

Explanatory Notes for 6.390

Shaanticlair Ruiz (Current TA)

Spring 2023

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	77
2.8	Terms	87
3	Gradient Descent	90
3.1	Gradient Descent in One Dimension	95
3.2	Multiple Dimensions	105
3.3	Application to Regression	116
3.4	Stochastic Gradient Descent	121
3.5	Terms	123
4	Classification	124
4.1	Classification	124
4.2	Linear Classifiers	129
4.3	Linear Logistic Classifiers	150
4.4	Gradient Descent for Logistic Regression	167
4.5	Handling Multiple Classes	171
4.6	Prediction Accuracy and Validation	182
4.7	Terms	183
5	Feature Representation	184
5.1	Gaining intuition about feature transformations	190
5.2	Systematic feature construction	195
5.3	Hand-constructing features for real domains	205
5.4	Terms	219
6	Clustering	220
6.1	Clustering Formalisms	222
6.2	The k-means formulations	224
6.3	How to evaluate clustering algorithms	235
6.4	Terms	244
7	Neural Networks 1 - Neurons, Layers, and Networks	245
7.1	Basic Element	249
7.2	Networks	261
7.3	Choices of activation function	284
7.4	Loss functions and activation functions	292
7	Neural Networks 1.5 - Back-Propagation and Training	295
7.5	Error back-propagation	295
7.6	Training	322
7	Neural Networks 2 - Training Techniques, Regularization	325
7.7	Optimizing neural network parameters	325
7.8	Regularization	350

7.9 Terms	366
8 Autoencoders	369
8.1 Autoencoder Structure	372
8.2 Autoencoder Learning	378
8.3 Evaluating an Autoencoder	380
8.4 Linear Encoders and Decoders	386
8.5 Advanced Encoders and Decoders (Optional)	392
8.6 Terms	405
9 Convolutional Neural Networks	406
9.1 Filters	413
9.2 Max-pooling	445
9.3 Typical architecture	452
9.4 Backpropagation in a simple CNN	455
9.5 Terms	464
A Matrix Derivatives	465
A.1 Introduction and Review	465
A.2 Derivative: Scalar/Vector (Gradient)	471
A.3 Derivative: Vector/Scalar	478
A.4 Derivative: Vector/Vector	481
A.5 General derivative (Vector/Vector)	483
A.6 Derivative: matrix/scalar	488
A.7 Derivative: scalar/matrix	490
A.8 Tensors	492
A.9 Chapter 7 Derivatives	499
A.10 Terms	510

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

$$P(A) \tag{5}$$

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:

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

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

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

- 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 (including the full space 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 Chains
- 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 TA as of Spring 2023. 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.

– **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.

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

- **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 set of models: 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.

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.

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

But first, one more detail.

1.1.2 How do we store our data?

A data point usually represents a single "thing". It's helpful to be able to use **multiple** facts about this one thing.

To store these facts, a data point requires **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.

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

- **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.

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

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

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 might be 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.

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

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

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

Concept 17

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

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.

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.

~~~~~  
**Parameters** tell you about your **model**, while the **input** variables describe one piece of **data**.

**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**,  $\theta$ . Just like for  $x$ , we'll use a **column vector**.

#### Notation 20

$\theta$  is our **vector of parameters**.

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

**Example:** Here is a vector  $\theta$  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  $i^{\text{th}}$  **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  $\theta$  (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  $h(x; \theta)$ .

$x$  is our main input, while  $\theta$  is used to **define** our function (using parameters).

**Example:** Consider every linear function  $y = mx_1 + b$ .

The input is a single value  $x_1$ , while the parameters are  $m$  and  $b$ : we can put those into  $\theta$ .

$$x = \begin{bmatrix} x_1 \end{bmatrix} \quad \theta = \begin{bmatrix} m \\ b \end{bmatrix} \quad h(x; \theta) = mx_1 + b$$

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

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

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  $\theta$ , 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  $\theta$ .

### 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 equation are put in the same **model class**. Since our models are defined by their **hypothesis**, we will often talk about the **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 description:

- 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  $\theta_i$  are our parameters.

## 1.5.5 Linear Model: Vector Form

$\theta$  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  $\theta$  must have the same **shape**.

**Concept 27**

When using a linear model,  $x$  and  $\theta$  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  $\theta_0$  out to make it work: if we want to talk about **all** parameters, we'll instead use the symbol  $\Theta$ .

**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: using matrices will make things easier (later in the class!)

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

Finally, we condense our vectors into symbols.

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

**Key Equation 29**

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

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

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 sparingly in this class.

Instead, we will focus on our **linear** model, in stages:

- We'll upgrade the linear model with a non-linear function, so we can solve **non-linear** problems.
- We will combine many of these "non-linear units" to create **neural networks**.

Arguably, neural networks are 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.

We'll explore some neural network variants on later on:

- Convolutional Neural Networks
- Recurrent Neural Networks

## 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  $\Theta$ : this vector defines our **model** within our **model class**.

This  $\Theta$  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  $\Theta$  is a **variable**.
- We'll focus on  $\Theta$  more if we assume we know which **hypothesis class** we're in - if we know that,  $\Theta$  fully **defines** our model.

Often, people will treat  $\Theta$  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  $\varphi(x)$ .

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

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

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**  $\mathcal{E}_n$  is written as:

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

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.

$$\mathcal{E}(h) = \frac{1}{m} \sum_i \mathcal{L}(h(x^{(i)}), y^{(i)}) \quad (2.2)$$

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

**Key Equation 35**

**Testing Error**  $\mathcal{E}$  is written as:

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

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** (work better for more situations).

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**  $\Theta$ , so we take that as an input:  $J(\Theta)$

We will be seeing a lot of  $\Theta$  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  $\Theta$  than before, 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  $\lambda$  so we can control this term:

**Key Equation 37**

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

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

- The left term is the **loss**: how well we perform on training data.
- The right term is the **regularization**: it improves how "general" (good for new situations) our model might be.

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: our data determines how well we're doing.

- Just like how we can use ";" when writing  $h(x; \Theta)$ , we'll use the same notation here:  $J(\Theta; \mathcal{D})$
- $\Theta$  is our "main" input variable, but if we switch out our data  $\mathcal{D}$ , we would get a different result.

A good solution for a political science exam is probably pretty bad for a geology exam.

**Clarification 38**

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

$$\overbrace{J(\Theta)}^{\text{Uses } \Theta} = \overbrace{\mathcal{E}_n(h)}^{\text{Uses } h} + \lambda R(\Theta)$$

The difference is that **training error**  $\mathcal{E}_n(h)$  is **more general** than the **objective function**  $J(\Theta)$ , and  **$h$**  is **more general** than  **$\Theta$** .

- By "more general", we mean that there are more situations where we can use  $h$  than  $\Theta$ .
  - $\Theta$  is a list of parameters: we only use it if we have a **parametric model**.
  - $h$  is used if we have **any kind of model**.

So, let's compare  $\mathcal{E}_n(h)$  and  $J(\Theta)$ :

- **Training error** can be used for any model  $h$ , so we don't want to use  $\Theta$ : our model could be **non-parametric**.
- Our **objective function assumes** we have parameters  $\Theta$ : we know our model class  $H$ , we just want to optimize  $\Theta$ .

### 2.2.3 Minimization Notation

Our goal is to minimize  $J$  by adjusting  $\Theta$ . 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**  $\Theta$ ?

#### 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:**

$$\min_x (x - 1)^2 = 0 \quad (2.3)$$

0 is the minimum value of  $J$  we can find by adjusting  $\Theta$ .

- What **value** of  $x$  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:**

$$\arg \min_x (x - 1)^2 = 1 \quad (2.4)$$

1 is the value of  $x$  which gives the minimum  $J$ .

**Clarification 41**

Why is it called "**argmin**"?

"**Argument**" is used as another word for "**input variable**".

And our argmin function returns the **argument** with the **minimum** output. Hence, **arg min**.



## 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  $\Theta$ .

**Notation 42**

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

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

**Example:**

$$x^* = 1 \text{ for the above example.} \quad (2.5)$$

So, if we want optimal  $\Theta$ , we're looking for:

**Key Equation 43**

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:

$$y = mx + b \quad (2.6)$$

A linear equation:

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

$b$  and  $m$  are our parameters: that means they're part of  $\Theta$ . We'll rename them  $b = \theta_0$  and  $m = \theta_1$ .

$$h(x) = \theta_1 x + \theta_0 \quad (2.7)$$

### 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  $x_1$ .

$$h(x) = \theta_1 x_1 + \theta_0 \quad (2.8)$$

The simplest way to include  $x_2$  by just **adding** it. We have a scaling factor  $\theta_1$  for  $x_1$ , so we'll give  $x_2$  its own **parameter**,  $\theta_2$ :

If  $\theta_1$  is the "slope" for  $x_1$ ,  $\theta_2$  is the "slope" for  $x_2$ .

$$h(x) = \theta_2 x_2 + \theta_1 x_1 + \theta_0 \quad (2.9)$$

### 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.

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots + \theta_d x_d \quad (2.10)$$

### 2.3.4 The Linear Model using Vectors

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

$$h(x) = \theta_0 + \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} \quad (2.11)$$

If we write this symbolically, we get:

$$h(x) = \theta_0 + \theta \cdot x \quad (2.12)$$

Unfortunately, we had to leave  $\theta_0$  out to make it work.  $\theta$  is used for the parameters of our **dot product**,  $\Theta$  is all parameters.

#### Notation 44

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 45

The **linear regression** hypothesis is written as

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

Remember that, when written out, this looks like:

$$h(x) = \begin{bmatrix} \theta_1 & \theta_2 & \theta_3 & \dots & \theta_d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_d \end{bmatrix} + \theta_0 \quad (2.13)$$

Make sure you know what  $\theta^T$  is: it's the transpose!

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 is 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**.

$$\mathcal{L}(g, a) = (g - a)^2 \quad (2.14)$$

Or, if we use  $y = a$ :

$$\mathcal{L}(g, y) = (g - y)^2 \quad (2.15)$$

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

Our slope  $\frac{d}{dx}x^2 = 2x$  gets larger as we move away from  $x = 0$ .

#### Concept 46

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

- 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.
  - We can see that  $\|w\|$  isn't smooth at  $w = 0$ .
- The **slope** is **small** when you're close to the correct answer: if you're close to right, the loss stays low.
  - Inversely, if you're really wrong, then the model really penalizes you for being more wrong.
  - For example: the difference between  $2^2$  and  $1^1$  ( $2^2 - 1^1 = 3$ ) is much smaller than between  $24^2$  and  $23^2$  ( $24^2 - 23^2 = 47$ )

### 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**:  $J(\Theta) = J(\theta, \theta_0)$

$$J(\theta, \theta_0) = \text{Training Loss} \quad (2.16)$$

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

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(g^{(i)}, y^{(i)}) \quad (2.17)$$

Substitute in **squared loss**:

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n (g^{(i)} - y^{(i)})^2 \quad (2.18)$$

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

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n (h(x^{(i)}; \Theta) - y^{(i)})^2 \quad (2.19)$$

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

### Key Equation 47

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

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

To clarify:

$$J(\theta, \theta_0) = \underbrace{\frac{1}{n} \sum_{i=1}^n}_{\text{Averaging}} \left( \underbrace{(\theta^T x^{(i)} + \theta_0)}_{\text{guess}} - \underbrace{y^{(i)}}_{\text{answer}} \right)^2 \quad (2.20)$$

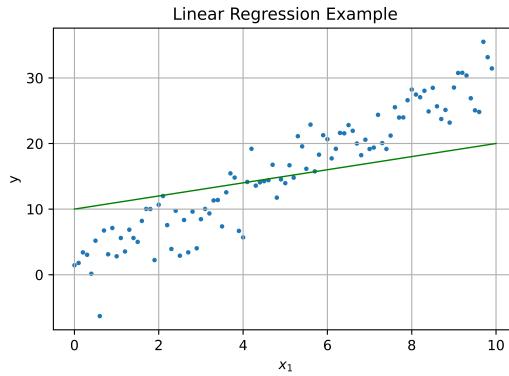
From this, we want to find

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

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

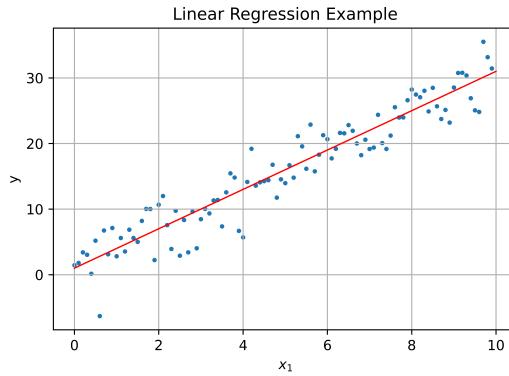
### 2.3.7 Visualizing our Model

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:  $(\theta_0 = 10, \theta_1 = 1)$

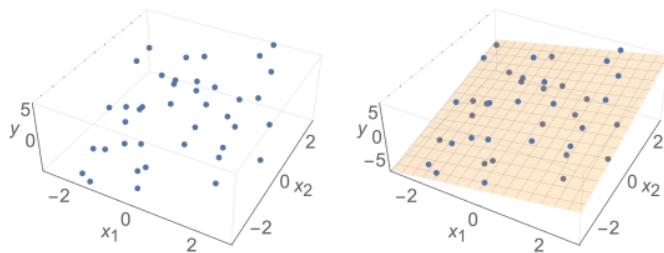
We're trying to get our line as **close as possible** to the points, hoping to find a linear pattern.  
We're **fitting** our line to the data.



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

What does this 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 even try to. Because they're a higher-dimensional version of a **plane**, we call it a **hyperplane**.

#### Definition 48

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**.

- We have  $d$  dimensions ( $d$  variables) in our input.
- To represent our output, we need one additional,  $(d + 1)^{\text{th}}$  dimension.

Thus, the "**height**" of our plane ( $(d + 1)^{\text{th}}$  dimension) at a particular point in the ( $d$ -dimensional) **input** space represents the **output** of our linear hypothesis, given those inputs.

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

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots + \theta_d x_d \quad (2.21)$$

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

- However, we can also think about the **relative scale** of each  $\theta_k$ : if  $\theta_2$  is **larger** than  $\theta_1$ , 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 this, we sometimes call  $\theta_k$  the **weight** for  $x_k$ .

#### Definition 49

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  $\theta$  and  $\theta_0$ . How do we do that? Let's start as simple as we possibly can.

We can't try **every** possible  $\Theta$ , 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  $\theta$  and  $\theta_0$ , 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.

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

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**  $\theta_0$  from our other  $\theta_k$  terms. Maybe we can **fix** that.

### 2.5.2 Combining $\theta$ and $\theta_0$

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

$$h(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots + \theta_d x_d \quad (2.23)$$

We converted this into a **dot product** because each  $\theta_n$  term is **multiplied** by an  $x_k$  term, except  $\theta_0$ .

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  $\theta_0$ , then could we? We know what's missing: " $\theta_0$  is **not** multiplied by an  $x_k$  term". So... could we get one? Is there a  $x_0$  factor we could **find**?

We need  $\theta_0$  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!

$$h(x) = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \dots + \theta_d x_d \quad (2.24)$$

So, this means we just have to **append** a 1 to our vector  $x$ . At the **same time**, we'll append  $\theta_0$  to  $\theta$ !

$$x = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_d \end{bmatrix}, \quad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_d \end{bmatrix}, \quad h(x) = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_d \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_d \end{bmatrix} \quad (2.25)$$

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

$$h(x) = \theta \cdot x = \theta^T x \quad (2.26)$$

### Concept 50

Sometimes, to simplify our algebra, we can **append**  $\theta_0$  to  $\theta$ .

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:

$$J = \frac{1}{n} \sum_{i=1}^n (\theta x^{(i)} - y^{(i)})^2 \quad (2.27)$$

- 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  $\theta_k$  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**.

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

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

$$J = \frac{1}{n} \sum_{i=1}^n (\theta x^{(i)} - y^{(i)}) (\theta x^{(i)} - y^{(i)}) \quad (2.29)$$

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

$$J = \frac{1}{n} \sum_{i=1}^n r^{(i)} * r^{(i)} \quad (2.30)$$

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

$$J = \frac{1}{n} \sum_{i=1}^n r^{(i)} * r^{(i)} = \frac{1}{n} \begin{bmatrix} r^{(1)} \\ r^{(2)} \\ r^{(3)} \\ \vdots \\ r^{(n)} \end{bmatrix} \cdot \begin{bmatrix} r^{(1)} \\ r^{(2)} \\ r^{(3)} \\ \vdots \\ r^{(n)} \end{bmatrix} \quad (2.31)$$

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**.

$$\theta = \left[ \begin{array}{c} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \vdots \\ \theta_d \end{array} \right] \left. \right\} \text{dimensions as rows...} \quad (2.32)$$

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

$$R = \overbrace{\begin{bmatrix} r^{(1)} & r^{(2)} & r^{(3)} & \dots & r^{(n)} \end{bmatrix}}^{\text{data points as columns!}} \quad (2.33)$$

## 2.5.5 Going from $x$ to $X$

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

**Notation 51**

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

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

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

We can write our **objective function** as

$$J = \frac{1}{n} \left[ r^{(1)} \quad r^{(2)} \quad r^{(3)} \quad \dots \quad r^{(n)} \right] \begin{bmatrix} r^{(1)} \\ r^{(2)} \\ r^{(3)} \\ \vdots \\ r^{(n)} \end{bmatrix} \quad (2.34)$$

Or more compactly:

$$J = \frac{1}{n} \mathbf{R} \mathbf{R}^T \quad (2.35)$$

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

$$\mathbf{R} = \theta \mathbf{X} - \mathbf{Y} \quad (2.36)$$

Still in the 1D case!

Let's use this to expand our objective function:

**Concept 52**

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

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

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:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_d \end{bmatrix} \quad (2.37)$$

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.

$$\mathbf{X} = \begin{bmatrix} x^{(1)} & x^{(2)} & x^{(3)} & \dots & x^{(n)} \end{bmatrix} \quad (2.38)$$

Why not combine them into a single object?

### Key Equation 53

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

$$\mathbf{X} = \underbrace{\begin{bmatrix} x_1^{(1)} & \dots & x_1^{(n)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \dots & x_d^{(n)} \end{bmatrix}}_{\text{n data points}} \}_{\text{d dimensions}} \quad (2.39)$$

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

$$\mathbf{X} = \underbrace{\begin{bmatrix} 1 & \dots & 1 \\ x_1^{(1)} & \dots & x_1^{(n)} \\ \vdots & \ddots & \vdots \\ x_d^{(1)} & \dots & x_d^{(n)} \end{bmatrix}}_{\text{n data points}} \}_{\text{d + 1 dimensions}} \quad (2.40)$$

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

### Key Equation 54

$\mathbf{Y}$  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  $\theta$  is a matrix, we have to **transpose** it.

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

**Key Equation 55**

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 56**

Our matrices have the shapes:

- $X$ :  $(d \times n)$  - matrix
- $Y$ :  $(1 \times n)$  - row vector
- $\theta$ :  $(d \times 1)$  - column vector
- $\theta_0$ :  $(1 \times 1)$  - scalar
- $J$ :  $(1 \times 1)$  - scalar

If we combine  $\theta_0$  into  $\theta$ , 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 57**

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  $\theta$ .

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

$$J(\theta) = \frac{1}{n} \sum_{i=1}^n (\theta^T x^{(i)} - y^{(i)})^2 \quad (2.41)$$

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

$$J(\theta) = (\theta^T x - y)^2 \quad (2.42)$$

And we'll start in 1D.

$$J(\theta) = (\theta x - y)^2 \quad (2.43)$$

If we treat  $\theta$  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  $\theta$  represents our **model**, we'll take  $\frac{d}{d\theta}$ .

$$J'(\theta) = 2x(\theta x - y) = 0 \quad (2.44)$$

We just find where the slope is 0, and solve for  $\theta$ !

$$\theta^* = \frac{y}{x} \quad (2.45)$$

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

### Concept 58

If our function  $J(\theta)$  has **one variable**, we can **explicitly** find **local minima** by solving for  $\theta$  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**:

$$J(\theta) = \frac{1}{n} \sum_{i=1}^n (\theta x^{(i)} - y^{(i)})^2 \quad (2.46)$$

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.

$$J'(\theta) = \frac{1}{n} \sum_{i=1}^n 2x^{(i)} \cdot (\theta x^{(i)} - y^{(i)}) = 0 \quad (2.47)$$

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 <sup>(i)</sup> notation since that's easy to add on afterwards.

$$J(\theta) = (\theta^T x - y)^2 \quad (2.48)$$

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

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

$$\Delta\theta = \begin{bmatrix} \Delta\theta_0 \\ \Delta\theta_1 \\ \Delta\theta_2 \\ \vdots \\ \Delta\theta_d \end{bmatrix} \quad (2.49)$$

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:

$$\underbrace{\Delta\theta \frac{dJ}{d\theta}}_{\text{The change in } J \text{ from } \theta \text{ overall}} \approx \underbrace{\Delta\theta_0 \frac{\partial J}{\partial \theta_0} + \Delta\theta_1 \frac{\partial J}{\partial \theta_1} + \cdots + \Delta\theta_d \frac{\partial J}{\partial \theta_d}}_{\text{The change in } J \text{ from each } \theta_k \text{ term}} \quad (2.50)$$

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 59**

If our function  $J(\theta)$  has **d+1 different parameters**, 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  $\theta$ .

- Why  $d + 1$  instead of  $d$ ? Because we're including  $\theta_0$  as one additional parameter.

The **solution** to this system of equations will be our **desired list of parameters**,  $\theta^*$ .

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 60**

The **gradient** of  $J$  with respect to  $\theta$  is

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

For example, our previous approach boiled down to saying

$$\nabla_{\theta} J = \begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \vdots \\ \frac{\partial J}{\partial \theta_d} \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix} = 0 \quad (2.51)$$

Note the subscript on the gradient! This emphasizes that our **space** is the components of  $\theta$ , not the components of our data  $x$ .

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

**Concept 61**

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

$$J = \frac{1}{n} (\tilde{\mathbf{X}}\theta - \tilde{\mathbf{Y}})^T (\tilde{\mathbf{X}}\theta - \tilde{\mathbf{Y}}) \quad (2.52)$$

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

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**.

When we take our derivative, we get

$$\nabla_{\theta} J = \frac{2}{n} \tilde{\mathbf{X}}^T (\tilde{\mathbf{X}}\theta - \tilde{\mathbf{Y}}) = 0 \quad (2.53)$$

From here, we just solve for  $\theta$  just like in the official notes.

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

**Key Equation 62**

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,

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

---

- Remember that if we're including  $\theta_0$ , we replace  $d$  with  $d + 1$ .

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 63

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** we didn't train with.

This function takes in our **vector of parameters**  $\Theta$  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 saw the problem with 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,  $\Theta_{\text{prior}}$ , to be **suspicious**, and less likely to be good.

**Concept 64**

If we have a **prior** hypothesis  $\Theta_{\text{prior}}$  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  $\theta_k$  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.

For each variable  $x_i$ , we don't know if it's likely to increase or decrease  $y$ . So we assume that the most likely effect is "none".

**Key Equation 65**

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

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

In this model, we are biasing our  $\|\theta\|$  towards 0.

This approach is called **Ridge Regression**.

In practice, this means that we're assuming that no particular variable is likely to affect the output very much.

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

- A small amount of change in the output would still be expected, because of random noise.

So, we start off assuming that we can't make a more detailed prediction, and allow the data to try to convince us otherwise.

- If the effect is small, then we're more sure that it's just noise.
- But, if the effect is large, it'll override the regularizer, and we expect that variable  $x_i$  really is affecting  $y$ .

#### 2.6.4 A second benefit: avoid error amplification

Let's demonstrate another possible problem: a large  $\theta_k$  can "amplify" the error due to randomness.

Let's see how this can happen.

- Suppose that two inputs are **almost exactly the same**:  $x_1^{(1)} = 0$ , and  $x_1^{(2)} = 0.01$ .
- But, the outputs are **somewhat** different. The outputs are  $y^{(1)} = 25$ ,  $y^{(2)} = 26$ .

Same variable  $x_1$ , but  
two different datapoints  
 $i = 1, i = 2$

If we assume **all** of the change in the output is a result of the **the input**, then it looks like a **tiny** change in  $x_1$  has a **huge** effect on the output  $y$ .

$$\frac{dy}{dx} = \frac{\Delta y}{\Delta x} = \frac{1}{.01} = 100 \quad (2.54)$$

This suggests that  $x_1$  has a **100x** effect on our output!

$$\theta_1 = 100 \quad (2.55)$$

$$100x_1 + 25 = h(x) \quad (2.56)$$

This model perfectly fits our data.

- But in reality, it's possible (and pretty common in real data) for  $y$  to change a bit (for example,  $\pm 3$ ) under the exact same circumstances, just because of **randomness**.
- If this is true,  $y$  only changed by 1. We probably shouldn't expect this change to give us much information!

Now, we have a problem. Even if  $x_1$  never mattered, we think it does. We see the effect for our new data point,  $x_1^{(3)} = 2$ .

$$100(2) + 25 = 225 \quad (2.57)$$

If we compare to  $x^{(1)}$ , our result has changed by 800%. But if  $x_1$  didn't really matter, and everything else is the same, then we've made a big mistake:  $y^{(3)} \approx 25$ .

This is what we mean by **error amplification**: a **small** difference in the input of +2 becomes a **very large** error in the output of +200.

### Concept 66

Regularization guards against the problem of **error amplification**.

Suppose the input  $x_1$  changes by a tiny amount, and the output  $y$  changes by a larger (but still small) amount.

- In this situation,  $x_1$  looks like it's a very important variable!
- $\theta_1$  becomes large.
- This can happen even if it's just random chance.

If  $x_1$  changes by a larger amount (again randomly), then you get a **huge change** in the **output**: this can cause a **huge error**!

Thus, the **error** resulting from random noise has been **amplified**.

If we bias  $\theta_1$  towards remaining smaller, then we avoid this problem.

Regularization biases against this kind of issue. Of course, including more data also helps fight overfitting.

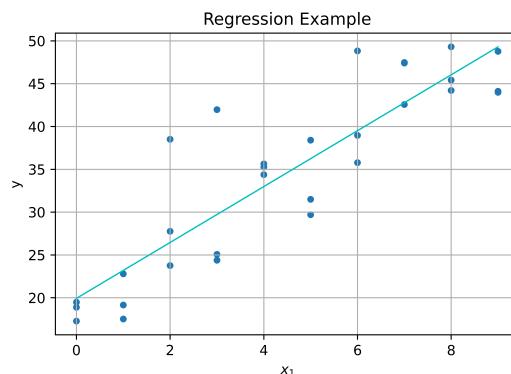
### 2.6.5 Why not include $\theta_0$ ?

One thing you might immediately notice is that we used the magnitude of  $\theta$  instead of  $\Theta$ : this omits  $\theta_0$ . 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.

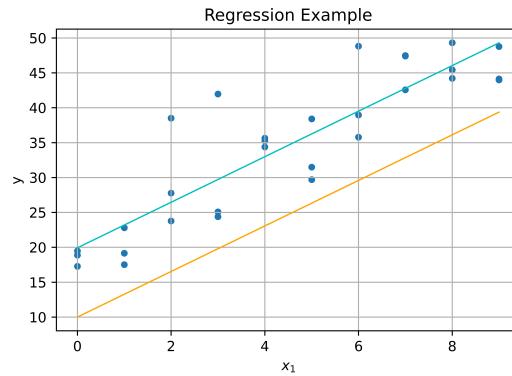
This is best shown with a **visual** example. Let's take an example with one input  $x_1$ . So, we have a **linear** function:  $h(x) = \theta_1 x_1 + \theta_0$ .

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)  $\theta_0$  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  $\theta_1$  wouldn't make this any better: it would just be flatter.

So, we'll keep  $\theta_0$  **separate** and **allow** it to take whatever value is **best**.

#### Concept 67

We **do not regularize** our **offset** term,  $\theta_0$ .

Instead, we allow  $\theta_0$  to **shift** our hyperplane wherever it **needs** to be.

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

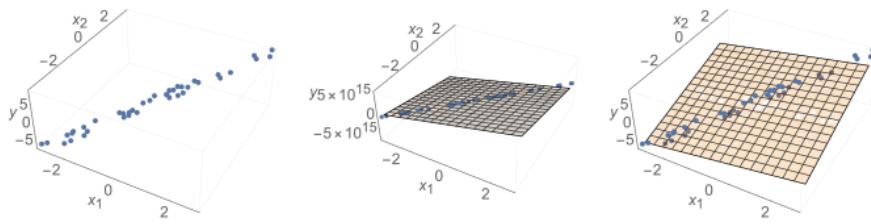
This will be discussed more in-depth in the Classification chapter!

## 2.6.6 A third benefit of regularization

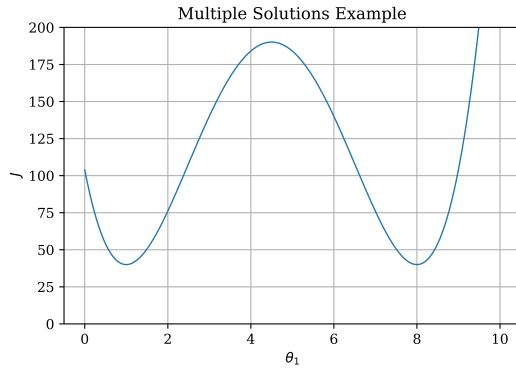
Another benefit of regularization is that it solves a more abstract problem: having **multiple optimal solutions**.

If we have **multiple** best outcomes, we have to pick one of them. 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 ( $\theta_1$ ) and sees how well they perform ( $J$ ): two are equally good!

Either way, we can pick a solution based on lowest  $\theta$  **magnitude**!

### 2.6.7 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!

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**

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

**magnitude.**

#### Concept 68

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**.

### 2.6.8 Lambda, a.k.a. $\lambda$

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  $\lambda$  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,  $\lambda$ .

#### Definition 69

**Lambda**, or  $\lambda$ , 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.

### 2.6.9 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 70**

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**.

### 2.6.10 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  $\theta$  and  $\theta_0$  is, frankly, annoying to **derive**. **Instead**, we'll cheat a little, and keep  $\theta_0$  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 71**

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,

$$\theta = (X^T X + n\lambda I)^{-1} X^T Y$$

### 2.6.11 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  $(3 \times 3)$  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  $\lambda$ s along the diagonal, is why we call it **ridge regression**.

## 2.6.12 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 72

**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 $\lambda$ should we choose?

There's something we ignored earlier: how do we pick the **best** value of  $\lambda$ ? We didn't go into detail, but that value of  $\lambda$  will affect our algorithm's **performance**.

- We mentioned that different  $\lambda$  values have different **tradeoffs**, so we need to figure out which  $\lambda$  value is best for our problem.
- This  $\lambda$  adjusts exactly how we learn: how do we balance learning from **data** against the need to **generalize**?

So, we need to **optimize** our  $\lambda$  value. Let's figure out how to go about that.

### 2.7.2 Tradeoffs: Estimation Error

High and low  $\lambda$  values have benefits and drawbacks. These tradeoffs can be loosely divided into **two categories**.

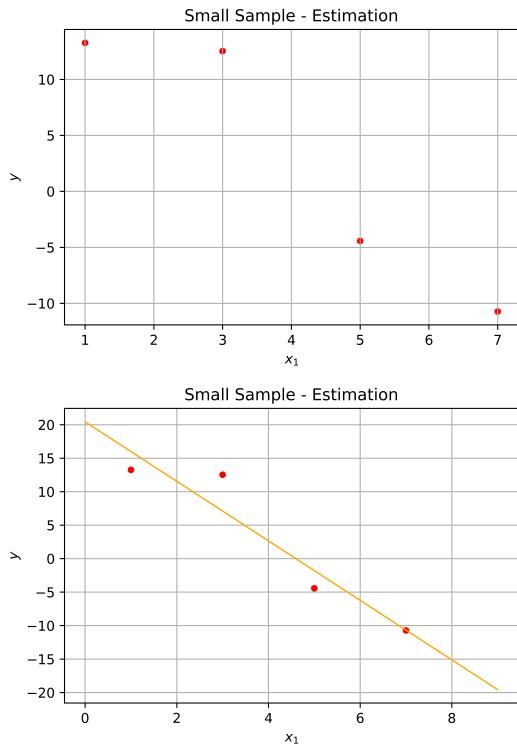
When we generalize, we're trying to avoid **estimation error**: we incorrectly guess the overall distribution we're trying to fit. We **estimate** poorly if we **generalize** poorly.

#### Definition 73

**Estimation error** is the error that results from poorly **estimating** the **solution** we're trying to find.

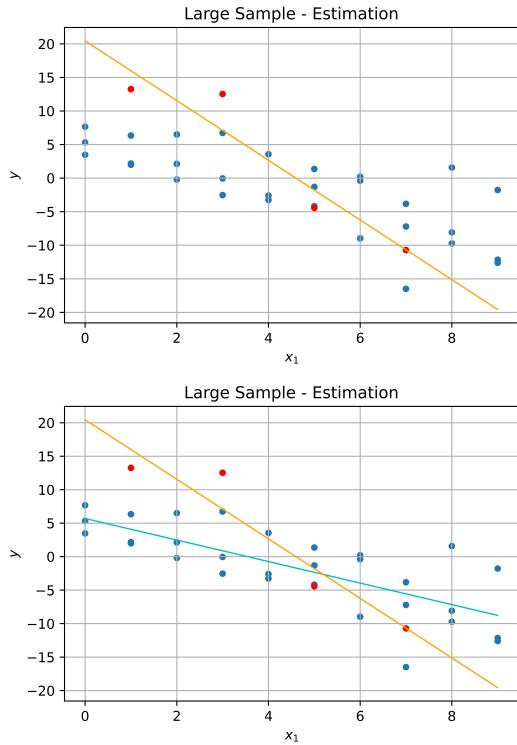
This can be caused by **overfitting**, getting a bad (**unrepresentative**) sample, or not having enough **data** to come to conclusion.

**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 74

$\lambda$  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**.

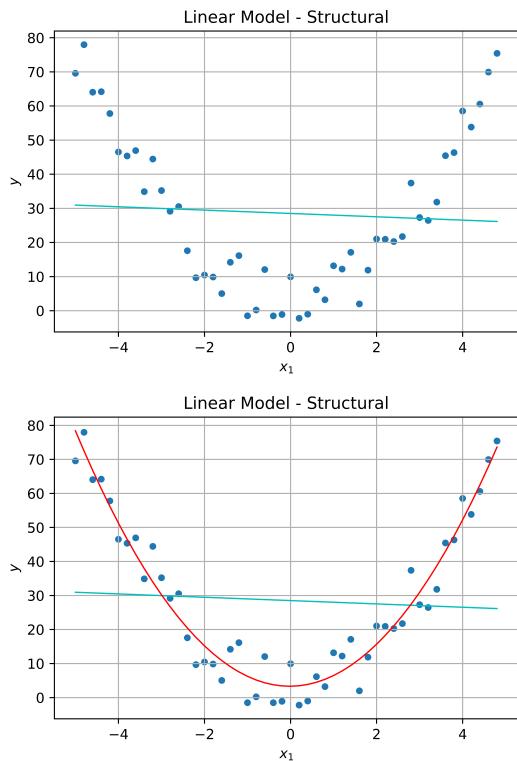
#### Definition 75

**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  $\lambda$  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 76

Note that  $\lambda$  does not restrict our model class **as severely** as **switching polynomial order**, like above.

But,  $\lambda$  **limits** the use of larger  $\theta$ , 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 $\lambda$

Based on these two categories, we can discuss the tradeoffs of  $\lambda$  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 77**

A large  $\lambda$  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 78**

A small  $\lambda$  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 79**

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 $\lambda$ '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  $\lambda$  is part of our **algorithm**: it determines how we find our hypothesis. So, let's talk about that.

Our hypothesis only includes the parameters  $\Theta$ : not  $\lambda$ !

**Definition 80**

A **learning algorithm** is our procedure for **learning** from data. It uses that data to create a **hypothesis**. We can diagram this as:

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

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

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

And compare it to our hypothesis  $h$ :

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

In a way, our learning algorithm is a function, that outputs another function!

This is similar to  $\mathcal{E}_n$ , which instead takes a function as **output**!

- Our **hypothesis** can be adjusted with our **parameter**  $\Theta$ : if we change  $\Theta$ , we change our **performance**.
- Our **learning algorithm** depends on  $\lambda$ : so,  $\lambda$  is like a **parameter**. But, it's different from  $\Theta$ :  $\Theta$  **is** our model,  $\lambda$  controls how we **choose** our model.

- So, it's a parameter ( $\lambda$ ) that affects other parameters ( $\Theta$ ). Because of that, we call it a **hyperparameter**.

It affects our hypothesis by pressuring it to have lower magnitude!

#### Definition 81

**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  $\lambda$ , but the **development** of hyperparameters is an ongoing area of **research**.

#### Concept 82

**Lambda**, or  $\lambda$ , 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 83

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 84**

**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 our data is limited?

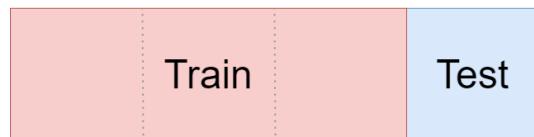
In this case, we'll assume that have some **finite** data,  $\mathcal{D}_n$ . We **can't get more**.

Data is often **expensive**. It might even be impossible to get more!

Previously, we solved validation by using **more data**, and generating **multiple hypotheses**.

- One set of data gives us one **hypothesis**.
- But, what if, rather than using **completely** new data for each hypothesis, we used **slightly different** data each time?

First, need to break  $\mathcal{D}_n$  into a chunk for training, and a chunk for testing.



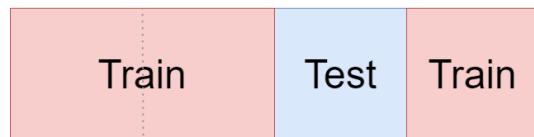
How do we get more hypotheses from this dataset?

### 2.7.11 Cross-Validation

We mentioned that we 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** training 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 85

**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 86

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**.

How do we **optimize** a learning algorithm?

Each  $\lambda$  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.

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**.

By "analytical", we mean directly creating an equation, and solving it.

So, we could **randomly** try  $\lambda$  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  $\lambda$  values:  $\lambda = .1, .2, .3 \dots$

#### Concept 87

**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  $\lambda$ . For each value, we use **cross-validation** to evaluate that learning algorithm.

Finally, we pick whichever  $\lambda$  gives you the **best** algorithm, and thus the **best** hypotheses.

### 2.7.14 Hyperparameter Tuning: Two kinds of optimization

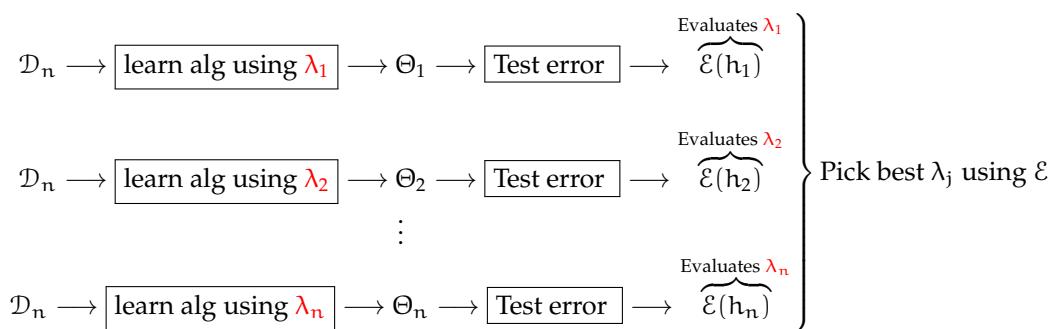
There's something often **confusing** about hyperparameter tuning to students:

- When we're **optimizing**  $\lambda$ , we try many values  $\lambda_j$ . Each  $\lambda_j$  create a **learning algorithm** we have to evaluate.
- But a single learning algorithm is already an optimization problem: the learning algorithm is supposed to find  $\Theta$ .
  - So, we have to **optimize**  $\Theta$ , while we're in the middle optimizing  $\lambda$ .

In case the word "optimization" starts to look like gibberish in this section, remember: it just means, "find the best option".

That means, **every time** we try a different  $\lambda$  value, we have to do one optimization problem. Optimizing  $\Theta$  many times, lets you optimize  $\lambda$  once.

Remember that, in this situation,  $\Theta$  and  $h$  are almost(but not quite) the same thing.



That means we have **two layers** of optimization!

#### Clarification 88

We **optimize**  $\lambda$  by trying many values.

- But, for each  $\lambda$  value, we have to **optimize**  $\Theta$ .

So, we have to optimize  $\Theta$  **repeatedly** in order to optimize  $\lambda$  **once**! This gives us  $\lambda^*$ .

Once we've found our best hyperparameter  $\lambda^*$ , we can use it to get our best parameters:  $\theta^*$ .

#### 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  $\Theta$  values.

We can represent this in pseudocode:

```
LAMBDA-OPTIMIZATION(D, lambda_values, theta_values)
1  for λ in lambda_values      #Try lambda values
2    for Θ in theta_values      #Try theta values
3      Calculate J(Θ)          #Compare values
4      Choose best theta value Θ*  #Best for each lambda
5  Choose best lambda value λ*
6
7  return λ*
```

If this pseudocode isn't helpful to you, don't worry! Some students like it, some don't.

To reiterate: this  $\lambda^*$  will then we used to get our final result,  $\theta^*$ .

## 2.8 Terms

- Hypothesis
- Theta ( $\Theta$ )
- Input Space
- Regression
- Feature
- Feature Transformation
- Training Error
- Test Error

- Objective Function
- Min function
- Argmin function
- Star Notation ( $\theta^*$ )
- 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 ( $\lambda$ )
- 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 89

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 90

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  $\theta$  by some amount: \_\_\_\_\_

$$\theta_{\text{new}} = \theta_{\text{old}} + \Delta\theta \quad (3.1)$$

We'll do the same for  $\theta_0$ , but we'll do it separately. We'll come back to that.

### Notation 91

In equations, we'll often use  $\theta_{\text{old}}$  and  $\theta_{\text{new}}$  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  $\theta$ ? 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  $\theta$ : ( $d \times 1$ ).

### Concept 92

We need a **separate** term  $\Delta\theta_i$  for each  $\theta_i$  we want to **improve**.

So, a vector of the **total** change,  $\Delta\theta$ , needs to have the **same shape** as  $\theta$ : ( $d \times 1$ ).

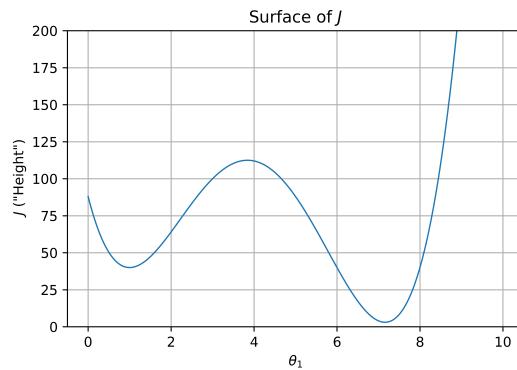
### 3.0.3 The name: "gradient descent"

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**"?

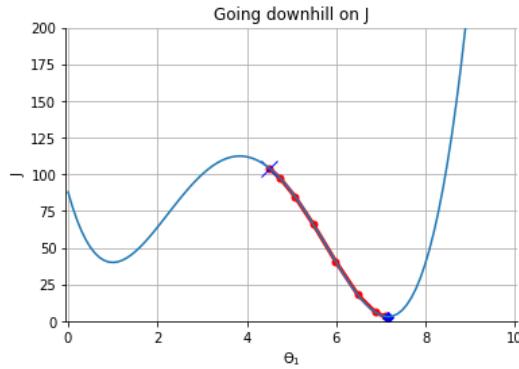
Our intuition is to imagine  $J$  as having a **height** at every input value. If you combine all of these different points, you get a **surface**, like the surface of a hill.

Reminder: Why are we "decreasing"  $J$ ? Because  $J$  represents the "badness" of our model. We want it to be, simply put, "less bad".



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  $\theta$  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  $\theta$  values. We also called this our "**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!

#### Definition 93

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.

- This can be seen as the "collection of **all possible models** in our model class".
- Why do we call it a "parameter **space**", not a "parameter set"?
  - It's the **structure**: the fact that some hypothesis are "closer" to each other:  
 $\theta = 1$  is closer to  $\theta = 2$  than  $\theta = 10$

We mentioned one useful feature: we have a concept of which hypotheses are "similar": those which are **closer** in parameter space. \_\_\_\_\_

This is the **space** we're exploring, as we try to move **downhill**.

We've already used this fact! When normalizing towards the previous hypothesis,  $\theta_{\text{old}} = 0$ , we minimized  $\|\theta - \theta_{\text{old}}\|$ .

#### Clarification 94

Pay attention to your **axes**!

Sometimes, we're doing a 2-D or 3-D plot of  $J$ , and our inputs are  $\theta_k$ . Other times, we're plotting hypothesis  $h$ , with our input axes  $x_i$ .

These two plots could have the same surface, but they **represent** completely different things.

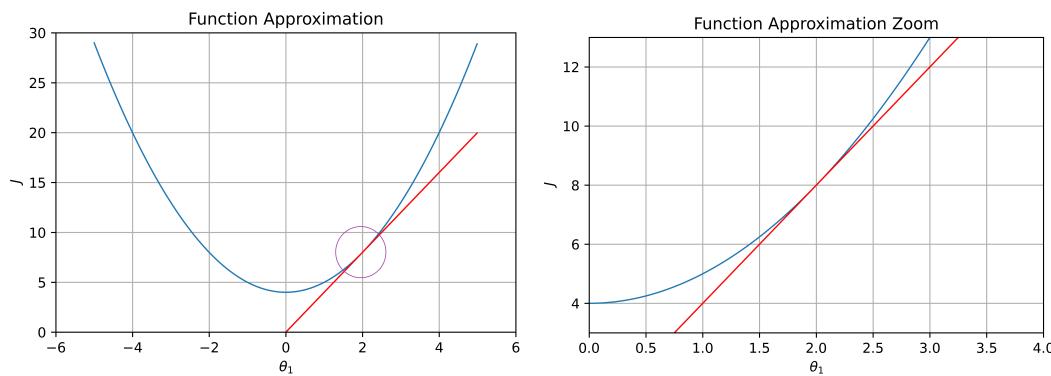
## 3.1 Gradient Descent in One Dimension

### 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** a smooth function as a straight line.



It looks more like a line as we zoom in: hence the **local** approximation.

#### Concept 95

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.

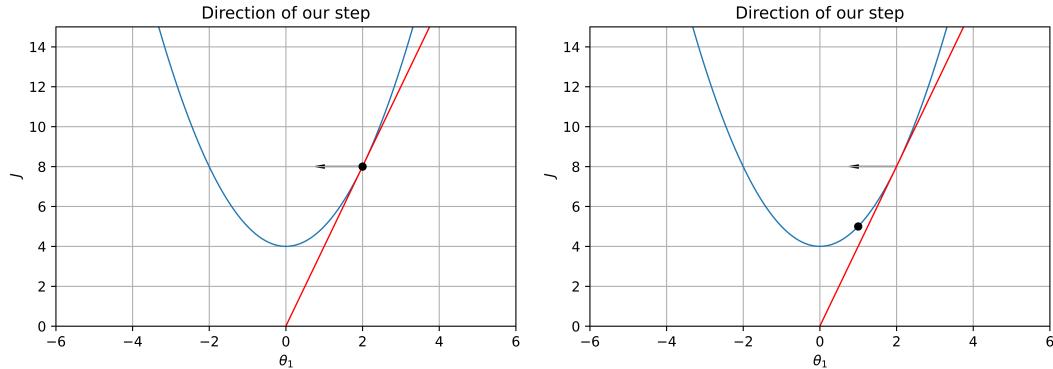
Well, what does our slope tell us? It tells us:

Because the best one might be expensive to find this way!

- How quickly  $J$  changes
- Whether it **increases** or decreases as we change  $\theta$

That second one tells us *how to change  $\theta$* : 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 96

In **1-D**, you can use the **derivative** to **optimize** our function  $J$ .

The **derivative** tells us how to immediately adjust  $\theta_i$  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  $\theta$  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 97**

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  $\epsilon$ . 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 98**

If a sequence  $S$  **converges** to  $m$ , then for all  $\epsilon > 0$ , we can say

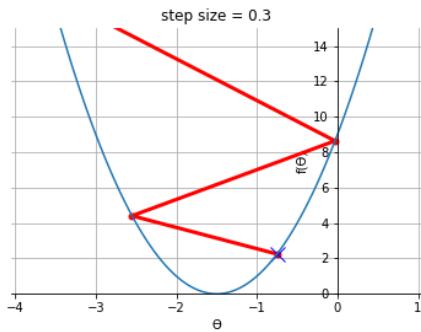
$$|m - s_i| < \epsilon \text{ for all } i > k \quad (3.8)$$

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.

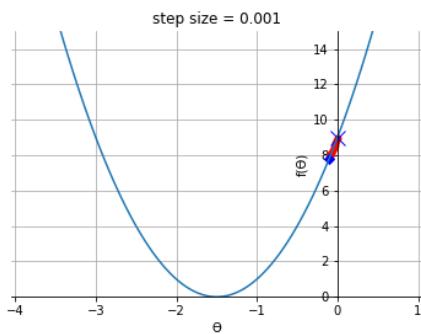
Remember, if it **diverges**, it never **approaches** a single value!



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!

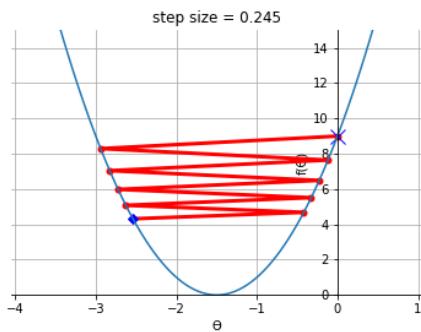
If they're too **small**, you might **converge** too slowly: it'll take way **too long** to make progress.

**Converging** means it successfully **approaches** an answer!



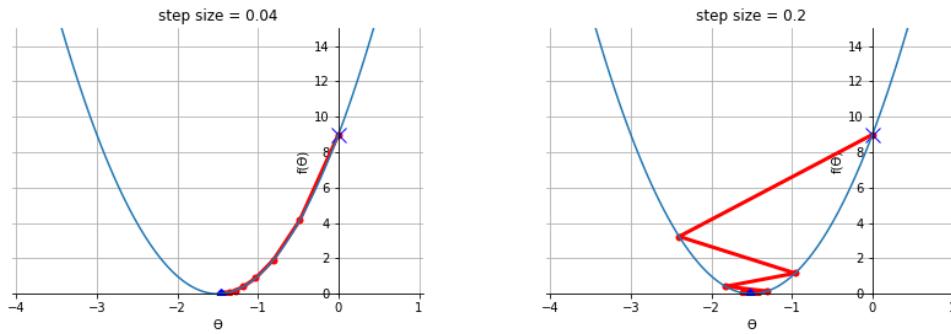
Our step size is too small: this is going to take too long!

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 99

Using the **wrong** step size can cause:

- Slow convergence
- Strong Oscillation
- Divergence

Which is why we **adjust** the step size using  $\eta$ .

### 3.1.6 Step size $\eta$

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**),  $\eta$ .

So, we can rewrite our **change** in  $\theta$  as:

$$\Delta\theta = -\eta \frac{dJ}{d\theta} = -\eta J'(\theta) \quad (3.9)$$

**Definition 100**

Our step size parameter  $\eta$ , or **eta**, **scales** how large each of our optimization steps are.

If  $\eta$  is bigger, we might **learn** faster, but we also risk **diverging**.

Different values of  $\eta$  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  $i^{\text{th}}$  **data point** with  $x^{(i)}$ . We'll reuse this **notation**.

**Notation 101**

Here, we're changing  $\theta$  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  $i^{\text{th}}$  data point.

In this case, we'll do  $\theta^{(t)}$ : the value of  $\theta$  after  $t$  **steps** are taken.

Earlier, we **introduced**  $\theta_{\text{old}}$  and  $\theta_{\text{new}}$ : 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 102**

In **1-D, Gradient Descent** is implemented as follows:

At each time step  $t$ , we **improve** our hypothesis  $\theta$  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:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \cdot \left( \frac{dJ}{d\theta} \Big|_{\theta=\theta_{\text{old}}} \right)$$

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  $\theta$  **isn't changing** much:  $|\Delta\theta| < \epsilon$ , for example.
  - If our  $\theta$  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  $\eta$ , we can guarantee convergence!

#### Theorem 103

##### Gradient Descent Convergence Theorem

We want to optimize function  $J$ . If  $J$  is

- Smooth enough
- Convex

And

- $\eta$  is small enough

Then gradient descent **will** converge to the **global minimum**!

- ~~~~~
- "Small enough" seems vague, but it basically means, "if the first **two statements** are true, then there **exists** a choice of  $\eta$  that converges."

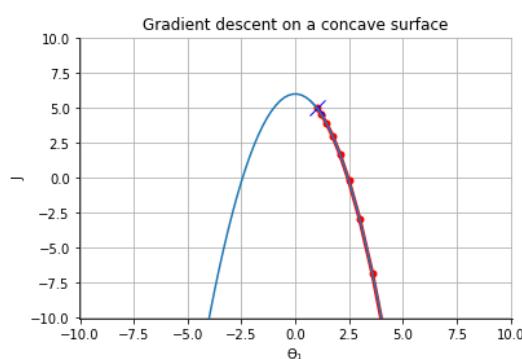
Or, if your  $\eta$  is too **big**, you can keep trying **smaller** ones, until it works.

This is amazing! We can **guarantee** a best solution in some cases!

### 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 104**

If our function  $J$  is **concave**, then our result will not **converge**: it will continue to **decrease** more and more indefinitely.

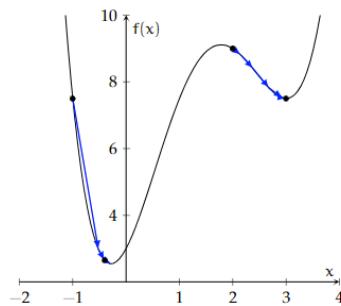
So, for future problems, let's assume it **doesn't** go down forever: if it was, then there is no best solution! We don't have a **valid** problem.

### 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 105**

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 **tiny** amount  $\epsilon$  to the input, the output will **increase**.

So, we **won't** necessarily end up with the **global** minimum, even with a *small*  $\eta$ .

This shows that **initialization matters**!

**Definition 106**

**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.

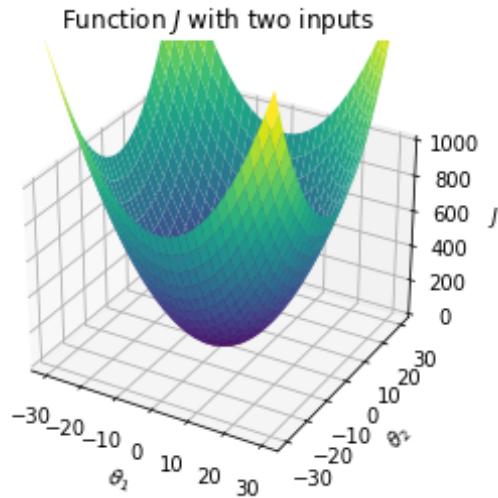
**Concept 107**

**Gradient descent** often finds **local** minima near the initialization, not necessarily **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,  $\theta_1$  and  $\theta_2$ , 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 view our **approximation** in 2-D: \_\_\_\_\_

- First, we could turn it back into 1-D: we remove one variables.

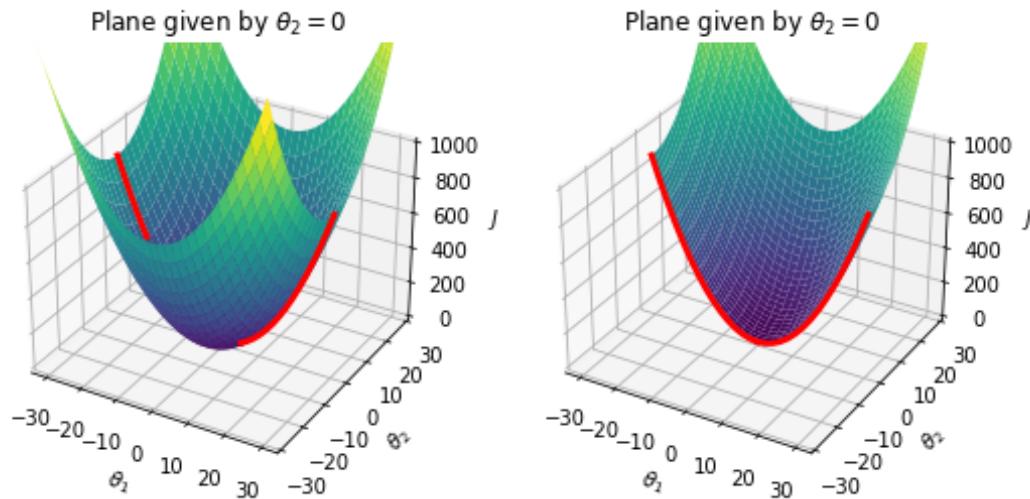
We do this by turning one variable constant: take  $\theta_2 = 0$ . Now, we have one free variable  $\theta_1$ . Same as 1-D.

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 108

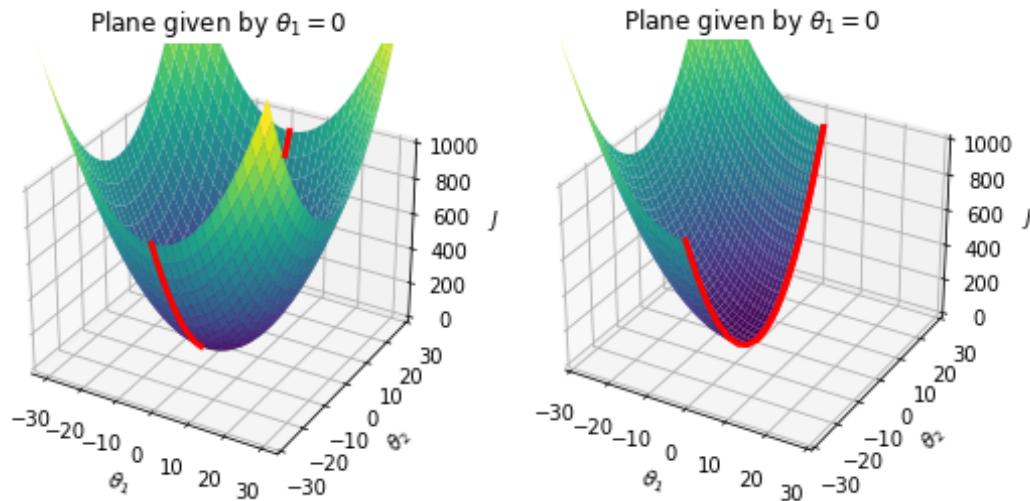
We can **reduce** the number of **variables** we have to work with, by holding some of them **constant**. That way, we have a **simpler** problem to work with.

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  $\theta_2$ .



We can slice along the other axis as well!

Along each **axis**,  $\theta_1$  and  $\theta_2$ , 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 109

If we have **two input variables** (a 2-D problem), we can **approximate** our surface as a **plane** if we **zoom** in enough.

If you look closely enough at any smooth surface in 3D space, it will look roughly "flat".

**Example:** The earth tends to look flat up close, even though it's a sphere.

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**.

$$\Delta\theta_1 = \frac{\partial J}{\partial\theta_1} \quad (3.12)$$

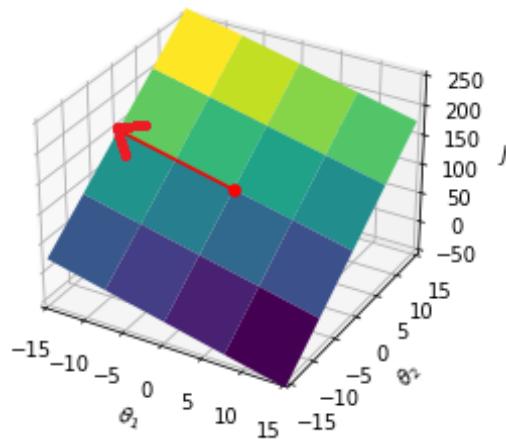
Note that we switched to **partial** derivatives, because we have **multiple** input variables  $\theta_i$ .

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  $\theta_1$  **axis**.

Movement in  $\theta_1$  on  $J$



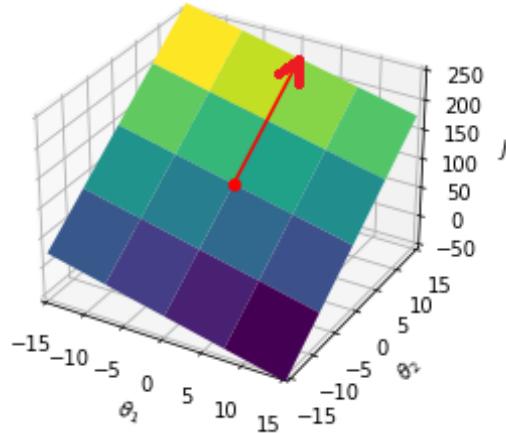
We can move along  $\theta_1$  just like on a line.

What if we treated this as a 1-D problem for the **other** variable,  $\theta_2$ ?

$$\Delta\theta = -\eta \begin{bmatrix} 0 \\ \partial J / \partial \theta_2 \end{bmatrix} \quad (3.14)$$

With this equation, we would be **moving** along the  $\theta_2$  axis.

Movement in  $\theta_2$  on  $J$



We can do the same with  $\theta_2$ .

Why not move in **both** directions **at once**? We can **combine** our two derivatives: we'll add up our two steps.

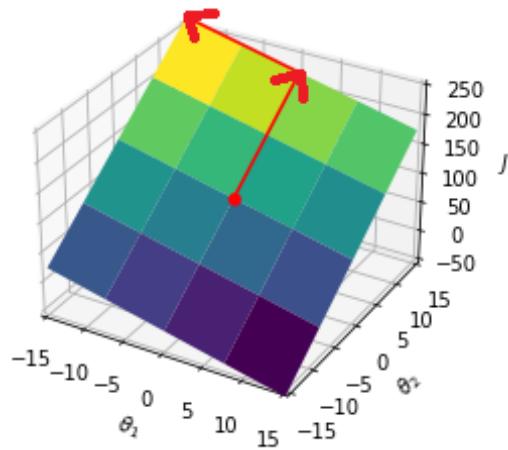
**Linearity** means that I can **add** them up without anything **weird** happening.

$$\Delta\theta = -\eta \begin{bmatrix} \partial J / \partial \theta_1 \\ 0 \end{bmatrix} - \eta \begin{bmatrix} 0 \\ \partial J / \partial \theta_2 \end{bmatrix} \quad (3.15)$$

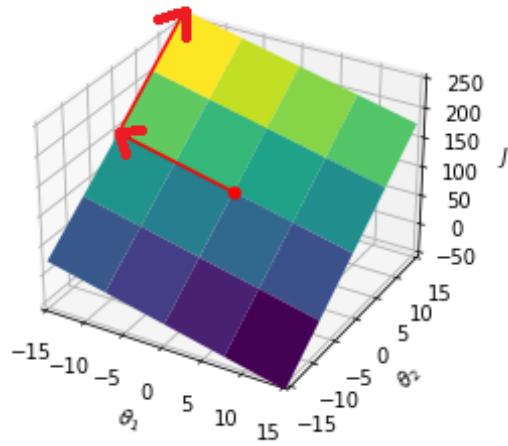
The relevant linearity rule:  $L(x + y) = L(x) + L(y)$ . In other words: taking two separate steps is the same as one big step.

These can be combined because we're treating our function as a **flat** plane: if I move in the  $\theta_1$  direction first, it doesn't change the  $\theta_2$  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 110**

In **2-D**, you can optimize your function  $J$  using this rule:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \begin{bmatrix} \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \end{bmatrix}$$

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  $\theta_1$  direction than  $\theta_2$ .

So, in that case, your step will move more in the  $\theta_1$  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  $\theta$  - 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  $\theta_k$  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**.

A hyperplane is just the equivalent of a plane in a higher dimensional space.

**Key Equation 111**

In **n-D**, you can optimize your function  $J$  using this rule:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \begin{bmatrix} \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \\ \vdots \\ \frac{\partial J}{\partial \theta_d} \end{bmatrix}$$

Using  $\theta_{\text{old}}$

In 3-D space, a plane fills one "slice" of 2-D space. In 4-D space, the hyperplane fills up one "slice" of 3-D space.

This is our **generalized gradient descent** rule.

**3.2.4 The Gradient**

We call this **gradient** descent because that right term we just invented **is** the gradient!

**Definition 112**

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 113**

The **gradient descent** rule can be generally written as:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \nabla_{\theta} J(\theta_{\text{old}})$$

$\theta_{\text{old}}$  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)  $\theta$ ? 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 114**

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**!

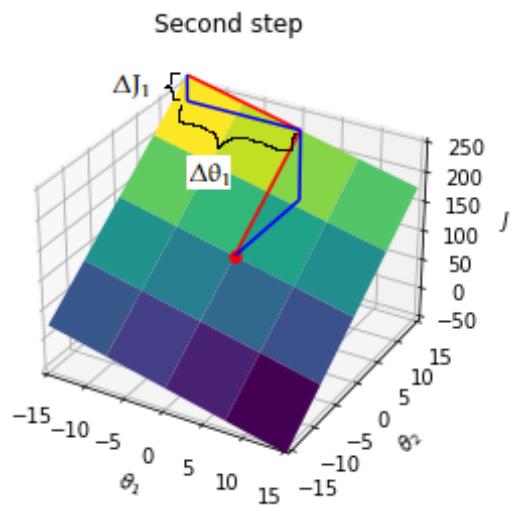
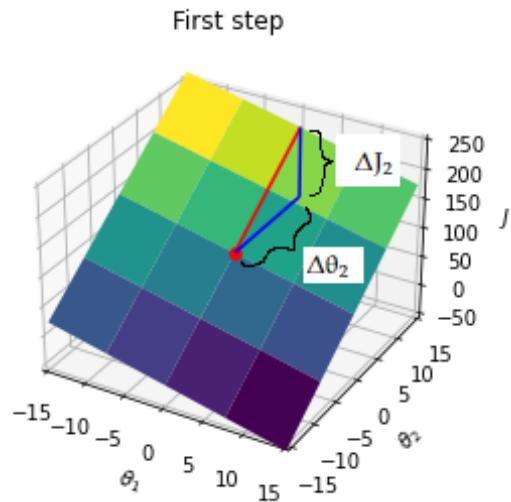
This is useful! We can **break down** any direction into the part that **affects** our function  $J$ , and the part that **doesn't**: \_\_\_\_\_

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 (planes are simple, which makes this possible!):



You can add up the results of our two steps:  $\Delta J_2$  and  $\Delta J_1$ .

$$\Delta J \approx \Delta J_1 + \Delta J_2 \quad (3.17)$$

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**

$(\Delta\theta_i$  matches  $i^{\text{th}}$  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  $\Delta J$ . How do we maximize a **dot product**?

- A dot product  $a \cdot b$  is maximized when the directions of  $a$  and  $b$  are **the same**!
- The direction of the gradient was given to us by our calculations.
- So, we want  $\Delta\theta$  to match the **gradient**! That way, they're in the same direction.

Maximizing this dot product means maximizing  $\Delta J$ , which is our goal.

So, we just demonstrated that the **gradient** gives us the **best** direction for  $\Delta\theta$ .

So, all we have to do is to **flip** the sign to **minimize**  $\Delta J$ .

And so, gradient descent is complete!

#### Concept 115

The **gradient**  $\nabla J$  is the **direction of greatest increase** for  $J$ .

That 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 2-D vector,  $\Delta\theta$  - no assurances about how good or bad it is.

Currently, our vector **components** are based on  $\theta_1$  and  $\theta_2$ . But, it can be useful 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  $\Delta B$ )
- Same height: the direction with no effect,  $\hat{u}_{none}$  (magnitude  $\Delta N$ )

As we established before, these two directions are perpendicular on the plane.

$$\Delta\theta = \underbrace{\hat{u}_{best}}_{\text{Full effect on } J} + \underbrace{\hat{u}_{none}}_{\text{No effect on } J} = \Delta B * \hat{u}_{best} + \Delta N * \hat{u}_{none} \quad (3.20)$$

So, all of the change in  $J$  just comes from  $\hat{u}_{best}$ . We **don't care** about the other direction!

What about higher dimensions?

If we have  $k$  more dimensions, we just include  $k$  more unit vectors which add nothing to  $\Delta J$ .

#### Concept 116

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  $\Delta J$  using **only that component**.

Thus,  $\hat{u}_{best}$  gives us the "direction of greatest increase": if we rotate our vector, we replace some of  $\hat{u}_{best}$  with  $\hat{u}_{none}$ .

- In which case, we're losing some  $\Delta J$ . So, we don't want to change direction like that.

However, this is the same kind of behavior as the dot product:

$$\Delta J \approx \Delta\theta \cdot \nabla_\theta J \quad (3.21)$$

- When we do the dot product  $a \cdot b$ , we take the **projection** of  $a$  onto  $b$ : we only take the component of  $a$  in the same direction as  $b$ .
- In this case, we only include the **component** of  $\Delta\theta$  which matches  $\nabla_\theta J$ .

So, let's compare the two.

- The  $u_{best}$  component of  $\Delta\theta$  is the only part that matters for  $\Delta J$ .
- The  $\nabla_\theta J$  component of  $\Delta\theta$  is the only part that matters for the **dot product**, which equals  $\Delta J$ .

They're the same!

### 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)  $\theta$ , 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.

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

Our gradient, according to **matrix derivative** rules, is

$$\nabla_{\theta} J(\theta) = \frac{2}{n} \tilde{X}^T (\tilde{X}\theta - \tilde{Y}) \quad (3.23)$$

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  $\theta_{\text{old}}$  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  $\theta_0$  as **separate** from our other data points, because of **regularization**: remember that it **doesn't** apply to  $\theta_0$ .

For  $\theta$ :

$$\nabla_{\theta} J_{\text{ridge}}(\theta, \theta_0) = \frac{2}{n} \sum_{i=1}^n \left( (\theta^T x^{(i)} + \theta_0) - y^{(i)} \right) x^{(i)} + 2\lambda\theta \quad (3.25)$$

For  $\theta_0$ :

$$\frac{\partial J_{\text{ridge}}(\theta, \theta_0)}{\partial \theta_0} = \frac{2}{n} \sum_{i=1}^n \left( (\theta^T x^{(i)} + \theta_0) - y^{(i)} \right) \quad (3.26)$$

Notice that we used a **gradient** for our vector  $\theta$ , but since  $\theta_0$  is a single variable, we just used a **simple derivative**!

### Concept 117

The **gradient**  $\frac{dJ}{d\theta}$  must have the **same shape as  $\theta$** : this shape-matching is why we can easily **subtract** it during gradient descent.

$$\underbrace{\theta_{\text{new}}}_{(d \times 1)} = \underbrace{\theta_{\text{old}}}_{(d \times 1)} - \eta \underbrace{\nabla_{\theta} J(\theta_{\text{old}})}_{(d \times 1)}$$

The derivative  $\frac{dJ}{d\theta_0}$  is based on  $\theta_0$ , a constant. So, the shape must be  $(1 \times 1)$ .

$$\theta_0^{(t)} = \theta_0^{(t-1)} - \eta \frac{dJ}{d\theta_0} \Big|_{\theta_0=\theta_0^{(t-1)}}$$

### 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

A derivative is just a 1-D gradient, after all!

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \quad (3.27)$$

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  $\delta$ .

### Concept 118

When **approximating** the derivative, we can choose a **small** finite width to measure, called  $\delta$ , so that

$$\frac{df}{dx} \approx \frac{f(x+\delta) - f(x)}{\delta}, \quad \delta \ll 1$$

So, let's **extend** that to the **gradient**:

$$\nabla_{\theta} J = \begin{bmatrix} \partial J / \partial \theta_1 \\ \partial J / \partial \theta_2 \\ \vdots \\ \partial J / \partial \theta_d \end{bmatrix} \quad (3.28)$$

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

$$J'(\theta) \approx \underbrace{\frac{J(\theta + \delta) - f(\theta)}{\delta}}_{\text{Not correct, but closer}} \quad (3.29)$$

This isn't quite right, because a **scalar**  $\delta$  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  $\delta$  to  $\theta_1$ : 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  $\theta_k$  term.

$$\Delta\theta = \begin{bmatrix} \Delta\theta_1 \\ \Delta\theta_2 \\ \vdots \\ \Delta\theta_d \end{bmatrix} \quad (3.30)$$

We want most terms **unchanged**, so we'll **add 0** to each of them, and we'll add  $\delta$  to the one term we want to **edit**.

$$\Delta\theta = \begin{bmatrix} \delta \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \quad (3.31)$$

We'll **create** one of these vectors for each **dimension**. We'll give them a special **name**:  $\delta_k$ , for the  $k^{\text{th}}$  dimension.

$$\delta_1 = \begin{bmatrix} \delta \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix} \quad \delta_2 = \begin{bmatrix} 0 \\ \delta \\ \vdots \\ 0 \\ 0 \end{bmatrix} \quad \delta_{d-1} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \delta \\ 0 \end{bmatrix} \quad \delta_d = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \delta \end{bmatrix} \quad (3.32)$$

Finally, we'll **divide** by  $\delta$ . We have what we need for our full equation:

### Key Equation 119

In order to **computationally find the gradient**, you need to find the **partial derivative** for each term  $\theta_k$ .

$$\frac{dJ}{d\theta_k} \approx \frac{J(\theta + \delta_k) - J(\theta)}{\delta}$$

Where

- $\delta$  is a small positive number
- $\delta_k$  is the  $(d \times 1)$  **column vector** with a  $\delta$  in the  $k^{\text{th}}$  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 **requirement**: our loss function has to be **smooth** and **differentiable**. If it isn't, we can't take the **gradient** of it.

**Concept 120**

**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

Some **benefits** of gradient descent come from the fact that GD **gradually** improves:

- You can pause early to check progress, or quit early if your model is good enough. We save on computation time.

But we can improve this feature: currently, we use a **sum** of all data points to get our gradient. Meaning, we have to compute all of our data before we take a single step.

### 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 for using less data, 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 121

**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. Randomly choosing data points is more unpredictable than using all of our data.

The random selection makes the data "look different" each time, so it's hard to perfectly match it.

**Concept 122**

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**.
  - 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 123**

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

In this case, though, we're often **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

Why? Because of our third condition,  $\sum_t \eta(t) < \infty$ . If  $\eta(t)$  decreases too slowly, this sum goes to infinity, and our GD algorithm might **diverge**.

# 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: sometimes, we just have a different kind of question.

Often, it's more useful to, rather than give numeric values, instead sort things into **categories**, or what we will call **classes**.

#### Definition 124

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 125

**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:** Suppose you want to classify different **animals** as a bird, a mammal, or a fish. You are given (as input) 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 126**

**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 you choose. Do you want to compare dogs vs. cats, mammals vs. fish, color of fur? What goes in different classes, or the same?
  - You can change units (lb vs kg), but you don't "decide" how the real numbers work.

**Concept 127**

**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 128

**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 129

**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.

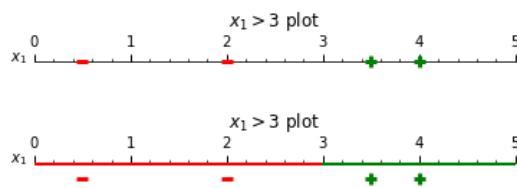
### 4.2.1 1-D Linear Classifiers

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.

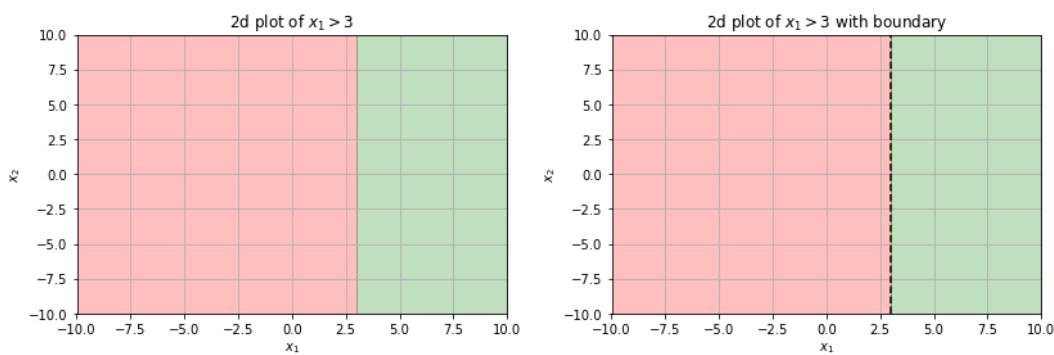
We could also call it  $\theta_0$ , in the spirit of our  $\theta$  notation for parameters.

$$x_1 + \theta_0 > 0 \quad (4.5)$$

### 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.

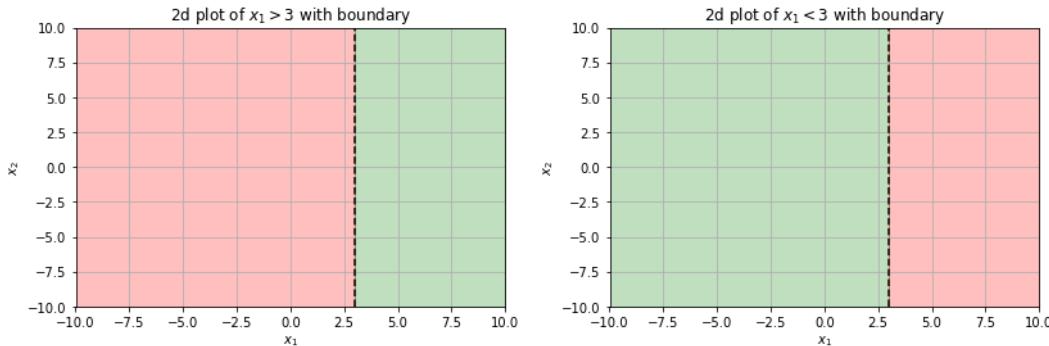
We'll omit the data points for now.



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**.

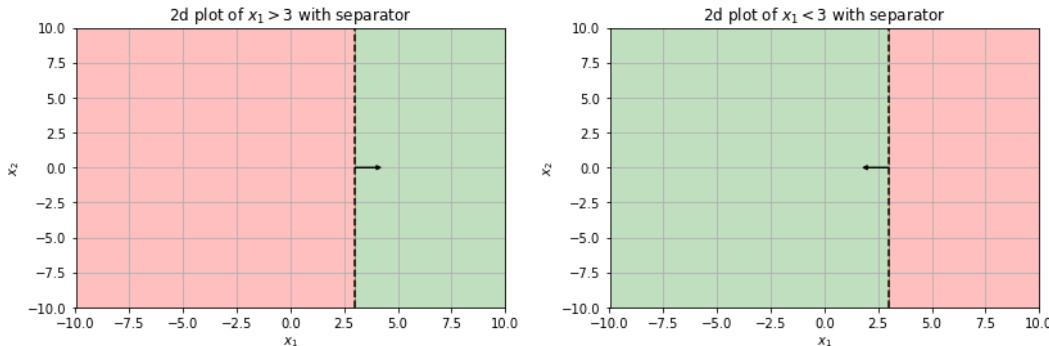
A vertical line is missing some information, though: compare  $x_1 > 3$  and  $x_1 < 3$ :



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 130

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 131

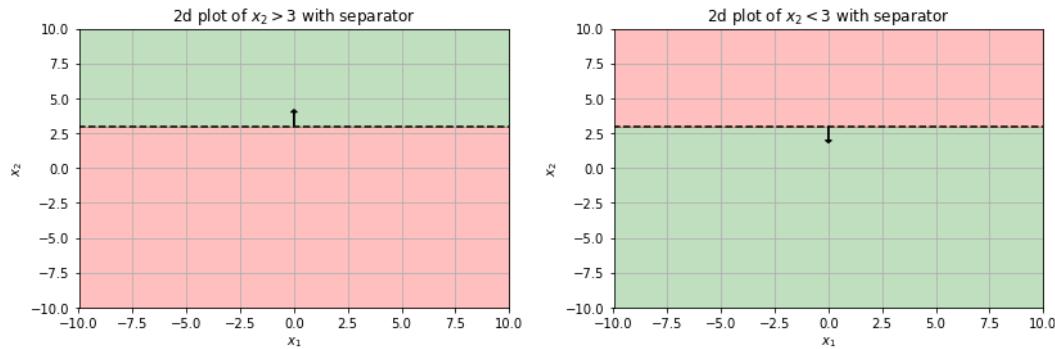
We can define our **1-D separator** using

- 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?

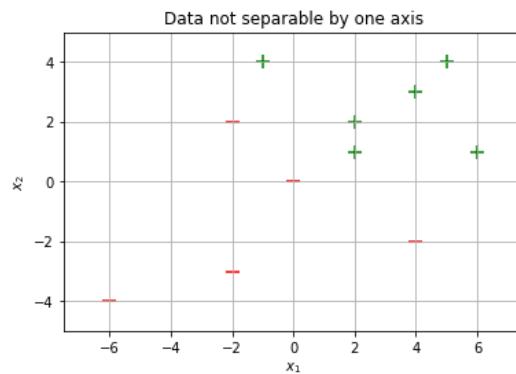


Instead of having a vertical separator, we have a **horizontal** one.

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?

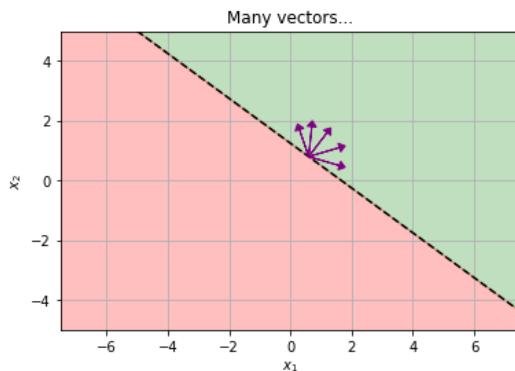
Just looking at our example, we might wonder, "well, if we can use **vertical** lines or **horizontal** lines, can't we just use a line in **another** orientation?"

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. The opposite side is just  $-\hat{n}$ .
- 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. This will be important later!

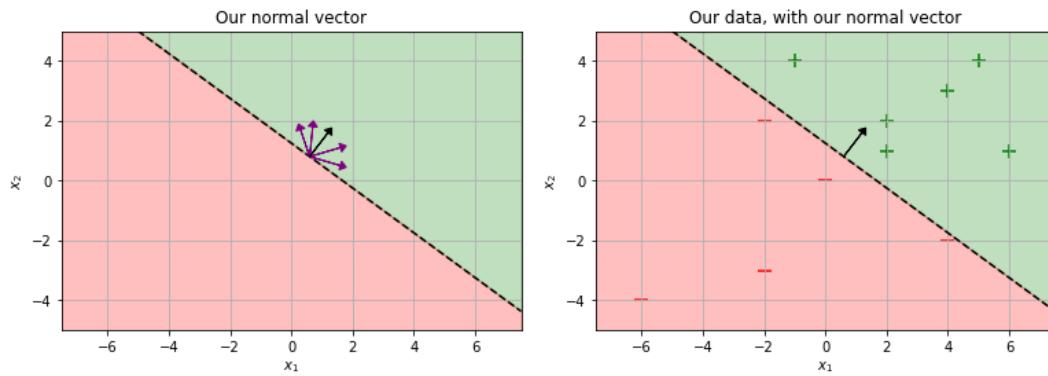
So, we have a **unique** vector that tells us which side is **positive**. Let's go with that!

### Concept 132

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  $\theta$ . We need **two** different  $\theta_k$ : one for the  $x_1$  component, another for the  $x_2$  component.

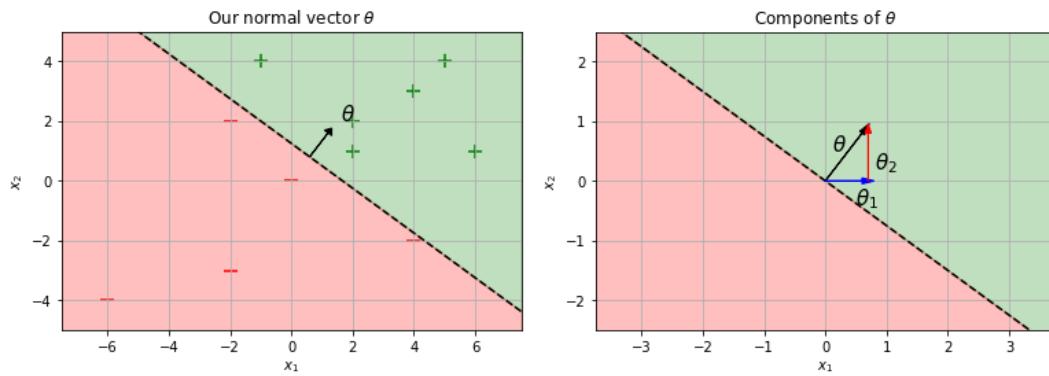
So, we'll use that.

### Notation 133

The vector  $\theta$  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:

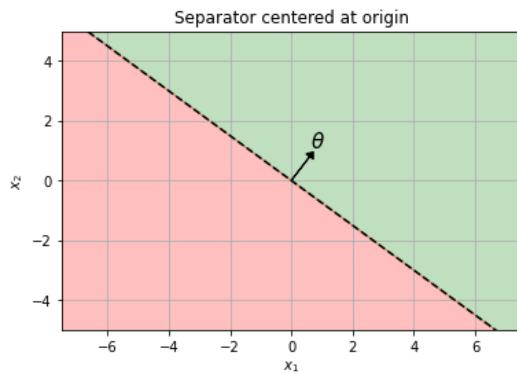


$\theta$  is our normal vector!

Nice work so far. The next question is: how do we describe this separator **mathematically**?

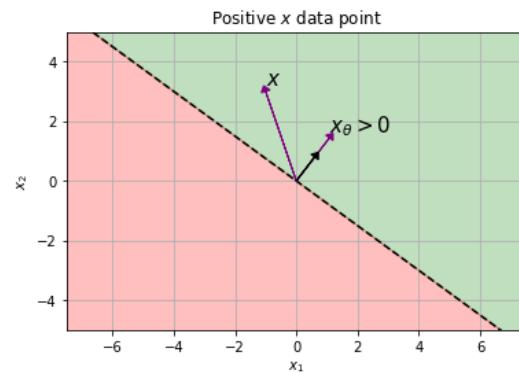
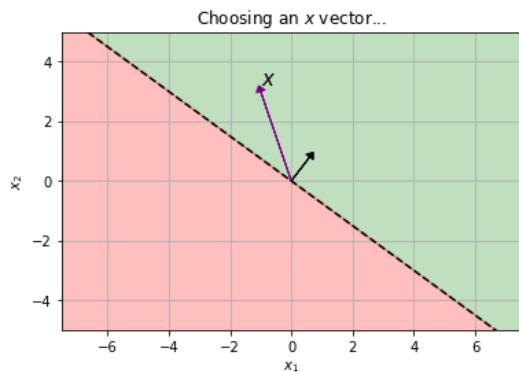
#### 4.2.5 2D Separator - Matching components

As always, we'll **simplify** the problem to make it more manageable: for now, we'll assume our **separator** is centered at the **origin**.

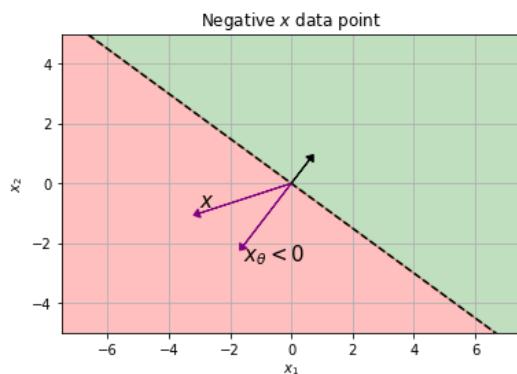


So, we have our vector,  $\hat{n}$ . As we mentioned above, anything on the **same** side as  $\hat{n}$  is **positive**, and anything on the **opposite** side is **negative**.

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  $\theta$  direction.



This vector has a **negative** component in the  $\theta$  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 134**

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 135**

You can use the **dot product** between non-unit vectors to measure their "similarity" **scaled by their magnitude**.

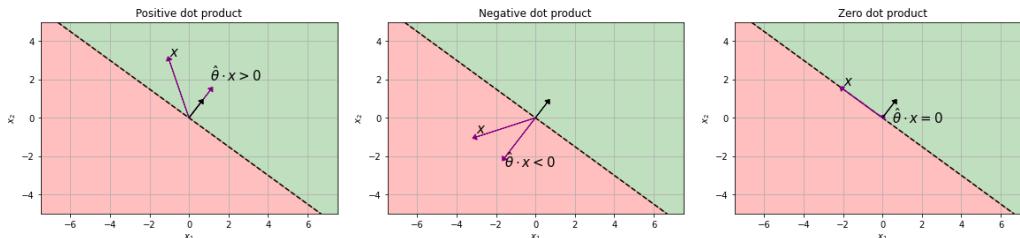
If two vectors are more **similar**, they have a **larger** dot product.

- If the vectors are **less** than  $90^\circ$  apart, they are more similar: they will share a **positive** component:  $\vec{a} \cdot \vec{b} > 0$
- If the vectors are **more** than  $90^\circ$  apart, they will share a **negative** component:  $\vec{a} \cdot \vec{b} < 0$
- If they are **perpendicular** ( $90^\circ$ ) 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  $\theta$ , our **normal vector**.

So, if a point has a **positive** dot product, it is on the **same side** as  $\theta$ , 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 136

For a **linear separator** centered on the **origin**, we can do **binary classification** using the hypothesis

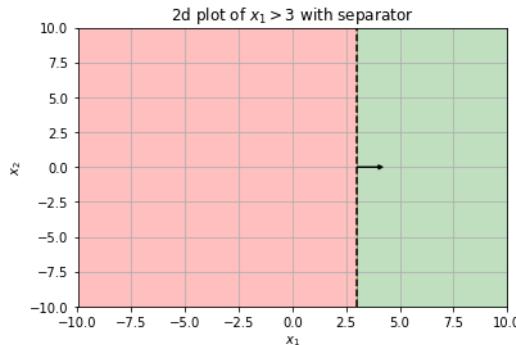
$$h(x; \theta) = \text{sign}(\theta \cdot x) = \begin{cases} +1 & \text{if } \theta \cdot x > 0 \\ -1 & \text{otherwise} \end{cases}$$

- Note that we assigned  $-1$  if  $\theta \cdot x = 0$ .
- This is an **arbitrary** convention: we could also have assigned  $+1$  for  $\theta \cdot x = 0$ .
- All that matters is to have a decision, and be consistent about it.

### 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!

$$x_1 > 3 \iff x_1 - 3 > 0 \quad (4.6)$$

So, instead, we could just add a constant to our expression, which we will call  $\theta_0$ .

We'll also switch out  $\theta \cdot x = \theta^T x$ .

### Key Equation 137

A general **linear separator** can do **binary classification** using the hypothesis

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

- Again, we assigned  $-1$  if  $\theta \cdot x = 0$ .
- Again, this choice is **arbitrary**. Consistency is what matters most.

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 138

For **binary classification**,  $\theta$  and  $\theta_0$  entirely **define** our **linear separator**.

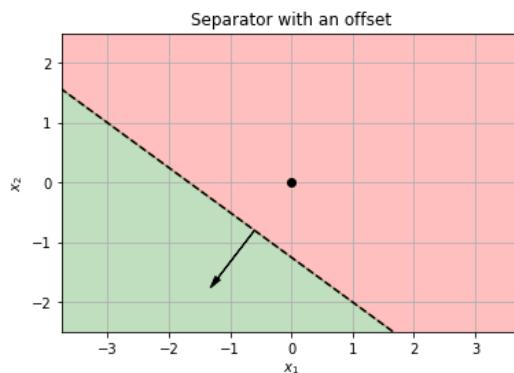
- $\theta$  gives us the **orientation** of our line.
- $\theta_0$  **shifts** that line around in **space**.

### 4.2.9 How does the offset affect our classifier?

So, how exactly does our offset  $\theta_0$  affect our **classifier**? Well, we mark our classifier with our **normal vector** and the **boundary**.

Our **normal vector** is entirely captured by  $\theta$ : it's unchanged by  $\theta_0$ .

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: if your formula is positive, you get +1. If your formula is negative, you get -1.

The **boundary** line is between positive and negative: it's at **zero**.

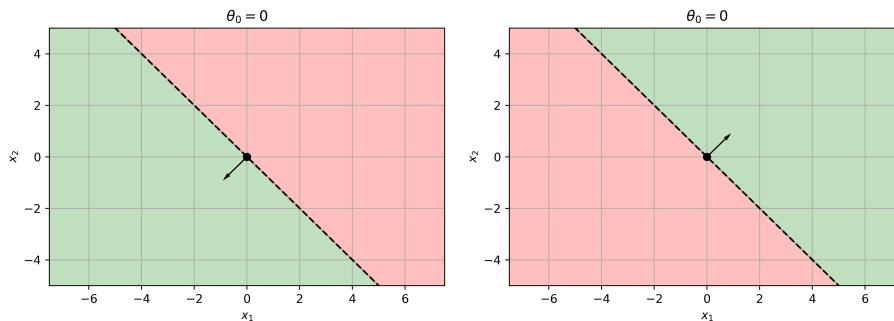
$$\theta^T x + \theta_0 = 0 \iff \theta^T x = -\theta_0 \quad (4.7)$$

We'll break the effects of  $\theta_0$  into three cases: \_\_\_\_\_

For each, we'll show a boundary, and a **flipped** version of that boundary.

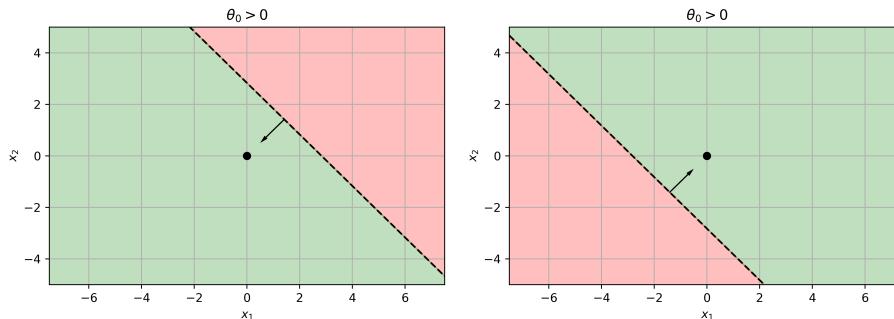
Note: the below statements are true no matter what  $\theta$  we choose!

- 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.

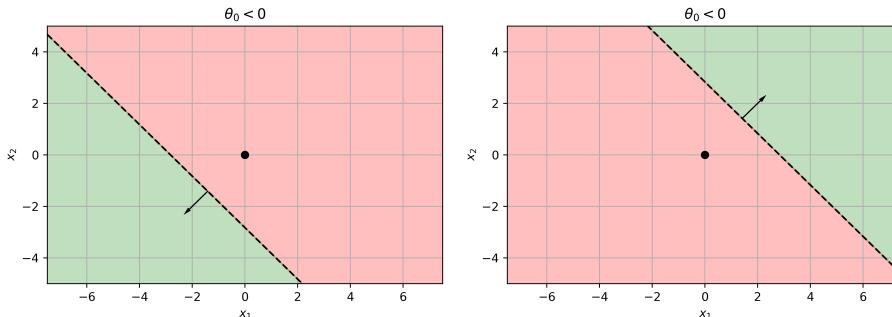
- If  $\theta_0 > 0$ , then the **origin** is in the **positive** region.
  - That means the positive region is "larger": some space that was negative, is positive.
  - The boundary line has moved in the  $-\theta$  direction.



If we have a **positive** constant, it's "easier" to get a positive **result**: more positive space.

- If  $\theta_0 < 0$ , then the **origin** is in the **negative** region.

- That means the positive region is "smaller": some space that was positive, is negative.
- The boundary line has moved in the  $+θ$  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 139

The **sign** of our  $\theta_0$  and the **direction** we move away from the origin are **opposite**.

If  $\theta_0 > 0$  (positive), our boundary moves in the  **$-θ$  direction**.

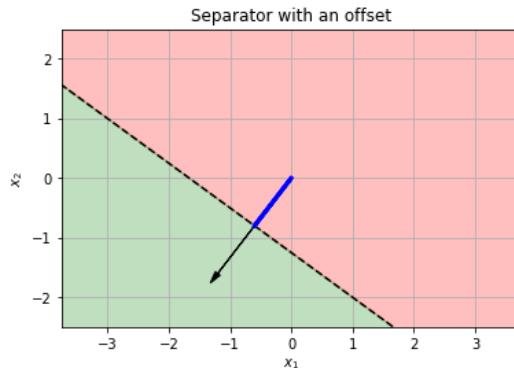
If  $\theta_0 < 0$  (negative), our boundary moves in the  **$+θ$  direction**.

This gives us a general idea of how the offset affects it, but what is the **exact** effect of  $\theta_0$  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!

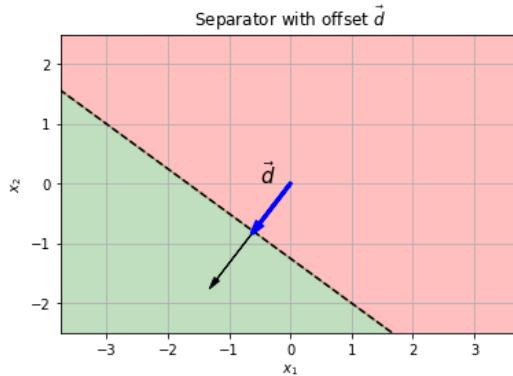
#### 4.2.10 Distance from the Origin to the Plane



Notice that the **shortest** path from the origin to the line is **parallel** to  $\theta$ !

So, we can think of our **line** as having been **pushed** in the  $\theta$  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  $\theta$ , 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  $\theta^\top \vec{d}$  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  $\hat{\theta} \cdot \hat{\theta} = 1$ :

$$\|\theta\| d + \theta_0 = 0 \quad (4.13)$$

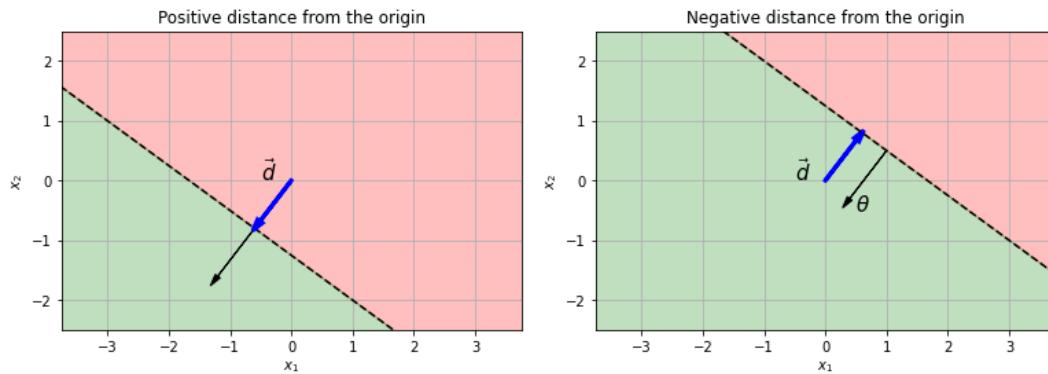
And now, we just solve for d:

**Concept 140**

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  $\theta$ .



Notice, again, that this agrees with our **earlier** thought: the sign of  $\theta_0$  is the opposite ( $-1$ ) of the  $\theta$  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  $\theta$ .

**Key Equation 141**

A general d-dimensional **linear separator** can do **binary classification** using the hypothesis

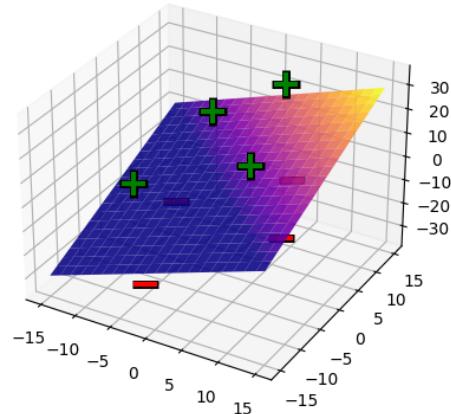
$$h(x; \theta) = \text{sign}(\theta^T x + \theta_0) = \begin{cases} +1 & \text{if } \theta^T x + \theta_0 > 0 \\ -1 & \text{otherwise} \end{cases}$$

Where

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_d \end{bmatrix} \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix}$$

What about how it looks? Well, if we have 3 input variables, our line turns into a **plane**:

Classification in 3D



Notice the green + signs are "above" the plane, while the red - signs are "below" the plane.

Just like with regression, this is when we introduce the **hyperplane**:

**Concept 142**

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  $\theta$  which defines the **orientation** of the hyperplane, and which side is **positive**.

It also has an **offset** term  $+\theta_0$ , that slides it in the  $-\theta$  direction **away** from the origin.

- $\theta_0$  can be any real number, but the shift will always be in the opposite direction.

For any dimensional input, we can use hyperplanes as separators.

#### 4.2.12 IMPORTANT: A difference between regression and classification

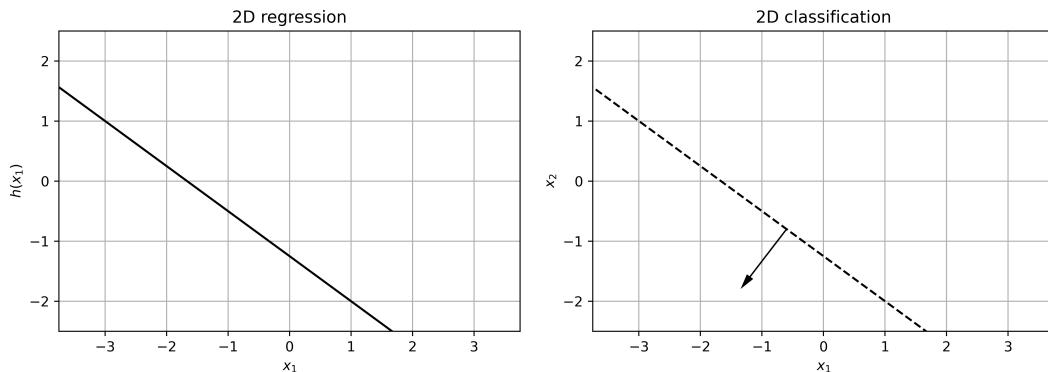
Here is an important misconception that comes up between regression and classification.

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.

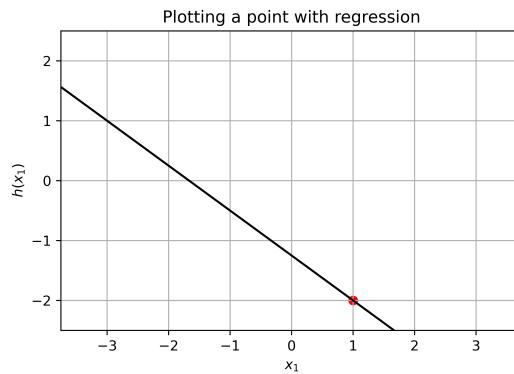
Notice, for example, that the regression plot has **only**  $x_1$ , while the classification plot has  $x_1$  **and**  $x_2$ .

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.

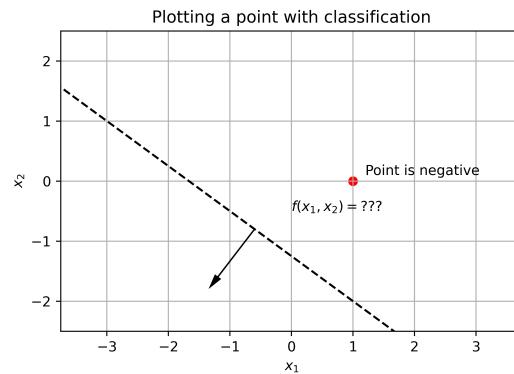
If we think in terms of  $f(x) = \theta^T x + \theta_0$ , we can compare them directly.

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. We only plot values on the line.

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 inputs, and we **don't** get the exact output. We can plot inputs anywhere in space.

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-dimensional** visualization.

### Clarification 143

**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 shows where  $f(x) = 0$ , and the sign of  $f(x)$  elsewhere.

In short:

Regression adds the whole **y number line**, classification only shows  $y = 0$ . This saves **one dimension of space**.



When plotting  $d$  inputs,

- A **regression** plot uses a  **$d+1$**  dimensions ( $d$ -dim hyperplane) to plot: +1 dimension for the **output axis**.
  - **Example:** You have one input dimension,  $d = 1$ . You need to plot the **input and output**: you'll be plotting it on a 2D plane.
  - However, on that 2D plane, you'll plot a **line**: despite existing in 2D space, a line is a 1-d hyperplane.
- A **classification** plot only needs  **$d$**  dimensions (( $d-1$ )-dim hyperplane): we don't need an output axis, because we only plot  **$y = 0$** .
  - **Example:** You have three input dimensions,  $d = 3$ . You'll be plotting every combination of **three inputs** that gives  **$y = 0$** : you'll be plotting it in 3D space.
  - However, in 3D space, you'll plot a **plane**: despite existing in 3D space, a plane is 2-d hyperplane.

**Notation 144**

A  $(d - 1)$ -dimensional hyperplane is a  $d$ -dimensional **separator**.

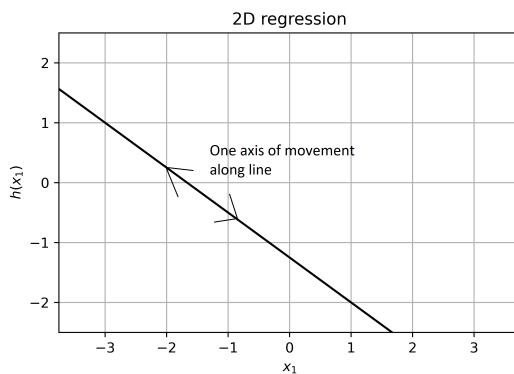
That's because it's **splitting** the  $d$ -dimensional space in half.

- **Example:** A **line** (1-dim) splits the **plane** (2-dim) in half.
- So, it separates the two halves of 2-d space.

Why do we need  $d + 1$  dimensions to plot a  $d$ -dimensional **hyperplane**? Because the hyperplane doesn't fill the whole space.

Here's an example: a **line** in 2-D space is a 1-D **hyperplane**: we have only **one axis** we can move on the line.

It's a 1-d object "embedded" in 2-d space.



Our plot is 2-D, but we can only move along one axis on our line!

- Notice that our line does not fill up the whole space: that's why it's a lower dimension.
- You need 2 dimensions to see **where** the line is, but the line itself is only 1-d.

Because of these differences,  $\theta$  also acts differently:

**Clarification 145**

$\theta$  appears differently in 2-D regression and classification:

- In **2-D regression**,  $\theta$  is the **slope** of the line

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

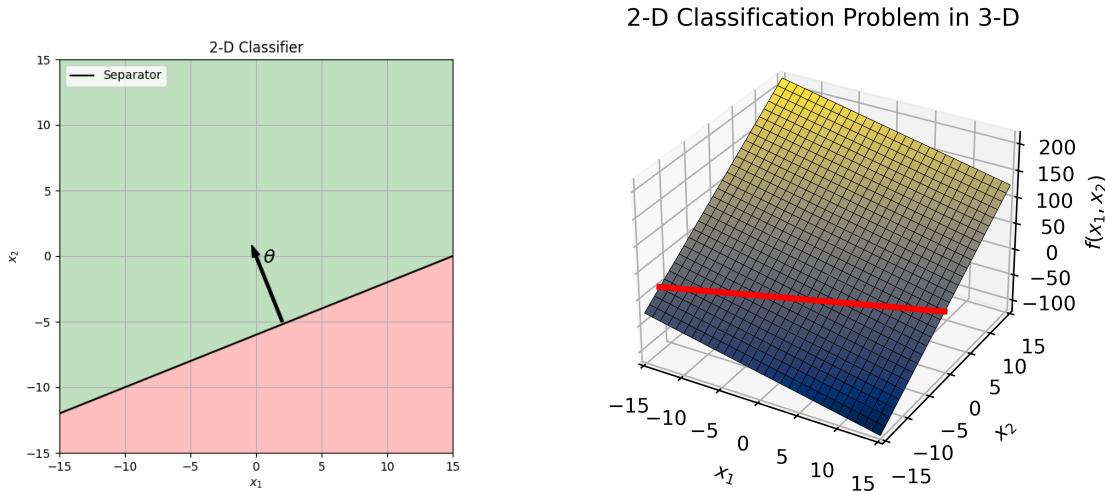
- In **2-D classification**,  $\theta$  is the **normal vector** of the line

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

### 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  $\{-1, +1\}$ .

The below plot "reveals" the 3rd dimension (output of  $h(x)$ ) that the classification plot usually hides.



We can **compare** what we usually see (left plot) to an **alternate** version that shows the 3rd dimension (right plot). These are the **same classifier**!

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).

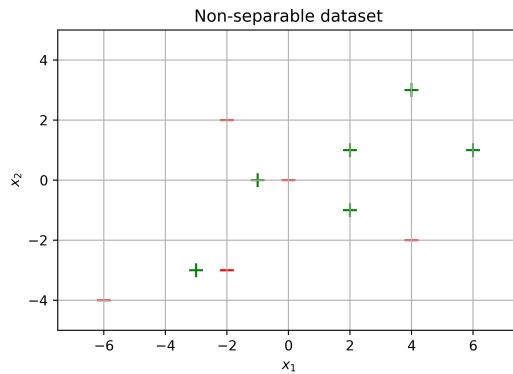
So here, we've done **exactly** that: the **height** is the output of  $h(x)$ .

But, because we don't **care** about the **exact** output in classification, we usually only graph the plane containing the **red line**: where  $f(x_1, x_2) = 0$ .

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 146**

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 that 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  $\mathcal{L}$ :

$$J(\theta, \theta_0) = \frac{1}{n} \sum_{i=1}^n \mathcal{L}(\text{sign}(\theta^T x^{(i)} + \theta_0), y^{(i)}) \quad (4.16)$$

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**.

First, we need to compute the **gradient**.

To be fair, this is true for most possible problems: most of them can't be solved analytically.

$$\nabla_{\theta} J = 0 \quad (4.17)$$

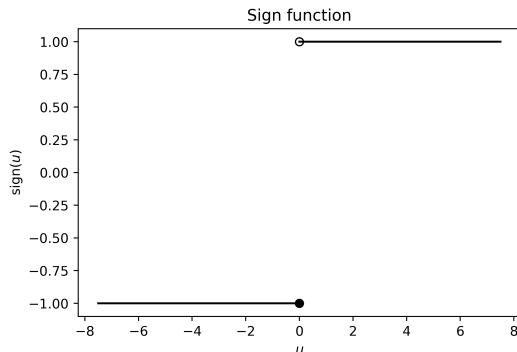
...Well that's not good.

Why not? Because we use our **gradient** to decide **how** to change  $\theta$ .

### 4.3.2 The real problem: $\text{sign}(u)$ is flat

What's going on here? Let's look at the sign function:

If the gradient is 0,  $\theta$  never changes, and we never **improve**  $\theta$  at all!



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.

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.

Remember, parameter space is what we move through as we change our parameter vector  $\theta$ .

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 147

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.

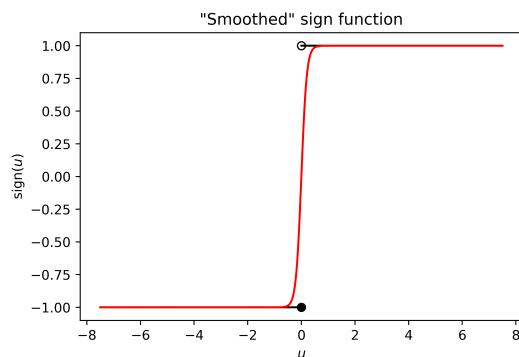
This causes two problems:

- 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 our 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**  $\text{sign}$  with? We like the way  $\text{sign}$  **works** (choosing between two different classes based on a **threshold**), so maybe we want a **smoother** version of it.

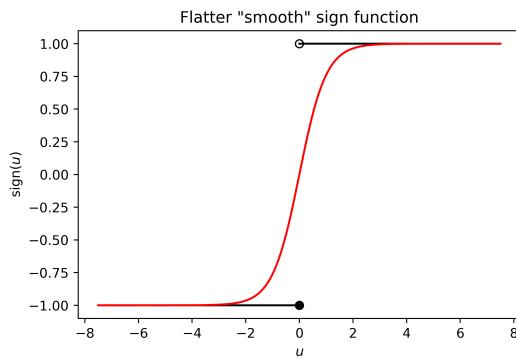


The red line shows a "smoother" sign function, that mostly behaves the same, while solving our problem.

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 148

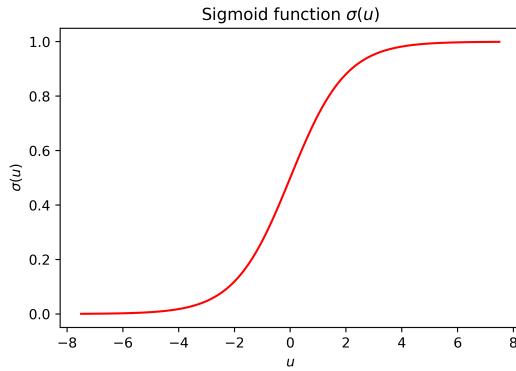
The **sigmoid** function

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

...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 149

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 150

**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:

$$h(x; \theta) = \frac{1}{1 + e^{-(\theta^T x + \theta_0)}}$$

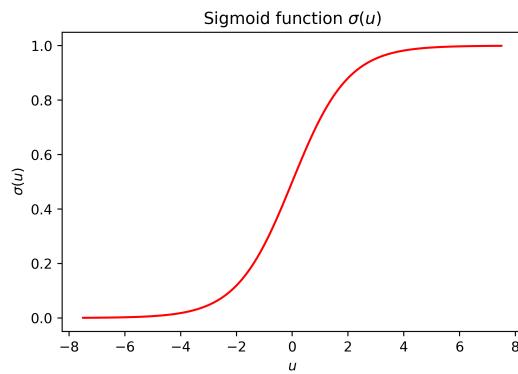
We have a problem, though: **logistic regression** is a... **regression** function. It takes in a real **vector**, and outputs a real **number**:  $\mathbb{R}^d \rightarrow \mathbb{R}$ .

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  $u(x) = 0$ .

What happens to  $\sigma$  if  $u = 0$ ?

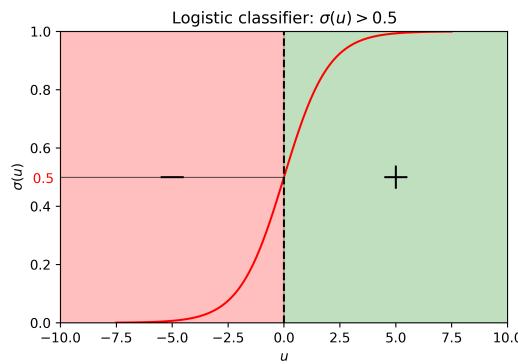


If we go to  $u = 0$  on the x-axis, we find  $\sigma = .5$  on the y-axis.

We get  $\sigma(u = 0) = 0.5$ . So, we could use that as our classification:  $\sigma(u) > 0.5$ .

$$u > 0 \iff \sigma(u) > 0.5 \quad (4.18)$$

If we want to plot the positive and negative regions:



But, we don't necessarily always want to use  $\sigma = 0.5$  as our boundary:

**Example:** Imagine if you wanted to **classify** whether someone needs a **test** for a disease. Classify  $-1$  if we test them,  $+1$  if we don't.

Let's say you got  $\sigma(u) = 0.6$ , so you're only 60% sure they **don't** need it. You'd classify that as  $\sigma(u) > 0.5$ : they're assigned "**no test**".

If the disease is life-threatening, and the test is cheap, then a 40% chance could justify getting the test.

- Whether or not they **should** get that test isn't usually decided by whether the chance is greater than 50%: that's a pretty **arbitrary** number.
- In real life, the **certainty** you want depends on the situation.

We call the **boundary** between positive and negative the **prediction threshold**.

How expensive is the test? How bad is it, if we don't catch the disease now? Etc.

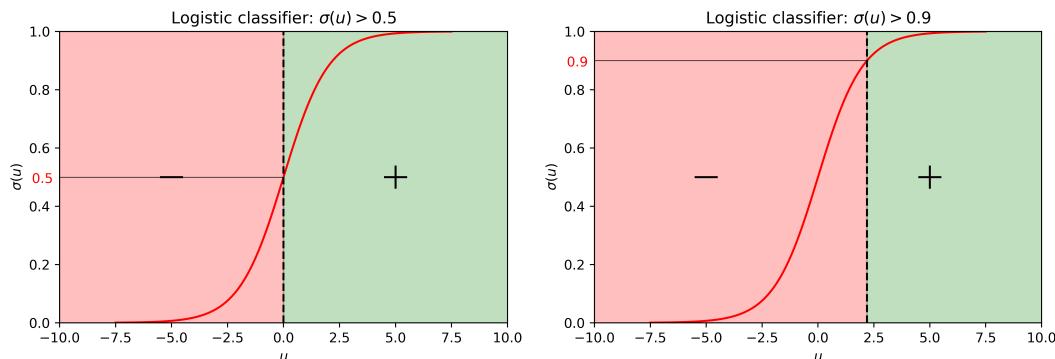
### Definition 151

The **prediction threshold**  $\sigma_{\text{thresh}}$  is the value where you go from **negative** classification to **positive**.

Our **default** value is a threshold of 0.5, but our threshold can be **anywhere** in the range

$$0 < \sigma_{\text{thresh}} < 1$$

**Example:** If  $\sigma_{\text{thresh}} = 0.9$ , we would see:



We switch from a 0.5 threshold (left) to a 0.9 threshold (right).

In this example, more things will be negatively classified.

### 4.3.7 Linear Logistic Classifier

This finally gives us our **linear logistic classifier** (LLC)

**Key Equation 152**

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

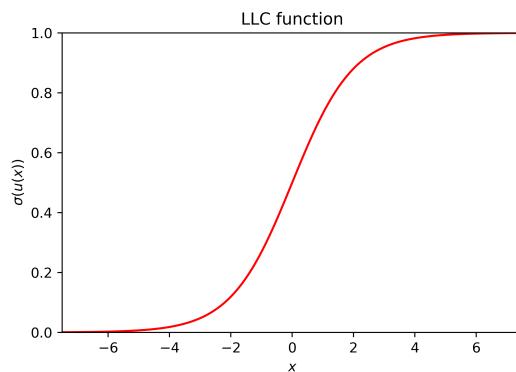
$$u = \theta^T x + \theta_0 \quad \sigma(u) = \frac{1}{1 + e^{-u}}$$

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

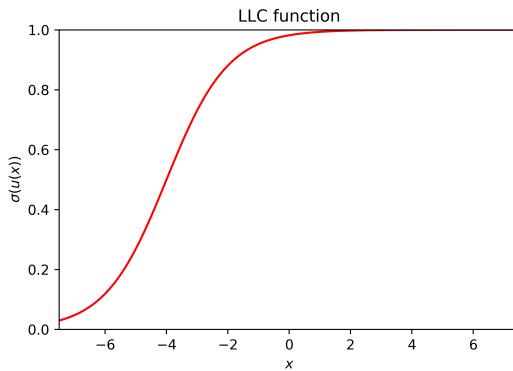
What happens when you modify the **parameters** of an LLC? Let's find out.

We'll use a 1-D input: our variables will be  $\theta$  (scalar) and  $\theta_0$ :  $\theta x + \theta_0$



Our baseline LLC:  $u(x) = 1x + 0$

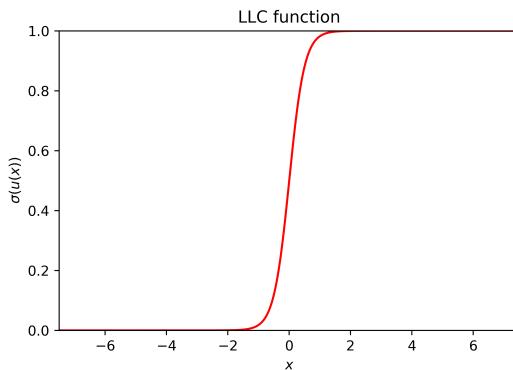
What if we shift by increasing  $\theta_0$ ?



Our shifted LLC:  $u(x) = 1x + 4$ .  $\theta_0$  shifts us along the x-axis!

Just like in linear regression, it **shifts** us in the **opposite** direction: if  $\theta_0$  is **positive**, we shift in the **negative** direction, and vice versa.

What if we increase the magnitude of  $\theta$ !



Our new LLC:  $u(x) = 4x$ . Increasing  $\theta$  makes our function steeper!

Making the magnitude of  $\theta$  larger makes our function **change** faster.

- This makes some sense: if  $\theta$  (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  $\theta_0$ , and also make it steeper/less steep by changing magnitude of  $\theta$ .

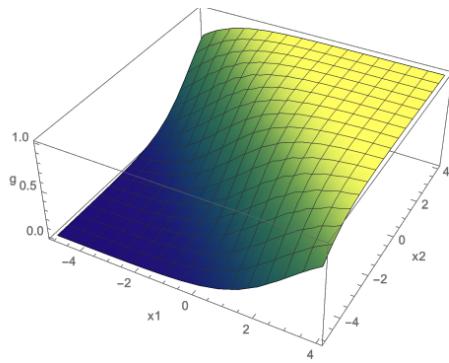
#### Concept 153

When working with **sigmoids**, you can **transform** them using your **parameters**:

- A higher **magnitude**  $\|\theta\|$  makes the slope **steeper**, and answers more **confident**.
- **Increasing**  $\theta_0$  **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.

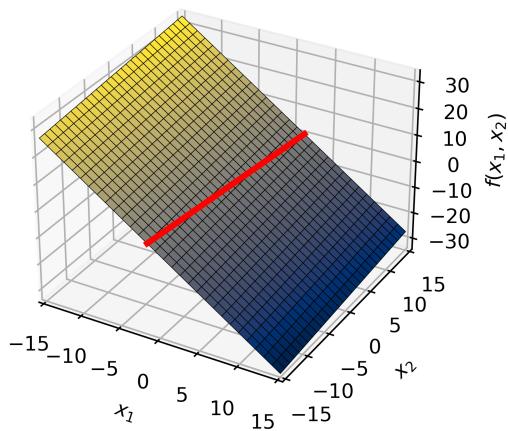
### 4.3.10 LLCs and LCs have the same boundary

One more important thing to note: noticed that we set  $\sigma_{\text{thresh}} = 0.5$ , because that was when  $u(x) = 0$ .

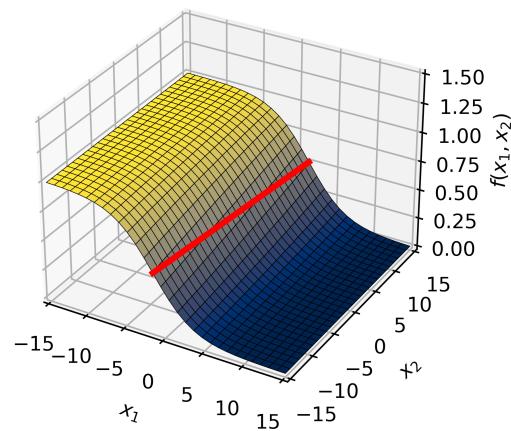
$$u = 0 \iff \sigma(u) = 0.5 \quad (4.19)$$

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  $\sigma_{\text{thresh}}$ , that will simply shift the linear classifier.

The probability interpretation is also more appropriate: we usually aren't fully confident in our answers.

#### Concept 154

**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.

- This is because the **same set of inputs**  $x$  that cause  $u(x) = 0$ , will also cause  $\sigma(u(x)) = 0.5$ .
- So, if  $u(x) = 0$  (LC) creates a hyperplane, then  $\sigma(u(x)) = 0.5$  (LLC) will, too.

### 4.3.11 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**:

$$\mathcal{L}_{01}(\mathbf{h}(\mathbf{x}; \Theta), y) = \begin{cases} 0 & \text{if } y = \mathbf{h}(\mathbf{x}; \Theta) \\ 1 & \text{otherwise} \end{cases}$$

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?

### 4.3.12 Building our new loss function

So, the **output** of our sigmoid  $\sigma(u)$  is a **probability**: it tells us, "how **likely** do we think a point is to be in class +1?"

We want a loss function

$$\mathcal{L}(g, y) \tag{4.20}$$

That considers two facts: the **correct** answer  $y$ , and how likely we **expected** +1 to be,  $g = \sigma(u)$ .

### Notation 155

For our **loss function**, rather than using  $\hat{y} \in \{-1, +1\}$ , we switch to **probabilities**:  $y \in \{0, 1\}$ .

$$\hat{y} \in \{-1, +1\} \rightarrow y \in \{0, 1\}$$

That way,  $\sigma(u)$  and  $y$  **match**:

$$y \in \{0, 1\} \quad g \in (0, 1)$$

Both represent "the chance that  $\hat{y} = +1$ ".  $\sigma(u)$  makes a prediction, while  $y$  uses the known result:

- If  $\hat{y} = +1$ , there's a **100% chance** that  $\hat{y} = +1$ .
  - So, the "true" output is  $y = 100\% = 1$
- If  $\hat{y} = -1$ , there's a **0% chance** that  $\hat{y} = +1$ .
  - So, the "true" output is  $y = 0\% = 0$

In this problem, it's easier to think in terms of "goodness" of the result. We'll use a "goodness" function  $G(g, y)$ .

- We want the "true" and "predicted" probabilities to be as close as possible.
  - If the correct answer is  $y = 1$ , then we want  $g = \sigma(u)$  to be **high**.
  - If the correct answer is  $y = 0$ , we want  $g = \sigma(u)$  to be **low**.

To get the loss function, we just take the negative of it later.

We represent this as

$$G(g, y) = \begin{cases} g & \text{if } y = 1 \\ 1 - g & \text{else } (y = 0) \end{cases} \quad (4.21)$$

**Remark (Optional) 156**

This loss function can be interpreted as, "if we had a  $g$  chance of picking +1, how often would we have been right?"

- If  $y = 1$ , then we want +1. The chance of choosing +1 is  $g$ .
- If  $y = 0$ , then we want -1. The chance of choosing -1 is  $1 - g$ .

0-1 loss works the same way for a non-random model (one that picks the larger odds every time): what's percentage of the data we get right.

As we mentioned, we'll need to take the negative of this later to get the "loss" function.

### 4.3.13 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  $E_i$ . If we want two independent events to **both** happen, we have to **multiply** their probabilities.

**Key Equation 157**

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:

$$P\{E_{\text{all}}\} = P\{E_1\} * P\{E_2\} * \dots * P\{E_n\} \quad (4.22)$$

Written using pi notation, and also  $g^{(i)}$  for multiple data points: \_\_\_\_\_

$$P\{E_{\text{all}}\} = \prod_{i=1}^n P\{E_i\} = \prod_{i=1}^n \begin{cases} g^{(i)} & \text{if } y^{(i)} = 1 \\ 1 - g^{(i)} & \text{if } y^{(i)} = 0 \end{cases} \quad (4.23)$$

Pi notation is described in the prerequisites chapter! The short version: instead of adding terms with  $\sum$ , you multiply with  $\prod$ .

### 4.3.14 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  $\textcolor{red}{g}^y$ : 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  $(1-\textcolor{red}{g})^{1-y}$ .

For one data point:

$$P\{E\} = \underbrace{\textcolor{blue}{g}^y}_{y=1} \underbrace{(1-\textcolor{red}{g})^{1-y}}_{y=0} \quad (4.24)$$

We've gotten rid of the piecewise function! Let's add back in the product:

$$P\{E_{all}\} = \prod_{i=1}^n P\{E_i\} = \prod_{i=1}^n \textcolor{red}{g}^{(i)} \textcolor{blue}{y}^{(i)} (1-\textcolor{red}{g}^{(i)})^{1-\textcolor{blue}{y}^{(i)}} \quad (4.25)$$

Looks pretty ugly, but we'll work on that.

### 4.3.15 Getting rid of the product

Our exponents look pretty **ugly**. Can we do something about that?

- More important than ugliness: **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

$$\overbrace{\log ab}^{\text{product}} = \overbrace{\log a + \log b}^{\text{sum}} \quad (4.26)$$

Aha! If we take the **log** of our function, we can turn a **product** into the **sum**!

$$\overbrace{\log \left( \prod_{i=1}^n p_i \right)}^{\text{product}} = \overbrace{\sum_{i=1}^n \log(p_i)}^{\text{sum}} \quad (4.27)$$

Plugging in  $p_i = P\{E_i\}$ :

$$\sum_{i=1}^n \log \left( g^{(i)y^{(i)}} (1 - g^{(i)})^{1-y^{(i)}} \right) \quad (4.28)$$

The below equation looks complicated, but all we've done is swap the product for a sum!

We can also separate our two **factors**,  $g^y$  and  $(1 - g)^{1-y}$ .

$$\sum_{i=1}^n \left( \log(g^{(i)y^{(i)}}) + \log((1 - g^{(i)})^{1-y^{(i)}}) \right) \quad (4.29)$$

And finally, we can remove the **exponents**:

$$\sum_{i=1}^n \left( y^{(i)} \log g^{(i)} + (1 - y^{(i)}) \log(1 - g^{(i)}) \right) \quad (4.30)$$

### Concept 158

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.

### 4.3.16 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 159**

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,

$$- \left( (\text{answer}) \log(\text{guess}) + (1 - \text{answer}) \log(1 - \text{guess}) \right)$$

Our total loss is

$$\sum_{i=1}^n \mathcal{L}_{\text{nll}}(g^{(i)}, y^{(i)}) \quad (4.31)$$

Finally, we add our **regularizer**:

$$J_{\text{lr}}(\theta, \theta_0; \mathcal{D}) = \frac{1}{n} \sum_{i=1}^n \left( \mathcal{L}_{\text{nll}}(g^{(i)}, y^{(i)}) \right) + \lambda \|\theta\|^2 \quad (4.32)$$

**Key Equation 160**

The full **objective function** for **LLC** is given as

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

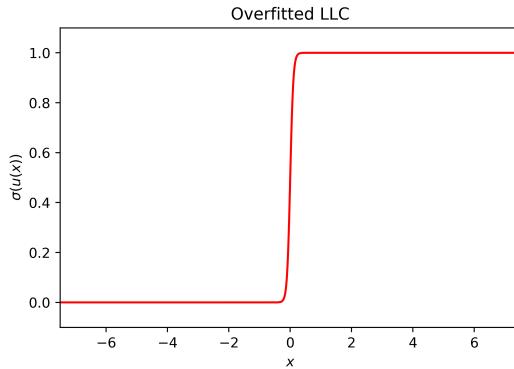
Using our **loss** function  $\mathcal{L}_{\text{nll}}$ , and our **logistic** function  $\sigma(u)$ .

### 4.3.17 LLCs and overfitting

In chapter 2, we reduced **overfitting** by limiting the **magnitude** of  $\theta$  using

$$R(\theta) = \lambda \|\theta\|^2 \quad (4.33)$$

In this chapter, it's more clear why reducing **magnitude** reduces **overfitting**. Let's see what happens when  $\theta$  is very **large**:

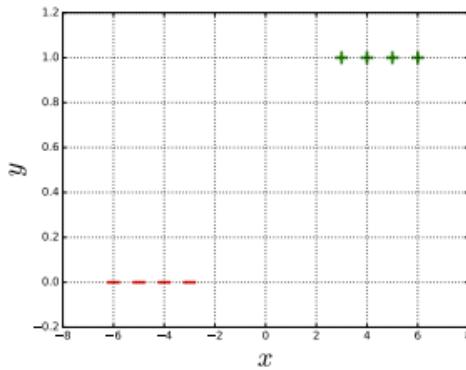


Our "squished" LLC:  $u(x) = 20x$ .

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.

- If we go from  $x = -0.5$  to  $x = +0.5$ , we go from "incredibly sure of  $-1$ " to "incredibly sure of  $+1$ ".
- That's a small change in input, for a big chance in confidence!

This sort of certainty isn't always appropriate: consider the below example.



In this case, you definitely want to separate the left and right side. Our  $\theta = 20$  separator above, would work just fine.

- But, is it **appropriate**? Let's try to use our model to predict new data.
- If you give that model  $x = 0.5$ , then our model believes  $\hat{y} = +1$  with .99995 confidence.
- When our closest data point to 0 is at  $\pm 3$ , that's unreasonably high certainty.

In other words, the model could create a very sensitive boundary, without having enough data to justify it: this could be a sign of **overfitting**.

Why would we train such an extreme model? Is this even a problem we have to worry about?

It's more serious than you'd expect: we can see why, by consider how our loss function works: here's an excerpt from earlier.

- We want the "true" and "predicted" probabilities to be as close as possible.
  - If the correct answer is  $y = 1$ , then we want  $g = \sigma(u)$  to be **high**.
  - If the correct answer is  $y = 0$ , we want  $g = \sigma(u)$  to be **low**.

The short version: we want to maximize our confidence in the correct answer. This is directly coded into our loss function, and all of the variations.

We just discussed that increasing  $\theta$  will increase your confidence in the answer. So, this loss function encourages our model to make  $\theta$  larger and larger!

So, we need to prevent  $\theta$  from **growing** to an unreasonably high value. Thus, we **penalize** a large  $\|\theta\|$ .

This means we're **penalizing** the machine's **overconfidence** in its answer, so that it **generalizes** better.

#### Concept 161

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 **sharp**, when it shouldn't be so **certain** of the boundary.
- It also prevents our model from becoming **overly confident** in its answer.

## 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 **input** variables into a single, predictive **number**:

$$u(x) = \theta^T x + \theta_0 \quad (4.34)$$

- A **logistic** model that turns this **number** into a **probability** of a classification,

$$\sigma(u) = \frac{1}{1 + e^{-u}} \quad (4.35)$$

- A **threshold value** we use to determine how to use this **probability** to **classify**:

$$h(x; \theta) = \begin{cases} +1 & \text{if } \sigma(u(x)) > \sigma_{\text{thresh}} \\ 0 & \text{otherwise} \end{cases} \quad (4.36)$$

- A **loss function** NLL we use to evaluate the **quality** of our **classifications**:

$$\mathcal{L}_{\text{nll}}(g^{(i)}, y^{(i)}) = - \left( y^{(i)} \log g^{(i)} + (1 - y^{(i)}) \log (1 - g^{(i)}) \right)$$

- And an **objective function** we can **optimize** and reduce our **loss**:

$$J_{\text{lr}}(\theta, \theta_0; \mathcal{D}) = \lambda \|\theta\|^2 + \frac{1}{n} \sum_{i=1}^n \mathcal{L}_{\text{nll}}(g^{(i)}, y^{(i)}) \quad (4.37)$$

We have everything we need to do optimization.

### 4.4.2 The problem: Gradient Descent

We want to do **gradient descent** to minimize  $J_{\text{lr}}$

$$J_{\text{lr}}(\theta, \theta_0) = R(\theta) + \frac{1}{n} \sum_{i=1}^n \mathcal{L}_{\text{nll}}(g^{(i)}, y^{(i)}) \quad (4.38)$$

We want repeatedly **adjust** our model  $\Theta = (\theta, \theta_0)$  to improve  $J_{\text{lr}}$ . To do that, we want the gradients for  $\theta$  and  $\theta_0$ . Let's start with  $\theta$ .

$$\nabla_{\theta} J_{\text{lr}} = \frac{\partial J_{\text{lr}}}{\partial \theta} \quad (4.39)$$

First,  $J_{\text{lr}}$  has **two** terms, so we'll separate them.

$$\nabla_{\theta} J_{lr} = \frac{\partial R}{\partial \theta} + \frac{1}{n} \sum_{i=1}^n \frac{\partial \mathcal{L}_{NLL}}{\partial \theta}(g^{(i)}, y^{(i)}) \quad (4.40)$$

The regularization term is pretty easy, because we did it last chapter:

$$\frac{\partial R}{\partial \theta} = 2\lambda\theta \quad (4.41)$$

But what about our first term?

### 4.4.3 Getting the gradient: Chain Rule

Now, we just need to do

$$\frac{\partial \mathcal{L}_{NLL}}{\partial \theta}(g, y) \quad (4.42)$$

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**:

This next line is a generic chain rule: not specific to our problem.

$$\frac{\partial A}{\partial C} = \frac{\partial A}{\partial B} \cdot \frac{\partial B}{\partial C} \quad (4.43)$$

So, we'll build up a **chain rule** for our needs. We'll use  $g = \sigma(u)$ .

$$\frac{\partial \mathcal{L}_{NLL}}{\partial \theta} = \frac{\partial \mathcal{L}_{NLL}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial \theta} \quad (4.44)$$

Sigma contains  $u$ , so we'll add that to the chain:

$$\frac{\partial \mathcal{L}_{NLL}}{\partial \theta} = \frac{\partial \mathcal{L}_{NLL}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial u} \cdot \frac{\partial u}{\partial \theta} \quad (4.45)$$

This is our full **chain rule**!

#### Key Equation 162

The **gradient** of **NLL** can be calculated using the **chain rule**:

$$\frac{\partial \mathcal{L}_{NLL}}{\partial \theta} = \frac{\partial \mathcal{L}_{NLL}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial u} \cdot \frac{\partial u}{\partial \theta} \quad (4.46)$$

### 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  $\frac{d}{dx} \log(x) = \frac{1}{x}$

$$\boxed{\frac{\partial \mathcal{L}_{\text{NLL}}}{\partial \sigma} = - \left( \frac{y}{\sigma} - \frac{1-y}{1-\sigma} \right)} \quad (4.47)$$

Now, we look at  $\sigma(u)$ :

$$\sigma(u) = \frac{1}{1 + e^{-u}} \quad (4.48)$$

If we take the derivative, we can get:

$$\frac{\partial \sigma}{\partial u} = \frac{-e^{-u}}{(1 + e^{-u})^2} \quad (4.49)$$

Which we can rewrite, conveniently, as

Try this yourself if you're curious!

$$\boxed{\frac{\partial \sigma}{\partial u} = \sigma(1 - \sigma)} \quad (4.50)$$

Finally, our last derivative:

$$u = \theta^T x + \theta_0 \quad (4.51)$$

$$\boxed{\frac{\partial u}{\partial \theta} = x} \quad (4.52)$$

#### 4.4.5 Simplifying our chain rule

So, now, we can put together our chain rule:

$$\frac{\partial \mathcal{L}_{\text{NLL}}}{\partial \theta} = \frac{\partial \mathcal{L}_{\text{NLL}}}{\partial \sigma} \cdot \frac{\partial \sigma}{\partial u} \cdot \frac{\partial u}{\partial \theta} \quad (4.53)$$

Plug in the derivatives:

$$\frac{\partial \mathcal{L}_{\text{NLL}}}{\partial \theta} = - \left( \frac{y}{\sigma} - \frac{1-y}{1-\sigma} \right) \cdot \sigma(1 - \sigma) \cdot x \quad (4.54)$$

Simplify:

$$\frac{\partial \mathcal{L}_{\text{NLL}}}{\partial \theta} = ((1 - \mathbf{y})\sigma - \mathbf{y}(1 - \sigma)) \cdot \mathbf{x} \quad (4.55)$$

And finally, we sum the terms. We can do the  $\theta_0$  gradient at the same time: the only difference is that  $\frac{\partial u}{\partial \theta_0} = 1$ , instead of  $x$ .

### Key Equation 163

The **gradients** of NLL for gradient descent are

$$\nabla_{\theta} \mathcal{L}_{\text{NLL}} = (\sigma - \mathbf{y}) \mathbf{x}$$

$$\frac{\partial \mathcal{L}_{\text{NLL}}}{\partial \theta_0} = (\sigma - \mathbf{y})$$

We can plug this into  $J_{\text{lr}}$ :

$$\nabla_{\theta} J_{\text{lr}} = \frac{1}{n} \sum_{i=1}^n \left( (\mathbf{g}^{(i)} - \mathbf{y}^{(i)}) \mathbf{x}^{(i)} \right) + 2\lambda\theta \quad (4.56)$$

One comment we didn't make: remember that  $R(\theta)$  won't show up in the  $\theta_0$  derivative!

$$\frac{\partial J_{\text{lr}}}{\partial \theta_0} = \frac{1}{n} \sum_{i=1}^n (\mathbf{g}^{(i)} - \mathbf{y}^{(i)}) \quad (4.57)$$

We can use this to do **gradient descent**!

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \nabla_{\theta} J_{\text{lr}}(\theta_{\text{old}}) \quad (4.58)$$

In  $\theta^{(t)}$  notation:

$$\theta^{(t)} = \theta^{(t-1)} - \eta \left( \nabla_{\theta} J_{\text{lr}}(\theta^{(t-1)}) \right) \quad (4.59)$$

$$\theta_0^{(t)} = \theta_0^{(t-1)} - \eta \left( \frac{\partial J_{\text{lr}}(\theta^{(t-1)})}{\partial \theta_0} \right) \quad (4.60)$$

- This chain-rule approach will return when we do Neural Networks in a later chapter.

## 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  $i^{\text{th}}$  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**:

$$\begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_k \end{bmatrix} \quad (4.61)$$

So, our **label** will be

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{bmatrix} \quad (4.62)$$

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 164

**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.

$$\begin{bmatrix} \text{table} \\ \text{bed} \\ \text{couch} \\ \text{chair} \end{bmatrix} \quad (4.63)$$

For each class:

$$y_{\text{chair}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad y_{\text{table}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad y_{\text{couch}} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad y_{\text{bed}} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad (4.64)$$

### 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.

$$g = \begin{bmatrix} P\{x \text{ in } C_1\} \\ P\{x \text{ in } C_2\} \\ \vdots \\ P\{x \text{ in } C_k\} \end{bmatrix} = \begin{bmatrix} g_1 \\ g_2 \\ \vdots \\ g_k \end{bmatrix} \quad (4.65)$$

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**.

$$g_1 + g_2 + \dots + g_k = \sum_i g_i = 1 \quad (4.66)$$

#### Concept 165

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 166**

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  $C$ :

$$C(r_1 + r_2 + \dots + r_k) = C \left( \sum_{i=1}^k r_i \right) = 1 \quad (4.67)$$

We can get our factor  $C$  just by dividing:

$$C = \frac{1}{\sum r_i} \quad (4.68)$$

We've got our desired  $g_i$  now!

$$g = \begin{bmatrix} r_1 / \sum r_i \\ r_2 / \sum r_i \\ \vdots \\ r_k / \sum r_i \end{bmatrix} \quad (4.69)$$

#### 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:

$$g = \sigma(u) = \frac{1}{1 + e^{-u}} \quad (4.70)$$

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.

$$g = \frac{e^u}{e^u + 1} \quad (4.71)$$

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 probability for our first class.

**Concept 167**

**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,  $g_2$ .

$$g_2 = 1 - g = \frac{1}{1 + e^u} \quad (4.72)$$

This follows an  $r_i / (\sum r_i)$  format! The numerators (1 and  $e^u$ ) add to **equal** the denominator ( $1 + e^u$ ).

$$g = \begin{bmatrix} e^u / (e^u + 1) \\ 1 / (e^u + 1) \end{bmatrix} = \begin{bmatrix} r_1 / (r_1 + r_2) \\ r_2 / (r_1 + r_2) \end{bmatrix} \quad (4.73)$$

How do we **extend** this to **more** classes? Well, 1 and  $e^u$  are **different** functions: this is 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**!

$$g_1 = \frac{e^u}{e^u + e^0} \quad g_2 = \frac{e^0}{e^u + e^0} \quad (4.74)$$

What if we want more classes? We just need more exponentials! They'll fit into the pattern from  $e^u$  and  $e^0$ :

$$g_i = \frac{r_i}{\sum r_j} = \frac{e^{u_i}}{\sum e^{u_j}} \quad (4.75)$$

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:

$$u(x) = \theta^T x + \theta_0 \quad (4.76)$$

Remember that  $u(x)$  is not a probability yet: we use 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  $(\theta, \theta_0)$ . We'll represent each vector as  $\theta_{(i)}$ .

$$\theta_{(1)} = \begin{bmatrix} \theta_{1(1)} \\ \theta_{2(1)} \\ \vdots \\ \theta_{d(1)} \end{bmatrix} \quad \theta_{(2)} = \begin{bmatrix} \theta_{1(2)} \\ \theta_{2(2)} \\ \vdots \\ \theta_{d(2)} \end{bmatrix} \quad \theta_{(k)} = \begin{bmatrix} \theta_{1(k)} \\ \theta_{2(k)} \\ \vdots \\ \theta_{d(k)} \end{bmatrix} \quad (4.77)$$

Each of these models could be seen as a "different perspective" of our data point: what about that data point is **prioritized** (large  $\theta_i$  magnitudes), and how do we **bias** the result ( $\theta_0$ )?

This "perspective" we call  $\theta_{(i)}$  will tell us if our data point is "closer" to the **class** it represents

$$u_1(x) = \theta_{(1)}^T x + \theta_{0(1)} \quad u_2(x) = \theta_{(2)}^T x + \theta_{0(2)} \quad u_k(x) = \theta_{(k)}^T x + \theta_{0(k)} \quad (4.78)$$

These equations allow us to directly compute each  $u_i$ .

- Intuitively, if  $u_i(x)$  is **larger than**  $u_j(x)$ , then our data point  $x$  is **more similar to** class  $i$  than class  $j$ .

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 168

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  $\theta_i$  is tedious. Instead, we can combine them all into a  $(d \times k)$  **matrix**.

$$\theta = \begin{bmatrix} \theta_{(1)} & \theta_{(2)} & \cdots & \theta_{(k)} \end{bmatrix} = \begin{bmatrix} \theta_{1(1)} & \theta_{1(2)} & \cdots & \theta_{1(k)} \\ \theta_{2(1)} & \theta_{2(2)} & \cdots & \theta_{2(k)} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{d(1)} & \theta_{d(2)} & \cdots & \theta_{d(k)} \end{bmatrix} \quad (4.79)$$

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,  $\theta_0$ , in a  $(k \times 1)$  matrix:

$$\theta_0 = \begin{bmatrix} \theta_{0(1)} \\ \theta_{0(2)} \\ \vdots \\ \theta_{0(k)} \end{bmatrix} \quad (4.80)$$

**Concept 169**

We can combine **multiple classifiers**  $\Theta_{(i)} = (\theta_{(i)}, \theta_{0(i)})$  into large **matrices**  $\theta$  and  $\theta_0$  to compute **multiple** outputs  $u_i$  at the **same** time.

This will put all of our terms into a  $(1 \times k)$  vector  $u$ .

$$u(x) = \theta^T x + \theta_0 = \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_k \end{bmatrix} \quad (4.81)$$

Which creates a deceptively simple formula: this is one of the perks of matrix multiplication!

#### 4.5.6 Softmax

We now have all the pieces we need.

- Our **linear regression** for each class:

$$\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_k \end{bmatrix} = \theta^T x + \theta_0 \quad (4.82)$$

- The **exponential** terms, to get **logistic** behavior

$$r_i = e^{u_i} \quad (4.83)$$

- The **averaging** to get the probabilities to add to 1:

$$g = \begin{bmatrix} r_1 / \sum r_i \\ r_2 / \sum r_i \\ \vdots \\ r_k / \sum r_i \end{bmatrix} \quad (4.84)$$

And so, our multiclass function is...

**Definition 170**

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

$$u_i(x) = \theta_{(i)}^T x + \theta_{0(i)} \quad (4.85)$$

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  $y = y_1$ .
  - Similarly,  $1 - g = g_2$  pairs with  $1 - y = y_2$ .

$$\mathcal{L}_{\text{nll}}(g, y) = - \left( y_1 \log g_1 + (y_2) \log (g_2) \right)$$

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 171

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** for  $y$ , 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.

#### 4.5.8 A side comment: Sigmoid vs. Softmax

Let's jump back to softmax real quick and clarify something.

Usually, we expect to use **softmax** if we have **more than 2** classes, because that's what we built it for.

- However, this isn't always the case – there's another aspect of softmax we haven't focused on:
- **Softmax** represents  $k$  different classes/events. These classes are assumed to be **mutually exclusive**: you can't be in multiple at the same time.

In other words, the classes of softmax are **disjoint**.

### Definition 172

If two events are **disjoint**, they **can't** happen at the **same time**.

If  $n$  events are **disjoint**, only **one** of them can happen at a time.

**Example:** We usually wouldn't classify an animal as both a cat and a dog: it's either one or the other.

When events are disjoint, their probabilities are separate:

**Concept 173**

If two events are **disjoint**, then they have **separate** probabilities: there's no overlap. Since  $P\{A \cap B\} = 0$ , we can say:

$$P\{A \cup B\} = P\{A\} + P\{B\}$$

If we have **every** event and they're all **disjoint**, then their probabilities sum to 1.

$$\sum_i p_i = 1 \quad (4.86)$$

**Example:** If the weather options are rain, cloudy, and sunny, and you have to only choose one, you should expect that:

$$P\{\text{Rain}\} + P\{\text{Cloudy}\} + P\{\text{Sunny}\} = 1 \quad (4.87)$$

~~~~~  
But this only makes sense **if** events can't happen at the same time.

- In some situations, multiple classes/events can happen at the same time! They might even be **independent**.
- **Example:** There might be k different people we could find in an **image**. But, there can be multiple people in the **same** image!

So, it doesn't always make sense to assume that each event is **mutually exclusive**.

- If our events are not mutually exclusive, then we **shouldn't use softmax**.

The solution: for each class, find the **probability** for that class, vs. the **absence** of that class.

- This brings us back to binary classification: we just use **one sigmoid per class**.

Clarification 174

Softmax is used when each of our k classes is **disjoint** (mutually exclusive).

- However, if they aren't, then we **can't use softmax**.

~~~~~  
If our classes are independent, we can use  $k$  **sigmoid** functions: one for each of our  $k$  classes. We're using **binary classification** on each class separately.

- The  $i^{\text{th}}$  sigmoid tells us how likely the **data point** is to be in the  $i^{\text{th}}$  class.

**Example:** We might have an algorithm figuring out which **products** a customer might want. They might want **multiple**, so we can't treat them as disjoint.

In this case, each product is a class, and we determine the result based on the matching sigmoid

What happens if the events aren't disjoint, but they also aren't independent? You need a more complex model.

## 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 practice, we often don't care about how close we were to right - we just care about how often we **were** right.

So, we use **accuracy**.

### Definition 175

The **accuracy**  $A$  of our model is the **percentage** of the time we get the **right** answer.

We can write this as

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

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-separate 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

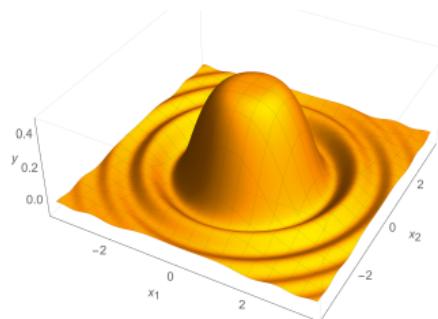
### What's still missing?

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  $\theta$  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 176**

**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.

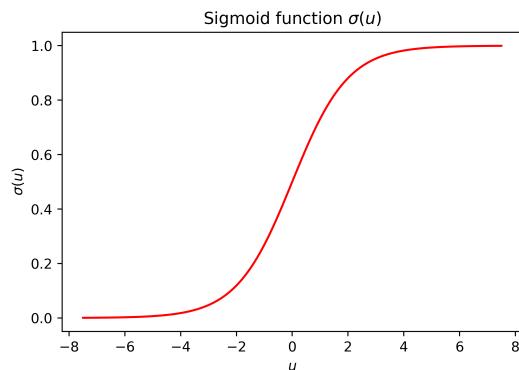
If we could create effective, non-linear models, we might even be able to deal with data that was previously "linearly inseparable".



## 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.

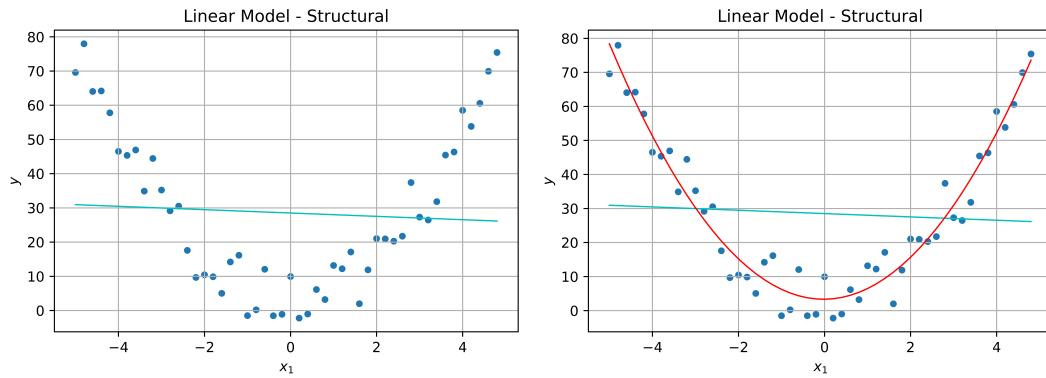


Figure 5.1: A linear regression can't represent this dataset. However, a parabola can!

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.

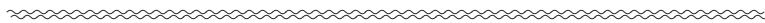
### Concept 177

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**.

This is called a **polynomial transformation** of our input data.



## Transformation

How do we *generalize* this concept? Well, we have a set of constant parameters  $A, B, C$ . These are similar to our constants  $\theta_i$ . 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  $\theta$  as a parameter:

Notice that  $\theta_0$  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  $\theta^T x$ . 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  $\phi(x)$ :

$$\phi(x) = \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^k \end{bmatrix} \quad h(x) = \theta^T \underbrace{\phi(x)}_{\text{New input}} \quad (5.4)$$

Compare the structure of  $\theta^T x$  versus  $\theta^T \phi(x)$ : you've replaced  $x$  with  $\phi(x)$ .

This is called **transforming** our input. However, polynomial is only one of our transformations!

### Definition 178

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
  - **Example:** Polynomial transformations
- **Pre-process** our data to make certain **patterns** more obvious, and easy for our model to detect.
  - **Example:** Radial transformations (to be discussed later!)
- Convert our data into a **usable** format (if the original format doesn't fit into our equations)
  - **Example:** One-hot encoding

**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 the input is that everything else we know about the model is still true: we can continue to use our linear representation.

- We will be able to optimize a "linear" model  $\theta$ , 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 ( $x = x_1$ ) with **itself**. But, often, we can combine multiple  $x_i$  together!

So, we need to be careful of our input variables  $x_i$ .

We'll cover this multi-dimensional polynomial transformation later in the chapter.

- We sometimes call a single input variable, a single "**feature**". However, we need to be careful: this word can have multiple meanings.

### Clarification 179

We often use the word **feature** in related (but not identical) contexts:

- A **feature** can be one unprocessed **aspect** of the **data**:
  - **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**.
  - $\phi(x)_j$  (one variable in  $\phi(x)$ ) is a **feature** of the **transformed data**.

Just like how we have an **input space** and a **hypothesis space**, we call the collection of possible values for our features the **feature space**.

Combined, this is why we called this technique the **feature transformation**:

- We apply a transform to the **features**  $x$  of our data, and create a new list of **features**  $\phi(x)$ .

Since these transforms only apply to our features, they don't affect the rest of our model. So, we can still use **linear** tools:

**Definition 180**

Feature transformation allows us to do linear regression or classification on a list of features we have non-linearly transformed:

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

- The  $\theta^T u$  operation is still linear ( $u = \phi(x)$ ).
- All non-linearity is stored in  $\phi(x)$ .

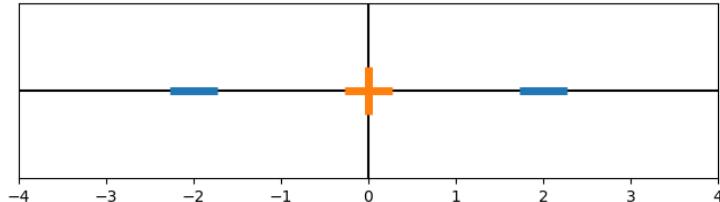


## 5.1 Gaining intuition about feature transformations

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 them **more accurate**.

So, we'll consider maybe the simplest (solvable) case of a nonlinear data set in 1-D:



The y-axis doesn't exist, it just has vertical height to make it easier to view on a page.

Figure 5.2: In this state, there's no 0-d plane (point) that would **separate** these data points.

This is where our transform comes in: we can't separate using just  $x$ . So, we'll introduce a second variable:  $x^2$ .

- We want a function that lets us classify some data points as positive, and some as negative.
- For the dataset in this image,  $-x^2 + 2 > 0$  gives us 100% accuracy. Let's see it in action:

We're replacing  
 $\theta_0 + \theta_1 x > 0$  with  
 $\theta_2 x^2 + \theta_1 x + \theta_0 > 0$ .

$$\begin{array}{lll} x = 2 & \parallel & y = -(2)^2 + 2 = -2 \\ x = 0 & \parallel & y = -(0)^2 + 2 = 2 \\ x = -2 & \parallel & y = -(-2)^2 + 2 = -2 \end{array} \quad \begin{array}{ll} y < 0 \implies \text{Negative} \\ y > 0 \implies \text{Positive} \\ y < 0 \implies \text{Negative} \end{array}$$

How do we visualize this? It turns out, there are different perspectives:

**Clarification 181**

There are **two** different ways we can **graph** a transformation:

We transform the **hyperplane**:

- **Example:** If our model is  $f(x) = -x^2 + 2$ , we just graph  $y = f(x)$  as our separator in 2D space.
  - This is the approach we used to start the chapter: we wanted a line that **fit** to our data.
  - In practice, this bends our **hyperplane** into a curve: at the start of the chapter, we transformed a line into a parabola.

Or, we transform the **data**:

- **Example:** We plot each data point in 2D as  $[x, -x^2 + 2]$ .
  - This model allows us to keep a "**linear** separator": we "shift" the data nonlinearly, **then** linearly separate it.

These models are mathematically **equivalent**, and we'll switch the approach we're using based on which is easier/more useful to graph.

See our plot examples for each below.

Note that our nonlinear transformation "adds" dimensions: we had a 1D problem, and we used a second dimension to separate it.

It may seem concerning to transform the **data**, rather than the **model**. The data is what we're using to make decisions, after all.

However, keep in mind that:

- Our model already was a sort of **transformation**: even the linear model  $\theta$  "transforms" each data point  $x$  into  $y = \theta^T x$ .
- 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 append 1 and  $x^2$ .

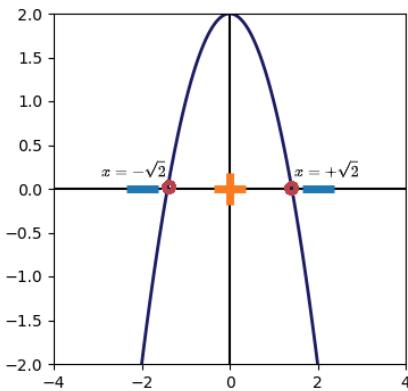
**Example:** Let's show both of these in action, using the 1-D dataset we showed above.



### 5.1.1 Transforming our separator

First, we transform our linear separator as desired: graphing  $-x^2 + 2 = f(x)$ .

- Our separator points are still on the  $x$ -axis: they "separate" our data wherever  $f(x) = 0$ .



In this version, we've taken our hyperplane separator and transformed it nonlinearly.

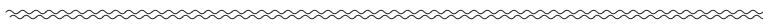
To correctly classify, we assign  $-x^2 + 2 > 0$  as positive.

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 direction and orientation of the "plane" now that it's been deformed into a parabola.

- 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**, instead of the separator. We'll do that next.

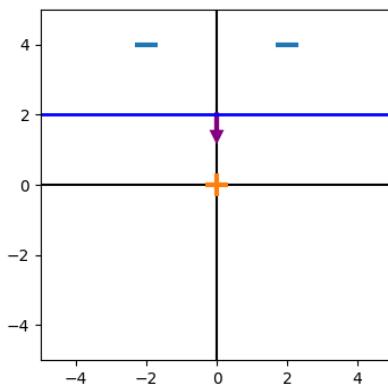


### 5.1.2 Transforming our data

In this case, every data point gets plotted on  $[x, x^2]$ . Our hyperplane is given by

$$-x^2 + 2 = \underbrace{\begin{bmatrix} 0 \\ -1 \end{bmatrix}}_{\theta^T}^T \begin{bmatrix} x \\ x^2 \end{bmatrix} + 2 = \theta^T \phi(x) + \theta_0 \quad (5.5)$$

Thus, we get a  $\theta$  plane 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 182

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?

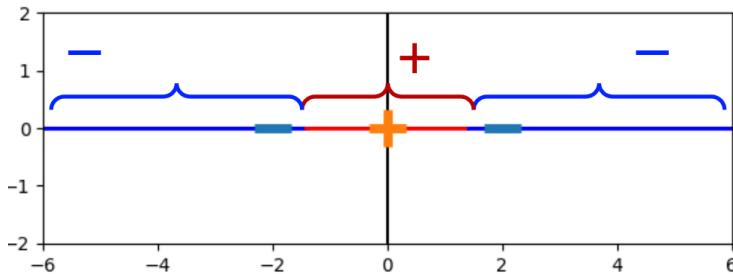
So, we'll create a third representation to correspond to that.

#### Concept 183

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:

$$h(x) = \sum_{i=1}^k \theta_i x^i = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_k \end{bmatrix} \cdot \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^k \end{bmatrix} = \theta_0 + \theta_1 x^1 + \theta_2 x^2 + \cdots + \theta_k x^k \quad (5.6)$$

Another word for these  $x_i$  terms might be a "polynomial **basis**".

- Why call it a basis? Well, we can use our  $x_i$  terms to create a polynomial, using

$$\sum_i \theta_i x^i \quad (5.7)$$

- Using this procedure, we can combine **basic** elements  $x_i$  to create any **polynomial**.

$$\{1, x_1, x_2, \dots, x_k\} \quad (5.8)$$

- This is what defines it as a **basis**: the ability to combine these terms, make any polynomial we want.

#### 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**.

We could also call this "problem independent": it works regardless of what kind of problem you have. Though, that doesn't mean problem type won't affect performance.

**Definition 184**

**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**.

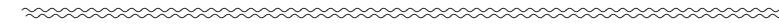
**Example:** We can compare different orders, by looking at the feature vector they create.

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     | $[1, x, x^2, x^3]^T$        | $x^3 + 8x^2 + x - \sqrt{2}$ |
| :     | :                           | :                           |
| $k$   | $[1, x, x^2, \dots, x^k]^T$ | $\sum_{j=1}^k \theta_j x^j$ |
| :     | :                           | :                           |

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 design 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 185**

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 186

**High-order polynomials** are very vulnerable to **overfitting**.

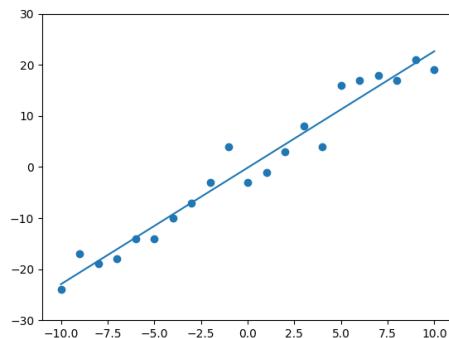
Because they can take on so many different, **complex** functions, they can very very closely **match** the original data set.

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.

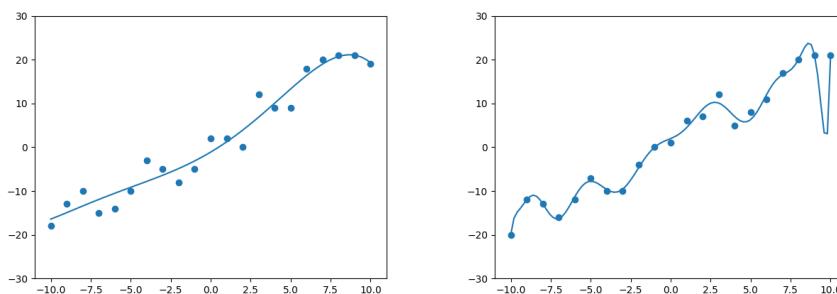
Let's see this in action: we'll generate some data based on  $2x + 1$ , while applying some random noise to it. We'll see the optimized linear regression model for each.

Rather than transform the data, we'll transform the separator: this really highlights the overfitting effect.

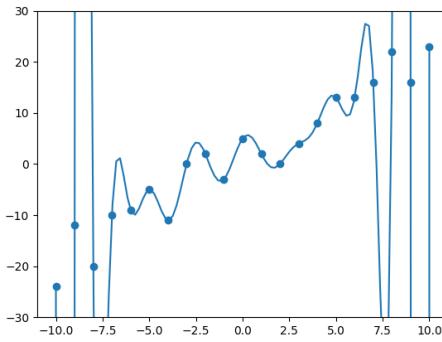
For ease, we'll exclude regularization: it does help mitigate this problem, but it doesn't totally solve it.



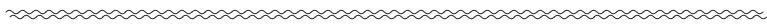
Here's the 1st order solution: in this case, correct for the underlying distribution. It fits our data fine.



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  $[x_1, x_2]^T$ .

We start with our typical 2D model:

$$\theta^T x + \theta_0 = \theta_1 x_1 + \theta_2 x_2 + \theta_0 \quad (5.9)$$

Is polynomial basis, with "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 **largest 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 187

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:

$$\begin{bmatrix} 1 & \textcolor{red}{x}_1 & \textcolor{red}{x}_1^2 & \textcolor{blue}{x}_2 & \textcolor{blue}{x}_2 x_1 & \textcolor{blue}{x}_2^2 \end{bmatrix}^\top \quad (5.10)$$

For  $d = 2$ , order=3, it starts getting a bit messy: we have 10 different basis terms.

You don't need to memorize these.

$$\begin{bmatrix} 1 & x_1 & x_1^2 & x_1^3 & x_2 & x_2 x_1 & x_2 x_1^2 & x_2^2 & x^2 x_1 & x^3 \end{bmatrix}^\top \quad (5.11)$$

~~~~~

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.

For example, here's how this would look for $d = 2$, $k = 2$ (5.10 above). All you need to notice is that the exponents add to 2.

$$\begin{bmatrix} 1^2 x_2^0 x_1^0 & 1^1 x_2^0 x_1^1 & 1^0 x_2^0 x_1^2 & 1^1 x_2^1 x_1^0 & 1^0 x_2^1 x_1^1 & 1^0 x_2^2 x_1^0 \end{bmatrix}^\top \quad (5.12)$$

This is a well-known problem in combinatorics: how many ways are there to add up $d + 1$ numbers to total k ? The solution to this problem gives us:

$$\binom{(d+1)+k-1}{k} = \binom{d+k}{k} = \frac{(d+k)!}{d!k!} \quad (5.13)$$

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 188

The **polynomial basis** of order k and dimension d includes **every feature**

$$\prod_{i=1}^d x_i^{c_i}$$

Where all of the integer exponents $c_i \geq 0$ add up to **at most** k .

Creating features such as:

$$x_1^k, x_1 x_2, x_2 x_3^3 x_6, 1, \dots$$

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!

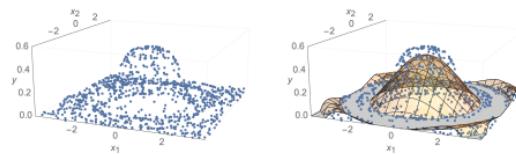
Concept 189

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.

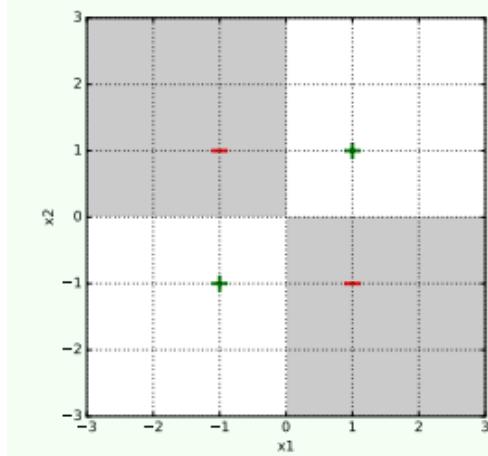
We'll train our model for various situations, to see what it can do. Different problems require different orders k , still.



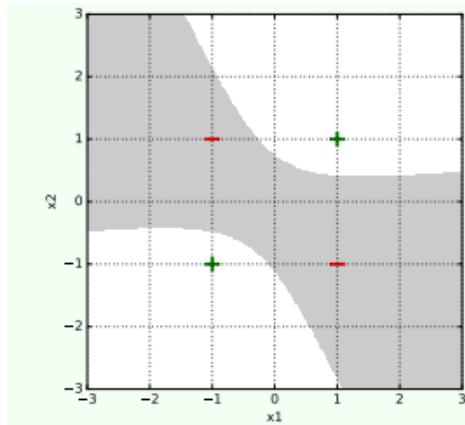
We start with the waveform we showed at the start of the chapter: on the left, we see a bunch of datapoints we want to take a regression over. With $k = 8$, we get a pretty good result.

For 2D separators, it's easier to show only the +/- classification, rather than the transformed data/boundary. That means, these below graphs are hiding the numeric outputs.

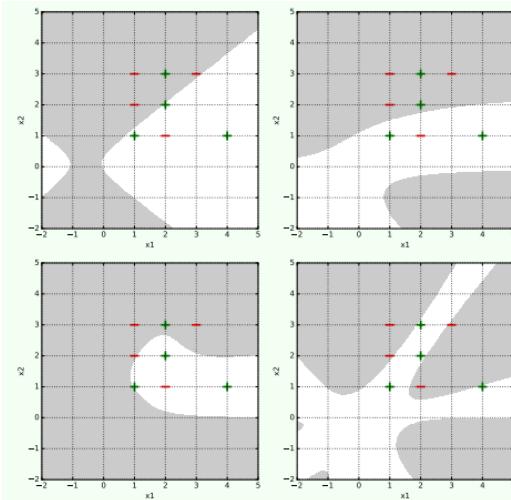
Light indicates "positive" for the model, dark indicates "negative" for the model.



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 rigid, 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.

$$\|p - x\|^2 \quad (5.14)$$

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 190

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 191

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.

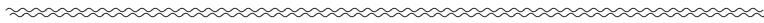
The **transformations** in this section address these kinds of problems. We take data that is informative, but not currently usable, and converting them to something we can **compute** with.

Concept 192

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 193

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 194

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: _____

$$\left[\frac{1}{k} \quad \frac{2}{k} \quad \dots \quad \frac{k-1}{k} \quad 1 \right], \quad \text{Class } i \longrightarrow \frac{i}{k} \quad (5.15)$$

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 195

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?"

$$\phi(x) = \begin{cases} \text{Class}(x) \geq 4 \\ \text{Class}(x) \geq 3 \\ \text{Class}(x) \geq 2 \\ \text{Class}(x) \geq 1 \end{cases} \quad (5.16)$$

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 196

Thermometer encoding is a **feature transform** where we take each class and turn it into a feature vector $\phi(x)$ where

$$\text{Class 1} \rightarrow \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \text{Class 2} \rightarrow \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 1 \end{bmatrix}, \quad \text{Class 3} \rightarrow \begin{bmatrix} 0 \\ \vdots \\ 1 \\ 1 \\ 1 \end{bmatrix}, \quad \text{Class k} \rightarrow \begin{bmatrix} 1 \\ \vdots \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

- 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 that implies no order.

- 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.

$$\begin{bmatrix} C_1 \\ C_2 \\ \vdots \\ C_k \end{bmatrix} \quad (5.17)$$

But in this case, we don't "build up" our vector: we replace $\text{Class}(x) \geq 4$ with $\text{Class}(x) = 4$.

$$\phi(x) = \begin{bmatrix} \text{Class}(x) = 4 \\ \text{Class}(x) = 3 \\ \text{Class}(x) = 2 \\ \text{Class}(x) = 1 \end{bmatrix} \quad (5.18)$$

This approach is called **one-hot encoding**.

Definition 197

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.

$$\begin{bmatrix} \text{table} \\ \text{bed} \\ \text{couch} \\ \text{chair} \end{bmatrix} \quad (5.19)$$

For each class:

$$y_{\text{chair}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad y_{\text{table}} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad y_{\text{couch}} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad y_{\text{bed}} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad (5.20)$$

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

$$\theta^T x = \sum_i \theta_i x_i \quad (5.21)$$

The vector format $\theta^T x$ is just a way to **condense** our equation: addition ignores ordering of elements!

Concept 198

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 doesn't encode that ordering.

Example: We'll take a vector, and rearrange it.

Despite shuffling, these two equations are equivalent!

$$\theta^T \phi(x) = [\theta_1 \ \theta_2 \ \theta_3 \ \theta_4] \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \longrightarrow (\theta^T)^* (\phi(x))^* = [\theta_3 \ \theta_1 \ \theta_4 \ \theta_2] \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

The math is the same, despite changing order: our model knows nothing about ordering.

Clarification 199

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.

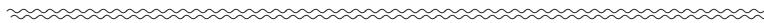
$$\theta^T \phi(x) = [\theta_1 \ \theta_2 \ \theta_3 \ \theta_4] \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix} \quad (5.22)$$

$$(\theta^T)^* (\phi(x))^* = [\theta_3 \ \theta_1 \ \theta_4 \ \theta_2] \begin{bmatrix} 1 \\ 0 \\ 1 \\ 1 \end{bmatrix} \quad (5.23)$$

Even though we've changed the order, we still know this is the **third** in the order, because we have **three 1's**!

Concept 200

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 201

Factored code is a **feature transformation** where we take one **discrete class** and break it up into other discrete classifications, called **factors**.

Class m and n \rightarrow Class m, Class n

- 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 202

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 203

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 204

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 205

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 206

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.

You could try to solve this by scaling down θ .

But, we're already using regularization to bias against large θ : that will affect small variables (big θ_i) more than larger ones (small θ_i).

$$\phi_1(x) = x - \bar{x} \quad (5.30)$$

- **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**, a measure of how much our data varies.

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 207

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 208

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 209

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

Clustering

6.0.1 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 has lots of benefits:

- It could be used to make **decisions**. Sometimes, knowing the class of an object is enough to make a decision, by itself.
 - We could use this to understand the structure 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, we've done this ourselves, giving us lots of useful classifications. But, **computers** allow us to do this in new situations:

- **High-dimensional** datasets, with too much **complex** information for a human to make sense of.
 - **Example:** Looking for patterns in hundreds of genetic factors at the same time.
- Discovering **new classes faster** than ever using computers.
 - And thus saving human labor.
- Finding **patterns** in creative ways humans would never think to, especially for really **abstract** problems.

Concept 210

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. How do we decide what things are "similar" or "different"? How do we create new classes, and know that they're meaningful?

6.1 Clustering Formalisms

6.1.1 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 211

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 $y^{(i)}$.

6.1.2 What is clustering?

So, if we don't know **what** our classes are, how do we figure out **which** classes to create?
Well, we have to think of what we expect in a class.

Intuitively, we think of a class as a **collection** of things that are **similar** to each other, and more different from other classes.

So, two points in the **same class** are more similar: in RBF, we decided that "more similar" meant "low distance" in the input space.

- **Example:** Two people might look more similar if their heights are numerically closer: shorter distance.

Remember that input space is where we represent each data point using input variables.

Meanwhile, two points in **different classes** are more **distant**: they're further apart in input space.

- **Example:** Two people might look more different if their weights are numerically further: greater distance.

We'll call these "possible classes" that we discover, **clusters**.

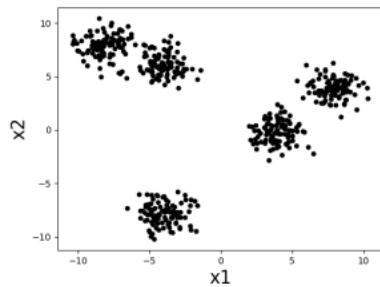
Definition 212

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 how "good" we are at clustering? Which clusters are closest to what we want?

6.2 The k-means formulations

In this section, we'll introduce a common way to do clustering, called the **k-means approach**.

6.2.1 Defining a cluster: The mean

We need to **define** what makes a "cluster" in order to move forward.

Suppose we have a collection of n points, which we informally think of as a "cluster".

We want the points within a cluster to be as **close together** as possible. So, you might ask, "how far is this point $x^{(i)}$ from the rest of the cluster?"

- How do we measure this? Well, we could average the distance to every other point. That could get time-consuming.
- Instead, could we somehow use one point that "**represents**" the whole cluster?
 - With this system, if you want the **distance from the cluster**, we can just use that "representative" as the "position" of our cluster.
 - That way, you only need to compute 1 distance, not $n - 1$ distances.
- This "representative" will be the **average** of all of our points: that way, it'll give us a rough idea of "where" all the points in the cluster are.

This is our **cluster mean**: when we want to know how far a data point is from the rest of the cluster, we'll compare it to the cluster mean.

Definition 213

We want to represent our **cluster** using its **mean**: the **average** of all of the data points in that **cluster**.

- This **cluster mean** will be treated as the "position" of our 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.

6.2.2 k-means

Now, we've created **one** cluster. To extend this to **many** clusters, we just need each cluster to have its **own** mean.

We say that 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 to their cluster. So, we'll assign it to the **closest** one.

Concept 214

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:
 - If they're closer to different cluster means, they're in different groups.
- 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.

6.2.3 k-means loss

Now, we know what we **want** out of our clusters. But, the problem is, we don't know **how** to get those nice clusters.

So, first, we will have to **assign** our initial "cluster means": often, we **randomly** select some points from our dataset.

Concept 215

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** specially designed clusters! They're just a random **initialization**: we have to **optimize**.

Clarification 216

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**.

6.2.4 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 217

In **k-means loss**, we want to minimize the **square distance** from each point $x^{(i)}$ to the **cluster mean** μ .

$$D_i = \|x^{(i)} - \mu\|^2 \quad (6.1)$$

We'll add this up for each of the n data points in our cluster.

$$\mathcal{L} = \sum_{i=1}^n \|x^{(i)} - \mu\|^2 \quad (6.2)$$

6.2.5 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 $\mu^{(j)}$.

$$\overbrace{\mathcal{L}_j}^{\text{Loss for only cluster } j} = \sum_{i=1}^n \|x^{(i)} - \mu^{(j)}\|^2 \quad (6.3)$$

Problem is, we're including **every** point $x^{(i)}$ in **every** cluster! We want a way to filter by **cluster**: we only put one data point in each cluster.

Remember that we **label** clusters the same way we labeled **classes** before:

Notation 218

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 $\mu^{(j)}$ **only** includes points in cluster j . So, when computing **loss**, we **only** want to include data point $x^{(i)}$ when:

$$\underbrace{y^{(i)}}_{x^{(i)} \text{ is in cluster } j} = j \quad (6.4)$$

We'll do this using the following helpful **function**:

Notation 219

The **indicator function** $\mathbb{1}$ tells you whether a statement p is true:

$$\mathbb{1}(p) = \begin{cases} 1 & \text{if } p \\ 0 & \text{otherwise} \end{cases}$$

Combined with our **condition** of matching clusters, this can be useful:

$$\mathbb{1}(y^{(i)} = j) = \begin{cases} 1 & x^{(i)} \text{ is in cluster } j \\ 0 & x^{(i)} \text{ is not in cluster } j \end{cases} \quad (6.5)$$

If we **multiply** this by our loss, it'll filter for situations where the clusters **match!** We can **eliminate** data points in a different cluster.

$$\mathbb{1}(y^{(i)} = j) \|x^{(i)} - \mu^{(j)}\|^2 = \begin{cases} \|x^{(i)} - \mu^{(j)}\|^2 & x^{(i)} \text{ is in cluster } j \\ 0 & x^{(i)} \text{ is not in cluster } j \end{cases} \quad (6.6)$$

6.2.6 k-mean loss: final form

So, we can **filter** by the data points in our cluster:

$$\mathcal{L}_j = \sum_{i=1}^n \underbrace{\mathbb{1}(y^{(i)} = j)}_{\text{Check cluster}} \cdot \underbrace{\|x^{(i)} - \mu^{(j)}\|^2}_{\text{Sq. dist from mean}} \quad (6.7)$$

And finally, we add up over many clusters:

$$\mathcal{L} = \sum_{j=1}^k \mathcal{L}_j \quad (6.8)$$

Using our equation, we get:

$$\mathcal{L} = \underbrace{\sum_{j=1}^k}_{\text{clusters}} \underbrace{\sum_{i=1}^n}_{\text{data points}} \underbrace{\mathbb{1}(y^{(i)} = j)}_{\text{Check cluster}} \cdot \underbrace{\|x^{(i)} - \mu^{(j)}\|^2}_{\text{Dist from mean}}$$

Let's clean that up:

Key Equation 220

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**.

6.2.7 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:

$$\mu^{(j)} = \frac{1}{N_j} \sum_{j=1}^k \underbrace{\mathbb{1}(y^{(i)} = j)}_{\text{check cluster}} \cdot \underbrace{x^{(i)}}_{\text{data points}} \quad (6.9)$$

And how large is N_j ? We can just **count** the number of data points in cluster j :

$$N_j = \sum_{j=1}^k \mathbb{1}(y^{(i)} = j) \quad (6.10)$$

One more loose end: we've been focusing on **square distance** as loss function. We want to

minimize this, but we're doing this over multiple data points.

So, really we want to minimize the **average** of that. This is a very common (and very useful!) property of a distribution called the **variance**.

Definition 221

The **variance** of a dataset is the **average square distance** from the **mean**:

$$\text{Var}[X] = \frac{1}{N} \sum_{i=1}^N \|x^{(i)} - \mu\|^2$$

It tells us how **spread out** our data is.

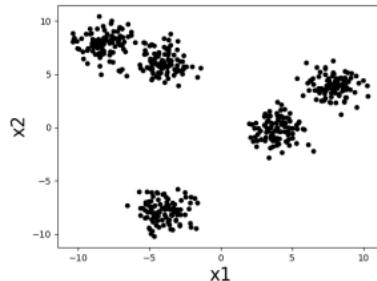
That means, our loss function is meant minimize the total **variance**.

Almost, the factor of $1/N$ is missing: since this constant won't change much as we improve our clustering, we'll leave it alone.

6.2.8 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:



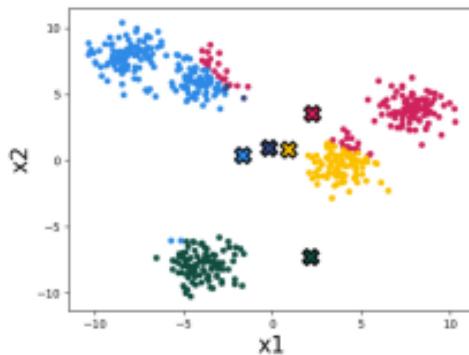
We could cluster this visually, but we want our machine to be able to do it for us.

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:

Above, we suggested selecting a random data point as our cluster mean, but you can also just pick a random position, like we did here.



1) Initial assignments

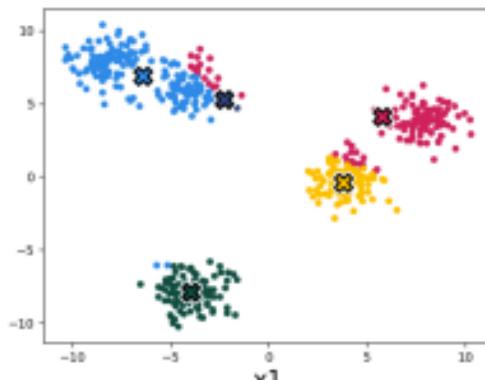
This is our starting point for the algorithm.

6.2.9 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 222

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.

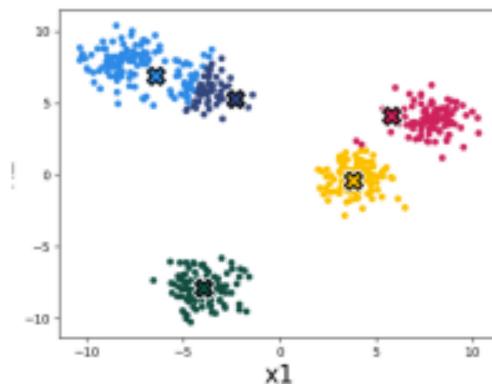
6.2.10 Second step: Reassign data points

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 223

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**.

6.2.11 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 224

Together, both 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.

6.2.12 The k-means algorithm

These two steps make up the **bulk** of our algorithm:

Definition 225

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:

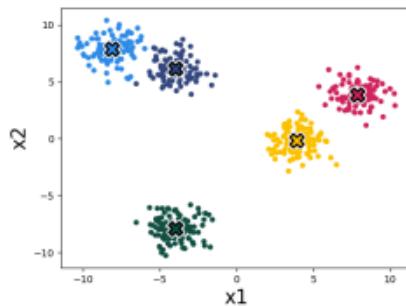


Figure 6.3: Converged result.

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 226

The **k-means algorithm** is guaranteed to **converge** to a **local minimum**.

6.2.13 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  $\mu, y$ 

```

6.2.14 Using gradient descent: minimizing distance to 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 227

We can solve the **k-means problem** using **gradient descent**!

So, we want to **optimize** \mathcal{L} using μ :

$$\mathcal{L}(\mu) = \sum_{i=1}^n \mathbb{1}(y^{(i)} = j) \|x^{(i)} - \mu^{(j)}\|^2 \quad (6.11)$$

Rather than dealing with the indicator function $1(\cdot)$, we could instead just consider whichever μ is closest: **minimum** distance.

$$\underbrace{\min_j}_{\text{Minimizing}} \underbrace{\|x^{(i)} - \mu^{(j)}\|^2}_{\text{distance}} \quad (6.12)$$

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 228

Instead of using an **indicator function** $\mathbb{1}(p)$, 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.

$$\mathcal{L}(\mu) = \sum_{i=1}^n \widehat{\min}_j^{\text{Nearest cluster}} \|x^{(i)} - \mu^{(j)}\|^2 \quad (6.13)$$

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!

6.2.15 Getting labels

Once we've finished gradient descent, and we've **minimized** our loss, we can get our **labels**.

The "min" function gives the **output** value that we get by minimizing. In this case, average **squared distance** from the cluster mean: the **loss**.

Meanwhile, argmin gives us the **input** value that gives us the minimum output. In this case, the **choice of cluster means** 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 229

After **optimizing** μ , our **labels** are given by:

$$y^{(i)} = \arg \min_j \|x^{(i)} - \mu^{(j)}\|^2$$

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.

Of course, this is also true for the k-means algorithm.

Even though individual terms of squared distance may be convex, adding min terms may not be convex.

6.3 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 230

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**.

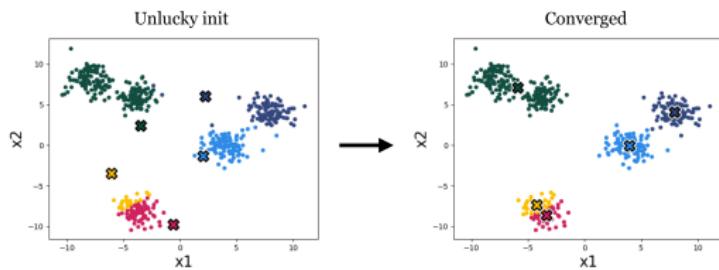
But, always remember that a "good" clustering is partly **subjective**, and depends on what you **want** to accomplish.

6.3.1 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 231

Getting an **unlucky initialization** can result in **clusters** that aren't **useful**.

We try to **solve** this by running our algorithm **multiple times**.

6.3.2 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 232

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 233

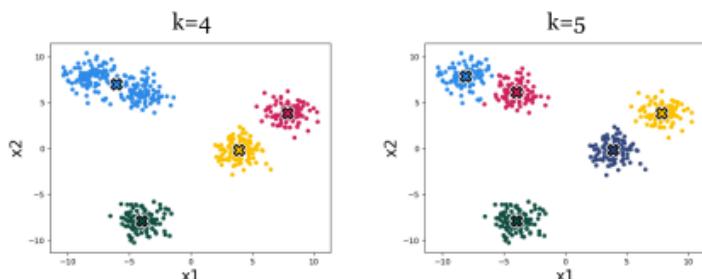
When choosing k (our **number of clusters**), we can cause **problems** by picking an inappropriate **value**:

- **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.

6.3.3 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 234

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).

6.3.4 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.

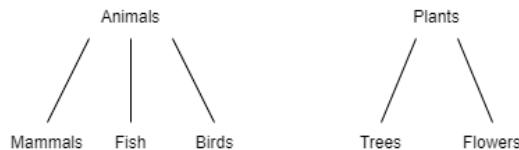
Definition 235

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!

6.3.5 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 236

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**.

6.3.6 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 237

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**.

6.3.7 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 238

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.

6.3.8 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 239

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**.

6.3.9 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 when we can **look** at our data, we can better **understand** what's going on.

In short: clustering allows humans to more easily make sense of data.

Concept 240

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**.

6.3.10 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 241

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** that clustering reveals.

- These "applications" rely on clustering for this improvement.

If our clustering is **good**, we would expect it to **improve** the performance of downstream tasks.

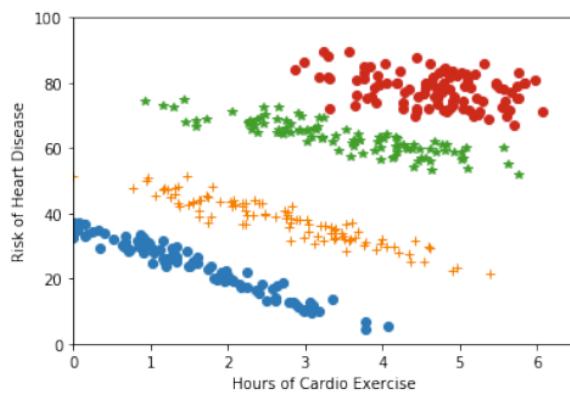
Concept 242

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**.

6.3.11 A benefit of clustering

One advantage for downstream applications is, there might be patterns that are more obvious if you only look at a 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 243

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**.

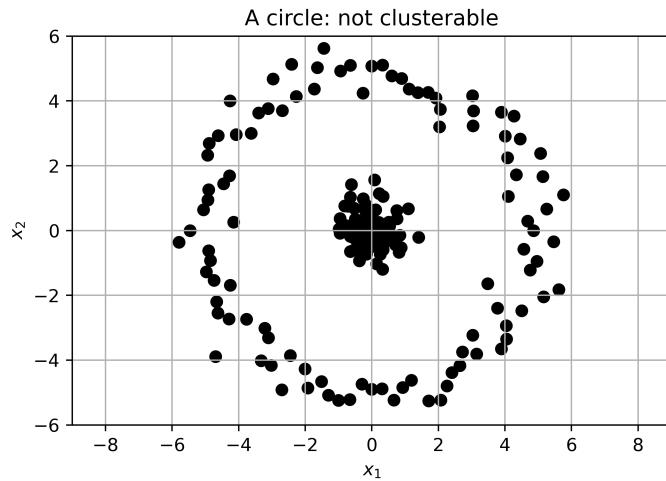
It shows that sometimes, **patterns** that we see may reflect how we're **looking** at the data.

Rest assured, you don't need to know this paradox by **name**. But it's important to **understand** possible problems like it: it'll help you make more responsible judgements in the future.

6.3.12 Weaknesses of k-means

There are some **weaknesses** to k-means clustering. Some patterns that a human eye can see, aren't easily **clustered** by our algorithms.

We can see this with a few **examples**:



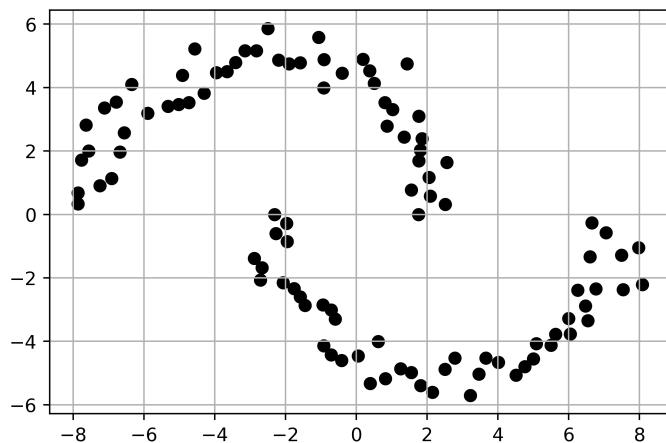
This data clearly seems to have a pattern. But does it have "clusters"?

This example can't be effectively **clustered**: and yet, most people would agree that the "outer ring" should be **one** cluster, while the "inner circle" should be **another**.

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. Or, two rings with different centers.

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.

These sorts of cases are often approached with either

- Attempts to directly visualize them
- Feature transformations
- Other algorithms not discussed here

6.4 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)

CHAPTER 7

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**.

7.0.1 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 discover the **important** information.

As you can see, **neural networks** are used in a wide array of very **difficult** problems. No wonder it's become so popular!

7.0.2 Neural Network Perspectives: The brain

So, what *is* a neural network? There are several perspectives we can take.

First, the **name** comes from the fact that NNs are inspired by the **brain**:

- We call the basic unit 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 244

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.

7.0.3 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.

- Our biggest problem so far is the "nonlinear task": tasks that can't be solved by our **linear** regression/classification models.
- In short: some problems require solutions outside of our **hypothesis space**.

Before, we used **feature representation** to solve this problem. Through the polynomial basis, we found a **richer** hypothesis class.

In this chapter, we present a different (but related) way of creating a richer hypothesis class. In this case, rather than transforming the input, we use a different **structure** for the model.

- 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 much more difficult problems.

Concept 245

Neural Networks make up a very **rich** hypothesis class, by combining many simple **units**.

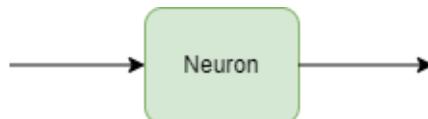
With this **hypothesis class**, we can handle **regression** or **classification** for very challenging **problems**.

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!

7.0.4 Building up a basic neural network

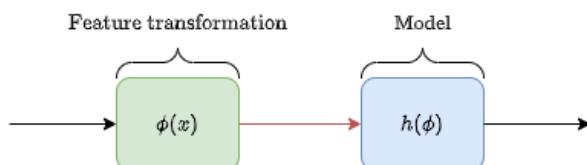
Let's make sense of what we said above, and **visualize** what a neural network might look like.

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!

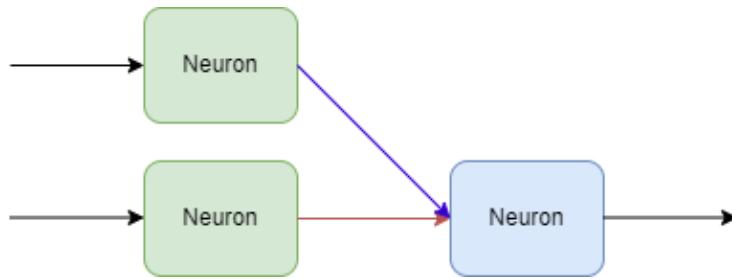
Last Updated: 11/08/23 21:00:09

Feature transformations, like polynomial or radial basis, are a bit more **complex** than 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!

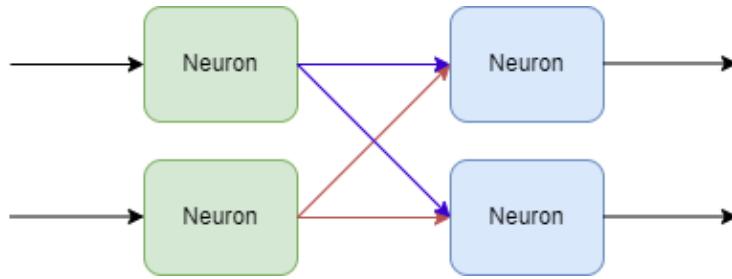
But, there's another type of **complexity** we haven't explored: we could have two neurons in **parallel**.



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 we can develop with simple instructions".

Definition 246

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.

7.0.5 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 247

Neural networks can be seen as a way to make **predictions** based on huge amount of **data** for very **complex** problems.

7.1 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 248

Neurons are also sometimes called **units** or **nodes**.

They are mostly **equivalent** names. They just reflect different **perspectives**.

7.1.1 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

$$h(x) = \theta^T x + \theta_0 \quad (7.1)$$

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 249

Our **neuron** contains a **linear** function as its **first** component.

7.1.2 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:

$$\overbrace{f_1(x)}^{\text{Neuron 1}} = Ax + B \quad \overbrace{f_2(x)}^{\text{Neuron 2}} = Cx + D \quad (7.2)$$

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

$$g(x) = 100x_1 + 2x_2 \quad (7.3)$$

Then, changing x_1 will have a much **bigger** effect on $g(x)$. Another way to say this is it **weights** more heavily: it matters **more**.

Because of that, we call the number we scale x_1 by a **weight**.

Notation 250

A **weight** w_k tells you how heavily a **variable** x_k **weights** 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 251

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 252

The **linear component** for a neuron is given by

$$z(x) = w^T x + w_0$$

where $w \in \mathbb{R}^m$ and $w_0 \in \mathbb{R}$.

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix}$$

7.1.3 Linear Diagram

Now, we want to be able to depict our **linear** subunit. Let's do it piece-by-piece.

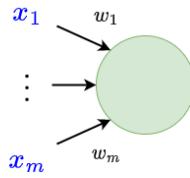
First, we have our vector $x = [x_1, x_2, \dots, x_m]^T$:

x_1

\vdots

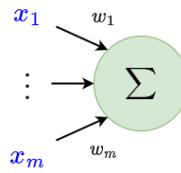
x_m

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.

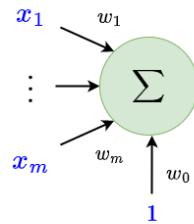
How are we combining them? Well, we're adding them together.



Note that we use the \sum symbol, because we're **adding** after we **multiply**. In fact, we can write this as

$$\mathbf{w}^T \mathbf{x} = \sum_{i=1}^m w_i x_i \quad (7.4)$$

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.

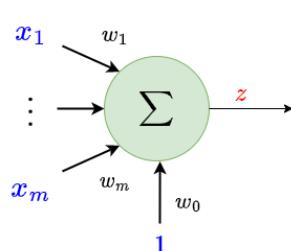


The blue "1" term is **multipled** by w_0 , just like how x_k gets multiplied by w_k .

We have our full function! All we need to do is include our output, z :

Notation 253

We can depict our linear function $z = \mathbf{w}^T \mathbf{x} + w_0$ as



Thus, z is a function of x :

$$z(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0 \quad (7.5)$$

Which, in \sum notation, we could write as

$$z(\mathbf{x}) = \left(\sum_{i=1}^m w_i x_i \right) + w_0 \quad (7.6)$$

7.1.4 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**.

You might remember that we had a problem with the logistic linear model still behaving linear, despite having a nonlinear function.

We'll show how we fix this later on.

Some activation functions reflect this, but they don't have to.

Definition 254

Our **neuron** contains a potentially **nonlinear** function f , called an **activation function**, as its **second** component.

We note this as

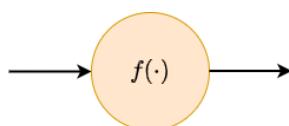
$$a = f(z)$$

Where z is the **output** of the **linear** component, and a is the **output** of the **activation** component.

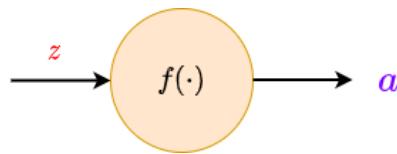
Note that z and a are **real numbers**: we have $f : \mathbb{R} \rightarrow \mathbb{R}$

7.1.5 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 255

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 256

a , the **output** of our **activation** function, is called the **activation**.

7.1.6 Putting it together

So now, our neuron is complete.

Definition 257

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

$$a = f(z) = f(w^T x + w_0)$$

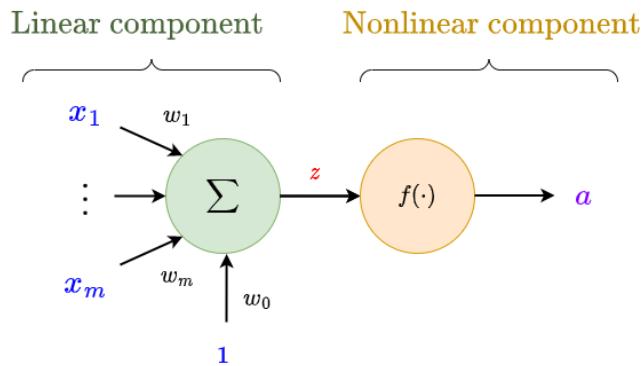
We can also use \sum notation to get:

$$a = f(z) = f\left(\left(\sum_{i=1}^m w_i x_i\right) + w_0\right)$$

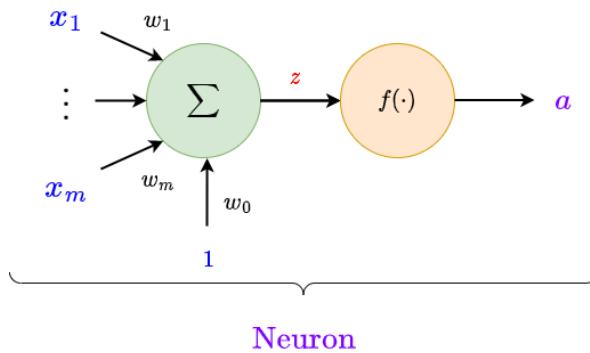
When we say "compose", we mean **function composition**: combining $f(x)$ and $g(x)$ into $f(g(x))$.

7.1.7 Neuron Diagram

Finally, we can **compose** our neuron into one big **diagram**:

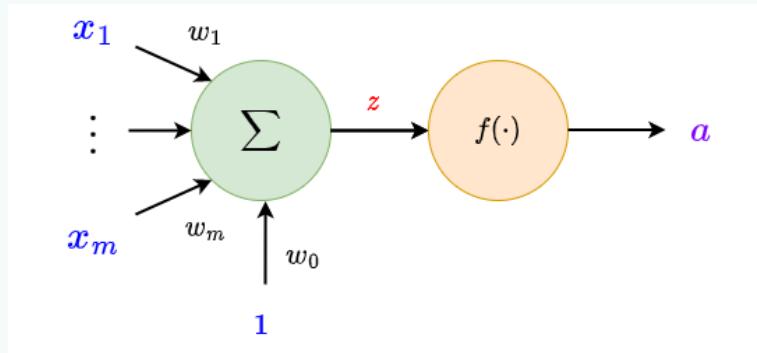


From here on out, we'll treat this as a **single** object:



Notation 258

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.

7.1.8 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

$$\mathcal{L}(g, y) \tag{7.7}$$

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.

$$\sum_i \mathcal{L} \left(h(\mathbf{x}^{(i)}; (\mathbf{w}, w_0)), \mathbf{y}^{(i)} \right)$$

And with this, not only is our neuron **complete**, but we have everything we need to **work** with it.

Concept 259

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**.

7.1.9 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: the most simple version that's useful. 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 260

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**:

$$f(g(x)) = g(x) \quad (7.8)$$

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 261

Linear Regression can be represented with a **single neuron** where

- We keep our **linear component**, but set $(\theta, \theta_0) = (w, w_0)$.

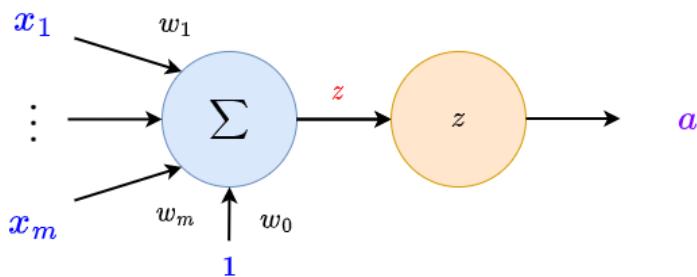
$$z(x) = w^T x + w_0$$

- Our **activation function** is the **identity** function,

$$f(z) = z$$

- Our **loss function** is **quadratic loss**.

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



7.1.10 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:

$$\textcolor{red}{z} = \theta^T x + \theta_0 \quad (7.9)$$

And then, the **logistic** component:

$$\sigma(\textcolor{red}{z}) = \frac{1}{1 + e^{-\textcolor{red}{z}}} \quad (7.10)$$

This second part is **nonlinear**: it's our **activation** function!

Concept 262

A **Linear Logistic Classifier** can be represented with a **single neuron** where

- We keep our **linear component**, but set $(\theta, \theta_0) = (w, w_0)$.

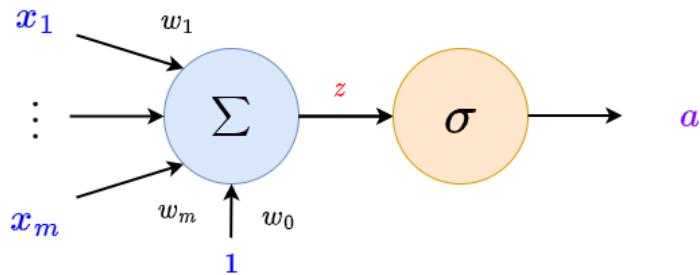
$$\textcolor{red}{z}(x) = w^T x + w_0$$

- 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)

$$\mathcal{L}_{\text{nll}}(\textcolor{violet}{a}, \textcolor{blue}{y}^{(i)}) = - \left(\textcolor{blue}{y}^{(i)} \log \textcolor{violet}{a} + (1 - \textcolor{blue}{y}^{(i)}) \log (1 - \textcolor{violet}{a}) \right)$$



7.2 Networks

Now, we have fully developed the individual **neuron**.

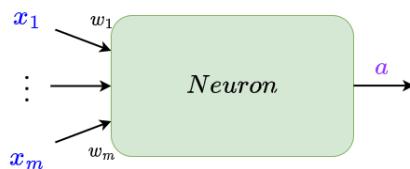
We can even do **gradient descent** on it: just like when we were doing LLCs, we can use the **chain rule**.

So, we return to the idea from the beginning of this chapter: combining multiple neurons into a **network**.

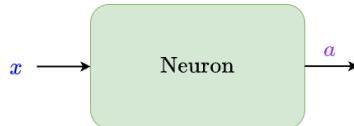
We'll get into this more, later in the chapter.

7.2.1 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 263

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".

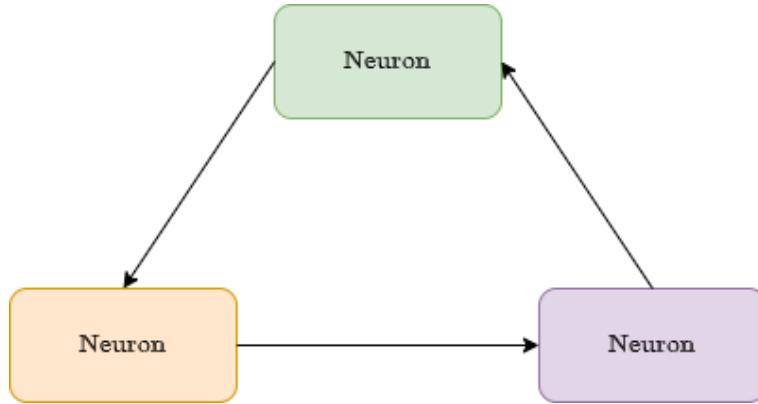
Example: Rather than thinking about all of the **mechanics** of how a car works, you might **abstract** it down to the pedals, the steering wheel, and how that causes the car to move.

7.2.2 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 264

For simple **neural networks**, we assume that they are **acyclic**: there are no **cycles**, or loops.

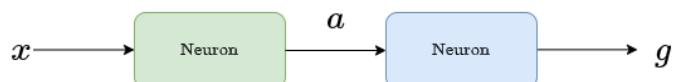
This means that **no neuron** has an output that affects its **input**, directly or indirectly.

We call these **feed-forward** networks: information can only go "forward", not "backward".

We'll show how to build up the rest of what we need.

7.2.3 How to build networks

Suppose we have two neurons in **series**, our **simplest** arrangement:



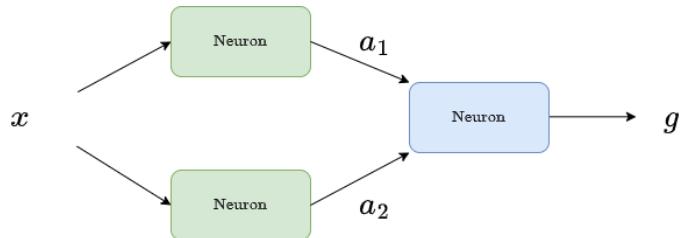
Our first neuron takes in a whole **vector** of values, $x = [x_1, x_2, \dots, x_m]^T$. But, it only **outputs** a single value, a .

- That means the second neuron only receives **one** value.

Remember that while we only see one arrow from x , each data point x_i is included.

But, just like our first neuron, it's capable of handling a full **vector**. We can add more values!

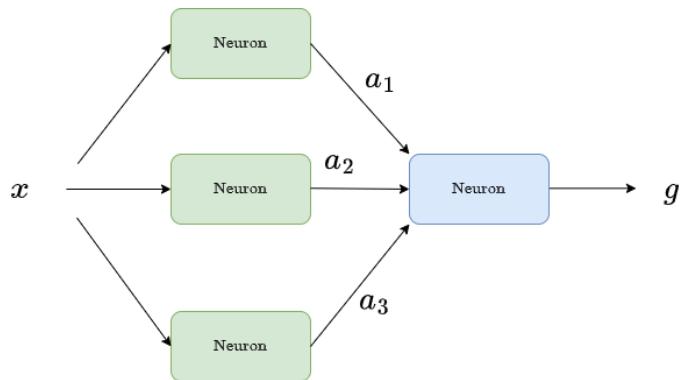
Let's add **another** neuron.



Our rightmost neuron now has **2 inputs**, which can be stored in a vector,

$$\mathbf{A} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \quad (7.11)$$

We could increase the **length** of this vector by adding more **neurons**.



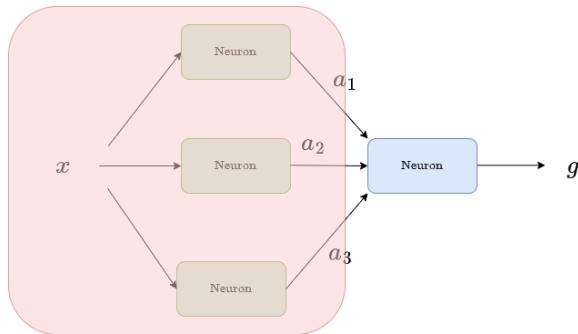
$$\mathbf{A} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad (7.12)$$

For our **rightmost** neuron, this is effectively the **same** as x : an **input vector**.

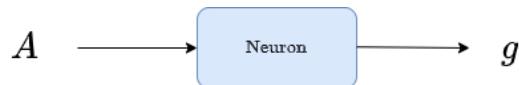
7.2.4 Layers

This gives us an idea for how to **build** our network: using multiple neurons in **parallel**, we can output a new vector \mathbf{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**.

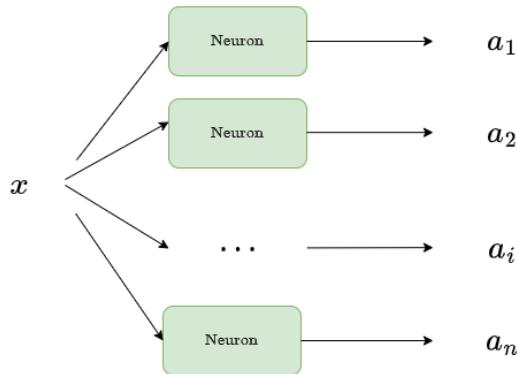
Definition 265

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 the neurons in the same layer 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.

7.2.5 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 266

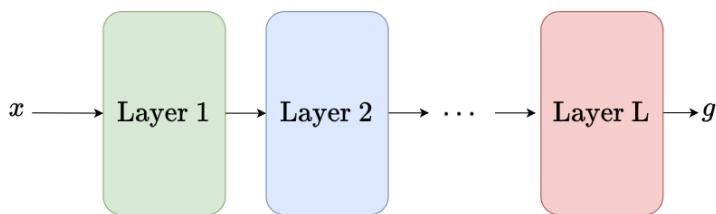
We structure our **neural networks** as a series of **layers**, where each layer is the **input** to the next layer.

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:

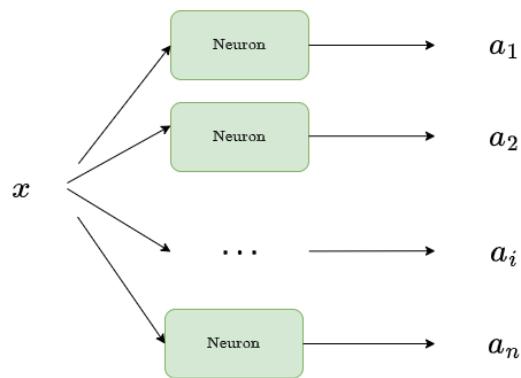


A neural network.

We now have a high-level view of our entire neural network, so now we dig into the details of a single layer.

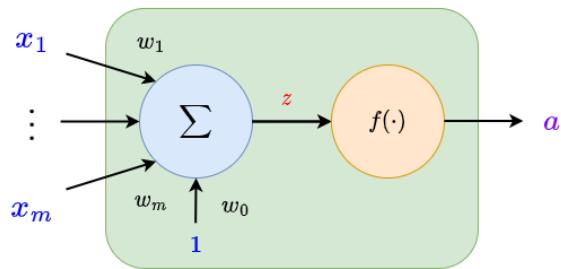
7.2.6 Single Layer: Visualizing our Components

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**:

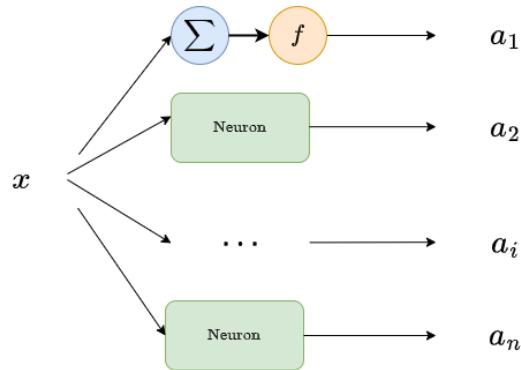


Each of those neurons looks like this.

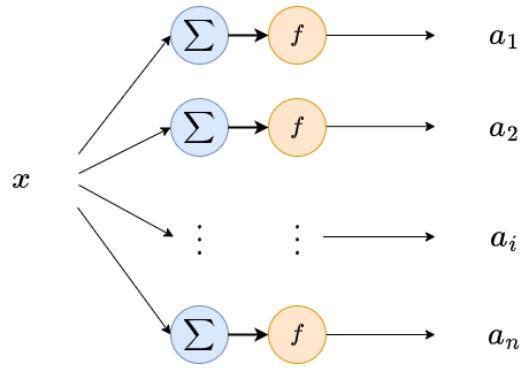
There are two important pieces of **information** we're hiding:

- 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.



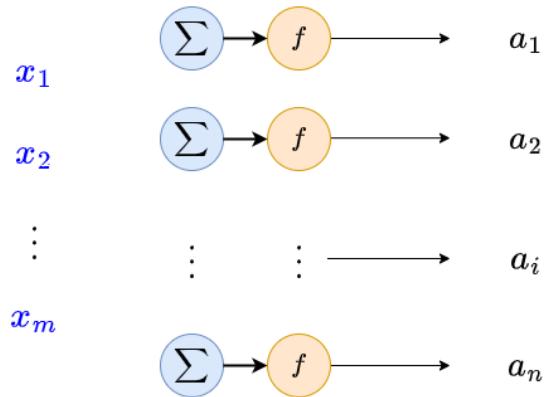
Replacing one neuron...



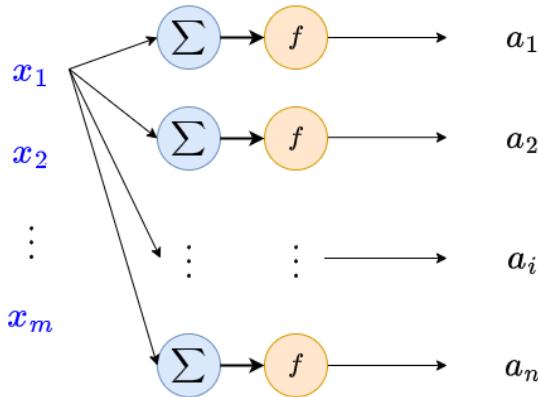
Replacing all neurons!

7.2.7 Single Layer: Visualizing our Inputs

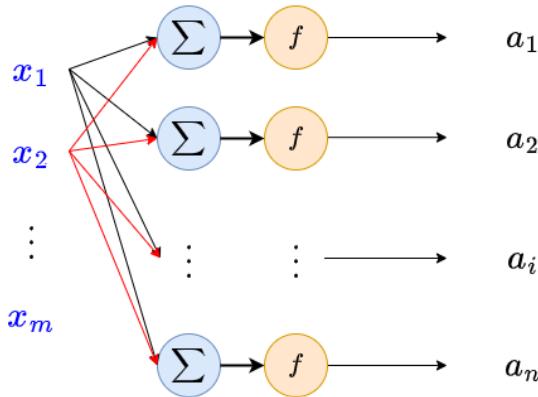
The second piece of information is much more difficult: we show all of the x_i outputs.



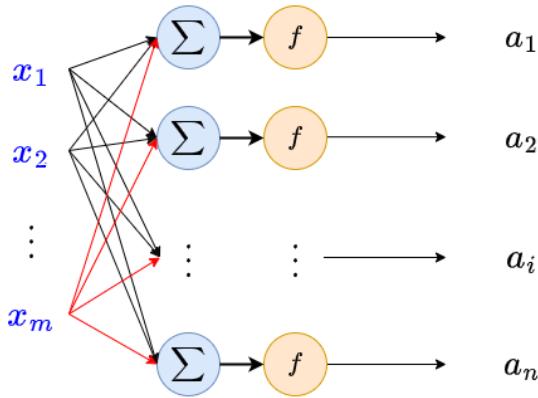
Now we have to draw the arrow for each input.



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 267

A layer is **fully connected** if every neuron has the **same input vector**.

In other words, every neuron in our layer is connected to every input value.

Example: If one of our neurons **ignored** x_1 , but the others did **not**, the layer would not be **fully connected**.

7.2.8 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 268

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 . Input dimension 3, output dimension 1.

Thus, our output dimension has been separated from our input dimension. Instead, it is the number of neurons.

So, in general, we can say:

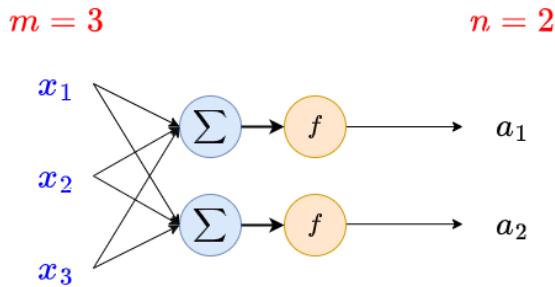
Notation 269

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:
 $A \in \mathbb{R}^n$

These dimensions can be any pair of numbers: the value of m doesn't affect the value of n .

Example: Suppose you have an **input** vector $x = [x_1, x_2, x_3]$ and two **neurons**. The dimensions are $m = 3$, and $n = 2$.



The input dimension and output dimensions are **separate**.

7.2.9 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

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} \quad A = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \quad (7.13)$$

But, there are a couple other things we haven't **generalized** for our entire **layer**:

- Our weights

- Our offsets
- Our preactivation

7.2.10 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 $w^T x$.

$$w = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} \quad (7.14)$$

To distinguish them from each other, we'll represent the i^{th} neuron's weights as \vec{w}_i .

$$\vec{w}_i = \begin{bmatrix} w_{1i} \\ w_{2i} \\ \vdots \\ w_{mi} \end{bmatrix} \quad (7.15)$$

Each weight 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:

$$W = \underbrace{\begin{bmatrix} \vec{w}_1 & \vec{w}_2 & \cdots & \vec{w}_n \end{bmatrix}}_{\text{Each neuron has a weight vector}} \quad (7.16)$$

If we expand it out, we get a full matrix...

$$W = \underbrace{\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m1} & \cdots & w_{mn} \end{bmatrix}}_{\text{n neurons}} \Bigg\} \text{m inputs} \quad (7.17)$$

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:

This is our offset vector, with the shape $(n \times 1)$.

Notation 270

We can store our **weights** and **offsets** as **matrices**:

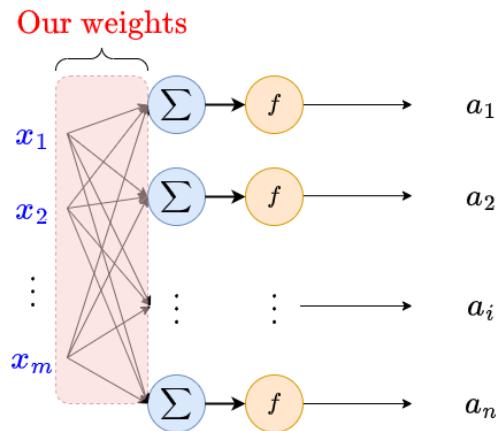
- **Weight** matrix W has the shape $(m \times n)$

$$W = \underbrace{\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{bmatrix}}_{m \text{ inputs}} \overbrace{\quad}^{n \text{ neurons}}$$

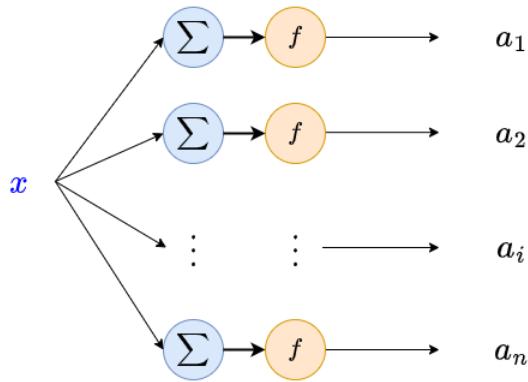
- **Offset** matrix W_0 has the shape $(n \times 1)$

$$W_0 = \underbrace{\begin{bmatrix} w_{01} \\ w_{02} \\ \vdots \\ w_{0n} \end{bmatrix}}_{\text{Each neuron has an offset}}$$

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.



7.2.11 Pre-activation

Now, all that remains is the pre-activation z .

Before, we did

$$w^T x + w_0 = z \quad (7.18)$$

Because we so carefully kept our weights and offsets separate, we can still do this!

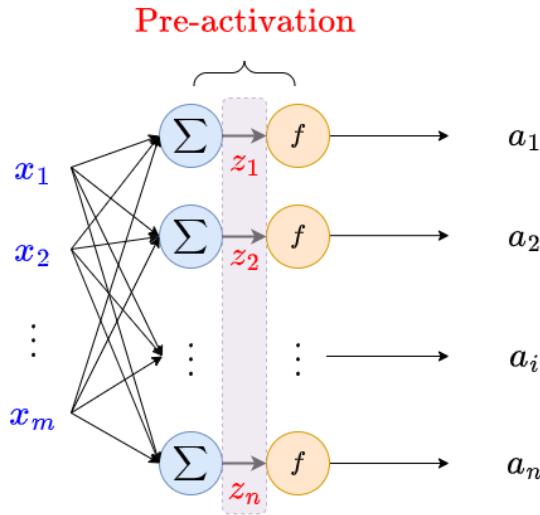
$$W^T x + W_0 = Z \quad (7.19)$$

You can check for yourself that this behaves the way you expect it to.

This pre-activation vector Z contains all of the outputs of our linear components:

$$Z = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} \quad (7.20)$$

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 :

$$A = f(Z) = \begin{bmatrix} f(z_1) \\ f(z_2) \\ \vdots \\ f(z_n) \end{bmatrix} \quad (7.21)$$

Because we run the function on each element in Z , we call this an **element-wise** use of our function.

7.2.12 Summary of a layer

So, we can now break our our layer up into pieces:

Notation 271

Our **layer** is a **function** that takes in $x \in \mathbb{R}^m$, and returns $A \in \mathbb{R}^n$.

It is defined by:

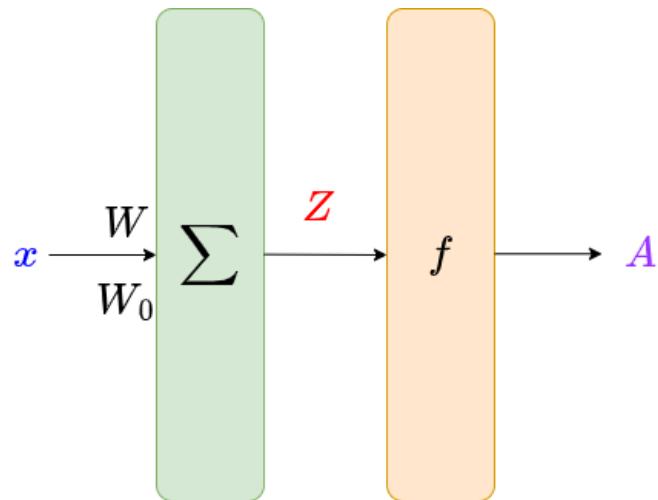
- **Dimensions:** m for **input**, n for **output** (number of neurons)

And our different **matrices**:

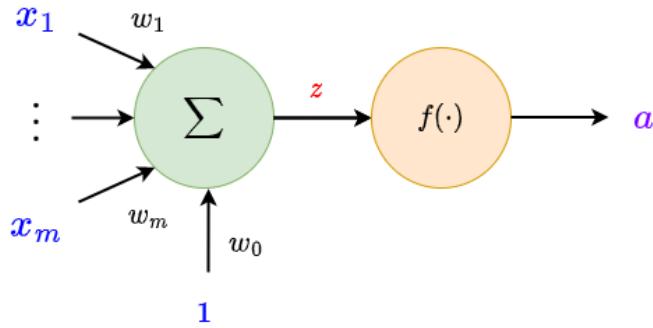
- **Input:** a **column vector** X in the shape $(m \times 1)$
- **Weights:** a **matrix** W in the shape $(m \times n)$
- **Offset:** a **column vector** W_0 in the shape $(n \times 1)$
- **Pre-activation:** a **column vector** Z in the shape $(n \times 1)$
- **Activation:** a **column vector** A in 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 272

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

$$A = f(Z) = f(W^T x + W_0)$$

7.2.13 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:

$$z = w^T x + w_0 = 0 \quad (7.22)$$

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

$$\sigma(z) = \sigma(0) = \frac{1}{2} \quad (7.23)$$

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!

$$\text{Linear boundary } z = 0 \iff f(z) = \frac{1}{2} \quad (7.24)$$

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.

$$z = C \iff f(z) = f(C) \quad (7.25)$$

Since $z = C$ is linear, we know that our activation separator $f(x) = f(C)$ is linear too.

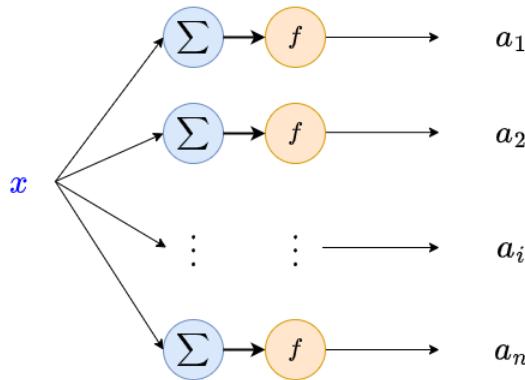
Concept 273

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.

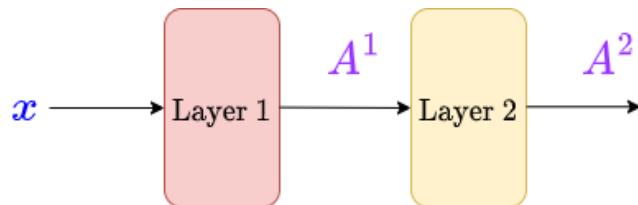
Concept 274

A **single layer** of neurons has **linear** behavior.

We need **multiple** layers to get a nonlinear **neural network**.

7.2.14 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 275

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".

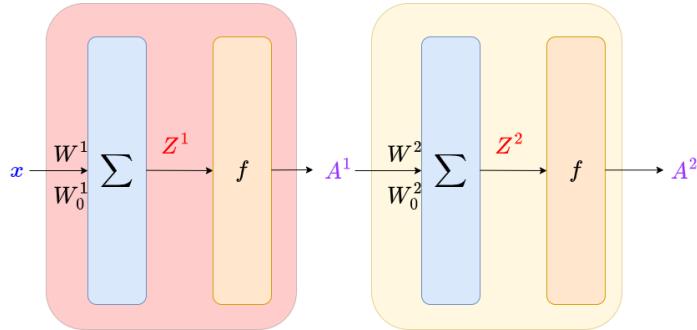
What can we learn from this?

- 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:

$$n^1 = m^2 \quad (7.26)$$

Let's break these into their components again.



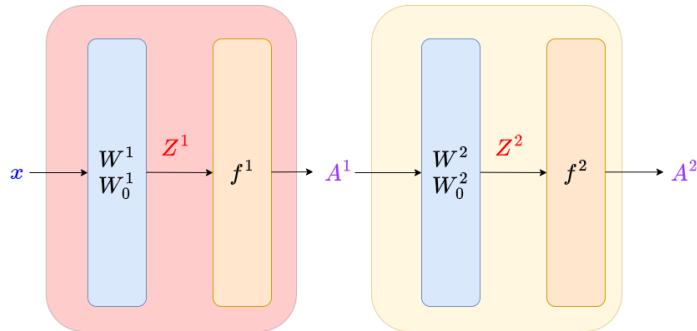
We have two separate outputs: A^1 and A^2 .

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:

$$A^1 = f(Z^1) = f((W^1)^T x + W_0^1) \quad (7.27)$$

And layer two:

$$A^2 = f(Z^2) = f((W^2)^T A^1 + W_0^2) \quad (7.28)$$

We can use this to build our **general** pattern.

7.2.15 Many Layers

We are finally ready to build our **complete** neural network. We'll just retrace the steps of the 2-layer case.

Notation 276

The total **number** of **layers** in our **neural network** is notated as L .

Typically we notate an **arbitrary** layer as ℓ (or l).

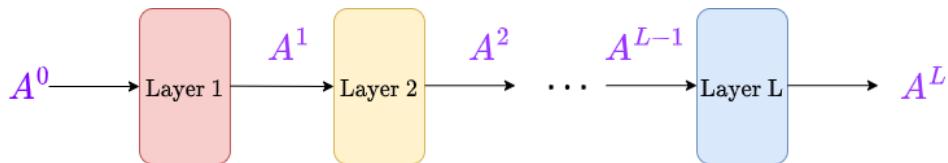
Since x is, for all purposes, **equivalent** to a vector A , we will call it A^0 .

Notation 277

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 278

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.

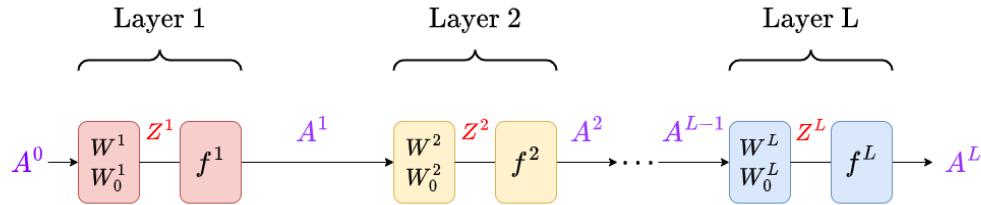
$$\underbrace{n^\ell}_{\text{Output}} = \underbrace{m^{\ell+1}}_{\text{Output}}$$

And the **dimension** of A^ℓ is $(n^\ell \times 1) = (m^{\ell+1} \times 1)$.

7.2.16 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 279

The calculations done by layer ℓ are broken into two parts:

- Input $A^{\ell-1}$ turns into pre-activation Z^ℓ

$$Z^\ell = (\mathbf{W}^\ell)^T A^{\ell-1} + \mathbf{W}_0^\ell$$

- Pre-activation Z^ℓ turns into A^ℓ

$$A^\ell = f(Z^\ell)$$

Which combine into:

$$A^\ell = f(Z^\ell) = f\left((\mathbf{W}^\ell)^T A^{\ell-1} + \mathbf{W}_0^\ell\right)$$

7.2.16.1 Hidden Layers and the "First Layer"

Now that we have a full network, we introduce some useful vocab.

Definition 280

A **hidden layer** is any functional layer except for the **output** (last) layer.

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)

You can't see the output of the **hidden layers** from outside the network.

Based on this definition, the **number of hidden layers** in a network is the layer count, minus one: $L - 1$.

Note that there's one point of confusion: online, you may see that the hidden layer is "any layer other than the **input** (first) or **output** (last) layer".

This is because, often, we consider the input itself to be a separate "**input layer**".

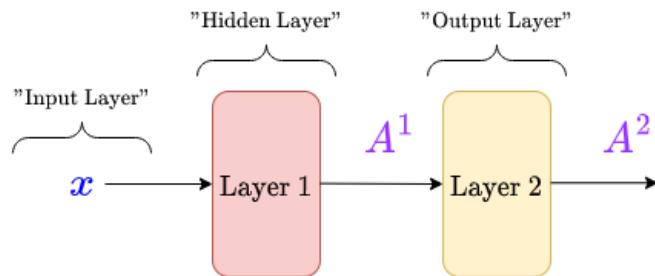
Despite this fact, when someone counts the number of layers in a neural network, they're usually only counting the hidden and output layers: we **don't count** the input layer. It confused me, too.

Definition 281

The **input layer** is a layer that brings the **input** into the network. It applies **no functions** to the data.

Because the input layer has **no effect** on our data (it just moves it), we **don't count the input layer** when we're saying how **many layers** a network has.

Example: Consider the following network from earlier:



In this network, x is passed into the network by the **input layer**. This layer is **before** layer 1 (you could think of it as "Layer 0").

Despite having the input layer, plus layer 1 and 2, we count only

- **Two** layers in our network:

- One hidden layer: Layer 1.
- One output layer: Layer 2.

7.3 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?

7.3.1 Trying out linear activation

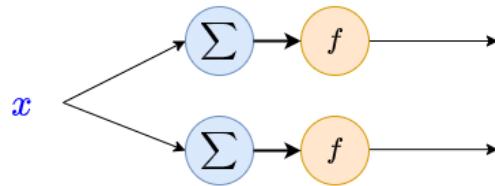
The simplest assumption would be to just use the **identity** function

$$f(z) = z \quad (7.29)$$

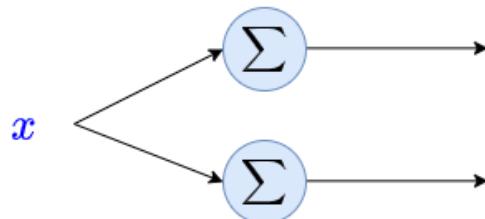
We might hope that we can combine a bunch of simple, **linear** models, and get a more sophisticated model. Why bother having a **nonlinear** activation at all?

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.

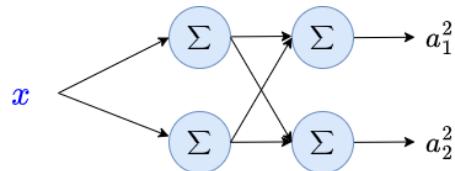
Let's look at layer 1:



Since the activation function has **no effect** on our result, we can **omit** it:

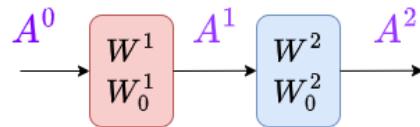


And now, we can show our **full** network:



7.3.2 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: _____



In each layer, we're "combining" the linear component with the linear activation:
linear * linear=linear.

We'll leave out W_0 terms to make it more readable, but the same will apply.

Layer 1:

$$A^1 = (\textcolor{red}{W}^1)^T \textcolor{blue}{A}_0 \quad (7.30)$$

Layer 2:

$$A^2 = \overbrace{(\textcolor{blue}{W}^2)^T (\textcolor{red}{W}^1)^T}^{\text{Weight matrices}} \textcolor{blue}{A}_0 \quad (7.31)$$

The full function for this equation is two matrices, **multiplied** by our input vector.

Let's take an arbitrary example:

$$\textcolor{red}{W}^1 = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \quad \textcolor{blue}{W}^2 = \begin{bmatrix} 5 & 6 \\ 7 & 8 \end{bmatrix} \quad (7.32)$$

Our equation becomes:

$$A^2 = \overbrace{\begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}}^{\text{Transposed matrices}} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad (7.33)$$

We created this function by applying two matrices separately. But, can't we **combine** them?

$$A^2 = \begin{bmatrix} 19 & 43 \\ 22 & 50 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad (7.34)$$

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...

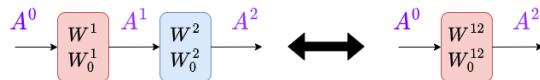
$$(\textcolor{blue}{W}^{12})^T = \begin{bmatrix} 19 & 43 \\ 22 & 50 \end{bmatrix} \quad (7.35)$$

7.3.3 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**.

$$(\mathbf{W}^2)^T (\mathbf{W}^1)^T = \mathbf{W}^{12} \quad (7.36)$$

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:

$$(\mathbf{W}^L)^T (\mathbf{W}^{L-1})^T \dots (\mathbf{W}^2)^T (\mathbf{W}^1)^T = \mathbf{W} \quad (7.37)$$

And so, we can't just use linear layers: we **need** a **nonlinear** activation function.

Concept 282

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.

This problem is even worse than it seems: let's see why. Since we can **combine** **n** linear layers together into one, what happens if we only have **one** linear layer?

Suppose layer ℓ is linear. The next layer contains a **linear** component and a non-linear **activation** component. We'll focus on just the linear part.

$$z^{\ell+1} = (\mathbf{W}^{\ell+1})^T \mathbf{x}^{\ell+1} = (\mathbf{W}^{\ell+1})^T (\mathbf{W}^\ell)^T \mathbf{x}^\ell \quad (7.38)$$

Activation comes after this step, so we would just use $f(z^{\ell+1})$.

Wait: we have **two** consecutive **linear** components. We can combine layer ℓ with the linear component of the next layer!

$$(\mathbf{W}^{\ell+1})^T (\mathbf{W}^\ell)^T \mathbf{x}^\ell = \mathbf{W} \mathbf{x}^\ell \quad (7.39)$$

Now, we've removed layer ℓ entirely: it makes no difference to have even just one **hidden** linear layer!

Concept 283

Even having one hidden **linear layer** is **redundant**: it's **equivalent** to not having that layer at all.

Since this requires **more computation** for no benefit, we **almost never** make linear hidden layers.

So, linear models are out. What if we use something **nonlinear**?

$$A^2 = f((W^2)^T A^1) \quad (7.40)$$

We get something that doesn't seem to **simplify**:

This is ugly, but we don't have to worry about the details.

$$A^2 = f((W^2)^T \boxed{f((W^1)^T x)}) \quad (7.41)$$

If we choose our function right (and avoid linearity), this cannot be simplified to a single layer! That means, this function is **different** (and likely more **complex**) than a one-layer model.

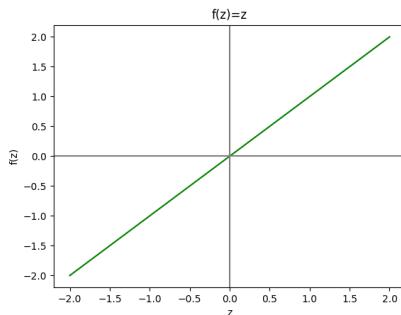
And this kind of **expressiveness** is exactly what we're looking for.

7.3.4 Example of Activation Functions

So, let's look at some possible **activation** functions:

- **Identity** function z :

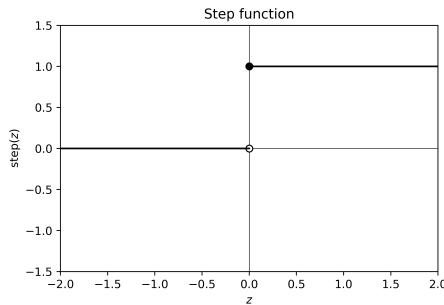
$$f(z) = z \quad (7.42)$$



- This function is called an **identity** function because it "preserves the identity" of the input: the output is the same.

- This is an example of a **linear** function.
 - * As we described in the last section, linear activation can't make our model more **expressive**.
 - * So, we **almost never** use it (or any other **linear** function) as an activation for a **hidden** layer.
- We mainly use this as an **output** activation function: it allows our final output to be any real number.
 - * This is a good activation function for a **regression** model, which returns a **real** number.
 - * It's a simple function, that can return **any** real number. By contrast, sigmoid and ReLU both have **limited** output ranges.
- **Step** function $\text{step}(z)$:

$$\text{step}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \quad (7.43)$$

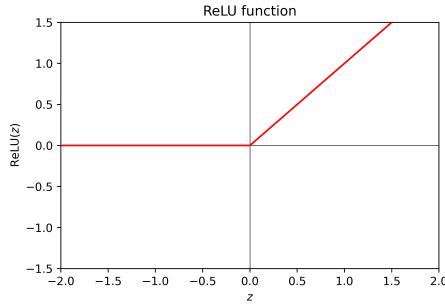


- This function is basically a **sign** function, but uses $\{0, 1\}$ instead of $\{-1, +1\}$.
- Step functions were a common early choice, but because they have a **zero** gradient, we can't use **gradient descent**, and so we basically **never** use them.

- **Rectified Linear Unit** $\text{ReLU}(z)$:

Same reason we replaced the sign function with sigmoid, when we were doing linear logistic classifiers.

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \quad (7.44)$$

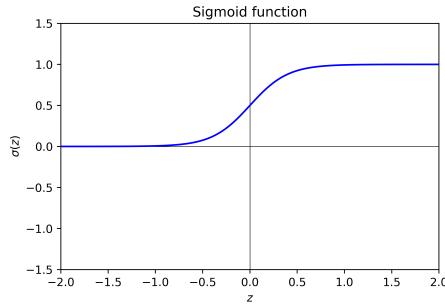


- This is a very **common** choice for activation function, even though the derivative is undefined at 0.
- We specifically use it for internal ("**hidden**") layers: layers that are neither the **first** nor **last** layer.

- **Sigmoid** function $\sigma(z)$:

They're "hidden" because they aren't visible to the input or output.

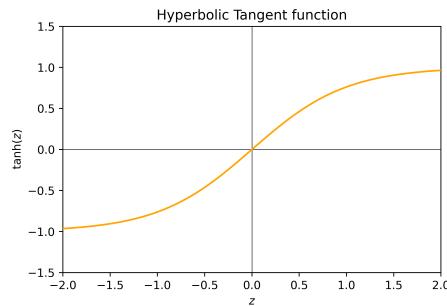
$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (7.45)$$



- This is the **activation** function for our **LLC** neuron from before.
- Just like LLC, it's useful for the **output neuron** in **binary classification**.
- Can be interpreted as the **probability** of a positive (+1) binary classification.
- We can also use this for multiclass when classes are **NOT** disjoint: we use one sigmoid per class.
 - * Each sigmoid tells us how likely the data point is to be in that class.

- **Hyperbolic Tangent** $\tanh(z)$:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \quad (7.46)$$



- This function looks similar to sigmoid over a different **range**.
- Unfortunately, it will not get much use in this class.
- **Softmax** function $\text{softmax}(z)$:

$$\text{softmax}(z) = \begin{bmatrix} \exp(z_1) / \sum_i \exp(z_i) \\ \vdots \\ \exp(z_n) / \sum_i \exp(z_i) \end{bmatrix} \quad (7.47)$$

- Behaves like a disjoint, **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.
 - * "Disjoint" probability: each option is separate. Sum of the rows adds up to 1.

Concept 284

For the different **activation functions**:

- $f(z) = z$ isn't used for **hidden** layers, but we can use it for regression **output**.
- $\text{sign}(z)$ 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.

Remember this caveat, though:

Clarification 285

Multi-class depends on whether a **data point** can be in **multiple classes at the same time**.

- $\text{softmax}(z)$ assumes our classes are **disjoint**: you can only be in **one** class.
 - This is usually what people mean by **multi-class**.
- $\sigma(z)$ can be used when classes are **not disjoint**: you can be in **multiple** classes.
 - You can think of this as **binary classification** for each class.

When using sigmoids, we need **one** sigmoid for each **class**.

Example: We can compare use cases for each of these:

- Softmax could be used to answer, "which word is the next one in the sentence?"
 - Every word in a sentence is only followed by one word: they're mutually exclusive.
- Sigmoids could be used to answer, "what genre of book is that?"
 - A book is often in more than one genre.

7.4 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 286

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	$\sum_j y_j \log(g_j)$

Special Case: If we allow **multiple** classes at the **same** time (non-disjoint), we use **binary** classification for each of them, rather than multi-class.

Example: An example for each type:

- **Regression:** Predicting the amount of rainfall in centimeters tomorrow.
- **Binary Classification:** Will the stock market go up or down tomorrow?
- **Multi-Class:** What species of tree is this?
- **Multiple Binary:** What are the themes in this movie?



7.4.1 Other Considerations

You might consider using other functions, based on the needs of a more specialized task. We'll ignore those cases, for the most part.

But, if you want to try a new function, the **data type** is the most important for whether we can use it.

Concept 287

If you want to use a new **activation** or **loss** function, you have to pay attention to the **input/output** type.

Example: $\tanh(z)$ outputs over the range $(-1, 1)$. We could use it, if that was the range we wanted.

Be careful, though:

Clarification 288

It's important to stress that while our **output activation** depends on the task, **hidden layers** don't have to.

Hidden layers can use one of several **different** activation functions, regardless of the **task**.

However, some activation functions tend to be **better** for making a model than others.

Example: Often, we use ReLU for hidden layers, but it's rarely used as an output activation function.

We also might use **sigmoid** as a hidden layer for a regression model, even though regression most commonly uses a **linear** output.

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 7

Neural Networks 1.5 - Back-Propagation and Training

7.5 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 289

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.

7.5.1 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 290

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

$$\Delta\theta = -\eta \frac{\partial \mathcal{L}}{\partial \theta} \quad (7.1)$$
- We take steps $\Delta\theta$ (and $\Delta\theta_0$) until we are satisfied with \mathcal{L} , or it **stops** improving.

7.5.2 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 .

$$z(x) = w^T x + w_0 \quad g(z) = \sigma(z) = \frac{1}{1 + e^{-z}} \quad (7.2)$$

$$\mathcal{L}(g, y) = y \log(g) + (1 - y) \log(1 - g) \quad (7.3)$$

Our goal is to minimize \mathcal{L} by adjusting θ and θ_0 .

So, we want

$$\frac{\partial \mathcal{L}}{\partial w} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial w_0} \quad (7.4)$$

We did this by using the **chain rule**:

We'll focus on w , but the same goes for w_0 .

$$\frac{\partial \mathcal{L}}{\partial w} = \overbrace{\frac{\partial \mathcal{L}}{\partial g}}^{\mathcal{L}(g)} \cdot \overbrace{\frac{\partial g}{\partial w}}^{\text{7.5}}$$

We can break it up further using **repeated chain rules**:

$$\frac{\partial \mathcal{L}}{\partial w} = \overbrace{\frac{\partial \mathcal{L}}{\partial g}}^{\mathcal{L}(g)} \cdot \underbrace{\frac{\partial g}{\partial z}}_{g(z)} \cdot \overbrace{\frac{\partial z}{\partial w}}^{\text{7.6}}$$

Plugging in our derivatives, we get:

$$\frac{\partial \mathcal{L}}{\partial w} = -\left(\frac{y}{\sigma} - \frac{1-y}{1-\sigma}\right) \cdot \overbrace{\frac{\partial g}{\partial z}}^{\sigma(1-\sigma)} \cdot \overbrace{\frac{\partial z}{\partial w}}^x \quad \text{7.7}$$

Concept 291

The **chain rule** allows us to take the gradient of **nested functions**, where each function is the **input** to the next one.

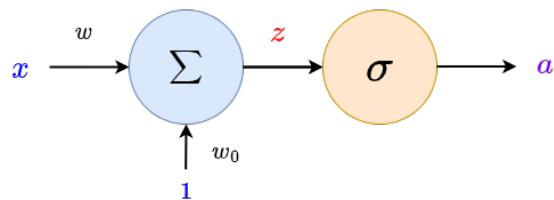
Another way to say this is that one function **feeds into** the next.

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 .

7.5.3 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**!



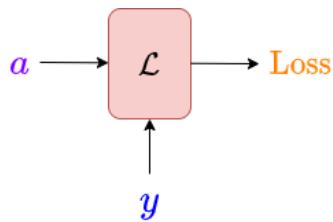
As usual, our first unit \sum is our **linear** component. The output is z , nothing different from before with LLC.

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 .

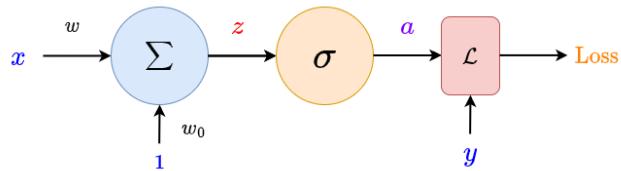
Something we neglected before: this diagram is **missing the loss function**. Let's create a small unit for that.

$\mathcal{L}(a, y)$ 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!

7.5.4 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 292

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:

- $f(10) = 10 + 2$
- $g(f) = 3 \cdot 12$
- $h(g) = \sin(36)$

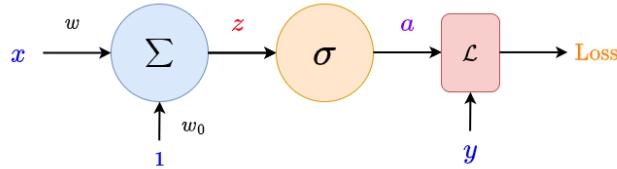
So, by "forward", we mean that we apply each function, one after another.

In our case, this means computing z , a , and $\mathcal{L}(a, y)$.



7.5.5 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.



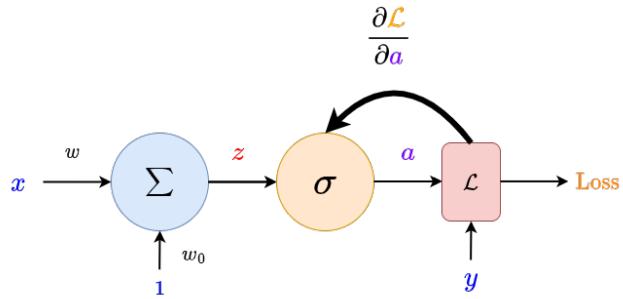
We can also call this "chaining together" lots of derivatives.

Loss \mathcal{L} is what we really care about. So, what is the loss directly **connected** to? The **activation**, a .

- Our loss function $\mathcal{L}(a, y)$ contains information about how \mathcal{L} is linked to a .

$$\underbrace{\frac{\partial \mathcal{L}}{\partial a}}_{\text{Loss unit}} \quad (7.8)$$

We send this information backwards, so it can be used later.

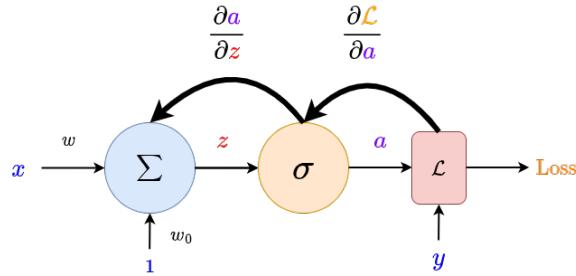


Now, we're on the $\sigma(z)$ unit.

- The $\sigma(z)$ unit contains information about how a is linked to z .
- We've connected \mathcal{L} to a , and a to z . We chain them together, connecting \mathcal{L} to z .

$$\overbrace{\frac{\partial \mathcal{L}}{\partial a}}^{\text{Loss unit}} \cdot \overbrace{\frac{\partial a}{\partial z}}^{\text{Activation function}} \quad (7.9)$$

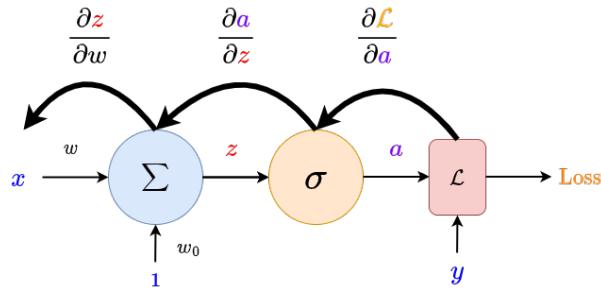
We haven't reached w yet, so we send this information further back.



Finally, we reach Σ .

- The Σ unit contains information about how z is linked to w .
- Finally, we have a chain of links, that allows us to connect \mathcal{L} to w .

This last derivative uses x , because $w^T x + w_0 = z$.



And, we built our chain rule! This contains the **information** of the derivatives from **every** unit.

$$\frac{\partial \mathcal{L}}{\partial w} = \underbrace{\frac{\partial \mathcal{L}}{\partial a}}_{\text{Loss unit}} \cdot \underbrace{\frac{\partial a}{\partial z}}_{\text{Activation}} \cdot \underbrace{\frac{\partial z}{\partial w}}_{\text{Linear subunit}} \quad (7.10)$$

Moving backwards like this is called **back-propagation**.

Definition 293

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.

7.5.6 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!

This summary covers some things we haven't fully discussed. We'll continue digging into the topic!

Concept 294

We can do **gradient descent** on a **neural network** using the ideas we've built up:

- 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.

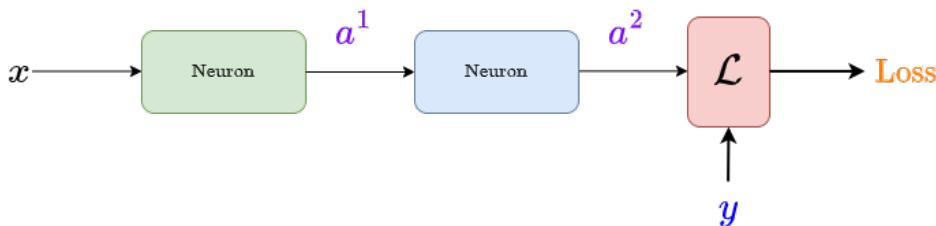
- Use your chain rule to get the **gradient** $\frac{\partial \mathcal{L}}{\partial w}$ for your **weight** vector(s). Take a **gradient descent** step.
- **Repeat** until satisfied, or your model **converges**.

7.5.7 A two-neuron network: starting backprop

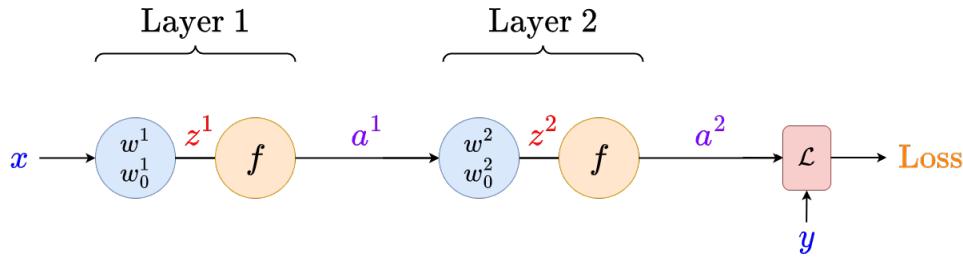
Above, we mention "**each layer**": we'll now transition to a **two-neuron** system, so we have "two layers". Then, we'll build up to many layers.

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:



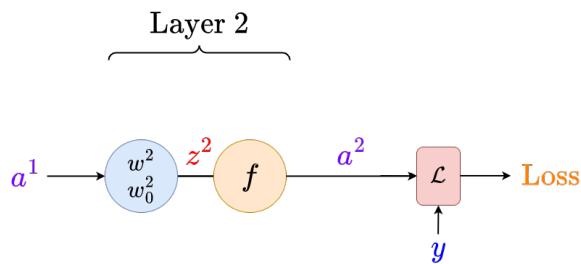
And unpack it:



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?

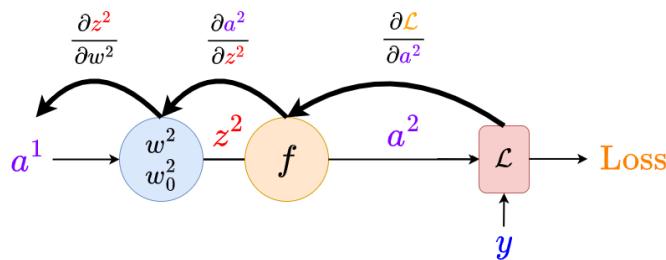
It turns out, it doesn't! We mentioned in the first part of chapter 7 that we can treat the **output** of the **first** layer a^1 as the same as if it were an **input** x .

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:

$$\frac{\partial \mathcal{L}}{\partial w^2} = \overbrace{\frac{\partial \mathcal{L}}{\partial a^2}}^{\text{Loss unit}} \cdot \overbrace{\frac{\partial a^2}{\partial z^2}}^{\text{Activation}} \cdot \overbrace{\frac{\partial z^2}{\partial w^2}}^{\text{Linear}} \quad (7.11)$$

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?

7.5.8 Continuing backprop: One more problem

We need to continue further to reach our **earlier** weights: this is why we have to work **backward**.

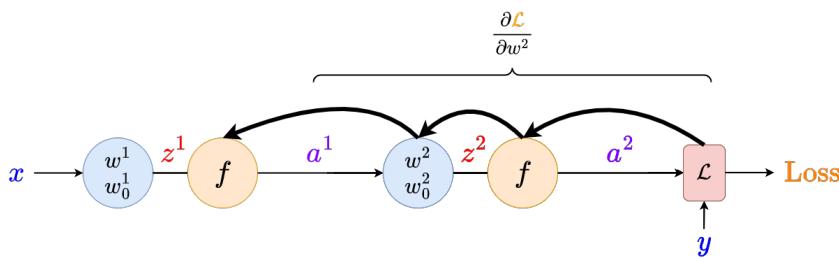
Concept 295

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 $\partial \mathcal{L} / \partial w^2$. If we wanted to continue the chain rule, we would expect to add more terms, like:

$$\frac{\partial w^2}{\partial a^1} \quad (7.12)$$

The problem is, what is w^2 ? It's a vector of constants.

$$w^2 = \begin{bmatrix} w_1^2 \\ w_2^2 \\ \vdots \\ w_n^2 \end{bmatrix}, \quad \text{Not a function of } a^1! \quad (7.13)$$

Since our current derivative includes w^2 , we would continue it with a w^2 in the "top" of a derivative,

$$\frac{\partial \mathcal{L}}{\partial w^2} \frac{\partial w^2}{\partial r}$$

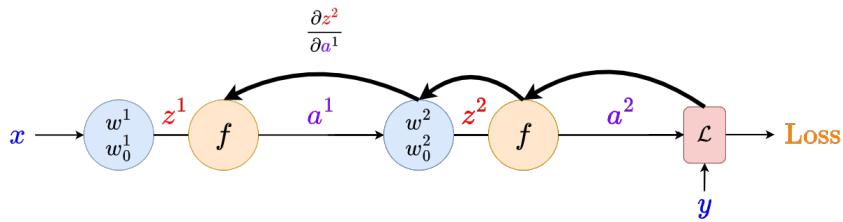
We're not sure what "r" is yet.

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:

$$\frac{\partial \mathcal{L}}{\partial a^1} = \underbrace{\frac{\partial \mathcal{L}}{\partial a^2}}_{\text{Other terms}} \cdot \underbrace{\frac{\partial a^2}{\partial z^2}}_{\text{Link Layers}} \cdot \underbrace{\frac{\partial z^2}{\partial a^1}}_{\text{Link Layers}} \quad (7.14)$$

Concept 296

For our **weight gradient** in layer l , we have to end our **chain rule** with

$$\frac{\partial z^l}{\partial w^l}$$

So we can get

$$\frac{\partial \mathcal{L}}{\partial w^l} = \underbrace{\frac{\partial \mathcal{L}}{\partial z^l}}_{\text{Other terms}} \cdot \underbrace{\frac{\partial z^l}{\partial w^l}}_{\text{Get weight grad}}$$

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

$$\frac{\partial z^l}{\partial a^{l-1}} \quad (7.15)$$

To "link together" two different layers l and $l - 1$ in a **chain rule**.

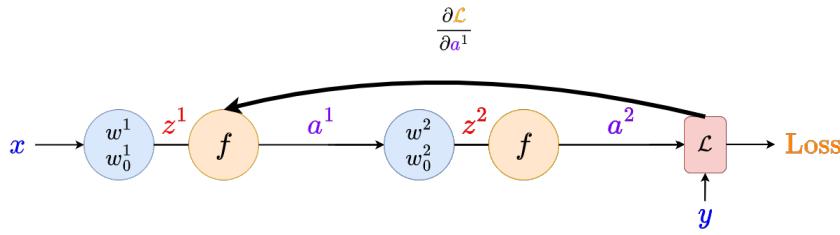
In this section, we compressed lots of derivatives into

$$\frac{\partial \mathcal{L}}{\partial z^l}$$

Don't let this alarm you, this just hides our long chain of derivatives!

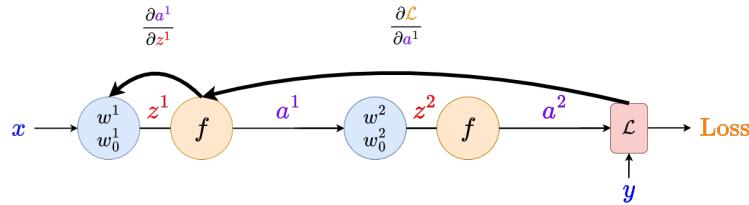
7.5.9 Finishing two-neuron backprop

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:

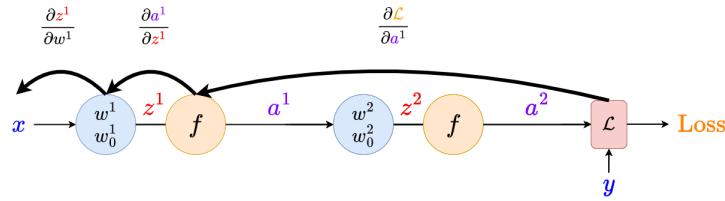


All the info we need is stored in this derivative: it can be written out using our friendly chain rule from earlier.

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

$$\frac{\partial \mathcal{L}}{\partial w^1} = \overbrace{\frac{\partial \mathcal{L}}{\partial a^1}}^{\text{Other layers}} \cdot \overbrace{\frac{\partial a^1}{\partial z^1} \cdot \frac{\partial z^1}{\partial w^1}}^{\text{Layer 1}} \quad (7.16)$$

Which, in reality, looks much bigger:

$$\frac{\partial \mathcal{L}}{\partial w^1} = \overbrace{\left(\frac{\partial \mathcal{L}}{\partial a^2} \right)}^{\text{Loss unit}} \cdot \overbrace{\left(\frac{\partial a^2}{\partial z^2} \cdot \frac{\partial z^2}{\partial a^1} \right)}^{\text{Layer 2}} \cdot \overbrace{\left(\frac{\partial a^1}{\partial z^1} \cdot \frac{\partial z^1}{\partial w^1} \right)}^{\text{Layer 1}} \quad (7.17)$$

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 297

We can get all of our **weight gradients** by repeatedly appending to the **chain rule**.

If we want to get the **weight gradient** of layer ℓ , we **terminate** with

$$\text{Within layer} \quad \frac{\partial \overbrace{\mathbf{a}^\ell}^{\text{a}}}{\partial \overbrace{\mathbf{z}^\ell}^{\text{z}}} \quad \cdot \quad \text{Get weight grad} \quad \frac{\partial \overbrace{\mathbf{z}^\ell}^{\text{z}}}{\partial \mathbf{w}^\ell}$$

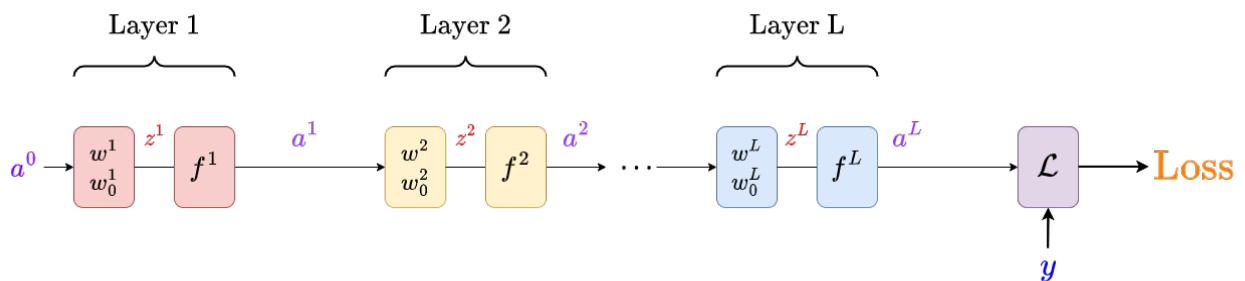
If we want to **extend** to the previous layer, we **instead** multiply by

$$\text{Within layer} \quad \frac{\partial \overbrace{\mathbf{a}^\ell}^{\text{a}}}{\partial \overbrace{\mathbf{z}^\ell}^{\text{z}}} \quad \cdot \quad \text{Link layers} \quad \frac{\partial \overbrace{\mathbf{z}^\ell}^{\text{z}}}{\partial \mathbf{a}^{\ell-1}}$$

7.5.10 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 298

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:

$$\frac{\partial \mathcal{L}}{\partial w^\ell} = \overbrace{\left(\frac{\partial \mathcal{L}}{\partial a^L} \right)}^{\text{Loss unit}} \cdot \overbrace{\left(\frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \right)}^{\text{Layer L}} \cdot \overbrace{\left(\frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^{L-1}}{\partial a^{L-2}} \right)}^{\text{Layer L-1}} \cdot \left(\dots \right) \cdot \overbrace{\left(\frac{\partial a^\ell}{\partial z^\ell} \cdot \frac{\partial z^\ell}{\partial w^\ell} \right)}^{\text{Layer } \ell} \quad (7.18)$$

That's pretty ugly. If we need to hide the complexity, we can:

Notation 299

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:

$$\frac{\partial \mathcal{L}}{\partial w^\ell} = \overbrace{\frac{\partial \mathcal{L}}{\partial z^\ell}}^{\text{Other}} \cdot \overbrace{\frac{\partial z^\ell}{\partial w^\ell}}^{\text{Weight term}}$$

But we should also explore what each of these terms *are*.

7.5.11 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**:

$$\frac{\partial \mathcal{L}}{\partial a^L} \tag{7.19}$$

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 :

$$\frac{\partial a^\ell}{\partial z^\ell} \tag{7.20}$$

What does the function between these **look** like?

$$a = f(z) \tag{7.21}$$

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.

$$\frac{\partial a^\ell}{\partial z^\ell} = \overbrace{(f^\ell)'}^{\text{deriv of func for layer } \ell} \overbrace{(z^\ell)}^{\text{Deriv input}} \tag{7.22}$$

This expression is a bit visually clunky, but it works. Without the annotation:

z^ℓ is not being multiplied by $(f^\ell)'$, it's the input to that derivative.

$$\frac{\partial a^\ell}{\partial z^\ell} = (f^\ell)'(z^\ell) \tag{7.23}$$

Between layers, we have

- We can also think about the derivative of the **linear function** that **connects two layers**:

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} \quad (7.24)$$

So, we want the function of these two:

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.

$$z^\ell = w^\ell a^{\ell-1} + w_0^\ell \quad (7.25)$$

This one is pretty simple! We just take the derivative manually:

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} = w^\ell \quad (7.26)$$

Finally, every gradient will end with...

- The derivative that directly connects to a **weight**, again using the **linear function**:

$$\frac{\partial z^\ell}{\partial w^\ell} \quad (7.27)$$

The linear function is the same:

$$z^\ell = w^\ell a^{\ell-1} + w_0^\ell \quad (7.28)$$

But with a different **variable**, the **derivative** comes out different:

$$\frac{\partial z^\ell}{\partial w^\ell} = a^{\ell-1} \quad (7.29)$$

Notation 300

Our **derivatives** for the **chain rule** in a **1-D neural network** take the form:

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \mathbf{a}^L} \\ \frac{\partial \mathbf{a}^\ell}{\partial \mathbf{z}^\ell} = (f^\ell)'(\mathbf{z}^\ell) \\ \frac{\partial \mathbf{z}^\ell}{\partial w^\ell} = \mathbf{a}^{\ell-1} \end{aligned} \tag{7.30}$$

Now, we can rewrite our generalized expression for gradient:

$$\frac{\partial \mathcal{L}}{\partial w^\ell} = \overbrace{\left(\frac{\partial \mathcal{L}}{\partial \mathbf{a}^L} \right)}^{\text{Loss unit}} \cdot \overbrace{\left((f^L)'(\mathbf{z}^L) \cdot w^L \right)}^{\text{Layer L}} \cdot \overbrace{\left((f^{L-1})'(\mathbf{z}^{L-1}) \cdot w^L \right)}^{\text{Layer L-1}} \cdot \left(\dots \right) \cdot \overbrace{\left((f^\ell)'(\mathbf{z}^\ell) \cdot \mathbf{a}^{\ell-1} \right)}^{\text{Layer } \ell} \tag{7.31}$$

Our expressions are more concrete now. It's still pretty visually messy, though.

7.5.12 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** $\text{step}(z)$:

$$\frac{d}{dz} \text{step}(z) = 0 \tag{7.32}$$

This is part of why we don't use this function: it has no gradient. We can show this by looking piecewise:

$$\text{step}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{7.33}$$

And take the derivative of each piece:

$$\frac{d}{dz} \text{ReLU}(z) = 0 = \begin{cases} 0 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \quad (7.34)$$

- **Rectified Linear Unit** $\text{ReLU}(z)$:

$$\frac{d}{dz} \text{ReLU}(z) = \text{step}(z) \quad (7.35)$$

This one might be a bit surprising at first, but it makes sense if you **also** break it up into cases:

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \quad (7.36)$$

And take the derivative of each piece:

$$\frac{d}{dz} \text{ReLU}(z) = \text{step}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases} \quad (7.37)$$

- **Sigmoid** function $\sigma(z)$:

$$\frac{d}{dz} \sigma(z) = \sigma(z)(1 - \sigma(z)) = \frac{e^{-z}}{(1 + e^{-z})^2} \quad (7.38)$$

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.

$$\sigma(z) = \frac{1}{1 + e^{-z}} \quad (7.39)$$

- **Identity** ("linear") function $f(z) = z$:

$$\frac{d}{dz} z = 1 \quad (7.40)$$

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 as the output activation 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 $\tanh(z)$:

$$\frac{d}{dz} \tanh(z) = 1 - \tanh(z)^2 \quad (7.41)$$

This strange little expression is 1 minus the "hyperbolic secant" squared. We won't bother further with it.

Notation 301

For our various **activation** functions, we have the **derivatives**:

Step:

$$\frac{d}{dz} \text{step}(z) = 0$$

ReLU:

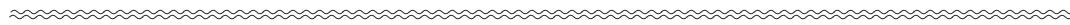
$$\frac{d}{dz} \text{ReLU}(z) = \text{step}(z)$$

Sigmoid:

$$\frac{d}{dz} \sigma(z) = \sigma(z)(1 - \sigma(z))$$

Identity/Linear:

$$\frac{d}{dz} z = 1$$



7.5.13 Loss derivatives

Now, we look at the loss derivatives.

- **Square loss** function $\mathcal{L}_{sq} = (a - y)^2$:

$$\frac{d}{da} \mathcal{L}_{sq} = 2(a - y) \quad (7.42)$$

Follows from chain rule+power rule, used for regression.

- **Linear loss** function $\mathcal{L}_{sq} = |a - y|$:

$$\frac{d}{da} \mathcal{L}_{lin} = \text{sign}(a - y) \quad (7.43)$$

This one can also be handled piecewise, like $\text{step}(z)$ and $\text{ReLU}(z)$:

$$|u| = \begin{cases} u & \text{if } z \geq 0 \\ -u & \text{if } z < 0 \end{cases} \quad (7.44)$$

We take the piecewise derivative:

$$\frac{d}{du}|u| = \text{sign}(u) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases} \quad (7.45)$$

- **NLL** (Negative-Log Likelihood) function $\mathcal{L}_{\text{NLL}} = -(y \log(a) + (1-y) \log(1-a))$

$$\frac{d}{da}\mathcal{L}_{\text{NLL}} = -\left(\frac{y}{a} - \frac{1-y}{1-a}\right) \quad (7.46)$$

- **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 302

For our various **loss** functions, we have the **derivatives**:

Square:

$$\frac{d}{da}\mathcal{L}_{\text{sq}} = 2(a - y)$$

Linear (Absolute):

$$\frac{d}{da}\mathcal{L}_{\text{lin}} = \text{sign}(a - y)$$

NLL (Negative-Log Likelihood):

$$\frac{d}{da}\mathcal{L}_{\text{NLL}} = -\left(\frac{y}{a} - \frac{1-y}{1-a}\right)$$

7.5.14 Many neurons per layer

Now, we just have left the elephant in the room: what do we do about the case where we have *big* layers? That is, what if we have **multiple** neurons per layer? This makes this more complex.

Well, the solution is the same as earlier in the course: we introduce **matrices**.

But this time, with a twist: we have to do serious **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.

7.5.15 The chain rule: Matrix form

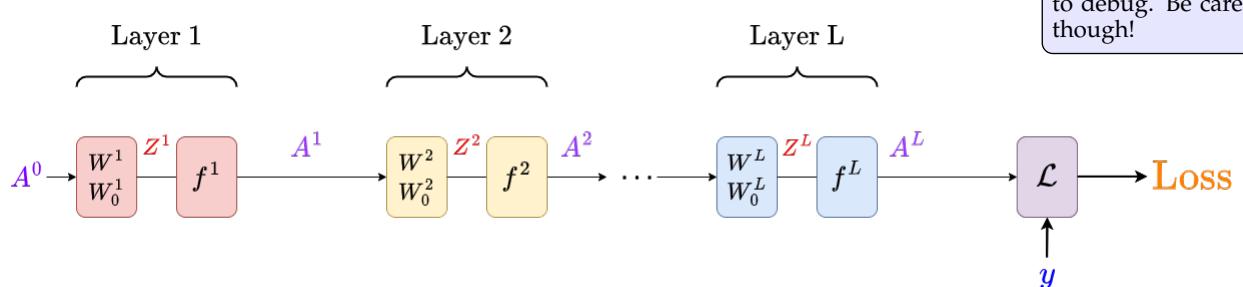
Let's start with the first: the punchline, how does the chain rule and our gradient descent **change** when we add **matrices**?

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!

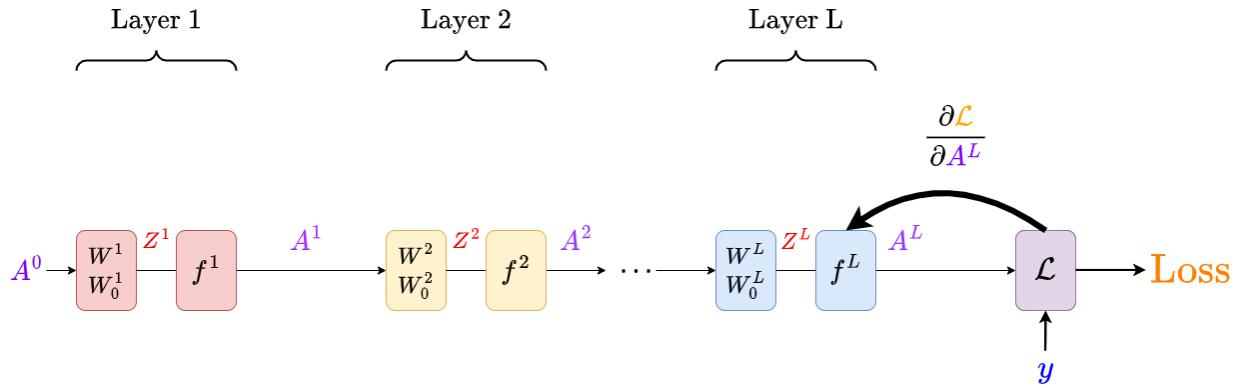
Pretty much the same! Only major difference: swapped scalars for vectors, and vectors for matrices (represented by switching to uppercase)

And, we do backprop the same way, too.

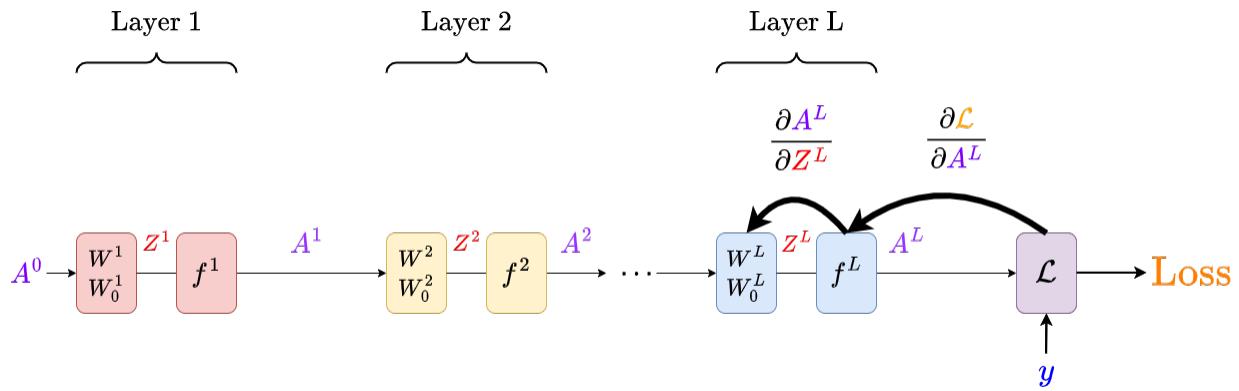
Here, we're not going to explain much as we go: all we're doing is getting the **derivatives** we need for our **chain rule**!

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:

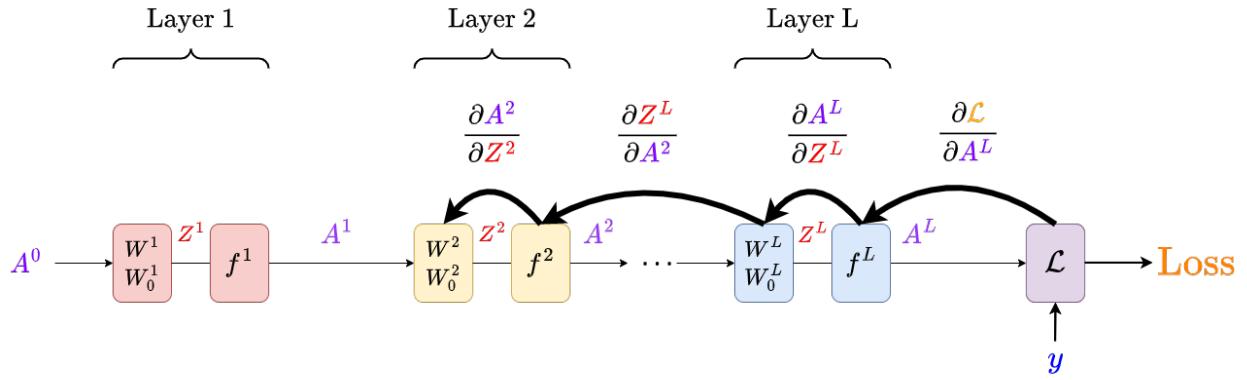


Take another step:

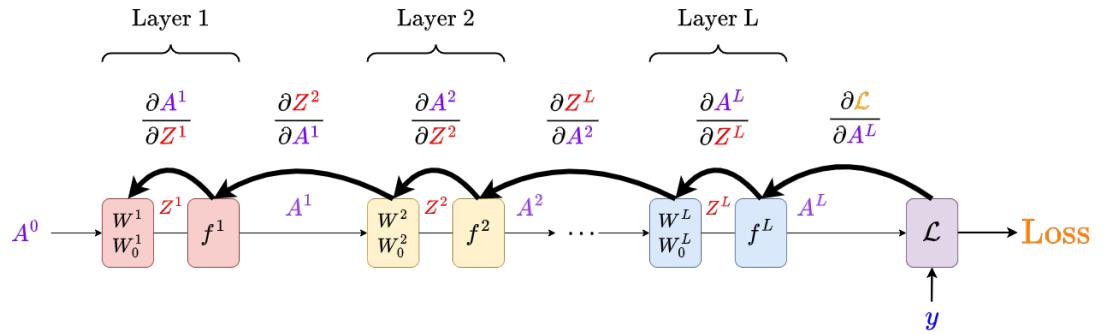


We'll pick up the pace: we'll jump to layer 2 and get its gradient.

The term $\frac{\partial Z^L}{\partial A^2}$ contains lots of derivatives from every layer between L and 2. But, all we're omitting is the same kinds of steps we're doing in layers 1, 2, and L.



Now, we finally get to layer 1!



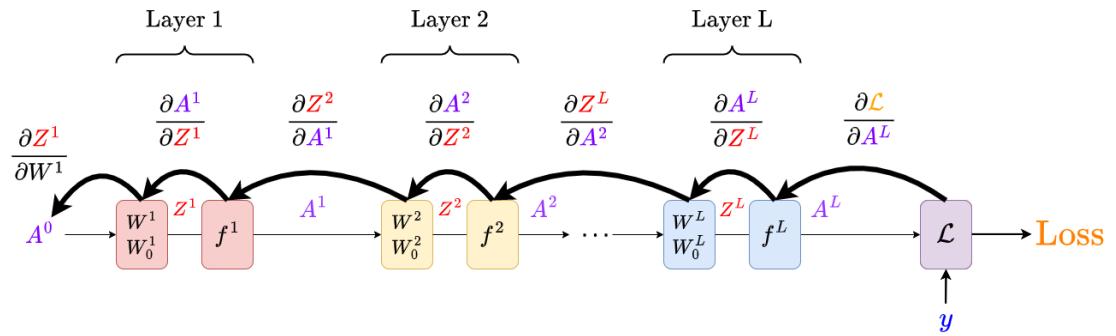
We finish off by getting what we're after: the gradient for W^1 .

Notation 303

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.

7.5.16 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

$$\frac{\partial Z^\ell}{\partial W^\ell} \quad (7.47)$$

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:

$$\frac{\partial \mathcal{L}}{\partial W^\ell} = \underbrace{\frac{\partial \mathcal{L}}{\partial Z^\ell}}_{\text{Weight link}} \cdot \overbrace{\left(\frac{\partial \mathcal{L}}{\partial Z^\ell} \right)^\top}^{\text{Other layers}}$$

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.

There are also deeper interpretations, but they aren't worth digging into for now.

Notation 304

The **gradient** $\nabla_{W^\ell} \mathcal{L}$ for a neural network is given as:

$$\frac{\partial \mathcal{L}}{\partial W^\ell} = \underbrace{\frac{\partial Z^\ell}{\partial W^\ell}}_{\text{Weight link}} \cdot \underbrace{\left(\frac{\partial \mathcal{L}}{\partial Z^\ell} \right)}_{\text{Other layers}}^\top$$

We get our remaining terms $\partial \mathcal{L} / \partial Z^\ell$ by our usual chain rule:

$$\frac{\partial \mathcal{L}}{\partial Z^\ell} = \underbrace{\left(\frac{\partial A^\ell}{\partial Z^\ell} \right)}_{\text{Layer } \ell} \cdot \left(\dots \right) \cdot \underbrace{\left(\frac{\partial Z^{L-1}}{\partial A^{L-2}} \cdot \frac{\partial A^{L-1}}{\partial Z^{L-1}} \right)}_{\text{Layer } L-1} \cdot \underbrace{\left(\frac{\partial Z^L}{\partial A^{L-1}} \cdot \frac{\partial A^L}{\partial Z^L} \right)}_{\text{Layer } L} \cdot \underbrace{\left(\frac{\partial \mathcal{L}}{\partial A^L} \right)}_{\text{Loss unit}}$$

This is likely our most important equation in this chapter!

7.5.17 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 305

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
- Why? Check the matrix derivative notes.

Example: Suppose you have the activation $f(\mathbf{z}) = \mathbf{z}^2$.

Your pre-activation might be

$$\mathbf{z}^\ell = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \quad (7.48)$$

The output would be

$$\mathbf{a}^{\ell} = f(\mathbf{z}^{\ell}) = \begin{bmatrix} 1 \\ 2^2 \\ 3^2 \end{bmatrix} \quad (7.49)$$

But the derivative would be:

$$f(z) = 2z \quad (7.50)$$

Which, gives our matrix derivative as:

$$\frac{\partial \mathbf{a}^{\ell}}{\partial \mathbf{z}^{\ell}} = \begin{bmatrix} f'(1) & 0 & 0 \\ 0 & f'(2) & 0 \\ 0 & 0 & f'(3) \end{bmatrix} = \begin{bmatrix} 2 \cdot 1 & 0 & 0 \\ 0 & 2 \cdot 2 & 0 \\ 0 & 0 & 2 \cdot 3 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 6 \end{bmatrix}$$

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!

$$\frac{\partial \mathbf{w}}{\partial \mathbf{v}} = \left\{ \begin{array}{cccc} \frac{\partial w_1}{\partial v_1} & \frac{\partial w_1}{\partial v_2} & \dots & \frac{\partial w_1}{\partial v_m} \\ \frac{\partial w_2}{\partial v_1} & \frac{\partial w_2}{\partial v_2} & \dots & \frac{\partial w_2}{\partial v_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial w_n}{\partial v_1} & \frac{\partial w_n}{\partial v_2} & \dots & \frac{\partial w_n}{\partial v_m} \end{array} \right\} \quad \begin{array}{l} \text{Column } j \text{ matches } w_j \\ \text{Row } i \text{ matches } v_i \end{array} \quad (7.51)$$

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.

Clarification 306

Note that we have chosen a **convention** for how our matrices work: plenty of other resources use a transposed version of matrix derivatives.

This alternate version means the exact **same** thing as our version. Our choice is called the **denominator layout notation** for matrix derivatives.

7.6 Training

7.6.1 Comments

A few important side notes on training. First, on derivatives:

Concept 307

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 308

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.

7.6.2 Pseudocode

Our training algorithm for backprop can follow smoothly from what we've laid out.

Here, we'll use the @ symbol to indicate matrix multiplication, following numpy conventions.

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( $\mathcal{D}_n, T, L, (m^1, \dots, m^L), (f^1, \dots, f^L), Loss$ )
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 Descend (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: 11/08/23 21:00:09

```

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

CHAPTER 7

Neural Networks 2 - Training Techniques, Regularization

7.7 Optimizing neural network parameters

We now understand both how neural networks work, and how to **train** them. We can use gradient descent to **optimize** their parameters.

But, we can do **better** than a simple SGD approach with step size $\eta(t)$. We'll try out some **modifications** that can speed up our training, and make better models.

7.7.1 Mini-batch

7.7.1.1 Review: Gradient Descent Notation

Let's review some gradient descent notation. We want to **optimize** our objective function J using W .

We do this using the gradient. This gradient depends on our current weights at time t , W_t .

$$\overbrace{\nabla_W J}^{\text{General Gradient}} \longrightarrow \overbrace{\nabla_W J(W_t)}^{\text{Gradient at time } t} \quad (7.1)$$

Our update rule is:

$$W_{\text{new}} = W_{\text{old}} - \eta \overbrace{(\nabla_W J(W_{\text{old}}))}^{\text{Gradient}} \quad (7.2)$$

Or, using timestep t :

$$W_{t+1} = W_t - \eta \overbrace{(\nabla_W J(W_t))}^{\text{Gradient}} \quad (7.3)$$

What is our objective function J ? Without regularization, it's based on our **loss** function.

We can get loss for each of our data points: _____

$$\mathcal{L}^{(i)} = \overbrace{\mathcal{L}(g^{(i)}, y^{(i)})}^{\text{Loss for data point } i} \quad (7.4)$$

We won't define J here, because it is slightly different for SGD and BGD. We'll get to that below.

Our guess $g^{(i)}$ depends on both our current data point $x^{(i)}$, and the current weights W_t :

$$\mathcal{L}^{(i)}(W_t) = \mathcal{L}(\underbrace{h(x^{(i)}; W_t)}_{g^{(i)}}, y^{(i)}) \quad (7.5)$$



7.7.1.2 Review: BGD vs. SGD

Let's review our two main types of gradient descent, using the equation

$$W_{t+1} = W_t - \eta \overbrace{(\nabla_W J(W_t))}^{\text{Gradient}} \quad (7.6)$$

First, we have **batch gradient descent**, where we use our **whole** training set each time we take a step.

Definition 309

Batch Gradient Descent (BGD) is a form of gradient descent where we get the **gradient** of our loss function using **all of our training data**.

$$\nabla_{\mathbf{W}} J(\mathbf{W}_t) = \sum_{i=1}^n \overbrace{\nabla_{\mathbf{W}} (\mathcal{L}^{(i)}(\mathbf{W}_t))}^{\text{Each data point}}$$

We get the gradient for each data point, and then **add** all of those gradients up. We use this **combined gradient** to take **one step**.

We **repeat** this process every time we want to take a new step.

Then, we have **stochastic gradient descent**, where we use only **one** data point for each step we take.

Definition 310

Stochastic Gradient Descent (SGD) is a form of gradient descent where we get the **gradient** of our loss function using **one data point at a time**.

$$\nabla_{\mathbf{W}} J(\mathbf{W}_t) = \overbrace{\nabla_{\mathbf{W}} (\mathcal{L}^{(i)}(\mathbf{W}_t))}^{\text{One data point}}$$

We **randomly** choose one data point $(x^{(i)}, y^{(i)})$ and get the **gradient**. Based on this one gradient, we take our **step**.

For each step, we choose a new **random** data point.

These two approaches have tradeoffs:

Concept 311

There are **tradeoffs** between **SGD** and **BGD**:

- Each step is **faster** in **SGD**: we only use one data point.
 - Meanwhile, **BGD** is **slower**: each step uses all of our data.
 - **SGD** could improve a lot with only a **small subset** of a data.
- Because **BGD** uses all our data, its gradient is much more **accurate**.
 - **SGD** often uses **smaller** steps: the gradient is less accurate, with less data.
 - This is worse if the data is **noisy**: each SGD step becomes less effective.
- **SGD randomly** chooses data points: this random noise makes it harder to over-fit.
 - **BGD** uses all of the data, so we don't reduce overfitting.

7.7.1.3 Mini-batch

Rather than picking one or the other, one might think, "why do we have to pick **every** data point or **one** data point? Couldn't we pick only a **few**?"

This is the premise of **mini-batch**: instead of making a batch out of the entire training set, we **randomly** select a few data points, and use that as our batch.

Definition 312

Mini-batch is a way to **compromise** between SGD and BGD.

To create a mini-batch, we **randomly** select K data points from our training data.

We treat this mini-batch the same way we would a regular **batch**: get the **gradient** of each data point, **add** those gradients, and take one step of gradient descent.

$$\nabla_{\mathbf{W}} J(\mathbf{W}_t) = \underbrace{\sum_{i=1}^K}_{\text{K data points in a mini-batch}} \nabla_{\mathbf{W}} (\mathcal{L}^{(i)}(\mathbf{W}_t))$$

We gather a **new** mini-batch for each step we want to take.

Mini-batch is the **default** used in most modern packages: it gives us more **control** over our

algorithm, and can often find the **best** of both worlds.

Concept 313

Mini-batch has a lot of benefits of both SGD and BGD:

- Steps are **faster** than BGD: we only need to get the gradient for K points.
 - The **speed** no longer depends on the total training data size (more data, more gradients): instead, it depends on our **batch size** K.
- Steps are more **accurate** than SGD: with more data, we have a better **gradient**.
 - This means we can take **bigger** steps.
- Our batches are **random**, like SGD: we reduce overfitting and escape local minima.

We do have to be careful to randomly select data in an efficient way, though. Packages usually take care of this.

One more important benefit:

- If we find that a particular problem is better suited for something closer to BGD or SGD, we can **adjust** our batch size K.
 - This gives us more **control** over our learning algorithm.

7.7.2 Adaptive Step Size - Challenges

We'll stop discussing mini-batches, and the SGD vs. BGD problem. Instead, let's improve our **step size**.

Step size η is a difficult problem:

- If η is **small**, then our training can take a long **time**.
- If η is too **large**, we might **diverge**: our answer gets way too large.
- A **large** step size might also cause **oscillation**: most of our step is wasted going back and forth, so we go **slowly** again.

SGD and mini-batch have a step size-related problem, too:

- In order to **converge** according to our theorems (see chapter 3), the step size $\eta(t)$ has to be **decreasing** in a certain way.

Check chapter 3 for the exact requirements of the theorem.

We'll spend the following sections coming up with solutions.

7.7.3 Vanishing/Exploding Gradient

Now, neural networks have one more **problem**, that we've ignored so far: **deep** neural networks can cause a problem called "**exploding/vanishing gradient**".

Here's an example: suppose you have a long chain rule, with 8 terms. Our chain rule gets **longer** with more layers, because each layer needs its own derivatives.

By "deep", we just mean "many layers".

$$\frac{\partial A}{\partial H} = \frac{\partial A}{\partial B} \cdot \frac{\partial B}{\partial C} \cdot \left(\dots \right) \cdot \frac{\partial G}{\partial H} \quad (7.7)$$

This chain rule gets **longer** as we move "**backwards**" through our network, so the chain rule is longest for the "**early**" layers: $\ell = 1, 2$, and so on.

Suppose all of our derivatives are roughly $.1$. What happens when we multiply them **together**?

$$\frac{\partial A}{\partial H} = .1 \cdot .1 \cdot \left(\dots \right) \cdot .1 = 10^{-8} \quad (7.8)$$

The derivative becomes really, really **tiny**! This is the case of the **vanishing** gradient: if our gradients are less than one, then as we append more layers, they multiply to get smaller and smaller.

- This is a problem: if our gradients in our earlier layers become too **small**, we'll never make any progress! They'll hardly change.

Definition 314

Vanishing gradient occurs when a deep neural network ends up with **very small gradients** in the **earlier** layers.

This happens because a deeper neural network has a **longer chain rule**: if all of the terms are **less than one**, they'll multiply into a very small value, "**vanishing**".

This means that our gradient descent will have **almost no effect** on these earlier weights, **slowing down** our algorithm considerably.

What if the gradients are larger than 1 ? Let's say our derivatives are 10 each.

$$\frac{\partial A}{\partial H} = 10 \cdot 10 \cdot \left(\dots \right) \cdot 10 = 10^8 \quad (7.9)$$

Now, the early derivatives are becoming **huge**! This is the case of **exploding** gradient: if our gradients are greater than one, then as we add layers, they multiply to get bigger.

- This is also a problem: we don't want to take **huge** steps, or we will **diverge**, or **oscillate**, and jump huge distances across the **hypothesis space**.

Definition 315

Exploding gradient occur when a deep neural network ends up with **very large gradients** in the **earlier** layers.

This happens because a deeper neural network has a **longer chain rule**: if all of the terms are much **greater than one**, they'll multiply into a very large value, "**exploding**".

This means that our gradient descent will take **huge steps** in the hypothesis space. This can cause us to **diverge**, miss local minima, or **oscillate**.

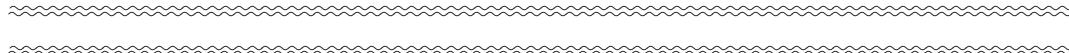
So, to avoid this, we can't just blindly multiply our gradients and keep a fixed step size.

The solution? Each **weight** gets its own step size η .

Concept 316

In order to avoid **vanishing/exploding** gradient problems, we give each **weight** in our network its own **step size** η .

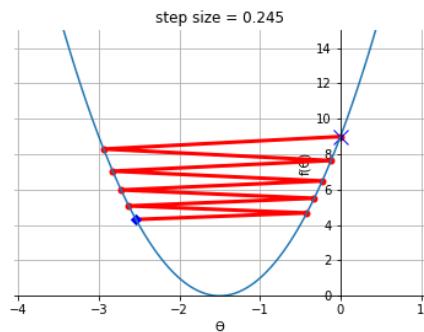
This allows us to **adjust** the step size for some weights more than others: if our gradient is too large or small, we can fix it.



7.7.4 Momentum

7.7.4.1 Solving Oscillation

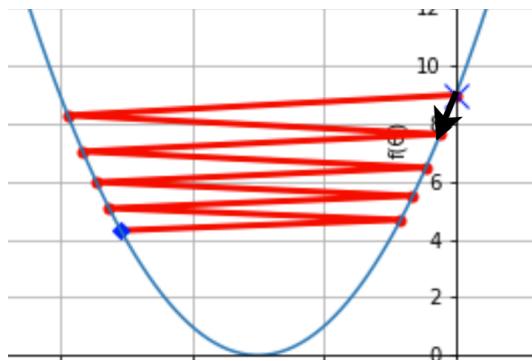
Let's look at one common problem we have with gradient descent: **oscillation**.



We overshoot our target, and then have to take another step that **undoes** most of what happened in the previous step. So, we waste a lot of time correcting the last step.

This can significantly **slow** down how quickly we converge.

For example, our first two steps land us in almost the same place we started!



The black arrow shows the combined effect of our first two steps: almost nothing!

We don't want to waste time, so we want to remove the "part" of the gradient that is likely to **cancel** out.

The **next** gradient cancels out some of the previous. Our first two steps add up, or "**average out**" to a small improvement.

If our steps are effectively "averaging", we'll speed up that process: we'll average together the gradients *before* taking our step!

This means we can take a bigger step in a direction we won't have to cancel!

Concept 317

Since our gradient descent steps **combine** to give us our new model, we can think of them as adding, or "**averaging**" to a more accurate improvement.

When our function **oscillates**, we get the same pattern **multiple** times: past steps indicate the sort of pattern we'll see in **future** steps.

The only real difference between adding and averaging is whether we divide by the number of terms.

So, we'll average our current gradient with past gradients: that way, the **component** that gets cancelled out is **removed**, and we won't have to undo our mistake over and over.

Concept 318

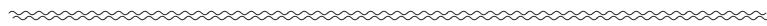
Oscillation causes us to move back and forth over the same region **multiple** times, where each step mostly **cancels** out the last.

One solution is to **average out** multiple of our gradients: the part that is "**cancelled out**" should be eliminated by the average.

So, we average our **past** gradients (past oscillation) with our **current** gradient, so we move in a more **efficient** direction, speeding up our algorithm.

Another way to think about it: when we **average** out our current and previous gradient, we're cancelling out what they "**disagree**" on, and keeping what they **agree** on.

So, we're taking a step in the direction that multiple gradients agree will **improve** our model!



7.7.4.2 Weighted averages

We could naively average all of our gradients **equally**. But, this would be a bad idea:

- It doesn't give you as much control of the algorithm: what if we care more about the **present** gradient, than the previous one?
- Gradients further in the **past** are **less likely** to matter: we've moved further away from those positions.
 - We also need to **scale** down past terms, so they don't take up most of the average.

The first problem is easy to solve: we'll **weigh** each of our terms differently.

If you're averaging 100 terms, and you add one more... it's not going to change much.

Concept 319

A **weighted average** is used when we want some terms to affect our **average** more than others.

We represent this with **weights**: each weight represents the **proportion** of our average from that term.

$$\text{Weighted Average} = x_1 w_1 + x_2 w_2 + \dots + x_n w_n$$

Example: If $w_1 = .6$, that means 60% of the average comes from x_1 .

Note that, since we're talking about **proportions**, they need to **sum to 1**: it wouldn't make sense to have more than 100% of the average.

At each time step, we're adding one new gradient: the **present** one.

We'll simplify our average to those two terms: the **present** gradient, versus all the **past** gradients.

These weights are **separate** from the weights inside our neural network.

They do, however, represent the same type of concept: the NN weights scale the **input**, while these weights scale the **gradients**.

- We represent the importance (**weight**) of our **past** gradients using the variable γ .
- We want the two terms to add to 1: so, the importance of **current** gradient is $1 - \gamma$.

$$\widehat{A}_t = \underbrace{\gamma G_{t-1}}_{\text{Old gradients}} + \underbrace{(1-\gamma)g_t}_{\text{New gradient}} \quad (7.10)$$

Now, we have **control** over how much the present or past gradient matters: we just have to adjust γ .



7.7.4.3 Running Average

We still have some work to do: first, we haven't made it clear how we're incorporating our old gradients: we lumped them into one term.

Let's try building up from $t = 1$. We'll assume our previous gradients are 0, for simplicity.

$$A_0 = g_0 = 0 \quad (7.11)$$

Our first step will average this with our **first** gradient:

$$A_1 = \gamma g_0 + (1-\gamma)g_1 \quad (7.12)$$

Simplifying to:

$$A_1 = (1 - \gamma)g_1 \quad (7.13)$$

What about our second step?

$$A_2 = \overbrace{\gamma G_{t-1}}^{\text{Old gradients}} + (1 - \gamma)g_2 \quad (7.14)$$

We could just plug in g_1 . But, A_1 contains the information about our first gradient g_1 , and the gradient before it, g_0 .

$$A_2 = \overbrace{\gamma A_1}^{\text{Contains } g_1, g_0} + (1 - \gamma)g_2 \quad (7.15)$$

We can repeat this process:

$$A_3 = \overbrace{\gamma A_2}^{\text{Contains } g_2, g_1, g_0} + \overbrace{(1 - \gamma)g_3}^{\text{New gradient}} \quad (7.16)$$

And so, we've created a general way to **average** as our program **runs** through different gradients.

$$A_t = \gamma A_{t-1} + (1 - \gamma)g_t \quad (7.17)$$

To allow more flexibility, we'll allow γ to **vary in time**, as γ_t .

$$A_t = \gamma_t A_{t-1} + (1 - \gamma_t)a_t \quad (7.18)$$

- We call this a **running average**.

Definition 320

A **running average** is a way to average past data with present data **smoothly**.

Our **initial** value for the average is typically zero:

$$A_0 = 0$$

Then, we begin introducing **new** data points.

- You use the parameter γ_t to indicate how much you want to prioritize **past data**.
- Thus, $1 - \gamma_t$ indicates the value of **new data**.

$$\overbrace{A_t}^{\text{Average}} = \underbrace{\gamma_t A_{t-1}}_{\text{Old gradients}} + \underbrace{(1 - \gamma_t) g_t}_{\text{New gradient}}$$

- Note that instead of γ , we write γ_t : this "discount factor" can vary with time.

Clarification 321

This is technically only one kind of **running average**: here, we use an "**exponential moving average**".

There are different ways to average past data points, with different **weighting** schemes.

- For example, you could do a "**simple moving average**", where you average equally over the last n data points.

7.7.4.4 Running Averages: The Distant Past

So, how does this "running average" approach affect our different data points, further in the past? Let's find out.

For simplicity, let's assume $\gamma_t = \gamma$: it's a **constant**.

$$A_t = \gamma A_{t-1} + (1 - \gamma) g_t \quad (7.19)$$

We can expand A_{t-1} to see further in the past:

$$A_t = \gamma(\gamma A_{t-2} + (1 - \gamma) g_{t-1}) + (1 - \gamma) g_t \quad (7.20)$$

And even further: _____

This is starting to get messy: don't worry if it's hard to read.

$$A_t = \gamma \left(\gamma (A_{t-3} + (1-\gamma)g_{t-2}) + (1-\gamma)g_{t-1} \right) + (1-\gamma)g_t \quad (7.21)$$

Let's rewrite this.

$$\gamma^3 A_{t-3} + \gamma^2 (1-\gamma)g_{t-2} + \gamma (1-\gamma)g_{t-1} + (1-\gamma)g_t \quad (7.22)$$

~~~~~

We see a "stacking" effect for  $\gamma$ :

- We only partly include our newest data point: we scale it by  $1 - \gamma$ , to make room for the past.

$$(1-\gamma)g_t \quad (7.23)$$

- But if your gradient is 1 time unit in the past, we apply  $\gamma$  **once**, "forgetting" some more of that gradient.

$$\gamma (1-\gamma)g_{t-1} \quad (7.24)$$

- But if your gradient is 2 units in the past, we apply  $\gamma$  **twice**: we've "forgotten" some of it twice.

$$\gamma^2 (1-\gamma)g_{t-2} \quad (7.25)$$

Each time we do an average, we scale down our older data points by  $\gamma$ . So, the further in the past, the less effect they have.

- This is exactly the kind of design we wanted!

### Concept 322

A **running average** tends to pay less attention to data further in the **past**.

- In general, if you are  $k$  time units in the past, we apply a factor of  $\gamma^k$ .

Because  $\gamma < 1$ , this **exponentially** decays to 0.

~~~~~

If we want to fully expand A_t , it's easiest to use a sum:

$$A_T = \sum_{t=0}^T \gamma^{(T-t)} \cdot (1-\gamma)^t \cdot g_t$$

You can compare this formulation against what we computed above.



7.7.4.5 Momentum

Applying a running average to our gradients gives us **momentum**.

- This analogy to **physics** represents how our point "moves" through the **weight space**, to optimize J .
- The gradient gives us a "direction" of motion. So, our **momentum** represents the direction we were "already moving": the **previous** averaged gradient.

We use M to represent the "averaged gradient" that we use to move. Our initial momentum is 0:

$$M_0 = 0 \quad (7.26)$$

Gradient descent adjusts our weights: we go from one list of weights, to another. That's why we say we're moving through the "weight space".

Because our hypothesis is determined by our weights, this is also a "hypothesis space".

And we want to average our current gradient with past gradients:

$$M_t = \gamma M_{t-1} + (1 - \gamma) g_t \quad (7.27)$$

What is our **past** gradient g_t ? Well, we want to use W to modify J : $\frac{\partial J}{\partial W}$, or $\nabla_W J$.

And we're moving through the **weight space**, so our input to the gradient is the previous set of weights, W_{t-1} .

$$g_t = \nabla_W J(W_{t-1}) \quad (7.28)$$

And finally, M_t , our "averaged gradient", determines how we move.

$$W_t = W_{t-1} - \eta M_t \quad (7.29)$$

Definition 323

Momentum is a technique for gradient descent where we do a **running average** between our current gradient, and our older gradients.

This approach reduces **oscillation**, and thus aims to improve the speed of convergence for our models.

- Our initial "momentum" (averaged gradient) is **0**.
- The amount we value new data is given by γ_t .
- Old data is scaled by $1 - \gamma_t$.

$$M_0 = 0$$

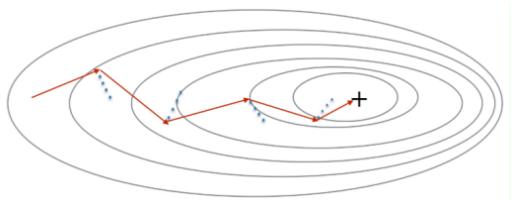
$$M_t = \underbrace{\gamma_t M_{t-1}}_{\text{Old gradients}} + \underbrace{(1 - \gamma_t) \nabla_W J(W_{t-1})}_{\text{New gradient}}$$

We use this momentum to take our step:

$$W_t = W_{t-1} - \eta M_t$$

~~~~~  
This approach puts more emphasis on newer data points, and less on older ones.

**Example:** We can see this "dampened oscillation" in action below:



The blue lines indicate the more "severe" path that we would've taken with normal gradient descent. the orange line is the line we take with momentum.

Notice that generally, the orange path is closer to correct! We still oscillate somewhat, but much less than we would have otherwise.

## 7.7.5 Adadelta

### 7.7.5.1 Per-weight step sizes

Earlier, we discussed creating **separate** step sizes for different weights in our network:

- Different weights could have different **sensitivities**: one weight could have a much larger/smaller impact on  $J$ , based on **structure**.
- We already have the challenge of figuring out what **step sizes** cause **slow convergence/divergence**: it would be even harder to find a step size that fit **all** of our weights, at the same time.

So, instead, we decided that each weight gets its **own step size**.

- But, we never elaborated on **how** we compute that (adjustable) step size.

#### Notation 324

Our base step size is indicated as  $\eta$ .

We'll call the **modified** step size for weight  $j$ ,  $\eta_j$ .

### 7.7.5.2 Scaling

Our goal now, is to come up with a **systematic** way to **assign step sizes**. This allows us to **adjust**, rather than run our whole gradient descent with a "bad" step size.

Why do we adjust our step size? To avoid slow convergence, oscillation, or divergence.

So, we have less of a problem if we choose  $\eta$  poorly.

- We might expect **slow convergence** if the derivative is too **small**: we carefully take small steps, but we aren't having much of an impact on  $J$ .
  - Across this "flatter" region of the surface, in one direction, we might expect it to be safer to move further.
- We might expect **divergence** or **oscillation** if the derivative is too **large**.
  - We might end up "missing" possible solutions/local minima by **overshooting** them.
  - So, "steeper" regions might be riskier.

**Concept 325**

Our goal is to **scale** our step size, so that it **adapts** to the situation:

- We want to **shrink** our step for **large** gradients
- We want to **increase** our step for **small** gradients

And we're interested in the **magnitude** of these gradients.

This sort of behavior is easily captured by including a factor of  $1/\|g_t\|$ . However, this has a smoothness problem: so, we'll use  $1/\|g_t\|^2$  instead.

Though, we need to keep it separate for each of our data points.

**Notation 326**

The gradient for **weight j** at **time t** is given as

$$g_{t,j} = \nabla_{W_j} J(W_{t-1})_j$$

Note the double-subscript.

- By isolating weight  $j$ , we have a constant, not a vector.

We can now write our gradient update rule:

$$W_{t,j} = W_{t-1,j} - \eta_j \cdot g_{t,j} \quad (7.30)$$

And we're currently using the step size

Don't save this equation! It isn't our final formula.

$$\eta_j = \frac{\eta}{g_{t,j}^2} \quad (7.31)$$

**7.7.5.3 Averaging**

But, we don't necessarily know how "steep" our region **generally** is, based on the current gradient  $g_t$ .  $g_t$  only gives us **one point** in space.

It would be helpful to include information from the **past**: we'll be re-using the **weighted average**, once again.

$$G_t = \gamma G_{t-1} + (1 - \gamma) g_t^2 \quad (\text{Maybe?})$$

- Note that we're averaging the **squared** magnitude.

Once again, it's helpful that the weighted average gradually "forgets" older information: we care less about gradients which are "further" from the present.

Technically this is still **incorrect**: we need the  $j$  notation.

$$G_{t,j} = \gamma G_{t-1,j} + (1 - \gamma) g_{t,j}^2 \quad (\text{Fixed!})$$

It's incorrect because  $g_t$  is a vector: we can't square it directly, we have to square its magnitude.

#### 7.7.5.4 Division by zero

There's a problem with our weight adjustment:

$$\eta_j = \frac{\eta}{G_{t,j}} \quad (7.32)$$

What happens if the denominator is near zero? It'll explode to a huge number! And at zero, it's undefined.

To solve this, we'll add a very small constant,  $\epsilon$ .

We won't prescribe any particular choice of  $\epsilon$  here.

$$\eta_j = \frac{\eta}{G_{t,j} + \epsilon} \quad (7.33)$$

Now, our scaling factor will never be bigger than  $1/\epsilon$ .

#### 7.7.5.5 Square root

One last concern, to wrap up: currently, we're diving by the **squared** gradient. This is actually somewhat overkill.

Remember that our goal is to do the following operation:

$$W_{t,j} = W_{t-1,j} - \underbrace{\eta_j \cdot g_{t,j}}_{\text{Update}} \quad (7.34)$$

With our current formula, our update has

- a factor of  $g_{t,j}$  in the **numerator**
- from  $\eta_j$ , a factor proportional to  $g_{t,j}^2$  in the **denominator**

This is "scaling" our gradient by more than we want to. So, we'll take the **square root** of the denominator.

$$\eta_j = \frac{\eta}{\sqrt{G_{t,j} + \epsilon}} \quad (7.35)$$

Our goal is to make the scales of different axes more similar, not to neglect dimensions with high gradient (high effect on loss)

This is the completed form of our **adadelta step size rule!**

**Definition 327**

**Adadelta** is a technique for **adaptive step size**, which:

- **Decreases** step size in dimensions with a history of **high-magnitude** gradients
- **Increases** step size in dimensions with a history of **low-magnitude** gradients

Suppose our gradient for weight  $W_j$  at time  $t$  is represented by

$$g_{t,j} = \nabla_W J(W_{t-1})_j$$

~~~~~

This is accomplished by **scaling** the step size η to create η_j :

$$\eta_j = \frac{\eta}{\sqrt{G_{t,j}} + \epsilon}$$

Where $G_{t,j}$ is a "**running average**" of the previous gradients for weight W_j .

$$\begin{aligned} G_{0,j} &= 0 \\ G_{t,j} &= \gamma G_{t-1,j} + (1 - \gamma) g_{t,j}^2 \end{aligned}$$

So, our completed gradient descent rule takes the form:

$$W_{t,j} = W_{t-1,j} - \frac{\eta}{\sqrt{G_{t,j}} + \epsilon} \cdot g_{t,j}$$

7.7.5.6 Sparse Data

One major advantage of adadelta is its use for **sparse datasets**, where many variables only show up in a small percentage of the data.

- If a variable is much less frequent, then the weighted average G_t will be much smaller.
- So, when those data points **do** appear, the step size is much larger.

This allows our model to learn more from variables that don't show up as frequently.

Concept 328

Adadelta often works well with **sparse data**: datasets where many variables rarely show up as non-zero.

The step sizes for these variables become much larger, so our model can learn more from less-common information.

However, this can run the risk of paying attention to variables that are sparse, but **not** especially meaningful.

- It's important to choose your variables carefully, so the model doesn't "learn" from noise.

7.7.5.7 Adagrad

Originally, adadelta derives from a **simpler** method, known as **adagrad**, or "adaptive gradient".

- The main difference with this approach is, rather than find the **weighted average** G_t , we simply **sum** the previous gradients.

The main problem with this approach is that, over time, G_t becomes too **large**. So, our step size becomes too small, and our algorithm slows down.

By averaging, we don't run into this problem of G_t "accumulating": older data is gradually **forgotten**.

7.7.6 Adam

Momentum and Adadelta both bring some benefits:

- **Momentum** averages our current gradient with **previous gradients**, to reduce **oscillation** and make a more direct path to the solution.
- **Adadelta** modifies our **step sizes**: it takes smaller steps in directions of high gradient (reduce overshooting) and takes bigger steps in directions of low gradient (converge faster).

There's nothing structurally incompatible between them. So, why not incorporate both?

- This combination is called **Adam**: it has become the most popular way to handle step sizes in neural networks.

Concept 329

Adam integrates the techniques of both **momentum** and **adadelta**.

7.7.6.1 Momentum and Adadelta

We used two different **running averages**, both using γ for their "**discount factor**": how much we discount the effect of older data.

- We'll replace γ with B_1 (momentum) and B_2 (adadelta).

It's "discounting", or reducing the effect of older data, because $\gamma < 1$.

Notation 330

In the adam algorithm, B_1 and B_2 are **discount factors** replacing γ from momentum and adadelta.

We use the same notation for gradients:

$$g_{t,j} = \nabla_{W_j} J(W_{t-1})_j \quad (7.36)$$

First, we'll keep track of our **averaged gradient**, $m_{t,j}$. This will be the **direction** we plan to move.

$$m_{t,j} = B_1 m_{t-1,j} + (1 - B_1) g_{t,j} \quad (7.37)$$

Then, we'll keep track of the **average squared gradient**, $v_{t,j}$. This will tell us whether to scale up/down our **step size** on each variable.

$$v_{t,j} = B_2 v_{t-1,j} + (1 - B_2) g_{t,j}^2 \quad (7.38)$$

We can combine these into our equation, the way you use momentum and adadelta:

$$W_{t,j} = W_{t-1,j} - \frac{\eta}{\sqrt{v_{t,j} + \epsilon}} \cdot m_{t,j} \quad (7.39)$$

We're not quite done yet, though.

7.7.6.2 Normalization

There's one issue with our running average, that we should address.

Suppose we have a sequence of numbers:

$$[1, 1, 1] \quad (7.40)$$

What's the running average of this sequence of numbers, with $\gamma = .1$? You'd expect it to be 1 for all elements, right?

- We start with $a_0 = 0$.

$$\begin{aligned} a_1 &= .1 * 0 + .9 * 1 = .9 \\ a_2 &= .1 * .9 + .9 * 1 = .99 \\ a_3 &= .1 * .99 + .9 * 1 = .999 \end{aligned} \quad (7.41)$$

It turns out not to be true! Why is that?

Because of our initial term, a_0 : it has nothing to do with the data. Because it'll always be 0, it **deflates** the value of all the remaining averages.

How much does it deflate the average? Well, let's consider the general equation:

$$A_T = \sum_{t=0}^T \gamma^{(T-t)} (1 - \gamma)^t g_t \quad (7.42)$$

What "fraction" of the total is missing, thanks to $A_0 = 0$?

- Well, all of our weighted terms $\gamma^{(T-t)} (1 - \gamma)^t$ should add up to 1: each one represents the "percent/fraction" of the average which comes from that g_t term.

A_0 matches $t = 0$, so we find:

$$A_T = \underbrace{A_0 \gamma^T}_{A_0=0} + \sum_{t=1}^T \gamma^{(T-t)} (1 - \gamma)^t g_t \quad (7.43)$$

So, out of our total 1, or 100%, we're missing γ^T .

- Our new total is $1 - \gamma^T$.
- In order to correct for the "zeroed out" part of our average, we multiply by

$$\frac{\text{Desired total}}{\text{Real total}} = \frac{1}{1 - \gamma^T} \quad (7.44)$$

Concept 331

We've chosen $A_0 = 0$: this is **unrelated** to our real data.

As a result, A_t doesn't accurately reflect our weighted average: it's slightly **smaller**.

- A_0 is "included" as a 0, even though it's not a **real** data point.
- So, whatever **percent** of our data is represented by A_0 , is "falsely empty".

A_0 is scaled by γ^t , so that's the amount **removed** from our "true" weighted average.

The "real" weighted average, that ignores A_0 , has to un-do the deflation caused by including fake, starting data:

$$\hat{A}_0 = A_0 \cdot \frac{1}{1 - \gamma^t}$$

We'll make this correction for our running averages:

$$\begin{aligned}\hat{m}_{t,j} &= \frac{m_{t,j}}{1 - B_1^t} \\ \hat{v}_{t,j} &= \frac{v_{t,j}}{1 - B_2^t}\end{aligned}\quad (7.45)$$

7.7.6.3 Adam

Now, we have our complete equation for adam:

Definition 332

Adam is a technique for improving **gradient descent** that integrates two other techniques. Our computed gradient is given as

$$g_{t,j} = \nabla_W J(W_{t-1})_j$$

- **Momentum** averages our previous gradients with our current one, to reduce oscillation and get a "averaged" view of data. B_1 gives the weight of old data.

$$m_{0,j} = 0 \quad m_{t,j} = B_1 m_{t-1,j} + (1 - B_1) g_{t,j}$$

- **Adadelta** estimates how "steep" our gradient has been recently, and uses it to adjust our step size, for better convergence. B_2 gives the weight of old data.

$$v_{0,j} = 0 \quad v_{t,j} = B_2 m_{t-1,j} + (1 - B_2) g_{t,j}^2$$

We include a **correction** factor for the zeroed initial conditions: $m_0 = v_0 = 0$.

$$\hat{m}_{t,j} = \frac{m_{t,j}}{1 - B_1^t} \quad \hat{v}_{t,j} = \frac{v_{t,j}}{1 - B_2^t}$$

Finally, our update rule takes the form:

$$W_{t,j} = W_{t-1,j} - \frac{\eta}{\sqrt{\hat{v}_{t,j} + \epsilon}} \cdot \hat{m}_{t,j}$$

Impressively, adam is not very sensitive to the initial conditions for ϵ , B_1 and B_2 .

To implement this for NNs, we just need to keep track of each value, for each weight. If you do it systematically, this is easier than it sounds.

It's able to "self-correct" its step sizes and gradient based on data.

7.8 Regularization

Something we haven't discussed in a while, that we might investigate, is **regularization**: techniques against overfitting.

- We teach our model using "training data", which, by chance, may not perfectly reflect the **true** distribution.

We want to apply this to our modern, deep neural networks, with their huge number of **parameters**, and huge amount of **data**. And yet...

These modern neural nets **don't** tend to have as much problem with **overfitting**, and we're not sure **why**!

Regardless, we do still have *some* methods for regularization: this will be our focus for the rest of this chapter.

7.8.1 Methods related to ridge regression

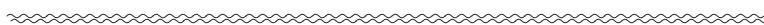
We'll start with methods that we can bring back from classic ridge regression.

7.8.1.1 Early Stopping

One component built into our learning algorithm for regression is **early stopping**: we check if the model is still making **progress**. If it isn't, then we **stop** training.

- The longer we train our model, the more time it has to "**memorize**" the exact structure of our training data: including **noise**.

We would typically either measure the size of the **gradient**, or the change in the **loss**. If either was small, then we might be in a local minimum: we've finished training.



So, we do the same here: after a period of training (over the whole dataset, called an **epoch**), we measure the **loss** on a validation set.

- If the loss stops decreasing, or begins **increasing**, our model is probably **overfitting**. We **stop early**.

Then, you return the weights with the lowest error.

Definition 333

An **epoch** is the time frame during which your model sees your whole **training data** set, **once**.

With **early stopping**, you evaluate your model using your **validation data**, computing the loss.

- If the loss has decreased from the last epoch, you **continue** training.
- If the loss has stopped decreasing, or is increasing, you **stop** training.

This continues until you've either run out of **epochs**, or you've met your **termination** condition, and stopped early.

- Note that sometimes, "epoch" just refers to how long you train before you check your loss.
- In this case, it might be smaller than the whole dataset.

7.8.1.2 Weight Decay

We can also use the same kind of regularization term that we used for linear regression: penalizing the **squared magnitude**.

- Starting with our loss function:

$$J_{\text{loss}} = \sum_{i=1}^n \mathcal{L}(\text{NN}(x^{(i)}), y^{(i)}; W) \quad (7.46)$$

- And we penalize based on the square magnitude of our weights:

$$J = \lambda \|W\|^2 + J_{\text{loss}} \quad (7.47)$$

If we take the gradient, we get:

$$\nabla_W J = 2\lambda W + \nabla_W J_{\text{loss}} \quad (7.48)$$

Let's see how the regularization affects our step:

$$W_t = W_{t-1} - \eta \left(2\lambda \|W_{t-1}\| + \nabla_W J_{\text{loss}} \right) \quad (7.49)$$

It directly subtracts from our weight, **decaying** it.

$$W_t = W_{t-1} \left(1 - 2\lambda\eta \right) - \eta \nabla_W J_{\text{loss}} \quad (7.50)$$

Thus, we call it **weight decay**.

Concept 334

When we apply **square magnitude** regularization to the weights of a neural network, we call it **weight decay**.

$$J_{\text{reg}} = J_{\text{loss}} + \lambda \|W\|^2$$

That's because, when you take the gradient, it directly subtracts from weight W , causing it to **decay** by a factor of $(1 - 2\lambda\eta)$.

$$W_t = W_{t-1} \left(1 - 2\lambda\eta \right) - \eta \nabla_W J_{\text{loss}}$$

7.8.1.3 Perturbation

One last way to reduce overfitting is to add some **random noise** to our data: each variable has a small, random number added to it.

This value is typically **zero-mean** and **normally distributed**:

- Zero-mean: it has 0 effect, on average, so it doesn't bias the data high or low.
- Normally distributed: the noise is **symmetric**: +2 and -2 are equally likely.

How large the noise is, depends on the chosen **variance**.

This reduces overfitting, because if the data is slightly different each time you see it, it's harder to perfectly "memorize" the exact shape and structure.

The "normal distribution" contains more information than that, but the symmetry is important.

Definition 335

Perturbation is a technique where you slightly modify your system.

- In our case, we're **randomly** adding small amount of **noise** to our input data.

This makes it more difficult for the model to **overfit**, because the patterns aren't always exactly the same.

- Only the "general", high-level patterns are preserved, each time you view the dataset.

Of course, if you perturb your data too strongly, you can miss real patterns. Your perturbations shouldn't be too large.

7.8.2 Dropout

We also have **structural** ways of dealing with overfitting. We discussed perturbing the **dataset**, but we could, instead, perturb the **model** itself!

We do this by randomly **removing** some weights from the neural network, and training.

- Each weight has a probability p of being "turned off": the **activation** is set to zero.

$$a_j^\ell = 0 \quad (7.51)$$

- Thus, that neuron's output is **ignored** by the next layer, and receives no training.
- At the next step, we remove a **different** random selection of weights.

Because our model keeps changing slightly, it's harder to exactly **overfit** to the data.



This particular approach also addresses a **different** kind of overfitting:

- One model might heavily "rely" on a **small** number of neurons to make decisions.
- This makes our model less flexible, uses the weights less efficiently.

To solve this, we prevent the neural network from using some of these weights, **randomly**.

Thus, the whole network "**shares**" some responsibility for getting the right answer.

Definition 336

Dropout is a process where, at each training interval, you **randomly** "drop out", or de-activate, some of the weights in the network.

- Each neuron has probability p of being turned off.
- These neurons have their **activation** set to zero: $a_j^\ell = 0$.

This process is designed to reduce **overfitting**. As the network randomly changes, it's harder for it to perfectly match the data structure.

- ~~~~~
- When the network is finished training, we multiply all the weights by p . Why?
 - Because during training, only p fraction of the neurons were active. We want to replicate that average activity level, even when we use all of our neurons.
- ~~~~~

This process is also designed to create "collective responsibility" for your neurons. It prevents your network from relying on a small number of neurons to solve problems.

It generally improves **robustness** against random variations in the data.

~~~~~

This approach has, in recent years, become somewhat less popular, for a couple reasons:

- **Very large** networks don't struggle as much with overfitting.
- CNNs tend not to benefit from this procedure, because of **weight-sharing**: the same weights are re-used in multiple places.
- Like most ML techniques, their usefulness depends on the situation.

We'll discuss CNNs in  
our next chapter.

It still finds use in some smaller models, RNNs, etc.

In many places, it has been replaced by **batch normalization**.

### 7.8.3 Batch Normalization

#### 7.8.3.1 Covariate Shift

Our last approach related to regularization was designed to handle a new type of problem we call **covariate shift**:

When you run **gradient descent** on a neural network, you're adjusting the weights of all of our layers, at the **same time**.

Let's focus on layer 1 and layer 2. By updating layer 1, we've changed the outputs it creates: the same  $x^{(i)}$  now creates a different output, going from  $a_{\text{old}}^1$  to  $a_{\text{new}}^1$ .

But, this output is the **input** of layer 2.

- This means that layer 2 is now receiving **different inputs** than it was before.
- This is a problem: layer 2 just received training based on the **old** inputs  $a_{\text{old}}^1$ !

This makes life a lot harder for our layer 2: not only is it learning to make better predictions, it also has to adjust for the change in layer 1. This is a form of **covariate shift**.

#### Definition 337

**Covariate shift** occurs when the distribution of input variables **changes** over time.

- This can cause our original model to become inaccurate, or "outdated": it was trained on **different** data.

**Internal covariate shift** occurs because of changes to the network itself.

- The distribution of inputs to **later layers** changes, because **earlier layers** have changed through training.

**Example:** If the weights in layer 1 all get smaller (as a side effect of correcting them), layer 2 may have to make all of its weights bigger to compensate.

- Our expectation is that this extra work would slow down training.

#### 7.8.3.2 Layer Normalization

So, we want to counter the problem of how the input to layer 2 **changes** based on layer 1's **learning**.

- But, at the same time, we don't want to **undo** the work that we did **training** layer 1.
- So, we want to preserve the information in layer 1, while making it easier to use.

In our example, we mentioned that the scale of the inputs might get larger: they all get bigger or smaller, at the same time.

But, we often want to know what makes these inputs **different** from each other, so we can compare them: it's not helpful if all of them become larger/smaller.

So, we'll **standardize** each of our mini-batches of data, between each layer: \_\_\_\_\_

- **Subtracting the mean:** we take the mean of the mini-batch input, and subtract it from each data point. So, our standardized data is always centered at 0.
- **Dividing by standard deviation:** we compute how "spread out" the data is, and scale by that factor. Our standardized data is always the same amount "spread out".

This is exactly the same as when we standardized in the Feature Transformation chapter.

We repeat this process in between each layer of our network. Each layer currently receives a set of inputs with mean 0, standard deviation 1. No matter how the earlier layers change.

Below, we exclude the  $\ell$  superscript in  $z^\ell$ ,  $\mu^\ell$ ,  $\sigma^\ell$ , and  $n^\ell$  for readability.

For now, we'll apply this to the pre-activation  $Z$ .

### Concept 338

In order to deal with **internal covariate shift**, we'll take each mini-batch of  $k$  **pre-activations** to each **layer** (pre-activation, not true "input") of our neural network, and we **standardize**/normalize it.

- Each **dimension** of the input is standardized **separately**.

~~~~~

Here, we focus on a single element: dimension i of the j^{th} data point in our batch, z_{ij} .

- We **compute** the mean μ and standard deviation σ , for batch size k .

$$\mu_i = \frac{1}{K} \sum_{j=1}^K z_{ij} \quad \sigma_i^2 = \frac{1}{K} \sum_{j=1}^K (z_{ij} - \mu_i)^2$$

- We **subtract** the mean, divide by the standard deviation (include ϵ term to avoid dividing by zero)

$$\bar{z}_{ij} = \frac{z_{ij} - \mu_i}{\sigma_i + \epsilon}$$

~~~~~

After **standardizing**, this data is sent forward through the network.

- This is equivalent to using a "standardizing function" **after** the weight function  $Z = W^T A$ , and **before** the activation function  $f(Z)$  (now  $f(\bar{Z})$ ).
- We could also standardize **after** activation, but it's unclear which approach is better.

We can insert our new module into our existing model:



Normalization/standardization can be treated just like any other module.

Note that this preserves the **information** in our input data:

- If one data point has one feature much larger than another, you'll still see that: the gap will just be shifted over to zero, and normalized.

**Example:** Suppose you have some data: [1, 2, **100**, 3, 4, 5]. If you standardize, you get

$$[-0.458, -0.433, \textcolor{red}{2.04}, -0.408, -0.383, -0.358,] \quad (7.52)$$

The larger data point still stands out above the rest!

We need to be careful of dimensions:

#### Clarification 339

Normalization relies on the distribution (mean, s.d.) of our **mini-batch**.

But that means we can't just compute one data point at a time: we need to include the whole mini-batch of  $k$  **at the same time**.

So, we have to change the dimensions of  $Z^\ell$ .

$k$  is our **batch size**, while  $n$  is the number of **dimensions**.

- $Z^\ell$  without norm:  $(n^\ell \times 1)$
- $Z^\ell$  with norm:  $(n^\ell \times k)$

We use  $Z_{ij}^\ell$  to indicate the  $i^{\text{th}}$  dimension of the  $j^{\text{th}}$  data point.

#### 7.8.3.3 Post-Normalization: Choose Mean and S.D.

Now, we've adjusted it so that our distribution doesn't "drift", based on our training.

But, now, we've **restricted** our model:

- We don't necessarily want our mean and standard deviation to be 0 and 1: it would be better to be able to **control** it.

To accomplish this, we'll **scale** and **shift** our input. Thus, we're choosing our mean/s.d. in a deliberate way.

- Each dimension needs its own mean and standard deviation.

- We have  $n$  dimensions, and we need one variable to handle mean (or s.d.) for each: we'll need an  $(n \times 1)$  vector.

### Concept 340

By doing **normalization**, we've transformed  $Z$  into  $\bar{Z}$ .

- This "resets" our mean and standard deviation to 0 and 1.

However, we want to be able to **control** our mean and standard deviation. To do so, we introduce two new parameters:

- $G$ : An  $(n \times 1)$  vector that **scales** each of our dimensions  $\bar{Z}_i$ , giving our **standard deviations**.
- $B$ : An  $(n \times 1)$  vector that **shifts** each of our dimensions  $\bar{Z}_i$ , giving our **means**.

Thus, we get the true output of **batch normalization**:

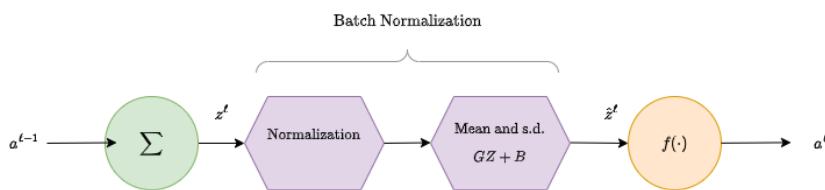
$$\hat{Z}_{ik} = G_i * \bar{Z}_{ij} + B_i$$

**Example:** Here's a sample example using a vector  $\bar{z}_j$ : only considering one, post-normalization data point  $j$ .

$$\begin{bmatrix} \hat{Z}_{1j} \\ \hat{Z}_{2j} \\ \vdots \\ \hat{Z}_{kj} \end{bmatrix} = \begin{bmatrix} G_1 \\ G_2 \\ \vdots \\ G_k \end{bmatrix} * \begin{bmatrix} \bar{Z}_{1j} \\ \bar{Z}_{2j} \\ \vdots \\ \bar{Z}_{kj} \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_k \end{bmatrix} \quad (7.53)$$

Where  $*$  indicates element-wise multiplication.

If we include this in our neuron graph, we now have two new modules:



#### 7.8.3.4 Full definition

**Definition 341**

**Batch Normalization** is a process where we

- Standardize the pre-activation for each layer using the mean  $\mu_i$  and standard deviation  $\sigma_i$  (for the  $i^{\text{th}}$  dimension). Select  $\epsilon: 0 < \epsilon \ll 1$ .

$$\bar{Z}_{ij} = \frac{Z_{ij} - \mu_i}{\sigma_i + \epsilon}$$

- Choose the new mean and standard deviation for the pre-activation using  $(n \times 1)$  vectors  $G$  and  $B$

$$\hat{Z}_{ik} = G_i * \bar{Z}_{ij} + B_i$$



This process is meant to accomplish the following:

- Remove possible **internal covariance shift**: training earlier layers may change the scale of inputs to later layers.
  - This could make training more difficult.
- Allow our model to **select** a particular mean and s.d. for its pre-activation values, rather than arriving at them by chance.

It also tends to have a regularizing effect, and, in some learning algorithms, has replaced dropout.

#### 7.8.3.5 Effectively Perturbs Data

We're not actually sure why normalization helps our models train. We originally designed it for **internal covariate shift**, but we're not sure if that's really **why** it works.

One explanation might be that, due to random sampling, each mini-batch ends up slightly **modified** by our normalization.

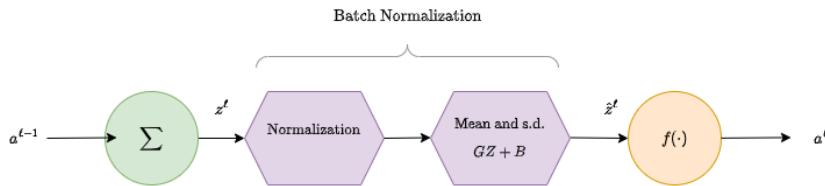
- Since each batch is likely to have a slightly different mean/standard deviation, each one ends up differently "perturbed" by normalization.

### 7.8.4 Applying batch normalization to backprop

**Remark (Optional)**

The following section mostly deals with the details of computing batch normalization derivatives.

As we showed with our figure,



Batch normalization adds two units that **interrupt** our chain of nested functions.

That means we need to figure out how to do backprop, bridging across the new gap between  $\textcolor{blue}{Z}$  and  $\hat{\textcolor{red}{Z}}$ .

So, that only leaves a couple problems:

- "Bridging the gap" between derivatives before and after BN, with  $\partial \hat{\textcolor{red}{Z}}^t / \partial \textcolor{blue}{Z}^t$ .
- Gradients for  $\textcolor{green}{G}$  and  $\textcolor{purple}{B}$ : they're parameters now, too, so we need to train them.

**Concept 342**

Introducing batch normalization add new functions **in between** our old ones.

- Our input data has to travel through those additional layers.
- This **changes** the relationship between our current value, and the output loss.

So, in order to do backprop correctly, we have to figure out the **derivatives** of those functions.

#### 7.8.4.1 Bridging the gap

We want to connect the start and end of batch normalization:

$$\frac{\partial \hat{\textcolor{red}{Z}}}{\partial \textcolor{blue}{Z}} \tag{7.54}$$

As usual, with the chain rule, we'll connect them by considering any values/function **between** them.

In this case,  $Z$  is normalized ( $\bar{Z}$ ), and **then** we apply G and B ( $\hat{Z}$ ). We're missing the "normalized" step.

$$\frac{\partial \hat{Z}}{\partial Z} = \frac{\partial \hat{Z}}{\partial \bar{Z}} \cdot \frac{\partial \bar{Z}}{\partial Z} \quad (7.55)$$

Let's compare any two of these :  $\hat{Z}$  and  $\bar{Z}$ , for example.

- These are both  $(n \times k)$  matrices.
- That means that each has two dimensions of variables. If we were to take the derivative between them, we would need  $2 * 2$  axes: a **4-tensor**.

That sounds terrible. Instead, we'll compute these derivatives **element-wise**.

$$\frac{\partial \hat{Z}_{ab}}{\partial Z_{ef}} = \frac{\partial \hat{Z}_{ab}}{\partial \bar{Z}_{cd}} \cdot \frac{\partial \bar{Z}_{cd}}{\partial Z_{ef}} \quad (7.56)$$


---

#### 7.8.4.2 Indexing

First, let's simplify these indices: only some pairs of elements matter.

$$\hat{Z}_{ik} = G_i * \bar{Z}_{ik} + B_i \quad (7.57)$$

It seems that  $\hat{Z}_{ik}$  is only affected by the element with the same indices:  $\bar{Z}_{ik}$ .

##### Concept 343

$\hat{Z}_{ik}$  is only a function of the same index element  $\bar{Z}_{ik}$ .

- Any other elements from  $\bar{Z}$  has no effect.

$$(a \neq c \text{ or } b \neq d) \implies \frac{\partial \hat{Z}_{ab}}{\partial \bar{Z}_{cd}} = 0$$

So, we write our remaining derivatives as  $\partial \hat{Z}_{ik} / \partial \bar{Z}_{ik}$ .

What about the other derivative?

$$\bar{Z}_{ik} = \frac{Z_{ik} - \mu_i}{\sigma_i + \epsilon}$$

$\mu_i$  and  $\sigma_i$  include various different data points  $Z_{ik}$ , but only the  $i^{\text{th}}$  dimension.  $Z_{ik}$  requires the exact same indices.

**Concept 344**

$\bar{Z}_{ik}$  is only a function of elements in the same dimension  $i$ ,  $Z_{ij}$ .

$$c \neq e \implies \frac{\partial \bar{Z}_{cd}}{\partial Z_{ef}} = 0$$

Our remaining derivatives take the form  $\partial \bar{Z}_{ik} / \partial Z_{ij}$ .

If we boil this down, we get all of our non-zero derivatives:

$$\frac{\partial \hat{Z}_{ik}}{\partial Z_{ij}} = \frac{\partial \hat{Z}_{ik}}{\partial \bar{Z}_{ik}} \cdot \frac{\partial \bar{Z}_{ik}}{\partial Z_{ij}} \quad (7.58)$$

**7.8.4.3 Computing  $\partial \hat{Z}_{ik} / \partial \bar{Z}_{ik}$** 

We return to our previous equation:

$$\hat{Z}_{ik} = G_i * \bar{Z}_{ik} + B_i \implies \frac{\partial \hat{Z}_{ik}}{\partial \bar{Z}_{ik}} = G_i \quad (7.59)$$

**7.8.4.4 Computing  $\partial \bar{Z}_{ik} / \partial Z_{ij}$** 

For the other derivative:

$$\bar{Z}_{ik} = \frac{Z_{ik} - \mu_i}{\sigma_i + \epsilon} \quad (7.60)$$

This gets a bit complicated, because  $Z_{ij}$  can affect three different terms:  $Z_{ik}$ ,  $\mu_i$ , and  $\sigma_i$ .

We'll solve this by using the multi-variable chain rule.

We're linearly adding the effect due to each of these three variables, separately.

$$\frac{\partial \bar{Z}_{ik}}{\partial Z_{ij}} = \underbrace{\frac{\partial \bar{Z}_{ik}}{\partial Z_{ik}} \cdot \frac{dZ_{ik}}{dZ_{ij}}}_{Z_{ik}'s \text{ effect}} + \underbrace{\frac{\partial \bar{Z}_{ik}}{\partial \mu_i} \cdot \frac{d\mu_i}{dZ_{ij}}}_{\mu_i's \text{ effect}} + \underbrace{\frac{\partial \bar{Z}_{ik}}{\partial \sigma_i} \cdot \frac{d\sigma_i}{dZ_{ij}}}_{\sigma_i's \text{ effect}} \quad (7.61)$$

In each of these terms, we treat the other two variables as "constant".

**7.8.4.5 Lots of mini-derivatives**

Let's compute the  $\bar{Z}$  derivatives.

$$\bar{Z}_{ik} = \frac{Z_{ik} - \mu_i}{\sigma_i + \epsilon} \quad (7.62)$$

gives us

$$\frac{\partial \bar{Z}_{ik}}{\partial Z_{ik}} = \frac{1}{\sigma_i + \epsilon} \quad \frac{\partial \bar{Z}_{ik}}{\partial \mu_i} = \frac{-1}{\sigma_i + \epsilon} \quad \frac{\partial \bar{Z}_{ik}}{\partial \sigma_i} = -\left(\frac{Z_{ik} - \mu_i}{(\sigma_i + \epsilon)^2}\right) \quad (7.63)$$

Now, let's compute the  $Z_{ij}$  derivatives.

$$\boxed{\frac{\partial Z_{ik}}{\partial Z_{ij}} = \delta_{jk}} = \mathbf{1}(j=k) = \begin{cases} 1 & j=k \\ 0 & j \neq k \end{cases} \quad (7.64)$$

$$\mu_i = \frac{1}{K} \sum_{j=1}^K Z_{ij} \implies \boxed{\frac{\partial \mu_i}{\partial Z_{ij}} = \frac{1}{K}} \quad (7.65)$$

$$\sigma_i^2 = \frac{1}{K} \sum_{j=1}^K (Z_{ij} - \mu_i)^2 \implies \boxed{\frac{\partial \sigma_i^2}{\partial Z_{ij}} = \frac{Z_{ij} - \mu_i}{K\sigma_i}} \quad (7.66)$$

#### 7.8.4.6 Assembling our derivatives

Now, we plug them in.

$$\frac{\partial \bar{Z}_{ik}}{\partial Z_{ij}} = \frac{\partial \bar{Z}_{ik}}{\partial Z_{ik}} \cdot \frac{dZ_{ik}}{dZ_{ij}} + \frac{\partial \bar{Z}_{ik}}{\partial \mu_i} \cdot \frac{d\mu_i}{dZ_{ij}} + \frac{\partial \bar{Z}_{ik}}{\partial \sigma_i} \cdot \frac{d\sigma_i}{dZ_{ij}} \quad (7.67)$$

First, the  $\bar{Z}$  derivatives.

$$\frac{\partial \bar{Z}_{ik}}{\partial Z_{ij}} = \left(\frac{1}{\sigma_i + \epsilon}\right) \cdot \frac{dZ_{ik}}{dZ_{ij}} - \left(\frac{1}{\sigma_i + \epsilon}\right) \cdot \frac{d\mu_i}{dZ_{ij}} - \left(\frac{Z_{ik} - \mu_i}{(\sigma_i + \epsilon)^2}\right) \cdot \frac{d\sigma_i}{dZ_{ij}} \quad (7.68)$$

And now the  $Z_{ij}$  derivatives.

$$\frac{\partial \bar{Z}_{ik}}{\partial Z_{ij}} = \left(\frac{1}{\sigma_i + \epsilon}\right) \cdot \delta_{jk} - \left(\frac{1}{\sigma_i + \epsilon}\right) \cdot \left(\frac{1}{K}\right) - \left(\frac{Z_{ik} - \mu_i}{(\sigma_i + \epsilon)^2}\right) \cdot \left(\frac{Z_{ij} - \mu_i}{K\sigma_i}\right) \quad (7.69)$$

#### Key Equation 345

We have found the batch normalization derivatives

$$\frac{\partial \hat{Z}_{ik}}{\partial Z_{ik}} = G_i$$

$$\frac{\partial \bar{Z}_{ik}}{\partial Z_{ij}} = \frac{1}{K(\sigma_i + \epsilon)} \left( K\delta_{jk} - 1 - \frac{(Z_{ik} - \mu_i)(Z_{ij} - \mu_i)}{\sigma_i(\sigma_i + \epsilon)} \right)$$

Which we multiply together to find  $\partial \hat{Z}_{ik} / \partial Z_{ij}$ .

Once we've computed these derivatives, we can use them to extend the chain of  $\partial L / \partial \hat{Z}_{ik}$ .

We're aiming to handle both of the batch normalization functions, with  $\partial L / \partial Z_{ij}$ .

We're going from  $\hat{Z}_{ik}$ , which is post-BN, to  $Z_{ij}$ , which is pre-BN.

- $Z_{ij}$  affects every data point  $\hat{Z}_{ik}$ , and every data point affects L.
- So, we'll have to consider every data point k using the multi-variable chain rule:

$$\frac{\partial L}{\partial Z_{ij}} = \underbrace{\sum_{k=1}^K}_{\text{MV Chain Rule}} \overbrace{\frac{\partial L}{\partial \hat{Z}_{ik}} \cdot \frac{\partial \hat{Z}_{ik}}{\partial Z_{ij}}}^{\text{Data point k's effect}} \quad (7.70)$$

We can use this to go further back in our layers: as far as we want, as long as we don't forget this derivative!

#### 7.8.4.7 Gradients for B and G

We want  $\partial L / \partial B$  and  $\partial L / \partial G$ . Thanks to the work we did just now, we can travel backwards through numerous layers, to reach any  $B^\ell$  and  $G^\ell$ .

So, we'll assume we have  $\partial L / \partial \hat{Z}^\ell$ . Once again, we'll omit the layer notation.

- We'll focus on a single bias,  $B_i$ : this biases one dimension of  $\hat{Z}_{ik}$ .

$$\hat{Z}_{ik} = G_i * \bar{Z}_{ik} + B_i \implies \frac{\partial \hat{Z}_{ik}}{\partial B_i} = 1 \quad (7.71)$$

- This bias is applied to every of our K data points: every data point affects the loss L. So, we'll have to sum them with the multi-variable chain rule.

This is exactly the same as how we did  $\partial L / \partial Z_{ij}$ .

$$\frac{\partial L}{\partial B_i} = \sum_{k=1}^K \frac{\partial L}{\partial \hat{Z}_{ik}} \cdot \frac{\partial \hat{Z}_{ik}}{\partial B_i} = \boxed{\sum_{k=1}^K \frac{\partial L}{\partial \hat{Z}_{ik}}} \quad (7.72)$$

Now, we focus on a single scaling factor  $G_i$ :

$$\hat{Z}_{ik} = G_i * \bar{Z}_{ik} + B_i \implies \frac{\partial \hat{Z}_{ik}}{\partial G_i} = \bar{Z}_{ik} \quad (7.73)$$

And once again, we add across different data points:

$$\frac{\partial L}{\partial G_i} = \sum_{k=1}^K \frac{\partial L}{\partial \hat{Z}_{ik}} \cdot \frac{\partial \hat{Z}_{ik}}{\partial G_i} = \boxed{\sum_{k=1}^K \frac{\partial L}{\partial \hat{Z}_{ik}} \bar{Z}_{ik}} \quad (7.74)$$

We're finished.

**Key Equation 346**

Here, we have the gradients for the batch scaling coefficients, B and G.

$$\frac{\partial L}{\partial B_i} = \sum_{k=1}^K \frac{\partial L}{\partial \hat{Z}_{ik}}$$

$$\frac{\partial L}{\partial G_i} = \sum_{k=1}^K \frac{\partial L}{\partial \hat{Z}_{ik}} \bar{Z}_{ik}$$

## 7.9 Terms

### Section 7-1

- 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  $A^\ell$
- Step function
- ReLU function
- Sigmoid function
- Hyperbolic tangent function
- Softmax function

## Section 7-1.5

- 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

## Section 7-2

- Mini-batch
- Vanishing/Exploding Gradient
- Weighted Average
- Running Average
- Exponential Moving Average
- Discount Factor
- Momentum
- Adadelta
- Adagrad (Optional)
- Adam
- Normalization
- Early stopping (Review)
- Weight Decay

- Perturbation
- Dropout
- Covariate Shift
- Internal Covariate Shift
- Batch Normalization
- Multivariable Chain Rule (Review)

# CHAPTER 8

---

## Autoencoders

---

Through neural networks, we've **upgraded** the set of models we can use to do classification and regression tasks.

- Neural networks become more complex and **expressive** as we add more **layers**.

We'll spend the next few chapters exploring NNs: creating new variants (CNNs, RNNs), for example.

In this chapter, we'll investigate a different kind of application: **autoencoders**.

### 8.0.1 Unsupervised Learning

In chapter 6, we discussed an **unsupervised** learning problem: clustering.

- Classification tells us which classes to use. By contrast, clustering **doesn't** "know" the classes we're looking for.
  - Instead, we discover new classes ("clusters"), using the k-means algorithm.
- This lack of guidance is what makes clustering **unsupervised**.

Clustering was used as a form of **data analysis**: we were able to learn more about the **structure** of our data, by finding what sorts of "groups" existed.

We'll use autoencoders for a different task, that follows the same theme: learning more about the data, by attempting to look for an **unknown solution** to a simple problem.

## 8.0.2 Autoencoders: Compression

This time, our problem is not cluster-finding, but instead, **compression**. We want to take our input data, and find a more **space-efficient** way to represent it.

Typically, this means reducing the number of **dimensions**/variables.

- **Example:** turning a 10-dim data point into a 4-dim data point.

Why would we do this? Because of what we gain from this task:

- A good compression algorithm should be able to be **decompressed**, while keeping the result mostly similar.
  - That means that our compression must preserve **essential** information, so it's possible to retrieve later.
- By observing what's the algorithm decides to preserve and discard, can teach us what matters, and what doesn't.

### Concept 347

Learning a **compression algorithm** for your dataset creates a **simplified** representation, that still keeps the most important, distinct information.

Based on the information we find in that representation, we can figure out what components were "**important**".

Our compression/decompression system needs to do two things:

- **Reproduce** the original data well
- Do so in a way that lets us **distinguish** one data point from another

Based on this, a trained autoencoder can provide us some unexpected insights.



We can better see this with an example.

**Example:** Consider a database of human faces.

- It's more space-efficient if you can just re-use a "**template**" for a face, and just modify it.
  - So, your model might memorize what a face "generally" looks like.
  - That way, it doesn't have to waste space in the compression for data that appears frequently.

- Then, your compressed data only needs to store what's special, or different, about the face it represents.
  - So, you learn what separates different faces from each other, based on the info in the compressed model.

### 8.0.3 Training

This representation might even be easier for a new model to **train** with:

- We've omitted some unnecessary information that can **distract** our model.
- With a simpler input, it's faster to train, and compute answers.

The model can overfit to **noise** in these extra variables.

#### Concept 348

Just like how **clustering** is used to improve downstream tasks, **compression** can be, too.

Compressing your input can improve learning, so it takes less data to train.

#### Clarification 349

Not all compression is created equal!

Auto-encoding compresses in a way that contains **relevant information**.

However, in the Feature representation chapter, we discussed **binary code**: representing a number in **binary**, because it requires fewer features.

- This kind of compression doesn't emphasize what's "important" about the feature representation.

Binary compression is **worse**, not better! Instead of isolating useful information, it **obscures** it, forcing the model to learn binary.

## 8.1 Autoencoder Structure

### 8.1.1 Visualization

Our encoder is a **compression** algorithm, which takes an input  $x^{(i)}$ , and returns a new "transformed" input  $a^{(i)}$ , with a lower dimensionality.

$$\underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_d \end{bmatrix}}_{k < d} \longrightarrow \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_k \end{bmatrix} \quad (8.1)$$

Note: you don't have to take the vectors in equation 8.1 literally.

You're not **required** to have  $\text{Dimension}(a) > 2$ , as in 8.1.

In other words, we're going from  $x \in \mathbb{R}^d$  to  $a \in \mathbb{R}^k$ .

**Example:** Take the classic problem of the MNIST dataset: identifying the identity of a digit based on a  $28 \times 28$  grid of pixels.



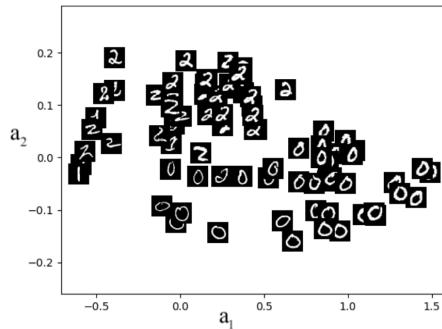
Here's an example of one data point: this represents the digit 9.

- That means our input data has 784 dimensions:  $x \in \mathbb{R}^{784}$ .
- Below, we've used an encoder to compress it down to 2 dimensions:  $a \in \mathbb{R}^2$ .

Each pixel is actually **restricted** to  $[0, 255]$ , but this statement is still technically true.

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{784} \end{bmatrix} \longrightarrow \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \quad (8.2)$$

- The x-axis and y-axis indicate the two dimensions of our output,  $a$ .



Notice that digits with the same identity are near each other: our compressed representation seems to be storing some useful data about digits!

Somehow, we've created a **simplified** representation that, despite having only 2 variables, has a lot of useful information!

- With only a couple labelled data points, we could guess the digits of most of these pictures, based on how close they are.

#### Clarification 350

This property, of "similar" data points, being **close** in the latent space, is **not** guaranteed, for any **compressed representation** you create.

However, it's a property we *hope* to find, in a useful one.

But whether or not our data "organizes" nicely like this, we've still demonstrated another benefit of compression:

- It allows us to transform our data into a lower-dimensional, more **digestible** format.
- So, we can create better visualizations.

#### Concept 351

One application of **compression** is using it for **visualization**.

A lower-dimensional version of a dataset is usually easier to diagram in a way humans can **interpret** directly.

- Because this representation tends to be more **dense** with information, it's often easier to draw useful conclusions.

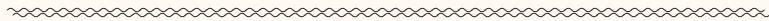
It can also let us make simple predictions, or find patterns, since there are **fewer** variables to pay attention to.

This compressed version of data is called a "latent representation".

**Definition 352**

The **latent representation** of our data is the **compressed** version, containing as much useful information as possible, with fewer variables.

- We call it **latent** because original data  $x$  is in a "hidden" form, but we can retrieve it by **decompressing**.



The **latent space** represents all of the possible latent representations.

- This "space" follows the tradition of "input/feature spaces": sets with structure. In this case, the **distance** between our data gives us the **structure**.

Generally, a good latent space preserves **information**: the reconstructed input still **communicates** what we were interested in, from the original input.

### 8.1.2 Anatomy of an Autoencoder

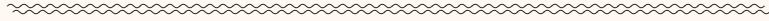
Our autoencoder's purpose is **compression**, but we need to make sure that this compression preserves the information that we want.

**Definition 353**

An **autoencoder** comes in two parts:

- An **encoder**, which **compresses** our ( $d$ -dim) input into the ( $k$ -dim) latent representation.
- A **decoder**, which **de-compresses** our latent representation, to try to re-create the input.
  - This is used to make sure that our representation contains the information we need, to accurately re-construct the input.

The goals are:

- To create a **smaller** latent representation, with dimensions  $k < d$
  - To make sure that this latent representation can **accurately** re-construct our input
- 
- The encoder and decoder can be any kind of function, but we will use **neural networks**.

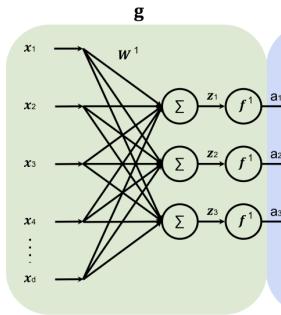
This is where neural networks shine: with more powerful model class, we can do more complex math to create our "encoding".

### 8.1.3 One layer encoder/one layer decoder

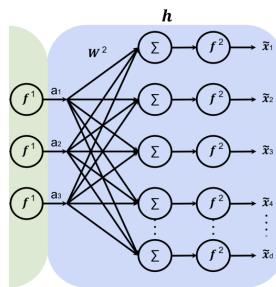
For a simple demonstration, we'll use a version with a one-layer encoder and a one-layer decoder.

We'll take our ( $\text{d-dim}$ ) input, and compress it into a ( $3\text{-dim}$ ) latent representation.

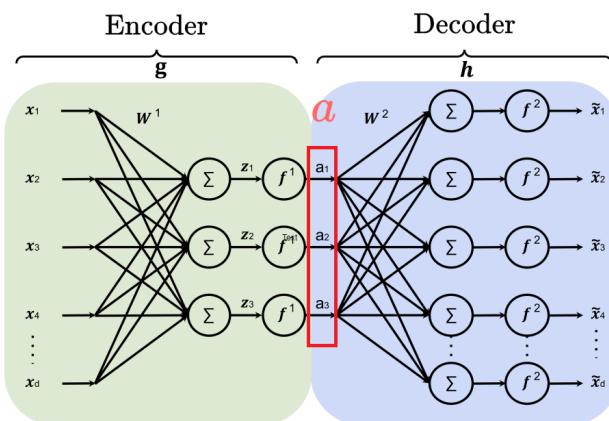
- Encoder: input  $x \in \mathbb{R}^d$ , output (compressed)  $a \in \mathbb{R}^3$ .



- Decoder: input (compressed)  $a \in \mathbb{R}^3$ , output (decompressed)  $\tilde{x} \in \mathbb{R}^d$ .



Taken together, we get our autoencoder:



Note that, in addition to  $W^1$  and  $W^2$ , we have a set of offsets that are not shown:  $W_0^1$  and  $W_0^2$ .

Let's run through our network:

- The **input**  $x$  goes through the encoder network. This compresses into the **latent representation**  $a$ .

- $W^1$  has shape  $(d \times k)$ , or equivalently,  $W^1 \in \mathbb{R}^{d \times k}$
- $W_0^1$  has shape  $(k \times 1)$ , or equivalently,  $W_0^1 \in \mathbb{R}^k$

$$\textcolor{brown}{z}^1 = (W^1)^T \textcolor{green}{x} + W_0^1 \quad \textcolor{red}{a} = f(\textcolor{brown}{z}^1) \quad (8.3)$$

- The **latent representation** goes through the decoder network. This de-compresses it into the **re-constructed input**.

- $W^2$  has shape  $(k \times d)$ , or equivalently,  $W^2 \in \mathbb{R}^{k \times d}$
- $W_0^2$  has shape  $(d \times 1)$ , or equivalently,  $W_0^2 \in \mathbb{R}^d$

$$\textcolor{brown}{z}^2 = (W^2)^T \textcolor{red}{a} + W_0^2 \quad \tilde{x} = f(\textcolor{brown}{z}^2) \quad (8.4)$$

The red layer in the center, is the "**latent representation**": the latent representation is **not** the output.

Our autoencoder can have more layers than we do here: this was just an example.

#### 8.1.4 Autoencoders in general

Let's focus on that point about the "red layer" in the center:

##### Clarification 354

When we train an autoencoder, our goal is to create a **latent representation**.

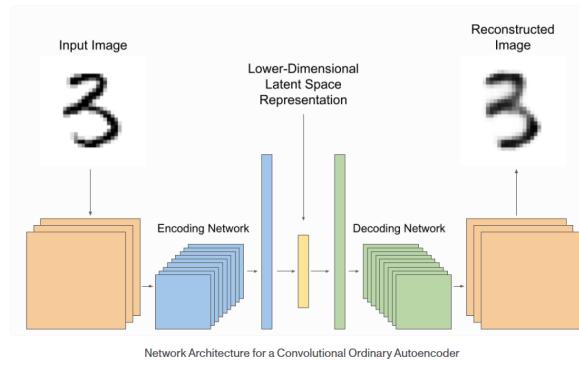
Deceptively, however, that representation is **not the output** of our autoencoder.

- Rather it's in the **middle** of the network: the output of the **encoder**, input to the decoder.

The actual autoencoder output is an **approximate** re-construction of our **input**.

Note that point: we're *approximately* re-constructing our input.

- The **quality** of the re-construction depends on our choice of encoder, and the size of our latent space.
- A bigger latent space (more dimensions) generally allows for a **better** re-construction, but takes up **more** space.



An example of the original vs reconstructed image from MNIST. Credit to [assemblyai.com](https://assemblyai.com)

The reconstructed 3 above is still very recognizable, but it's clearly been "degraded" somewhat: it's not the same.

#### Clarification 355

The reconstructed input is usually **not exactly the same** as the input.

$$x \neq \tilde{x}$$

Our re-construction is an **approximation**.

We're essentially running our input through a "**bottleneck**", in a lower dimension.

- We do so, hoping that the right compression size "squeezes out" only unnecessary information.

#### Concept 356

An autoencoder is, technically, just a normal neural network, where:

- We want the **input** to equal the **output** (re-constructed input)
- We monitor an **intermediate layer** (latent representation), where the layer output size ( $k$ ) **decreases** below the input size ( $d$ )
- If our input and output match well enough, then we use the **output** of that **intermediate layer**.

## 8.2 Autoencoder Learning

Now that we've defined our autoencoder, it's time to **train** it.

What's our objective? Well, we want to create a lower-dimensional representation that can be used to **re-construct** the original.

- We've handled the lower-dim aspect with the **structure** of the neural network: the last layer of the **encoder** will output  $k$  values, giving our **latent representation**. 
- So now, we need to show that this representation contains the information we want: it's able to **re-construct** the original.
  - This is the job of the **decoder**.

Where  $k < d$ , given  $d$  is the original input dimension.

That latter point is what we need to address: we need to check the quality of our re-construction,  $\tilde{x}$ .

### Concept 357

We measure the **quality** of our autoencoder by measuring the **similarity** between the original input  $x$ , and the re-constructed input  $\tilde{x}$ .

- If the re-construction is similar to the original input, that means our latent representation successfully **encodes** information about  $x$ .

So, we need some kind of **similarity** metric. We'll encode this into our loss function,  $\mathcal{L}$ .  $\mathcal{L}$  tells us how **different** our re-construction is, from the original.

- For **continuous** variables, you might use **squared distance** between  $x$  and  $\tilde{x}$ : loss  $\mathcal{L}_{SE}$ . 

$$\mathcal{L}_{SE} = \|x - \tilde{x}\|^2 = \sum_{i=1}^d (x_i - \tilde{x}_i)^2 \quad (8.5)$$

"SE" stands for "squared error". It's different from MSE, "mean squared error", because we're not dividing by  $d$ .

Note that often, an input does not only contain one data type: it may include different types of **discrete** data.

So, you may need to use different loss functions for different dimensions of the input.



Now that our problem is fully framed, we can simply **optimize** it, to minimize loss, using our parameters.

**Concept 358**

Once we've chosen our loss function, we can **optimize** our autoencoder as an ordinary neural network, in order to create our **encoder** for latent representations.

- $W_{en}$  and  $W_{de}$  are our encoder and decoder weights, respectively.

Our goal is to optimize these weights, with respect to the **loss**, over our  $n$  data points:

$$W_{en}^*, W_{de}^* = \operatorname{argmin}_{W_{en}, W_{de}} \left( \sum_{i=1}^n \mathcal{L}(\tilde{x}^{(i)}, x^{(i)}) \right)$$

Or, if we want to be more explicit,

$$W_{en}^*, W_{de}^* = \operatorname{argmin}_{W_{en}, W_{de}} \left( \sum_{i=1}^n \mathcal{L}(\text{NN}(x^{(i)}; W_{en}, W_{de}), x^{(i)}) \right)$$

~~~~~

- As usual for neural networks, we often optimize using gradient descent via **back-propagation**.

Example: For our one-layer encoder/decoder above, we would optimize over W^1, W_0^1, W^2, W_0^2 .

Remember that W^* notation is used to indicate "optimal" parameters.

8.3 Evaluating an Autoencoder

After **training** our autoencoder, we want to be able to **confirm** that it does what we want.

What do we **want**?

- A representation that contains **fewer** dimensions than the input: $k < d$.
 - Remember that k , in this case, is the dimension of our **latent** representation, and d is the dimension of our **input**.
- For this representation to contain useful **information** about our input.
 - This second aspect is (hopefully) addressed by our **loss** function.
 - If our **re-constructions** are really good, then we've managed to preserve our information.

Often, a latent representation can be **much** smaller than the input.

Remember our MNIST example at the start of the chapter, going from 784 dimensions, to 2.

8.3.1 Dimensionality of a

Our loss function handles the latter of these two problems, but the **dimensionality** is based on our **choice** of NN structure.

Well, if we're compressing our data, we want to reduce our number of dimensions, typically. But there's a tradeoff:

Concept 359

The **smaller** our latent representation is, the **less information** we can store in it.

- So, our re-constructions will be generally **worse**.

But a **larger** latent representation uses more **space**/computation, and doesn't **filter** out as much unnecessary, distracting information.

Clarification 360

Because we want to **compress** our data, we require $k < d$.

If we allow $k = d$, then our "latent representation" could be the **exact same** as our original representation.

- In fact, that would be the most **efficient** way to always be correct.
- If $k > d$, then we have extra dimensions, prone to overfitting.

If we **remove** the $k < d$ requirement, we're not forcing our network to do any real work, and our representation **doesn't** have to become more efficient.

In short: if we're not making the dimensions smaller, then we're not really compressing our data!

But what if we wanted to try making the dimensions **larger** on purpose? Does that have applications?

Concept 361

You might wonder if $k > d$ would let us "**unpack**" some of our input data into those extra dimensions.

An encoder with $k > d$ is called an **overcomplete autoencoder**.

Usually, this is **not** desirable:

- Our goal of "recreating the input" doesn't lend it itself well to "unpacking the data": it's usually easier to just **copy** the input.
- Moreover, **overfitting** makes these autoencoders hard to train.

That said, this sort of approach does have applications in de-noising, and learning sparse representations.



Just like in clustering, your exact choice of latent dimensionality k (usually $k < d$) is often subjective or task-based, and requires some trial and error.

We might have different considerations:

- How well does the plot seem to organize our data?
- Are we missing some crucial kind of information?
- Have we encoded information we don't care about on accident?

And more.

8.3.2 Data Analysis

One of the reasons we wanted to design autoencoders was to learn more about our data.

So, a useful encoder might be one that gives us some new or interesting insights.

Concept 362

We can learn more about our (already trained) encoding by **experimenting** with it.

How? By **modifying** the latent representation a , and seeing how that affects the reconstructed version \tilde{x} .

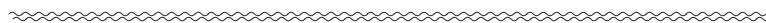
- We could start with a known data point, and modify one **dimension** of it, a_j .
- If we increase a_j and get a noticeable change, we can make guesses about what that dimension "**represents**".

Example: Suppose that we embedded the MNIST digits with k dimensions.

- We select one random data point, $x^{(i)}$. This data point happens to be a picture of the number 6.
- We scale up/down one dimension, a_j .
- We might see that the line thickness of the 6 increases. Maybe a_j represents line thickness.

There could be many possible features: how "angular" the number is, how it loops, etc.

Not to say that those are the particular features you *will* find. In fact, sometimes, it's totally unclear what your latent representation is "representing".



A few other examples of how to do our data analysis:

Concept 363

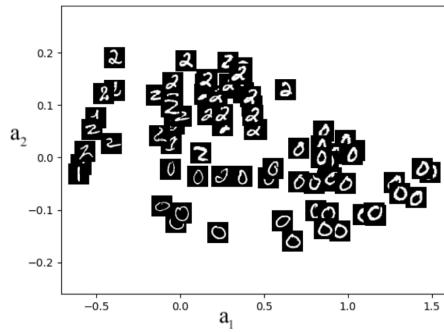
Rather than "experimenting" with individual axes, we could **plot** data points in the **latent space**.

There are two ways to do this:

- Take **real** data points, and plot where they appear in the latent space, compared to how the **input** looks.
- Directly **sample** points from the latent space, and see what their **re-construction** looks like.

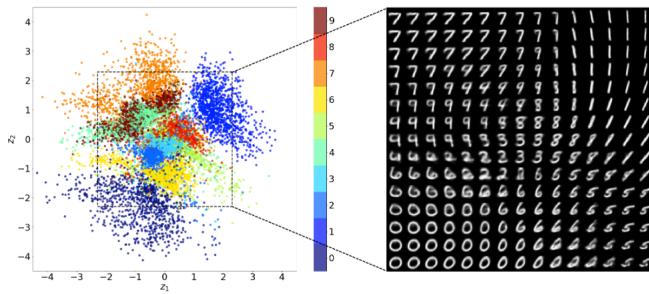
In both cases, we get an idea of how the autoencoder interprets the data it receives.

Example: We started the chapter with an example of the former technique:



Here, we take real data points, and plot them in the latent space.

Example: We could also use the latter: seeing how our re-construction appears, as we modify our latent variables.



The left shows the position of real digits in latent space, while the right shows our reconstructions, moving in a grid across the space. Credit to [argmax.ai](#).

Concept 364

Lastly, we could see what the model thinks the data **generally** looks like, based on the biases/offsets in the network.

- The offset values (W_0^ℓ) are the **same**, regardless of the data point.
- We could set all of the values of α to 0: in a **linear** autoencoder, this would give the **average** of our data points.

If our data clusters around the average, it would be reasonable to expect the average to be **somewhat similar** to all of our data.

- And if it looks similar to each data point, it could somewhat **represent** what it looks like in general.

In a **non-linear** autoencoder, our above approach doesn't give us the **average**, but could be useful regardless.

After analyzing all our dimensions, we can try to figure out what **information** the encoding decided was "important", or at least, necessary for re-construction.

This kind of analysis has a wide array of applications, including natural language processing, disease subtyping, and image processing.

8.3.3 Downstream Tasks

If we're using the encoder for downstream tasks, we can evaluate the autoencoder based on the performance of those downstream tasks.

- We mentioned the same kind of metric when we discussed clustering.

Concept 365

Performance on **downstream tasks** is one way to evaluate the quality of an encoding.

One simple approach, is to use **semi-supervised learning**.

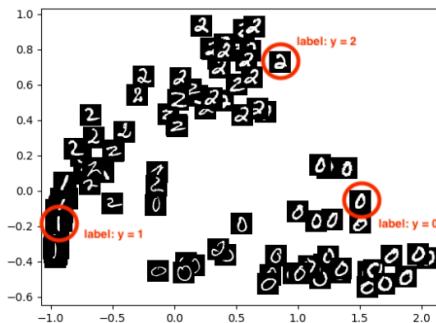
Definition 366

Semi-supervised learning provides labels for only **some** of the data we use to train. The rest of it is unlabelled.

So, the model has to **extrapolate** from that data, to figure out a pattern for the rest of the data.

If we have a **meaningful** latent representation, we could use it to help with this type of problem.

Example: We'll re-use our MNIST example. Suppose we were only given a **few** labelled digits:



Only the three labelled data points are "supervised".

- In this case, we could label many of these digits, just based on what we have.

3 data points is a really small amount of information! This makes it even more impressive.

The fact that this approach works so well, with only 2 dimensions, tells us that our encoding is impressively effective.

8.4 Linear Encoders and Decoders

Even simpler than our one-layer example above, is the **linear** autoencoder.

It turns out that, despite its simplicity, even a linear autoencoder can be useful! In some contexts, it's even **better**:

- A linear autoencoder often has a **closed-form** solution: we don't have to do gradient descent.
- The linear autoencoder tends to create a very simple kind of **interpretability**.

This closed form solution is equivalent to **principle component analysis** (PCA), which you might be familiar with from linear algebra.

- We obtain this solution using a technique called **singular value decomposition** (SVD).
-

We can draw some parallels to a paradigm we've seen before:

- **PCA** can be thought of as the simplified, linear version of the **autoencoder** problem.
- This is similar to how **linear regression** is the simple, linear version of a **neural network**.

8.4.1 Principle Component Analysis (Optional)

Remark (Optional)

The following section briefly covers the concepts of PCA.

These may provide some intuition for what autoencoders do, in a simple, linear environment.

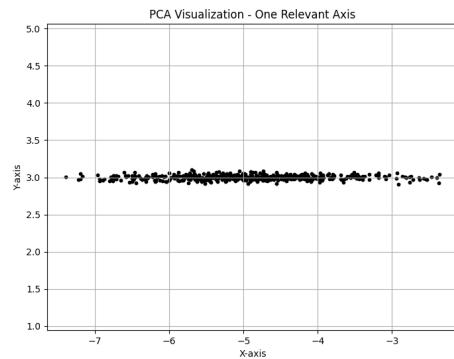
We mentioned that autoencoders need to make their data points clearly **distinct** from one another.

- In other words, it's best to focus on ways that they're **different** from each other.

The easiest way to do that is to focus on sources of **variance** in the data.

8.4.2 Low-variance: less important (Optional)

Let's give a motivating example:



Almost all data is encoded on the x-axis.

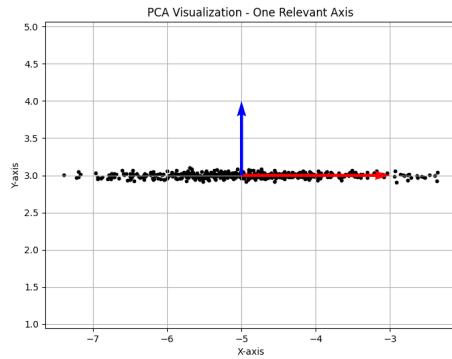
If we look at this dataset, there's a clear **difference** between our two axes: one is very high-variance (x), one is low (y).

If I told you "the y coordinate of this data point is roughly 3", that tells you essentially **nothing**: that's true of all of our data.

Meanwhile, the x coordinate is much more **informative**.

- It seems that **high-variance** dimensions tend to contain **more information** than low-variance dimensions.

So, if we break our data up based on coordinates:



We could remove the y-axis while preserving most of our information! This would be a good target for **omitting** from the latent space.

$$\begin{bmatrix} x \\ y \end{bmatrix} \longrightarrow \begin{bmatrix} x \end{bmatrix} \quad (8.6)$$

Concept 367

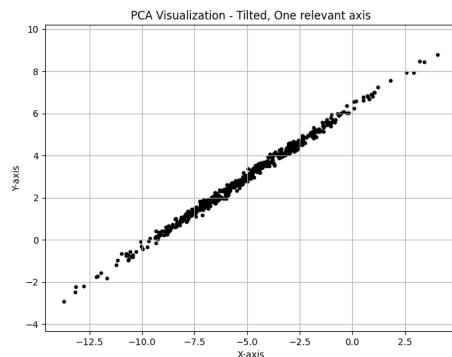
When doing PCA, we tend to focus on axes of **high variance**.

Axes with low variance carry **less information**.

So, if we want to **compress** our data, we can remove those low-variance dimensions.

8.4.3 Different axes (Optional)

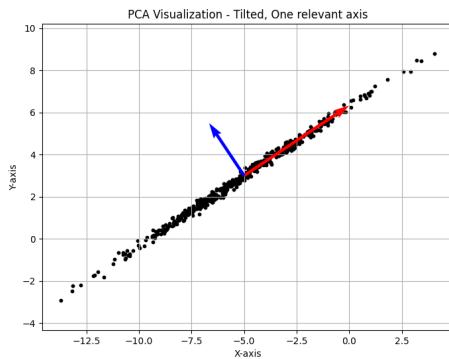
Our data doesn't always (or even usually) line up perfectly with our axes, though.



Almost all data is on a single line.

In this case, it looks very **similar** to our x-axis data. But the problem is, we can't just reduce it to one of our two axes.

The solution? **Change coordinate systems.**



Now, we have a high variance axis, and a low variance axis.

Now, we have the same situation as before! Almost all of our data is on **one axis**: we can omit the other.

Concept 368

Often, the most "**information-heavy**" directions in our data, aren't our default axes.

So, we look for a **new coordinate system**, where most of our variance is contained ("can be explained") by only a few dimensions.

- That way, we can **remove** the other remaining dimensions, which contribute very little to our understanding of the data.
- With fewer dimensions, our data is often more **interpretable**.

This idea, of finding high-variance components, and discarding low-variance components, is the core principle of **principle component analysis**.

- This is equivalent to how our linear autoencoders remove extra dimensions.

Sometimes, it can be a bit difficult to **interpret** these high-variance components: what does it mean to have high variance along a "different axis", conceptually?

But other times, we find that two "separate" variables, are both giving us the **same information**: they would correlate strongly, and thus, would give us the line we see above!

- This is why we can **remove** one dimension: we're using two axes to represent basically the same thing.

Example: If something is sold for a fixed price, then "number of units sold" and "profits from sales" are telling us **the same thing**.

So, we can compress into one dimension with low information loss.

Last Updated: 11/08/23 21:00:09

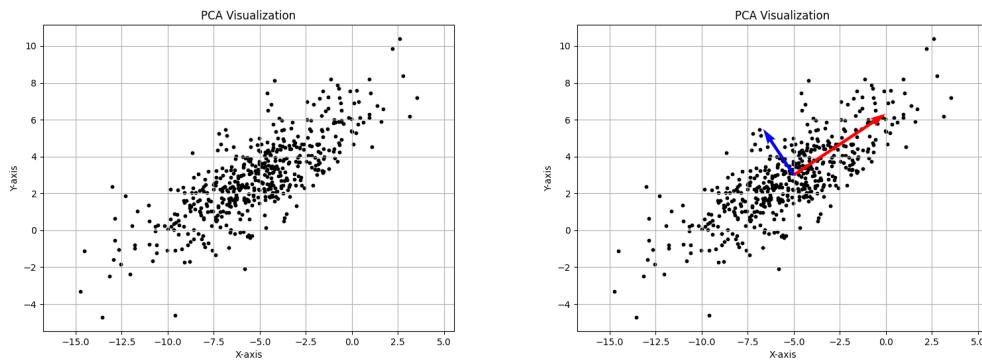
Many examples aren't perfectly correlated like this, but are still pretty similar: for example, sales and advertising spending at a company.

8.4.4 General example (Optional)

How many dimensions we need to capture most of our variance **depends** on the situation.

The goal is often, for visualization, to reduce it to **2 or 3** primary components.

But even if we can't remove axes, PCA can give us a useful way to focus in on the **relationship** between our variables:



Here, we have identified which axis matters "more" to our data, and how much.

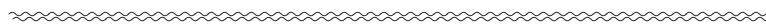
8.4.5 Non-linear encoders (Optional)

Of course, things aren't always so simple: we often won't have this nice blob, which we can measure across a few perpendicular axes.

Our real data might take on a different form, that has a coherent "shape", or "structure", but not one that fits our **linear** model.

This is where our **non-linear** autoencoders come in. They follow the same kind of principles as PCA:

- We want to distinguish patterns and curves that contain **variance**, and allow us to **distinguish** data points.
- Those which contain more information, we **keep**. The rest, we discard.
- We can then **visualize** those remaining dimensions, or use them for data analysis.



One more detail of PCA we've ignored so far: all of variance comes from some "baseline": an **average** position, which all of our data points are **shifted** from.

- We could view this as a "**template**", which our PCA components point away from, for any given data point.

In PCA, we didn't focus on this as much, because it was just the average of all of our data points: useful, but simple.

This concept becomes interesting in the more complex case of non-linear autoencoders: it isn't so easy to **guess** what's the "average"/typical example in the latent space.

- **Example:** Suppose we plot all of the data points labeled "7", based on their latent representations.
- We could see what happens if we average those, and then convert it back into a picture: it would teach us about how our model sees the number 7 in general.

The result depends on your exact choice of encoder, but it's not always what you expect: that's why it's so **informative!**

8.5 Advanced Encoders and Decoders (Optional)

We can build on this framework to create models for different, practical tasks.

8.5.1 Generative Networks (Optional)

One useful application of our latent space is to **artificially** create more data which is **similar** to the data we already have.

How?

We saw that, with **some** very effective autoencoders, we find some useful **structure** in our latent space:

- Data points which were **nearby** in latent space, appeared **similar** in the input space.

If we successfully train an autoencoder with this property, then creating new data is possible:

- We take our real data, and slightly modify it in the latent space. This should be similar to our real data, but not exactly the same.

How do we train our autoencoder to do this? We'll discuss one popular approach below: the "variational autoencoder" (VAE).

This is the basis of a **generative network**:

Definition 369

Generative networks are networks which are used to **generate** artificial data, which is similar to **real** data used to train it.

These often come in the form of an autoencoder:

- We start by using real data to **train** the autoencoder, and create a **latent space**.
- We use those same data, and **create** new data that is "close" to the real data, within our latent space.
- Finally, we use the decoder to **reconstruct** our **new data**.

~~~~~  
 This technique relies on the assumption that data points which are **close** in the latent space, are **similar** in the input space.

These models have many applications:

- Augmenting (increasing the size of) **smaller** datasets
- Reducing **overfitting**, by perturbing data
- Art and Media

- **Example:** Superresolution: filling in "details" in images where they need to be believable, not necessarily real

And more, which we'll discuss below.

This, of course, can cause problems if you don't know you're working with fictional data!

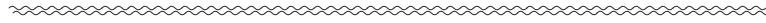
### 8.5.1.1 Variational Autoencoders (Optional)

We've introduced the general outline of a **generative network**: we create artificial data, using our latent space.

But we have **two problems**:

- First: how do we **generate new** data points, "close" to the real data? What's our procedure?
- Second: this relies on the assumption that our **encoding** is **smooth**: nearby points in latent space represent similar information. How do we guarantee that?

We'll find that one technique addresses both of these problems: representing our data as a **probability distribution**.



One simple way to generate new data, is to **randomly** perturb a data point in the latent space, by a small amount.

The outcomes of random processes, like this, have a certain **probability** of occurring.

- So, the output of our encoder isn't a point in the latent space, but a **probability distribution** of possible points.

#### Definition 370

A **variational autoencoder**, rather than creating a **single** encoding for a data point, encodes a **probability distribution**.

- This distribution represents different possible encodings, for the **same** input.

Then, we **sample** from this distribution, to randomly select an encoding.

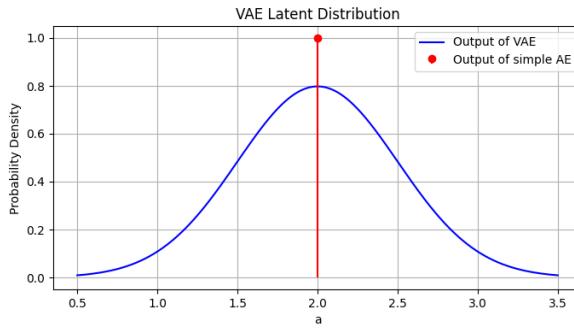
Finally, we **decode** this to get our "modified" data point.



This process actually **regularizes** our model: it needs to be able to re-produce the input, even if it's slightly modified in the latent space.

- We're also restricting our probability distribution to have a certain shape/structure (like a normal distribution).

**Example:** Suppose that we created a 1-D encoding, and for this particular data point, we encode it as  $a = 2$ . If we compare the simple AE to the VAE:

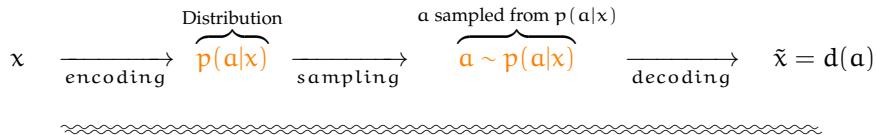


We've gone from "always output 2" to "output a normal distribution centered on 2".

We can directly compare the process in both cases. First, the **simple autoencoder**:

$$x \xrightarrow{\text{encoding}} a = e(x) \xrightarrow{\text{decoding}} \tilde{x} = d(a) \quad (8.7)$$

Next, the **variational autoencoder**:



Now, we can handle the second problem: how do we ensure that nearby points in the latent space are **similar**?

Well, if we want our model to do something, we **train** it with that goal in mind.

- As we've designed it, our VAE **already** generates points "nearby" to our encoding, using its probability distribution.
- Because they're nearby, we want the **original** data and the **sampled** data to be relatively similar.

Technical comment, that isn't important to this class:

The simple AE distribution depicted is the **dirac delta function**, where all the probability is 'stored' at  $a = 2$ .

### Concept 371

Unlike a simple AE, our VAE **doesn't** return exactly the same data that it started with:

- Instead, our VAE **samples** the latent space, **nearby** to the "pure" encoding.

We usually use something like the normal distribution above: most data is close to the mean.

Because the sampled and original data are nearby, we want them to be **similar**.

How do we compare the original and sampled data? The original data is given as the input. Meanwhile, we convert the sampled data by **decoding** (reconstructing) it.

We want our original (input) data  $x$  and sampled (+decoded) data  $\tilde{x}$  to be **similar**.

- This is similar to our goal for the simple autoencoder: there, too, we wanted our original and re-constructed data to match.

### Concept 372

Just like with a simple autoencoder, we want the **input** and **output** of our VAE to be close to **equal**.

$$x \approx \tilde{x}$$

This serves a few functions:

- Same goal as we had before: being able to **reconstruct** the input shows that our encoding preserves meaningful **information**.
- The simple encoding of  $x$ , and the encoding that produces  $\tilde{x}$ , are **not** the same, but they are **similar**.
  - This helps us train our model to "**smooth out**" its surface: we teach it that nearby points should be similar.

So, we **train** our model with this goal in mind.

---

To help "smooth out" the VAE representation further, we often add a **regularizer** term to the loss.

- We won't go into detail on this feature here.

## 8.5.2 Adversarial Optimization (Optional)

We'll make a brief detour, to discuss a different way we can generate **artificial** data.

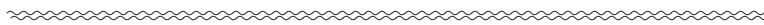
In order to "test" your network, and see how **robust** it is, you might want to deliberately find examples it **struggles** with, but *shouldn't*.

- We could do this by just running a large amount of data, and manually looking for examples that fail, despite seeming "obvious" to humans.
- But this is labor-intensive, and **expensive**.

Instead, we introduce a different way to create so-called "**adversaries**": examples specifically designed to "trick" our model.

We want a data point that:

- **Should** be classified correctly, and would be classified correctly by humans
- But the machine fails, despite looking **similar** to valid data points.



We'll start with a valid, **correctly** labelled data point: this handles the first condition: "should have been labelled correctly".

But we want to **modify** it so that it's labelled incorrectly.

- If our model isn't **robust** enough, we might be able to **confuse** it, without changing the data very much at all.

How do we modify it most efficiently? Using the **gradient**.

Previously, we used the gradient to compute, "what's the most efficient way to change my **model**, to **decrease** my loss?"

$$W - \eta \frac{\partial L}{\partial W} \quad (8.8)$$

This time, we want to ask, "what's the most efficient way to change my **data point**, to **increase** my loss?"

So, we want the gradient between the loss and the data point.

$$x + \eta \frac{\partial L}{\partial x} \quad (8.9)$$

We take **steps** in this process, repeatedly changing our gradient to match the model.

We continue this process until our data point is successfully classified **incorrectly**.

- Often, we can accomplish this without significantly modifying our data point.

This time, instead of traveling across the parameter space, we're traveling across the input space!

**Definition 373**

**Adversarial training** is a way to "trick" a model into making inaccurate predictions:

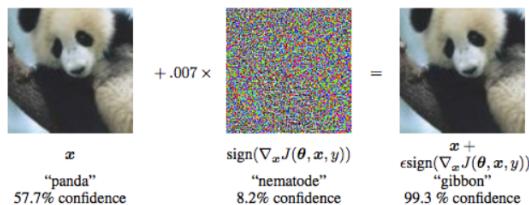
This is done by **training** data points that appear very similar to real data, but exploit weaknesses of the model.

To accomplish this, we take a real, correctly labelled data point, and slowly apply gradient descent **to the data point**, gradually increasing the loss:

$$x_{\text{new}} = x_{\text{old}} + \eta \frac{\partial L}{\partial x}$$

With some training, we can design a new data point that our model evaluates incorrectly, despite being very similar to the original data point.

**Example:** Here's a *real example* of applying this to image classification:



This panda, despite no visible change in appearance, is now a gibbon. (Source: Ian Goodfellow et al., 2014)

This kind of data is actually incredibly valuable:

- If we use it to train our model further (i.e., teaching it that it's wrong about these data points), it tends to become **more robust** against this kind of attack.

But, we need to be careful in this process:

**Clarification 374**

When creating adversarial data, we have to set a **termination** condition:

If we continue gradient descent too long, our data point will change **classification**, but it can also actually change enough that it **should** be identified differently.

**Example:** If you do gradient descent to turn a picture of a "9" into a "1", and it looks like a "1" to a human, it should be reasonable that the model makes the same decision.

We want to find the point where the data is different enough to be deceptive, but not different enough that it's obviously, genuinely a new data point.

Adversarial data is widely useful:

- Improving model robustness
- Defending against security threats
- Learning more about the nature of your model

### 8.5.3 Generative Adversarial Networks (**Optional**)

We've developed two useful ideas:

- Generative networks: used to generate artificial, plausible data
- Adversarial data: data designed to exploit weaknesses in a model

We can combine these ideas to create a powerful tool, called, appropriately, a **Generative Adversarial Network**.

Here's the general idea:

- **Generators**: we want to generate artificial data, that closely reflects the **original** distribution.
- We found that an **adversary** could teach us (and in turn, our models), their weaknesses, by actively seeking them out.
  - So, we'll create an **adversary** for the generator: the **discriminator**. This punishes our model for creating data that is "detectably different" from real data.

The generator will create "adversarial data": this data is designed to **trick** the discriminator into thinking it's real.

The discriminator will try to learn how to tell the two apart, and provide feedback to the generator, telling us how it made a mistake the previous time.

This feedback loop gradually improves how "realistic" our data generated is, through a form of **unsupervised** learning.

The two models are "supervising" each other!

**Definition 375**

A **Generative Adversarial Network** is a model that comes into two opposing, or "adversarial", parts:

- The **generator** is trained to create "fake" data, that looks as plausible and realistic as possible.
- The **discriminator** is trained to detect fake data

When one of these models fails, they teach the other model what they did wrong, to improve.

- This process repeats until the discriminator accuracy reaches 50%: if it's equally likely to mix up real/fake data, our generator has become indistinguishable from real data.

This is a sort of "arms race" between the two halves of our model.

**Clarification 376**

Why is best performance for the generator at **50%**?

If it was 100%, then the model always assumes the generator is real, and the real data is fake.

- But that's not true: the real data is *also* real. The discriminator isn't doing its job correctly anymore.

Another perspective: if you knew your discriminator was always **wrong**, then you could easily create a discriminator that was always right: just do the opposite of what you got before.

In short: 50% is the best you can do, because your discriminator is completely **unsure** of its answer: which is what it really means to be "**indistinguishable**".

GANs have been highly successful: this procedure not only improves robustness, but often creates a generally improved model.

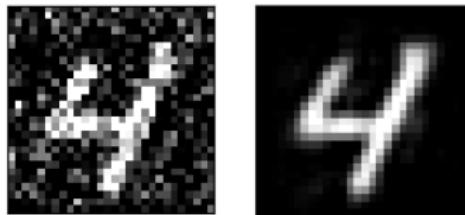
They're useful for creating very effective generators of new data, and have been particularly useful in the past for image generation.

GANs are still relevant, but if you want a more **modern** approach, you could look into **Diffusion** models, like Stable Diffusion.

### 8.5.4 De-noising (Optional)

One useful application of autoencoders is **de-noising**.

- We want to turn a noisy input into a less-noisy input.



The left is our input, right our desired, cleaner output.

The process for creating this kind of autoencoder is straightforward enough: we give noisy data and encourage the model to match the original, noiseless version.

#### Concept 377

Autoencoders can be used for **de-noising**:

By training with

- Noisy data as input
- Noiseless data as desired output

You can teach the model to create a latent representation that's resistant to this kind of noise.

### 8.5.5 Attention (Optional)

Transformer networks are a very modern, very powerful approach to many problems, most famously **language processing**.

In order to create detailed context within a sentence, these models use a technique called "**self-attention**", which has proven to be incredibly powerful for working with language.

As of writing, GPT-4 is the most famous example.

- Attention is an in-depth topic that deserves its own section; we'll **skip** the math.
- The main idea: attention helps determine how words create **context** for each other.

#### Concept 378

**Attention**, at a very high level, is based on the idea that:

- Different **words** in a sentence have different levels of **importance** to each other.

The words that are more **important** to a particular word X, are a bigger part of the **context** you use to understand that word.

With that in mind, you can apply the **context** from each other word Y, to better understand the meaning of word X.

Transformers aren't **limited** to language, but we'll focus on that use case, for ease of explanation.

**Example:** "The **blue dog** bites the **red ball**": the word '**red**' is describing the '**ball**', so it is less important for understanding '**dog**' in this sentence.

Based on that...

#### Definition 379

**Attention** is a mechanism for determining how, for a **pair of words**, one word X might be **important** to understanding the other word Y, contextually.

In other words, X might require your **attention**, if you want to understand Y.

- Attention is applied to **every** pair of words in a sentence: we measure every word's impact on every other word.
- Finally, for each word, we **integrate** information from the other words, based on how "important" they were.

~~~~~  
Attention has the benefit of allowing us to analyze our entire prompt simultaneously, speeding up the process.

Self-attention is used by a single sequence of text, by itself: it "learns about" itself.

Clarification 380

A few lingering comments:

- Word X and Y are typically asymmetric: X may not affect the meaning of Y the same way as Y affects X.
- This meaning is **contextual**, not the "importance" of each word in isolation.
- We "integrate information" by taking a linear combination of each word: a larger weight is given to words which are more "important" to word Y.

"Multi-headed attention" simply refers to having several of these attention mechanisms, applied to the same input in parallel.

So, each "attention head" receives the same data, like neurons in the same layer of a network.

8.5.6 Transformer Networks (Optional)

Now that we loosely understand attention, we can think about the bigger picture.

The goal of a transformer is to **predict text**.

At a very high level, transformers have a structure *similar* to **autoencoders**. They're broken into the same sort of two parts, though their internal structure is a bit **different**:

- **Encoder**: this is actually a **stack** of several encoders, one after another. This (presumably) creates an **internal representation** of the "meaning" of the input text, similar to a latent representation.

– Each encoder contains a fully connected network, as we've shown above, but also a "self-attention mechanism".

You could say that this converts the input "prompt" into something similar to the "latent representation", which retains our key information.

- **Decoder**: a stack of decoders, one after another. Based on the **internal representation**, it creates predicted text.

– Each decoder also contains an FC network+attention mechanisms.

This would create the "response" to your "prompt".

– Once our decoder predicts some text, that's part of the "**past text**": this gets included alongside the internal representation, for future predictions.

Concept 381

A transformer network follows a structure with some similarities to auto-encoders: using an **internal representation**.

- Converting its **input** (prior text) into an **internal representation**, similar to a latent representation.
- Converting that **representation** into an **output** (predicted text)

However, there are key differences:

- As the output creates new text, that is **included** with the internal representation.
- Our "predicted text" will **not** be the **same** as our past text.

Why multiple encoders?

Concept 382

The **multiple** encoders are used to gradually find "more **complex**" patterns (or concepts), as we move through more layers.

-
- The idea is that, the first encoder combines words to finds **basic**, simple language structures.
 - The second encoder has access to these simple structures, and can **combine** them together to create a more complex structure.
 - Each encoder is combining the results of the **previous** layer, to create something more complex.

This is similar to the structure and motivation behind **convolutional neural networks**, which we will cover later.

Definition 383

A **transformer network** is a model made of encoders and decoders that use **attention** to determine the **structure** and **context** of the input "prompt", and create an output "response".

The structure of a transformer network is:

- Several "layers" of **encoders**, that take the input and interpret it, creating the internal representation
- An **internal representation** of the input, and the previous output of the decoder
- Several "layers" of **decoders** that turn this representation+output, into more **predicted text**

Predicted text is given as a **probability distribution**, using softmax.

- We select an element from this probability distribution before moving on to the next word.

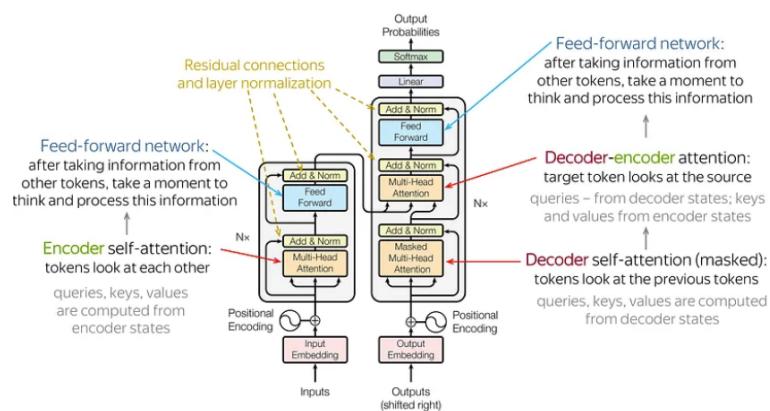
We've skipped over many important details of transformers, but this gives us the gist.

In particular, we've skipped some components, like normalization layers.

Clarification 384

A few key differences between an autoencoder and transformer networks:

- The input and output are the same in an autoencoder.
- Transformers use attention mechanisms.
- The internal representation in an autoencoder ("latent representation") is **smaller** than the input or output.
- They serve different purposes.



If you want a more in-depth explanation, go to [this very helpful resource](#), and the source of this lovely diagram!

8.6 Terms

- Unsupervised Learning (Review)
- Clustering (Review)
- Compression
- Decompression/Re-construction
- Encoder
- Decoder
- Autoencoder
- Latent Representation
- Latent Space
- Bottleneck
- Dimensionality (Review)
- Overcomplete Autoencoder
- Downstream Task (Review)
- Semi-supervised Learning
- Principle Component Analysis
- Singular Value Decomposition
- Generative Networks
- Variational Autoencoders
- Transformer Networks

Optional

- Adversarial Data
- Generative Adversarial Networks
- De-noising
- Attention
- Transformer Networks
- Internal Representation

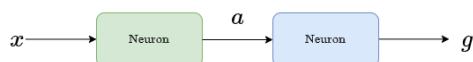
CHAPTER 9

Convolutional Neural Networks

9.0.1 Fully Connected Networks

Up to this point, we've focus on "**fully connected**" neural networks.

- "Connected" refers to the "connection" between neurons in **adjacent** layers: one neuron provides the input for another.



These two neurons are connected.

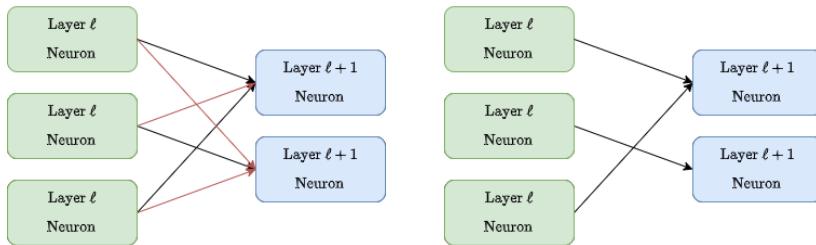
Thus, "fully connected" means that every possible connection between pairs of neurons **exists**.

Definition 385

A **fully connected** (FC) layer is one where every **input** neuron is connected to every **output** neuron.

The network layer only needs to be missing **one** connection between neurons to not be fully connected.

Example: We compare two networks:



The left network is fully connected: the right is not, having removed the red arrows.

9.0.2 The drawbacks of fully-connected networks

The "fully connected" approach includes a **weight** $w_{a,b}$ for every pair of input neuron a , and output neuron b .

- Each of these weights determines the **relationship** between our two neurons.
- In an FC settings, we're allowing for every possible pattern between pairs.

This is a very, very flexible model: any combination of patterns is possible.

Concept 386

Fully-connected networks are very useful when we **know very little** about how to **predict** our result.

By including so many possible connections and patterns, we're open to lots of **different models** we could try.

- This is especially helpful if we expect these relationships to be complex.

With a non-FC model, on the other hand, some connections have been severed. With this model, we're creating making some **assumptions** about which patterns **don't** exist.

- **Example:** If you think fact A is irrelevant for computing fact B, you wouldn't include it in the equation.
- This is similar to how you want to exclude inputs that won't help you predict your output.

Concept 387

Removing a connection in a neural network is equivalent to saying, "I don't think this variable a **should** affect this other variable b ".

This highlights a major **drawback** of fully connected networks: sometimes, it's inappropriate to allow for every possible connection.

Having connections we don't need can cause plenty of problems:

Concept 388

Fully-connected networks come with some problems:

- Having many parameters can risk **overfitting**,
- Our model takes **more time** to converge
 - Both because it has to train **more weights**,
 - And because our model can get "distracted" by dead-end possibilities, that a simpler model wouldn't consider.
- It's often difficult to interpret **how** our neural network comes to the conclusions it does.

In this chapter, we'll introduce some more specific problems, and one model type that allows us to overcome these problems: **Convolutional Neural Networks**.

9.0.3 Intro to Image Processing

An excellent example for how FC neural networks can fail, is **image processing**.

Example: Facial recognition, self-driving vehicles, classifying the object in a picture

Let's give it a try: suppose we have an image. For simplicity, it's black and white. We'll need to represent the **brightness** of each pixel with a number.

- We'll use the most common range of values: the integers [0, 255].

$$\begin{bmatrix} 255 & 0 & 255 \\ 0 & 0 & 0 \\ 255 & 0 & 255 \end{bmatrix}$$

Our machine stores the "picture" on the right.

Our neural network takes a single $d \times 1$ vector as its input. But right now, we have an $(r \times k)$ matrix. How do we solve that? With **flattening**.

Definition 389

Flattening is the process of taking a **matrix** of inputs, and transforming it into a single **vector**.

We usually do this by **concatenating** (combining consecutively) each row/column, in order.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \rightarrow \begin{bmatrix} a \\ c \\ b \\ d \end{bmatrix}$$

If the input is an $(r \times k)$ matrix, the result is an $(rk \times 1)$ vector.

Example: We can apply this to our data above.

We'll represent it with the transpose to save space.

$$\begin{bmatrix} 255 & 0 & 255 \\ 0 & 0 & 0 \\ 255 & 0 & 255 \end{bmatrix} \rightarrow \begin{bmatrix} 255 & 0 & 255 & 0 & 0 & 0 & 255 & 0 & 255 \end{bmatrix}^T \quad (9.1)$$

And so transform from (3×3) to (9×1) .

Looking at the image, or even the **matrix**, it's relatively easy to see the "**cross**" pattern.

But, when we **flatten** it, those patterns immediately stop being obvious.

- We've lost information above which pixels are "**beside**" one another, for example.

Based on this, we'll find **two** main problems, that our CNNs will hopefully solve:

Remember that we only took the row vector for visualization: they're stacked vertically based on column!

9.0.4 Spatial Locality

First: as we just mentioned, we need an idea of which pixels are close **horizontally**.

- In fact, our network doesn't even care which pixels are **above** each other **vertically**:

Concept 390

Review from the Feature Representation chapter

The **order** we choose for elements in a vector **doesn't** affect the behavior of our model, so long as we **consistently** use that order.

This is because a linear model is a **sum**:

$$w^T a = \sum_i w_i a_i$$

And sums are the same, regardless of **order**.

$$a + b = b + a \implies w_1 a_1 + w_2 a_2 = w_2 a_2 + w_1 a_1$$

- We emphasize that the order does need to be **consistent** between data points.

Example: Suppose x_1 represents height and x_2 represents weight.

- We could do either [weight, height] or [height, weight]: it doesn't matter.

In other words, we could **shuffle** the order of our pixels, and as long as we shuffled it the **same way** for all of our training data, it wouldn't matter to our **model**.

BUT, we have to be **consistent** with which order we pick: otherwise, someone is measured as 180 feet tall, instead of 180 pounds.

- This was fine for the above example, but it doesn't make sense for **image processing**, where each dimension is just a **pixel**:

Human vision works differently: we look for shapes, which are often made up of pixels **near** each other: edges, points, corners, curves.

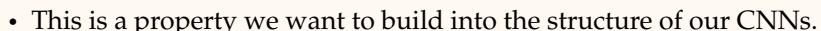
- In other words, we want to encode **local** information, across the physical **space** of our image. Thus, we call it **spatial locality**.

Definition 391

Spatial locality is the knowledge of which objects are **close** in **space** to each other.

In an **image**, we might think of which pixels are "close" in that image.

- Which pixels are next to, or on top of each other? How far apart? How are they "arranged"?

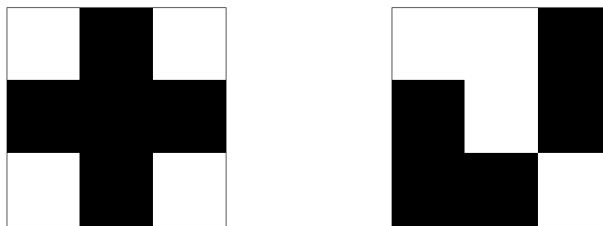


- This is a property we want to build into the structure of our CNNs.

When we use "space" here, it's **different** from "latent space", or "input space".

Instead, we're talking about physical space: how physically **close** real things are, measured in **meters**, or in this case, **pixels**.

Example: If told that the white was blank space, and the black represented "objects", a human would have a concrete understanding of how these two images might be different:



The left image contains **one** object, while the right image contains **two** objects.

We figure this out based on which pixels are **toucning** or not: a spatial property.

- The neural network would struggle to encode anything like that.

9.0.5 Translation Invariance

A second problem is that, if the same **pattern** occupies different pixels, then it's completely new to the model.

- Example:** Suppose you have a cat on the left side of an image. You **move** it to the right side of the image.
- A person would consider that image "**almost the same**".
- But our FC NN does not: the cat is occupying a completely **different set of pixels**, which have a completely separate set of weights attached.

So, our NN can't find structures that are **similar** across different parts of the input.

Instead, we want a different behavior: we want our model to treat our input as the **same** (invariant), even if we move, or **translate** it.

- Thus, we're looking for **translation invariance**.

Not language translation: "translation" as in "moving around in space".

Definition 392

Translation invariance is the property of treating patterns as the **same** even if we **translate** them in space.

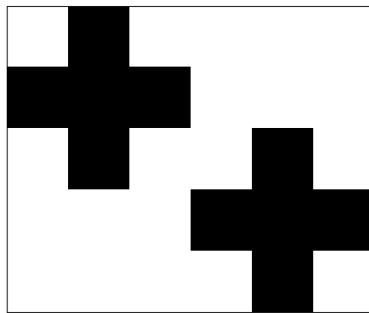
In an **image**, we might want to recognize the same **pattern** in two different **positions** on the image.

- In other words, the pattern has "translated" from one of those positions, to the other.



- This is a property we want to build into our CNN.

Example: In the following image, you would probably recognize "two crosses".



We have two of the **same** object: just **translated** over.

But, because the top left pixels have separate weights from the bottom right pixels, the NN will react differently to each.

Now that we've defined our problem, we can come up with a solution: **filters**.

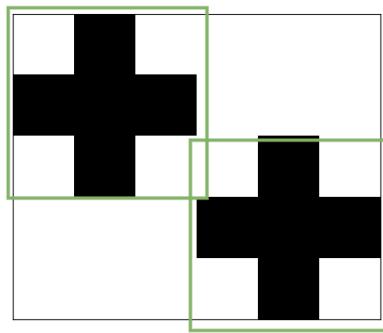
9.1 Filters

9.1.1 Motivating the Filter

So, we want a technique that handles both of these problems.

First, **translation invariance**: we want a calculation that can find the same pattern, in multiple locations.

- So, we'll apply the same calculation repeatedly, in **multiple positions** on our image.
- We'll **move** across our image, shifting to a new position each time we **scan** for that pattern.



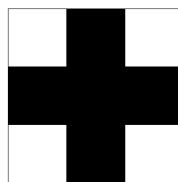
As we "scan over" our image, we'll hopefully find both of our crosses separately.

Next, **spatial locality**: we want this calculation to encode **spatial** information.

- As we "scan" across our image, each computation will look for a particular "shape", or "**pattern**" for our pixels.
- This pattern will be based on the **relative location** of each pixel.

So, we're looking for a tool that repeatedly shifts (or **translates**) across our image, and looks for a spatial **pattern** in the image.

- **Example:** Above, we would be looking for the "3x3 cross" pattern, and shift across rows/columns.



This is the shape we are looking for at each position.

The tool in question is "looking" for a pattern. Another way to see it, is that it's **filtering** out everything that doesn't match that pattern.

- Thus, we call it a **filter**.

Concept 393

Filters handle both the problems of **spatial locality** and **translation invariance** at the same time.

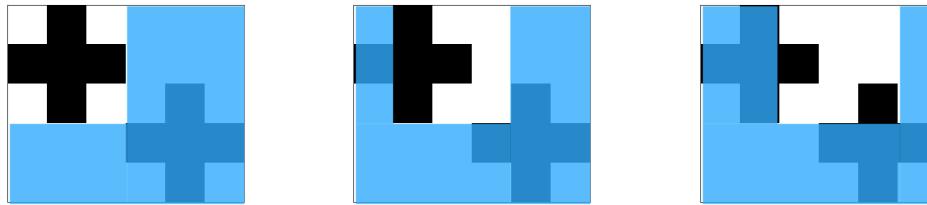
Notably, all of this works better if we keep our data in the **matrix** format, not the **flattened** one.

9.1.2 Windowing

We still need to figure out **how** we're going to find these patterns.

We've already established that our algorithm will look at a **local** region of the image, and search for the pattern.

To make life easier, we'll cut out a **piece** of the image, and only compare that to the pattern.



If we're viewing the top-left corner, we're ignoring everything else (blue-shaded). Then, we'll check the next position.

This region is the only part we "see", so we call it a **window**.

Definition 394

The **window** v is the region of our image that we are, at a given moment, looking for our **pattern** in.

This is the region we are applying our **filter** to.

The window has the **same dimensions** as our filter, so we can compare them directly.

-
- As we continue filtering, we'll repeatedly move our filter, shifting it to every valid position on the image.

9.1.3 1-D case

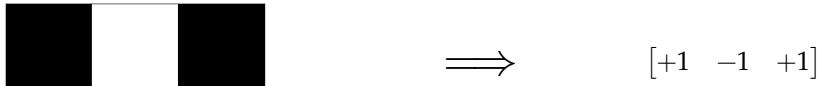
To get going, we'll start with a 1D example.

- To make the math easier, we'll replace 0 and 255 with +1 and -1.

Suppose we're looking for "bright spots": pixels that are much brighter than their surroundings.

This isn't just a simplification: when processing sound data, it'll be in a 1D form.

We've decided to make dark pixels +1, and bright pixels -1. Which convention we choose isn't important: it's just more easily visible.



So, we're looking for something like this.

How do we find "bright spots", like this? Well, we want to find regions which are **similar** to our pattern.

- Our sequence is a vector, so we want to **get the similarity between two vectors**.
- We have a tool for this! The **dot product** $a \cdot b$.

Concept 395

Review from the Classification chapter

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 $\text{angle} < 90^\circ$ they are "similar": $\vec{a} \cdot \vec{b} > 0$
- If $\text{angle} > 90^\circ$ they are "different": $\vec{a} \cdot \vec{b} < 0$
- If they are **perpendicular** ($\text{angle}=90^\circ$) to each other, $\vec{a} \cdot \vec{b} = 0$

So: as an approximation, the higher the dot product, the more similar they are!

Now, we know what to do: we'll get the **dot product** between our window, and the **filter**, to see how similar they are.

- If they're similar enough, then we found the pattern!

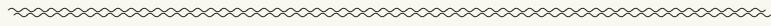
This works extra well for something like an image, where the pixels have a restricted "range" of values: it's not as easy to get an extra-high dot product just because the magnitude are too large.

Concept 396

To determine whether the window contains our pattern, we take the **dot product** between our **window v** and our **filter f**.

$$v \cdot f$$

The **higher** the dot product, the **more likely** that we have our pattern.



- There's no exactly dot product value to be "sure" you've found your pattern: you have to choose your threshold based on context.

We'll show our example below.

9.1.4 1D Example

So, suppose we have our input image:

$$\xrightarrow{[+1 \quad +1 \quad -1 \quad -1 \quad -1 \quad +1 \quad -1 \quad +1 \quad +1 \quad +1]}$$

Our filter is size 3 (3 elements), so we'll grab a window of 3 elements.

$$\xrightarrow{\overbrace{[+1 \quad -1 \quad +1]}^f \quad -1 \quad -1 \quad +1 \quad -1 \quad +1 \quad +1 \quad +1}$$

We ignore everything after the first three elements.

We then compute the result:

$$\overbrace{[+1 \quad +1 \quad -1]}^v \xrightarrow{f} \overbrace{\begin{bmatrix} +1 \\ -1 \\ +1 \end{bmatrix}}^f \cdot \overbrace{\begin{bmatrix} +1 \\ +1 \\ -1 \end{bmatrix}}^v = +1 - 1 - 1 = -1 \quad (9.2)$$

This is our first filtering; we get -1. This is the first element of our output:

$$y = x * f = [-1 \quad ? \quad ? \quad ? \quad ? \quad ? \quad ?] \quad (9.3)$$

We'll repeat for the rest of our 1d signal.



$$[-1 \quad +1 \quad \dots \quad ?] \xrightarrow{} [-1 \quad +1 \quad -1 \quad \dots \quad ?] \xrightarrow{} [-1 \quad +1 \quad -1 \quad +1 \quad \dots \quad ?]$$

This is **convolution**.

9.1.5 Convolution

Convolution simply applies our filter, at each position:

Concept 397

When filtering (doing **convolution**), the **output** at the i^{th} index is given by having **shifted** your window over from 0, $(i - 1)$ times.

- The **indices** for our output usually start from index 1.

Example: The last example above, ending at index 3, outputs +1 after shifting right 3 times.

The result is a new vector:

$$y = x * f = [-1 \quad +1 \quad -1 \quad +1 \quad -3 \quad \textcolor{orange}{+3} \quad -1 \quad +1] \quad (9.4)$$

The pixel we have labelled in orange corresponds to the "bright spot" in our sequence:



As we hoped, the "matching" pattern is the highest positive magnitude!

With this, we've fully demonstrated 1-d **convolution**.

Definition 398

Convolution $x * f$ is the process of searching through a **signal** x for a particular **pattern**, using a **filter** f .

- The filter **matches** the pattern we're looking for.

The convolution process follows the following steps:

- Taking a **window** v in the same shape as the filter, **isolating** a section of your signal
- Applying a **dot product-like** operation between your filter f and your window v .
- **Sliding** your window, and **repeating**, until every output is computed.



- The $(i + 1)^{\text{th}}$ index is given for the dot product computed by shifting over i times from index 1.

We call this a "**dot product-like operation**" to prepare us for **higher-dimensional** equiva-

lents.

We can even write this in formula terms. To represent **windowing**, we'll use python slices, with the same conventions.

Key Equation 399

If we have **signal** x , **filter** f of **size** k , we can create a **window** v_i by...

Starting at the **leftmost** pixel, and shifting right by i units:

$$v_{i+1} = x[i : i + k] = \begin{bmatrix} x_{i+1} \\ x_{i+2} \\ \vdots \\ x_{i+k} \end{bmatrix}$$

- Note the subscript v_{i+1} : we start from $i = 0$, and thus v_1 .

This is used to create our **convolution** $y = x * f$:

$$y_i = f \cdot v_i$$

You might see a different version of indexing in some situations: _____

The fact that x is 1-indexed, but python slicing is 0-indexed, is the reason why $x[i : i + k]$ starts at x_{i+1} .

Clarification 400

Above, we used i to give us the leftmost slice of our input.

Like in the official notes!

We did this because we assumed the **leftmost** pixel would be assigned $i = 0$.

- However, in some cases, the **middle** pixel is assigned $i = 0$: the pixels indices go equally positive or negative.

In which case, we would need to **replace** our slicing procedure above:

$$x[i : i + k] \rightarrow x[(i - \lfloor k/2 \rfloor) : (i + \lfloor k/2 \rfloor)]$$

- We use the floor operator $\lfloor x \rfloor$ so that we index correctly, by integers.

One more thing: we need to be careful when we say we're doing "Convolution".

Clarification 401

In other fields, convolution requires **reversing the order** of your filter, before you apply it to your input.

However, this is typically **not** the case in machine learning.

9.1.6 Convolution Output Size

Something you might notice is that our output is **smaller** than our input was.

How much shorter? 2 elements: in general, the output of a convolution is $k - 1$ elements **shorter** than the input.

Why is this? We can see why, by focusing on the **leftmost** element of our filter: we can only shift it until our vector ends.

Where k is, again, the size of our filter.

$$\begin{array}{ccccccccc} & & & & & & \begin{bmatrix} +1 & -1 & +1 \\ +1 & +1 & +1 \end{bmatrix} \\ +1 & +1 & -1 & -1 & -1 & +1 & -1 & \underbrace{\quad}_{\begin{bmatrix} +1 & +1 & +1 \end{bmatrix}} \end{array} \quad (9.5)$$

But, our leftmost element hasn't reached the end of the vector: if it did, then the rest of the vector would be **sticking out**, with nothing to multiply with:

$$\begin{array}{ccccccccc} & & & & & & \begin{bmatrix} +1 & -1 & +1 \\ +1 & ? & ? \end{bmatrix} \\ +1 & +1 & -1 & -1 & -1 & +1 & -1 & +1 & +1 \end{array} \quad (9.6)$$

When our leftmost position is as far right as we can go, there are $k - 1$ positions remaining: the rest of the filter is "in the way".

Concept 402

For a length- n input and a length- k filter, **1d convolution** creates an output of size:

$$n - (k - 1)$$

9.1.7 Padding

We don't necessarily want to be shrinking the size of our output. How do we solve this?

Well, our equation above gives us two options: increase the input size, or decrease the filter size.

- Decreasing filter size is **restrictive**: the smaller the filter, the smaller the pattern we can search for.
- So, we'll just increase the size of our input.

We'll increase input size with **padding**: adding extra elements to the ends of our vector.

- Typically, we pad with 0's, to have the most neutral effect possible on our output.

Definition 403

Padding is a technique for increasing the size of the output of convolution.

To pad an input, you add filler values (usually 0's) to the **edges** of the input vector.

This allows the filter shift further in both directions.



A padding of p adds p values to **both sides** of our input vector, transforming our n -sized input into a $n + 2p$ sized input. Thus, our output size is:

$$(n + 2p) - (k - 1)$$

Where k is our filter size.

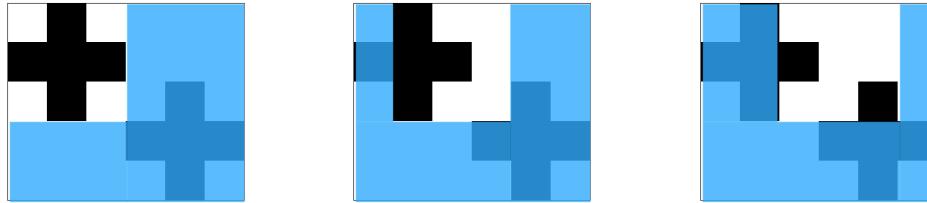
Example: Here's an example of **zero-padding** with $p = 2$:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 1 & 2 & 3 & 4 & 0 & 0 \end{bmatrix} \quad (9.7)$$

Often, we select our padding size so the output size is the same as the input size: $2p = k - 1$.

9.1.8 2D Filter

Now, we want to extrapolate this idea to higher dimensions: in particular, 2D, but this approach will work for any dimension.



We're back to this example.

Let's review what we can already guess: first, we now have a 2-D pattern we're looking for, and a 2-D window cut out of our image.

$$\begin{bmatrix} -1 & +1 & -1 \\ +1 & +1 & +1 \\ -1 & +1 & -1 \end{bmatrix}$$

This is our filter.

$$\begin{bmatrix} +1 & -1 & -1 \\ +1 & +1 & -1 \\ +1 & -1 & -1 \end{bmatrix}$$

This is our window $w_{0,1}$: we've shifted over by one column.

How do we measure their similarity?

$$\begin{bmatrix} -1 & +1 & -1 \\ +1 & +1 & +1 \\ -1 & +1 & -1 \end{bmatrix} \text{ vs } \begin{bmatrix} +1 & -1 & -1 \\ +1 & +1 & -1 \\ +1 & -1 & -1 \end{bmatrix} \quad (9.8)$$

We measured similarity between vectors using the **dot product**.

We can break our matrices up into vectors, by treating them as vectors of vectors.

$$\begin{bmatrix} [-1] \\ [+1] \\ [-1] \end{bmatrix} \quad \begin{bmatrix} [+1] \\ [+1] \\ [+1] \end{bmatrix} \quad \begin{bmatrix} [-1] \\ [+1] \\ [-1] \end{bmatrix} \quad \text{vs} \quad \begin{bmatrix} [+1] \\ [+1] \\ [+1] \end{bmatrix} \quad \begin{bmatrix} [-1] \\ [+1] \\ [-1] \end{bmatrix} \quad \begin{bmatrix} [-1] \\ [-1] \\ [-1] \end{bmatrix} \quad (9.9)$$

So, we can compare the similarity between the first vector in our **filter**, and the first vector in the **window** using the **dot product**.

Concept 404

If the j^{th} vector that makes up **matrix A** is similar to the j^{th} vector that makes up **matrix B**, then A and B are **similar**.

$$\vec{d} \approx \vec{d}$$

$$\vec{b} \approx \vec{e} \implies [\vec{a} \quad \vec{b} \quad \vec{c}] \approx [\vec{d} \quad \vec{e} \quad \vec{f}]$$

$$\vec{c} \approx \vec{f}$$

- We'll repeat this process for each column.

$$\underbrace{\begin{bmatrix} [-1] \\ [+1] \\ [-1] \end{bmatrix} \cdot \begin{bmatrix} [+1] \\ [+1] \\ [+1] \end{bmatrix}}_{\text{Col 1}} + \underbrace{\begin{bmatrix} [+1] \\ [+1] \\ [+1] \end{bmatrix} \cdot \begin{bmatrix} [-1] \\ [+1] \\ [-1] \end{bmatrix}}_{\text{Col 2}} + \underbrace{\begin{bmatrix} [-1] \\ [+1] \\ [-1] \end{bmatrix} \cdot \begin{bmatrix} [-1] \\ [-1] \\ [-1] \end{bmatrix}}_{\text{Col 3}} \quad (9.10)$$

So, we've matched the j^{th} column of our window with the j^{th} column of our filter.

- And a dot product matches the i^{th} row of that window vector, with the i^{th} row of the filter vector.

That means, we're multiplying **element-wise** across our matrix: the (i, j) element of f is multiplied by the (i, j) element of w .

Definition 405

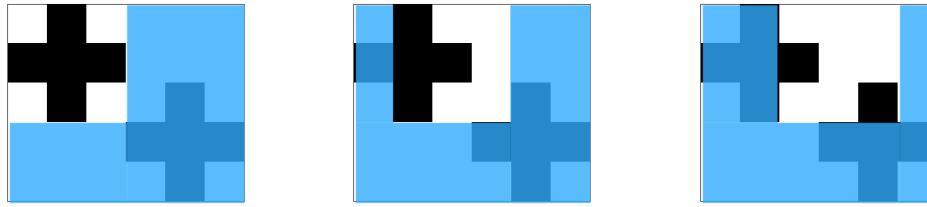
We introduce a **dot product generalization**, for a **matrix** (2-tensor):

- We compute it by multiplying **element-wise**, then **summing** all the elements.

$$A \cdot B = \sum_i \sum_j A_{ij} B_{ij}$$

This operation measures **similarity** between our two vectors.

This allows us to do higher-dimensional convolution.



We're back to this example.

9.1.9 2-D convolution

We'll need to shift around the image, in the same way that we did for the 1-d case.

- Before, we shifted over our **window** (size s) across our **input** (size k) by every possible position: this created $n - (k - 1)$ outputs.

The process is the same here: we just need to shift along two axes.

- We'll need to consider every combination of shifting i rows down, and j rows right.
- In python, this is equivalent to a double for-loop: "for i in m, for j in n".

Concept 406

For **2-D convolution**, we need to **shift** our window along two axes.

- So, we have one window for each **combination** of shifting i rows down, j columns right.

If we have an **input** with an axis of length **n**, and a **filter** of size **k**, that output axis has **length**

$$(n + 2p) - (k - 1)$$

- k is typically the same on both of the 2d axes: it's usually **square**.

Remark (Optional)

Convolution was originally designed based on the way human eyes work: we use it to look for edges, and other distinct features in our vision.

9.1.10 Dot Product Generalization

Later, we'll need to generalize this to higher dimensions: we'll review the higher-dimensional version of a matrix, the **tensor**:

Definition 407

Review from the Matrix Derivatives Chapter:

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.

Thus, a vector is a 1-d array, and a matrix is a 2-d array.

We can extend this to any number of dimensions. We call this kind of generalization a **tensor**.

Definition 408

Review from the Matrix Derivatives Chapter:

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**.

Example: If we stacked a bunch of matrices in a box in 3-d, that would be a 3-tensor.

To get element-wise multiplication, we'll need a way to index into tensors: we'll use numpy notation.

Notation 409

We want to **index** into a tensor T , with n axes ("dimensions")

We'll use indices $i_1, i_2, i_3, \dots, i_n$ to get an element:

$$T[i_1, i_2, i_3, \dots, i_n]$$

Finally, we can show the dot-product generalization for tensors: _____

Definition 410

The **dot product generalization** for an arbitrary **n-Tensor**

- We compute it by multiplying **element-wise**, then **summing** all the elements.

$$A \cdot B = \sum_{i_1, i_2, i_3, \dots, i_n} A[i_1, i_2, i_3, \dots, i_n] \cdot B[i_1, i_2, i_3, \dots, i_n]$$

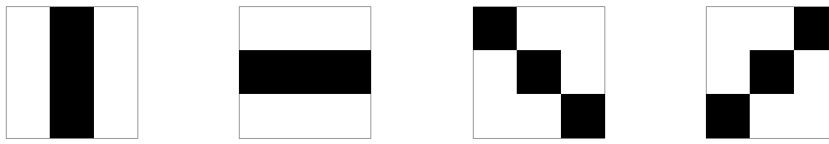
This is where python slicing really shines: it makes it easier to talk about grabbing an element from an unknown tensor.

9.1.11 Filter Banks

As we've shown, one filter produces one 2-d output: telling us where it "finds" or does not find the given pattern.

But, typically, when doing complex image analysis, we don't just want **one** filter. There are lots of different patterns we might be looking for.

- **Example:** Rather than programming every larger shape **directly**, it might be easier to look for smaller edges.
- You'll need a **different** filter for a vertical edge, or a horizontal edge, or a diagonal edge.



All four of these might be useful for the same image.

In practice, you almost always want to look for more than one pattern at the same time, in an image.

We'll store all of these filters together. Suppose we have m of these filters: each filter has size k .

- Each is a 2d matrix, so we'll **stack** them in the third dimension.
- This creates a **tensor** in the shape $(k \times k \times m)$.

This collection is called a **filter bank**.

Definition 411

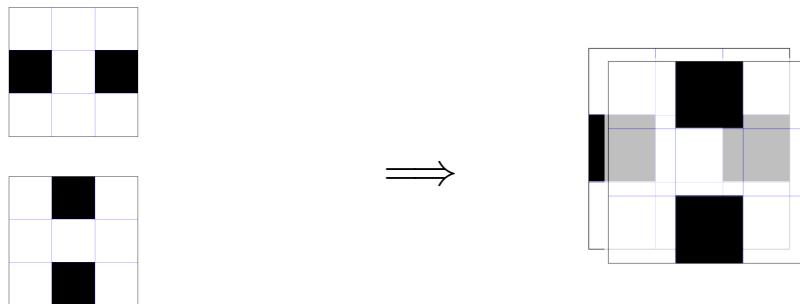
A **filter bank** is a collection of all of our $(k \times k)$ filters stacked into a **3-tensor**.

- Thus, if we have m of these filters, the **shape** is $(k \times k \times m)$.

These filters are all applied to our image in **parallel**: meaning, each is applied to the original image, and each creates a separate output.

Example: We might, for example, combine the two following filters:

It's difficult to visualize a 3d thing like this, so if this looks strange, don't worry.



Now, we have a very simple filter bank.

Clarification 412

This $(k \times k \times m)$ object could be a **filter bank**, but it could **also** be a single **3-tensor filter**, for a 3-tensor input.

Why would our **input** be a 3-tensor? We'll see why in a bit.

So, we'll use each of these filters, and **convolve** them with the input. Each creates a separate output stored in a separate **channel**.

Concept 413

A **channel** is the output of convolving **one filter** with our **image**.

In a 2d image problem, one channel is a **matrix**.

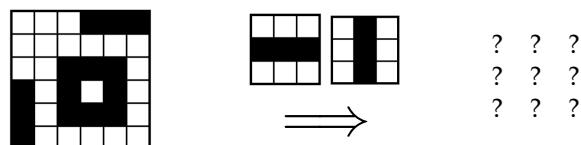
Each filter creates one channel. So, in order to depict all of our channels of output, we'll need another 3-tensor.

Concept 414

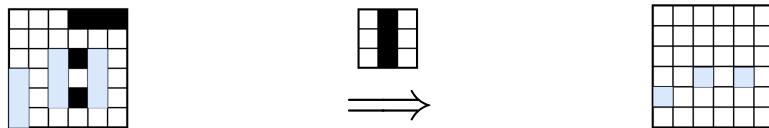
Suppose we have our 2d input.

If we have m filters in our **filter bank**, we end up with m **channels** in our output.

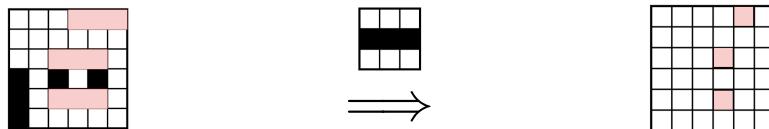
- Example:** Here, we'll apply two filters: one detecting vertical lines, one detecting horizontal lines. It'll create two channels of output.



We're applying a simple filter bank to the image on the left.

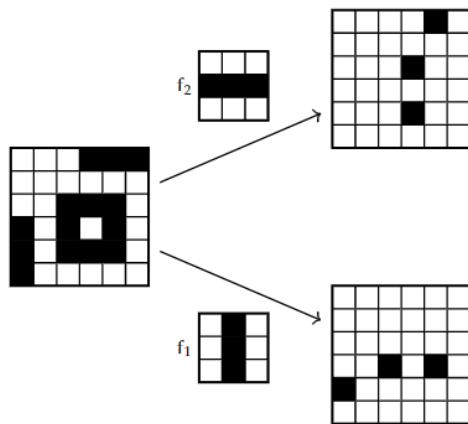


Our vertical detection.



Our horizontal detection.

Together, these create two channels:



9.1.12 Tensor Filters

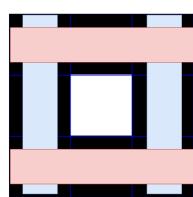
Now, we have two different channels in our output. What do we do with this result?

Each of our filters was designed to find a particular **pattern**: you could say it represents one "**perspective**" on the data.

- Our two filters above think about the data in terms of vertical lines, and horizontal lines.

We want to **combine** those perspectives to get useful information.

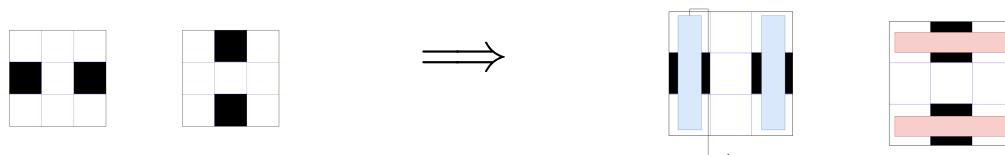
Example: A "square" is made out of two vertical lines, and two horizontal lines.



That means we want to find two **vertical** lines, and two **horizontal** lines: each on the opposite side of our center pixel.

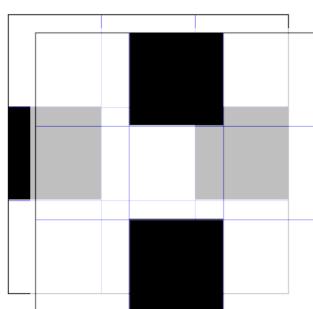
- This is a kind of **pattern** we could search for, but it's a pattern across **two channels**.
- That means that we need a filter occupying **multiple channels**.

Let's see the pattern we want to see on each channel:



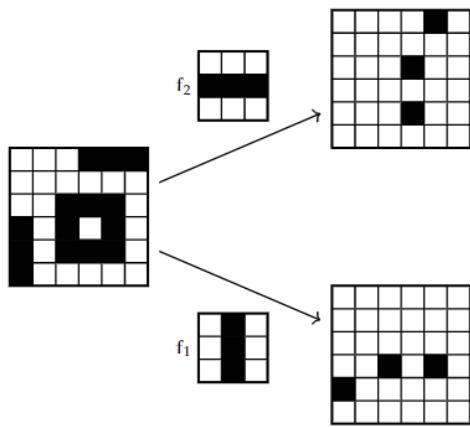
The right side shows what each pixel on the filter "represents".

We want both at the same time, so we create a **3d filter**:



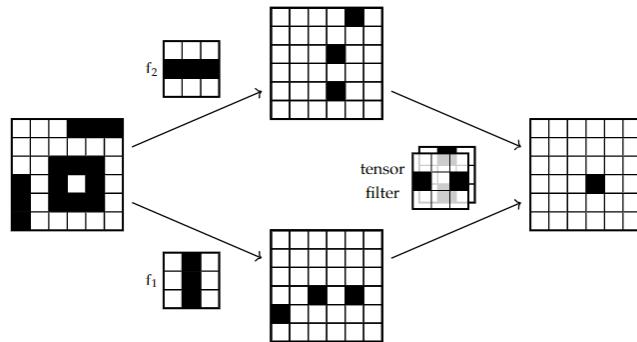
This looks like our 3d filter bank, of **2 filters**. But in this case, it's a **single** 3d filter.

Let's apply this trick to the two channels we designed earlier: we're going to find "donut" patterns in the original image.



Couldn't we have just used a donut-shaped filter in the first place, and then only need one filter?
Yes, but this is useful for demonstrative purposes.

Here's our previous work, finding vertical and horizontal lines.



And now, we'll combine those lines to create a square.

Look at that: our result is, we **only** get an output where there's a hollow square in the **original** input!

We've found a more interesting pattern, using a **second layer** of convolution.

9.1.13 Tensor Filters: All channels

When we introduce a **3-tensor filter**, we could imagine not only moving across the rows and columns of the input, as we convolve, but also the **channels**.

- Thus, we would need to shift our filter along 3 axes.

However, in practice, we frequently **avoid** this: instead, our tensor filter tends to have the **same number of channels** as the input tensor.

- If they have the same number of channels, then there's no space to "shift" along the third axis.

- That means that the output of this filtering is a single matrix again!

Concept 415

Typically, our **3-tensor filters** occupy **all channels of the input**.

- So, if the input is shape $(a \times b \times c)$, the filter has the shape $(k \times k \times c)$.

That means that our 3-tensor filter creates a **matrix** as its output, when we do convolution.

Technically, it's a tensor with the shape $(m \times n \times 1)$. The last dimension being 1 is why it's effectively a "matrix".

This allows us to add more tensor filters: our filter bank can contain multiple 3-tensors, and each one creates one channel of the output.

Concept 416

Often, we use several **3-tensor** filters: each one occupies **all of the input channels** at the same time.

- And each one outputs a single **matrix**.

That means that we can stack these 3-tensors into a **filter bank**: this object is now a **4-tensor**.

- When we apply this **4-tensor filter bank** to our **3-tensor input**, we get a **3-tensor output**.

This means that our filter bank is a 4-tensor..... don't think too hard about it.

9.1.14 Convolution is Linear

You might have noticed that, above, we could have **replaced** all of our filters with a single, donut-shaped filter.

- We didn't do this, so we could **demonstrate** how convolution works conceptually.

This is possible because convolution, being entirely made out of **multiplication** and **addition**, is a **linear** operation.

So, two consecutive convolutions are "**compressible**" to one, just like linear layers.

Concept 417

Machine learning "convolution", or "cross-correlation", is a **linear** operation.

- Here, we'll summarize linearity as "**multiplying** our variables x by scalars (in this case, weights from f), and **adding** the result together.

Summing Scalar
Variables Product

$$v \cdot f = \sum_i \overbrace{v_i}^{\text{Scalar}} \overbrace{f_i}^{\text{Variable}}$$

In fact, as we'll see later when doing backprop, **convolution** between x and f can be represented using a particular **matrix multiplication**.

This isn't a problem in practice because we'll add ReLU and max-pool layers in between: both are **nonlinear**.

That said, a "convolutional layer" is not a "linear layer":

We haven't discussed max-pool yet.

Clarification 418

While convolution is **linear**, a **convolutional layer** is different from a **linear layer**.

Why is that?

Because convolution is a very **restricted** kind of linear:

- In **convolution**, we use the **same filter** for every dot product – every operation uses the same weights in f .
- In a **fully connected**, "**linear layer**", every input-output pair has a **separate, independent** weight, that can be tuned freely.



You could think that a FC/"linear" layer refers to the broadest, **least-restricted** kind of linear transformation:

- Every possible linear relationship is allowed.

You could also think of it as the "simplest" linear layer: it makes the least assumptions.

9.1.15 RGB colors (Optional)

One quick concern, that's more pragmatic: all of our images, so far, have been in black-and-white.

How do real pictures create color? Using the **RGB** system: each pixel has a certain bright-

ness of red, green, and blue.

- So, to represent the pixel output, you need **three** numbers: a brightness in the $[0, 255]$ range for each color.

That means that our input isn't a 2d image: it's actually 3d.

Concept 419

If we're using **RGB** color instead of black-and-white (BW), each pixel requires **3 values** to represent the **brightness** of each color.

Thus, if our BW image had the shape $(m \times n)$, our RGB image has the shape $(m \times n \times 3)$: we use a **3-tensor** to store the extra information about color.

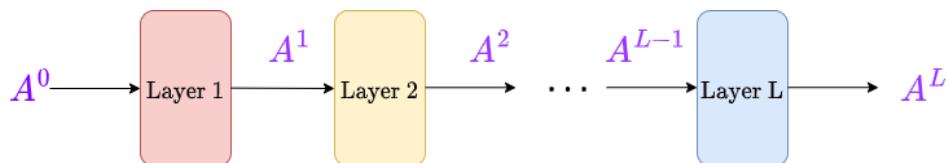
- Thus, our filters have to be 3d filters, as well.

9.1.16 Adding Convolution to our Neural Networks

So, we've developed a complete system for **convolution**, using **filter banks**. How do we apply this to **machine learning**?

Well, thankfully, our neural networks are very **modular**:

- Each FC layer is **self-contained**, and **abstracted**: when we depict it this way, we only care about the input and output dimensions.



By "self-contained" and "abstracted", we mean that we can hide the contents of the layer, while still having a useful representation.

We've broken our model into "modules" that we could swap out: this representation doesn't acknowledge the weights, or the structure.

So, it's not too different if, instead of a **fully-connected** layer, we were to have a **convolutional** layer.

Concept 420

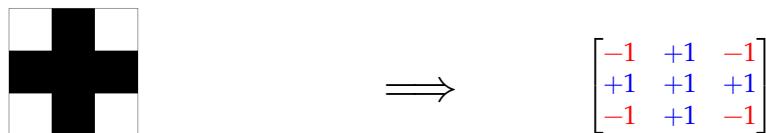
A **convolutional layer** can be inserted into a neural network by placing it **between** two other layers.

- You just need to make sure the input/output have the right **dimensions**.

Let's figure out how to **implement** that, while thinking in familiar NN terminology.

Convolution is based on your window and filter.

- Your **window** is simply given by your input tensor, and how far you've **shifted**.
- Your **filter** is what really defines your convolution: it chooses the **pattern** you're looking for.



So, we'll focus on the filter: this is how we determine the behavior of our convolutional layer.

- This is similar to how our **weight** matrix W is used to configure a layer of our FC NN.

- So, we say that our convolutional layer is defined by the **weights** in our filters.

Note that we say **filters**: we already established that we can use multiple filters. If we do, our output will have multiple channels.

Definition 421

Our **convolutional layer** is entirely determined by the **weights** we choose for our **filter bank**.

- For **each filter** in our filter bank, we'll also include a single **offset**, or bias term.

~~~~~

Suppose that, in the 1d case, we have a window of size  $k$ , for our input  $x$ . Our window shifted by  $i$ , is labelled  $v_{i+1}$ .

$$v_{i+1} = x[i : i + k] = \begin{bmatrix} x_{i+1} \\ x_{i+2} \\ \vdots \\ x_{i+k} \end{bmatrix}$$

We've chosen weights for our filter  $f$ , with a bias term  $f_0$ . The  $i^{\text{th}}$  element of our **output** for that filter is:

$$y_i = v_i \cdot f + f_0$$

**Example:** If we have a 2d filter of length  $k$ , then we need  $k^2$  weights, and 1 bias. We have  $k^2 + 1$  parameters.

$$f = \begin{bmatrix} W_{11} & W_{12} & \cdots & W_{1n} \\ W_{21} & W_{22} & \cdots & W_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ W_{m1} & W_{m2} & \cdots & W_{mn} \end{bmatrix} \quad f_0 = W_0 \quad (9.11)$$

Each convolutional layer has one filter bank, typically.

We casually toss in a **bias** term, similar to our neural network structure.

- But we should ask, what effect does that bias term have?

**Concept 422**

A **higher** filter output suggests a **higher chance** that our desired **pattern** is found at that position.

Thus, our **bias/offset** term can increase or decrease how "**sensitive**" we are to inputs similar to our pattern.

- If we increase the offset, we get a higher filter output: more inputs will appear as **positive**: possibly matching our pattern.

### 9.1.17 Training our Convolutional Layer

In the past, experts would **hand-craft** their filters, manually experimenting with them.

However, we've defined our convolutional layer in terms of **trainable** weights and biases.

- So, as long as we can take the **derivative**, we can use **gradient descent** to find filters which are more suitable for the task.

**Concept 423**

We can **train** the weights and biases used in our filter bank.

This requires doing **backpropagation** to find the gradient, but that's possible so long as we have well-defined **derivatives**.

We'll come back to how to compute these derivatives in the last section of this chapter.

Sometimes, these filters teach us interesting things about the **structure** of the data, based on which ones ended up being **useful**!

- They've even been found to sometimes recreate the types of successful designs made by humans.

### 9.1.18 Benefits of Convolution

We've already discussed some of the benefits of convolution:

**Concept 424**

Convolution provides **spatial locality** and **translation invariance**.

- **Spatial Locality:** our "filter" focuses on a **local** region of the image, and looks for a specific, **spatial** arrangement of pixels.
- **Translation Invariance:** we repeatedly apply the **same** filter as we **move** across the image. So, it will find our pattern and recognize it the same, no matter the position.

Of course, there's some caveats:

**Clarification 425**

Convolution doesn't perfectly provide translation invariance.

This is because of the **edges** of our image.

- If we don't use zero-padding, then information close to the edge of the image is scanned over **fewer** times.
- If we do use zero-padding, then the information close to the edge is **distorted** by the zeroes.

But there's one more surprising benefit.

The same filter is used, over and over again, as we move over the image.

- That means we repeatedly re-use the **same weights** for multiple different calculations.

This can be a bit confusing: the same weights will appear in different calculations, and thus different derivatives.

**Definition 426**

**Weight sharing** is a useful property of convolution, where the **same weights** are re-used for **multiple calculations**.

- In particular, the weights in a filter are used for many **dot products**, in the same convolution.

Having fewer weights allows our model to **train faster**, and possibly **overfit** less: it's a form of **regularization**.

**Example:** Let's compare two situations: in both, we have a  $(5 \times 5)$  image, and we want a  $(5 \times 5)$  output.

- FC Layer: We flatten our input and output to  $(25 \times 1)$ .
  - To get every combination of input and output, we need  $25 * 25$  weights.
  - 1 bias for each output:  $25 * 1$  biases.
  - **Total:** 650 parameters.
- Conv. Layer: We keep our current shape and use a single filter.
  - We use a  $(3 \times 3)$  filter, with one unit of padding ( $p = 1$ ). That means 9 weights.
  - We have one bias: 1 bias term.
  - **Total:** 10 parameters.

In a way, weight-sharing makes our model more **efficient**. In exchange, it's less **flexible**: it makes some assumptions about how our data is structured.

### 9.1.19 Our NN dimensions

So, we are considering introducing a convolutional layer with layer  $\ell$ .

We should be careful of how to notate our dimensions:

#### Notation 427

For a convolutional layer on layer  $\ell$ :

- **Input length:**  $n^{\ell-1}$
- **Input channels:**  $m^{\ell-1}$
- **Filter size:**  $k^\ell$
- **Number of filters:**  $m^\ell$
- **Padding length:**  $p^\ell$

A few notes:

- The input parameters are  $\ell - 1$ , because they're the **output** of the previous layer  $\ell - 1$ .
- Notice that  $m$  is used for the filter count, and the channels of the input.
  - The input is a previous output, and the **output** has the same number of **channels** as the **filter bank**.

Now, we can use these to get the shapes of some objects:

#### Definition 428

For a convolutional layer on layer  $\ell$ :

- **Input tensor shape:**  $(n^{\ell-1} \times n^{\ell-1} \times m^{\ell-1})$
- **Filter shape:**  $(k^\ell \times k^\ell)$
- **Filter bank shape:**  $(k^\ell \times k^\ell \times m^\ell)$

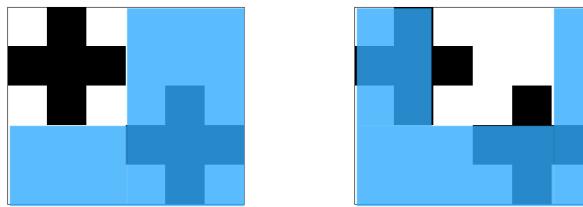
Again, we see that our input can be a **3-tensor**, if it has multiple channels.

### 9.1.20 Stride

There's one more thing we've skipped over: until now, we've assumed that, as we take a convolution, we move over, **one index** at a time.

But, this isn't required: we could move by **multiple** units.

This can either be due to filter banks, or the structure of the data (like in the RGB section).



This is the same as what we did before, except now, we've "skipped" one of our intermediate steps.

There are multiple reasons to do this, one of which involving "max pooling", which we discuss in the next section.

#### Definition 429

**Stride** is the distance you travel each time you **move indices** to take another **window** from your input.

**Example:** The stride we were using until now is 1. The stride we used in the above diagram is 2.

Naturally, this will shrink the size of your output:

#### Concept 430

Increasing **stride**  $s$  decreases the **size** of your output.

$$\text{New Size} = \left\lceil \frac{\text{Old Size}}{s} \right\rceil$$

We divide by  $s$ , because we're "skipping over" some windows: we are only taking a "fraction" of them.

Note the symbols on the side:

#### Notation 431

$\lceil x \rceil$  takes the "**ceiling**" of  $x$ : if  $x$  is between two integers, we round up.

We need this because the size of our output matrix is an **integer**.

- If we have a length-5 output with stride 1, but we take stride 2 instead, without rounding, we end up with size 2.5.
- So, instead we round.

### 9.1.21 Output shape

Now, we have all the tools we need, to compute the output shape, based on the input shape.

Three things can affect our input shape:

- Filter size:  $n - (k - 1)$
- Padding:  $n + 2p$
- Stride:  $\lceil n/s \rceil$

Taking all of these variables together, we get this result (which is important, and worth saving!):

#### Key Equation 432

Suppose we apply **convolution** to a matrix, with

- **Input size**  $n^{\ell-1}$
- **Filter size**  $k$
- **Padding**  $p$
- **Stride**  $s$

The **output size** will be

$$n^\ell = \left\lceil \frac{n^{\ell-1} - (k^\ell - 1) + 2p^\ell}{s^\ell} \right\rceil$$

~~~~~

More commonly, you will see a (surprisingly) equivalent expression:

$$n^\ell = \left\lfloor \frac{n^{\ell-1} - k^\ell + 2p^\ell}{s^\ell} + 1 \right\rfloor$$

- Instead of the ceiling function $\lceil x \rceil$, which rounds up, we have the floor function $\lfloor x \rfloor$, which rounds down.

Example: Let's take an input tensor of shape $(64 \times 64 \times 3)$.

Our filter is size 2 ($k = 2$), with stride 2 ($s = 2$).

- It needs to have 3 channels, to match the input. Thus, $(2 \times 2 \times 3)$.

Using our equation for the size of our output, we get

$$\lceil(64 - (2 + 1) + 2 \cdot 0)/2\rceil = \lceil(63)/2\rceil = 32$$

So, our output dimensions are $(32 \times 32 \times 1)$.

This example is slightly different from the official notes: there, we were doing max-pool, so we keep all 3 channels.

9.2 Max-pooling

So, we've used our filters to find **basic** patterns, roughly matching the filter.

- But earlier, we showed an example where we used **two layers** of convolution, to create a more complex pattern from a simpler one.

Of course, in that case, the two layers were reducible to a **single** layer, because convolution is a **linear** operation.

But we could get a more "true" version of this idea, by introducing a new function: the **max-pool** operation.

9.2.1 Aggregating information

Let's clearly state our goal:

Concept 433

One goal of **multi-layer convolution** is to

- Find **local**, smaller patterns
- Combine them to create **bigger**, more complex patterns

With each layer, we find broader and broader patterns.

~~~~~  
However, we need a way to truly "aggregate" those patterns together.

This is the goal of our **max-pool** function.

- **Example:** We combine simple edges, into larger, **longer** edges, then into shapes like **squares**.
- Then, those combine into **windows** and doors and roofs, and finally, if they're arranged correctly, we use them to draw a "**house**".

We need a function that allows us to "**aggregate**" data this way.

~~~~~  
Here's one idea: what happens as we move to higher size scales, building up a more **complex** object?

- We tend to care **less** about the smaller, individual details.

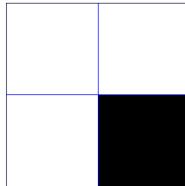
Rather than knowing **exactly** where a pattern is, we might simplify that to knowing **approximately** where that pattern is.

Our house picture doesn't stop being a house, if we slightly corrupt some of the edges, for example.

That way, we can gather information: "in this general **section** of the image, I found the pattern we're looking for!"

- How do we implement this?

It sounds like we want to replace "here's the exact pixel we found the pattern in", with "we found the pattern in this **general area**".



If black pixel indicates a "success" for finding the pattern, then this (2×2) grid does contain our pattern!

Definition 434

The **region** we're looking for our pattern in is called our **receptive field**.

Typically, it's square grid: $(k \times k)$.

9.2.2 Deriving max-pool

In this image, how did we know that the pattern was here? We noticed, "**at least** one pixel in this grid detected the pattern".

- In other words, we don't care about pixels that **don't** detect the pattern.
- And we don't care if there are **multiple**.

We can get our desired behavior by taking the **max** of these pixels: the pixel with the greatest output, is the **most similar** to our pattern.

- So, if the "most similar" pixel doesn't match our pattern, then none of the others will, either.
- Naturally, this also ignores multiple instances of our pattern.

This process, of **pooling** together all the pixels in our receptive field, and taking their **maximum**, is called the **max-pool** operation.

We'll repeatedly apply this process, all over our image: that way, we can aggregate over each region.

Definition 435

The **max-pool** operation, applied to a **receptive field**, takes the **max** of all values over that region.

Similar to convolution, we repeatedly **shift** our receptive field, and compute the max again.

- Also similar to convolution: the **number of times** we've **shifted** over, is the **index** of the output.

Example: Here's a 2x2 max-pool operation, with a **stride** of 2:

$$\begin{bmatrix} 1 & 3 & -9 & -22 \\ 44 & -10 & -11 & -1 \\ 0 & 10 & 4 & -3 \\ 11 & 9 & 321 & 99 \end{bmatrix} \xrightarrow{\text{max-pool}} \begin{bmatrix} 44 & -1 \\ 11 & 321 \end{bmatrix} \quad (9.12)$$

Concept 436

Max-pool typically only uses a **2d matrix** for its **receptive field**: we apply the max-pool operation separately, for each channel of our input.

So, the number of channels is the same, before and after our input.

9.2.3 Max-pool stride

Notice that, in our example above, we've **shrunk** the image, while preserving some general data.

Max-pooling is typically designed to "gather" data across our image:

- If we apply it after a **convolutional layer**, it can help us figure out if the receptive field contains a **pattern**.

In other words, we're **searching** for our pattern over a **larger** region.

So, it might be natural to take **bigger steps** in between each max-pool, since each max-pool condenses a whole receptive field of information.

Key Equation 437

In order to get a simplified, "broader" view of the data, our **max-pool** often uses a larger **stride s**:

$$s > 1$$

This means we move our receptive field by a larger amount, between max-pools. This **shrinks** our output.

- If we apply this for multiple consecutive layers, we get a "**pyramid**" shape, where our output gradually shrinks in size.

With this approach, we can store the information we care about ("pattern found roughly here/not here"), without focusing on the exact, **pixel-perfect** detail.

- It's also useful for building larger objects: if two patterns (from two **filter channels**) are **roughly** nearby, it'll be easier to recognize a **larger shape**.

If we don't want to shrink our image as much, we can use **zero-padding**, usually on our convolution step.

Because often, different instances of a shape won't be exactly the same: this makes it easier to recognize them anyway.

That said, while we want a stride that's bigger than 1, we don't want it to be so big that we **skip** some of the input.

Key Equation 438

In order to avoid **skipping** portions of the input, we don't want our **stride s** to be larger than our filter of **size k**.

$$k \geq s$$

9.2.4 Clarifications on max-pooling

Our max-pool essentially chooses the "most likely to match" output, after the filtering. Based on that result, we can guess whether this region contains our pattern.

Clarification 439

Neither **convolution** nor **max-pooling** exactly tell us if we found a pattern **match** at a particular **index**.

Instead, they give us a **number**, based on a (generalized) **dot product**, that can be **interpreted** to check for a pattern match.

- Whether that number confirms our pattern, depends on how **high** it is.

Our **offset** can help with this problem: it helps us set a "**threshold**":

$$v_i \cdot f > -b \implies v \cdot f + b > 0$$

If we set b correctly, we could simply say, "the pattern appears if $v \cdot f + b > 0$ ".

- This threshold, like our other parameters, will be **learned** by the neural network.

Example: Consider the following example:

$$\begin{bmatrix} +1 \\ +1 \\ +1 \end{bmatrix} \cdot \begin{bmatrix} +1 \\ +1 \\ -1 \end{bmatrix} = 1 + 1 - 1 = +1 \quad (9.13)$$

This output is **positive**, but the two patterns aren't visually the same. Whether they're **similar** enough depends on the context.

- If they're not similar enough to justify a positive output, we could use a **negative offset** to make our filter less sensitive.

A related comment: some pattern matches might be ambiguous.

Clarification 440

In this chapter, for our images, we've exclusively used simplified images, with only **extreme** brightnesses (-1 or +1).

However, in most real images, there's a **spectrum** of brightnesses.

This can make it **harder** to figure out whether you really find a pattern, in a particular place.

Example: Whether the following image contains our pattern is unclear. The left is our filter,

the right is our window.



The pixel in the center of the window is a bit brighter than the surroundings, but... not by very much.

This is another place where our bias can be useful to set a threshold.

9.2.5 Max-pool: A functional layer

Max-pool, as a specific variant of the max function, has **no parameters**: "compute the maximum input value" isn't a function that requires **adjustment**.

Concept 441

max-pool has no **parameters**: it always behaves the same way.

This also means that it doesn't need to be **trained**.

Because it has no weights, and behaves like a **function**, we can think of this a **pure functional layer**.

- Activation functions for linear layers, like **ReLU**, behave the same way: they require **no parameters**.

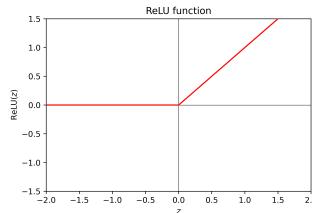
On that same note: while max-pool is *technically* nonlinear, we usually don't use it to provide nonlinearity to our model.

Instead, we accomplish this by applying ReLU **after** our convolution.

Concept 442

After convolution, we often apply a **ReLU function** to provide **nonlinearity** to our model.

Typically, we follow a sequence of **convolution**, **relu**, and then **max-pool**, before repeating.



A reminder of how the relu function appears.

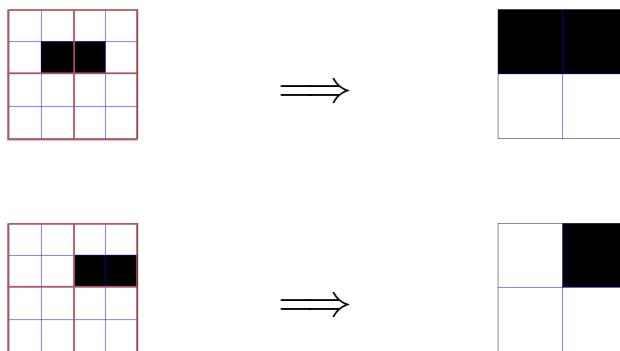
9.2.6 Max-pool: Some problems with "translation invariance".

There's one problem with having a larger stride:

- We skip some of the possible receptive fields.

This means that the same pattern could look different, based on where it's placed.

Example: We can get markedly different results of our max-pooling, just by shifting the input:



We shifted over the input, and lost one of our two black pixels: that doesn't seem very translation-invariant.

Concept 443

Using a stride s greater than one creates an output that isn't translation-invariant:

- if you shift part of the input slightly, it can alter the pattern recognition of the output.

This is notable for max-pool, which almost always uses $s > 1$.

Here, we provide a paper that provides a potential solution.

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

- In short, the idea is to max-pool with stride 1, and then scale down our output by averaging over the results.

9.3 Typical architecture

Now that we've built all the pieces of our new neural net, we can get the general flow of a neural network that implements convolution: a **Convolutional Neural Network** (CNN).

First, let's consider what we need for each layer of convolution:

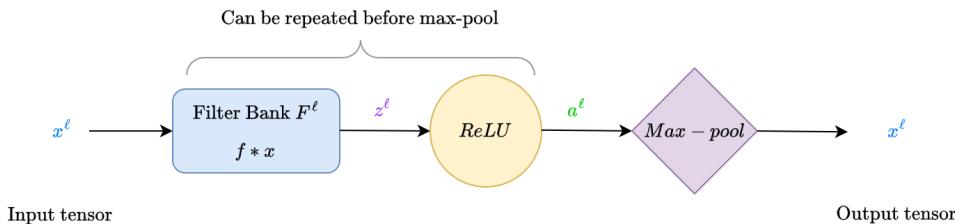
Concept copied from above, for convenience.

Concept 444

After convolution, we often apply a **ReLU function** to provide **nonlinearity** to our model.

We typically use layers of **convolution**, **relu**, and then **max-pool**, then repeat.

- Notably, we may do conv+relu multiple times before one max-pool.



This is our basic structure: often, we repeat this multiple times.

Once we reduced our input to a reasonably small size, we finish by using a **fully connected layer**.

The convolutional layers can be seen as "preparing" the data, so that our fully-connected network can more easily find the patterns it needs.

We could even model it as a very complex feature transformation!

Definition 445

A **Convolutional Neural Network** (CNN) is a neural network which uses **convolution** to transform data.

Most typically, we take the following structure:

- Several layers of **conv-relu-maxpool**, gradually shrinking the **output** size
- A **fully-connected** network, typically intended for classification or regression.

The model is **evaluated** based on the performance on the chosen classification/regression task.

- This can be viewed as several layers of convolution, applied before a regular neural network.

We can refer to our earlier comments for some benefits of CNNs:

Concept 446

Convolution provides benefits through **spatial locality**, **translation invariance**, and **weight sharing**.

- **Spatial Locality**: our "filter" focuses on a **local** region of the image, and looks for a specific, **spatial** arrangement of pixels.
- **Translation Invariance**: we repeatedly apply the **same** filter as we **move** across the image. So, it will find our pattern and recognize it the same, no matter the position.
- **Weight sharing**: the same filter weights are **re-used** for many calculations. This can speed up training, and reduce overfitting.

As a result, CNNs tend to perform well for image-based problems.

One might ask: how many **layers** of convolution? How many **filters** per layer? What **size** should these filters have?

- These questions are good ones, but they're very **difficult** to answer: very few hard rules exist for how to design this kind of network.
- Often, designs are based on what has worked in the **past**, or some **intuition** about the data.

Once we've designed our network, we can begin training.

Concept 447

CNNs can be trained just like normal neural networks: we train both the **fully connected** network, and the **filter weights** throughout the convolutional layers.

To do gradient descent, we measure the **performance** of our CNN on the classification/regression task in question.

We close out this section with an example of a "typical" CNN:

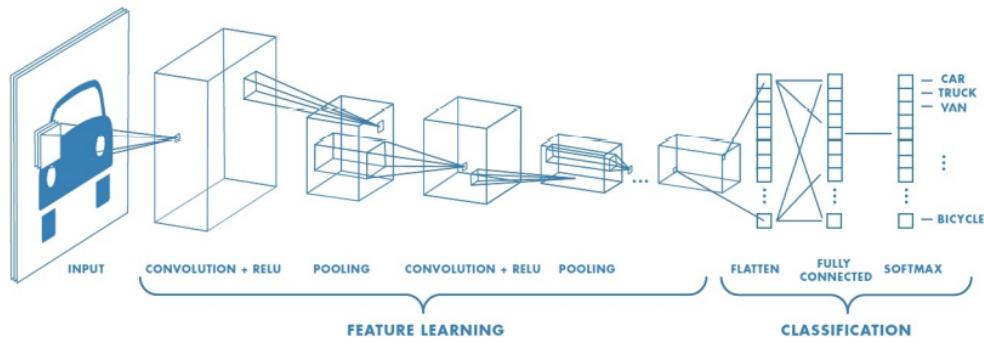


Figure source: <https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html>

A few comments:

- Our input has **three** layers, to show the **RGB** color channels.
- In our convolution and pooling sections, we have **boxes**, not flat matrices: those represent **3-tensors**.
- Our last layer is **softmax**: our typical output for multi-class classification.
- We do end up **flattening** after our convolution is finished: that's necessary for feeding our FC network.

9.4 Backpropagation in a simple CNN

Now that we have a new type of neural network, it's only appropriate that we learn how to **train** it!

- Our filtering, ReLu, and maxpool functions are (mostly) **continuously differentiable**, so we can get useful derivative for **gradient descent**.

9.4.1 Our Simplest Example

We'll consider the simplest possible example, with a **1d** input.

- An **n**-length, one-channel, 1d input x : shape $(n \times 1 \times 1)$.

- We'll use zero-padding of length **p**, to create our input $X = A^0$.

Reminder that we're using A^ℓ notation to indicate the ℓ^{th} layer of our network.

With padding, our shape is $((n+2p) \times 1 \times 1)$.

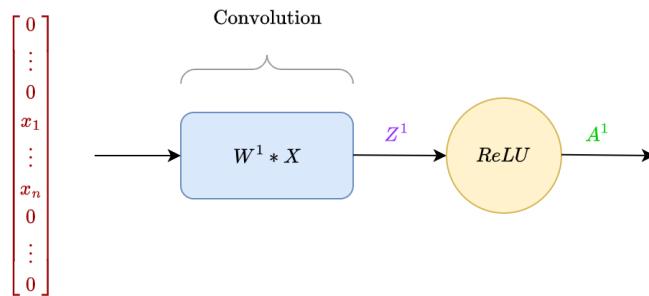
$$A^0 = X = \begin{bmatrix} 0 & \cdots & 0 & \overbrace{x_1 \cdots x_n}^x & 0 & \cdots & 0 \end{bmatrix}^T \quad (9.14)$$

- One layer of **conv-relu**

- Our convolution has one size-**k** filter, with weights W^1 : shape $(k \times 1 \times 1)$.
- Stride **s = 1**.

$$Z^1 = \underbrace{W^1 * A^0}_{\text{Convolution}} \quad \rightarrow \quad A^1 = \text{ReLU}(Z^1) \quad (9.15)$$

We can visualize our results so far:



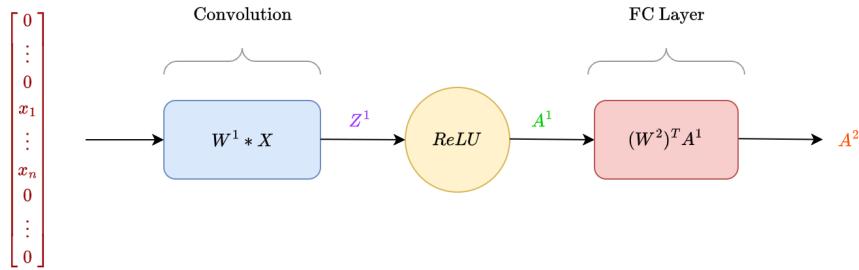
- A single FC layer for **regression**

- Using weights W^2 , with no bias.

$$A^2 = (W^2)^T A^1 \quad (9.16)$$

- A loss function using **squared difference**.

$$\mathcal{L}(A^2, y) = (A^2 - y)^2 \quad (9.17)$$



Notation 448

Reminder that, in this chapter, $a * b$ refers to the machine learning convolution/[cross-correlation](#) between a and b .

- In other fields, $a * b$ refers to the "true" convolution, where we flip the order of either a or b before using the same operation.

9.4.2 Chain rule to get full derivative

We know how to do gradient-descent on our **fully connected** layer already, and our **ReLU** layer is purely functional: no trainable weights.

So, all that's left is our **filter** derivative: our filter is parametrized by W^1 .

$$\frac{\partial \mathcal{L}}{\partial W^1} \quad (9.18)$$

Looking at our above diagram, we can "move backwards" in steps, building up a **chain rule**, until we reach W^1 .

$$\frac{\partial \mathcal{L}}{\partial A^2} \rightarrow \frac{\partial \mathcal{L}}{\partial A^2} \cdot \frac{\partial A^2}{\partial A^1} \rightarrow \frac{\partial \mathcal{L}}{\partial A^2} \cdot \frac{\partial A^2}{\partial A^1} \cdot \frac{\partial A^1}{\partial Z^1} \quad (9.19)$$

Finally, we get:

$$\frac{\partial \mathcal{L}}{\partial W^1} = \frac{\partial \mathcal{L}}{\partial A^2} \cdot \frac{\partial A^2}{\partial A^1} \cdot \frac{\partial A^1}{\partial Z^1} \cdot \frac{\partial Z^1}{\partial W^1} \quad (9.20)$$

9.4.3 Easy, Familiar Derivatives

We already know several of these terms:

$$\mathcal{L}(A^2, y) = (A^2 - y)^2 \implies \frac{\partial \mathcal{L}}{\partial A^2} = 2(A^2 - y) \quad (9.21)$$

Remember that A^2 is not "A squared": it's A for layer 2.

$$A^2 = (W^2)^T A^1 \implies \frac{\partial A^2}{\partial A^1} = W^2 \quad (9.22)$$

9.4.4 ReLU Derivative

Our next derivative is ReLU, one of the tricky **functional layers**.

$$A^1 = \text{ReLU}(Z^1) \quad (9.23)$$

For a full dive explanation of this derivative, go to [Explanatory Notes – Matrix Derivatives, A.9.4](#).

For now, we'll take the result for granted.

Concept 449

Review from Matrix Derivative Chapter:

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

$$\frac{\partial a_j}{\partial z_i} = 0 \quad \text{if } i \neq j$$

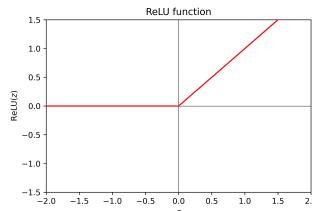
So, our only nonzero derivatives are

$$\frac{\partial a_i}{\partial z_i}$$

So, our result is a **diagonal** matrix: the off-diagonal elements are all zero.

$$\frac{\partial A_i}{\partial Z_j} = \begin{cases} f'(Z_i) & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \quad (9.24)$$

What about the **diagonal** elements? Well, that's based on the **ReLU** function.



Another picture of our ReLU function.

$$f(Z_i) = \begin{cases} Z_i & \text{if } Z_i > 0 \\ 0 & \text{otherwise} \end{cases} \implies f'(Z_i) = \begin{cases} 1 & \text{if } Z_i > 0 \\ 0 & \text{otherwise} \end{cases} \quad (9.25)$$

We get our final result by combining these two facts: the diagonal structure, with the ReLU derivative.

Key Equation 450

The **derivative** between the length- m input Z and output A of the **ReLU** function is an $(m \times m)$ **diagonal matrix**, whose diagonals are

$$\frac{\partial A_i}{\partial Z_i} = \begin{cases} 1 & \text{if } Z_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

Being a diagonal matrix, the off-diagonal elements are all zero.

$$\frac{\partial A_i}{\partial Z_j} = \begin{cases} 1 & \text{if } Z_i > 0 \text{ and } i = j \\ 0 & \text{otherwise} \end{cases}$$

Example: One possible matrix might look like

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (9.26)$$

Where Z_3 was negative, and thus it was in the 0, flat region of ReLU.

9.4.5 Filter Derivative

Lastly, we want to compute a **derivative**, based on the output of our **filter**.

$$Z^1 = W^1 * A^0 \implies \frac{\partial Z^1}{\partial W^1} = ??? \quad (9.27)$$

The problem is: we don't have a derivative for our **convolution**. Let's find an **expression** to get the derivative from.

The easiest way to do that is to look at our **equations** for convolution:

Key Equation 451

Review from above, section 9.1.5

If we have **signal** x , **filter** f of **size** k , we can create a **window** v_i by...

Starting at the **leftmost** pixel, and shifting right by i units:

$$v_{i+1} = x[i:i+k] = \begin{bmatrix} x_{i+1} \\ x_{i+2} \\ \vdots \\ x_{i+k} \end{bmatrix}$$

- Note the subscript v_{i+1} : we start from $i = 0$, and thus v_1 .

This is used to create our **convolution** $y = x * f$:

$$y_i = f \cdot v_i$$

Let's try to create something differentiable out of our equation for elements of Z : _____

$$Z_i = W \cdot v_i = \begin{bmatrix} W_1 \\ W_2 \\ \vdots \\ W_k \end{bmatrix} \cdot \begin{bmatrix} X_i \\ X_{i+1} \\ \vdots \\ X_{i+k-1} \end{bmatrix} = \sum_{j=1}^k W_j X_{i+j-1} \quad (9.28)$$

We'll swap out v_{i+1} for v_i . This makes our slicing a little ugly, but we'll just omit that.

Now we have something differentiable: a sum of products! Let's find $\frac{\partial Z_i^1}{\partial W_j^1}$.

- If we're differentiating with W_j^1 , we can ignore every term of the sum that doesn't include it.
- All that remains is the one term containing W_j .

$$W_j^1 X_{i+j-1} \quad (9.29)$$

Thus, we find:

$$Z_i^1 = \sum_{j=1}^k W_j^1 X_{i+j-1} \implies \frac{\partial Z_i^1}{\partial W_j^1} = X_{i+j-1} \quad (9.30)$$

Key Equation 452

The derivative between the output of the convolution and its weights are given by a matrix, containing elements from the input:

$$\frac{\partial Z_i}{\partial W_j} = X_{i+j-1}$$

The matrix for $\frac{\partial Z^1}{\partial W^1}$ has the shape $(k \times n)$.

Example: Suppose we had a simple example: 4 weights in W , 6 variables in X . With stride 1, that gives 3 outputs in Z .

$$\frac{\partial Z^1}{\partial W^1} = \begin{bmatrix} X_1 & X_2 & X_3 \\ X_2 & X_3 & X_4 \\ X_3 & X_4 & X_5 \\ X_4 & X_5 & X_6 \end{bmatrix} \quad (9.31)$$

With this last derivative, we can assemble our chain rule, and compute the gradient for $\partial \mathcal{L} / \partial W^1$.

We can confirm this with some shapes:

- Size of X is m : $(m \times 1)$.
- Size of Z^1 and A^1 is n : $(n \times 1)$.
- Size of A^2 is 1: it's a scalar.
- Size of filter W^1 is k : $(k \times 1)$.

Using our knowledge from the matrix derivatives chapter, we can confirm our shapes:

$$\underbrace{\frac{\partial \mathcal{L}}{\partial W^1}}_{(k \times 1)} = \underbrace{\frac{\partial Z^1}{\partial W^1}}_{(k \times n)} \cdot \underbrace{\frac{\partial A^1}{\partial Z^1}}_{(n \times n)} \cdot \underbrace{\frac{\partial A^2}{\partial A^1}}_{(n \times 1)} \cdot \underbrace{\frac{\partial \mathcal{L}}{\partial A^2}}_{(1 \times 1)} \quad (9.32)$$

9.4.6 Maxpool derivative

We didn't include a **maxpool** unit in our CNN. How do we compute the **derivative** of that?

Well, let's consider a simplified case: a 1d window of 2 elements: a_1 and a_2 .

$$\text{maxpool}(A) = \max \left(\begin{bmatrix} a_1 & a_2 \end{bmatrix} \right) = \begin{cases} a_1 & \text{if } a_1 \geq a_2 \\ a_2 & \text{if } a_1 < a_2 \end{cases} \quad (9.33)$$

- Notice that if $a_1 = a_2$, it **doesn't matter** which of the two you select.

We can just take the derivative from one of these inputs, let's say a_1 .

$$\text{maxpool}(A) = \begin{cases} a_1 & \text{if } a_1 \geq a_2 \\ a_2 & \text{if } a_1 < a_2 \end{cases} \implies \frac{\partial \text{maxpool}(A)}{\partial a_1} = \begin{cases} 1 & \text{if } a_1 \geq a_2 \\ 0 & \text{if } a_1 < a_2 \end{cases} \quad (9.34)$$

This gives us something we can **generalize** to more a_i terms:

- If a_i is **biggest**, then it'll be the output of maxpool, and it'll have an effect – a **nonzero** derivative.
- If a_i is **not biggest**, then it's not included in the maxpool output, and it has **no effect** – a zero derivative.

Example: If you're taking the maximum, and the largest number is 1000, it doesn't matter if the second largest number is 999 or 2.

Key Equation 453

The **maxpool derivative** is only nonzero for its **maximum** value.

$$\frac{\partial \text{maxpool}(A)}{\partial a_i} = \begin{cases} 1 & \text{if } a_i = \text{maxpool}(A) \\ 0 & \text{otherwise} \end{cases}$$

When we take all of the a_i derivatives and combine them into a **vector**, we realize that we have a **one-hot vector**, telling us which output was the **maximum**.

Example: Suppose a_4 was the largest out of 6 inputs in a column vector a .

$$\frac{\partial \text{maxpool}(A)}{\partial a} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad (9.35)$$

9.4.7 Maxpool derivative: somewhat similar to sign function (Optional)

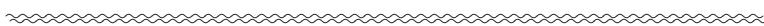
Interestingly, when two values a_i and a_j are both max, maxpool behaves like ReLU.

If $a_1 = a_2$, and both are max, we have two cases:

- If a_1 decreases, it has no effect on the output: **derivative 0**.
- If a_1 increases, it increases the maxpool output directly: **derivative 1**.

So, the derivative is different moving left or right: it's **undefined**.

We'll ignore this edge case, just like we usually do for ReLU.



But what if we're not at the edge case, and a_i is our max value?

- Well, **decreasing** or **increasing** a_i will have a 1-1, linear effect, until we reach the **second-largest** term, a_j .
- Then once we're below a_j , a_i it has no effect.

We still see that, for any one particular a_i term, maxpool behaves like a shifted ReLU, and its derivative like the step function.

Key Equation 454

The derivative of maxpool for a particular a_i term behaves like the **step function**.

- The transition from 0 to 1 occurs at the **highest a_j term, excluding a_i** .

This is true regardless of whether a_i is currently the max.



By integrating, we see that maxpool, then, behaves like a **ReLU function**, shifted on the input/output dimensions.

9.5 Terms

- Connected
- Fully Connected
- Flattening
- Spatial Locality
- Translation Invariance
- Window
- Dot Product (Review)
- Filter
- Convolution
- Cross-Correlation
- Padding
- Dot Product Generalization
- Filtering
- Tensor (Review)
- Filter bank
- Channel
- 3-tensor Filter
- Linear Layer (Review)
- Convolutional Layer
- Weight sharing
- Stride
- Max-pooling
- Receptive Field
- Functional Layer
- Convolutional Neural Network

APPENDIX A

Matrix Derivatives

A.1 Introduction and Review

Our goal here, is 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

$$\mathbf{w}^T \mathbf{x} = \begin{bmatrix} w_1 & w_2 & \dots & w_m \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \sum_{i=1}^m x_i w_i \quad (\text{A.1})$$

In this case, we're able to do m different **multiplications** at the same time! This is what we like about matrices.

In this case, we're thinking about vectors as $(m \times 1)$ matrices.

- **Calculus:** analyzing the way different variables are **related**: how does changing x affect y ?

Example: Suppose we have

$$\frac{\partial f}{\partial x_1} = 10 \quad \frac{\partial f}{\partial x_2} = -5 \quad (\text{A.2})$$

Now we know that, if we increase x_1 , we increase f . This **understanding** of variables

is what we like about derivatives.

Concept 455

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**.
- These "data" are 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.

$$\underbrace{X}_{(a \times b)} \quad \underbrace{Y}_{(b \times c)} \quad (A.3)$$

We can't do anything that would **violate** matrix rules like these: otherwise, we're not really doing "matrix **calculus**" anymore. This means we need to build our math carefully.

First, we'll look at the **properties** of derivatives. Then, we figure out how to usefully apply them to **vectors**, and finally, to **matrices**.

A.1.1 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 456

The **partial derivative**

$$\frac{\partial B}{\partial A}$$

Is useful when there may be **multiple variables** in our functions.

The rule of the partial derivative is that we keep every variable **other** than A and B **fixed**.

- This is different from the "total" derivative $\frac{dB}{dA}$, where we consider how B affects A, without keeping other variables fixed.
- The total derivative is also used in multi-variable calculus, but serves a different role: we'll come back to this later.

Example: Consider $f(x, y) = 2x^2y$.

$$\frac{\partial f}{\partial x} = 2(2x)y \quad (\text{A.4})$$

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.
- Since this is too complicated, we'll instead change these 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 457

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:

$$\frac{df}{ds} = \sum_{i=1}^k \underbrace{\frac{\partial f}{\partial x_i} \frac{\partial x_i}{\partial s}}_{x_i \text{ component}}$$

- We justify a lot of aspects of the multivariable chain rule, and the gradient, in our Gradient Descent chapter. Feel free to review there.

A.1.2 Thinking about derivatives

The typical definition of derivatives

$$\lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h} \quad (\text{A.5})$$

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".

$$A = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad B = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \quad (\text{A.6})$$

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, $\partial b_i / \partial a_j$.

3 dimensions of A times
2 dimensions of B: 6
combinations.

- Every component of the input A can potentially modify **every** component of the output B .

One solution we could try is to just collect all of these derivatives into a **vector** or **matrix**.

Concept 458

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?



A.1.3 Derivatives: Approximation

To answer this, we need to ask ourselves *why* we care about these derivatives: we'll define them based on what we **need** them for.

- We care about the **direction of greatest decrease**: the negative of the **gradient**.
 - One application: we might want to adjust weight vector w to reduce loss \mathcal{L} .
- To find the gradient, we'll need some other **matrix** derivatives that will be useful in a **chain rule**.
 - The chain rule creates lots of intermediate derivatives, that we need to work with.

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} .

$$\frac{\partial \mathcal{L}}{\partial w} \approx \frac{\text{Change in } \mathcal{L}}{\text{Change in } w} = \frac{\Delta \mathcal{L}}{\Delta w} \quad (\text{A.7})$$

Thus, we **solve** for $\Delta\mathcal{L}$:

All we do is multiply both sides by Δw .

$$\Delta \mathcal{L} \approx \frac{\partial \mathcal{L}}{\partial w} \Delta w \quad (\text{A.8})$$

Since this derivation was gotten using scalars, we might need a **different** type of multiplication for our **vector** and **matrix** derivatives.

Concept 459

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**.

A.2 Derivative: Scalar/Vector (Gradient)

Our plan is to look at every derivative combination of scalars, vectors, and matrices we can.

First, we consider:

$$\frac{\partial(\text{Scalar})}{\partial(\text{Vector})} = \frac{\partial s}{\partial v} \quad (\text{A.9})$$

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**.

$$\Delta v \longrightarrow [f] \longrightarrow \Delta s \quad (\text{A.10})$$

How do we make sense of this? Well, let's write Δv_i explicitly:

$$\underbrace{\begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix}}_{\Delta v} \longrightarrow \Delta s \quad (\text{A.11})$$

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 .

A.2.1 Finding the scalar/vector derivative

But how do we shape our matrix? Let's look at our **rule**.

$$\Delta s \approx \frac{\partial s}{\partial v} * \Delta v \quad (\text{A.12})$$

We can transform using our shapes:

$$\Delta s \approx \frac{\partial s}{\partial v} * \underbrace{\begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix}}_{\Delta v} \quad (\text{A.13})$$

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**.

Last Updated: 11/08/23 21:00:09

It's usually possible to change each v_i , so we have to look at every one of them.

One problem, though: how can we treat each **derivative** separately?

- Suppose we only start by considering Δv_1 : it'll move us along one axis.
- But our derivative is based on our position! If we've just moved, all of our derivatives have changed.
- So, choosing Δv_1 first, had an effect on the other Δv_i terms: they're not really "separate".

So, what do we do?

A.2.2 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 460

If you have a **smooth function**, then...

If you take sufficiently **small steps**, then you can treat the derivatives as **constant**.

This is called the **planar approximation** because that's how a smooth surface looks, at small scales:

- The derivatives of a flat plane are the same at every position: they're **constant** (we show this below).
- This is, of course, an **approximation**: we can't use it at every position, just "very nearby positions".

Clarification

This clarification 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. In other words, $\Delta f'(x)$ is **small**.

Let's say we don't want it to change more than δ : this is our definition of "very small".

If you want

- $\Delta f'(x)$ to be very small ($|\Delta f'(x)| < \delta$)
- It has been proven that...
 - it's possible to 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 461

Why is this **true**? Because a **hyperplane** can be represented using our **linear** function

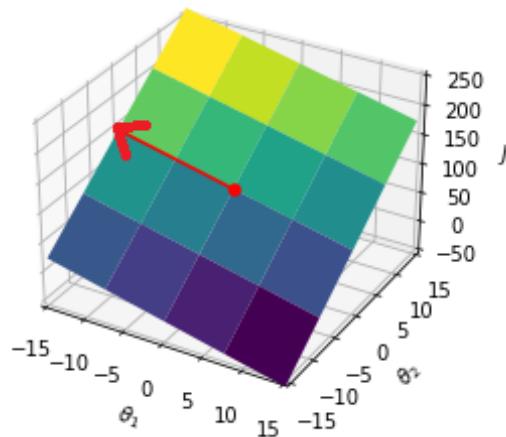
$$f(x) \approx \theta^T x + \theta_0 = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_m x_m$$

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**.

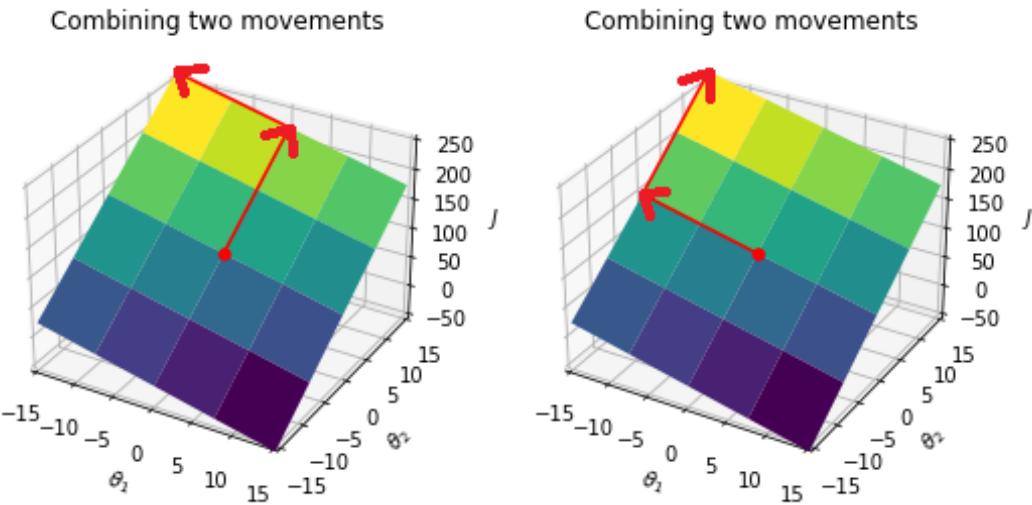
Movement in θ_1 on J



If we take very small steps, we can approximate our function as **flat**.

Why does this help? If our derivative doesn't **change**, you can take multiple steps Δv_i and the order doesn't matter.

So, you can combine your steps or separate them easily.



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:

$$\Delta s = \Delta s_{\text{from } v_1} + \Delta s_{\text{from } v_2} + \dots + \Delta s_{\text{from } v_m} \quad (\text{A.14})$$

A.2.3 Our completed scalar/vector derivative

From this, we can get an **approximated** version of the MV chain rule.

Definition 462

The **multivariable chain rule approximation** looks similar to the multivariable chain rule, but for finite changes Δx_i .

In 3-D, we get

$$\Delta f \approx \underbrace{\frac{\partial f}{\partial x} \Delta x}_{x \text{ component}} + \underbrace{\frac{\partial f}{\partial y} \Delta y}_{y \text{ component}} + \underbrace{\frac{\partial f}{\partial z} \Delta z}_{z \text{ component}}$$

In general, we have

$$\Delta f \approx \sum_{i=1}^m \underbrace{\frac{\partial f}{\partial x_i} \Delta x_i}_{x_i \text{ component}}$$

This function lets us add up the effect each component has on our output, using **derivatives**.

This gives us what we're looking for:

$$\Delta s \approx \sum_{i=1}^m \frac{\partial s}{\partial v_i} \Delta v_i \quad (\text{A.15})$$

If we circle back around to our original approximation:

$$\sum_{i=1}^m \frac{\partial s}{\partial v_i} \Delta v_i = \frac{\partial s}{\partial v} * \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix} \quad (\text{A.16})$$

When we look at the left side, we're multiplying pairs of components, and then adding them. That sounds similar to a **dot product**.

$$\sum_{i=1}^m \frac{\partial s}{\partial v_i} \Delta v_i = \begin{bmatrix} \frac{\partial s}{\partial v_1} \\ \frac{\partial s}{\partial v_2} \\ \vdots \\ \frac{\partial s}{\partial v_m} \end{bmatrix} \cdot \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix} \quad (\text{A.17})$$

This gives us our derivative: it contains all of the **element-wise** derivatives we need, and in a **useful** form!

Definition 463

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:

$$\underbrace{\Delta s}_{(1 \times 1)} = \underbrace{\left(\frac{\partial s}{\partial v} \right)^T}_{(1 \times m)} \underbrace{\Delta v}_{(m \times 1)} \quad (\text{A.18})$$

A.3 Derivative: Vector/Scalar

Now, we want to try the flipped version: we swap our vector and our scalar.

$$\frac{\partial(\text{Vector})}{\partial(\text{Scalar})} = \frac{\partial w}{\partial s} \quad (\text{A.19})$$

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**.

$$\Delta s \rightarrow [f] \rightarrow \Delta w \quad (\text{A.20})$$

Written explicitly, like before:

$$\Delta s \rightarrow \underbrace{\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix}}_{\Delta w} \quad (\text{A.21})$$

Note that we're using vector w instead of v this time: this will be helpful for our vector/vector derivative: we'll need two different symbols for "a vector".

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**:

$$\Delta w \approx \frac{\partial w}{\partial s} \star \Delta s \quad \text{or} \quad \underbrace{\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix}}_{\Delta w} \approx \frac{\partial w}{\partial s} \star \Delta s \quad (\text{A.22})$$

Here, we can't do a **dot product**: we're multiplying our derivative by a **scalar**.

A.3.1 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 one of these terms **individually**, treating it as a scalar derivative, we get:

$$\Delta w_i = \frac{\partial w_i}{\partial s} \Delta s \quad (\text{A.23})$$

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:

$$\mathbf{w} = \begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix} = \begin{bmatrix} \Delta s(\partial w_1 / \partial s) \\ \Delta s(\partial w_2 / \partial s) \\ \vdots \\ \Delta s(\partial w_n / \partial s) \end{bmatrix} \quad (\text{A.24})$$

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:

$$\Delta w = \left(\frac{\partial w}{\partial s} \right)^T \Delta s \quad (\text{A.25})$$

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.

$$\Delta w = \left[\frac{\partial w_1}{\partial s}, \frac{\partial w_2}{\partial s}, \dots, \frac{\partial w_n}{\partial s} \right]^T \Delta s \quad (\text{A.26})$$

Clarification 464

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!

$$\overbrace{\Delta w}^{(\text{orange} \times 1)} = \overbrace{\left(\frac{\partial w}{\partial s} \right)^T}^{(\text{orange} \times 1)} \overbrace{\Delta s}^{(1 \times 1)} \quad (\text{A.27})$$

As we can see, the dimensions check out.

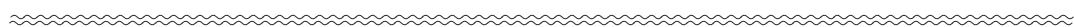
Definition 465

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

$$\frac{\partial w}{\partial s} = \left[\frac{\partial w_1}{\partial s}, \frac{\partial w_2}{\partial s}, \dots, \frac{\partial w_n}{\partial s} \right]$$



A.4 Derivative: Vector/Vector

We'll be combining our two previous derivatives:

$$\frac{\partial(\text{Vector})}{\partial(\text{Vector})} = \frac{\partial w}{\partial v} \quad (\text{A.28})$$

v and w are both **vectors**: thus, input and output are both **vectors**.

$$\Delta v \rightarrow [f] \rightarrow \Delta w \quad (\text{A.29})$$

Written out, we get:

$$\underbrace{\begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix}}_{\Delta v} \rightarrow \underbrace{\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix}}_{\Delta w} \quad (\text{A.30})$$

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!

A.4.1 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:

$$\Delta w = \left(\frac{\partial w}{\partial v} \right)^T \Delta v \quad (\text{A.31})$$

Hopefully, since we're combining two different derivatives, we should be able to use the same rule here.

With mn different elements, this could get messy very fast. Let's see if we can focus on only **part** of our problem:

$$\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix} = \left(\frac{\partial w}{\partial v} \right)^T \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix} \quad (\text{A.32})$$

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 .

$$\underbrace{\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix}}_{\Delta w \text{ from } v_i} = \left(\frac{\partial w}{\partial v_i} \right)^T \Delta v_i \quad (\text{A.33})$$

We now have a simpler case: $\partial \text{Vector} / \partial \text{Scalar}$. We're familiar with this case!

$$\frac{\partial w}{\partial v_i} = \left[\frac{\partial w_1}{\partial v_i}, \frac{\partial w_2}{\partial v_i}, \dots, \frac{\partial w_n}{\partial v_i} \right] \quad (\text{A.34})$$

We get a vector. What if the **output** is a scalar instead?

One output

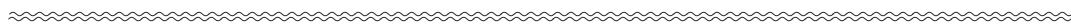
$$\Delta w_j = \left(\frac{\partial w_j}{\partial v} \right)^T \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix} \quad (\text{A.35})$$

We have $\partial \text{Scalar} / \partial \text{Vector}$:

$$\frac{\partial w_j}{\partial v} = \begin{bmatrix} \partial w_j / \partial v_1 \\ \partial w_j / \partial v_2 \\ \vdots \\ \partial w_j / \partial v_m \end{bmatrix} \quad (\text{A.36})$$

So, our vector-vector derivative is a **generalization** of the two derivatives we did before!

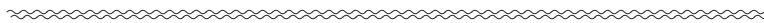
- It seems that extending along the **vertical** axis changes our v_i value, while moving along the **horizontal** axis changes our w_j value.



A.5 General derivative (Vector/Vector)

You might have a hint of what we get: one derivative stretches us along **one** axis, the other along the **second**.

But now, let's prove it to ourselves.



Our biggest problem is that we developed these tools for working with **vectors**, and now we have a **matrix**.

Rather than giving up this perspective, we'll instead take it further:

- We can think of a matrix as a "stack of vectors".

$$M = \begin{bmatrix} a_1 & b_1 & \cdots & z_1 \\ a_2 & b_2 & \cdots & z_2 \\ \vdots & \vdots & \ddots & \vdots \\ a_n & b_n & \cdots & z_n \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} & \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} & \cdots & \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} \end{bmatrix} \quad (\text{A.37})$$

- Or, a **row vector of column vectors**.

$$M = \begin{bmatrix} \vec{a} & \vec{b} & \cdots & \vec{z} \end{bmatrix} \quad (\text{A.38})$$

So, that's our plan: we first view our matrix as a row vector, hiding the column vectors inside.

- This requires treating our column vectors as if they were "scalars".

Then, we'll expand those column vectors, and see what we get.

Concept 466

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 pretend "**scalar**", has to match how the **vector** would behave.



This is **equivalent** to when just focused on one scalar inside our vector, and then stacked all those scalars back into the vector.

Here's how we apply this to our situation.

- First, we treat each **column** as a **scalar**, and the whole object as a **vector**.
 - This will use one familiar derivative.

- Then, we'll expand each **column** into a whole **vector**. Then, we have a **matrix**.
 - This will us our other derivative.

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.

$$\Delta w = \left(\frac{\partial w}{\partial v} \right)^T \Delta v \quad (\text{A.39})$$

We'll only expand Δw , because that's a simpler case we know how to manage.

$$\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix} = \left(\frac{\partial w}{\partial v} \right)^T \Delta v \quad (\text{A.40})$$

- Notice that we **didn't** simplify v to v_i . We aren't using only one element of v : we're pretending as if the whole vector v (including all elements) is a scalar.

We compute our derivative, based on earlier work:

$$\frac{\partial w}{\partial v} = \underbrace{\left[\frac{\partial w_1}{\partial v}, \frac{\partial w_2}{\partial v}, \dots, \frac{\partial w_n}{\partial v} \right]}_{\text{Column } j \text{ matches } w_j} \quad (\text{A.41})$$

- 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 : each is a **column vector**!

First, for w_1 :

$$\frac{\partial \mathbf{w}}{\partial \mathbf{v}} = \left[\begin{array}{c} \frac{\partial w_1}{\partial v_1} \\ \frac{\partial w_1}{\partial v_2} \\ \vdots \\ \frac{\partial w_1}{\partial v_m} \end{array}, \frac{\partial w_2}{\partial v_1}, \dots, \frac{\partial w_n}{\partial v_m} \right] \quad \text{Column } j \text{ matches } w_j \quad \text{Row } i \text{ matches } v_i \quad (\text{A.42})$$

And again, for w_2 :

$$\frac{\partial \mathbf{w}}{\partial \mathbf{v}} = \left[\begin{array}{c} \frac{\partial w_1}{\partial v_1} \\ \frac{\partial w_1}{\partial v_2} \\ \vdots \\ \frac{\partial w_1}{\partial v_m} \end{array}, \begin{array}{c} \frac{\partial w_2}{\partial v_1} \\ \frac{\partial w_2}{\partial v_2} \\ \vdots \\ \frac{\partial w_2}{\partial v_m} \end{array}, \dots, \frac{\partial w_n}{\partial v_m} \right] \quad \text{Column } j \text{ matches } w_j \quad \text{Row } i \text{ matches } v_i \quad (\text{A.43})$$

And again, for w_n :

$$\frac{\partial \mathbf{w}}{\partial \mathbf{v}} = \left[\begin{array}{c} \frac{\partial w_1}{\partial v_1} \\ \frac{\partial w_1}{\partial v_2} \\ \vdots \\ \frac{\partial w_1}{\partial v_m} \end{array}, \begin{array}{c} \frac{\partial w_2}{\partial v_1} \\ \frac{\partial w_2}{\partial v_2} \\ \vdots \\ \frac{\partial w_2}{\partial v_m} \end{array}, \dots, \begin{array}{c} \frac{\partial w_n}{\partial v_1} \\ \frac{\partial w_n}{\partial v_2} \\ \vdots \\ \frac{\partial w_n}{\partial v_m} \end{array} \right] \quad \text{Column } j \text{ matches } w_j \quad \text{Row } i \text{ matches } v_i \quad (\text{A.44})$$

We have column vectors in our row vector... based on our new perspective, this is basically the same as a **matrix**.

Definition 467

If

- v is an $(m \times 1)$ vector
- w is an $(n \times 1)$ vector

Then we define the **vector derivative** $\partial w / \partial v$ as fulfilling:

$$\Delta w = \left(\frac{\partial w}{\partial v} \right)^T \Delta v$$

Thus, our derivative must be a $(m \times n)$ vector

$$\frac{\partial w}{\partial v} = \left[\begin{array}{cccc} \frac{\partial w_1}{\partial v_1} & \frac{\partial w_2}{\partial v_1} & \cdots & \frac{\partial w_n}{\partial v_1} \\ \frac{\partial w_1}{\partial v_2} & \frac{\partial w_2}{\partial v_2} & \cdots & \frac{\partial w_n}{\partial v_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial w_1}{\partial v_m} & \frac{\partial w_2}{\partial v_m} & \cdots & \frac{\partial w_n}{\partial v_m} \end{array} \right] \quad \begin{array}{l} \text{Column } j \text{ matches } w_j \\ \text{Row } i \text{ matches } v_i \end{array}$$

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.

A.5.1 More about the vector/vector derivative

Let's show a specific example: w is (3×1) , v is (2×1) .

$$\frac{\partial w}{\partial v} = \left[\begin{array}{ccc} \overbrace{\frac{\partial w_1}{\partial v_1}}^{w_1} & \overbrace{\frac{\partial w_2}{\partial v_1}}^{w_2} & \overbrace{\frac{\partial w_3}{\partial v_1}}^{w_3} \\ \overbrace{\frac{\partial w_1}{\partial v_2}}^{w_1} & \overbrace{\frac{\partial w_2}{\partial v_2}}^{w_2} & \overbrace{\frac{\partial w_3}{\partial v_2}}^{w_3} \end{array} \right] \quad \begin{array}{l} \} v_1 \\ \} v_2 \end{array} \quad (\text{A.45})$$

Another way to describe the general case:

Notation 468

Our matrix $\frac{\partial \mathbf{w}}{\partial \mathbf{v}}$ is entirely filled with **scalar derivatives**

$$\frac{\partial w_j}{\partial v_i}$$

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 469

Our matrix $\frac{\partial \mathbf{w}}{\partial \mathbf{v}}$ can be written as

$$\frac{\partial \mathbf{w}}{\partial \mathbf{v}} = \left[\underbrace{\frac{\partial w_1}{\partial v_1}, \frac{\partial w_2}{\partial v_1}, \dots, \frac{\partial w_n}{\partial v_1}}_{\text{Column } j \text{ matches } w_j} \right]$$

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.

A.6 Derivative: matrix/scalar

Now, we have our general form for creating derivatives.

We'll get our derivative of the form

$$\frac{\partial(\text{Matrix})}{\partial(\text{Scalar})} = \frac{\partial \textcolor{blue}{M}}{\partial \textcolor{red}{s}} \quad (\text{A.46})$$

We have a matrix $\textcolor{blue}{M}$ in the shape $(r \times k)$ and a scalar $\textcolor{red}{s}$. Our **input** is a **scalar**, and our **output** is a **matrix**.

$$\textcolor{blue}{M} = \begin{bmatrix} \textcolor{blue}{m}_{11} & \textcolor{blue}{m}_{12} & \cdots & \textcolor{blue}{m}_{1k} \\ \textcolor{blue}{m}_{21} & \textcolor{blue}{m}_{22} & \cdots & \textcolor{blue}{m}_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ \textcolor{blue}{m}_{r1} & \textcolor{blue}{m}_{r2} & \cdots & \textcolor{blue}{m}_{rk} \end{bmatrix} \quad (\text{A.47})$$

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 470

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**.

$$\textcolor{blue}{M} = \begin{bmatrix} \textcolor{blue}{M}_1 \\ \textcolor{blue}{M}_2 \\ \vdots \\ \textcolor{blue}{M}_r \end{bmatrix} \quad (\text{A.48})$$

Now, we can pretend our matrix is a vector! We've got a derivative for that:

$$\frac{\partial \mathbf{M}}{\partial s} = \begin{bmatrix} \frac{\partial M_1}{\partial s} & \frac{\partial M_2}{\partial s} & \dots & \frac{\partial M_r}{\partial s} \end{bmatrix} \quad (\text{A.49})$$

Aha - we have the same form that we did for our vector/scalar derivative! Each derivative is a column vector. Let's expand it out:

$$\frac{\partial \mathbf{M}}{\partial s} = \left[\begin{array}{c} \left[\frac{\partial m_{11}}{\partial s}, \frac{\partial m_{21}}{\partial s}, \dots, \frac{\partial m_{r1}}{\partial s} \right] \\ \vdots \\ \left[\frac{\partial m_{1k}}{\partial s}, \frac{\partial m_{2k}}{\partial s}, \dots, \frac{\partial m_{rk}}{\partial s} \right] \end{array} \right], \quad \text{Column } j \text{ matches } m_j? \quad (\text{A.50})$$

Row i matches $m_{?i}$

Definition 471

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:

$$\frac{\partial \mathbf{M}}{\partial s} = \left[\begin{array}{cccc} \frac{\partial m_{11}}{\partial s} & \frac{\partial m_{21}}{\partial s} & \dots & \frac{\partial m_{r1}}{\partial s} \\ \frac{\partial m_{12}}{\partial s} & \frac{\partial m_{22}}{\partial s} & \dots & \frac{\partial m_{r2}}{\partial s} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial m_{1k}}{\partial s} & \frac{\partial m_{2k}}{\partial s} & \dots & \frac{\partial m_{rk}}{\partial s} \end{array} \right], \quad \text{Column } j \text{ matches } m_j? \quad (\text{A.51})$$

Row i matches $m_{?i}$

- This matrix has the shape of \mathbf{M}^T .

A.7 Derivative: scalar/matrix

We'll get our derivative of the form

$$\frac{\partial(\text{Scalar})}{\partial(\text{Matrix})} = \frac{\partial s}{\partial M} \quad (\text{A.51})$$

We have a matrix 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**.

$$M = \begin{bmatrix} M_1 \\ M_2 \\ \vdots \\ M_r \end{bmatrix} \quad (\text{A.52})$$

The gradient for this "vector" gives us a **column vector**:

$$\frac{\partial s}{\partial M} = \begin{bmatrix} \frac{\partial s}{\partial M_1} \\ \frac{\partial s}{\partial M_2} \\ \vdots \\ \frac{\partial s}{\partial M_r} \end{bmatrix} \quad (\text{A.53})$$

This time, each derivative is a **row vector**. Let's **expand**:

$$\frac{\partial s}{\partial M} = \begin{bmatrix} \left[\frac{\partial s}{\partial m_{11}} \quad \frac{\partial s}{\partial m_{12}} \quad \cdots \quad \frac{\partial s}{\partial m_{1k}} \right] \\ \left[\frac{\partial s}{\partial m_{21}} \quad \frac{\partial s}{\partial m_{22}} \quad \cdots \quad \frac{\partial s}{\partial m_{2k}} \right] \\ \vdots \\ \left[\frac{\partial s}{\partial m_{r1}} \quad \frac{\partial s}{\partial m_{r2}} \quad \cdots \quad \frac{\partial s}{\partial m_{rk}} \right] \end{bmatrix} \quad (\text{A.54})$$

Definition 472

If \mathbf{M} is a matrix in the shape $(r \times k)$ and s is a scalar,

Then we define the **matrix derivative** $\partial s / \partial \mathbf{M}$ as the $(r \times k)$ matrix:

$$\frac{\partial s}{\partial \mathbf{M}} = \left[\begin{array}{cccc} \frac{\partial s}{\partial m_{11}} & \frac{\partial s}{\partial m_{12}} & \cdots & \frac{\partial s}{\partial m_{1k}} \\ \frac{\partial s}{\partial m_{21}} & \frac{\partial s}{\partial m_{22}} & \cdots & \frac{\partial s}{\partial m_{2k}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial s}{\partial m_{r1}} & \frac{\partial s}{\partial m_{r2}} & \cdots & \frac{\partial s}{\partial m_{rk}} \end{array} \right] \quad \begin{array}{l} \text{Column } j \text{ matches } m_{?j} \\ \text{Row } i \text{ matches } m_i? \end{array}$$

This matrix has the same shape as \mathbf{M} .

A.8 Tensors

A.8.1 Other Derivatives

After these, you might ask yourself, what about other derivative combinations?

$$\frac{\partial \textcolor{blue}{v}}{\partial \textcolor{blue}{M}}? \quad \frac{\partial \textcolor{blue}{M}}{\partial \textcolor{blue}{v}}? \quad \frac{\partial \textcolor{blue}{M}}{\partial \textcolor{blue}{M}^2}?$$
 (A.55)

There's a problem with all of these: the total number of axes is **too large**.

What do we mean by an **axis**?

Definition 473

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

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{bmatrix} \left. \right\} \text{Axis 1}$$

~~~~~

- A **matrix** has **2 axes**: we can move **horizontally** or **vertically**.

$$\overbrace{\begin{bmatrix} m_{11} & m_{12} & \cdots & m_{1r} \\ m_{21} & m_{22} & \cdots & m_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ m_{k1} & m_{k2} & \cdots & m_{kr} \end{bmatrix}}^{\substack{\text{Axis 2: Columns} \\ \text{Axis 1: Rows}}} \left. \right\}$$

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**? We'll clarify this in an optional following section.

$$\frac{\partial \mathbf{w}}{\partial \mathbf{v}} = \left[ \begin{array}{cccc} \frac{\partial w_1}{\partial v_1} & \frac{\partial w_2}{\partial v_1} & \cdots & \frac{\partial w_n}{\partial v_1} \\ \frac{\partial w_1}{\partial v_2} & \frac{\partial w_2}{\partial v_2} & \cdots & \frac{\partial w_n}{\partial v_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial w_1}{\partial v_m} & \frac{\partial w_2}{\partial v_m} & \cdots & \frac{\partial w_n}{\partial v_m} \end{array} \right]$$

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:

- $\partial s / \partial v$ : we need one axis to represent each term  $v_i$ .
  - 0 axis + 1 axis  $\rightarrow$  1 axis: the output is a (column) **vector**.
- $\partial v / \partial s$ : we need one axis to represent each term  $w_j$ .
  - 1 axis + 0 axis  $\rightarrow$  1 axis: the output is a (row) **vector**.
- $\partial w / \partial v$ : 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**.
- $\partial M / \partial s$ : 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**.
- $\partial s / \partial M$ : 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 474**

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:

$$\text{Axes}\left(\frac{\partial \textcolor{violet}{y}}{\partial \textcolor{orange}{x}}\right) = \text{Axes}(\textcolor{violet}{y}) + \text{Axes}(\textcolor{orange}{x})$$

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.

---

### A.8.2 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 475

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**.

$$\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_k \end{bmatrix} \quad \left. \right\} \text{We can adjust each of our } k \text{ scalars.}$$

- 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.

In other words:

- Vector-style dimensionality: number of separate scalars in your vector.
- Matrix(tensor)-style dimensionality: number of separate indices we can use to find scalars.

If you jumped here from X.16, feel free to follow this [link](#) back. Otherwise, continue on.

### A.8.3 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 476

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 477

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**.

**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.

- We can imagine stacking the following two matrices in the third dimension, with the leftmost in front, and the rightmost in the back, in a  $(2 \times 3 \times 2)$  block:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad \begin{bmatrix} 5 & 7 & 11 \\ 13 & 17 & 19 \end{bmatrix} \quad (\text{A.56})$$

### Clarification 478

Note that what we call a tensor is **not** a mathematical (or physics) tensor.

- We don't make use of most of the crucial properties of a mathematical tensor.

Our tensor can be better thought of as a "**generalized matrix**", or a "multidimensional array".

This is important, because a "mathematical" tensor has properties that can be confusing from an ML perspective:

- The "tensor product" doesn't behave at all like a matrix product.
- Rather, the generalized version of the **matrix product** is something called **tensor contraction**.

Tensor contraction examples, like the einsum function in numpy ("einstein summation notation"), can get very complex for higher-dimensional tensors.

If tensors don't really behave quite like matrices, and their math is complex, 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, in this matrix, we make a vector out of the diagonal:

$$\begin{bmatrix} 3 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 3 \\ 9 \\ 4 \end{bmatrix} \quad (\text{A.57})$$

These examples aren't especially important, but you can see different variations used for different problems!

- We can also flatten it into a matrix or vector by **stacking** layers next to each other in the same dimension.

- For example, we could stack the two columns of a matrix:

This version is used when we are willing to give up our n-D structure: we lose a lot of information.

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 1 \\ 3 \\ 2 \\ 4 \end{bmatrix} \quad (\text{A.58})$$

- We cleverly "**hide**" dimensions of a matrix, as we have before: treat a scalar as a vector, etc.

This works best for high-level, conceptual stuff. We'll use it below.

- We **systematically** multiply and add our elements in a way that gives us the derivatives we want, replicating matrix multiplication.
  - This is **tensor contraction**.
  - This method is often used by softwares to implement the chain rule.

#### Clarification 479

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: shape, structure, etc.

These techniques are part of why: there are **different** approaches for how to approach these objects.

The solution tends to be directly computing the chain rule, rather than figuring out the structure of the abstract object.

As we will see in the next chapter, tensors are **very** important to machine learning.

However, because they're so troublesome to work with, we'll convert to matrices in most cases.

---

## A.9 Chapter 7 Derivatives

### A.9.1 The loss derivative

Finally, we apply all of our new knowledge to the common derivatives in section 7.5.

$$\overbrace{\frac{\partial \mathcal{L}}{\partial A^L}}^{(n^L \times 1)} \quad (\text{A.59})$$

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  $A^L$ .

#### Notation 480

Our derivative

$$\frac{\partial \mathcal{L}}{\partial A^L}$$

Is a scalar/vector derivative, and thus the shape  $(n^L \times 1)$ .

### A.9.2 The weight derivative

$$\overbrace{\frac{\partial \mathcal{L}}{\partial W^\ell}}^{(m^\ell \times 1) ?} \quad (\text{A.60})$$

This derivative is difficult - it's a derivative in the form vector/matrix. With **three** axes, a 3-tensor seems suitable.

But, our goal is to use this for the **chain rule**: so, we need to make matrix shapes that **match**.

$$\frac{\partial \mathcal{L}}{\partial W^\ell} = \underbrace{\frac{\partial \mathcal{L}}{\partial Z^\ell}}_{\text{Weight link}} \cdot \underbrace{\left( \frac{\partial \mathcal{L}}{\partial Z^\ell} \right)^\top}_{\text{Other layers}} \quad (\text{A.61})$$

Remember that we had to do weird reordering and transposing in chapter 7 to flip the order of the chain rule? This is why: shape consistency.

Our problem is we have **too many axes**: the easiest way to resolve this to **break up** our matrix.

$$W = [W_1 \ W_2 \ \dots \ W_n] \quad (\text{A.62})$$

Notice that, this time, we broke  $W$  into **column vectors**  $W_i$ , rather than row vectors. Each **neuron**'s weights are represented by one column vector.

So, for now, we focus on only **one neuron** at a time: it has a column vector  $W_i$ . We'll ignore everything except  $W_i$ .

For simplicity, we're gonna ignore the  $\ell$  notation: just be careful, because  $Z$  and  $A$  are from two different layers!

$$W_i = \begin{bmatrix} w_{1i} \\ w_{2i} \\ \vdots \\ w_{mi} \end{bmatrix} \quad (\text{A.63})$$

### Concept 481

When you have a derivative that has **too many dimensions**, it's easier to only focus on **one** of the elements/dimensions, and find the derivative of that first.

- In this example, rather than finding  $\frac{\partial Z}{\partial W}$ , we're finding  $\frac{\partial Z}{\partial W_i}$ .
- We're simplifying from **matrix**  $W$  to **vector**  $W_i$ .

~~~~~  
So, we're doing $\frac{\partial Z}{\partial W_i}$: we need equations relating these variables.

- W_i represents the weights of one neuron, while Z represents the pre-activation of all the neurons.

Let's focus on one pre-activation neuron at a time.

$$z_j = W_j^T A \quad (\text{A.64})$$

Notably, this equation reminds us that pre-activation z_j is only affected by weights from the same neuron, W_j .

- Weights from a different neuron have no effect on z_j .
- So, the derivative between them will be 0: we can check this using the equation above.

Concept 482

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 neuron** k .
- 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 out of the way, let's actually **expand** our expression from above, to compute $\frac{\partial z_i}{\partial W_{ji}}$.

$$z_i = W_i^T A \quad (\text{A.65})$$

$$z_i = \begin{bmatrix} w_{1i} & w_{2i} & \cdots & w_{mi} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \quad (\text{A.66})$$

This matrix multiplication can be written as an easier-to-use **sum**:

$$z_i = \sum_{j=1}^n W_{ji} a_j \quad (\text{A.67})$$

Finally, we can get our derivatives, for the **non-zero** terms:

$$\frac{\partial z_i}{\partial W_{ji}} = a_j \quad (\text{A.68})$$

Now, we can combine them into a vector.

$$\frac{\partial z_i}{\partial W_i} = \begin{bmatrix} \frac{\partial z_i}{\partial W_{1i}} \\ \frac{\partial z_i}{\partial W_{2i}} \\ \vdots \\ \frac{\partial z_i}{\partial W_{mi}} \end{bmatrix} \quad (\text{A.69})$$

We plug in our new values:

$$\frac{\partial z_i}{\partial W_i} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} = A \quad (\text{A.70})$$

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 from one neuron don't affect different neurons.

$$\frac{\partial z_i}{\partial W_{jk}} = 0 \quad \text{if } i \neq k \quad (\text{A.71})$$

We can combine these into a **zero vector**:

$$\frac{\partial z_i}{\partial W_k} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \vec{0} \quad \text{if } i \neq k \quad (\text{A.72})$$

So, now, we can describe all of our vector components:

$$\frac{\partial z_i}{\partial W_k} = \begin{cases} A & \text{if } i = k \\ \vec{0} & \text{if } i \neq k \end{cases}$$

(A.73)

Again, we see: "same neuron? they're related. Different neurons? They're not."

~~~~~

This derivative, despite being a vector, can be represented with a **single** symbol ($\vec{0}$ or A), for each element $\frac{\partial z_i}{\partial W_k}$.

- For the sake of trying to wrangle our 3-tensor, we'll "hide" one dimension using this fact.
- If this is the case, then we have to pretend that W_k is a "scalar": so, W is now a "vector".

We compute $\frac{\partial Z}{\partial W}$ as a "vector/vector" derivative: this is a **matrix**!

$$\frac{\partial Z}{\partial W} = \begin{bmatrix} A & \vec{0} & \cdots & \vec{0} \\ \vec{0} & A & \cdots & \vec{0} \\ \vdots & \vdots & \ddots & \vec{0} \\ \vec{0} & \vec{0} & \vec{0} & A \end{bmatrix} \quad (\text{A.74})$$

We have our result: it turns out, despite being stored in a **matrix**-like format, this is actually a **3-tensor**! Each "scalar" of our **matrix** is a **vector**: we really have 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 483

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.

$$\begin{bmatrix} A & \vec{0} & \cdots & \vec{0} \\ \vec{0} & A & \cdots & \vec{0} \\ \vdots & \vdots & \ddots & \vec{0} \\ \vec{0} & \vec{0} & \vec{0} & A \end{bmatrix} \longrightarrow \begin{bmatrix} A \\ A \\ \vdots \\ A \end{bmatrix} \quad (\text{A.75})$$

Then, we'll combine all of our redundant  $A$  values.

$$\begin{bmatrix} \textcolor{orange}{A} \\ \textcolor{orange}{A} \\ \vdots \\ \textcolor{orange}{A} \end{bmatrix} \longrightarrow \textcolor{orange}{A} \quad (\text{A.76})$$

We have our final answer!

#### Notation 484

Our derivative

$$\underbrace{\frac{\partial \textcolor{red}{Z}^\ell}{\partial \textcolor{blue}{W}^\ell}}_{(\textcolor{blue}{m}^\ell \times 1)} = \textcolor{orange}{A}^{\ell-1}$$

Is a vector/matrix derivative, and thus should be a 3-tensor.

But, we have turned it into the shape  $(\textcolor{blue}{m}^\ell \times 1)$ .

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 multiplication order of the chain rule, etc.)

However, at the end, we get the right shape for our chain rule!

This is the weird order change we mentioned in chapter 7, where we reverse the order of elements.

### A.9.3 Linking Layers $\ell - 1$ and $\ell$

$$\frac{\partial \textcolor{red}{Z}^\ell}{\partial \textcolor{orange}{A}^{\ell-1}} \quad (\text{A.77})$$

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  $\ell$ , as before.

$$\textcolor{red}{Z} = \textcolor{blue}{W}^T \textcolor{orange}{A} \quad (\text{A.78})$$

We'll use the same approach we did last section:  $\textcolor{blue}{W}$  is a "vector", and we'll focus on  $\textcolor{blue}{W}_i$ . This will allow us to break it up **element-wise**.

- We could treat  $\textcolor{blue}{W}$  as a whole matrix, but our approach will be cleaner: otherwise, we'd have to depict every  $\textcolor{blue}{W}_i$  at **once**.

$$W = \begin{bmatrix} W_1 & W_2 & \dots & W_n \end{bmatrix} \quad W_i = \begin{bmatrix} w_{1i} \\ w_{2i} \\ \vdots \\ w_{mi} \end{bmatrix} \quad (\text{A.79})$$

Here's our equation:

$$z_i = \begin{bmatrix} w_{1i} & w_{2i} & \dots & w_{mi} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \quad (\text{A.80})$$

We matrix multiply:

$$z_i = \sum_{j=1}^n W_{ji} a_j \quad (\text{A.81})$$

The derivative can be gotten from here -

$$\frac{\partial z_i}{\partial a_j} = W_{ji} \quad (\text{A.82})$$

We look at our whole matrix derivative:

$$\frac{\partial Z}{\partial A} = \left\{ \begin{array}{c} \text{Column } i \text{ matches } z_i \\ \left[ \begin{array}{ccc} \ddots & \vdots & \ddots \\ \cdots & \frac{\partial z_i}{\partial a_j} & \cdots \\ \ddots & \vdots & \ddots \end{array} \right] \end{array} \right\} \text{Row } j \text{ matches } a_j \quad (\text{A.83})$$

This notation looks a bit weird, but it's just a way to represent that all of our elements follow the same pattern.

Wait.

- The derivative  $\frac{\partial z_i}{\partial a_j}$  is in the  $j^{\text{th}}$  row,  $i^{\text{th}}$  column of  $\frac{\partial Z}{\partial A}$ .
- $W_{ji}$  is the element of  $W$  in the  $j^{\text{th}}$  row,  $i^{\text{th}}$  column.

$W_{ji}$  goes in the same place in both matrices! They're the same matrix:  $W = \frac{\partial Z}{\partial A}$

We get our final result:

If two matrices have exactly the same shape and elements, they're the same matrix.

**Notation 485**

Our derivative

$$\underbrace{\frac{\partial \mathbf{Z}^\ell}{\partial \mathbf{A}^{\ell-1}}}_{(m^\ell \times n^\ell)} = \mathbf{W}^\ell$$

Is a vector/vector derivative, and thus a matrix.

It takes the shape  $(m^\ell \times n^\ell)$ .

- This matches the intuition we have from the simplified, 1d version, where we have  $z = w a$ , instead of  $Z = W^T A$ .

**A.9.4 Activation Function**

$$\frac{\partial \mathbf{A}^\ell}{\partial \mathbf{Z}^\ell} \quad (\text{A.84})$$

The last derivative is less strange than its solution looks.

$$\mathbf{A}^\ell = f(\mathbf{Z}^\ell) \longrightarrow \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = f\left( \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} \right) \quad (\text{A.85})$$

We can apply our function element-wise:

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} f(z_1) \\ f(z_2) \\ \vdots \\ f(z_n) \end{bmatrix} \quad (\text{A.86})$$

As we can see, each activation is a function of only **one** pre-activation.

**Concept 486**

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

$$\frac{\partial a_i}{\partial z_i}$$

As for our **non-zero** terms, they all rely on the equation:

$$a_i = f(z_i) \tag{A.87}$$

Our derivative is:

$$\frac{\partial a_i}{\partial z_i} = f'(z_i) \tag{A.88}$$

In general, including the non-diagonals:

$$\frac{\partial a_i}{\partial z_j} = \begin{cases} f'(z_i) & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{A.89}$$

This gives us our result:

**Notation 487**

Our derivative

$$\underbrace{\frac{\partial \mathbf{A}^\ell}{\partial \mathbf{Z}^\ell}}_{(n^\ell \times n^\ell)} = \left[ \begin{array}{ccccc} f'(z_1^\ell) & 0 & 0 & \cdots & 0 \\ 0 & f'(z_2^\ell) & 0 & \cdots & 0 \\ 0 & 0 & f'(z_3^\ell) & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & f'(z_n^\ell) \end{array} \right] \quad \begin{array}{l} \text{Column } j \text{ matches } a_j \\ \text{Row } i \text{ matches } z_i \end{array}$$

Is a vector/vector derivative, and thus a matrix.

It takes the shape  $(n^\ell \times n^\ell)$ .

**A.9.5 Element-wise multiplication**

Notice that, in the previous section, we could've compressed this matrix down to remove the unnecessary 0's:

$$\begin{bmatrix} f'(z_1^\ell) \\ f'(z_2^\ell) \\ \vdots \\ f'(z_n^\ell) \end{bmatrix} \quad (\text{A.90})$$

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 488**

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:

$$RQ = \underbrace{R * Q}_{\text{Element-wise multiplication}} = \begin{bmatrix} r_1 q_1 \\ r_2 q_2 \\ r_3 q_3 \\ \vdots \\ r_n q_n \end{bmatrix}$$

So, we can substitute the chain rule this way.

## A.10 Terms

- Matrix/scalar derivative
- Scalar/Matrix derivative
- Axis
- Dimension (vector)
- Dimension (array)
- Array
- "Tensor" (Generalized matrix)
- $c$ -Tensor (2-tensor, 3-tensor, etc.)