

Explanatory Notes for 6.390

Shaanticlair Ruiz (Current TA)

Fall 2022

Contents

0 Prerequisites - Explanatory Notes	4
0.1 Multi-variable Calculus (MIT's 18.02)	4
0.2 Vectors and Matrices (MIT's 18.02)	5
0.3 Linear Algebra (18.06)	6
0.4 Programming (6.100A, 6.1010)	7
0.5 Algorithms (6.1210)	7
0.6 Probability	8
0.7 Notation: Sets	9
0.8 Notation: Numbers and functions	11
0.9 Notation: Vectors Spaces	12
0.10 Optional	13
1 Introduction - Explanatory Notes	14
1.1 Problem Class	20
1.2 Assumptions	24
1.3 Evaluation Criteria	28
1.4 Model Type	30
1.5 Model Class	34
1.6 Algorithm	38
1.7 Overview of the Course	38
1.8 Terms	39
2 Regression	41
2.1 Problem Formulation	41
2.2 Regression as an optimization problem	46
2.3 Linear Regression	50
2.4 The stupidest possible linear regression algorithm	56

2.5	Analytical solution: ordinary least squares	57
2.6	Regularization	68
2.7	Evaluating Learning Algorithms	75
2.8	Terms	85
3	Gradient Descent	87
3.1	Gradient Descent in One Dimension	92
3.2	Multiple Dimensions	102
3.3	Application to Regression	112
3.4	Stochastic Gradient Descent	117
3.5	Terms	119
4	Classification	120
4.1	Classification	120
4.2	Linear Classifiers	125
4.3	Linear Logistic Classifiers	145
4.4	Gradient Descent for Logistic Regression	160
4.5	Handling Multiple Classes	164
4.6	Prediction Accuracy and Validation	172
4.7	Terms	173
5	Feature Representation	174
5.1	Gaining intuition about feature transformations	179
5.2	Systematic feature construction	183
5.3	Hand-constructing features for real domains	193
5.4	Terms	207
6	Neural Networks 1 - Neurons, Layers, and Networks	208
6.1	Basic Element	212
6.2	Networks	224
6.3	Choices of activation function	245
6.4	Loss functions and activation functions	251
6	Neural Networks 1.5 - Back-Propagation and Training	253
6.5	Error back-propagation	253
6.6	Training	279
7	Convolutional Neural Networks - Supplementary Notes	319
7.1	Introduction	319
7.2	Filters - Introduced	322
7.3	Filters - In higher dimensions	325
7.4	Filter Banks Pt. 1	326
7.5	How to store matrices - Tensors	327
7.6	Filter Banks Pt. 2	330
7.7	Tensor Filters	332

7.8 Machine learning and Convolution	334
7.9 Output dimensions	337
7.10 Max Pooling	339
7.11 Typical architecture	341
7.12 Training our Network	342
7.13 Backpropagation in a simple CNN	348
12 Clustering	350
12.1 Clustering Formalisms	352
12.2 The k-means formulations	354
12.3 How to evaluate clustering algorithms	365
12.4 Terms	374

CHAPTER 0

Prerequisites - Explanatory Notes

This course assumes knowledge of several topics. Here, we'll outline them: hopefully, this will make it easier to get up to speed if you have a gap in your background, and to know what you're looking for.

This is designed to be somewhat comprehensive, so if you've taken a class, you can likely skip the corresponding section.

If a class has its own prerequisite, then we assume the understanding of that class as well. For example, we assume you know multi-variable calculus, so single-variable is assumed as well.

0.1 Multi-variable Calculus (MIT's 18.02)

You will need most of the differential aspects of multi-variable calculus:

- The concept of partial derivatives and how to find them
- The **multivariable** chain rule
- An intuition for the gradient as the **direction** of greatest increase in a function, and the **magnitude** of that increase.

It is helpful to able to visualize a surface created by a function. You won't need to memorize the shapes of specific surfaces; just the 3D intuition in general.

You should also be able to imagine "zooming in" to that surface, and seeing it "locally" as a **plane**, just like how we zoom in to a one-variable function, and see the **tangent line**.

Sometimes, in this class, we will also do derivatives **numerically**, where we approximate the derivative with finite steps, of the form

$$\frac{dy}{dx} \approx \frac{\Delta y}{\Delta x} \quad (1)$$

You should also be comfortable with some basic ideas of "infinity": what happens in the "limit as we approach infinity", for example.

You will not need double/triple/line integrals, curl, divergence, or greens/stokes theorem.

0.2 Vectors and Matrices (MIT's 18.02)

You will need an understanding of vectors:

- You need to know what a vector is, with two interpretations:
 - an **ordered list** of numbers
 - an object in some "space" with **magnitude** and **direction**
- You should know that the **length** or **dimension** of a vector is just how many numbers(**or elements**) that vector contains.
- You should know that a **scalar** is just a number, or in some perspectives, a 1-element vector.
- You need to be able to **add** or **subtract** pairs of vectors. You should also be able to **scale** them (in other words, multiply by a **scalar**).
- You should know how to take the **derivative** of a vector \vec{v} , with respect to a scalar x , in the form

$$\frac{d\vec{v}}{dx} \quad (2)$$

You will also need to understand the **dot product**, and its intuition as the **similarity** between vectors.

You will need to understand matrices:

- You should know what a matrix is, with three perspectives:
 - a 2D grid of numbers (in a "rectangle")
 - an **ordered list** of equal-length vectors.
 - a **transformation** of vectors.

- You should understand the **dimensions** of a matrix, and the common notation (# of rows \times # of columns)
- You need to be able to multiply two matrices, or a matrix times a vector.
 - You need to understand when you are able to multiply matrices, based on dimensions.
 - The dimensions of the new matrix after multiplication.
 - How to do the calculation of multiplying matrices by hand.

You should understand the **determinant**: both how to calculate it, and the intuition behind it.

Finally, you should know what a matrix **inverse** is, and that a **zero-determinant** matrix has **no inverse**.

0.3 Linear Algebra (18.06)

Currently, linear algebra is a **new** prerequisite.

You need all of the concepts mentioned in the "vectors and matrices" segment.

You should also understand **independence** between vectors, the **rank** of a matrix, and how to take the **transpose** of a matrix.

Linearity is a really nice (and important!) property , where a function doesn't get in the way of some simple operations: **addition** or **scalar multiplication**. The order doesn't matter.

Definition 1

For **linear** function/operator \mathbb{L} ,

Addition (of any kind) has the same effect, before or after the function.

$$\mathbb{L}(x + y) = \mathbb{L}(x) + \mathbb{L}(y)$$

Multiplication by a **scalar** also has the same effect.

$$\mathbb{L}(3z) = 3\mathbb{L}(z)$$

Example: The **derivative** is **linear**:

$$\frac{d}{dx}[f + g] = \frac{d}{dx}[f] + \frac{d}{dx}[g] \quad (3)$$

$$\frac{d}{dx}[10h] = 10 \frac{d}{dx}[h] \quad (4)$$

Often, we talk about **operators**. They're like functions, where they have an input and an output. But sometimes, we use this word when we have **another** function as an input.

Definition 2

An **operator** often takes in a **function** as an input, and gives another **function** as an output.

Example: The **derivative** is also an **operator**. If you input $f(x) = x^2$, the output is $\frac{d}{dx}[f(x)] = 2x$: another **function**.

An operator doesn't have a really "unique" definition in math: it's used for convenience.

Thus, we can call the derivative a **linear operator**.

The **visual** intuition of a matrix as a spatial transformation is useful in this class.

It would be helpful to understand **nullspace**, **column space**, and **vector spaces** in general.

A good reference for intuition is the linear algebra series by YouTube channel, 3blue1brown! Each video averages 11 minutes, and has been helpful for many past students.

0.4 Programming (6.100A, 6.1010)

For 6.100A:

You should be familiar with object-oriented programming in **Python**: you will be implementing various classes and simple algorithms in this class.

You should have a basic understanding of **time complexity** and big-O notation. Ease with reading basic pseudocode would be helpful as well.

For 6.1010:

Either 6.1010 or 6.1210 are counted as a prereq: they are not equivalent, and neither is individually required to understand the course, but either will make your work in this class much easier.

6.1010 offers more coding experience, which makes the process of implementation much smoother.

Misc:

Prior understanding of numpy is not mandatory, but would be very helpful. Pytorch would also be helpful, but is not used until much later in the course.

0.5 Algorithms (6.1210)

Either 6.1010 or 6.1210 are counted as a prereq: they are not equivalent, and neither is individually required to understand the course, but either will make your work in this class much easier.

6.1210 is about **algorithms**, and thus makes it easier to understand our discussions of different algorithms throughout this class.

Concepts of dynamic programming, complexity, and reading/writing pseudocode are all helpful for this course.

0.6 Probability

You don't need to have taken a full probability course, but there are some core concepts you must understand:

- **Probability** (or chance) is the relative **frequency** of a particular outcome, if you were to run many trials - specifically, it is the proportion of those trials that gave this particular outcome.

For example, if $p = .4$, then you should expect to have that event occur 40 times for every 100, on average.

- Probabilities p are between 0 and 1:
 - If $p = 0$, the event **will not** occur
 - If $p = 1$, the event **will definitely** occur.
 - If $p = .5$, the event has a 50% chance of happening if you try once.
 - Any other probability between 0 and 1 will give some corresponding percentile chance of occurring ($100 * p\%$)
- You can represent probability of event A as

$$P(A) \quad (5)$$

Treating P as a function that returns the **probability**.

- If you want two events to both occur, you can write that with an **and** statement:

$$P(A \text{ and } B) = P(A \cap B) \quad (6)$$

- If you want at least one of two events to occur, you can write that with an **or** statement:

$$P(A \text{ or } B) = P(A \cup B) \quad (7)$$

- The **conditional probability** is the probability of an event **given** that another event has already occurred:

$$P(B|A) \quad (8)$$

This is read as "The probability of B **given** A".

- Two events are **independent** if knowing the outcome of one event does not affect the odds of another. You can write this as

$$P(A|B)P(B) = P(A \text{ and } B) \quad (9)$$

You can equivalently use the next bullet point's definition.

- The chance of two **independent** events both happening is their odds multiplied.

$$P(A \text{ and } B) = P(A)P(B) \quad (10)$$

- The **sum** of the probabilities of all outcomes **must add** to 1: otherwise, there's a chance of getting none of the listed outcomes.
- The chance of a particular event not occurring is called the **complement**, and has a chance of $1 - p$.

0.7 Notation: Sets

There are some common definitions and notations you should be familiar with (though they may be introduced if necessary).

If you understand an equation, move on to the next one: each explanation is mostly basic.

Definition 3

An **element** is a single object in a collection of objects.

This definition is often linked to **sets**.

Definition 4

A **set** is a collection of distinct elements with no given order: if you shuffle the elements in a different order, you have the same set.

- We can **define** a set by listing out its elements. For example, this says, "The set A contains the numbers 1, 2, and 3"

$$A = \{1, 2, 3\} \quad (11)$$

- Shows that an element is **in** a set. The following says "x is an element of the set A".

$$x \in A \quad (12)$$

- Shows that an element is **not in** a set. The following says "x is **not** an element of the set A".

$$x \notin A \quad (13)$$

- Natural numbers**, or the counting numbers.

$$\mathbb{N} = \{1, 2, 3, 4, 5, \dots\} \quad (14)$$

- Real numbers**, or all of the numbers on the number line (include those between integers).

$$\mathbb{R} \quad (15)$$

- We can also define a set by starting with another set, and then listing a **restriction**:

The following says, "Include each natural number n".

$$\{n \in \mathbb{N}\} \quad (16)$$

This seems redundant, but now, we can choose to only include natural numbers **larger than 10**:

$$\{n \in \mathbb{N} \mid n > 10\} = \{11, 12, 13, 14, \dots\} \quad (17)$$

- A set A is a **subset** of B if all elements in A are contained in B. B is then said to be a **superset** of A.

For example, the set of natural numbers is a **subset** of the set of real numbers: every natural number is also a real number.

Here, this says " \mathbb{N} is a subset of \mathbb{R} ".

$$\mathbb{N} \subseteq \mathbb{R} \quad (18)$$

Notice that this symbol looks similar to \leq : that's intentional! This is because the two sets could be the same set, while one is a subset of the other.

- The previous symbol allowed for the two sets to be the same. But if we know they aren't, we can use a **proper subset**.

Here, this says " \mathbb{N} is a **proper** subset of \mathbb{R} ".

$$\mathbb{N} \subset \mathbb{R} \quad (19)$$

Since we know that some real numbers are not natural numbers, they can't be the same.

0.8 Notation: Numbers and functions

- **Sum notation:** adding up elements in a sequence. For example:

$$\sum_{n=1}^6 n^2 = 1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 6^2 \quad (20)$$

- **Product notation:** multiplying elements in a sequence. For example:

$$\prod_{n=1}^5 n = 1 \times 2 \times 3 \times 4 \times 5 \quad (21)$$

- **Rounding up:** round up real numbers. The following says, "round 2.5 up to the nearest whole number"

$$\lceil 2.5 \rceil = 3 \quad (22)$$

- **Rounding down:** round down real numbers. The following says, "round 2.5 down to the nearest whole number"

$$\lfloor 2.5 \rfloor = 2 \quad (23)$$

- **Function** notation: shows the name of the function, the set of inputs, and the set of outputs.

For example, this below says, "the function f takes real numbers as inputs, and outputs natural numbers."

$$f : \mathbb{R} \longrightarrow \mathbb{N} \quad (24)$$

- If you want to get the **maximum** or **minimum** output of a function, you use the function with the corresponding name: max or min.

For example:

$$\min_{x \in \mathbb{R}} x^2 = 0 \quad (25)$$

$$\max_{x \in \mathbb{R}} \sin x = 1 \quad (26)$$

Below the max or min declaration you can denote the domain over which to find the maximum or minimum, respectively.

- Sometimes, you don't want the minimum or maximum output: you want to know the **input** that gives you the minimum or maximum output. If the domain can be inferred from context, it may be omitted.

So, you pick an **argument** (input variable) and get the **argmax** or **argmin**

The following says, "x = 1 **gives you** the minimum output for $f(x) = (x - 1)^2$ ".

$$\arg \min_x (x - 1)^2 = 1 \quad (27)$$

The following says, "f(x) = 0 **is** the minimum output for $f(x) = (x - 1)^2$ ".

$$\min_x (x - 1)^2 = 0 \quad (28)$$

Make sure to keep track of the **difference** between min and argmin, or max and argmax!

0.9 Notation: Vectors Spaces

Here, we'll build up some notation for representing sets of vectors, by representing them as ordered sequences.

- Often, we care about **ordered sequence** of numbers. Maybe you want to return the entire sequence.

We start with ordered pairs of numbers: you can represent every pair of elements from two sets with \times .

For example, here we have "every pair of two natural numbers":

$$\mathbb{N} \times \mathbb{N} = \{(1, 1), (1, 2), (2, 1), (2, 2)\dots\} \quad (29)$$

Notice that this can be used to fill in an grid of numbers: with real numbers, you can fill in the whole space with no gaps. Note that since sets do not contain duplicate elements, (1, 1) is not included twice.

- If you want **more than two** elements, you can simply use more **crosses** \times .

For example, here is every trio of natural numbers:

$$\mathbb{N} \times \mathbb{N} \times \mathbb{N} = \{(1, 1, 1), (1, 1, 2), (1, 2, 1)\dots\} \quad (30)$$

- Here, we introduce a shorthand, because writing every cross (example: $\mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R}$) can get tiring.

We can compress multiplication with **exponents**, so we'll do the same here:

$$\mathbb{R}^5 = \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \times \mathbb{R} \quad (31)$$

- Because one of our perspectives on **vectors** is as "an ordered list of numbers", we can represent all of our desired vectors using this notation.

In general, the set of all length-n vectors can be represented as

$$\mathbb{R}^n \quad (32)$$

0.10 Optional

Here, we list concepts that could be **helpful** for understanding this course more easily, but are entirely **not required**, as we'll be teaching what we need.

- Tensors
- Convolution
- Examples of Optimization (Least-Squares, etc.)
- Markov Models
- Probability and Statistics (Expectation, Variance, Distributions...)
- Mathematical maturity (from upper-level math courses, etc.)
- Matrix Calculus

CHAPTER 1

Introduction - Explanatory Notes

These are the explanatory course notes produced by **Shauntclair Ruiz**, a current TA who has also served as an LA in Spring 2022. They are intended to be **supplementary** to the official lectures notes.

The official course lecture notes are designed to be **minimal**, and present what the instructors think that you absolutely **need** to **know** to understand and interact with the current state of machine learning.

These notes, by contrast, are designed to provide more thorough **explanations**. We **explain** certain logical leaps, **break down** concepts into smaller parts, and try to make the notes more **accessible** to students who find the primary notes too dense.

These notes cover the **same** topics as the primary notes, just with a different **presentation**. Most of the explanations in this document are a reaction to **difficulties** that students have had in previous semesters.

If the concepts in these explanatory note chapters are **familiar** to you, or if you find them to be too **drawn-out**, you can **skim** sections that you're not concerned about.

We, again, stress that neither set of notes is more "advanced", as they cover the **same material**. They simply reflect **different** learning styles and backgrounds.

It may be helpful to refer to these explanatory notes as you digest the official lecture notes: the main section numbers (1.1, 1.2, 1.3...) should **match** with the official notes.

If you have any concerns or points of **confusion**, feel free provide **feedback** on this ongoing project.

1.0.1 What is machine learning?

Why study machine learning? To answer that, let's learn what machine learning really *is*. Fortunately, it's all in the name.

Machine learning is a broad field. We use **machines**, or computers, and give them data to **learn** from.

Why are we teaching machines? Same as why we want **people** to learn: so they can use that learning to make good **decisions**.

So, in short, we can say:

Concept 5

The main focus of **machine learning** is making **decisions** or **predictions** based on **data**.

1.0.2 Why do Machine Learning: The Benefits

Why use machine learning? What is it good for?

The techniques used in machine learning have many applications. It has become the best way to handle many different problems:

- Facial detection
- Speech recognition
- Language processing
- Many problems that involve data or signal processing

Based on speed, time to develop, "robustness", etc.

Different ML techniques have become the best way to handle many problems in many fields. As a result, it has become very popular!

1.0.3 The role of humans

If machines can solve all of these problems, where do humans play a role?

Well, these machines aren't (yet) able to **set themselves up** to solve these problems: humans have to set up the system so the machines can succeed. We call this **framing** the problem.

We'll use an example to help explain.

- A human has to **recognize** that there is a problem to solve.
 - **Example:** You want to have self-driving cars. Your problem: those cars need to be able to **watch** the road.

- They have to decide what kind of **solutions** you want to try, and use that as the basis for training.
 - **Example:** You decide to create a **model** that can replicate vision for our car.
 - This **model** represents the kinds of **solutions** you expect to work: a particular model will allow for a certain approach to a situation.
- They have to **gather** data to train with.
 - **Example:** You might gather **videos** from dashcam footage, or create a virtual simulator for your car to drive in.
- They have to choose the **algorithms** we'll use for learning: what **instructions** do we give our computer?
 - **Example:** To "train" your model, you could need to adjust it to perform **better**. How do you adjust it, using the videos?
- They have to look at the final result and **validate** whether it's a good enough solution to use.
 - **Example:** You **test** out your model in a car: does it notice obstacles?
- They have to consider the possible **ethics** or other consequences of this solution.
 - **Example:** What's the most "responsible" way of driving? When should a car prioritize its own safety, or the safety of pedestrians? How much control should the user have?

This is over-simplified, but it gives us a high-level view of what we'll need going forward.

These are all important steps, and they require the human in question to make smart and responsible choices. That's why you need to learn machine learning: in order to use it **effectively**, you have to **understand** it!

We want to understand machine learning, so we'll break it down into different parts. We'll do this by **asking** ourselves a couple **questions**, and thinking about machines in the broadest sense we can: as the **solution** to a **problem**.

This breakdown is different from the one above!

1.0.4 What's the plan?

We know that, in machine learning, we want to make **decisions** or **predictions** using **data**.

Let's frame this more generally: we want to **solve** a **problem** presented to us, using our **machine** and some **data**.

This brings up some questions:

- What exactly is our **problem**?
- And **solution** do we want to use?

The answer depends on the situation, but we can break down these questions into simpler, easier ones.

1.0.5 The Problem

Simply put, our goal is to create a **machine** that **takes in** data and **spits out** some kind of results.

In that way, our machine is just a **function**.

The **problem**, then, is to reach that goal: to get our desired output from our input.

That means we're focused on what's **outside** of our machine - here, we don't know or care how the machine works, we just know what we've got (input), and what we want (output).

- **Assumptions:** What do we **assume** about our **problem**? What do we expect about our **data**, or our possible **solutions**? How do we use this knowledge?

– This step is important because these assumptions can allow us to **simplify** the problem, and often, our approach **depends** on them.

We often use these assumptions to come up with solutions: if they aren't true, your approach may fail!

– **Example:** We might be looking at our patients (several adorable puppies), and **assume** that they are all the same **age**: we can simplify by not including age as a variable.

- **Problem Class:** What are the **needs** of our particular problem? What **kind** of inputs and outputs are expected?

– In this situation, "class" means, "set of things with something in **common**". So, our "problem class" tells us, "what **kind** of problem do we have?"

"Which **group** of problems does ours **fit into**?"

– This is important for choosing our solution: our solution follows from the problem.

In order to answer a question, you need to know what you're being asked!

- * We might also use **existing** solutions to similar **problems** as inspiration for our own work.
 - **Example:** Our inputs are weight, blood pressure, and breed. Our output is a number: how long do they have to live? This will be a real number, in years.
- **Evaluation Criteria:** What is our goal? We know the **kind** of output we want (structure, type, etc.), but how do we measure the **quality** of an answer?
 - This evaluation criteria is crucial, both for telling our machine how to **improve**, and to **show** other humans how well it **performs**.

Example: We could use the absolute difference between the lifetime predicted, and the lifetime the puppies actually experience.

These aspects together make up our problem, that we now need a **solution** for.

1.0.6 Solution Setup: What is a model?

Remember: our goal is to create a **machine** that **takes in** data and **spits out** some kind of results.

The **solution** is what's **inside** the machine - how do we do it? What approach do we use?

First, let's dig a little into what a **solution** is: we've mentioned before that our solution will often rely on a **model**, but what exactly *is* a model?

For our purposes, a model is a way to **simplify reality**: we strip away everything that doesn't matter, and just leave a system that can work *well enough*, in the ways that matter.

In machine learning, we sometimes care less about how **realistic** the model is, than its ability to get **good results**. That means our model is not always structured to match reality.

Definition 6

A **model** is a way of mathematically **representing** a **system**.

This system is **simplified** to only include the **details** we care about and give us the level of **accuracy** we want.

We do this sometimes because we don't *know* the true model, and sometimes because simulating the true model is too expensive and time-consuming.

We boil down a **system** into the values we **care about**, and how those values **affect** each other (in terms of math equations).

Example: A planetary model that simulates **gravity** between Mars and the sun may not account for the density of the planet, or everything that happens on the surface... but that might be good enough to predict the **length of a year** on Mars.

However, in this example, we knew all of the values of the model (the weight of the planet and sun, the distance from the sun...). We have no need for machine learning: the model is already **complete**.

Again, we emphasize that a model doesn't have to be structured to match reality - but if we know the true model, this can help.

In the problems we face, we **don't know** those values, or even always what **model** will work best. That's where the techniques we will learn come in.

1.0.7 The Solution

So now, we have a vague idea of what our solution might look like. So, let's break it into parts, like we did for the problem.

- **Model Type:** Will we make a model? What kinds of **data** will we **include** in our model?
 - Sometimes, a model isn't necessary: do we really need it? If we do, how do we **use** that model?
- **Model Class:** What **kind** of model will we use? What sort of **variables** will we use, and what **structure** will our math use?
 - Just like with problem classes, a model class is a "type" of model: a collection of models with similar structure.
 - We will spend much of this class exploring **different** model classes: each has benefits in different circumstances.
- **Algorithm:** Once we have a model, how do we "teach" it what we want it to know? We'll need a **procedure** for this - an algorithm.
 - Which algorithms we choose will affect how well our machine can learn: how quickly will it learn, and how good is the end result?

Now, we take a deeper dive into each aspect listed about, starting with our **base assumptions**.

1.1 Problem Class

"Problem class" is, from the name, the type of problem you are presented with: what inputs and outputs are expected?

But there is a second aspect of the problem we haven't discussed - what **data** is does the machine have available when it is **training**?

1.1.1 Supervised vs. Unsupervised

We know that, to train our computer, we have to give it **input** data, but how does the machine know whether it's doing well? We could, for example, give it an "**answer key**": the correct outputs we expect from it.

If we do, we call that **supervised learning**.

Or, do we find a different way to measure its success? We break it down into a few common cases:

In a way, we're "supervising" it by giving it the right answer: we're guiding it and making sure it does what we want it to!

- **Supervised Learning** is when we train our machine using a set of **inputs** and the correct matching **outputs**.

– **Example:** You show your machine a bunch of **pictures** (inputs), and then **label** what is in each picture: like a dog (output).

- **Unsupervised Learning** is when we **don't** give our machine the answers, and it has to guess without having a "correct" answer.

– This is often used in cases where we don't know a "correct" answer in advance. For example, we might want to find some kind of **pattern** in our data, and we have no way of predicting that!

– **Example:** You look at a bunch of animals (input) and try to invent species for groups of animals.

- There are other cases that we will save for the end of this section.

You don't need to know the species "cow" and "pig" to figure out that they're different from each other!

We'll list some common problem classes for each of these types. You don't have to memorize these types, but they will come back later in the course.

But first, one more detail.

1.1.2 How do we store our data?

Each data point typically contains **multiple** values: pieces of information you want to draw **conclusions** from.

We often standardize the information our machine receives by storing this information in a **vector**.

Being consistent makes it easier to develop the techniques we need!

Our models are usually made up of **equations**, so we want to be able to **compute** with these values. So, each variable will be represented with a real number, or multiple if necessary.

Thus, one data point is a **vector of real numbers**. Specifically, a **column vector**.

Notation 7

x is our **vector of inputs**.

It is a column vector. Its matrix shape is $(d \times 1)$.

Example: Suppose we have a data point x that's a vector of 3 numbers: its shape is (3×1) .

We write this as:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad (1.1)$$

Since this is so common, we introduce some notation.

The real numbers are represented with \mathbb{R} . Since we're combining **multiple** real numbers, we use **exponent** notation to represent this.

Notation 8

The **set of length-d vectors** is written as \mathbb{R}^d .

Example: \mathbb{R}^2 represents all of the length-2 vectors: all of the vectors/points on the 2D plane.

So, we might say a data point $x \in \mathbb{R}^d$ if all we know is that x is a length-d vector.

1.1.3 Supervised Learning

- **Regression** takes in a vector of numbers, and predicts some **real number** as an output. Our goal is to correctly guess the desired output.

– **Example:** You want to predict how much a worker makes based on their job, where they live, and how many years they've worked.

- **Classification** also takes in a vector of numbers, but outputs a **label**: we have a set of **classes**, and we want to **label** each data point as a member of one class.

– This means our output is **discrete**: each class is separate output value, and we have k classes.

– **Example:** You have several documents and want to **label** which **language** each is written in.

Notice that "where they live" isn't usually represented as a number: we often have to convert certain data types.

As before, a "class" is just another word for a group of related things.

1.1.4 Unsupervised Learning

- **Density Estimation** takes in data, and tries to approximate the **distribution** of that data: what is the chance of getting a new data point x ? _____

– **Example:** You want to get the **distribution** of human **heights** in a particular city.

We define "distributions" in section 1.2.

- **Clustering** is when you want to sort data points into groups of similar points, without knowing the groups in advance.

– **Example:** You want to sort patients with a disease into groups, where each group might need different treatments.

- **Dimensionality Reduction** is a bit different: the goal is to take a vector, and reduce the length of the vector, while still keeping the information that's important.

– You may not need every dimension to store the information you need, so you can save on space and time by storing it in a smaller vector.

– **Example:** You find out height has no effect on income, so you ignore height. Or maybe you find that having both education and literacy is redundant. _____

– Notice that what information is relevant depends on what you're using it for.

Since we often automate this process, real examples might not be so simple!

1.1.5 Other Types of Learning

Now, we turn to some types of learning that are, arguably, neither supervised nor unsupervised.

- **Reinforcement Learning** is used when you have an "environment" you can interact with. Different choices will change what that environment looks like, and may reward or punish you. _____

– The goal is to pick the actions that give you the best rewards.

We represent the current environment with something called a **state**.

– **Example:** You have a robot on Mars, and you want to move your robot (action) to reach certain goals (rewards)

– This isn't **supervised** because you **don't know** the correct action. But it isn't fully unsupervised because you **do know** when you get a reward.

- **Sequence Learning** is used to take one sequence, and turn it into another. In these sequences, each output depends on all of the previous inputs.

– This means you need to store information about previous inputs using a **state**.

– **Example:** Predicting the next word in a sentence, based on the words so far. You **predict** one word for each new one you receive, so you return a **sequence**.

- We're partly "supervised" by being given the output sequence, but we don't know what our states need to look like.

1.1.6 Types of Learning not covered in this class (Optional)

These will not be covered, but are worth mentioning.

- **Semi-supervised Learning** gives us some supervised training data that has been labelled, but also some that has not.
- **Active Learning** gives our computer the ability to **choose** which data points it receives: this is used when data is **expensive**, and we want to learn efficiently.
- **Transfer Learning** is used when we apply learning from one task to another, related task. That way, the new task can be learned faster.

1.2 Assumptions

Let's look at our underlying assumptions: the rest of this class relies on these assumptions.

1.2.1 An assumption about data

Let's return back to our original goal: we want to use **data** to teach our machine to give us **results** we want. Just like how a person might learn from their **experience** and use it to make **judgments**.

However, there's an **assumption** built in to this statement, one we need to look at more closely: we are assuming that **past** data allows us to predict **future** data.

This may seem obvious, but it isn't always: past data may not be **representative** of the future, for example.

- **Example:** We can't use the weather over the month of July to predict the weather in the month of December.

This often called the problem of **induction**: using the past to predict the future.

1.2.2 Is our data representative?

First, let's solve the problem presented above:

- **Example:** We got our weather from a **different** month than we're trying to predict.

So, it seems our problem is that our **data** and what we're trying to **predict** are from **two different sources**.

We want them to come from the **same source**, then. In this case, we could say we want them to be from the **same** month. Great. But how do we say this in general?

1.2.3 How do we compare data?

We got down to the real problem: we want our new data to be from a similar source to the old data. One month couldn't **represent** another, because they **behave** differently.

- **Example:** For different months, we get different rainy days, different temperature ranges, so on: they can't be compared.

In general, we need a way to describe what we mean by "different": what describes one of these months?

- **Example:** To us, all that matters is the weather: how **likely** are we have a rainy day, for example? In fact, we'd like to know how **likely** every outcome is.

We represent this with something called a **distribution**. A distribution gives us exactly what we just described: **how likely** different events are to occur. _____

This is how our system "behaves", in a way.

Definition 9

A **distribution** is a **function** that gives us the **probability** of different **outcomes**.

Example: The **distribution** of outcomes on a coin is 50% chance of heads, 50% chance of tails.

Notice that distributions are **probabilistic**: outcomes have a certain **chance** of occurring. Otherwise, these problems would be simple.

Why is it called a distribution? Well, we're taking the **odds**, and spreading them out (or **distributing** them) over multiple different outcomes!

1.2.4 Identically Distributed Data

We can think of this distribution as a **simplified** view of the **source** of our data. Each "outcome" is a data point; one we can use to **learn**.

We want our **past** data we **learn** from, and our **future** data with **test** with, to have the **same** distribution.

We also want different points in the **same** dataset (past *or* future) to be from the same **distribution**: if they aren't, then why are we lumping them together?

We want them to be the "same", or **identical**: they have **exactly** the same chances for each outcome.

We want to focus on one problem at a time - one distribution.

In other words: we want our sets of data to be **identically distributed**.

Definition 10

If two **data points**(or datasets) are **identically distributed**, then they have the **same** underlying **distributions**.

In other words, they have the **same probabilities** for each possible **outcome**.

Example: Two fair coins will behave the same as each other: they both have 50-50 odds. Thus, they're **identically distributed**.

1.2.5 Independence (Review)

There's a second assumption that is just as important: when we draw two different data points, we are also **assuming** that the results of one do not **affect** the other.

If one point **depended** on another, then there's no **new** information: you could have used the last point to guess this one.

This means you're **not learning**, which is a problem: you need many experiences to come to a good **conclusion**, that will apply well in the future.

Because we don't want the result of one data point to **depend** on another, we call this assumption **independence**.

Definition 11

Two **data points** are **independent** if **knowledge** of the outcome for one data point does not affect the **probabilities** for the other.

Example: If you flip two coins, knowing that one coin comes up heads does not tell you anything about the other coin: the two coin tosses are **independent**.

This definition is a bit informal: the proper definition is to say that, for two events A and B, $P(A)P(B) = P(A \text{ and } B)$

1.2.6 Independent and Identically Distributed

We combine both of these assumptions into our final result: we want our data points and data sets to be both **independent** and **identically distributed**.

Definition 12

IID, or **Independent and Identically Distributed**, means that if you draw two data points, they

- Come from the **same distribution**: they have the same **probabilities** for each outcome,
- They **aren't related** in any other way: they are **independent**, meaning the **outcome** of one **does not** affect the other.

Example: Based on the two examples above, flipping two coins (or rolling a die twice) is IID.

We shorten this to one acronym, which tells you how important it is: it is the base assumption in many different statistics, inference, and machine learning settings.

We will assume this to be true, and use that assumption throughout the class. We expect our data to be IID in most cases.

1.2.7 Estimation and Generalization

In this section, the main theme has been applying knowledge about **training** data to **new**, unfamiliar situations, like our **testing** data.

We have a word for this that we haven't used so far: **generalization**.

Definition 13

Generalization is the **problem** of applying **current** knowledge to **new** situations we've never seen before.

We want to be able to take the **specific** case of our training data, and apply it to the more **general** case of any of the possible **new** data.

A second problem is the **nature** of our training data: because we **randomly** select it, we don't have a perfect idea of what the true distribution looks like.

The randomness means that our sample will look a bit different each time we generate it.

This creates some **noise**: something that interferes with what we're trying to focus on.

The problem of using our sample to **estimate** the true distribution, despite imperfect, "noisy" data, is **estimation**.

Just like how background **noise** can make it harder to listen to a phone call!

Definition 14

Estimation is the **problem** of taking **imperfect** data and using it to **estimate** the "true" information we're looking for.

1.2.8 Other Assumptions

There are some other assumptions we will make, that will not go into as much detail on:

- We know the set of possible answers: the type of answer we should give back, whether number, label, making a choice...
 - If we don't know what kind of answer we're supposed to give, how can we build a model to give back that answer?
- Our problem is solvable: the "true" model can be represented and answered using our computer.

Imagine if you were supposed to write an essay, but could only answer with real numbers between 0 and 1 - this is what we want to avoid.

Here are some more which are less universal.

- The data is generated by a Markov chain.
- The data might be **adversarial**: designed to specifically exploit weaknesses in the machine.

If you don't know what this is, don't worry! We come back to it later.

Some of these assumptions are required in order to move forward at all. Others narrow down the options we have to work with, so we can find a good solution in a reasonable amount of time.

1.3 Evaluation Criteria

1.3.1 What is a loss function?

In order to solve our task, we want to be able to measure how our machine is performing. We do this by creating a measure of success or failure, called a **loss function**.

Definition 15

A **loss function** measures how **poorly** your machine is **performing** on a **task**.

The output is a **real number**. If your machine is performing **well**, then you will have a **low** output. And vice versa: if it is doing **badly**, it will have a **high** output.

Example: If you counted the number of questions you got **wrong** on a test, that could be a **loss function**.

A loss function usually has the **correct** and **predicted** guesses as inputs: it has to compare them to know how well it's doing.

Notation 16

Often, we will use g to represent our **guess** as to the correct answer: this is the output of our model; our **prediction**.

The **true answer** is often represented by either a or y .

Our **loss** is the function \mathcal{L} , so altogether, our computed loss is $\mathcal{L}(g, a)$.

1.3.2 Examples of Loss Functions

Different loss functions are useful for different situations.

- **0-1 Loss** is a simple kind of loss: if our answer is correct, the value is 0. If our answer is incorrect, the value is 1.

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

This matches our earlier example of "number of questions wrong on a test".

- This kind of loss is often used for **discrete** situations, where there are k options and one is correct - like on a multiple-choice test.
- **Linear loss** is the **absolute difference** between your answer and the correct one.

$$\mathcal{L}(g, a) = |g - a| \tag{1.2}$$

- **Square loss** is the **square difference**.

$$\mathcal{L}(g, a) = (g - a)^2 \quad (1.3)$$

- Because the slope increases as you get further away from 0, it punishes large errors more aggressively than small errors.
- **Asymmetric Loss** punishes some outcomes more than others. It may be worse to miss a heart attack, than to expect one and be wrong.

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

1.3.3 How to use loss

We want to reduce loss as much as we can: in other words, **minimize** it. But there are lots of ways to do that.

In this class we will minimize the **expected loss**: the average loss we would *expect* based on the probability of each outcome.

We could the "worst-case" loss, or the average loss, etc...

We do this because, over the **long-term**, the **expected loss** should reflect what we actually get.

This is called the **law of large numbers**: if you have a large number of trials, the average value should be close to the expected value.

Concept 17

In most machine learning problems, we want to **minimize** our **expected loss**.

But we also need to be careful when choosing our loss function: if we get to choose how we're grading ourselves, then we need to pick an accurate way to measure progress!

1.4 Model Type

Now, we start on the solution. Do we choose to use a model? If we do, there are some other details we have to consider.

1.4.1 No Model

A model allows us to **simplify** what we learn from our data. So, if we don't use a model, we have to use our data **directly**.

One way to do this is to simply average some known data points that "seem" similar to the newest query. This is called the **nearest neighbor** approach. _____

Example: You measure a chemical's physical properties, and **label** it based on which one you've seen before is the most **similar**.

When we say "similar", there are multiple ways to interpret this, but often we use distance in \mathbb{R}^d space.

1.4.2 Models using Parameters

These days, we're much more likely to **use** a model to make our prediction.

But, as we mentioned before, a model can be adjusted, and for almost any problem, we'll need to adjust it to fit our needs. This is the process of **training**.

How do we adjust a model? Our models will be a **function**, that has several values it uses to do calculations on our inputs. For example, here's a simple model:

$$f(x) = A \sin(Bx) + C \quad (1.4)$$

In this case, we have one input variable, x . And we have three values that don't change based on the input: A , B , and C . These values are called **parameters**.

Definition 18

Parameters are the **non-input variables** in a model that can be **adjusted** to adjust the model.

Example: When using the linear equation $f(x) = mx + b$, m and b are your parameters.

You can think of a parameter as a dial on a machine that you can "tune" to different values, like a radio.

Adjusting these parameters will change how the model behaves - different outputs for each input - but it keeps the same overall **structure**.

By structure, we mean the formula: the way variables and parameters **interact**. The above model will (almost) always be **different** from _____

"Almost" because, if $A = B = 0$ for both cases, they're both the constant function $f(x) = C$.

$$f(x) = Ax^2 + Bx + C \quad (1.5)$$

They both have three parameters, and one input, but they are different models.

1.4.3 Prediction Rule

Our goal is to use one of these equations to **directly** calculate our prediction. For that reason, we call this equation our **prediction rule**, but more often, we will call it our **hypothesis**.

Definition 19

A **hypothesis** is the **function** that defines our model, using a fixed number of **parameters**.

The **output** of our hypothesis is typically the **prediction** our model is designed to create.

1.4.4 Hypothesis Notation

For simplicity's sake, we often lump all of our **parameters** into a single **vector**, θ . Just like for x , we'll use a **column vector**.

Notation 20

θ is our **vector of parameters**.

It is a column vector. Its matrix shape is $(d \times 1)$.

Example: Here is a vector θ with 4 parameters.

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \end{bmatrix} \quad (1.6)$$

Often, a vector is represented by bolding a variable (x), or putting an arrow over it (\vec{x}). Since we work with vectors so often in this class, we will omit this notation.

Similarly, we lump all of our inputs into a single **vector**, x .

But, if we have **multiple** data points, we need to label them **separately**.

Notation 21

$x^{(i)}$ is the i^{th} **data point**, represented as a vector.

Sometimes, you may instead see the notation x_i .

x is the **input** to our hypothesis h , but since θ (our parameters) can be **adjusted**, we can think of it as a **second** "type" of input.

To represent this, we use $f(a; b)$ notation: a is our input to a single **function**, but b allows us to describe a whole **family** of functions (by adjusting parameters).

Notation 22

Our **hypothesis** is shown in the form $h(x; \theta)$.

1.4.5 Fitting

The process of **adjusting** our model (i.e. its **parameters**) to match our data is called **fitting**.

As we mentioned before, our goal is typically to **minimize** expected loss. But this expected loss is based on knowing the **true** distribution of our data. We call this loss our **test error**.

Since we usually don't know the true distribution, we have to settle for our best guess - the **training data** that we've gathered.

We call it this because we're "testing" our machine in the real world.

Instead, we could minimize the **training** error: we average it out, to see our performance. Let's write that out.

The loss for our i^{th} data point is $\mathcal{L}(g^{(i)}, a^{(i)})$. So, we average out n of those points:

$$\frac{1}{n} \sum_{i=1}^n \mathcal{L}(g^{(i)}, a^{(i)}) \quad (1.7)$$

Let's write this in terms of x and y . a is just another name for y .

Our guess is given by the hypothesis, so $g^{(i)} = h(x^{(i)}; \theta)$.

In this equation, we'll leave off θ , to allow for non-parametric hypotheses.

Key Equation 23

The **expected loss** for a hypothesis is:

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

This is the equation we would **minimize** with θ .

1.4.6 Overfitting

But, we have to be careful - we mentioned before that the randomness of our sampling can introduce **noise**.

If we too heavily emphasize the current values, we may not **generalize** well to new data. This problem is called **overfitting**, and we will talk about it a lot in this course.

Definition 24

Overfitting happens when we fit **too strongly** to a particular dataset.

Because we focus too much on that **dataset**, our machine **learns** incorrect facts about the overall distribution.

This makes our model worse at **generalizing** to new situations.

Example: You want to know what cats are like. By coincidence, you see three black cats in a row. You assume all cats are probably black: you've **overfit** to your data.

In this course, we will discuss many ways to tackle **overfitting**.

1.5 Model Class

In this section, we'll assume we're using a model.

1.5.1 Hypothesis Class

As we mentioned before, changing our **parameters** will change the specific model we have, but it will have the same overall **structure**.

Models with the same overall equation are put in the same **model class**. Since our models are defined by their **hypothesis**, we will more often talk about **hypothesis class**.

Definition 25

A **hypothesis class** is a collection of **hypotheses** with the **same type of equation**: the only difference between them is the **value** of their **parameters**.

Another way to view it is that the **hypothesis class** represents all of the **possibilities** for a model class: we can get every option based on our **parameters**.

Example: Every hypothesis of the form $mx + b$ is in the same **hypothesis class**.

Another way to say "same type of equation" is "same functional form".

1.5.2 Expressiveness

Note that some hypothesis classes are capable of things that others are **not**. For example, our linear function $mx + b$ could never produce a **parabola** x^2 .

That means if our problem **requires** a more complicated model, then we can't ever get a good result!

This can be summarized by **expressiveness** or "richness" of a hypothesis class.

Definition 26

If one **hypothesis class** is more **expressive** than another, it can represent a **larger** collection of possible hypotheses.

Sometimes, if a problem can't be solved in one model class, it might be solvable in a more **expressive** one.

Example: **Quadratic** equations ($Ax^2 + Bx + C$) are more expressive than **linear** equations ($mx + b$). Every linear equation can be **created** using quadratics, but not the other way around.

1.5.3 Choosing Model Classes

So, the question is - which model class should you use for a given problem?

Your first instinct might be to use the most **expressive** one you can. However, this can become very **expensive** to compute, because there are many more options you have to explore.

Often, it is already **known** what kinds of models work well for what kinds of problems - we'll explore some of those options in this class.

It's also more likely to overfit! We'll discuss why another time.

As an ML researcher gains more **experience**, they can use that experience to make **educated** guesses: they may look at multiple possible models, and pick one based on theory or practice.

Choosing the **class** of model we want is called the **model selection** problem. Choosing the **parameters** for our model, on the other hand, is **model fitting**.

Research on this is ongoing: we continue to develop new model classes to try to better handle new and old problems!

1.5.4 Our Linear Model

We will start this class off using one of the simplest models we know: one that only uses **addition** and **scalar multiplication**.

We have our input variables, $x_1, x_2, x_3 \dots$ that we can combine using these two operations. We can add them together, add a constant, or multiply by a constant.

We can write this in general as:

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots + \theta_d x_d \quad (1.8)$$

Where θ_i are our parameters.

1.5.5 Linear Model: Vector Form

θ and x , both being vectors, are being multiplied in a way that looks similar to the **dot product**: multiplying together elements, and then adding.

$$h(x) = \theta_0 + \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \quad (1.9)$$

So, we can rewrite it more compactly this way:

$$h(x) = \theta_0 + \theta \cdot x \quad (1.10)$$

Note that this looks very similar to the $y = mx + b$ formula, our original linear function!

Note that, in order for the dot product to work, x and θ must have the same **shape**.

Concept 27

When using a linear model, x and θ must have the **same shape**. They both have length d .

Meaning, they are both $(d \times 1)$ column vectors.

1.5.6 Linear Model: Cleaning Up

Unfortunately, we had to leave θ_0 out to make it work: if we want to talk about **all** parameters, we'll instead use the symbol Θ .

Notation 28

We represent the **parameters** of our **linear** equation as $\Theta = (\theta, \theta_0)$

We'll swap out the dot product for matrix multiplication, because this allows us to use matrices instead of just vectors (which we'll need later!)

$$h(x) = \theta_0 + \begin{bmatrix} \theta_1 & \theta_2 & \dots & \theta_d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \quad (1.11)$$

Finally, we condense our vectors into symbols.

In order to make the matrix multiplication work, we have to take the **transpose** θ^T .

Key Equation 29

The **linear model** has a hypothesis of the form

$$h(x) = \theta^T x + \theta_0 \quad (1.12)$$

This is the form you will use through a significant portion of this course - it's good to get used to it!

1.5.7 Other Models

We will explore several different kinds of models in this course.

In general, we will assume that we have a fixed, finite number of parameters. Models that don't have this restriction are called **non-parametric** models. We will use them in only one chapter.

Instead, we will focus on our **linear** model, and functions we can apply to allow it to handle non-linear problems.

We will combine many of these "non-linear units" to create **neural networks** later on - arguably the most **powerful** tool in the ML arsenal, and key to machine learning's modern explosion in usage.

Many of the modern models used in complex and high-performing system are **variations** on neural networks, so we will give them all the attention they need.

1.6 Algorithm

Finally, once we have our model class and a tool for evaluating our model, we can finally begin the process of **fitting** our model.

This is where the problem of developing an **algorithm** comes in - we need to decide on what set of instructions will best help us find a good model.

Our problem will typically boil down to a kind of optimization: minimizing a loss function, or more often, a modified loss function called an **objective function**.

Different problems will require different algorithms and techniques: some are general-purpose optimizers, others are specially tailors for the needs of machine learning.

One of our most powerful tools will be **gradient descent**; so much so that it has its own devoted chapter.

But, we will leave that to the next chapters.

1.7 Overview of the Course

Here is a short summary of each chapter.

- **Introduction:** an introduction to the basic concepts of the course, and what to expect going forward.
- **Regression:** using our linear model to learn to make numeric predictions about future data.
- **Gradient Descent:** learning to use the gradient, our "multivariable derivative", to optimize functions, like loss.
- **Classification:** using our model to sort data into different classes, and introducing some non-linear functions into that model.
- **Feature Representation:** transforming the data we receive, both to make them usable by a computer, and expanding our hypotheses to non-linear functions.
- **Neural Networks:** showing how you can combine multiple non-linear functions, to create a much more powerful function for new, exciting problems.
- **Convolutional Neural Networks:** building on neural networks with convolution, making it easier to handle images, signals, and other problems.
- **Sequential Models:** introducing "states", a way to store information over time, and how to do decision-making using that information.
- **Recurrent Neural Networks:** We combine neural networks with states to build up a sequence of outputs over time, allowing us to do some language processing.

- **Reinforcement Learning:** making decisions in a changing environment, where some states and choices reward you more than other.
- **Non-parametric methods:** introducing some different tools, which are often cheaper to develop and sometimes just as effective as more complex methods.
- **Clustering:** trying to find hidden patterns and structures in data, and making that data easier to visualize for human usage.

1.8 Terms

- Machine Learning
- Problem Class
- Model
- Model Class
- Distribution
- Identically Distributed
- Independence
- IID
- Induction
- Generalization
- Estimation
- Supervised Learning
- Unsupervised Learning
- Regression
- Classification
- Loss Function
- Expected Loss
- Parameter
- Non-Parametric Model
- Hypothesis
- Fitting

- Overfitting
- Hypothesis Class
- Expressiveness
- Linear Model

CHAPTER 2

Regression

2.1 Problem Formulation

In the last chapter, we discussed many broad ideas in machine learning.

In this section, we will **review** those concepts, and use one concrete example to help make them clearer: **regression**.

2.1.1 Hypothesis (Review)

In the last chapter, we distinguished between problem and solution. Our **problem** is getting from our input x to our desired output y .

$$x \rightarrow \boxed{?} \rightarrow y$$

Meanwhile, our **solution** is some model we place in between those two: some **function** we can use to compute output based on input. This is our **hypothesis** h .

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

Remember that we store our **parameters** in a vector Θ : this vector defines our **model** within our **model class**.

This Θ is what we hope to be able to improve, and use to find a **better model**.

Depending on how **precise** we want to be, remember that we can write our hypothesis as $h(x)$ or $h(x; \Theta)$: both are **valid**, the latter emphasizes the fact that Θ is a **variable**.

We'll focus on Θ more if we assume we know which **hypothesis class** we're in - if we know that, Θ fully **defines** our model.

Often, people will treat Θ and h as almost interchangeable: be careful when you're doing this!

2.1.2 The Problem of Regression

In regression, our problem is taking data in a vector, and converting it into a **real number**.

This is the mission of our function, the **hypothesis**. Functions are important, so we'll introduce some common notation.

Remember the notation for real-numbered vectors we introduced in the last chapter!

Notation 30

A **function** is notated based on what sorts of **inputs** it can take, and the **outputs** it can return.

A function f is written like this:

$$f : \text{set of inputs} \rightarrow \text{set of outputs}$$

Often, instead of the "set of inputs", you'll hear people talk about the **input space**. This is essentially the same thing: it is a term worth getting used to.

Now, we can write this more efficiently:

Technically, a **space** is a set "with **added structure**", which is about as broad as it sounds.

Definition 31

Regression is a **machine learning setting** where we use a **vector** of real numbers to return a **real-valued number**.

In other words, we want a **hypothesis** h of the form:

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

Example: If you have **3 values** in your input vector (height, weight, age) and **1 real output** (life expectancy), you would need a hypothesis

$$h : \mathbb{R}^3 \rightarrow \mathbb{R}$$

A more visual example:



In this example, you have one input (x-axis), and you want to **predict** the output (y-axis) based on that. These points are the dataset you want to **learn** to match.

2.1.3 Converting our data

Often, our data will not be in the right format - maybe we have a car brand, or a color as a variable.

This requires converting this data into real numbers. We do this using something called a **feature transformation**.

Definition 32

A **feature** is one distinct piece of **information** in our input.

A **feature transformation** takes those pieces of information, and **transforms** them into something more **useful** - often a better data type or format.

Sometimes, we use it on already-valid formats to find **new patterns** in data.

Example: You have three car brands. Instead of representing them normally, you instead turn them into vectors:

$$\text{Brand A} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad \text{Brand B} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad \text{Brand C} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad (2.1)$$

We do our feature transformation with a function: we often notate this function as $\varphi(x)$.

There are many different feature transformations for different needs. We will come back to this in a later chapter.

This particular feature transformation is called **one-hot encoding!** We'll return to it later.

For now, we will simply assume that all of our inputs x are already in \mathbb{R}^d (vectors of real numbers).

2.1.4 Our dataset

Regression is **supervised**, meaning we have training data with a correct output included: we have an "answer key" to a practice quiz.

We want to pair up inputs with their correct outputs, so we'll write our first data point as

$$(x^{(1)}, y^{(1)})$$

And so we have a set of n data points, \mathcal{D}_n :

Remember that the superscript tells us that this is the 1st data point.

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

2.1.5 Measuring our performance

In the last chapter, we discussed that we use our past **training data** to learn a good **model** for our future **testing data**.

So, even though the training data is what we have in front of us, the **testing data** is what we care the most about. We want our machine to handle new situations.

We'll assume our data are IID, so that we can **learn** and **generalize** effectively. And we'll evaluate ourselves using a **loss function** \mathcal{L} , a measure of how poorly our model is running.

We care about **expected loss**, so we'll take an **average**. We'll start with our training data, so we call it **training error**:

Key Equation 33

Training Error is written as:

$$\mathcal{E}_n(h) = \frac{1}{n} \sum_{i=1}^n \text{Loss}^{(i)} = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(h(x^{(i)}), y^{(i)}) \quad (2.2)$$

Notice that we treat this error as a function of h , our hypothesis - our **hypothesis** is what we're **adjusting** to try to improve the error.

Clarification 34

h is a **function** that takes a **variable**, x , as its input. This is the usual format.

\mathcal{E}_n is also a **function**, but it takes in h , a **different function**, as an input!

That means one function is using another function as an input. This can sometimes cause confusion.

Make sure to keep track of the difference as you go through this chapter - the idea will come back later.

2.1.6 Learning to Generalize

Our goal, though, is to perform well on testing data: to be able to **generalize** to data we haven't seen before.

So, let's define **test error**. This time, we have m new data points.

$$\mathcal{E}(h) = \frac{1}{m} \sum_i \mathcal{L}(h(x^{(i)}), y^{(i)}) \quad (2.3)$$

We'll start counting from $n+1$ because we've already used the first n points when training.

Key Equation 35

Testing Error is written as:

$$\mathcal{E}(h) = \frac{1}{m} \sum_{i=n+1}^{n+m} \mathcal{L}(h(x^{(i)}), y^{(i)}) \quad (2.4)$$

We want to minimize **test error**, but by definition of "data we haven't seen before", we can't use it to design our hypothesis.

So, for now, the next best thing to "minimize test error" is "**minimize training error**", while doing our best to help our hypothesis **generalize**. We'll discuss how to do that.

2.2 Regression as an optimization problem

We've mentioned that we want to make our loss (error) as low as we can: we want to **minimize** it. This is a form of **optimization** - getting the best results from our system. Here, we'll introduce some of the terms and notation of optimization.

Lot of research has gone into solving optimization problems!

2.2.1 Objective Function

Our goal is to minimize our loss. But, **training loss** is not our main goal: to compensate for that, we might include other terms to make it more **general**.

To distinguish between our **loss** versus our overall goal, we will define an **objective function**.

In later sections, we will introduce something called a **regularizer** to do just that!

Definition 36

An **objective function** is the function we are **optimizing** for: usually, this means that our goal is **minimizing** it.

This term usually contains the **loss**, and sometimes, a **regularizer**.

We minimize our objective function using our **parameters** Θ , so we take that as an input: $J(\Theta)$

We will be seeing a lot of Θ for a while.

2.2.2 Parts of an Objective Function

For now, we will just consider loss, so our **objective function** will be the same as our **training error**, dependent only on the **loss function**.

Since we are focusing more on Θ , we'll replace $h(x^{(i)})$ with $h(x^{(i)}; \Theta)$:

$$J(\Theta) = \text{Training Error} = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(h(x^{(i)}; \Theta), y^{(i)})$$

But once we introduce the **regularizer** $R(\Theta)$ to improve **generalization**, we'll add that term:

$$J(\Theta) = \text{Training Error} + \text{Regularizer}$$

We will discuss this term in a later section; don't worry about it for now.

We'll also include a scaling factor λ so we can control this term:

Key Equation 37

In general, we write the **objective function** as:

$$J(\Theta) = \left(\frac{1}{n} \sum_{i=1}^n \mathcal{L}(h(x^{(i)}; \Theta), y^{(i)}) \right) + \lambda R(\Theta)$$

In the long run, this is the function we want to **optimize**.

Notice that our objective function depends on our training data \mathcal{D} as well: what data we have affects what the best result is.

Just like how we can use $h(x; \Theta)$, we'll use the same notation here: $J(\Theta; \mathcal{D})$

Θ is our "main" input variable, but if we switch out our data \mathcal{D} , we would get a different result.

Clarification 38

Students often get confused by the fact that our **objective function** J is a function of Θ , while **training error** \mathcal{E}_n is a function of h .

This may seem strange: if **training error** is included in the **objective function**, why have different variables?

The difference is that **training error** is **more general** than the **objective function**, and h is **more general** than Θ .

- **Training error** can be used for a **non-parametric model**. Since we don't know if we have parameters, we can't assume Θ , our vector of **parameters**. So, we use h .
- Our **objective function assumes** we have parameters Θ : since we want to optimize with Θ , we use it instead.

2.2.3 Minimization Notation

Our goal is to minimize J by adjusting Θ . If we accomplish this, there are two questions we can ask ourselves, and some corresponding notation.

To show our point, we'll use the following example:

Example: Take $f(x) = (x - 1)^2$. The minimum output is 0, which happens at $x = 1$. So, we have a minimum at $(1, 0)$.

- What is the **minimum** value of J we can find **by adjusting** Θ ?

Notation 39

The **min function** gives you the **minimum output** of a function we get by adjusting one chosen **variable**.

$$\min_{\Theta} J(\Theta)$$

The **function we want to minimize** is written to the right, while the **variable we adjust** is written below.

Example:

$$\min_x (x - 1)^2 = 0 \quad (2.5)$$

- What **value** of Θ gives us **minimum** J ?

Notation 40

The **argmin function** tells you the value of the **input variable** that gives the **minimum output**.

$$\arg \min_{\Theta} J(\Theta)$$

The **function we want to minimize** is written to the right, while the **variable we adjust** is written below.

Example:

$$\arg \min_x (x - 1)^2 = 1 \quad (2.6)$$

Why is it called "argmin"? Well, "argument" is used as another word for "input variable". And our argmin function returns the **argument** with the **minimum** output.

2.2.4 Optimal Value Notation

So, we want to know what the best model we want get is, where this model is represented by Θ .

Notation 41

We add a **star** * to indicate the **optimal** variable choice.

If that variable is z^* , you would say it as "z-star".

Example:

$$x^* = 1 \text{ for the above example.} \quad (2.7)$$

So, if we want optimal Θ , we're looking for:

Key Equation 42

Our **optimal parameter** vector is written as

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

2.3 Linear Regression

Now that we understand the problem of **regression**, and the concept of **optimizing** over it, we can pick a concrete example.

We want a function that can use information to **predict** outputs.

2.3.1 The Linear Model, 1-D

We'll start off small: we have one variable, and something we want to predict. And we'll pick the simplest pattern we can:

$$y = mx + b \quad (2.8)$$

A linear equation: m tells us how much our input affects our output. b accounts for everything unrelated to x : what is y when $x = 0$?

b and m are our parameters: that means they're part of Θ . We'll rename them θ_0 and θ_1 .

$$h(x) = \theta_1 x + \theta_0 \quad (2.9)$$

2.3.2 The Linear Model, 2-D

We want to have **multiple** input variables: x will be a **vector**, not a number. So, for our above example, we'll **replace** x with x_1 .

$$h(x) = \theta_1 x_1 + \theta_0 \quad (2.10)$$

The simplest way to include x_2 by just **adding** it. We have a scaling factor θ_1 for x_1 , so we'll give x_2 its own **parameter**, θ_2 :

If θ_1 is the "slope" for x_1 , θ_2 is the "slope" for x_2 .

$$h(x) = \theta_2 x_2 + \theta_1 x_1 + \theta_0 \quad (2.11)$$

2.3.3 The Linear Model, d-D

You can **expand** this to d dimensions by simply adding more terms:

This is the "dimension" of our input space: the **number** of input variables we have.

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots + \theta_d x_d \quad (2.12)$$

2.3.4 The Linear Model using Vectors

Here, we are **multiplying** components of x and θ together, then **adding**. This looks like a **dot product**:

$$h(x) = \theta_0 + \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \quad (2.13)$$

If we write this symbolically, we get:

$$h(x) = \theta_0 + \theta \cdot x \quad (2.14)$$

Unfortunately, we had to leave θ_0 out to make it work. θ is used for the parameters of our **dot product**, Θ is **all** parameters.

Notation 43

We represent the **parameters** of our **linear** equation as $\Theta = (\theta, \theta_0)$.

This formula looks similar to $y = mx + b$ again! Only this time, we have **vectors** instead.

We'll swap out the dot product for **matrix multiplication**: we'll use matrix multiplication a lot in this chapter, and course.

One benefit is that we can use **matrices** instead of just **vectors**!

Key Equation 44

The **linear regression** hypothesis is written as

$$h(x) = \theta^T x + \theta_0$$

Remember that, when written out, this looks like:

Make sure you know what θ^T is: it's the transpose!

$$h(x) = \begin{bmatrix} \theta_1 & \theta_2 & \theta_3 & \dots & \theta_d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_d \end{bmatrix} + \theta_0 \quad (2.15)$$

This is the **hypothesis class** of **linear hypotheses** we will reuse throughout the class.

2.3.5 Regression Loss

We need to decide on our **loss function** for regression: how **badly** is our model performing?

We have an **actual** output value a that we want to compare to our **guessed** output g . If they're more **different**, that's worth punishing **more**.

$g - a$ is our difference, but we want to punish it being both too **high** or too **low**: we'll **square the difference**.

$$\mathcal{L}(g, a) = (g - a)^2 \quad (2.16)$$

Or, if we use $y = a$:

$$\mathcal{L}(g, y) = (g - y)^2 \quad (2.17)$$

We call this **square loss**. It punishes high and low guesses equally, and the punishments become more **severe** as the **difference** increases.

We use **square** distance for a few good reasons:

Our slope $2x$ gets larger as we move away from $x = 0$.

- Always positive: high and low guesses are treated equally.
- When we have vectors, we can **represent** it with a **dot product** $w \cdot w$, or **matrix multiplication** $w^T w$: tools we like using.
- $\|w\|$ is not **smooth**: this isn't good for derivatives! $\|w\|^2$ is smooth.

To see this, compare $|x|$ to x^2 .

2.3.6 Our Goal

Our goal is to minimize the **loss** \mathcal{L} on our data set, using the **linear** model.

We want the smallest distance between our **hypothesis** and the **data points**: we want it as **close** as possible.

Let's write this into a **single equation**: $J(\Theta) = J(\theta, \theta_0)$

$$J(\theta, \theta_0) = \text{Training Loss} \quad (2.18)$$

Get the equation for **expected loss**, using ⁽ⁱ⁾ to show which data point we're using in the sum:

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(g_i, y^{(i)}) \quad (2.19)$$

Substitute in **squared loss**:

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n (g_i - y^{(i)})^2 \quad (2.20)$$

Our guess is given by our **hypothesis**, $h(x^{(i)}; \Theta)$

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n \left(h(x^{(i)}; \Theta) - y^{(i)} \right)^2 \quad (2.21)$$

And finally, we use our linear model, $\theta^T x^{(i)} + \theta_0$.

Key Equation 45

The **ordinary least squares objective function** for **linear regression** is written as

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n \left((\theta^T x^{(i)} + \theta_0) - y^{(i)} \right)^2 \quad (2.22)$$

To clarify:

$$J(\theta, \theta_0) = \underbrace{\frac{1}{n} \sum_{i=1}^n}_{\text{Averaging}} \left(\underbrace{(\theta^T x^{(i)} + \theta_0)}_{\text{guess}} - \underbrace{y^{(i)}}_{\text{answer}} \right)^2 \quad (2.23)$$

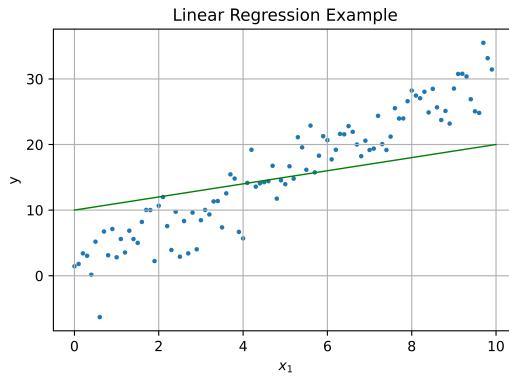
From this, we want to find

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

We now have two parameters in our argmin function, but aside from listing both of them, the notation is the same. We just substituted $\Theta = (\theta, \theta_0)$

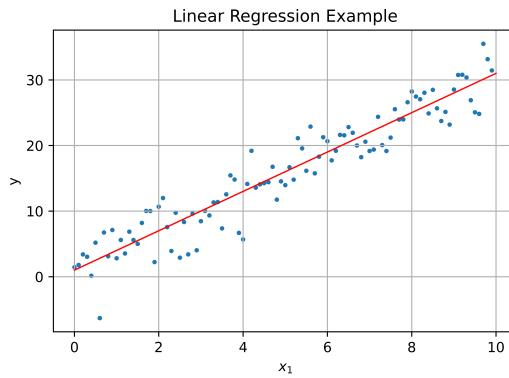
2.3.7 Visualizing our Model

With **one variable**, we've seen that our linear model simply turns into $\theta_1 x_1 + \theta_0$. As you'd expect, on a plot, this looks like a **line** in the **2D plane**.



This example of linear regression is not a great fit: $(\theta_0 = 10, \theta_1 = 1)$

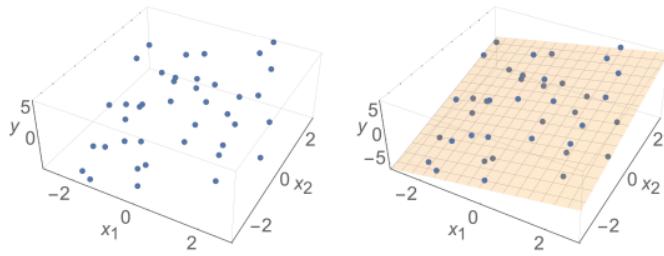
We're trying to get our line as **close as possible** to the points, hoping to find a linear pattern. We're **fitting** our line to the data.



This line is much better fitted to the data: $(\theta_0 = 1, \theta_1 = 3)$

What does this look like if we have **two** variables? You need a 3D space, with 2 dimensions for the input.

Extending our line into a second dimension, we create a **plane**.



This plane is **fitted** the same way our line was. Notice that y is our **height**: this is the **output** of our regression.

Higher-dimension versions are hard to visualize. So, instead, we don't try, and call it a **hyperplane**.

Definition 46

A **hyperplane** is a **higher-dimensional version** of a **plane** - a **flat** surface that continues on forever.

We use it to represent our **linear** hypothesis for the purpose of **regression**.

The "**height**" ($(d+1)^{\text{th}}$ dimension) of this **plane** at a certain point represents the **output** of our **linear** hypothesis at that point.

Our line was a **1-D** object in a **2-D** plane. Our plane was a **2-D** object in a **3-D** space. So, our hyperplane is a d dimensional object in a $d + 1$ dimensional space.

With this intuition, we can imagine our **hyperplane** as trying to get as **close** to all of the data points as it possibly can.

2.3.8 Another Interpretation

There's another, similar way to interpret our model

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots + \theta_d x_d \quad (2.24)$$

Before, we took θ_k as just an **extension** of the $mx + b$ formula: θ_k tells us how much x_k affects our output.

However, we can also think about the **relative** scale of each θ_k : if θ_2 is **larger** than θ_1 , then x_2 has a **stronger** effect on the output than x_1 .

We can say that x_2 **weighs** more heavily in our calculation: it has more say in the **result**.

Because of that, we sometimes call θ_k the **weight** for x_k .

Definition 47

A **weight** is a **parameter** that tells us how **strongly** a variable **influences** our **output**.

It is usually a **scalar** that we **multiply** by our variable.

2.4 The stupidest possible linear regression algorithm

So, now we want to try to optimize J based on θ and θ_0 . How do we do that? Let's start as simple as we possibly can.

We can't try **every** possible Θ , because there are an **infinite** number of them. Rather than thinking too hard about a possible pattern, or an **algorithm**, let's just **randomly** try options.

We'll try **random** values for θ and θ_0 , and **pick** whichever option gives us the best result. Seems simple, if inefficient.

Why introduce such a silly algorithm? For two reasons:

- It gives us an **example** of an optimization algorithm that's very **simple**.
- **Randomly** generated results create a good **baseline** - more intelligent algorithms can be compared to this one, to see how well we're doing.

2.5 Analytical solution: ordinary least squares

We can do better than randomly **generate** parameters, though. In fact, in this rare case, we can actually **solve** for optimal parameters!

2.5.1 Trying to Simplify

Our approach will involve a lot of **algebra**. Because of that, it's worth it to **simplify** our formula as much as possible beforehand.

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n \left(\underbrace{(\theta^T x^{(i)} + \theta_0)}_{\text{guess}} - \underbrace{y^{(i)}}_{\text{answer}} \right)^2 \quad (2.25)$$

Most parts of this equation can't really be **simplified**: y and x are just variables, and we can't do anything with the **sum** without knowing our data points.

But, one thing that was strange is that we **separated** θ_0 from our other θ_k terms. Maybe we can **fix** that.

2.5.2 Combining θ and θ_0

Let's go back to our **original** equation for $(\theta^T x + \theta_0)$, before we switched to **vectors**.

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots + \theta_d x_d \quad (2.26)$$

We converted this into a **dot product** because each θ_n term is **multiplied** by an x_k term, except θ_0 .

We drop the $^{(i)}$ notation whenever it isn't necessary, to de-clutter the equations. We only do this when we don't care which data point we're using.

But if we **really** want to include θ_0 , then could we? We know what's missing: " θ_0 is **not** multiplied by an x_k term". So... could we get one? Is there a x_0 factor we could **find**?

We need θ_0 to be **multiplied** by something. Is there something we could "factor out"? How about: $x_0 = 1$?

You can always factor out 1 without changing the value!

$$h(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots + \theta_d x_d \quad (2.27)$$

So, this means we just have to **append** a 1 to our vector x . At the **same time**, we'll append θ_0 to θ !

$$x = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_d \end{bmatrix}, \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_d \end{bmatrix}, \quad h(x) = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_d \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_d \end{bmatrix} \quad (2.28)$$

We'll write that symbolically, and then apply a transpose.

$$h(x) = \theta \cdot x = \theta^T x \quad (2.29)$$

Concept 48

Sometimes, to simplify our algebra, we can **append** θ_0 to θ .

In order to do this, we have to **append** a value of 1 to x as well.

Once we do this, we can **write**

$$h(x) = \theta^T x$$

We **have** to append this 1 to every single $x^{(i)}$ in order for this to **work**. But, now we can treat our parameters as **one vector**.

2.5.3 Summing over data points

Currently, when using our **objective** function, we have to **sum** over **every** single data point. For the 1D case, this means we have to do:

$$J = \frac{1}{n} \sum_{i=1}^n (\theta x^{(i)} - y^{(i)})^2 \quad (2.30)$$

This is a bit of a hassle - it **forces** us to use $x^{(i)}$ notation, and we have to be conscious of that **sum**.

By using **vectors** above, we were able to work with **many** variables θ_k at the same time, making it easier to **represent** and **work** with them in the future.

Can we do the **same** here - combining many **data points** into one object, rather than many **variables**?

2.5.4 Summing with Vectors: Row Vectors

We want to represent **addition** using **vectors**. We did that when we were adding $x_k \theta_k$ terms with a **dot product**.

$$\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_d \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_d \end{bmatrix} \quad (2.31)$$

But, dot products also include **multiplication**. Above, our terms are **squared**. So, we can multiply $(\theta x^{(i)} - y^{(i)})$ times itself!

$$J = \frac{1}{n} \sum_{i=1}^n (\theta x^{(i)} - y^{(i)}) (\theta x^{(i)} - y^{(i)}) \quad (2.32)$$

We'll write $r^{(i)} = \theta x^{(i)} - y^{(i)}$ to simplify our work.

$$J = \frac{1}{n} \sum_{i=1}^n r^{(i)} * r^{(i)} \quad (2.33)$$

In a dot product, we **add** the **dimensions** together. So, we'll give each term in our sum its own **dimension**.

$$J = \frac{1}{n} \sum_{i=1}^n r^{(i)} * r^{(i)} = \frac{1}{n} \begin{bmatrix} r^{(1)} \\ r^{(2)} \\ r^{(3)} \\ \vdots \\ r^{(n)} \end{bmatrix} \cdot \begin{bmatrix} r^{(1)} \\ r^{(2)} \\ r^{(3)} \\ \vdots \\ r^{(n)} \end{bmatrix} \quad (2.34)$$

We've got a single vector we could call R .

We could make it a **column vector**, but we already use the **rows** to indicate the **dimensions**.

$$\theta = \left[\begin{array}{c} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_d \end{array} \right] \left. \right\} \text{dimensions as rows...} \quad (2.35)$$

So, let's use **columns** instead: each **column** will be a **data point**: we'll use a **row vector** ($1 \times n$).

$$R = \overbrace{\begin{bmatrix} r^{(1)} & r^{(2)} & r^{(3)} & \dots & r^{(n)} \end{bmatrix}}^{\text{data points as columns!}} \quad (2.36)$$

2.5.5 Going from x to X

We can do the same for our input data $x^{(i)}$:

Notation 49

We can store all of our 1-D **data points** in a **row vector**:

$$\begin{aligned} X &= \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \dots & x^{(n)} \end{bmatrix} \\ Y &= \begin{bmatrix} y^{(1)} & y^{(2)} & y^{(3)} & \dots & y^{(n)} \end{bmatrix} \end{aligned}$$

We can write our **objective function** as

$$J = \frac{1}{n} \begin{bmatrix} r^{(1)} & r^{(2)} & r^{(3)} & \dots & r^{(n)} \end{bmatrix} \begin{bmatrix} r^{(1)} \\ r^{(2)} \\ r^{(3)} \\ \vdots \\ r^{(n)} \end{bmatrix} \quad (2.37)$$

Or more compactly:

$$J = \frac{1}{n} RR^T \quad (2.38)$$

Since we had $r^{(i)} = (\theta x^{(i)} - y^{(i)})$, we can write

$$R = \theta X - Y \quad (2.39)$$

Still in the 1D case!

Let's use this to expand our objective function:

Concept 50

In 1-D, we can use row vectors to sum our data points as

$$J = \frac{1}{n} (\theta X - Y)(\theta X - Y)^T$$

We've successfully **removed the sum!**

This format **stores** all of our **data points** in **one object**, just like how we wanted.

2.5.6 Putting it together: Matrices

Now, we have shown both a way to express x_1, x_2, x_3 as a single ($d \times 1$) matrix:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_d \end{bmatrix} \quad (2.40)$$

And a way to express $x^{(1)}, x^{(2)}, x^{(3)}$ as a single $(1 \times n)$ matrix:

We'll leave off the appended 1 for now.

$$\mathbf{X} = \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \dots & x^{(n)} \end{bmatrix} \quad (2.41)$$

Why not combine them into a single object?

Key Equation 51

\mathbf{X} is our **input matrix** in the shape $(d \times n)$ that contains information about both **dimension** and **data points**.

$$\mathbf{X} = \underbrace{\begin{bmatrix} x_1^{(1)} & \dots & x_1^{(n)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \dots & x_d^{(n)} \end{bmatrix}}_{\text{n data points}}}_{\text{d dimensions}} \quad (2.42)$$

If we include the appended 1, we write this as the $((d + 1) \times n)$ matrix

$$\mathbf{X} = \underbrace{\begin{bmatrix} 1 & \dots & 1 \\ x_1^{(1)} & \dots & x_1^{(n)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \dots & x_d^{(n)} \end{bmatrix}}_{\text{n data points}}}_{\text{d + 1 dimensions}} \quad (2.43)$$

Because each data point $y^{(i)}$ has only one dimension, it's the same as in the last section:

Key Equation 52

\mathbf{Y} is our **output matrix** in the shape $(1 \times n)$ that contains all data points.

$$\mathbf{Y} = \begin{bmatrix} y^{(1)} & \dots & y^{(n)} \end{bmatrix}$$

All we have to do is combine our **equations**: We can use the one in the last section, but because θ is a matrix, we have to **transpose** it.

There was no point in transposing it when it was just a constant!

Key Equation 53

Using our **appended** matrix, we can write our **objective function** for **multiple** variables and **multiple** data points as

$$J = \frac{1}{n} (\theta^T X - Y) (\theta^T X - Y)^T$$

It is important to **remember** the **shape** of our objects, as well.

Concept 54

Our matrices have the shapes:

- X : $(d \times n)$ - matrix
- Y : $(1 \times n)$ - row vector
- θ : $(d \times 1)$ - column vector
- θ_0 : (1×1) - scalar
- J : (1×1) - scalar

If we combine θ_0 into θ , replace every use of d with $d + 1$.

These shapes are worth **memorizing**.

2.5.7 Alterate Notation

One side problem: some ML texts use the **transpose** of X and Y .

Notice that these shapes make sense for our above equation! Try working through the matrix multiplication to verify this.

Notation 55

Some subjects use **different notation** for **matrices**. The main difference is that X and Y use their **transpose**, which we'll notate as

$$\tilde{X} = X^T \quad \tilde{Y} = Y^T$$

Thus, our equation above becomes

$$J = \frac{1}{n} (\tilde{X}\theta - \tilde{Y})^T (\tilde{X}\theta - \tilde{Y})$$

2.5.8 Optimization in 1-D - Calculus Returns!

Now that we have our **problem** presented the way we **want**, we can figure out how to **optimize** our θ .

For now, we'll revert to **sum** notation, but we'll come back to our **matrices** later.

$$J(\theta) = \frac{1}{n} \sum_{i=1}^n (\theta^T x^{(i)} - y^{(i)})^2 \quad (2.44)$$

How do we **optimize** this? Let's just take **one data point**:

$$J(\theta) = (\theta^T x - y)^2 \quad (2.45)$$

And we'll start in 1D.

$$J(\theta) = (\theta x - y)^2 \quad (2.46)$$

If we treat θ like any ordinary **variable**, this is just a simple function! How would we find the **minimum**?

Using **calculus**! Anywhere there's a local **minimum**, we typically know the **derivative** is 0.

Assuming a "smooth" surface...

Note that we aren't taking $\frac{d}{dx}$: we want to change our **model**, not our **data**! So, since θ represents our **model**, we'll take $\frac{d}{d\theta}$.

$$J'(\theta) = 2x(\theta x - y) = 0 \quad (2.47)$$

We just find where the slope is 0, and solve for θ !

$$\theta^* = \frac{y}{x} \quad (2.48)$$

We technically need to prove whether this is minimum, maximum, or neither. For now, we'll assume we have a minimum.

Concept 56

If our function $J(\theta)$ has **one variable**, we can **explicitly** find **local minima** by solving for θ when the **derivative** is zero.

$$\frac{dJ}{d\theta} = 0$$

Then, you check each candidate using the second derivative to see if it is a minimum ($J''(\theta) > 0$). In this class we will often be able to ignore this step.

This concept is review from 18.01 (Single-variable calculus), but is worth repeating!

2.5.9 Using our sum

Now, we'll go back to having multiple data points we want to **average**:

$$J(\theta) = \frac{1}{n} \sum_{i=1}^n (\theta x^{(i)} - y^{(i)})^2 \quad (2.49)$$

We want to do the same optimization. Thankfully, derivatives are **linear**: addition and scalar multiplication are not affected!

Check the prerequisites chapter, chapter 0, for a full definition of linearity.

$$J'(\theta) = \frac{1}{n} \sum_{i=1}^n 2x^{(i)} \cdot (\theta x^{(i)} - y^{(i)}) = 0 \quad (2.50)$$

And we can **solve** this just the same way.

2.5.10 Optimizing for multiple variables

Now, the tricky part is working with **vectors**.

We'll ignore the averaging and ⁽ⁱ⁾ notation since that's easy to add on afterwards.

$$J(\theta) = (\theta^T x - y)^2 \quad (2.51)$$

We want to **optimize** this. In the **one-dimensional** case, we wanted to set the **derivative** of J to **zero**, using a single θ variable. Now, we have **multiple** variables θ_k to **change**.

Derivatives are all about **change** in variables, and our **change** $\Delta\theta$ is a **combination** of changing the different **components**, $\Delta\theta_k$.

$$\Delta\theta = \begin{bmatrix} \Delta\theta_0 \\ \Delta\theta_1 \\ \Delta\theta_2 \\ \vdots \\ \Delta\theta_d \end{bmatrix} \quad (2.52)$$

So, maybe it would be reasonable to just set **every** derivative to **zero**? It turns out, the answer is **yes**!

We can show this by using the **chain rule** definition:

$$\underbrace{\Delta\theta \frac{dJ}{d\theta}}_{\text{The change in } J \text{ from } \theta \text{ overall}} \approx \underbrace{\Delta\theta_0 \frac{\partial J}{\partial \theta_0} + \Delta\theta_1 \frac{\partial J}{\partial \theta_1} + \cdots + \Delta\theta_d \frac{\partial J}{\partial \theta_d}}_{\text{The change in } J \text{ from each } \theta_k \text{ term}} \quad (2.53)$$

So, if all the derivatives are zero, the **overall** derivative is zero.

This approximation formula becomes exact as the step size shrinks: we go from $\Delta\theta$ to $d\theta$.

Concept 57

If our function $J(\theta)$ has d different variables, we can explicitly find local minima by getting all of the equations

$$\frac{\partial J}{\partial \theta_0} = 0, \quad \frac{\partial J}{\partial \theta_1} = 0, \quad \frac{\partial J}{\partial \theta_2} = 0, \dots, \quad \frac{\partial J}{\partial \theta_d} = 0$$

Or in general,

$$\frac{\partial J}{\partial \theta_k} = 0 \quad \text{for all } k \in \{0, 1, 2, \dots, d\}$$

...And solving this system of equations for the components of θ .

The solution to this system of equations will be our desired result, θ^* .

Again, we ignore the second requirement of making sure this isn't a maximum or saddle point.

2.5.11 Gradient Notation

Writing it this way can be a hassle. So, we'll continue our tradition of using matrix-based notation to make our lives easier.

You may recognize these component-wise derivatives as part of the "multivariable version" of the derivative: the gradient.

Key Equation 58

The gradient of J with respect to θ is

$$\nabla_{\theta} J = \frac{\partial J}{\partial \theta} = \begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \vdots \\ \frac{\partial J}{\partial \theta_d} \end{bmatrix} \quad (2.54)$$

For example, our previous approach boiled down to saying

$$\nabla_{\theta} J = \begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \vdots \\ \frac{\partial J}{\partial \theta_d} \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} = 0 \quad (2.55)$$

Note the subscript on the gradient! This emphasizes that our space is the components of θ , not the components of our data x .

For our purposes, we will simply represent that zero vector with a single 0.

Concept 59

If our function $J(\theta)$ has a **vector variable**, we can **explicitly** find **local minima** by solving for θ when the **gradient** is 0.

$$\nabla_{\theta} J = 0$$

This is the form we will be solving.

Ignoring the requirement from earlier!
We're assuming it's a minimum.

2.5.12 Matrix Calculus

Taking derivatives of vectors falls under **vector calculus**. That would solve our **above** problem.

But, before, we showed that it's more **convenient** if we can store these instead as **matrices**: that way, we don't need the **sum**. Luckily, we can generalize our work with **matrix calculus**.

Below, we will use the alternative notation, to be consistent with the official notes.

$$J = \frac{1}{n} (\tilde{X}\theta - \tilde{Y})^T (\tilde{X}\theta - \tilde{Y}) \quad (2.56)$$

We will not show here how to **find** these derivatives, but the important rules you need to compute our derivatives are in the **appendix**.

There's a document explaining vector derivatives coming soon!

Note that **matrix** derivatives often look **similar** to **traditional** derivatives, but they are **not the same**. Most often, making this mistake will result in **shape errors**.

Sometimes, you can guess a derivative by using the familiar rules and fixing shape errors with transposing/changing multiplication order. But be careful!

When we take our derivative, we get

$$\nabla_{\theta} J = \frac{2}{n} \tilde{X}^T (\tilde{X}\theta - \tilde{Y}) = 0 \quad (2.57)$$

From here, we just solve for θ just like in the official notes.

Key Equation 60

The **solution** for **OLS optimization** is

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

Or, in our **original** notation,

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

And we're done with OLS!

2.6 Regularization

So far, we've shown how to make the **best** model for our **training data**. But now, we want to move to our **real** goal: performing well on **test data**.

This means we want to make a model that is **general**: it can apply well to **new data**.

2.6.1 Regularizers

Only focusing on training data is a **weakness** for our model - if by chance, we have a training data that doesn't **match** our overall distribution, we are likely to make a **bad model**.

Example: You flip **4 coins**, and get **3 heads**. You determine that this coin has a **75% chance** of landing heads. It turns out this **isn't true**: it's a fair coin, and you got **unlucky**.

We may need a **second** way to measure our performance: one that focuses **less** on **current** performance, and **more** on predicting how **generalizable** it is.

You can also increase sample size (flip more times), but for complex problems this isn't always an option!

We call this type of function a **regularizer**.

Definition 61

A **regularizer** is an added term to our **loss function** that helps measure how **general** our hypothesis is.

By **optimizing** with this term, we hope to create a model that works better with **new data**.

This function takes in our **vector of parameters** Θ as an input: $R(\Theta)$

Example: You figure that the coin is **equally likely** to bias towards heads or tails: even if it's **weighted**, you don't know **which way**. So, you start with **50-50 odds**, and **adjust** that based on evidence.

Instead of just focusing on the **specific** data for our coin, we consider how coins act in **general**.

2.6.2 Regularizer for Regression: Prior Knowledge

Now, the question is, **how** do we choose our regularizer? What will make our model more **general**?

We want to **resist** the effects of random **chance**, like in the **coin** example above. In that example, we improved our guess by starting with a **prior assumption**.

If you have some **previous** guess, or past experience, you might have some **model** you **expect** to work well: the data has to **convince** you otherwise.

So, we might consider a model **more different** from that past one, Θ_{prior} , to be **suspicious**, and less likely to be good.

Concept 62

If we have a **prior** hypothesis Θ_{prior} to work with, we might improve our **new** model by encouraging it to be **closer** to the old one.

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

We measure how **similar** they are using **square distance**.

Example: You have a **pretty good** model for **predicting** company profits, but it isn't perfect. You decide to train a **better** one, but you expect it to be **similar** to your old one.

2.6.3 Regularizer for Regression: No Prior Hypothesis

But, what if we **don't have** a prior hypothesis? What if we have **no clue** what a **good** solution looks like?

Well, just like in the **coin** example, we don't expect it to be **more likely** to be **weighted** towards heads or tails.

So, even if we **didn't know** most coins are fair coins, we still would've chosen **50-50** as our guess.

In this case, as far as we know, every θ_k term is **equally likely** to be **positive or negative** - we have no clue.

So, **on average**, we could push for it to be **closer to zero**, so it doesn't drift in any direction too strongly.

Key Equation 63

In general, our **regularizer for regression** will be given by **square magnitude** of θ :

$$R(\Theta) = \|\theta\|^2 = \theta \cdot \theta$$

This approach is called **Ridge Regression**.

We'll discuss why it's called "ridge" regression once we find our solution.

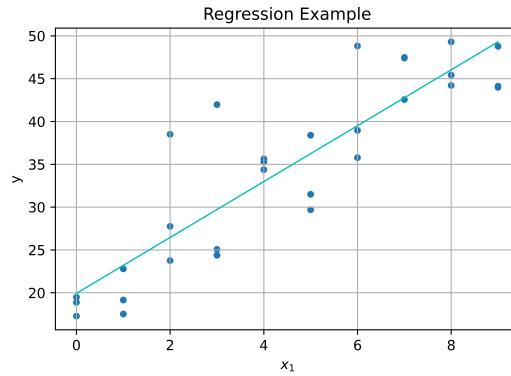
2.6.4 Why not include θ_0 ?

One thing you might immediately notice is that we used the magnitude of θ instead of Θ : this omits θ_0 . Why would we do that?

We'll show that we need to **allow** the **offset** to have whatever value works best, and we shouldn't **punish** it.

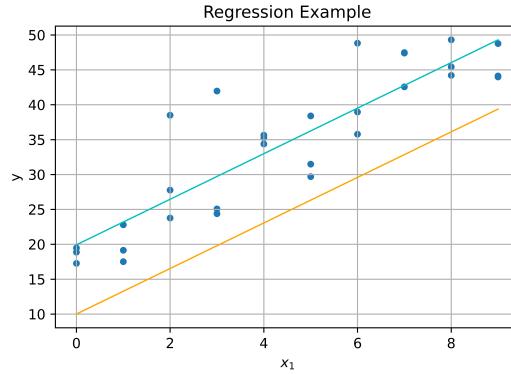
This is best shown with a **visual** example. Let's take an example with one input x_1 . So, we have a **linear** function: $h(x) = \theta_1 x_1 + \theta_0$.

For simplicity, we won't do any regularization here: we can make our point without it.



Our regression example.

Let's suppose we **push** for a **much lower** (offset) θ_0 term, while keeping everything else the **same**:



Reducing our offset pulls our line further away from all of our data! That's not helpful.

This shows that we **need** our offset! We use it to **slide** our hyperplane around the space: if all of our data is **far** from $(0, 0)$, we need to be able to **move** our **entire** line.

And regularizing θ_1 wouldn't make this any better: it would just be flatter.

So, we'll keep θ_0 **separate** and **allow** it to take whatever value is **best**.

Concept 64

We **do not regularize** our **offset** term, θ_0 .

Instead, we allow θ_0 to **shift** our hyperplane wherever it **needs** to be.

The other terms θ control the **orientation** of the hyperplane: the **direction** it is **facing**. We **regularize** this to push it towards less "complicated" orientations.

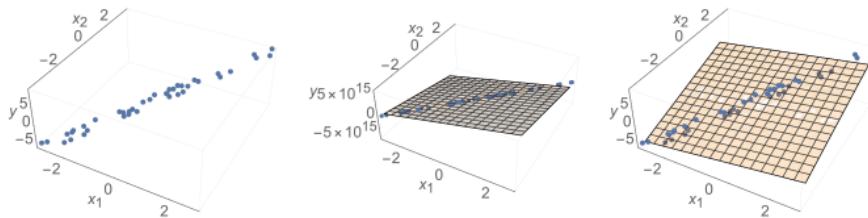
This will be discussed more in-depth in the Classification chapter!

2.6.5 A second benefit of regularization

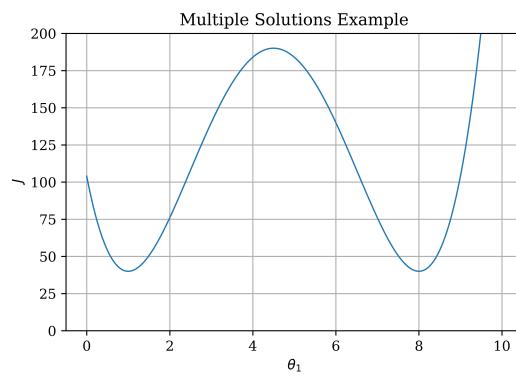
Another benefit of regularization is that it solves a second problem: having **multiple optimal solutions**.

If we have **multiple** best outcomes, we have to pick which one to **choose**. We can make this choice by **picking** the one with the **smallest** magnitude.

We can **visualize** the problem of "multiple best solutions" a couple different ways:



There are many **planes** that can go through this line: multiple equally good solutions!



This compares different hypotheses (θ_1) and sees how well they perform (J): two are equally good!

Either way, we can pick a solution based on lowest θ **magnitude**!

2.6.6 A Math Perspective: Unique Solutions

We can also view this problem more **mathematically**.

Let's look at our **analytical** solution:

$$\theta = (XX^T)^{-1}XY^T \quad (2.58)$$

This solution only works if $(XX^T)^{-1}$ is **valid**. But we have a problem: **not all matrices** have **inverses**.

If XX^T has a **determinant of zero**, then we cannot find an inverse.

Without an inverse, we have **no unique solution**! This is a problem.

This is one thing our **regularizer** $R(\Theta)$ helps us solve: we'll see that our **new solution** will not have this problem!

This is an important idea in linear algebra! If you don't know what this means, here's a [great video](#).

The reason will be clear in the **algebra**, but it's **equivalent** to the reason we discussed the above: we take the best **models** that are all **equally good**, and pick the one with **lowest magnitude**.

Concept 65

Ridge Regression helps **improve** our model by

- Making our model more **general** and resistant to **overfitting**
- Making sure **solutions** are **unique**
- Keeping our matrix XX^T **invertible**, so we can find a **solution**.

2.6.7 Lambda, a.k.a. λ

We now have a term that can help us choose a more **general** hypothesis. One important question is, **how general** do we want it to be?

The more general we make our model, the **less specific** to our current data it is. This may seem like a good thing, but too much can make our model **worse**!

If λ is **too large**, then your model will stay **very close** to $\|\theta\| = 0$. This probably isn't a good solution for most cases.

But if it's **too small**, then it **won't** have enough of an **effect**. So, we need to be able to adjust how **much** we're regularizing.

To do this, we will **scale** our regularizer by a **constant** factor, λ .

Definition 66

Lambda, or λ , is the constant we **scale** our **regularizer** by.

It represents **how strongly** we want to regularize: how much we prioritize **general** understanding over **specific** understanding.

2.6.8 Our new objective function

Now that we have our regularizer,

$$R(\Theta) = \lambda \|\theta\|^2 \quad (2.59)$$

We can add it to our objective function:

Key Equation 67

The **objective function** for **ridge regression** is given as

$$J(\theta) = \frac{1}{n} \sum_{i=1}^n \left(\underbrace{(\theta^T x^{(i)} + \theta_0)}_{\text{guess}} - \underbrace{y^{(i)}}_{\text{answer}} \right)^2 + \underbrace{\lambda \|\theta\|^2}_{\text{Regularizer}}$$

This is the form we will **solve**.

2.6.9 Matrix Form Ridge Regression

Just like before, we'll switch from a **sum** to a **matrix** in order to solve this problem.

Creating an **equation** for both θ and θ_0 is, frankly, **annoying to derive**. **Instead**, we'll cheat a little, and keep θ_0 in and create our **matrix-form** objective function:

$$J = \frac{1}{n} (\tilde{X}\theta - \tilde{Y})^T (\tilde{X}\theta - \tilde{Y}) + \lambda (\theta^T \theta) \quad (2.60)$$

Our work begins. Let's take the **gradient**: what we want to set to zero.

$$\nabla_{\theta} J = \frac{2}{n} \tilde{X}^T (\tilde{X}\theta - \tilde{Y}) + 2\lambda\theta = 0 \quad (2.61)$$

We do some algebra and **solve** as we do in the **official notes**:

Key Equation 68

The **solution** for **ridge regression optimization** is

$$\theta = (\tilde{X}^T \tilde{X} + n\lambda I)^{-1} \tilde{X}^T \tilde{Y}$$

Or, in our **original** notation,

$$\theta = (X^T X + n\lambda I)^{-1} X^T Y$$

2.6.10 Our new term, $n\lambda I$

So, we already established that **regularization** helps us create more **general** hypotheses that are lower in magnitude.

But, how does this **mathematically** solve our invertibility problem?

$$\theta = (X^T X + n\lambda I)^{-1} X^T Y \quad (2.62)$$

This term, $n\lambda I$, is added to the matrix we want to invert. Let's see what this matrix looks like. We'll use a (3×3) example: _____

I is the **identity matrix** in our notation.

$$n\lambda I = n\lambda \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = n \begin{bmatrix} \lambda & 0 & 0 \\ 0 & \lambda & 0 \\ 0 & 0 & \lambda \end{bmatrix} \quad (2.63)$$

This visual, having a "ridge" of λ s along the diagonal, is why we call it **ridge regression**.

2.6.11 Invertibility

This term $n\lambda I$ shifts the values of XX^T so that we **avoid** having a **determinant of zero**.

Since the **determinant is nonzero**, we don't have to worry about an **uninvertible matrix**: we now have a **unique** inverse, and thus a **unique** solution.

Concept 69

Ridge Regression solves the problem of **matrix invertibility** (non-unique solutions) by adding a term $n\lambda I$, our **ridge** of diagonals.

This turns the inverse $(XX^T)^{-1}$ into

$$(XX^T + n\lambda I)^{-1}$$

Which can prevent a **determinant** of zero in our solution, given $\lambda > 0$.

2.7 Evaluating Learning Algorithms

Now, we have successfully developed an **algorithm** for **learning** from our data. But, did our algorithm make a **good** hypothesis? How do we do **better**?

2.7.1 What λ should we choose?

There's something we ignored earlier: how do we pick the **best** value of λ ? We didn't go into detail, but that value of λ will affect our algorithm's **performance**.

This λ adjusts exactly how we **learn**: how much more do we learn **specifically**, versus **generally**?

We mentioned that different λ values have different **tradeoffs**, so we need to figure out which λ value is best for our problem.

We'll need to **optimize** our λ value! Let's figure out how to go about that.

2.7.2 Tradeoffs: Estimation Error

High and low λ values have benefits and drawbacks. These tradeoffs can be loosely divided into **two categories**.

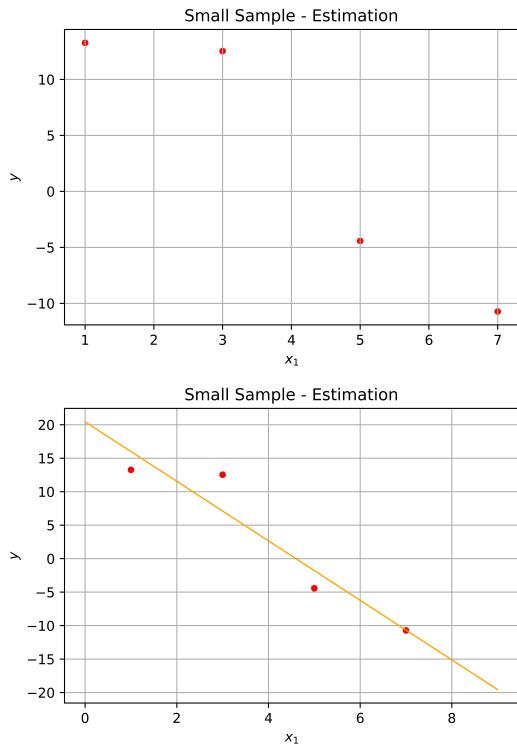
When we generalize, we're trying to avoid **estimation error**: we incorrectly guess the overall distribution we're trying to fit. We **estimate** poorly if we **generalize** poorly.

Definition 70

Estimation error is the error that results from poorly **estimating** the **solution** we're trying to find.

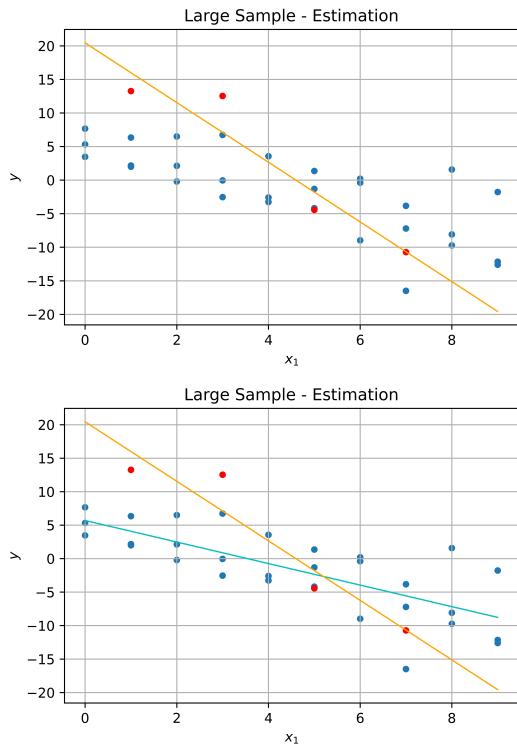
This can be caused by **overfitting**, getting a bad (**unrepresentative**) sample, or not having enough **data** to come to conclusion.

Example: Let's try a regression problem, but we'll use only 4 points to make our plot.



This is the regression solution we get based on our small dataset.

We might be suspicious. One way to reduce **estimation error** is to increase our number of data points (though this isn't always an option, or sufficient!)



Our regression from before doesn't look so good on this model... We make an updated regression, and get a more accurate result.

Clarification 71

λ doesn't lower **estimation error** in the **same way** that increasing **sample size** does, but the problem is **similar**.

2.7.3 Tradeoffs: Structural Error

However, not all problems are caused by estimation error: sometimes, it **isn't even possible** to get a good result - you chose the wrong **model class**.

This means the **structure** of your model is the problem, not your method of **estimation**. Thus, we call this **structural error**.

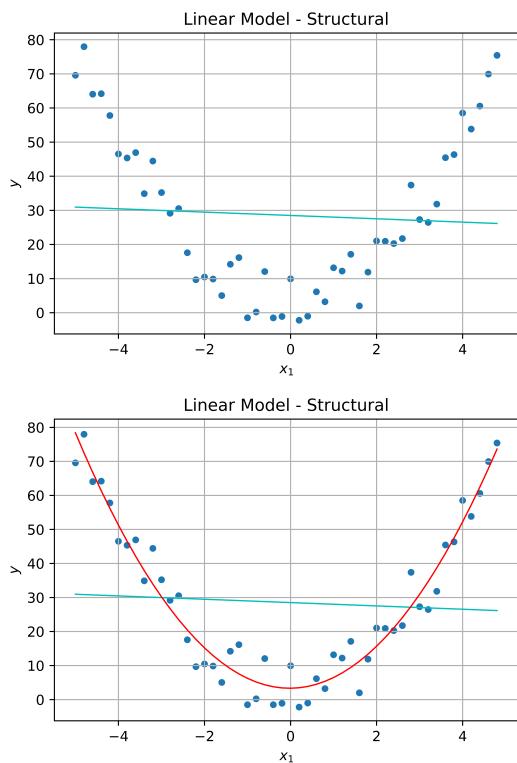
Definition 72

Structural error is the error that results from having the wrong **structure** for the **task** you are trying to accomplish.

This can result from the **wrong class** of model, but sometimes, your model class doesn't have the **expressiveness** it needs for a complex problem.

It can also happen if your algorithm **limits** the available models in some way, like how λ does.

Example: If the **true shape** of a distribution is a parabola x^2 , there is **no** linear function $mx + b$ that can match that: this creates **structural error**.



Our **linear** model isn't able to represent a quadratic function... so, we switch to a more expressive model: a **quadratic** equation.

Clarification 73

Note that λ does not restrict our model class **as severely** as **switching polynomial order**, like above.

But, λ **limits** the use of larger θ , which does make it **unable** to solve some problems. So, the **structural error** problem is similar.

Remember that **expressiveness** is about how many possible models you have: if you have more models, you can solve more problems.

2.7.4 Tradeoffs of λ

Based on these two categories, we can discuss the tradeoffs of λ more easily.

As we mentioned, regularization **reduces** estimation error:

If we overfit to our current data, we are poorly **estimating** the distribution, because the training data may not perfectly **represent** it.

Concept 74

A large λ means **more regularization**: we more strongly push for a more **general** model, over a more **specific** one.

This results in...

- **Reduced** estimation error
- **Increased** structural error

However, **regularization** also **limits** the possible models we can use - those it views as less "general", it **penalizes**.

That means the scope of possible models is **smaller** - some models are no longer **acceptable**. What if the only valid solution was in that space we **restricted**? Well, then we can't **find** it.

That means there are certain **structural** limits on our model: that means that regularization **increases** structural error!

Concept 75

A small λ means **less regularization**: we care less about a more **general** model, allowing more **specific** data to come into play.

This results in...

- **Increased** estimation error
- **Reduced** structural error

2.7.5 Evaluating Hypotheses

So, we know that we have these **two** types of **error**. But it's **difficult** to **measure** them separately.

So instead, we just want to measure the **overall performance** of our hypothesis.

We do this using our **testing error**: this tells us how good our hypothesis is **after** training.

$$\mathcal{E}(h) = \frac{1}{m} \sum_{i=n+1}^{n+m} (h(x^{(i)}) - y^{(i)})^2 \quad (2.64)$$

Note that, before, we were using **regularization**. This is so we can **make** a more **general** model.

But here, we've **removed** it, because training is **done**: we're **not** going to make our hypothesis **better**. We just care about how **good** it came out.

We're already measuring the generalizability by using new data!

Clarification 76

When we **evaluate a hypothesis** using **testing error**, we are **done training**: our hypothesis will not change.

Because of this, we **do not** include the **regularizer** when **evaluating** our hypothesis.

2.7.6 λ 's purpose: learning algorithms

Notice that we **removed** regularization when we were **evaluating** our hypothesis: regularization was used to **create** our hypothesis, but it is not **part** of that hypothesis.

That's because λ is part of our **algorithm**: it determines how we run our algorithm. So, let's talk about that.

Our hypothesis only includes the parameters Θ : not λ !

Definition 77

A **learning algorithm** is our procedure for **learning** from data. It uses that data to create a **hypothesis**. We can diagram this as:

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

In a way, it's a function that takes in **data** \mathcal{D}_n , and outputs a **hypothesis** h .

We're choosing **one hypothesis** h from the hypothesis class \mathcal{H} : this is why \mathcal{H} appears in the notation above.

We can write this as
 $h \in \mathcal{H}$

2.7.7 Comparing Hypotheses and Learning Algorithms

We can take our learning algorithm

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

And compare it to our hypothesis h :

$$x \rightarrow [h] \rightarrow y$$

In a way, our learning algorithm is a function, that outputs another function!

Our **hypothesis** can be adjusted with our **parameter** Θ : if we change Θ , we change our **performance**.

This is similar to \mathcal{E}_n , which instead **takes in** a function!

Our **learning algorithm** depends on λ : so, it's like a **parameter**. But, it's different from Θ : Θ is our model, λ controls how we **choose** our model.

So, it's a parameter (λ) that affects other parameters (Θ). Because of that, we call it a **hyperparameter**.

It affects our hypothesis by pressuring it to have lower magnitude!

Definition 78

Parameters are **variables** that adjust the behavior of **our model**: our hypothesis.

A **hyperparameter** is a **variable** that can adjust **how we make models**: our learning algorithm.

The **only** hyperparameter we have for now is λ , but the **development** of hyperparameters is an ongoing area of **research**.

Concept 79

Lambda, or λ , is a **hyperparameter**: it controls our **learning algorithm**.

2.7.8 Evaluating our Learning Algorithm

So, while we can evaluate each **hypothesis**, it's also important to measure how our **learning algorithm** is performing.

How do we measure it? Well, the job of our **learning algorithm** is to **pick good hypotheses**.

Concept 80

We can **evaluate** the performance of a **learning algorithm** using **testing loss**: a good learning algorithm will create **hypotheses** with **low testing loss**.

You could think of this as measuring the **skill** of a **teacher** (the learning algorithm) by the **success** of their **student** (the hypothesis) on a **test** (testing loss).

2.7.9 Validation: Evaluating with lots of data

When we were creating hypotheses, **randomness** caused some problems: you might not get **training data** that matched the **testing data** very well.

The **same** can happen here, when **evaluating your algorithm**: maybe your model happened to create a bad (or unusually good!) hypothesis because of **luck**.

The easy solution to **randomness** is to add **more data**: we get more **consistency** that way.

So, we **repeatedly** get new training data and test data. For each, we **train** a different hypothesis. We can **average** their performance out, and use that to **estimate** the quality of our algorithm.

Definition 81

Validation is a way to **evaluate a learning algorithm** using **large amounts of data**.

We do this by **running** our algorithm **many times** with new data, and **averaging** the testing error of all the hypotheses.

This process is often requires having **lots of data** to train with, but is a **provably** good approach.

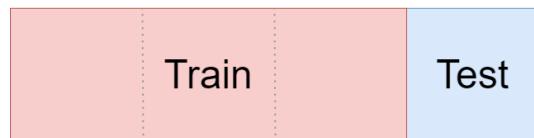
2.7.10 Our Problem: When data is less available

As mentioned, this takes up **lots of data**. What if we can't get as much: it's **expensive**, or not even possible? In this case, we have some **finite** data, \mathcal{D}_n . We **can't get more**.

We solved the **randomness** problem by using **more** training and testing **data**. So, we need some way to **get more distinct** hypotheses.

One set of data gives us one **hypothesis**. But, what if, rather than using **completely** new data, we used **slightly different** data each time?

First, need to break \mathcal{D}_n into a chunk for training, and a chunk for testing.



How do we get more hypotheses from this dataset?

2.7.11 Cross-Validation

We mentioned that we just want **different** hypotheses. Our hypotheses depend on our **training data**. So we want to **change** our training data.

We can't **add** data to it, because then we **lose** testing data. We shouldn't **remove** data, because then we're just making a hypothesis that's **less well-informed**.

Instead, we'll **swap** some of the training data for testing data.



This will create a new hypothesis, and the data is partially different! In fact, we can do this for each of our chunks:



We now have **four different hypotheses** for the price of one!

Definition 82

Cross-validation is a way to **evaluate a learning algorithm** using **limited data**.

We do this by **breaking** our data into **chunks** to create **multiple hypotheses** from one dataset.

For each **chunk**, we train one dataset on all the data **not in that chunk**. We get our **test error** using the chunk **we left out**.

For k chunks, we end up with k hypotheses. By **averaging** out their performance, we can **approximate** the quality of our algorithm.

This approach is much **less expensive**, and very common in machine learning! But, some of the theoretical **benefits** of validation are not **proven** to be true for cross-validation.

Clarification 83

Note that the goal of validation and cross-validation is **not** to evaluate **one hypothesis**.

Instead, it is instead meant to evaluate a **learning algorithm**. This is why we have to create **many** hypotheses: we want to see that our algorithm is **generally** good!

2.7.12 Hyperparameter Tuning

Now, we know how to **evaluate** a learning algorithm, just like how we **evaluate** a hypothesis.

Once we knew how to evaluate a hypothesis, we started **optimizing** our parameters for the **best** hypothesis. So, we could do the same for our **learning algorithm**.

Each λ value creates a slightly **different** learning algorithm: we can **optimize** this **hyperparameter** to create the **best** learning algorithm.

2.7.13 How to tune our algorithm

When we were **optimizing** our hypothesis, we started by **randomly** trying hypotheses. Then, we used an **analytical** approach.

Last Updated: 03/09/23 21:42:28

By "analytical", we mean directly creating an equation, and solving it.

We don't always have **simple** equations to work with: with all of our data, it's hard to come up with **manageable** equations. So, we **won't** try doing it **analytically**.

So, we could **randomly** try λ values and pick the **best** one. This is pretty **close** to what we usually end up doing. For each value we pick, we'll use **cross-validation** to evaluate.

For now, we'll systematically go through λ values: $\lambda = .1, .2, .3 \dots$

Concept 84

Hyperparameter tuning is how we **optimize** our **learning algorithm** to create the **best** hypotheses.

The simplest way to do this is to try **multiple** different values of λ . For each value, we use **cross-validation** to evaluate that learning algorithm.

Finally, we pick whichever λ gives you the **best** algorithm, and thus the **best** hypotheses.

2.7.14 Hyperparameter Tuning: Two kinds of optimization

There's something often **confusing** about hyperparameter tuning to students:

When we're **optimizing** λ , we have to determine the quality of **each** learning algorithm.

But, to get the **quality** of that algorithm, we have to optimize Θ based on that **single** learning algorithm.

That means, **every time** we try a different λ value, we have to do one optimization problem. But trying different λ values is a **different** kind of optimization.

If we do cross-validation, then we have to optimize k times!

That means we have **two layers** of optimization!

Clarification 85

We **optimize** λ by trying many values.

But, for each λ value, we have to **optimize** Θ .

So, we have to optimize Θ **repeatedly** in order to optimize λ **once**! This gives us λ^* .

But, our goal is to get a **hypothesis**. So we use that λ^* to, finally, get our θ^*

2.7.15 Pseudocode Example

This technique is **not** limited to regression. Thus, we'll be a bit more **general**: we won't assume an **analytical** solution. Instead, we **optimize** by just trying different Θ values.

We can represent this in pseudocode:

If this pseudocode isn't helpful to you, don't worry! Some students like it, some don't.

```
LAMBDA-OPTIMIZATION( $\mathcal{D}$ , lambda_values, theta_values)
1  for  $\lambda$  in lambda_values      #Try lambda values
2      for  $\Theta$  in theta_values    #Try theta values
3          Calculate  $J(\Theta)$         #Compare values
4          Choose best theta value  $\Theta^*$  #Best for each lambda
5  Choose best lambda value  $\lambda^*$ 
6
7  return  $\lambda^*$ 
```

To reiterate: this λ^* will then we used to get our final result, θ^* .

2.8 Terms

- Hypothesis
- Theta (Θ)
- Input Space
- Regression
- Feature
- Feature Transformation
- Training Error
- Test Error
- Objective Function
- Min function
- Argmin function
- Star Notation (θ^*)
- Linear Regression
- Hypothesis Class
- Square Loss
- Ordinary Least Squares (OLS) Problem
- OLS Objective Function
- Hyperplane
- Weight
- Input Matrix

- Output Matrix
- Gradient
- OLS Solution
- Regularization
- Regularizer
- Regularizer for Regression
- Lambda (λ)
- Ridge Regression
- RR Objective Function
- RR Solution
- Invertibility
- Estimation Error
- Structural Error
- Expressiveness
- Learning Algorithm
- Hyperparameter
- Validation
- Cross-Validation
- Chunk (Cross-Validation)
- Hyperparameter Tuning

CHAPTER 3

Gradient Descent

What is gradient descent?

3.0.1 Why do we need gradient descent?

In the last chapter, we used an **analytical** approach to solve the OLS and RR problems.

By "analytical", we mean we got an **explicit** answer: an equation we can use to directly compute the correct answer.

The trouble is, we can't always do this:

- Sometimes the problem or the loss function can't be **rearranged** into a simple **equation**.
- Or, we have **too much** data, and directly computing the answer would take way **too long**.

Concept 86

Most **problems** we come across cannot be solved **analytically**.

Well, if we can't **directly** find the **best** answer, what's the next best thing? Finding a **better** solution than your current one.

So, our mission is to gradually try to find a better and better answer. This type of approach has a couple benefits:

- It's **quicker** to see if we're using a good model: if we're making very little progress, we can **quit** early and try something else.
- If we don't need **all** of our data to get the answer, we don't need to spend as much time. If our answer is **good** and not getting better, we can **stop**.
- It's easier to find a **better** answer than the **best** answer: our equations will be **simpler**. In some case, it might not have even been **possible** without this gradual approach!

Concept 87

When we can't reasonably find a **best** answer, it's often easier to find a **better** answer and gradually **improve**.

Gradient descent follows this philosophy: we gradually **update** our solution to make it better and better.

3.0.2 How do we improve?

So, now, the question is: how do we **improve** our hypothesis? We'll be modifying our hypothesis θ by some amount: _____

$$\theta_{\text{new}} = \theta_{\text{old}} + \Delta\theta \quad (3.1)$$

We'll do the same for θ_0 , but we'll do it separately. We'll come back to that.

Notation 88

In equations, we'll often use θ_{old} and θ_{new} to represent **before** and **after** we take a step.

We will use this notation **elsewhere** in the class.

So, we are interesting in $\Delta\theta$: how do we plan to change θ ? What does $\Delta\theta$ look like?

Well, we want to modify

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix} \quad (3.2)$$

So, we want to modify each of those terms.

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix} + \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \\ \vdots \\ \Delta\theta_d \end{bmatrix} \quad (3.3)$$

So, we have our total change!

$$\Delta\theta = \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \\ \vdots \\ \Delta\theta_d \end{bmatrix} \quad (3.4)$$

Notice that the shape of this change matches the shape of θ : ($d \times 1$).

Concept 89

We need a **separate** term $\Delta\theta_i$ for each θ_i we want to **improve**.

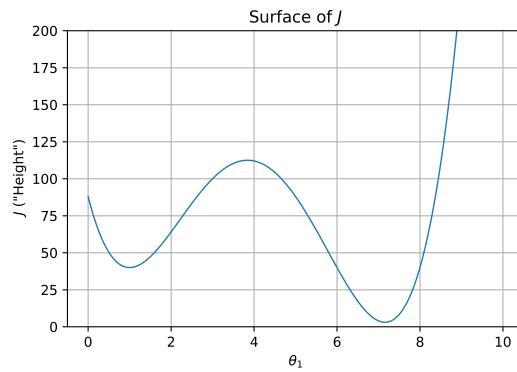
So, a vector of the **total** change, $\Delta\theta$, needs to have the **same shape** as θ : ($d \times 1$).

3.0.3 The name: "gradient descent"

Our goal is to gradually **decrease** J , step-by-step. We do this using the **gradient**, hence "gradient descent". Why the gradient? We'll discuss that later.

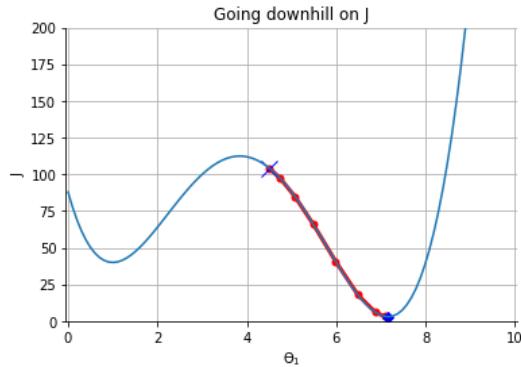
But why the word "**descent**"?

Our intuition is to imagine J as having a **height** at every input value. If you combine all of these different points, you get a **surface**, like the surface of a hill.



You can imagine this like some hills we want to "descend".

Then, decreasing J is moving **down** the function, similar to rolling a ball down a hill. In other words, we **descend** the hill.



Like this! Starting from the blue "X", moving 'downhill'.

3.0.4 Input Space vs. Parameter Space

One more thing to note: we have two similar situations.

- J is a **function** with θ as an **input**: $J(\theta)$.
- h is a **function** with x as an **input**: $h(x)$.

In both cases, we can imagine the **output** as the "**height**" of our function: the **hill** we mentioned before. This **physical** intuition is useful to **gradient descent**.

But, what about **input** to our function? That's the x -axis our hill is floating above:

- With $h(x)$, our x -axis was our **input space**, all possible x_1 values: the "space" containing all of our possible inputs.
- With $J(\theta)$, our x -axis is the **parameter space**, all possible θ values. We also called this our "**hypothesis space**".

Definition 90

The **parameter space** is our set of all **possible** parameter combinations.

This is the same as the **hypothesis space**, because our parameters **define** our hypothesis.

When we **optimize** our hypothesis, we are "**exploring**" the hypothesis space.

We're assuming 1-D right now for simplicity. If we were 2-D, we'd have an entire 2D grid under our hill!

This also gives us an idea of which hypotheses are "**similar**": those which are **closer** in parameter space (which we used, when we were doing regularization $\|\theta - \theta_{\text{old}}\|$).

This is the **space** we're exploring, as we try to move **downhill**.

Clarification 91

Pay attention to your **axes**!

Sometimes, we're doing a 2-D or 3-D plot of J , and our inputs are θ_k . Other times, we're plotting hypothesis h , with our axes x_i .

These two plots could have the same surface, but they **represent** completely different things.

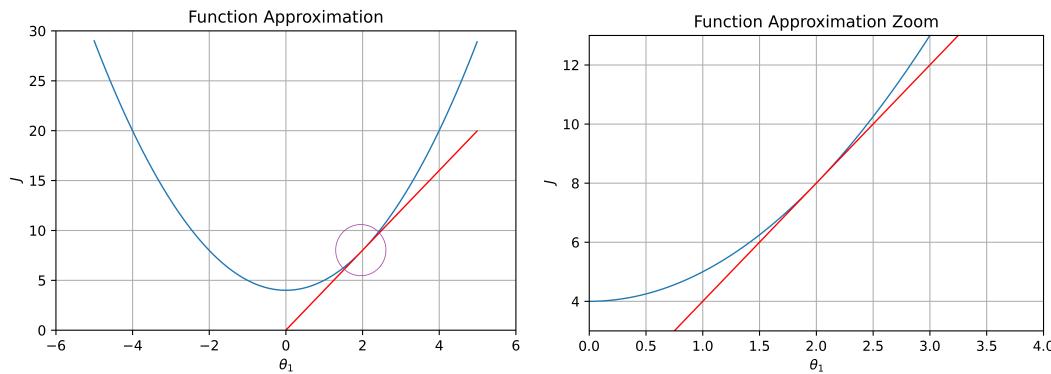
3.1 Gradient Descent in One Dimension

3.1.1 Derivatives (Review)

Here, we'll use some concepts from **calculus**.

We'll make improvements in small **steps**. And, we measure our improvement against the **loss function**, J : that's what we want to **optimize**.

In calculus, we found that, over **small** enough steps, you can **approximate** as function as a straight line.



It looks more like a line as we zoom in: hence the **local** approximation.

Concept 92

A **smooth** (enough) function can be **approximated** with a **straight line** if you **zoom** in on it enough.

Looking at it this way is called a **local** view.

3.1.2 Optimize with Derivatives: 1-D

This gives us the **slope** of the function locally. Last chapter, we used $\frac{dJ}{d\theta} = 0$ to get our **minimum**.

But, let's not get too greedy - we want to **improve** our hypothesis, **not** immediately try to find the **best** one.

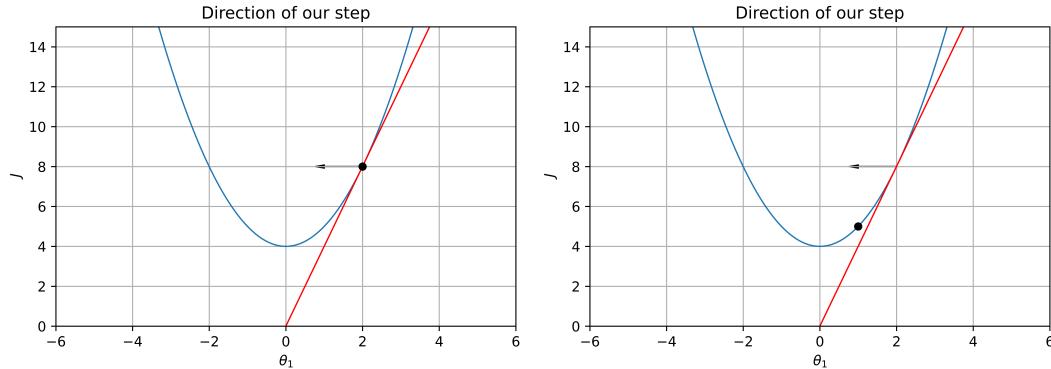
Well, what does our slope tell us? It tells us:

Because the best one might be expensive to find this way!

- How quickly J changes
- Whether it **increases** or decreases as we change θ

That second one tells us *how to change* θ : we want to move in the direction that **decreases** J .

If the slope is **positive**, then we want to **decrease** $\Delta\theta$: the sign of $\Delta\theta$ is the opposite of our desired change!



Our slope is **positive**. We want to **decrease** our function, so we move in the **negative** direction, and "fall down" the surface.

And so, for now, we have

$$\Delta\theta = -\frac{dJ}{d\theta} \quad (3.5)$$

Concept 93

In **1-D**, you can use the **derivative** to **optimize** our function J .

The **derivative** tells us how to immediately adjust θ_i to **improve** our J **locally**: we move in the **opposite direction**.

This gives us a procedure for optimizing J : get the derivative $J'(\theta)$, and repeatedly adjust θ in the opposite direction until you're satisfied.

There's a certain way this feels like we're moving "**downhill**": we're moving "down" the slope, to try to find a local **minimum**.

We'll need to pick a condition for being satisfied, but we'll get to this later

3.1.3 Convergence

If you do this procedure with the above equation, though, you'll often run into **problems**. Why is that?

Well, because each of your steps is too **big** or too **small**: we won't be able to find a **stable** answer, i.e. **converge**!

What does it mean to **converge**?

It means we get a **single answer** after repeated steps: given enough time, we'll get **close as we want** to one number, and **stay there**.

Definition 94

If a sequence **converges**, then our result gets as **close as we want** to a **single number**, without going **further away**.

Example: The numbers $1/n: \{1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \dots\}$ converges to 0.

If our answer **doesn't converge**, then it **diverges**. We can see why this might be bad: if we never **approach** a single answer, how do we know what value to **pick**?

3.1.4 Convergence: A little more formally (Optional)

Let's be more specific. Our sequence S will converge to r .

$$S = \{s_1, s_2, s_3, s_4, \dots\} \quad (3.6)$$

"As close as we want": let's say we want the maximum distance to be ϵ . That means, no matter what $\epsilon > 0$, we'll get closer at some point: $|m - s_i| < \epsilon$

$$|m - s_i| < \epsilon \text{ for some } i \quad (3.7)$$

"And stay there": at some time k , we never move further away again:

Definition 95

If a sequence S **converges** to m , then for all $\epsilon > 0$, we can say

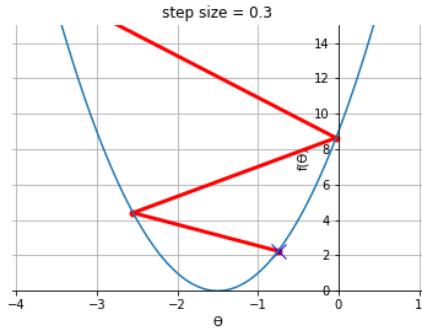
$$|m - s_i| < \epsilon \text{ for all } i > k \quad (3.8)$$

This is a "formal" definition of convergence.

3.1.5 Step size

If your steps are too **big**, your result might **diverge**: you make such big jumps, you move **away** from the minimum, and get worse.

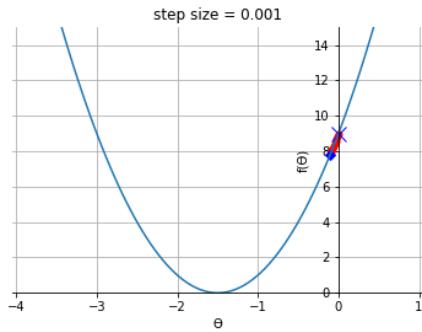
Remember, if it **diverges**, it never **approaches** a single value!



We start at the blue "x" mark. Notice that, even though we try to move toward the minimum, we go too far and accidentally get further and further!

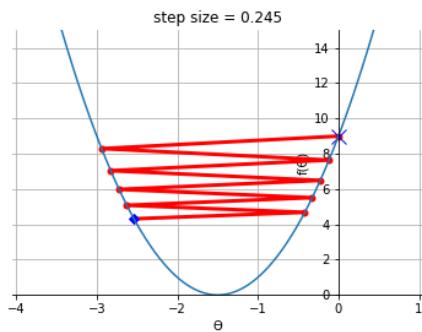
If they're too **small**, you might **converge** too slowly: it'll take way **too long** to make progress.

Converging means it successfully approaches an answer!



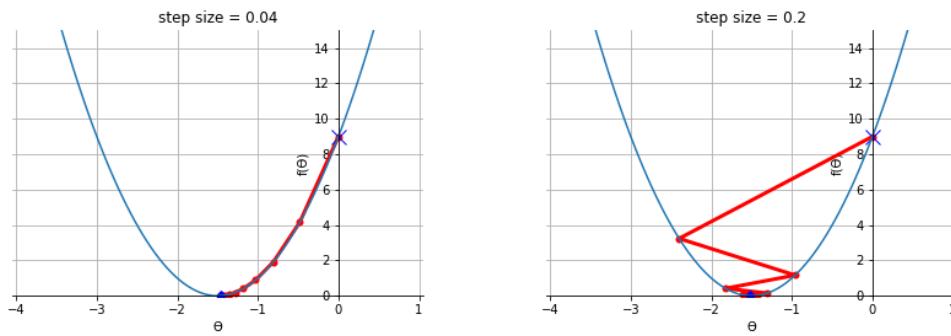
Our step size is too small: this is going to take too long!

In-between, it might converge, but **oscillate** a bunch: this can slow down getting an answer!



Most of our step is spent undoing the last step... we get better very slowly.

But, if we get the right step size, it'll converge nice and reasonably!



Both of these look pretty good! One of them oscillating a bit is fine.

One question you might ask is, "how much oscillation is too much? Am I converging **fast enough?**"

This is a good question, but the simple answer is that there is **no objective answer**: it depends on what you **need** and how much **time** you have. But you should strive to do **better** when you can!

Concept 96

Using the **wrong** step size can cause:

- Slow convergence
- Strong Oscillation
- Divergence

Which is why we **adjust** the step size using η .

3.1.6 Step size η

Right now, our step size is at the mercy of $J'(\theta)$. But, we don't have to be: we could **scale** our step size up or down.

We do this with our **scaling** factor (also called a **learning rate**), η .

So, we can rewrite our **change** in θ as:

$$\Delta\theta = -\eta \frac{dJ}{d\theta} = -\eta J'(\theta) \quad (3.9)$$

Definition 97

Our step size parameter η , or **eta**, **scales** how large each of our optimization steps are.

If η is bigger, we might **learn** faster, but we also risk **diverging**.

Different values of η are good for **different situations**.

3.1.7 Our procedure

So, we have our parameter **update**, $\Delta\theta$. We'll start at $t = 0$.

Before, we represented the i^{th} **data point** with $x^{(i)}$. We'll reuse this **notation**.

Notation 98

Here, we're changing θ over **time**: each step happens at $t = \{1, 2, 3, \dots\}$ so we need **notation** for that.

We'll **reuse** the notation from $x^{(i)}$, for the i^{th} data point.

In this case, we'll do $\theta^{(t)}$: the value of θ after t **steps** are taken.

Earlier, we **introduced** θ_{old} and θ_{new} : these are $\theta^{(t-1)}$ and $\theta^{(t)}$.

Example: After **10 steps** of 1-D gradient descent, we have gone from $\theta^{(0)}$ to $\theta^{(10)}$.

So, we move the **first** time using $J'(\theta^{(0)})$.

Once we've moved in parameter space **one** time, though, our **derivative** has changed: we're in a different part of the **surface**.

So, we'll take a **second** step with a **new** derivative, $J'(\theta^{(1)})$.

We want to do this **repeatedly**. We'll take our equation

$$\theta_{\text{new}} = \theta_{\text{old}} + \Delta\theta \quad (3.10)$$

And combine it with our **chosen** step size.

Key Equation 99

In **1-D, Gradient Descent** is implemented as follows:

At each time step t , we **improve** our hypothesis θ using the following rule:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta J'(\theta_{\text{old}})$$

Using $\theta^{(t)}$ notation:

$$\theta^{(t)} = \theta^{(t-1)} - \eta J'(\theta^{(t-1)})$$

We repeat until we reach whatever our chosen **termination condition** is.

We can also write it as:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \left(\frac{dJ}{d\theta_{\text{old}}} \right)$$

We've got our gradient descent **update** rule in 1-D!

3.1.8 Termination Conditions

When do we **stop**? We can't let it run forever.

We have some options:

- Stop after a **fixed** T steps.
 - This has the advantage of being **simple**, but how do you know what the **correct** number of steps is?
- Stop when θ **isn't changing** much: $|\Delta\theta| < \epsilon$, for example.
 - If our θ isn't changing much, our algorithm isn't **improving** our hypothesis much. So, it makes sense to stop: we've stabilized.
- Stop when the **derivative is small**: $|J'(\theta)| < \epsilon$.
 - Mathematically **equivalent** to our last choice. But a different **perspective**: if the slope is small, our surface is relatively **flat**, and we're near a **minimum** (probably).
 - "The derivative is **small**" is weaker, but in the same spirit as "the derivative is **zero**", $J'(\theta) = 0$, from last chapter.

3.1.9 Convergence Theorem

It turns out, if our function is **nice** enough, and we pick the **right** value of η , we can guarantee convergence!

Theorem 100

We want to optimize function J . If J is

- Smooth enough
- Convex

And

- η is small enough

Then gradient descent **will** converge to the **global minimum**!

"Small enough" seems vague, but it basically means, "an η small enough to converge **exists**".

Or, if your η is too **big**, you can keep trying **smaller** ones, until it works.

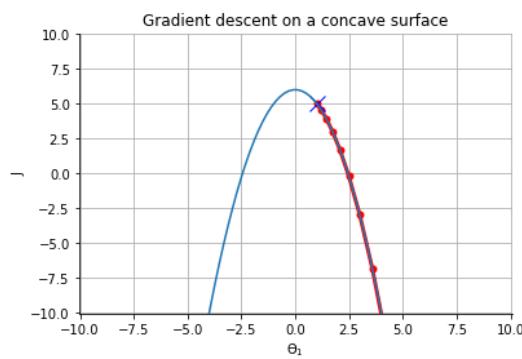
This is amazing! We can **guarantee** a best solution in some cases!

If we want to say it in a mathy way, we can say "**there exists** an η small enough to converge"

3.1.10 Concavity

One requirement we haven't focused on "J is **convex**". Why do we need J to be convex?

Well, if it's **concave**, there is no **global minimum**: it goes down forever!



Our gradient just leads us downhill forever.

Concept 101

If our function J is **concave**, then our result will not **converge**: it will continue to **decrease** more and more indefinitely.

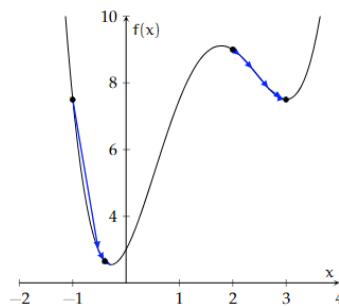
So, for future problems, let's assume it **doesn't** go down forever: if it was, then there is no best solution! We don't have a **valid** problem.

3.1.11 Local minima

Even if we don't have that problem, we have a **different** one:

Gradient descent **gradually** improves our solution until it reaches one it's **satisfied** with. But, what if there are **multiple** solutions we could reach?

Are they all equally good?



Depending on your starting position (**initialization**), you could find a different local minimum!

Maybe not! So, if our function isn't **always convex**, we can end up with **multiple** "valleys", or **local** minima.

Definition 102

A **global** minimum is the **lowest** point on our entire function: the one with the lowest **output**.

A **local** minimum is one that is the **lowest** point among those points that are **near** it.

- For **local minima**, if you add or subtract a **small** amount ϵ , the value will **increase**.

So, we **won't** necessarily end up with the **global** minimum, even with a *small* η .

This shows that **initialization matters**!

Definition 103

Initialization is our "starting point": when we first **start** our algorithm, what are our **parameters** set to?

If we have a **different** starting position, we can find a **different** local minimum.

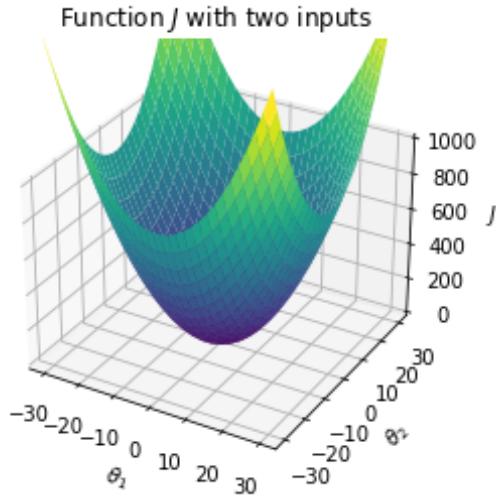
Concept 104

Gradient descent finds **local** minima near the initialization, not **global** minima.

This means, if our function has **multiple local minima** (not fully convex), our **initialization** can affect our **solution**.

3.2 Multiple Dimensions

Now that we've handled the 1-D case, we'll move into 2-D: now, we have **two** parameters, θ_1 and θ_2 , as the input to J .



The "height" of your plot in 3D, is, again, your output! You want to move **downhill**.

3.2.1 Multivariable Local Approximation (Review)

Again, we rely on **calculus**. We want to move up to having more parameters: more **dimensions**.

Before, in 1-D, we found that, if you **zoomed** in enough on a function (using a "**local view**"), we could **approximate** it as a **straight line**, and move up or down that slope.

There are **two** ways we can **approximate** like we want to in 2-D: _____

- First, we could turn it back into 1-D: we remove one variables.

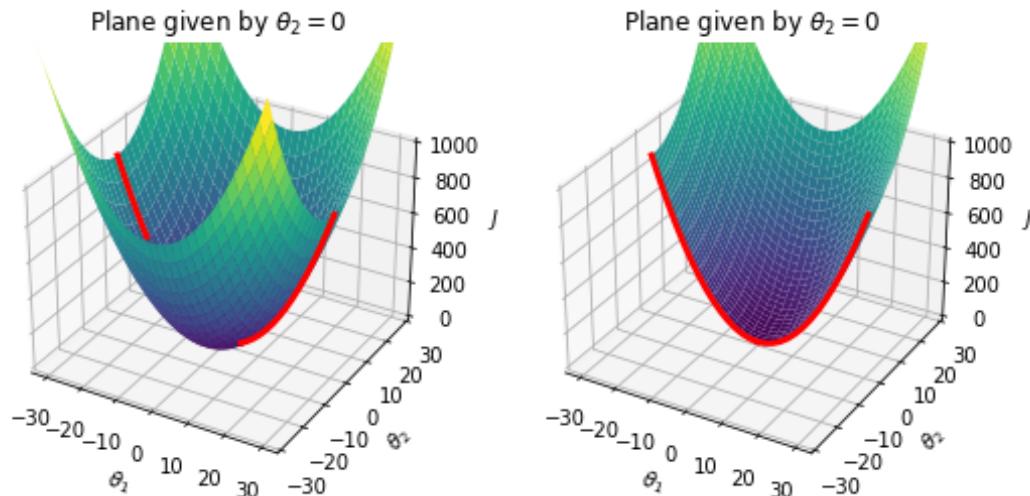
We do this by turning one variable constant: take $\theta_2 = 0$. Now, we have one free variable θ_1 . Same as 1-D.

Remember that, by 2-D, we mean two **parameters**/inputs to J . If we add in the **height** of our function, that means our plot will **look** like 3-D!

Concept 105

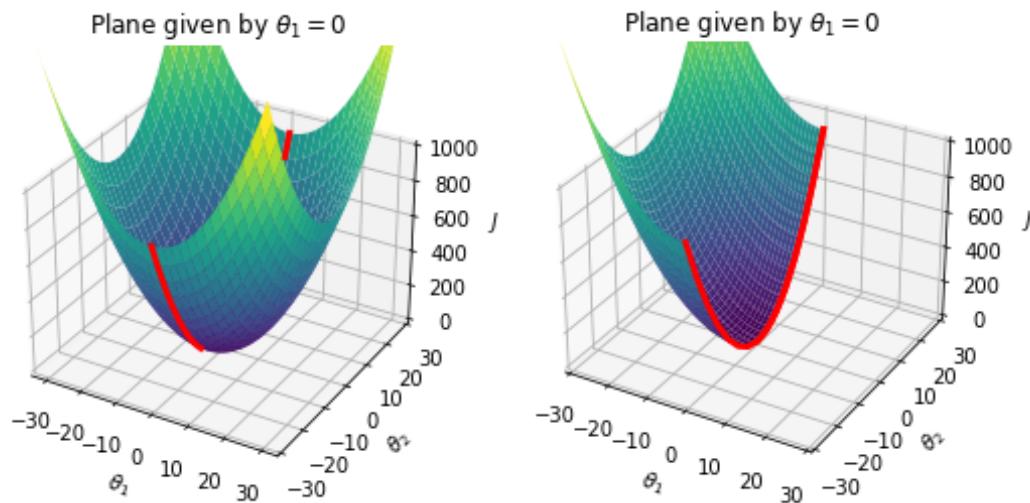
We can **reduce** the number of **variables** we have to work with, by holding some of them **constant**. That way, we have a **simpler** problem to work with.

This is **the same** as taking a single 2-D plane in a 3-D plot.



If we focus on a single plane of this surface, we end up with a **parabola**.

We can do the same the other way: we take $\theta_1 = 0$, and now we have a 1-D problem in θ_2 .



We can slice along the other axis as well!

Along each **axis**, θ_1 and θ_2 , you can **approximate** our function as **two** different straight lines. Which leads into our next point...

- Second way: if we take the two perpendicular **lines** we got from each dimension, we can combine them into a **plane**.

Concept 106

If we have **two input variables** (a 2-D problem), we can **approximate** our surface as a **plane** if we **zoom** in enough.

These **approximations** will allow us to **optimize**.

3.2.2 2-D: One dimension at a time

How do we **improve** our function J ? Now that we have **two** dimensions, we have to store our change $\Delta\theta$ in a **vector**:

$$\Delta\theta = \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \end{bmatrix} \quad (3.11)$$

This **complicates** things: we have two different things to consider **at once**.

Well, the **simplest** way would be to treat it as a **1-D** problem, and do exactly what we did **before**.

$$\Delta\theta_1 = \frac{\partial J}{\partial\theta_1} \quad (3.12)$$

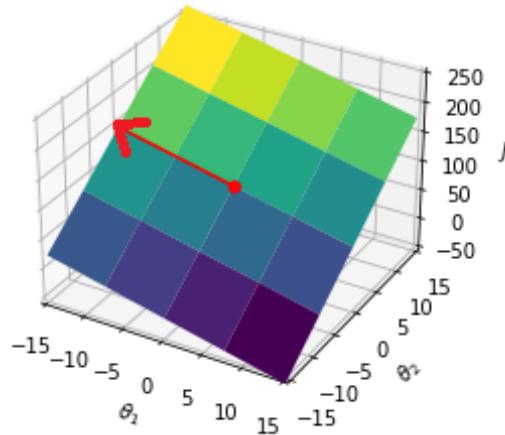
Note that we switched to **partial** derivatives, because we have **multiple** input variables θ_i .

Writing this in our **new** notation, we get:

$$\Delta\theta = -\eta \begin{bmatrix} \partial J / \partial\theta_1 \\ 0 \end{bmatrix} \quad (3.13)$$

And then we would take a **step**, moving along the θ_1 **axis**.

Movement in θ_1 on J



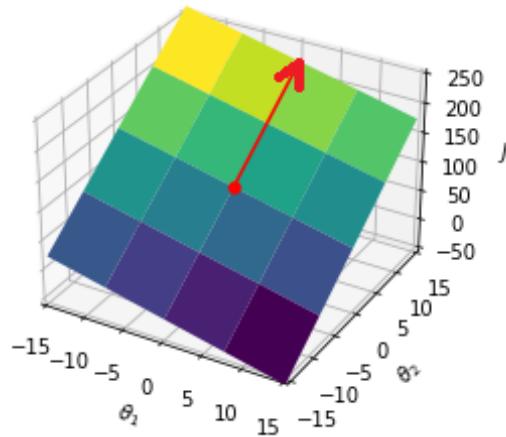
We can move along θ_1 just like on a line.

What if we treated this as a 1-D problem for the **other** variable, θ_2 ?

$$\Delta\theta = -\eta \begin{bmatrix} 0 \\ \partial J / \partial\theta_2 \end{bmatrix} \quad (3.14)$$

With this equation, we would be **moving** along the θ_2 axis.

Movement in θ_2 on J



We can do the same with θ_2 .

Why not move in **both** directions **at once**? We can **combine** our two derivatives: we'll add up our two steps.

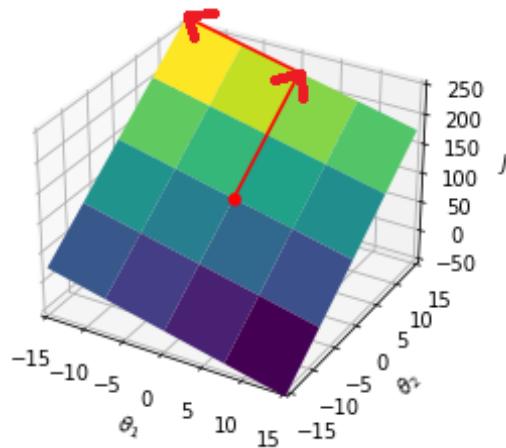
Linearity means that I can **add** them up without anything **weird** happening.

$$\Delta\theta = -\eta \begin{bmatrix} \partial J / \partial \theta_1 \\ 0 \end{bmatrix} - \eta \begin{bmatrix} 0 \\ \partial J / \partial \theta_2 \end{bmatrix} \quad (3.15)$$

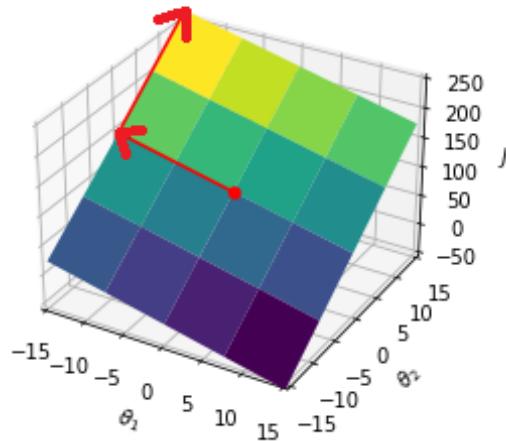
The relevant linearity rule: $L(x + y) = L(x) + L(y)$. In other words: taking two separate steps is the same as one big step.

These can be combined because we're treating our function as a **flat** plane: if I move in the θ_1 direction first, it doesn't change the θ_2 slope, and vice versa.

Combining two movements



Combining two movements



Our plane being flat means we can take both operations, back-to-back! Notice that the order doesn't matter.

$$\Delta\theta = -\eta \begin{bmatrix} \partial J / \partial \theta_1 \\ \partial J / \partial \theta_2 \end{bmatrix} \quad (3.16)$$

So, let's use that to optimize:

Key Equation 107

In **2-D**, you can optimize your function J using this rule:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \underbrace{\begin{bmatrix} \partial J / \partial \theta_1 \\ \partial J / \partial \theta_2 \end{bmatrix}}_{\text{Using } \theta_{\text{old}}}$$

This is our **gradient descent** rule for 2-D.

This sort of approach makes some **sense**: if $\frac{\partial J}{\partial \theta_1}$ is **bigger** than $\frac{\partial J}{\partial \theta_2}$, that means that you can get **more benefit** from moving in the θ_1 direction than θ_2 .

So, in that case, your step will move more in the θ_1 direction: it's a more **efficient** way to get a **better** hypothesis!

But for now, we **don't know** that this is necessarily the **optimal** way to change θ - we'll explore that later.

3.2.3 Gradient Descent in n-D

This idea can be built up in **any number** of dimensions: each variable θ_k creates a **different** line we can use to **approximate**.

And, we can combine them into a **flat** hyperplane: so, we can **add up** all of the different **derivatives**.

Key Equation 108

In **n-D**, you can optimize your function J using this rule:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \underbrace{\begin{bmatrix} \partial J / \partial \theta_1 \\ \partial J / \partial \theta_2 \\ \vdots \\ \partial J / \partial \theta_d \end{bmatrix}}_{\text{Using } \theta_{\text{old}}}$$

This is our **generalized gradient descent** rule.

3.2.4 The Gradient

We call this **gradient** descent because that right term **is** the gradient!

Definition 109

The gradient can be written as

$$\nabla_{\theta} J = \begin{bmatrix} \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \\ \vdots \\ \frac{\partial J}{\partial \theta_d} \end{bmatrix} = \frac{dJ}{d\theta}$$

So, our rule can be rewritten (for the last time) as:

Key Equation 110

The **gradient descent** rule can be generally written as:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \nabla_{\theta} J(\theta_{\text{old}})$$

θ_{old} is the input to $\nabla_{\theta} J$, not multiplication!

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta J'(\theta_{\text{old}})$$

Now, is $\nabla_{\theta} J$ the **optimal** way to improve(optimize) θ ? Let's find out.

3.2.5 The Plane Approximation

So, what is the best direction? Which way will increase/decrease J **fastest**?

Is it the gradient? Let's explore a bit to figure that out. Let's look at our plane, and see what hints it might provide: _____

For explanation purposes, we'll assume 2-D, but the explanation extends to n-D.

Concept 111

Assume your function is, at least locally, a **flat plane**.

- A **flat plane** has only **one** direction of **maximum increase**: this is the direction you might call, "directly **uphill**" if you think of elevation.
- The **opposite** direction is the direction of **maximum decrease**, or "**downhill**".
- If you move at a **right angle** to the "best" direction (maximum increase/decrease), the function **will not change**. In elevation, you stay at the **same height**!

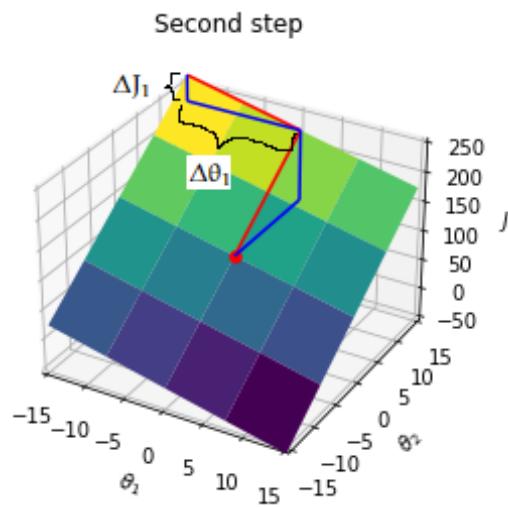
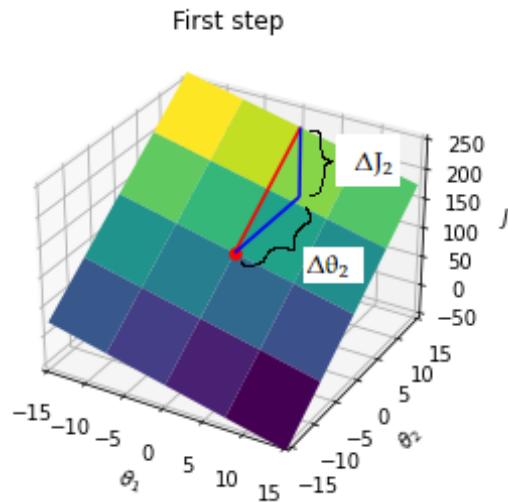
This is useful! We can **break down** any direction into the part that **affects** our function J , and the part that **doesn't**: _____

In the n-D case, we have **more** perpendicular directions. But, all of them have **no effect**!

3.2.6 The Optimal Direction: The Gradient

How do we get the optimal direction?

The **total** change in J is gotten by just **adding** the change in each direction (thank you planes!):



You can add up the results of our two steps: ΔJ_2 and ΔJ_1 .

$$\Delta J \approx \Delta J_1 + \Delta J_2 \quad (3.17)$$

Let's convert that using derivatives:

$$\Delta J \approx \Delta \theta_1 \frac{\partial J}{\partial \theta_1} + \Delta \theta_2 \frac{\partial J}{\partial \theta_2} \quad (3.18)$$

Now we've got a useful equation: the total change. As a bonus we can see a clear **pattern**

(i^{th} θ matches i^{th} derivative).

So, **condense** this pattern, like we did for our linear model: using a **dot product**.

$$\Delta J \approx \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \end{bmatrix} \cdot \begin{bmatrix} \partial J / \partial\theta_1 \\ \partial J / \partial\theta_2 \end{bmatrix} = \Delta\theta \cdot \nabla_{\theta} J \quad (3.19)$$

The **gradient** shows up! Interesting. But what does that **mean**?

Well, we want to **maximize** (or minimize!) our ΔJ . How do we maximize a **dot product**?

By making sure the directions are **the same**! So, we can confirm that the **gradient** gives us the **best** direction.

So, all we have to do is to **flip** the sign to **minimize** ΔJ .

And so, gradient descent is already complete!

Concept 112

The **gradient** ∇J is the **direction of greatest increase** for J .

That means means the opposite direction $-\nabla J$ is the **direction of greatest decrease** in J .

This is the single **most important concept** in this entire chapter!

3.2.7 Termination Condition

We can still use our termination conditions from before, but we need to be careful to make sure they extrapolate to n-D.

- Stop after a fixed T steps.
 - Nothing to change here.
- Stop when $\|\theta\|$ isn't changing much: $\|\Delta\theta\| < \epsilon$, for example.
 - We just had to replace **absolute** value with **magnitude**.
- Stop when the derivative is small: $|J'(\theta)| < \epsilon$
 - Nothing to change here.

We don't use this one often, though!

3.2.8 Another explanation of gradient (OPTIONAL)

Some students may not like the **first** explanation given for why gradient is the **direction of greatest increase**. So here, we use a slightly **different** approach, one that's more **geometric**.

Feel free to skip this section if you are not interested.

We look at a random vector, $\Delta\theta$ - no assurances about how good or bad it is.

Currently, our vector **components** are based on θ_1 and θ_2 . But, we want to **switch perspectives**.

Our vector can **also** be broken up into **parts** based on whether it **affects** J . This will let us take a **look** at the "best direction" we're trying to **find**.

- Uphill: the "best" direction \hat{u}_{best} (magnitude ΔB)
- Same height: the direction with no effect, \hat{u}_{none} (magnitude ΔN)

$$\Delta \theta = \underbrace{\Delta B \hat{u}_{\text{best}}}_{\text{Full effect on } J} + \underbrace{\Delta N \hat{u}_{\text{none}}}_{\text{No effect on } J} \quad (3.20)$$

So, all of the change in J just comes from \hat{u}_{best} . We **don't care** about the other direction!

Concept 113

In a **local planar approximation**, the **only** component of $\Delta \theta$ that **affects** J is the **direction of greatest increase**, \hat{u}_{best} .

So, we can determine ΔJ using **only that component**.

But, $\nabla_{\theta} J$ **also** gives us change in ΔJ : it contains all of the **derivatives**, and thus gives us the effect θ has on ΔJ .

So we can find:

$$\Delta \theta \cdot \frac{dJ}{d\theta} = \Delta J = (\Delta \theta \cdot \hat{u}_{\text{best}}) \frac{dJ}{dB} \quad (3.21)$$

$\Delta \theta$ is being dotted with both the **gradient** and the **best direction**: they both give the change in J . To keep the **dot product** consistent as $\Delta \theta$ changes, they need to be in the **same direction**.

3.3 Application to Regression

One nice thing about **gradient descent** is that it is **easy** to switch the kind of problem you're applying it to: all you need is your **parameters**(s) θ , and a function to optimize, J .

From there, you can just **compute** the gradient.

3.3.1 Ordinary Least Squares

Our **loss** function is

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n \left((\theta^T x^{(i)} + \theta_0) - y^{(i)} \right)^2 \quad (3.22)$$

Or, in **matrix** terms,

Including the appended row of 1's from before.

$$J = \frac{1}{n} (\tilde{\mathbf{X}}\theta - \tilde{\mathbf{Y}})^T (\tilde{\mathbf{X}}\theta - \tilde{\mathbf{Y}})$$

Our gradient, according to **matrix derivative** rules, is

$$\nabla_{\theta} J(\theta) = \frac{2}{n} \tilde{\mathbf{X}}^T (\tilde{\mathbf{X}}\theta - \tilde{\mathbf{Y}}) \quad (3.23)$$

Before, we set it equal to **zero**. But here, we can instead take **steps** towards the solution, using **gradient descent**.

We could use the **matrix** form, but sometimes it's easier to use a **sum**. Fortunately, derivatives are easy with a sum. If so, here's **another** way to write it:

$$\nabla_{\theta} J(\theta) = \frac{2}{n} \sum_{i=1}^n (\theta^T x^{(i)} - y^{(i)}) x^{(i)} \quad (3.24)$$

Either way, we use gradient descent **normally**:

Remember that θ_{old} is an **input** to the gradient, not multiplied by it!

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \nabla_{\theta} J(\theta_{\text{old}})$$

Using $\theta^{(t)}$ notation:

$$\theta^{(t)} = \theta^{(t-1)} - \eta \nabla_{\theta} J(\theta^{(t-1)})$$

3.3.2 Ridge Regression

Ridge regression is similar.

$$J(\theta) = \frac{1}{n} \sum_{i=1}^n \left(\underbrace{(\theta^T x^{(i)} + \theta_0)}_{\text{guess}} - \underbrace{y^{(i)}}_{\text{answer}} \right)^2 + \underbrace{\lambda \|\theta\|^2}_{\text{Regularizer}}$$

However, we have to treat θ_0 as **separate** from our other data points, because of **regularization**: remember that it **doesn't** apply to θ_0 .

For θ :

$$\nabla_{\theta} J_{\text{ridge}}(\theta, \theta_0) = \frac{2}{n} \sum_{i=1}^n \left((\theta^T x^{(i)} + \theta_0) - y^{(i)} \right) x^{(i)} + 2\lambda\theta \quad (3.25)$$

For θ_0 :

$$\frac{\partial J_{\text{ridge}}(\theta, \theta_0)}{\partial \theta_0} = \frac{2}{n} \sum_{i=1}^n \left((\theta^T x^{(i)} + \theta_0) - y^{(i)} \right) \quad (3.26)$$

Notice that we used a **gradient** for our vector θ , but since θ_0 is a single variable, we just used a **simple derivative**!

Concept 114

The **gradient** $\frac{dJ}{d\theta}$ must have the **same shape as θ** : this shape-matching is why we can easily **subtract** it during gradient descent.

$$\underbrace{\theta_{\text{new}}}_{(d \times 1)} = \underbrace{\theta_{\text{old}}}_{(d \times 1)} - \eta \underbrace{\nabla_{\theta} J(\theta_{\text{old}})}_{(d \times 1)}$$

3.3.3 Computational Gradient

Sometimes, we **can't** easily find the **equation** for our gradient: maybe our loss isn't a simple **equation**, or we have some **other** kind of problem. So, rather than getting the **exact** gradient, we **approximate** it.

But how do we **approximate** the gradient? Well, first, we could **reference** how we approximate a **simple derivative**.

The definition of the **derivative** can be gotten as

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \quad (3.27)$$

But, what if we **can't** take the **limit**? Or, we just don't **want** to?

We can **approximate** by taking h to be a small, **finite** number.

Instead of h , we'll call this δ .

A derivative is just a 1-D gradient, after all!

Concept 115

When **approximating** the derivative, we can choose a **small** finite width to measure, called δ , so that

$$\frac{df}{dx} \approx \frac{f(x + \delta) - f(x)}{\delta}, \quad \delta \ll 1 \quad (3.28)$$

So, let's **extend** that to the **gradient**:

$$\nabla_{\theta} J = \begin{bmatrix} \partial J / \partial \theta_1 \\ \partial J / \partial \theta_2 \\ \vdots \\ \partial J / \partial \theta_d \end{bmatrix} \quad (3.29)$$

Luckily, the **gradient** is just a bunch of derivatives **stacked in a vector!**

So, we can just **compute** each of them **separately**, and then put them together.

Let's show how we'd **write** that in **vector** form, for just one of them. We want something like

$$J'(\theta) \approx \underbrace{\frac{J(\theta + \delta) - f(\theta)}{\delta}}_{\text{Not correct, but closer}} \quad (3.30)$$

This isn't quite right, because a **scalar** δ would **add to every term**.

We **only** want to shift **one** variable at a time, so we can do a **simple** derivative.

Let's say we want $dJ/d\theta_1$. We would **only** want to add δ to θ_1 : the other parameters are **unchanged**.

So, we **can't** add a **scalar**. Instead, we need a $(d \times 1)$ vector: one term to **separately** add to each θ_k term.

$$\Delta \theta = \begin{bmatrix} \Delta \theta_1 \\ \Delta \theta_2 \\ \vdots \\ \Delta \theta_d \end{bmatrix} \quad (3.31)$$

We want most terms **unchanged**, so we'll **add 0** to each of them, and we'll add δ to the one term we want to **edit**.

$$\Delta\theta = \begin{bmatrix} \delta \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \quad (3.32)$$

We'll **create** one of these vectors for each **dimension**. We'll give them a special **name**: δ_k , for the k^{th} dimension.

$$\delta_1 = \begin{bmatrix} \delta \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \quad \delta_2 = \begin{bmatrix} 0 \\ \delta \\ \vdots \\ 0 \\ 0 \end{bmatrix} \quad \delta_{d-1} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \delta \\ 0 \end{bmatrix} \quad \delta_d = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \delta \end{bmatrix} \quad (3.33)$$

Finally, we'll **divide** by δ . We have what we need for our full equation:

Key Equation 116

In order to **computationally find the gradient**, you need to find the **partial derivative** for each term θ_k .

$$\frac{dJ}{d\theta_k} \approx \frac{J(\theta + \delta_k) - f(\theta)}{\delta}$$

Where

- δ is a small positive number
- δ_k is the $(d \times 1)$ **column vector** with a δ in the k^{th} row, and a 0 in every other row.

3.3.4 Problems with Gradient Descent

Gradient descent is very handy, but it's important to be aware of some of its **problems**.

We've discussed a couple: diverging, oscillating, and converging slowly. We also have to worry about **local minima** that aren't as good as other answers.

But there's also a **limitation**: our loss function has to be **smooth** and **differentiable**. If it isn't, we can't take the **gradient** of it.

Concept 117

Gradient descent requires for your **functions** to be (at least mostly) **smooth and differentiable**.

Our **answer** is also only as good as our **loss function**: if our loss function is not good for what we actually want to **accomplish**, then we can easily create a **bad** model.

3.4 Stochastic Gradient Descent

3.4.1 Another problem with gradient descent

One **advantage** we mentioned for gradient descent at the beginning of the chapter is that we can **stop early** if we think we're done, saving on **time**.

This is helpful for really **large** datasets, and it's also more **computationally manageable** than inverting a gigantic XX^T matrix.

But, we **haven't** taken **full advantage** of it: above, we used a **sum** to get our gradient - we're assuming we'll use all of our **data points**.

But, what if we don't want to have to use all of them at once? That might be **expensive**.

And in fact, there are **other** reasons not to: maybe the gradient will change directions after only a **small** distance.

Right now, we're getting **lots of data** before even taking a **single step**: if we start moving **immediately**, our program could **adapt** to the "terrain" more quickly!

3.4.2 A better way: stochastic GD

Instead, why wait until we have **added** up over all the data? We could just **compute** the gradient over **one** data point **at a time**. In fact, to be fair, we'll do it **randomly**.

But wait, this **seems** like it would be **less** effective - after all, how much does **one** data point tell you?

To compensate, our steps will have to be **smaller**!

Well, even if it isn't much, this isn't very **different** from adding them up all at **once**: in **theory**, taking lots of **little** steps should average out to the **same** information as if we do it all at once.

Definition 118

Stochastic Gradient Descent (SGD) is the process of applying **gradient descent** on **randomly** selected data points.

This should **average** out to being **similar** to regular (batch) gradient descent, but the **randomness** often lets it improve **faster** and **avoid** some common problems.

There are more possible benefits, too: **randomly** choosing data points adds some **noise**, and random movement might be able to pull us out of local minima we don't want.

Stochastic is just a very mathematically precise word for "random".

This sort of **noise** and **randomness** can make it hard for our model to **perfectly** fit the training data: this can reduce **overfitting**, too!

We mean "noise" in the signals sense: random **variation** in our data that **isn't** part of what we're trying to pay attention to: in this case, the **distribution**.

For example, it's hard to focus on the details of someone's eyelashes (unimportant details) if your vision is blurry.

Concept 119

There are many **benefits** to **SGD** (Stochastic Gradient Descent) over regular BGD (Batch Gradient Descent).

- SGD can sometimes **learn** a good model **without** using all of our **data**, which can **save us time** when data sets are **too large**.
 - It can also let us address problems **early** if the model **isn't** improving.
- The noise produced by the random sampling in SGD can sometimes help it **avoid local minima** that aren't very good models,
 - This is because the model might be moved in a **random direction** in **parameter space**, and randomly **pulled out** of that minimum, even if BGD would have gotten **stuck**.
- The noise also **reduces overfitting**, because it's **harder** for the model to **memorize** the exact details of the **distribution**.

3.4.3 Ensuring Convergence

How do we make sure that our SGD method converges? We need some kind of termination criteria. Thankfully, there's a useful theorem on the matter:

Theorem 120

SGD **converges** with *probability one* to the **optimal** Θ if

- f is convex

And our step size(learning rule) $\eta(t)$ follows these rules:

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

Why these rules? Let's see:

- The **first** rule is for the **same** reason as for regular BGD: if it isn't **convex**, we can get stuck in **local minima**, or if it's **concave**, decrease **forever**.
- The **second** rule means that your steps need to add up to an **infinite distance**: this allows you to reach **any** possible point in your **parameter space**.
- The **third** one is a bit **trickier**, but basically means the steps need to get **smaller**, so we can approach the **minimum** (otherwise we might **diverge**!)

One option is $\eta(t) = 1/t$. But often, we use rules that **decrease more slowly**, so that it doesn't take as **long**.

But technically, we're no longer guaranteed convergence!

3.5 Terms

- Gradient Descent
- Parameter Space
- Local view (Calculus)
- Linear Function Approximation
- Planar Function Approximations
- Convergence
- Divergence
- Oscillation
- Step size
- Termination Condition
- Concavity/Convexity
- Global Minimum
- Local Minimum
- Initialization
- Gradient (Direction of Maximum Increase)
- Gradient Descent Rule
- Gradient Shape
- Gradient Approximation
- Stochastic Gradient Descent
- Batch Gradient Descent
- BGD Convergence Theorem
- SGD Convergence Theorem

CHAPTER 4

Classification

4.0.1 Regression (Review)

In chapter 2, we handled the problem of **Regression**: taking in lots of data (stored as a **vector of real numbers**), and returning another single **real number**.

Remember that we used a $(d \times 1)$ column vector for our data points $x^{(i)}$.

$$h_{reg} : \mathbb{R}^d \rightarrow \mathbb{R} \quad (4.1)$$

This was good for when we wanted to predict some **numeric** output: stock prices, height, life expectancy, and so on.

But, this isn't the **only** type of problem we might have to deal with.

4.1 Classification

4.1.1 Motivation: Putting things into classes

We don't *always* want a **real number** output: that can be complicated, or not match some problems well.

Often, it's more useful to, rather than give exact values, sort things into **categories**, or what we will call **classes**.

Definition 121

A **class** is **set** of things that have something relevant in **common**.

Example: A beagle and a golden retriever could both be put in a **class** called "dog". This is useful if you just want to know whether you have a dog or not!

4.1.2 What is classification?

This is the goal of **classification**: we want to take lots of **information**, and use them to **predict** what **class** a data point belongs in.

Definition 122

Classification is the **machine learning problem** of sorting items into different, **discrete** classes.

In this setting, we take **real-valued data**, stored in a $(d \times 1)$ **vector**, and return one of our **classes**.

$$h : \mathbb{R}^d \rightarrow \{C_1, C_2, C_3, \dots, C_n\}$$

Where $\{C_1, C_2, C_3, \dots, C_n\}$ are all **classes**. Sometimes, we call the value we return a **label** instead.

Example: You might classify different **animals** as a bird, a mammal, or a fish. You have 5 pieces of useful data to **classify** with.

As a refresher, the function notation here just says, "take in a d -dimensional vector, and output one of our n discrete classes."

$$h : \mathbb{R}^5 \rightarrow \{\text{Bird, Mammal, Fish}\} \quad (4.2)$$

Classification can be useful for lots of situations:

- **Deciding** which **action** to take in a difficult situation
- **Diagnosing** a patient, and determining the best **treatment**
- **Sort** information to be **processed** later
- And more!

Just like with regression, we can depict our **hypothesis** as the function

$$x \rightarrow [h] \rightarrow y \quad (4.3)$$

Concept 123

Classification is also **supervised**: meaning, you have **training** data \mathcal{D}_n with the **correct** answers given:

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

In **unsupervised** problems, you're not told the "correct" answer and have to just guess one!

4.1.3 Important Facts about Classes

There's a few important things we should remember about classes moving forward.

- Classes are **discrete**: each class is a distinct "thing", **separate** from other classes.
 - This is unlike real numbers, which are **continuous**: you can **smoothly** transition between them.
- This isn't **always** true, but usually, classes are **finite**: there are only so many of them, which we write as n .
 - Meanwhile, there are **infinitely many** real numbers.
- These classes may not have a natural **order**: is there a correct way to order "[Bird, Mammal, Fish]"? Not really.
 - The real numbers are ordered, too.
- In some problems, you get to **decide** what classes are acceptable: where do you draw the **line** between two categories? Do you care about dogs, or just mammals? And so on.
 - You can change units, but the **structure** of the real numbers is very **consistent**.

Concept 124

Classes are

- **discrete**
- **finite** (usually)
- **not necessarily ordered**
- often **defined** based on your **needs**

4.1.4 Binary Classification

So, how do we get **started**? Well, we want to create the **simplest** case, and maybe we can get the **general** idea.

Two is the **smallest** number of useful classes: often, this boils down to a **yes-or-no** question. Typically, we **represent** these two choices as $+1$ and -1 , respectively.

Definition 125

Binary classification is the **problem** of sorting elements into one of **two categories**.

Often, these categories are defined by a "**yes-or-no**" question.

$$h : \mathbb{R}^d \rightarrow \{-1, +1\}$$

Example: You could look at a person and say, "are they sick?" or, "is that a dog"? You can **classify** data in a binary way **based** on those questions.

4.1.5 Classification Performance

And how do we measure how well this model is doing? The easiest way might be, "count the number of wrong guesses".

This is captured by **0-1 Loss**:

Definition 126

0-1 Loss is a way of measuring **classification** performance: if you get the **wrong** answer, you get a loss of **1**. If you're **right**, then **0** loss.

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

This type of loss is as **simple** as we can get: similar to counting how many wrong answers you get on a **multiple-choice** test.

If we want to get our training error, we'll just average over the data points:

$$\mathcal{E}_n(h) = \frac{1}{n} \sum_{i=1}^n \begin{cases} 0 & \text{if } g_i = a_i \\ 1 & \text{otherwise} \end{cases}$$

Just like before, we care about **testing loss** more than **training loss**: we want our model to **generalize**.

This relies on our typical IID assumption from chapter 1.

$$\mathcal{E}(h) = \frac{1}{m} \sum_{i=n+1}^{n+m} \begin{cases} 0 & \text{if } g = a \\ 1 & \text{otherwise} \end{cases}$$

Next, we figure out what **model** we use to do our classification.

4.2 Linear Classifiers

If you wanted to break up your data into two parts (+1 and -1), how might you do it? Let's explore that question.

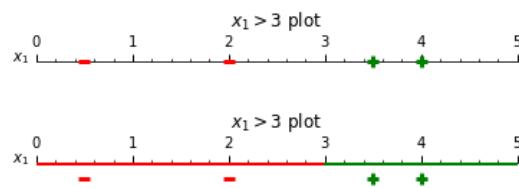
4.2.1 1-D Linear Classifiers

As usual, we'll start with the **simplest** case we can think of: 1-D. So, we only have one variable x_1 to **classify** with.

The simplest version might be to just **split** our space in **half**: those above or below a certain **value**. This is our parameter, C .

$$x_1 > C \quad \text{or} \quad x_1 - C > 0 \quad (4.4)$$

Example: For the below data (where green gives positive and red gives negative), could classify positive as $x_1 > 3$.



We plot everything above $x = 3$ as **positive**, and **negative** otherwise.

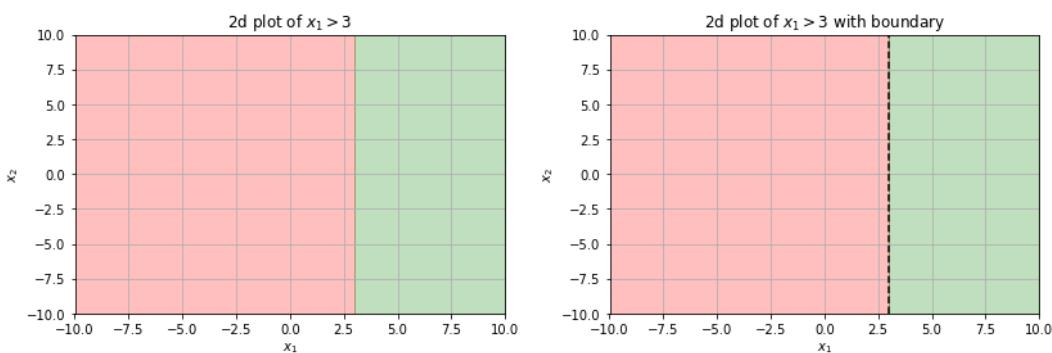
We could also call it θ_0 , in the spirit of our θ notation for parameters.

$$x_1 + \theta_0 > 0 \quad (4.5)$$

4.2.2 1-D classifiers in 2-D

Let's add a variable and see how our classifier looks on a 2-D plot.

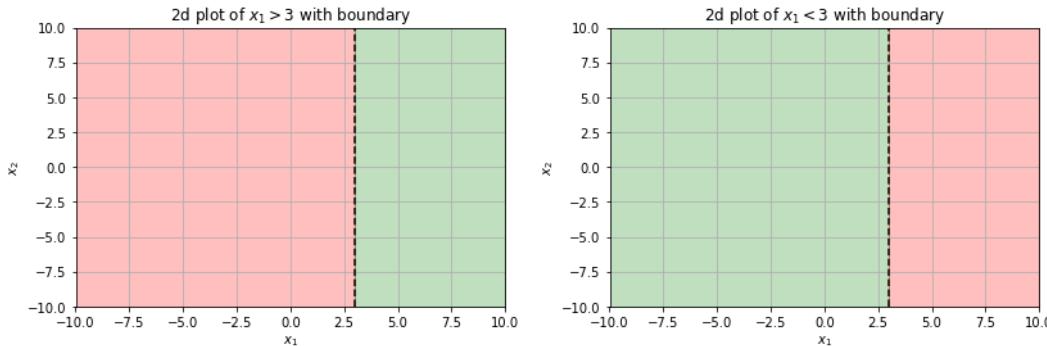
We'll omit the data points for now.



On the right, we've drawn the **dividing** line between our two regions.

Interesting - the **boundary** between positive and negative is defined by a **vertical line**.

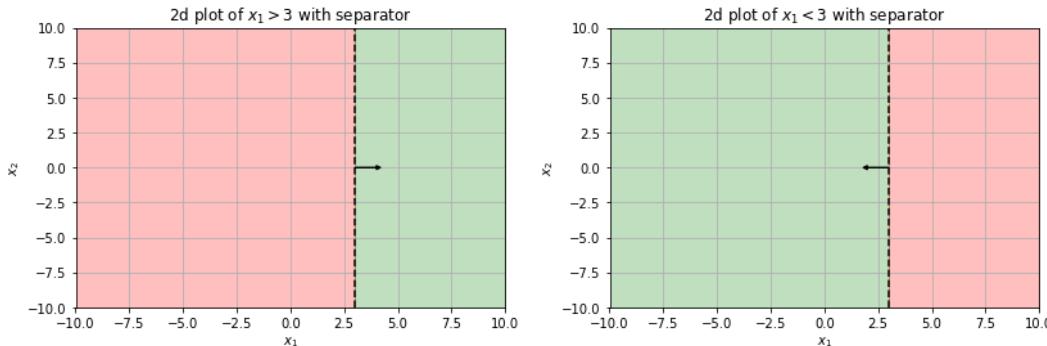
Or, almost. Compare $x_1 > 3$ and $x_1 < 3$:



These two plots have the same line, but have their sides flipped.

So, we have a **line** that gives us the boundary, but we **also** need to include information about which way is the **positive** direction.

What tool best represents **direction**? We could use angles, but we haven't used that much so far. Instead, let's use a **vector** to **point** in the right direction.



Now, it's clear which plot is which, just using our **line** and **vector**!

The object that represents our classification is called a **separator**!

Since our variables are x_1 and x_2 , this is a separator in **input space**.

Definition 127

A **separator** defines how we **separate** two different classes with our **hypothesis**.

It includes

- The **boundary**: the **surface** where we **switch** from one **class** to another.
- The **orientation**: a **description** of which **side** of the boundary is assigned to **which class**.

For example, let's take our specific separator from above.

Concept 128

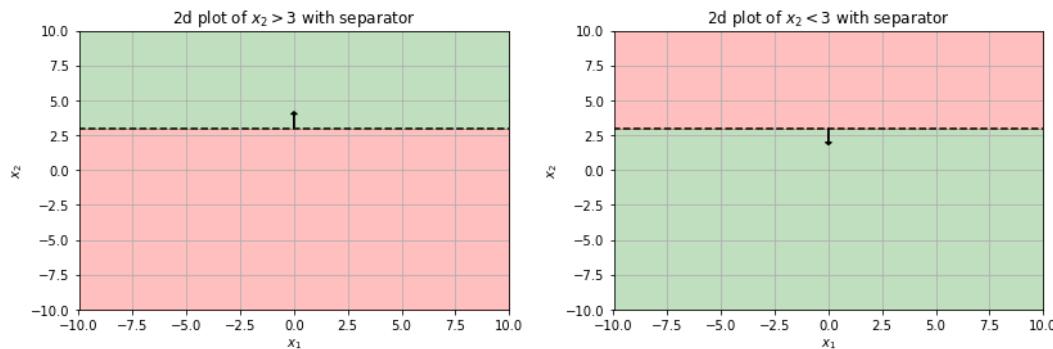
We can define our **1-D separator** using

- The **boundary** between the **positive** and **negative** regions: in 2-D input space, this looks like a vertical or horizontal **line**.
- A **vector** pointing towards whichever side is given a **+1 value**.

We call it "orientation" because you could imagine "flipping over" the space, so the positive and negative regions are swapped.

4.2.3 A second 1-D separator, and our problem

What if we use x_2 to **separate** our data?

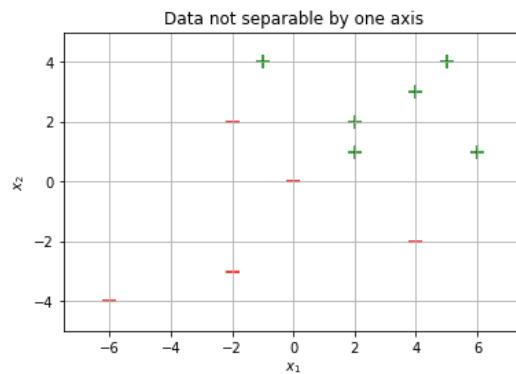


Instead of having a vertical separator, we have a **horizontal** one.

We get the same sort of plot along the **other axis**!

So, this is cool so far, but it's not a very **powerful** model: we can only handle a situation where the data is evenly divided by **one axis**.

And if that's the case, what's the point of our **other** variable?



There's no vertical or horizontal line we can use to split this space!

4.2.4 The 2-D Separator: What vector do we use?

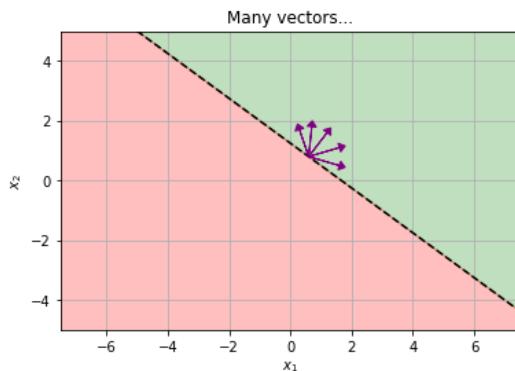
Just looking at our example, we might wonder, "well, if we can use **vertical** lines or **horizontal** lines, can't we just use a line in **another** orientation?"

It turns out, we **can**!



If we allow lines at an angle, we can classify all of our data correctly!

So, we've got our **boundary**. But we still need a vector to tell us which side is **positive**. But there are **many** possible vectors we could choose:



All of these vectors point towards the **correct** side of the plane. Is there a **best** one to use?

Above, we used the vector that was **vertical** or **horizontal**. This makes sense: if we're doing $x_1 > 3$, it seems reasonable to have the arrow **point** in the positive- x_1 direction.

But this vector also happened to be **perpendicular** to our **line**: this is the line's **normal vector**, \hat{n} . This vector has a couple nice properties:

- It is **unique**: in 2-D, there is only 1 **normal** direction. _____
- It points directly **away** from the plane. _____
- If our plane is at the **origin**, any point with a **positive** \hat{n} component is on the **positive** side. _____

The opposite side is just $-\hat{n}$.

This will be important later!

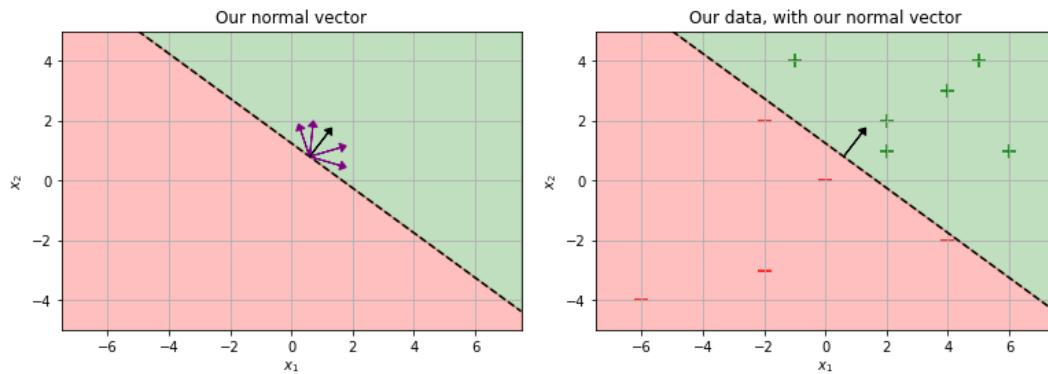
So, we have a **unique** vector that tells us which side is **positive**. Let's go with that!

Concept 129

Every **line** in 2-D has a **unique normal vector** that can be used to **define** the **angle/direction** of the line.

The **direction** the vector is "facing" is also called the **orientation**.

Our normal vector for the above separator:



We can define our plane using the **normal vector**!

It's clear that this vector in some way is a **parameter**: if we change this vector, we get a different **orientation**, and a different **classifier**.

We have **represented** parameters in the past using θ . We need **two** different θ_k : one for the x_1 component, another for the x_2 component.

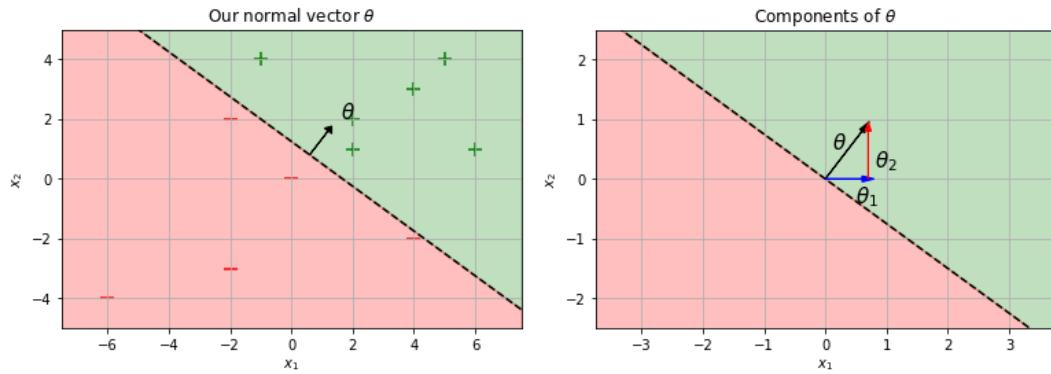
So, we'll use that.

Notation 130

The vector θ represents the **normal vector** to our line in 2D.

$$\hat{\theta} = \theta = \begin{bmatrix} \theta_1 \\ \theta_2 \end{bmatrix}$$

We add this to our diagram:

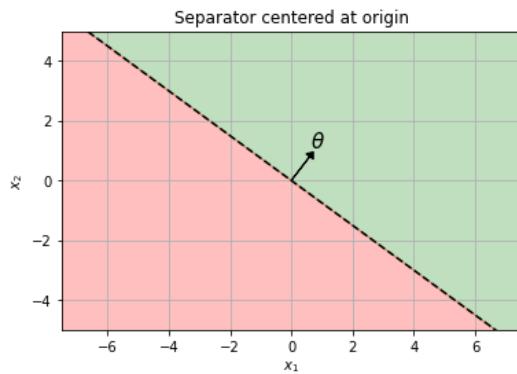


θ is our normal vector!

Nice work so far. The next question is: how do we describe this separator **mathematically**?

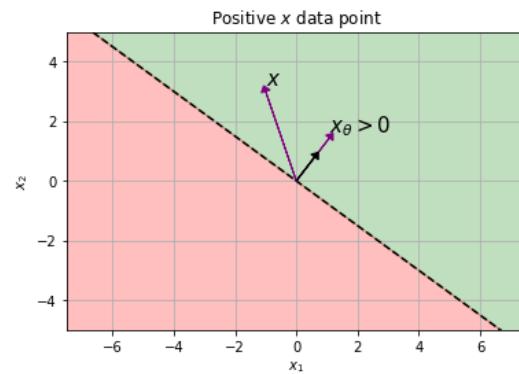
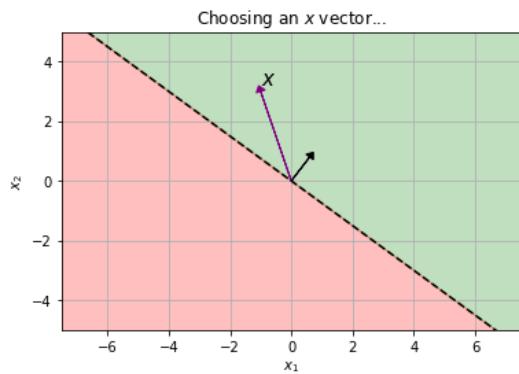
4.2.5 2D Separator - Matching components

As always, we'll **simplify** the problem to make it more manageable: for now, we'll assume our **separator** is centered at the **origin**.

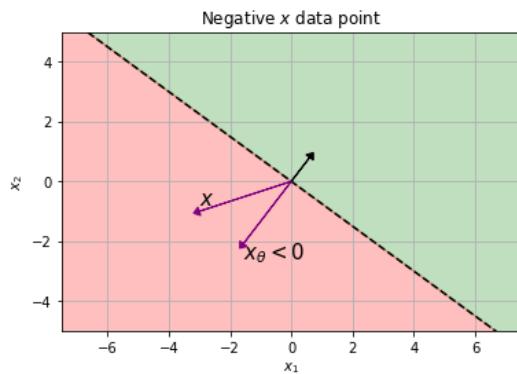


So, we have our vector, \hat{n} . As we mentioned above, anything on the **same** side as \hat{n} is **positive**, and anything on the **opposite** side is **negative**.

For a line on the origin, "On the same side of the line" can be interpreted as "has a positive \hat{n} component". We'll find that component next.



This vector has a **positive** component in the θ direction.



This vector has a **negative** component in the θ direction.

How do we represent "on the same side" mathematically? How do we **find** whether the component is **positive or negative**? We use the **dot product**.

4.2.6 The Dot Product (Review)

How to calculate the dot product should be familiar to you, but we'll talk about some **intuition** that you may not be exposed to.

Concept 131

You can use the **dot product** between unit vectors to measure their "similarity": if two vectors are more **similar**, they have a **larger** dot product.

In the most clear cases, take unit vectors \hat{a} and \hat{b} :

- If they are in the **exact same** direction, $\hat{a} \cdot \hat{b} = 1$
- If they are in the **exact opposite** direction, $\hat{a} \cdot \hat{b} = -1$
- If they are **perpendicular** to each other, $\hat{a} \cdot \hat{b} = 0$

Remember, **unit vectors** have a length of 1.

What about non-unit vectors?

These unit vectors are then scaled up by the **magnitude** of each of our vectors. Because magnitudes are **always positive**, the dot product sign doesn't change.

Concept 132

You can use the **dot product** between non-unit vectors to measure their "similarity" **scaled by their magnitude**.

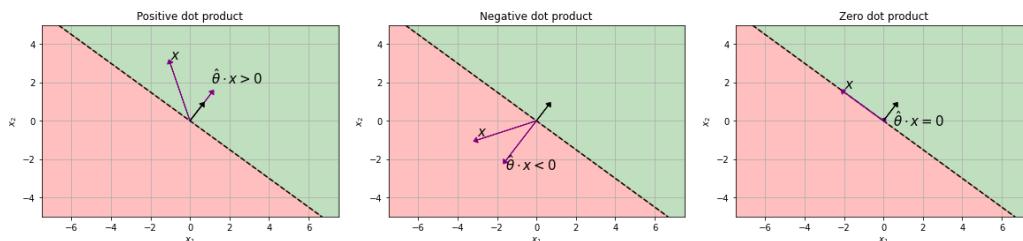
If two vectors are more **similar**, they have a **larger** dot product.

- If the vectors are **less** than 90° apart, they are more similar: they will share a **positive** component: $\vec{a} \cdot \vec{b} > 0$
- If the vectors are **more** than 90° apart, they will share a **negative** component: $\vec{a} \cdot \vec{b} < 0$
- If they are **perpendicular** (90°) to each other, $\vec{a} \cdot \vec{b} = 0$

4.2.7 Using the dot product

So, the **sign** of the dot product is a useful tool. If a point is on the line, it is **perpendicular** to θ , our **normal vector**.

So, if a point has a **positive** dot product, it is on the **same side** as θ , and if it's **negative**, it's on the **opposite side**.



Our various dot products can show us where in the space we are.

So, we can classify things based on the **sign** of it. Written as an equation, we can define the sign function:

Key Equation 133

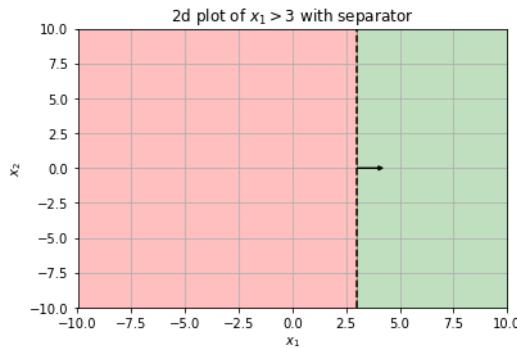
For a **linear separator** centered on the **origin**, we can do **binary classification** using the hypothesis

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

4.2.8 Introducing our offset

Now that we have handled the case where our linear separator is on the **origin**, we want to **shift** our separator **away** from it.

In our **1-D** case, we easily **shifted** away from the origin: any separator $x_1 > C$ where C **isn't zero**, we shift by C units.



By making our inequality $x_1 > 3$ **nonzero**, we moved away from the origin by 3 units!

We could make our inequality **nonzero**, then! That could move us **away** from the origin, just in a different **direction**.

Or, we could equivalently do this... Note: $A \iff B$ means A and B are equivalent!

$$x_1 > 3 \iff x_1 - 3 > 0 \tag{4.6}$$

So, instead, we could just add a constant to our expression, which we will call θ_0 .

We'll also switch out $\theta \cdot x = \theta^T x$.

Key Equation 134

A general **linear separator** can do **binary classification** using the hypothesis

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

Notice that this looks very similar to what we did in regression! We'll get into that in a bit.

First, a quick look at the components of our equation:

Concept 135

For **binary classification**, θ and θ_0 entirely **define** our **linear separator**.

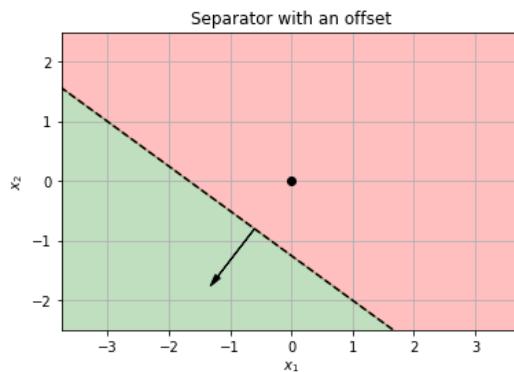
- θ gives us the **orientation** of our line.
- θ_0 **shifts** that line around in **space**.

4.2.9 How does the offset affect our classifier?

So, how exactly does our offset θ_0 affect our **classifier**? Well, we mark our classifier with our **normal vector** and the **boundary**.

Our **normal vector** is entirely captured by θ : it's unchanged by θ_0 .

What about our **boundary**? We have its **orientation**, but we don't know where it has shifted to.



Note that the origin has been marked.

Well, let's use our equation: the **boundary** line is given by

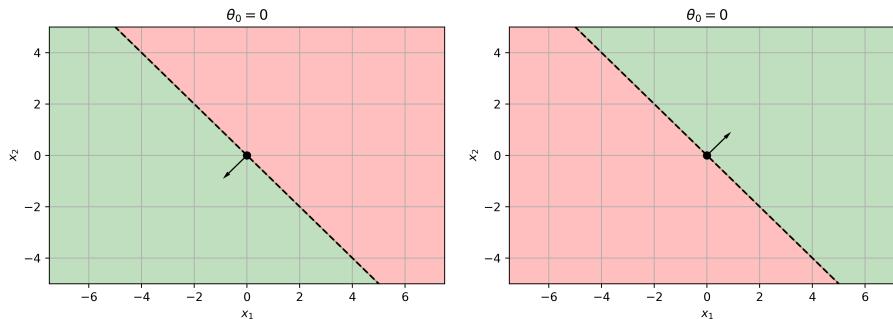
$$\theta^T x + \theta_0 = 0 \iff \theta^T x = -\theta_0 \quad (4.7)$$

We'll break the effects of θ_0 into three cases: _____

For each, we'll show two different θ values.

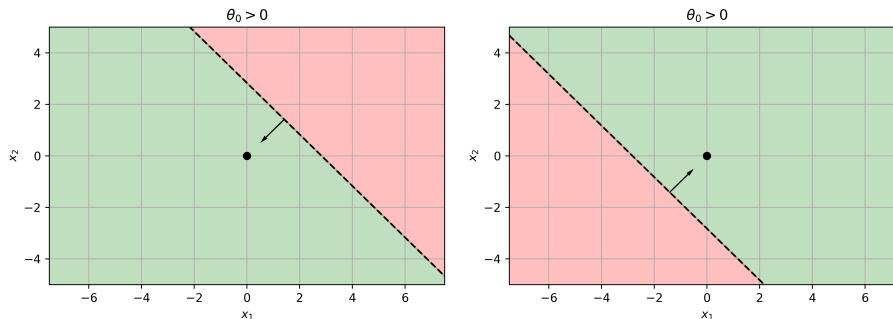
Note: the below statements are true no matter what θ we choose!

- If $\theta_0 = 0$, then $x = (0, 0)$ is **on the line**.
 - Without an **offset**, our line goes through the **origin**.



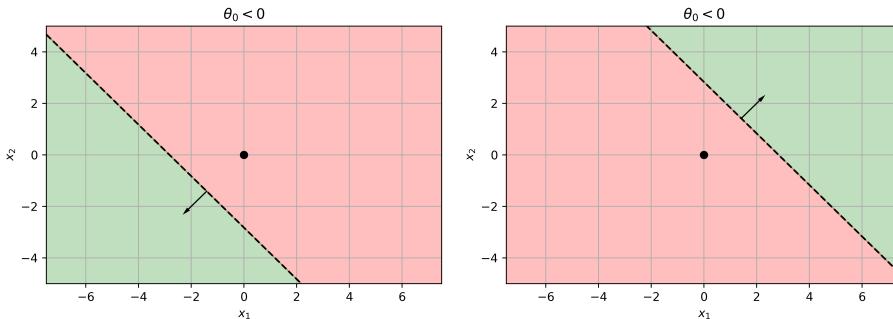
The boundary is on the origin.

- If $\theta_0 > 0$, then $x = (0, 0)$ is in the **positive** region.
 - That means the positive region is **larger**: the line must have moved in the -0 direction.



If we have a **positive** constant, it's "easier" to get a positive **result**: more positive space.

- If $\theta_0 < 0$, then $x = (0, 0)$ is in the **negative** region.
 - That means the positive region is **smaller**: the line must have moved in the $+0$ direction.



If we have a **negative** constant, it's "harder" to get a positive **result**: more negative space.

This can be a bit confusing, so we'll summarize:

Concept 136

The **sign** of our θ_0 and the **direction** we move away from the origin are **opposite**.

If $\theta_0 > 0$ (positive), our boundary moves in the **$-\theta$ direction**.

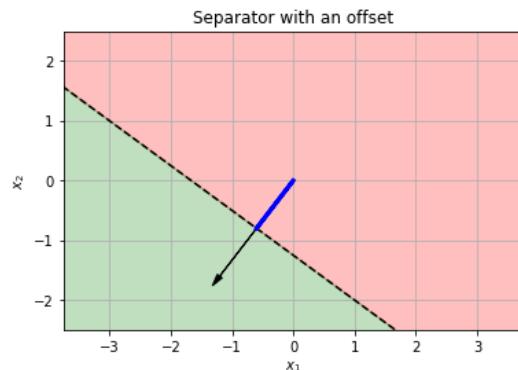
If $\theta_0 < 0$ (negative), our boundary moves in the **$+\theta$ direction**.

This gives us a general idea of how the offset affects it, but what is the **exact** effect of θ_0 on the line?

We'll focus on one point on the line: the **closest point to the origin**. We want to look at this point because it's **unique**.

Points that aren't unique are hard to keep track of!

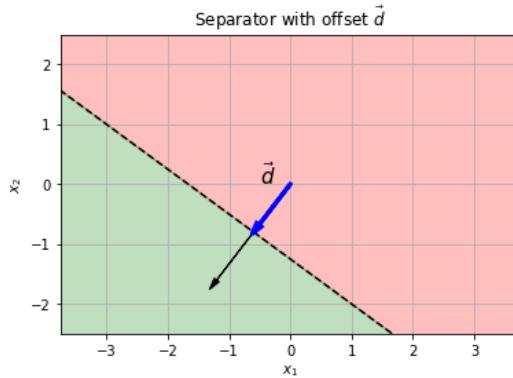
4.2.10 Distance from the Origin to the Plane



Notice that the **shortest** path from the origin to the line is **parallel** to θ !

So, we can think of our **line** as having been **pushed** in the θ direction. This **matches** what we did for 1-D separators: $x_1 > 3$ was moved in the x_1 direction.

So, we'll take the closest point on the line, \vec{d} . The **magnitude** d will give us the **distance** that the separator has been **shifted**.



Since \vec{d} is in the direction of θ , the direction can be captured by the unit vector $\hat{\theta}$. Let's take a look at that:

$$\theta = \|\theta\| \hat{\theta} \quad (4.8)$$

Remember, a vector is direction (unit vector) times magnitude (scalar).

$$\vec{d} = d \hat{\theta} \quad (4.9)$$

They're in the same **direction**, so they have the same **unit vector** $\hat{\theta}$.

\vec{d} is on the **line**, so it satisfies:

We'll use $\theta \cdot \vec{d}$ instead of $\theta^T \vec{d}$ here.

$$\theta \cdot \vec{d} + \theta_0 = 0 \quad (4.10)$$

We can plug our equations 4.8 and 4.9, where we've separated magnitude from unit vector:

$$\underbrace{(\|\theta\| \hat{\theta})}_{\theta} \cdot \underbrace{(d \hat{\theta})}_{\vec{d}} + \theta_0 \quad (4.11)$$

We can move the scalars $\|\theta\|$ and d out of the way of the dot product:

$$\|\theta\| d (\hat{\theta} \cdot \hat{\theta}) + \theta_0 \quad (4.12)$$

We know that $\hat{\theta} \cdot \hat{\theta} = 1$:

$$\|\theta\| d + \theta_0 = 0 \quad (4.13)$$

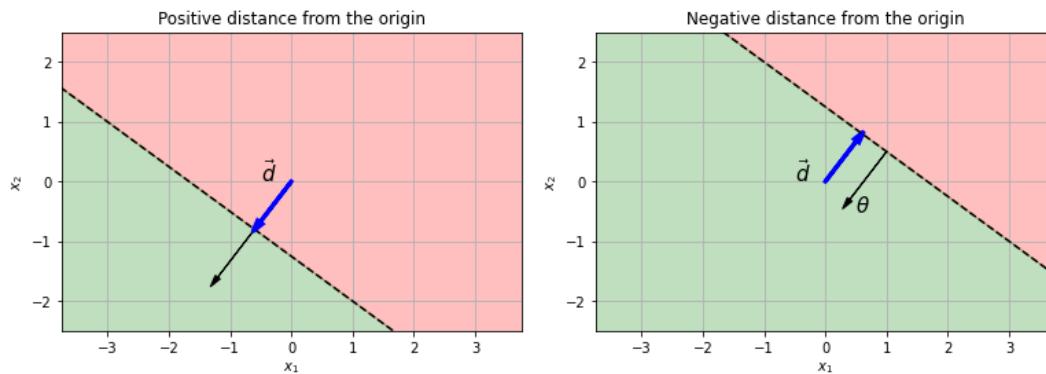
And now, we just solve for d :

Concept 137

The **distance** d from the **origin** to our **linear separator** is

$$d = \frac{-\theta_0}{\|\theta\|} \quad (4.14)$$

A "negative" distance means \vec{d} (the vector from the origin to the line) is pointed in the opposite direction of θ .



Notice, again, that this agrees with our **earlier** thought: the sign of θ_0 is the opposite (-1) of the θ direction we move in.

4.2.11 Extending to higher dimensions

We've now fully conquered the 2D problem! Now, we can move up in **dimensions**.

In terms of equations, the answer is simple, just like it is for regression: just add more terms to θ .

Key Equation 138

A general d-dimensional **linear separator** can do **binary classification** using the hypothesis

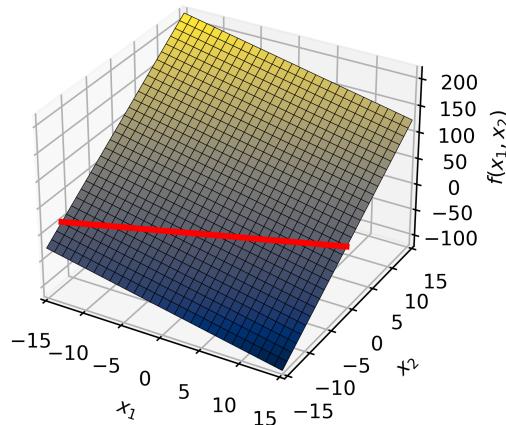
$$h(x; \theta) = \text{sign}(\theta^T x + \theta_0) = \begin{cases} +1 & \text{if } \theta^T x + \theta_0 > 0 \\ -1 & \text{otherwise} \end{cases}$$

Where

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix}$$

What about how it looks? Well, if we have 3 input variables, our line turns into a **plane**:

2-D Classification Problem in 3-D



As we move into 3D, we can **separate** points on **three different axes**.

Just like with regression, this is when we introduce the **hyperplane**:

Concept 139

Our n -dimensional **linear separator** solution to the **binary classification** problem **splits** our space into two **halves**: a positive and a negative half.

The **surface** that **splits** space like this is a $(n - 1)$ -dimensional **hyperplane**.

The hyperplane is **oriented**: there is a **normal** vector θ which defines the **orientation** of the hyperplane, and which side is **positive**.

It also has an **offset** term θ_0 , that slides it in the θ direction **away** from the origin.

For any dimensional input, we can use hyperplanes as separators.

4.2.12 IMPORTANT: A difference between regression and classification

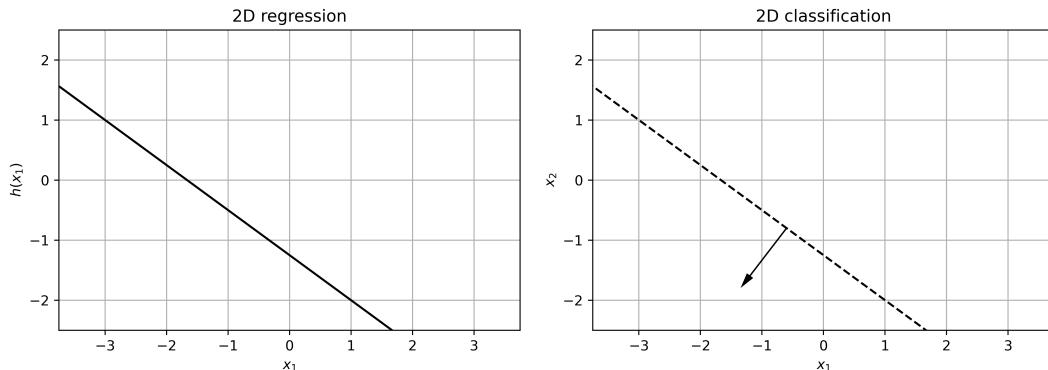
Here is an important misconception that comes up between regression and classification.

Both functions use the equation

$$\theta^T x + \theta_0 \quad (4.15)$$

So, one might think of them as interchangeable.

However, they are **not**. Why is that?



These two plots look almost the same, but represent completely different things!

Notice that these two plots are **both** plotted in 2-D, and both have a **line** plotted. But, they **aren't** as **similar** as they look.

Notice, for example, that the regression plot has **only** x_1 , while the classification plot has x_1 **and** x_2 .

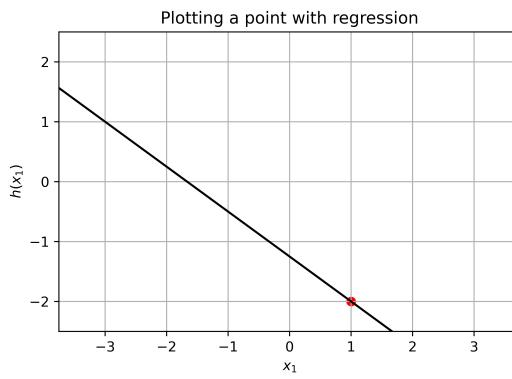
The reason why? The **output**.

- In **regression**, the output is a **real number**: every point on that line represents an input x_1 , and an output $h(x_1)$.

- This plot can only contain **one** input variable: the **second** axis is reserved for the **output!**
- In **classification**, the output is **binary**. So, that line represents only the **values** where the output is $h(x) = 0$.
 - This plot can contain **two** input variables: x_1 and x_2 . Rather than **displaying** the output, we only show one **slice** of the output: the $h(x) = 0$ slice.

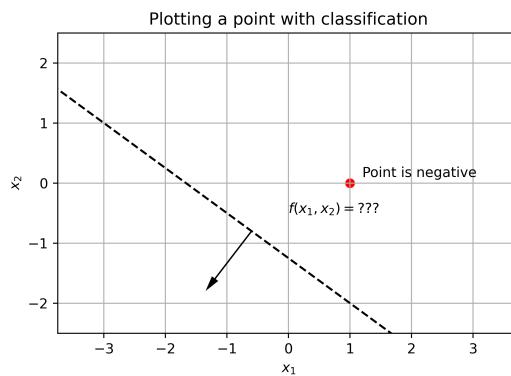
If we think in terms of $f(x) = \theta^T x + \theta_0$, we can compare them directly.

The regression plot shows the exact value on the y-axis. If we want to know what $f(x_1 = 1)$ looks like, we can check the plot: we just get $f(1) = -2$.



We have one input, and we get the exact value of our output.

But the classification plot **doesn't!** We aren't given the value of $\theta^T x + \theta_0$ at $x = (1, 0)$: we just know that it's **negative**.



We have two input, and we **don't** get the exact output.

If we wanted to know the exact value of our 2-D classification, we would need to view it as a plane in 3-D space.

This is the trade-off between these two plots: one gives more information about the output,

and the other allows for more inputs in a lower dimension.

Clarification 140

Regression and **classification** plots that look the same, have **different functions**:

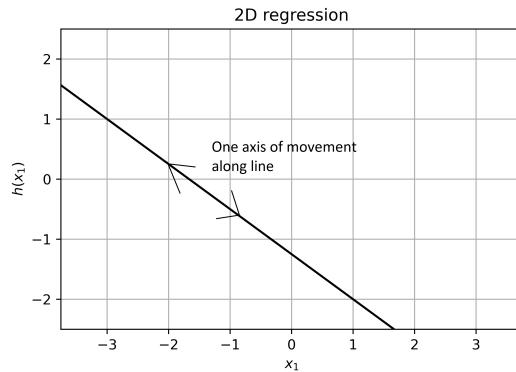
When looking at the output of $f(x) = \theta^T x + \theta_0$,

- A **regression** plot gives the **exact numeric** $f(x)$.
- A **classification** plot only gives the **sign** of the $f(x)$.

When plotting n inputs,

- A **regression** plot uses $d + 1$ dimensions (d -dim hyperplane) to plot: +1 for the **output**.
- A **classification** plot only needs d dimensions (($d - 1$)-dim hyperplane): we only see the $f(x) = 0$ **hyperplane**.

Why do we need $d + 1$ dimensions to plot a d -dimensional **hyperplane**? You can think of it this way: a **line** in 2-D space is a 1-D **hyperplane**: we have only **one axis** we can move on the line.



Our plot is 2-D, but we can only move along one axis on our line!

Because of these differences, θ also acts differently:

Clarification 141

θ appears differently in 2-D regression and classification:

- In **2-D regression**, θ is the **slope** of the line

$$h(x) = \theta x + \theta_0 \quad (4.16)$$

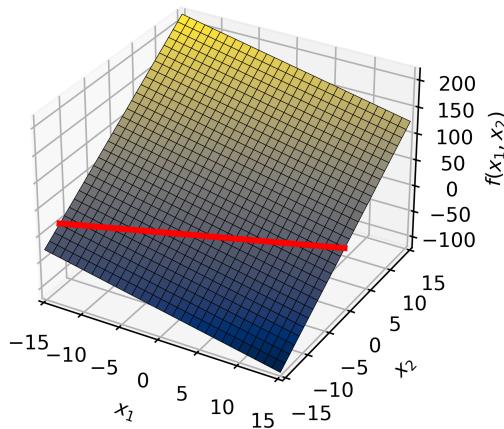
- In **2-D classification**, θ is the **normal vector** of the line

$$0 = \theta^T x + \theta_0 \quad (4.17)$$

4.2.13 3d plot of 2d separator

For additional understanding, you might view the full output of $\theta^T x + \theta_0$, before we simplify the output to $\{-1, +1\}$.

2-D Classification Problem in 3-D



The red line represents where $f(x) = 0$. The black line is our **normal vector**: notice that it's normal to the **line**, not the **plane**.

We mentioned before that, if we wanted to show the exact value of $f(x)$ for our 2-D classifier, we'd need a 3-D plot (just like for regression).

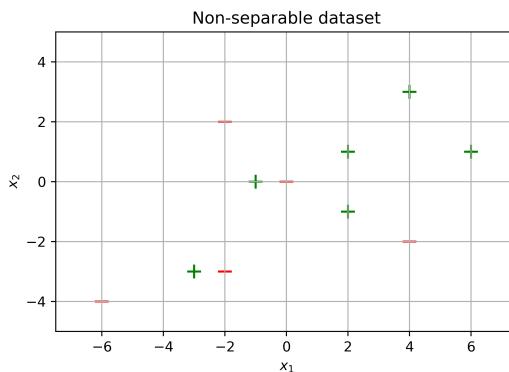
So here, we've done **exactly** that: the **height** is the output of $h(x)$.

But, because we don't **care** about the **exact** output in classification, we usually only graph the **red line**: where $f(x) = 0$.

This shows how we're taking a 2D slice ($f(x) = 0$) out of a 3-D plot (full hyperplane), to **save** on one dimension of **plotting**.

4.2.14 Separable vs Non-separable data

One more consideration: **not all** data can be correctly **divided** by a linear separator!



There's no line we could draw through this data to **separate** the points from each other.

If we can, we call it **linearly separable**.

Definition 142

A **dataset** is **linearly separable** if you can **perfectly** classify it with a **linear classifier**.

A couple common reasons for data to not be linearly separable:

- A positive and negative data point have the exact **same position** in input space.
- Two points on either **side** of a point with opposite classification: $+ - +$ or $- + -$, for example.

Very often, real-world datasets **can't** linearly separated, because of **complexities** in the real world, or random **noise**.

But, sometimes, we can **almost** linearly separate it: we get very high **accuracy**. In those cases, it may be **fine** to use a linear separator: we might risk **overfitting** if we use a more complex model.

Still, if a dataset is not **linearly separable**, or at least **high-accuracy** with a linear separator, that could mean we need a **richer** hypothesis class.

We'll get into ways to make a **richer** class in the **next** chapter: **feature transformations**.

What is "high enough accuracy"? Depends on what you need it for!

Remember: a "richer" or more "expressive" hypothesis class is one that can create more hypotheses than our current one can't!

4.3 Linear Logistic Classifiers

4.3.1 The problem

Now, our goal is to create a **good model** for our problem, **binary classification**.

To do this, we can **try** using our 0-1 loss \mathcal{L} :

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(\text{sign}(\theta^T x^{(i)} + \theta_0), y^{(i)}) \quad (4.18)$$

The **first** thing to note is that there isn't an easy **analytical** solution, no simple **equation**: $\text{sign}(u)$ isn't a function that we can explicitly **solve**, like we could for **linear regression**.

So, we refer to our other approach, **gradient descent**.

But in order to do that, we'll just need to get the **gradient**.

To be fair, this is true for most possible problems: most of them can't be solved analytically.

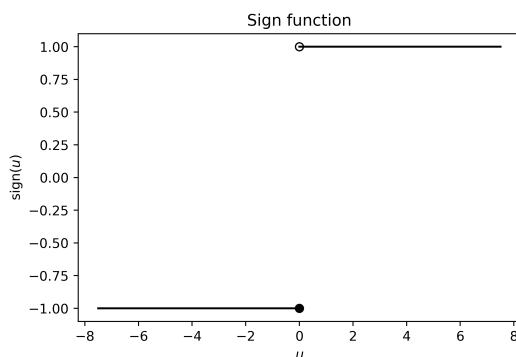
$$\nabla_{\theta} J = 0 \quad (4.19)$$

...Well that's not good.

Why not? Because we use our **gradient** to decide **how** to change θ , if the gradient is 0, we'll never **improve** θ at all!

4.3.2 The real problem: $\text{sign}(u)$ is flat

What's going on here? Let's look at the sign function:



Sign is a flat function! The slope is 0 everywhere, except $u = 0$, where it's **undefined**.

Well, that explains why we can't use the gradient: the function is **flat**.

Another way to say this is that our function doesn't **tell** us when we're **closer** to being right.

There's **no difference** between being **wrong** by 1 unit or being wrong by 10 units: you can't tell if you're getting **closer** to a correct answer.

And the **gradient** doesn't tell you which way to move in **parameter space** to further improve.

Remember, parameter space is what we move through as we change our parameter vector θ .

In fact, the best way we know how to approach this kind of problem takes **exponential** time: it takes exponentially **longer** to solve based on our **number** of data points.

That's way too **slow**. So, we'll have to come up with a **better** function: something to **replace** $\text{sign}(u)$, that still serves the same role.

Concept 143

The **sign function** is difficult to optimize, because it isn't **smooth**: not only is the slope undefined at 0, it is 0 everywhere else.

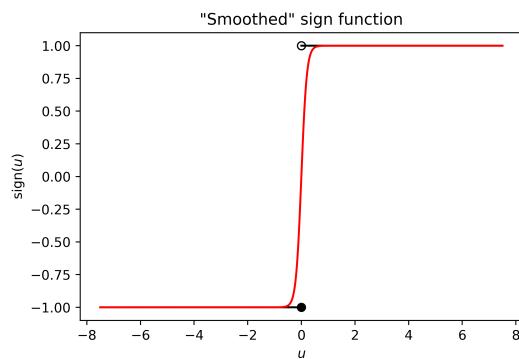
This causes two problems:

- We can't tell whether one **hypothesis** is **closer** to being **correct**, if it has gotten **better**, unless its accuracy has increased.
 - This makes it harder to **improve**.
- We can't indicate how **certain** we are in our answer: $\text{sign}(u)$ is **all-or-nothing**: we choose one class, with no information about how **confident** we are in our choice.
 - Knowing how **uncertain** we are can be **helpful**, both for **improving** our machine and also **judging** the choices or machine makes.

So, we need to explore a **new** approach: we'll **replace** $\text{sign}(u)$ with something else.

4.3.3 The sigmoid function

So, what do we **replace** sign with? We like the way sign **works** (choosing between two different classes based on a **threshold**), so maybe we want a **smoother** version of it.

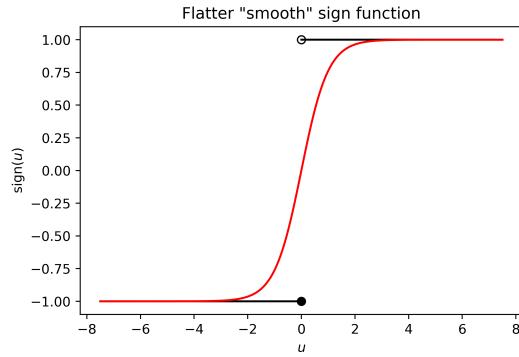


The red line shows a "smoother" sign function, that mostly behaves the same, while solving our problem.

This solves **one** of our two problems: the **gradient is nonzero**.

We could also make it less steep:

It's hard to see visually, but the function is **smooth**, and the slope is **nonzero everywhere**!



So, we need a **function** that accomplishes this. It turns out there are **several** that work: $\tanh u$, for example.

For our purposes, we'll use the following function:

Definition 144

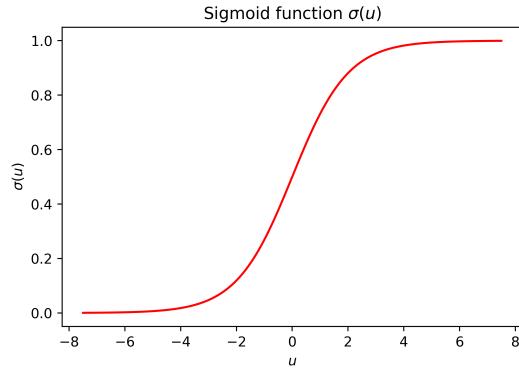
The **sigmoid** function

$$\sigma(u) = \frac{1}{1 + e^{-u}} \quad (4.20)$$

...is a nonlinear function that we use to **compute** the output of our **classification** problem.

It is also called the **logistic** function.

The function looks like this:



4.3.4 Sigmoid as a probability

Something you may **notice** is that $\sigma(x)$ is always between 0 and 1. But before, $\text{sign}(x)$ was **always** between -1 and +1. Why would we use *this* function?

Because going between 0 and 1 has a different advantage: we can interpret it as a **probability**.

Your **value** of $\sigma(u)$ can be stated as, "what does the machine think is the **probability** we **classify** this data point as +1".

And, on the **flip** side, $1 - \sigma(u)$ is the **probability** we **classify** as -1.

This solves the second problem we mentioned **earlier**: we can indicate how **confident** the machine is in its answer!

Concept 145

The output of the **sigmoid function** $\sigma(u(x))$ gives the **probability** that the data point x is classified **positively**.

$$\sigma(u) = P\{x \text{ is classified } +1\}$$

$$1 - \sigma(u) = P\{x \text{ is classified } -1\}$$

Note that this works because $\sigma(u) \in (0, 1)$.

4.3.5 Logistic Regression

So, we've seen the benefits of switching from $\text{sign}(u)$ to $\sigma(u)$. So we'll do that:

We're using $u(x) = \theta^T x + \theta_0$

Key Equation 146

Logistic Regression is a **modification** of **linear regression**.

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

where

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

It outputs the **probability** of a **positive** classification.

If we **plug** this in, we get this slightly ugly expression:

$$h(x; \theta) = \frac{1}{1 + e^{-(\theta^T x + \theta_0)}}$$

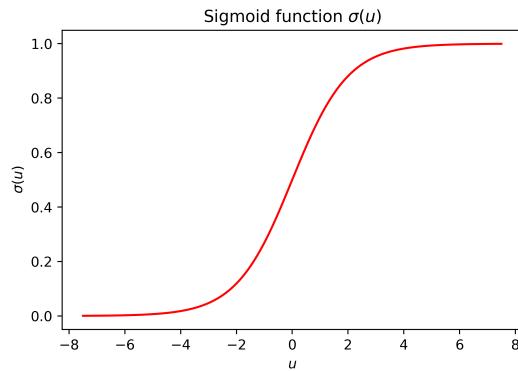
We have a problem, though: **logistic regression** is a... **regression** function. It takes in a **real vector**, and outputs a **real number**: $\mathbb{R}^d \rightarrow \mathbb{R}$.

We can't use this to do **classification**, where want $\mathbb{R}^d \rightarrow \{-1, +1\}$!

4.3.6 Prediction Threshold

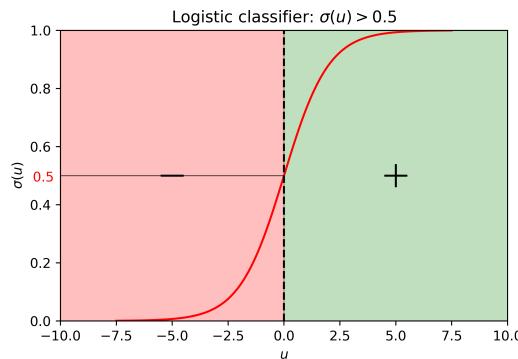
When we were just using $u(x) = \theta^T x + \theta_0$, we classified data points by saying whether $u(x) > 0$. Our boundary was $u(x) = 0$.

We can't quite do that here, because $\sigma(u) = 0$ is **impossible**: $\sigma(u)$ is **always** greater than 0.



$\sigma(u)$ approaches 0 as u approaches $-\infty$, but it never reaches it.

Well, what happens when $u(x) = 0$? We get $\sigma(0) = .5$. So, we could use that as our classification: $\sigma(u) > .5$



But, we don't necessarily always want to use .5:

Example: Imagine if you wanted to **classify** whether someone needs **life-saving** treatment. Classify -1 if sick (they need it), $+1$ if healthy (they don't).

Let's say you got $\sigma(u) = .6$, so you're only 60% sure they **don't** need it. You'd classify that as $\sigma(u) > .5$: they're '**healthy**'.

Even so, you probably shouldn't **refuse** someone treatment that's 40% likely to **save** their life. We might not want to use $\sigma(u) > .5$ after all.

We call the **boundary** between positive and negative the **prediction threshold**.

Definition 147

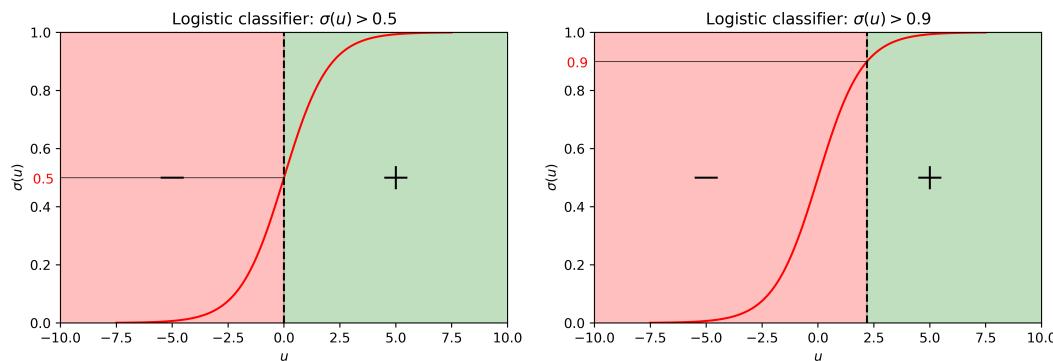
The **prediction threshold** σ_{thresh} is the value where you go from **negative** classification to **positive**.

In general, we say

Our **default** value is a threshold of .5, but our threshold can be **anywhere** in the range

$$0 < \sigma_{\text{thresh}} < 1$$

Example: If $\sigma_{\text{thresh}} = .9$, we would see:



We switch from a .5 threshold to a .9 threshold.

4.3.7 Linear Logistic Classifier

This finally gives us our **linear logistic classifier** (LLC)

Key Equation 148

The **linear logistic classifier** is a **binary** classifier of the form

$$h(x; \theta) = \begin{cases} +1 & \text{if } \sigma(u(x)) > \sigma_{\text{thresh}} \\ -1 & \text{otherwise} \end{cases}$$

where

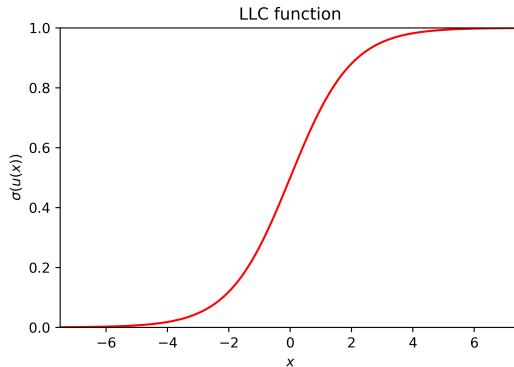
$$u = \theta^T x + \theta_0 \quad \sigma(u) = \frac{1}{1 + e^{-u}} \quad (4.21)$$

We call it linear because of the linear inner function $u(x)$, and logistic because of the outer function $\sigma(u)$.

4.3.8 Modifying our sigmoid

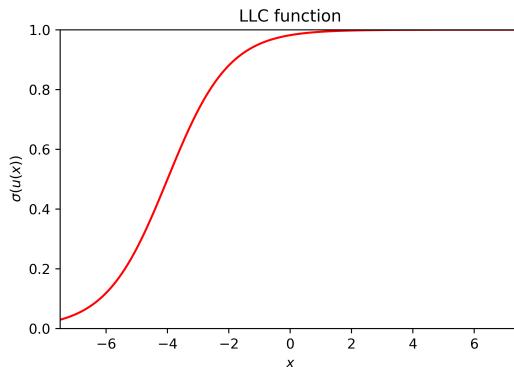
What happens when you modify the **parameters** of an LLC? Let's find out.

We'll use a 1-D input: our variables will be θ (scalar) and θ_0 : $\theta x + \theta_0$



Our baseline LLC: $u(x) = 1x + 0$

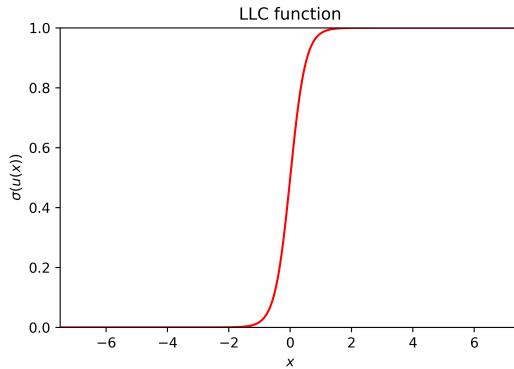
What if we shift by increasing θ_0 ?



Our shifted LLC: $u(x) = 1x + 4$. θ_0 shifts us along the x -axis!

Just like before, it **shifts** us in the **opposite** direction: if θ_0 is **positive**, we shift in the **negative** direction, and vice versa.

What if we increase the magnitude of θ !



Our new LLC: $u(x) = 4x$. Increasing θ makes our function steeper!

Making the magnitude of θ larger makes our function **change** faster.

This makes some sense: if θ (linear slope of $u(x)$) makes $u(x)$ **change** faster, it will make $\sigma(u)$ change faster **too**.

You can combine these changes as well: you can shift your LLC with θ_0 , and also make it steeper/less steep by changing magnitude of θ .

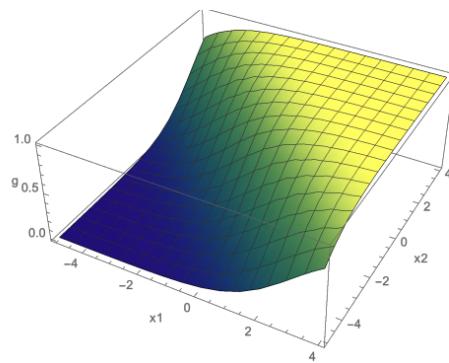
Concept 149

When working with **sigmoids**, you can **transform** them using your **parameters**:

- A higher **magnitude** $\|\theta\|$ makes the slope **steeper**, and answers more **confident**.
- **Increasing** θ_0 **shifts** the sigmoid in the $-\theta$ **direction**, and vice versa.

4.3.9 Viewing our sigmoid in 3D

Let's quickly take a look at a sigmoid in 3D, with two inputs:



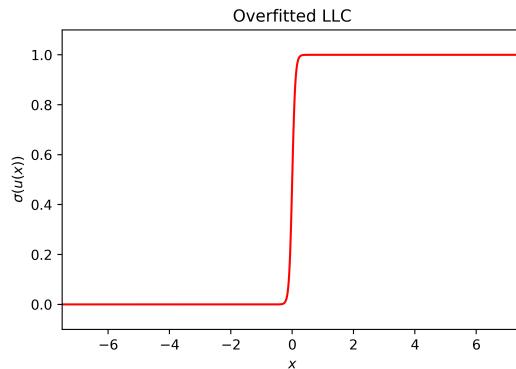
As you can see, you get mostly the same shape: if you look at it from the side, it's exactly the same, in fact! Just stretched out into 3D.

4.3.10 LLCs and overfitting

In chapter 2, we reduced **overfitting** by limiting the **magnitude** of θ using

$$R(\theta) = \lambda \|\theta\|^2 \quad (4.22)$$

In this chapter, it's more clear why reducing **magnitude** reduces **overfitting**. Let's see what happens when θ is very **large**:



Our shifted LLC: $u(x) = 20x$.

This function starts looking more and more like the **sign** function. This means we very, **very quickly** go from **confident** in one answer, to confident in another.

But if you have a limited dataset, and you're very carefully tuned to it, it's doesn't make sense to be **very confident** in your answer. Especially when you're **close** to your **boundary**.

You have closely **fitted** your sigmoid function to your data: in a way, you may have **overfitted** it.

The problem is, you're **rewarded** for increasing θ ! If you're just a little bit **more sure** of the answers you get correct, the loss function continues going **down**.

So, we need to prevent θ from **growing** to an unreasonably high value. We'll **penalize** a large $\|\theta\|$.

This means we're **penalizing** the machine's **overconfidence** in its answer, so that it **generalizes** better.

Concept 150

In **classification**, the **regularizer** follows the form

$$R(\theta) = \lambda \|\theta\|^2$$

Regularization in this form reduces **overfitting** to our data by

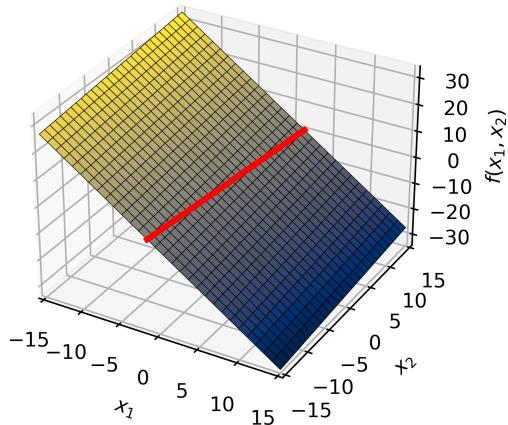
- Making the **transition** between classifications less **sudden**, when it shouldn't be so **certain** of the boundary.
- It also prevents our model from becoming **overly confident** in its answer.

4.3.11 LLCs and LCs have the same boundary

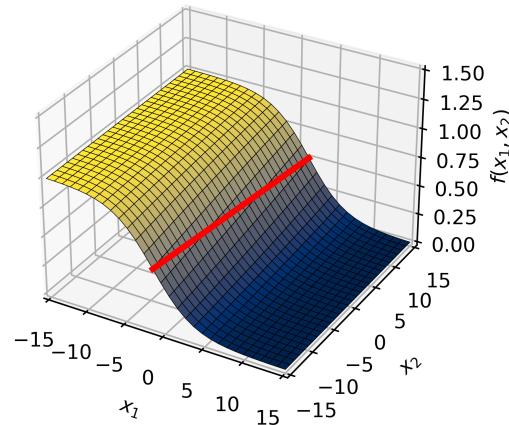
One more important thing to note: noticed that we set $\sigma_{\text{thresh}} = .5$, because that was when $u(x) = 0$.

This means that, if our threshold is 0.5, then the boundary of our LLC should look exactly the same as if it were LC: the only difference is the values that we *can't* see:

2-D Classification Problem in 3-D



2-D Classification Problem in 3-D



Despite having different shapes in 3D, they both create 2-D **linear** classifiers: on the left, $u(x) = 0$, and on the right, $\sigma(u) = .5$.

One way to think about this difference is that while one may be logistic, they are both **linear**: they both create the same **linear separator**.

The main benefit of switching to LLC is that $\sigma(u)$ has a useful **gradient**, while $\text{sign}(u)$ does **not**, so we can do **gradient descent**.

Even if we adjust our threshold σ_{thresh} , that will simply shift the linear classifier.

The probabilistic interpretation is also more appropriate: we shouldn't be fully confident in our answers.

Concept 151

LLCs (Linear Logistic Classifiers) and **LCs** (Linear Classifiers) both create a **linear hyperplane separator** in $d - 1$ dimensional **space**.

If the **threshold value** is 0.5, then they have the **exact same** separator.

4.3.12 Learning LLCs: Loss Functions

Now that we have fully **built up** LLCs, we can start trying to **train** our own.

In order to do that, we need a way to **evaluate** our hypotheses: a **loss function**.

Earlier in the chapter, we tried **0-1 Loss**:

$$\mathcal{L}_{01}(h(x; \Theta), y) = \begin{cases} 0 & \text{if } y = h(x; \Theta) \\ 1 & \text{otherwise} \end{cases}$$

But, this **loss** function has the same problem our **sign** function did: it isn't **smooth**!

It's a **discrete** function based on our **discrete classes**: so, it won't have a smooth **gradient** we can do **descent** on.

For our **sign** function, we switched to the **sigmoid** function, which measures in terms of **probabilities**: this gave us some **smoothness** to our classification.

Could we do the same here?

4.3.13 Building our new loss function

So, the **output** of our sigmoid $\sigma(u)$ is a **probability**: how **likely** do we think a point is to be in class +1?

We want a loss function

$$\mathcal{L}(g, y) \tag{4.23}$$

That considers two facts: the **correct** answer y , and how likely we **expected** +1 to be, $g = \sigma(u)$.

Notation 152

For our **loss function**, rather than using $y \in \{-1, +1\}$, we'll use $y \in \{0, 1\}$.

That way, $\sigma(u)$ and y **match**:

$$y \in \{0, 1\} \quad g \in (0, 1)$$

So, if the correct is 1, then we want $\sigma(u)$ to be **high**. If the correct answer is 0, we want $\sigma(u)$ to be **low**.

For **one** data point, then, we can consider, "how likely did we think the right answer was?"

$$G(g, y) = \begin{cases} g & \text{if } y = 1 \\ 1 - g & \text{else } (y = 0) \end{cases} \quad (4.24)$$

If we choose 1 with probability g , this could also mean, "how likely were we to be **right**?"

This G is how "**good**" our function is, so the **loss** would need for us to take the **negative**: we'll do that later.

4.3.14 Loss Function for Multiple Data Points

Now, how do we consider **multiple** data points? Well, let's think in terms of **probability**: guessing each point is a separate **event**.

We *could* add or **average** our guesses. But, since we're working with **probabilities**, there's a natural way to **combine** them: multiple events **occurring** at the same time.

Before, we asked, "how likely were we to be **right**?" for **one** data point. We could **extend** this question to, "how likely are we to get **every** question right?"

Well, each question we get right is an **independent** event C_i . If we want two independent events to **both** happen, we have to **multiply** their probabilities.

Key Equation 153

The probability of two independent events A and B happening at the same time is

$$P\{A \text{ and } B\} = P\{A\} * P\{B\}$$

So if we want **all** of them, we just multiply:

$$P\{E_{\text{all}}\} = P\{E_1\} * P\{E_2\} * \dots * P\{E_n\} \quad (4.25)$$

Written using pi notation, and also $g^{(i)}$ for multiple data points: _____

$$P\{E_{\text{all}}\} = \prod_{i=1}^n P\{E_i\} = \prod_{i=1}^n \begin{cases} g^{(i)} & \text{if } y^{(i)} = 1 \\ 1 - g^{(i)} & \text{if } y^{(i)} = 0 \end{cases} \quad (4.26)$$

This notation is described in the prerequisites chapter! The short version: instead of adding terms with \sum , you multiply with \prod .

4.3.15 Simplifying our expression - Piecewise

Our piecewise function is a bit **annoying**, though: is there a way to **simplify** it so that it doesn't have to be **piecewise**?

Our goal is to **combine** our two piecewise cases into a **single** equation. That means one of them needs to **cancel out** whenever the other is true.

Well, let's see what we have to **work** with.

Our **two** cases happen when $y = 0$ or $y = 1$: these are **nice** numbers! Why? Because of the **exponent** rules for these two:

- $c^0 = 1$: an exponent of 0 outputs 1: a factor of 1 in a product might as well **not be there**. It has been effectively **cancelled** out.
- $c^1 = c$: an **exponent** of 1 leaves the factor **unaffected**.

So, let's consider the **first** case, g . we can use g^y : if $y = 1$, it's **unaffected**. If $y = 0$, the term is **removed**.

We want the **opposite** for $1-g$. We can **swap** 1 and 0 by doing $1-y$. This gives us $(1-g)^{1-y}$.

For one data point:

$$P\{E\} = \underbrace{g^y}_{y=1} \underbrace{(1-g)^{1-y}}_{y=0} \quad (4.27)$$

We've gotten rid of the piecewise function! Let's add back in the product:

$$P\{E_{all}\} = \prod_{i=1}^n P\{E_i\} = \prod_{i=1}^n g^{(i)y^{(i)}} (1-g^{(i)})^{1-y^{(i)}} \quad (4.28)$$

Looks pretty ugly, but we'll work on that.

4.3.16 Getting rid of the product

Our exponents look pretty **ugly**. Can we do something about that?

More importantly, **products** are also pretty unpleasant: we can't use **linearity**!

Linearity uses **addition** between variables. What sort of **function** could change a **product** into a **sum**?

Linearity makes lots of problems easy to work with, so we try to keep it.

Well, we could **list** out different basic functions, to see which ones connect sums and products. It turns out, one **interesting** function is

$$\log ab = \overbrace{\log a + \log b}^{\text{sum}} \quad (4.29)$$

Aha! If we take the **log** of our function, we can turn a **product** into the **sum**!

The below equation looks complicated, but all we've done is swap the product for a sum!

$$\underbrace{\log \left(\prod_{i=1}^n P\{E_i\} \right)}_{\text{product}} = \underbrace{\sum_{i=1}^n \log (P\{E_i\})}_{\text{sum}} \quad (4.30)$$

We can also separate our two **factors**:

$$\sum_{i=1}^n \left(\log(g^{(i)} y^{(i)}) + \log((1 - g^{(i)})^{1-y^{(i)}}) \right) \quad (4.31)$$

And **finally**, we can remove the **exponents**:

$$\sum_{i=1}^n \left(y^{(i)} \log g^{(i)} + (1 - y^{(i)}) \log (1 - g^{(i)}) \right) \quad (4.32)$$

Concept 154

Our **negative log likelihood** (NLL) comes from a couple steps:

- Use $y \in \{0, 1\}$ instead of $y \in \{-1, +1\}$ so that y and g have **matching** outcomes.
- Get the **chance** the model is right on every **guess**: a **product**.
- Use **exponents** to convert the **piecewise** expression into a single **equation**.
- Take the **log** of our expression to switch from a **product** to a **sum**.
- Take the **negative** to get the **loss** rather than the "**goodness**" of our function.

4.3.17 Negative Log Likelihood

Remember, at the **beginning**, we said that we need to take the **negative**: our function represents how **good** our function is, but we want the **loss**.

With this, our function is in its final form:

Key Equation 155

We can get the loss of our **linear logistic classifier (LLC)** using the **negative log likelihood (NLL)** loss function

$$\mathcal{L}_{\text{nll}}(g^{(i)}, y^{(i)}) = - \left(y^{(i)} \log g^{(i)} + (1 - y^{(i)}) \log (1 - g^{(i)}) \right)$$

Or,

$$-(\text{answer} \log \text{guess} + (1 - \text{answer}) \log (1 - \text{guess}))$$

Our total loss is

$$\sum_{i=1}^n \mathcal{L}_{\text{nll}}(\mathbf{g}^{(i)}, \mathbf{y}^{(i)}) \quad (4.33)$$

Finally, we add our **regularizer**:

$$J_{\text{lr}}(\theta, \theta_0; \mathcal{D}) = \frac{1}{n} \sum_{i=1}^n \left(\mathcal{L}_{\text{nll}}(\mathbf{g}^{(i)}, \mathbf{y}^{(i)}) \right) + \lambda \|\theta\|^2 \quad (4.34)$$

Key Equation 156

The full **objective function** for **LLC** is given as

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

Using our **loss** function \mathcal{L}_{nll} , and our **logistic** function $\sigma(u)$.

4.4 Gradient Descent for Logistic Regression

4.4.1 Summary

Now, we have developed all the tool we need to do binary classification with LLC:

- A **linear** model that lets us **combine** our variables,

$$u(x) = \theta^T x + \theta_0 \quad (4.35)$$

- A **logistic** model that lets us get the **probability** of a classification,

$$\sigma(u) = \frac{1}{1 + e^{-u}} \quad (4.36)$$

- A **threshold value** we use to determine how to **classify** our data,

$$h(x; \theta) = \begin{cases} +1 & \text{if } \sigma(u(x)) > \sigma_{\text{thresh}} \\ 0 & \text{otherwise} \end{cases} \quad (4.37)$$

- A **loss function** NLL we use to **evaluate** our model performance:

$$\mathcal{L}_{\text{nll}}(g^{(i)}, y^{(i)}) = - \left(y^{(i)} \log g^{(i)} + (1 - y^{(i)}) \log (1 - g^{(i)}) \right)$$

- And an **objective function** we can **optimize**:

$$J_{\text{lr}}(\theta, \theta_0; \mathcal{D}) = \lambda \|\theta\|^2 + \frac{1}{n} \sum_{i=1}^n \mathcal{L}_{\text{nll}}(g^{(i)}, y^{(i)}) \quad (4.38)$$

We have everything we need to do optimization.

4.4.2 The problem: Gradient Descent

We want to do **gradient descent** to minimize J_{lr}

$$R(\theta) + J_{\text{lr}}(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}_{\text{nll}}(g^{(i)}, y^{(i)}) \quad (4.39)$$

We want repeatedly **adjust** our model $\Theta = (\theta, \theta_0)$ to improve J_{lr} . To do that, we want the gradients for θ and θ_0 . Let's start with θ .

$$\nabla_{\theta} J_{\text{lr}} = \frac{\partial J_{\text{lr}}}{\partial \theta} \quad (4.40)$$

First, J_{lr} has **two** terms, so we'll separate them.

$$\nabla_{\theta} J_{lr} = \frac{\partial R}{\partial \theta} + \frac{1}{n} \sum_{i=1}^n \frac{\partial \mathcal{L}_{NLL}}{\partial \theta}(g^{(i)}, y^{(i)}) \quad (4.41)$$

The regularization term is pretty easy, because we did it last chapter:

$$\frac{\partial R}{\partial \theta} = 2\lambda\theta \quad (4.42)$$

But what about our first term?

4.4.3 Getting the gradient: Chain Rule

Now, we just need to do

$$\frac{\partial \mathcal{L}_{NLL}}{\partial \theta}(g, y) \quad (4.43)$$

With our \mathcal{L}_{NLL} term, we run into an issue: how do we take the **derivative**? The function is very, very deeply **nested**. In our case:

x **affects** $u(x)$. $u(x)$ **affects** $\sigma(u)$. $\sigma(u) = g$ **affects** $\mathcal{L}_{NLL}(g, y)$, which finally **affects** $J(\theta, \theta_0)$.

How do we represent this **chain** of functions? With the **chain rule**:

$$\frac{\partial A}{\partial C} = \frac{\partial A}{\partial B} \cdot \frac{\partial B}{\partial C} \quad (4.44)$$

So, we'll build up a **chain rule** for our needs. We'll use $g = \sigma(u)$.

$$\frac{\partial \mathcal{L}_{NLL}}{\partial \theta} = \frac{\partial \mathcal{L}_{NLL}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial \theta} \quad (4.45)$$

Sigma contains u , so we'll use that instead:

$$\frac{\partial \mathcal{L}_{NLL}}{\partial \theta} = \frac{\partial \mathcal{L}_{NLL}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial u} \cdot \frac{\partial u}{\partial \theta} \quad (4.46)$$

This is our full **chain rule**!

Key Equation 157

The **gradient** of **NLL** can be calculated using the **chain rule**:

$$\frac{\partial \mathcal{L}_{NLL}}{\partial \theta} = \frac{\partial \mathcal{L}_{NLL}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial u} \cdot \frac{\partial u}{\partial \theta} \quad (4.47)$$

4.4.4 Getting our individual derivatives

We can take the derivative of each of these objects. First, let's look at \mathcal{L}_{NLL}

$$\mathcal{L}_{\text{NLL}}(\sigma, y) = - \left(y \log \sigma + (1-y) \log (1-\sigma) \right)$$

And we'll use $\frac{d}{dx} \log(x) = \frac{1}{x}$

$$\boxed{\frac{\partial \mathcal{L}_{\text{NLL}}}{\partial \sigma} = - \left(\frac{y}{\sigma} - \frac{1-y}{1-\sigma} \right)} \quad (4.48)$$

Now, we look at $\sigma(u)$:

$$\sigma(u) = \frac{1}{1 + e^{-u}} \quad (4.49)$$

If we take the derivative, we can get:

$$\frac{\partial \sigma}{\partial u} = \frac{-e^{-u}}{(1 + e^{-u})^2} \quad (4.50)$$

Which we can rewrite, conveniently, as

Try this yourself if you're curious!

$$\boxed{\frac{\partial \sigma}{\partial u} = \sigma(1 - \sigma)} \quad (4.51)$$

Finally, our last derivative:

$$u = \theta^T x + \theta_0 \quad (4.52)$$

$$\boxed{\frac{\partial u}{\partial \theta} = x} \quad (4.53)$$

4.4.5 Simplifying our chain rule

So, now, we can put together our chain rule:

$$\frac{\partial \mathcal{L}_{\text{NLL}}}{\partial \theta} = \frac{\partial \mathcal{L}_{\text{NLL}}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial u} \cdot \frac{\partial u}{\partial \theta} \quad (4.54)$$

Plug in the derivatives:

$$\frac{\partial \mathcal{L}_{\text{NLL}}}{\partial \theta} = - \left(\frac{y}{\sigma} - \frac{1-y}{1-\sigma} \right) \cdot \sigma(1-\sigma) \cdot x \quad (4.55)$$

Simplify:

$$\frac{\partial \mathcal{L}_{\text{NLL}}}{\partial \theta} = \left((1-y)\sigma - y(1-\sigma) \right) \cdot x \quad (4.56)$$

And finally, we sum the terms. We can do the θ_0 gradient at the same time: the only difference is that $\frac{\partial u}{\partial \theta_0} = 1$, instead of x .

Key Equation 158

The **gradients** of NLL for gradient descent are

$$\nabla_{\theta} \mathcal{L}_{\text{NLL}} = (\sigma - y)x$$

$$\frac{\partial \mathcal{L}_{\text{NLL}}}{\partial \theta_0} = (\sigma - y)$$

We can plug this into J_{lr} :

$$\nabla_{\theta} J_{\text{lr}} = \frac{1}{n} \sum_{i=1}^n \left((g^{(i)} - y^{(i)})x^{(i)} \right) + 2\lambda\theta \quad (4.57)$$

One comment we didn't make: remember that $R(\theta)$ won't show up in the θ_0 derivative!

$$\frac{\partial J_{\text{lr}}}{\partial \theta_0} = \frac{1}{n} \sum_{i=1}^n (g^{(i)} - y^{(i)}) \quad (4.58)$$

We can use this to do **gradient descent**!

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \nabla_{\theta} J_{\text{lr}}(\theta_{\text{old}}) \quad (4.59)$$

In $\theta^{(t)}$ notation:

$$\theta^{(t)} = \theta^{(t-1)} - \eta \left(\nabla_{\theta} J_{\text{lr}}(\theta^{(t-1)}) \right) \quad (4.60)$$

$$\theta_0^{(t)} = \theta_0^{(t-1)} - \eta \left(\frac{\partial J_{\text{lr}}(\theta^{(t-1)})}{\partial \theta_0} \right) \quad (4.61)$$

This also corresponds to some basic math within Neural Networks, which we will return to **later** in the course.

4.5 Handling Multiple Classes

Now, we have developed a **binary** classifier, using logistic regression. But, many (almost all) problems have more than two classes!

Example: Different animals, genres of movies, sub-types of disease, etc.

4.5.1 Approaches to multi-class classification

So, we need to a way to do **multi-classing**. Consider two main approaches:

- Train many binary classifiers on different **classes** and **combine** them into a single model.
 - There are several ways to **combine** these **classifiers**. We won't go over them here, but some **names**: OVO (one-versus-one), OVA (one-versus-all).
- Make **one** classifier that handles the multi-class problem by itself.
 - This model will be A **modified** version of logistic regression, using a variant of NLL.

The **latter** approach is what we will use in this **next** section.

4.5.2 Extending our Approach: One-Hot Encoding

Rather than being **restricted** to classes 0 and 1, we'll have k **distinct** classes. Our **hypothesis** will be

$$h : \mathbb{R}^d \rightarrow \{C_1, C_2, C_3, \dots, C_k\}$$

Where C_i is the i^{th} class. Meaning, we want to **output** one of those k **classes**.

Because we'll be using our computer to do **math** to get the **answer**, we need to represent this with **numbers**. Before, we would simply **label** with 0 or 1.

We could return $\{1, 2, 3, 4, 5, \dots, k\}$ for each **label**. But this is **not** a good idea: it implies that there's a natural **order** to the classes, which isn't necessarily true.

If we don't **actually** think C_1 is closer to C_2 than to C_5 , we probably shouldn't represent them with numbers that are **closer** to each other.

Instead, each class needs to be a **separate** variable. We can store them in a vector:

$$\begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_k \end{bmatrix} \quad (4.62)$$

So, our **label** will be

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix} \quad (4.63)$$

In binary classification, we used 0 or 1 to indicate whether we fit into one **class**. So, that's how we'll do each class: 0 if our data point is **not** in this class, 1 if it **is**.

This approach is called **one-hot encoding**.

Definition 159

One-hot encoding is a way to represent **discrete** information about a data point.

Our k classes are stored in a length- k column **vector**. For **each** variable in the vector,

- The value is **0** if our data point is **not in that class**.
- The value is **1** if our data point is **in that class**.

In one-hot encoding, items are **never** labelled as being in **two** classes at the **same time**.

Example: Suppose that we want to classify **furniture** as table, bed, couch, or chair.

$$\begin{bmatrix} \text{table} \\ \text{bed} \\ \text{couch} \\ \text{chair} \end{bmatrix} \quad (4.64)$$

For each class:

$$\mathbf{y}_{\text{chair}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad \mathbf{y}_{\text{table}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{y}_{\text{couch}} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \mathbf{y}_{\text{bed}} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad (4.65)$$

4.5.3 Probabilities in multi-class

So, we now know our **problem**: we're taking in a data point $x \in \mathbb{R}^d$, and **outputting** one of the classes as a **one-hot vector**.

So, now that we know what sorts of data we're **expecting**, we need to decide on the formats of our **answer**.

We'll be returning a vector of length- k : **one** for each **class**. When we were doing **binary** classification, we estimated the **probability** of the positive class.

So, it should make sense to do the same **here**: for each class, we'll return the estimated **probability** of our data point being in that class.

$$g = \begin{bmatrix} P\{x \text{ in } C_1\} \\ P\{x \text{ in } C_2\} \\ \vdots \\ P\{x \text{ in } C_k\} \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_k \end{bmatrix} \quad (4.66)$$

We need one **additional** rule: the probabilities need to add up to **one**: we should assume our point ends up in some class or **another**.

$$g_1 + g_2 + \dots + g_k = 1 \quad (4.67)$$

Concept 160

The different terms of our **multi-class** guess g_i represent the **probability** of our data point being in class C_i .

Because we should assume our data point is in **some** class, all of these probabilities have to **add** to 1.

Let's be careful, though: this is only true for probabilities within a single data point.

Example: Suppose you have two animals (data points).

- It's impossible for the first animal to be **both** 90% cat and 90% dog.
- *But*, there's no issue with the first animal being 90% cat and the second animal being 90% dog.

Clarification 161

It's only true that all of the probabilities for the **same data point** need to add to 1.

If you have $P\{\text{class 1}\}$ for one data point and $P\{\text{class 2}\}$ for another data point, those **aren't related**.

So, we want to scale our values so they add to 1: this is called **normalization**. How do we do that?

Well, let's say each class gets a **value** of r_i , before being **normalized**. For now, let's ignore how we got r_i , just know that we have it.

To make the total 1, we'll **scale** our terms by a factor C :

$$C(r_1 + r_2 + \dots + r_k) = C \left(\sum_{i=1}^k r_i \right) = 1 \quad (4.68)$$

We can get our factor C just by dividing:

$$C = \frac{1}{\sum r_i} \quad (4.69)$$

We've got our desired g_i now!

$$g = \begin{bmatrix} r_1 / \sum r_i \\ r_2 / \sum r_i \\ \vdots \\ r_k / \sum r_i \end{bmatrix} \quad (4.70)$$

4.5.4 Turning sigmoid multi-class

Now, we just need to compute r_i terms to plug in. To do that, we'll see how we did it using sigmoid:

$$g = \sigma(u) = \frac{1}{1 + e^{-u}} \quad (4.71)$$

This function is 0 to 1, which is good for being a probability.

Just for our convenience, we'll switch to positive exponents: all we have to do is multiply by e^u/e^u .

Negative numbers are easy to mess up in algebra.

$$g = \frac{e^u}{e^u + 1} \quad (4.72)$$

We'll think of **binary** classification as a special case of **multi-class** classification. The above probability could be thought of as g_1 : the chance of our first class.

Concept 162

Binary classification is a **special** case of **multi-class** classification with only **two** classes.

So, we can use it to figure out the **general** case.

So, what was our **second** probability, $1 - g$? This will be our second class, g_2 .

$$g_2 = 1 - g = \frac{1}{1 + e^u} \quad (4.73)$$

This follows an $r_i/(\sum r_i)$ format: the numerators (1 and e^u) add to **equal** the denominator ($1 + e^u$).

$$g = \begin{bmatrix} 1/(1 + e^u) \\ e^u/(1 + e^u) \end{bmatrix} \quad (4.74)$$

How do we **extend** this to **more** classes? Well, 1 and e^u are **different** functions: this a problem. We want to be able to **generalize** to many r_i .

How do they make them **equivalent**? We could say $1 = e^0$. So, we could treat both terms as **exponentials!**

$$g_1 = \frac{e^u}{e^0 + e^u} \quad (4.75)$$

We can do this for an **arbitrary** number of terms. We'll treat them as **exponentials**, just like for e^u and e^0

$$g_i = \frac{r_i}{\sum r_j} = \frac{e^{u_i}}{\sum e^{u_j}} \quad (4.76)$$

Now, we have a template for expanding into higher dimensions!

4.5.5 Our Linear Classifiers

What are each of those u_i terms? When we were doing **binary classification**, we used a **linear regression** function to help generate the probability:

$$u(x) = \theta^T x + \theta_0 \quad (4.77)$$

Remember that $u(x)$ is not a probability yet: we used a sigmoid to turn it *into* a probability.

Now, we want multiple probabilities. So, we create multiple different functions u_i : k different linear regression models (θ, θ_0) . We'll represent each vector as $\theta_{(i)}$.

$$\theta_{(1)} = \begin{bmatrix} \theta_{1(1)} \\ \theta_{2(1)} \\ \vdots \\ \theta_{d(1)} \end{bmatrix} \quad \theta_{(2)} = \begin{bmatrix} \theta_{1(2)} \\ \theta_{2(2)} \\ \vdots \\ \theta_{d(2)} \end{bmatrix} \quad \theta_{(k)} = \begin{bmatrix} \theta_{1(k)} \\ \theta_{2(k)} \\ \vdots \\ \theta_{d(k)} \end{bmatrix} \quad (4.78)$$

Each of these models could be seen as a "different perspective" of our data point: what about that data point is prioritized (large θ_i magnitudes), how do we bias the result (θ_0)?

This "perspective" we call $\theta_{(i)}$ will tell us if our data point is "closer" to the class it represents. And we compute the result with:

$$u_1(x) = \theta_{(1)}^T x + \theta_{0(1)} \quad u_2(x) = \theta_{(2)}^T x + \theta_{0(2)} \quad u_k(x) = \theta_{(k)}^T x + \theta_{0(k)} \quad (4.79)$$

In the last section, we emphasized that we can only use $\sum p_i = 1$ for the probabilities of a **single** data point. Based on this, we'll focus on only one data point.

Clarification 163

In this section, x represents only **one data point** $x^{(i)}$.

Softmax treats each data point **individually**, so it's easier to not group them together.

Having all these separate equations for θ_i is tedious. Instead, we can combine them all into a $(d \times k)$ **matrix**.

$$\theta = [\theta_{(1)} \quad \theta_{(2)} \quad \dots \quad \theta_{(k)}] = \begin{bmatrix} \theta_{1(1)} & \theta_{1(2)} & \dots & \theta_{1(k)} \\ \theta_{2(1)} & \theta_{2(2)} & \dots & \theta_{2(k)} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{d(1)} & \theta_{d(2)} & \dots & \theta_{d(k)} \end{bmatrix} \quad (4.80)$$

k classes, so we need k classifiers. We'll stack them side-by-side like how we stacked multiple data points to create X .

And our constants, θ_0 , in a $(k \times 1)$ matrix:

$$\theta_0 = \begin{bmatrix} \theta_{0(1)} \\ \theta_{0(2)} \\ \vdots \\ \theta_{0(k)} \end{bmatrix} \quad (4.81)$$

Concept 164

We can combine **multiple classifiers** $\Theta_{(i)} = (\theta_{(i)}, \theta_{0(i)})$ into large **matrices** θ and θ_0 to compute **multiple** outputs u_i at the **same** time.

This will put all of our terms into a $(1 \times k)$ vector u .

$$u(x) = \theta^T x + \theta_0 = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_k \end{bmatrix} \quad (4.82)$$

4.5.6 Softmax

We now have all the pieces we need. Our **linear regression** for each class:

$$\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_k \end{bmatrix} = \theta^T x + \theta_0 \quad (4.83)$$

The **exponential** terms, to get **logistic** behavior:

$$r_i = e^{u_i} \quad (4.84)$$

The **averaging** to get probability = 1:

$$g = \begin{bmatrix} r_1 / \sum r_i \\ r_2 / \sum r_i \\ \vdots \\ r_k / \sum r_i \end{bmatrix} \quad (4.85)$$

And so, our multiclass function is...

Definition 165

The **softmax function** allows us to calculate the probability of a point being in each class:

$$g = \begin{bmatrix} e^{u_1} / \sum e^{u_i} \\ e^{u_2} / \sum e^{u_i} \\ \vdots \\ e^{u_k} / \sum e^{u_i} \end{bmatrix}$$

Where

$$u_i(x) = \theta_{(i)}^T x + \theta_{0(i)} \quad (4.86)$$

If we are forced to make a **choice**, we choose the class with the **highest probability**: we return a **one-hot encoding**.

4.5.7 NLLM

One loose end left to tie up: our **loss function**. We need to evaluate our hypothesis, and be able to improve it.

For **binary classification**, we did **NLL**:

$$\mathcal{L}_{\text{nll}}(g, y) = - \left(y \log g + (1 - y) \log (1 - g) \right)$$

How do we make this work in **general**? Well, we want to make our two terms have a **similar** form, so we can generalize to more classes.

- g and $1 - g$ are both probabilities: we can think of them as g_1 and g_2 , respectively.

- If $g = g_1$, then we would expect $y = y_1$. And indeed: it gives a 1 if we're in the first class (+1).
 - Similarly, $1 - y = y_2$.

$$\mathcal{L}_{\text{nll}}(\mathbf{g}, \mathbf{y}) = - \left(y_1 \log(g_1) + (1 - y_1) \log(g_2) \right)$$

They have the **same** format now! Much tidier. And it tracks: when one **label** is correct, the other term is $y_j = 0$, and **vanishes**.

Does this **generalize** well? It turns out it does: with **one-hot encoding**, the correct label is **always** $y_j = 1$, and the incorrect labels are **all** $y_j = 0$.

So, we'll write it out:

Key Equation 166

The **loss** function for **multi-class** classification, **Negative Log Likelihood Multiclass (NLLM)**, is written as:

$$\mathcal{L}_{\text{NLLM}}(\mathbf{g}, \mathbf{y}) = - \sum_{j=1}^k y_j \log(g_j)$$

Because of **one-hot encoding**, all terms except one have $y_j = 0$, and thus **vanish**.

Using all of these functions, we can finally do gradient descent on our multi-class classifier. However, we won't go through that work in these notes.

4.6 Prediction Accuracy and Validation

We've been working in **probabilities**, but in the end, the goal is usually to make a **decision** or **prediction**: which class do we pick?

In general, we just pick the class we predict with the **highest** probability.

And in the real world, we don't care about how close we were to right - we just care about how often we **were** right.

So, we use **accuracy**.

Definition 167

The **accuracy** of our model is the **percentage** of the time we get the **right** answer.

We can write this as

$$A(h; D) = 1 - \frac{1}{n} \sum_{i=1}^n \mathcal{L}(h(x^{(i)}), y^{(i)}) \quad (4.87)$$

Where \mathcal{L} is 0-1 loss (**counting** the number of **wrong** answers)

Or, "one minus how often we get the answer **wrong**".

4.7 Terms

- Class
- Classification
- Label
- Binary Classification
- 0-1 Loss
- Linear Classifier
- Separator
- Orientation
- Boundary
- Normal Vector
- Dot Product (Conceptual)
- Linear Separator
- Sign Function
- Hyperplane
- Separability
- Non-separable data
- Sigmoid Function
- Logistic Regression
- Prediction Threshold
- Linear Logistic Classifier (LLC)
- Negative Log Likelihood (NLL)
- Multi-class Classification
- One-Hot Encoding
- Normalization
- Softmax Function
- Negative Log Likelihood Multi-Class (NLLM)
- Accuracy

CHAPTER 5

Feature Representation

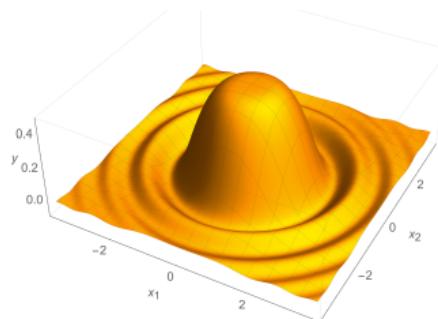
What's still missing?

Last chapter, we used our linear regression model to do classification: we created a "hyperplane" to **separate** the data that we placed in each class.

We also mentioned that regularization can increase **structural error**, by limiting what possible θ models we're allowed to use.

But, what if our linear model is already **too limited**? What if we need a more complicated model? This is true in a lot of real-world problems, like vibration:

Our goal was to decrease estimation error, but that's beside the point right now.



This wave doesn't seem particularly friendly to a planar approximation.

These kinds of situations are called, appropriately, **non-linear**.

Concept 168

Non-linear behavior cannot be accurately represented by any **linear** model.

In order to create an accurate model, we have to use some **nonlinear** operation.

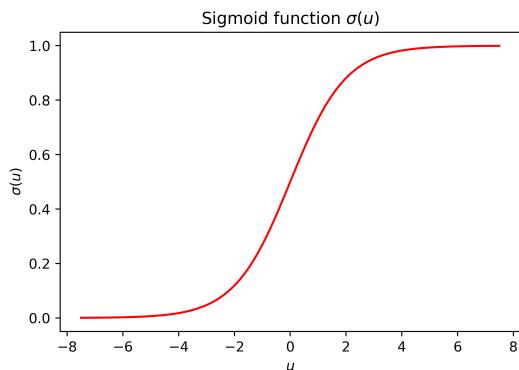
If we could create effective, non-linear models, we might even be able to deal with data that was previously "linearly inseparable".



Possible Solutions: Polynomials

Let's try to think of ways to approach this problem. We'll start with a 1-D input, for simplicity.

Upon hearing "non-linear", we might remember the function we introduced last chapter: the **sigmoid**.



Your friendly neighborhood sigmoid.

Can we use this to create a new model class? For now, unfortunately not: remember that we used this in the last chapter, and we still got a **linear** separator. The reasons were discussed there.

Instead, we can get inspiration from our example of "structural error". For now, let's focus on **regression** (though classification isn't too different):

We'll show ways we can use this kind of approach, when we discuss Neural Networks.

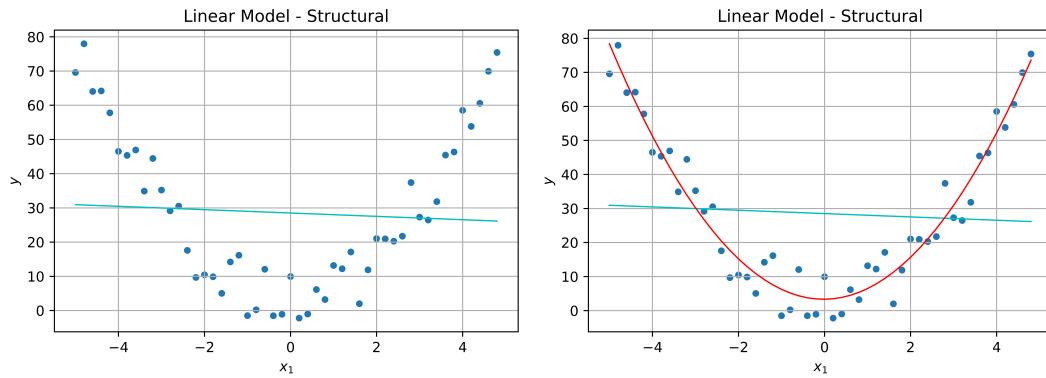


Figure 5.1: A linear function can't represent this dataset. However, a parabola can!

We're still using our input variable x , but this time, we've "transformed" it: we have squared x , giving us a model of the form

Remember that x is 1-D right now!

$$h(x) = Ax^2 + Bx + C \quad (5.1)$$

It should be clear that this model is more **expressive** than the one before: it can create every model that our linear approach could (just by setting $A = 0$), and it can create new models in a parabola shape.

Concept 169

We can make our **linear** model more **expressive** by add a squared term, and turning it into a **parabolic** function.

Reminder: "expressiveness" or "richness" of a hypothesis class is how many models it can represent: a more expressive model can handle more different situations.

This concept can be extended even further, to any **polynomial**.

Transformation

How do we *generalize* this concept? Well, we have a set of constant parameters A, B, C . These are similar to our constants θ_i . Let's change our notation:

$$h(x) = \theta_2 x^2 + \theta_1 x + \theta_0 \quad (5.2)$$

Now, we've got something more familiar. We could imagine extending this to any number of terms $\theta_i x^i$: if we needed a cubic function, for example, we could include $\theta_3 x^3$.

This is starting to look pretty similar to our previous model: in fact, we could even separate out θ as a variable:

Notice that θ_0 corresponds to $x^0 = 1$.

$$h(x) = \underbrace{\sum_{i=1}^k \theta_i x^i}_{\text{Polynomial sum}} = \underbrace{\begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_k \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^k \end{bmatrix}}_{\text{Store as vectors}} = \underbrace{\theta^T}_{\text{Simplify}} \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^k \end{bmatrix} \quad (5.3)$$

This really *is* starting to look like our linear transformation! That's helpful: we might be able to use the techniques we developed before.

In fact, we can argue that they're **equivalent**: we've just changed what our input vector is. Consider our new input $\phi(x)$:

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^k \end{bmatrix} \quad h(x) = \theta^T \underbrace{\phi(x)}_{\text{New input}} \quad (5.4)$$

This is called **transforming** our input.

Definition 170

A **transformation** $\phi(x)$ takes our input vector x and converts it into a **new** vector.

This transformation can be used to:

- Allow our model to handle new, more **complex** situations (more **expressiveness**)
- Pre-process** our data to make it **easier** for our model to find **patterns**.
- Convert our data into a **usable** format (if, say, the original format doesn't fit into our equations)

Example: Taking our input x and converting it into a polynomial is a **transformation** of our input.

This chapter will focus on these kinds of transformations.



Features

One benefit of only changing our input is that we can continue to use our linear representation: we will be able to optimize a "linear" model θ , over data that has been made **nonlinear**.

These transformations can be complex, especially for multi-dimensional inputs. In this

first case, we only combined one input with **itself**. But, often, we can combine multiple together!

Thus, we should be careful to distinguish each input variable from each other. We often call these "**features**". However, we need to be careful:

Clarification 171

We often use the word **feature** in related (but not identical) contexts:

- A **feature** can be one **aspect** of our **original data**: for example, whether or not something is a cat or a dog, or the height of a patient.
- A **feature** can also be one mathematical **variable** in our **transformed input**. x_i is a feature of the data, while each variable of $\phi(x)$ is a feature of the transformed data.

Just like how we have an input space, we call the collection of possible values for our features the **feature space**.

Example: x in our previous example was a feature of the data, while x^i is a feature of our transformed vector.

Combined, this is why we called this technique the **feature transformation**: we apply some *transform* to the *features* of a data, to create a new set of *features*.

Since these transforms only apply to our features, we can keep the structure of a linear function:

Definition 172

Feature transformation allows us to do **linear** regression or classification on a set of **features** we have **non-linearly transformed**:

$$h(x) = \theta^T \phi(x)$$

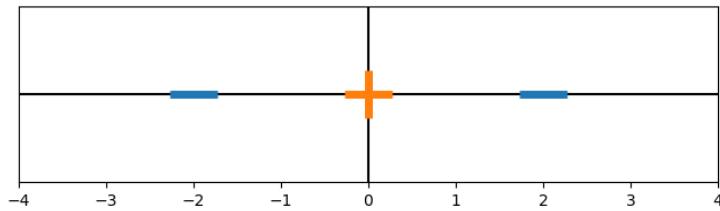
$\phi(x)$ is our (often nonlinear) transformation of our features x .

5.1 Gaining intuition about feature transformations

Now that we understand the general idea of feature transformations, we can begin work with them, particularly for classification.

Our goal is often to take data that linear models couldn't handle, and make it **separable**.

So, we'll consider maybe the simplest (solvable) case of a nonlinear data set:



In its current state, there's no one plane that would go through these data points: this is where our transform comes in. We'll try using a polynomial transform with x^2 . It turns out, $-x^2 + 2$ works pretty well.

How do we visualize this? It turns out, there are different perspectives:

Clarification 173

There are **two** different ways we can **graph** a transformation:

- We transform the **separator**: if our model is $-x^2 + 2$, we just graph that function over the data.
 - This is the approach we used above: we wanted a line that **fit** to our data.
- We transform the **data**: we graph each data point according to our model $-x^2 + 2$.
 - This model allows us to keep a "**linear separator**": we effectively "shift" the data to be linear.

These models are mathematically **equivalent**, and we'll switch the approach we're using based on which is easier/more useful to graph.

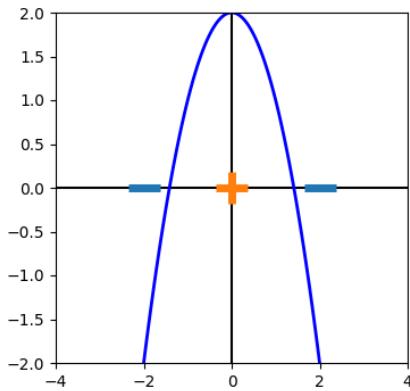
It may seem concerning to transform the data, rather than the model. However, keep in mind that:

- If you switch perspectives, they're the same: a different model will just "transform" the data when calculating its output.
- Usually, we try to preserve the original structure of the data, so we don't lose information: we just add more.
 - For example, $[1, x, x^2]$ still contains the information x : we just add x^2 .

Example: Let's show both of these in action.

5.1.1 Transforming our separator

First, we transform our linear separator as desired: graphing $-x^2 + 2 = 0$ on our plot.



In this version, we've taken our hyperplane separator and transformed it nonlinearly.

In this case, we have assigned $-x^2 + 2 < 0$ as positive.

In this version, we preserve the structure of the data, making it easier to see the original shape. However, it's not as easy to think about the shape and orientation of the "plane" now that it's been deformed.

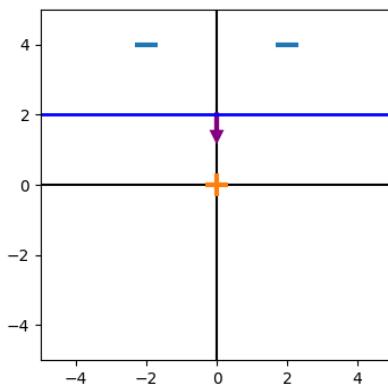
For example, we don't really have a good "normal" vector, even if we know which side is positive. This is why, to keep our model "linear", we can transform the data, and find the corresponding plane. We'll do that next.

5.1.2 Transforming our data

In this case, every data point gets plotted on $[x, x^2]$. Our hyperplane is given by

$$-x^2 + 2 = \underbrace{\begin{bmatrix} 0 \\ -1 \end{bmatrix}}_{\theta^T}^T \begin{bmatrix} x \\ x^2 \end{bmatrix} + 2 \quad (5.5)$$

Thus, we get a θ pointing downward, with an offset of 2.



This time, we've transformed our data: the math is totally the same, but now we can identify our separator more easily.

Note that our transformation makes the data linearly separable!

Concept 174

Features transformations allow us to **non-linearly** transform our data, in order to make that data **linearly separable**, or at least, more **accurate** with a linear separator.

Often, we do this by transforming into a **higher dimensional** space.



5.1.3 Positive vs. Negative

While these perspectives are helpful, they can become too complicated with more dimensions/higher-dimensional transformations.

In an effort to simplify, we might ask ourselves, "what do we really want to know"? In the end, all we typically care about is classification: which data points are positive or negative?

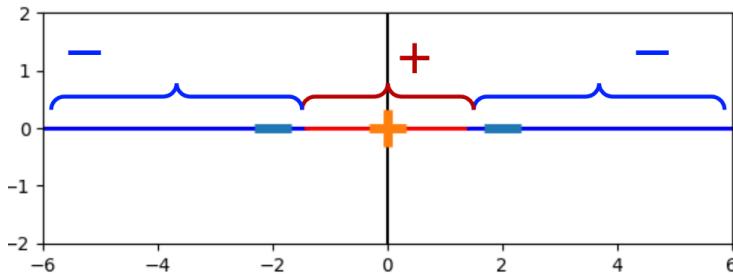
So, we'll create a third representation to correspond to that.

Concept 175

A third, **simplified** representation of our transformation doesn't show how it affects our data points or classifier. Instead, we just show the **result**: which regions are classified as positive, and which are classified as negative?

This allows us to see which points get **classified** in which way, without considering the high-dimensional details of the model itself.

Example: We can graph this for our sample data:



This way, we can stay in a 1-D space, while showing the information we need!

Note that the points where we switch between positive and negative, $\pm\sqrt{2}$, are the points corresponding to $-x^2 + 2 = 0$: they're the only part of the separator surface visible in our 1D plot.

They match our nonlinear hyperplane separator from section 5.1.1

5.2 Systematic feature construction

Now that we've established feature transformations, let's consider a couple options for how we'd want to do it, and how we can generalize to higher dimensions.

Here, we'll present two common ways to construct features, in a way that's consistent across problems, or "systematic".

5.2.1 Polynomial Basis

At the start of this chapter, we introduced the idea of polynomial transformations. If a linear function isn't "expressive" enough to solve a problem, then we can create a more complex model, based on how many x^i we include. This can be written as:

$$h(x) = \sum_{i=1}^k \theta_i x^i = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_k \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^k \end{bmatrix} \quad (5.6)$$

Another word for this might be a "polynomial **basis**".

Definition 176

A "**basis**" is the set of basic, **distinct** elements where every **linear** combination creates a unique item in the **space**.

In this case, each term x^i is part of the "basis": we can use any combination

$$\sum_i \theta_i x^i$$

to create a unique polynomial. This allows us to cover our "feature space".

5.2.1.1 Order

An important question to ask is, "how many terms do we include"?

We categorize our polynomials based on the highest exponent included: this is called the **order**.

Definition 177

Order k , also known as **degree**, is the **largest** exponent allowed in our **polynomial**.

Every higher exponential x^j can be thought of as having a coefficient $\theta_k = 0$: as far as we're concerned, it **doesn't exist**.

We could also call this "problem independent": it works regardless of what kind of problem you have. Though, that doesn't mean problem time won't affect performance.

Example: We can compare different orders, by looking at the feature vector they create:

Here's a table of the first few:

Order	$d = 1$	Example
0	[1]	3.5
1	$[1, x]^T$	$2.5x - 1$
2	$[1, x, x^2]^T$	$4.1x^2 - 10x + 1$
3	$[1, x, x^2, x^3]^T$	$x^3 + 8x^2 + x - \sqrt{2}$
:	:	:

Note that, while we chose every coefficient to be nonzero here, they don't have to be! $-x^2 + 2$ from before is a valid second-order polynomial.

The order we choose is an important decision choice.



5.2.1.2 Overfitting with order

It's difficult to know how many terms to include in our polynomial, but we run into two problems if our order is **too high**:

- It becomes time-consuming to calculate, with little benefit
- We start overfitting more and more.

The first part makes sense: with more terms, we have to do more multiplications, more additions, etc.

Concept 178

More **complex models** tend to be more **expensive** to train, and slower to use. This is a trade-off for more **accuracy**.

Usually, there's a point where cost **outweights** benefits. A problem is rarely perfectly solved, even by an excellent model, so you can't just continue until it's "perfect".

But what about the second part? Why do we increase overfitting?

With a higher order, our polynomial becomes more complex: it can take on more shapes, which are increasingly complex and perfectly fit to the data.

This can cause our data to overlook obvious patterns, and instead create a very precise shape that is paying attention to the noise in our model.

Concept 179

High-order polynomials are very vulnerable to **overfitting**.

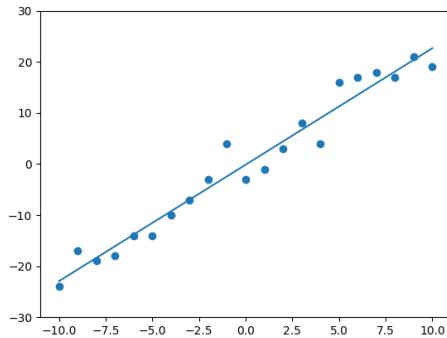
Because they can take on so many different, **complex** functions, they can very closely **match** the original data set.

This can cause the model to "learn" noise, and **miss** broader and simpler patterns that actually exist. It may fail to learn something broad and useful, while **memorizing** the dataset with its expressiveness.

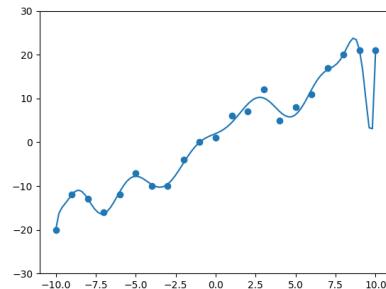
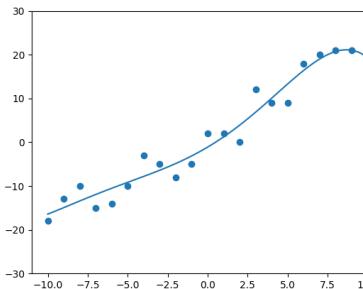
Let's see this in action: we'll generate some data based on $2x + 1$, while applying some random noise to it. We'll see the optimized linear regression model for each.

Rather than transform the data, we'll transform the separator: this really highlights the overfitting effect.

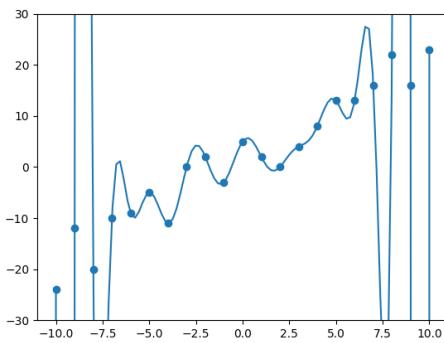
For ease, we'll exclude regularization: it does help mitigate this problem, but it doesn't totally solve it.



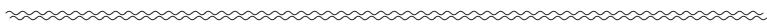
Here's the 1st order solution: in this case, correct for the underlying distribution. It fits our data fine.



5th and 15th order. The left model looks suspicious, and the right is way overfit. It's very unlikely that we know such an intricate pattern, from so little data.



20th order. We have one order for each data point: now, our model is capable of doing regression going through every single data point: as overfit as physically possible, perfectly matching the data.



5.2.1.3 Higher dimensions

Until now, we've only been focusing on the 1-D case of data. Let's change that. Let's consider a 2D dataset $[x_1, x_2]^T$.

Our typical model

$$\theta^T x + \theta_0 = \theta_1 x_1 + \theta_2 x_2 + \theta_0 \quad (5.7)$$

Is "order 1": the largest exponent is 1. This is still a "linear" model.

If we want to move up to order 2, we increase the max exponent, adding x_1^2 and x_2^2 to the basis.

However, this doesn't take full advantage of the expressiveness of our model: this only creates parabolas aligned with the x_1 and x_2 axes. How do we create other options?

Well, we created these options by multiplying x_1 with another x_1 . It seems like we could logically expand to multiplying x_1 by x_2 .

Definition 180

For **higher dimension** $d > 1$ **polynomials**, we allow for multiplication **between variables** x_i and x_j .

The **order** of the polynomial is the maximum number of times you can **multiply variables** together.

For order k , the **sum of exponents** must be **less than or equal to** the order.

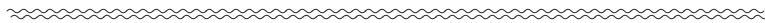
So, for $d = 2$, order=2, we get the basis:

$$\begin{bmatrix} 1 & x_1 & x_1^2 & x_2 & x_2x_1 & x_2^2 \end{bmatrix}^\top \quad (5.8)$$

For $d = 2$, $\text{order}=3$, it starts getting a bit messy: we have 10 different basis terms.

You don't need to memorize these.

$$\begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & x_2 & x_2x_1 & x_2x_1^2 & x_2^2 & x^2x_1 & x^3 \end{bmatrix}^\top \quad (5.9)$$



5.2.1.4 Total number of features

How many do we have in general? Well, every term results from multiplying variables **at most** k times. Or, the exponents **add up** to at most k .

If we count 1 as a factor, we can say that the exponents always add up to k (since 1^j has no effect). So, we have $d + 1$ different numbers, which add up to exactly k .

We have d variables, so $d + 1$ if we include 1 as a variable.

This is a well-known problem in combinatorics: how many ways are there to add up m numbers to total n ? The solution to this problem gives us:

$$\binom{(d+1)+k-1}{k} = \binom{d+k}{k} = \frac{(d+k)!}{d!k!} \quad (5.10)$$

Explaining the math here is beside the point of this course. If you're curious, search up "stars and bars math", or visit [here](#).

5.2.1.5 Summary of Polynomial Basis

Definition 181

The **polynomial basis** of order k and dimension d includes **every feature**

$$\prod_{i=1}^d x_i^{c_i} \quad (5.11)$$

Where all of the integer exponents $c_i \geq 0$ add up to **at most** k .

Creating features such as:

$$x_1^k, x_1x_2, x_2x_3^3x_6, 1, \dots \quad (5.12)$$

We can represent this in a table:

This table is different from the one we saw earlier!

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	$[1, x, x^2, x^3]^T$	$[1, x_1, \dots, x_1^3, x_1x_2, \dots, x_1x_2x_3, \dots]^T$
\vdots	\vdots	\vdots

5.2.1.6 Polynomial Basis in Action

Below, we'll show examples of how polynomial basis being used, to demonstrate just how useful it is.

But how do we use feature transformations? The best part: remember that we can view it as a linear separator? We can train it just the same way!

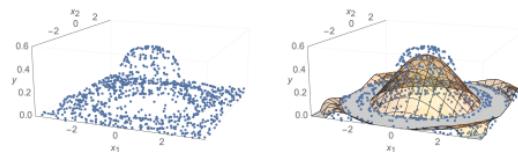
Concept 182

Feature transformations don't change how we train our model. We can still treat our model as a **linear** vector θ , even if our data has been **non-linearly** transformed.

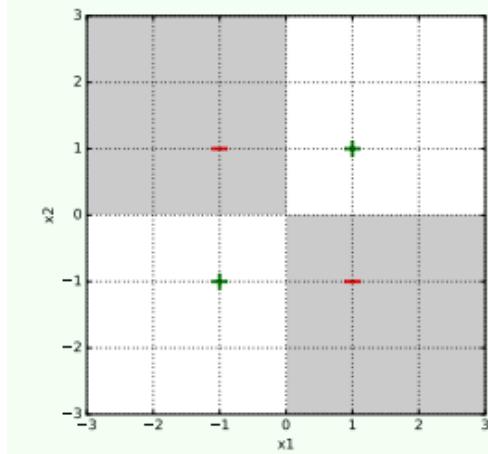
So, after we transform our data, we can use normal techniques (OLS, gradient descent, SGD) to fit our model.

In this situation, the benefits of regularization become more clear: by preventing θ from becoming too large, we discourage a surface that is too "extreme", with larger changes across its surface.

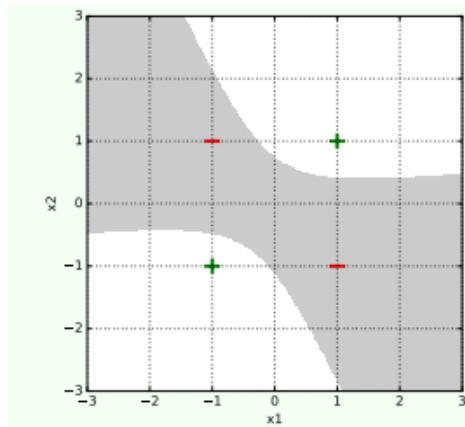
We'll train our model for various situations, to see what it can do. Different problems require different orders k , still.



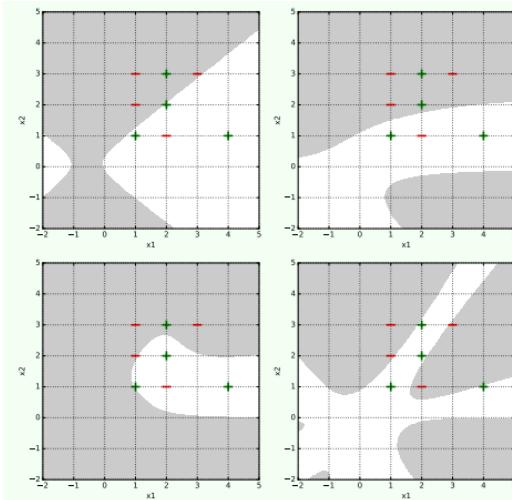
We start with the waveform we showed at the start of the chapter: on the left, we see a bunch of datapoints we want to take a regression over. With $k = 8$, we get a pretty good result.



This is the classic "xor" problem: a typical case of "linearly unseparable". With $k = 2$, we can classify it well with the chosen model $4x_1x_2 = 0$.



This time we use gradient descent and a random initialization to get a less 'perfect', but still effective classification.



This dataset is pretty brutal: we try with $k = 2, 3, 4$, and finally 5. The shapes we get are... complex, to say the least. But, we successfully solve it with $k = 5$.



5.2.2 Radial Basis

Finally, we consider an alternative way to create a feature space.

- With the "polynomial basis" approach, we **combined features** to create more complex surfaces to **fit** the structure of the data.
- This "radial basis" approach, on the other hand, **combines data points** to **learn** more about the structure of the data.

What do we mean by that? Well, let's consider what we mean by "structure": when we're judging data, what sorts of patterns are we looking for?

Often, we're looking to see, "what data is near/similar to other data?" Similar data is more likely to behave similarly, after all.

So, it might be useful to include distance between data points as a feature: how do we implement this? Well, let's do this one-by-one: we'll create a feature for the distance to a single data point p .

We'll come back to these ideas when we talk about clustering!

We start with squared distance, for smoothness reasons.

$$\|p - x\|^2 \quad (5.13)$$

This feature would *grow* as data points get further apart, though. We want to see what data is *close*: the opposite.

We could use a function like $\frac{1}{u}$. However, this would explode to infinity as distances shrink: not good.

e^{-u} is a better fit: it approaches 1 when $u = 0$, and, relatedly, it tends to drop off more smoothly and gradually than $1/u$.

Finally, we add a coefficient β to the exponent to give us more control: it will tell us how quickly our function decays with distance.

The word "decay" is used commonly for exponential decrease.

Definition 183

We define the **radial basis function**

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

As a **feature** in the RBF feature transformation.

This transformation takes a data point p and provides a feature $f_p(x)$ that represents "**closeness**" of x to p .

Note some useful properties of this transform:

- For small distances, this feature creates a **connection** between p and our data point: representing some local "structure" of **closeness**.
- If points are far away, this effect gradually **vanishes**: points which are **far** away have very little to do with each other.
- β controls what is considered "close" and "far":
 - if β is large, points have to be very close for an effect.
 - if β is small, we have a larger "neighborhood" of points with a relevant $f_p(x)$.

Definition 184

The **radial basis functions (RBF)** transform takes each of the data points in the input, and uses it to create a set of **radial basis function** features.

Collectively, they make the **feature space**:

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

Where:

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

This transform allows us to represent "closeness" within our dataset. With it, we can compare new data points to some "reference" points $x^{(i)}$.

It's often used to allow us to represent our dataset in a way that is approximate, but still useful.

This general idea is useful for problems like:

- Function approximation,
- Optimization,
- Reducing noise in signals

This approach is not limited to the "squared distance" idea of closeness, either: if you can come up with another way to define distance, you can use the same approach.

Reminder that "noise" just refers to anything undesired in the signal. Usually added by randomness or the environment.

These ways to define distance are called "distance metrics".

5.3 Hand-constructing features for real domains

So far, we've focused on transformations that handle two of our main problems (which have a lot of overlap): _____

- Allow our model to handle new, more **complex** situations (more **expressiveness**)
- **Pre-process** our data to make it **easier** for our model to find **patterns**.

Borrowed from the transformation definition above.

Now, we'd like to turn our attention to the last of the three:

- Convert our data into a **usable** format (if, say, the original format doesn't fit into our equations)

One challenge with our models are they rely on computation and calculation. This usually require our input to be something like a **number**.

But we don't always receive data in this way: words, brands, colors, and odd others data types, are often presented instead. Frequently, we even need to **adjust** our numerical inputs.

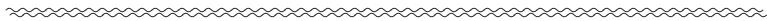
The **transformations** in this section address these kinds of problems. We take data that is informative, but not currently usable, and converting them to something we can **compute** with.

Concept 185

Often, we have to **convert** between data types, in order to do the machine learning work we want to do.

This requires using the appropriate **transforms** to get data we can work with, without **losing** important **information**.

As mentioned above, we have to be **careful**: if we use the wrong data type, we can **lose** crucial information that our model could have made use of.



5.3.1 Discrete Features

One of the most common issues with data types is figuring out what to do with **discrete** features: ones that are broken up into categories.

These categories may or may not have an order, or some other important information. We need to use the right data type to keep as much information as possible. This will allow our model to more easily discover patterns.

We'll make the following assumption:

Clarification 186

In this textbook/course, we assume that all **input vectors** x should be **real-valued vectors** (or: $x \in \mathbb{R}^d$)

And now, we go through some common examples of feature transformations:

5.3.1.1 Numeric

First, we consider the case where our pre-processing **feature** is "almost" in a number format: each class could reasonably correspond to a **number**.

Definition 187

In the **numeric** transformation, we convert each of our k classes into a **number**.

- This approach is only appropriate if each class is roughly "numeric": it fits appropriately into the **real numbers**.
 - We have a clear **ordering**, and
 - The numbers have the **structure** of real numbers: **distance** between points, or the idea of **adding/multiplying**, makes sense.

Example: There are many ways to do this. Here, we evenly distribute values evenly between 0 and 1: _____

$$\left[\frac{1}{k} \quad \frac{2}{k} \quad \dots \quad \frac{k-1}{k} \quad 1 \right], \quad \text{Class } i \longrightarrow \frac{i}{k} \quad (5.14)$$

Remember: which way you transform should reflect the nature of your data!

5.3.1.2 Thermometer Code

Next, we'll relax how number-like our feature is. This time, we don't need our data to behave like a number, but it does have an **ordering**. _____

Some examples:

- Results of an opinion poll:
 - "Strongly Agree", "Agree", "Neutral", "Disagree", "Strongly Disagree"
- Education level:
 - "Below High School", "High School Degree", "Associates Degree", "Bachelors" "Advanced Degree"
- Ranking of athletes

By "relax", we mean we'll remove some requirements for our feature, like being able to add them together.

In this case, we can't just use numbers {1, 2, 3, ...}. Why not?

Because that implies that there's a specific "scaling" between points: Is the #1 athlete twice as good as the #2 athlete? Maybe, but that's not what the ranking tells us!

Concept 188

Data that is **ordered** but not **numerical** cannot be represented with **a single real number**.

Otherwise, we might consider one element to be a certain amount "larger" or "smaller" than another, when that's not what **ordering** means.

Example: Suppose we assign {1, 2, 3} for {Disagree, Neutral, Agree}. The person who writes 'agree' is doesn't "agree three times as much" as the person who writes 'disagree'!

But, we still want to keep that ordering: counting up from one element to the next. How do we create an order without creating an exact, numeric difference?

Just now, we tried to count by increasing a single variable. But, there's another way to count: counting up using multiple different variables!

This approach is more similar to counting on your fingers.

$$\begin{array}{llll} \text{Class 1} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} & \text{Class 2} \rightarrow \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} & \text{Class 3} \rightarrow \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} & \text{Class 4} \rightarrow \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \end{array}$$

This version allows us to avoid the problems we had before: this doesn't behave the same way as a **numerical** value.

To better understand what's going on, here's another way to frame it:

Class(x) is just shorthand for, "which class is x in?"

$$\phi(x) = \begin{cases} \text{Class}(x) \geq 4 \\ \text{Class}(x) \geq 3 \\ \text{Class}(x) \geq 2 \\ \text{Class}(x) \geq 1 \end{cases} \quad (5.15)$$

Example: Suppose x is in class 3. The bottom three statements are all true, the top one is false: so we get $[0, 1, 1, 1]^T$.

This helps us understand why this encoding is so useful:

- We aren't directly "adding" variables to each other: they stay separated by **index**.
- When using a linear model $\theta^T \phi(x)$, each class matches a different θ_i .
- Despite not behaving like numbers, "higher" classes in the order still keep track of all of the classes below them.

θ_i scales the i^{th} variable. So, each class can be scaled differently!

- **Example:** Class 2-4 all share the feature $\text{Class}(x) \geq 2$ (equivalent to $\text{Class}(x) > 1$).

This technique is called **thermometer encoding**.

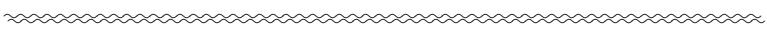
Definition 189

Thermometer encoding is a **feature transform** where we take each class and turn it into a feature vector $\phi(x)$ where

$$\begin{array}{ll} \text{Class 1} \longrightarrow \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ 1 \end{bmatrix}, & \text{Class 2} \longrightarrow \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 1 \end{bmatrix} \\ \text{Class 3} \longrightarrow \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 1 \\ 1 \end{bmatrix}, & \text{Class k} \longrightarrow \begin{bmatrix} 1 \\ \vdots \\ 1 \\ 1 \\ 1 \end{bmatrix} \end{array}$$

- The **length of the vector** is the **number of classes** k we have.
- The i^{th} class has i **ones**.
- This transformation is only appropriate if the data
 - Is **ordered**,
 - But not **real number-compatible**: we can't add the values, or compare the "amount" of each feature.

Example: We reuse our example from earlier:

$$\phi(x_{\text{Class 1}}) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \phi(x_{\text{Class 2}}) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}, \quad \phi(x_{\text{Class 3}}) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \quad \phi(x_{\text{Class 4}}) = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$


5.3.1.3 One-hot Code

We introduced this technique in the **previous chapter**:

When there's no clear way to **simplify** our data, we accept the current discrete classes, and **convert** them to a number-like form.

- Examples:

- Colors: {Red, Orange, Yellow, Green, Blue, Purple}
- Animals: {Dog, Cat, Bird, Spider, Fish, Scorpion}
- Companies: {Walmart, Costco, McDonald's, Twitter}

We can't use thermometer code, because that suggests a natural **order**. And we definitely can't use real numbers.

Example: {Brown, Pink, Green} doesn't necessarily have an obvious order: you could force one, but there's no reason to.

But, we can use one idea from thermometer code: each class in a different variable.

$$\begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_k \end{bmatrix} \quad (5.16)$$

But in this case, we don't "build up" our vector: we replace $\text{Class}(x) \geq 4$ with $\text{Class}(x) = 4$.

$$\phi(x) = \begin{bmatrix} \text{Class}(x) = 4 \\ \text{Class}(x) = 3 \\ \text{Class}(x) = 2 \\ \text{Class}(x) = 1 \end{bmatrix} \quad (5.17)$$

This approach is called **one-hot encoding**.

Definition 190

One-hot encoding is a way to represent **discrete** information about a data point.

Our k classes are stored in a length- k column **vector**. For **each** variable in the vector,

- The value is **0** if our data point is **not in that class**.
- The value is **1** if our data point is **in that class**.

$$\text{Class 1} \rightarrow \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \text{Class 2} \rightarrow \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \end{bmatrix}, \quad \text{Class 3} \rightarrow \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 0 \\ 0 \end{bmatrix}, \quad \text{Class } k \rightarrow \begin{bmatrix} 1 \\ \vdots \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

In one-hot encoding, items are **never** labelled as being in **two** classes at the **same time**.

- This transformation is only appropriate if the data is
 - Does not have another **structure** we can reduce it to: it's neither like a **real number** nor **ordered**
 - We don't have an **alternative** representation that contains more (accurate) information.

Example: Suppose that we want to classify **furniture** as table, bed, couch, or chair.

$$\begin{bmatrix} \text{table} \\ \text{bed} \\ \text{couch} \\ \text{chair} \end{bmatrix} \quad (5.18)$$

For each class:

$$y_{\text{chair}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad y_{\text{table}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad y_{\text{couch}} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad y_{\text{bed}} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad (5.19)$$

5.3.1.4 One-hot versus Thermometer

One common question is, "why can't we use one-hot for ordered data? We could sort the indices so they're in order".

However, there's a problem with this logic: the computer **doesn't care** about the order of the variables in an array: it contains no information!

Why is that? If the vector has an order, shouldn't that **affect** the model?

Well, remember that our model is represented by

$$\theta^T x = \sum_i \theta_i x_i \quad (5.20)$$

The vector format $\theta^T x$ is just a way to **condense** our equation: the ordering goes away when we compute the sum.

Concept 191

Order of elements in a vector **don't** affect the behavior of our model.

This is because a linear model is a **sum**, and sums are the same regardless of **order**.

If our model has the same math regardless of order, then it can't use order information.

Example: We'll take a vector, and rearrange it.

Despite shuffling, these two equations are equivalent!

$$\theta^T \phi(x) = \begin{bmatrix} \theta_1 & \theta_2 & \theta_3 & \theta_4 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \longrightarrow (\theta^T)^* (\phi(x))^* = \begin{bmatrix} \theta_3 & \theta_1 & \theta_4 & \theta_2 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

The math is the same, despite changing order: our model knows nothing about ordering.

Clarification 192

One hot encoding **cannot** encode information about ordering.

Thermometer encoding is required to **represent ordered objects**.

Why is thermometer encoding able to represent ordering? Let's try shuffling it, too.

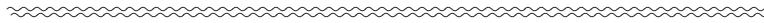
$$\theta^T \phi(x) = \begin{bmatrix} \theta_1 & \theta_2 & \theta_3 & \theta_4 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad (5.21)$$

$$(\theta^T)^* (\phi(x))^* = \begin{bmatrix} \theta_3 & \theta_1 & \theta_4 & \theta_2 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad (5.22)$$

Even though we've changed the order, we still know this is the **third** in the order, because we have **three 1's**!

Concept 193

Even though the **order of elements** in a vector **doesn't matter**, we can retrieve the order of **thermometer coding** based on the **number of 1's in the vector**.



5.3.1.5 Factored Code

Now, we move away from number-like properties. Instead, what other patterns of our feature could be **useful**?

Sometimes, a single feature will contain multiple pieces of information. Separating those pieces (or **factors**) from each other can make it easier for our machine to understand.

- A **car** is often described by the "make" (brand) and "model" (which exact type of car by that brand).
 - These could be broken into two **features**: "make" is one feature, "model" is another.
 - **Example:** "Nissan Altima" becomes "Make: Nissan" and "Model: Altima".
- Most **blood types** are in the following categories: {A+, A-, B+, B-, AB+, AB-, O+, O-}.
 - You could factor this based on the letter, and positive/negative: {A, B, AB, O} and {+,-}.
 - Since "O" means we contain neither A nor B, we could factor the first feature further: {A, not A}, {B, not B}
 - Example: Using the first factoring, A- becomes [A, -]. Using the second it becomes [A, not B, -].
- **Addresses** have many parts: street number, zip code, state, etc.
 - Each of these can be given their own factor.

Definition 194

Factored code is a **feature transformation** where we take one **discrete class** and break it up into other discrete classifications, called **factors**.

$$\text{Class } m \text{ and } n \longrightarrow \text{Class } m, \text{Class } n \quad (5.23)$$

- This transformation is only appropriate if
 - We have some feature(s) that can be **broken up** into **simpler**, meaningful parts.

Often, we apply **other** feature transformations after factored coding.

Note the final comment: often, we turn a discrete class into multiple new discrete classes.

But, we still need to convert these into a usable, numeric-vector form!

Example: We can re-use our blood type example from above.

$$\phi(x) = \begin{bmatrix} x \text{ contains A} \\ x \text{ contains B} \\ x \text{ is +} \end{bmatrix} \quad (5.24)$$

Each of these are binary features. For example:

$$\phi(AB-) = \begin{bmatrix} \text{True} \\ \text{True} \\ \text{False} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix} \quad (5.25)$$

5.3.1.6 Binary Code

One possible way to encode data is to **compress** data using a **binary code**.

This might be tempting, because k values can be represented by $\log_2(k)$ values.

Example: Suppose you have the one-hot code for 6, and want to represent it with binary:

$$\text{Class 6} \xrightarrow{\text{One-hot}} [0 \ 1 \ 0 \ 0 \ 0 \ 0]^T \xrightarrow{\text{Binary}} [1 \ 1 \ 0]^T \quad (5.26)$$

Please do not do this

Concept 195

Using **binary code** to compress your features is usually a **bad idea**.

This forces your model to spend resources learning how to **decode** the binary code, before it can do the task you want it to!

5.3.2 Text

Just now, we showed different ways to transform **discrete** features.

Another very common data type we work with is **language**: bodies of text, online articles, corpora, etc.

Later in this course, we will discuss more powerful ways to analyze text, such as **sequential models**, and **transformers**.

Obligatory chatgpt reference.

There's a very simple encoding that we'll focus on here: the **bag of words** approach.

This approach is meant to be as simple as possible: for each word, we ask ourselves, "if this word in the text?", and answer yes (1) or no (0) for every single word.

Definition 196

The **bag of words** feature transformation takes a body of text, and creates a **feature** for every **word**: is that word in the text, or not?

$$\phi(x) = \begin{bmatrix} \text{Word 1 in } x \\ \text{Word 2 in } x \\ \vdots \\ \text{Word k in } x \end{bmatrix} \quad (5.27)$$

This approach is used for **bodies of text**.

Example: Consider the following sentence: "She read a book."

With the words: {She, he, a, read, tired, water, book}

We get:

$$\phi(\text{"She read a book."}) = [1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1] \quad (5.28)$$

A couple weaknesses to this approach:

- Ignores the order of words and syntax of the sentence.
- Doesn't encode meaning directly.

- Duplicate words are only included once.
- It doesn't create much structure for our model to use.

But, it's very easy to implement.

5.3.3 Numeric values

Now, on to the (typically) more manageable data type:

Concept 197

Typically, if your feature is **already a numeric value**, then we usually want to **keep it as a data value**.

Example: Heart rate, stock price, distance, reaction time, etc.

However, this may not be true if there is some difference between different ranges of numbers:

- Being below or above the age of 18 (or 21) for legal reasons
- Temperature above or below boiling
- Different age ranges of children might need different range sizes: the difference between ages 1-2 is very different from ages 7-8.

Concept 198

Sometimes, if there are distinct **breakpoints**/boundaries between different values of a numerical feature, we might use **discrete** features to represent those.

5.3.3.1 Standardizing Values

We still aren't done, if our data is numeric. We likely want to **scale** our features, so that they all tend to be in similar ranges.

Why is that? If some features are much **larger** than others, then they will have a much larger impact on the answer.

For example, suppose we have $x_1 = 4000$, $x_2 = 7$:

$$h(x) = \theta^T x = 4000\theta_1 + 7\theta_2 \quad (5.29)$$

The first term is going to have a way bigger impact on $h(x)$. If we change x_1 by 10%, that's going to be bigger than if we changed x_2 by 100%!

$4000 * 10\% = 400$
$7 * 100\% = 7$

Concept 199

If one **feature** is much **larger** than **another** feature, it will tend to have a much **larger** effect on the result.

This is often a bad thing: just because one feature is **larger**, doesn't mean it's more **important**!

Example: Income might be in the range of tens of thousands (10,000-100,000), while age is a two-digit number(20-100). Income will be weighed more heavily.

How do we solve that problem? We need to do two things:

- **Shift** the data so that our range is not too high/low. Our goal is to have it centered on 0.
 - We want it centered on 0 so we can distinguish between the above-average and below-average data points.
 - We do this by subtracting the **mean**, or the **average** of all of our data points.

Plus, it's easier to get all of our data to 0, rather than picking some arbitrary value.

$$\phi_1(x) = x - \bar{x} \quad (5.30)$$

- Scale the **range** of possible values, so they all vary by roughly the same amount.
- So, if one variable tends to vary by a **larger** amount, it doesn't have a bigger impact on the result.

$$\phi(x_i) = \frac{x_i - \bar{x}_i}{\sigma_i} \quad (5.31)$$

Where σ is the **standard deviation**.

If you are interested, we define **standard deviation** below.

Note that each feature has its own σ_i : we have to compute this equation for each feature.

Definition 200

To make sure that all of our data is **on the same size scale**, we **normalize/standardize** our dataset using the operation

$$\phi(x_i) = \frac{x_i - \bar{x}_i}{\sigma_i}$$

For every variable x_i in a data point x .

- \bar{x}_i is the **mean** of x_i
- σ_i is the **standard deviation** of x_i

This results in a dataset which has

- A mean \bar{x}_i of **0**
- A standard deviation σ_i of **1**

So, all of our features have the same **average**, and **vary** by the same amount.

This prevents some features getting prioritized because they're on different size scales.

Example: Suppose we have 1-D data $x = [1, 2, 3, 4, 5, 6]$

The mean is

$$\bar{x} = \frac{1 + 2 + 3 + 4 + 5 + 6}{6} = 3.5 \quad (5.32)$$

And the standard deviation is

$$\sigma = \sqrt{\frac{2.5^2 + 1.5^2 + .5^2 + .5^2 + 1.5^2 + 2.5^2}{6}} = \sqrt{\frac{35}{12}} \approx 1.7078 \quad (5.33)$$

5.3.3.2 Variance and Standard Deviation (Optional)

This section describes the origin of σ above. Feel free to skip if you're familiar.

In order to scale our data, we need a measure of how much our data **varies**. So, if our data varies by more, we can scale it down, and vice versa.

We can measure this using the **variance**.

Definition 201

We can measure how spread out/varying our data with **variance**

$$\sigma^2 = \sum_i \frac{(x^{(i)} - \bar{x})^2}{n} \quad (5.34)$$

In other words, the **average squared distance** from the **mean**.

Why do we square the terms? Same reason we square our loss:

- We want only positive values, for distance.
- We don't want to use absolute value, for smoothness.

However, this is too large: we want something similar to "average distance from the mean".

This is the average **squared** distance.

We also get nicer statistical properties we won't discuss here.

So, we take a square root!

Definition 202

A more common way to measure how our data varies is using **standard deviation** σ

$$\sigma = \sqrt{\sigma^2} = \sqrt{\sum_i \frac{(x - \bar{x})^2}{n}}$$

This term is **not** the average distance from the mean, but can be used for **scaling** our data in the same way.

This term allows us to scale our data appropriately. If our data varies by a larger amount, σ will be larger. So, $\frac{1}{\sigma}$ will cancel that variance out.

5.4 Terms

- Non-linear
- Transformation
- Feature
- Feature Transformation
- Polynomial Basis
- Order/Degree of a polynomial
- Radial Basis
- Discrete Feature
- Numeric transformation
- Thermometer Code
- One-hot Code
- Factored Code
- Binary Code
- Bag of Words
- Standardization
- Normalization
- Standard Deviation

CHAPTER 6

Neural Networks 1 - Neurons, Layers, and Networks

The tools we've developed so far are interesting, and **varied**. We've discussed:

- **Regression**: the problem of creating **real-number** outputs based on data.
- **Classification**: the problem of **sorting** data points into **categories**.
- Gradient **descent**: A technique for gradually **improving** your model using **calculus**.

These concepts are fascinating in their own right, and can be used to handle some **simple** problems. But, when they are **combined** together, we get something much more **powerful**: **neural networks**.

6.0.1 Machine Learning Applications

Neural networks in the modern area are used to tackle complex and challenging problems:

- Image labelling and generation
 - **Example**: Recognizing a picture of a dog. Or, creating a picture of a dog when prompted.
- Physics simulation
 - **Example**: Simulating water flow realistically, or special-effects smoke for a movie.
- Financial prediction

- **Example:** Predicting how the **market** moves over time, and what the best **financial** choices in the present are.
- Text processing and generation
 - **Example:** Creating machines that can understand human text **prompts**, and writing useful **explanations** for humans.
- Data analysis
 - **Example:** **Compressing** data, or processing it to isolate the **important** aspects, without the noise.

As you can see, **neural networks** are used in a wide array of very **difficult** problems. No wonder it's rapidly becoming so popular!

6.0.2 Neural Network Perspectives: The brain

So, what *is* a neural network? Well, there's a **couple** ways of looking at it:

First, the name comes from the fact that NNs are inspired by the **brain**: we call the basic units of a neural network a **neuron**.

This gives us some general idea of the **structure** of a neural network:

Just like in the **brain**, we take many individual units, called **neurons**, which we connect together to do more **complicated** tasks. That combined structure is a **neural network**.

Concept 203

Neural networks are inspired by the brain and its **neurons**, in an effort to do better, **human-like** computation.

Based on this, neural networks are **built** out of simple **units** called **neurons**, connected to each other.

Funny enough, as effective as neural networks are, we now think they don't work very much like the human brain! But we keep the terminology.

6.0.3 Neural Network Perspectives: Classification and Regression

In this class, we **won't** focus on the brain analogy, though it did inspire the model.

Instead, we will mostly think of **neural networks** in terms of what they're able to do, and how they work.

One problem we have struggled with is certain tasks that can't be handled by **linear** models. We have used **feature representations** to work on this problem.

Simply, some problems are outside our **hypothesis** space. But, there's another way: this is where **neural networks** come in.

By combining lots of simple **units** ("neurons"), we can get a very **complex** model for solving our problems.

With such a **rich** hypothesis class, combined with the power of **gradient descent**, we can create a model that can do **classification** or **regression** for very difficult problems!

Concept 204

Neural Networks can create a very **rich hypothesis class** by combining many simple **units**.

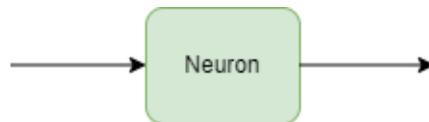
With this **hypothesis class**, we can handle **regression** or **classification** for very challenging **problems**.

Reminder: "richness" or "expressiveness" of a hypothesis reflect how wide our options are. Neural networks give us many possibilities for models. With more options, we can handle more problems!

6.0.4 Building up a basic neural network

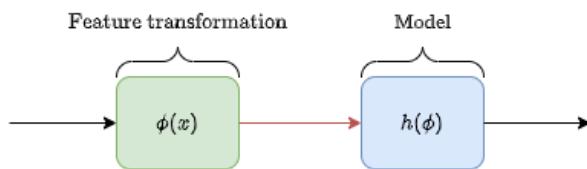
Let's make sense of what we said above, and **visualize** what a neural network might look like.

We start with one function: a **neuron**. This function could be, for example, one we've used before: our logistic **classifier**, or linear **regression**. We'll ignore the details for now.



One neuron might not be very powerful, or **expressive**. It's useful, but limited. We've seen its weaknesses.

We could try to use **feature transformations** to help us. But, let's think in a more **general** way: a transformation is just another **function** we apply to our input!



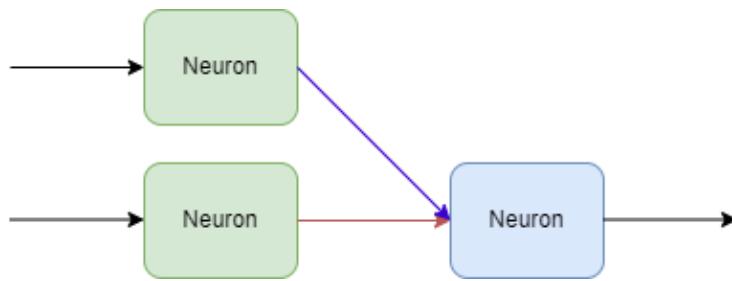
This gives us an **idea**: rather than trying to think of a single, more **complex** model, we could **combine** multiple simple models!



Note that feature transformations are a bit complex for what we'd usually put in a neuron. But, it gives us the right inspiration.

We could repeatedly add more neurons in **series**: each one being the input to another. And we'll do that later!

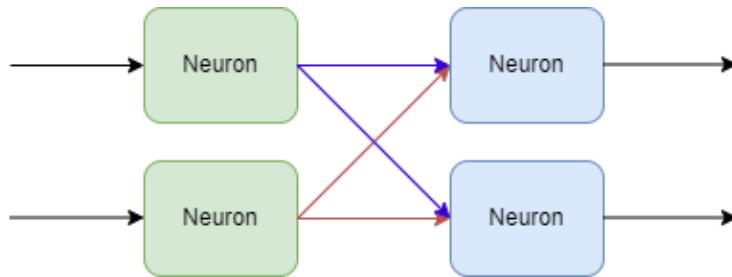
But, there's another type of **complexity** we haven't explored: we could have two neurons in **parallel**.



This parallel/series vocabulary is borrowed from circuits. We'll just use it for demonstration: you don't need to remember it.

Now, we have **two** neurons feeding into one output neuron! This already looks like a more **complicated** model.

We can go even further: what if we have two outputs as well?



Because we had two **inputs**, we had to add two new **links** when we added the output neuron. This is getting difficult to **view**!

We'll stop here for now, but you can imagine repeatedly **adding** more neurons in **parallel** (with the same inputs/outputs) or in **series** (as an input or output).

And with each addition, the function gets more and more **complex**: you can create a **richer** hypothesis class!

We'll explore how to do this **systematically** later in the chapter.

By "systematic", we just mean "in a way that's consistent and makes sense".

Definition 205

Neural Networks are a **class** of models that can be used to solve **classification**, **regression**, or other interesting problems.

They create very **rich** hypothesis classes by combining many **simple** models, called **neurons**, into a **complex** model.

We do this combination **systematically**, so that it is easy to **analyze** and work with our **model**.

This creates a very **flexible** hypothesis, which can be **broken down** into its **simple** parts and what **connects** them.

6.0.5 Neural Network Perspectives: Predictions with Big Data

Our last major **perspective** on neural networks is one that you see in lots of modern **applications**. We won't work much with this perspective in this **class**, but our techniques **enable** it.

Neural networks, because they can create such **sophisticated** models, can be used for problems in very **complex** domains: the kind of **applications** we discussed at the beginning of this chapter.

These applications require a lot of **data** to build a good **model**, however. So, machine learning models often take **huge** amounts of data, with lots of energy and time to train them.

But, once they are fully **trained**, they can give predictions very **quickly**, and often very **accurately**.

Concept 206

Neural networks can be seen as a way to make **predictions** based on huge amount of **data** for very **complex** problems.

6.1 Basic Element

Now, we have idea of what neural networks **are**. But, we have yet to handle the **details**:

- What **is** a neuron?
- How do we "systematically" **combine** our neurons?
- How do we **train** this, like we would a **simple** model?

We'll handle all of these steps and more - the above description was just to give a **high-level** view of what we want to **accomplish**.

Now, we go down to the **bottom** level, and think about just **one neuron**: what does it look like, and how does it work?

First, some terminology:

Notation 207

Neurons are also sometimes called **units** or **nodes**.

They are mostly **equivalent** names. They just reflect different **perspectives**.

6.1.1 What's in a neuron: The Linear Component

As we mentioned before, our goal is to combine **simple** units into a **bigger** one. So, we want a model that's **simple**.

Well, let's start with what we've done before: we've worked with the **linear** model

$$h(x) = \theta^T x + \theta_0 \quad (6.1)$$

This model has lots of nice properties:

- It limits itself to **addition** and **multiplication** (easy to compute)
- **Linearity** lets us prove some mathematical things, and use vector/**matrix** math
- The dot product between θ and x has a nice **geometric** interpretation.

This will make up the **first** part of our model.

Concept 208

Our **neuron** contains a **linear** function as its **first** component.

6.1.2 Weights and Biases

But, there's one minor **change**: before, we used θ because it represented our **hypothesis**.

But, every neuron is going to have its own **values** for its **linear** model:

$$\begin{array}{ll} \text{Neuron 1} & \text{Neuron 2} \\ \overbrace{f_1(x)} = Ax + B & \overbrace{f_1(x)} = Cx + D \end{array} \quad (6.2)$$

It wouldn't make much **sense** to call both A and C by the name θ .

We could use some clever **notation**, but why treat them as **hypotheses**? They are each only a **part** of our hypothesis Θ .

So, instead of thinking of each as a "hypothesis", let's switch perspectives.

Each value θ_k **scales** how much x_k affects the **output**: if we're doing

$$g(x) = 100x_1 + 2x_2 \quad (6.3)$$

Then, changing x_1 will have a much **bigger** effect on $g(x)$. Another way to say this is it **weighs** more heavily: it matters **more**.

Because of that, we call the number we scale x_1 by a **weight**.

Notation 209

A **weight** w_k tells you how heavily a **variable** x_k **weighs** into the output.

w_k is **equivalent** to θ_k : it's a **scalar** $w_k \in \mathbb{R}$.

$$(\theta_1 x_1 + \theta_2 x_2) \iff (w_1 x_1 + w_2 x_2)$$

We can combine it into a vector $w \in \mathbb{R}^m$.

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} \quad \theta^T x \iff w^T x$$

What about our other term, θ_0 ? We call it an **offset**: it's the value we **shift** our linear model away from **origin**.

Remember that $a \iff b$ means a and b are equivalent!

We'll use the same notation:

Notation 210

An **offset** w_0 tells you how far we **shift** $h(x)$ away from the origin.

w_0 is **equivalent** to θ_0 : it's a **scalar** $w_0 \in \mathbb{R}$

$$((\theta^T x) + \theta_0) \iff ((w^T x) + w_0)$$

We also sometimes call this the **threshold** or the **bias**.

Alternate notation: we might call this variable **b**, for bias.

This gives us our linear model using our new notation:

Definition 211

The **linear component** for a neuron is given by

$$z(x) = w^T x + w_0$$

where $w \in \mathbb{R}^m$ and $w_0 \in \mathbb{R}$.

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$$

6.1.3 Linear Diagram

Now, we want to be able to depict our **linear** subunit. Let's do it piece-by-piece.

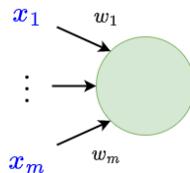
First, we have our vector $x = [x_1, x_2, \dots, x_m]^T$:

x_1

\vdots

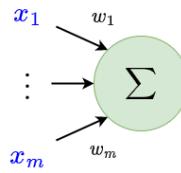
x_m

Now, we want to **multiply** each term x_i by its corresponding **weight** w_i . We'll combine them into a **function**:



The circle represents our function.

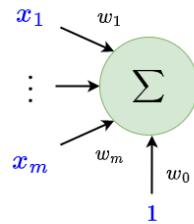
How are we combining them? Well, we're adding them together.



Note that we use the \sum symbol, because we're **adding** after we **multiply**. In fact, we can write this as

$$\mathbf{w}^T \mathbf{x} = \sum_{i=1}^m w_i x_i \quad (6.4)$$

We'll include the bias term as well: remember that we can represent w_0 as $1 * w_0$ to match with the other terms.

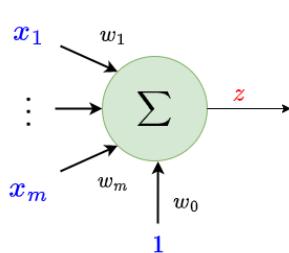


The blue "1" term is **multiplied** by w_0 , just like how x_k gets multiplied by w_k .

We have our full function! All we need to do is include our output, z :

Notation 212

We can depict our linear function $z = \mathbf{w}^T \mathbf{x} + w_0$ as



Thus, z is a function of x :

$$z(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0 \quad (6.5)$$

Which, in \sum notation, we could write as

$$\textcolor{red}{z}(\textcolor{blue}{x}) = \left(\sum_{i=1}^m w_i \textcolor{blue}{x}_i \right) + w_0 \quad (6.6)$$

6.1.4 Adding nonlinearity

We'll continue building our neuron based on what we've done **before**. When doing linear regression, that linear unit was all we had.

But, once we do classification, we found that it was helpful to have a second, **non-linear** component: we used **sigmoid** $\sigma(u)$.

We might not necessarily want the **same** nonlinear function, so instead, we'll just generalize: we have *some* second component, which is allowed to be **nonlinear**.

We call this component our **activation** function. Why do we call it that? It comes from the historical **inspiration** of neurons in the brain.

Biological neurons only "fire" (give an output) above a certain threshold of **input**: that's when they **activate**.

Some activation functions reflect this, but they don't have to.

Definition 213

Our **neuron** contains a potentially **nonlinear** function f called an **activation function** as its **second** component.

We note this as

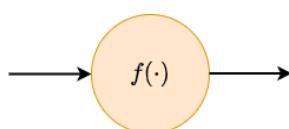
$$\textcolor{blue}{a} = f(\textcolor{red}{z}) \quad (6.7)$$

Where z is the **output** of the **linear** component, and a is the **output** of the **activation** component.

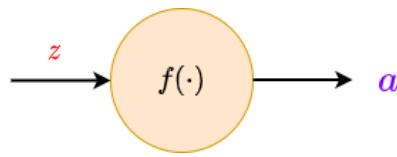
Note that z and a are **real numbers**: we have $f : \mathbb{R} \rightarrow \mathbb{R}$

6.1.5 Nonlinear Diagram

We'll depict a function f .



It takes in our **linear** output, z , and outputs our **neuron** output, a .



Note some vocabulary used for z :

Notation 214

z , the **output** of our **linear** function, is called the **pre-activation**.

This is because it is the result **before** we run the **activation** function.

And for a :

Notation 215

a , the **output** of our **activation** function, is called the **activation**.

6.1.6 Putting it together

So now, our neuron is complete.

Definition 216

Our **neuron** is made of

- A **linear** component that takes the neuron's input x , and applies a linear function

$$z = w^T x + w_0$$

- The **pre-activation** z is the **output** of the **linear** function.
- It is also the **input** of the **activation function** f .

- A (potentially nonlinear) **activation** component that takes the pre-activation z and applies an **activation function** f :

$$a = f(z)$$

- The **activation** a is the **output** of this **activation function**.

When we **compose** them together, we get

$$a = f(z) = f(w^T x + w_0)$$

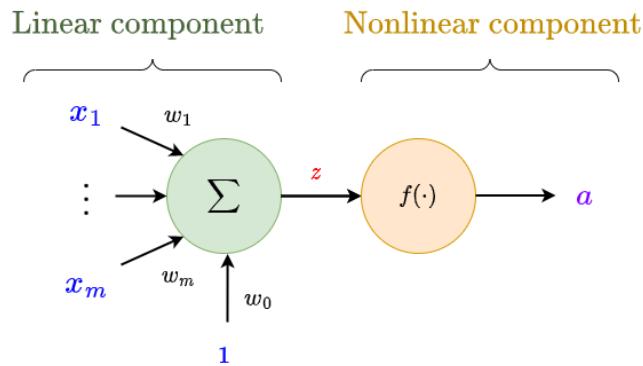
We can also use \sum notation to get:

$$a = f(z) = f\left(\left(\sum_{i=1}^m w_i x_i\right) + w_0\right)$$

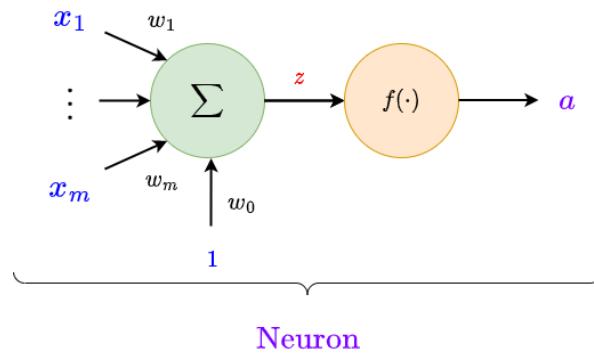
When we say "compose", we mean **function composition**: combining $f(x)$ and $g(x)$ into $f(g(x))$.

6.1.7 Neuron Diagram

Finally, we can **compose** our neuron into one big **diagram**:

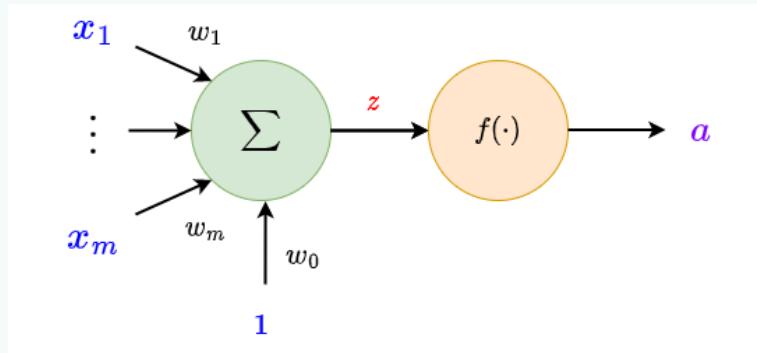


From here on out, we'll treat this as a **single** object:



Notation 217

We can depict our **neuron** $f(w^T x + w_0)$ as



- x is our **input** (neuron input, linear input)
- z is our **pre-activation** (linear output, activation input)
- a is our **activation** (neuron output, activation output)

This neuron will be the **basic unit** we work with for the rest of this **chapter** - it's one of the most **important** objects in all of machine learning.

6.1.8 Our Loss Function

One more detail: we will want to **train** these neurons. In order to be able to **measure** their performance, we'll need a **loss** function.

This **isn't** any different from usual: we just need a **function** of the form

$$\mathcal{L}(g, y) \tag{6.8}$$

In **regression**, we wrote our loss as

$$\mathcal{L}\left(h(x; \Theta), y \right)$$

The right term, $y^{(i)}$, is unchanged: we still need to compare against the **correct** answer.

The main change is we aren't using Θ notation: we'll **replace** it with (w, w_0)

$$\mathcal{L}\left(h(x; (w, w_0)), y \right)$$

And finally, we get the loss for multiple data points: _____

We skip doing $1/n$ averaging because we often use this for SGD: we plan to take small steps as we go, rather than adding up our steps all at once.

$$\sum_i \mathcal{L} \left(h(x^{(i)}; (w, w_0)), y^{(i)} \right)$$

And with this, not only is our neuron **complete**, but we have everything we need to **work** with it.

Concept 218

For a **complete neuron**, we need to specify

- Our **weights** and **offset**
- Our **activation** function
- Our **loss** function

From here, we could do **stochastic gradient descent** as we usually do, to **optimize** this neuron's **performance**.

6.1.9 Example: Linear Regression

Let's go through some **examples**. We mentioned in the **beginning** of this chapter that our neuron could be most of the simple **models** we've worked with.

So, let's give that a go: we'll start by doing **linear regression**.

$$h(x) = \theta^T x + \theta_0$$

This model is exclusively **linear**: we just have to replace θ with w .

$$z(x) = w^T x + w_0$$

So, our linear component is **done**: $(\theta, \theta_0) = (w, w_0)$.

What about our **activation** function?

Well, activation allows for **nonlinear** functions. But, we don't **want** to make it nonlinear.

In fact, we've already got what we **want**: we don't want the **activation** to do anything at all.

So, we'll use **this** function:

Concept 219

The **identity function** $f(z)$ is a function that has no **effect** on your **input**.

$$f(z) = z$$

By "having no effect", we mean that the input is **unchanged**: this is true even if your input is **another function**:

$$f(g(x)) = g(x) \quad (6.9)$$

So, the **identity** function is our activation function: it keeps our **linearity**.

We call it the "identity" because the input's identity is unchanged!

Concept 220

Linear Regression can be represented with a **single neuron** where

- We keep our **linear component**, but set $(\theta, \theta_0) = (w, w_0)$.

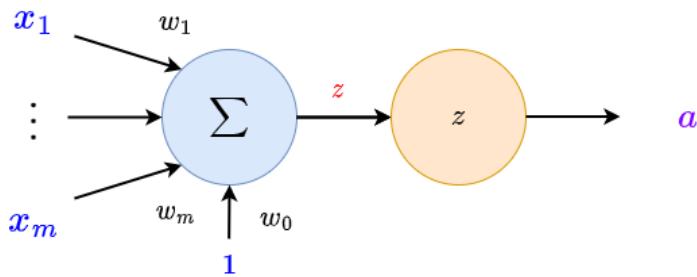
$$z(x) = w^T x + w_0$$

- Our **activation function** is the **identity** function,

$$f(z) = z$$

- Our **loss function** is **quadratic loss**.

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



6.1.10 Example: Linear Logistic Classifiers

Now, we do the same for LLCs: it's already broken up into **two** parts in our **classification** chapter.

First, the **linear** component. This is the same as linear regression:

$$\textcolor{red}{z} = \theta^T x + \theta_0 \quad (6.10)$$

And then, the **logistic** component:

$$\sigma(\textcolor{red}{z}) = \frac{1}{1 + e^{-\textcolor{red}{z}}} \quad (6.11)$$

This second part is **nonlinear**: it's our **activation** function!

Concept 221

A **Linear Logistic Classifier** can be represented with a **single neuron** where

- We keep our **linear component**, but set $(\theta, \theta_0) = (w, w_0)$.

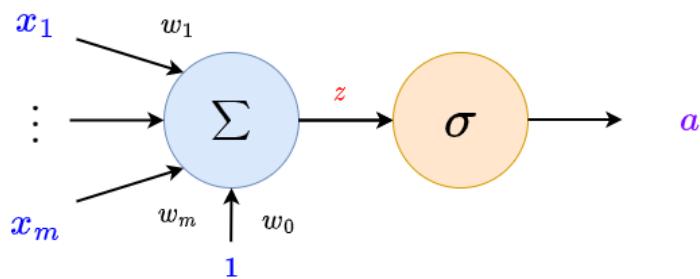
$$\textcolor{red}{z}(x) = w^T x + w_0$$

- Our **activation function** is the **sigmoid** function,

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

- Our **loss function** is **negative-log likelihood** (NLL)

$$\mathcal{L}_{\text{nll}}(\textcolor{violet}{a}, \textcolor{blue}{y}^{(i)}) = - \left(\textcolor{blue}{y}^{(i)} \log \textcolor{violet}{a} + (1 - \textcolor{blue}{y}^{(i)}) \log (1 - \textcolor{violet}{a}) \right)$$



6.2 Networks

Now, we have fully developed the individual **neuron**.

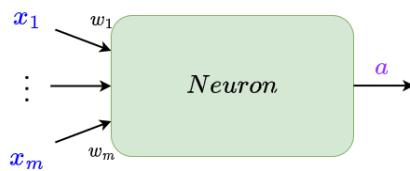
We can even do **gradient descent** on it: just like when we were doing LLCs, we can use the **chain rule**.

So, we return to the idea from the beginning of this chapter: combining multiple neurons into a **network**.

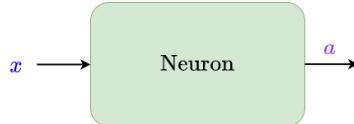
We'll get into this more, later in the chapter.

6.2.1 Abstraction

For this next section, we'll **simplify** the above diagram to this:



In fact, for more **simplicity**, we'll draw **one** arrow to represent the whole vector x . However, nothing about the **actual** math has changed.



This is also called **abstraction** - we need it a lot in this chapter.

Definition 222

Abstraction is a way to view your system more **broadly**: removing excess details, to make it **easier** to work with.

Abstraction takes a **complicated** system, and focuses on only the **important** details. Everything else is **excluded** from the model.

Often, this **simplified** view boils a system down to its the **inputs** and **outputs**: the "interface".

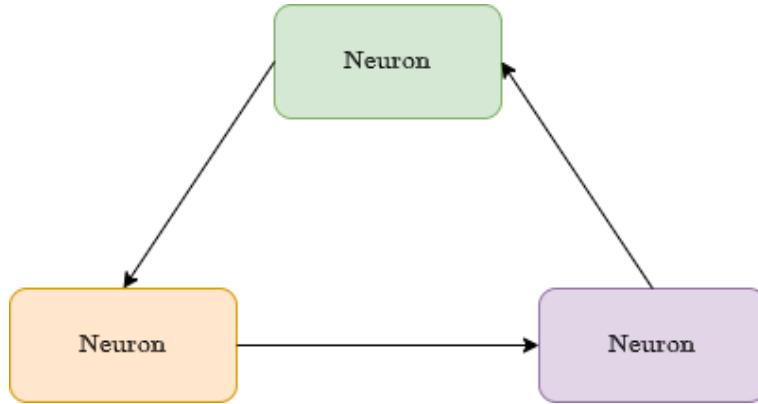
Example: Rather than thinking about all of the **mechanics** of how a car works, you might **abstract** it down to the pedals, the steering wheel, and how that causes the car to move.

6.2.2 Some limitations: acyclic networks

We won't allow for just **any** kind of network: we can create ones that might be unhelpful, or just very **difficult** to **analyze**.

For now, we can get interesting and **useful** behavior while keeping it **systematic**. We'll define this "system" later.

We'll assume our networks are **acyclic**: they do not create closed **loops**, where something can affect its own input.



This is a cyclic network: this is messy and we won't worry about this for now.

This means information only **flows** in one direction, "forward": it never flows "backwards".

Concept 223

For simple **neural networks**, we assume that they are **acyclic**: there are no **cycles**, or loops.

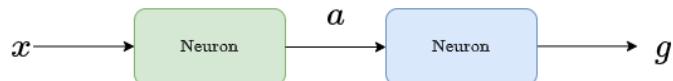
This means that **no neuron** has an output that affects its **input**, directly or indirectly.

We call these **feed-forward** networks.

We'll show how to build up the rest of what we need.

6.2.3 How to build networks

Suppose we have two neurons in **series**, our **simplest** arrangement:

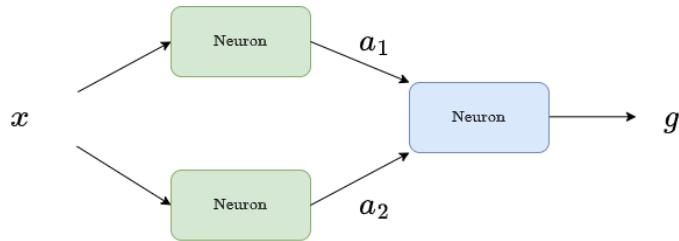


Our first neuron takes in a whole **vector** of values, $x = [x_1, x_2, \dots, x_m]^T$. But, it only **outputs** a single value, a .

That means the second neuron only receives **one** value, but it's capable of handling a full **vector**. We can add more values!

Remember that while we only see one arrow from x , each data point x_i is included.

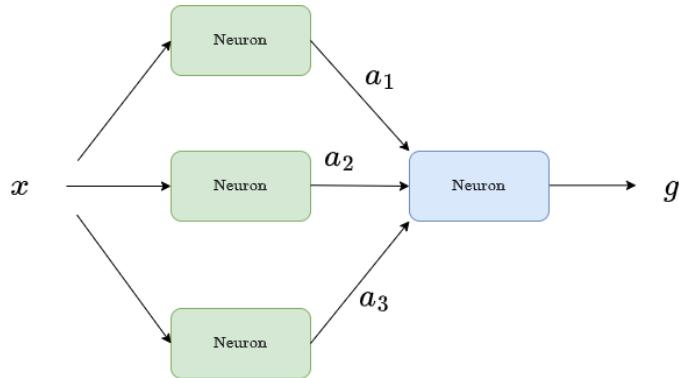
Let's add **another** neuron.



Our rightmost neuron now has **2 inputs**, which can be stored in a vector,

$$A = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \quad (6.12)$$

We could increase the **length** of this vector by adding more **neurons**.



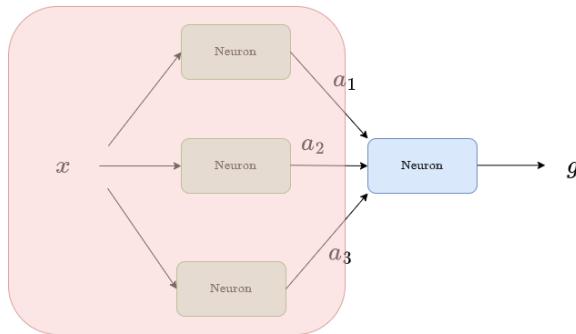
$$A = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad (6.13)$$

For our **rightmost** neuron, this is effectively the **same** as x : an **input vector**.

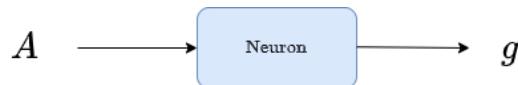
6.2.4 Layers

This gives us an idea for how to **build** our network: using multiple neurons in **parallel**, we can output a new vector A !

This is useful, because it means we can **simplify**: from the rightmost neuron's perspective, it just sees that **vector** as an input.



We can take this entire layer...



And just reduce it down to the vector A .

Because it's so useful, we'll give this set of neurons a name: a **layer**.

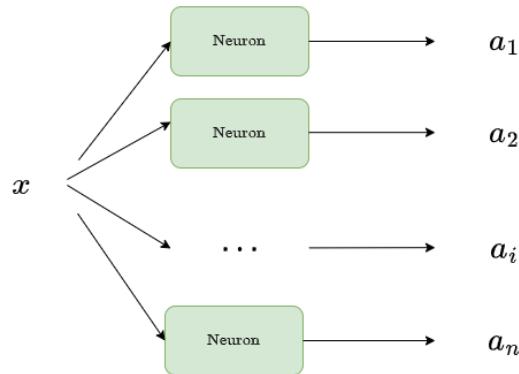
Definition 224

A **layer** is a set of **neurons** that are in "parallel":

- They all have **inputs** from the same **previous layer**
 - This **previous layer** could also be the **original input** x .
- They all have **outputs** to the same **next layer**
 - This **next layer** could also be the **final output** of the neural network.
- And none of these neurons are directly **connected** to each other.

This **layering** structure allows us to simplify our **analysis**: anything that comes after the layer only has to work with a **single vector**.

A layer in general might look like this:



A general layer in a neural network.

6.2.5 The Basic Structure of a Neural Network

We could pick many structures for neural networks, but for simplicity, this will define our **template** for this chapter.

Definition 225

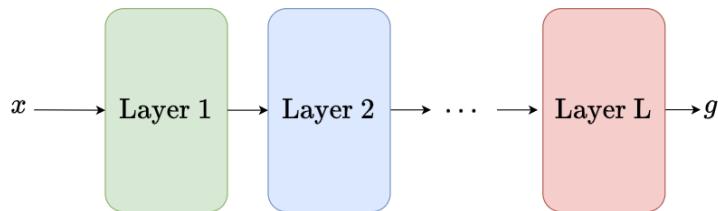
We structure our **neural networks** as a series of **layers**, where each layer is the **input** to the next layer.

This means that **layers** are a basic unit of a neural network, one level above a **neuron**.

In short, we have:

- A **neuron**, made of a linear and an activation component
- A **layer**, made of many neurons in parallel
- A **neural** network, made of many layers in series

Our goal is some kind of structure that looks something like this:

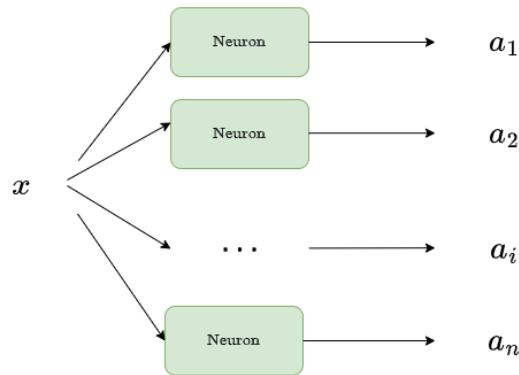


A neural network.

We now have a high-level view of our entire neural network, so now we dig into the details of a single layer.

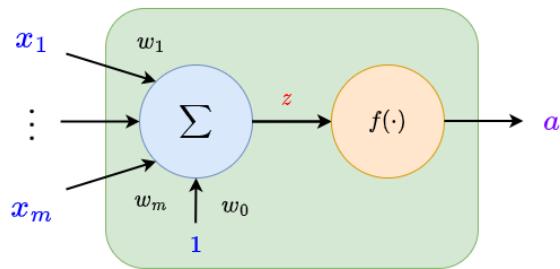
6.2.6 Single Layer: Visualizing our Components

Now, rather than analyzing a single neuron, we will analyze a single layer.



Our first layer.

In order to **analyze** this layer, we have to open back up the **abstraction**:

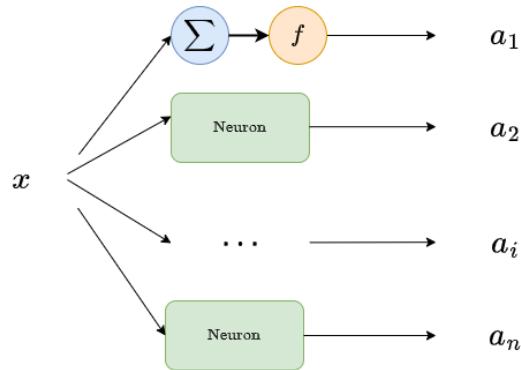


Each of those neurons looks like this.

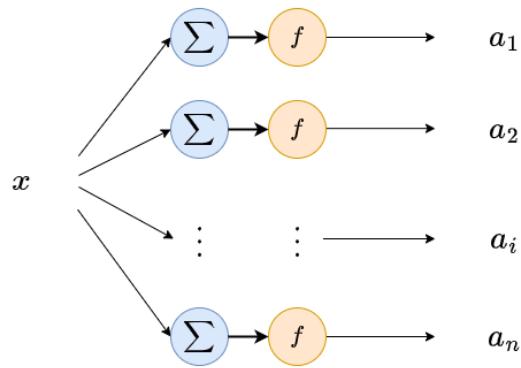
There are two important pieces of **information** we're hiding:

- We have two components inside of our neuron.
- We have many inputs x_i for one neuron.

The first piece of information is easier to visualize: we just replace each neuron with the two components.



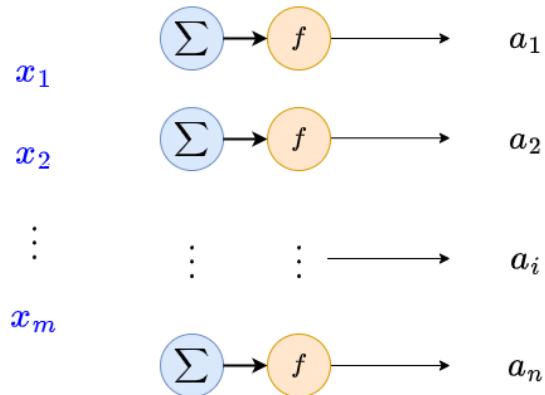
Replacing one neuron...



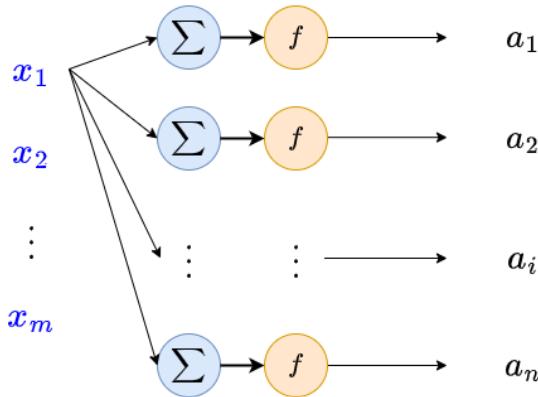
Replacing all neurons!

6.2.7 Single Layer: Visualizing our Inputs

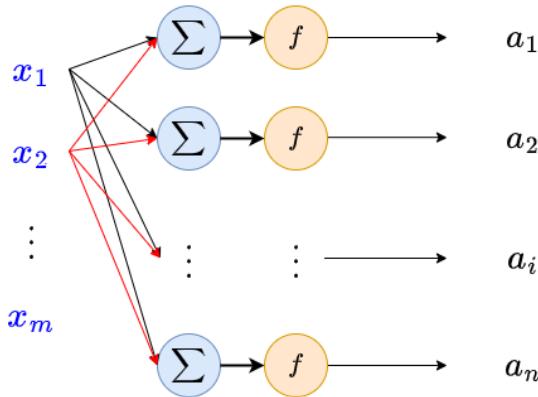
The second piece of information is much more difficult: we show all of the x_i outputs.



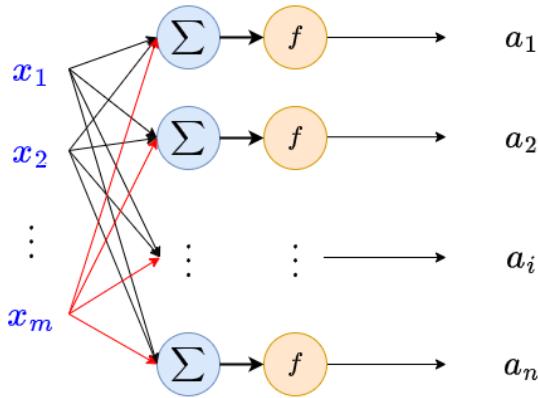
Now we have to draw the arrow for each input.



Every neuron receives the first input.



Every neuron receives the second input, too. This is getting messy...



The completed version: this is hard to look at.

Don't worry if this looks **confusing!** It's natural for it to be **hard** to read: the only thing you need to know is that we pair **every** input with **every** neuron.

This is our **final** view of this layer: because each of our m inputs has to go to every of n neurons, we end up with mn different **weights**.

This is a ton of **information**, and its only one layer! This shows how **complex** a neural network can be, just by **combining** simple neurons.

Note that this is a **fully connected** network: not all networks are FC.

Definition 226

A layer is **fully connected** if every neuron has the **same input vector**.

Example: If one of our neurons **ignored** x_1 , but the others did **not**, the layer would not be **fully connected**.

6.2.8 Dimensions of a layer

Now that we've seen the **full** view, we can **analyze** it. Our goal is to create a more **useful** and **accurate** simplification.

Our first point: note that the input and output have a **different** dimensions!

Clarification 227

A **layer** can have a different **input** and **output** dimension. In fact, they are completely **separate** variables.

This is because **every** input variable is allowed to be applied to the **same** neuron:

Example: You can have one neuron of the form

$$z = \begin{bmatrix} w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + w_0$$

In this case, our neuron has **one** output variable $f(z)$, but **three** inputs x_1, x_2, x_3 .

Thus, our output dimension has been separated from our input dimension. Instead, it is the number of neurons.

So, in general, we can say:

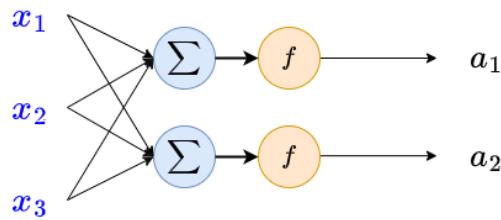
Notation 228

A **layer** has two associated **dimensions**: the **input** dimension m and the **output** dimension n .

- The **input** dimension m is based on the vector output from the **previous layer**:
 $x \in \mathbb{R}^m$
- The **output** dimension n is equal to the **number of neurons** in the **current** layer:
 $A \in \mathbb{R}^n$

Example: Suppose you have an **input** vector $x = [x_1, x_2, x_3]$ and two **neurons**. The dimensions are $m = 3$, and $n = 2$.

$$m = 3 \qquad \qquad n = 2$$



The input dimension and output dimensions are **separate**.

6.2.9 The known objects of our layer

So, we know we have two objects so far:

- Our **input** vector $x \in \mathbb{R}^m$
- Our **output** vector $A \in \mathbb{R}^n$

Where they each take the form

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \qquad A = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \quad (6.14)$$

But, there are a couple other things we haven't **generalized** for our entire **layer**:

- Our weights
- Our offsets
- Our preactivation

6.2.10 The other variables of our layer: weights and offsets

First, our **weights**: each neuron has its own vector of weights $w \in \mathbb{R}^m$.

The dimension needs to match x so we can compute $w^T x$.

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} \quad (6.15)$$

To distinguish them from each other, we'll represent the i^{th} neuron's weights as \vec{w}_i .

$$\vec{w}_i = \begin{bmatrix} w_{1i} \\ w_{2i} \\ \vdots \\ w_{mi} \end{bmatrix} \quad (6.16)$$

Each needs to be used to **compute** a_i , but having so many objects is annoying.

Remember that, when we had **multiple** data points $x^{(i)}$, we worked with them at the **same time** by stacking them in a **matrix**. Let's do the same here:

$$W = \overbrace{\begin{bmatrix} \vec{w}_1 & \vec{w}_2 & \cdots & \vec{w}_n \end{bmatrix}}^{\text{Each neuron has a weight vector}} \quad (6.17)$$

If we expand it out, we get a full matrix...

$$W = \left\{ \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m1} & \cdots & w_{mn} \end{bmatrix} \right\}^{\text{n neurons}}_{\text{m inputs}} \quad (6.18)$$

This is our **weight matrix** W : it's an $(m \times n)$ matrix. It contains all of our mn weights, sorted by

- **Input variable** (row)
- **Neuron** (column)

We can do this for our **offsets** too: thankfully, there is only **one** offset per neuron, so we can write:

$$W_0 = \begin{bmatrix} w_{01} \\ w_{02} \\ \vdots \\ w_{0n} \end{bmatrix} \left. \right\} \text{Each neuron has an offset} \quad (6.19)$$

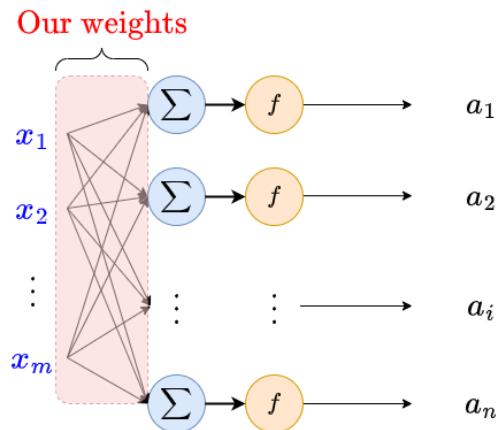
This is our offset vector, with the shape $(n \times 1)$.

Notation 229

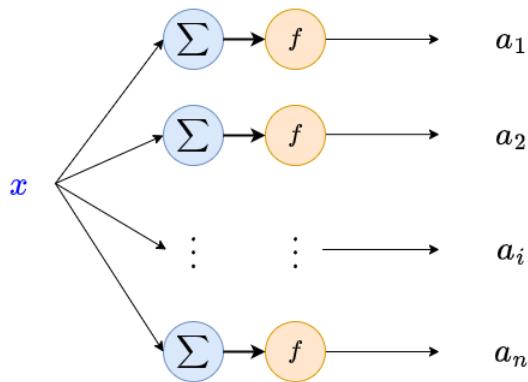
We can store our **weights** and **offsets** as **matrices**:

- **Weight** matrix W has the shape $(m \times n)$
- **Offset** matrix W_0 has the shape $(n \times 1)$

These matrices give us a tidy way to understand all of this mess:



Now that we understand it, we'll **hide** those weights again, for readability.



6.2.11 Pre-activation

Now, all that remains is the pre-activation z .

Before, we did

$$w^T x + w_0 = z \quad (6.20)$$

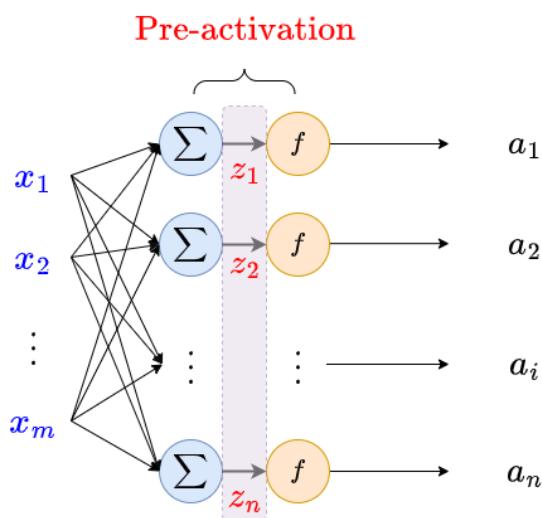
Because we so carefully kept our weights and offsets separate, we can still do this!

$$W^T x + W_0 = Z \quad (6.21)$$

This pre-activation vector Z contains all of the outputs of our linear components:

$$Z = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} \quad (6.22)$$

On our diagram, we can see it here:



This section is what Z details with.

And we can connect this to our activation: each a_i is the result of running our function f on z_i :

Because we run the function on each element in Z , we call this an **element-wise** use of our function.

$$A = f(Z) = \begin{bmatrix} f(z_1) \\ f(z_2) \\ \vdots \\ f(z_n) \end{bmatrix} \quad (6.23)$$

6.2.12 Summary of layer

So, we can now break our our layer up into pieces:

Notation 230

Our **layer** is a **function** that takes in $x \in \mathbb{R}^m$, and returns $A \in \mathbb{R}^n$.

It is defined by:

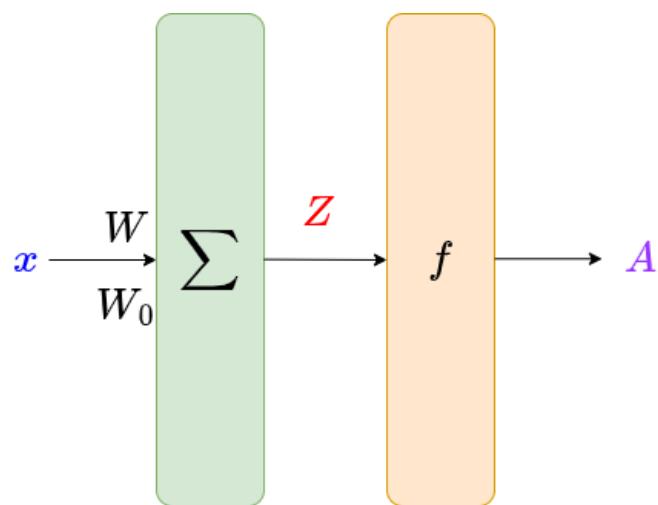
- **Dimensions:** m for **input**, n for **output** (number of neurons)

And our different **matrices**:

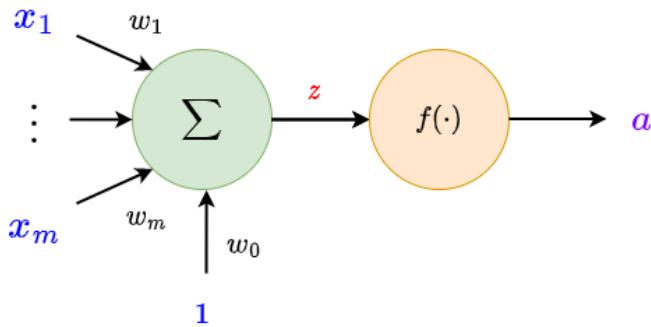
- **Input:** a **column vector** X in the shape $(m \times 1)$
- **Weights:** a **matrix** W in the shape $(m \times n)$
- **Offset:** a **column vector** W_0 in the shape $(n \times 1)$
- **Pre-activation:** a **column vector** Z in the shape $(n \times 1)$
- **Activation:** a **column vector** A in the shape $(n \times 1)$

We've now accomplished our goal: **simplify** the layer into its **base** components, without losing any crucial **information**.

We've can represent an entire layer like this:



Note how similar this looks to a **single** neuron: this works because the neurons in a **layer** are in **parallel**!



The math is very similar as well:

Definition 231

Our **layer** can be represented by

- A **linear** component that takes in x , and outputs **pre-activation** Z :

$$Z = W^T x + W_0$$

- A (potentially nonlinear) **activation** component that takes in Z , and outputs **activation** A :

$$A = f(Z)$$

When we **compose** them together, we get

$$A = f(Z) = f(W^T x + W_0)$$

6.2.13 The weakness of a single layer

What can we do with a single layer? Well, our LLC model gives us an example: it has the **nonlinear** sigmoid activation, but acts as a **linear** separator.

Why is that? Why is the separator still linear, if the **activation** isn't?

Well, let's take the **linear** separator created by the pre-activation:

$$z = W^T x + W_0 = 0 \tag{6.24}$$

This is our **boundary** for just a linear function. But adding the nonlinear activation should

make it more **complex**, right?

Well, it turns out, we can represent our **activation** boundary with a **linear** boundary.

Example: Continue our LLC example. If $z = 0$, then $\sigma(z) = \sigma(0)$. Our boundary is

$$\sigma(z) = \sigma(0) = \frac{1}{2} \quad (6.25)$$

Wait. But that means that $\sigma(z) = .5$ is the same as $z = 0$: the same inputs x cause both of them, so they have the same boundary!

$$\text{Linear boundary } z = 0 \iff f(z) = \frac{1}{2} \quad (6.26)$$

Summary:

- $\sigma(z) = .5$ is the **same** as $z = 0$.
- $z = 0$ is **linear**.
- Thus, our sigmoid boundary is **linear**.

We can apply this to other activation functions. In general, any constant boundary for most $f(z)$ is equivalent to some linear boundary $z = C$:

Assuming that f is invertible, which it often is.

$$z = C \iff f(z) = f(C) \quad (6.27)$$

Since $z = C$ is linear, we know that our activation separator $f(x) = f(C)$ is linear too.

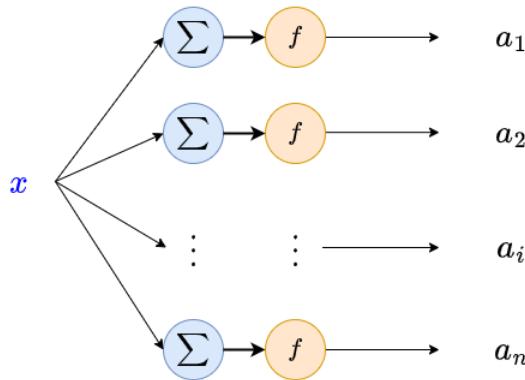
Concept 232

A single neuron creates a **linear separator**, even if it has a **nonlinear** activation.

This is because any **boundary** for $f(z)$ we can create, can be represented by some **linear** boundary in z .

It turns out, adding more neurons **within** the layer doesn't change much: because they act in **parallel**, each neuron acts separately, and the things we said above are still **true** for each output a_i .

There are exceptions, but this is true for most useful activation functions.



Each of these neurons has the same input, x .

So, in order to create nonlinear behavior, we need at least two layers of neurons in **series**.

So, we'll start **stacking** layers on each other: each layer **feeds** into the next one.

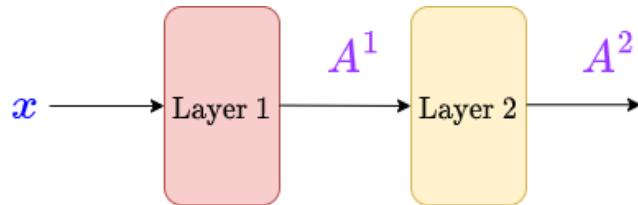
Concept 233

A **single layer** of neurons has **linear** behavior.

We need **multiple** layers to get a nonlinear **neural network**.

6.2.14 Adding a second layer

So, let's add one more **layer**. We'll label layers by using a **superscript**: W^1 is the set of **weights** for the **first** layer, for example.



We have two separate outputs: A^1 and A^2 .

Clarification 234

Superscripts in our notation indicate the **layer** that our value is associated with.

They do **not** represent exponentiation!

Example: Z^3 would be the **pre-activation** for layer 3: it is **not** Z "cubed".

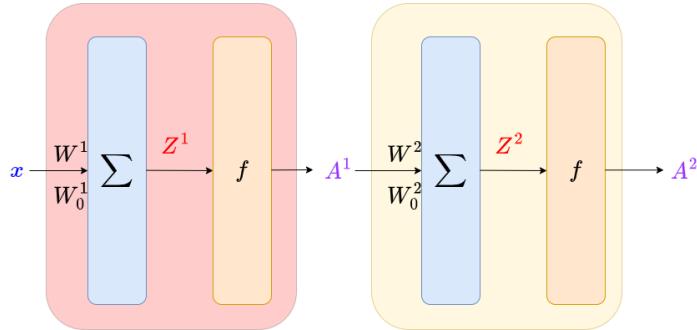
What can we learn from this?

- The **output** of layer 1, A^1 , is the **input** to layer 2.

- Thus, the output dimension n^1 of layer 1 must **match** the input m^2 of layer 2:

$$n^1 = m^2 \quad (6.28)$$

Let's break these into their components again.



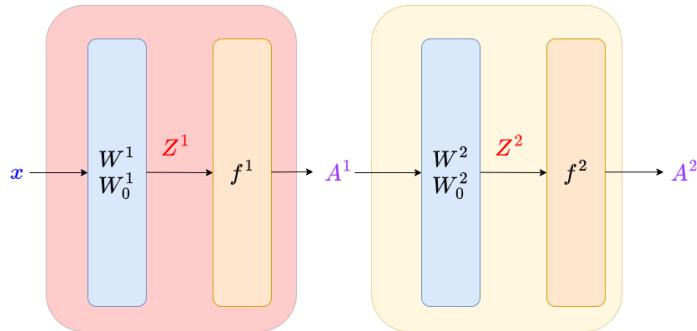
We have two separate outputs: A^1 and A^2 .

To distinguish between the linear functions in each layer, we'll just notate them using the weights and offsets.

$$\begin{matrix} W^1 \\ W_0^1 \end{matrix} \leftrightarrow \sum$$

These two are equivalent (if in the same layer)! We'll use the notation on the left, so that you know which layer our unit is in.

And this gives us:



Now, we can make our functions. For layer one:

$$A^1 = f(Z^1) = f((W^1)^T x + W_0^1) \quad (6.29)$$

And layer two:

$$\mathbf{A}^2 = f(Z^2) = f((\mathbf{W}^2)^T \mathbf{A}^1 + \mathbf{W}_0^2) \quad (6.30)$$

We can use this to build our **general** pattern.

6.2.15 Many Layers

We are finally ready to build our **complete** neural network. We'll just retrace the steps of the 2-layer case.

Notation 235

The total **number** of **layers** in our **neural network** is notated as L .

Typically we notate an **arbitrary** layer as ℓ (or l).

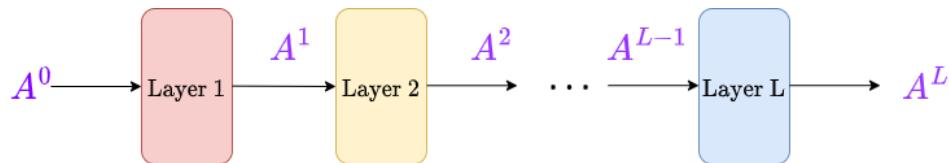
Since x is, for all purposes, **equivalent** to a vector A , we will call it A^0 .

Notation 236

Our **neural network**'s input x is used in the **same** way as every term A^ℓ .

So, we will **represent** it as

$$x = A^0$$



Again, we see that the **output** of layer ℓ is the **input** of layer $\ell + 1$.

Concept 237

Each layer **feeds** into the next layer.

A^ℓ is the **output** of layer ℓ , and the **input** of layer $\ell + 1$.

This means that the **output** dimension must **match** the next **input** dimension.

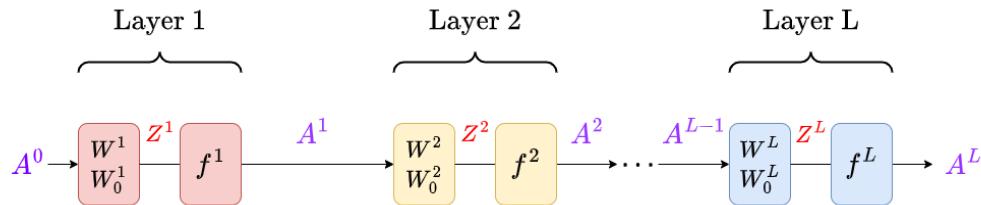
$$\underbrace{n^\ell}_{\text{Output}} = \underbrace{m^{\ell+1}}_{\text{Output}}$$

And the **dimension** of A^ℓ is $(n^\ell \times 1) = (m^{\ell+1} \times 1)$.

6.2.16 Our Complete Neural Network

We can break our layers into components, so we can see the functions involved.

With this, we build our final neural network:



With this, we can see how each layer is **related** to each other: as we **mentioned**, the **output** of one layer is the **input** of the next layer.

Here is the computation we do for layer ℓ :

Key Equation 238

The calculations done by layer ℓ are given by

$$Z^\ell = (W^\ell)^T A^{\ell-1} + W_0^\ell$$

and

$$A^\ell = f(Z^\ell)$$

Which combine into:

$$A^\ell = f(Z^\ell) = f\left((W^\ell)^T A^{\ell-1} + W_0^\ell\right)$$

One more comment: a useful definition.

Definition 239

A **hidden layer** is any layer except for the **last** one.

It is called a "**hidden**" layer because, if you're viewing the whole neural network based on

- **Input** x (first input)
- **Output** A^L (final output)

Then you can't see the **output** of any of the layers except for the **last** one.

Sometimes you'll hear someone say that a hidden layer is any except the "**first or last**": by that, they mean you can view the **input** for the **first** layer, as well as the **output** for the **last** layer.

But, when we're talking about **activation** functions (which we often are when we mention hidden layers, see below), we only care about whether the **output** is hidden!

6.3 Choices of activation function

Our linear model is entirely **defined** by its input: the number of **weights** in a neuron is just the number of **inputs** m .

But our **activation** function is up to us to decide: what works best?

6.3.1 Trying out linear activation

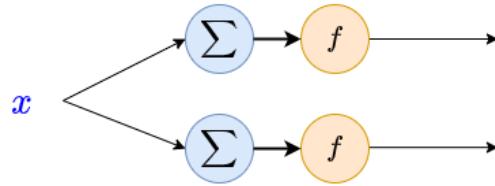
The simplest assumption would be to just use the **identity** function

$$f(z) = z \quad (6.31)$$

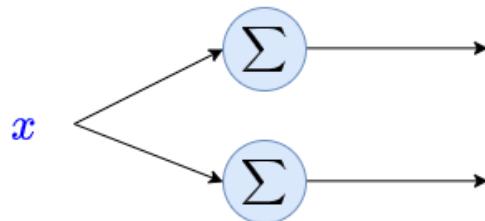
We might hope that we can combine a bunch of simple, **linear** models, and get a more sophisticated model. Why bother having a **nonlinear** activation at all?

Well, it turns out, combining **multiple** linear layers doesn't make our model any stronger. Let's try an example: we'll take a network with 2 layers, two neurons each.

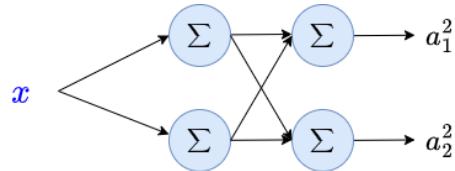
Let's look at layer 1:



Since the activation function has **no effect** on our result, we can **omit** it:

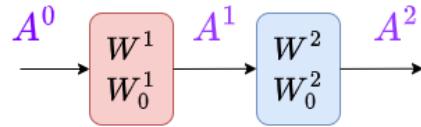


And now, we can show our **full** network:



6.3.2 Linear Layers: An example

We'll assume **two** inputs $A_0 = [x_1, x_2]^T$. For our sanity, we'll lump all of the weights in each layer:



We'll leave out W_0 terms to make it more readable, but the same will apply.

Layer 1:

$$A^1 = (\textcolor{red}{W}^1)^T \textcolor{blue}{A}_0 \quad (6.32)$$

Layer 2:

$$A^2 = \overbrace{(\textcolor{blue}{W}^2)^T (\textcolor{red}{W}^1)^T}^{\text{Weight matrices}} \textcolor{blue}{A}_0 \quad (6.33)$$

The full function for this equation is two matrices, **multiplied** by our input vector.

Let's take an arbitrary example:

$$\textcolor{red}{W}^1 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad \textcolor{blue}{W}^2 = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \quad (6.34)$$

Our equation becomes:

$$A^2 = \overbrace{\begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}}^{\text{Transposed matrices}} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad (6.35)$$

We created this function by applying two matrices separately. But, can't we **combine** them?

$$A^2 = \begin{bmatrix} 19 & 43 \\ 22 & 50 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad (6.36)$$

Wait, but this looks like a **one-layer** network with those weights! The second layer is **pointless**, we could have represented it with a single layer...

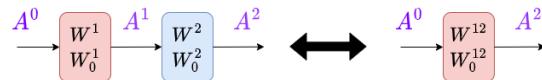
$$(\textcolor{blue}{W}^{12})^T = \begin{bmatrix} 19 & 43 \\ 22 & 50 \end{bmatrix} \quad (6.37)$$

6.3.3 The problem with linear networks

In fact, this is true in general: we can always take our **two** linear layers and combine them into **one**.

$$(\mathbf{W}^2)^T (\mathbf{W}^1)^T = \mathbf{W}^{12} \quad (6.38)$$

Our network is **equivalent** to the supposedly "simpler" one-layer network.



What if we have more layers? Well, we can just combine them one-by-one. At the end, we're just left with one layer:

$$(\mathbf{W}^L)^T (\mathbf{W}^{L-1})^T \dots (\mathbf{W}^2)^T (\mathbf{W}^1)^T = \mathbf{W}^X \quad (6.39)$$

And so, we can't just use linear layers: we **need** a **nonlinear** activation function.

Concept 240

Having multiple consecutive **linear layers** (i.e. layers with linear **activation** functions) is **equivalent** to having one linear layer in its place.

This means that we do not expand our **hypothesis** class by using more linear layers: we have to use **nonlinear** activation functions.

If we use something **nonlinear**...

$$\mathbf{A}^2 = \mathbf{f} \left((\mathbf{W}^2)^T \mathbf{A}^1 \right) \quad (6.40)$$

We get something that doesn't **simplify**:

This is ugly, but we don't have to worry about the details.

$$\mathbf{A}^2 = \mathbf{f} \left((\mathbf{W}^2)^T \boxed{\mathbf{f} \left((\mathbf{W}^1)^T \mathbf{x} \right)} \right) \quad (6.41)$$

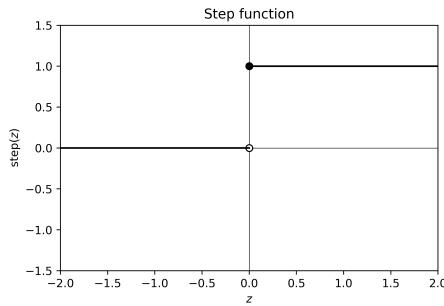
Which is what we want!

6.3.4 Example of Activation Functions

So, let's look at some possible **activation** functions:

- **Step** function $\text{step}(z)$:

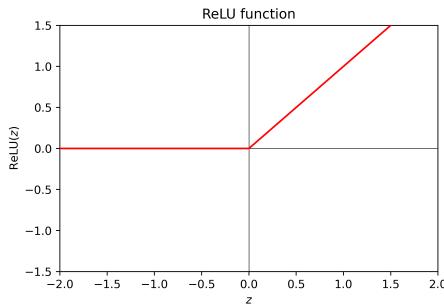
$$\text{step}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \quad (6.42)$$



- This function is basically a **sign** function, but uses $\{0, 1\}$ instead of $\{-1, +1\}$.
- Step functions were a common early choice, but because they have a **zero gradient**, we can't use **gradient descent**, and so we basically **never** use them.
- **Rectified Linear Unit** $\text{ReLU}(z)$:

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \quad (6.43)$$

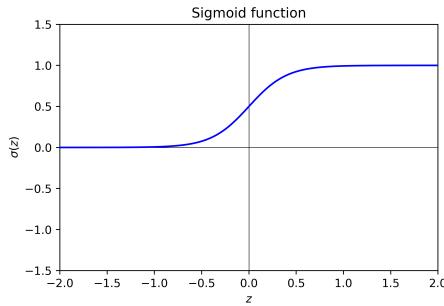
Same reason we replaced the sign function with sigmoid.



- This is a very **common** choice for activation function, even though the derivative is undefined at 0.
- We specifically use it for internal ("hidden") layers: layers that are neither the **first** nor **last** layer.
- **Sigmoid** function $\sigma(z)$:

They're "hidden" because they aren't visible to the input or output.

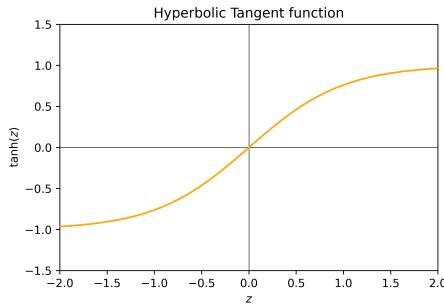
$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (6.44)$$



- This is the **activation** function for our **LLC** neuron from before.
- Just like it was then, it's useful for the **output neuron** in **binary classification**.
- Can be interpreted as the **probability** of a positive (+1) binary classification.

- **Hyperbolic Tangent** $\tanh(z)$:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad (6.45)$$



- This function looks similar to sigmoid over a different **range**.
- Unfortunately, it will not get much use in this class.

- **Softmax** function $\text{softmax}(z)$:

$$\text{softmax}(z) = \begin{bmatrix} \exp(z_1) / \sum_i \exp(z_i) \\ \vdots \\ \exp(z_n) / \sum_i \exp(z_i) \end{bmatrix} \quad (6.46)$$

- Behaves a like a **multi-class** version of **sigmoid**.
- Appropriately, we use it as the **output neuron** for **multi-class** classification.
- Can be interpreted as the **probability** of our k possible classifications.

Concept 241

For the different **activation functions**:

- $\text{sign}(z)$ is **rarely** used.
- $\text{ReLU}(z)$ is often used for "**hidden**" layers.
- $\sigma(z)$ is often used as the **output** for **binary classification**.
- $\text{softmax}(z)$ is often used as the **output** for **multi-class classification**.

$\tanh(z)$ is useful, but not a focus of this class.

6.4 Loss functions and activation functions

As we can see above, your **activation** function depends on what kind of **problem** you're dealing with.

The same is true for our **loss** function: we used **different** loss functions for classification and regression.

Classification can be further broken up into **binary** versus **multiclass** classification.

To summarize our findings, we'll **sort** this information:

Concept 242

Each of our **tasks** requires a different **loss** and output **activation** function.

We emphasize that we specifically mean the **output** activation function: the activation function used in **hidden layers** doesn't have to match the loss function.

task	f^L	Loss	
Regression	Linear z	Squared	$(g - y)^2$
Binary Class	Sigmoid $\sigma(z)$	NLL	$y \log g + (1 - y) \log(1 - g)$
Multi-Class	Softmax $\text{softmax}(z)$	NLLM	$\sum_j y_j \log(g_j)$

Terms

- Neuron (Unit, Node)
- Neural Network
- Series and Parallel
- Linear Component
- Weight w
- Offset (Bias, Threshold) w_0
- Activation Function f
- Pre-activation z
- Activation a
- Identity Function
- Acyclic Networks
- Feed-forward Networks
- Layer
- Fully Connected
- Input dimension m
- Output dimension n
- Weight Matrix
- Offset Matrix
- Layer Notation A^ℓ
- Step function
- ReLU function
- Sigmoid function
- Hyperbolic tangent function
- Softmax function

CHAPTER 6

Neural Networks 1.5 - Back-Propagation and Training

6.5 Error back-propagation

We have a complete neural network: a **model** we can use to make predictions or calculations.

Now, our mission is to **improve** this neural network: even if our hypothesis class is good, we still have to **find** the hypotheses that are useful for our problem.

As usual, we will start out with **randomized** values for our weights and biases: this **initial** neural network will not be useful for anything in particular, but that's why we need to improve it.

For such a complex problem, we definitely can't find an explicit solution, like we did for ridge regression. Instead, we will have to rely on **gradient descent**.

Concept 243

Neural networks are typically optimized using **gradient descent**.

We randomize them because otherwise, if our initialization is $w_i = 0$, we get

$$w^T x + w_0 = 0$$

no matter what input x we have.

6.5.1 Review: Gradient Descent

What does it really mean to do gradient descent on our **network**? Let's remind ourselves of how gradient descent works, and then **build** up to a network.

Concept 244

Gradient descent works based on the following reasoning:

- We have a function we want to **minimize**: our loss function \mathcal{L} , which tells us how **badly** we're doing.
 - We want to perform "less badly".
- Our main tool for **improving** \mathcal{L} is to alter θ and θ_0 .
 - These are our **parameters**: we're adjusting our model.
- The **gradient** is our main tool: $\frac{\partial \mathcal{L}}{\partial \theta}$ tells you the direction to **change** A in order to **increase** B.
 - Remember that η is our **step size**: we can take bigger or smaller steps in each direction.
- We want to **change** θ to **decrease** \mathcal{L} . Thus, we move in the direction of

$$\Delta\theta = -\eta \frac{\partial \mathcal{L}}{\partial \theta} \quad (6.1)$$
- We take steps $\Delta\theta$ (and $\Delta\theta_0$) until we are satisfied with \mathcal{L} , or it **stops** improving.

6.5.2 Review: Gradient Descent with LLCs

Let's start with a familiar example: **LLCs**.

Our LLC model uses the following equations:

We'll use w instead of θ .

$$z(x) = w^T x + w_0 \quad g(z) = \sigma(z) = \frac{1}{1 + e^{-z}} \quad (6.2)$$

$$\mathcal{L}(g, y) = y \log(g) + (1 - y) \log(1 - g) \quad (6.3)$$

Our goal is to minimize \mathcal{L} by adjusting θ and θ_0 .

So, we want

$$\frac{\partial \mathcal{L}}{\partial w} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial w_0} \quad (6.4)$$

We did this by using the **chain rule**:

We'll focus on w , but the same goes for w_0 .

$$\frac{\partial \mathcal{L}}{\partial w} = \overbrace{\frac{\partial \mathcal{L}}{\partial g}}^{\mathcal{L}(g)} \cdot \overbrace{\frac{\partial g}{\partial w}}^{\text{6.5}}$$

We can break it up further using **repeated chain rules**:

$$\frac{\partial \mathcal{L}}{\partial w} = \overbrace{\frac{\partial \mathcal{L}}{\partial g}}^{\mathcal{L}(g)} \cdot \underbrace{\frac{\partial g}{\partial z} \cdot \frac{\partial z}{\partial w}}_{g(z)} \quad (6.6)$$

Plugging in our derivatives, we get:

$$\frac{\partial \mathcal{L}}{\partial w} = -\left(\frac{y}{\sigma} - \frac{1-y}{1-\sigma}\right) \cdot \underbrace{\sigma(1-\sigma)}_{\text{6.6}} \cdot \underbrace{\frac{\partial z}{\partial w}}_{x} \quad (6.7)$$

Concept 245

The **chain rule** allows us to take the gradient of **nested functions**, where each function is the **input** to the next one.

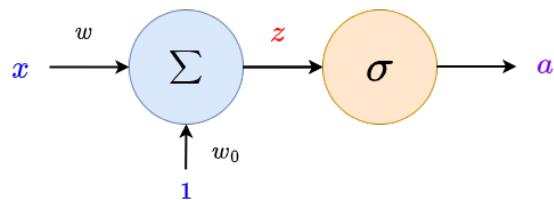
Another way to say this is that one function **feeds into** the next.

If you aren't familiar with "nested" functions, consider this example:

If you have functions $f(x)$ and $g(x)$, then $g(f(x))$ is the **nested** combination, where the output of f is the input of g .

6.5.3 Review: LLC as Neuron

Remember that we can represent our LLC as a **neuron**: this could give us the first idea for how to train our **neural network**!



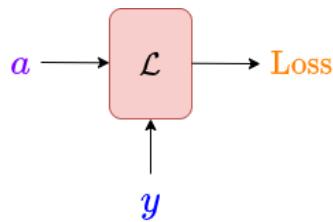
As usual, our first unit \sum is our **linear** component. The output is z , nothing different from before with LLC.

Remember that x is a whole vector of values, which we've condensed into one variable.

The **output** of σ , which we wrote before as g , is now a .

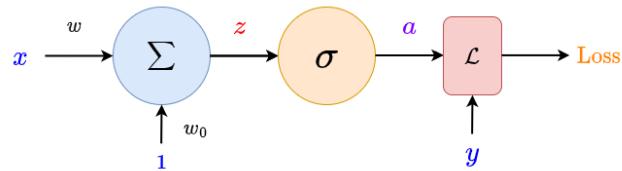
Something we neglected before: this diagram is **missing the loss function**. Let's create a small unit for that.

$\mathcal{L}(a, y)$ has **two** inputs: our predicted value a , and the correct value y .



We have two inputs to our loss function.

We **combine** these into a single unit to get:



Our full unit!

6.5.4 LLC Forward-Pass

Now, we can do gradient descent like before. We want to get the effect our **weight** has on our **loss**.

But, this time, we'll pair it with a **visual** that is helpful for understanding how we **train** neural networks.

First, one important consideration:

As we saw above, the **gradient** we get might rely on z , a , or $\mathcal{L}(a, y)$. So, before we do anything, we have to **compute** these values.

Each step **depends** on the last: this is what the **forward** arrows represent. We call this a **forward pass** on our neural network.

Definition 246

A **forward pass** of a neural network is the process of sending information "forward" through the neural network, starting from the **input**.

This means the **input** is fed into the **first** layer, and that output is fed into the **next** layer, and so on, until we reach our **final** result and **loss**.

Example: If we had

- $f(x) = x + 2$

- $g(f) = 3f$
- $h(g) = \sin(g)$

Then, a forward pass with the input $x = 10$ would have us go function-by-function:

- $f(10) = 10 + 2$
- $g(f) = 3 \cdot 12$
- $h(g) = \sin(36)$

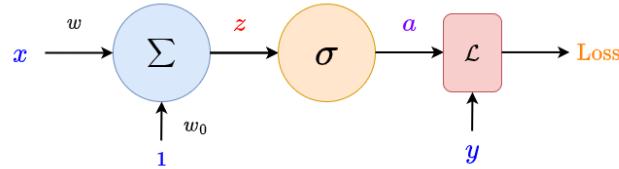
So, by "forward", we mean that we apply each function, one after another.

In our case, this means computing z , a , and $\mathcal{L}(a, y)$.



6.5.5 LLC Back-propagation

Now that we have all of our values, we can get our gradient. Let's **visualize** this process.

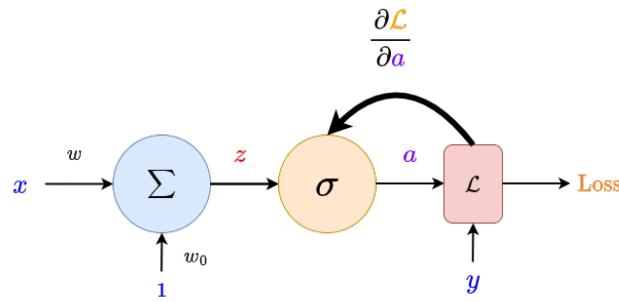


We want to link \mathcal{L} to w . In order to do that, we need to **connect** each thing in between.

This lets us **combine** lots of simple **links** to get our more complicated result.

Loss is what we really care about. So, what is the loss directly **connected** to? The **activation**, a .

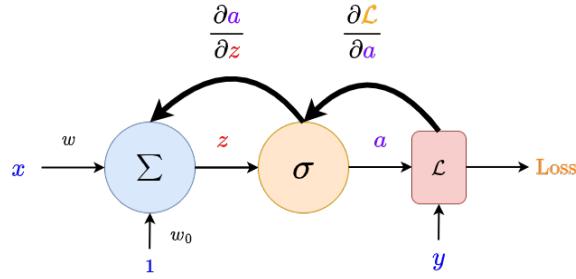
We can also call this "chaining together" lots of derivatives.



So, our σ unit has information about the derivative that comes after it: the **loss** derivative

$$\overbrace{\frac{\partial \mathcal{L}}{\partial a}}^{\text{Loss unit}} \quad (6.8)$$

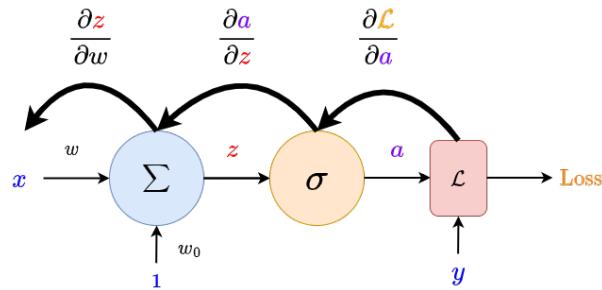
And what is that connected to? The **pre-activation** z :



Now, our Σ unit has information about both the **loss** derivative and the σ derivative:

$$\begin{array}{c} \text{Loss unit} \\ \overbrace{\frac{\partial \mathcal{L}}{\partial a}}^{\text{Activation function}} \end{array} \cdot \begin{array}{c} \text{Activation function} \\ \overbrace{\frac{\partial a}{\partial z}}^{\text{Linear subunit}} \end{array} \quad (6.9)$$

And finally, we've reached w :



And, we built our chain rule! This contains the **information** of the derivatives from **every** unit.

$$\frac{\partial \mathcal{L}}{\partial w} = \begin{array}{c} \text{Loss unit} \\ \overbrace{\frac{\partial \mathcal{L}}{\partial a}}^{\text{Activation}} \end{array} \cdot \begin{array}{c} \text{Activation} \\ \overbrace{\frac{\partial a}{\partial z}}^{\text{Linear subunit}} \end{array} \cdot \begin{array}{c} \text{Linear subunit} \\ \overbrace{\frac{\partial z}{\partial w}}^{\text{Linear subunit}} \end{array} \quad (6.10)$$

Moving backwards like this is called **back-propagation**.

Definition 247

Back-propagation is the process of moving "backwards" through your network, starting at the **loss** and moving back layer-by-layer, and gathering terms in your **chain rule**.

We call it "**propagation**" because we send backwards the **terms** of our chain rule about later derivatives.

An **earlier** unit (closer to the "left") has all of the **derivatives** that come after (to the "right" of) it, along with its own term.

6.5.6 Summary of neural network gradient descent: a high-level view

So, with just this, we have built up the basic idea of how we **train** our model: now that we have the gradient, we can do **gradient descent** like we normally do!

Concept 248

We can do **gradient descent** on a **neural network** using the ideas we've built up:

This summary covers some things we haven't fully discussed. We'll continue digging into the topic!

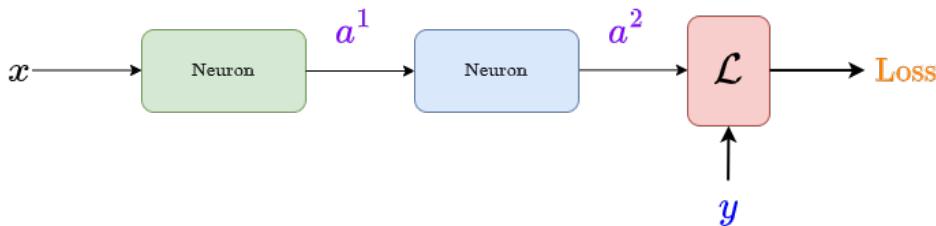
- Do a **forward pass**, where we compute the value of each **unit** in our model, passing the information **forward** - each layer's **output** is the next layer's **input**.
 - We finish by getting the **loss**.
- Do **back-propagation**: build up a **chain rule**, starting at the **loss** function, and get each unit's **derivative** in **reverse order**.
 - **Reverse** order: if you have 3 layers, you want to get the 3rd layer's **derivatives**, then the 2nd layer, then the 1st.
 - **Each weight** vector has its own **gradient**: we'll deal with this later, but we need to calculate one for each of them.
- Use your chain rule to get the **gradient** $\frac{\partial \mathcal{L}}{\partial w}$ for your **weight** vector(s). Take a **gradient descent** step.
- **Repeat** until satisfied, or your model **converges**.

6.5.7 A two-neuron network: starting backprop

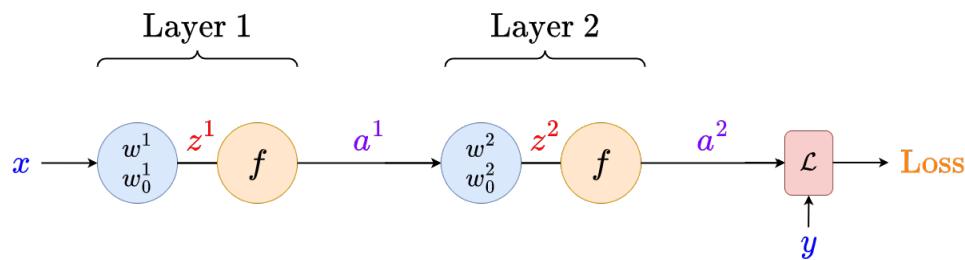
Above, we mention "each layer": we'll now transition to a **two-neuron** system, so we have "two layers". Then, we'll build up to many layers.

Remember, though, that the **ideas** represented here are just extensions of what we did **above**.

Let's get a look at our **two-neuron** system, now with our **loss** unit:



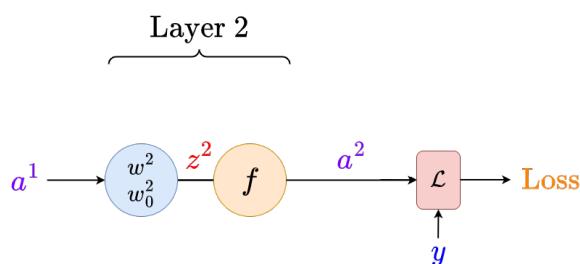
And unpack it:



We want to do **back-propagation** like we did before. This time, we have **two** different layers of weights: w^1 and w^2 . Does this cause any problems?

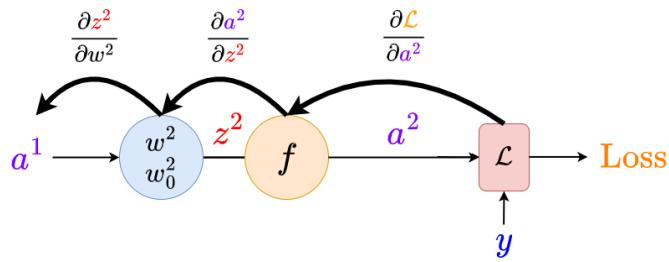
It turns out, it doesn't! We mentioned in the first part of chapter 7 that we can treat the **output** of the **first** layer a^1 as the same as if it were an **input** x .

This is one of the biggest benefits of neural network layers!



Now, we can do backprop safely.

"Backprop" is a common shortening of "back-propagation".



We can get:

$$\frac{\partial \mathcal{L}}{\partial w^2} = \underbrace{\frac{\partial \mathcal{L}}{\partial a^2}}_{\text{Loss unit}} \cdot \underbrace{\frac{\partial a^2}{\partial z^2}}_{\text{Activation}} \cdot \underbrace{\frac{\partial z^2}{\partial w^2}}_{\text{Linear}} \quad (6.11)$$

The same format as for our **one-neuron** system! We now have a gradient we can update for our **second** weight vector.

But what about our **first** weight vector?

6.5.8 Continuing backprop: One more problem

We need to continue further to reach our **earlier** weights: this is why we have to work **backward**.

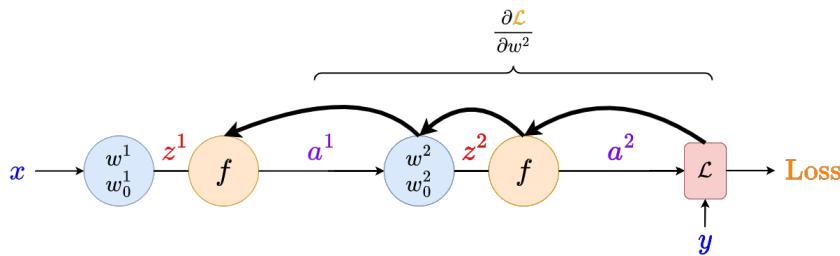
Concept 249

We work **backward** in **back-propagation** because every layer after the **current** one **affects** the gradient.

Our current layer **feeds** into the next layer, which feeds into the layer after that, and so on. So this layer affects **every** later layer, which then affect the loss.

So, to see the effect on the **output**, we have to **start** from the **loss**, and get every layer **between** it and our weight vector.

Remember that when we say "f feeds into g", we mean that the output of f is the input to g.



We have one problem, though:

We just gathered the derivative $\frac{\partial \mathcal{L}}{\partial w^2}$. If we wanted to continue the chain rule, we would expect to add more terms, like: _____

$$\frac{\partial w^2}{\partial a^1} \quad (6.12)$$

The problem is, what is w^2 ? It's a vector of constants.

$$w^2 = \begin{bmatrix} w_1^2 \\ w_2^2 \\ \vdots \\ w_n^2 \end{bmatrix}, \quad \text{Not a function of } a^1! \quad (6.13)$$

Since our current derivative includes w^2 , we would continue it with a w^2 in the "top" of a derivative,

$$\frac{\partial \mathcal{L}}{\partial w^2} \frac{\partial w^2}{\partial r}$$

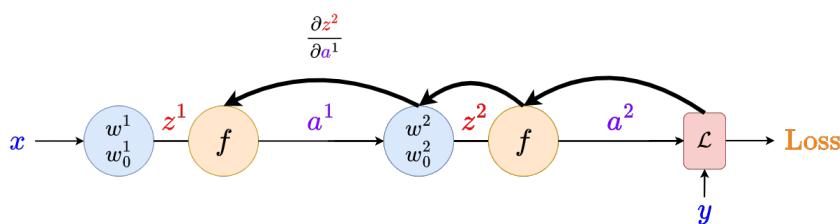
We're not sure what "r" is yet.

That derivative above is going to be **zero!** In other words, w^2 isn't really the **input** to z^2 : it's a **parameter**.

So, we can't end our derivative with w^2 . Instead, we have to use something else. z^2 's real input is a^1 , so let's go directly to that!

We were building our chain rule by combining inputs with outputs: that's what links two layers together.

So, it should make sense that using something like w (that doesn't link two layers) prevents us from making a longer chain rule.



Using this allows us to move from layer 2 to layer 1.

Now, we have our new chain rule:

$$\frac{\partial \mathcal{L}}{\partial a^1} = \overbrace{\frac{\partial \mathcal{L}}{\partial a^2} \cdot \frac{\partial a^2}{\partial z^2}}^{\text{Other terms}} \cdot \overbrace{\frac{\partial z^2}{\partial a^1}}^{\text{Link Layers}} \quad (6.14)$$

Concept 250

For our **weight gradient** in layer ℓ , we have to end our **chain rule** with

$$\frac{\partial \mathcal{L}}{\partial w^\ell}$$

So we can get

$$\frac{\partial \mathcal{L}}{\partial w^\ell} = \underbrace{\frac{\partial \mathcal{L}}{\partial z^\ell}}_{\text{Other terms}} \cdot \underbrace{\frac{\partial z^\ell}{\partial w^\ell}}_{\text{Get weight grad}}$$

However, because w^ℓ is not the **input** of layer ℓ , we can't use it to find the gradient of **earlier layers**.

Instead, we use

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} \quad (6.15)$$

To "link together" two different layers ℓ and $\ell - 1$ in a **chain rule**.

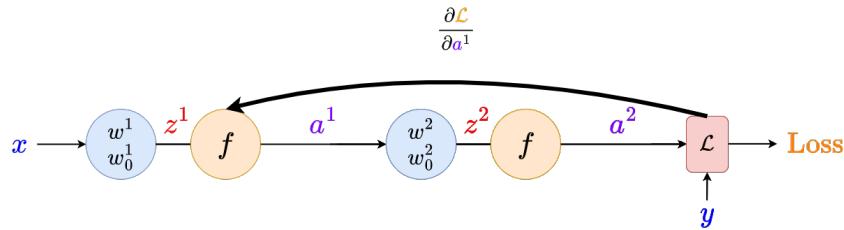
In this section, we compressed lots of derivatives into

$$\frac{\partial \mathcal{L}}{\partial z^\ell}$$

Don't let this alarm you, this just hides our long chain of derivatives!

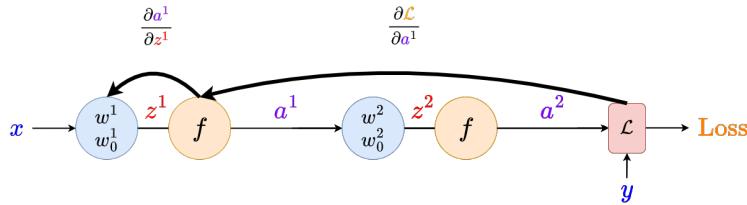
6.5.9 Finishing two-neuron backprop

Now that we have safely connected our layers, we can do the rest of our gradient. First, let's lump together everything we did before:

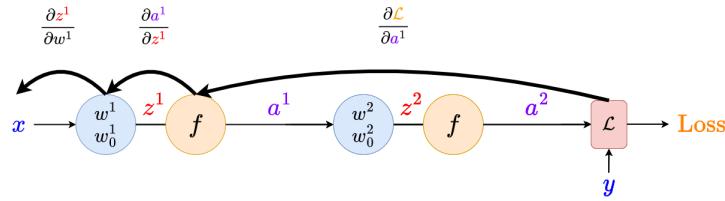


All the info we need is stored in this derivative: it can be written out using our friendly chain rule from earlier.

Now, we can add our remaining terms. It's the same as before: we want to look at the pre-activation



And finally, our input:



We can get our second chain rule

$$\frac{\partial \mathcal{L}}{\partial w^1} = \underbrace{\frac{\partial \mathcal{L}}{\partial a^1}}_{\text{Other layers}} \cdot \underbrace{\left(\frac{\partial a^1}{\partial z^1} \cdot \frac{\partial z^1}{\partial w^1} \right)}_{\text{Layer 1}} \quad (6.16)$$

Which, in reality, looks much bigger:

$$\frac{\partial \mathcal{L}}{\partial w^1} = \underbrace{\left(\frac{\partial \mathcal{L}}{\partial a^2} \right)}_{\text{Loss unit}} \cdot \underbrace{\left(\frac{\partial a^2}{\partial z^2} \cdot \frac{\partial z^2}{\partial a^1} \right)}_{\text{Layer 2}} \cdot \underbrace{\left(\frac{\partial a^1}{\partial z^1} \cdot \frac{\partial z^1}{\partial w^1} \right)}_{\text{Layer 1}} \quad (6.17)$$

We see a clear **pattern** here! In fact, this is the procedure we'll use for a neural network with **any** number of layers.

Concept 251

We can get all of our **weight gradients** by repeatedly appending to the **chain rule**.

For each layer, we multiply by

$$\text{Within layer} \quad \overbrace{\frac{\partial a^\ell}{\partial z^\ell}} \quad \cdot \quad \text{Get weight grad} \quad \overbrace{\frac{\partial z^\ell}{\partial w^\ell}}$$

To get the **weight gradient** $\partial \mathcal{L} / \partial w^\ell$.

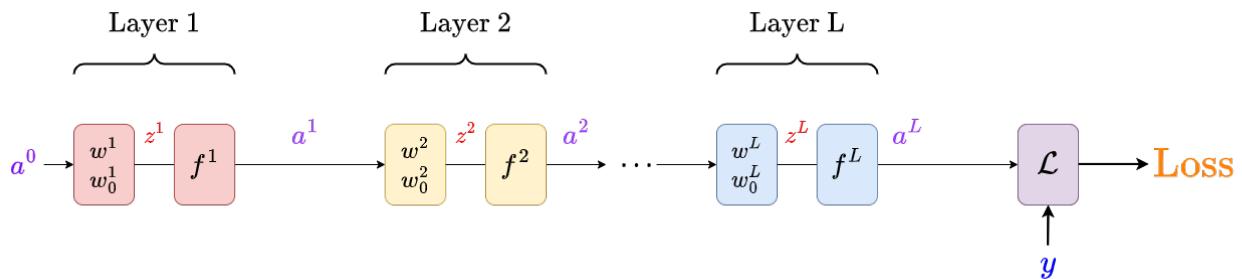
If we want to **extend** to the next layer, we **instead** multiply by

$$\text{Within layer} \quad \overbrace{\frac{\partial a^\ell}{\partial z^\ell}} \quad \cdot \quad \text{Link layers} \quad \overbrace{\frac{\partial z^\ell}{\partial a^{\ell-1}}}$$

6.5.10 Many layers: Doing back-propagation

Now, we'll consider the case of many possible layers.

To make it more readable, we'll use boxes instead of circles for units.



This may look intimidating, but we already have all the tools we need to handle this problem.

Our goal is to get a **gradient** for each of our **weight** vectors w^ℓ , so we can do gradient descent and **improve** our model.

According to our above analysis in Concept 9, we need only a few steps to get all of our gradients.

Concept 252

In order to do **back-propagation**, we have to build up our **chain rule** for each weight gradient.

- We start our chain rule with one term shared by every gradient:

$$\overbrace{\frac{\partial \mathcal{L}}{\partial a^L}}$$

Loss unit

Then, we follow these two steps until we run out of layers:

- We're at layer ℓ . We want to get the **weight gradient** for this layer. We get this by **multiplying** our chain rule by

$$\begin{array}{c} \text{Within layer} \quad \text{Get weight grad} \\ \overbrace{\frac{\partial a^\ell}{\partial z^\ell}} \quad \cdot \quad \overbrace{\frac{\partial z^\ell}{\partial w^\ell}} \end{array}$$

We **exclude** this term for any other gradients we want.

- If we aren't at layer 1, there's a previous layer we want to get the weight for. We reach layer $\ell - 1$ by multiplying our chain rule by

$$\begin{array}{c} \text{Within layer} \quad \text{Link layers} \\ \overbrace{\frac{\partial a^\ell}{\partial z^\ell}} \quad \cdot \quad \overbrace{\frac{\partial z^\ell}{\partial a^{\ell-1}}} \end{array}$$

Once we reach layer 1, we have **every single** weight vector we need! Repeat the process for w_0 gradients and then do **gradient descent**.

Let's get an idea of what this looks like in general:

$$\frac{\partial \mathcal{L}}{\partial w^\ell} = \overbrace{\left(\frac{\partial \mathcal{L}}{\partial a^L} \right)}^{\text{Loss unit}} \cdot \overbrace{\left(\frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \right)}^{\text{Layer L}} \cdot \overbrace{\left(\frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^{L-1}}{\partial a^{L-2}} \right)}^{\text{Layer L-1}} \cdot \left(\dots \right) \cdot \overbrace{\left(\frac{\partial a^\ell}{\partial z^\ell} \cdot \frac{\partial z^\ell}{\partial w^\ell} \right)}^{\text{Layer } \ell} \quad (6.18)$$

That's pretty ugly. If we need to hide the complexity, we can:

Notation 253

If you need to do so for **ease**, you can **compress** your **derivatives**. For example, if we want to only have the last weight term **separate**, we can do:

$$\frac{\partial \mathcal{L}}{\partial w^\ell} = \overbrace{\frac{\partial \mathcal{L}}{\partial z^\ell}}^{\text{Other}} \cdot \overbrace{\frac{\partial z^\ell}{\partial w^\ell}}^{\text{Weight term}} \quad (6.19)$$

But we should also explore what each of these terms *are*.

6.5.11 What do these derivatives equal?

Let's look at each of these derivatives and see if we can't simplify them a bit.

First, every gradient needs

- The **loss derivative**:

$$\frac{\partial \mathcal{L}}{\partial a^L} \quad (6.20)$$

This **depends** on our loss function, so we're **stuck** with that one.

Next, within each layer, we have

- The **activation function** - between our activation a and preactivation z :

$$\frac{\partial a^\ell}{\partial z^\ell} \quad (6.21)$$

What does the function between these **look** like?

$$a = f(z) \quad (6.22)$$

Well, that's not super interesting: we **don't know** our function. But, at least we can **write** it using f : that way, we know that this term only depends on our **activation** function.

$$\frac{\partial a^\ell}{\partial z^\ell} = \left(\overbrace{f^\ell}^{\text{func for layer } \ell} \right)'(z^\ell) \quad (6.23)$$

This expression is a bit visually clunky, but it works.

Between layers, we have

- We can also think about the derivative of the **linear function** that **connects two layers**:

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} \quad (6.24)$$

So, we want the function of these two:

$$z^\ell = w^\ell a^{\ell-1} + w_0^\ell \quad (6.25)$$

Be careful not to get this mixed up with the last one!
They look similar, but one is within the layer, and the other is between layers.

This one is pretty simple! We just take the derivative manually:

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} = w^\ell \quad (6.26)$$

Finally, every gradient will end with

- The derivative that directly connects to a **weight**, again using the **linear function**:

$$\frac{\partial z^\ell}{\partial w^\ell} \quad (6.27)$$

The linear function is the same:

$$z^\ell = w^\ell a^{\ell-1} + w_0^\ell \quad (6.28)$$

But with a different **variable**, the **derivative** comes out different:

$$\frac{\partial z^\ell}{\partial w^\ell} = a^{\ell-1} \quad (6.29)$$

Notation 254

Our **derivatives** for the **chain rule** in a **1-D neural network** take the form:

$$\frac{\partial \mathcal{L}}{\partial a^L} \quad (6.30)$$

$$\frac{\partial a^\ell}{\partial z^\ell} = (f^\ell)'(z^\ell) \quad (6.31)$$

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} = w^\ell \quad (6.32)$$

$$\frac{\partial z^\ell}{\partial w^\ell} = a^{\ell-1} \quad (6.33)$$

Now, we can rewrite our generalized expression for gradient:

$$\frac{\partial \mathcal{L}}{\partial w^\ell} = \overbrace{\left(\frac{\partial \mathcal{L}}{\partial a^L} \right)}^{\text{Loss unit}} \cdot \overbrace{\left((f^L)'(z^L) \cdot w^L \right)}^{\text{Layer L}} \cdot \overbrace{\left((f^{L-1})'(z^{L-1}) \cdot w^L \right)}^{\text{Layer L-1}} \cdot \left(\dots \right) \cdot \overbrace{\left((f^\ell)'(z^\ell) \cdot a^{\ell-1} \right)}^{\text{Layer } \ell} \quad (6.34)$$

Our expressions are more concrete now. It's still pretty visually messy, though.

6.5.12 Activation Derivatives

We weren't able to **simplify** our expressions above, partly because we didn't know which **loss** or **activation** function we were going to use.

So, here, we will look at the **common** choices for these functions, and **catalog** what their derivatives look like.

- **Step function** $\text{step}(z)$:

$$\frac{d}{dz} \text{step}(z) = 0 \quad (6.35)$$

This is part of why we don't use this function: it has no gradient. We can show this by looking piecewise:

$$\text{step}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \quad (6.36)$$

And take the derivative of each piece:

$$\frac{d}{dz} \text{ReLU}(z) = 0 = \begin{cases} 0 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \quad (6.37)$$

- **Rectified Linear Unit** $\text{ReLU}(z)$:

$$\frac{d}{dz} \text{ReLU}(z) = \text{step}(z) \quad (6.38)$$

This one might be a bit surprising at first, but it makes sense if you **also** break it up into cases:

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \quad (6.39)$$

And take the derivative of each piece:

$$\frac{d}{dz} \text{ReLU}(z) = \text{step}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \quad (6.40)$$

- **Sigmoid** function $\sigma(z)$:

$$\frac{d}{dz} \sigma(z) = \sigma(z)(1 - \sigma(z)) = \frac{e^{-z}}{(1 + e^{-z})^2} \quad (6.41)$$

This derivative is useful for simplifying NLL, and has a nice form.

As a reminder, the function looks like:

We can just compute the derivative with the single-variable chain rule.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (6.42)$$

- **Identity** ("linear") function $f(z) = z$:

$$\frac{d}{dz} z = 1 \quad (6.43)$$

This one follows from the definition of the derivative.

We cannot rely on a linear activation function for our **hidden** layers, because a linear neural network is no more **expressive** than one layer.

But, we use it for **regression**.

- **Softmax** function $\text{softmax}(z)$:

This function has a difficult derivative we won't go over here.

If you're curious, here's a [link](#).

- **Hyperbolic tangent** function $\tanh(z)$:

$$\frac{d}{dz} \tanh(z) = 1 - \tanh(z)^2 \quad (6.44)$$

This strange little expression is the "hyperbolic secant" squared. We won't bother further with it.

Notation 255

For our various **activation** functions, we have the **derivatives**:

Step:

$$\frac{d}{dz} \text{step}(z) = 0$$

ReLU:

$$\frac{d}{dz} \text{ReLU}(z) = \text{step}(z)$$

Sigmoid:

$$\frac{d}{dz} \sigma(z) = \sigma(z)(1 - \sigma(z))$$

Identity/Linear:

$$\frac{d}{dz} z = 1$$



6.5.13 Loss derivatives

Now, we look at the loss derivatives.

- **Square loss** function $\mathcal{L}_{\text{sq}} = (a - y)^2$:

$$\frac{d}{da} \mathcal{L}_{\text{sq}} = 2(a - y) \quad (6.45)$$

Follows from chain rule+power rule, used for regression.

- **Linear loss** function $\mathcal{L}_{\text{lin}} = |a - y|$:

$$\frac{d}{da} \mathcal{L}_{\text{lin}} = \text{sign}(a - y) \quad (6.46)$$

This one can also be handled piecewise, like $\text{step}(z)$ and $\text{ReLU}(z)$:

$$|u| = \begin{cases} u & \text{if } z \geq 0 \\ -u & \text{if } z < 0 \end{cases} \quad (6.47)$$

We take the piecewise derivative:

$$\frac{d}{du}|u| = \text{sign}(u) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases} \quad (6.48)$$

- **NLL** (Negative-Log Likelihood) function $\mathcal{L}_{\text{NLL}} = -(y \log(a) + (1-y) \log(1-a))$

$$\frac{d}{da}\mathcal{L}_{\text{NLL}} = -\left(\frac{y}{a} - \frac{1-y}{1-a}\right) \quad (6.49)$$

- **NLLM** (Negative-Log Likelihood Multiclass) function $\mathcal{L}_{\text{NLL}} = -\sum_j y_j \log(a_j)$

Similar to softmax, we will omit this derivative.

Notation 256

For our various **loss** functions, we have the **derivatives**:

Square:

$$\frac{d}{da}\mathcal{L}_{\text{sq}} = 2(a - y) \quad (6.50)$$

Linear (Absolute):

$$\frac{d}{da}\mathcal{L}_{\text{lin}} = \text{sign}(a - y) \quad (6.51)$$

NLL (Negative-Log Likelihood):

$$\frac{d}{da}\mathcal{L}_{\text{NLL}} = -\left(\frac{y}{a} - \frac{1-y}{1-a}\right) \quad (6.52)$$

6.5.14 Many neurons per layer

Now, we just have left the elephant in the room: what do we do about the case where we have *full* layers? That is, what if we have **multiple** neurons per layer? This makes this more complex.

Well, the solution is the same as in the first part of chapter 7: we introduce **matrices**.

But this time, with a twist: we have to do **matrix** calculus: a difficult topic indeed.

To handle this, we will go in somewhat **reversed** order, but one that better fits our needs.

- We begin by considering how the chain rule looks when we switch to matrix form.
- We give a general idea of what matrix derivatives look like.

- We list some of the results that matrix calculus gives us, for particular derivatives.
- We actually reason about how matrix calculus *works*.

The last of these is by far the **hardest**, and warrants its own section. Nevertheless, even without it, you can more or less get the idea of what we need - hence why we're going in reversed order.

6.5.15 The chain rule: Matrix form

Let's start with the first: the punchline, how does the chain rule and our gradient descent **change** when we add **matrices**?

It turns out, not much: by using **layers** in the last section, we were able to create a pretty powerful and mathematically **tidy** object.

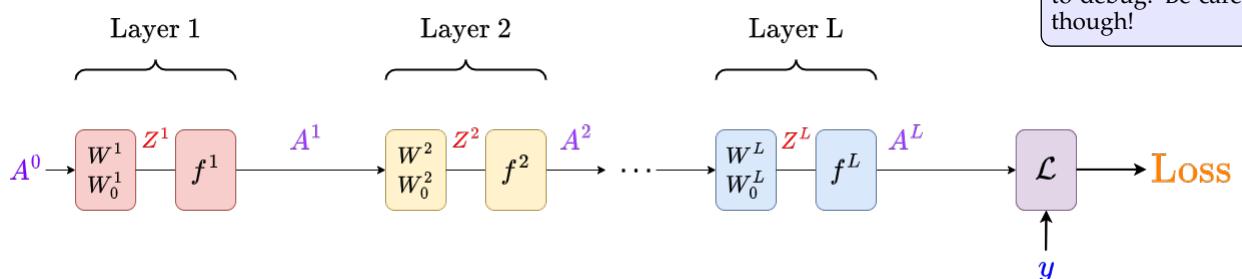
With layers, each layer feeds into the **next**, with no other interaction. And neurons within the same layer do not directly interact with each other, which simplifies our math greatly.

Basically, we have a bunch of functions (neurons) that, within a layer, have nothing to do with each other, and only **output** to the **next** layer of similar functions.

So, we can often **oversimplify** our model by thinking of each layer as like a "big" function, taking in a vector of size m^l and outputting a vector of size n^l .

Our main concern is making sure we have agreement of **dimensions!**

So, here's how our model looks now:



In fact, if you just rearranging your matrices and transposing them can be a helpful way to debug. Be careful, though!

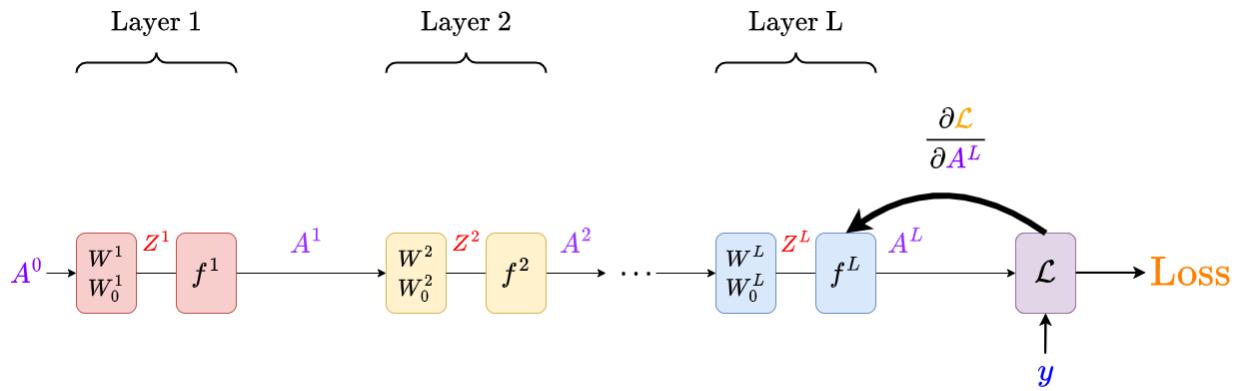
Pretty much the same! Only major difference: swapped scalars for vectors, and vectors for matrices (represented by switching to uppercase)

And, we do backprop the same way, too.

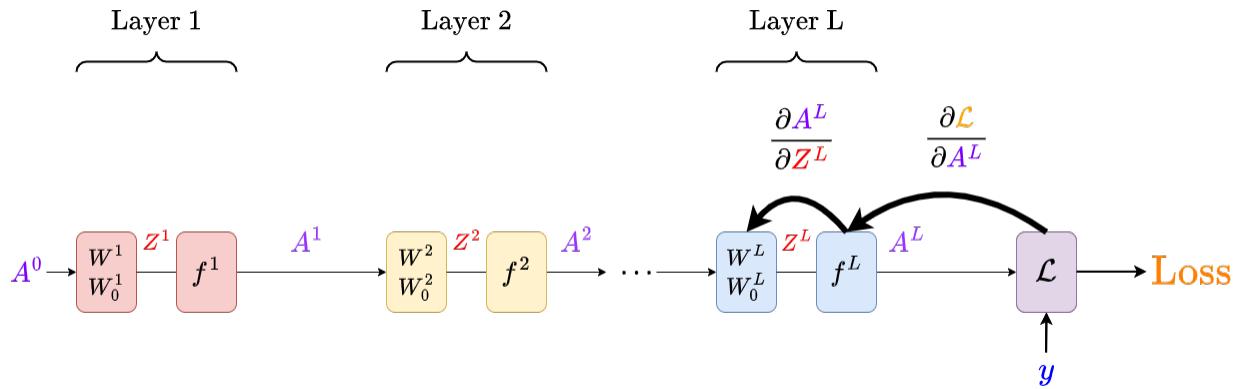
Here, we're not going to explain much as we go: all we're doing is getting the **derivatives** we need for our **chain rule**!

As we go **backwards**, we can build the gradient for each **weight** we come across, in the way we described above.

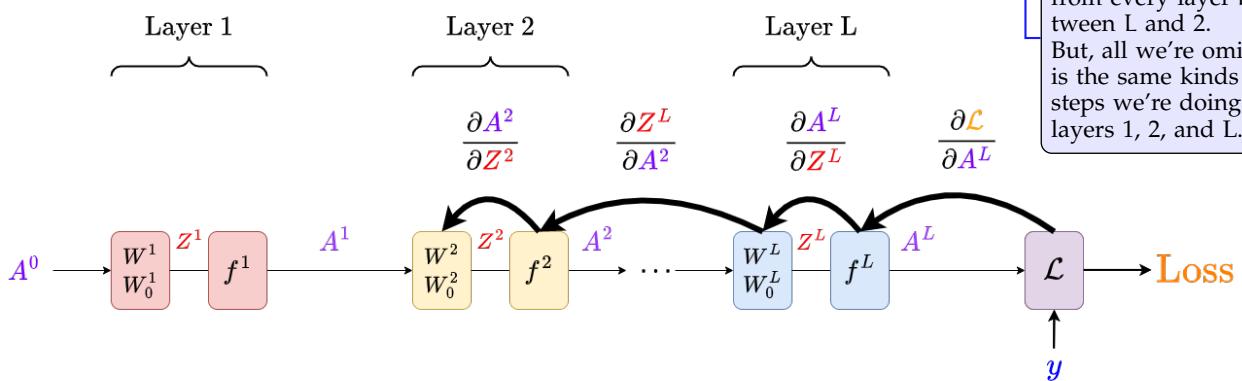
As always, we start from the loss function:



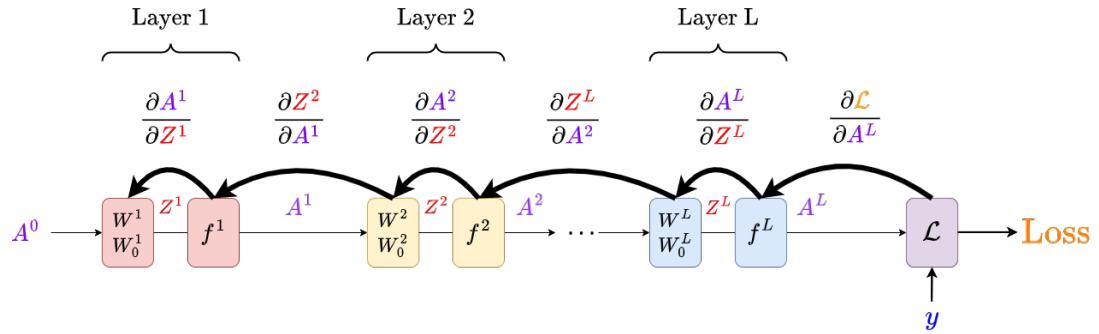
Take another step:



We'll pick up the pace: we'll jump to layer 2 and get its gradient.



Now, we finally get to layer 1!



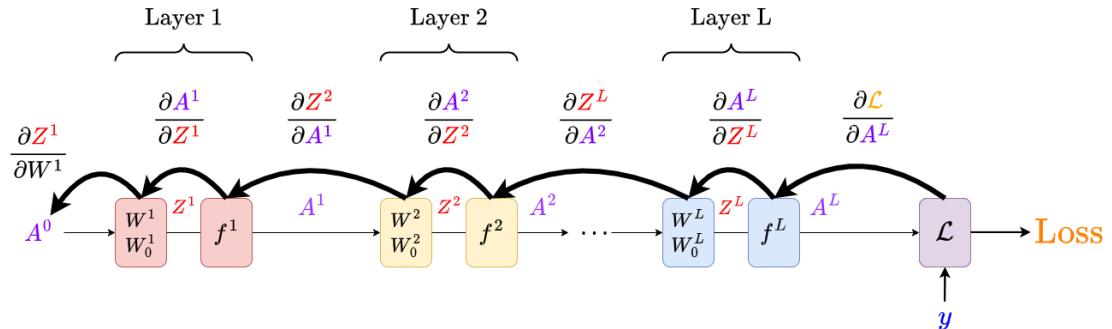
We finish off by getting what we're after: the gradient for W^1 .

Notation 257

We depict neural network gradient descent using the below diagram (outside the box):

The **right-facing straight** arrows come **first**: they're part of the **forward pass**, where we get all of our values.

The **left-facing curved** arrows come **after**: they represent the **back-propagation** of the gradient.



And, with this, we can rewrite our general equation for neural network gradients.

6.5.16 How the Chain Rule changes in Matrix form

As we discussed before, we can't just add onto our weight gradient to reach another layer: the final term

$$\frac{\partial Z^\ell}{\partial W^\ell} \quad (6.53)$$

Ends our chain rule when we add it: W^ℓ isn't part of the input or output, so it doesn't connect to the previous layer.

So, for this section, we'll add it **separately** at the end of our chain rule:

$$\frac{\partial \mathcal{L}}{\partial W^\ell} = \underbrace{\frac{\partial \mathcal{L}}{\partial Z^\ell}}_{\text{Weight link}} \cdot \underbrace{\left(\frac{\partial \mathcal{L}}{\partial Z^\ell} \right)^\top}_{\text{Other layers}}$$

That way, we can add onto $\partial \mathcal{L} / \partial Z^\ell$ without worrying about the weight derivative.

Notice two minor changes caused by the switch to matrices:

- The order has to be **reversed**.
- We also have to do some weird **transposing**.

Both of these mostly boil down to trying to be careful about **shape**/dimension agreement.

Notation 258

The **gradient** $\nabla_{W^\ell} \mathcal{L}$ for a neural network is given as:

$$\frac{\partial \mathcal{L}}{\partial W^\ell} = \underbrace{\frac{\partial \mathcal{L}}{\partial Z^\ell}}_{\text{Weight link}} \cdot \underbrace{\left(\frac{\partial \mathcal{L}}{\partial Z^\ell} \right)^\top}_{\text{Other layers}}$$

There are also deeper interpretations, but they aren't worth digging into for now.

We get our remaining terms $\partial \mathcal{L} / \partial Z^\ell$ by our usual chain rule:

$$\frac{\partial \mathcal{L}}{\partial Z^\ell} = \underbrace{\left(\frac{\partial A^\ell}{\partial Z^\ell} \right)}_{\text{Layer } \ell} \cdot \underbrace{\left(\dots \right)}_{\text{Layer L-1}} \cdot \underbrace{\left(\frac{\partial Z^{L-1}}{\partial A^{L-2}} \cdot \frac{\partial A^{L-1}}{\partial Z^{L-1}} \right)}_{\text{Layer L}} \cdot \underbrace{\left(\frac{\partial Z^L}{\partial A^{L-1}} \cdot \frac{\partial A^L}{\partial Z^L} \right)}_{\text{Loss unit}} \cdot \underbrace{\left(\frac{\partial \mathcal{L}}{\partial A^L} \right)}_{\text{Loss unit}}$$

This is likely our most important equation in this chapter!

6.5.17 Relevant Derivatives

If you aren't interesting in understanding matrix derivatives, here we provide the general format of each of the derivatives we care about.

Notation 259

Here, we give useful **derivatives** for **neural network gradient descent**.

Loss is not given, so we can't compute it, as before:

$$\overbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{A}^L}}$$

We get the same result for each of these terms as we did before, except in matrix form.

$$\overbrace{\frac{\partial \mathbf{Z}^\ell}{\partial \mathbf{W}^\ell}}^{(m^\ell \times 1)} = \mathbf{A}^{\ell-1}$$

$$\overbrace{\frac{\partial \mathbf{Z}^\ell}{\partial \mathbf{A}^{\ell-1}}}^{(m^\ell \times n^\ell)} = \mathbf{W}^\ell$$

The last one is actually pretty different from before:

$$\overbrace{\frac{\partial \mathbf{a}^\ell}{\partial \mathbf{z}^\ell}}^{(n^\ell \times n^\ell)} = \begin{bmatrix} f'(\mathbf{z}_1^\ell) & 0 & 0 & \cdots & 0 \\ 0 & f'(\mathbf{z}_2^\ell) & 0 & \cdots & 0 \\ 0 & 0 & f'(\mathbf{z}_3^\ell) & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & f'(\mathbf{z}_r^\ell) \end{bmatrix}$$

Where r is the length of \mathbf{Z}^ℓ .

In short, we only have the \mathbf{z}_i derivative on the i^{th} diagonal.

Example: Suppose you have the activation $f(\mathbf{z}) = \mathbf{z}^2$.

Why this is will be explained in the matrix derivative notes.

Your pre-activation might be

$$\mathbf{z}^\ell = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \tag{6.54}$$

The output would be

$$\mathbf{a}^{\ell} = f(\mathbf{z}^{\ell}) = \begin{bmatrix} 1 \\ 2^2 \\ 3^2 \end{bmatrix} \quad (6.55)$$

But the derivative would be:

$$f(z) = 2z \quad (6.56)$$

Which, gives our matrix derivative as:

$$\frac{\partial \mathbf{a}^{\ell}}{\partial \mathbf{z}^{\ell}} = \begin{bmatrix} 2 \cdot 1 & 0 & 0 \\ 0 & 2 \cdot 2 & 0 \\ 0 & 0 & 2 \cdot 3 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 6 \end{bmatrix}$$

If you want to be able to **derive** some of the derivatives, without reading the matrix derivative section, just use this formula for vector derivatives:

If you have time, do read - you won't understand what you're doing otherwise!

$$\frac{\partial \mathbf{w}}{\partial \mathbf{v}} = \left\{ \begin{array}{cccc} \frac{\partial w_1}{\partial v_1} & \frac{\partial w_1}{\partial v_2} & \dots & \frac{\partial w_1}{\partial v_m} \\ \frac{\partial w_2}{\partial v_1} & \frac{\partial w_2}{\partial v_2} & \dots & \frac{\partial w_2}{\partial v_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial w_n}{\partial v_1} & \frac{\partial w_n}{\partial v_2} & \dots & \frac{\partial w_n}{\partial v_m} \end{array} \right\} \quad (6.57)$$

Column j matches w_j

Row i matches v_i

We can use this for scalars as well: we just treat them as a vector of length 1.

With some cleverness, you can derive the Scalar/Matrix and Matrix/Scalar derivatives as well.

This is contained in the matrix derivatives chapter.

6.6 Training

6.6.1 Comments

A few important side notes on training. First, on derivatives:

Concept 260

Sometimes, depending on your **loss** and **activation** function, it may be easier to directly compute

$$\frac{\partial \mathcal{L}}{\partial Z^L}$$

Than it is to find

$$\partial \mathcal{L} / \partial A^L \text{ and } \partial A^L / \partial Z^L$$

So, our algorithm may change slightly.

Another thought: initialization.

Concept 261

We typically try to pick a **random initialization**. This does two things:

- Allows us to avoid weird **numerical** and **symmetry** issues that happen when we start with $W_{ij} = 0$.
- We can hopefully find different **local minima** if we run our algorithm multiple times.
 - This is also helped by picking **random data points** in **SGD** (our typical algorithm).

Here, we choose our **initialization** from a **Gaussian** distribution, if you know what that is.

6.6.2 Pseudocode

Our training algorithm for backprop can follow smoothly from what we've laid out.

If you do not know a gaussian distribution, that shouldn't be a problem. It is also known as a "normal" distribution.

SGD-NEURAL-NET($\mathcal{D}_n, T, L, (m^1, \dots, m^L), (f^1, \dots, f^L), \text{Loss}$)

```

1  for every layer:
2      Randomly initialize
3          the weights in every layer
4          the biases in every layer
5
6  While termination condition not met:
7      Get random data point i
8      Keep track of time t
9
10     Do forward pass
11         for every layer:
12             Use previous layer's output: get pre-activation
13             Use pre-activation: get new output, activation
14
15     Get loss: forward pass complete
16
17     Do back-propagation
18         for every layer in reversed order:
19             If final layer: #Loss function
20                 Get  $\partial \mathcal{L} / \partial A^L$ 
21
22             Else:
23                 Get  $\partial \mathcal{L} / \partial A^\ell$ : #Link two layers
24                  $(\partial Z^{\ell+1} / \partial A^\ell) * (\partial \mathcal{L} / \partial Z^{\ell+1})$ 
25
26                 Get  $\partial \mathcal{L} / \partial Z^\ell$ : #Within layer
27                  $(\partial A^\ell / \partial Z^\ell) * (\partial \mathcal{L} / \partial A^\ell)$ 
28
29             Compute weight gradients:
30                 Get  $\partial \mathcal{L} / \partial W^\ell$ : #Weights
31                  $\partial Z^\ell / \partial W^\ell = A^{\ell-1}$ 
32                  $(\partial Z^\ell / \partial W^\ell) * (\partial \mathcal{L} / \partial Z^\ell)$ 
33
34             Get  $\partial \mathcal{L} / \partial W_0^\ell$ : #Biases
35                  $\partial \mathcal{L} / \partial W_0^\ell = (\partial \mathcal{L} / \partial Z^\ell)$ 
36
37             Follow Stochastic Gradient Descent (SGD): #Take step
38                 Update weights:
39                      $W^\ell = W^\ell - (\eta(t) * (\partial \mathcal{L} / \partial W^\ell))$ 
40
41                 Update biases:
42                      $W_0^\ell = W_0^\ell - (\eta(t) * (\partial \mathcal{L} / \partial W_0^\ell))$ 
43
44     Return final neural network with weights and biases    Last Updated: 03/09/23 21:42:28

```

Terms

- Forward pass
- Back-Propagation
- Weight gradient
- Matrix Derivative
- Partial Derivative
- Multivariable Chain Rule
- Total Derivative
- Size of a matrix
- Planar Approximation
- Scalar/scalar derivative
- Vector/scalar derivative
- Scalar/vector derivative
- Vector/vector derivative

X. Matrix Derivatives

In general, we want to be able to combine the powers of matrices and calculus:

- **Matrices:** the ability to store lots of **data**, and do fast linear operations on all that data at the **same time**.

Example: Consider

$$\mathbf{w}^T \mathbf{x} = \begin{bmatrix} w_1 & w_2 & \cdots & w_m \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \sum_{i=1}^m x_i w_i \quad (6.58)$$

In this case, we're able to do m different **multiplications** at the same time! This is what we like about matrices.

- **Calculus:** analyzing the way different variables are **related**: how does changing x affect y ?

In this case, we're thinking about vectors as $(m \times 1)$ matrices.

Example: Suppose we have

$$\frac{\partial f}{\partial x_1} = 10 \quad \frac{\partial f}{\partial x_2} = -5 \quad (6.59)$$

Now we know that, if we increase x_1 , we increase f . This **understanding** of variables is what we like about derivatives.

Concept 262

Matrix derivatives allow us to find **relationships** between large volumes of **data**.

- These "relationships" are **derivatives**: consider dy/dx . How does y change if we modify x ? Currently, we only have **scalar derivatives**.
- This "data" is stored as **matrices**: blocks of data, that we can do linear operations (matrix multiplication) on.

Our goal is to work with many scalar derivatives at the **same time**.

In order to do that, we can apply some **derivative** rules, but we have to do it in a way that **agrees** with **matrix** math.

Our work is a careful balancing act between getting the **derivatives** we want, without violating the **rules** of matrices (and losing what makes them useful!)

Example: When we multiply two matrices, their inner shape has to match: in the below case, they need to share a dimension b .

$$\underbrace{X}_{(a \times b)} \quad \underbrace{Y}_{(b \times c)} \quad (6.60)$$

We can't do anything that would **violate** this rule: otherwise, our **equations** don't make sense, and we get stuck. This means we need to build our math carefully.

First, we'll look at the **properties** of derivatives. Then figure out how to usefully apply them to **vectors**, and then **matrices**.

X.1 Review: Partial Derivatives

One more comment, though - we may have many different variables floating around. This means we **have** to use the multivariable **partial derivative**.

Definition 263

The **partial derivative**

$$\frac{\partial B}{\partial A}$$

Is used when there may be **multiple variables** in our functions.

The rule of the partial derivative is that we keep every **independent** variable other than A and B **fixed**.

Example: Consider $f(x, y) = 2x^2y$.

$$\frac{\partial f}{\partial x} = 2(2x)y \quad (6.61)$$

Here, we kept y *fixed* - we treat it as if it were an unchanging **constant**.

Using the partial derivative lets us keep our work tidy: if **many** variables were allowed to **change** at the same time, it could get very confusing.

If this is too complicated, we can change those variables *one at a time*. We get a partial derivative for each of them, holding the others **constant**.

Imagine keeping track of k different variables x_i with k different changes Δx_i at the same time! That's a headache.

Our **total** derivative is the result of all of those different variables, **added** together. This is how we get the **multi-variable chain rule**.

Definition 264

The **multi-variable chain rule** in 3-D ($\{x, y, z\}$) is given as

$$\frac{df}{ds} = \underbrace{\frac{\partial f}{\partial x} \frac{\partial x}{\partial s}}_{\text{only modify } x} + \underbrace{\frac{\partial f}{\partial y} \frac{\partial y}{\partial s}}_{\text{only modify } y} + \underbrace{\frac{\partial f}{\partial z} \frac{\partial z}{\partial s}}_{\text{only modify } z}$$

If we have k variables $\{x_1, x_2, \dots, x_k\}$ we can generalize this as:

$$\frac{df}{ds} = \sum_{i=1}^k \underbrace{\frac{\partial f}{\partial x_i} \frac{\partial x_i}{\partial s}}_{x_i \text{ component}}$$

X.2 Thinking about derivatives

The typical definition of derivatives

$$\lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \quad (6.62)$$

Gives an *idea* of what sort of things we're looking for. It reminds us of one piece of information we need:

- Our derivative **depends** on the **current position** x we are taking the derivative at.

We need this because derivative are **local**: the relationship between our variables might change if we move to a different **position**.

But, the problem with vectors is that each component can act **separately**: if we have a vector, we can change in many different "directions".

$$A = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad B = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad (6.63)$$

Example: Suppose we want a derivative $\partial B / \partial A$: $\Delta a_1, \Delta a_2$, and Δa_3 could each, separately, have an effect on Δb_1 and/or Δb_2 . That requires 6 different derivatives, $\partial b_1 / \partial a_j$.

Every component of the input A can potentially modify **every** component of the output B .

3 dimensions of A times
2 dimensions of B: 6
combinations.

One solution we could try is to just collect all of these derivatives into a **vector** or **matrix**.

Concept 265

For the **derivative** between two objects (scalars, vectors, matrices) A and B

$$\frac{\partial B}{\partial A}$$

We need to get the **derivatives**

$$\frac{\partial b_j}{\partial a_i}$$

between every **pair** of elements a_i, b_j : each pair of elements could have a **relationship**.

The total number of elements (or "size") is...

$$\text{Size}\left(\frac{\partial B}{\partial A}\right) = \text{Size}(B) * \text{Size}(A)$$

Collecting these values into a **matrix** will give us all the information we need.

But, how do we gather them? What should the **shape** look like? Should we **transpose** our matrix or not?



X.3 Derivatives: Approximation

To answer this, we need to ask ourselves *why* we care about these derivatives: their **structure** will be based on what we need them for.

- We care about the **direction of greatest decrease**, the gradient. For example, we might want to adjust weight vector w to reduce \mathcal{L} .
- We also want other derivatives that have the **same** behavior, so we can combine them using the **chain rule**.

Let's focus on the first point: we want to **minimize** \mathcal{L} . Our focus is the **change** in \mathcal{L} , $\Delta\mathcal{L}$.

We want to take steps that reduce our loss \mathcal{L} .

$$\frac{\partial \mathcal{L}}{\partial w} \approx \frac{\text{Change in } \mathcal{L}}{\text{Change in } w} = \frac{\Delta \mathcal{L}}{\Delta w} \quad (6.64)$$

Thus, we **solve** for $\Delta\mathcal{L}$:

All we do is multiply both sides by Δw .

$$\Delta \mathcal{L} \approx \frac{\partial \mathcal{L}}{\partial w} \Delta w \quad (6.65)$$

Since this derivation was gotten using scalars, we might need a **different** type of multiplication for our **vector** and **matrix** derivatives.

Concept 266

We can use derivatives to **approximate** the change in our output based on our input:

$$\Delta \mathcal{L} \approx \frac{\partial \mathcal{L}}{\partial w} * \Delta w$$

Where the $*$ symbol represents some type of **multiplication**.

We can think of this as a **function** that takes in change in Δw , and returns an **approximation** of the loss.

We already understand **scalar** derivatives, so let's move on to the **gradient**.

X.4 The Gradient: a vector input, scalar output

Our plan is to look at every derivative combination of scalars, vectors, and matrices we can.

First, we consider:

$$\frac{\partial(\text{Scalar})}{\partial(\text{Vector})} = \frac{\partial s}{\partial v} \quad (6.66)$$

We'll take s to be our scalar, and v to be our vector. So, our input is a **vector**, and our output is a **scalar**.

$$\Delta v \rightarrow [f] \rightarrow \Delta s \quad (6.67)$$

How do we make sense of this? Well, let's write Δv_i explicitly:

$$\underbrace{\begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix}}_{\Delta v} \rightarrow \Delta s \quad (6.68)$$

We can see that we have m different **inputs** we can change in order to change our **one** output.

So, our derivative needs to have m different **elements**: one for each element v_i .

X.5 Finding the scalar/vector derivative

But how do we shape our matrix? Let's look at our rule.

$$\Delta s \approx \frac{\partial s}{\partial v} \star \Delta v \quad \text{or} \quad \Delta s \approx \frac{\partial s}{\partial v} \star \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix} \quad (6.69)$$

How do we get Δs ? We have so many variables. Let's focus on them one at a time: breaking Δv into Δv_i , so we'll try to consider each v_i separately.

One problem, though: how can we treat each **derivative** separately? Each Δv_i will move our position, which can change a different derivative v_k : they can **affect** each other.

It's usually possible to change each v_i , so we have to look at every one of them.

X.6 Review: Planar Approximation

We'll resolve this the same way we did in chapter 3, **gradient descent**: by taking advantage of the "planar approximation".

The solution is this: assume your function is **smooth**. The **smaller** a step you take, the **less** your derivative has a chance to change.

Example: Take $f(x) = x^2$.

- If we go from $x = 1 \rightarrow 2$, then our derivative goes from $f'(x) = 2 \rightarrow 4$.
- Let's **shrink** our step. We go from $x = 1 \rightarrow 1.01$, our derivative goes from $f'(x) = 2 \rightarrow 2.02$.
 - Our derivative is almost the same!

This isn't true for big steps, but eventually, if your step is small enough, then the derivative will barely change.

if we take a small enough step Δv_i , then, if our function is **smooth**, then the derivative will hardly change!

So, if we zoom in enough (shrink the scale of change), then we can **pretend** the derivative is **constant**.

You could imagine repeatedly shrinking the size of our step, until the change in the derivatives is basically unnoticeable.

Concept 267

If you have a **smooth function**, then...

If you take sufficiently **small steps**, then you can treat the derivatives as **constant**.

Clarification

This section is **optional**.

We can describe "sufficiently small steps" in a more mathematical way:

Our goal is for $f'(x)$ to be **basically constant**: it doesn't change much. $\Delta f'(x)$ is **small**.

Let's say it can't change more than δ .

If you want

- $\Delta f'(x)$ to be very small ($|\Delta f'(x)| < \delta$)
- It has been proven that...
 - can take a small enough step $|\Delta x| < \epsilon$, and to get that result.

One way to describe this is to say that our function is (locally) **flat**: it looks like some kind of plane/hyperplane.

The word "locally" represents the small step size: we stay in the "local area".

Clarification 268

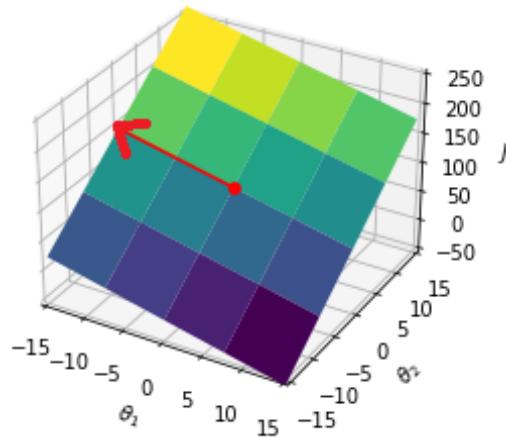
Why is this **true**? Because a **hyperplane** can be represented using our **linear** function

$$f(x) \approx \theta^T x + \theta_0 = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_m x_m$$

If we take a derivative:

$$\frac{\partial f}{\partial x_i} = \theta_i$$

That derivative is a **constant**! It's doesn't change based on **position**.

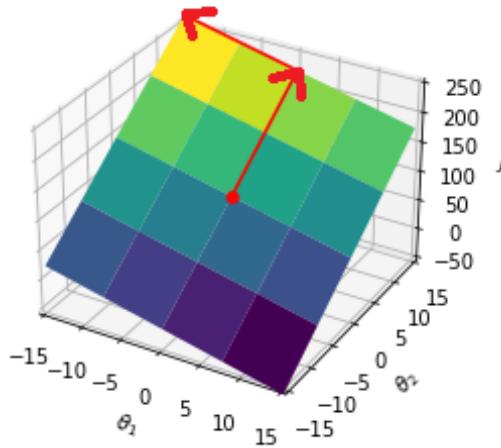
Movement in θ_1 on J 

If we take very small steps, we can approximate our function as **flat**.

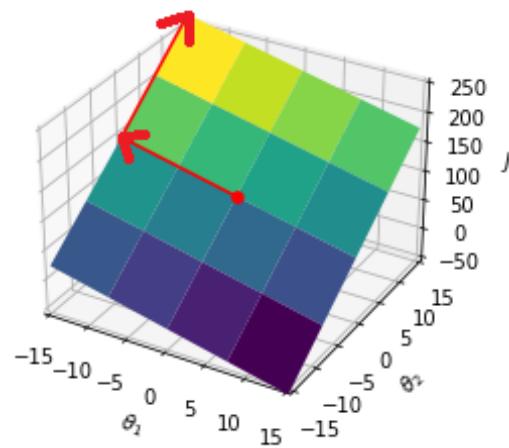
Why does this help? If our derivative doesn't **change**, we can combine multiple steps. You can take multiple steps Δv_i and the order doesn't matter.

So, you can combine your steps or separate them easily.

Combining two movements



Combining two movements



We can break up our big step into two smaller steps that are truly independent: order doesn't matter.

With that, we can add up all of our changes:

$$\Delta s = \Delta s_{\text{from } v_1} + \Delta s_{\text{from } v_2} + \dots + \Delta s_{\text{from } v_m} \quad (6.70)$$

X.7 Our scalar/vector derivative

From this, we can get an **approximated** version of the MV chain rule.

Definition 269

The **multivariable chain rule approximation** looks similar to the multivariable chain rule, but for finite changes Δx_i .

In 3-D, we get

$$\Delta \mathbf{f} = \underbrace{\frac{\partial \mathbf{f}}{\partial \mathbf{x}} \Delta \mathbf{x}}_{\text{x component}} + \underbrace{\frac{\partial \mathbf{f}}{\partial \mathbf{y}} \Delta \mathbf{y}}_{\text{y component}} + \underbrace{\frac{\partial \mathbf{f}}{\partial \mathbf{z}} \Delta \mathbf{z}}_{\text{z component}}$$

In general, we have

$$\Delta \mathbf{f} = \sum_{i=1}^m \underbrace{\frac{\partial \mathbf{f}}{\partial \mathbf{x}_i} \Delta \mathbf{x}_i}_{\text{x}_i \text{ component}}$$

This function lets us add up the effect each component has on our output, using **derivatives**.

This gives us what we're looking for:

$$\Delta s \approx \sum_{i=1}^m \frac{\partial s}{\partial v_i} \Delta v_i \quad (6.71)$$

If we circle back around to our original approximation:

$$\sum_{i=1}^m \frac{\partial s}{\partial v_i} \Delta v_i = \frac{\partial s}{\partial \mathbf{v}} * \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix} \quad (6.72)$$

When we look at the left side, we're multiplying pairs of components, and then adding them. That sounds similar to a **dot product**.

$$\sum_{i=1}^m \frac{\partial s}{\partial v_i} \Delta v_i = \begin{bmatrix} \frac{\partial s}{\partial v_1} \\ \frac{\partial s}{\partial v_2} \\ \vdots \\ \frac{\partial s}{\partial v_m} \end{bmatrix} \cdot \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix} \quad (6.73)$$

This gives us our derivative: it contains all of the **element-wise** derivatives we need, and in a **useful** form!

Definition 270

If s is a **scalar** and v is an $(m \times 1)$ **vector**, then we define the **derivative** or **gradient** $\frac{\partial s}{\partial v}$ as fulfilling:

$$\Delta s = \frac{\partial s}{\partial v} \cdot \Delta v$$

Or, equivalently,

$$\Delta s = \left(\frac{\partial s}{\partial v} \right)^T \Delta v$$

Thus, our derivative must be an $(m \times 1)$ vector

$$\frac{\partial s}{\partial v} = \begin{bmatrix} \frac{\partial s}{\partial v_1} \\ \frac{\partial s}{\partial v_2} \\ \vdots \\ \frac{\partial s}{\partial v_m} \end{bmatrix} = \begin{bmatrix} \frac{\partial s}{\partial v_1} \\ \frac{\partial s}{\partial v_2} \\ \vdots \\ \frac{\partial s}{\partial v_m} \end{bmatrix}$$

We can see the shapes work out in our matrix multiplication:

$$\Delta s = \underbrace{\left(\frac{\partial s}{\partial v} \right)^T}_{(1 \times 1)} \underbrace{\Delta v}_{(1 \times m)} \underbrace{\Delta s}_{(m \times 1)} \quad (6.74)$$

X.8 Vector derivative: a scalar input, vector output

Now, we want to try the flipped version: we swap our vector and our scalar.

$$\frac{\partial(\text{Vector})}{\partial(\text{Scalar})} = \frac{\partial w}{\partial s} \quad (6.75)$$

We'll take s to be our scalar, and w to be our vector. So, our input is a **scalar**, and our output is a **vector**.

$$\Delta s \longrightarrow [f] \longrightarrow \Delta w \quad (6.76)$$

Note that we're using vector w instead of v this time: this will be helpful for our vector/vector derivative: we can use both.

Written explicitly, like before:

$$\Delta s \longrightarrow \underbrace{\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix}}_{\Delta w} \quad (6.77)$$

We have 1 **input**, that can affect n different **outputs**. So, our derivative needs to have n elements.

Again, let's look at our **approximation rule**:

$$\Delta w \approx \frac{\partial w}{\partial s} * \Delta s \quad \text{or} \quad \underbrace{\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix}}_{\Delta w} \approx \frac{\partial w}{\partial s} * \Delta s \quad (6.78)$$

Here, we can't do a **dot product**: we're multiplying our derivative by a **scalar**. Plus, we'd get the **same shape** as before: we might **mix up** our derivatives.

X.9 Working with the vector derivative

How do we get each of our terms Δw_i ?

Well, each term is **separately** affected by Δs : we have our terms $\partial w_i / \partial s$.

So, if we take these terms **individually**, treating it as a scalar derivative, we get:

$$\Delta w_i = \frac{\partial w_i}{\partial s} \Delta s \quad (6.79)$$

If you're ever confused with matrix math, thinking about individual elements is often a good way to figure it out!

Since we only have **one** input, we don't have to worry about **planar** approximations: we only take one step, in the s direction.

In our matrix, we get:

$$\mathbf{w} = \begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix} = \begin{bmatrix} \Delta s(\partial w_1 / \partial s) \\ \Delta s(\partial w_2 / \partial s) \\ \vdots \\ \Delta s(\partial w_n / \partial s) \end{bmatrix} \quad (6.80)$$

This works out for our equation above!

It could be tempting to think of our derivative $\partial w / \partial s$ as a **column vector**: we just take w and just differentiate each element. Easy!

In fact, this *is* a valid convention. However, this conflicts with our previous derivative: they're both column vectors!

Not only is it **confusing**, but it also will make it harder to do our **vector/vector** derivative.

So, what do we do? We refer back to the equation we used last time:

$$\Delta w = \left(\frac{\partial w}{\partial s} \right)^T \Delta s \quad (6.81)$$

We take the **transpose**! That way, one derivative is a column vector, and the other is a row vector. And, we know that this equation works out from the work we just did.

$$\Delta w = \left[\frac{\partial w_1}{\partial s}, \frac{\partial w_2}{\partial s}, \dots, \frac{\partial w_n}{\partial s} \right]^T \Delta s \quad (6.82)$$

Clarification 271

We mentioned that it is a valid **convention** to have that **vector derivative** be a **column vector**, and have our **gradient** be a **row vector**.

This is **not** the convention we will use in this class - you will be confused if we try!

That means, for whatever **notation** we use here, you might see the **transposed** version elsewhere. They mean exactly the **same** thing!

$$\underbrace{\Delta w}_{\substack{(n \times 1)}} = \underbrace{\left(\frac{\partial w}{\partial s} \right)}_{\substack{(n \times 1)}}^T \underbrace{\Delta s}_{(1 \times 1)} \quad (6.83)$$

As we can see, the dimensions check out.

Definition 272

If s is a **scalar** and w is an $(n \times 1)$ **vector**, then we define the **vector derivative** $\partial w / \partial s$ as fulfilling:

$$\Delta w = \left(\frac{\partial w}{\partial s} \right)^T \Delta s$$

Thus, our derivative must be a $(1 \times n)$ vector

$$\frac{\partial w}{\partial s} = \left[\frac{\partial w_1}{\partial s}, \frac{\partial w_2}{\partial s}, \dots, \frac{\partial w_n}{\partial s} \right]$$

X.10 Vectors and vectors: vector input, vector output

We'll be combining our two previous derivatives:

$$\frac{\partial(\text{Vector})}{\partial(\text{Vector})} = \frac{\partial w}{\partial v} \quad (6.84)$$

v and w are both **vectors**: thus, input and output are both **vectors**.

$$\Delta v \rightarrow [f] \rightarrow \Delta w \quad (6.85)$$

Written out, we get:

$$\underbrace{\begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix}}_{\Delta v} \rightarrow \underbrace{\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix}}_{\Delta w} \quad (6.86)$$

Something pretty complicated! We have m inputs and n outputs. Every input can interact with every output.

So, our derivative needs to have mn different elements. That's a lot!

X.11 The vector/vector derivative

We return to our rule from before. We'll skip the star notation, and jump right to the equation we've gotten for both of our two previous derivatives: _____

Hopefully, since we're combining two different derivatives, we should be able to use the same rule here.

$$\Delta \mathbf{w} = \left(\frac{\partial \mathbf{w}}{\partial \mathbf{v}} \right)^T \Delta \mathbf{v} \quad (6.87)$$

With mn different elements, this could get messy very fast. Let's see if we can focus on only **part** of our problem:

$$\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix} = \left(\frac{\partial \mathbf{w}}{\partial \mathbf{v}} \right)^T \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix} \quad (6.88)$$

One input

We could try focusing on just a single **input** or a single **output**, to simplify things. Let's start with a single v_i .

$$\underbrace{\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix}}_{\Delta w \text{ from } v_i} = \left(\frac{\partial \mathbf{w}}{\partial v_i} \right)^T \Delta v_i \quad (6.89)$$

We now have a simpler case: $\partial \text{Vector} / \partial \text{Scalar}$. We're familiar with this case!

$$\frac{\partial \mathbf{w}}{\partial v_i} = \left[\frac{\partial w_1}{\partial v_i}, \frac{\partial w_2}{\partial v_i}, \dots, \frac{\partial w_n}{\partial v_i} \right] \quad (6.90)$$

We get a vector. What if the **output** is a scalar instead?

One output

$$\Delta w_j = \left(\frac{\partial w_j}{\partial \mathbf{v}} \right)^T \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix} \quad (6.91)$$

We have $\partial \text{Scalar} / \partial \text{Vector}$:

$$\frac{\partial \mathbf{w}_j}{\partial \mathbf{v}} = \begin{bmatrix} \partial w_j / \partial v_1 \\ \partial w_j / \partial v_2 \\ \vdots \\ \partial w_j / \partial v_m \end{bmatrix} \quad (6.92)$$

So, our vector-vector derivative is a **generalization** of the two derivatives we did before!

It seems that extending along the **vertical** axis changes our v_i value, while moving along the **horizontal** axis changes our w_j value.

X.12 General derivative

You might have a hint of what we get: one derivative stretches us along **one** axis, the other along the **second**.

To prove it to ourselves, we can **combine** these concepts. We'll handle solve as if we have one vector, and then **substitute** in the second one.

Concept 273

One way to **simplify** our work is to treat **vectors** as **scalars**, and then convert them back into **vectors** after applying some math.

We have to be careful - any operation we apply to the **scalar**, has to match how the **vector** would behave.

This is **equivalent** to if we just focused on one scalar inside our vector, and then stacked all those scalars back into the vector.

This isn't just a cute trick: it relies on an understanding that, at its **basic** level, we're treating **scalars** and **vectors** and **matrices** as the same type of object: a structured array of numbers.

We'll get into "arrays" later.

As always, our goal is to **simplify** our work, so we can handle each piece of it.

- We treat Δv as a scalar so we can get the simplified derivative.

$$\Delta \mathbf{w} = \left(\frac{\partial \mathbf{w}}{\partial \mathbf{v}} \right)^T \Delta \mathbf{v} \quad (6.93)$$

We'll only expand **one** of our vectors, since we know how to manage **one** of them.

$$\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix} = \left(\frac{\partial w}{\partial v} \right)^T \Delta v \quad (6.94)$$

This time, notice that we **didn't** simplify v to v_i . We didn't **remove** the other elements - we still have a full **vector**. But, let's treat it as if it *were* a scalar.

This comes out to:

$$\frac{\partial w}{\partial v} = \underbrace{\left[\frac{\partial w_1}{\partial v}, \frac{\partial w_2}{\partial v}, \dots, \frac{\partial w_n}{\partial v} \right]}_{\text{Column } j \text{ matches } w_j} \quad (6.95)$$

- Our "answer" is a row vector. But, each of those derivatives is a **column** vector!

Now that we've taken care of ∂w_j (one for each column), we can expand our derivatives in terms of ∂v_i .

First, for w_1 :

$$\frac{\partial w}{\partial v} = \left[\underbrace{\left[\frac{\partial w_1}{\partial v_1}, \frac{\partial w_1}{\partial v_2}, \dots, \frac{\partial w_1}{\partial v_m} \right]}_{\text{Column } j \text{ matches } w_j}, \frac{\partial w_2}{\partial v}, \dots, \frac{\partial w_n}{\partial v} \right] \quad \left. \right\} \text{Row } i \text{ matches } v_i \quad (6.96)$$

And again, for w_2 :

$$\frac{\partial w}{\partial v} = \left[\underbrace{\left[\frac{\partial w_1}{\partial v_1}, \frac{\partial w_2}{\partial v_1} \right]}, \underbrace{\left[\frac{\partial w_1}{\partial v_2}, \frac{\partial w_2}{\partial v_2} \right]}, \dots, \frac{\partial w_n}{\partial v} \right] \quad \left. \right\} \text{Row } i \text{ matches } v_i \quad (6.97)$$

And again, for w_n :

$$\frac{\partial \mathbf{w}}{\partial \mathbf{v}} = \left[\begin{array}{c|c|c} \hline & \text{Column } j \text{ matches } w_j & \\ \hline \begin{bmatrix} \frac{\partial w_1}{\partial v_1} \\ \frac{\partial w_1}{\partial v_2} \\ \vdots \\ \frac{\partial w_1}{\partial v_m} \end{bmatrix}, & \begin{bmatrix} \frac{\partial w_2}{\partial v_1} \\ \frac{\partial w_2}{\partial v_2} \\ \vdots \\ \frac{\partial w_2}{\partial v_m} \end{bmatrix}, & \cdots & \begin{bmatrix} \frac{\partial w_n}{\partial v_1} \\ \frac{\partial w_n}{\partial v_2} \\ \vdots \\ \frac{\partial w_n}{\partial v_m} \end{bmatrix} \\ \hline \end{array} \right] \quad \left. \begin{array}{l} \\ \\ \\ \end{array} \right\} \text{Row } i \text{ matches } v_i \quad (6.98)$$

We have column vectors in our row vector... let's just combine them into a **matrix**.

Definition 274

If

- \mathbf{v} is an $(m \times 1)$ **vector**
- \mathbf{w} is an $(n \times 1)$ **vector**

Then we define the **vector derivative** $\partial \mathbf{w} / \partial \mathbf{v}$ as fulfilling:

$$\Delta \mathbf{w} = \left(\frac{\partial \mathbf{w}}{\partial \mathbf{s}} \right)^T \Delta \mathbf{s}$$

Thus, our derivative must be a $(1 \times n)$ vector

$$\frac{\partial \mathbf{w}}{\partial \mathbf{v}} = \left[\begin{array}{cccc} \hline & \text{Column } j \text{ matches } w_j & & \\ \hline \begin{matrix} \frac{\partial w_1}{\partial v_1} & \frac{\partial w_2}{\partial v_1} & \cdots & \frac{\partial w_n}{\partial v_1} \\ \frac{\partial w_1}{\partial v_2} & \frac{\partial w_2}{\partial v_2} & \cdots & \frac{\partial w_n}{\partial v_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial w_1}{\partial v_m} & \frac{\partial w_2}{\partial v_m} & \cdots & \frac{\partial w_n}{\partial v_m} \end{matrix} & & & \\ \hline \end{array} \right] \quad \left. \begin{array}{l} \\ \\ \\ \end{array} \right\} \text{Row } i \text{ matches } v_i$$

This general form can be used for **any** of our matrix derivatives.

So, our matrix can represent any **combination** of two elements! We just assign each **row** to a v_i component, and each **column** with a w_j component.

X.13 More about the vector/vector derivative

Let's show a specific example: w is (3×1) , v is (2×1) .

$$\frac{\partial w}{\partial v} = \begin{bmatrix} \underbrace{\frac{\partial w_1}{\partial v_1}}_{w_1} & \underbrace{\frac{\partial w_2}{\partial v_1}}_{w_2} & \underbrace{\frac{\partial w_3}{\partial v_1}}_{w_3} \\ \underbrace{\frac{\partial w_1}{\partial v_2}}_{v_1} & \underbrace{\frac{\partial w_2}{\partial v_2}}_{v_2} & \underbrace{\frac{\partial w_3}{\partial v_2}}_{v_2} \end{bmatrix} \quad (6.99)$$

Another way to describe the general case:

Notation 275

Our matrix $\partial w / \partial v$ is entirely filled with **scalar derivatives**

$$\frac{\partial w_j}{\partial v_i} \quad (6.100)$$

Where any one **derivative** is stored in

- Row i
 - m rows total
- Column j
 - n columns total

We can also compress it along either axis (just like how we did to derive this result):

Notation 276

Our matrix $\frac{\partial \mathbf{w}}{\partial \mathbf{v}}$ can be written as

$$\frac{\partial \mathbf{w}}{\partial \mathbf{v}} = \underbrace{\left[\frac{\partial w_1}{\partial v_1}, \frac{\partial w_2}{\partial v_2}, \dots, \frac{\partial w_n}{\partial v} \right]}_{\text{Column } j \text{ matches } w_j}$$

or

$$\frac{\partial \mathbf{w}}{\partial \mathbf{v}} = \left[\begin{array}{c} \frac{\partial \mathbf{w}}{\partial v_1} \\ \frac{\partial \mathbf{w}}{\partial v_2} \\ \vdots \\ \frac{\partial \mathbf{w}}{\partial v_m} \end{array} \right] \left. \right\} \text{Row } i \text{ matches } v_i$$

These compressed forms will be useful for deriving our new and final derivatives, **matrix-scalar** pairs.

X.14 Derivative: matrix/scalar

Now, we have our general form for creating derivatives.

We'll get our derivative of the form

$$\frac{\partial (\text{Matrix})}{\partial (\text{Scalar})} = \frac{\partial \mathbf{M}}{\partial s} \quad (6.101)$$

We have a matrix \mathbf{M} in the shape $(r \times k)$ and a scalar s . Our **input** is a **scalar**, and our **output** is a **matrix**.

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & \cdots & m_{1r} \\ m_{21} & m_{22} & \cdots & m_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ m_{k1} & m_{k2} & \cdots & m_{kr} \end{bmatrix} \quad (6.102)$$

This may seem concerning: before, we divided **inputs** across **rows**, and **outputs** across **columns**. But in this case, we have **no** input axes, and **two** output axes.

Well, let's try to make this work anyway.

What did we do before, when we didn't know how to handle a **new** derivative? We compared it to **old** versions: we built our vector/vector case using the vector/scalar case and the scalar/vector case.

We did this by **compressing** one of our *vectors* into a *scalar* temporarily: this works, because we want to treat each of these objects the **same way**.

We don't know how to work with Matrix/Scalar, but what's the **closest** thing we do know? **Vector/Scalar**.

How do we accomplish that? As we saw above, a matrix is a **vector of vectors**. We could turn it into a **vector of scalars**.

Concept 277

A **matrix** can be thought of as a **column vector** of **row vectors** (or vice versa).

So, we can use our earlier technique and convert the **row vectors** into **scalars**.

We'll replace the **row vectors** in our matrix with **scalars**.

$$\mathbf{M} = \begin{bmatrix} \mathbf{M}_1 \\ \mathbf{M}_2 \\ \vdots \\ \mathbf{M}_k \end{bmatrix} \quad (6.103)$$

Now, we can pretend our matrix is a vector! We've got a derivative for that:

$$\frac{\partial \mathbf{M}}{\partial s} = \left[\frac{\partial \mathbf{M}_1}{\partial s} \quad \frac{\partial \mathbf{M}_2}{\partial s} \quad \dots \quad \frac{\partial \mathbf{M}_r}{\partial s} \right] \quad (6.104)$$

Aha - we have the same form that we did for our vector/vector derivative! Each derivative is a column vector. Let's expand it out:

$$\frac{\partial \mathbf{M}}{\partial s} = \left\{ \left[\begin{array}{c} \frac{\partial m_{11}}{\partial s} \\ \frac{\partial m_{12}}{\partial s} \\ \vdots \\ \frac{\partial m_{1r}}{\partial s} \end{array} \right], \left[\begin{array}{c} \frac{\partial m_{21}}{\partial s} \\ \frac{\partial m_{22}}{\partial s} \\ \vdots \\ \frac{\partial m_{2r}}{\partial s} \end{array} \right], \dots, \left[\begin{array}{c} \frac{\partial m_{k1}}{\partial s} \\ \frac{\partial m_{k2}}{\partial s} \\ \vdots \\ \frac{\partial m_{kr}}{\partial s} \end{array} \right] \right\} \quad \text{Row } i \text{ matches } m_{?i} \quad (6.105)$$

Definition 278

If \mathbf{M} is a matrix in the shape $(r \times k)$ and s is a scalar,

Then we define the **matrix derivative** $\partial\mathbf{M}/\partial s$ as the $(k \times r)$ matrix:

$$\frac{\partial \mathbf{M}}{\partial s} = \left[\begin{array}{cccc} \frac{\partial m_{11}}{\partial s} & \frac{\partial m_{21}}{\partial s} & \cdots & \frac{\partial m_{k1}}{\partial s} \\ \frac{\partial m_{12}}{\partial s} & \frac{\partial m_{22}}{\partial s} & \cdots & \frac{\partial m_{k2}}{\partial s} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial m_{1r}}{\partial s} & \frac{\partial m_{2r}}{\partial s} & \cdots & \frac{\partial m_{kr}}{\partial s} \end{array} \right] \quad \begin{array}{l} \text{Column } j \text{ matches } m_{j?} \\ \text{Row } i \text{ matches } m_{?i} \end{array}$$

This matrix has the transpose of the shape of \mathbf{M} .

X.15 Derivative: scalar/matrix

We'll get our derivative of the form

$$\frac{\partial(\text{Scalar})}{\partial(\text{Matrix})} = \frac{\partial s}{\partial \mathbf{M}} \quad (6.106)$$

We have a matrix \mathbf{M} in the shape $(r \times k)$ and a scalar s . Our **input** is a **matrix**, and our **output** is a **scalar**.

Let's do what we did last time: break it into **row vectors**.

$$\mathbf{M} = \begin{bmatrix} \mathbf{M}_1 \\ \mathbf{M}_2 \\ \vdots \\ \mathbf{M}_k \end{bmatrix} \quad (6.107)$$

The gradient for this "vector" gives us a **column vector**:

$$\frac{\partial \mathbf{s}}{\partial \mathbf{M}} = \begin{bmatrix} \frac{\partial \mathbf{s}}{\partial \mathbf{M}_1} \\ \frac{\partial \mathbf{s}}{\partial \mathbf{M}_2} \\ \vdots \\ \frac{\partial \mathbf{s}}{\partial \mathbf{M}_k} \end{bmatrix} \quad (6.108)$$

This time, each derivative is a **row vector**. Let's **expand**:

$$\frac{\partial \mathbf{s}}{\partial \mathbf{M}} = \left[\begin{bmatrix} \frac{\partial \mathbf{s}}{\partial m_{11}} & \frac{\partial \mathbf{s}}{\partial m_{12}} & \cdots & \frac{\partial \mathbf{s}}{\partial m_{1r}} \end{bmatrix} \right. \right. \\ \left. \left. \begin{bmatrix} \frac{\partial \mathbf{s}}{\partial m_{21}} & \frac{\partial \mathbf{s}}{\partial m_{22}} & \cdots & \frac{\partial \mathbf{s}}{\partial m_{2r}} \end{bmatrix} \right. \right. \\ \vdots \\ \left. \left. \begin{bmatrix} \frac{\partial \mathbf{s}}{\partial m_{k1}} & \frac{\partial \mathbf{s}}{\partial m_{k2}} & \cdots & \frac{\partial \mathbf{s}}{\partial m_{kr}} \end{bmatrix} \right] \right] \quad (6.109)$$

Definition 279

If \mathbf{M} is a matrix in the shape $(r \times k)$ and \mathbf{s} is a scalar,

Then we define the **matrix derivative** $\partial \mathbf{s}/\partial \mathbf{M}$ as the $(r \times k)$ matrix:

$$\frac{\partial \mathbf{s}}{\partial \mathbf{M}} = \left\{ \begin{array}{c} \text{Column } j \text{ matches } m_{?j} \\ \left[\begin{array}{cccc} \frac{\partial \mathbf{s}}{\partial m_{11}} & \frac{\partial \mathbf{s}}{\partial m_{12}} & \cdots & \frac{\partial \mathbf{s}}{\partial m_{1r}} \\ \frac{\partial \mathbf{s}}{\partial m_{21}} & \frac{\partial \mathbf{s}}{\partial m_{22}} & \cdots & \frac{\partial \mathbf{s}}{\partial m_{2r}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial \mathbf{s}}{\partial m_{k1}} & \frac{\partial \mathbf{s}}{\partial m_{k2}} & \cdots & \frac{\partial \mathbf{s}}{\partial m_{kr}} \end{array} \right] \end{array} \right\} \text{Row } i \text{ matches } m_{i?}$$

This matrix has the same shape as \mathbf{M} .

X.16 Other Derivatives

After these, you might ask yourself, what about other derivative combinations?

$$\frac{\partial \textcolor{blue}{v}}{\partial \textcolor{blue}{M}}? \quad \frac{\partial \textcolor{blue}{M}}{\partial \textcolor{orange}{v}}? \quad \frac{\partial \textcolor{blue}{M}}{\partial \textcolor{blue}{M}^2}? \quad (6.110)$$

There's a problem with all of these: the total number of axes is **too large**.

What do we mean by an **axis**?

Definition 280

An **axis** is one of the **indices** we can adjust to get a different scalar in our array: each index is a "direction" we can move along our object to **store** numbers.

- A **scalar** has **0 axes**: we only have one scalar, so we have no indices to adjust.
-
- A **vector** has **1 axis**: we can get different scalars by moving **vertically** (for column vectors): $v_1, v_2, v_3\dots$

$$\left[\begin{array}{c} v_1 \\ v_2 \\ \vdots \\ v_m \end{array} \right] \quad \left. \right\} \text{Axis 1}$$

- A **matrix** has **2 axes**: we can move **horizontally** or **vertically**.

$$\overbrace{\left[\begin{array}{cccc} m_{11} & m_{12} & \cdots & m_{1r} \\ m_{21} & m_{22} & \cdots & m_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ m_{k1} & m_{k2} & \cdots & m_{kr} \end{array} \right]}^{\text{Axis 2: Columns}} \quad \left. \right\} \text{Axis 1: Rows}$$

These can also be called **dimensions**.

Why does the number of **axes** matter? Remember that, so far, for our derivatives, each axis of the output represented an axis of the **input** or **output**.

Note that last bit: we're saying a vector has one dimension. Can't a vector have **multiple** dimensions? Jump to [X.17](#) for a clarification.

$$\frac{\partial \mathbf{w}}{\partial \mathbf{v}} = \left[\begin{array}{cccc} \frac{\partial w_1}{\partial v_1} & \frac{\partial w_2}{\partial v_1} & \cdots & \frac{\partial w_n}{\partial v_1} \\ \frac{\partial w_1}{\partial v_2} & \frac{\partial w_2}{\partial v_2} & \cdots & \frac{\partial w_n}{\partial v_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial w_1}{\partial v_m} & \frac{\partial w_2}{\partial v_m} & \cdots & \frac{\partial w_n}{\partial v_m} \end{array} \right]$$

Column j: vertical axis of \mathbf{w}

Row i: vertical axis of \mathbf{v}

The way we currently build derivatives, we try to get **every pair** of input-output variables: we use **one** axis for each **axis** of either the **input** or **output**.

Take some examples:

- $\partial \mathbf{s}/\partial \mathbf{v}$: we need one axis to represent each term v_i .
 - 0 axis + 1 axis \rightarrow 1 axis: the output is a (column) **vector**.
- $\partial \mathbf{v}/\partial \mathbf{s}$: we need one axis to represent each term w_j .
 - 1 axis + 0 axis \rightarrow 1 axis: the output is a (row) **vector**.
- $\partial \mathbf{w}/\partial \mathbf{v}$: we need one axis to represent each term v_i , and another to represent each term w_j .
 - 1 axis + 1 axis \rightarrow 2 axes: the output is a **matrix**.
- $\partial \mathbf{M}/\partial \mathbf{s}$: we need one axis to represent the rows of M , and another to represent the columns of M .
 - 2 axis + 0 axis \rightarrow 2 axes: the output is a **matrix**.
- $\partial \mathbf{s}/\partial \mathbf{M}$: we need one axis to represent the rows of M , and another to represent the columns of M .
 - 0 axis + 2 axis \rightarrow 2 axes: the output is a **matrix**.

Notice the pattern!

Concept 281

A **matrix derivative** needs to be able to account for each type/**index** of variable in the input **and** the output.

So, if the **input** x has m axes, and the **output** y has n axes, then the derivative needs to have the same **total** number:

$$\text{Axes}\left(\frac{\partial \textcolor{violet}{y}}{\partial \textcolor{orange}{x}}\right) = \text{Axes}(\textcolor{violet}{y}) + \text{Axes}(\textcolor{orange}{x}) \quad (6.111)$$

This is where our problem comes in: if we have a vector and a matrix, we need **3 axes!** That's more than a matrix.

X.17 Dimensions (Optional)

Here's a quick aside to clear up possible confusion from the last section: our definition of axes and "dimensions".

We said a vector has 1 axis, or "dimension" of movement. But, can't a vector have **multiple** dimensions?

Clarification 282

We have two competing definition of **dimension**: this explains why we can say seemingly conflicting things about derivatives.

So far, by "**dimension**", we mean, "a separate **value** we can **adjust**".

- Under this definition, a $(k \times 1)$ column **vector** has **k** dimensions: it contains **k** different scalars we can **adjust**.

$$\left[\begin{array}{c} v_1 \\ v_2 \\ \vdots \\ v_k \end{array} \right] \quad \text{We can adjust each of our } k \text{ scalars.}$$

- You might say a $(k \times r)$ **matrix** has **k** dimensions, too: based on the **dimensionality** of its column vectors.
 - Since we prioritize the size of the vectors, we could say this is a very "vector-centric" definition.

In this section, by "dimension", we mean, "an **index** we can **adjust** (move along) to find another scalar.

- Under this definition, a $(k \times 1)$ column **vector** has **1** dimension: we only have **1** axis of **movement**.
- You might say a $(k \times r)$ **matrix** has **2** dimensions: a **horizontal** one, and a **vertical** one.
 - This **definition** is the kind we use in the following sections.

If you jumped here from X.16, feel free to follow this [link](#) back. Otherwise, continue on.

X.18 Dealing with Tensors

If a vector looks like a "line" of numbers, and a matrix looks like a "rectangle" of numbers, then a **3-axis** version would look like a "box" of numbers. How do we make sense of this?

First, what is this kind of object we've been working with? Vectors, matrices, etc. This collection of numbers, organized neatly, is an **array**.

Definition 283

An **array** of objects is an **ordered sequence** of them, stored together.

The most typical example is a **vector**: an ordered sequence of **scalars**.

A **matrix** can be thought of as a **vector** of **vectors**. For example: it could be a row vector, where every column is a column vector.

So, we think of a matrix as a "two-dimensional array".

We can extend this to any number of dimensions. We call this kind of generalization a **tensor**.

Definition 284

In **machine learning**, we think of a **tensor** as a "**multidimensional array**" of numbers.

Each "dimension" is what we have been calling an "**axis**".

A tensor with c axes is called a **c -Tensor**.

Note that what we call a tensor is **not** a mathematical (or physics) tensor: we do not often use the "tensor product", or other tensor properties.

Our tensor can be better thought of as a "**generalized matrix**".

Example: The 3-D box we are talking about above is called a 3-Tensor. We can simply think of it as a stack of matrices.

How do we handle **tensors**? Simply, we convert them into regular **matrices** in some way, and then do our usual math on them:

- If a tensor has a pattern of zeroes, we might be able to flatten it into a matrix.
 - For example, if we wanted to flatten a matrix into a vector (which we sometimes do!), we could do

These examples aren't especially important, but you'll see different variations in different softwares!

$$\begin{bmatrix} 3 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 3 \\ 9 \\ 4 \end{bmatrix} \quad (6.112)$$

- We can also flatten it into a matrix or vector by placing the layers next to each other.
- We cleverly do regular matrix multiplication in a way that's compatible with our tensors.
 - Note that tensors do not have a matrix multiplication-like multiplication by default: several have been designed, however.

- We ignore the structure of the tensor, and just look at the individual elements: we take the scalar chain rule for each of them, without respecting the overall tensor.

Clarification 285

If you look into **derivatives** that would result in a **3-tensor** or higher, you'll find that there's no consistent **notation** for what these derivatives look like.

These techniques are part of why: there are **different** approaches for how to approach these objects.

As we will see in the next chapter, tensors are **very** important to machine learning.

However, because they don't have a natural matrix multiplication, we'll try to convert it into a matrix in most cases.

X.19 The loss derivative

Finally, we apply this to our common derivatives in section 7.5.

$$\underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{A}^L}}_{(n^L \times 1)} \quad (6.113)$$

Loss is not given, so we can't compute it. But, we can get the shape: we have a scalar/vector derivative, so the shape matches \mathbf{A}^L .

Notation 286

Our derivative

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}^L} \quad (6.114)$$

Is a scalar/vector derivative, and thus the shape $(n^L \times 1)$.

X.20 The weight derivative

$$\underbrace{\frac{\partial \mathcal{Z}^L}{\partial \mathbf{W}^L}}_{(m^L \times 1)?} \quad (6.115)$$

This derivative is difficult - it's a derivative in the form vector/matrix. With **three** axes, we might imagine representing as a 3-tensor.

In fact, this can be manipulated into multiple different interesting **shapes** based on your **interpretation**: as we mentioned, there's no consistent rule for these variables.

But, our goal is to use this for the **chain rule**: so, we need to make the shapes **match**. This is why we do that strange transposing for our complete derivative.

$$\frac{\partial \mathcal{L}}{\partial W^\ell} = \underbrace{\frac{\partial Z^\ell}{\partial W^\ell}}_{\text{Weight link}} \cdot \underbrace{\left(\frac{\partial \mathcal{L}}{\partial Z^\ell} \right)^T}_{\text{Other layers}} \quad (6.116)$$

Our problem is we have **too many axes**: the easiest way to resolve this to **break up** our matrix. So, for now, we focus on only **one neuron** at a time: it has a column vector W_i .

$$W = \begin{bmatrix} W_1 & W_2 & \cdots & W_n \end{bmatrix} \quad (6.117)$$

For simplicity, we're gonna ignore the ℓ notation: just be careful, because Z and A are from two different layers!

Notice that, this time, we broke it into **column vectors**, rather than row vectors: each neuron's **weights** are represented by a column vector.

We'll ignore everything except W_i .

$$W_i = \begin{bmatrix} w_{1i} \\ w_{2i} \\ \vdots \\ w_{mi} \end{bmatrix} \quad (6.118)$$

Finally, we get into our equation: notice that a **single** neuron has only **one** pre-activation z_i , so we don't need the whole vector.

$$z_i = W_i^T A \quad (6.119)$$

Wait: there's something to notice, right off the bat. z_i is **only** a function of W_i : that means the derivative for every other term $\partial/\partial W_k$ is **zero**!

For example, changing W_2 would have **no effect** on z_1 .

Concept 287

The i^{th} neuron's **weights**, W_i , have **no effect** on a different neuron's **pre-activation** z_j .

So, if the **neurons** don't match, then our derivative is zero:

- i is the neuron for pre-activation z_i
- j is the j^{th} **weight** in a neuron.
- k is the neuron for weight vector W_k

$$\frac{\partial z_i}{\partial W_{jk}} = 0 \quad \text{if } i \neq k$$

So, our only nonzero derivatives are

$$\frac{\partial z_i}{\partial W_{ji}}$$

With that done, let's substitute in our values:

$$z_i = \begin{bmatrix} w_{1i} & w_{2i} & \cdots & w_{mi} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \quad (6.120)$$

And we'll do our **matrix multiplication**:

$$z_i = \sum_{j=1}^n w_{ji} a_j \quad (6.121)$$

Finally, we can get our derivatives:

$$\frac{\partial z_i}{\partial w_{ji}} = a_j \quad (6.122)$$

So, if we combine that into a vector, we get:

$$\frac{\partial z_i}{\partial W_i} = \begin{bmatrix} \frac{\partial z_i}{\partial W_{1i}} \\ \frac{\partial z_i}{\partial W_{2i}} \\ \vdots \\ \frac{\partial z_i}{\partial W_{mi}} \end{bmatrix} \quad (6.123)$$

We can use our equation:

$$\frac{\partial z_i}{\partial W_i} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} = A \quad (6.124)$$

We get a result!

~~~~~

What if the pre-activation  $z_i$  and weights  $W_k$  don't match? We've already seen: the derivative is 0: weights don't affect different neurons.

$$\frac{\partial z_i}{\partial W_{jk}} = 0 \quad \text{if } i \neq k \quad (6.125)$$

We can combine these into a **zero vector**:

$$\frac{\partial z_i}{\partial W_k} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \vec{0} \quad \text{if } i \neq k \quad (6.126)$$

So, now, we can describe all of our vector components:

$$\frac{\partial z_i}{\partial W_k} = \begin{cases} A & \text{if } i = k \\ \vec{0} & \text{if } i \neq k \end{cases} \quad (6.127)$$

These are all the elements of our matrix  $\partial z_i / \partial W_k$ : so, we can get our result.

$$\frac{\partial \mathbf{Z}}{\partial \mathbf{W}} = \begin{bmatrix} \mathbf{A} & \vec{0} & \cdots & \vec{0} \\ \vec{0} & \mathbf{A} & \cdots & \vec{0} \\ \vdots & \vdots & \ddots & \vec{0} \\ \vec{0} & \vec{0} & \vec{0} & \mathbf{A} \end{bmatrix} \quad (6.128)$$

We have our result: it turns out, despite being stored in a **matrix**-like format, this is actually a **3-tensor**! Each entry of our **matrix** is a **vector**: 3 axes.

~~~~~  
But, we don't really... *want* a tensor. It doesn't have the right shape, and we can't do matrix multiplication.

We'll solve this by **simplifying**, without losing key information.

Concept 288

For many of our "tensors" resulting from matrix derivatives, they contain **empty** rows or **redundant** information.

Based on this, we can **simplify** our tensor into a fewer-dimensional (fewer axes) object.

We can see two types of **redundancy** above:

- Every element **off** the diagonal is 0.
- Every element **on** the diagonal is the same.

Let's fix the first one: we'll go from a diagonal matrix to a column vector.

$$\begin{bmatrix} \mathbf{A} & \vec{0} & \cdots & \vec{0} \\ \vec{0} & \mathbf{A} & \cdots & \vec{0} \\ \vdots & \vdots & \ddots & \vec{0} \\ \vec{0} & \vec{0} & \vec{0} & \mathbf{A} \end{bmatrix} \longrightarrow \begin{bmatrix} \mathbf{A} \\ \mathbf{A} \\ \vdots \\ \mathbf{A} \end{bmatrix} \quad (6.129)$$

Then, we'll combine all of our redundant \mathbf{A} values.

$$\begin{bmatrix} \mathbf{A} \\ \mathbf{A} \\ \vdots \\ \mathbf{A} \end{bmatrix} \longrightarrow \mathbf{A} \quad (6.130)$$

We have our big result!

Notation 289

Our derivative

$$\underbrace{\frac{\partial \mathbf{Z}^\ell}{\partial \mathbf{W}^\ell}}_{(m^\ell \times 1)} = \mathbf{A}^{\ell-1}$$

Is a vector/matrix derivative, and thus should be a 3-tensor.

But, we have turned it into the shape $(m^\ell \times 1)$.

This is as **condensed** as we can get our information: if we compress to a scalar, we lose some of our elements.

Even with this derivative, we still have to do some clever **reshaping** to get the result we need (transposing, changing derivative order, etc.)

However, at the end, we get the right shape for our chain rule!



X.21 Linking Layers

$$\frac{\partial \mathbf{Z}^\ell}{\partial \mathbf{A}^{\ell-1}} \quad (6.131)$$

This derivative is much more manageable: it's just the derivative between a vector and a vector. Let's look at our equation again:

Ignoring superscripts ℓ , as before.

$$\mathbf{Z} = \mathbf{W}^T \mathbf{A} \quad (6.132)$$

We'll use the same approach we did last time: \mathbf{W} is a vector, and we'll focus on \mathbf{W}_i . This will allow us to break it up **element-wise**, and get all of our **derivatives**.

We could treat \mathbf{W} as a whole matrix, but this will give us our results without as much clutter: the only **difference** is that we would have to depict every \mathbf{W}_i at **once**.

$$\mathbf{W} = [W_1 \ W_2 \ \dots \ W_n] \quad \mathbf{W}_i = \begin{bmatrix} w_{1i} \\ w_{2i} \\ \vdots \\ w_{mi} \end{bmatrix} \quad (6.133)$$

Here's our equation:

$$z_i = \begin{bmatrix} w_{1i} & w_{2i} & \cdots & w_{mi} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \quad (6.134)$$

We matrix multiply:

$$z_i = \sum_{j=1}^n W_{ji} a_j \quad (6.135)$$

The derivative can be gotten from here -

$$\frac{\partial z_i}{\partial a_j} = W_{ji} \quad (6.136)$$

We look at our whole matrix derivative:

$$\frac{\partial Z}{\partial A} = \left\{ \begin{array}{c} \text{Column } j \text{ matches } z_i \\ \left[\begin{array}{ccc} \ddots & \vdots & \ddots \\ \cdots & \frac{\partial z_i}{\partial a_j} & \cdots \\ \ddots & \vdots & \ddots \end{array} \right] \end{array} \right\} \text{Row } i \text{ matches } a_j \quad (6.137)$$

This notation looks a bit weird, but it's just a way to represent that all of our elements follow this pattern.

Wait.

- The derivative $\partial z_i / \partial a_j$ is in the j^{th} row, i^{th} column.
- W_{ji} represents the element in the j^{th} row, i^{th} column.

They're the same matrix!

We get our final result:

If two matrices have exactly the same shape and elements, they're the same matrix.

Notation 290

Our derivative

$$\overbrace{\frac{\partial Z^\ell}{\partial A^{\ell-1}}}^{(m^\ell \times n^\ell)} = W^\ell$$

Is a vector/vector derivative, and thus a matrix.

But, we have turned it into the shape $(m^\ell \times n^\ell)$.

X.22 Activation Function

$$\frac{\partial \mathbf{A}^l}{\partial \mathbf{Z}^l} \quad (6.138)$$

The last derivative is less unusual than it looks.

$$\mathbf{A}^l = f(\mathbf{Z}^l) \longrightarrow \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = f\left(\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} \right) \quad (6.139)$$

We can apply our function element-wise:

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} f(z_1) \\ f(z_2) \\ \vdots \\ f(z_n) \end{bmatrix} \quad (6.140)$$

As we can see, each activation is a function of only **one** pre-activation.

Concept 291

Each **activation** is only affected by the **pre-activation** in the **same neuron**.

So, if the **neurons** don't match, then our derivative is zero:

- i is the neuron for pre-activation z_i
- j is the neuron for activation a_j

$$\frac{\partial a_j}{\partial z_i} = 0 \quad \text{if } i \neq j$$

So, our only nonzero derivatives are

$$\frac{\partial a_j}{\partial z_i}$$

As for our remaining term, we'll describe any row of the above vectors:

$$a_i = f(z_i) \quad (6.141)$$

Our derivative is:

$$\frac{\partial a_i}{\partial z_i} = f'(z_i) \quad (6.142)$$

In general, including the non-diagonals:

$$\frac{\partial \mathbf{a}_i}{\partial z_i} = \begin{cases} f'(z_i) & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \quad (6.143)$$

This gives us our result:

Notation 292

Our derivative

$$\underbrace{\frac{\partial \mathbf{A}^l}{\partial z^l}}_{(n^l \times n^l)} = \left[\begin{array}{ccccc} f'(z_1^l) & 0 & 0 & \cdots & 0 \\ 0 & f'(z_2^l) & 0 & \cdots & 0 \\ 0 & 0 & f'(z_3^l) & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & f'(z_n^l) \end{array} \right] \right\} \begin{array}{l} \text{Column } j \text{ matches } a_j \\ \text{Row } i \text{ matches } z_i \end{array} \quad (6.144)$$

Is a vector/vector derivative, and thus a matrix.

But, we have turned it into the shape $(n^l \times n^l)$.

X.23 Element-wise multiplication

Notice that, in the previous section, we would've compressed this matrix down to remove the unnecessary 0's:

$$\begin{bmatrix} f'(z_1^l) \\ f'(z_2^l) \\ \vdots \\ f'(z_n^l) \end{bmatrix} \quad (6.145)$$

This is a valid way to interpret this matrix! The only thing we need to be careful of: if we were to use this in a chain rule, we couldn't do normal matrix multiplication.

However, because of how this matrix works, you can just do **element-wise** multiplication instead!

You can check it for yourself: each index is separately scaled.

Concept 293

When multiplying two vectors R and Q, if they take the form

$$R = \begin{bmatrix} r_1 & 0 & 0 & \cdots & 0 \\ 0 & r_2 & 0 & \cdots & 0 \\ 0 & 0 & r_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & r_n \end{bmatrix} \quad Q = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ \vdots \\ q_n \end{bmatrix}$$

Then we can write their product each of these ways:

$$RQ = \underbrace{R * Q}_{\text{Element-wise multiplication}} = \begin{bmatrix} r_1 q_1 \\ r_2 q_2 \\ r_3 q_3 \\ \vdots \\ r_n q_n \end{bmatrix} \quad (6.146)$$

So, we can substitute the chain rule this way.

Terms

- Matrix/scalar derivative
- Scalar/Matrix derivative
- Axis
- Dimension (vector)
- Dimension (array)
- Array
- "Tensor" (Generalized matrix)
- c-Tensor (2-tensor, 3-tensor, etc.)
- Gaussian/Normal Distribution (Optional)

CHAPTER 7

Convolutional Neural Networks - Supplementary Notes

Unofficial notes by Shauntclair Ruiz

7.1 Introduction

7.1.1 Fully Connected Networks

Up to this point, we've focus on "fully connected" neural networks.

They're called "fully connected" because between two layer, every single unit in one layer can affect every single unit in the next layer, via some weight.

Definition 294

A **fully connected** layer is ones where every number in the **input** is connected (by a weight) to every element in the **output**.

For example, unit a in the first layer is multiplied by $w_{a,b}$ to affect unit b in the second layer, for every single a^{th} input and b^{th} output.

In this case, we have as many connections as possible, so we use this when we don't know enough about our problem: we let the machine decide how to handle it.

But, this isn't always the most effective way to structure our algorithm. To show why this is, we'll introduce a particular type of problem: image processing.

7.1.2 A different type of problem

For example, we might think of face recognition, or the vision used by self-driving cars.

We'll simplify for now by assuming we're working with black and white images. Let's start by thinking about how we'd put our image into our fully connected network.

Our image has the dimensions ($r \times k$): r pixels wide, k pixels tall.

So, we list our pixels out in a single line of rk inputs: this process of turning a 2d input into a 1d input is called "flattening".

Definition 295

Flattening: Taking an entire **matrix** of inputs, and turning them into a single **vector**, by listing all the numbers in a single line.

7.1.3 What's wrong?

Here, we present two issues:

- First: our computer has no concept of pixels being near or far from each other, except in the same row.

If two pixels were on top of each other, the computer loses that information when you line up the pixels. Are they diagonal? How close are they?

Arguably, the computer could figure out the shape eventually. But, that's time wasted, and could create a less robust solution.

We want our computer to know how the 2d **space** in our image works, so it's "spatial".

Specifically, we want to know what pixels are nearby- which ones are **local**? We're talking about "locality".

The combination of these two is

Definition 296

Spatial locality asks: Which pixels are **near** each other in **space**, and how?

How big a block of pixels do we need to know in order to find, say, a cat?

These kinds of questions fall under "spatial locality".

- Second: imagine if your computer is looking for something: maybe a cat, for example. If you move the cat within the image, it should still recognize it, right?

The problem is, in a fully connected network, it doesn't!

If it finds the cat in the top-left of one image, that includes different inputs than if it finds the cat in the bottom right of the image.

We want our computer to know that these two are the same: meaning nothing *changes*. Something that doesn't change is "invariant".

The shift we just described (sliding the cat around the image) is called a "translation".

So, the combination here is

Definition 297

Translation Invariance: The pattern you're looking for is the **same**, no matter **where** in the image you look for it.

We'll come up with a strategy to handle both of these problems.

7.2 Filters - Introduced

7.2.1 The pieces of our solution

Let's figure out how to solve those two problems at once - we want to do a calculation that doesn't depend on location, for example.

So, we'll make a smaller calculation at multiple different positions.

And we want this calculation that gives an idea of "locality": which pixels are near each other.

The solution is to do a calculation over a small region of an image (local) and repeat it all over!

That might not be entirely clear right now, but we'll get into the details.

7.2.2 Looking for patterns

We call this calculation a filter. Here's how it works:

Let's say we're looking for some pattern; we could say we're "filtering for" this pattern.

So, we'll take each section of the image, and look for this matching patterns.

We need an idea of how similar these patterns are. We're not sure how to do that... let's try to do a 1-D example, for inspiration.

This isn't an entirely useless example, either - when you want to do sound processing, you'll have 1D data, just like this!

7.2.3 The 1D case

We have some string of numbers that's length r , and a pattern we're looking for, length n . So, we'll look at every length- n segment of our input.

Which means, we'll be comparing two length- n lists of numbers, i.e. vectors.

This might be registering something familiar, or maybe it isn't - but, it turns out, you can measure how similar two vectors are using the dot product!

The dot-product, as we've seen in cases like linear classification ($\theta^T x$, for example), has two interesting ideas we care about.

7.2.4 Dot Products

To make explaining easier, let's assume i represents the dimension we're focused on, and v and w are our two vectors.

- The dot product can be calculated by multiplying together the each dimension of each vector ($v_i w_i, v_j w_j$, so on) and adding all of those together

- This operation can sort of measure how "similar" these vectors are

The "similarity" idea makes the most sense if you imagine we have a binary input with 1's and -1's:

- if you multiply 1 by 1, you get 1, and the two vectors are the same in that dimensions.
- if you multiply -1 by 1, you get -1: the two vectors are opposite (most dissimilar!) in that dimension.
- if you multiply -1 by -1, you get 1, the two dimensions are more similar again.

So, if our dot product is more positive, then our vectors are more similar. The same general logic works (sort of) for non-binary outputs.

7.2.5 Finally, 1D filtering

Back to the example we're working on: one of these two vectors is the pattern we're looking for.

So, this dot product is a good measure of how similar this piece is to that pattern! That's exactly what we're looking for - a pattern detector.

So, for every substring inside this input, we'll calculate this measure of similarity - the resulting new vector is our output.

If all of the numbers are too positive or negative, maybe we'll add a bias, where we either subtract from or add to our dot product.

This procedure is called **convolution**.

Definition 298

Convolution: the process of looking for a **pattern** in an image (or any signal) by **sliding** it over the input, and repeatedly doing a **dot product**-like calculation.

7.2.6 A comment about our output

Notice that, in this case, our output smaller than our input: because our filter can only go as far right as our vector allows, our length has shrunk by $n - 1$.

Why that number? Well, if we have a length-1 filter, we're all good. If we have a length-2 filter, our filter can start on every input, except for the very last one, so it has shrunk by 1.

So on and so forth: every element after the first of our filter "blocks" the start of our filter from using that element.

How do we deal with that? Well...

7.2.7 Padding

One very common solution is to do something called "padding".

The idea is: our output shrank because our filter is limited by how long it is - it can't shift past the matrix; it has nothing to multiply with.

So, what if we... let it stick out the side?

How do we do this? Well, the reason we can't shift the filter outside the matrix is because we are missing some values.

So... we'll just add some extra values to the edge of the vector.

We can pad with whatever values we want, but most often, you pad with 0's. You can also adjust how many 0's you pad with, depending on how big you want your output to be.

Definition 299

Padding is when you add filler values (usually 0's) to the **edge** of your input so your filter can scan over the whole input, without being stopped by the edge.

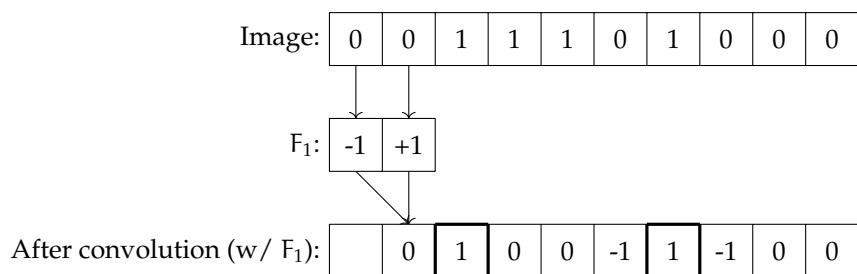
7.2.8 A specific example - a length-2 filter

Here, we introduce a concrete example. We have a filter of size two, and take a 1-dimensional, binary "image" to start off with.

First, our first filter is $F_1 = (-1, +1)$.

This filter looks for a very simple pattern: a smaller element on the left, and a larger element on the right. You could think of this as the numbers "increasing" from left to right.

Then given the first image below, we can "convolve" it with filter F_1 , like so.



As shown above, we do this operation for every pair of numbers in the image, to get the post-convolution result.

Notice that this calculation detects exactly what we're looking for - any time we have an increasing sequence of two numbers, we get a 1 in the output!

7.2.9 Another example - a length-3 filter

Let's try another example: to show that we can apply convolutions one-after-another, we'll use the previous output as our new input - our new image.

This filter will be length 3: we'll use $F_1 = (-1, +1, -1)$

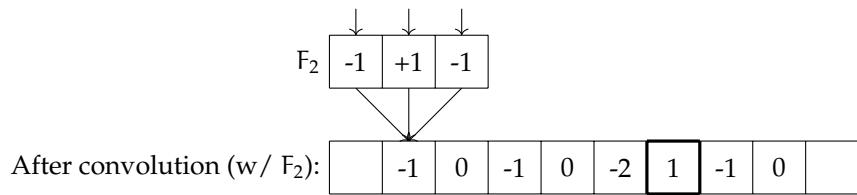
This pattern looks like a point greater than those around it - you might think of it as a local maximum.

We'll pad one zero on the left (not shown below) to make our vector shrink by only 1, instead of 2.

So, we'll see how this filter does.

After convolution (w/ F_1):

	0	1	0	0	-1	1	-1	0	0
--	---	---	---	---	----	---	----	---	---



And again, it works! It only gives a positive result on the sequence $(-1, +1, -1)$, where we have a local maximum.

It also gives an especially negative result at a local minimum: $(0, -1, +1)$.

7.3 Filters - In higher dimensions

So, how do we extrapolate this to higher dimensions?

7.3.1 Multiplication in 2D

Surprisingly, the answer is pretty simple: we just do the same operation as before, but without the perspective of vectors.

Before, we did the dot product for each sub-segment. This means multiplying the components "element-wise" (multiply i^{th} dimension of each vector together), and adding together those results.

Now, we do the exact same: our filter is 2D, so we take every continuous 2D frame of our input, and multiply, element-by-element.

The same logic as before applies: if two elements are similar, we get a larger result. If they are more different, we'll get a smaller result.

7.3.2 Dot products, but... bigger?

Here's one way to think about this that might be helpful, might be confusing.

Break your matrix up into rows. For each of the rows, you're multiplying element-wise - like a dot product.

So, you're taking a dot product for each row, which gives the similarity for those rows.

Then you add up all these dot products ("similarities") to measure the total similarity between two matrices.

In a way, because we're still multiplying and adding, you might intuitively think about this as a "bigger" dot product.

7.3.3 Comments - Evolution

As a fun fact, these kinds of 2D filters appear in the visual part of mammal brains - including yours!

This seems to show that this way of thinking about visual information is similar to how evolution settled on it - and so it probably does a pretty good job.

7.4 Filter Banks Pt. 1

7.4.1 More Filters!

One filter can encode lots of information, but different filters have different data: one might look for edges, maybe another can find circles, maybe a more complicated one can find a face!

What if we want to have different kinds of information, at the same time? What if we want to be able to combine these pieces of information?

You might think "well, I could use multiple filters", but, how do we do that? If we have multiple filters one after another, the first one may have removed information the later ones need.

So, instead, what if we use multiple filters at the same time ("in parallel") - we run each filter over the original image, and save each result?

Each of these calculations creates a new matrix output, which we store separately. We end up with a bunch of matrices as our output. Each of these matrices is called a **channel**.

Definition 300

Channel: the result of using(convolving) **one filter** on an **image**. If multiple filters are used, we can end up with many channels.

7.5 How to store matrices - Tensors

7.5.1 Our goal

Now that we have all this data to store, we need to be careful how we work with it - this is a lot to keep track of, after all.

In order to do that, we'll introduce what's called a "tensor".

7.5.2 Building Up

Consider the following review of what we know so far.

- If we only have one number, we store that as a **scalar**.

This scalar can be thought of like a single point - since there's no axis along which we can move, sometimes this is called "0-dimensional".

- If we have a sequence of numbers, we store that as a **vector**.

You can do this by stacking scalars on top of each other in a line.

You can also think of this as a 1D object, where the "dimension" is the index we can move along to get different scalars.

In a way, we have a "line" of numbers, at least visually.

- If you have several sequences of numbers, you can store that as a **matrix**.

You can do this by stacking vectors next to each other. If you break your matrix up into rows or columns, you get a bunch of vectors!

This is a 2D object: you can move along the rows, or the columns: each of these is a dimension.

We have a grid of numbers, or maybe you can imagine it as a square or "plane".

- What if we want to stack several matrices on each other?

This would create a sort of 3D object, where we have rows, columns, and height.

Visually, you might imagine the 1d case as a line of numbers, the 2d case as a square of numbers.

Now, we have what you could think of as a "cube" of numbers.

This cube of numbers, this 3D object, is what we call a **tensor**.

Definition 301

A **tensor** is the more "generalized" version of a matrix: it's just a "**box**" of numbers in some **dimension**. In our case, our **tensor** is **cube** (or cuboid) of numbers.

As you see in the definition above, a tensor is technically any version of this kind of object, in any dimension. A scalar, vector, and matrix are all tensors.

But, for our purposes, we only care about what is called a **3-tensor**, or a 3D box of numbers. The one that looks like a **cube**.

7.5.3 An important warning: dimensions

Before we continue, we should make a quick warning about dimensions.

You might have noticed that the way we are using the word "dimension" here is different from how we have used it in other parts of the course.

For example, if you have a vector that has 3 elements, you could call that a "3D vector", because you can imagine it as an arrow in 3D space.

But above we said vectors were "1D" objects. What gives?

This is because, unfortunately, there are different ways you can talk about "dimensions".

All have some different mathematical reason behind them, where in some way, they are related, **but these definitions are different and you should be careful**.

Let's get into that.

7.5.4 Dimensions, Clarified

We'll break this down. Dimensions are about asking the question, "how many separate ways can I change what I'm looking at?"

For example, we live in 3D space because we can vary our position in the x , y , and z directions.

A paper is 2D because there are only two separate ways to move on the paper.

Based on this, we can think of two definitions for dimensions:

- Version 1: If we drew our vector in space, how many ways can we vary the direction it points in?

Or, in other words, how many *dimensions* of space are we thinking of when we draw our vector?

This definition focused on vectors, and each number in that vector as a unique dimension.

In this perspective, we imagine that we can vary each scalar in our vector to create a different vector, and we have n scalars that we can vary.

To summarize:

- A scalar is a 1D vector.

- A vector with 2 numbers is 2D by this definition.
- A (4×4) matrix (4 vectors, each in 4D), is 4D.

Definition 302

One way to define **dimensions** is to ask how many different **numbers** (scalars) we have in a **vector**.

- Version 2: How many axes of numbers do we have to work with?

This definition is more abstract, and talks about the dimension of what we're storing our numbers in.

In this perspective, we imagine that we can vary the index on each axis (row, column, height, etc.) to find a different scalar, and we have k axes we can vary.

To summarize:

- A scalar is a 0D object.
- A vector with 2 numbers is 1D by this definition.
- A (4×4) matrix (4 vectors, each in 4D), is 2D.

Definition 303

Another way to define **dimensions** is to ask how many **axes** our **box** of numbers, or **tensor**, has.

Now that we've clarified the difference, back to tensors.

7.5.5 Back on track: What's a tensor?

Using what we've carefully built up, we have everything we need to understand the basics of tensors:

A tensor is, more or less, a bunch of matrices of the same shape stacked on top of each other.

You can visualize it as a **cube of numbers**.

Not too bad, right?

7.5.6 Tensor Math (Optional)

We went through all this work to carefully build understanding because tensors are not easy, but they are very, very important.

Unfortunately, just like how matrix multiplication, algebra, and calculus have their own special operations and rules, so too, do tensors.

However, simply put: tensor multiplication and calculus are really hard, so we're not going to do it. We'll let the computer handle it.

The rules involved are incredibly general and very, very powerful.

Physicists, computer scientists, mathematicians, and other smart people solving hard problems have to grapple with the details for years, sometimes, to understand everything they need.

We do not have that time. So, we'll leave those details to the computer.

7.5.7 What about higher dimensions? (Optional)

One little plot twist: what if we need something even bigger than a tensor?

What if we need to stack multiple tensors, for an even more complicated problem?

Well, thankfully, tensors actually include things higher than 3 dimensions!

If you have a "4D cube of numbers", that still follows the rules of tensors.

Tensor math can be applied for any higher dimensions. So, if we were to ever need it, our computer knows how to handle it.

7.6 Filter Banks Pt. 2

7.6.1 Review

Now that we know what a tensor is, let's pick back up where we left off:

We wanted to convolve multiple different filters with our image at the same time ("in parallel"). We call this collection of filters a "filter bank".

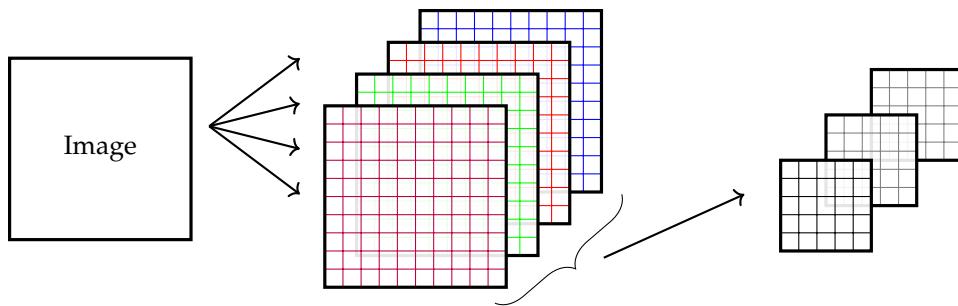
Definition 304

A **filter bank** is a collection of **filters** used on the **same image**.

(Side comment: notice you can store this filter bank as a tensor too! Since each filter is a matrix, and you can just stack them).

7.6.2 Using filter banks

The output after applying all these filters is a tensor: each filter outputs to one "channel", which is what we call the third dimension of our tensor (row, column, channel)



This diagram loosely shows the process: the first object on the left is our image.

The many arrows represent that the image is applied to each of the filters in multi-color filter bank. Finally, each of those outputs is combined into the tensor on the right.

If we apply k filters, we will end up with k channels: one for each filter.

So, if a single filter resulted in a $(r \times r)$ -sized matrix, then the filter bank results in a $(r \times r \times k)$ tensor.

7.6.3 An example: Using a Filter Bank

Let's show an example of 2d filtering, with a filter bank.

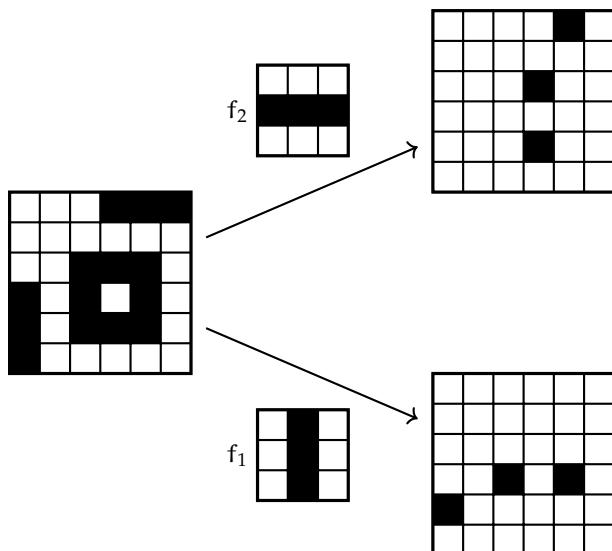
Our filter bank has two filters, f_1 and f_2 . So in this case, $k = 2$.

Each filter is (3×3) in size. So we preserve the size in our output, we'll add 1 layer of 0-padding all the way around our input matrix (not shown).

As mentioned before, each filter will be convolved with the input image, "looking" for the pattern they represent.

In this case, we see that f_1 seems to be looking for a vertical line, and f_2 is looking for a horizontal line; both 3 pixels long.

Our input image is $(n \times n)$, so our output of the first step will be an $(n \times n \times 2)$ tensor.



Notice that the top output puts three dots where it finds horizontal lines(you can match them up visually!)

The bottom output puts three dots where it finds vertical lines. So, it seems to be working!

7.7 Tensor Filters

7.7.1 What now?

Now we have used multiple filters, and we got a tensor... great!

But... what on earth do we do with a tensor?

7.7.2 Motivation

So, the whole reason we used multiple filters, was so we could look for multiple patterns at the same time.

Often, the goal of your trained machine is to recognize certain shapes.

It may be easier to look for smaller, simpler shapes first, and then combine those results to look for bigger shapes; and so on.

For example: imagine you're looking in an image for a child's depiction of a house.

You might look for the signature rectangle on the bottom, and a triangle "roof" on top. These are two distinct features you can find with different filters.

So, using our previous steps, we have a filter that finds squares, and one that finds triangles.

But what about combining them?

7.7.3 Filters, again!

The solution is to use another filter!

But this time, we need to go up a dimension again.

We went from 1D filters to 2D filters because we wanted to look for 2D patterns, and we did this by multiplying element-wise as we moved across the grid.

Now, we have a 3D tensor we're analyzing.

In this case, we want to find a point on one layer (representing the roof), closer to the top of the image than a point in the next layer (representing the "body" of the house)

So, if this is the pattern we're looking for, we could replicate this pattern in a new filter! We'll create a tensor that represents this pattern, and again, slide this pattern around the image.

For simplicity's sake, we'll assume this tensor has the same number of channels as the input, so we only have to slide it around the image, and not through the different channels (doesn't that sound complicated...)

Thus, we have invented the "tensor filter". All the same math from before (multiplying element-wise, sliding around the image, maybe padding...) is the same.

7.7.4 An example

We'll take the output from our earlier example and run a tensor filter over it.

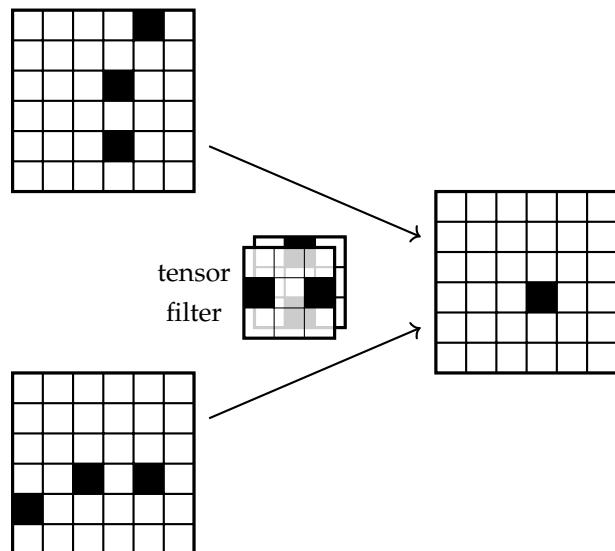
Let's say the output we're looking for a "circle", or really, a (3×3) square with a hole in the center.

This is gotten by having two vertical lines on the sides, and two horizontal lines on top and bottom of the section of the image.

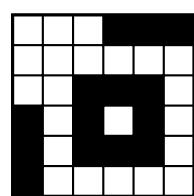
So, we'll create a filter so that's exactly what we're looking for!

One channel will look for the two horizontal lines, the other will look for the two vertical lines.

Remember that below, the top image gave us horizontal lines, and the bottom image gave us vertical lines.

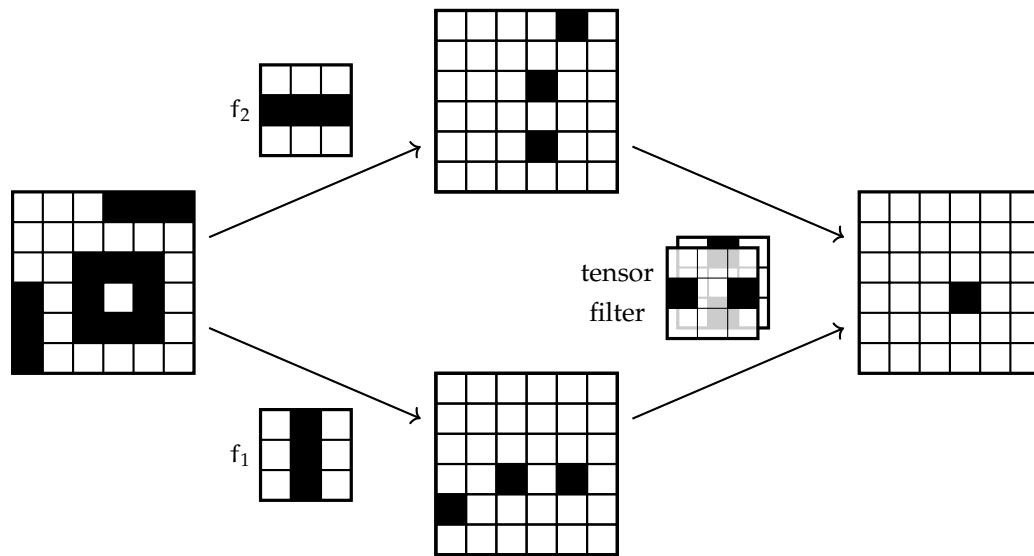


Like magic, we get exactly what we're looking for! Notice that it has a dot in exactly one place. If we go back to the original...



We have our dot right in the center of our empty square/"circle"! We can see that the process works.

Here's what the whole thing looks like when we put it together:



We can see the full process of running two separate filter banks, and getting the results we're looking for.

7.7.5 Really small side comment: RGB color (Optional)

If your image is RGB (made up of red, green, blue), you'll need three values at every pixel: this is most easily described with a tensor!

So, if you have an image that is $(n \times n)$ in size, the RGB representation will be a tensor of the shape $(n \times n \times 3)$.

7.8 Machine learning and Convolution

7.8.1 Applying to ML: Layering

Now, we have learned how to use convolution to process information.

We can see it's good for finding patterns, so it makes some sense we'd want to add that ability to our network.

So, let's try that: how do we use convolution in our neural network?

Well, because our network is modular (broken up into separate pieces, or "modules", that can be thought of separately), the process isn't too painful.

All we have to do is insert a new layer of convolution into our neural network.

It takes in inputs from whatever the last layer gave, and sends its output to the next layer after.

In fact, our example above shows two consecutive layers of convolution.

This can be used to build larger features (circles) from smaller ones (lines).

7.8.2 Thinking in ML terms

Each filter bank we use will represent a single layer.

If we want to think in the terms of ML, we can think of each number inside a filter as a "weight": a number used for calculation we can adjust.

As we mentioned early on, we might also have one bias for our filter - making the output of our element-wise multiplication more positive, more negative, etc.

7.8.3 Training Filters

In the past, machine vision experts would hand-craft their filters, and experiment with them.

However, because we can think of our filters in terms of their weights, we can use gradient descent to train them!

As long as we can take our derivatives, we can use gradient descent to find better filters. Our computers know how to do this.

Sometimes, these filters can even tell us about the structure of the data!

7.8.4 Benefits of Convolution

This convolution has a lot of benefits. Two of those were the original motivations for trying out convolution:

- **Spatial Locality:** A filter tells you about what the "local" area around a pixel looks like.

Thus, it can give the neural network the ability to associate that information.

- **Translation Invariance:** A filter also, by virtue of being slid around, will find the same pattern no matter where in the image it appears.

This means the neural network can now more easily tell that an object in the bottom right is often the same as an image in the top left.

But there's one more surprising benefit: efficiency.

7.8.5 Efficiency of Convolution: Fewer Weights

The interesting detail: a filter is (typically) much smaller than an image, and each pixel in the filter only has one weight.

So, rather than having many weights for every pixel in the input, we have a small number of weights for our filter.

On top of that, these weights are reused multiple times during convolution, as we slide around the image.

So, we have a much, much smaller number of weights to train.

Because we are "sharing" weights between different calculations, we call this benefit **weight sharing**.

Definition 305

Weight sharing is where the **same weights** are used for **multiple calculations**. Thus, the model trains **faster**.

For example:

if we have a (5×5) image, and want a (5×5) output, using a fully connected layer, we would need 25 inputs, 25 outputs.

This creates 25×25 weights, and 25×1 biases: 650 parameters.

If we have a (3×3) filter, and we use length-1 padding all the way around the image, we can get the same size output.

This has 3×3 weights, and 1 bias: 10 parameters.

And this efficiency grows the larger your input and output are.

This does mean we have to be careful when training, though!

7.8.6 Variable definitions

Let's carefully define this with some variables. Let's say we place our filter bank on layer l .

Note that we'll assume the image and filter are both square (same number of rows and columns).

This isn't necessary, but makes our definition simpler.

Additionally, remember that exponents in this notation only represent which layer you're referring to.

Our variables:

- **Length of the square input:** n^{l-1}

We use $l - 1$ instead of l because this is technically the **output** of the previous layer.

- **Number of filters:** m^l

- **Length of each square filter:** k^l

- **Padding around the input:** p^l

Remember that we usually pad with the number 0.

For each filter, we also have one additional bias term.

To use this bias, we do the element-wise multiplication, add up the terms, and *then* add the bias.

We have one bias per filter, or m^l total biases in a layer.

7.8.7 Shapes

Now, we need to find the shape of each object.

If we have only **one filter**, we just end up with some squares:

- Our filter: $(k^l \times k^l)$
- Our input: $(n^{l-1} \times n^{l-1})$

What if our previous layer had filters, though?

Well, if our previous layer had m^{l-1} filters, then it would give the output (our new input) a third dimension of m^{l-1} .

We mentioned before we only want to slide our filter across rows and columns, so the third dimension needs to match.

Thus, we need our filter to have as many channels as these previous output: our third dimension is m^{l-1} .

Thus, our shapes are

- Our filter: $(k^l \times k^l \times m^{l-1})$
- Our input: $(n^{l-1} \times n^{l-1} \times m^{l-1})$

7.9 Output dimensions

Let's round out some other important considerations related to size. Specifically, we'll look at things affecting the output dimensions.

7.9.1 Filter Size

Recall that, in the 1D case, we showed that our output (vector) dimension is decreased by $k - 1$.

This was because every element of the filter after the first prevented the filter from moving further to the right.

We can extrapolate the same concept to multidimensional inputs: each dimension is shrunk by $k - 1$.

So, an $(n \times n)$ input turns into a $((n - k + 1) \times (n - k + 1))$ output.

For the rest of the problem, we shall refer to n -squares to mean squares of length n , rather than writing out the full dimensions.

So in this case, the length is

$$n^l = n^{l-1} - k^l + 1$$

7.9.2 Padding

We have mentioned padding, but it's important to remember how this affects the input and output.

If we add padding symmetrically all the way around the input, then we add p 0's to the left of our image, and p 0's to the right.

So, the dimension increases by $2p$.

Thus, we can say our n -square has its length changed to

$$n^l = n^{l-1} + 2p^l$$

7.9.3 Stride

Here, we introduce a new concept: stride.

When we convolve normally, we move our filter over by 1 for each step.

But, we could also move over by r for each step. Basically, we "skip over" several pixels each time.

A stride of 1 means we move our pixel over by one each time, same as normal.

A stride of 2 means we move our pixel over by 2 each time: you "skip" every even pixel.

In 1D, you would convolve with pixel 1, 3, 5...

Definition 306

Stride: How many **pixels** you **slide** over with each **of convolution**.

As you can see, we skip one pixel each time, the size of our output image becomes half the size.

So we start by guessing we get a square of length n/s .

We have to round, though: if we have a stride of 2, and 5 pixels, that would imply a length 2.5 square.

But as we can see above, if we have 5 pixels, we have 3 pixels to convolve with.

Rounding up has its own notation: if you want to round up x , you write that as $\lceil x \rceil$

So, we get a square of length

$$n^l = \lceil n^{l-1}/s^l \rceil$$

7.9.4 The convolution output size

We combine all three of these factors to get a single equation.

- Filter size: $n - k + 1$
- Padding: $n + 2p$
- Stride: $\lceil n/s \rceil$

We get this result (which is important, and worth saving!):

$$n^l = \lceil (n^{l-1} - k^l + 1 + 2p^l)/s^l \rceil$$

7.10 Max Pooling

7.10.1 Gathering information

As we've alluded to, convolution can be used to get basic features, which we combine into larger features later.

Whether lines into circles, or shapes into childlike houses, we often want to combine these patterns.

However, as we gather this information, it might make sense for us to shrink the output.

Why? Well, there are going to be many more small, simple patterns (like edges) in an image than large, complex ones (like a house).

This is natural: if it takes many edges to make one house, there will be fewer houses than edges.

So, combining our outputs into a smaller image let's us get a "bigger picture" on whatever is going on.

7.10.2 How to combine patterns

Our first thought might be to increase the filter size, or increase the stride size.

However, these both present problems: we may not want a larger filter, and using one might focus on patterns we don't care about.

Meanwhile, even though stride size makes the image smaller, you lose some information:

If the pattern is between two of your stride steps, for example, you might miss it. So, greater stride doesn't collect all the information over that range.

What we'll do instead is a new tool called **max pooling**.

7.10.3 What is max pooling?

Max pooling is a tool similar to ones we've used before, but with its own twist.

It behaves like a filter: we pick a size we're looking over, do a calculation, and then move our "window" around all over the input.

Sometimes we call the area that we're currently applying our filter to (to get the maximum value) the "receptive field" or just a "field".

So, we get the maximum value over our "receptive field".

Definition 307

Receptive Field: the portion of our input we're currently applying our **filter** to.

In this case, however, the calculation is different: all we do is **return the largest value over a certain range**.

Definition 308

Max pooling is the process of looking over our **field** and returning the **maximum** value over that region.

For an example: consider a (3×3) max pool window.

We would take a 3-length square and slide it around our input image, and for every position, we put the maximum value in a new output matrix.

7.10.4 Considerations for Max Pooling

A few comments:

First, notice that, since we're taking the maximum, there are no weights: the layer always operates in the same way.

In fact, there is no bias, either.

You can think of this kind of like a ReLU layer, which also has no weights, because it only relies on a simple function.

Because of this, we might call it a "pure functional layer".

Second, it has filter size, so it decreases size according to its size and stride, just like a regular filter.

Finally, remember that the goal of our max pooling layer is meant to aggregate information. So, we put two restrictions:

- Our stride $s > 1$, so we end up with a smaller output.

Hopefully, this is how we collect our information together.

- Our filter width $k \geq s$. This is so we don't skip over some values.

To understand why, consider that if we had a stride of 5, and a filter width of 1, then we would never touch a large part of our input!

We could miss crucial information that way.

Because we're only taking the maximum over a broad area, we lose some of the precise information about where a pattern was found.

This can take away some information, but it lets our machine learn to find patterns regardless of where they are.

7.10.5 An example

For example, let's take a matrix of size $(64 \times 64 \times 3)$.

We'll set our filter to size 2, with stride 2.

Using our equation for the size of our output, we get

$$\lceil (64 - 2 + 1)/2 \rceil = \lceil (63)/2 \rceil = 32$$

So, our output dimensions are $(32 \times 32 \times 3)$.

7.10.6 Concerns about Max Pooling

One problem with max pooling is that they don't perfectly handle "translation invariance", or, our algorithm not caring exactly where our pattern is.

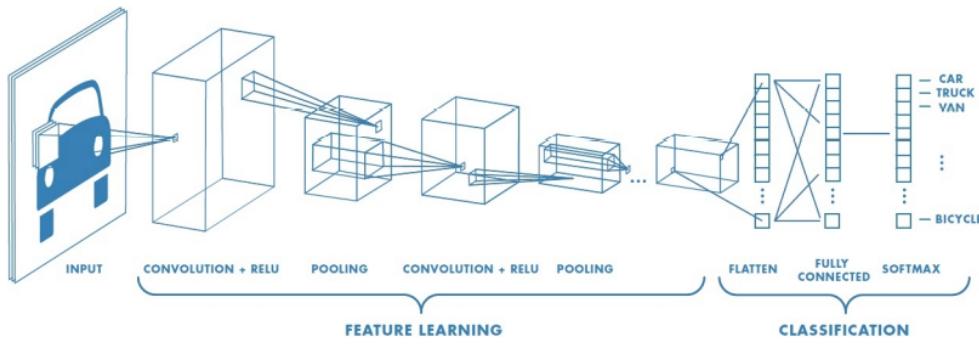
This is because, if your stride $s > 1$, then you could shift the pattern within that stride to change the max pooling output significantly.

This problem will not be elaborated further here, but here's a relevant paper (<https://arxiv.org/pdf/1904.11486.pdf>)

7.11 Typical architecture

7.11.1 An example of a CNN

Here, we give the form for 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>)

7.11.2 Common features of CNNs

First and foremost: a CNN is just a regular neural network, with some convolutional layers, like filters and max pooling.

Here, we get into some common features:

- After each filter layer there is generally a ReLU layer
- there maybe be multiple filter/ReLU layers
- Max pooling is often used after filtering to shrink the output.
- Once the output is relatively small, we finish with a fully-connected layer
- We close out with our activation function. For example, we might use softmax in a classification problem.

There's very few hard rules for how exactly to design this kind of network.

There are no good guidelines, either in theory or in practice, to tell us how our choice affects how well the networks does.

7.12 Training our Network

7.12.1 Can we train it?

Given the subject matter of this course, "Machine Learning", your first question might be, can we train this, and if so, how?

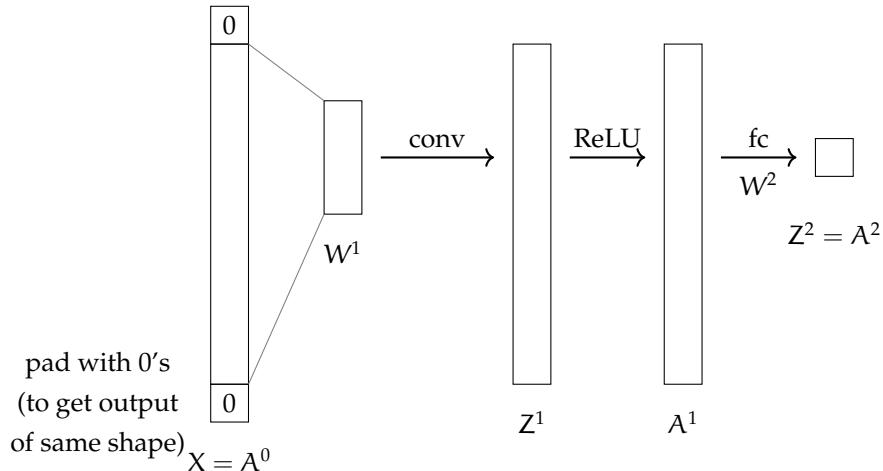
Well, thankfully, it turns out we can!

The various weights of filters and other components are (mostly) continuously differentiable, so we can return to our old friend, back-propagation.

(Technically ReLU and max pooling both don't have continuous derivatives, but we can usually get away with this)

7.12.2 An example

We'll try a simple example, just to understand how we can do back-propagation on convolutional networks.



In this example, we take our input, and run a convolution (getting Z^1), a ReLU (getting A^1), and then a fully connected network (finally, $Z^2 = A^2$).

We'll assume we have a 1D, single-channel input image, so its shape is $(n \times x1 \times 1)$.

We also have a single filter with no bias, of the shape $(k \times 1 \times 1)$.

7.12.3 Forward pass - Convolution

For starters, we need to figure out the output of each sub-layer.

First, we want to apply the convolution.

The convolution, as always, involves taking some slice of the input, and multiplying it (in a dot product, in this case) with our filter.

Our filter weights can be represented with a series of numbers as W^1 .

We take our input, X , or A^0 , and decide to index into it.

We'll use python notation for indexing, using square brackets. We'll also assume our filter is odd, because that's more common.

We'll use i to represent us sliding our filter. If we want to index the right length, we can do it two ways:

- We index starting from the very left of our field.

This would be $[i : i + k]$.

- We index starting in the middle of our field: we have an odd number on our filter, so we need to round down.

Thus, we get $[i - \lfloor k/2 \rfloor : i + \lfloor k/2 \rfloor]$

Which version we use depends on how we handle indices after padding.

For simplicity, we'll use the former (note that the official notes use the latter; do with this what you will)

Thus, we're doing a dot product: we'll represent this the same way we normally do, when we're doing linear regression: $\theta^T X$.

Thus, we have

$$Z_i^1 = (W^1)^T \cdot \{A^0[i : i + k]\}$$

7.12.4 Forward pass - The other steps

The rest of the steps are, thankfully, much simpler. They look more or less the same as before.

First, our ReLU step (remember that ReLU generally follows convolution):

$$A^1 = \text{ReLU}(Z^1)$$

Now, we do the fully connected step:

$$A^2 = Z^2 = W^2^T A^1$$

Finally, we get the square loss

$$L(A^2, y) = (A^2 - y)^2$$

7.12.5 Updating our filter

Now, how do we update our filter?

Well, we can use the chain rule.

First, we start at the end: how does our output, A^2 , affect the loss, L ?

$$\frac{\partial L}{\partial A^2}$$

We can write that with this simple derivative, and we have previously derived it.

Now, we move a step back. How does A_1 affect L ?

We combine how much A_1 affects A_2 , with how A_2 affects L .

After all, if we double the change in A_1 , that should double the chance in A_2 (since the derivative pretends our function is linear), and thus doubles the change in L .

$$\frac{\partial L}{\partial A^1} = \frac{\partial L}{\partial A^2} \cdot \frac{\partial A^2}{\partial A^1}$$

We repeatedly move backwards, and get

$$\frac{\partial L}{\partial W^1} = \frac{\partial L}{\partial A^2} \cdot \frac{\partial A^2}{\partial A^1} \cdot \frac{\partial A^1}{\partial Z^1} \cdot \frac{\partial Z^1}{\partial W^1}$$

7.12.6 Familiar Derivatives

Each of these derivatives except one is familiar.

We start with the square loss; we get it using the power rule.

$$L(A^2, y) = (A^2 - y)^2$$

$$\frac{\partial L}{\partial A^2} = 2(A^2 - y) \tag{7.1}$$

Then, we handle the fully connected layer. We just take the derivative normally, as if we were doing $\frac{d}{da}(wa) = w$.

$$A^2 = Z^2 = W^2 A^1 \tag{7.2}$$

$$\frac{\partial A^2}{\partial A^1} = W^2$$

We finally handle the ReLU.

As discussed previously (check the matrix derivatives notes!), we get it as follows:

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

$$\frac{\partial A_i^1}{\partial Z_i^1} = \begin{cases} 1 & \text{if } Z_i^1 > 0 \\ 0 & \text{otherwise} \end{cases}$$

7.12.7 Our New Derivative

(Note that this gets complicated.)

Finally, we have

$$Z_i^1 = (W^1)^T \cdot \{A^0[i : i+k]\}$$

We take the derivative like we did for the fully connected network, but instead of with respect to a , we'll do it for w : $\frac{d}{dw}(wa) = a$.

$$\frac{\partial Z^1}{\partial W^1} = A^0[i : i+k] ???$$

This is a bit confusing: what do we make of this?

Well, we'll use the rule built in the matrix derivative notes:

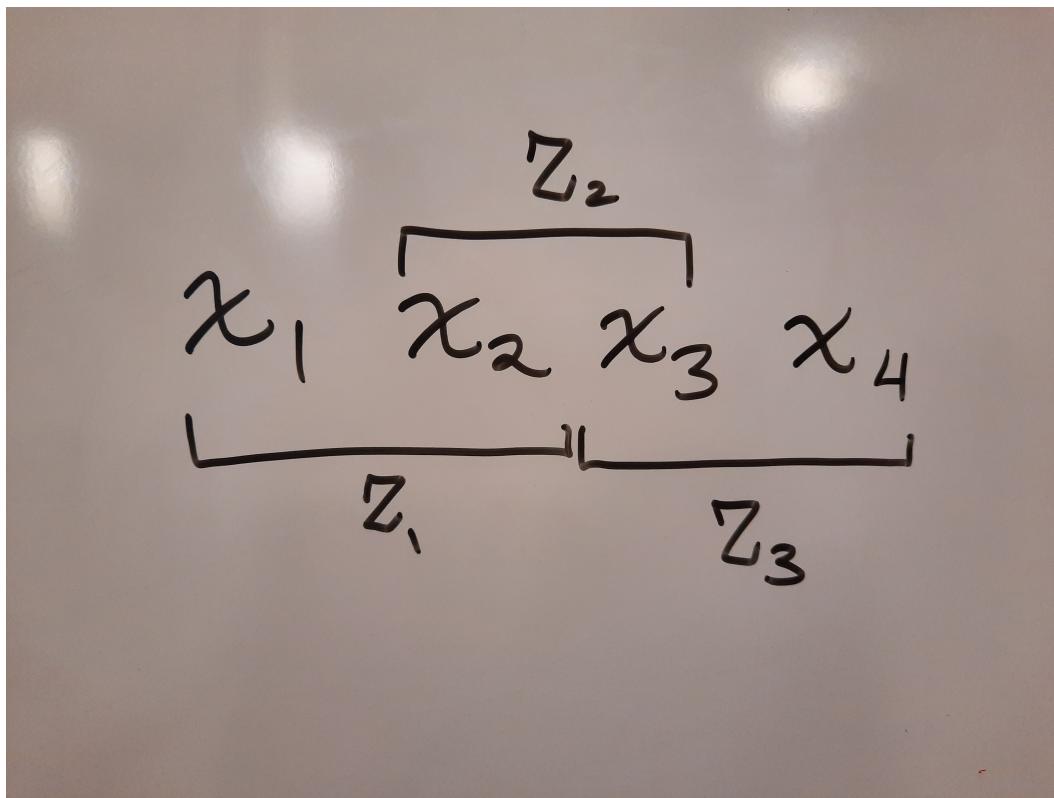
$$\left(\frac{\partial Z^1}{\partial W^1} \right)^T \Delta W^1 = \Delta Z^1$$

7.12.8 Derivatives: what's going on?

To make it clearer what's going on, we'll create a concrete example:

We'll do a 1D example. Our filter is 2 numbers long, and our input is 4 numbers long.

So, we'll get three outputs: z_1 , z_2 , and z_3 . Like so:



So our equations are:

$$z_1 = w_1 x_1 + w_2 x_2$$

$$z_2 = w_1 x_2 + w_2 x_3$$

$$z_3 = w_1 x_3 + w_2 x_4$$

Let's show what our matrix looks like:

$$\left(\frac{\partial z}{\partial w} \right)^T \Delta w \quad \Delta z$$

$$\begin{bmatrix} ab \\ cd \\ ef \end{bmatrix} \begin{bmatrix} \Delta w_1 \\ \Delta w_2 \end{bmatrix} = \begin{bmatrix} \Delta z_1 \\ \Delta z_2 \\ \Delta z_3 \end{bmatrix}$$

We'll focus on the first row.

$$a\Delta w_1 + b\Delta w_2 = \Delta z_1 \tag{7.3}$$

Notice that only w_1 and w_2 affect z_1 .

How much w_1 affects z_1 is proportional to x_1 . Same for w_2 and z_2 . So, we get

$$x_1 \Delta w_1 + x_2 \Delta w_2 = \Delta z_1 \tag{7.4}$$

We can use the same logic for each of these. We get

$$\begin{bmatrix} x_1 & x_2 \\ x_2 & x_3 \\ x_3 & x_4 \end{bmatrix} \begin{bmatrix} \Delta \omega_1 \\ \Delta \omega_2 \end{bmatrix} = \begin{bmatrix} \Delta Z_1 \\ \Delta Z_2 \\ \Delta Z_3 \end{bmatrix}$$

This sort of pattern can be used for all of our derivatives.

So, each row is represented by $A^0[i : i + k]$, where i is given by the row.

This explains why it appears in the derivative we calculated before!

So, now we just take the transpose, and this gives us our derivatives.

Thus, our derivative is

$$\frac{\partial Z_1}{\partial W_1} = [A^0[:, k], A^0[1 : 1 + k], \dots] \quad (7.5)$$

Where $A^0[i : i + k]$ is a column vector.

We have k rows, because that's the length of $A^0[i : i + k]$.

We have n^{l+1} columns - the number of outputs we expect.

So, the shape of our derivative is $(n^{l+1} \times k)$.

7.12.9 Buffer

7.13 Backpropagation in a simple CNN

7.13.1 Max Pooling

One last detail: how do we handle max-pooling?

Well, our function is

$$\text{maxpool} = \max(a_1, a_2, a_3\dots)$$

If a_1 is largest, then this simplifies to

$$\text{maxpool} = a_1$$

So, if we take the derivative with respect to each component:

$$\frac{d}{da_1} \text{maxpool} = 1$$

While

$$\frac{d}{da_2} \text{maxpool} = 0$$

This means that the back-propagation will affect a_1 and all of the parts of the network that fed into a_1 .

Meanwhile, there will be no change to any other a_n value, or anything that feeds into it.

So, that's how we handle derivatives for maxpool.

Thankfully, most of this mess is handled by existing software.

CHAPTER 12

Clustering

12.0.1 Why do clustering?

In chapter 4, we discussed **classification**: sorting data points into different groups, or **classes**.

Example: We might sort animals by **genetics**, or different sub-diseases that need different **treatments**.

Simplifying our data into **categories** can allow us to do better work, more easily.

This had lots of benefits:

- It could be used to make **decisions**. For example, a binary classifier could be used to decide "yes" or "no".
- We could use this to understand the item. It could be used to make yes or no decisions, and **distribution** of our data.
- We could sort different types of data to be processed **separately**.

The problem is, this relied on us **knowing** what classes we plan to sort into.

This may seem obvious, but what if we're looking at something **new**? A disease we don't fully **understand**, or animals we've never **seen** before? How do we classify them?

In the past, doing this ourselves has given rise to many of the **classes** we use to sort things today. But, computers allow us to do this in situations we **never** could before:

- **High-dimensional** datasets, with too much **complex** information for a human to make sense of.
- Discovering new classes **faster** than ever using computers.
- Finding **patterns** in creative ways humans would never think to, especially for really **abstract** problems.

Concept 309

Clustering is like **classification**, where we want to assign things to **classes**: we call them **clusters**.

But, we use it when we **don't know** what groupings we want, so we have to **find** them.

We have some challenges ahead of us, though. Not only do we need to create **new classes**, we *still* need to classify our points based on them!

12.1 Clustering Formalisms

12.1.1 Unsupervised Learning

The first thing we should note:

This problem is similar to classification, a **supervised** problem.

It was **supervised** because we knew the **correct** labels for our data in advance. We just wanted to **teach** it to our computer.

The problem here: we **don't** know the correct labels! In fact, we're making them up as we go. Because we aren't being "supervised" by a correct answer, we call this **unsupervised learning**.

Concept 310

Clustering is a type of **unsupervised learning**: meaning, we don't have a **correct** answer in advance.

The labels we create are not based on a **known** truth.

The **label** for data point $x^{(i)}$ is written as $y^{(i)}$.

12.1.2 What is clustering?

So, if we don't know **what** our classes are, how do we figure out **which** classes to create?

Intuitively, we think of classes as a **collection** of things that are **similar** to each other. Before, we've considered things "**close**" if they have a **low distance** in input space.

Example: We can tell two animals are **both** cats because they both have fur and sharp claws, among other things: they're **similar**.

Remember that input space is where we represent each data point using input variables.

Meanwhile, two points in different classes are more **different**: they're further apart in input space.

Example: We can tell a dolphin is **distinct** from a cat because one lives in water and one doesn't: their clusters are more **different**.

We call each of these groupings **clusters**.

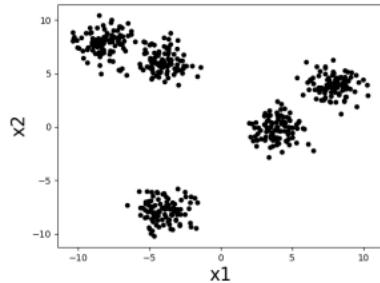
Definition 311

Informally, a **cluster** is a collection of **data points** that are all

- **Near** each other
- **Far** from the other clusters.

We use clusters as our way to **discover** new classifications.

Example: Below, we can visually mark out what looks like 5 distinct **clusters** in input space $(x_1, x_2) \in \mathbb{R}^2$:



This is an informal way to understand clusters, though. If we want to be more precise, we need to ask ourselves questions like:

- What does it mean for points to be "close" or "far"? How are we measuring distance?
- How many clusters do we want?
- How do we evaluate our clustering?

12.2 The k-means formulations

In this section, we'll introduce a common way to do clustering called the **k-means approach**.

12.2.1 Defining a cluster: The mean

We need to define what makes a "cluster" in order to move **forward**.

We want the points within a cluster to be as **close together** as possible. So, you might measure the **distance** from one point to all the others.

So, it would make sense to **average** them out. And we need to average every pair of points. That's a lot of work: can we **simplify** it?

Well, if we're trying to **average** the result of many data points, it would make sense to use the **mean**!

That's how we'll **define** our cluster: as the **mean**, the point that is the **average** of all the other points in the cluster.

Definition 312

We want to represent our **cluster** using its **mean**: the **average** of all of the data points in that **cluster**.

Our goal is for the **cluster mean** to have the **minimum average distance** possible to all of our data points: it's as **close** to our points as we can get.

Example: We describe the "male lifespan" using **life expectancy**: the **average** time a male human lives for. Same for women as well.

12.2.2 k-means

Now, we've created **one** cluster. To extend this to **many** clusters, we just need each cluster to have its **own** mean.

There are k of these clusters: this is why we call this the **k-means formulation**.

How do we decide which point goes in which **cluster**? Well, we want our points to be close. So, we'll assign it to the **closest** one.

Concept 313

A **point** is assigned to the **closest cluster mean**.

For a point $x^{(i)}$, the **output** is which **cluster** ("new class") it has been assigned to: $y^{(i)}$.

Once we've successfully clustered using our **algorithm** below, we will find that both of these goals are met:

- Our points are **assigned** to the **closest** cluster mean.
 - This separates **different** clusters of points from each other.
- The cluster mean is the **average** of all of our points: the **minimum distance** to them.
 - This makes sure our cluster is made up of points that are **similar** to each other.
 - If our point is close to the **mean**, it's probably close to the **other** points in the cluster.

12.2.3 k-means loss

Now, we know what we want out of our **clusters**. But, the problem is, we don't know **which** points will give us our nice clusters.

So, first, we will have to **assign** our initial "cluster means": often, we **randomly** select some points from our dataset.

Concept 314

We **initialize** our clustering by **randomly** selecting one point to **represent** each cluster, which we call the **cluster mean**.

At first, each point is assigned to the **closest** cluster mean.

But as you'll notice, these points are **not** the cluster means we're looking for! They're just a random **initialization**. So, we have to **optimize**.

Clarification 315

Notice that, when we **first** select our "cluster means", we don't get them by **averaging** any points: we choose them **randomly**.

That means, at first, is our cluster mean **isn't a true mean!**

Our k-means algorithm is designed to **fix** this problem.

In order to **improve** our clustering, it helps to have a way to measure the **quality** of a clustering: we need a **loss function**.

12.2.4 One-cluster loss

Let's start with just one cluster: what do we want to **minimize**?

Well, we want the points within a cluster to be as **close together** as possible. So, we want to minimize the **distance** to the mean, μ .

To make our function smooth, we'll use **squared distance** instead.

Concept 316

In **k-means loss**, we want to minimize the **square distance** from each point $x^{(i)}$ to the **cluster mean** μ .

$$D_i = \|x^{(i)} - \mu\|^2 \quad (12.1)$$

We'll add this up for each of the n data points in our cluster.

$$\mathcal{L} = \sum_{i=1}^n \|x^{(i)} - \mu\|^2 \quad (12.2)$$

12.2.5 Building up to k clusters

So, what do we do for each of our k clusters? Well, we can just **add** up the **loss** for them.

We'll use $j \in \{1, 2, 3, \dots, k\}$ to represent our j^{th} cluster. Each cluster has a mean $\mu^{(j)}$.

$$\mathcal{L}_j = \sum_{i=1}^n \|x^{(i)} - \mu^{(j)}\|^2 \quad (12.3)$$

Problem is, we're including **every** point $x^{(i)}$ in **every** cluster! We want a way to filter by **cluster**.

Remember that we **label** clusters the same way we labeled **classes** before:

Notation 317

For a **data point** $x^{(i)}$, its **cluster** is given by

$$y^{(i)} \in \{1, 2, \dots, k\}$$

Where j represents the j^{th} cluster.

Cluster mean $\mu^{(j)}$ is the j^{th} cluster mean: it only counts for points in c_j . So, we **only** want to add up the loss when

$$y^{(i)} = j \quad (12.4)$$

We'll do this using the following helpful **function**:

Notation 318

The **indicator function** $\mathbb{1}$ tells you whether a statement p is true:

$$\mathbb{1}(p) = \begin{cases} 1 & \text{if } p = \text{True} \\ 0 & \text{otherwise (if } p = \text{False)} \end{cases}$$

Combined with our **condition** of matching clusters, this can be useful:

$$\mathbb{1}(y^{(i)} = j) \quad (12.5)$$

If we **multiply** this by our loss, it'll **only** appear if the clusters **match!** We can **eliminate** data points in a different cluster.

12.2.6 k-mean loss: final form

So, we can **filter** by the data points in our cluster:

$$\mathcal{L}_j = \sum_{i=1}^n \underbrace{\mathbb{1}(y^{(i)} = j)}_{\text{Check cluster}} \underbrace{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|^2}_{\text{Dist from mean}} \quad (12.6)$$

And finally, we add up over many clusters:

$$\mathcal{L} = \sum_{j=1}^k \mathcal{L}_j \quad (12.7)$$

Using our equation, we get:

$$\mathcal{L} = \underbrace{\sum_{j=1}^k}_{\text{clusters}} \underbrace{\sum_{i=1}^n}_{\text{data points}} \underbrace{\mathbb{1}(y^{(i)} = j)}_{\text{Check cluster}} \underbrace{\left\| \mathbf{x}^{(i)} - \boldsymbol{\mu}^{(j)} \right\|^2}_{\text{Dist from mean}}$$

Let's clean that up:

Key Equation 319

The **k-means loss** is given as:

$$\mathcal{L} = \sum_{j=1}^k \sum_{i=1}^n \mathbb{1}(y^{(i)} = j) \|x^{(i)} - \mu^{(j)}\|^2$$

Where:

- μ_j is the **cluster mean**: the **average** of the points in the j^{th} cluster.
- $\mathbb{1}(y^{(i)} = j)$ is the **indicator function**: meaning that we only **include** terms where the data point and mean are in the **same cluster**.

12.2.7 Making further use of the indicator function (Optional)

We can actually use our **indicator function** to represent some of our **other** variables:

For example: the **cluster mean** is the average of data points, but **only** those belonging to that cluster.

So, we can use $\mathbb{1}(\cdot)$ to **filter** those other data points out:

$$\mu^{(j)} = \frac{1}{N_j} \sum_{i=1}^k \underbrace{\mathbb{1}(y^{(i)} = j)}_{\text{check cluster}} \underbrace{x^{(i)}}_{\text{data points}} \quad (12.8)$$

And how large is N_j ? We can just **add 1** for every data point in cluster, 0 otherwise:

$$N_j = \sum_{i=1}^k \mathbb{1}(y^{(i)} = j) \quad (12.9)$$

One more loose end:

Definition 320

The **variance** of a dataset is the **average distance from the mean**:

$$\text{Var}[X] = \frac{1}{N} \sum_{i=1}^N x^{(i)}$$

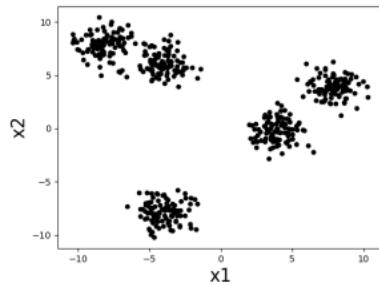
That means, our loss function is meant minimize the total **variance**. _____

Almost, the factor of $1/N$ is missing: since this won't change as we improve our clustering, we'll leave it alone.

12.2.8 Initializing the k-means algorithm

Now that we have our **clusters**, **means**, and a **loss** function for evaluating them, we can begin looking for a better **clustering**.

We'll start out with a **dataset** we want to cluster: we'll use the one from the **beginning** of the chapter:

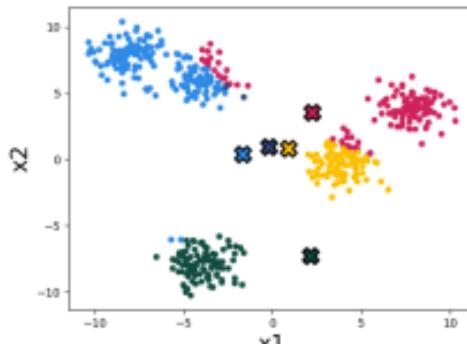


We could cluster this visually, but we want our machine to be able to do it for us.

First, we need to decide on our **number** of clusters. When you can't **visualize** it, this can be **difficult** - how many is too many or too few?

But, for now, we'll **ignore** that problem, and say $k = 5$.

Let's **randomly** assign our initial cluster means, and assign each point to the **closest** cluster:



1) Initial assignments

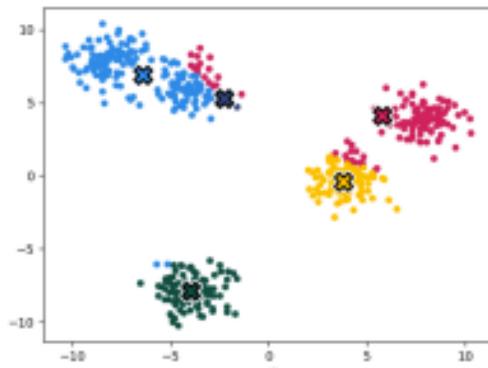
This is our starting point for the algorithm.

12.2.9 First step: moving our cluster means

As we mentioned before, these points aren't **actually** the average of their cluster: you can tell that by looking at it.

We want to **minimize** the variation in our cluster: that's why we're using the mean.

So, let's fix this: we'll take the **average** of all the points in each **cluster**, and **move** the cluster mean to that position.



2) Update means

And now, our cluster means are closer to all our data points!

Concept 321

One way **minimize** the **distance** between the **cluster mean** and its **data points** is:

- Take the **average** of all the points in the cluster, and **reassign** the cluster mean to that average.

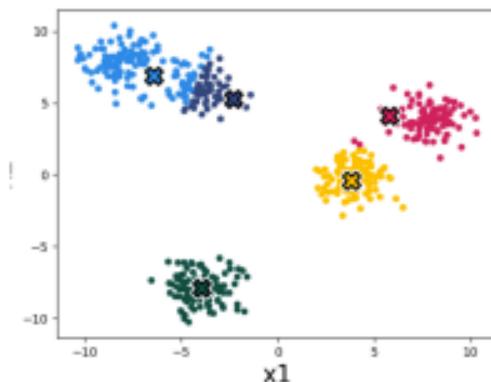
12.2.10 Second step: Reassign data points

We've **improved** our model by moving the cluster mean.

The problem is, we originally **assigned** every point to the **closest** cluster mean.

If the cluster means **move**, then some points might be closer to a **different** cluster now!

If so, we can **improve** our clustering further by reassigning points to the cluster they're **closest** to!



3) Update assignments

Concept 322

Another way **minimize** the **distance** between the **cluster mean** and its **data points** is:

- After the **means** have been **moved**, **reassign** the **data points** to whichever mean is **closest**.

12.2.11 The cycle continues

But wait - now that we've changed the points in each cluster, our cluster mean might not be the **true** average!

So, we can, again, improve our loss by taking the average of each cluster, and moving the cluster mean.

This creates a cycle that continues until we **converge** on our final answer.

Concept 323

Together, of our steps for **improving** our clusters create a **cycle** of **optimization**:

- **Moving** our cluster mean **changes** which point should go in each cluster.
- **Reassigning** points to different clusters **changes** our cluster mean.

12.2.12 The k-means algorithm

These two steps make up the **bulk** of our algorithm:

Definition 324

The **k-means algorithm** uses the following steps:

- First, we **randomly** choose our **initial** cluster means.

Then, we **cycle** through the following two steps:

- **Reassign points** to the cluster mean they're closest to.
- **Move** each **cluster mean** to the average of all the points in that cluster.

Until our clusters means **stop** changing.

When we run our algorithm on the above dataset, we get:

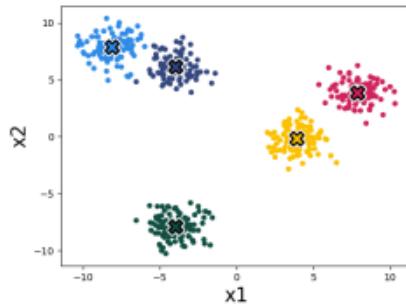


Figure 6.3: Converged result.

Note that, our cycle **works** because changing **either** cluster mean or point assignments allows you to further **improve** the **other** step.

So, if we're **not** changing one of them, the other one won't **change** either: the cycle is **broken**, and we can **stop**.

Another nice fact: it can be shown that this algorithm does **converge** to a local minimum!

This is our termination condition!

Concept 325

The **k-means algorithm** is guaranteed to **converge** to a **local minimum**.

12.2.13 Pseudocode

```

K-MEANS( $k, \tau, \{x^{(i)}\}_{i=1}^n$ )
1  $\mu, y = \text{randinit}$       #Random initialization
2 for  $t = 1$  to  $\tau$       #Begin cycling
3
4      $y_{\text{old}} = y$       #Keep track of last step
5
6     for  $i = 1$  to  $n$ 
7          $y^{(i)} = \arg \min_j \|x^{(i)} - \mu^{(j)}\|_2^2$       #Reassign data point to closest mean
8
9     for  $j = 1$  to  $k$ 
10         $\mu^{(j)} = \frac{1}{N_j} \sum_{i=1}^n \mathbb{1}(y^{(i)} = j) x^{(i)}$       #Move cluster mean to average of cluster
11
12    if  $\mathbb{1}(y = y_{\text{old}})$ 
13        break      #If nothing has changed, then the cycle is done. Terminate
14
15 return  $\mu, y$ 

```

12.2.14 Using gradient descent: minimizing μ

We can also use **gradient descent** to solve this problem!

We want to **minimize** our loss \mathcal{L} , and we do this by **adjusting** our cluster means $\mu^{(j)}$ until they're in the **best** position.

Concept 326

We can solve the **k-means problem** using **gradient descent**!

So, we want to **optimize** \mathcal{L} using μ :

$$\mathcal{L}(\mu) = \sum_{i=1}^n \mathbb{1}(y^{(i)} = j) \|x^{(i)} - \mu^{(j)}\|^2 \quad (12.10)$$

Rather than dealing with the indicator function $1(\cdot)$, we could instead just consider whichever μ is closest: **minimum** distance.

$$\underbrace{\min_j}_{\text{Minimizing}} \underbrace{\|x^{(i)} - \mu^{(j)}\|^2}_{\text{distance}} \quad (12.11)$$

This **automatically** assigns every point to the closest **cluster** before we get our loss! So, all we need to worry about is μ_j .

Notation 327

Instead of using an **indicator function**, we can represent **cluster assignment** another way: using the **function** \min_j .

It can give **minimum distance** from $x^{(i)}$ to one of the cluster means: it picks the **closest** mean.

This **automatically** assigns the point to the **closest** cluster, making our job easier.

$$\mathcal{L}(\mu) = \sum_{i=1}^n \widehat{\min}_j^{\text{Nearest cluster}} \|x^{(i)} - \mu^{(j)}\|^2 \quad (12.12)$$

Now, we can do gradient descent using $\frac{\partial \mathcal{L}(\mu)}{\partial \mu}$.

$\mathcal{L}(\mu)$ is **mostly** smooth, except when the cluster assignment of a **point** changes. So, it's usually smooth **enough** to do gradient descent.

We move our means until they're minima!

12.2.15 Getting labels

Once we've finished gradient descent, and we've **minimized** our loss, we can get our **labels**.

\min gives the **output** value that we get by minimizing. In this case, average **squared distance** from the cluster mean.

Meanwhile, $\arg \min$ gives us the **input** value that gives us the minimum output. In this case, the **cluster** that gives the minimum distance.

So, $\arg \min$ gives us the cluster closest to each point: that's our **label**!

We can use this notation to get our **labels**.

Notation 328

After **optimizing** μ , our **labels** are given by:

$$y^{(i)} = \arg \min_j \|x^{(i)} - \mu^{(j)}\|^2 \quad (12.13)$$

Using gradient descent can give us a **local** minimum, but our surface is not fully **convex**: so, we don't necessarily get a **global** minimum.

Even though individual terms of squared distance may be convex, adding min terms may not be convex.

12.3 How to evaluate clustering algorithms

The biggest problem with clustering algorithms is that they're **unsupervised**: this makes it much harder to know if we've gotten a **good** result.

This is partly because our **loss** function doesn't necessarily tell us if clustering is **useful**, or represents the data **accurately**.

It just tells us if our points are **close** to their cluster **mean**. That doesn't always mean the clustering is **good**.

Example: Imagine **every** single point in the dataset gets its **own** cluster mean. The **distance** to the cluster mean would be 0 (low loss), but this isn't very **useful**!

Clarification 329

The **k-means loss function** does **not** tell us if we have a good and **useful** clustering or not.

This isn't useful because nothing has changed: we've gone from having n separate data points, to having... n separate clusters.

It only tells us if the points in our clusters are **close** to their **cluster mean**.

This can help us make **better** clusters, but that does not mean they are **good** or what we **want**.

Without having "true" labels, we have to find other ways to **verify** our approach.

We'll do two things to **approach** this problem:

- We'll look at some of the ways our **algorithm** can go wrong (or right).
- Then, we'll find **better** ways to evaluate our clusterings than just looking at the **loss**.

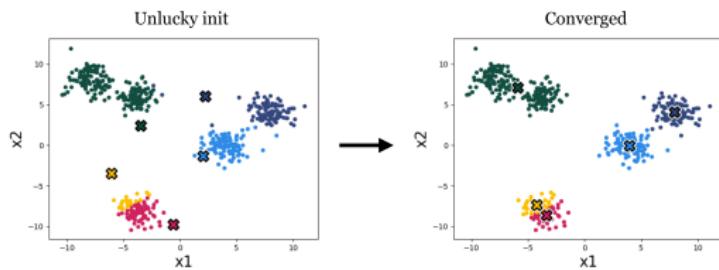
But, always remember that a "good" clustering is partly **subjective**, and depends on what you **want** to accomplish.

12.3.1 Initialization

The first problem we have is related to something we mentioned at the end of the last section: k-means is not **convex**.

That means we can find **local** minima that are not the **global** minimum: our **initialization** (our **starting** clusters) can affect whether we end up in a useful minimum.

The reason why is, mathematically, the same as when we first introduced the idea of a local minimum.



In this example, notice that we ended up with convergence on some very **bad** clusters: the bottom cluster is split in **half**!

The easiest way to resolve this is to run k-means multiple times with different initializations.

Other techniques exist, but this is the simplest one.

Concept 330

Getting an **unlucky initialization** can result in **clusters** that aren't **useful**.

We try to **solve** this by running our algorithm **multiple times**.

12.3.2 Choice of k

One important question we decided to **ignore** earlier was: **how many** clusters should we pick in advance?

Especially for **complex** data, we **don't know** how many natural clusters there will be.

But our number of clusters matter: because it's a parameter determines **how** our learning algorithm runs (rather than being chosen *by* the algorithm), it's a **hyperparameter**:

Concept 331

Our **number of clusters** k is a **hyperparameter**.

And, choosing too high *or* too low can both be **problematic**:

- If we set k too **high**, then we have more clusters than actually **exist**.
 - This can cause us to **split** real clusters in half, or find **patterns** that don't exist.
 - In a way, this resembles a kind of **overfitting**: we try to **closely** match the data, but end up fitting **too closely** and not **generalizing** well: **estimation error**.
 - **Example:** The **extreme** case looks like the example we mentioned **before**: when labeling animals, we could make... a different **species** for every single instance of **any** animal we find.
- If we set k too **low**, we don't have **enough** clusters to represent our data.

That doesn't sound very helpful.

- This means some clusters will be **lumped together** as a single thing: we **lose** some information.
- In this case, it's **impossible** to cluster everything in the way that would make the most **sense**: we have **structural error**.
- **Example:** Let's say we wanted to **sort** fish, birds, and mammals into **two** categories: we might just **divide** them into "flies" and "doesn't fly".

That's some information, but often not enough!

Concept 332

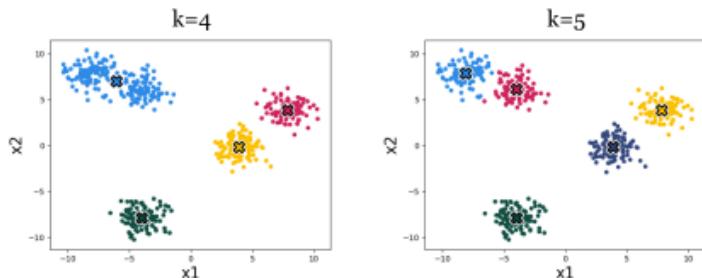
When choosing k (our **number of clusters**), we can cause **problems** by picking an inappropriate **value**:

- **Too many** clusters (large k) can cause **overfitting** and **estimation error**: we find patterns we don't want.
- **Not enough** clusters (small k) causes **structural error**: it prevents us from correctly **separating** data.

12.3.3 Subjectivity of k

Not only is it hard to choose a "good" value of k , what a good value of k is can really depend on your opinion, and what you know about reality.

For example, consider the following example:



Which of these two clusterings is more accurate?

Should the top left be **one** cluster, or **two**? It's hard to say!

Even if you're **sure**, you might **disagree** with others, or find that the best one depends on your **needs**.

So not only can k values be too high or too low, they can also be **debatably** better or worse!

Concept 333

The **best** choice of **clustering** is not entirely objective: it can depend on your **opinion**, or how you plan to **use** the clustering.

What do we mean by, what we're "**using**" the clustering for? We'll get into that later, but in short: we might use **clusters** to make sense of **information**, or to make better **decisions**.

Different clusterings might be good when you want a different kind of understanding.

Example: The understanding you get from high-level comparisons (plants vs animals vs bacteria) is different from low-level comparison (cats vs dogs).

12.3.4 Hierarchical Clustering

That last example reveals something: not all types of groups are the same! Some are much broader than others, for example.

If two types of groups are different, then why do we only have to have one type in our clustering? We don't have to restrict ourselves to a single k.

Instead, we could treat some groups as inside of other groups: we call this a *hierarchy*, because some groups are "higher" on the scale.

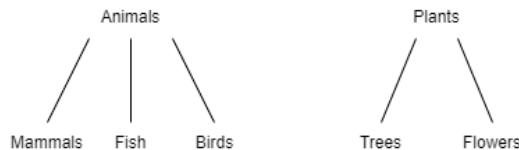
Definition 334

Hierarchical Clustering is when we cluster at multiple different **levels**.

Some groups are **high-level**, or **coarse**: they are groupings that contains **more elements**: items in the same group can be more **different**.

Some groups are **low-level**, or **fine**: they are groupings that contain **fewer elements**: items in the same group have to be very **similar**.

Example: Categorizing living things is done using hierarchical clustering: some groupings **contain** other groupings.



The top row of clusters are more **coarse**, while the bottom row is more **fine**.

We can **split** our groupings into **smaller** and smaller ones, to be as **fine-grained** as we need. Useful!

12.3.5 k-means in feature space

One **important** consideration: when working with **feature** representations, we found that sometimes, feature transformations made it **easier** to **classify** data.

Clustering is very **similar**, so, could we do the same here? It turns out, we **can!**

Often, it wasn't even possible without those transformations!

Rather than directly clustering in **input** space x , we can **process** our data using features, and then cluster in **feature** space $\phi(x)$.

Concept 335

Feature transformations can be used to make it easier to **accurately** cluster our data in a **meaningful** way.

There are some **other** reasons to do feature transformations, though: imagine that our data is **stretched** out along axis x_1 , but not x_2 .

x_1 distances would be **larger** in general: it would contribute more to our distance metric! We could correct for this by **standardizing** our data: **scaling down** the more stretched axis.

This would be a **feature** transformation, and would make it **easier** to do our **clustering**.

12.3.6 Solutions: Validation

Now, we start trying to answer the **question**: how do we **check** whether we have a **good** clustering?

Well, first, we can check for a **poor fit** (or overfitting) using new, **held-out** testing data: do we get **low loss** on that testing data?

If we **don't**, then our clusters definitely aren't **representative** of the overall **dataset**: they don't **generalize** to new data.

Concept 336

If our clusters give **large testing loss**, then they aren't **generalizing** well, and are probably **not representative** of the overall distribution.

So, we already know our clusters **don't fit the distribution**.

12.3.7 Solutions: Consistency

But, just like for classification/regression **validation**, we don't only run our algorithm **one time**: we'll run it **many** times, with different training and testing sets.

We can't **just** use the loss, though: having **more** clusters could make our error lower, without making a better clustering, for example.

Another thought: we're trying to find some patterns **inherent** in the data. The idea is: if the pattern we're finding is **real**, we should find a similar pattern **each time**!

So, we look to see if our clusters are **consistent** when we generate them using different training data: if they aren't, then it's possible we're not finding the "real" patterns in the data.

Different training data from the same distribution, of course.

Concept 337

If our **clusters** accurately **reflect** the underlying classes of data, then we should expect some **consistency** of which clusters we **generate** by running k-means many times.

If our clusters aren't **consistent**, then we might doubt if any of them especially reflect the **distribution**, rather than **noise**.

If our clusters are **consistent**, then we're probably seeing something about the **real** dataset.

12.3.8 Solutions: Ground Truth

But, even if we're getting something **consistent**, that doesn't mean we're seeing the patterns that **matter**.

If it was based on random noise, then the odds of getting matching results would be really low!

One way to **check** this is, if we have some idea of what the "**true**" clustering looks like for just a few data points, we can compare those results to ours.

We call this "real" clustering the "ground truth".

Definition 338

In machine learning, the **ground truth** is what we know about the "real world".

In general, we want our models to be able to **reproduce** this reality: it is the data that we tend to **trust** the most, if it is gathered correctly.

That way, we can use a very **small** amount of **supervision** to get an idea of whether our clustering is on the **right track**.

12.3.9 Applications: Visualization and Interpretability

We've discussed some ways to **abstractly** test whether our clustering might be **accurate** the data.

But, when it comes down to it, often, the **quality** of a clustering is based on how **useful** it is. So, what sorts of **uses** does clustering **have**?

Well, we're organizing our data into **groups**: this **simplifies** how we look at our data. And it allows us to **view** at our data, and **understand** what's going on.

In short: clustering allows humans to more easily make sense of data.

Concept 339

One of the the main **goals** of **clustering** is to make it easier for humans to **understand** the data.

This happens in two ways:

- We can **visualize** the data: we can **see** it, and more easily use our **intuition** to make sense of it.
- We can **interpret** our data: by seeing what sorts of **groupings** we create, we learn about the **structure** of the data.

So, machine learning experts judge partly based on how well a clustering **helps** them **achieve** these two goals.

Evaluating clusterings is **subjective** for exactly this reason: what is **good** "visually", or is the **best** "interpretation" of data, is often up to **debate**.

So, **human judgement** is important for this type of **problem**.

12.3.10 Applications: Downstream Tasks

Finally, there's one more way to think about clustering that is more **practical**, and closer to **objective**.

We use clustering to **sort** different data points that need **different** processing: this can make our model more **effective**, since different parts of the dataset may work **better** with different **treatment**.

Example: We could train a different regression model on each cluster: this can create a more accurate model.

We call this next problem a **downstream application**.

Definition 340

A **downstream application** is a **problem** that relies on a **different** process to make its work better or easier.

In this case, **clustering** has **downstream applications** that can **take advantage** of the **structure** it reveals.

If our clustering is **good**, we would expect it to **improve** the performance of downstream tasks.

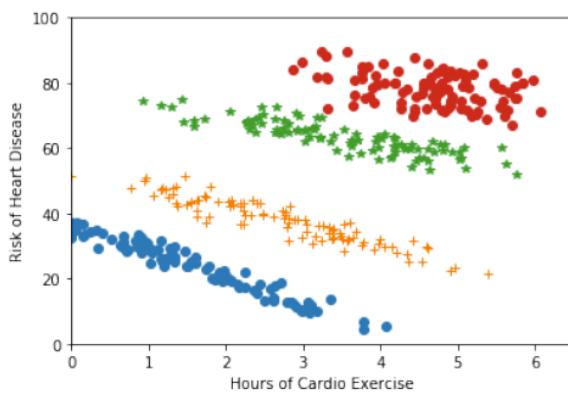
Concept 341

We can indirectly **evaluate** a **clustering algorithm** based on how **successful** the **downstream application** is.

If it **improves** the performance of a downstream application, we could say it works **well**.

12.3.11 A benefit of clustering

One advantage for downstream applications is, there might be patterns that are more obvious if you only look at related segment of the data. For example:



If we take the data as a whole (**no clustering**), we would draw a **positive** regression: it seems that exercise and heart disease increase **together**. That doesn't make sense!

But, if we divide it into **clusters**, based on age, we see a **negative** relationship: each individual group experiences **benefits** from exercise.

This particular issue is called **Simpson's Paradox**.

Definition 342

Simpson's Paradox is when a **trend** that appears in groups of data either **vanishes** or **reverses** when we look at all the data **together**.

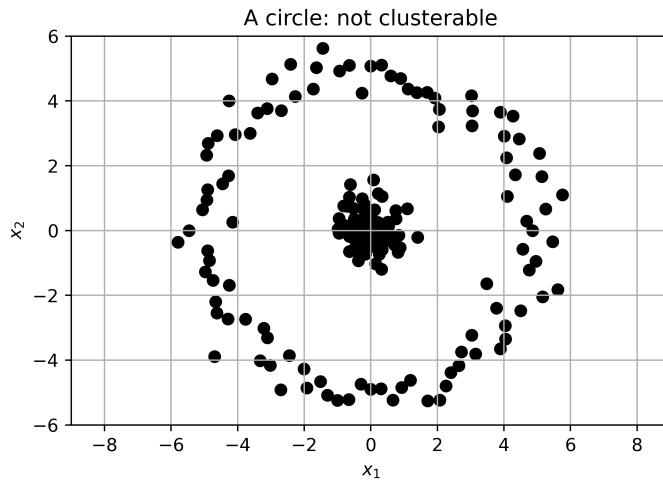
It shows that sometimes, **patterns** that we see may reflect how we're **looking** at the data.

Rest assured, you don't need to know this paradox by **name**! But it's important to **understand** possible problems like it: it'll make you more **responsible** in the future!

12.3.12 Weaknesses of k-means

There are some **weaknesses** to k-means clustering. Certain patterns that we can **see** aren't easily **clustered**.

We can see this with a few **examples**:



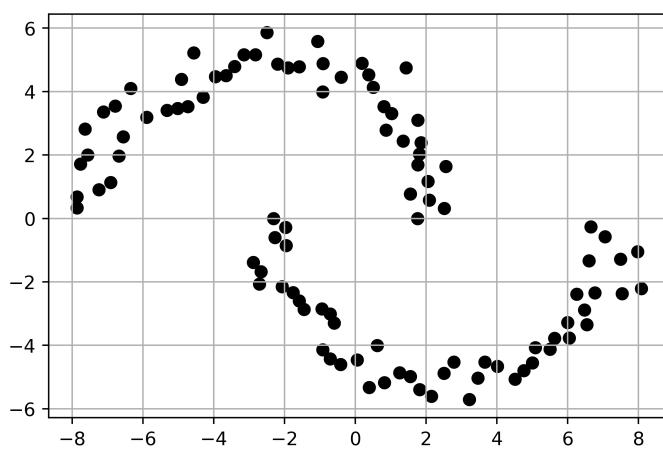
This data can't be simply clustered.

This example can't be effectively **clustered**: most people would agree that the "outer ring" should be **one** cluster, while the "inner circle" should be **another**.

But, assuming we (correctly) place one cluster mean in the **center**, there's **nowhere** we can put our other cluster mean to be **closest** to all of the **outer** points, but **not** the inner points.

We might be able to **resolve** this using a **feature** transformation. But, the problem remains.

Another example works for clusters that aren't very centralized:



For example, we could have a feature represent the radius! But then, we would still struggle with a ring not centered on the origin.

This data can't be clustered either!

The edge of one cluster is too close to the other: we can't easily create a good pair of cluster means for each semi-circle.

12.4 Terms

- Clustering
- Unsupervised Learning
- Cluster
- Cluster mean
- k-means problem
- k-means loss
- Initialization
- Indicator Function
- Variance (Optional)
- k-means algorithm
- Hierarchical Clustering
- Consistency
- Ground Truth
- Visualization
- Interpretability
- Downstream Application
- Simpson's Paradox (Optional)