Explanatory Notes for 6.3900

Shaunticlair Ruiz

Fall 2024

Contents

CHAPTER 6

Neural Networks 2 - Back-Propagation and Training

6.5 Error back-propagation

We have a complete neural network: a **model** we can use to make predictions or calculations.

Now, our mission is to **improve** this neural network: even if our hypothesis class is good, we still have to **find** the hypotheses that are useful for our problem.

As usual, we will start out with **randomized** values for our weights and biases: this **initial** neural network will not be useful for anything in particular, but that's why we need to improve it.

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.

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 1

Neural networks are typically optimized using **gradient descent**.

6.5.1 Review: Gradient Descent

What does it really mean to do gradient descent on our **network**? Let's remind ourselves of how gradient descent works, and then **build** up to a network.

Concept 2

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: $\frac{\partial B}{\partial A}$ tells you the direction to **change** A in order to **increase** B.
- We want to **change** θ to **decrease** $\mathcal{L}$. Thus, we move in the direction of

$$\Delta\theta = -\eta \frac{\partial \mathcal{L}}{\partial \theta}$$

- We take steps $\Delta\theta$ (and $\Delta\theta_0$) until we are satisfied with $\mathcal{L}$, or it **stops** improving.

Remember that η is our **step size**: we can take bigger or smaller steps in each direction.

6.5.2 Review: Gradient Descent with LLCs

Let's start with a familiar example: **LLCs**.

Our LLC model uses the following equations:

We'll use w instead of θ.

$$z(x) = w^T x + w_0 \qquad g(z) = \sigma(z) = \frac{1}{1+e^{-z}} \tag{6.1}$$

$$\mathcal{L}(g, y) = y \log(g) + (1-y)\log(1-g) \tag{6.2}$$

Our goal is to minimize $\mathcal{L}$ by adjusting θ and θ_0.

So, we want

$$\frac{\partial \mathcal{L}}{\partial w} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial w_0} \tag{6.3}$$

We did this by using the **chain rule**:

We'll focus on w, but the same goes for w_0.

$$\frac{\partial \mathcal{L}}{\partial w} = \overbrace{\frac{\partial \mathcal{L}}{\partial g} \cdot \frac{\partial g}{\partial w}}^{\mathcal{L}(g)} \tag{6.4}$$

We can break it up further using **repeated** chain rules:

$$\frac{\partial \mathcal{L}}{\partial w} = \overbrace{\frac{\partial \mathcal{L}}{\partial g} \cdot \underbrace{\frac{\partial g}{\partial z} \cdot \frac{\partial z}{\partial w}}_{g(z)}}^{\mathcal{L}(g)} \tag{6.5}$$

Plugging in our derivatives, we get:

$$\frac{\partial \mathcal{L}}{\partial w} = \overbrace{-\left(\frac{y}{\sigma} - \frac{1-y}{1-\sigma}\right)}^{\partial \mathcal{L}/\partial g} \cdot \overbrace{\sigma(1-\sigma)}^{\partial g/\partial z} \cdot \overbrace{x}^{\partial z/\partial w} \tag{6.6}$$

Concept 3

The **chain rule** allows us to take the gradient of **nested functions**, where each function is the **input** to the next one.

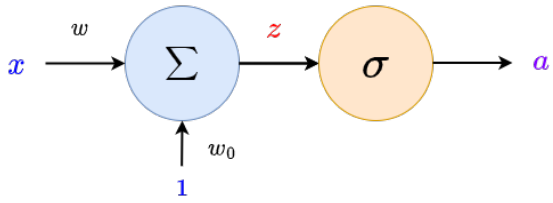
Another way to say this is that one function **feeds into** the next.

If you aren't familiar with "nested" functions, consider this example:

If you have functions $f(x)$ and $g(x)$, then $g(f(x))$ is the **nested** combination, where the output of f is the input of g.

6.5.3 Review: LLC as Neuron

Remember that we can represent our LLC as a **neuron**: this could give us the first idea for how to train our **neural network**!



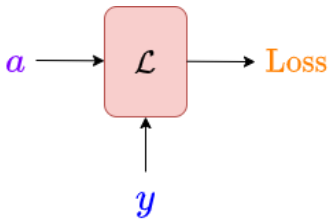
As usual, our first unit $\sum$ is our **linear** component. The output is z, nothing different from before with LLC.

Remember that x is a whole vector of values, which we've condensed into one variable.

The **output** of σ, which we wrote before as g, is now a.

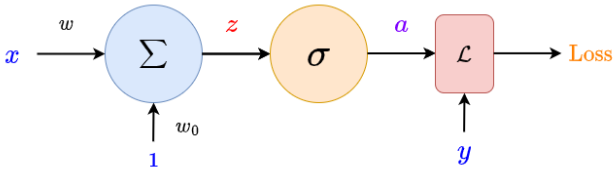
Something we neglected before: this diagram is **missing** the **loss function**. Let's create a small unit for that.

$\mathcal{L}(a, y)$ has **two** inputs: our predicted value a, and the correct value y.



We have two inputs to our loss function.

We **combine** these into a single unit to get:



Our full unit!

6.5.4 LLC Forward-Pass

Now, we can do gradient descent like before. We want to get the effect our **weight** has on our **loss**.

But, this time, we'll pair it with a **visual** that is helpful for understanding how we **train** neural networks.

First, one important consideration:

As we saw above, the **gradient** we get might rely on z, a, or $\mathcal{L}(a, y)$. So, before we do anything, we have to **compute** these values.

Each step **depends** on the last: this is what the **forward** arrows represent. We call this a **forward pass** on our neural network.

Definition 4

A **forward pass** of a neural network is the process of sending information "**forward**" through the neural network, starting from the **input**.

This means the **input** is fed into the **first** layer, and that output is fed into the **next** layer, and so on, until we reach our **final** result and **loss**.

Example: If we had

- $f(x) = x + 2$

- $g(f) = 3f$
- $h(g) = \sin(g)$

Then, a forward pass with the input $x = 10$ would have us go function-by-function:

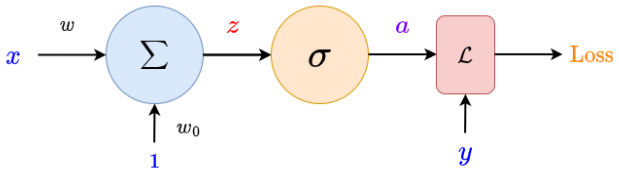
- $f(10) = 10 + 2$
- $g(f) = 3 \cdot 12$
- $h(g) = \sin(36)$

So, by "forward", we mean that we apply each function, one after another.

In our case, this means computing z, a, and $\mathcal{L}(a, y)$.

6.5.5 LLC Back-propagation

Now that we have all of our values, we can get our gradient. Let's **visualize** this process.



We want to link $\mathcal{L}$ to w. In order to do that, we need to **connect** each thing in between.

- This lets us **combine** lots of simple **links** to get our more complicated result.

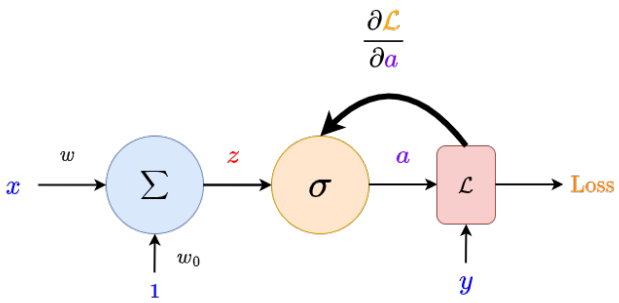
We can also call this "chaining together" lots of derivatives.

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.

$$\overbrace{\frac{\partial \mathcal{L}}{\partial a}}^{\text{Loss unit}} \tag{6.7}$$

We send this information backwards, so it can be used later.

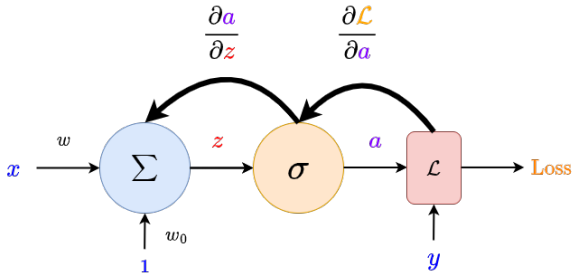


Now, we're on the $\sigma(z)$ unit.

- 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.

$$\overbrace{\frac{\partial \mathcal{L}}{\partial a}}^{\text{Loss unit}} \cdot \overbrace{\frac{\partial a}{\partial z}}^{\text{Activation function}} \tag{6.8}$$

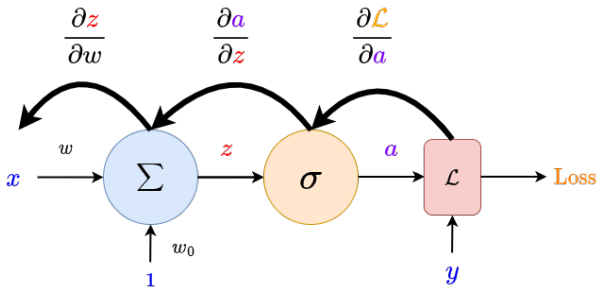
We haven't reached w yet, so we send this information further back.



Finally, we reach $\sum$.

- The $\sum$ 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.

$$\frac{\partial \mathcal{L}}{\partial w} = \overbrace{\frac{\partial \mathcal{L}}{\partial a}}^{\text{Loss unit}} \cdot \overbrace{\frac{\partial a}{\partial z}}^{\text{Activation}} \cdot \overbrace{\frac{\partial z}{\partial w}}^{\text{Linear subunit}} \tag{6.9}$$

Moving backwards like this is called **back-propagation**.

Definition 5

Back-propagation is the process of moving "**backwards**" through your network, starting at the **loss** and moving back layer-by-layer, and gathering terms in your **chain rule**.

We call it "**propagation**" because we send backwards the **terms** of our chain rule about later derivatives.

An **earlier** unit (closer to the "left") has all of the **derivatives** that come after (to the "right" of) it, along with its own term.

6.5.6 Summary of neural network gradient descent: a high-level view

So, with just this, we have built up the basic idea of how we **train** our model: now that we have the gradient, we can do **gradient descent** like we normally do!

This summary covers some things we haven't fully discussed. We'll continue digging into the topic!

Concept 6

We can do **gradient descent** on a **neural network** using the ideas we've built up:

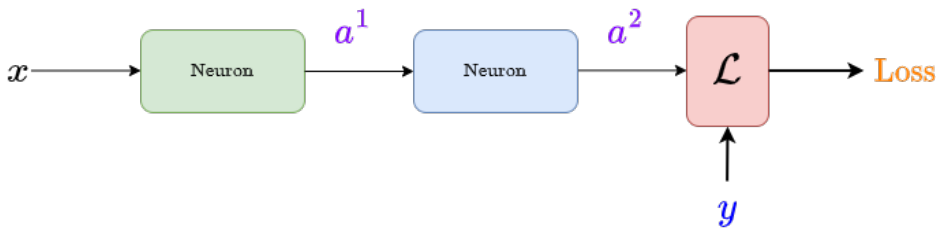
- Do a **forward pass**, where we compute the value of each **unit** in our model, passing the information **forward** - each layer's **output** is the next layer's **input**.
  - We finish by getting the **loss**.
- Do **back-propagation**: build up a **chain rule**, starting at the **loss** function, 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**.

6.5.7 A two-neuron network: starting backprop

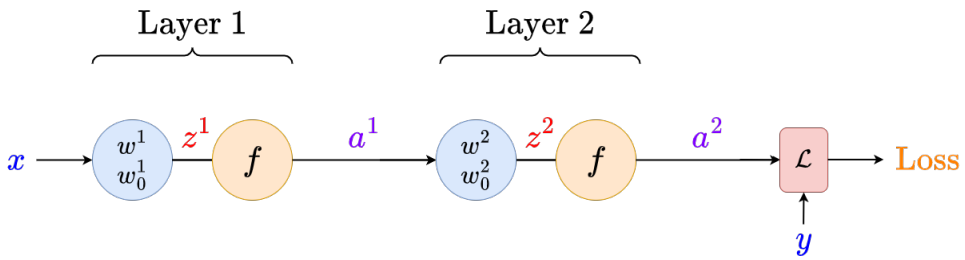
Above, we mention "**each** layer": we'll now transition to a **two-neuron** system, so we have "two layers". Then, we'll build up to many layers.

Remember, though, that the **ideas** represented here are just extensions of what we did **above**.

Let's get a look at our **two-neuron** system, now with our **loss** unit:



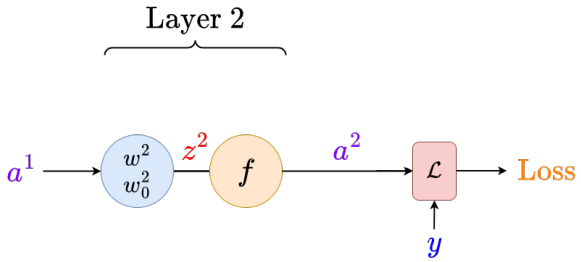
And unpack it:



We want to do **back-propagation** like we did before. This time, we have **two** different layers of weights: w^1 and w^2. Does this cause any problems?

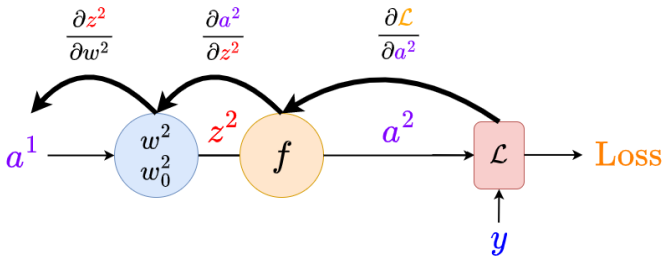
It turns out, it doesn't! We mentioned in the first part of chapter 7 that we can treat the **output** of the **first** layer a^1 as the same as if it were an **input** x.

This is one of the biggest benefits of neural network layers!



Now, we can do backprop safely.

"Backprop" is a common shortening of "back-propagation".



We can get:

$$\frac{\partial \mathcal{L}}{\partial w^2} = \overbrace{\frac{\partial \mathcal{L}}{\partial a^2}}^{\text{Loss unit}} \cdot \overbrace{\frac{\partial a^2}{\partial z^2}}^{\text{Activation}} \cdot \overbrace{\frac{\partial z^2}{\partial w^2}}^{\text{Linear}} \tag{6.10}$$

The same format as for our **one-neuron** system! We now have a gradient we can update for our **second** weight vector.

But what about our **first** weight vector?

6.5.8 Continuing backprop: One more problem

We need to continue further to reach our **earlier** weights: this is why we have to work **backward**.

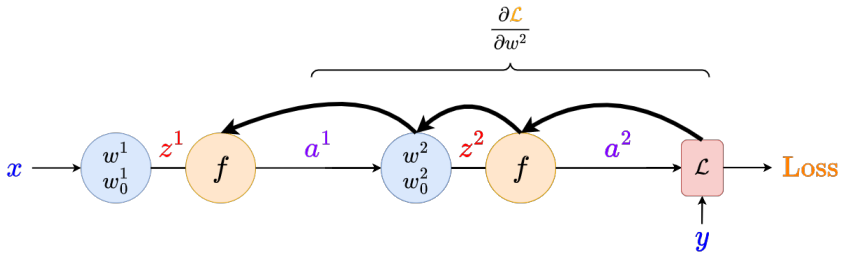
Concept 7

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:

$$\frac{\partial w^2}{\partial a^1} \tag{6.11}$$

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.

The problem is, what is w^2? It's a vector of constants.

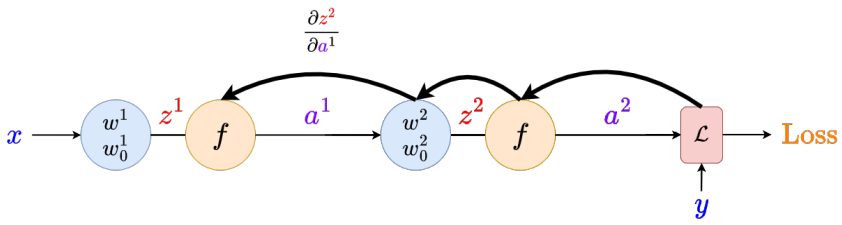
$$w^2 = \begin{bmatrix} w_1^2 \\ w_2^2 \\ \vdots \\ w_n^2 \end{bmatrix}, \qquad \text{Not a function of } a^1! \tag{6.12}$$

That derivative above is going to be **zero**! In other words, w^2 isn't really the **input** to z^2: it's a **parameter**.

So, we can't end our derivative with w^2. Instead, we have to use something else. z^2's real input is a^1, so let's go directly to that!

We were building our chain rule by combining inputs with outputs: that's what links two layers together.

So, it should make sense that using something like w (that doesn't link two layers) prevents us from making a longer chain rule.



Using this allows us to move from layer 2 to layer 1.

Now, we have our new chain rule:

$$\frac{\partial \mathcal{L}}{\partial a^1} = \overbrace{\frac{\partial \mathcal{L}}{\partial a^2} \cdot \frac{\partial a^2}{\partial z^2}}^{\text{Other terms}} \cdot \overbrace{\frac{\partial z^2}{\partial a^1}}^{\text{Link Layers}} \tag{6.13}$$

Concept 8

For our **weight gradient** in layer l, we have to end our **chain rule** with

$$\frac{\partial z^\ell}{\partial w^\ell}$$

So we can get

$$\frac{\partial \mathcal{L}}{\partial w^\ell} = \overbrace{\frac{\partial \mathcal{L}}{\partial z^\ell}}^{\text{Other terms}} \cdot \overbrace{\frac{\partial z^\ell}{\partial w^\ell}}^{\text{Get weight grad}}$$

However, because w^l is not the **input** of layer l, we can't use it to find the gradient of **earlier layers**.

Instead, we use

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} \tag{6.14}$$

To "**link together**" two different layers ℓ and $\ell - 1$ in a **chain rule**.

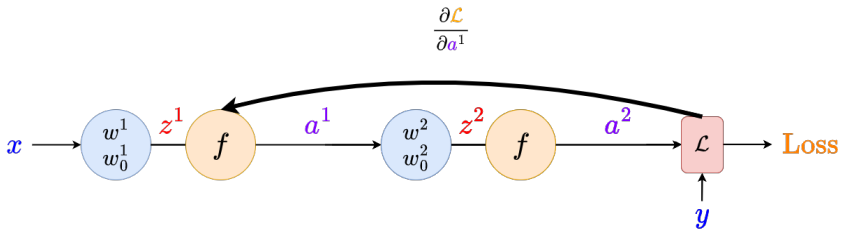
In this section, we compressed lots of derivatives into

$$\frac{\partial \mathcal{L}}{\partial z^\ell}$$

Don't let this alarm you, this just hides our long chain of derivatives!

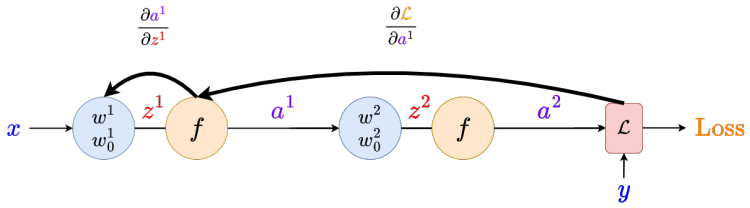
6.5.9 Finishing two-neuron backprop

Now that we have safely connected our layers, we can do the rest of our gradient. First, let's lump together everything we did before:

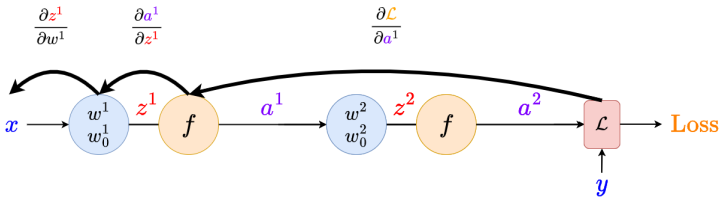


All the info we need is stored in this derivative: it can be written out using our friendly chain rule from earlier.

Now, we can add our remaining terms. It's the same as before: we want to look at the pre-activation



And finally, our input:



We can get our second chain rule

$$\frac{\partial \mathcal{L}}{\partial w^1} = \overbrace{\frac{\partial \mathcal{L}}{\partial a^1}}^{\text{Other layers}} \cdot \overbrace{\frac{\partial a^1}{\partial z^1} \cdot \frac{\partial z^1}{\partial w^1}}^{\text{Layer 1}} \tag{6.15}$$

Which, in reality, looks much bigger:

$$\frac{\partial \mathcal{L}}{\partial w^1} = \overbrace{\left(\frac{\partial \mathcal{L}}{\partial a^2}\right)}^{\text{Loss unit}} \cdot \overbrace{\left(\frac{\partial a^2}{\partial z^2} \cdot \frac{\partial z^2}{\partial a^1}\right)}^{\text{Layer 2}} \cdot \overbrace{\left(\frac{\partial a^1}{\partial z^1} \cdot \frac{\partial z^1}{\partial w^1}\right)}^{\text{Layer 1}} \tag{6.16}$$

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 9

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

$$\overbrace{\frac{\partial a^\ell}{\partial z^\ell}}^{\text{Within layer}} \cdot \overbrace{\frac{\partial z^\ell}{\partial w^\ell}}^{\text{Get weight grad}}$$

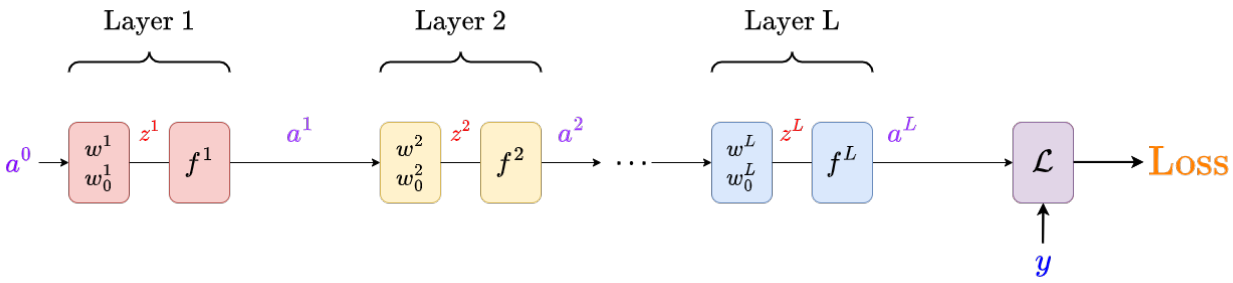
If we want to **extend** to the previous layer, we **instead** multiply by

$$\overbrace{\frac{\partial a^\ell}{\partial z^\ell}}^{\text{Within layer}} \cdot \overbrace{\frac{\partial z^\ell}{\partial a^{\ell-1}}}^{\text{Link layers}}$$

6.5.10 Many layers: Doing back-propagation

Now, we'll consider the case of many possible layers.

To make it more readable, we'll use boxes instead of circles for units.



This may look intimidating, but we already have all the tools we need to handle this problem.

Our goal is to get a **gradient** for each of our **weight** vectors w^ℓ, so we can do gradient descent and **improve** our model.

According to our above analysis in Concept 9, we need only a few steps to get all of our gradients.

Concept 10

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:

$$\overbrace{\frac{\partial \mathcal{L}}{\partial a^{L}}}^{\text{Loss unit}}$$

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

$$\overbrace{\frac{\partial a^{\ell}}{\partial z^{\ell}}}^{\text{Within layer}} \cdot \overbrace{\frac{\partial z^{\ell}}{\partial w^{\ell}}}^{\text{Get weight grad}}$$

  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

$$\overbrace{\frac{\partial a^{\ell}}{\partial z^{\ell}}}^{\text{Within layer}} \cdot \overbrace{\frac{\partial z^{\ell}}{\partial a^{\ell-1}}}^{\text{Link layers}}$$

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:

$$\frac{\partial \mathcal{L}}{\partial w^{\ell}} = \overbrace{\left(\frac{\partial \mathcal{L}}{\partial a^{L}}\right)}^{\text{Loss unit}} \cdot \overbrace{\left(\frac{\partial a^{L}}{\partial z^{L}} \cdot \frac{\partial z^{L}}{\partial a^{L-1}}\right)}^{\text{Layer L}} \cdot \overbrace{\left(\frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^{L-1}}{\partial a^{L-2}}\right)}^{\text{Layer L}-1} \cdot \left(\cdots\right) \cdot \overbrace{\left(\frac{\partial a^{\ell}}{\partial z^{\ell}} \cdot \frac{\partial z^{\ell}}{\partial w^{\ell}}\right)}^{\text{Layer } \ell} \tag{6.17}$$

That's pretty ugly. If we need to hide the complexity, we can:

Notation 11

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*.

6.5.11 What do these derivatives equal?

Let's look at each of these derivatives and see if we can't simplify them a bit.

First, every gradient needs

- The **loss derivative**:

$$\frac{\partial \mathcal{L}}{\partial a^L} \tag{6.18}$$

  This **depends** on 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:

$$\frac{\partial a^\ell}{\partial z^\ell} \tag{6.19}$$

  What does the function between these **look** like?

$$a = f(z) \tag{6.20}$$

  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.

$$\frac{\partial a^\ell}{\partial z^\ell} = \overbrace{\left(f^\ell\right)'}^{\text{deriv of func for layer } \ell} \overbrace{(z^\ell)}^{\text{Deriv input}} \tag{6.21}$$

  This expression is a bit visually clunky, but it works. Without the annotation:

$$\frac{\partial a^\ell}{\partial z^\ell} = \left(f^\ell\right)'(z^\ell) \tag{6.22}$$

z^ℓ is not being multiplied by $(f^\ell)'$, it's the input to that derivative.

Between layers, we have

- We can also think about the derivative of the **linear function** that **connects two layers**:

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} \tag{6.23}$$

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.

So, we want the function of these two:

$$z^\ell = w^\ell a^{\ell-1} + w_0^\ell \tag{6.24}$$

This one is pretty simple! We just take the derivative manually:

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} = w^\ell \tag{6.25}$$

Finally, every gradient will end with...

- The derivative that directly connects to a **weight**, again using the **linear function**:

$$\frac{\partial z^\ell}{\partial w^\ell} \tag{6.26}$$

The linear function is the same:

$$z^\ell = w^\ell a^{\ell-1} + w_0^\ell \tag{6.27}$$

But with a different **variable**, the **derivative** comes out different:

$$\frac{\partial z^\ell}{\partial w^\ell} = a^{\ell-1} \tag{6.28}$$

Notation 12

Our **derivatives** for the **chain rule** in a **1-D neural network** take the form:

$$\frac{\partial \mathcal{L}}{\partial a^L}$$

$$\frac{\partial a^\ell}{\partial z^\ell} = (f^\ell)'(z^\ell)$$

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} = w^\ell$$

$$\frac{\partial z^\ell}{\partial w^\ell} = a^{\ell-1}$$

Now, we can rewrite our generalized expression for gradient:

$$\frac{\partial \mathcal{L}}{\partial w^\ell} = \overbrace{\left(\frac{\partial \mathcal{L}}{\partial a^L}\right)}^{\text{Loss unit}} \cdot \overbrace{\left((f^L)'(z^L) \cdot w^L\right)}^{\text{Layer L}} \cdot \overbrace{\left((f^{L-1})'(z^{L-1}) \cdot w^{L-1}\right)}^{\text{Layer L}-1} \cdot \left(\cdots\right) \cdot \overbrace{\left((f^\ell)'(z^\ell) \cdot a^{\ell-1}\right)}^{\text{Layer } \ell} \tag{6.29}$$

Our expressions are more concrete now. It's still pretty visually messy, though.

6.5.12 Activation Derivatives

We weren't able to **simplify** our expressions above, partly because we didn't know which **loss** or **activation** function we were going to use.

So, here, we will look at the **common** choices for these functions, and **catalog** what their derivatives look like.

- **Step function** $\text{step}(z)$:

$$\frac{d}{dz}\text{step}(z) = 0 \tag{6.30}$$

This is part of why we don't use this function: it has no gradient. We can show this by looking piecewise:

$$\text{step}(z) = \begin{cases} 1 & \text{if } z \geqslant 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{6.31}$$

And take the derivative of each piece:

$$\frac{d}{dz}\text{ReLU}(z) = 0 = \begin{cases} 0 & \text{if } z \geqslant 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{6.32}$$

- **Rectified Linear Unit** ReLU(z):

$$\frac{d}{dz}\text{ReLU}(z) = \text{step}(z) \tag{6.33}$$

This one might be a bit surprising at first, but it makes sense if you **also** break it up into cases:

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z \geqslant 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{6.34}$$

And take the derivative of each piece:

$$\frac{d}{dz}\text{ReLU}(z) = \text{step}(z) = \begin{cases} 1 & \text{if } z \geqslant 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{6.35}$$

- **Sigmoid** function $\sigma(z)$:

$$\frac{d}{dz}\sigma(z) = \sigma(z)(1 - \sigma(z)) = \frac{e^{-z}}{(1 + e^{-z})^2} \tag{6.36}$$

This derivative is useful for simplifying NLL, and has a nice form.

We can just compute the derivative with the single-variable chain rule.

As a reminder, the function looks like:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{6.37}$$

- **Identity** ("linear") function $f(z) = z$:

$$\frac{d}{dz}z = 1 \tag{6.38}$$

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 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 $\tanh(z)$:

$$\frac{d}{dz}\tanh(z) = 1 - \tanh(z)^2 \tag{6.39}$$

  This strange little expression is 1 minus the "hyperbolic secant" squared. We won't bother further with it.

Notation 13

For our various **activation** functions, we have the **derivatives**:

Step:

$$\frac{d}{dz}\text{step}(z) = 0$$

ReLU:

$$\frac{d}{dz}\text{ReLU}(z) = \text{step}(z)$$

Sigmoid:

$$\frac{d}{dz}\sigma(z) = \sigma(z)(1 - \sigma(z))$$

Identity/Linear:

$$\frac{d}{dz}z = 1$$

6.5.13 Loss derivatives

Now, we look at the loss derivatives.

- **Square loss** function $\mathcal{L}_{sq} = (a - y)^2$:

$$\frac{d}{da}\mathcal{L}_{sq} = 2(a - y) \tag{6.40}$$

  Follows from chain rule+power rule, used for regression.

- **Linear loss** function $\mathcal{L}_{sq} = |a - y|$:

$$\frac{d}{da}\mathcal{L}_{lin} = \text{sign}(a - y) \tag{6.41}$$

This one can also be handled piecewise, like step(z) and ReLU(z):

$$|u| = \begin{cases} u & \text{if } z \geqslant 0 \\ -u & \text{if } z < 0 \end{cases} \tag{6.42}$$

We take the piecewise derivative:

$$\frac{d}{du}|u| = \text{sign}(u) = \begin{cases} 1 & \text{if } z \geqslant 0 \\ -1 & \text{if } z < 0 \end{cases} \tag{6.43}$$

- **NLL** (Negative-Log Likelihood) function $\mathcal{L}_{\text{NLL}} = -(y \log(a) + (1-y)\log(1-a))$

$$\frac{d}{da}\mathcal{L}_{\text{NLL}} = -\left(\frac{y}{a} - \frac{1-y}{1-a}\right) \tag{6.44}$$

- **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 14

For our various **loss** functions, we have the **derivatives**:

Square:

$$\frac{d}{da}\mathcal{L}_{sq} = 2(a-y)$$

Linear (Absolute):

$$\frac{d}{da}\mathcal{L}_{lin} = \text{sign}(a-y)$$

NLL (Negative-Log Likelihood):

$$\frac{d}{da}\mathcal{L}_{\text{NLL}} = -\left(\frac{y}{a} - \frac{1-y}{1-a}\right)$$

6.5.14 Many neurons per layer

Now, we just have left the elephant in the room: what do we do about the case where we have *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.

6.5.15 The chain rule: Matrix form

Let's start with the first: the punchline, how does the chain rule and our gradient descent **change** when we add **matrices**?

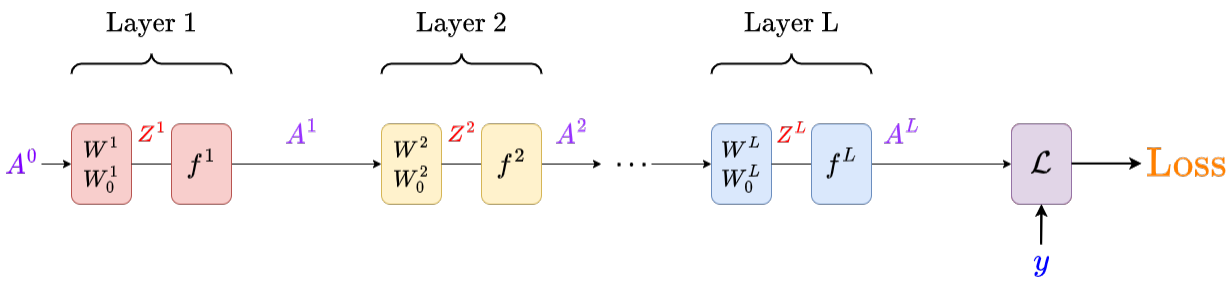
It turns out, not much: by using **layers** in the last section, we were able to create a pretty powerful and mathematically **tidy** object.

- With layers, each layer feeds into the **next**, with no other interaction. And neurons **within** the same layer do **not** directly **interact** with each other, which simplifies our math greatly.
  - Basically, we have a bunch of functions (neurons) that, within a layer, have **nothing** to do with each other, and only **output** to the **next** layer of similar functions.
- So, we can often **oversimplify** our model by thinking of each **layer** as like a "big" function, taking in a vector of size m^ℓ and outputting a vector of size n^ℓ.

Our main concern is making sure we have agreement of **dimensions**!

In fact, if you just rearranging your matrices and transposing them can be a helpful way to debug. Be careful, though!

So, here's how our model looks now:



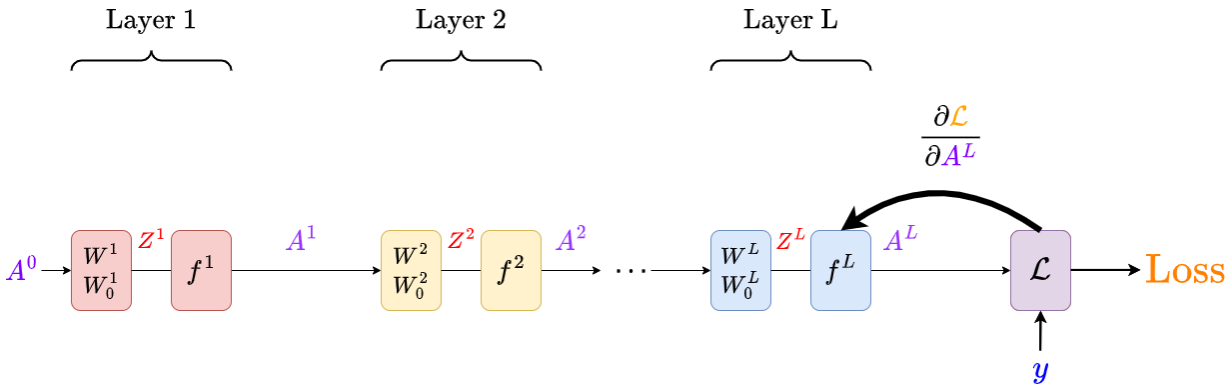
Pretty much the same! Only major difference: swapped scalars for vectors, and vectors for matrices (represented by switching to uppercase)

And, we do backprop the same way, too.

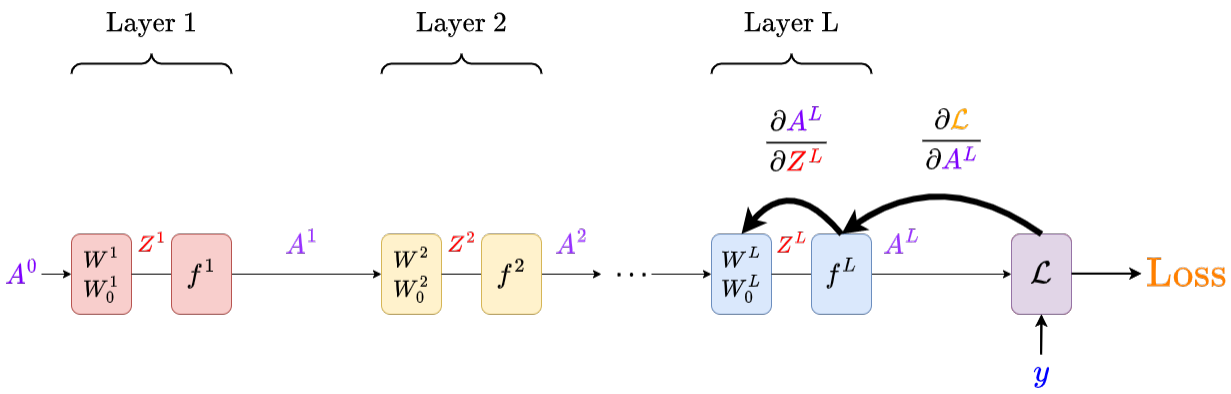
Here, we're not going to explain much as we go: all we're doing is getting the **derivatives** we need for our **chain rule**!

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:

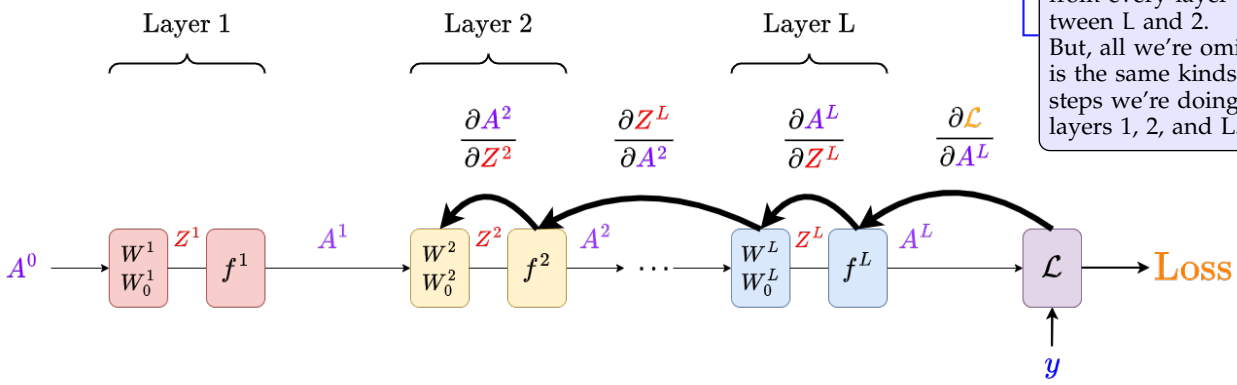


Take another step:

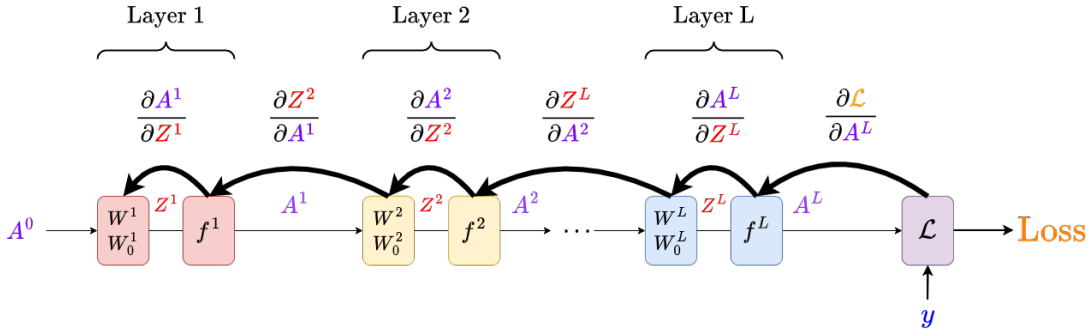


We'll pick up the pace: we'll jump to layer 2 and get its gradient.

The term $\partial Z^L / \partial A^2$ 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!



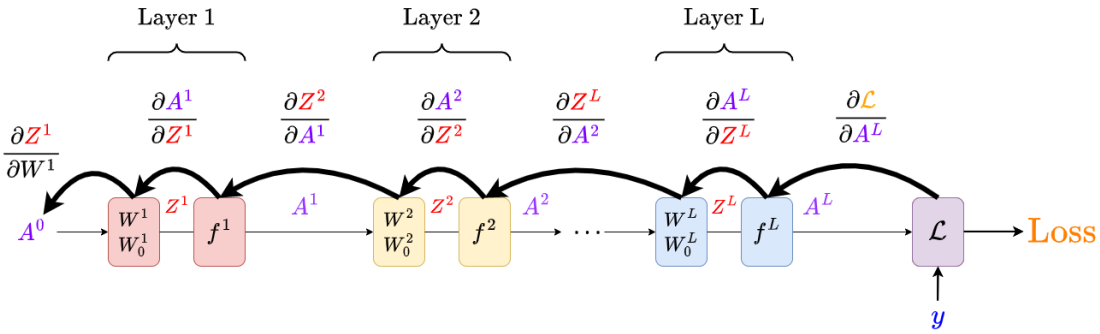
We finish off by getting what we're after: the gradient for W^1.

Notation 15

We depict neural network gradient descent using the below diagram (outside the box):

The **right**-facing **straight** arrows come **first**: they're part of the **forward pass**, where we get all of our values.

The **left**-facing **curved** arrows come **after**: they represent the **back-propagation** of the gradient.



And, with this, we can rewrite our general equation for neural network gradients.

6.5.16 How the Chain Rule changes in Matrix form

As we discussed before, we can't just add onto our weight gradient to reach another layer: the final term

$$\frac{\partial Z^\ell}{\partial W^\ell} \tag{6.45}$$

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:

$$\frac{\partial \mathcal{L}}{\partial W^\ell} = \overbrace{\frac{\partial Z^\ell}{\partial W^\ell}}^{\text{Weight link}} \cdot \overbrace{\left(\frac{\partial \mathcal{L}}{\partial Z^\ell}\right)^T}^{\text{Other layers}}$$

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 16

The **gradient** $\nabla_{W^\ell}\mathcal{L}$ for a neural network is given as:

$$\frac{\partial \mathcal{L}}{\partial W^\ell} = \overbrace{\frac{\partial Z^\ell}{\partial W^\ell}}^{\text{Weight link}} \cdot \overbrace{\left(\frac{\partial \mathcal{L}}{\partial Z^\ell}\right)^T}^{\text{Other layers}}$$

We get our remaining terms $\partial\mathcal{L}/\partial Z^\ell$ by our usual chain rule:

$$\frac{\partial \mathcal{L}}{\partial Z^\ell} = \overbrace{\left(\frac{\partial A^\ell}{\partial Z^\ell}\right)}^{\text{Layer } \ell} \cdot \left(\cdots\right) \cdot \overbrace{\left(\frac{\partial Z^{L-1}}{\partial A^{L-2}} \cdot \frac{\partial A^{L-1}}{\partial Z^{L-1}}\right)}^{\text{Layer } L-1} \cdot \overbrace{\left(\frac{\partial Z^L}{\partial A^{L-1}} \cdot \frac{\partial A^L}{\partial Z^L}\right)}^{\text{Layer } L} \cdot \overbrace{\left(\frac{\partial \mathcal{L}}{\partial A^L}\right)}^{\text{Loss unit}}$$

This is likely our most important equation in this chapter!

6.5.17 Relevant Derivatives

If you aren't interesting in understanding matrix derivatives, here we provide the general format of each of the derivatives we care about.

Notation 17

Here, we give useful **derivatives** for **neural network gradient descent**.

Loss is not given, so we can't compute it, as before:

$$\overbrace{\frac{\partial \mathcal{L}}{\partial A^L}}^{(n^L \times 1)}$$

We get the same result for each of these terms as we did before, except in matrix form.

$$\overbrace{\frac{\partial Z^\ell}{\partial W^\ell}}^{(m^\ell \times 1)} = A^{\ell-1}$$

$$\overbrace{\frac{\partial Z^\ell}{\partial A^{\ell-1}}}^{(m^\ell \times n^\ell)} = W^\ell$$

The last one is actually pretty different from before:

$$\overbrace{\frac{\partial A^\ell}{\partial Z^\ell}}^{(n^\ell \times n^\ell)} = \begin{bmatrix} f'(z_1^\ell) & 0 & 0 & \cdots & 0 \\ 0 & f'(z_2^\ell) & 0 & \cdots & 0 \\ 0 & 0 & f'(z_3^\ell) & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & f'(z_r^\ell) \end{bmatrix}$$

Where r is the length of Z^ℓ.

- In short, we only have the z_i derivative on the i^{th} diagonal
- Why? Check the **matrix derivative explanatory notes**.

For $\partial Z^\ell / \partial W^\ell$, check section A.9.2.

For $\partial Z^\ell / \partial A^{\ell-1}$, check section A.9.3.

For $\partial a^\ell / \partial z^\ell$, check section A.9.4.

Example: Suppose you have the activation $f(z) = z^2$.

Your pre-activation might be

$$z^\ell = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \tag{6.46}$$

The output would be

$$a^\ell = f(z^\ell) = \begin{bmatrix} 1 \\ 2^2 \\ 3^2 \end{bmatrix} \tag{6.47}$$

But the derivative would be:

$$f(z) = 2z \tag{6.48}$$

Which, gives our matrix derivative as:

$$\frac{\partial a^\ell}{\partial z^\ell} = \begin{bmatrix} f'(1) & 0 & 0 \\ 0 & f'(2) & 0 \\ 0 & 0 & f'(3) \end{bmatrix} = \begin{bmatrix} 2\cdot 1 & 0 & 0 \\ 0 & 2\cdot 2 & 0 \\ 0 & 0 & 2\cdot 3 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 6 \end{bmatrix}$$

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!

$$\frac{\partial w}{\partial v} = \overbrace{\begin{bmatrix} \frac{\partial w_1}{\partial v_1} & \frac{\partial w_2}{\partial v_1} & \cdots & \frac{\partial w_n}{\partial v_1} \\ \frac{\partial w_1}{\partial v_2} & \frac{\partial w_2}{\partial v_2} & \cdots & \frac{\partial w_n}{\partial v_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial w_1}{\partial v_m} & \frac{\partial w_2}{\partial v_m} & \cdots & \frac{\partial w_n}{\partial v_m} \end{bmatrix}}^{\text{Column j matches } w_j} \left.\vphantom{\begin{bmatrix} a \\ a \\ a \\ a \end{bmatrix}}\right\} \text{Row i matches } v_i \tag{6.49}$$

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 18

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.

6.6 Training

6.6.1 Comments

A few important side notes on training. First, on derivatives:

Concept 19

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: intialization.

Concept 20

We typically try to pick a **random initalization**. 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.

If you do not know a gaussian distribution, that shouldn't be a problem. It is also known as a "normal" distribution.

6.6.2 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.

SGD-NEURAL-NET($\mathcal{D}_n$, T, L, $(m^1, \ldots, m^L)$, $(f^1, \ldots, f^L)$, Loss)

for every **layer**:
    Randomly initialize
        the **weights** in every layer
        the **biases** in every layer

While **termination condition** not met:
    Get random data point i
    Keep track of time t

    Do forward pass
        for every **layer**:
            Use previous **layer**'s **output**: get **pre-activation**
            Use **pre-activation**: get new output, **activation**

        Get **loss**: forward pass complete

    Do back-propagation
        for every **layer** in reversed order:
            If **final** layer: #Loss function
                Get $\partial \mathcal{L} / \partial A^L$

            Else:
                Get $\partial \mathcal{L} / \partial A^\ell$: #Link two layers
                    $(\partial Z^{\ell+1} / \partial A^\ell)$ @ $(\partial \mathcal{L} / \partial Z^{\ell+1})$

                Get $\partial \mathcal{L} / \partial Z^\ell$: #Within layer
                    $(\partial A^\ell / \partial Z^\ell)$ @ $(\partial \mathcal{L} / \partial A^\ell)$

            Compute weight gradients:
                Get $\partial \mathcal{L} / \partial W^\ell$: #Weights
                    $\partial Z^\ell / \partial W^\ell = A^{\ell-1}$
                    $(\partial Z^\ell / \partial W^\ell)$ @ $(\partial \mathcal{L} / \partial Z^\ell)$

                Get $\partial \mathcal{L} / \partial {W_0}^\ell$: #Biases
                    $\partial \mathcal{L} / \partial {W_0}^\ell = (\partial \mathcal{L} / \partial Z^\ell)$

            Follow Stochastic Gradient Descend (SGD): #Take step
                Update **weights**:
                    $W^\ell = W^\ell - \left(\eta(t) * (\partial \mathcal{L} / \partial W^\ell)\right)$

                Update **biases**:
                    ${W_0}^\ell = {W_0}^\ell - \left(\eta(t) * (\partial \mathcal{L} / \partial {W_0}^\ell)\right)$

Return final neural network with weights and biases

6.7 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.

6.7.1 Mini-batch

6.7.1.1 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.

$$\overbrace{\nabla_W J}^{\text{General Gradient}} \longrightarrow \overbrace{\nabla_W J(W_t)}^{\text{Gradient at time } t} \tag{6.50}$$

Our update rule is:

$$W_{\text{new}} = W_{\text{old}} - \eta \overbrace{\left(\nabla_W J(W_{\text{old}})\right)}^{\text{Gradient}} \tag{6.51}$$

Or, using timestep t:

$$W_{t+1} = W_t - \eta \overbrace{\left(\nabla_W J(W_t)\right)}^{\text{Gradient}} \tag{6.52}$$

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:

We won't define J here, because it is slightly different for SGD and BGD. We'll get to that below.

$$\mathcal{L}^{(i)} = \overbrace{\mathcal{L}(g^{(i)}, y^{(i)})}^{\text{Loss for data point } i} \tag{6.53}$$

Our guess $g^{(i)}$ depends on both our current data point $x^{(i)}$, and the current weights W_t:

$$\mathcal{L}^{(i)}(W_t) = \mathcal{L}(\overbrace{h(x^{(i)}; W_t)}^{g^{(i)}}, y^{(i)}) \tag{6.54}$$

6.7.1.2 Review: BGD vs. SGD

Let's review our two main types of gradient descent, using the equation

$$W_{t+1} = W_t - \eta \overbrace{\left(\nabla_W J(W_t)\right)}^{\text{Gradient}} \tag{6.55}$$

First, we have **batch gradient descent**, where we use our **whole** training set each time we take a step.

Definition 21

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**.

$$\nabla_W J(W_t) = \sum_{i=1}^{n} \overbrace{\nabla_W \left(\mathcal{L}^{(i)}(W_t)\right)}^{\text{Each data point}}$$

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 22

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**.

$$\nabla_W J(W_t) = \overbrace{\nabla_W \left(\mathcal{L}^{(i)}(W_t)\right)}^{\text{One data point}}$$

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 23

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 overfit.
  - **BGD** uses all of the data, so we don't reduce overfitting.

6.7.1.3 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 24

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.

$$\nabla_W J(W_t) = \overbrace{\sum_{i=1}^{K}}^{\text{K data points in a mini-batch}} \nabla_W \left(\mathcal{L}^{(i)}(W_t) \right)$$

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.

We do have to be careful to randomly select data in an efficient way, though. Packages usually take care of this.

Concept 25

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.

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.

6.7.2 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.

In Appendix B, we discuss some common techniques:

- Momentum
- Adadelta
- Adam

6.7.3 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**".

By "deep", we just mean "many layers".

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.

$$\frac{\partial A}{\partial H} = \frac{\partial A}{\partial B} \cdot \frac{\partial B}{\partial C} \cdot \left(\cdots \right) \cdot \frac{\partial G}{\partial H} \tag{6.56}$$

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**?

$$\frac{\partial A}{\partial H} = .1 \cdot .1 \cdot \left(\cdots \right) \cdot .1 = 10^{-8} \tag{6.57}$$

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 26

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.

$$\frac{\partial A}{\partial H} = 10 \cdot 10 \cdot \left(\cdots \right) \cdot 10 = 10^{8} \tag{6.58}$$

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 27

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 28

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.

6.8 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.

6.8.1 Methods related to ridge regression

We'll start with methods that we can bring back from classic ridge regression.

6.8.1.1 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 29

An **epoch** is the time frame during which your model sees your whole **training data** set, **once**.

- 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.

Definition 30

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.

6.8.1.2 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:

$$J_{loss} = \sum_{i=1}^{n} \mathcal{L}(NN(x^{(i)}),\ y^{(i)};\ W) \tag{6.59}$$

- And we penalize based on the square magnitude of our weights:

$$J = \lambda \|W\|^2 + J_{loss} \tag{6.60}$$

If we take the gradient, we get:

$$\nabla_W J = 2\lambda W + \nabla_W J_{loss} \tag{6.61}$$

Let's see how the regularization affects our step:

$$W_t = W_{t-1} - \eta\left(2\lambda \|W_{t-1}\| + \nabla_W J_{loss}\right) \tag{6.62}$$

It directly subtracts from our weight, **decaying** it.

$$W_t = W_{t-1}\left(1 - 2\lambda\eta\right) - \eta\nabla_W J_{loss} \tag{6.63}$$

Thus, we call it **weight decay**.

Concept 31

When we apply **square magnitude** regularization to the weights of a neural network, we call it **weight decay**.

$$J_{reg} = J_{loss} + \lambda\|W\|^2$$

That's because, when you take the gradient, it directly subtracts from weight W, causing it to **decay** by a factor of $(1 - 2\lambda\eta)$.

$$W_t = W_{t-1}\left(1 - 2\lambda\eta\right) - \eta\nabla_W J_{loss}$$

6.8.1.3 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.

The "normal distribution" contains more information than that, but the symmetry is important.

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.

Definition 32

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.

6.8.2 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.

$$a_j^\ell = 0 \tag{6.64}$$

- 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 33

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.

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.

Clarification 34

When a network using dropout 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 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**.

6.8.3 Batch Normalization

6.8.3.1 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^1_{old} to a^1_{new}.

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^1_{old}!

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 35

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.

6.8.3.2 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:

This is exactly the same as when we standardized in the Feature Transformation chapter.

- **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".

We repeat this process in between each layer of our network. Each layer receives a set of inputs with mean 0, standard deviation 1, no matter how the earlier layers change.

Below, we exclude the ℓ superscript in z^ℓ, μ^ℓ, σ^ℓ, and n^ℓ for readability.

For now, we'll apply this to the pre-activation Z.

Notation 36

We'll represent the element in the i^{th} row, j^{th} column, as Z_{ij}.

Key Equation 37

*Here, we focus on a single element: dimension i of the j^{th} data point in our batch, z_{ij}.

When we **standardize**, we first compute the **mean** and **standard deviation**:

$$\mu_i = \frac{1}{K}\sum_{j=1}^{K} Z_{ij} \qquad \sigma_i^2 = \frac{1}{K}\sum_{j=1}^{K}(Z_{ij} - \mu_i)^2$$

Once we've done that, we can properly standardize, creating data with mean 0, and sd 1.

- We include a very small ϵ term to avoid dividing by zero.

$$\overline{Z}_{ij} = \frac{Z_{ij} - \mu_i}{\sigma_i + \epsilon}$$

Concept 38

In order to deal with **internal covariate shift**, we'll take each mini-batch of k **pre-activations** to each **layer** of our neural network, and **standardize**/normalize it.

- Each **dimension** of the input is standardized **separately**.

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

$$[-0.458, -0.433, 2.04, -0.408, -0.383, -0.358,] \tag{6.65}$$

The larger data point still stands out above the rest!

We need to be careful of dimensions:

Clarification 39

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^ℓ.

k is our **batch size**, while n is the number of **dimensions**.

- Z^ℓ without norm: $(n^\ell \times 1)$
- Z^ℓ with norm: $(n^\ell \times k)$

We use Z^ℓ_{ij} to indicate the i^{th} dimension of the j^{th} data point.

6.8.3.3 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 40

By doing **normalization**, we've transformed Z into $\overline{Z}$.

- 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 $\overline{Z}_i$, giving our **standard deviations**.
- B: An $(n \times 1)$ vector that **shifts** each of our dimensions $\overline{Z}_i$, giving our **means**.

Thus, we get the true output of **batch normalization**:

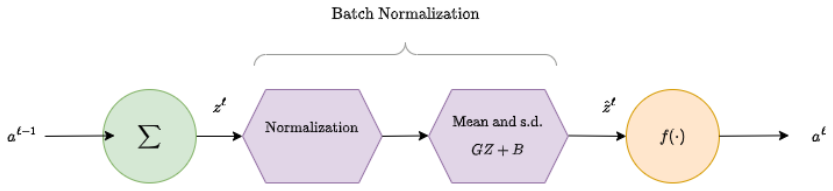
$$\widehat{Z}_{ik} = G_i * \overline{Z}_{ij} + B_i$$

Example: Here's a sample example using a vector $\overline{z}_j$: only considering one, post-normalization data point j.

$$\begin{bmatrix} \widehat{Z}_{1j} \\ \widehat{Z}_{2j} \\ \vdots \\ \widehat{Z}_{kj} \end{bmatrix} = \begin{bmatrix} G_1 \\ G_2 \\ \vdots \\ G_k \end{bmatrix} * \begin{bmatrix} \overline{Z}_{1j} \\ \overline{Z}_{2j} \\ \vdots \\ \overline{Z}_{kj} \end{bmatrix} + \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_k \end{bmatrix} \tag{6.66}$$

Where $*$ indicates element-wise multiplication.

If we include this in our neuron graph, we now have two new modules:



6.8.3.4 Full definition

Definition 41

Batch Normalization is a process where we

- Standardize the pre-activation for each layer using the mean μ_i and standard deviation σ_i (for the i^{th} dimension). Select ϵ: $0 < \epsilon << 1$.

$$\overline{Z}_{ij} = \frac{Z_{ij} - \mu_i}{\sigma_i + \epsilon}$$

- Choose the new mean and standard deviation for the pre-activation using $(n \times 1)$ vectors G and B

$$\widehat{Z}_{ik} = G_i * \overline{Z}_{ij} + B_i$$

Concept 42

Batch Normalization 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.

6.8.3.5 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.

6.8.3.6 Applying batch normalization to backprop

We defer discussion of backprop to Appendix B.

6.9 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^{ℓ}
- Step function
- ReLU function
- Sigmoid function
- Hyperbolic tangent function
- Softmax function

Section 7-2

- 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
- Mini-batch
- Vanishing/Exploding Gradient
- Momentum (Optional)
- Adadelta (Optional)
- Adagrad (Optional)
- Adam (Optional)
- Normalization
- Early stopping (Review)
- Weight Decay
- Perturbation
- Dropout
- Covariate Shift
- Internal Covariate Shift
- Batch Normalization

- Multivariable Chain Rule (Review)