text{loss}}{\partial A^L} \tag{6.5}$$

$$= \frac{\partial A^l}{\partial Z^l} \cdot W^{l+1} \cdot \dots \cdot \frac{\partial A^{L-1}}{\partial Z^{L-1}} \cdot W^L \cdot \frac{\partial A^L}{\partial Z^L} \cdot \frac{\partial \text{loss}}{\partial A^L} \,. \tag{6.6}$$

First, compare each equation to its one-dimensional counterpart, and make sure you see the similarities. That is, compare the general weight derivatives in Eq. 6.4 to the one-dimensional case in Eq. 6.1. Compare the intermediate derivative of loss with respect to the pre-activations $Z^l$ in Eq. 6.5 to the one-dimensional case in Eq. 6.2. And finally compare the version where we've substituted in some of the derivatives in Eq. 6.6 to Eq. 6.3. Hopefully you see how the forms are very analogous. But in the matrix case, we now have to be careful about the matrix dimensions. We'll check these matrix dimensions below.

Let's start by talking through each of the terms in the matrix version of these equations. Recall that loss is a scalar, and $W^l$ is a matrix of size $m^l \times n^l$. You can read about the conventions in the course for derivatives starting in this chapter in Appendix A. By these conventions (not the only possible conventions!), we have that $\partial \text{loss}/\partial W^l$ will be a matrix of size $m^l \times n^l$ whose $(i, j)$ entry is the scalar $\partial \text{loss}/\partial W^l_{i,j}$. In some sense, we're just doing a bunch of traditional scalar derivatives, and the matrix notation lets us write them all simultaneously and succinctly. In particular, for SGD, we need to find the derivative of the loss with respect to every scalar component of the weights because these are our model's parameters and therefore are the things we want to update in SGD.

The next quantity we see in Eq. 6.4 is $A^{l-1}$, which we recall has size $m^l \times 1$ (or equivalently $n^{l-1} \times 1$ since it represents the outputs of the $l-1$ layer). Finally, we see $\partial \text{loss}/\partial Z^l$. Again, loss is a scalar, and $Z^l$ is a $n^l \times 1$ vector. So by the conventions in Appendix A, we have that $\partial \text{loss}/\partial Z^l$ has size $n^l \times 1$. The transpose then has size $1 \times n^l$. Now you should be able to check that the dimensions all make sense in Eq. 6.4; in particular, you can check that inner dimensions agree in the matrix multiplication and that, after the multiplication, we should be left with something that has the dimensions on the lefthand side.

Now let's look at Eq. 6.6. We're computing $\partial \text{loss}/\partial Z^l$ so that we can use it in Eq. 6.4. The weights are familiar. The one part that remains is terms of the form $\partial A^l/\partial Z^l$. Checking out Appendix A, we see that this term should be a matrix of size $n^l \times n^l$ since $A^l$ and $Z^l$ both have size $n^l \times 1$. The $(i, j)$ entry of this matrix is $\partial A^l_j/\partial Z^l_i$. This scalar derivative is something that you can compute when you know your activation function. If you're not using a softmax activation function, $A^l_j$ typically is a function only of $Z^l_j$, which means that $\partial A^l_j/\partial Z^l_i$ should equal 0 whenever $i \neq j$, and that $\partial A^l_j/\partial Z^l_j = (f^l)'(Z^l_j)$.

**Study Question:** Compute the dimensions of every term in Eqs. 6.5 and 6.6 using Appendix A. After you've done that, check that all the matrix multiplications work; that is, check that the inner dimensions agree and that the lefthand side and right-hand side of these equations have the same dimensions.

**Study Question:** If I use the identity activation function, what is $\partial A^l_j/\partial Z^l_j$ for any j? What is the full matrix $\partial A^l/\partial Z^l$?

### 6.5.3 Derivations for the general case

You can use everything above without deriving it yourself. But if you want to find the gradients of loss with respect to $W^l_0$ (which we need for SGD!), then you'll want to know how to actually do these derivations. So next we'll work out the derivations.

The key trick is to just break every equation down into its scalar meaning. For instance, the $(i, j)$ element of $\partial\text{loss}/\partial W^l$ is $\partial\text{loss}/\partial W^l_{i,j}$. If you think about it for a moment (and it might help to go back to the one-dimensional case), the loss is a function of the elements of $Z^l$, and the elements of $Z^l$ are a function of the $W^l_{i,j}$. There are $n^l$ elements of $Z^l$, so we can use the chain rule to write

$$\frac{\partial\text{loss}}{\partial W^l_{i,j}} = \sum_{k=1}^{n^l} \frac{\partial\text{loss}}{\partial Z^l_k}\frac{\partial Z^l_k}{\partial W^l_{i,j}}. \tag{6.7}$$

To figure this out, let's remember that $Z^l = (W^l)^\top A^{l-1} + W^l_0$. We can write one element of the $Z^l$ vector, then, as $Z^l_b = \sum_{a=1}^{m^l} W^l_{a,b}A^{l-1}_a + (W^l_0)_b$. It follows that $\partial Z^l_k/\partial W^l_{i,j}$ will be zero except when $k = j$ (check you agree!). So we can rewrite Eq. 6.7 as

$$\frac{\partial\text{loss}}{\partial W^l_{i,j}} = \frac{\partial\text{loss}}{\partial Z^l_j}\frac{\partial Z^l_j}{\partial W^l_{i,j}} = \frac{\partial\text{loss}}{\partial Z^l_j}A^{l-1}_i. \tag{6.8}$$

Finally, then, we match entries of the matrices on both sides of the equation above to recover Eq. 6.4.

**Study Question:** Check that Eq. 6.8 and Eq. 6.4 say the same thing.

**Study Question:** Convince yourself that $\partial Z^l/\partial A^{l-1} = W^l$ by comparing the entries of the matrices on both sides on the equality sign.

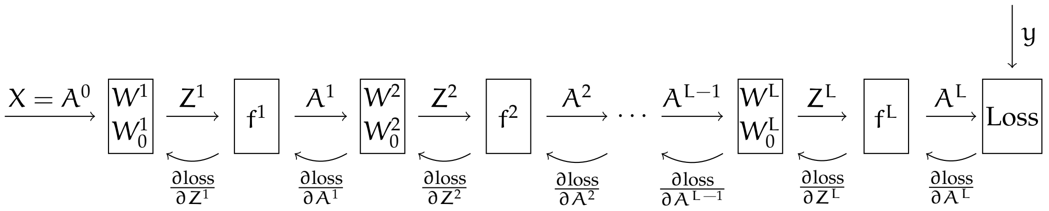
**Study Question:** Convince yourself that Eq. 6.5 is true.

**Study Question:** Apply the same reasoning to find the gradients of loss with respect to $W^l_0$.

### 6.5.4 Reflecting on backpropagation

This general process of computing the gradients of the loss with respect to the weights is called *error back-propagation*. The idea is that we first do a *forward pass* to compute all the $a$ and $z$ values at all the layers, and finally the actual loss. Then, we can work backward and compute the gradient of the loss with respect to the weights in each layer, starting at layer L and going back to layer 1.

We could call this "blame propagation". Think of loss as how mad we are about the prediction just made. Then $\partial\text{loss}/\partial A^L$ is how much we blame $A^L$ for the loss. The last module has to take in $\partial\text{loss}/\partial A^L$ and compute $\partial\text{loss}/\partial Z^L$, which is how much we blame $Z^L$ for the loss. The next module (working backwards) takes in $\partial\text{loss}/\partial Z^L$ and computes $\partial\text{loss}/\partial A^{L-1}$. So every module is accepting its blame for the loss, computing how much of it to allocate to each of its inputs, and passing the blame back to them.


$X = A^0$
$W^1$ $W^1_0$
$Z^1$
$f^1$
$A^1$
$W^2$ $W^2_0$
$Z^2$
$f^2$
$A^2$
$\cdots$
$A^{L-1}$
$W^L$ $W^L_0$
$Z^L$
$f^L$
$A^L$
Loss
$y$
$\frac{\partial\text{loss}}{\partial Z^1}$
$\frac{\partial\text{loss}}{\partial A^1}$
$\frac{\partial\text{loss}}{\partial Z^2}$
$\frac{\partial\text{loss}}{\partial A^2}$
$\frac{\partial\text{loss}}{\partial A^{L-1}}$
$\frac{\partial\text{loss}}{\partial Z^L}$
$\frac{\partial\text{loss}}{\partial A^L}$

If we view our neural network as a sequential composition of modules (in our work so far, it has been an alternation between a linear transformation with a weight matrix, and a component-wise application of a non-linear activation function), then we can define a simple API for a module that will let us compute the forward and backward passes, as well as do the necessary weight updates for gradient descent. Each module has to provide the following "methods." We are already using letters $a, x, y, z$ with particular meanings, so here we will use $u$ as the vector input to the module and $v$ as the vector output:

- forward: $u \rightarrow v$
- backward: $u, v, \partial L/\partial v \rightarrow \partial L/\partial u$
- weight grad: $u, \partial L/\partial v \rightarrow \partial L/\partial W$ only needed for modules that have weights $W$

In homework we will ask you to implement these modules for neural network components, and then use them to construct a network and train it as described in the next section.

## 6.6 Training

Here we go! Here's how to do stochastic gradient descent training on a feed-forward neural network. After this pseudo-code, we motivate the choice of initialization in lines 2 and 3. The actual computation of the gradient values (e.g., $\partial\text{loss}/\partial A^L$) is not directly defined in this code, because we want to make the structure of the computation clear.

**Study Question:** What is $\partial Z^l/\partial W^l$?

**Study Question:** Which terms in the code below depend on $f^L$?

```
SGD-NEURAL-NET($\mathcal{D}_n, T, L, (m^1, \dots, m^L), (f^1, \dots, f^L)$, Loss)
1   for $l = 1$ to $L$
2       $W^l_{ij} \sim \text{Gaussian}(0, 1/m^l)$
3       $W^l_{0j} \sim \text{Gaussian}(0, 1)$
4   for $t = 1$ to $T$
5       $i$ = random sample from $\{1, \dots, n\}$
6       $A^0 = x^{(i)}$
7       // forward pass to compute the output $A^L$
8       for $l = 1$ to $L$
9           $Z^l = W^{l^T} A^{l-1} + W^l_0$
10          $A^l = f^l(Z^l)$
11      loss = Loss($A^L, y^{(i)}$)
12      for $l = L$ to 1:
13          // error back-propagation
14          $\partial\text{loss}/\partial A^l$ = if $l < L$ then $\partial Z^{l+1}/\partial A^l \cdot \partial\text{loss}/\partial Z^{l+1}$ else $\partial\text{loss}/\partial A^L$
15          $\partial\text{loss}/\partial Z^l = \partial A^l/\partial Z^l \cdot \partial\text{loss}/\partial A^l$
16          // compute gradient with respect to weights
17          $\partial\text{loss}/\partial W^l = A^{l-1} \cdot \left(\partial\text{loss}/\partial Z^l\right)^T$
18          $\partial\text{loss}/\partial W^l_0 = \partial\text{loss}/\partial Z^l$
19          // stochastic gradient descent update
20          $W^l = W^l - \eta(t) \cdot \partial\text{loss}/\partial W^l$
21          $W^l_0 = W^l_0 - \eta(t) \cdot \partial\text{loss}/\partial W^l_0$
```

Initializing $W$ is important; if you do it badly there is a good chance the neural network training won't work well. First, it is important to initialize the weights to random values. We want different parts of the network to tend to "address" different aspects of the problem; if they all start at the same weights, the symmetry will often keep the values from moving in useful directions. Second, many of our activation functions have (near) zero slope when the pre-activation $z$ values have large magnitude, so we generally want to keep the initial weights small so we will be in a situation where the gradients are non-zero, so that gradient descent will have some useful signal about which way to go.

One good general-purpose strategy is to choose each weight at random from a Gaussian (normal) distribution with mean 0 and standard deviation $(1/m)$ where $m$ is the number of inputs to the unit.

**Study Question:** If the input $x$ to this unit is a vector of 1's, what would the expected pre-activation $z$ value be with these initial weights?

We write this choice (where $\sim$ means "is drawn randomly from the distribution") as $W^l_{ij} \sim \text{Gaussian}\left(0, \frac{1}{m^l}\right)$. It will often turn out (especially for fancier activations and loss functions) that computing $\frac{\partial \text{loss}}{\partial Z^L}$ is easier than computing $\frac{\partial \text{loss}}{\partial A^L}$ and $\frac{\partial A^L}{\partial Z^L}$. So, we may instead ask for an implementation of a loss function to provide a backward method that computes $\partial \text{loss}/\partial Z^L$ directly.

## 6.7 Optimizing neural network parameters

Because neural networks are just parametric functions, we can optimize loss with respect to the parameters using standard gradient-descent software, but we can take advantage of the structure of the loss function and the hypothesis class to improve optimization. As we have seen, the modular function-composition structure of a neural network hypothesis makes it easy to organize the computation of the gradient. As we have also seen earlier, the structure of the loss function as a sum over terms, one per training data point, allows us to consider stochastic gradient methods. In this section we'll consider some alternative strategies for organizing training, and also for making it easier to handle the step-size parameter.

### 6.7.1 Batches

Assume that we have an objective of the form

$$J(W) = \sum_{i=1}^{n} \mathcal{L}(h(x^{(i)}; W), y^{(i)}) \ ,$$

where $h$ is the function computed by a neural network, and $W$ stands for all the weight matrices and vectors in the network.

Recall that, when we perform *batch* (or the vanilla) gradient descent, we use the update rule

$$W_t = W_{t-1} - \eta \nabla_W J(W_{t-1}) \ ,$$

which is equivalent to

$$W_t = W_{t-1} - \eta \sum_{i=1}^{n} \nabla_W \mathcal{L}(h(x^{(i)}; W_{t-1}), y^{(i)}) \ .$$

So, we sum up the gradient of loss at each training point, with respect to $W$, and then take a step in the negative direction of the gradient.

In *stochastic* gradient descent, we repeatedly pick a point $(x^{(i)}, y^{(i)})$ at random from the data set, and execute a weight update on that point alone:

$$W_t = W_{t-1} - \eta \nabla_W \mathcal{L}(h(x^{(i)}; W_{t-1}), y^{(i)}) \ .$$

As long as we pick points uniformly at random from the data set, and decrease $\eta$ at an appropriate rate, we are guaranteed, with high probability, to converge to at least a local optimum.

These two methods have offsetting virtues. The batch method takes steps in the exact gradient direction but requires a lot of computation before even a single step can be taken, especially if the data set is large. The stochastic method begins moving right away, and can sometimes make very good progress before looking at even a substantial fraction of the whole data set, but if there is a lot of variability in the data, it might require a very small $\eta$ to effectively average over the individual steps moving in "competing" directions.

An effective strategy is to "average" between batch and stochastic gradient descent by using *mini-batches*. For a mini-batch of size $K$, we select $K$ distinct data points uniformly at random from the data set and do the update based just on their contributions to the gradient

$$W_t = W_{t-1} - \eta \sum_{i=1}^{K} \nabla_W \mathcal{L}(h(x^{(i)}; W_{t-1}), y^{(i)}) \ .$$

Most neural network software packages are set up to do mini-batches.

**Study Question:** For what value of $K$ is mini-batch gradient descent equivalent to stochastic gradient descent? To batch gradient descent?

Picking $K$ unique data points at random from a large data-set is potentially computationally difficult. An alternative strategy, if you have an efficient procedure for randomly shuffling the data set (or randomly shuffling a list of indices into the data set) is to operate in a loop, roughly as follows:

```
MINI-BATCH-SGD(NN, data, K)
1  $n$ = length(data)
2  while not done:
3      RANDOM-SHUFFLE(data)
4      for $i = 1$ to $\lceil n/K \rceil$
5          BATCH-GRADIENT-UPDATE(NN, data[$(i-1)K : iK$])
```

See note on the ceiling[1] function, for the case when $n/K$ is not an integer.

### 6.7.2 Adaptive step-size

Picking a value for $\eta$ is difficult and time-consuming. If it's too small, then convergence is slow and if it's too large, then we risk divergence or slow convergence due to oscillation. This problem is even more pronounced in stochastic or mini-batch mode, because we know we need to decrease the step size for the formal guarantees to hold.

It's also true that, within a single neural network, we may well want to have different step sizes. As our networks become *deep* (with increasing numbers of layers) we can find that magnitude of the gradient of the loss with respect the weights in the last layer, $\partial \text{loss}/\partial W_L$, may be substantially different from the gradient of the loss with respect to the weights in the first layer $\partial \text{loss}/\partial W_1$. If you look carefully at Eq. 6.6, you can see that the output gradient is multiplied by all the weight matrices of the network and is "fed back" through all the derivatives of all the activation functions. This can lead to a problem of *exploding* or *vanishing* gradients, in which the back-propagated gradient is much too big or small to be used in an update rule with the same step size.

[1] In line 4 of the algorithm above, $\lceil \cdot \rceil$ is known as the *ceiling* function; it returns the smallest integer greater than or equal to its input. E.g., $\lceil 2.5 \rceil = 3$ and $\lceil 3 \rceil = 3$.

So, we can consider having an independent step-size parameter *for each weight*, and updating it based on a local view of how the gradient updates have been going. Some common strategies for this include *momentum* (“averaging” recent gradient updates), *Adadelta* (take larger steps in parts of the space where $J(W)$ is nearly flat), and *Adam* (which combines these two previous ideas). Details of these approaches are described in Appendix B.0.1.

This section is very strongly influenced by Sebastian Ruder’s excellent blog posts on the topic: `ruder.io/optimizing-gradient-descent`

## 6.8 Regularization

So far, we have only considered optimizing loss on the training data as our objective for neural network training. But, as we have discussed before, there is a risk of overfitting if we do this. The pragmatic fact is that, in current deep neural networks, which tend to be very large and to be trained with a large amount of data, overfitting is not a huge problem. This runs counter to our current theoretical understanding and the study of this question is a hot area of research. Nonetheless, there are several strategies for regularizing a neural network, and they can sometimes be important.

### 6.8.1 Methods related to ridge regression

One group of strategies can, interestingly, be shown to have similar effects to each other: early stopping, weight decay, and adding noise to the training data.

Result is due to Bishop, described in his textbook and here `doi.org/10.1162/neco.1995.7.1.108`.

Early stopping is the easiest to implement and is in fairly common use. The idea is to train on your training set, but at every *epoch* (a pass through the whole training set, or possibly more frequently), evaluate the loss of the current $W$ on a *validation set*. It will generally be the case that the loss on the training set goes down fairly consistently with each iteration, the loss on the validation set will initially decrease, but then begin to increase again. Once you see that the validation loss is systematically increasing, you can stop training and return the weights that had the lowest validation error.

Another common strategy is to simply penalize the norm of all the weights, as we did in ridge regression. This method is known as *weight decay*, because when we take the gradient of the objective

$$J(W) = \sum_{i=1}^{n} \mathcal{L}(\mathrm{NN}(x^{(i)}), y^{(i)}; W) + \lambda \|W\|^2$$

we end up with an update of the form

$$\begin{aligned} W_t &= W_{t-1} - \eta \left( \left( \nabla_W \mathcal{L}(\mathrm{NN}(x^{(i)}), y^{(i)}; W_{t-1}) \right) + 2\lambda W_{t-1} \right) \\ &= W_{t-1}(1 - 2\lambda\eta) - \eta \left( \nabla_W \mathcal{L}(\mathrm{NN}(x^{(i)}), y^{(i)}; W_{t-1}) \right) \quad . \end{aligned}$$

This rule has the form of first “decaying” $W_{t-1}$ by a factor of $(1 - 2\lambda\eta)$ and then taking a gradient step.

Finally, the same effect can be achieved by perturbing the $x^{(i)}$ values of the training data by adding a small amount of zero-mean normally distributed noise before each gradient computation. It makes intuitive sense that it would be more difficult for the network to overfit to particular training data if they are changed slightly on each training step.

### 6.8.2 Dropout

Dropout is a regularization method that was designed to work with deep neural networks. The idea behind it is, rather than perturbing the data every time we train, we’ll perturb the network! We’ll do this by randomly, on each training step, selecting a set of units in each layer and prohibiting them from participating. Thus, all of the units will have to take a kind of "collective" responsibility for getting the answer right, and will not be able to rely on any small subset of the weights to do all the necessary computation. This tends also to make the network more robust to data perturbations.

During the training phase, for each training example, for each unit, randomly with probability $p$ temporarily set $a^{\ell}_{j} = 0$. There will be no contribution to the output and no gradient update for the associated unit.

**Study Question:** Be sure you understand why, when using SGD, setting an activation value to 0 will cause that unit's weights not to be updated on that iteration.

When we are done training and want to use the network to make predictions, we multiply all weights by $p$ to achieve the same average activation levels.

Implementing dropout is easy! In the forward pass during training, we let

$$a^{\ell} = f(z^{\ell}) * d^{\ell}$$

where $*$ denotes component-wise product and $d^{\ell}$ is a vector of 0's and 1's drawn randomly with probability $p$. The backwards pass depends on $a^{\ell}$, so we do not need to make any further changes to the algorithm.

It is common to set $p$ to 0.5, but this is something one might experiment with to get good results on your problem and data.

### 6.8.3 Batch normalization

Another strategy that seems to help with regularization and robustness in training is *batch normalization*. It was originally developed to address a problem of *covariate shift*: that is, if you consider the second layer of a two-layer neural network, the distribution of its input values is changing over time as the first layer's weights change. Learning when the input distribution is changing is extra difficult: you have to change your weights to improve your predictions, but also just to compensate for a change in your inputs (imagine, for instance, that the magnitude of the inputs to your layer is increasing over time—then your weights will have to decrease, just to keep your predictions the same).

For more details see `arxiv.org/abs/1502.03167`.

So, when training with mini-batches, the idea is to *standardize* the input values for each mini-batch, just in the way that we did it in Section 5.3.3 of Chapter 5, subtracting off the mean and dividing by the standard deviation of each input dimension. This means that the scale of the inputs to each layer remains the same, no matter how the weights in previous layers change. However, this somewhat complicates matters, because the computation of the weight updates will need to take into account that we are performing this transformation. In the modular view, batch normalization can be seen as a module that is applied to $z^l$, interposed after the product with $W^l$ and before input to $f^l$.

We follow here the suggestion from the original paper of applying batch normalization before the activation function. Since then it has been shown that, in some cases, applying it after works a bit better. But there aren't any definite findings on which works better and when.

Although batch-norm was originally justified based on the problem of covariate shift, it's not clear that that is actually why it seems to improve performance. Batch normalization can also end up having a regularizing effect for similar reasons that adding noise and dropout do: each mini-batch of data ends up being mildly perturbed, which prevents the network from exploiting very particular values of the data points. For those interested, the equations for batch normalization, including a derivation of the forward pass and backward pass, are described in Appendix B.0.2.

# CHAPTER 7

# Clustering

Oftentimes a dataset can be partitioned into different categories. A doctor may notice that their patients come in cohorts and different cohorts respond to different treatments. A biologist may gain insight by identifying that bats and whales, despite outward appearances, have some underlying similarity, and both should be considered members of the same category, i.e., "mammal". The problem of automatically identifying meaningful groupings in datasets is called clustering. Once these groupings are found, they can be leveraged toward interpreting the data and making optimal decisions for each group.

## 7.1 Clustering formalisms

Mathematically, clustering looks a bit like classification: we wish to find a mapping from datapoints, $x$, to categories, $y$. However, rather than the categories being predefined labels, the categories in clustering are automatically discovered *partitions* of an unlabeled dataset.

Because clustering does not learn from labeled examples, it is an example of an *unsupervised* learning algorithm. Instead of mimicking the mapping implicit in supervised training pairs $\{x^{(i)}, y^{(i)}\}_{i=1}^n$, clustering assigns datapoints to categories based on how the unlabeled data $\{x^{(i)}\}_{i=1}^n$ is *distributed* in data space.

Intuitively, a "cluster" is a group of datapoints that are all nearby to each other and far away from other clusters. Let's consider the following scatter plot. How many clusters do you think there are?

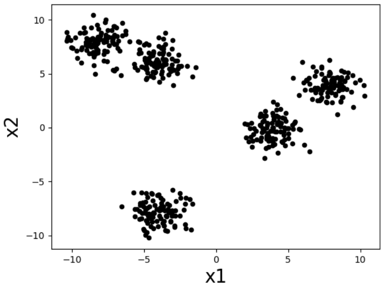


Figure 7.1: A dataset we would like to cluster. How many clusters do you think there are?

There seem to be about five clumps of datapoints and those clumps are what we would like to call clusters. If we assign all datapoints in each clump to a cluster corresponding to that clump, then we might desire that nearby datapoints are assigned to the same cluster, while far apart datapoints are assigned to different clusters.

In designing clustering algorithms, three critical things we need to decide are:

- How do we measure *distance* between datapoints? What counts as "nearby" and "far apart"?
- How many clusters should we look for?
- How do we evaluate how good a clustering is?

We will see how to begin making these decisions as we work through a concrete clustering algorithm in the next section.

## 7.2 The k-means formulation

One of the simplest and most commonly used clustering algorithms is called k-means. The goal of the k-means algorithm is to assign datapoints to $k$ clusters in such a way that the variance within clusters is as small as possible. Notice that this matches our intuitive idea that a cluster should be a tightly packed set of datapoints.

Similar to the way we showed that supervised learning could be formalized mathematically as the minimization of an objective function (loss function + regularization), we will show how unsupervised learning can also be formalized as minimizing an objective function. Let us denote the cluster assignment for a datapoint $x^{(i)}$ as $y^{(i)} \in \{1, 2, \ldots, k\}$, i.e., $y^{(i)} = 1$ means we are assigning datapoint $x^{(i)}$ to cluster number 1. Then the k-means objective can be quantified with the following objective function (which we also call the "k-means loss"):

$$\sum_{j=1}^{k} \sum_{i=1}^{n} \mathbb{1}(y^{(i)} = j) \left\| x^{(i)} - \mu^{(j)} \right\|^2 , \tag{7.1}$$

where $\mu^{(j)} = \frac{1}{N_j} \sum_{i=1}^{n} \mathbb{1}(y^{(i)} = j) x^{(i)}$ and $N_j = \sum_{i=1}^{n} \mathbb{1}(y^{(i)} = j)$, so that $\mu^{(j)}$ is the mean of all datapoints in cluster $j$, and using $\mathbb{1}(\cdot)$ to denote the indicator function (which takes on value of 1 if its argument is true and 0 otherwise). The inner sum (over data points) of the loss is the variance of datapoints within cluster $j$. We sum up the variance of all $k$ clusters to get our overall loss.

We will be careful to distinguish between the k-means *algorithm* and the k-means *objective*. As we will see, the k-means algorithm can be understood to be just one optimization algorithm which finds a local optimum of the k-means objective.

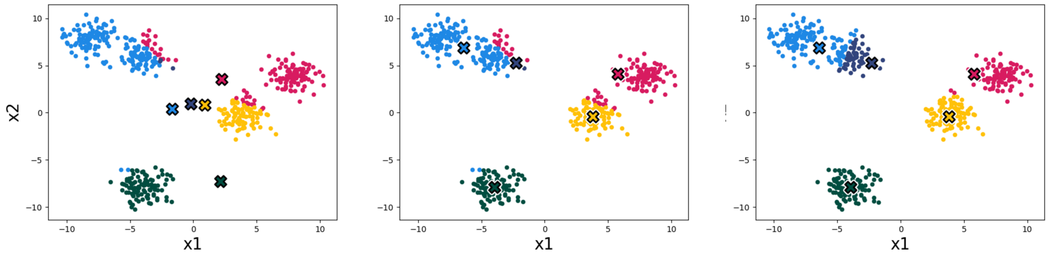
Recall that *variance* is a measure of how "spread out" data is, defined as the mean squared distance from the average value of the data.

### 7.2.1 K-means algorithm

The k-means algorithm minimizes this loss by alternating between two steps: given some initial cluster assignments: 1) compute the mean of all data in each cluster and assign this as the "cluster mean", and 2) reassign each datapoint to the cluster with nearest cluster mean. Fig. 7.2 shows what happens when we repeat these steps on the dataset from above.

Each time we reassign the data to the nearest cluster mean, the k-means loss decreases (the datapoints end up closer to their assigned cluster mean), or stays the same. And each time we recompute the cluster means the loss *also* decreases (the means end up closer to their assigned datapoints) or stays the same. Overall then, the clustering gets better and better, according to our objective – until it stops improving.

After four iterations of cluster assignment + update means in our example, the k-means algorithm stops improving. We say it has converged, and its final solution is shown in Fig. 7.3.



1) Initial assignments 2) Update means 3) Update assignments

Figure 7.2: The first three steps of running the k-means algorithm on this data. Datapoints are colored according to the cluster to which they are assigned. Cluster means are the larger X's with black outlines.

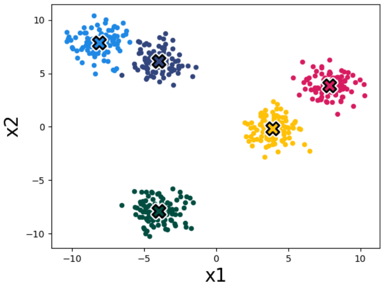


Figure 7.3: Converged result.

It seems to converge to something reasonable! Now let's write out the algorithm in complete detail:

```
K-MEANS($k, \tau, \{x^{(i)}\}_{i=1}^{n}$)
1   $\mu, y$ = random initialization
2   for $t = 1$ to $\tau$
3       $y_{\text{old}} = y$
4       for $i = 1$ to $n$
5           $y^{(i)} = \arg\min_j \left\| x^{(i)} - \mu^{(j)} \right\|^2$
6       for $j = 1$ to $k$
7           $\mu^{(j)} = \frac{1}{N_j} \sum_{i=1}^{n} \mathbb{1}(y^{(i)} = j) x^{(i)}$
8       if $\mathbb{1}(y = y_{\text{old}})$
9           break
10  return $\mu, y$
```

**Study Question:** Why do we have the "break" statement on line 9? Could the clustering improve if we ran it for more iterations after this point? Has it converged?

The for-loop over the $n$ datapoints assigns each datapoint to the nearest cluster center. The for-loop over the $k$ clusters updates the cluster center to be the mean of all datapoints currently assigned to that cluster. As suggested above, it can be shown that this algorithm reduces the loss in Eq. 7.1 on each iteration, until it converges to a local minimum of the loss.

It's like classification except it *picked* what the classes are rather than being given examples of what the classes are.

### 7.2.2 Using gradient descent to minimize k-means objective

We can also use gradient descent to optimize the k-means objective. To show how to apply gradient descent, we first rewrite the objective as a differentiable function *only of* $\mu$:

$$L(\mu) = \sum_{i=1}^{n} \min_j \left\| x^{(i)} - \mu^{(j)} \right\|^2 \quad . \tag{7.2}$$

The k-means algorithm presented above is a form of *block coordinate descent*, rather than gradient descent. For certain problems, and in particular k-means, this method can converge faster than gradient descent.

$L(\mu)$ is the value of the k-means loss given that we pick the *optimal* assignments of the datapoints to cluster means (that's what the $\min_j$ does). Now we can use the gradient $\frac{\partial L(\mu)}{\partial \mu}$ to find the values for $\mu$ that achieve minimum loss when cluster assignments are optimal. Finally, we read off the optimal cluster assignments, given the optimized $\mu$, just by assigning datapoints to their nearest cluster mean:

$$y^{(i)} = \arg\min_j \left\| x^{(i)} - \mu^{(j)} \right\|^2 \quad . \tag{7.3}$$

$L(\mu)$ is a smooth function except with kinks where the nearest cluster changes; that means it's differentiable almost everywhere, which in practice is sufficient for us to apply gradient descent.

This procedure yields a local minimum of Eq. 7.1, as does the standard k-means algorithm we presented (though they might arrive at different solutions). It might not be the global optimum since the objective is not convex (due to $\min_j$, as the minimum of multiple convex functions is not necessarily convex).

### 7.2.3 Importance of initialization

The standard k-means algorithm, as well as the variant that uses gradient descent, both are only guaranteed to converge to a local minimum, not necessarily the global minimum of the loss. Thus the answer we get out depends on how we initialize the cluster means. Figure 7.4 is an example of a different initialization on our toy data, which results in a worse converged clustering:

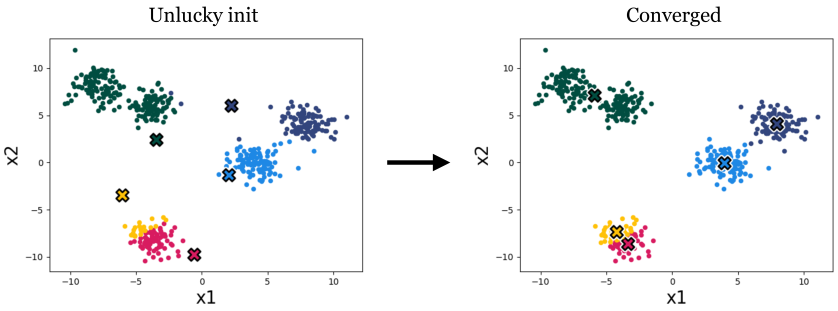


Figure 7.4: With the initialization of the means to the left, the yellow and red means end up splitting what perhaps should be one cluster in half.

A variety of methods have been developed to pick good initializations (see, for example, the *k-means++* algorithm). One simple option is to run the standard k-means algorithm multiple times, with different random initial conditions, and then pick from these the clustering that achieves the lowest k-means loss.

### 7.2.4 Importance of k

A very important parameter in cluster algorithms is the number of clusters we are looking for. Some advanced algorithms can automatically infer a suitable number of clusters, but

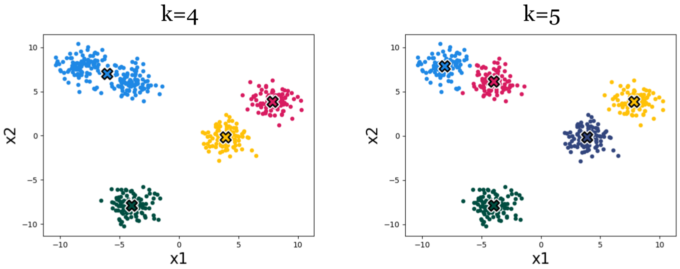


Figure 7.5: Example of k-means run on our toy data, with two different values of k. Setting k=4, on the left, results in one cluster being merged, compared to setting k=5, on the right. Which clustering do you think is better? How could you decide?

most of the time, like with k-means, we will have to pick k – it's a hyperparameter of the algorithm.

Figure 7.5 shows an example of the effect. Which result looks more correct? It can be hard to say! Using higher k we get more clusters, and with more clusters we can achieve lower within-cluster variance – the k-means objective will never increase, and will typically strictly decrease as we increase k. Eventually, we can increase k to equal the total number of datapoints, so that each datapoint is assigned to its own cluster. Then the k-means objective is zero, but the clustering reveals nothing. Clearly, then, we cannot use the k-means objective itself to choose the best value for k. In Section 7.3, we will discuss some ways of evaluating the success of clustering beyond its ability to minimize the k-means objective, and it's with these sorts of methods that we might decide on a proper value of k.

Alternatively, you may be wondering: why bother picking a single k? Wouldn't it be nice to reveal a *hierarchy* of clusterings of our data, showing both coarse and fine groupings? Indeed *hierarchical clustering* is another important class of clustering algorithms, beyond k-means. These methods can be useful for discovering tree-like structure in data, and they work a bit like this: initially a coarse split/clustering of the data is applied at the root of the tree, and then as we descend the tree we split and cluster the data in ever more fine-grained ways. A prototypical example of hierarchical clustering is to discover a taxonomy of life, where creatures may be grouped at multiple granularities, from species to families to kingdoms. You may find a suite of clustering algorithms in SKLEARN's cluster module.

### 7.2.5 k-means in feature space

Clustering algorithms group data based on a notion of *similarity*, and thus we need to define a *distance metric* between datapoints. This notion will also be useful in other machine learning approaches, such as nearest-neighbor methods that we see in Chapter 13. In k-means and other methods, our choice of distance metric can have a big impact on the results we will find.

Our k-means algorithm uses the Euclidean distance, i.e., $\left\|x^{(i)} - \mu^{(j)}\right\|$, with a loss function that is the square of this distance. We can modify k-means to use different distance metrics, but a more common trick is to stick with Euclidean distance but measured in a *feature space*. Just like we did for regression and classification problems, we can define a feature map from the data to a nicer feature representation, $\phi(x)$, and then apply k-means to cluster the data in the feature space.

In fact, using a simple distance metric in feature space can be equivalent to using a more sophisticated distance metric in the data space, and this trick forms the basis of *kernel methods*, which you can learn about in more advanced machine learning classes.

As a simple example, suppose we have two-dimensional data that is very stretched out in the first dimension and has less dynamic range in the second dimension. Then we may want to scale the dimensions so that each has similar dynamic range, prior to clustering. We could use standardization, like we did in Chapter 5.

If we want to cluster more complex data, like images, music, chemical compounds, etc., then we will usually need more sophisticated feature representations. One common practice these days is to use feature representations learned with a neural network. For example, we can use an autoencoder to compress images into feature vectors, then cluster those feature vectors.

## 7.3 How to evaluate clustering algorithms

One of the hardest aspects of clustering is knowing how to evaluate it. This is actually a big issue for all unsupervised learning methods, since we are just looking for patterns in the data, rather than explicitly trying to predict target values (which was the case with supervised learning).

Remember, evaluation metrics are *not* the same as loss functions, so we can't just measure success by looking at the k-means loss. In prediction problems, it is critical that the evaluation is on a held-out test set, while the loss is computed over training data. If we evaluate on training data we cannot detect overfitting. Something similar is going on with the example in Section 7.2.4, where setting k to be too large can precisely "fit" the data (minimize the loss), but yields no general insight.

One way to evaluate our clusters is to look at the **consistency** with which they are found when we run on different subsamples of our training data, or with different hyperparameters of our clustering algorithm (e.g., initializations). For example, if running on several bootstrapped samples (random subsets of our data) results in very different clusters, it should call into question the validity of any of the individual results.

If we have some notion of what **ground truth** clusters should be, e.g., a few data points that we know should be in the same cluster, then we can measure whether or not our discovered clusters group these examples correctly.

Clustering is often used for **visualization** and **interpretability**, to make it easier for humans to understand the data. Here, human judgment may guide the choice of clustering algorithm. More quantitatively, discovered clusters may be used as input to **downstream tasks**. For example, as we saw in the lab, we may fit a different regression function on the data within each cluster. Figure 7.6 gives an example where this might be useful. In cases like this, the success of a clustering algorithm can be indirectly measured based on the success of the downstream application (e.g., does it make the downstream predictions more accurate).

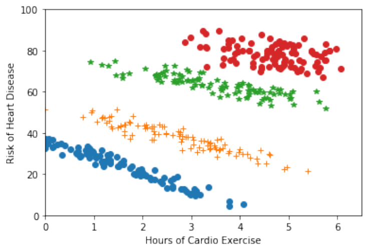


Figure 7.6: Averaged across the whole population, risk of heart disease positively correlates with hours of exercise. However, if we cluster the data, we can observe that there are four subgroups of the population which correspond to different age groups, and within each subgroup the correlation is negative. We can make better predictions, and better capture the presumed true effect, if we cluster this data and then model the trend in each cluster separately.

# CHAPTER 8

## Convolutional Neural Networks

So far, we have studied what are called *fully connected* neural networks, in which all of the units at one layer are connected to all of the units in the next layer. This is a good arrangement when we don't know anything about what kind of mapping from inputs to outputs we will be asking the network to learn to approximate. But if we *do* know something about our problem, it is better to build it into the structure of our neural network. Doing so can save computation time and significantly diminish the amount of training data required to arrive at a solution that generalizes robustly.

One very important application domain of neural networks, where the methods have achieved an enormous amount of success in recent years, is signal processing. Signals might be spatial (in two-dimensional camera images or three-dimensional depth or CAT scans) or temporal (speech or music). If we know that we are addressing a signal-processing problem, we can take advantage of *invariant* properties of that problem. In this chapter, we will focus on two-dimensional spatial problems (images) but use one-dimensional ones as a simple example. In a later chapter, we will address temporal problems.

Imagine that you are given the problem of designing and training a neural network that takes an image as input, and outputs a classification, which is positive if the image contains a cat and negative if it does not. An image is described as a two-dimensional array of *pixels*, each of which may be represented by three integer values, encoding intensity levels in red, green, and blue color channels.

A *pixel* is a "picture element."

There are two important pieces of prior structural knowledge we can bring to bear on this problem:

- **Spatial locality:** The set of pixels we will have to take into consideration to find a cat will be near one another in the image.
- **Translation invariance:** The pattern of pixels that characterizes a cat is the same no matter where in the image the cat occurs.

So, for example, we won't have to consider some combination of pixels in the four corners of the image, in order to see if they encode cat-ness.

Cats don't look different if they're on the left or the right side of the image.

We will design neural network structures that take advantage of these properties.

## 8.1 Filters

We begin by discussing *image filters*. An image filter is a function that takes in a local spatial neighborhood of pixel values and detects the presence of some pattern in that data.

Unfortunately in AI/ML/CS/Math, the word "filter" gets used in many ways: in addition to the one we describe here, it can describe a temporal process (in fact, our moving averages are a kind of filter) and even a somewhat esoteric algebraic structure.

Let's consider a very simple case to start, in which we have a 1-dimensional binary "image" and a filter $F$ of size two. The filter is a vector of two numbers, which we will move along the image, taking the dot product between the filter values and the image values at each step, and aggregating the outputs to produce a new image.

Let $X$ be the original image, of size $d$; then pixel $i$ of the the output image is specified by

$$Y_i = F \cdot (X_{i-1}, X_i) \ .$$

To ensure that the output image is also of dimension $d$, we will generally "pad" the input image with 0 values if we need to access pixels that are beyond the bounds of the input image. This process of applying the filter to the image to create a new image is called "convolution."

And filters are also sometimes called *convolutional kernels*.

If you are already familiar with what a convolution is, you might notice that this definition corresponds to what is often called a correlation and not to a convolution. Indeed, correlation and convolution refer to different operations in signal processing. However, in the neural networks literature, most libraries implement the correlation (as described in this chapter) but call it convolution. The distinction is not significant; in principle, if convolution is required to solve the problem, the network could learn the necessary weights. For a discussion of the difference between convolution and correlation and the conventions used in the literature you can read Section 9.1 in this excellent book: `https://www.deeplearningbook.org`.

Here is a concrete example. Let the filter $F_1 = (-1, +1)$. Then given the image in the first line below, we can convolve it with filter $F_1$ to obtain the second image. You can think of this filter as a detector for "left edges" in the original image—to see this, look at the places where there is a 1 in the output image, and see what pattern exists at that position in the input image. Another interesting filter is $F_2 = (-1, +1, -1)$. The third image (the last line below) shows the result of convolving the first image with $F_2$, where we see that the output pixel $i$ corresponds to when the center of $F_2$ is aligned at input pixel $i$.

**Study Question:** Convince yourself that filter $F_2$ can be understood as a detector for isolated positive pixels in the binary image.

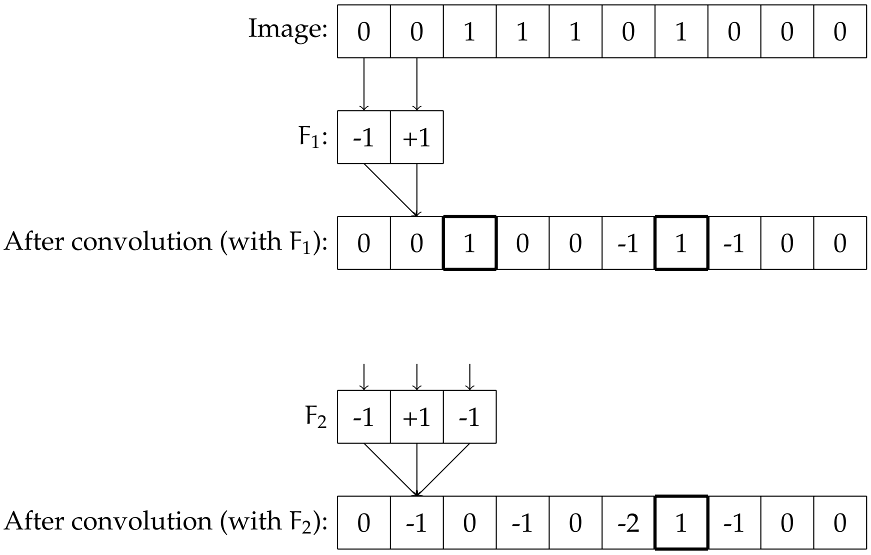


Image: 0 0 1 1 1 0 1 0 0 0

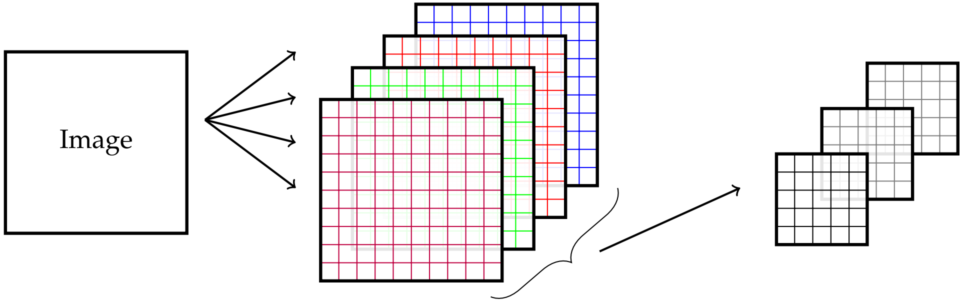
$F_1$: -1 +1

After convolution (with $F_1$): 0 0 1 0 0 -1 1 -1 0 0

$F_2$ -1 +1 -1

After convolution (with $F_2$): 0 -1 0 -1 0 -2 1 -1 0 0

Two-dimensional versions of filters like these are thought to be found in the visual cortex of all mammalian brains. Similar patterns arise from statistical analysis of natural images. Computer vision people used to spend a lot of time hand-designing *filter banks*. A filter bank is a set of sets of filters, arranged as shown in the diagram below.



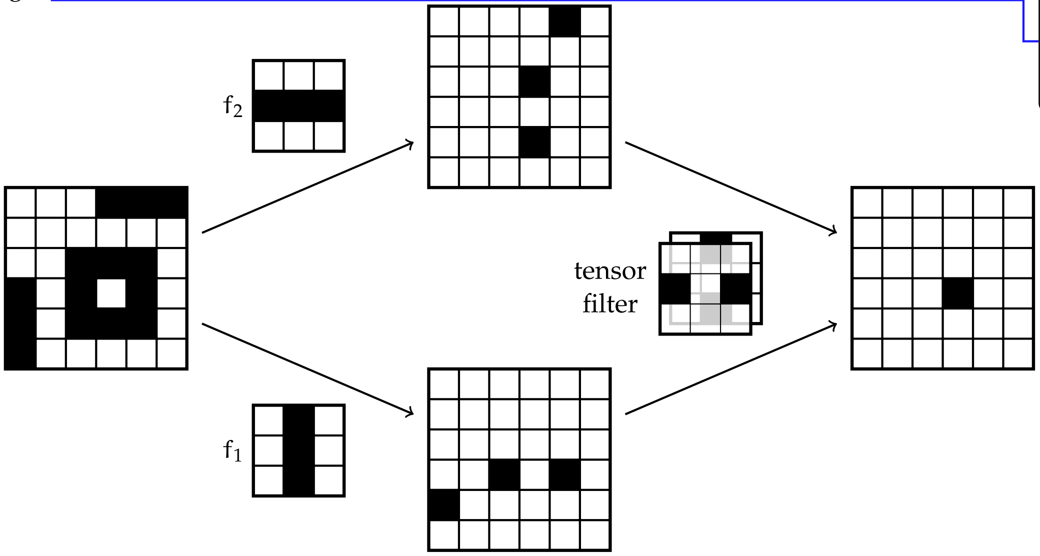
All of the filters in the first group are applied to the original image; if there are $k$ such filters, then the result is $k$ new images, which are called *channels*. Now imagine stacking all these new images up so that we have a cube of data, indexed by the original row and column indices of the image, as well as by the channel. The next set of filters in the filter bank will generally be *three-dimensional*: each one will be applied to a sub-range of the row and column indices of the image and to all of the channels.

These 3D chunks of data are called *tensors*. The algebra of tensors is fun, and a lot like matrix algebra, but we won't go into it in any detail.

There are now many useful neural-network software packages, such as *TensorFlow* and *PyTorch* that make operations on tensors easy.

Here is a more complex example of two-dimensional filtering. We have two $3 \times 3$ filters in the first layer, $f_1$ and $f_2$. You can think of each one as "looking" for three pixels in a row, $f_1$ vertically and $f_2$ horizontally. Assuming our input image is $n \times n$, then the result of filtering with these two filters is an $n \times n \times 2$ tensor. Now we apply a tensor filter (hard to draw!) that "looks for" a combination of two horizontal and two vertical bars (now represented by individual pixels in the two channels), resulting in a single final $n \times n$ image.

When we have a color image as input, we treat it as having three channels, and hence as an $n \times n \times 3$ tensor.



We are going to design neural networks that have this structure. Each "bank" of the filter bank will correspond to a neural-network layer. The numbers in the individual filters will be the "weights" (plus a single additive bias or offset value for each filter) of the network, that we will train using gradient descent. What makes this interesting and powerful (and somewhat confusing at first) is that the same weights are used many many times in the computation of each layer. This *weight sharing* means that we can express a transformation on a large image with relatively few parameters; it also means we'll have to take care in figuring out exactly how to train it!

We will define a filter layer $l$ formally with:

For simplicity, we are assuming that all images and filters are square (having the same number of rows and columns). That is in no way necessary, but is usually fine and definitely simplifies our notation.

- *number* of filters $m^l$;
- *size* of one filter is $k^l \times k^l \times m^{l-1}$ plus 1 bias value (for this one filter);
- *stride* $s^l$ is the spacing at which we apply the filter to the image; in all of our examples so far, we have used a stride of 1, but if we were to "skip" and apply the filter only at odd-numbered indices of the image, then it would have a stride of two (and produce a resulting image of half the size);
- *input tensor size* $n^{l-1} \times n^{l-1} \times m^{l-1}$
- *padding*: $p^l$ is how many extra pixels – typically with value 0 – we add around the edges of the input. For an input of size $n^{l-1} \times n^{l-1} \times m^{l-1}$, our new effective input size with padding becomes $(n^{l-1} + 2 \cdot p^l) \times (n^{l-1} + 2 \cdot p^l) \times m^{l-1}$.

This layer will produce an output tensor of size $n^l \times n^l \times m^l$, where $n^l = \lceil (n^{l-1} + 2 \cdot p^l - (k^l - 1))/s^l \rceil$.[1] The weights are the values defining the filter: there will be $m^l$ different $k^l \times k^l \times m^{l-1}$ tensors of weight values; plus each filter may have a bias term, which means there is one more weight value per filter. A filter with a bias operates just like the filter examples above, except we add the bias to the output. For instance, if we incorporated a bias term of 0.5 into the filter $F_2$ above, the output would be $(-0.5, 0.5, -0.5, 0.5, -1.5, 1.5, -0.5, 0.5)$ instead of $(-1, 0, -1, 0, -2, 1, -1, 0)$.

This may seem complicated, but we get a rich class of mappings that exploit image structure and have many fewer weights than a fully connected layer would.

**Study Question:** How many weights are in a convolutional layer specified as above?

**Study Question:** If we used a fully-connected layer with the same size inputs and outputs, how many weights would it have?

## 8.2 Max pooling

It is typical to structure filter banks into a *pyramid*, in which the image sizes get smaller in successive layers of processing. The idea is that we find local patterns, like bits of edges in the early layers, and then look for patterns in those patterns, etc. This means that, effectively, we are looking for patterns in larger pieces of the image as we apply successive filters. Having a stride greater than one makes the images smaller, but does not necessarily aggregate information over that spatial range.

Both in engineering and in nature

Another common layer type, which accomplishes this aggregation, is *max pooling*. A max pooling layer operates like a filter, but has no weights. *You can think of it as purely functional, like a ReLU in a fully connected network.* It has a filter size, as in a filter layer, but simply returns the maximum value in its field. Usually, we apply max pooling with the following traits:

We sometimes use the term *receptive field* or just *field* to mean the area of an input image that a filter is being applied to.

- stride $> 1$, so that the resulting image is smaller than the input image; and
- $k \geqslant$ stride, so that the whole image is covered.

[1] Recall that $\lceil \cdot \rceil$ is the *ceiling* function; it returns the smallest integer greater than or equal to its input. E.g., $\lceil 2.5 \rceil = 3$ and $\lceil 3 \rceil = 3$.

As a result of applying a max pooling layer, we don't keep track of the precise location of a pattern. This helps our filters to learn to recognize patterns independent of their location.

Consider a max pooling layer where both the strides and $k$ are set to be 2. This would map a $64 \times 64 \times 3$ image to a $32 \times 32 \times 3$ image. Note that max pooling layers do not have additional bias or offset values.

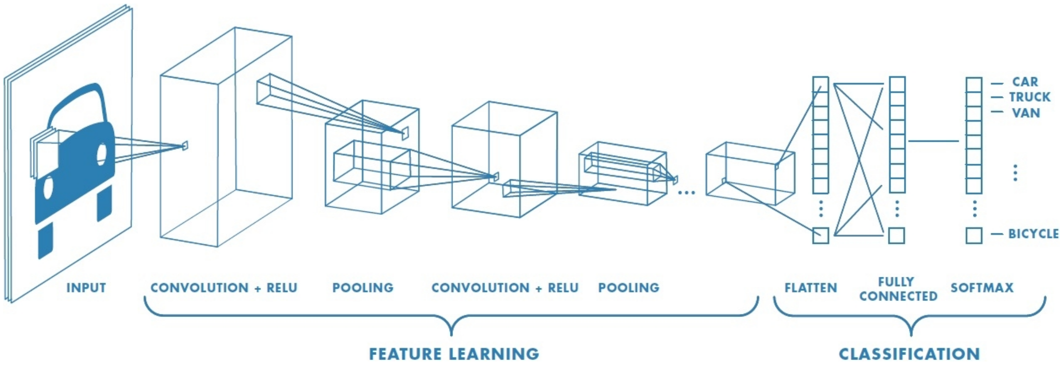
**Study Question:** Maximilian Poole thinks it would be a good idea to add two max pooling layers of size $k$, one right after the other, to their network. What single layer would be equivalent?

One potential concern about max-pooling layers is that they actually don't completely preserve translation invariance. If you do max-pooling with a stride other than 1 (or just pool over the whole image size), then shifting the pattern you are hoping to detect within the image by a small amount can change the output of the max-pooling layer substantially, just because there are discontinuities induced by the way the max-pooling window matches up with its input image. Here is an interesting paper that illustrates this phenomenon clearly and suggests that one should first do max-pooling with a stride of 1, then do "downsampling" by averaging over a window of outputs.

https://arxiv.org/pdf/1904.11486.pdf

## 8.3 Typical architecture

Here is the form of a typical convolutional network:



The "depth" dimension in the layers shown as cuboids corresponds to the number of channels in the output tensor. (Figure source: https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html)

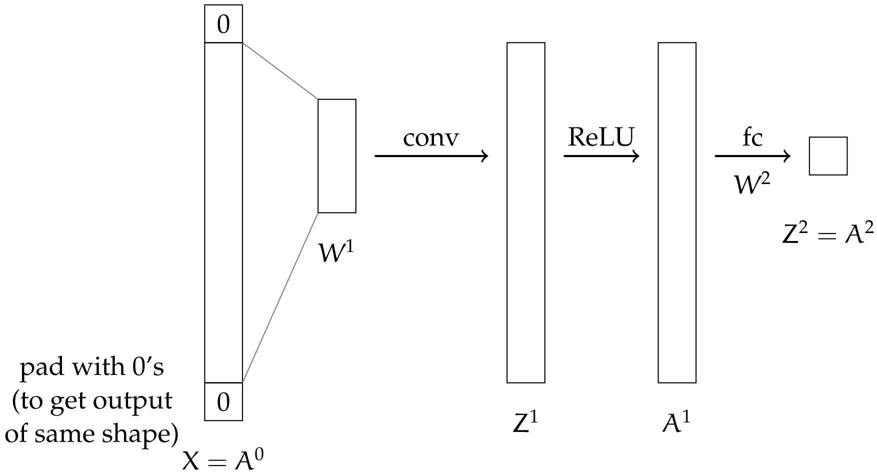
At the end of each filter layer, we typically apply a ReLU activation function. There may be multiple filter plus ReLU layers. Then we have a max pooling layer. Then we have some more filter + ReLU layers. Then we have max pooling again. Once the output is down to a relatively small size, there is typically a last fully-connected layer, leading into an activation function such as softmax that produces the final output. The exact design of these structures is an art—there is not currently any clear theoretical (or even systematic empirical) understanding of how these various design choices affect overall performance of the network.

The critical point for us is that this is all just a big neural network, which takes an input and computes an output. The mapping is a differentiable function of the weights, which means we can adjust the weights to decrease the loss by performing gradient descent, and we can compute the relevant gradients using back-propagation!

Well, techinically the derivative does not exist at every point, both because of the ReLU and the max pooling operations, but we ignore that fact.

## 8.4 Backpropagation in a simple CNN

Let's work through a *very* simple example of how back-propagation can work on a convolutional network. The architecture is shown below. Assume we have a one-dimensional single-channel image $X$ of size $n \times 1 \times 1$, and a single filter $W^1$ of size $k \times 1 \times 1$ (where we omit the filter bias) for the first convolutional operation denoted "conv" in the figure below. Then we pass the intermediate result $Z^1$ through a ReLU layer to obtain the activation $A^1$, and finally through a fully-connected layer with weights $W^2$, denoted "fc" below, with no additional activation function, resulting in the output $A^2$.



For simplicity assume $k$ is odd, let the input image $X = A^0$, and assume we are using squared loss. Then we can describe the forward pass as follows:

$$\begin{aligned}
Z_i^1 &= {W^1}^T A^0_{[i-\lfloor k/2 \rfloor : i+\lfloor k/2 \rfloor]} \\
A^1 &= \mathrm{ReLU}(Z^1) \\
A^2 &= Z^2 = {W^2}^T A^1 \\
\mathcal{L}_{square}(A^2, y) &= (A^2 - y)^2
\end{aligned}$$

**Study Question:** Assuming a stride of 1, for a filter of size $k$, how much padding do we need to add to the top and bottom of the image? We see one zero at the top and bottom in the figure just above; what filter size is implicitly being shown in the figure? (Recall the padding is for the sake of getting an output the same size as the input.)

### 8.4.1 Weight update

How do we update the weights in filter $W^1$?

$$\frac{\partial \text{loss}}{\partial W^1} = \frac{\partial Z^1}{\partial W^1} \frac{\partial A^1}{\partial Z^1} \frac{\partial \text{loss}}{\partial A^1}$$

- $\partial Z^1 / \partial W^1$ is the $k \times n$ matrix such that $\partial Z_i^1 / \partial W_j^1 = X_{i - \lfloor k/2 \rfloor + j - 1}$. So, for example, if $i = 10$, which corresponds to column 10 in this matrix, which illustrates the dependence of pixel 10 of the output image on the weights, and if $k = 5$, then the elements in column 10 will be $X_8, X_9, X_{10}, X_{11}, X_{12}$.

- $\partial A^1/\partial Z^1$ is the $n \times n$ diagonal matrix such that

$$\partial A^1_i/\partial Z^1_i = \begin{cases} 1 & \text{if } Z^1_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

- $\partial \text{loss}/\partial A^1 = (\partial \text{loss}/\partial A^2)(\partial A^2/\partial A^1) = 2(A^2 - y)W^2$, an $n \times 1$ vector

Multiplying these components yields the desired gradient, of shape $k \times 1$.

### 8.4.2 Max pooling

One last point is how to handle back-propagation through a max-pooling operation. Let's study this via a simple example. Imagine

$$y = \max(a_1, a_2) \ \ ,$$

where $a_1$ and $a_2$ are each computed by some network. Consider doing back-propagation through the maximum. First consider the case where $a_1 > a_2$. Then the error value at $y$ is propagated back entirely to the network computing the value $a_1$. The weights in the network computing $a_1$ will ultimately be adjusted, and the network computing $a_2$ will be untouched.

**Study Question:** What is $\nabla_{(x,y)} \max(x, y)$ ?

# CHAPTER 9

# Transformers

Transformers are a very recent family of architectures that have revolutionized fields like natural language processing (NLP), image processing, and multi-modal generative AI.

Transformers were originally introduced in the field of NLP in 2017, as an approach to process and understand human language. Human language is inherently sequential in nature (e.g., characters form words, words form sentences, and sentences form paragraphs and documents). Prior to the advent of the transformers architecture, recurrent neural networks (RNNs) briefly dominated the field for their ability to process sequential information (RNNs are described in Appendix C for reference). However, RNNs, like many other architectures, processed sequential information in an iterative/sequential fashion, whereby each item of a sequence was individually processed one after another. Transformers offer many advantages over RNNs, including their ability to process all items in a sequence in a *parallel* fashion (as do CNNs).

Like CNNs, transformers factorize the signal processing problem into stages that involve independent and identically processed chunks. However, they also include layers that mix information across the chunks, called *attention layers*, so that the full pipeline can model dependencies between the chunks.

In this chapter, we describe transformers from the bottom up. We start with the idea of embeddings and tokens (Section 9.1). We then describe the attention mechanism (Section 9.2). And finally we then assemble all these ideas together to arrive at the full transformer architecture in Section 9.3.

## 9.1 Vector embeddings and tokens

Before we can understand the attention mechanism in detail, we need to first introduce a new data structure and a new way of thinking about neural processing for language.

The field of NLP aims to represent words with vectors (aka *word embeddings*) such that they capture semantic meaning. More precisely, the degree to which any two words are related in the 'real-world' to us humans should be reflected by their corresponding vectors (in terms of their numeric values). So, words such as 'dog' and 'cat' should be represented by vectors that are more similar to one another than, say, 'cat' and 'table' are. Nowadays, it's also typical for every individual occurrence of a word to have its own distinct representation/vector. So, a story about a dog may mention the word 'dog' a dozen times, with each vector being slightly different based on its context in the sentence and story at large.

To measure how similar any two word embeddings are (in terms of their numeric values) it is common to use *cosine similarity* as the metric:

$$\frac{u^T v}{|u|\,|v|} = \cos\langle u, v\rangle\,, \tag{9.1}$$

where $|u|$ and $|v|$ are the lengths of the vectors, and $\langle u, v\rangle$ is the angle between $u$ and $v$. The cosine similarity is $+1$ when $u = v$, zero when the two vectors are perpendicular to each other, and $-1$ when the two vectors are diametrically opposed to each other. Thus, higher values correspond to vectors that are numerically more similar to each other.

While word embeddings – and various approaches to create them – have existed for decades, the first approach that produced astonishingly effective word embeddings was *word2vec* in 2012. This revolutionary approach was the first highly-successful approach of applying deep learning to NLP, and it enabled all subsequent progress in the field, including Transformers. The details of *word2vec* are beyond the scope of this course, but we note two facts: (1) it created a single word embedding for each distinct word in the training corpus (not on a per-occurrence basis); (2) it produced word embeddings that were so useful, many relationships between the vectors corresponded with real-world semantic relatedness. For example, when using *Euclidean distance* as a distance metric between two vectors, word2vec produced word embeddings with properties such as (where $v_{\texttt{word}}$ is the vector for `word`):

$$v_{\texttt{paris}} - v_{\texttt{france}} + v_{\texttt{italy}} \approx v_{\texttt{rome}} \tag{9.2}$$

This corresponds with the real-world property that Paris is to France what Rome is to Italy. This incredible finding existed not only for geographic words but all sorts of real-world concepts in the vocabulary. Nevertheless, to some extent, the exact values in each embedding is arbitrary, and what matters most is the holistic relation between all embeddings, along with how performant/useful they are for the exact task that we care about.

For example, an embedding may be considered good if it accurately captures the conditional probability for a given word to appear next in a sequence of words. You probably have a good idea of what words might typically fill in the blank at the end of this sentence:

After the rain, the grass was ___________

Or a model could be built that tries to correctly predict words in the middle of sentences:

The child fell ___________ during the long car ride

The model can be built by minimizing a loss function that penalizes incorrect word guesses, and rewards correct ones. This is done by training a model on a very large corpus of written material, such as all of Wikipedia, or even all the accessible digitized written materials produced by humans.

While we will not dive into the full details of *tokenization*, the high-level idea is straightforward: the individual inputs of data that are represented and processed by a model are referred to as *tokens*. And, instead of processing each word as a whole, words are typically split into smaller, meaningful pieces (akin to syllables). Thus, when we refer to tokens, know that we're referring to each individual input, and that in practice, nowadays, they tend to be sub-words (e.g., the word 'talked' may be split into two tokens, 'talk' and 'ed').

## 9.2 Query, key, value, and attention

Attention is a strategy for processing global information efficiently, focusing just on the parts of the signal that are most salient to the task at hand.

What we present below is the so-called "dot-product attention" mechanism; there can be other variants that involve more complex attention functions

It might help our understanding of the "attention" mechanism to think about a dictionary look-up scenario. Consider a dictionary with keys $k$ mapping to some values $v(k)$. For example, let $k$ be the name of some foods, such as `pizza`, `apple`, `sandwich`, `donut`, `chili`, `burrito`, `sushi`, `hamburger`, . . .. The corresponding values may be information about the food, such as where it is available, how much it costs, or what its ingredients are.

Suppose that instead of looking up foods by a specific name, we wanted to query by cuisine, e.g., "`mexican`" foods. Clearly, we cannot simply look for the word "`mexican`" among the dictionary keys, since that word is not a food. What does work is to utilize again the idea of finding "similarity" between vector embeddings of the query and the keys. The end result we'd hope to get, is a probability distribution over the foods, $p(k|q)$ indicating which are best matches for a given query $q$. With such a distribution, we can look for keys that are semantically close to the given query.

More concretely, to get such distribution, we follow these steps: First, embed the word we are interested in ("`mexican`" in our example) into a so-called query vector, denoted simply as $q \in \mathbb{R}^{d_k \times 1}$ where $d_k$ is the embedding dimension.

Next, suppose our given dictionary has $n$ number of entries/entries, we embed each one of these into a so-called key vector. In particular, for each of the $j^{th}$ entry in the dictionary, we produce a $k_j \in \mathbb{R}^{d_k \times 1}$ key vector, where $j = 1, 2, 3, \ldots, n$.

We can then obtain the desired probability distribution using a softmax (see Chapter 6) applied to the inner-product between the key and query:

$$p(k|q) = \mathtt{softmax}\left([q^T k_1; q^T k_2; q^T k_3; \ldots, q^T k_n]\right)$$

This vector-based lookup mechanism has come to be known as "attention" in the sense that $p(k|q)$ is a conditional probability distribution that says how much attention should be given to the key $k_j$ for a given query $q$.

In other words, the conditional probability distribution $p(k|q)$ gives the "attention weights," and the weighted average value

$$\sum_j p(k_j|q)\, v_j \tag{9.3}$$

is the "attention output."

The meaning of this weighted average value may be ambiguous when the values are just words. However, the attention output really becomes meaningful when the values are projected in some semantic embedding space (and such projections are typically done in transformers via learned embedding weights).

The same weighted-sum idea generalizes to multiple query, key, and values. In particular, suppose there are $n_q$ number of queries, $n_k$ number of keys (and therefore $n_k$ number of values), one can compute an attention matrix

$$\alpha = \begin{bmatrix} \text{softmax}\left(\begin{bmatrix} q_1^\top k_1 & q_1^\top k_2 & \cdots & q_1^\top k_{n_k} \end{bmatrix} / \sqrt{d_k}\right) \\ \text{softmax}\left(\begin{bmatrix} q_2^\top k_1 & q_2^\top k_2 & \cdots & q_2^\top k_{n_k} \end{bmatrix} / \sqrt{d_k}\right) \\ \vdots \\ \text{softmax}\left(\begin{bmatrix} q_{n_q}^\top k_1 & q_{n_q}^\top k_2 & \cdots & q_{n_q}^\top k_{n_k} \end{bmatrix} / \sqrt{d_k}\right) \end{bmatrix} \tag{9.4}$$

Here, softmax$_j$ is a softmax over the $n_k$-dimensional vector indexed by $j$, so in Eq. 9.10 this means a softmax computed over keys. In this equation, the normalization by $\sqrt{d_k}$ is done to reduce the magnitude of the dot product, which would otherwise grow undesirably large with increasing $d_k$, making it difficult for (overall) training.

Let $\alpha_{ij}$ be the entry in ith row and jth column in the attention matrix $\alpha$. Then $\alpha_{ij}$ helps answer the question "which tokens $x^{(j)}$ help the most with predicting the corresponding output token $y^{(i)}$?" The attention output is given by a weighted sum over the values:

$$y^{(i)} = \sum_{j=1}^{n} \alpha_{ij} v_j$$

### 9.2.1 Self Attention

Self-attention is an attention mechanism where the keys, values, and queries are all generated from the same input.

At a very high level, typical transformer with self-attention layers maps $\mathbb{R}^{n \times d} \longrightarrow \mathbb{R}^{n \times d}$. In particular, the transformer takes in data (a sequence of tokens) $X \in \mathbb{R}^{n \times d}$ and for each token $x^{(i)} \in \mathbb{R}^{d \times 1}$, it computes (via learned projection, to be discussed in Section 9.3.1), a query $q_i \in \mathbb{R}^{d_q \times 1}$, key $k_i \in \mathbb{R}^{d_k \times 1}$, and value $v_i \in \mathbb{R}^{d_v \times 1}$. In practice, $d_q = d_k = d_v$ and we often denote all three embedding dimension via a unified $d_k$.

Note that $d_k$ differs from $d$: $d$ is the dimension of raw input token $\in \mathbb{R}^{d_q \times 1}$

The self-attention layer then takes in these query, key, and values, and compute a self-attention matrix

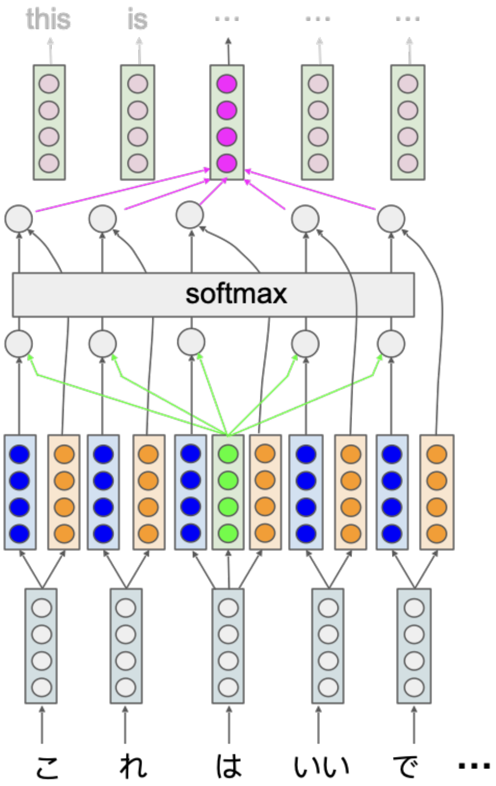
$$\alpha = \begin{bmatrix} \text{softmax}\left(\begin{bmatrix} q_1^\top k_1 & q_1^\top k_2 & \cdots & q_1^\top k_n \end{bmatrix} / \sqrt{d_k}\right) \\ \text{softmax}\left(\begin{bmatrix} q_2^\top k_1 & q_2^\top k_2 & \cdots & q_2^\top k_n \end{bmatrix} / \sqrt{d_k}\right) \\ \vdots \\ \text{softmax}\left(\begin{bmatrix} q_n^\top k_1 & q_n^\top k_2 & \cdots & q_n^\top k_n \end{bmatrix} / \sqrt{d_k}\right) \end{bmatrix} \tag{9.5}$$

Comparing this self-attention matrix with the attention matrix described in Equation 9.10, we notice the only difference lies in the dimensions: since in self-attention, the query, key, and value all come from the same input, we have $n_q = n_k = n_v$, and we often denote all three with a unified $n$.

The self-attention output is then given by a weighted sum over the values:

$$y^{(i)} = \sum_{j=1}^{n} \alpha_{ij} v_j \tag{9.6}$$

This diagram below shows (only) the middle input token generating a query that is then combined with the keys computed with all tokens to generate the attention weights via a softmax. The output of the softmax is then combined with values computed from all tokens, to generate the attention output corresponding to the middle input token. Repeating this for each input token then generates the output.



**Study Question:** We have five colored tokens in the diagram above (gray, navy blue, orange, lime green, magenta). Could you read off the diagram the correspondence between the color and input, query, value, output?

Note that the size of the output is the same as the size of the input. Also, observe that there is no apparent notion of ordering of the input words in the depicted structure. Positional information can be added by encoding a number for token (giving say, the token's position relative to the start of the sequence) into the vector embedding of each token. And note that a given query need not pay attention to all other tokens in the input; in this example, the token used for the query is not used for a key or value.

More generally, a *mask* may be applied to limit which tokens are used in the attention computation. For example, one common mask limits the attention computation to tokens that occur previously in time to the one being used for the query. This prevents the attention mechanism from "looking ahead" in scenarios where the transformer is being used to generate one token at a time.

Each self-attention stage is trained to have key, value, and query embeddings that lead it to pay specific attention to some particular feature of the input. We generally want to pay attention to many different kinds of features in the input; for example, in translation one feature might be be the verbs, and another might be objects or subjects. A transformer utilizes multiple instances of self-attention, each known as an "attention head," to allow combinations of attention paid to many different features.

## 9.3 Transformers

A transformer is the composition of a number of transformer blocks, each of which has multiple attention heads. At a very high-level, the goal of a transformer block is to output a really rich, useful representation for each input token, all for the sake of being high-performant for whatever task the model is trained to learn.

Rather than depicting the transformer graphically, it is worth returning to the beauty of the underlying equations[1].

### 9.3.1 Learned embedding

For simplicity, we assume the transformer internally uses self-attention. Full general attention layers work out similarly.

Formally, a transformer block is a parameterized function $f_\theta$ that maps $\mathbb{R}^{n\times d} \to \mathbb{R}^{n\times d}$, where the input data $X \in \mathbb{R}^{n\times d}$ is often represented as a sequence of $n$ tokens, with each token $x^{(i)} \in \mathbb{R}^{d\times 1}$.

Three projection matrices (weights) $W_q, W_k, W_v$ are to be learned, such that, for each token $x^{(i)} \in \mathbb{R}^{d\times 1}$, we produce 3 distinct vectors: a query vector $q_i = W_q^T x^{(i)}$; a key vector $k_i = W_k^T x^{(i)}$; and a value vector $v_i = W_v^T x^{(i)}$. All three of these vectors are in $\mathbb{R}^{d_k\times 1}$ and the learned weights are $W_q, W_k, W_v \in \mathbb{R}^{d\times d_k}$.

If we stack these $n$ query, key, value vectors into matrix- form, such that $Q \in \mathbb{R}^{n\times d_k}$, $K \in \mathbb{R}^{n\times d_k}$, and $V \in \mathbb{R}^{n\times d_k}$, then we can more compactly write out the learned transformation from the sequence of input token $X$:

$$\begin{aligned} Q &= XW_q \\ K &= XW_k \\ V &= XW_v \end{aligned}$$

This $Q, K, V$ triple can then be used to produce one (self)attention-layer output. One such layer is called one "attention head."

One can have more than one "attention head," such that: the queries, keys, and values are embedded via encoding matrices:

$$\begin{aligned} Q^{(h)} &= XW_{h,q} && (9.7) \\ K^{(h)} &= XW_{h,k} && (9.8) \\ V^{(h)} &= XW_{h,v} && (9.9) \end{aligned}$$

and $W_{h,q}, W_{h,k}, W_{h,v} \in \mathbb{R}^{d\times d_k}$where $d_k$ is the size of the key/query embedding space, and $h \in \{1, \cdots, H\}$ is an index over "attention heads."

for each attention-head $h$, we learn one set of $W_{h,q}, W_{h,k}, W_{h,v}$.

The attention weights $\alpha_{ij}^{(h)}$ for head $h$, query $i$ and key $j$ are then given by

$$\alpha_{ij}^{(h)} = \text{softmax}_j\left(\frac{{Q_i^{(h)}}^T K_j^{(h)}}{\sqrt{d_k}}\right). \tag{9.10}$$

Here, $Q_i^{(h)}$ denotes the row of $Q^{(h)}$ for input element $x^{(i)}$, and similarly $K_j^{(h)}$ corresponds to $x^{(j)}$. Also, $\text{softmax}_j$ is a softmax over the $d_k$-dimensional vector indexed by $j$, so in Eq. 9.10 this means a softmax computed over keys. In this equation, the normalization by $\sqrt{d_k}$ is done to reduce the magnitude of the dot product, which would otherwise grow undesirably large with increasing $d_k$.

We then perform a weighted sum over all the outputs for each head,

$$u'^{(i)} = \sum_{h=1}^{H} W_{h,c}^T \sum_{j=1}^{n} \alpha_{ij}^{(h)} V_j^{(h)}, \tag{9.11}$$

where $W_{h,c} \in \mathbb{R}^{d_k\times d}$, $u'^{(i)} \in \mathbb{R}^{d\times 1}$,. The indices $i \in \{1, \cdots, n\}$ and $j \in \{1, \cdots, n\}$ are integer indices over tokens. Observe that this is just a weighted combination of self-attention

$V_j^{(h)}$ is the $d_k \times 1$ value embedding vector that corresponds to the input token $x^j$ for attention head $h$.

[1] The presentation here follows the notes by John Thickstun.

outputs expressed more simply by Eq. 9.6.

The weighted sum ${u'}^{(i)}$ is then standardized and combined with $x^{(i)}$ using a LayerNorm function (defined below) to become

$$u^{(i)} = \text{LayerNorm}\left(x^{(i)} + {u'}^{(i)}; \gamma_1, \beta_1\right) \tag{9.12}$$

with parameters $\gamma_1, \beta_1 \in \mathbb{R}^d$.

To get the final output, we follow the "intermediate output then layer norm" recipe again. In particular, we first get the transformer block output ${z'}^{(i)}$ given by

$${z'}^{(i)} = W_2^T \text{ReLU}\left(W_1^T u^{(i)}\right) \tag{9.13}$$

with weights $W_1 \in \mathbb{R}^{d \times m}$ and $W_2 \in \mathbb{R}^{m \times d}$. This is then standardized and combined with $u^{(i)}$ to give the final output $z^{(i)}$:

$$z^{(i)} = \text{LayerNorm}\left(u^{(i)} + {z'}^{(i)}; \gamma_2, \beta_2\right) , \tag{9.14}$$

with parameters $\gamma_2, \beta_2 \in \mathbb{R}^d$. These vectors are then assembled (e.g., through parallel computation) to produce $z \in \mathbb{R}^{n \times d}$.

The LayerNorm function transforms a d-dimensional input $z$ with parameters $\gamma, \beta \in \mathbb{R}^d$ into

$$\text{LayerNorm}(z; \gamma, \beta) = \gamma \frac{z - \mu_z}{\sigma_z} + \beta\, , \tag{9.15}$$

where $\mu_z$ is the mean and $\sigma_z$ the standard deviation of $z$:

$$\mu_z = \frac{1}{d} \sum_{i=1}^{d} z_i \tag{9.16}$$

$$\sigma_z = \sqrt{\frac{1}{d} \sum_{i=1}^{d} (z_i - \mu_z)^2}\, . \tag{9.17}$$

Layer normalization is done to improve convergence stability during training.

The model parameters comprise the weight matrices $W_{h,q}, W_{h,k}, W_{h,v}, W_{h,c}, W_1, W_2$ and the LayerNorm parameters $\gamma_1, \gamma_2, \beta_1, \beta_2$. A *transformer* is the composition of L transformer blocks, each with its own parameters:

$$f_{\theta_L} \circ \cdots \circ f_{\theta_2} \circ f_{\theta_1}(x) \in \mathbb{R}^{n \times d}\, . \tag{9.18}$$

The hyperparameters of this model are $d, d_k, m, H$, and $L$.

### 9.3.2 Variations and training

Many variants on this transformer structure exist. For example, the LayerNorm may be moved to other stages of the neural network. Or a more sophisticated attention function may be employed instead of the simple dot product used in Eq. 9.10. Transformers may also be used in pairs, for example, one to process the input and a separate one to generate the output given the transformed input. Self-attention may also be replaced with cross-attention, where some input data are used to generate queries and other input data generate keys and values. Positional encoding and masking are also common, though they are left implicit in the above equations for simplicity.

How are transformers trained? The number of parameters in $\theta$ can be very large; modern transformer models like GPT4 have tens of billions of parameters or more. A great deal of data is thus necessary to train such models, else the models may simply overfit small datasets.

Training large transformer models is thus generally done in two stages. A first “pre-training” stage employs a very large dataset to train the model to extract patterns. This is done with unsupervised (or self-supervised) learning and unlabelled data. For example, the well-known BERT model was pre-trained using sentences with words masked. The model was trained to predict the masked words. BERT was also trained on sequences of sentences, where the model was trained to predict whether two sentences are likely to be contextually close together or not. The pre-training stage is generally very expensive.

The second “fine-tuning” stage trains the model for a specific task, such as classification or question answering. This training stage can be relatively inexpensive, but it generally requires labeled data.

# CHAPTER 10

## Autoencoders

In previous chapters, we have largely focused on classification and regression problems, where we use supervised learning with training samples that have both features/inputs and corresponding outputs or labels, to learn hypotheses or models that can then be used to predict labels for new data.

In contrast to supervised learning paradigm, we can also have an unsupervised learning setting, where we only have features but no corresponding outputs or labels for our dataset. On natural question aries then: if there are no labels, what are we learning?

One canonical example of unsupervised learning is clustering, which we learned about in Chapter 7. In clustering, the goal is to develop algorithms that can reason about "similarity" among data points's features, and group the data points into clusters.

*Autoencoders* are another family of unsupervised learning algorithms, in this case seeking to obtain insights about our data by learning compressed versions of the original data, or, in other words, by finding a good lower-dimensional feature representations of the same data set. Such insights might help us to discover and characterize underlying factors of variation in data, which can aid in scientific discovery; to compress data for efficient storage or communication; or to pre-process our data prior to supervised learning, perhaps to reduce the amount of data that is needed to learn a good classifier or regressor.

## 10.1 Autoencoder structure

Assume that we have input data $\mathcal{D} = \{x^{(1)}, \dots, x^{(n)}\}$, where $x^{(i)} \in \mathbb{R}^d$. We seek to learn an autoencoder that will output a new dataset $\mathcal{D}_{\text{out}} = \{a^{(1)}, \dots, a^{(n)}\}$, where $a^{(i)} \in \mathbb{R}^k$ with $k < d$. We can think about $a^{(i)}$ as the new *representation* of data point $x^{(i)}$. For example, in Fig. 10.1 we show the learned representations of a dataset of MNIST digits with $k = 2$. We see, after inspecting the individual data points, that unsupervised learning has found a compressed (or *latent*) representation where images of the same digit are close to each other, potentially greatly aiding subsequent clustering or classification tasks.

Formally, an autoencoder consists of two functions, a vector-valued *encoder* $g : \mathbb{R}^d \to \mathbb{R}^k$ that deterministically maps the data to the representation space $a \in \mathbb{R}^k$, and a *decoder* $h : \mathbb{R}^k \to \mathbb{R}^d$ that maps the representation space back into the original data space.

In general, the encoder and decoder functions might be any functions appropriate to the domain. Here, we are particularly interested in neural network embodiments of encoders

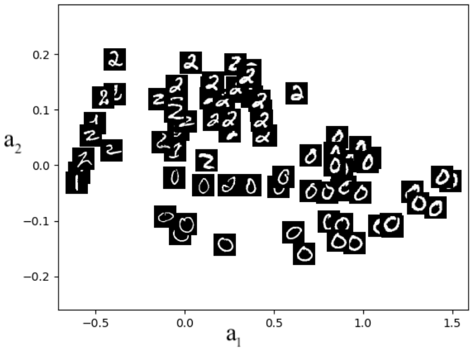


Figure 10.1: Compression of digits dataset into two dimensions. The input $x^{(i)}$, an image of a handwritten digit, is shown at the new low-dimensional representation $(a_1, a_2)$.

and decoders. The basic architecture of one such autoencoder, consisting of only a single layer neural network in each of the encoder and decoder, is shown in Figure 10.2; note that bias terms $W_0^1$ and $W_0^2$ into the summation nodes exist, but are omitted for clarity in the figure. In this example, the original d-dimensional input is compressed into $k = 3$ dimensions via the encoder $g(x; W^1, W_0^1) = f_1({W^1}^T x + W_0^1)$ with $W^1 \in \mathbb{R}^{d \times k}$ and $W_0^1 \in \mathbb{R}^k$, and where the non-linearity $f_1$ is applied to each dimension of the vector. To recover (an approximation to) the original instance, we then apply the decoder $h(a; W^2, W_0^2) = f_2({W^2}^T a + W_0^2)$, where $f_2$ denotes a different non-linearity (activation function). In general, both the decoder and the encoder could involve multiple layers, as opposed to the single layer shown here. Learning seeks parameters $W^1, W_0^1$ and $W^2, W_0^2$ such that the reconstructed instances, $h(g(x^{(i)}; W^1, W_0^1); W^2, W_0^2)$, are close to the original input $x^{(i)}$.

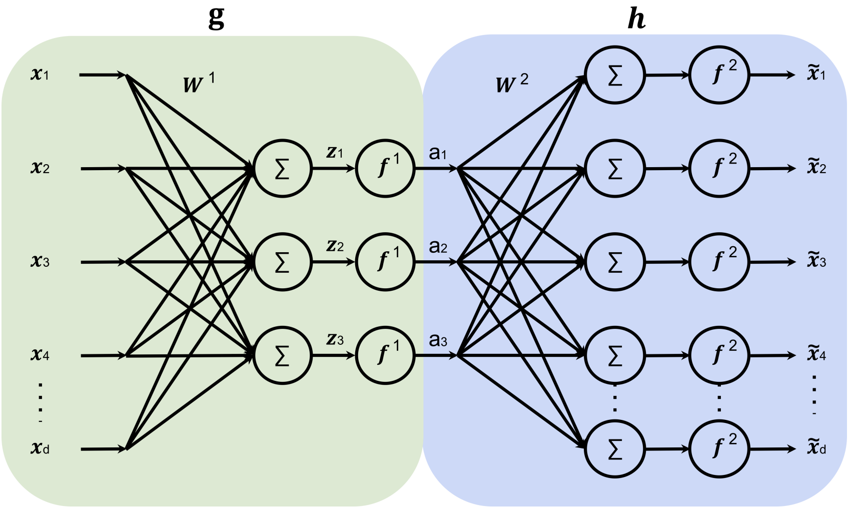


Figure 10.2: Autoencoder structure, showing the encoder (left half, light green), and the decoder (right half, light blue), encoding inputs x to the representation a, and decoding the representation to produce $\tilde{x}$, the reconstruction. In this specific example, the representation $(a_1, a_2, a_3)$ only has three dimensions.

## 10.2 Autoencoder Learning

We learn the weights in an autoencoder using the same tools that we previously used for supervised learning, namely (stochastic) gradient descent of a multi-layer neural network to minimize a loss function. All that remains is to specify the loss function $\mathcal{L}(\tilde{x}, x)$, which tells us how to measure the discrepancy between the reconstruction $\tilde{x} = h(g(x; W^1, W_0^1); W^2, W_0^2)$ and the original input $x$. For example, for continuous-valued $x$ it might make sense to use squared loss, i.e., $\mathcal{L}_{SE}(\tilde{x}, x) = \sum_{j=1}^{d}(x_j - \tilde{x}_j)^2$. Learning then seeks to optimize the parameters of $h$ and $g$ so as to minimize the reconstruction error, measured according to this loss function:

$$\min_{W^1, W_0^1, W^2, W_0^2} \sum_{i=1}^{n} \mathcal{L}_{SE}\left(h(g(x^{(i)}; W^1, W_0^1); W^2, W_0^2), x^{(i)}\right)$$

Alternatively, you could think of this as *multi-task learning*, where the goal is to predict each dimension of $x$. One can mix-and-match loss functions as appropriate for each dimension's data type.

## 10.3 Evaluating an autoencoder

What makes a good learned representation in an autoencoder? Notice that, without further constraints, it is always possible to perfectly reconstruct the input. For example, we could let $k = d$ and $h$ and $g$ be the identity functions. In this case, we would not obtain any compression of the data.

To learn something useful, we must create a *bottleneck* by making $k$ to be smaller (often much smaller) than $d$. This forces the learning algorithm to seek transformations that describe the original data using as simple a description as possible. Thinking back to the digits dataset, for example, an example of a compressed representation might be the digit label (i.e., 0–9), rotation, and stroke thickness. Of course, there is no guarantee that the learning algorithm will discover precisely this representation. After learning, we can inspect the learned representations, such as by artificially increasing or decreasing one of the dimensions (e.g., $a_1$) and seeing how it affects the output $h(a)$, to try to better understand what it has learned.

As with clustering, autoencoders can be a preliminary step toward building other models, such as a regressor or classifier. For example, once a good encoder has been learned, the decoder might be replaced with another neural network that is then trained with supervised learning (perhaps using a smaller dataset that does include labels).

## 10.4 Linear encoders and decoders

We close by mentioning that even linear encoders and decoders can be very powerful. In this case, rather than minimizing the above objective with gradient descent, a technique called *principal components analysis* (PCA) can be used to obtain a closed-form solution to the optimization problem using a singular value decomposition (SVD). Just as a multilayer neural network with nonlinear activations for regression (learned by gradient descent) can be thought of as a nonlinear generalization of a linear regressor (fit by matrix algebraic operations), the neural network based autoencoders discussed above (and learned with gradient descent) can be thought of as a generalization of linear PCA (as solved with matrix algebra by SVD).

## 10.5 Advanced encoders and decoders

Advanced neural networks that build on the encoder-decoder conceptual decomposition have become increasingly powerful in recent years. One family of applications are *generative* networks, where new outputs that are "similar to" but different from any existing training sample are desired. In *variational autoencoders* the compressed representation encompasses information about the probability distribution of training samples, e.g., learning both mean and standard deviation variables in the bottleneck layer or latent representation. Then, new outputs can be generated by random sampling based on the latent representation variables and feeding those samples into the decoder.

*Transformer* neural networks use *multiple* encoder and decoder blocks, together with a self-attention mechanism to make predictions about potential next outputs resulting from sequences of inputs. Such transformer networks have many applications in natural language processing and elsewhere. We will learn about sequential models in Chapter C.

CHAPTER 11

# Markov Decision Processes

So far, most of the learning problems we have looked at have been *supervised*, that is, for each training input $x^{(i)}$, we are told which value $y^{(i)}$ should be the output. From a traditional machine-learning viewpoint, there're two other major groups of learning problems: one is the *unsupervised* learning problems, in which we are given data and no expected outputs, and we will look at later in Chapter 7 and Chapter 10.

The other major type is the so-called *Reinforcement learning* (RL) problems. Reinforcement learning differs significantly from supervised learning problems, and we will delve into the details later in in Chapter 12. However, it's worth pointing out one major difference at a very high level: in supervised learning, our goal is to learn a one-time static mapping to make predictions, whereas in RL, the setup requires us to sequentially take actions to maximize cumulative rewards.

This setup change necessitates additional mathematical and algorithmic tools for us to understand RL. *Markov decision process* (MDP) is precisely such a classical and fundamental tool.

## 11.1 Definition and value functions

Formally, a Markov decision process is $\langle \mathcal{S}, \mathcal{A}, T, R, \gamma \rangle$ where $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, and:

- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is a *transition model*, where

  $$T(s, a, s') = \Pr(S_t = s' | S_{t-1} = s, A_{t-1} = a) \enspace ,$$

  specifying a conditional probability distribution;

- $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is a *reward function*, where $R(s, a)$ specifies an immediate reward for taking action $a$ when in state $s$; and

- $\gamma \in [0, 1]$ is a *discount factor*, which we'll discuss in Section 11.1.2.

The notation $S_t = s'$ uses a capital letter $S$ to stand for a random variable, and small letter $s$ to stand for a concrete value. So $S_t$ here is a random variable that can take on elements of $\mathcal{S}$ as values.

In this class, we assume the rewards are deterministic functions. Further, in this MDP chapter, we assume the state space and action space are finite (in fact, typically small).

The following description of a simple machine as Markov decision process provides a concrete example of an MDP. The machine has three possible operations (*actions*): "wash", "paint", and "eject" (each with a corresponding button). Objects are put into the machine. Each time you push a button, something is done to the object. However, it's an old machine, so it's not very reliable. The machine has a camera inside that can clearly detect what is going on with the object and will output the state of the object: "dirty", "clean", "painted", or "ejected". For each action, this is what is done to the object:

**Wash**:

- If you perform the "wash" operation on any object, whether it's dirty, clean, or painted, it will end up "clean" with probability 0.9 and "dirty" otherwise.

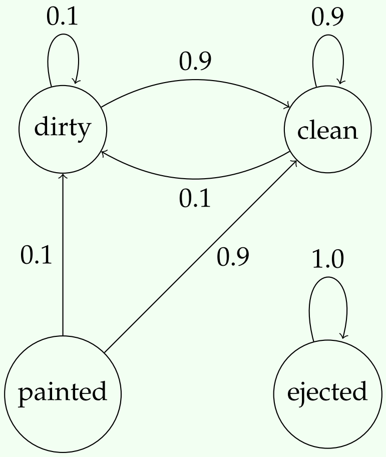
**Paint**:

- If you perform the "paint" operation on a clean object, it will become nicely "painted" with probability 0.8. With probability 0.1, the paint misses but the object stays clean, and also with probability 0.1, the machine dumps rusty dust all over the object and it becomes "dirty".
- If you perform the "paint" operation on a "painted" object, it stays "painted" with probability 1.0.
- If you perform the "paint" operation on a "dirty" part, it stays "dirty" with probability 1.0.

**Eject**:

- If you perform an "eject" operation on any part, the part comes out of the machine and this fun game is over. The part remains "ejected" regardless of any further action.

These descriptions specify the transition model T, and the transition function for each action can be depicted as a state machine diagram. For example, here is the diagram for "wash":



You get reward +10 for ejecting a painted object, reward 0 for ejecting a non-painted object, reward 0 for any action on an "ejected" object, and reward -3 otherwise. The MDP description would be completed by also specifying a discount factor.

A *policy* is a function $\pi$ that specifies what action to take in each state. The policy is what we will want to learn; it is akin to the strategy that a player employs to win a given game. Below, we take just the initial steps towards this eventual goal. We describe how to evaluate how good a policy is, first in the *finite horizon* case (Section 11.1.1) when the total number of transition steps is finite. In the finite horizon case, we typically denote the policy as $\pi_h(s)$, where $h \in \mathbb{N}$ is a non-negative integer denoting the number of steps remaining and $s \in \mathcal{S}$ is the current state. Then we consider the *infinite horizon* case (Section 11.1.2), when you don't know when the game will be over.

### 11.1.1 Finite-horizon value functions

The goal of a policy is to maximize the expected total reward, averaged over the stochastic transitions that the domain makes. Let's first consider the case where there is a finite *horizon* $H$, indicating the total number of steps of interaction that the agent will have with the MDP.

In the finite-horizon case, we usually set the discount factor $\gamma$ to 1.

We seek to measure the goodness of a policy. We do so by defining for a given horizon $h$ and MDP policy $\pi_h$, the "horizon $h$ *value*" of a state, $V^h_\pi(s)$. We do this by induction on the horizon, which is the *number of steps left to go*.

The base case is when there are no steps remaining, in which case, no matter what state we're in, the value is 0, so

$$V^0_\pi(s) = 0 \ . \tag{11.1}$$

Then, the value of a policy in state $s$ at horizon $h+1$ is equal to the reward it will get in state $s$ plus the next state's expected horizon $h$ value, discounted by a factor $\gamma$. So, starting with horizons 1 and 2, and then moving to the general case, we have:

$$V^1_\pi(s) = R(s, \pi_1(s)) + 0 \tag{11.2}$$

$$V^2_\pi(s) = R(s, \pi_2(s)) + \gamma \sum_{s'} T(s, \pi_2(s), s') V^1_\pi(s') \tag{11.3}$$

$$\vdots$$

$$V^h_\pi(s) = R(s, \pi_h(s)) + \gamma \sum_{s'} T(s, \pi_h(s), s') V^{h-1}_\pi(s') \tag{11.4}$$

The sum over $s'$ is an *expectation*: it considers all possible next states $s'$, and computes an average of their $(h-1)$-horizon values, weighted by the probability that the transition function from state $s$ with the action chosen by the policy $\pi_h(s)$ assigns to arriving in state $s'$, and discounted by $\gamma$.

**Study Question:** What is $\sum_{s'} T(s, a, s')$ for any particular $s$ and $a$?

**Study Question:** Convince yourself that Eqs. 11.1 and 11.3 are special cases of Eq. 11.4.

Then we can say that a policy $\pi$ is better than policy $\bar{\pi}$ for horizon $h$ if and only if for all $s \in \mathcal{S}$, $V^h_\pi(s) \geqslant V^h_{\bar{\pi}}(s)$ and there exists at least one $s \in \mathcal{S}$ such that $V^h_\pi(s) > V^h_{\bar{\pi}}(s)$.

### 11.1.2 Infinite-horizon value functions

More typically, the actual finite horizon is not known, i.e., when you don't know when the game will be over! This is called the *infinite horizon* version of the problem. How does one evaluate the goodness of a policy in the infinite horizon case?

If we tried to simply take our definitions above and use them for an infinite horizon, we could get in trouble. Imagine we get a reward of 1 at each step under one policy and a reward of 2 at each step under a different policy. Then the reward as the number of steps grows in each case keeps growing to become infinite in the limit of more and more steps. Even though it seems intuitive that the second policy should be better, we can't justify that by saying $\infty < \infty$.

One standard approach to deal with this problem is to consider the *discounted* infinite horizon. We will generalize from the finite-horizon case by adding a discount factor.

In the finite-horizon case, we valued a policy based on an expected finite-horizon value:

$$\mathbb{E}\left[\sum_{t=0}^{h-1} \gamma^t R_t \mid \pi, s_0\right] \quad , \tag{11.5}$$

where $R_t$ is the reward received at time t.

What is $\mathbb{E}[\cdot]$? This mathematical notation indicates an *expectation*, i.e., an average taken over all the random possibilities which may occur for the argument. Here, the expectation is taken over the *conditional probability* $\Pr(R_t = r \mid \pi, s_0)$, where $R_t$ is the random variable for the reward, subject to the policy being $\pi$ and the state being $s_0$. Since $\pi$ is a function, this notation is shorthand for conditioning on all of the random variables implied by policy $\pi$ and the stochastic transitions of the MDP.

A very important point is that $R(s, a)$ is always deterministic (in this class) for any given s and a. Here $R_t$ represents the set of all possible $R(s_t, a)$ at time step t; this $R_t$ is a random variable because the state we're in at step t is itself a random variable, due to prior stochastic state transitions up to but not including at step t and prior (deterministic) actions dictated by policy $\pi$.

Now, for the infinite-horizon case, we select a discount factor $0 \leqslant \gamma \leqslant 1$, and evaluate a policy based on its expected *infinite horizon discounted value*:

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t \mid \pi, s_0\right] = \mathbb{E}\left[R_0 + \gamma R_1 + \gamma^2 R_2 + \ldots \mid \pi, s_0\right] \quad . \tag{11.6}$$

Note that the t indices here are not the number of steps to go, but actually the number of steps forward from the starting state (there is no sensible notion of "steps to go" in the infinite horizon case).

Eqs. 11.5 and 11.6 are a conceptual stepping stone. Our main objective is to get to Eq. 11.8, which can also be viewed as including $\gamma$ in Eq. 11.4, with the appropriate definition of the infinite-horizon value.

There are two good intuitive motivations for discounting. One is related to economic theory and the present value of money: you'd generally rather have some money today than that same amount of money next week (because you could use it now or invest it). The other is to think of the whole process terminating, with probability $1 - \gamma$ on each step of the interaction. This value is the expected amount of reward the agent would gain under this terminating model.

At every step, your expected future lifetime, given that you have survived until now, is $1/(1-\gamma)$.

**Study Question:** Verify this fact: if, on every day you wake up, there is a probability of $1-\gamma$ that today will be your last day, then your expected lifetime is $1/(1-\gamma)$ days.

Let us now evaluate a policy in terms of the expected discounted infinite-horizon value that the agent will get in the MDP if it executes that policy. We define the infinite-horizon value of a state s under policy $\pi$ as

$$V_\pi(s) = \mathbb{E}[R_0 + \gamma R_1 + \gamma^2 R_2 + \cdots \mid \pi, S_0 = s] = \mathbb{E}[R_0 + \gamma(R_1 + \gamma(R_2 + \gamma \ldots))) \mid \pi, S_0 = s] \quad . \tag{11.7}$$

Because the expectation of a linear combination of random variables is the linear combination of the expectations, we have

$$\begin{aligned} V_\pi(s) &= \mathbb{E}[R_0 \mid \pi, S_0 = s] + \gamma \mathbb{E}[R_1 + \gamma(R_2 + \gamma \dots))) \mid \pi, S_0 = s] \\ &= R(s, \pi(s)) + \gamma \sum_{s'} T(s, \pi(s), s') V_\pi(s') \ . \end{aligned} \tag{11.8}$$

The equation defined in Eq. 11.8 is known as the Bellman Equation, which breaks down the value function into the immediate reward and the (discounted) future value function. You could write down one of these equations for each of the $n = |\mathcal{S}|$ states. There are $n$ unknowns $V_\pi(s)$. These are linear equations, and standard software (e.g., using Gaussian elimination or other linear algebraic methods) will, in most cases, enable us to find the value of each state under this policy.

This is *so* cool! In a discounted model, if you find that you survived this round and landed in some state $s'$, then you have the same expected future lifetime as you did before. So the value function that is relevant in that state is exactly the same one as in state $s$.

## 11.2 Finding policies for MDPs

Given an MDP, our goal is typically to find a policy that is optimal in the sense that it gets as much total reward as possible, in expectation over the stochastic transitions that the domain makes. We build on what we have learned about evaluating the goodness of a policy (Sections 11.1.1 and 11.1.2), and find optimal policies for the finite horizon case (Section 11.2.1), then the infinite horizon case (Section 11.2.2).

### 11.2.1 Finding optimal finite-horizon policies

How can we go about finding an optimal policy for an MDP? We could imagine enumerating all possible policies and calculating their value functions as in the previous section and picking the best one – but that's too much work!

The first observation to make is that, in a finite-horizon problem, the best action to take depends on the current state, but also on the horizon: imagine that you are in a situation where you could reach a state with reward 5 in one step or a state with reward 100 in two steps. If you have at least two steps to go, then you'd move toward the reward 100 state, but if you only have one step left to go, you should go in the direction that will allow you to gain 5!

One way to find an optimal policy is to compute an optimal *action-value function*, Q. For the finite-horizon case, we define $Q^h(s, a)$ to be the expected value of

- starting in state $s$,
- executing action $a$, and
- continuing for $h - 1$ more steps executing an optimal policy for the appropriate horizon on each step.

Similar to our definition of $V^h$ for evaluating a policy, we define the $Q^h$ function recursively according to the horizon. The only difference is that, on each step with horizon $h$, rather than selecting an action specified by a given policy, we select the value of $a$ that will maximize the expected $Q^h$ value of the next state.

$$Q^0(s,a) = 0 \tag{11.9}$$

$$Q^1(s,a) = R(s,a) + 0 \tag{11.10}$$

$$Q^2(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q^1(s',a') \tag{11.11}$$

$$\vdots$$

$$Q^h(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q^{h-1}(s',a') \tag{11.12}$$

where $(s', a')$ denotes the next time-step state/action pair. We can solve for the values of $Q^h$ with a simple recursive algorithm called *finite-horizon value iteration* that just computes $Q^h$ starting from horizon 0 and working backward to the desired horizon $H$. Given $Q^h$, an optimal $\pi^*_h$ can be found as follows:

$$\pi^*_h(s) = \arg\max_a Q^h(s,a) \quad . \tag{11.13}$$

which gives the *immediate* best action(s) to take when there are $h$ steps left; then $\pi^*_{h-1}(s)$ gives the best action(s) when there are $(h-1)$ steps left, and so on. In the case where there are multiple best actions, we typically can break ties randomly.

Additionally, it is worth noting that in order for such an optimal policy to be computed, we assume that the reward function $R(s,a)$ is bounded on the set of all possible (state, action) pairs. Furthermore, we will assume that the set of all possible actions is finite.

**Study Question:** The optimal value function is unique, but the optimal policy is not. Think of a situation in which there is more than one optimal policy.

**Dynamic programming** (somewhat counter-intuitively, dynamic programming is neither really "dynamic" nor a type of "programming" as we typically understand it) is a technique for designing efficient algorithms. Most methods for solving MDPs or computing value functions rely on dynamic programming to be efficient. The *principle of dynamic programming* is to compute and store the solutions to simple sub-problems that can be re-used later in the computation. It is a very important tool in our algorithmic toolbox.

Let's consider what would happen if we tried to compute $Q^4(s, a)$ for all $(s, a)$ by directly using the definition:

- To compute $Q^4(s_i, a_j)$ for any one $(s_i, a_j)$, we would need to compute $Q^3(s, a)$ for all $(s, a)$ pairs.
- To compute $Q^3(s_i, a_j)$ for any one $(s_i, a_j)$, we'd need to compute $Q^2(s, a)$ for all $(s, a)$ pairs.
- To compute $Q^2(s_i, a_j)$ for any one $(s_i, a_j)$, we'd need to compute $Q^1(s, a)$ for all $(s, a)$ pairs.
- Luckily, those are just our $R(s, a)$ values.

So, if we have $n$ states and $m$ actions, this is $O((mn)^3)$ work — that seems like way too much, especially as the horizon increases! But observe that we really only have $mnh$ values that need to be computed: $Q^h(s, a)$ for all $h, s, a$. If we start with $h = 1$, compute and store those values, then using and reusing the $Q^{h-1}(s, a)$ values to compute the $Q^h(s, a)$ values, we can do all this computation in time $O(mnh)$, which is much better!

### 11.2.2 Finding optimal infinite-horizon policies

In contrast to the finite-horizon case, the best way of behaving in an infinite-horizon discounted MDP is not time-dependent. That is, the decisions you make at time $t = 0$ looking forward to infinity, will be the same decisions that you make at time $t = T$ for any positive $T$, also looking forward to infinity.

An important theorem about MDPs is: in the infinite-horizon case, there exists a stationary optimal policy $\pi^*$ (there may be more than one) such that for all $s \in \mathcal{S}$ and all other policies $\pi$, we have

$$V_{\pi^*}(s) \geqslant V_\pi(s) \quad . \tag{11.14}$$

Stationary means that it doesn't change over time; in contrast, the optimal policy in a finite-horizon MDP is *non-stationary.*

There are many methods for finding an optimal policy for an MDP. We have already seen the finite-horizon value iteration case. Here we will study a very popular and useful method for the infinite-horizon case, *infinite-horizon value iteration*. It is also important to us, because it is the basis of many *reinforcement-learning* methods.

We will again assume that the reward function $R(s, a)$ is bounded on the set of all possible (state, action) pairs and additionally that the number of actions in the action space is finite. Define $Q(s, a)$ to be the expected infinite-horizon discounted value of being in state $s$, executing action $a$, and executing an optimal policy $\pi^*$ thereafter. Using similar reasoning to the recursive definition of $V_\pi$, we can express this value recursively as

$$Q(s, a) = R(s, a) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(s', a') \quad . \tag{11.15}$$

This is also a set of equations, one for each $(s, a)$ pair. This time, though, they are not linear (due to the max operation), and so they are not easy to solve. But there is a theorem that says they have a unique solution!

Once we know the optimal action-value function, then we can extract an optimal policy $\pi^*$ as

$$\pi^*(s) = \arg\max_a Q(s,a) \ . \tag{11.16}$$

We can iteratively solve for the $Q^*$ values with the infinite-horizon value iteration algorithm, shown below:

As in the finite-horizon case, there may be more than one optimal policy in the infinite-horizon case.

```
INFINITE-HORIZON-VALUE-ITERATION($\mathcal{S}, \mathcal{A}, T, R, \gamma, \epsilon$)
for $s \in \mathcal{S}, a \in \mathcal{A}$ :
    $Q_{\text{old}}(s,a) = 0$
while not converged:
    for $s \in \mathcal{S}, a \in \mathcal{A}$ :
        $Q_{\text{new}}(s,a) = R(s,a) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q_{\text{old}}(s',a')$
    if $\max_{s,a} |Q_{\text{old}}(s,a) - Q_{\text{new}}(s,a)| < \epsilon$ :
        return $Q_{\text{new}}$
    $Q_{\text{old}} = Q_{\text{new}}$
```

**Theory** There are a lot of nice theoretical results about infinite-horizon value iteration. For some given (not necessarily optimal) Q function, define $\pi_Q(s) = \arg\max_a Q(s,a)$.

- After executing infinite-horizon value iteration with convergence hyper-parameter $\epsilon$,

$$\|V_{\pi_{Q_{\text{new}}}} - V_{\pi^*}\|_{\max} < \epsilon \ . \tag{11.17}$$

- There is a value of $\epsilon$ such that

$$\|Q_{\text{old}} - Q_{\text{new}}\|_{\max} < \epsilon \Longrightarrow \pi_{Q_{\text{new}}} = \pi^* \tag{11.18}$$

Note the new notation! Given two functions $f$ and $f'$, we write $\|f - f'\|_{\max}$ to mean $\max_x |f(x) - f'(x)|$. It measures the maximum absolute disagreement between the two functions at any input $x$.

- As the algorithm executes, $\|V_{\pi_{Q_{\text{new}}}} - V_{\pi^*}\|_{\max}$ decreases monotonically on each iteration.
- The algorithm can be executed asynchronously, in parallel: as long as all $(s,a)$ pairs are updated infinitely often in an infinite run, it still converges to the optimal value.

This is very important for reinforcement learning.

# CHAPTER 12

# Reinforcement learning

So far, all the learning problems we have looked at have been *supervised*, that is, for each training input $x^{(i)}$, we are told which value $y^{(i)}$ should be the output. *Reinforcement learning* differs from previous learning problems in several important ways:

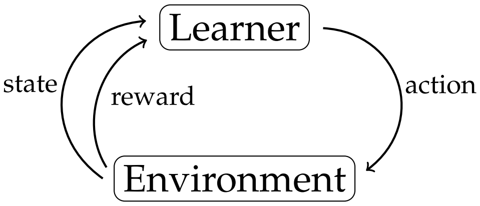
- The learner interacts explicitly with an environment, rather than implicitly (as in supervised learning) through an available training data set of $(x^{(i)}, y^{(i)})$ pairs drawn from the environment.

- The learner has some choice over what new information it seeks to gain from the environment.

- The learner updates models incrementally as additional information about the environment becomes available.

*Online learning* is a variant of supervised learning in which new data pairs become available over time and the model is updated, e.g., by retraining over the entire larger data set, or by weight update using just the new data.

In a reinforcement learning problem, the interaction with the environment takes a particular form:



- Learner observes *input* state $s^{(i)}$
- Learner generates *output* action $a^{(i)}$
- Learner observes *reward* $r^{(i)}$
- Learner observes *input* state $s^{(i+1)}$
- Learner generates *output* action $a^{(i+1)}$
- Learner observes *reward* $r^{(i+1)}$
- ...

Similar to MDPs, the learner is supposed to find a *policy*, mapping a state $s$ to action $a$, that maximizes expected reward over time.

## 12.1 Reinforcement learning algorithms overview

A *reinforcement learning (RL) algorithm* is a kind of a policy that depends on the whole history of states, actions, and rewards and selects the next action to take. There are several different ways to measure the quality of an RL algorithm, including:

- Ignoring the $r^{(i)}$ values that it gets *while* learning, but considering how many interactions with the environment are required for it to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$ that is nearly optimal.

- Maximizing the expected sum of discounted rewards while it is learning.

Most of the focus is on the first criterion (which is called "sample efficiency"), because the second one is very difficult. The first criterion is reasonable when the learning can take place somewhere safe (imagine a robot learning, inside the robot factory, where it can't hurt itself too badly) or in a simulated environment.

Approaches to reinforcement learning differ significantly according to what kind of hypothesis or model is being learned. Roughly speaking, RL methods can be categorized into model-free methods and model-based methods. The main distinction is that model-based methods explicitly learn the transition and reward models to assist the end-goal of learning a policy; model-free methods do not. We will start our discussion with the model-free methods, and introduce two of the arguably most popular types of algorithms, Q-learning (Section 12.2.1) and policy gradient (Section 12.2.4). We then describe model-based methods (Section 12.3). Finally, we briefly consider "bandit" problems (Section 12.4), which differ from our MDP learning context by having probabilistic rewards.

## 12.2 Model-free methods

Model-free methods are methods that do not explicitly learn transition and rewards models. Depending on what is explicitly being learned, model-free methods are sometimes further categorized into value-based methods (where the goal is to learn/estimate a value function) and policy-based methods (where the goal is to directly learn an optimal policy). It's important to note that such categorization is approximate and the boundaries are blurry. In fact, current RL research tends to combine the learning of value functions, policies, and transition and reward models all into a complex learning algorithm, in an attempt to combine the strengths of each approach.

### 12.2.1 Q-learning

Q-learning is a frequently used class of RL algorithms that concentrates on learning (estimating) the state-action value function, i.e., the Q function. Specifically, recall the MDP value-iteration update:

$$Q(s, a) = R(s, a) + \gamma \sum_{s'} T(s, a, s') \max_{a'} Q(s', a') \tag{12.1}$$

The Q-learning algorithm below adapts this value-iteration idea to the RL scenario, where we do not know the transition function T or reward function R, and instead rely on samples to perform the updates.

The thing that most students seem to get confused about is when we do value iteration and when we do Q learning. Value iteration assumes you know T and R and just need to *compute* Q. In Q learning, we don't know or even directly estimate T and R: we estimate Q directly from experience!

```
Q-LEARNING($\mathcal{S}, \mathcal{A}, \gamma, \alpha, s_0$, max-iter)
$i = 0$
**for** $s \in \mathcal{S}, a \in \mathcal{A}$ :
    $Q_{\mathtt{old}}(s, a) = 0$
$s = s_0$
**while** $i <$ max-iter:
    $a =$ select_action$(s, Q_{\mathtt{old}}(s, a))$
    $r, s' =$ execute$(a)$
    $Q_{\text{new}}(s, a) = (1 - \alpha)Q_{\text{old}}(s, a) + \alpha(r + \gamma \max_{a'} Q_{\text{old}}(s', a'))$
    $s = s'$
    $i = (i + 1)$
    $Q_{\text{old}} = Q_{\text{new}}$
**return** $Q_{\text{new}}$
```

With the pseudo-code provided for Q-learning, there are a few key things to note. First, we must determine which state to initialize the learning from. In the context of a game, this initial state may be well defined. In the context of a robot navigating an environment, one may consider sampling the initial state at random. In any case, the initial state is necessary to determine the trajectory the agent will experience as it navigates the environment. Second, different contexts will influence how we want to choose when to stop iterating through the while loop. Again, in some games there may be a clear terminating state based on the rules of how it is played. On the other hand, a robot may be allowed to explore an environment *ad infinitum*. In such a case, one may consider either setting a fixed number of transitions (as done explictly in the pseudo-code) to take; or we may want to stop iterating in the example once the values in the Q-table are not changing, after the algorithm has been running for a while. Finally, a single trajectory through the environment may not be sufficient to adequately explore all state-action pairs. In these instances, it becomes necessary to run through a number of iterations of the Q-learning algorithm, potentially with different choices of initial state $s_0$. Of course, we would then want to modify Q-learning such that the Q table is not reset with each call.

This notion of running a number of instances of Q-learning is often referred to as experiencing multiple *episodes*.

Now, let's dig in to what is happening in Q-learning. Here, $\alpha \in (0, 1]$ represents the "learning rate," which needs to decay for convergence purposes, but in practice is often set to a constant. It's also worth mentioning that Q-learning assumes a discrete state and action space where states and actions take on discrete values like $1, 2, 3, \ldots$ etc. In contrast, a continuous state space would allow the state to take values from, say, a continuous range of numbers; for example, the state could be any real number in the interval $[1, 3]$. Similarly, a continuous action space would allow the action to be drawn from, e.g., a continuous range of numbers. There are now many extensions developed based on Q-learning that can handle continuous state and action spaces (we'll look at one soon), and therefore the algorithm above is also sometimes referred to more specifically as tabular Q-learning.

In the Q-learning update rule

$$Q[s, a] \leftarrow (1 - \alpha)Q[s, a] + \alpha(r + \gamma \max_{a'} Q[s', a']) \tag{12.2}$$

the term $r + \gamma \max_{a'} Q[s', a']$ is often referred to as the (one-step look-ahead) *target*. The update can be viewed as a combination of two different iterative processes that we have already seen: the combination of an old estimate with the target using a running average with a learning rate $\alpha$, and the dynamic-programming update of a Q value from value iteration.

Eq. 12.2 can also be equivalently rewritten as

$$Q[s, a] \leftarrow Q[s, a] + \alpha \left( (r + \gamma \max_{a'} Q[s', a']) - Q[s, a] \right), \tag{12.3}$$

which allows us to interpret Q-learning in yet another way: we make an update (or correction) based on the temporal difference between the target and the current estimated value $Q[s, a]$.

The Q-learning algorithm above includes a procedure called *select_action*, that, given the current state $s$ and current Q function, has to decide which action to take. If the Q value is estimated very accurately and the agent is behaving in the world, then generally we would want to choose the apparently optimal action $\arg\max_{a \in \mathcal{A}} Q(s, a)$. But, during learning, the Q value estimates won't be very good and exploration is important. However, exploring completely at random is also usually not the best strategy while learning, because it is good to focus your attention on the parts of the state space that are likely to be visited when executing a good policy (not a bad or random one).

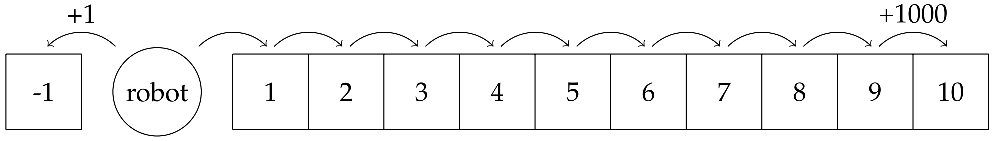
A typical action-selection strategy that attempts to address this *exploration versus exploitation* dilemma is the so-called $\epsilon$-*greedy* strategy:

- with probability $1 - \epsilon$, choose $\arg\max_{a \in \mathcal{A}} Q(s, a)$;
- with probability $\epsilon$, choose the action $a \in \mathcal{A}$ uniformly at random.

where the $\epsilon$ probability of choosing a random action helps the agent to explore and try out actions that might not seem so desirable at the moment.

Q-learning has the surprising property that it is *guaranteed* to converge to the actual optimal Q function under fairly weak conditions! Any exploration strategy is okay as long as it tries every action infinitely often on an infinite run (so that it doesn't converge prematurely to a bad action choice).

Q-learning can be very inefficient. Imagine a robot that has a choice between moving to the left and getting a reward of 1, then returning to its initial state, or moving to the right and walking down a 10-step hallway in order to get a reward of 1000, then returning to its initial state.



The first time the robot moves to the right and goes down the hallway, it will update the Q value just for state 9 on the hallway and action "right" to have a high value, but it won't yet understand that moving to the right in the earlier steps was a good choice. The next time it moves down the hallway it updates the value of the state before the last one, and so on. After 10 trips down the hallway, it now can see that it is better to move to the right than to the left.

More concretely, consider the vector of Q values $Q(i = 0, \ldots, 9; \text{right})$, representing the Q values for moving right at each of the positions $i = 0, \ldots, 9$. Position index 0 is the starting position of the robot as pictured above.

Then, for $\alpha = 1$ and $\gamma = 0.9$, Eq. 12.3 becomes

$$Q(i,\ \text{right}) = R(i,\ \text{right}) + 0.9 \cdot \max_{a} Q(i+1, a). \tag{12.4}$$

Starting with Q values of 0,

$$Q^{(0)}(i = 0, \ldots, 9;\ \text{right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{12.5}$$

Since the only nonzero reward from moving right is $R(9,\ \text{right}) = 1000$, after our robot makes it down the hallway once, our new Q vector is

$$Q^{(1)}(i = 0, \ldots, 9;\ \text{right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1000 \end{bmatrix}. \tag{12.6}$$

We are violating our usual notational conventions here, and writing $Q^{(i)}$ to mean the Q value function that results after the robot runs all the way to the end of the hallway, when executing the policy that always moves to the right.

After making its way down the hallway again, $Q(8, \text{right}) = 0 + 0.9 \cdot Q(9, \text{right}) = 900$ updates:

$$Q^{(2)}(i = 0, \ldots, 9; \text{ right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 900 & 1000 \end{bmatrix}. \tag{12.7}$$

Similarly,

$$Q^{(3)}(i = 0, \ldots, 9; \text{ right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 810 & 900 & 1000 \end{bmatrix} \tag{12.8}$$

$$Q^{(4)}(i = 0, \ldots, 9; \text{ right}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 729 & 810 & 900 & 1000 \end{bmatrix} \tag{12.9}$$

$$\vdots \tag{12.10}$$

$$Q^{(10)}(i = 0, \ldots, 9; \text{ right}) = \begin{bmatrix} 387.4 & 420.5 & 478.3 & 531.4 & 590.5 & 656.1 & 729 & 810 & 900 & 1000 \end{bmatrix}, \tag{12.11}$$

and the robot finally sees the value of moving right from position 0.

Here, we can see the exploration/exploitation dilemma in action: from the perspective of $s_0 = 0$, it will seem that getting the immediate reward of 1 is a better strategy without exploring the long hallway.

**Study Question:** Determine the Q value functions that will result from updates due to the robot always executing the "move left" policy.

### 12.2.2 Function approximation: Deep Q learning

In our Q-learning algorithm above, we essentially keep track of each Q value in a table, indexed by $s$ and $a$. What do we do if $\mathcal{S}$ and/or $\mathcal{A}$ are large (or continuous)?

We can use a function approximator like a neural network to store Q values. For example, we could design a neural network that takes in inputs $s$ and $a$, and outputs $Q(s, a)$. We can treat this as a regression problem, optimizing this loss:

This is the so-called squared Bellman error; as the name suggests, it's closely related to the Bellman equation we saw in MDPs in Chapter 11. Roughly speaking, this error measures how much the Bellman equality is violated.

$$\left(Q(s, a) - (r + \gamma \max_{a'} Q(s', a'))\right)^2 , \tag{12.12}$$

where $Q(s, a)$ is now the output of the neural network.

There are several different architectural choices for using a neural network to approximate Q values:

- One network for each action $a$, that takes $s$ as input and produces $Q(s, a)$ as output;
- One single network that takes $s$ as input and produces a vector $Q(s, \cdot)$, consisting of the Q values for each action; or
- One single network that takes $s, a$ concatenated into a vector (if $a$ is discrete, we would probably use a one-hot encoding, unless it had some useful internal structure) and produces $Q(s, a)$ as output.

The first two choices are only suitable for discrete (and not too big) action sets. The last choice can be applied for continuous actions, but then it is difficult to find $\arg\max_{a \in \mathcal{A}} Q(s, a)$.

For continuous action spaces, it is popular to use a class of methods called *actor-critic* methods, which combine policy and value-function learning. We won't get into them in detail here, though.

There are not many theoretical guarantees about Q-learning with function approximation and, indeed, it can sometimes be fairly unstable (learning to perform well for a while, and then getting suddenly worse, for example). But neural network Q-learning has also had some significant successes.

One form of instability that we do know how to guard against is *catastrophic forgetting*. In standard supervised learning, we expect that the training $x$ values were drawn independently from some distribution. But when a learning agent, such as a robot, is moving through an environment, the sequence of states it encounters will be temporally correlated. For example, the robot might spend 12 hours in a dark environment and then 12 in a light

And, in fact, we routinely shuffle their order in the data file, anyway.

one. This can mean that while it is in the dark, the neural-network weight-updates will make the Q function "forget" the value function for when it's light.

One way to handle this is to use *experience replay*, where we save our $(s, a, s', r)$ experiences in a *replay buffer*. Whenever we take a step in the world, we add the $(s, a, s', r)$ to the replay buffer and use it to do a Q-learning update. Then we also randomly select some number of tuples from the replay buffer, and do Q-learning updates based on them, as well. In general it may help to keep a *sliding window* of just the 1000 most recent experiences in the replay buffer. (A larger buffer will be necessary for situations when the optimal policy might visit a large part of the state space, but we like to keep the buffer size small for memory reasons and also so that we don't focus on parts of the state space that are irrelevant for the optimal policy.) The idea is that it will help us propagate reward values through our state space more efficiently if we do these updates. We can see it as doing something like value iteration, but using samples of experience rather than a known model.

### 12.2.3 Fitted Q-learning

An alternative strategy for learning the Q function that is somewhat more robust than the standard Q-learning algorithm is a method called *fitted Q*.

```
FITTED-Q-LEARNING($\mathcal{A}, s_0, \gamma, \alpha, \epsilon, m$)
1   $s = s_0$ // (e.g., $s_0$ can be drawn randomly from $\mathcal{S}$)
2   $\mathcal{D} = \{\,\}$
3   initialize neural-network representation of Q
4   while True:
5       $\mathcal{D}_{\text{new}}$ = experience from executing $\epsilon$-greedy policy based on Q for $m$ steps
6       $\mathcal{D} = \mathcal{D} \cup \mathcal{D}_{\text{new}}$ represented as $(s, a, s', r)$ tuples
7       $\mathcal{D}_{\text{supervised}} = \{(x^{(i)}, y^{(i)})\}$ where $x^{(i)} = (s, a)$ and $y^{(i)} = r + \gamma \max_{a' \in \mathcal{A}} Q(s', a')$
8         for each tuple $(s, a, s', r)^{(i)} \in \mathcal{D}$
9       re-initialize neural-network representation of Q
10      $Q$ = SUPERVISED-NN-REGRESSION($\mathcal{D}_{\text{supervised}}$)
```

Here, we alternate between using the policy induced by the current Q function to gather a batch of data $\mathcal{D}_{\text{new}}$, adding it to our overall data set $\mathcal{D}$, and then using supervised neural-network training to learn a representation of the Q value function on the whole data set. This method does not mix the dynamic-programming phase (computing new Q values based on old ones) with the function approximation phase (supervised training of the neural network) and avoids catastrophic forgetting. The regression training in line 10 typically uses squared error as a loss function and would be trained until the fit is good (possibly measured on held-out data).

### 12.2.4 Policy gradient

A different model-free strategy is to search directly for a good policy. The strategy here is to define a functional form $f(s; \theta) = a$ for the policy, where $\theta$ represents the parameters we learn from experience. We choose $f$ to be differentiable, and often define $f(s, a; \theta) = \Pr(a|s)$, a conditional probability distribution over our possible actions.

This means the chance of choosing an action depends on which state the agent is in. Suppose, e.g., a robot is trying to get to a goal and can go left or right. An unconditional policy can say: I go left 99% of the time; a conditional policy can consider the robot's state, and say: if I'm to the right of the goal, I go left 99% of the time.

Now, we can train the policy parameters using gradient descent:

- When $\theta$ has relatively low dimension, we can compute a numeric estimate of the gradient by running the policy multiple times for different values of $\theta$, and computing the resulting rewards.

- When $\theta$ has higher dimensions (e.g., it represents the set of parameters in a complicated neural network), there are more clever algorithms, e.g., one called REINFORCE, but they can often be difficult to get to work reliably.

Policy search is a good choice when the policy has a simple known form, but the MDP would be much more complicated to estimate.

## 12.3 Model-based RL

The conceptually simplest approach to RL is to model R and T from the data we have gotten so far, and then use those models, together with an algorithm for solving MDPs (such as value iteration) to find a policy that is near-optimal given the current models.

Assume that we have had some set of interactions with the environment, which can be characterized as a set of tuples of the form $(s^{(t)}, a^{(t)}, s^{(t+1)}, r^{(t)})$.

Because the transition function $T(s, a, s')$ specifies probabilities, multiple observations of $(s, a, s')$ may be needed to model the transition function. One approach to this task of building a model $\hat{T}(s, a, s')$ for the true $T(s, a, s')$ is to estimate it using a simple counting strategy,

$$\hat{T}(s, a, s') = \frac{\#(s, a, s') + 1}{\#(s, a) + |\mathcal{S}|} . \tag{12.13}$$

Here, $\#(s, a, s')$ represents the number of times in our data set we have the situation where $s^{(t)} = s, a^{(t)} = a, s^{(t+1)} = s'$ and $\#(s, a)$ represents the number of times in our data set we have the situation where $s^{(t)} = s, a^{(t)} = a$.

**Study Question:** Prove to yourself that $\#(s, a) = \sum_{s'} \#(s, a, s')$.

Adding 1 and $|\mathcal{S}|$ to the numerator and denominator, respectively, are a form of smoothing called the *Laplace correction*. It ensures that we never estimate that a probability is 0, and keeps us from dividing by 0. As the amount of data we gather increases, the influence of this correction fades away.

Conceptually, this is also similar to having "initialized" our estimate for the transition function with uniform random probabilities, before having made any observations.

In contrast, the reward function $R(s, a)$ (as we have specified it in this text) is a *deterministic* function, such that knowing the reward $r$ for a given $(s, a)$ is sufficient to fully determine the function at that point. In other words, our model $\hat{R}$ can simply be a record of observed rewards, such that $\hat{R}(s, a) = r = R(s, a)$.

Given empirical models $\hat{T}$ and $\hat{R}$ for the transition and reward functions, we can now solve the MDP $(\mathcal{S}, \mathcal{A}, \hat{T}, \hat{R})$ to find an optimal policy using value iteration, or use a search algorithm to find an action to take for a particular state.

This approach is effective for problems with small state and action spaces, where it is not too hard to get enough experience to model T and R well; but it is difficult to generalize this method to handle continuous (or very large discrete) state spaces, and is a topic of current research.

## 12.4 Bandit problems

Bandit problems differ from our reinforcement learning setting as described above in two ways: the reward function is probabilistic, and the key decision is usually framed as whether or not to continue exploring (to improve the model) versus exploiting (take actions to maximize expected rewards based on the current model).

A basic bandit problem is given by

- A set of actions $\mathcal{A}$;

- A set of reward values $\mathcal{R}$; and
- A probabilistic reward function $R_p : \mathcal{A} \times \mathcal{R} \rightarrow \mathbb{R}$, i.e., $R_p$ is a function that takes an action and a reward and returns the probability of getting that reward conditioned on that action being taken, $R_p(a, r) = \Pr(\text{reward} = r \mid \text{action} = a)$. This is analogous to how the transition function $T$ is defined. Each time the agent takes an action, a new value is drawn from this distribution.

The most typical bandit problem has $\mathcal{R} = \{0, 1\}$ and $|\mathcal{A}| = k$. This is called a $k$-*armed bandit problem*, where the decision is which "arm" (action $a$) to select, and the reward is either getting a payoff (1) or not (0). There is a lot of mathematical literature on optimal strategies for $k$-armed bandit problems under various assumptions. The important question is usually one of *exploration versus exploitation*. Imagine that you have tried each action 10 times, and now you have estimates $\hat{R}_p(a, r)$ for the probabilities $R_p(a, r)$ for reward $r$ given action $a$. Which arm should you pick next? You could

Why? Because in English slang, "one-armed bandit" is a name for a slot machine (an old-style gambling machine where you put a coin into a slot and then pull its arm to see if you get a payoff) because it has one arm and takes your money! What we have here is a similar sort of machine, but with $k$ arms.

**exploit** your knowledge, and for future trials choose the arm with the highest value of expected reward; or

**explore** further, by trying some or all actions more times, hoping to get better estimates of the $R_p(a, r)$ values.

The theory ultimately tells us that, the longer our horizon $h$ (or, similarly, closer to 1 our discount factor), the more time we should spend exploring, so that we don't converge prematurely on a bad choice of action.

**Study Question:** Why is it that "bad" luck during exploration is more dangerous than "good" luck? Imagine that there is an action that generates reward value 1 with probability 0.9, but the first three times you try it, it generates value 0. How might that cause difficulty? Why is this more dangerous than the situation when an action that generates reward value 1 with probability 0.1 actually generates reward 1 on the first three tries?

Bandit problems are reinforcement learning problems (and are very different from batch supervised learning) in that:

There is a setting of supervised learning, called *active learning*, where instead of being given a training set, the learner gets to select a value of $x$ and the environment gives back a label $y$; the problem of picking good $x$ values to query is interesting, but the problem of deriving a hypothesis from $(x, y)$ pairs is the same as the supervised problem we have been studying.

- The agent gets to influence what data it obtains (selecting $a$ gives it another sample from $R(a, r)$), and
- The agent is penalized for mistakes it makes while it is learning (if it is trying to maximize the expected reward $\sum_r r \cdot \Pr(R_p(a, r) = r)$ it gets while behaving).

In a *contextual* bandit problem, you have multiple possible states, drawn from some set $\mathcal{S}$, and a separate bandit problem associated with each one.

Bandit problems are an essential subset of reinforcement learning. It's important to be aware of the issues, but we will not study solutions to them in this class.

# CHAPTER 13

# Non-parametric methods

Neural networks have adaptable complexity, in the sense that we can try different structural models and use cross validation to find one that works well on our data. Beyond neural networks, we may further broaden the class of models that we can fit to our data, for example as illustrated by the techniques introduced in this chapter.

Here, we turn to models that automatically adapt their complexity to the training data. The name *non-parametric methods* is misleading: it is really a class of methods that does not have a fixed parameterization in advance. Rather, the complexity of the parameterization can grow as we acquire more data.

Some non-parametric models, such as nearest-neighbor, rely directly on the data to make predictions and do not compute a model that summarizes the data. Other non-parametric methods, such as decision trees, can be seen as dynamically constructing something that ends up looking like a more traditional parametric model, but where the actual training data affects exactly what the form of the model will be.

These are sometimes called *classification trees*; the decision analysis literature uses "decision tree" for a structure that lays out possible future events that consist of choices interspersed with chance nodes.

The non-parametric methods we consider here tend to have the form of a composition of simple models:

- *Nearest neighbor models*: (Section 13.1) where we don't process data at training time, but do all the work when making predictions, by looking for the closest training example(s) to a given new data point.

- *Tree models*: (Section 13.2) where we partition the input space and use different simple predictions on different regions of the space; the hypothesis space can become arbitrarily large allowing finer and finer partitions of the input space.

- *Ensemble models*: (Section 13.2.3) in which we train several different classifiers on the whole space and average the answers; this decreases the estimation error. In particular, we will look at bootstrap aggregation, or *bagging* of trees.

- *Boosting* is a way to construct a model composed of a sequence of component models (e.g., a model consisting of a sequence of trees, each subsequent tree seeking to correct errors in the previous trees) that decreases both estimation and structural error. We won't consider this in detail in this class.

Why are we studying these methods, in the heyday of complicated models such as neural networks?

- They are fast to implement and have few or no hyperparameters to tune.
- They often work as well as or better than more complicated methods.
- Predictions from some of these models can be easier to explain to a human user: decision trees are fairly directly human-interpretable, and nearest neighbor methods can justify their decision to some extent by showing a few training examples that the prediction was based on.

## 13.1 Nearest Neighbor

In nearest-neighbor models, we don't do any processing of the data at training time – we just remember it! All the work is done at prediction time.

Input values $x$ can be from any domain $\mathcal{X}$ ($\mathbb{R}^d$, documents, tree-structured objects, etc.). We just need a distance metric, $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}^+$, which satisfies the following, for all $x, x', x'' \in \mathcal{X}$:

$$\begin{aligned} d(x, x) &= 0 \\ d(x, x') &= d(x', x) \\ d(x, x'') &\leqslant d(x, x') + d(x', x'') \end{aligned}$$

Given a data-set $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=1}^n$, our predictor for a new $x \in \mathcal{X}$ is

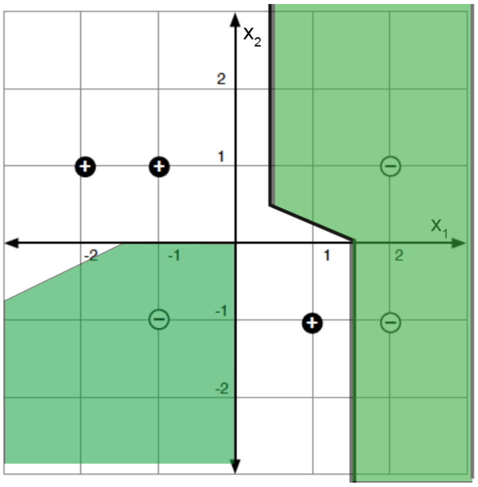
$$h(x) = y^{(i)} \quad \text{where} \quad i = \arg\min_i d(x, x^{(i)}) \ , \tag{13.1}$$

that is, the predicted output associated with the training point that is closest to the query point $x$. Tie breaking is typically done at random.

This same algorithm works for regression *and* classification!

The nearest neighbor prediction function can be described by dividing the space up into regions whose closest point is each individual training point as shown below :

Decision boundary regions can also be described by Voronoi diagrams. In a Voronoi diagram, each of the data points would have its own "cell" or region in the space that is closest to the data point in question. In the diagram provided here, cells have been merged if the predicted value is the same in adjacent cells.



In each region, we predict the associated $y$ value.

**Study Question:** Convince yourself that these boundaries do represent the nearest-neighbor classifier derived from these six data points.

There are several useful variations on this method. In $k$-*nearest-neighbors*, we find the $k$ training points nearest to the query point $x$ and output the majority $y$ value for classification or the average for regression. We can also do *locally weighted regression* in which we fit locally linear regression models to the $k$ nearest points, possibly giving less weight to those that are farther away. In large data-sets, it is important to use good data structures (e.g., ball trees) to perform the nearest-neighbor look-ups efficiently (without looking at all the data points each time).

## 13.2 Tree Models

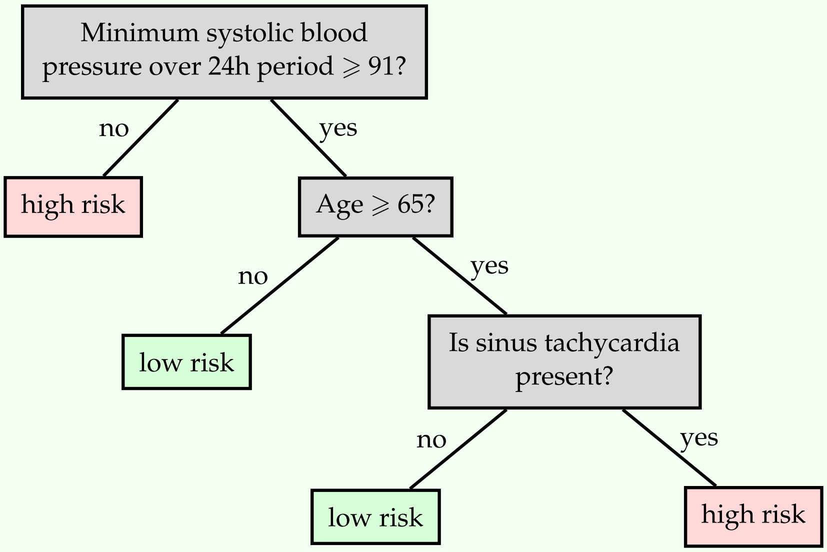
The idea here is that we would like to find a partition of the input space and then fit very simple models to predict the output in each piece. The partition is described using a (typically binary) "tree" that recursively splits the space.

Tree methods differ by:

- The class of possible ways to split the space at each node; these are typically linear splits, either aligned with the axes of the space, or sometimes using more general classifiers.
- The class of predictors within the partitions; these are often simply constants, but may be more general classification or regression models.
- The way in which we control the complexity of the hypothesis: it would be within the capacity of these methods to have a separate partition element for each individual training example.
- The algorithm for making the partitions and fitting the models.

One advantage of tree models is that they are easily interpretable by humans. This is important in application domains, such as medicine, where there are human experts who often ultimately make critical decisions and who need to feel confident in their understanding of recommendations made by an algorithm. Below is an example decision tree, illustrating how one might be able to understand the decisions made by the tree.

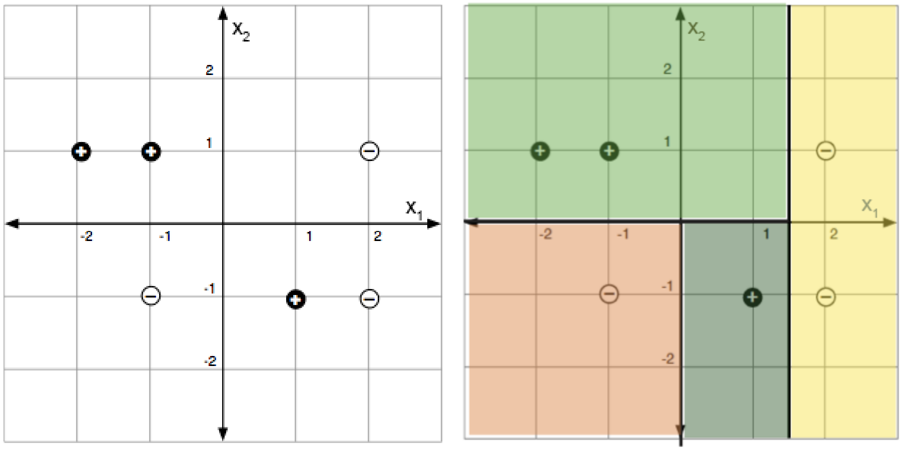
**Example:** Here is a sample tree (reproduced from Breiman, Friedman, Olshen, Stone (1984)):


Minimum systolic blood pressure over 24h period ⩾ 91?
no
yes
high risk
Age ⩾ 65?
no
yes
low risk
Is sinus tachycardia present?
no
yes
low risk
high risk

These methods are most appropriate for domains where the input space is not very high-dimensional and where the individual input features have some substantially useful information individually or in small groups. Trees would not be good for image input, but might be good in cases with, for example, a set of meaningful measurements of the condition of a patient in the hospital, as in the example above.

We'll concentrate on the CART/ID3 ("classification and regression trees" and "iterative dichotomizer 3", respectively) family of algorithms, which were invented independently in the statistics and the artificial intelligence communities. They work by greedily constructing a partition, where the splits are *axis aligned* and by fitting a *constant* model in the leaves. The interesting questions are how to select the splits and how to control complexity. The regression and classification versions are very similar.

As a concrete example, consider the following images:



The left image depicts a set of labeled data points in a two-dimensional feature space. The right shows a partition into regions by a decision tree, in this case having no classification errors in the final partitions.

### 13.2.1 Regression

The predictor is made up of

- a partition function, $\pi$, mapping elements of the input space into exactly one of $M$ regions, $R_1, \ldots, R_M$, and
- a collection of $M$ output values, $O_m$, one for each region.

If we already knew a division of the space into regions, we would set $O_m$, the constant output for region $R_m$, to be the average of the training output values in that region. For a training data set $\mathcal{D} = \left\{\left(x^{(i)}, y^{(i)}\right)\right\}, i = 1, \ldots n$, we let $I$ be an indicator set of all of the elements within $\mathcal{D}$, so that $I = \{1, \ldots, n\}$ for our whole data set. We can define $I_m$ as the subset of data set samples that are in region $R_m$, so that $I_m = \{i \mid x^{(i)} \in R_m\}$. Then

$$O_m = \text{average}_{i \in I_m} \; y^{(i)} \; .$$

We can define the error in a region as $E_m$. For example, $E_m$ as the sum of squared error would be expressed as

$$E_m = \sum_{i \in I_m} (y^{(i)} - O_m)^2 \; . \tag{13.2}$$

Ideally, we would select the partition to minimize

$$\lambda M + \sum_{m=1}^{M} E_m \ , \tag{13.3}$$

for some regularization constant $\lambda$. It is enough to search over all partitions of the training data (not all partitions of the input space!) to optimize this, but the problem is NP-complete.

**Study Question:** Be sure you understand why it's enough to consider all partitions of the training data, if this is your objective.

#### 13.2.1.1 Building a tree

So, we'll be greedy. We establish a criterion, given a set of data, for finding the best single split of that data, and then apply it recursively to partition the space. For the discussion below, we will select the partition of the data that *minimizes the sum of the sum of squared errors of each partition element.* Then later, we will consider other splitting criteria.

Given a data set $\mathcal{D} = \left\{\left(x^{(i)}, y^{(i)}\right)\right\}, i = 1, \ldots n$, we now consider $I$ to be an indicator of the subset of elements within $\mathcal{D}$ that we wish to build a tree (or subtree) for. That is, $I$ may already indicate a subset of data set $\mathcal{D}$, based on prior splits in constructing our overall tree. We define terms as follows:

- $I^+_{j,s}$ indicates the set of examples (subset of $I$) whose feature value in dimension $j$ is greater than or equal to split point $s$;
- $I^-_{j,s}$ indicates the set of examples (subset of $I$) whose feature value in dimension $j$ is less than $s$;
- $\hat{y}^+_{j,s}$ is the average $y$ value of the data points indicated by set $I^+_{j,s}$; and
- $\hat{y}^-_{j,s}$ is the average $y$ value of the data points indicated by set $I^-_{j,s}$.

Here is the pseudocode. In what follows, $k$ is the largest leaf size that we will allow in the tree, and is a hyperparameter of the algorithm.

```
BUILDTREE($I, k$)
if $|I| \leqslant k$
    Set $\hat{y} = \text{average}_{i \in I}\, y^{(i)}$
    return LEAF(value $= \hat{y}$)
else
    for each split dimension $j$ and split value $s$
        Set $I^+_{j,s} = \{i \in I \mid x^{(i)}_j \geqslant s\}$
        Set $I^-_{j,s} = \{i \in I \mid x^{(i)}_j < s\}$
        Set $\hat{y}^+_{j,s} = \text{average}_{i \in I^+_{j,s}}\, y^{(i)}$
        Set $\hat{y}^-_{j,s} = \text{average}_{i \in I^-_{j,s}}\, y^{(i)}$
        Set $E_{j,s} = \sum_{i \in I^+_{j,s}} (y^{(i)} - \hat{y}^+_{j,s})^2 + \sum_{i \in I^-_{j,s}} (y^{(i)} - \hat{y}^-_{j,s})^2$
    Set $(j^*, s^*) = \arg\min_{j,s} E_{j,s}$
return NODE($j^*, s^*$, BUILDTREE($I^-_{j^*,s^*}, k$), BUILDTREE($I^+_{j^*,s^*}, k$))
```

In practice, we typically start by calling BUILDTREE with the first input equal to our whole data set (that is, with $I = \{1, \ldots, n\}$). But then that call of BUILDTREE can recursively lead to many other calls of BUILDTREE.

Let's think about how long each call of BUILDTREE takes to run. We have to consider all possible splits. So we consider a split in each of the $d$ dimensions. In each dimension, we only need to consider splits between two data points (any other split will give the same error on the training data). So, in total, we consider $O(dn)$ splits in each call to BUILDTREE.

**Study Question:** Concretely, what would be a good set of split-points to consider for dimension $j$ of a data set indicated by $I$?

#### 13.2.1.2 Pruning

It might be tempting to regularize by using a somewhat large value of $k$, or by stopping when splitting a node does not significantly decrease the error. One problem with short-sighted stopping criteria is that they might not see the value of a split that will require one more split before it seems useful.

**Study Question:** Apply the decision-tree algorithm to the XOR problem in two dimensions. What is the training-set error of all possible hypotheses based on a single split?

So, we will tend to build a tree that is too large, and then prune it back.

We define *cost complexity* of a tree $T$, where $m$ ranges over its leaves, as

$$C_\alpha(T) = \sum_{m=1}^{|T|} E_m(T) + \alpha|T| \quad , \tag{13.4}$$

and $|T|$ is the number of leaves. For a fixed $\alpha$, we can find a $T$ that (approximately) minimizes $C_\alpha(T)$ by "weakest-link" pruning:

- Create a sequence of trees by successively removing the bottom-level split that minimizes the increase in overall error, until the root is reached.
- Return the $T$ in the sequence that minimizes the cost complexity.

We can choose an appropriate $\alpha$ using cross validation.

### 13.2.2 Classification

The strategy for building and pruning classification trees is very similar to the strategy for regression trees.

Given a region $R_m$ corresponding to a leaf of the tree, we would pick the output class $y$ to be the value that exists most frequently (the *majority value*) in the data points whose $x$ values are in that region, i.e., data points indicated by $I_m$:

$$O_m = \text{majority}_{i \in I_m} \, y^{(i)} \quad .$$

Let's now define the error in a region as the number of data points that do not have the value $O_m$:

$$E_m = \left| \{ i \mid i \in I_m \text{ and } y^{(i)} \neq O_m \} \right| \quad .$$

We define the *empirical probability* of an item from class $k$ occurring in region $m$ as:

$$\hat{P}_{m,k} = \hat{P}(I_m, k) = \frac{\left| \{ i \mid i \in I_m \text{ and } y^{(i)} = k \} \right|}{N_m} \quad ,$$

where $N_m$ is the number of training points in region $m$; that is, $N_m = |I_m|$. For later use, we'll also define the empirical probabilities of split values, $\hat{P}_{m,j,s}$, as the fraction of points with dimension $j$ in split $s$ occurring in region $m$ (one branch of the tree), and $1 - \hat{P}_{m,j,s}$ as the complement (the fraction of points in the other branch).

**Splitting criteria** In our greedy algorithm, we need a way to decide which split to make next. There are many criteria that express some measure of the "impurity" in child nodes. Some measures include:

- *Misclassification error*:
$$Q_m(T) = \frac{E_m}{N_m} = 1 - \hat{P}_{m,O_m} \tag{13.5}$$
- *Gini index*:
$$Q_m(T) = \sum_k \hat{P}_{m,k}(1 - \hat{P}_{m,k}) \tag{13.6}$$
- *Entropy*:
$$Q_m(T) = H(I_m) = -\sum_k \hat{P}_{m,k} \log_2 \hat{P}_{m,k} \tag{13.7}$$

So that the entropy H is well-defined when $\hat{P} = 0$, we will stipulate that $0 \log_2 0 = 0$.

These splitting criteria are very similar, and it's not entirely obvious which one is better. We will focus on entropy, just to be concrete.

Analogous to how for regression we choose the dimension j and split s that minimizes the sum of squared error $E_{j,s}$, for classification, we choose the dimension j and split s that minimizes the weighted average entropy over the "child" data points in each of the two corresponding splits, $I^+_{j,s}$ and $I^-_{j,s}$. We calculate the entropy in each split based on the empirical probabilities of class memberships in the split, and then calculate the weighted average entropy $\hat{H}$ as

$$\begin{aligned}
\hat{H} &= (\text{fraction of points in left data set}) \cdot H(I^-_{j,s}) \\
&\qquad + (\text{fraction of points in right data set}) \cdot H(I^+_{j,s}) \\
&= (1 - \hat{P}_{m,j,s})H(I^-_{j,s}) + \hat{P}_{m,j,s}H(I^+_{j,s}) \\
&= \frac{|I^-_{j,s}|}{N_m} \cdot H(I^-_{j,s}) + \frac{|I^+_{j,s}|}{N_m} \cdot H(I^+_{j,s}) \; .
\end{aligned} \tag{13.8}$$

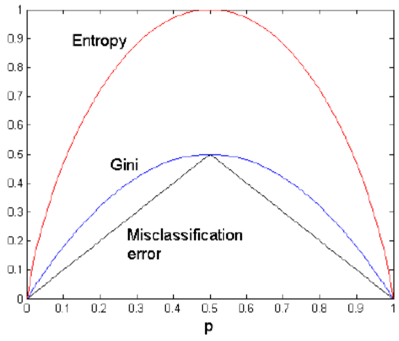
Choosing the split that minimizes the entropy of the children is equivalent to maximizing the *information gain* of the test $x_j = s$, defined by

$$\text{INFOGAIN}(x_j = s, I_m) \quad = \quad H(I_m) - \left( \frac{|I^-_{j,s}|}{N_m} \cdot H(I^-_{j,s}) + \frac{|I^+_{j,s}|}{N_m} \cdot H(I^+_{j,s}) \right) \tag{13.9}$$

In the two-class case (with labels 0 and 1), all of the splitting criteria mentioned above have the values

$$\begin{cases} 0.0 & \text{when } \hat{P}_{m,0} = 0.0 \\ 0.0 & \text{when } \hat{P}_{m,0} = 1.0 \end{cases} .$$

The respective impurity curves are shown below, where $p = \hat{P}_{m,0}$; the vertical axis plots $Q_m(T)$ for each of the three criteria.



There used to be endless haggling about which impurity function one should use. It seems to be traditional to use *entropy* to select which node to split while growing the tree, and *misclassification error* in the pruning criterion.

### 13.2.3 Bagging

One important limitation or drawback in conventional trees is that they can have high estimation error: small changes in the data can result in very big changes in the resulting tree.

*Bootstrap aggregation* is a technique for reducing the estimation error of a non-linear predictor, or one that is adaptive to the data. The key idea applied to trees, is to build multiple trees with different subsets of the data, and then create an ensemble model that combines the results from multiple trees to make a prediction.

- Construct $B$ new data sets of size $n$. Each data set is constructed by sampling $n$ data points with replacement from $\mathcal{D}$. A single data set is called *bootstrap sample* of $\mathcal{D}$.
- Train a predictor $\hat{f}^b(x)$ on each bootstrap sample.
- *Regression case*: bagged predictor is

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^b(x)\ . \tag{13.10}$$

- *Classification case*: Let $K$ be the number of classes. We find a majority bagged predictor as follows. We let $\hat{f}^b(x)$ be a "one-hot" vector with a single 1 and $K-1$ zeros, and define the predicted output $\hat{y}$ for predictor $f^b$ as $\hat{y}^b(x) = \arg\max_k \hat{f}^b(x)_k$. Then

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^b(x), \tag{13.11}$$

which is a vector containing the proportion of classifiers that predicted each class $k$ for input $x$. Then the overall predicted output is

$$\hat{y}_{\text{bag}}(x) = \arg\max_k \hat{f}_{\text{bag}}(x)_k\ . \tag{13.12}$$

There are theoretical arguments showing that bagging does, in fact, reduce estimation error. However, when we bag a model, any simple intrepetability is lost.

### 13.2.4 Random Forests

Random forests are collections of trees that are constructed to be de-correlated, so that using them to vote gives maximal advantage. In competitions, they often have excellent classification performance among large collections of much fancier methods.

In what follows, $B$, $m$, and $n$ are hyperparameters of the algorithm.

```
RandomForest($\mathcal{D}$; $B, m, n$)
for $b = 1, \dots, B$
    Draw a bootstrap sample $\mathcal{D}_b$ of size $n$ from $\mathcal{D}$
    Grow a tree $T_b$ on data $\mathcal{D}_b$ by recursively:
        Select $m$ variables at random from the $d$ variables
        Pick the best variable and split point among the $m$ variables
        Split the node
return tree $T_b$
```

Given the ensemble of trees, vote to make a prediction on a new $x$.

### 13.2.5 Tree variants and tradeoffs

There are many variations on the tree theme. One is to employ different regression or classification methods in each leaf. For example, a linear regression might be used to model the examples in each leaf, rather than using a constant value.

In the relatively simple trees that we've considered, splits have been based on only a single feature at a time, and with the resulting splits being axis-parallel. Other methods for splitting are possible, including consideration of multiple features and linear classifiers based on those, potentially resulting in non-axis-parallel splits. Complexity is a concern in such cases, as many possible combinations of features may need to be considered, to select the best variable combination (rather than a single split variable).

Another generalization is a *hierarchical mixture of experts*, where we make a "soft" version of trees, in which the splits are probabilistic (so every point has some degree of membership in every leaf). Such trees can be trained using a form of gradient descent. Combinations of bagging, boosting, and mixture tree approaches (e.g., *gradient boosted trees*) and implementations are readily available (e.g., XGBoost).

Trees have a number of strengths, and remain a valuable tool in the machine learning toolkit. Some benefits include being relatively easy to interpret, fast to train, and ability to handle multi-class classification in a natural way. Trees can easily handle different loss functions; one just needs to change the predictor and loss being applied in the leaves. Methods also exist to identify which features are particularly important or influential in forming the tree, which can aid in human understanding of the data set. Finally, in many situations, trees perform surprisingly well, often comparable to more complicated regression or classification models. Indeed, in some settings it is considered good practice to start with trees (especially random forest or boosted trees) as a "baseline" machine learning model, against which one can evaluate performance of more sophisticated models.

# APPENDIX A

## Matrix derivative common cases

What are some conventions for derivatives of matrices and vectors? It will always work to explicitly write all indices and treat everything as scalars, but we introduce here some shortcuts that are often faster to use and helpful for understanding.

There are at least two consistent but different systems for describing shapes and rules for doing matrix derivatives. In the end, they all are correct, but it is important to be consistent.

We will use what is often called the 'Hessian' or denominator layout, in which we say that for

$\mathbf{x}$ of size $n \times 1$ and $\mathbf{y}$ of size $m \times 1$, $\partial \mathbf{y}/\partial \mathbf{x}$ is a matrix of size $n \times m$ with the $(i, j)$ entry $\partial y_j/\partial x_i$. This denominator layout convention has been adopted by the field of machine learning to ensure that the shape of the gradient is the same as the shape of the shape of the respective derivative. This is somewhat controversial at large, but alas, we shall continue with denominator layout.

The discussion below closely follows the Wikipedia on matrix derivatives.

## A.1 The shapes of things

Here are important special cases of the rule above:

- Scalar-by-scalar: For $x$ of size $1 \times 1$ and $y$ of size $1 \times 1$, $\partial y/\partial x$ is the (scalar) partial derivative of $y$ with respect to $x$.
- Scalar-by-vector: For $\mathbf{x}$ of size $n \times 1$ and $y$ of size $1 \times 1$, $\partial y/\partial \mathbf{x}$ (also written $\nabla_{\mathbf{x}} y$, the gradient of $y$ with respect to $\mathbf{x}$) is a column vector of size $n \times 1$ with the $i^{\text{th}}$ entry $\partial y/\partial x_i$:

$$\partial y/\partial \mathbf{x} = \begin{bmatrix} \partial y/\partial x_1 \\ \partial y/\partial x_2 \\ \vdots \\ \partial y/\partial x_n \end{bmatrix}.$$

- Vector-by-scalar: For $x$ of size $1 \times 1$ and $\mathbf{y}$ of size $m \times 1$, $\partial \mathbf{y}/\partial x$ is a row vector of size $1 \times m$ with the $j^{\text{th}}$ entry $\partial \mathbf{y}_j/\partial x$:

$$\partial \mathbf{y}/\partial x = \begin{bmatrix} \partial y_1/\partial x & \partial y_2/\partial x & \cdots & \partial y_m/\partial x \end{bmatrix}.$$

- Vector-by-vector: For $\mathbf{x}$ of size $n \times 1$ and $\mathbf{y}$ of size $m \times 1$, $\partial \mathbf{y}/\partial \mathbf{x}$ is a matrix of size $n \times m$ with the $(i,j)$ entry $\partial y_j/\partial x_i$:

$$\partial \mathbf{y}/\partial \mathbf{x} = \begin{bmatrix} \partial y_1/\partial x_1 & \partial y_2/\partial x_1 & \cdots & \partial y_m/\partial x_1 \\ \partial y_1/\partial x_2 & \partial y_2/\partial x_2 & \cdots & \partial y_m/\partial x_2 \\ \vdots & \vdots & \ddots & \vdots \\ \partial y_1/\partial x_n & \partial y_2/\partial x_n & \cdots & \partial y_m/\partial x_n \end{bmatrix}.$$

- Scalar-by-matrix: For $\mathbf{X}$ of size $n \times m$ and $y$ of size $1 \times 1$, $\partial y/\partial \mathbf{X}$ (also written $\nabla_{\mathbf{X}} y$, the gradient of $y$ with respect to $\mathbf{X}$) is a matrix of size $n \times m$ with the $(i,j)$ entry $\partial y/\partial X_{i,j}$:

$$\partial y/\partial \mathbf{X} = \begin{bmatrix} \partial y/\partial X_{1,1} & \cdots & \partial y/\partial X_{1,m} \\ \vdots & \ddots & \vdots \\ \partial y/\partial X_{n,1} & \cdots & \partial y/\partial X_{n,m} \end{bmatrix}.$$

You may notice that in this list, we have not included matrix-by-matrix, matrix-by-vector, or vector-by-matrix derivatives. This is because, generally, they cannot be expressed nicely in matrix form and require higher order objects (e.g., tensors) to represent their derivatives. These cases are beyond the scope of this course.

Additionally, notice that for all cases, you can explicitly compute each element of the derivative object using (scalar) partial derivatives. You may find it useful to work through some of these by hand as you are reviewing matrix derivatives.

## A.2 Some vector-by-vector identities

Here are some examples of $\partial \mathbf{y}/\partial \mathbf{x}$. In each case, assume $\mathbf{x}$ is $n \times 1$, $\mathbf{y}$ is $m \times 1$, $a$ is a scalar constant, $\mathbf{a}$ is a vector that does not depend on $\mathbf{x}$ and $\mathbf{A}$ is a matrix that does not depend on $\mathbf{x}$, $u$ and $v$ are scalars that do depend on $\mathbf{x}$, and $\mathbf{u}$ and $\mathbf{v}$ are vectors that do depend on $\mathbf{x}$. We also have vector-valued functions $\mathbf{f}$ and $\mathbf{g}$.

### A.2.1 Some fundamental cases

First, we will cover a couple of fundamental cases: suppose that $\mathbf{a}$ is an $m \times 1$ vector which is not a function of $\mathbf{x}$, an $n \times 1$ vector. Then,

$$\frac{\partial \mathbf{a}}{\partial \mathbf{x}} = \mathbf{0}, \tag{A.1}$$

is an $n \times m$ matrix of 0s. This is similar to the scalar case of differentiating a constant. Next, we can consider the case of differentiating a vector with respect to itself:

$$\frac{\partial \mathbf{x}}{\partial \mathbf{x}} = \mathbf{I} \tag{A.2}$$

This is the $n \times n$ identity matrix, with 1's along the diagonal and 0's elsewhere. It makes sense, because $\partial \mathbf{x}_j/\mathbf{x}_i$ is 1 for $i = j$ and 0 otherwise. This identity is also similar to the scalar case.

### A.2.2 Derivatives involving a constant matrix

Let the dimensions of $\mathbf{A}$ be $m \times n$. Then the object $\mathbf{Ax}$ is an $m \times 1$ vector. We can then compute the derivative of $\mathbf{Ax}$ with respect to $\mathbf{x}$ as:

$$\frac{\partial \mathbf{Ax}}{\partial \mathbf{x}} = \begin{bmatrix} \partial(\mathbf{Ax})_1/\partial x_1 & \partial(\mathbf{Ax})_2/\partial x_1 & \cdots & \partial(\mathbf{Ax})_m/\partial x_1 \\ \partial(\mathbf{Ax})_1/\partial x_2 & \partial(\mathbf{Ax})_2/\partial x_2 & \cdots & \partial(\mathbf{Ax})_m/\partial x_2 \\ \vdots & \vdots & \ddots & \vdots \\ \partial(\mathbf{Ax})_1/\partial x_n & \partial(\mathbf{Ax})_2/\partial x_n & \cdots & \partial(\mathbf{Ax})_m/\partial x_n \end{bmatrix} \tag{A.3}$$

Note that any element of the column vector $\mathbf{Ax}$ can be written as, for $j = 1, \dots, m$:

$$(\mathbf{Ax})_j = \sum_{k=1}^{n} A_{j,k} x_k.$$

Thus, computing the $(i, j)$ entry of $\frac{\partial \mathbf{Ax}}{\partial \mathbf{x}}$ requires computing the partial derivative $\partial(\mathbf{Ax})_j/\partial x_i$ :

$$\partial(\mathbf{Ax})_j/\partial x_i = \partial \left( \sum_{k=1}^{n} A_{j,k} x_k \right) /\partial x_i = A_{j,i}$$

Therefore, the $(i, j)$ entry of $\frac{\partial \mathbf{Ax}}{\partial \mathbf{x}}$ is the $(j, i)$ entry of $\mathbf{A}$:

$$\frac{\partial \mathbf{Ax}}{\partial \mathbf{x}} = \mathbf{A}^T \tag{A.4}$$

Similarly, for objects $\mathbf{x}, \mathbf{A}$ of the same shape, on can obtain,

$$\frac{\partial \mathbf{x}^T \mathbf{A}}{\partial \mathbf{x}} = \mathbf{A} \tag{A.5}$$

### A.2.3 Linearity of derivatives

Suppose that $\mathbf{u}, \mathbf{v}$ are both vectors of size $m \times 1$. Then,

$$\frac{\partial(\mathbf{u} + \mathbf{v})}{\partial \mathbf{x}} = \frac{\partial \mathbf{u}}{\partial \mathbf{x}} + \frac{\partial \mathbf{v}}{\partial \mathbf{x}} \tag{A.6}$$

Suppose that $a$ is a scalar constant and $\mathbf{u}$ is an $m \times 1$ vector that is a function of $\mathbf{x}$. Then,

$$\frac{\partial a\mathbf{u}}{\partial \mathbf{x}} = a \frac{\partial \mathbf{u}}{\partial \mathbf{x}} \tag{A.7}$$

One can extend the previous identity to vector- and matrix-valued constants. Suppose that $\mathbf{a}$ is a vector with shape $m \times 1$ and $v$ is a scalar which depends on $\mathbf{x}$. Then,

$$\frac{\partial v\mathbf{a}}{\partial \mathbf{x}} = \frac{\partial v}{\partial \mathbf{x}} \mathbf{a}^T \tag{A.8}$$

First, checking dimensions, $\partial v/\partial \mathbf{x}$ is $n \times 1$ and $\mathbf{a}$ is $m \times 1$ so $\mathbf{a}^T$ is $1 \times m$ and our answer is $n \times m$ as it should be. Now, checking a value, element $(i, j)$ of the answer is $\partial v \mathbf{a}_j/\partial x_i = (\partial v/\partial x_i)\mathbf{a}_j$ which corresponds to element $(i, j)$ of $(\partial v/\partial \mathbf{x})\mathbf{a}^T$.

Similarly, suppose that $\mathbf{A}$ is a matrix which does not depend on $\mathbf{x}$ and $\mathbf{u}$ is a column vector which does depend on $\mathbf{x}$. Then,

$$\frac{\partial \mathbf{Au}}{\partial \mathbf{x}} = \frac{\partial \mathbf{u}}{\partial \mathbf{x}} \mathbf{A}^T \tag{A.9}$$

### A.2.4 Product rule (vector-valued numerator)

Suppose that $v$ is a scalar which depends on $\mathbf{x}$, while $\mathbf{u}$ is a column vector of shape $m \times 1$ and $\mathbf{x}$ is a column vector of shape $n \times 1$. Then,

$$\frac{\partial v\mathbf{u}}{\partial \mathbf{x}} = v\frac{\partial \mathbf{u}}{\partial \mathbf{x}} + \frac{\partial v}{\partial \mathbf{x}}\mathbf{u}^T \tag{A.10}$$

One can see this relationship by expanding the derivative as follows:

$$\frac{\partial v\mathbf{u}}{\partial \mathbf{x}} = \begin{bmatrix} \partial(vu_1)/\partial x_1 & \partial(vu_2)/\partial x_1 & \cdots & \partial(vu_m)/\partial x_1 \\ \partial(vu_1)/\partial x_2 & \partial(vu_2)/\partial x_2 & \cdots & \partial(vu_m)/\partial x_2 \\ \vdots & \vdots & \ddots & \vdots \\ \partial(vu_1)/\partial x_n & \partial(vu_2)/\partial x_n & \cdots & \partial(vu_m)/\partial x_n \end{bmatrix}.$$

Then, one can use the product rule for scalar-valued functions,

$$\partial(vu_j)/\partial x_i = v(\partial u_j/\partial x_i) + (\partial v/\partial x_i)u_j,$$

to obtain the desired result.

### A.2.5 Chain rule

Suppose that $\mathbf{g}$ is a vector-valued function with output vector of shape $m \times 1$, and the argument to $\mathbf{g}$ is a column vector $\mathbf{u}$ of shape $d \times 1$ which depends on $\mathbf{x}$. Then, one can obtain the chain rule as,

$$\frac{\partial \mathbf{g}(\mathbf{u})}{\partial \mathbf{x}} = \frac{\partial \mathbf{u}}{\partial \mathbf{x}}\frac{\partial \mathbf{g}(\mathbf{u})}{\partial \mathbf{u}} \tag{A.11}$$

Following "the shapes of things," $\partial\mathbf{u}/\partial\mathbf{x}$ is $n \times d$ and $\partial\mathbf{g}(\mathbf{u})/\partial\mathbf{u}$ is $d \times m$, where element $(i, j)$ is $\partial\mathbf{g}(\mathbf{u})_j/\partial\mathbf{u}_i$. The same chain rule applies for further compositions of functions:

$$\frac{\partial \mathbf{f}(\mathbf{g}(\mathbf{u}))}{\partial \mathbf{x}} = \frac{\partial \mathbf{u}}{\partial \mathbf{x}}\frac{\partial \mathbf{g}(\mathbf{u})}{\partial \mathbf{u}}\frac{\partial \mathbf{f}(\mathbf{g})}{\partial \mathbf{g}} \tag{A.12}$$

## A.3 Some other identities

You can get many scalar-by-vector and vector-by-scalar cases as special cases of the rules above, making one of the relevant vectors just be 1 x 1. Here are some other ones that are handy. For more, see the Wikipedia article on Matrix derivatives (for consistency, only use the ones in *denominator layout*).

$$\frac{\partial \mathbf{u}^T\mathbf{v}}{\partial \mathbf{x}} = \frac{\partial \mathbf{u}}{\partial \mathbf{x}}\mathbf{v} + \frac{\partial \mathbf{v}}{\partial \mathbf{x}}\mathbf{u} \tag{A.13}$$

$$\frac{\partial \mathbf{u}^T}{\partial x} = \left(\frac{\partial \mathbf{u}}{\partial x}\right)^T \tag{A.14}$$

## A.4 Derivation of gradient for linear regression

Applying identities A.5, A.13, A.6, A.4 A.1

$$\begin{aligned}\frac{\partial(\tilde{\mathbf{X}}\theta-\tilde{\mathbf{Y}})^T(\tilde{\mathbf{X}}\theta-\tilde{\mathbf{Y}})/n}{\partial\theta} &= \frac{2}{n}\frac{\partial(\tilde{\mathbf{X}}\theta-\tilde{\mathbf{Y}})}{\partial\theta}(\tilde{\mathbf{X}}\theta-\tilde{\mathbf{Y}}) \\ &= \frac{2}{n}\Big(\frac{\partial\tilde{\mathbf{X}}\theta}{\partial\theta}-\frac{\partial\tilde{\mathbf{Y}}}{\partial\theta}\Big)(\tilde{\mathbf{X}}\theta-\tilde{\mathbf{Y}}) \\ &= \frac{2}{n}\big(\tilde{\mathbf{X}}^T-\mathbf{0}\big)(\tilde{\mathbf{X}}\theta-\tilde{\mathbf{Y}}) \\ &= \frac{2}{n}\tilde{\mathbf{X}}^T(\tilde{\mathbf{X}}\theta-\tilde{\mathbf{Y}})\end{aligned}$$

## A.5 Matrix derivatives using Einstein summation

*You do not have to read or learn this! But you might find it interesting or helpful.*

Consider the objective function for linear regression, written out as products of matrices:

$$J(\theta)=\frac{1}{n}(\tilde{X}\theta-\tilde{Y})^T(\tilde{X}\theta-\tilde{Y})\,, \tag{A.15}$$

where $\tilde{X}=X^T$ is $n\times d$, $\tilde{Y}=Y^T$ is $n\times 1$, and $\theta$ is $d\times 1$. How does one show, with no shortcuts, that

$$\nabla_\theta J=\frac{2}{n}\tilde{X}^T(\tilde{X}\theta-\tilde{Y})\ \ ? \tag{A.16}$$

One neat way, which is very explicit, is to simply write all the matrices as variables with row and column indices, e.g., $\tilde{X}_{ab}$ is the row $a$, column $b$ entry of the matrix $\tilde{X}$. Furthermore, let us use the convention that in any product, all indices which appear more than once get summed over; this is a popular convention in theoretical physics, and lets us suppress all the summation symbols which would otherwise clutter the following expresssions. For example, $\tilde{X}_{ab}\theta_b$ would be the implicit summation notation giving the element at the $a^{\text{th}}$ row of the matrix-vector product $\tilde{X}\theta$.

Using implicit summation notation with explicit indices, we can rewrite $J(\theta)$ as

$$J(\theta)=\frac{1}{n}\left(\tilde{X}_{ab}\theta_b-\tilde{Y}_a\right)\left(\tilde{X}_{ac}\theta_c-\tilde{Y}_a\right)\,. \tag{A.17}$$

Note that we no longer need the transpose on the first term, because all that transpose accomplished was to take a dot product between the vector given by the left term, and the vector given by the right term. With implicit summation, this is accomplished by the two terms sharing the repeated index $a$.

Taking the derivative of $J$ with respect to the $d^{\text{th}}$ element of $\theta$ thus gives, using the chain rule for (ordinary scalar) multiplication:

$$\frac{dJ}{d\theta_d} = \frac{1}{n}\left[\tilde{X}_{ab}\delta_{bd}\left(\tilde{X}_{ac}\theta_c-\tilde{Y}_a\right)+\left(\tilde{X}_{ab}\theta_b-\tilde{Y}_a\right)\tilde{X}_{ac}\delta_{cd}\right] \tag{A.18}$$

$$= \frac{1}{n}\left[\tilde{X}_{ad}\left(\tilde{X}_{ac}\theta_c-\tilde{Y}_a\right)+\left(\tilde{X}_{ab}\theta_b-\tilde{Y}_a\right)\tilde{X}_{ad}\right] \tag{A.19}$$

$$= \frac{2}{n}\tilde{X}_{ad}\left(\tilde{X}_{ab}\theta_b-\tilde{Y}_a\right)\,, \tag{A.20}$$

where the second line follows from the first, with the definition that $\delta_{bd} = 1$ only when $b = d$ (and similarly for $\delta_{cd}$). And the third line follows from the second by recognizing that the two terms in the second line are identical. Now note that in this implicit summation notation, the $a, b$ element of the matrix product of $A$ and $B$ is $(AB)_{ac} = A_{ab}B_{bc}$. That is, ordinary matrix multiplication sums over indices which are adjacent to each other, because a row of $A$ times a column of $B$ becomes a scalar number. So the term in the above equation with $\tilde{X}_{ad}\tilde{X}_{ab}$ is not a matrix product of $\tilde{X}$ with $\tilde{X}$. However, taking the transpose $\tilde{X}^T$ switches row and column indices, so $\tilde{X}_{ad} = \tilde{X}^T_{da}$. And $\tilde{X}^T_{da}\tilde{X}_{ab}$ *is* a matrix product of $\tilde{X}^T$ with $\tilde{X}$! Thus, we have that

$$\begin{aligned} \frac{dJ}{d\theta_d} &= \frac{2}{n}\tilde{X}^T_{da}\left(\tilde{X}_{ab}\theta_b - \tilde{Y}_a\right) && \text{(A.21)} \\ &= \frac{2}{n}\left[\tilde{X}^T\left(\tilde{X}\theta - \tilde{Y}\right)\right]_d , && \text{(A.22)} \end{aligned}$$

which is the desired result.

# APPENDIX B

# Optimizing Neural Networks

### B.0.1 Strategies towards adaptive step-size

#### B.0.1.1 Running averages

We'll start by looking at the notion of a *running average*. It's a computational strategy for estimating a possibly weighted average of a sequence of data. Let our data sequence be $c_1, c_2, \ldots$; then we define a sequence of running average values, $C_0, C_1, C_2, \ldots$ using the equations

$$\begin{aligned} C_0 &= 0 \\ C_t &= \gamma_t C_{t-1} + (1-\gamma_t)c_t \end{aligned}$$

where $\gamma_t \in (0,1)$. If $\gamma_t$ is a constant, then this is a *moving* average, in which

$$\begin{aligned} C_T &= \gamma C_{T-1} + (1-\gamma)c_T \\ &= \gamma(\gamma C_{T-2} + (1-\gamma)c_{T-1}) + (1-\gamma)c_T \\ &= \sum_{t=1}^{T} \gamma^{T-t}(1-\gamma)c_t \end{aligned}$$

So, you can see that inputs $c_t$ closer to the end of the sequence T have more effect on $C_T$ than early inputs.

If, instead, we set $\gamma_t = (t-1)/t$, then we get the actual average.

**Study Question:** Prove to yourself that the previous assertion holds.

#### B.0.1.2 Momentum

Now, we can use methods that are a bit like running averages to describe strategies for computing $\eta$. The simplest method is *momentum*, in which we try to "average" recent gradient updates, so that if they have been bouncing back and forth in some direction, we take out that component of the motion. For momentum, we have

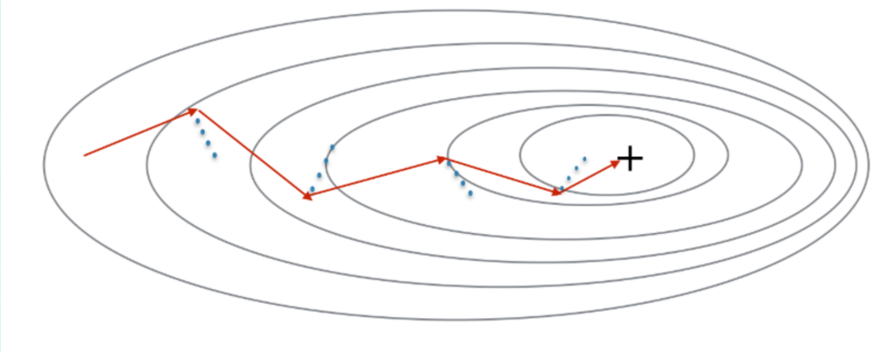
$$\begin{aligned} V_0 &= 0 \\ V_t &= \gamma V_{t-1} + \eta \nabla_W J(W_{t-1}) \\ W_t &= W_{t-1} - V_t \end{aligned}$$

This doesn't quite look like an adaptive step size. But what we can see is that, if we let $\eta = \eta'(1-\gamma)$, then the rule looks exactly like doing an update with step size $\eta'$ on a moving average of the gradients with parameter $\gamma$:

$$\begin{aligned} M_0 &= 0 \\ M_t &= \gamma M_{t-1} + (1-\gamma)\nabla_W J(W_{t-1}) \\ W_t &= W_{t-1} - \eta' M_t \end{aligned}$$

**Study Question:** Prove to yourself that these formulations are equivalent.

We will find that $V_t$ will be bigger in dimensions that consistently have the same sign for $\nabla_W$ and smaller for those that don't. Of course we now have *two* parameters to set ($\eta$ and $\gamma$), but the hope is that the algorithm will perform better overall, so it will be worth trying to find good values for them. Often $\gamma$ is set to be something like 0.9.



The red arrows show the update after each successive step of mini-batch gradient descent with momentum. The blue points show the direction of the gradient with respect to the mini-batch at each step. Momentum smooths the path taken towards the local minimum and leads to faster convergence.

**Study Question:** If you set $\gamma = 0.1$, would momentum have more of an effect or less of an effect than if you set it to 0.9?

### B.0.1.3 Adadelta

Another useful idea is this: we would like to take larger steps in parts of the space where $J(W)$ is nearly flat (because there's no risk of taking too big a step due to the gradient being large) and smaller steps when it is steep. We'll apply this idea to each weight independently, and end up with a method called *adadelta*, which is a variant on *adagrad* (for adaptive gradient). Even though our weights are indexed by layer, input unit and output unit, for simplicity here, just let $W_j$ be any weight in the network (we will do the same thing for all of them).

$$\begin{aligned} g_{t,j} &= \nabla_W J(W_{t-1})_j \\ G_{t,j} &= \gamma G_{t-1,j} + (1-\gamma) g_{t,j}^2 \\ W_{t,j} &= W_{t-1,j} - \frac{\eta}{\sqrt{G_{t,j} + \epsilon}} g_{t,j} \end{aligned}$$

The sequence $G_{t,j}$ is a moving average of the square of the jth component of the gradient. We square it in order to be insensitive to the sign—we want to know whether the magnitude is big or small. Then, we perform a gradient update to weight j, but divide the step size by $\sqrt{G_{t,j} + \epsilon}$, which is larger when the surface is steeper in direction j at point $W_{t-1}$ in weight space; this means that the step size will be smaller when it's steep and larger when it's flat.

#### B.0.1.4 Adam

Adam has become the default method of managing step sizes in neural networks. It combines the ideas of momentum and adadelta. We start by writing moving averages of the gradient and squared gradient, which reflect estimates of the mean and variance of the gradient for weight j:

Although, interestingly, it may actually violate the convergence conditions of SGD: arxiv.org/abs/1705.08292

$$\begin{aligned} g_{t,j} &= \nabla_W J(W_{t-1})_j \\ m_{t,j} &= B_1 m_{t-1,j} + (1 - B_1) g_{t,j} \\ v_{t,j} &= B_2 v_{t-1,j} + (1 - B_2) g_{t,j}^2 \quad . \end{aligned}$$

A problem with these estimates is that, if we initialize $m_0 = v_0 = 0$, they will always be biased (slightly too small). So we will correct for that bias by defining

$$\begin{aligned} \hat{m}_{t,j} &= \frac{m_{t,j}}{1 - B_1^t} \\ \hat{v}_{t,j} &= \frac{v_{t,j}}{1 - B_2^t} \\ W_{t,j} &= W_{t-1,j} - \frac{\eta}{\sqrt{\hat{v}_{t,j} + \epsilon}} \hat{m}_{t,j} \quad . \end{aligned}$$

Note that $B_1^t$ is $B_1$ raised to the power t, and likewise for $B_2^t$. To justify these corrections, note that if we were to expand $m_{t,j}$ in terms of $m_{0,j}$ and $g_{0,j}, g_{1,j}, \ldots, g_{t,j}$ the coefficients would sum to 1. However, the coefficient behind $m_{0,j}$ is $B_1^t$ and since $m_{0,j} = 0$, the sum of coefficients of non-zero terms is $1 - B_1^t$, hence the correction. The same justification holds for $v_{t,j}$.

Now, our update for weight j has a step size that takes the steepness into account, as in adadelta, but also tends to move in the same direction, as in momentum. The authors of this method propose setting $B_1 = 0.9, B_2 = 0.999, \epsilon = 10^{-8}$. Although we now have even more parameters, Adam is not highly sensitive to their values (small changes do not have a huge effect on the result).

**Study Question:** Define $\hat{m}_j$ directly as a moving average of $g_{t,j}$. What is the decay ($\gamma$ parameter)?

Even though we now have a step-size for each weight, and we have to update various quantities on each iteration of gradient descent, it's relatively easy to implement by maintaining a matrix for each quantity ($m_t^\ell, v_t^\ell, g_t^\ell, {g_t^2}^\ell$) in each layer of the network.

### B.0.2 Batch Normalization Details

Let's think of the batch-norm layer as taking $Z^l$ as input and producing an output $\widehat{Z}^l$ as output. But now, instead of thinking of $Z^l$ as an $n^l \times 1$ vector, we have to explicitly think about handling a mini-batch of data of size K, all at once, so $Z^l$ will be $n^l \times K$, and so will the output $\widehat{Z}^l$.

Our first step will be to compute the *batchwise* mean and standard deviation. Let $\mu^l$ be the $n^l \times 1$ vector where

$$\mu_i^l = \frac{1}{K}\sum_{j=1}^{K} Z_{ij}^l \ ,$$

and let $\sigma^l$ be the $n^l \times 1$ vector where

$$\sigma_i^l = \sqrt{\frac{1}{K}\sum_{j=1}^{K}(Z_{ij}^l - \mu_i)^2} \ .$$

The basic normalized version of our data would be a matrix, element $(i, j)$ of which is

$$\overline{Z}_{ij}^l = \frac{Z_{ij}^l - \mu_i^l}{\sigma_i^l + \epsilon} \ ,$$

where $\epsilon$ is a very small constant to guard against division by zero. However, if we let these be our $\widehat{Z}^l$ values, we really are forcing something too strong on our data—our goal was to normalize across the data batch, but not necessarily force the output values to have exactly mean 0 and standard deviation 1. So, we will give the layer the "opportunity" to shift and scale the outputs by adding new weights to the layer. These weights are $G^l$ and $B^l$, each of which is an $n^l \times 1$ vector. Using the weights, we define the final output to be

$$\widehat{Z}_{ij}^l = G_i^l \overline{Z}_{ij}^l + B_i^l \ .$$

That's the forward pass. Whew!

Now, for the backward pass, we have to do two things: given $\partial L/\partial \widehat{Z}^l$,

- Compute $\partial L/\partial Z^l$ for back-propagation, and
- Compute $\partial L/\partial G^l$ and $\partial L/\partial B^l$ for gradient updates of the weights in this layer.

Schematically

For simplicity we will drop the reference to the layer $l$ in the rest of the derivation

$$\frac{\partial L}{\partial B} = \frac{\partial L}{\partial \widehat{Z}}\frac{\partial \widehat{Z}}{\partial B} \ .$$

It's hard to think about these derivatives in matrix terms, so we'll see how it works for the components. $B_i$ contributes to $\widehat{Z}_{ij}$ for all data points $j$ in the batch. So

$$\begin{aligned}\frac{\partial L}{\partial B_i} &= \sum_j \frac{\partial L}{\partial \widehat{Z}_{ij}}\frac{\partial \widehat{Z}_{ij}}{\partial B_i} \\ &= \sum_j \frac{\partial L}{\partial \widehat{Z}_{ij}} \ ,\end{aligned}$$

Similarly, $G_i$ contributes to $\widehat{Z}_{ij}$ for all data points $j$ in the batch. So

$$\begin{aligned}\frac{\partial L}{\partial G_i} &= \sum_j \frac{\partial L}{\partial \widehat{Z}_{ij}}\frac{\partial \widehat{Z}_{ij}}{\partial G_i} \\ &= \sum_j \frac{\partial L}{\partial \widehat{Z}_{ij}} \overline{Z}_{ij} \ .\end{aligned}$$

Now, let's figure out how to do backprop. We can start schematically:

$$\frac{\partial L}{\partial Z} = \frac{\partial L}{\partial \widehat{Z}}\frac{\partial \widehat{Z}}{\partial Z} \ .$$

And because dependencies only exist across the batch, but not across the unit outputs,

$$\frac{\partial L}{\partial Z_{ij}} = \sum_{k=1}^{K} \frac{\partial L}{\partial \widehat{Z}_{ik}} \frac{\partial \widehat{Z}_{ik}}{\partial Z_{ij}} \quad .$$

The next step is to note that

$$\begin{aligned} \frac{\partial \widehat{Z}_{ik}}{\partial Z_{ij}} &= \frac{\partial \widehat{Z}_{ik}}{\partial \overline{Z}_{ik}} \frac{\partial \overline{Z}_{ik}}{\partial Z_{ij}} \\ &= G_i \frac{\partial \overline{Z}_{ik}}{\partial Z_{ij}} \quad . \end{aligned}$$

And now that

$$\frac{\partial \overline{Z}_{ik}}{\partial Z_{ij}} = \left(\delta_{jk} - \frac{\partial \mu_i}{\partial Z_{ij}}\right) \frac{1}{\sigma_i} - \frac{Z_{ik} - \mu_i}{\sigma_i^2} \frac{\partial \sigma_i}{\partial Z_{ij}} \quad ,$$

where $\delta_{jk} = 1$ if $j = k$ and $\delta_{jk} = 0$ otherwise. Getting close! We need two more small parts:

$$\begin{aligned} \frac{\partial \mu_i}{\partial Z_{ij}} &= \frac{1}{K} \, , \\ \frac{\partial \sigma_i}{\partial Z_{ij}} &= \frac{Z_{ij} - \mu_i}{K \sigma_i} \quad . \end{aligned}$$

Putting the whole crazy thing together, we get

$$\frac{\partial L}{\partial Z_{ij}} = \sum_{k=1}^{K} \frac{\partial L}{\partial \widehat{Z}_{ik}} G_i \frac{1}{K \sigma_i} \left( \delta_{jk} K - 1 - \frac{(Z_{ik} - \mu_i)(Z_{ij} - \mu_i)}{\sigma_i^2} \right) \quad .$$

# APPENDIX C

## Recurrent Neural Networks

So far, we have limited our attention to domains in which each output $y$ is assumed to have been generated as a function of an associated input $x$, and our hypotheses have been "pure" functions, in which the output depends only on the input (and the parameters we have learned that govern the function's behavior). In the next few chapters, we are going to consider cases in which our models need to go beyond functions. In particular, behavior as a function of *time* will be an important concept:

- In *recurrent neural networks*, the hypothesis that we learn is not a function of a single input, but of the whole sequence of inputs that the predictor has received.
- In *reinforcement learning*, the hypothesis is either a *model* of a domain (such as a game) as a recurrent system or a *policy* which is a pure function, but whose loss is determined by the ways in which the policy interacts with the domain over time.

In this chapter, we introduce *state machines*. We start with deterministic state machines, and then consider recurrent neural network (RNN) architectures to model their behavior. Later, in Chapter 11, we will study *Markov decision processes* (MDPs) that extend to consider probabilistic (rather than deterministic) transitions in our state machines. RNNs and MDPs will enable description and modeling of temporally sequential patterns of behavior that are important in many domains.

## C.1 State machines

A *state machine* is a description of a process (computational, physical, economic) in terms of its potential sequences of *states*.

This is such a pervasive idea that it has been given many names in many subareas of computer science, control theory, physics, etc., including: *automaton*, *transducer*, *dynamical system*, etc.

The *state* of a system is defined to be all you would need to know about the system to predict its future trajectories as well as possible. It could be the position and velocity of an object or the locations of your pieces on a game board, or the current traffic densities on a highway network.

Formally, we define a *state machine* as $(\mathcal{S}, \mathcal{X}, \mathcal{Y}, s_0, f_s, f_o)$ where

- $\mathcal{S}$ is a finite or infinite set of possible states;
- $\mathcal{X}$ is a finite or infinite set of possible inputs;

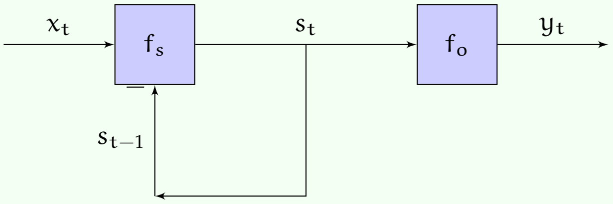
- $\mathcal{Y}$ is a finite or infinite set of possible outputs;
- $s_0 \in \mathcal{S}$ is the initial state of the machine;
- $f_s : \mathcal{S} \times \mathcal{X} \to \mathcal{S}$ is a *transition function*, which takes an input and a previous state and produces a next state;
- $f_o : \mathcal{S} \to \mathcal{Y}$ is an *output function*, which takes a state and produces an output.

The basic operation of the state machine is to start with state $s_0$, then iteratively compute for $t \geqslant 1$:

In some cases, we will pick a starting state from a set or distribution.

$$s_t = f_s(s_{t-1}, x_t) \tag{C.1}$$
$$y_t = f_o(s_t) \tag{C.2}$$

The diagram below illustrates this process. Note that the "feedback" connection of $s_t$ back into $f_s$ has to be buffered or delayed by one time step—-otherwise what it is computing would not generally be well defined.



So, given a sequence of inputs $x_1, x_2, \ldots$ the machine generates a sequence of outputs

$$\underbrace{f_o(f_s(s_0, x_1))}_{y_1}, \underbrace{f_o(f_s(f_s(s_0, x_1), x_2))}_{y_2}, \ldots \ .$$

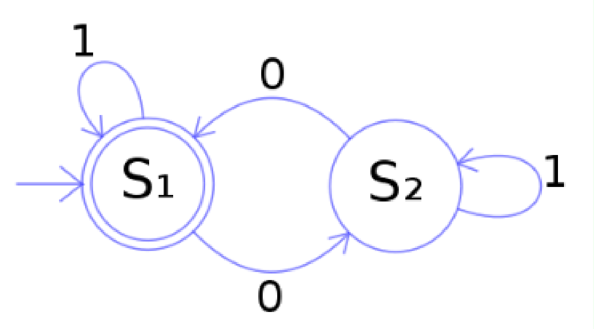
We sometimes say that the machine *transduces* sequence $x$ into sequence $y$. The output at time $t$ can have dependence on inputs from steps 1 to $t$.

There are a huge number of major and minor variations on the idea of a state machine. We'll just work with one specific one in this section and another one in the next, but don't worry if you see other variations out in the world!

One common form is *finite state machines*, in which $\mathcal{S}$, $\mathcal{X}$, and $\mathcal{Y}$ are all finite sets. They are often described using *state transition diagrams* such as the one below, in which nodes stand for states and arcs indicate transitions. Nodes are labeled by which output they generate and arcs are labeled by which input causes the transition.

All computers can be described, at the digital level, as finite state machines. Big, but finite!

One can verify that the state machine below reads binary strings and determines the parity of the number of zeros in the given string. Check for yourself that all input binary strings end in state $S_1$ if and only if they contain an even number of zeros.



Another common structure that is simple but powerful and used in signal processing and control is *linear time-invariant (LTI) systems*. In this case, all the quantities are real-valued vectors: $\mathcal{S} = \mathbb{R}^m$, $\mathcal{X} = \mathbb{R}^l$ and $\mathcal{Y} = \mathbb{R}^n$. The functions $f_s$ and $f_o$ are linear functions of their inputs. The transition function is described by the state matrix $A$ and the input matrix $B$; the output function is defined by the output matrix $C$, each with compatible dimensions. In discrete time, they can be defined by a linear difference equation, like

$$s_t = f_s(s_{t-1}, x_t) = As_{t-1} + Bx_t, \tag{C.3}$$

$$y_t = f_o(s_t) = Cs_t, \tag{C.4}$$

and can be implemented using state to store relevant previous input and output information. We will study *recurrent neural networks* which are a lot like a non-linear version of an LTI system.

## C.2 Recurrent neural networks

In Chapter 6, we studied neural networks and how the weights of a network can be obtained by training on data, so that the neural network will model a function that approximates the relationship between the $(x, y)$ pairs in a supervised-learning training set. In Section C.1 above, we introduced state machines to describe sequential temporal behavior. Here in Section C.2, we explore recurrent neural networks by defining the architecture and weight matrices in a neural network to enable modeling of such state machines. Then, in Section C.3, we present a loss function that may be employed for training *sequence to sequence* RNNs, and then consider application to language translation and recognition in Section C.4. In Section C.5, we'll see how to use gradient-descent methods to train the weights of an RNN so that it performs a *transduction* that matches as closely as possible a training set of input-output *sequences*.

A *recurrent neural network* is a state machine with neural networks constituting functions $f_s$ and $f_o$:

$$s_t = f_s\left(W^{sx}x_t + W^{ss}s_{t-1} + W_0^{ss}\right) \tag{C.5}$$

$$y_t = f_o\left(W^{o}s_t + W_0^{o}\right) \quad . \tag{C.6}$$

The inputs, states, and outputs are all vector-valued:

$$x_t : \ell \times 1 \tag{C.7}$$

$$s_t : m \times 1 \tag{C.8}$$

$$y_t : v \times 1 \quad . \tag{C.9}$$

The weights in the network, then, are

$$W^{sx} : m \times \ell \tag{C.10}$$

$$W^{ss} : m \times m \tag{C.11}$$

$$W_0^{ss} : m \times 1 \tag{C.12}$$

$$W^{o} : v \times m \tag{C.13}$$

$$W_0^{o} : v \times 1 \tag{C.14}$$

with activation functions $f_s$ and $f_o$.

**Study Question:** Check dimensions here to be sure it all works out. Remember that we apply $f_s$ and $f_o$ elementwise, unless $f_o$ is a softmax activation.

## C.3 Sequence-to-sequence RNN

Now, how can we set up an RNN to model and be trained to produce a transduction of one sequence to another? This problem is sometimes called *sequence-to-sequence* mapping. You can think of it as a kind of regression problem: given an input sequence, learn to generate the corresponding output sequence.

One way to think of training a sequence **classifier** is to reduce it to a transduction problem, where $y_t = 1$ if the sequence $x_1, \ldots, x_t$ is a *positive* example of the class of sequences and $-1$ otherwise.

A training set has the form $\left[\left(x^{(1)}, y^{(1)}\right), \ldots, \left(x^{(q)}, y^{(q)}\right)\right]$, where

- $x^{(i)}$ and $y^{(i)}$ are length $n^{(i)}$ sequences;
- sequences in the *same pair* are the same length; and sequences in different pairs may have different lengths.

Next, we need a loss function. We start by defining a loss function on sequences. There are many possible choices, but usually it makes sense just to sum up a per-element loss function on each of the output values, where $g$ is the predicted sequence and $y$ is the actual one:

$$\mathcal{L}_{\text{seq}}\left(g^{(i)}, y^{(i)}\right) = \sum_{t=1}^{n^{(i)}} \mathcal{L}_{\text{elt}}\left(g_t^{(i)}, y_t^{(i)}\right) \quad . \tag{C.15}$$

The per-element loss function $\mathcal{L}_{\text{elt}}$ will depend on the type of $y_t$ and what information it is encoding, in the same way as for a supervised network.

So it could be NLL, squared loss, etc.

Then, letting $W = \left(W^{sx}, W^{ss}, W^{o}, W_0^{ss}, W_0^{o}\right)$, our overall goal is to minimize the objective

$$J(W) = \frac{1}{q}\sum_{i=1}^{q} \mathcal{L}_{\text{seq}}\left(\text{RNN}(x^{(i)}; W), y^{(i)}\right) \quad , \tag{C.16}$$

where $\text{RNN}(x; W)$ is the output sequence generated, given input sequence $x$.

It is typical to choose $f_s$ to be *tanh* but any non-linear activation function is usable. We choose $f_o$ to align with the types of our outputs and the loss function, just as we would do in regular supervised learning.

Remember that it looks like a sigmoid but ranges from -1 to +1.

## C.4 RNN as a language model

A *language model* is a sequence to sequence RNN which is trained on a token sequence of the form, $c = (c_1, c_2, \ldots, c_k)$, and is used to predict the next token $c_t, t \leqslant k$, given a sequence of the previous $(t-1)$ tokens:

A "token" is generally a character, common word fragment, or a word.

$$c_t = \text{RNN}\left((c_1, c_2, \ldots, c_{t-1}); W\right) \tag{C.17}$$

We can convert this to a sequence-to-sequence training problem by constructing a data set of $q$ different $(x, y)$ sequence pairs, where we make up new special tokens, start and end, to signal the beginning and end of the sequence:

$$x = (\langle\text{start}\rangle, c_1, c_2, \ldots, c_k) \tag{C.18}$$
$$y = (c_1, c_2, \ldots, \langle\text{end}\rangle) \tag{C.19}$$

## C.5 Back-propagation through time

Now the fun begins! We can now try to find a $W$ to minimize $J$ using gradient descent. We will work through the simplest method, *back-propagation through time* (BPTT), in detail. This is generally not the best method to use, but it's relatively easy to understand. In Section C.6 we will sketch alternative methods that are in much more common use.

What we want you to take away from this section is that, by "unrolling" a recurrent network out to model a particular sequence, we can treat the whole thing as a feed-forward network with a lot of parameter sharing. Thus, we can tune the parameters using stochastic gradient descent, and learn to model sequential mappings. The concepts here are very important. While the details are important to get right if you need to implement something, we present the mathematical details below primarily to convey or explain the larger concepts.

**Calculus reminder: total derivative** Most of us are not very careful about the difference between the *partial derivative* and the *total derivative*. We are going to use a nice example from the Wikipedia article on partial derivatives to illustrate the difference. The volume of a circular cone depends on its height and radius:

$$V(r, h) = \frac{\pi r^2 h}{3} \ . \tag{C.20}$$

The partial derivatives of volume with respect to height and radius are

$$\frac{\partial V}{\partial r} = \frac{2\pi r h}{3} \quad \text{and} \quad \frac{\partial V}{\partial h} = \frac{\pi r^2}{3} \ . \tag{C.21}$$

They measure the change in $V$ assuming everything is held constant except the single variable we are changing. Now assume that we want to preserve the cone's proportions in the sense that the ratio of radius to height stays constant. Then we can't really change one without changing the other. In this case, we really have to think about the *total derivative*. If we're interested in the total derivative with respect to $r$, we sum the "paths" along which $r$ might influence $V$:

$$\begin{aligned} \frac{dV}{dr} &= \frac{\partial V}{\partial r} + \frac{\partial V}{\partial h}\frac{dh}{dr} && \text{(C.22)} \\ &= \frac{2\pi r h}{3} + \frac{\pi r^2}{3}\frac{dh}{dr} && \text{(C.23)} \end{aligned}$$

Or if we're interested in the total derivative with respect to $h$, we consider how $h$ might influence $V$, either directly or via $r$:

$$\begin{aligned} \frac{dV}{dh} &= \frac{\partial V}{\partial h} + \frac{\partial V}{\partial r}\frac{dr}{dh} && \text{(C.24)} \\ &= \frac{\pi r^2}{3} + \frac{2\pi r h}{3}\frac{dr}{dh} && \text{(C.25)} \end{aligned}$$

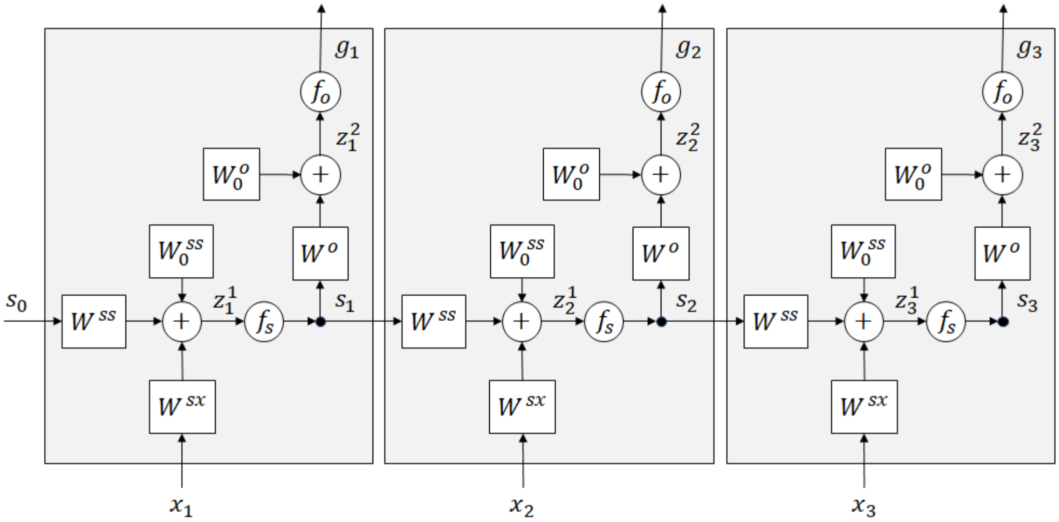
Just to be completely concrete, let's think of a right circular cone with a fixed angle $\alpha = \tan r/h$, so that if we change $r$ or $h$ then $\alpha$ remains constant. So we have $r = h\tan^{-1}\alpha$; let constant $c = \tan^{-1}\alpha$, so now $r = ch$. Thus, we finally have

$$\begin{aligned} \frac{dV}{dr} &= \frac{2\pi r h}{3} + \frac{\pi r^2}{3}\frac{1}{c} && \text{(C.26)} \\ \frac{dV}{dh} &= \frac{\pi r^2}{3} + \frac{2\pi r h}{3}c \ . && \text{(C.27)} \end{aligned}$$

The BPTT process goes like this:

(1) Sample a training pair of sequences $(x, y)$; let their length be $n$.

(2) "Unroll" the RNN to be length $n$ (picture for $n = 3$ below), and initialize $s_0$:



Now, we can see our problem as one of performing what is almost an ordinary back-propagation training procedure in a feed-forward neural network, but with the difference that the weight matrices are shared among the layers. In many ways, this is similar to what ends up happening in a convolutional network, except in the conv-net, the weights are re-used spatially, and here, they are re-used temporally.

(3) Do the *forward pass*, to compute the predicted output sequence $g$:

$$z_t^1 = W^{sx} x_t + W^{ss} s_{t-1} + W_0^{ss} \tag{C.28}$$

$$s_t = f_s(z_t^1) \tag{C.29}$$

$$z_t^2 = W^o s_t + W_0^o \tag{C.30}$$

$$g_t = f_o(z_t^2) \tag{C.31}$$

(4) Do *backward pass* to compute the gradients. For both $W^{ss}$ and $W^{sx}$ we need to find

$$\frac{d\mathcal{L}_{\text{seq}}(g, y)}{dW} = \sum_{u=1}^{n} \frac{d\mathcal{L}_{\text{elt}}(g_u, y_u)}{dW}$$

Letting $\mathcal{L}_u = \mathcal{L}_{\text{elt}}(g_u, y_u)$ and using the *total derivative*, which is a sum over all the ways in which $W$ affects $\mathcal{L}_u$, we have

$$= \sum_{u=1}^{n} \sum_{t=1}^{n} \frac{\partial s_t}{\partial W} \frac{\partial \mathcal{L}_u}{\partial s_t}$$

Re-organizing, we have

$$= \sum_{t=1}^{n} \frac{\partial s_t}{\partial W} \sum_{u=1}^{n} \frac{\partial \mathcal{L}_u}{\partial s_t}$$

Because $s_t$ only affects $\mathcal{L}_t, \mathcal{L}_{t+1}, \dots, \mathcal{L}_n$,

$$= \sum_{t=1}^{n} \frac{\partial s_t}{\partial W} \sum_{u=t}^{n} \frac{\partial \mathcal{L}_u}{\partial s_t}$$

$$= \sum_{t=1}^{n} \frac{\partial s_t}{\partial W} \left( \frac{\partial \mathcal{L}_t}{\partial s_t} + \underbrace{\sum_{u=t+1}^{n} \frac{\partial \mathcal{L}_u}{\partial s_t}}_{\delta^{s_t}} \right) . \tag{C.32}$$

where $\delta^{s_t}$ is the dependence of the future loss (incurred after step t) on the state $S_t$.

That is, $\delta^{s_t}$ is how much we can blame state $s_t$ for all the future element losses.

We can compute this backwards, with t going from n down to 1. The trickiest part is figuring out how early states contribute to later losses. We define the *future loss* after step t to be

$$F_t = \sum_{u=t+1}^{n} \mathcal{L}_{\text{elt}}(g_u, y_u) \ , \tag{C.33}$$

so

$$\delta^{s_t} = \frac{\partial F_t}{\partial s_t} \ . \tag{C.34}$$

At the last stage, $F_n = 0$ so $\delta^{s_n} = 0$.

Now, working backwards,

$$\delta^{s_{t-1}} = \frac{\partial}{\partial s_{t-1}} \sum_{u=t}^{n} \mathcal{L}_{\text{elt}}(g_u, y_u) \tag{C.35}$$

$$= \frac{\partial s_t}{\partial s_{t-1}} \frac{\partial}{\partial s_t} \sum_{u=t}^{n} \mathcal{L}_{\text{elt}}(g_u, y_u) \tag{C.36}$$

$$= \frac{\partial s_t}{\partial s_{t-1}} \frac{\partial}{\partial s_t} \left[ \mathcal{L}_{\text{elt}}(g_t, y_t) + \sum_{u=t+1}^{n} \mathcal{L}_{\text{elt}}(g_u, y_u) \right] \tag{C.37}$$

$$= \frac{\partial s_t}{\partial s_{t-1}} \left[ \frac{\partial \mathcal{L}_{\text{elt}}(g_t, y_t)}{\partial s_t} + \delta^{s_t} \right] \tag{C.38}$$

Now, we can use the chain rule again to find the dependence of the element loss at time t on the state at that same time,

$$\underbrace{\frac{\partial \mathcal{L}_{\text{elt}}(g_t, y_t)}{\partial s_t}}_{(m \times 1)} = \underbrace{\frac{\partial z_t^2}{\partial s_t}}_{(m \times v)} \underbrace{\frac{\partial \mathcal{L}_{\text{elt}}(g_t, y_t)}{\partial z_t^2}}_{(v \times 1)} , \tag{C.39}$$

and the dependence of the state at time t on the state at the previous time,

$$\underbrace{\frac{\partial s_t}{\partial s_{t-1}}}_{(m \times m)} = \underbrace{\frac{\partial z_t^1}{\partial s_{t-1}}}_{(m \times m)} \underbrace{\frac{\partial s_t}{\partial z_t^1}}_{(m \times m)} = W^{ss\,T} \frac{\partial s_t}{\partial z_t^1} \tag{C.40}$$

Note that $\partial s_t / \partial z_t^1$ is formally an $m \times m$ diagonal matrix, with the values along the diagonal being $f_s'(z_{t,i}^1)$, $1 \leqslant i \leqslant m$. But since this is a diagonal matrix, one could represent it as an $m \times 1$ vector $f_s'(z_t^1)$. In that case the product of the matrix $W^{ss^T}$ by the vector $f_s'(z_t^1)$, denoted $W^{ss^T} * f_s'(z_t^1)$, should be interpreted as follows: take the first column of the matrix $W^{ss^T}$ and multiply each of its elements by the first element of the vector $\partial s_t / \partial z_t^1$, then take the second column of the matrix $W^{ss^T}$ and multiply each of its elements by the second element of the vector $\partial s_t / \partial z_t^1$, and so on and so forth ...

Putting this all together, we end up with

$$\delta^{s_{t-1}} = \underbrace{W^{ss^T} \frac{\partial s_t}{\partial z_t^1}}_{\frac{\partial s_t}{\partial s_{t-1}}} \underbrace{\left( W^{o^T} \frac{\partial \mathcal{L}_t}{\partial z_t^2} + \delta^{s_t} \right)}_{\frac{\partial F_{t-1}}{\partial s_t}} \tag{C.41}$$

We're almost there! Now, we can describe the actual weight updates. Using Eq. C.32 and recalling the definition of $\delta^{s_t} = \partial F_t / \partial s_t$, as we iterate backwards, we can accumulate the terms in Eq. C.32 to get the gradient for the whole loss.

$$\frac{d\mathcal{L}_{\text{seq}}}{dW^{ss}} = \sum_{t=1}^{n} \frac{d\mathcal{L}_{\text{elt}}(g_t, y_t)}{dW^{ss}} = \sum_{t=1}^{n} \frac{\partial z_t^1}{\partial W^{ss}} \frac{\partial s_t}{\partial z_t^1} \frac{\partial F_{t-1}}{\partial s_t} \tag{C.42}$$

$$\frac{d\mathcal{L}_{\text{seq}}}{dW^{sx}} = \sum_{t=1}^{n} \frac{d\mathcal{L}_{\text{elt}}(g_t, y_t)}{dW^{sx}} = \sum_{t=1}^{n} \frac{\partial z_t^1}{\partial W^{sx}} \frac{\partial s_t}{\partial z_t^1} \frac{\partial F_{t-1}}{\partial s_t} \tag{C.43}$$

We can handle $W^o$ separately; it's easier because it does not affect future losses in the way that the other weight matrices do:

$$\frac{d\mathcal{L}_{\text{seq}}}{dW^o} = \sum_{t=1}^{n} \frac{d\mathcal{L}_t}{dW^o} = \sum_{t=1}^{n} \frac{\partial \mathcal{L}_t}{\partial z_t^2} \frac{\partial z_t^2}{\partial W^o} \tag{C.44}$$

Assuming we have $\frac{\partial \mathcal{L}_t}{\partial z_t^2} = (g_t - y_t)$, (which ends up being true for squared loss, softmax-NLL, etc.), then

$$\underbrace{\frac{d\mathcal{L}_{\text{seq}}}{dW^o}}_{v \times m} = \sum_{t=1}^{n} \underbrace{(g_t - y_t)}_{v \times 1} \underbrace{s_t^T}_{1 \times m} . \tag{C.45}$$

Whew!

**Study Question:** Derive the updates for the offsets $W_0^{ss}$ and $W_0^o$.

## C.6 Vanishing gradients and gating mechanisms

Let's take a careful look at the backward propagation of the gradient along the sequence:

$$\delta^{s_{t-1}} = \frac{\partial s_t}{\partial s_{t-1}} \left[ \frac{\partial \mathcal{L}_{\text{elt}}(g_t, y_t)}{\partial s_t} + \delta^{s_t} \right] . \tag{C.46}$$

Consider a case where only the output at the end of the sequence is incorrect, but it depends critically, via the weights, on the input at time 1. In this case, we will multiply the loss at step $n$ by

$$\frac{\partial s_2}{\partial s_1}\frac{\partial s_3}{\partial s_2}\cdots\frac{\partial s_n}{\partial s_{n-1}} \quad . \tag{C.47}$$

In general, this quantity will either grow or shrink exponentially with the length of the sequence, and make it very difficult to train.

**Study Question:** The last time we talked about exploding and vanishing gradients, it was to justify per-weight adaptive step sizes. Why is that not a solution to the problem this time?

An important insight that really made recurrent networks work well on long sequences is the idea of *gating*.

### C.6.1 Simple gated recurrent networks

A computer only ever updates some parts of its memory on each computation cycle. We can take this idea and use it to make our networks more able to retain state values over time and to make the gradients better-behaved. We will add a new component to our network, called a *gating network*. Let $g_t$ be a $m \times 1$ vector of values and let $W^{gx}$ and $W^{gs}$ be $m \times l$ and $m \times m$ weight matrices, respectively. We will compute $g_t$ as

It can have an offset, too, but we are omitting it for simplicity.

$$g_t = \text{sigmoid}(W^{gx}x_t + W^{gs}s_{t-1}) \tag{C.48}$$

and then change the computation of $s_t$ to be

$$s_t = (1 - g_t) * s_{t-1} + g_t * f_s(W^{sx}x_t + W^{ss}s_{t-1} + W_0^{ss}) \ , \tag{C.49}$$

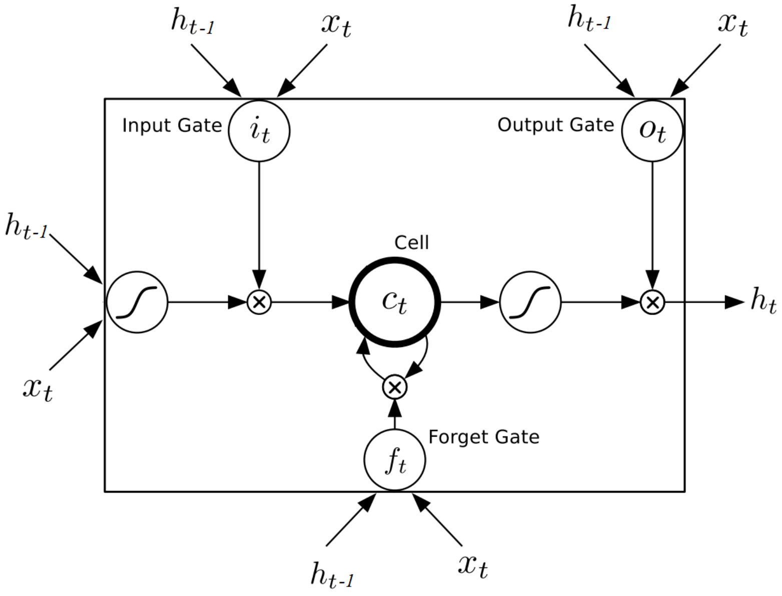
where $*$ is component-wise multiplication. We can see, here, that the output of the gating network is deciding, for each dimension of the state, how much it should be updated now. This mechanism makes it much easier for the network to learn to, for example, "store" some information in some dimension of the state, and then not change it during future state updates, or change it only under certain conditions on the input or other aspects of the state.

**Study Question:** Why is it important that the activation function for $g$ be a sigmoid?

### C.6.2 Long short-term memory

The idea of gating networks can be applied to make a state machine that is even more like a computer memory, resulting in a type of network called an LSTM for "long short-term memory." We won't go into the details here, but the basic idea is that there is a memory cell (really, our state vector) and three (!) gating networks. The *input* gate selects (using a "soft" selection as in the gated network above) which dimensions of the state will be updated with new values; the *forget* gate decides which dimensions of the state will have its old values moved toward 0, and the *output* gate decides which dimensions of the state will be used to compute the output value. These networks have been used in applications like language translation with really amazing results. A diagram of the architecture is shown below:

Yet another awesome name for a neural network!


$h_{t\text{-}1}$
$x_t$
$h_{t\text{-}1}$
$x_t$
Input Gate
$i_t$
Output Gate
$o_t$
$h_{t\text{-}1}$
Cell
$c_t$
$h_t$
$x_t$
Forget Gate
$f_t$
$h_{t\text{-}1}$
$x_t$

# APPENDIX D

## Supervised learning in a nutshell

In which we try to describe the outlines of the "lifecycle" of supervised learning, including hyperparameter tuning and evaluation of the final product.

## D.1 General case

We start with a very generic setting.

### D.1.1 Minimal problem specification

Given:

- Space of inputs $\mathcal{X}$
- Space of outputs $\mathcal{Y}$
- Space of possible *hypotheses* $\mathcal{H}$ such that each $h \in \mathcal{H}$ is a function $h : \mathcal{X} \to \mathcal{Y}$
- Loss function $\mathcal{L} : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$

a *supervised learning algorithm* $\mathcal{A}$ takes as input a *data set* of the form

$$\mathcal{D} = \left\{ \left(x^{(1)}, y^{(1)}\right), \dots, \left(x^{(n)}, y^{(n)}\right) \right\} \quad ,$$

where $x^{(i)} \in \mathcal{X}$ and $y^{(i)} \in \mathcal{Y}$ and returns an $h \in \mathcal{H}$.

### D.1.2 Evaluating a hypothesis

Given a problem specification and a set of data $\mathcal{D}$, we evaluate hypothesis $h$ according to average loss, or *error*,

$$\mathcal{E}(h, \mathcal{L}, \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{i=1}^{\mathcal{D}} \mathcal{L}(h(x^{(i)}), y^{(i)})$$

If the data used for evaluation *were not used during learning of the hypothesis* then this is a reasonable estimate of how well the hypothesis will make additional predictions on new data from the same source.

### D.1.3 Evaluating a supervised learning algorithm

A *validation strategy* $\mathcal{V}$ takes an algorithm $\mathcal{A}$, a loss function $\mathcal{L}$, and a data source $\mathcal{D}$ and produces a real number which measures how well $\mathcal{A}$ performs on data from that distribution.

#### D.1.3.1 Using a validation set

In the simplest case, we can divide $\mathcal{D}$ into two sets, $\mathcal{D}^{\text{train}}$ and $\mathcal{D}^{\text{val}}$, train on the first, and then evaluate the resulting hypothesis on the second. In that case,

$$\mathcal{V}(\mathcal{A}, \mathcal{L}, \mathcal{D}) = \mathcal{E}(\mathcal{A}(\mathcal{D}^{\text{train}}), \mathcal{L}, \mathcal{D}^{\text{val}}) \ .$$

#### D.1.3.2 Using multiple training/evaluation runs

We can't reliably evaluate an algorithm based on a single application of it to a single training and test set, because there are many aspects of the training and testing data, as well as, sometimes, randomness in the algorithm itself, that cause *variance* in the performance of the algorithm. To get a good idea of how well an algorithm performs, we need to, multiple times, train it and evaluate the resulting hypothesis, and report the average over $K$ executions of the algorithm of the error of the hypothesis it produced each time.

We divide the data into $2K$ random non-overlapping subsets: $\mathcal{D}_1^{\text{train}}, \mathcal{D}_1^{\text{val}}, \dots, \mathcal{D}_K^{\text{train}}, \mathcal{D}_K^{\text{val}}$. Then,

$$\mathcal{V}(\mathcal{A}, \mathcal{L}, \mathcal{D}) = \frac{1}{K}\sum_{k=1}^{K} \mathcal{E}(\mathcal{A}(\mathcal{D}_k^{\text{train}}), \mathcal{L}, \mathcal{D}_k^{\text{val}}) \ .$$

#### D.1.3.3 Cross validation

In *cross validation*, we do a similar computation, but allow data to be re-used in the $K$ different iterations of training and testing the algorithm (but never share training and testing data for a single iteration!). See Section 2.7.2.2 for details.

### D.1.4 Comparing supervised learning algorithms

Now, if we have two different algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$, we might be interested in knowing which one will produce hypotheses that generalize the best, using data from a particular source. We could compute $\mathcal{V}(\mathcal{A}_1, \mathcal{L}, \mathcal{D})$ and $\mathcal{V}(\mathcal{A}_\in, \mathcal{L}, \mathcal{D})$, and prefer the algorithm with lower validation error. More generally, given algorithms $\mathcal{A}_1, \dots, \mathcal{A}_M$, we would prefer

$$\mathcal{A}^* = \arg\min_m \mathcal{V}(\mathcal{A}_M, \mathcal{L}, \mathcal{D}) \ .$$

### D.1.5 Fielding a hypothesis

Now what? We have to deliver a hypothesis to our customer. We now know how to find the algorithm, $\mathcal{A}^*$, that works best for our type of data. We can apply it to all of our data to get the best hypothesis we know how to create, which would be

$$h^* = \mathcal{A}^*(\mathcal{D}) \ ,$$

and deliver this resulting hypothesis as our best product.

### D.1.6 Learning algorithms as optimizers

A majority of learning algorithms have the form of optimizing some objective involving the training data and a loss function.

Interestingly, this loss function is *not always the same* as the loss function that is used for evaluation! We will see this in logistic regression.

So for example, (assuming a perfect optimizer which doesn't, of course, exist) we might say our algorithm is to solve an optimization problem:

$$\mathcal{A}(\mathcal{D}) = \arg\min_{h \in H} \mathcal{J}(h; \mathcal{D}) \enspace .$$

Our objective often has the form

$$\mathcal{J}(h; \mathcal{D}) = \mathcal{E}(h, \mathcal{L}, \mathcal{D}) + \mathcal{R}(h) \enspace ,$$

where $\mathcal{L}$ is a loss to be minimized during training and $\mathcal{R}$ is a regularization term.

### D.1.7 Hyperparameters

Often, rather than comparing an arbitrary collection of learning algorithms, we think of our learning algorithm as having some parameters that affect the way it maps data to a hypothesis. These are not parameters of the hypothesis itself, but rather parameters of the *algorithm*. We call these *hyperparameters*. A classic example would be to use a hyperparameter $\lambda$ to govern the weight of a regularization term on an objective to be optimized:

$$\mathcal{J}(h; \mathcal{D}) = \mathcal{E}(h, \mathcal{L}, \mathcal{D}) + \lambda\mathcal{R}(h) \enspace .$$

Then we could think of our algorithm as $\mathcal{A}(\mathcal{D}; \lambda)$. Picking a good value of $\lambda$ is the same as comparing different supervised learning algorithms, which is accomplished by validating them and picking the best one!

## D.2 Concrete case: linear regression

In linear regression the problem formulation is this:

- $\mathcal{X} = \mathbb{R}^d$
- $\mathcal{Y} = \mathbb{R}$
- $\mathcal{H} = \{\theta^T x + \theta_0\}$ for values of parameters $\theta \in \mathbb{R}^d$ and $\theta_0 \in \mathbb{R}$.
- $\mathcal{L}(g, y) = (g - y)^2$

Our learning algorithm has hyperparameter $\lambda$ and can be written as:

$$\mathcal{A}(\mathcal{D}; \lambda) = \Theta^*(\lambda, \mathcal{D}) = \arg\min_{\theta, \theta_0} \frac{1}{|\mathcal{D}|} \sum_{(x,y) \in \mathcal{D}} (\theta^T x + \theta_0 - y)^2 + \lambda\|\theta\|^2 \enspace .$$

For a particular training data set and parameter $\lambda$, it finds the best hypothesis on this data, specified with parameters $\Theta = (\theta, \theta_0)$, written $\Theta^*(\lambda, \mathcal{D})$.

Picking the best value of the hyperparameter is choosing among learning algorithms. We could, most simply, optimize using a single training / validation split, so $\mathcal{D} = \mathcal{D}^{\text{train}} \cup \mathcal{D}^{\text{val}}$, and

$$\begin{aligned}
\lambda^* &= \arg\min_{\lambda} \mathcal{V}(\mathcal{A}_\lambda, \mathcal{L}, \mathcal{D}^{\text{val}}) \\
&= \arg\min_{\lambda} \mathcal{E}(\Theta^*(\lambda, \mathcal{D}^{\text{train}}), \text{mse}, \mathcal{D}^{\text{val}}) \\
&= \arg\min_{\lambda} \frac{1}{|\mathcal{D}_{\text{val}}|} \sum_{(x,y) \in \mathcal{D}_{\text{val}}} (\theta^*(\lambda, \mathcal{D}^{\text{train}})^T x + \theta_0^*(\lambda, \mathcal{D}^{\text{train}}) - y)^2
\end{aligned}$$

It would be much better to select the best $\lambda$ using multiple runs or cross-validation; that would just be a different choices of the validation procedure $\mathcal{V}$ in the top line.

Note that we don't use regularization here because we just want to measure how good the output of the algorithm is at predicting values of *new* points, and so that's what we measure. We use the regularizer during training when we don't want to focus only on optimizing predictions on the training data.

Finally! To make a predictor to ship out into the world, we would use all the data we have, $\mathcal{D}$, to train, using the best hyperparameters we know, and return

$$\begin{aligned}\Theta^* &= \mathcal{A}(\mathcal{D};\lambda^*) \\ &= \Theta^*(\lambda^*, \mathcal{D}) \\ &= \arg\min_{\theta,\theta_0} \frac{1}{|\mathcal{D}|} \sum_{(x,y)\in\mathcal{D}} (\theta^T x + \theta_0 - y)^2 + \lambda^* \|\theta\|^2\end{aligned}$$

Finally, a customer might evaluate this hypothesis on their data, which we have never seen during training or validation, as

$$\begin{aligned}\mathcal{E}^{\text{test}} &= \mathcal{E}(\Theta^*, \text{mse}, \mathcal{D}^{\text{test}}) \\ &= \frac{1}{|\mathcal{D}^{\text{test}}|} \sum_{(x,y)\in\mathcal{D}^{\text{test}}} (\theta^{*T} x + \theta_0^* - y)^2\end{aligned}$$

Here are the same ideas, written out in informal pseudocode.

```
# returns theta_best(D, lambda)
define train(D, lambda):
    return minimize(mse(theta, D) + lambda * norm(theta)**2, theta)

# returns lambda_best using very simple validation
define simple_tune(D_train, D_val, possible_lambda_vals):
    scores = [mse(train(D_train, lambda), D_val)
                  for lambda in possible_lambda_vals]
    return possible_lambda_vals[least_index[scores]]

# returns theta_best overall
define theta_best(D_train, D_val, possible_lambda_vals):
  return train(D_train + D_val,
               simple_tune(D_train, D_val, possible_lambda_vals))

# customer evaluation of the theta delivered to them
define customer_val(theta):
    return mse(theta, D_test)
```

## D.3 Concrete case: logistic regression

In binary logistic regression the problem formulation is as follows. We are writing the class labels as 1 and 0.

- $\mathcal{X} = \mathbb{R}^d$

- $\mathcal{Y} = \{+1, 0\}$
- $\mathcal{H} = \{\sigma(\theta^T x + \theta_0)\}$ for values of parameters $\theta \in \mathbb{R}^d$ and $\theta_0 \in \mathbb{R}$.
- $\mathcal{L}(g, y) = \mathcal{L}_{01}(g, h)$
- Proxy loss $\mathcal{L}_{\text{nll}}(g, y) = -(y \log(g) + (1 - y) \log(1 - g))$

Our learning algorithm has hyperparameter $\lambda$ and can be written as:

$$\mathcal{A}(\mathcal{D}; \lambda) = \Theta^*(\lambda, \mathcal{D}) = \arg\min_{\theta, \theta_0} \frac{1}{|\mathcal{D}|} \sum_{(x,y) \in \mathcal{D}} \mathcal{L}_{\text{nll}}(\sigma(\theta^T x + \theta_0), y) + \lambda \|\theta\|^2 \quad .$$

For a particular training data set and parameter $\lambda$, it finds the best hypothesis on this data, specified with parameters $\Theta = (\theta, \theta_0)$, written $\Theta^*(\lambda, \mathcal{D})$ according to the proxy loss $\mathcal{L}_{\text{nll}}$.

Picking the best value of the hyperparameter is choosing among learning algorithms based on their actual predictions. We could, most simply, optimize using a single training / validation split, so $\mathcal{D} = \mathcal{D}^{\text{train}} \cup \mathcal{D}^{\text{val}}$, and *we use the real 01 loss*:

$$\begin{aligned}
\lambda^* &= \arg\min_{\lambda} \mathcal{V}(\mathcal{A}_\lambda, \mathcal{L}_{01}, \mathcal{D}^{\text{val}}) \\
&= \arg\min_{\lambda} \mathcal{E}(\Theta^*(\lambda, \mathcal{D}^{\text{train}}), \mathcal{L}_{01}, \mathcal{D}^{\text{val}}) \\
&= \arg\min_{\lambda} \frac{1}{|\mathcal{D}_{\text{val}}|} \sum_{(x,y) \in \mathcal{D}_{\text{val}}} \mathcal{L}_{01}(\sigma(\theta^*(\lambda, \mathcal{D}^{\text{train}})^T x + \theta_0^*(\lambda, \mathcal{D}^{\text{train}})), y)
\end{aligned}$$

It would be much better to select the best $\lambda$ using multiple runs or cross-validation; that would just be a different choices of the validation procedure $\mathcal{V}$ in the top line.

Finally! To make a predictor to ship out into the world, we would use all the data we have, $\mathcal{D}$, to train, using the best hyperparameters we know, and return

$$\Theta^* = \mathcal{A}(\mathcal{D}; \lambda^*)$$

**Study Question:** What loss function is being optimized inside this algorithm?

Finally, a customer might evaluate this hypothesis on their data, which we have never seen during training or validation, as

$$\mathcal{E}^{\text{test}} = \mathcal{E}(\Theta^*, \mathcal{L}_{01}, \mathcal{D}^{\text{test}})$$

The customer just wants to buy the right stocks! So we use the real $\mathcal{L}_{01}$ here for validation.

# Index