Explanatory Notes for 6.390

Shaunticlair Ruiz (Current TA)

Fall 2022

Contents

CHAPTER 6

Neural Networks 1.5 - Back-Propagation and Training

6.5 Error back-propagation

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

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

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

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

$$\Delta\theta = -\eta \frac{\partial \mathcal{L}}{\partial \theta} \tag{6.1}$$

  - Remember that η is our **step size**: we can take bigger or smaller steps in each direction.
- We take steps $\Delta\theta$ (and $\Delta\theta_0$) until we are satisfied with $\mathcal{L}$, or it **stops** improving.

6.5.2 Review: Gradient Descent with LLCs

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

Our LLC model uses the following equations:

We'll use w instead of θ.

$$z(x) = w^T x + w_0 \qquad g(z) = \sigma(z) = \frac{1}{1+e^{-z}} \tag{6.2}$$

$$\mathcal{L}(g, y) = y \log(g) + (1-y)\log(1-g) \tag{6.3}$$

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

So, we want

$$\frac{\partial \mathcal{L}}{\partial w} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial w_0} \tag{6.4}$$

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

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

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

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

$$\frac{\partial \mathcal{L}}{\partial w} = \overbrace{\frac{\partial \mathcal{L}}{\partial g} \cdot \underbrace{\frac{\partial g}{\partial z}}^{\mathcal{L}(g)} \cdot \frac{\partial z}{\partial w}}_{g(z)} \tag{6.6}$$

Plugging in our derivatives, we get:

$$\frac{\partial \mathcal{L}}{\partial w} = \overbrace{-\left(\frac{y}{\sigma} - \frac{1-y}{1-\sigma}\right)}^{\partial \mathcal{L}/\partial g} \cdot \overbrace{\sigma(1-\sigma)}^{\partial g/\partial z} \cdot \overbrace{x}^{\partial z/\partial w} \tag{6.7}$$

Concept 3

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

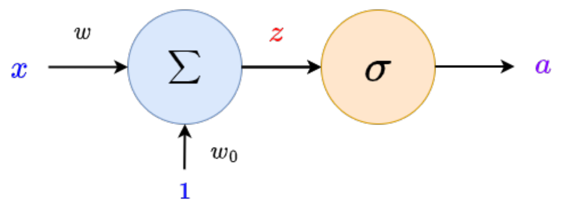
Another way to say this is that one function **feeds into** the next.

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**!



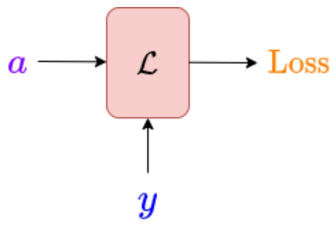
As usual, our first unit $\sum$ is our **linear** component. The output is z, nothing different from before with LLC.

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.

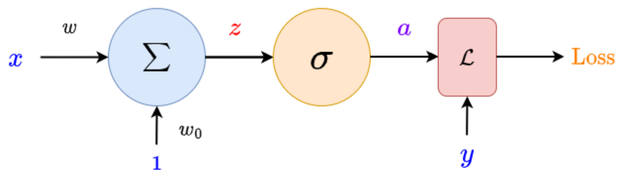
Something we neglected before: this diagram is **missing** the **loss function**. Let's create a small unit for that.

$\mathcal{L}(a, y)$ 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:

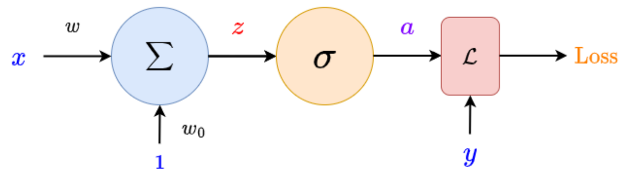
- $f(10) = 10 + 2$
- $g(f) = 3 \cdot 12$
- $h(g) = \sin(36)$

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.

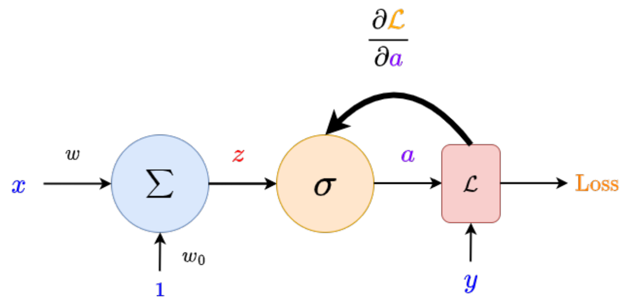


We want to link $\mathcal{L}$ to w. In order to do that, we need to **connect** each thing in between.

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

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

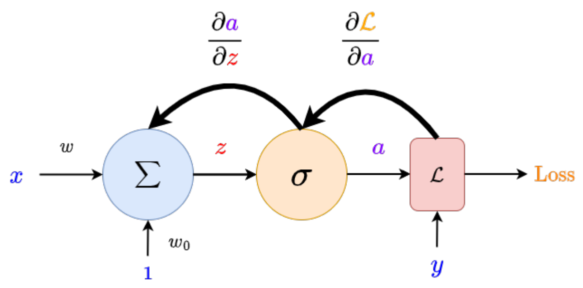
Loss is what we really care about. So, what is the loss directly **connected** to? The **activation**, a.



So, our σ unit has information about the derivative that comes after it: the **loss** derivative

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

And what is that connected to? The **pre-activation** z:

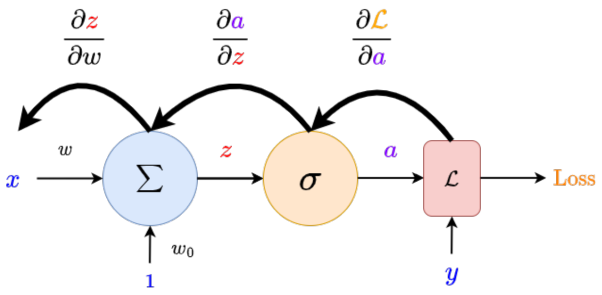


$\frac{\partial a}{\partial z}$
$\frac{\partial \mathcal{L}}{\partial a}$
x
w
$\sum$
z
σ
a
$\mathcal{L}$
Loss
w_0
1
y

Now, our $\sum$ unit has information about both the **loss** derivative and the σ derivative:

$$\overbrace{\frac{\partial \mathcal{L}}{\partial a}}^{\text{Loss unit}} \cdot \overbrace{\frac{\partial a}{\partial z}}^{\text{Activation function}} \tag{6.9}$$

And finally, we've reached w:



$\frac{\partial z}{\partial w}$
$\frac{\partial a}{\partial z}$
$\frac{\partial \mathcal{L}}{\partial a}$
x
w
$\sum$
z
σ
a
$\mathcal{L}$
Loss
w_0
1
y

And, we built our chain rule! This contains the **information** of the derivatives from **every** unit.

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

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:

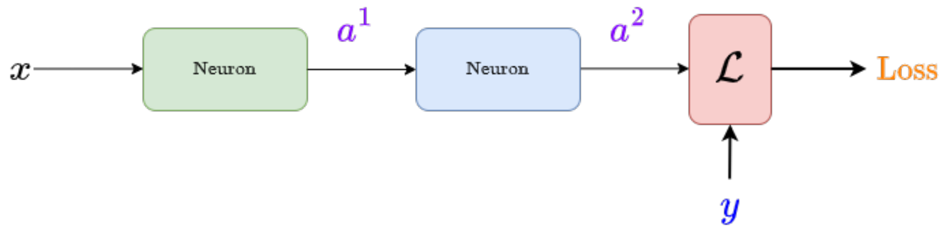
- Do a **forward pass**, where we compute the value of each **unit** in our model, passing the information **forward** - each layer's **output** is the next layer's **input**.
  - We finish by getting the **loss**.
- Do **back-propagation**: build up a **chain rule**, starting at the **loss** function,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

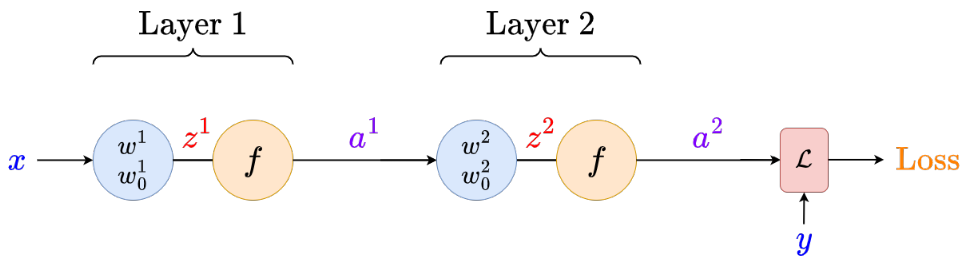
Above, we mention "**each** layer": we'll now transition to a **two-neuron** system, so we have "two layers". Then, we'll build up to many layers.

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:



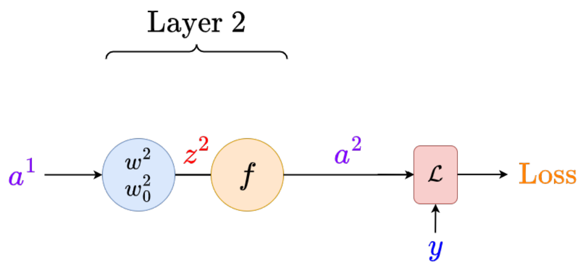
And unpack it:



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?

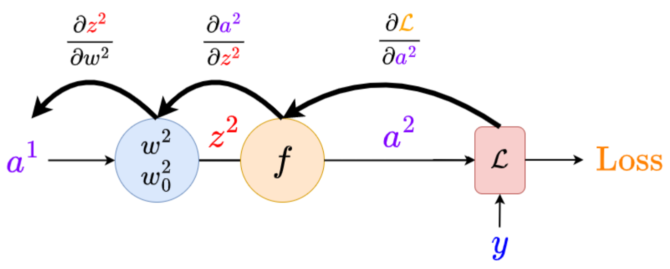
It turns out, it doesn't! We mentioned in the first part of chapter 7 that we can treat the **output** of the **first** layer a^1 as the same as if it were an **input** x.

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".



$\frac{\partial z^2}{\partial w^2}$
$\frac{\partial a^2}{\partial z^2}$
$\frac{\partial \mathcal{L}}{\partial a^2}$
a^1
w^2
w_0^2
z^2
f
a^2
$\mathcal{L}$
Loss
y

We can get:

$$\frac{\partial \mathcal{L}}{\partial w^2} = \overbrace{\frac{\partial \mathcal{L}}{\partial a^2}}^{\text{Loss unit}} \cdot \overbrace{\frac{\partial a^2}{\partial z^2}}^{\text{Activation}} \cdot \overbrace{\frac{\partial z^2}{\partial w^2}}^{\text{Linear}} \tag{6.11}$$

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

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

6.5.8 Continuing backprop: One more problem

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

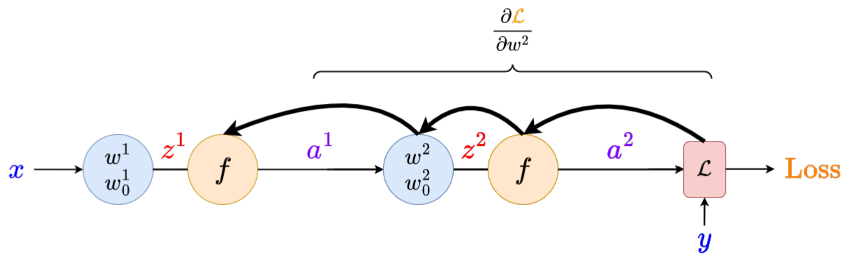
Concept 7

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:

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.

$$\frac{\partial w^2}{\partial a^1} \tag{6.12}$$

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

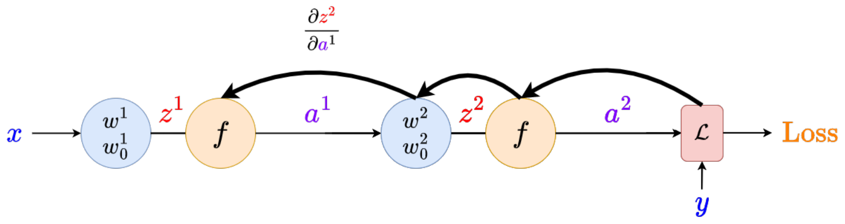
$$w^2 = \begin{bmatrix} w_1^2 \\ w_2^2 \\ \vdots \\ w_n^2 \end{bmatrix}, \qquad \text{Not a function of } a^1! \tag{6.13}$$

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**.

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.

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!



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

Now, we have our new chain rule:

$$\frac{\partial\mathcal{L}}{\partial a^1} = \overbrace{\frac{\partial\mathcal{L}}{\partial a^2}\cdot\frac{\partial a^2}{\partial z^2}}^{\text{Other terms}} \cdot \overbrace{\frac{\partial z^2}{\partial a^1}}^{\text{Link Layers}} \tag{6.14}$$

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

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} \tag{6.15}$$

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

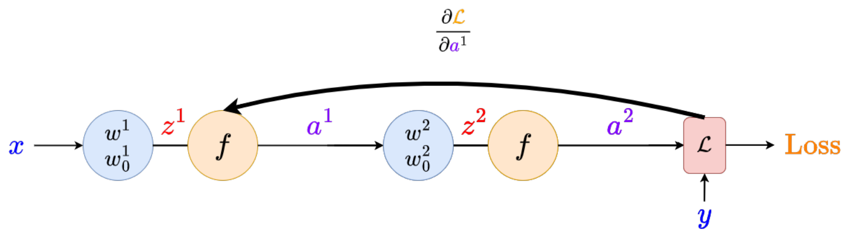
In this section, we compressed lots of derivatives into

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

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

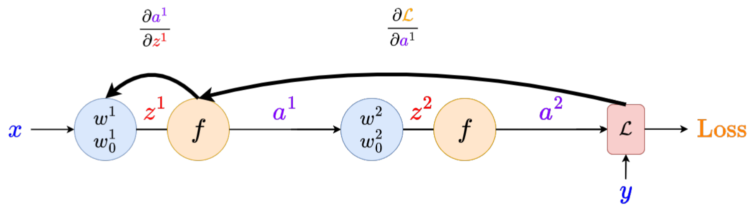
6.5.9 Finishing two-neuron backprop

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



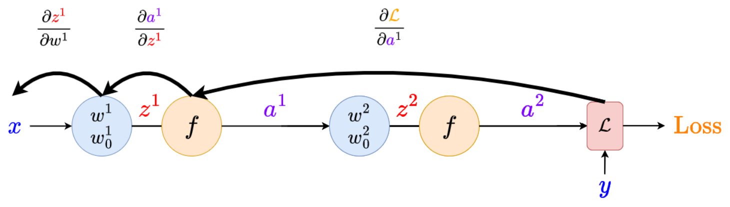
All the info we need is stored in this derivative: it can be written out using our friendly chain rule from earlier.

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



$\frac{\partial a^1}{\partial z^1}$
$\frac{\partial \mathcal{L}}{\partial a^1}$
x
w^1 w_0^1
z^1
f
a^1
w^2 w_0^2
z^2
f
a^2
$\mathcal{L}$
Loss
y

And finally, our input:



$\frac{\partial z^1}{\partial w^1}$
$\frac{\partial a^1}{\partial z^1}$
$\frac{\partial \mathcal{L}}{\partial a^1}$
x
w^1 w_0^1
z^1
f
a^1
w^2 w_0^2
z^2
f
a^2
$\mathcal{L}$
Loss
y

We can get our second chain rule

$$\frac{\partial \mathcal{L}}{\partial w^1} = \overbrace{\frac{\partial \mathcal{L}}{\partial a^1}}^{\text{Other layers}} \cdot \overbrace{\frac{\partial a^1}{\partial z^1} \cdot \frac{\partial z^1}{\partial w^1}}^{\text{Layer 1}} \tag{6.16}$$

Which, in reality, looks much bigger:

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

We see a clear **pattern** here! In fact, this is the procedure we'll use for a neural network with **any** number of layers.

Concept 9

We can get all of our **weight gradients** by repeatedly appending to the **chain rule**.

For each layer, we multiply by

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

To get the **weight gradient** $\partial \mathcal{L}/\partial w^\ell$.

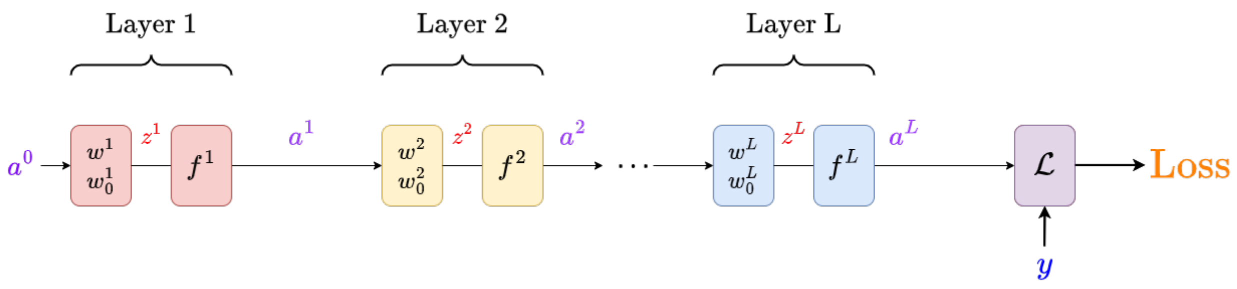
If we want to extend to the next layer, we instead multiply by

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

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:

$$\frac{\partial \mathcal{L}}{\partial w^\ell} = \overbrace{\left(\frac{\partial \mathcal{L}}{\partial a^L}\right)}^{\text{Loss unit}} \cdot \overbrace{\left(\frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}}\right)}^{\text{Layer L}} \cdot \overbrace{\left(\frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^{L-1}}{\partial a^{L-2}}\right)}^{\text{Layer L}-1} \cdot \left(\cdots\right) \cdot \overbrace{\left(\frac{\partial a^\ell}{\partial z^\ell} \cdot \frac{\partial z^\ell}{\partial w^\ell}\right)}^{\text{Layer } \ell} \tag{6.18}$$

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:

$$\frac{\partial \mathcal{L}}{\partial w^\ell} = \overbrace{\frac{\partial \mathcal{L}}{\partial z^\ell}}^{\text{Other}} \cdot \overbrace{\frac{\partial z^\ell}{\partial w^\ell}}^{\text{Weight term}} \tag{6.19}$$

But we should also explore what each of these terms *are*.

6.5.11 What do these derivatives equal?

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

First, every gradient needs

- The **loss derivative**:

$$\frac{\partial \mathcal{L}}{\partial a^L} \tag{6.20}$$

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:

$$\frac{\partial a^\ell}{\partial z^\ell} \tag{6.21}$$

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

$$a = f(z) \tag{6.22}$$

Well, that's not super interesting: we **don't know** our function. But, at least we can **write** it using f: that way, we know that this term only depends on our **activation** function.

$$\frac{\partial a^\ell}{\partial z^\ell} = \Bigg(\overbrace{f^\ell}^{\text{func for layer } \ell} \Bigg)'(z^\ell) \tag{6.23}$$

This expression is a bit visually clunky, but it works.

Between layers, we have

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

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.

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} \tag{6.24}$$

So, we want the function of these two:

$$z^\ell = w^\ell a^{\ell-1} + w_0^\ell \tag{6.25}$$

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

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} = w^\ell \tag{6.26}$$

Finally, every gradient will end with

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

$$\frac{\partial z^\ell}{\partial w^\ell} \tag{6.27}$$

The linear function is the same:

$$z^\ell = w^\ell a^{\ell-1} + w_0^\ell \tag{6.28}$$

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

$$\frac{\partial z^\ell}{\partial w^\ell} = a^{\ell-1} \tag{6.29}$$

Notation 12

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

$$\frac{\partial \mathcal{L}}{\partial a^L} \tag{6.30}$$

$$\frac{\partial a^\ell}{\partial z^\ell} = (f^\ell)'(z^\ell) \tag{6.31}$$

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} = w^\ell \tag{6.32}$$

$$\frac{\partial z^\ell}{\partial w^\ell} = a^{\ell-1} \tag{6.33}$$

Now, we can rewrite our generalized expression for gradient:

$$\frac{\partial \mathcal{L}}{\partial w^\ell} = \overbrace{\left(\frac{\partial \mathcal{L}}{\partial a^L}\right)}^{\text{Loss unit}} \cdot \overbrace{\left((f^L)'(z^L) \cdot w^L\right)}^{\text{Layer L}} \cdot \overbrace{\left((f^{L-1})'(z^{L-1}) \cdot w^L\right)}^{\text{Layer L}-1} \cdot \left(\cdots\right) \cdot \overbrace{\left((f^\ell)'(z^\ell) \cdot a^{\ell-1}\right)}^{\text{Layer }\ell} \tag{6.34}$$

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

6.5.12 Activation Derivatives

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

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

- **Step function** step(z):

$$\frac{d}{dz}\text{step}(z) = 0 \tag{6.35}$$

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

$$\text{step}(z) = \begin{cases} 1 & \text{if } z \geqslant 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{6.36}$$

  And take the derivative of each piece:

$$\frac{d}{dz}\text{ReLU}(z) = 0 = \begin{cases} 0 & \text{if } z \geqslant 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{6.37}$$

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

$$\frac{d}{dz}\text{ReLU}(z) = \text{step}(z) \tag{6.38}$$

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

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z \geqslant 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{6.39}$$

And take the derivative of each piece:

$$\frac{d}{dz}\text{ReLU}(z) = \text{step}(z) = \begin{cases} 1 & \text{if } z \geqslant 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{6.40}$$

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

$$\frac{d}{dz}\sigma(z) = \sigma(z)(1-\sigma(z)) = \frac{e^{-z}}{(1+e^{-z})^2} \tag{6.41}$$

  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:

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{6.42}$$

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

$$\frac{d}{dz}z = 1 \tag{6.43}$$

  This one follows from the definition of the derivative.

  We cannot rely on a linear activation function for our **hidden** layers, because a linear neural network is no more **expressive** than one layer.

  But, we use it for **regression**.

- **Softmax** function 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 $\tanh(z)$:

$$\frac{d}{dz}\tanh(z) = 1 - \tanh(z)^2 \tag{6.44}$$

  This strange little expression is 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 $\mathcal{L}_{sq} = (a-y)^2$:

  $$\frac{d}{da}\mathcal{L}_{sq} = 2(a-y) \tag{6.45}$$

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

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

  $$\frac{d}{da}\mathcal{L}_{lin} = \text{sign}(a-y) \tag{6.46}$$

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

  $$|u| = \begin{cases} u & \text{if } z \geqslant 0 \\ -u & \text{if } z < 0 \end{cases} \tag{6.47}$$

  We take the piecewise derivative:

$$\frac{d}{du}|u| = \text{sign}(u) = \begin{cases} 1 & \text{if } z \geqslant 0 \\ -1 & \text{if } z < 0 \end{cases} \tag{6.48}$$

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

$$\frac{d}{da}\mathcal{L}_{NLL} = -\left(\frac{y}{a} - \frac{1-y}{1-a}\right) \tag{6.49}$$

- **NLLM** (Negative-Log Likelihood Multiclass) function $\mathcal{L}_{NLL} = -\sum_j y_j \log(a_j)$

  Similar to softmax, we will omit this derivative.

Notation 14

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

Square:

$$\frac{d}{da}\mathcal{L}_{sq} = 2(a-y) \tag{6.50}$$

Linear (Absolute):

$$\frac{d}{da}\mathcal{L}_{lin} = \text{sign}(a-y) \tag{6.51}$$

NLL (Negative-Log Likelihood):

$$\frac{d}{da}\mathcal{L}_{NLL} = -\left(\frac{y}{a} - \frac{1-y}{1-a}\right) \tag{6.52}$$

6.5.14 Many neurons per layer

Now, we just have left the elephant in the room: what do we do about the case where we have *full* layers? That is, what if we have **multiple** neurons per layer? This makes this more complex.

Well, the solution is the same as in the first part of chapter 7: we introduce **matrices**.

But this time, with a twist: we have to do **matrix** calculus: a difficult topic indeed.

To handle this, we will go in somewhat **reversed** order, but one that better fits our needs.

- We begin by considering how the chain rule looks when we switch to matrix form.

- We give a general idea of what matrix derivatives look like.

- We list some of the results that matrix calculus gives us, for particular derivatives.
- We actually reason about how matrix calculus *works*.

The last of these is by far the **hardest**, and warrants its own section. Nevertheless, even without it, you can more or less get the idea of what we need - hence why we're going in reversed order.

6.5.15 The chain rule: Matrix form

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

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

With layers, each layer feeds into the **next**, with no other interaction. And neurons within the same layer do not directly interact with each other, which simplifies our math greatly.

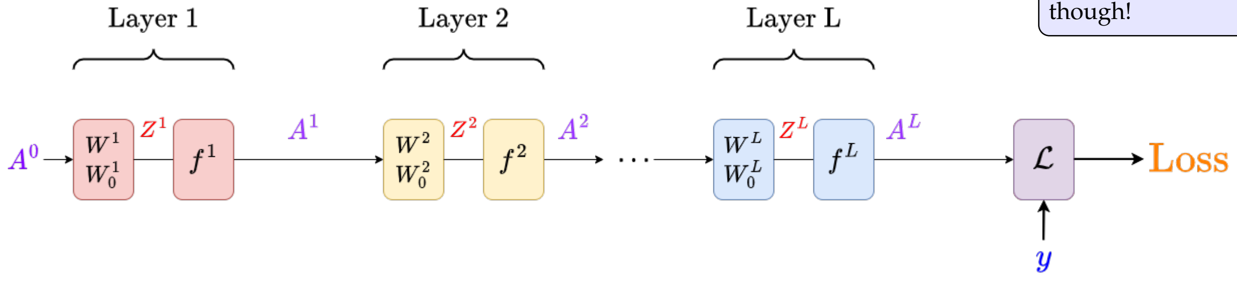
Basically, we have a bunch of functions (neurons) that, within a layer, have nothing to do with each other, and only **output** to the **next** layer of similar functions.

So, we can often **oversimplify** our model by thinking of each layer as like a "big" function, taking in a vector of size m^ℓ and outputting a vector of size n^ℓ.

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

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:



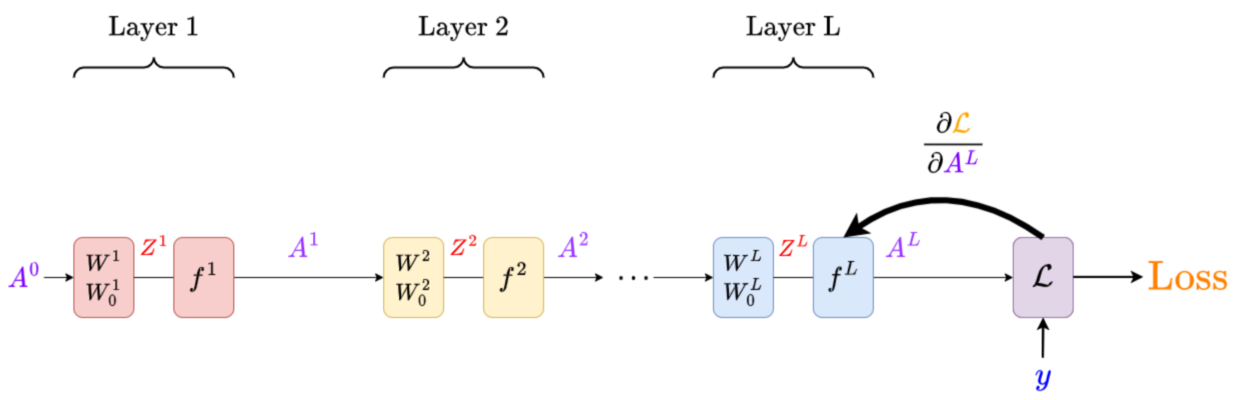
Pretty much the same! Only major difference: swapped scalars for vectors, and vectors for matrices (represented by switching to uppercase)

And, we do backprop the same way, too.

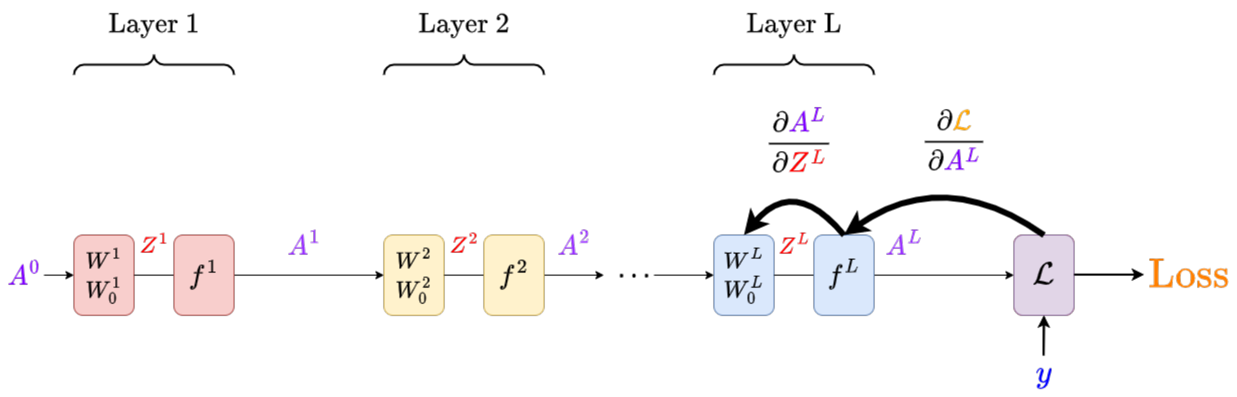
Here, we're not going to explain much as we go: all we're doing is getting the **derivatives** we need for our **chain rule**!

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:

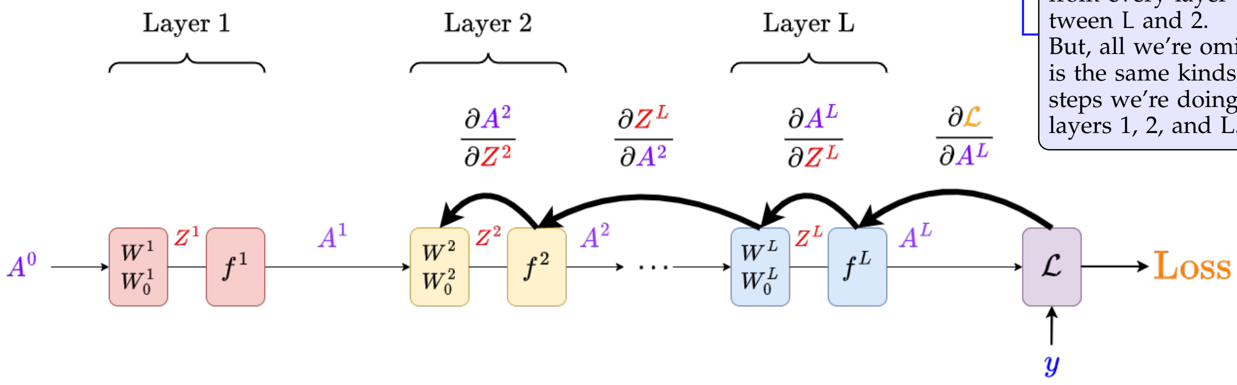


Take another step:

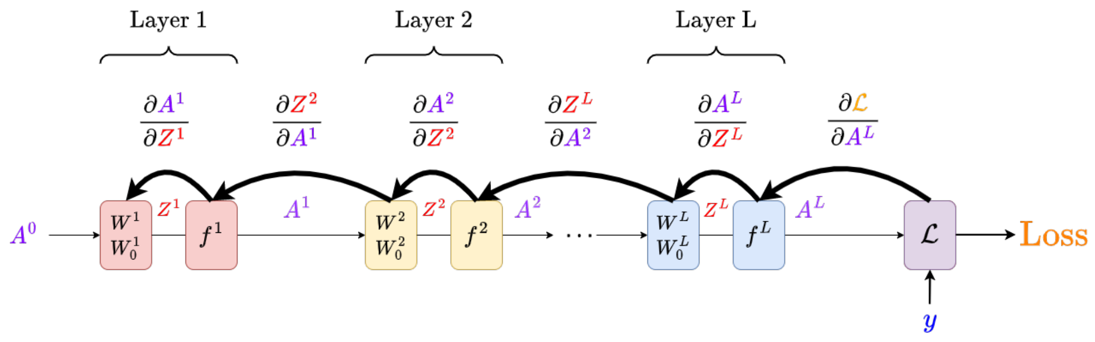


We'll pick up the pace: we'll jump to layer 2 and get its gradient.

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.


Layer 1
Layer 2
Layer L
$\frac{\partial A^2}{\partial Z^2}$
$\frac{\partial Z^L}{\partial A^2}$
$\frac{\partial A^L}{\partial Z^L}$
$\frac{\partial \mathcal{L}}{\partial A^L}$
A^0
W^1 W_0^1
Z^1
f^1
A^1
W^2 W_0^2
Z^2
f^2
A^2
$\cdots$
W^L W_0^L
Z^L
f^L
A^L
$\mathcal{L}$
Loss
y

Now, we finally get to layer 1!



Layer 1
Layer 2
Layer L
$\frac{\partial A^1}{\partial Z^1}$
$\frac{\partial Z^2}{\partial A^1}$
$\frac{\partial A^2}{\partial Z^2}$
$\frac{\partial Z^L}{\partial A^2}$
$\frac{\partial A^L}{\partial Z^L}$
$\frac{\partial \mathcal{L}}{\partial A^L}$
A^0
W^1 W_0^1
Z^1
f^1
A^1
W^2 W_0^2
Z^2
f^2
A^2
$\cdots$
W^L W_0^L
Z^L
f^L
A^L
$\mathcal{L}$
Loss
y

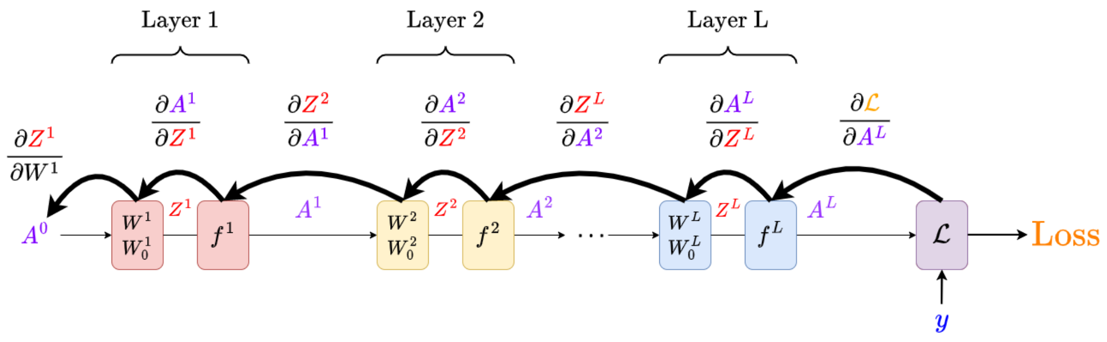
We finish off by getting what we're after: the gradient for W^1.

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.



Layer 1
Layer 2
Layer L
$\frac{\partial Z^1}{\partial W^1}$
$\frac{\partial A^1}{\partial Z^1}$
$\frac{\partial Z^2}{\partial A^1}$
$\frac{\partial A^2}{\partial Z^2}$
$\frac{\partial Z^L}{\partial A^2}$
$\frac{\partial A^L}{\partial Z^L}$
$\frac{\partial \mathcal{L}}{\partial A^L}$
A^0
W^1 W_0^1
Z^1
f^1
A^1
W^2 W_0^2
Z^2
f^2
A^2
$\cdots$
W^L W_0^L
Z^L
f^L
A^L
$\mathcal{L}$
Loss
y

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

$$\frac{\partial Z^\ell}{\partial W^\ell} \tag{6.53}$$

Ends our chain rule when we add it: W^ℓ isn't part of the input or output, so it doesn't connect to the previous layer.

So, for this section, we'll add it **separately** at the end of our chain rule:

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

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 this is will be explained in the matrix derivative notes.

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

Your pre-activation might be

$$z^\ell = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \tag{6.54}$$

The output would be

$$a^\ell = f(z^\ell) = \begin{bmatrix} 1 \\ 2^2 \\ 3^2 \end{bmatrix} \tag{6.55}$$

But the derivative would be:

$$f(z) = 2z \tag{6.56}$$

Which, gives our matrix derivative as:

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

If you want to be able to **derive** some of the derivatives, without reading the matrix derivative section, just use this formula for vector derivatives:

If you have time, do read - you won't understand what you're doing otherwise!

$$\frac{\partial w}{\partial v} = \overbrace{\begin{bmatrix} \frac{\partial w_1}{\partial v_1} & \frac{\partial w_2}{\partial v_1} & \cdots & \frac{\partial w_n}{\partial v_1} \\ \frac{\partial w_1}{\partial v_2} & \frac{\partial w_2}{\partial v_2} & \cdots & \frac{\partial w_n}{\partial v_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial w_1}{\partial v_m} & \frac{\partial w_2}{\partial v_m} & \cdots & \frac{\partial w_n}{\partial v_m} \end{bmatrix}}^{\text{Column } j \text{ matches } w_j} \left.\vphantom{\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}}\right\} \text{Row } i \text{ matches } v_i \tag{6.57}$$

We can use this for scalars as well: we just treat them as a vector of length 1.

With some cleverness, you can derive the Scalar/Matrix and Matrix/Scalar derivatives as well.

This is contained in the matrix derivatives chapter.

6.6 Training

6.6.1 Comments

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

Concept 18

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

$$\partial \mathcal{L}/\partial A^L \quad \text{and} \quad \partial A^L/\partial Z^L$$

So, our algorithm may change slightly.

Another thought: intialization.

Concept 19

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.

```
SGD-NEURAL-NET($\mathcal{D}_n, T, L, (m^1, \ldots, m^L), (f^1, \ldots, f^L), \text{Loss}$)
for every layer:
    Randomly initialize
        the weights in every layer
        the biases in every layer

While termination condition not met:
    Get random data point $i$
    Kepp 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
```

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