Explanatory Notes for 6.390

Shaunticlair Ruiz (Current TA)

Fall 2022

Contents

CHAPTER 7

Neural Networks 1.5 - Back-Propagation and Training

To jump directly to matrix derivatives, click here.

7.5 Error back-propagation

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

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

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

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**.

7.5.1 Review: Gradient Descent

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

Concept 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

$$\Delta\theta = -\eta \frac{\partial \mathcal{L}}{\partial \theta} \tag{7.1}$$

  - 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.

7.5.2 Review: Gradient Descent with LLCs

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

Our LLC model uses the following equations:

We'll use w instead of θ.

$$z(x) = w^T x + w_0 \qquad g(z) = \sigma(z) = \frac{1}{1+e^{-z}} \tag{7.2}$$

$$\mathcal{L}(g, y) = y\log(g) + (1-y)\log(1-g) \tag{7.3}$$

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

So, we want

$$\frac{\partial \mathcal{L}}{\partial w} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial w_0} \tag{7.4}$$

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

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

$$\frac{\partial \mathcal{L}}{\partial w} = \overbrace{\frac{\partial \mathcal{L}}{\partial g}}^{\mathcal{L}(g)} \cdot \frac{\partial g}{\partial w} \tag{7.5}$$

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

$$\frac{\partial \mathcal{L}}{\partial w} = \overbrace{\frac{\partial \mathcal{L}}{\partial g} \cdot \underbrace{\frac{\partial g}{\partial z}}_{}}^{\mathcal{L}(g)}\underbrace{\cdot \frac{\partial z}{\partial w}}_{g(z)} \tag{7.6}$$

Plugging in our derivatives, we get:

$$\frac{\partial \mathcal{L}}{\partial w} = \overbrace{-\left(\frac{y}{\sigma} - \frac{1-y}{1-\sigma}\right)}^{\partial\mathcal{L}/\partial g} \cdot \overbrace{\sigma(1-\sigma)}^{\partial g/\partial z} \cdot \overbrace{x}^{\partial z/\partial w} \tag{7.7}$$

Concept 3

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

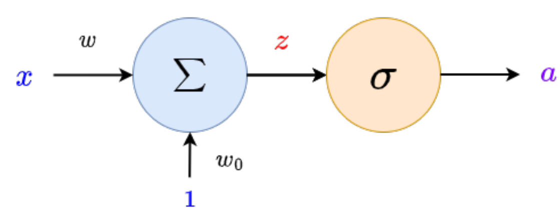
Another way to say this is that one function **feeds into** the next.

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.

7.5.3 Review: LLC as Neuron

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



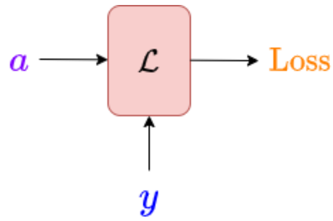
As usual, our first unit $\sum$ is our **linear** component. The output is z, nothing different from before with LLC.

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

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

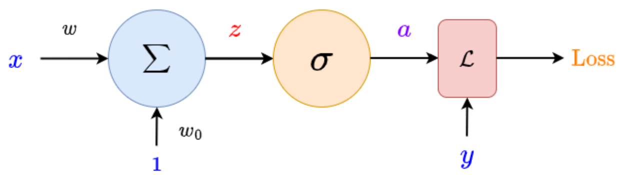
Something we neglected before: this diagram is **missing** the **loss function**. Let's create a small unit for that.

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



We have two inputs to our loss function.

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



Our full unit!

7.5.4 LLC Forward-Pass

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

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

First, one important consideration:

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

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

Definition 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:

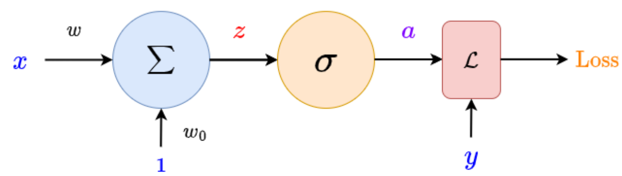
- $f(10) = 10 + 2$
- $g(f) = 3 \cdot 12$
- $h(g) = \sin(36)$

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

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

7.5.5 LLC Back-propagation

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

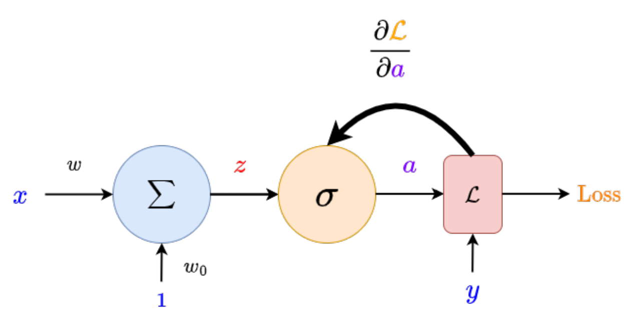


We want to link $\mathcal{L}$ to w. In order to do that, we need to **connect** each thing in between.

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

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

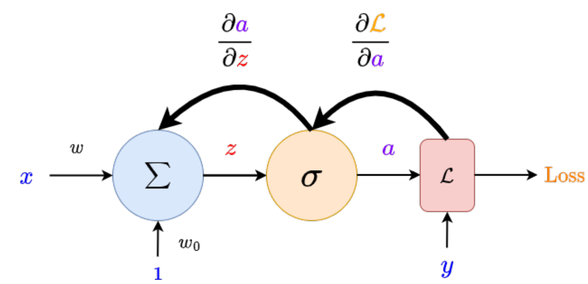
Loss is what we really care about. So, what is the loss directly **connected** to? The **activation**, a.



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

$$\frac{\overbrace{\partial \mathcal{L}}^{\text{Loss unit}}}{\partial a} \tag{7.8}$$

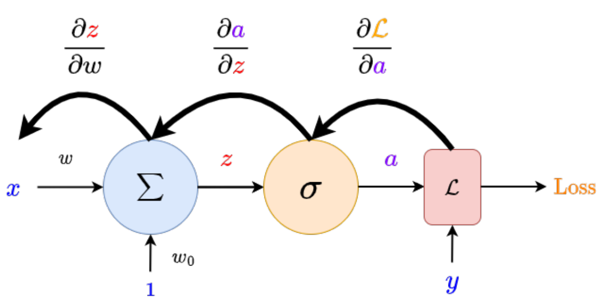
And what is that connected to? The **pre-activation** z:



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

$$\overbrace{\frac{\partial \mathcal{L}}{\partial a}}^{\text{Loss unit}} \cdot \overbrace{\frac{\partial a}{\partial z}}^{\text{Activation function}} \tag{7.9}$$

And finally, we've reached w:



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

$$\frac{\partial \mathcal{L}}{\partial w} = \overbrace{\frac{\partial \mathcal{L}}{\partial a}}^{\text{Loss unit}} \cdot \overbrace{\frac{\partial a}{\partial z}}^{\text{Activation}} \cdot \overbrace{\frac{\partial z}{\partial w}}^{\text{Linear subunit}} \tag{7.10}$$

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.

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

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

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

Concept 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.

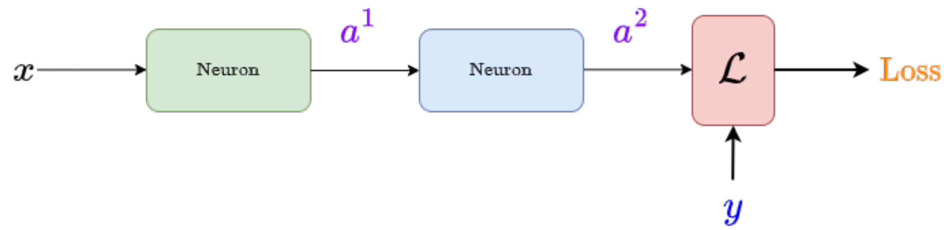
- Use your chain rule to get the **gradient** $\frac{\partial \mathcal{L}}{\partial w}$ for your **weight** vector(s). Take a **gradient descent** step.
- **Repeat** until satisfied, or your model **converges**.

7.5.7 A two-neuron network: starting backprop

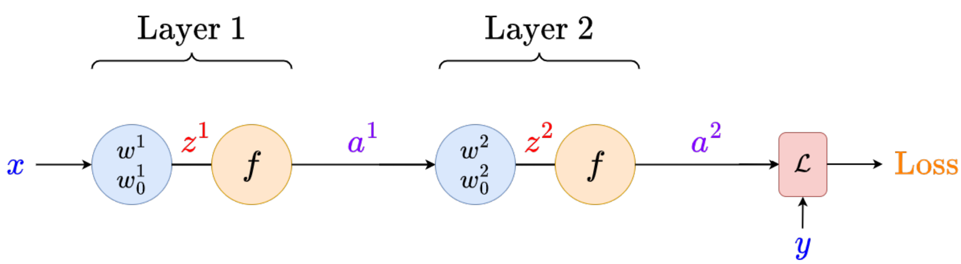
Above, we mention "**each** layer": we'll now transition to a **two-neuron** system, so we have "two layers". Then, we'll build up to many layers.

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

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



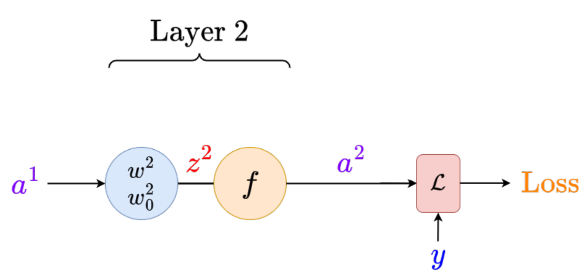
And unpack it:



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

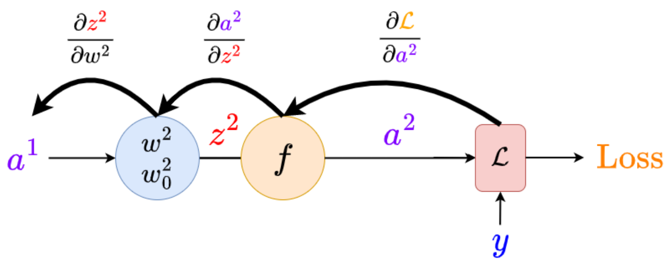
It turns out, it doesn't! We mentioned in the first part of chapter 7 that we can treat the **output** of the **first** layer a^1 as the same as if it were an **input** x.

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



Now, we can do backprop safely.

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



We can get:

$$\frac{\partial \mathcal{L}}{\partial w^2} = \overbrace{\frac{\partial \mathcal{L}}{\partial a^2}}^{\text{Loss unit}} \cdot \overbrace{\frac{\partial a^2}{\partial z^2}}^{\text{Activation}} \cdot \overbrace{\frac{\partial z^2}{\partial w^2}}^{\text{Linear}} \tag{7.11}$$

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

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

7.5.8 Continuing backprop: One more problem

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

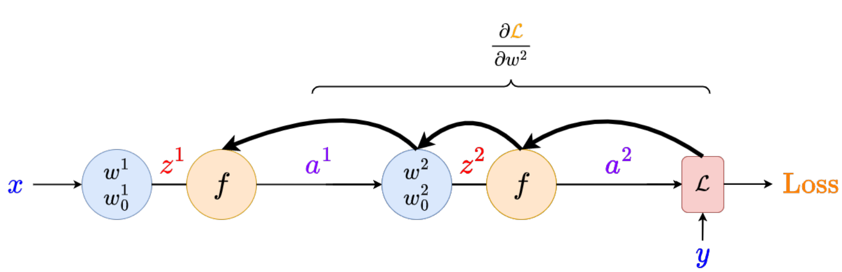
Concept 7

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:

$$\frac{\partial w^2}{\partial a^1} \tag{7.12}$$

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.

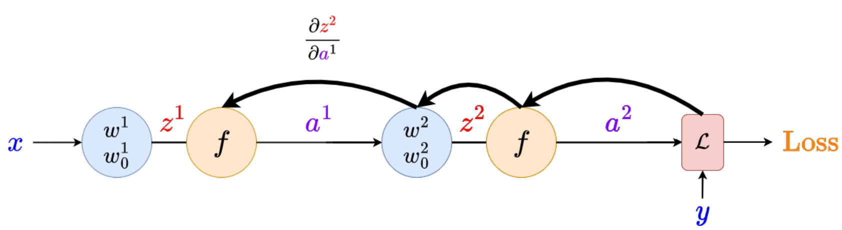
$$w^2 = \begin{bmatrix} w_1^2 \\ w_2^2 \\ \vdots \\ w_n^2 \end{bmatrix}, \qquad \text{Not a function of } a^1! \tag{7.13}$$

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:

$$\frac{\partial\mathcal{L}}{\partial a^1} = \overbrace{\frac{\partial\mathcal{L}}{\partial a^2}\cdot\frac{\partial a^2}{\partial z^2}}^{\text{Other terms}} \cdot \overbrace{\frac{\partial z^2}{\partial a^1}}^{\text{Link Layers}} \tag{7.14}$$

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

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} \tag{7.15}$$

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

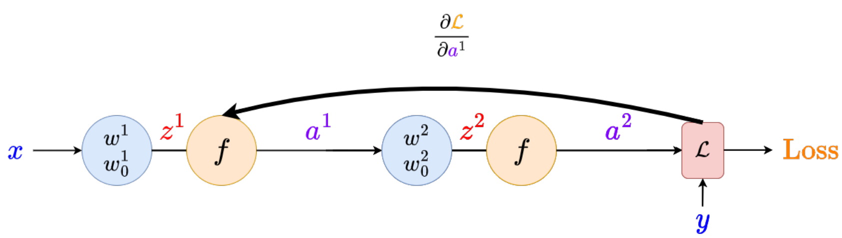
In this section, we compressed lots of derivatives into

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

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

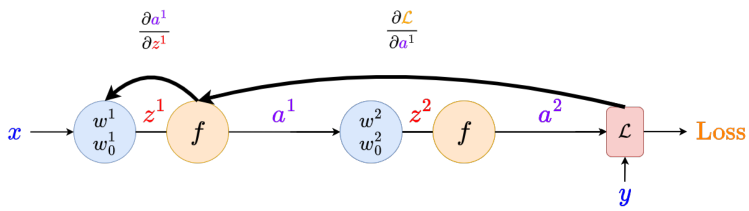
7.5.9 Finishing two-neuron backprop

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

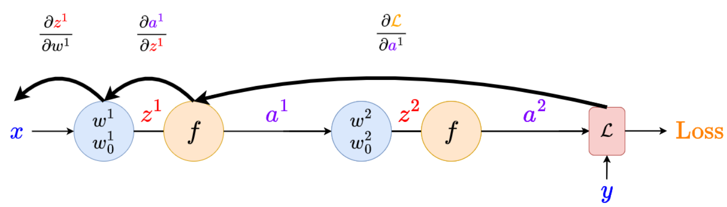


All the info we need is stored in this derivative: it can be written out using our friendly chain rule from earlier.

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



And finally, our input:



We can get our second chain rule

$$\frac{\partial \mathcal{L}}{\partial w^1} = \overbrace{\frac{\partial \mathcal{L}}{\partial a^1}}^{\text{Other layers}} \cdot \overbrace{\frac{\partial a^1}{\partial z^1} \cdot \frac{\partial z^1}{\partial w^1}}^{\text{Layer 1}} \tag{7.16}$$

Which, in reality, looks much bigger:

$$\frac{\partial \mathcal{L}}{\partial w^1} = \overbrace{\left(\frac{\partial \mathcal{L}}{\partial a^2}\right)}^{\text{Loss unit}} \cdot \overbrace{\left(\frac{\partial a^2}{\partial z^2} \cdot \frac{\partial z^2}{\partial a^1}\right)}^{\text{Layer 2}} \cdot \overbrace{\left(\frac{\partial a^1}{\partial z^1} \cdot \frac{\partial z^1}{\partial w^1}\right)}^{\text{Layer 1}} \tag{7.17}$$

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

Concept 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** $\partial\mathcal{L}/\partial w^\ell$.

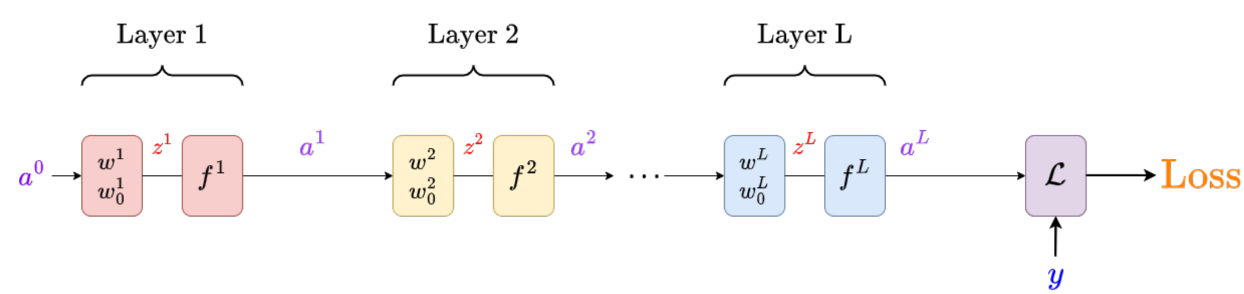
If we want to **extend** to the next layer, we instead multiply by

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

7.5.10 Many layers: Doing back-propagation

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

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



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

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

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

Concept 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:

$$\frac{\partial \mathcal{L}}{\partial w^\ell} = \overbrace{\left(\frac{\partial \mathcal{L}}{\partial a^L}\right)}^{\text{Loss unit}} \cdot \overbrace{\left(\frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}}\right)}^{\text{Layer } L} \cdot \overbrace{\left(\frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^{L-1}}{\partial a^{L-2}}\right)}^{\text{Layer } L-1} \cdot \left(\cdots\right) \cdot \overbrace{\left(\frac{\partial a^\ell}{\partial z^\ell} \cdot \frac{\partial z^\ell}{\partial w^\ell}\right)}^{\text{Layer } \ell} \tag{7.18}$$

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:

$$\frac{\partial \mathcal{L}}{\partial w^\ell} = \overbrace{\frac{\partial \mathcal{L}}{\partial z^\ell}}^{\text{Other}} \cdot \overbrace{\frac{\partial z^\ell}{\partial w^\ell}}^{\text{Weight term}} \tag{7.19}$$

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

7.5.11 What do these derivatives equal?

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

First, every gradient needs

- The **loss derivative**:

$$\frac{\partial \mathcal{L}}{\partial a^L} \tag{7.20}$$

  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:

$$\frac{\partial a^\ell}{\partial z^\ell} \tag{7.21}$$

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

$$a = f(z) \tag{7.22}$$

  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.

$$\frac{\partial a^\ell}{\partial z^\ell} = \Big(\overbrace{f^\ell}^{\text{func for layer } \ell} \Big)'(z^\ell) \tag{7.23}$$

  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.

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} \tag{7.24}$$

So, we want the function of these two:

$$z^\ell = w^\ell a^{\ell-1} + w_0^\ell \tag{7.25}$$

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

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} = w^\ell \tag{7.26}$$

Finally, every gradient will end with

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

$$\frac{\partial z^\ell}{\partial w^\ell} \tag{7.27}$$

The linear function is the same:

$$z^\ell = w^\ell a^{\ell-1} + w_0^\ell \tag{7.28}$$

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

$$\frac{\partial z^\ell}{\partial w^\ell} = a^{\ell-1} \tag{7.29}$$

Notation 12

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

$$\frac{\partial \mathcal{L}}{\partial a^L} \tag{7.30}$$

$$\frac{\partial a^\ell}{\partial z^\ell} = (f^\ell)'(z^\ell) \tag{7.31}$$

$$\frac{\partial z^\ell}{\partial a^{\ell-1}} = w^\ell \tag{7.32}$$

$$\frac{\partial z^\ell}{\partial w^\ell} = a^{\ell-1} \tag{7.33}$$

Now, we can rewrite our generalized expression for gradient:

$$\frac{\partial \mathcal{L}}{\partial w^\ell} = \overbrace{\left(\frac{\partial \mathcal{L}}{\partial a^L}\right)}^{\text{Loss unit}} \cdot \overbrace{\left((f^L)'(z^L) \cdot w^L\right)}^{\text{Layer } L} \cdot \overbrace{\left((f^{L-1})'(z^{L-1}) \cdot w^L\right)}^{\text{Layer } L-1} \cdot \left(\cdots\right) \cdot \overbrace{\left((f^\ell)'(z^\ell) \cdot a^{\ell-1}\right)}^{\text{Layer } \ell} \tag{7.34}$$

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

7.5.12 Activation Derivatives

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

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

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

$$\frac{d}{dz}\text{step}(z) = 0 \tag{7.35}$$

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

$$\text{step}(z) = \begin{cases} 1 & \text{if } z \geqslant 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{7.36}$$

  And take the derivative of each piece:

$$\frac{d}{dz}\text{ReLU}(z) = 0 = \begin{cases} 0 & \text{if } z \geqslant 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{7.37}$$

- **Rectified Linear Unit** $\text{ReLU}(z)$:

$$\frac{d}{dz}\text{ReLU}(z) = \text{step}(z) \tag{7.38}$$

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

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{if } z \geqslant 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{7.39}$$

And take the derivative of each piece:

$$\frac{d}{dz}\text{ReLU}(z) = \text{step}(z) = \begin{cases} 1 & \text{if } z \geqslant 0 \\ 0 & \text{if } z < 0 \end{cases} \tag{7.40}$$

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

$$\frac{d}{dz}\sigma(z) = \sigma(z)(1 - \sigma(z)) = \frac{e^{-z}}{(1 + e^{-z})^2} \tag{7.41}$$

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:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{7.42}$$

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

$$\frac{d}{dz} z = 1 \tag{7.43}$$

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 $\tanh(z)$:

$$\frac{d}{dz}\tanh(z) = 1 - \tanh(z)^2 \tag{7.44}$$

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:

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

7.5.13 Loss derivatives

Now, we look at the loss derivatives.

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

$$\frac{d}{da}\mathcal{L}_{sq} = 2(a - y) \tag{7.45}$$

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

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

$$\frac{d}{da}\mathcal{L}_{lin} = \text{sign}(a - y) \tag{7.46}$$

  This one can also be handled piecewise, like $\text{step}(z)$ and $\text{ReLU}(z)$:

$$|u| = \begin{cases} u & \text{if } z \geqslant 0 \\ -u & \text{if } z < 0 \end{cases} \tag{7.47}$$

  We take the piecewise derivative:

$$\frac{d}{du}|u| = \text{sign}(u) = \begin{cases} 1 & \text{if } z \geqslant 0 \\ -1 & \text{if } z < 0 \end{cases} \tag{7.48}$$

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

$$\frac{d}{da}\mathcal{L}_{NLL} = -\left(\frac{y}{a} - \frac{1-y}{1-a}\right) \tag{7.49}$$

- **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:

$$\frac{d}{da}\mathcal{L}_{sq} = 2(a-y) \tag{7.50}$$

Linear (Absolute):

$$\frac{d}{da}\mathcal{L}_{lin} = \text{sign}(a-y) \tag{7.51}$$

NLL (Negative-Log Likelihood):

$$\frac{d}{da}\mathcal{L}_{NLL} = -\left(\frac{y}{a} - \frac{1-y}{1-a}\right) \tag{7.52}$$

7.5.14 Many neurons per layer

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

7.5.15 The chain rule: Matrix form

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

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

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

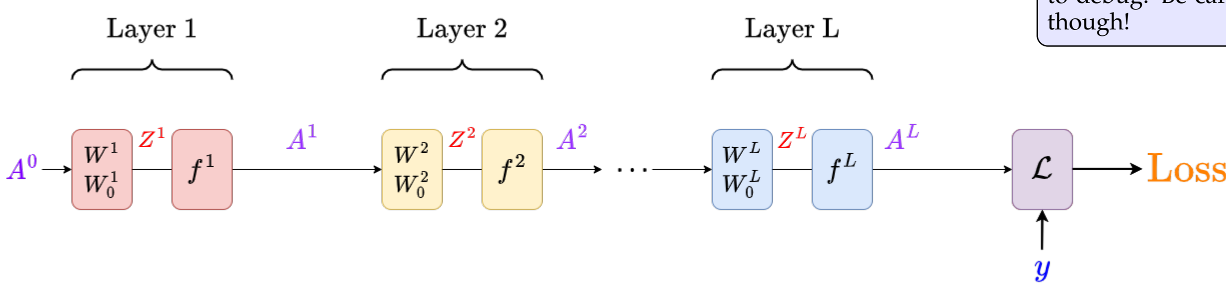
Basically, we have a bunch of functions (neurons) that, within a layer, have nothing to do with each other, and only **output** to the **next** layer of similar functions.

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

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

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:



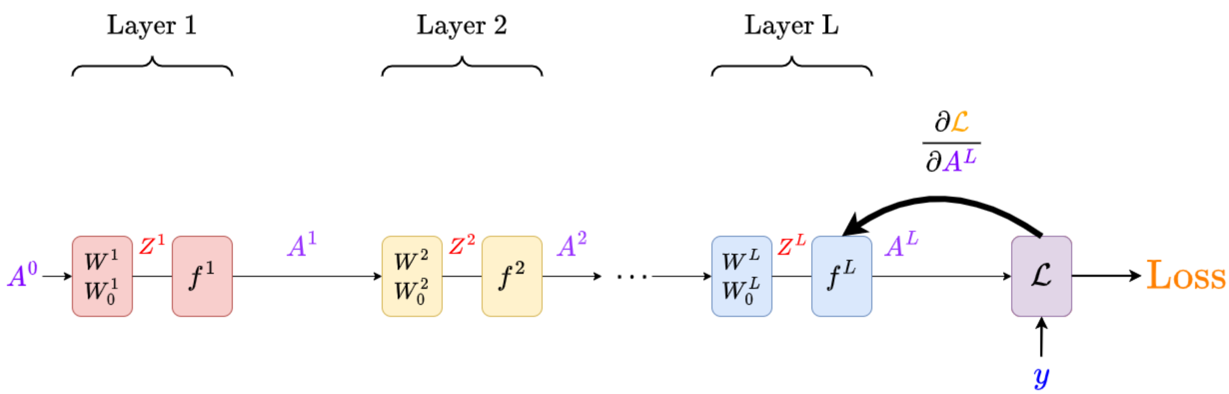
Pretty much the same! Only major difference: swapped scalars for vectors, and vectors for matrices (represented by switching to uppercase)

And, we do backprop the same way, too.

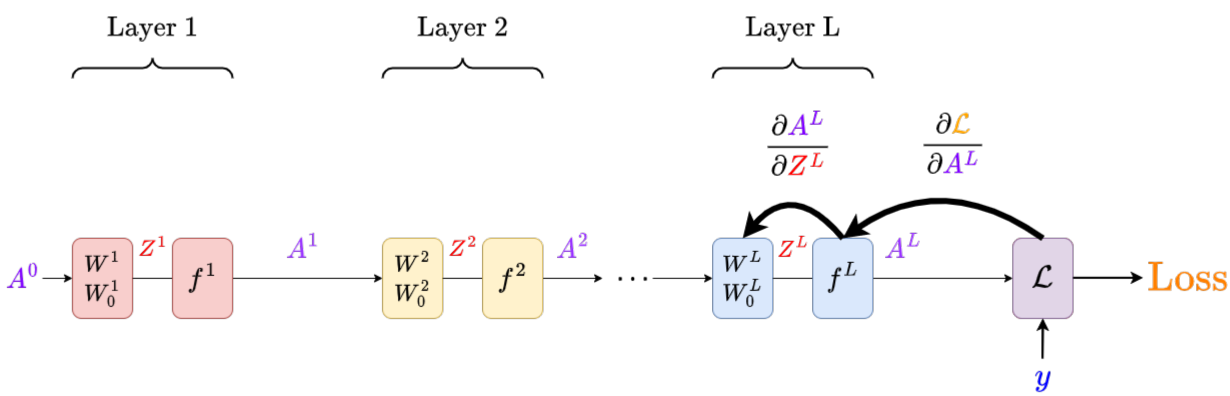
Here, we're not going to explain much as we go: all we're doing is getting the **derivatives** we need for our **chain rule**!

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:

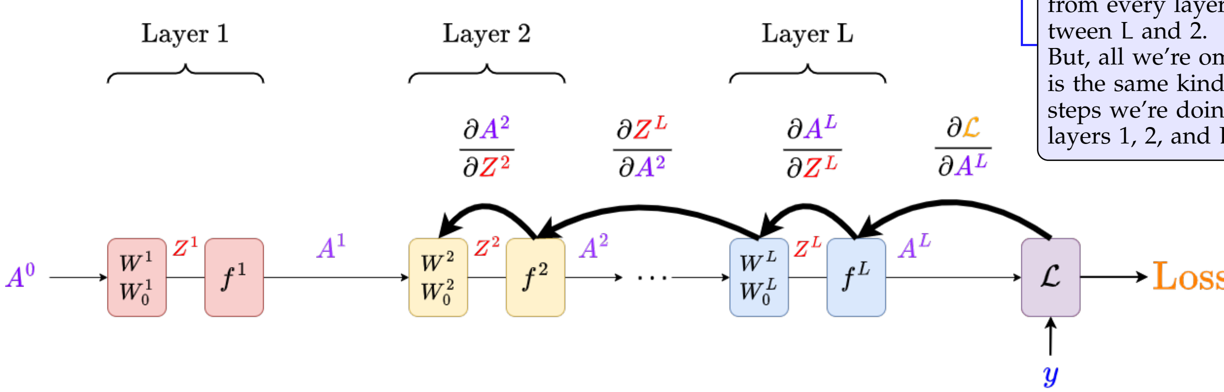


Take another step:

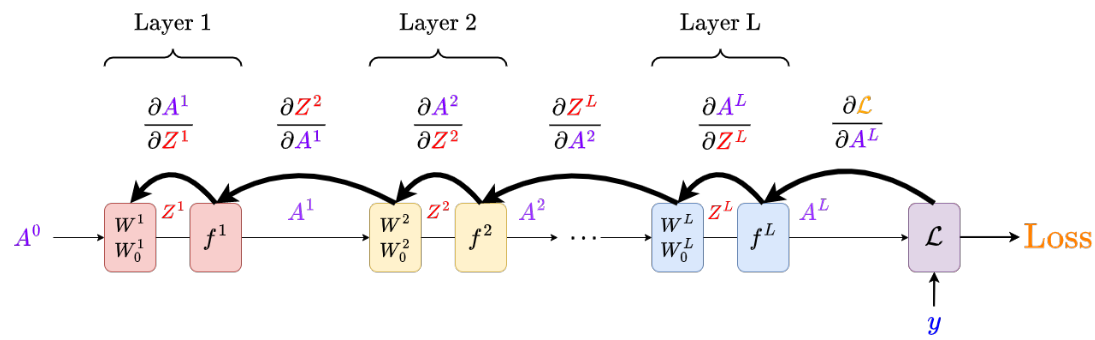


We'll pick up the pace: we'll jump to layer 2 and get its gradient.

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.

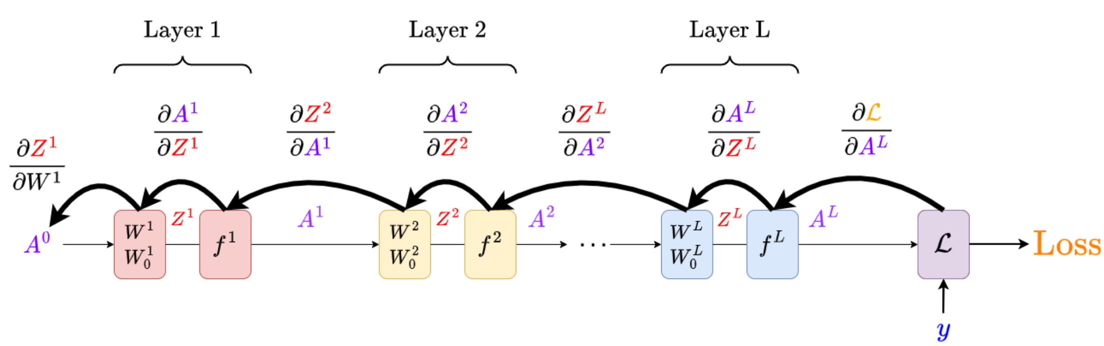


Now, we finally get to layer 1!



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.

7.5.16 How the Chain Rule changes in Matrix form

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

$$\frac{\partial Z^\ell}{\partial W^\ell} \tag{7.53}$$

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!

7.5.17 Relevant Derivatives

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

Notation 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

$$z^\ell = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \tag{7.54}$$

The output would be

$$\mathbf{a}^{\ell} = f(\mathbf{z}^{\ell}) = \begin{bmatrix} 1 \\ 2^2 \\ 3^2 \end{bmatrix} \tag{7.55}$$

But the derivative would be:

$$f(\mathbf{z}) = 2z \tag{7.56}$$

Which, gives our matrix derivative as:

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

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

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

$$\frac{\partial w}{\partial v} = \left.\overbrace{\begin{bmatrix} \frac{\partial w_1}{\partial v_1} & \frac{\partial w_2}{\partial v_1} & \cdots & \frac{\partial w_n}{\partial v_1} \\ \frac{\partial w_1}{\partial v_2} & \frac{\partial w_2}{\partial v_2} & \cdots & \frac{\partial w_n}{\partial v_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial w_1}{\partial v_m} & \frac{\partial w_2}{\partial v_m} & \cdots & \frac{\partial w_n}{\partial v_m} \end{bmatrix}}^{\text{Column } j \text{ matches } w_j}\right\} \text{Row } i \text{ matches } v_i \tag{7.57}$$

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.

Part of what the next section covers.

If you want to skip the matrix derivatives section, click here.

7.X Matrix Derivatives

In general, we want to be able to combine the powers of matrices and calculus:

- **Matrices**: the ability to store lots of **data**, and do fast linear operations on all that data at the **same time**.

  Example: Consider

$$w^T x = \begin{bmatrix} w_1 & w_2 & \cdots & w_m \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{bmatrix} = \sum_{i=1}^{m} x_i w_i \tag{7.58}$$

  In this case, we're able to do m different **multiplications** at the same time! This is what we like about matrices.

In this case, we're thinking about vectors as $(m \times 1)$ matrices.

- **Calculus**: analyzing the way different variables are **related**: how does changing x affect y?

  Example: Suppose we have

$$\frac{\partial f}{\partial x_1} = 10 \qquad \frac{\partial f}{\partial x_2} = -5 \tag{7.59}$$

  Now we know that, if we increase x_1, we increase f. This **understanding** of variables is what we like about derivatives.

Concept 18

Matrix derivatives allow us to find **relationships** between large volumes of **data**.

- These "relationships" are **derivatives**: consider dy/dx. How does y change if we modify x? Currently, we only have **scalar derivatives**.
- This "data" is stored as **matrices**: blocks of data, that we can do linear operations (matrix multiplication) on.

Our goal is to work with many scalar derivatives at the **same time**.

In order to do that, we can apply some **derivative** rules, but we have to do it in a way that **agrees** with **matrix** math.

Our work is a careful balancing act between getting the **derivatives** we want, without violating the **rules** of matrices (and losing what makes them useful!)

Example: When we multiply two matrices, their inner shape has to match: in the below case, they need to share a dimension b.

$$\overbrace{X}^{(a\times b)}\overbrace{Y}^{(b\times c)} \tag{7.60}$$

We can't do anything that would **violate** this rule: otherwise, our **equations** don't make sense, and we get stuck. This means we need to build our math carefully.

First, we'll look at the **properties** of derivatives. Then figure out how to usefully apply them to **vectors**, and then **matrices**.

7.X.1 Review: Partial Derivatives

One more comment, though - we may have many different variables floating around. This means we **have** to use the multivariable **partial derivative**.

Definition 19

The **partial derivative**

$$\frac{\partial B}{\partial A}$$

Is used when there may be **multiple variables** in our functions.

The rule of the partial derivative is that we keep every **independent** variable other than A and B **fixed**.

Example: Consider $f(x, y) = 2x^2y$.

$$\frac{\partial f}{\partial x} = 2(2x)y \tag{7.61}$$

Here, we kept y *fixed* - we treat it as if it were an unchanging **constant**.

Using the partial derivative lets us keep our work tidy: if **many** variables were allowed to **change** at the same time, it could get very confusing.

Imagine keeping track of k different variables x_i with k different changes Δx_i at the same time! That's a headache.

If this is too complicated, we can change those variables *one at a time*. We get a partial derivative for each of them, holding the others **constant**.

Our **total** derivative is the result of all of those different variables, **added** together. This is how we get the **multi-variable chain rule**.

Definition 20

The **multi-variable chain rule** in 3-D ($\{x, y, z\}$) is given as

$$\frac{df}{ds} = \overbrace{\frac{\partial f}{\partial x}\frac{\partial x}{\partial s}}^{\text{only modify } x} + \overbrace{\frac{\partial f}{\partial y}\frac{\partial y}{\partial s}}^{\text{only modify } y} + \overbrace{\frac{\partial f}{\partial z}\frac{\partial z}{\partial s}}^{\text{only modify } z}$$

If we have k variables $\{x_1, x_2, \dots x_k\}$ we can generalize this as:

$$\frac{df}{ds} = \sum_{i=1}^{k} \overbrace{\frac{\partial f}{\partial x_i}\frac{\partial x_i}{\partial s}}^{x_i \text{ component}}$$

7.X.2 Thinking about derivatives

The typical definition of derivatives

$$\lim_{h \to 0} \frac{f(x+h) - f(x)}{h} \tag{7.62}$$

Gives an *idea* of what sort of things we're looking for. It reminds us of one piece of information we need:

- Our derivative **depends** on the **current position** x we are taking the derivative at.

We need this because derivative are **local**: the relationship between our variables might change if we move to a different **position**.

But, the problem with vectors is that each component can act **separately**: if we have a vector, we can change in many different "directions".

$$A = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \qquad B = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{7.63}$$

Example: Suppose we want a derivative $\partial B/\partial A$: $\Delta a_1, \Delta a_2$, and Δa_3 could each, separately, have an effect on Δb_1 and/or Δb_2. That requires 6 different derivatives, $\partial b_i/\partial a_j$.

3 dimensions of A times 2 dimensions of B: 6 combinations.

Every component of the input A can potentially modify **every** component of the output B.

One solution we could try is to just collect all of these derivatives into a **vector** or **matrix**.

Concept 21

For the **derivative** between two objects (scalars, vectors, matrices) A and B

$$\frac{\partial B}{\partial A}$$

We need to get the **derivatives**

$$\frac{\partial b_j}{\partial a_i}$$

between every **pair** of elements a_i, b_j: each pair of elements could have a **relationship**.

The total number of elements (or "size") is...

$$\text{Size}\left(\frac{\partial B}{\partial A}\right) = \text{Size}(B) * \text{Size}(A)$$

Collecting these values into a **matrix** will gives us all the information we need.

But, how do we gather them? What should the **shape** look like? Should we **transpose** our matrix or not?

7.X.3 Derivatives: Approximation

To answer this, we need to ask ourselves *why* we care about these derivatives: their **structure** will be based on what we need them for.

- We care about the **direction of greatest decrease**, the gradient. For example, we might want to adjust weight vector w to reduce $\mathcal{L}$.
- We also want other derivatives that have the **same** behavior, so we can combine them using the **chain rule**.

Let's focus on the first point: we want to **minimize** $\mathcal{L}$. Our focus is the **change** in $\mathcal{L}$, $\Delta\mathcal{L}$.

We want to take steps that reduce our loss $\mathcal{L}$.

$$\frac{\partial \mathcal{L}}{\partial w} \approx \frac{\text{Change in } \mathcal{L}}{\text{Change in w}} = \frac{\Delta \mathcal{L}}{\Delta w} \tag{7.64}$$

Thus, we **solve** for $\Delta\mathcal{L}$:

All we do is multiply both sides by Δw.

$$\Delta \mathcal{L} \approx \frac{\partial \mathcal{L}}{\partial w} \Delta w \tag{7.65}$$

Since this derivation was gotten using scalars, we might need a **different** type of multiplication for our **vector** and **matrix** derivatives.

Concept 22

We can use derivatives to **approximate** the change in our output based on our input:

$$\Delta \mathcal{L} \approx \frac{\partial \mathcal{L}}{\partial w} \star \Delta w$$

Where the $\star$ symbol represents some type of **multiplication**.

We can think of this as a **function** that takes in change in Δw, and returns an **approximation** of the loss.

We already understand **scalar** derivatives, so let's move on to the **gradient**.

7.X.4 The Gradient: a vector input, scalar output

Our plan is to look at every derivative combination of scalars, vectors, and matrices we can.

First, we consider:

$$\frac{\partial(\text{Scalar})}{\partial(\text{Vector})} = \frac{\partial s}{\partial v} \tag{7.66}$$

We'll take s to be our scalar, and v to be our vector. So, our input is a **vector**, and our output is a **scalar**.

$$\Delta v \longrightarrow \boxed{f} \longrightarrow \Delta s \tag{7.67}$$

How do we make sense of this? Well, let's write Δv_i explicitly:

$$\overbrace{\begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix}}^{\Delta v} \longrightarrow \Delta s \tag{7.68}$$

We can see that we have m different **inputs** we can change in order to change our **one** output.

So, our derivative needs to have m different **elements**: one for each element v_i.

7.X.5 Finding the scalar/vector derivative

But how do we shape our matrix? Let's look at our **rule**.

$$\Delta s \approx \frac{\partial s}{\partial v} \star \Delta v \qquad \text{or} \qquad \Delta s \approx \frac{\partial s}{\partial v} \star \overbrace{\begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix}}^{\Delta v} \tag{7.69}$$

How do we get $\Delta reds$? We have so many variables. Let's focus on them one at a time: breaking Δv into Δv_i, so we'll try to consider each v_i **separately**.

It's usually possible to change each v_i, so we have to look at every one of them.

One problem, though: how can we treat each **derivative** separately? Each Δv_i will move our position, which can change a different derivative v_k: they can **affect** each other.

7.X.6 Review: Planar Approximation

We'll resolve this the same way we did in chapter 3, **gradient** descent: by taking advantage of the "planar approximation".

The solution is this: assume your function is **smooth**. The **smaller** a step you take, the **less** your derivative has a chance to change.

This isn't true for big steps, but eventually, if your step is small enough, then the derivative will barely change.

Example: Take $f(x) = x^2$.

- If we go from $x = 1 \to 2$, then our derivative goes from $f'(x) = 2 \to 4$.
- Let's **shrink** our step. We go from $x = 1 \to 1.01$, our derivative goes from $f'(x) = 2 \to 2.02$.
  - Our derivative is almost the same!

if we take a small enough step Δv_i, then, if our function is **smooth**, then the derivative will hardly change!

You could imagine repeatedly shrinking the size of our step, until the change in the derivatives is basically unnoticeable.

So, if we zoom in enough (shrink the scale of change), then we can **pretend** the derivative is **constant**.

Concept 23

If you have a **smooth function**, then...

If you take sufficiently **small steps**, then you can treat the derivatives as **constant**.

Clarification

This section is **optional**.

We can describe "sufficiently small steps" in a more mathematical way:

Our goal is for $f'(x)$ to be **basically constant**: it doesn't change much. $\Delta f'(x)$ is **small**.

Let's say it can't change more then δ.

If you want

- $\Delta f'(x)$ to be very small ($|\Delta f'(x)| < \delta$)
- It has been proven that...
  - can take a small enough step $|\Delta x| < \epsilon$, and to get that result.

One way to describe this is to say that our function is (locally) **flat**: it looks like some kind of plane/hyperplane.

The word "locally" represents the small step size: we stay in the "local area".

Clarification 24

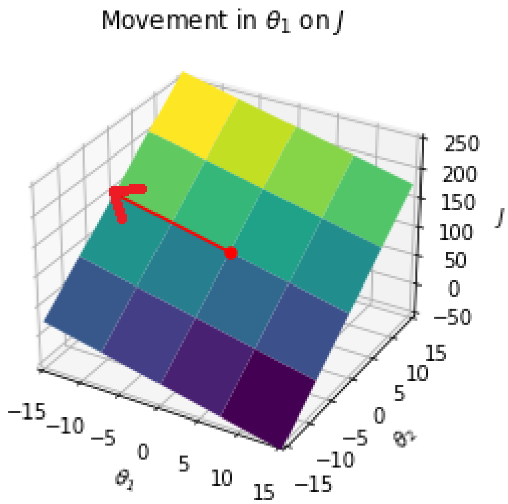
Why is this **true**? Because a **hyperplane** can be represented using our **linear** function

$$f(x) \approx \theta^T x + \theta_0 = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_m x_m$$

If we take a derivative:

$$\frac{\partial f}{\partial x_i} = \theta_i$$

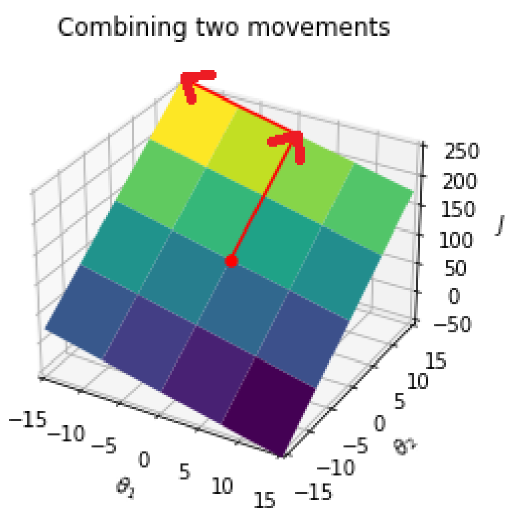
That derivative is a **constant**! It's doesn't change based on **position**.

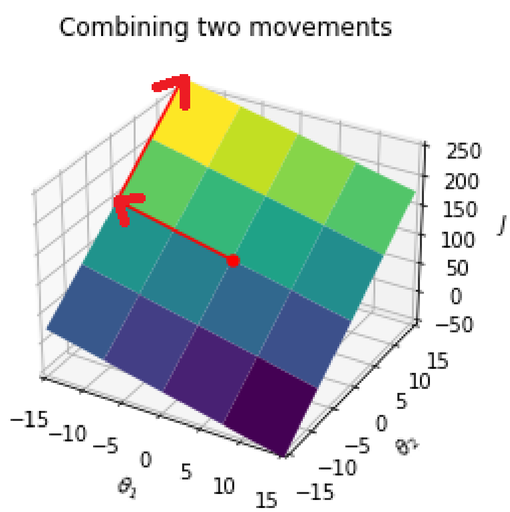


If we take very small steps, we can approximate our function as **flat**.

Why does this help? If our derivative doesn't **change**, we can combine multiple steps You can take multiple steps Δv_i and the order doesn't matter.

So, you can combine your steps or separate them easily.





We can break up our big step into two smaller steps that are truly independent: order doesn't matter.

With that, we can add up all of our changes:

$$\Delta s = \Delta s_{\text{from } v_1} + \Delta s_{\text{from } v_2} + \cdots + \Delta s_{\text{from } v_m} \tag{7.70}$$

7.X.7 Our scalar/vector derivative

From this, we can get an **approximated** version of the MV chain rule.

Definition 25

The **multivariable chain rule approximation** looks similar to the multivariable chain rule, but for finite changes Δx_i.

In 3-D, we get

$$\Delta f = \overbrace{\frac{\partial f}{\partial x}\Delta x}^{x\text{ component}} + \overbrace{\frac{\partial f}{\partial y}\Delta y}^{y\text{ component}} + \overbrace{\frac{\partial f}{\partial z}\Delta z}^{z\text{ component}}$$

In general, we have

$$\Delta f = \sum_{i=1}^{m} \overbrace{\frac{\partial f}{\partial x_i}\Delta x_i}^{x_i\text{ component}}$$

This function lets us add up the effect each component has on our output, using **derivatives**.

This gives us what we're looking for:

$$\Delta s \approx \sum_{i=1}^{m} \frac{\partial s}{\partial v_i}\Delta v_i \tag{7.71}$$

If we circle back around to our original approximation:

$$\sum_{i=1}^{m} \frac{\partial s}{\partial v_i}\Delta v_i = \frac{\partial s}{\partial v} \star \overbrace{\begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix}}^{\Delta v} \tag{7.72}$$

When we look at the left side, we're multiplying pairs of components, and then adding them. That sounds similar to a **dot product**.

$$\sum_{i=1}^{m} \frac{\partial s}{\partial v_i} \Delta v_i \quad = \quad \overbrace{\begin{bmatrix} \partial s/\partial v_1 \\ \partial s/\partial v_2 \\ \vdots \\ \partial s/\partial v_m \end{bmatrix}}^{\partial s/\partial v} \cdot \overbrace{\begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix}}^{\Delta v} \tag{7.73}$$

This gives us our derivative: it contains all of the **element-wise** derivatives we need, and in a **useful** form!

Definition 26

If s is a **scalar** and v is an $(m \times 1)$ **vector**, then we define the **derivative** or **gradient** $\partial s/\partial v$ as fulfilling:

$$\Delta s = \frac{\partial s}{\partial v} \cdot \Delta v$$

Or, equivalently,

$$\Delta s = \left(\frac{\partial s}{\partial v}\right)^T \Delta v$$

Thus, our derivative must be an $(m \times 1)$ vector

$$\frac{\partial s}{\partial v} \quad = \quad \begin{bmatrix} \partial s/\partial v_1 \\ \partial s/\partial v_2 \\ \vdots \\ \partial s/\partial v_m \end{bmatrix} \quad = \quad \begin{bmatrix} \frac{\partial s}{\partial v_1} \\ \frac{\partial s}{\partial v_2} \\ \vdots \\ \frac{\partial s}{\partial v_m} \end{bmatrix}$$

We can see the shapes work out in our matrix multiplication:

$$\overbrace{\Delta s}^{(1\times 1)} \quad = \overbrace{\left(\frac{\partial s}{\partial v}\right)^T}^{(1\times m)} \overbrace{\Delta v}^{(m\times 1)} \tag{7.74}$$

7.X.8 Vector derivative: a scalar input, vector output

Now, we want to try the flipped version: we swap our vector and our scalar.

$$\frac{\partial(\text{Vector})}{\partial(\text{Scalar})} = \frac{\partial w}{\partial s} \tag{7.75}$$

We'll take s to be our scalar, and w to be our vector. So, our input is a **scalar**, and our output is a **vector**.

Note that we're using vector w instead of v this time: this will be helpful for our vector/vector derivative: we can use both.

$$\Delta s \longrightarrow \boxed{\text{f}} \longrightarrow \Delta w \tag{7.76}$$

Written explicitly, like before:

$$\Delta s \longrightarrow \overbrace{\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix}}^{\Delta w} \tag{7.77}$$

We have 1 **input**, that can affect n different **outputs**. So, our derivative needs to have n elements.

Again, let's look at our **approximation** rule:

$$\Delta w \approx \frac{\partial w}{\partial s} \star \Delta s \qquad \text{or} \qquad \overbrace{\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix}}^{\Delta w} \approx \frac{\partial w}{\partial s} \star \Delta s \tag{7.78}$$

Here, we can't do a **dot product**: we're multiplying our derivative by a **scalar**. Plus, we'd get the **same shape** as before: we might **mix up** our derivatives.

7.X.9 Working with the vector derivative

How do we get each of our terms Δw_i?

Well, each term is **separately** affected by Δs: we have our terms $\partial w_i / \partial s$.

So, if we take these terms **individually**, treating it as a scalar derivative, we get:

If you're ever confused with matrix math, thinking about individual elements is often a good way to figure it out!

$$\Delta w_i = \frac{\partial w_i}{\partial s} \Delta s \tag{7.79}$$

Since we only have **one** input, we don't have to worry about **planar** approximations: we only take one step, in the s direction.

In our matrix, we get:

$$w = \begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix} = \begin{bmatrix} \Delta s(\partial w_1/\partial s) \\ \Delta s(\partial w_2/\partial s) \\ \vdots \\ \Delta s(\partial w_n/\partial s) \end{bmatrix} \tag{7.80}$$

This works out for our equation above!

It could be tempting to think of our derivative $\partial w/\partial s$ as a **column vector**: we just take w and just differentiate each element. Easy!

In fact, this *is* a valid convention. However, this conflicts with our previous derivative: they're both column vectors!

Not only is it **confusing**, but it also will make it harder to do our **vector/vector** derivative.

So, what do we do? We refer back to the equation we used last time:

$$\Delta w = \left(\frac{\partial w}{\partial s}\right)^T \Delta s \tag{7.81}$$

We take the **transpose**! That way, one derivative is a column vector, and the other is a row vector. And, we know that this equation works out from the work we just did.

$$\Delta w = \begin{bmatrix} \frac{\partial w_1}{\partial s}, & \frac{\partial w_2}{\partial s}, & \cdots & \frac{\partial w_n}{\partial s} \end{bmatrix}^T \Delta s \tag{7.82}$$

Clarification 27

We mentioned that it is a valid **convention** to have that **vector derivative** be a **column vector**, and have our **gradient** be a **row vector**.

This is **not** the convention we will use in this class - you will be confused if we try!

That means, for whatever **notation** we use here, you might see the **transposed** version elsewhere. They mean exactly the **same** thing!

$$\overbrace{\Delta w}^{(n\times 1)} = \overbrace{\left(\frac{\partial w}{\partial s}\right)^T}^{(n\times 1)} \overbrace{\Delta s}^{(1\times 1)} \tag{7.83}$$

As we can see, the dimensions check out.

Definition 28
If s is a **scalar** and w is an $(n \times 1)$ **vector**, then we define the **vector derivative** $\partial w/\partial s$ as fulfilling:

$$\Delta w = \left(\frac{\partial w}{\partial s}\right)^T \Delta s$$

Thus, our derivative must be a $(1 \times n)$ vector

$$\frac{\partial w}{\partial s} = \begin{bmatrix} \frac{\partial w_1}{\partial s}, & \frac{\partial w_2}{\partial s}, & \cdots & \frac{\partial w_n}{\partial s} \end{bmatrix}$$

7.X.10 Vectors and vectors: vector input, vector output

We'll be combining our two previous derivatives:

$$\frac{\partial(\text{Vector})}{\partial(\text{Vector})} = \frac{\partial w}{\partial v} \tag{7.84}$$

v and w are both **vectors**: thus, input and output are both **vectors**.

$$\Delta v \longrightarrow \boxed{f} \longrightarrow \Delta w \tag{7.85}$$

Written out, we get:

$$\overbrace{\begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix}}^{\Delta v} \longrightarrow \overbrace{\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix}}^{\Delta w} \tag{7.86}$$

Something pretty complicated! We have m inputs and n outputs. Every input can interact with every output.

So, our derivative needs to have mn different elements. That's a lot!

7.X.11 The vector/vector derivative

We return to our rule from before. We'll skip the star notation, and jump right to the equation we've gotten for both of our two previous derivatives:

Hopefully, since we're combining two different derivatives, we should be able to use the same rule here.

$$\Delta w = \left(\frac{\partial w}{\partial v}\right)^{\mathsf{T}} \Delta v \tag{7.87}$$

With mn different elements, this could get messy very fast. Let's see if we can focus on only **part** of our problem:

$$\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix} = \left(\frac{\partial w}{\partial v}\right)^{\mathsf{T}} \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix} \tag{7.88}$$

One input

We could try focusing on just a single **input** or a single **output**, to simplify things. Let's start with a single v_i.

$$\overbrace{\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix}}^{\Delta w \text{ from } v_i} = \left(\frac{\partial w}{\partial v_i}\right)^{\mathsf{T}} \Delta v_i \tag{7.89}$$

We now have a simpler case: ∂Vector/∂Scalar. We're familiar with this case!

$$\frac{\partial w}{\partial v_i} = \begin{bmatrix} \frac{\partial w_1}{\partial v_i}, & \frac{\partial w_2}{\partial v_i}, & \cdots & \frac{\partial w_n}{\partial v_i} \end{bmatrix} \tag{7.90}$$

We get a vector. What if the **output** is a scalar instead?

One output

$$\Delta w_j = \left(\frac{\partial w_j}{\partial v}\right)^{\mathsf{T}} \begin{bmatrix} \Delta v_1 \\ \Delta v_2 \\ \vdots \\ \Delta v_m \end{bmatrix} \tag{7.91}$$

We have ∂Scalar/∂Vector:

$$\frac{\partial w_j}{\partial v} = \begin{bmatrix} \partial w_j/\partial v_1 \\ \partial w_j/\partial v_2 \\ \vdots \\ \partial w_j/\partial v_m \end{bmatrix} \tag{7.92}$$

So, our vector-vector derivative is a **generalization** of the two derivatives we did before!

It seems that extending along the **vertical** axis changes our v_i value, while moving along the **horizontal** axis changes our w_j value.

7.X.12 General derivative

You might have a hint of what we get: one derivative stretches us along **one** axis, the other along the **second**.

To prove it to ourselves, we can **combine** these concepts. We'll handle solve as if we have one vector, and then **substitute** in the second one.

Concept 29

One way to **simplify** our work is to treat **vectors** as **scalars**, and then convert them back into **vectors** after applying some math.

We have to be careful - any operation we apply to the **scalar**, has to match how the **vector** would behave.

This is **equivalent** to if we just focused on one scalar inside our vector, and then stacked all those scalars back into the vector.

This isn't just a cute trick: it relies on an understanding that, at its **basic** level, we're treating **scalars** and **vectors** and **matrices** as the same type of object: a structured array of numbers.

We'll get into "arrays" later.

As always, our goal is to **simplify** our work, so we can handle each piece of it.

- We treat Δv as a scalar so we can get the simplified derivative.

$$\Delta w = \left(\frac{\partial w}{\partial v}\right)^{T} \Delta v \tag{7.93}$$

We'll only expand **one** of our vectors, since we know how to manage **one** of them.

$$\begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \vdots \\ \Delta w_n \end{bmatrix} = \left(\frac{\partial w}{\partial v}\right)^T \Delta v \tag{7.94}$$

This time, notice that we **didn't** simplify v to v_i. We didn't **remove** the other elements - we still have a full **vector**. But, let's treat it as if it *were* a scalar.

This comes out to:

$$\frac{\partial w}{\partial v} = \overbrace{\begin{bmatrix} \frac{\partial w_1}{\partial v}, & \frac{\partial w_2}{\partial v}, & \cdots & \frac{\partial w_n}{\partial v} \end{bmatrix}}^{\text{Column j matches } w_j} \tag{7.95}$$

- Our "answer" is a row vector. But, each of those derivatives is a **column** vector!

Now that we've taken care of ∂w_j (one for each column), we can expand our derivatives in terms of ∂v_i.

First, for w_1:

$$\frac{\partial w}{\partial v} = \left.\overbrace{\begin{bmatrix} \begin{bmatrix} \frac{\partial w_1}{\partial v_1} \\ \frac{\partial w_1}{\partial v_2} \\ \vdots \\ \frac{\partial w_1}{\partial v_m} \end{bmatrix}, & \frac{\partial w_2}{\partial v}, & \cdots & \frac{\partial w_n}{\partial v} \end{bmatrix}}^{\text{Column j matches } w_j}\right\} \text{Row i matches } v_i \tag{7.96}$$

And again, for w_2:

$$\frac{\partial w}{\partial v} = \left.\overbrace{\begin{bmatrix} \begin{bmatrix} \frac{\partial w_1}{\partial v_1} \\ \frac{\partial w_1}{\partial v_2} \\ \vdots \\ \frac{\partial w_1}{\partial v_m} \end{bmatrix}, & \begin{bmatrix} \frac{\partial w_2}{\partial v_1} \\ \frac{\partial w_2}{\partial v_2} \\ \vdots \\ \frac{\partial w_2}{\partial v_m} \end{bmatrix}, & \cdots & \frac{\partial w_n}{\partial v} \end{bmatrix}}^{\text{Column j matches } w_j}\right\} \text{Row i matches } v_i \tag{7.97}$$

And again, for w_n:

$$\frac{\partial w}{\partial v} = \overbrace{\left[\begin{bmatrix} \frac{\partial w_1}{\partial v_1} \\ \frac{\partial w_1}{\partial v_2} \\ \vdots \\ \frac{\partial w_1}{\partial v_m} \end{bmatrix}, \begin{bmatrix} \frac{\partial w_2}{\partial v_1} \\ \frac{\partial w_2}{\partial v_2} \\ \vdots \\ \frac{\partial w_2}{\partial v_m} \end{bmatrix}, \cdots \begin{bmatrix} \frac{\partial w_n}{\partial v_1} \\ \frac{\partial w_n}{\partial v_2} \\ \vdots \\ \frac{\partial w_n}{\partial v_m} \end{bmatrix} \right]}^{\text{Column } j \text{ matches } w_j} \left.\vphantom{\begin{bmatrix} a \\ a \\ a \\ a \\ a \\ a \end{bmatrix}}\right\} \text{Row } i \text{ matches } v_i \tag{7.98}$$

We have column vectors in our row vector... let's just combine them into a **matrix**.

Definition 30

If

- v is an $(m \times 1)$ **vector**
- w is an $(n \times 1)$ **vector**

Then we define the **vector derivative** $\partial w / \partial v$ as fulfilling:

$$\Delta w = \left(\frac{\partial w}{\partial s} \right)^T \Delta s$$

Thus, our derivative must be a $(1 \times n)$ vector

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

This general form can be used for **any** of our matrix derivatives.

So, our matrix can represent any **combination** of two elements! We just assign each **row** to a v_i component, and each **column** with a w_j component.

7.X.13 More about the vector/vector derivative

Let's show a specific example: w is (3×1), v is (2×1).

$$\frac{\partial w}{\partial v} = \begin{array}{cl} \begin{bmatrix} \overbrace{\frac{\partial w_1}{\partial v_1}}^{w_1} & \overbrace{\frac{\partial w_2}{\partial v_1}}^{w_2} & \overbrace{\frac{\partial w_3}{\partial v_1}}^{w_3} \\[2em] \frac{\partial w_1}{\partial v_2} & \frac{\partial w_2}{\partial v_2} & \frac{\partial w_3}{\partial v_2} \end{bmatrix} & \begin{array}{l} \big\}\, v_1 \\[3em] \big\}\, v_2 \end{array} \end{array} \tag{7.99}$$

Another way to describe the general case:

Notation 31

Our matrix $\partial w / \partial v$ is entirely filled with **scalar derivatives**

$$\frac{\partial w_j}{\partial v_i} \tag{7.100}$$

Where any one **derivative** is stored in

- Row i
  - m rows total
- Column j
  - n columns total

We can also compress it along either axis (just like how we did to derive this result):

Notation 32

Our matrix $\partial w/\partial v$ can be written as

$$\frac{\partial w}{\partial v} = \overbrace{\begin{bmatrix} \frac{\partial w_1}{\partial v}, & \frac{\partial w_2}{\partial v}, & \cdots & \frac{\partial w_n}{\partial v} \end{bmatrix}}^{\text{Column j matches } w_j}$$

or

$$\frac{\partial w}{\partial v} = \left.\begin{bmatrix} \frac{\partial w}{\partial v_1} \\ \frac{\partial w}{\partial v_2} \\ \vdots \\ \frac{\partial w}{\partial v_m} \end{bmatrix}\right\} \text{Row i matches } v_i$$

These compressed forms will be useful for deriving our new and final derivatives, **matrix-scalar** pairs.

7.X.14 Derivative: matrix/scalar

Now, we have our general form for creating derivatives.

We'll get our derivative of the form

$$\frac{\partial(\text{Matrix})}{\partial(\text{Scalar})} = \frac{\partial M}{\partial s} \tag{7.101}$$

We have a matrix M in the shape $(r \times k)$ and a scalar s. Our **input** is a **scalar**, and our **output** is a **matrix**.

$$M = \begin{bmatrix} m_{11} & m_{12} & \cdots & m_{1r} \\ m_{21} & m_{22} & \cdots & m_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ m_{k1} & m_{k2} & \cdots & m_{kr} \end{bmatrix} \tag{7.102}$$

This may seem concerning: before, we divided **inputs** across **rows**, and **outputs** across **columns**. But in this case, we have **no** input axes, and **two** output axes.

Well, let's try to make this work anyway.

What did we do before, when we didn't know how to handle a **new** derivative? We compared it to **old** versions: we built our vector/vector case using the vector/scalar case and the scalar/vector case.

We did this by **compressing** one of our *vectors* into a *scalar* temporarily: this works, because we want to treat each of these objects the **same way**.

We don't know how to work with Matrix/Scalar, but what's the **closest** thing we do know? **Vector/Scalar**.

How do we accomplish that? As we saw above, a matrix is a **vector** of **vectors**. We could turn it into a **vector** of **scalars**.

Concept 33

A **matrix** can be thought of as a **column vector** of **row vectors** (or vice versa).

So, we can use our earlier technique and convert the **row vectors** into **scalars**.

We'll replace the **row vectors** in our matrix with **scalars**.

$$M = \begin{bmatrix} M_1 \\ M_2 \\ \vdots \\ M_k \end{bmatrix} \tag{7.103}$$

Now, we can pretend our matrix is a vector! We've got a derivative for that:

$$\frac{\partial M}{\partial s} = \begin{bmatrix} \frac{\partial M_1}{\partial s} & \frac{\partial M_2}{\partial s} & \cdots & \frac{\partial M_r}{\partial s} \end{bmatrix} \tag{7.104}$$

Aha - we have the same form that we did for our vector/vector derivative! Each derivative is a column vector. Let's expand it out:

$$\frac{\partial M}{\partial s} = \left.\overbrace{\begin{bmatrix} \begin{bmatrix} \frac{\partial m_{11}}{\partial s} \\ \frac{\partial m_{12}}{\partial s} \\ \vdots \\ \frac{\partial m_{1r}}{\partial s} \end{bmatrix}, & \begin{bmatrix} \frac{\partial m_{21}}{\partial s} \\ \frac{\partial m_{22}}{\partial s} \\ \vdots \\ \frac{\partial m_{2r}}{\partial s} \end{bmatrix}, & \cdots & \begin{bmatrix} \frac{\partial m_{k1}}{\partial s} \\ \frac{\partial m_{k2}}{\partial s} \\ \vdots \\ \frac{\partial m_{kr}}{\partial s} \end{bmatrix} \end{bmatrix}}^{\text{Column } j \text{ matches } m_{j?}}\right\}\text{Row } i \text{ matches } m_{?i} \tag{7.105}$$

Definition 34

If M is a matrix in the shape $(r \times k)$ and s is a scalar,

Then we define the **matrix derivative** $\partial M/\partial s$ as the $(k \times r)$ matrix:

$$\frac{\partial M}{\partial s} = \left. \overbrace{\begin{bmatrix} \frac{\partial m_{11}}{\partial s} & \frac{\partial m_{21}}{\partial s} & \cdots & \frac{\partial m_{k1}}{\partial s} \\ \frac{\partial m_{12}}{\partial s} & \frac{\partial m_{22}}{\partial s} & \cdots & \frac{\partial m_{k2}}{\partial s} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial m_{1r}}{\partial s} & \frac{\partial m_{2r}}{\partial s} & \cdots & \frac{\partial m_{kr}}{\partial s} \end{bmatrix}}^{\text{Column } j \text{ matches } m_{j?}} \right\} \text{Row } i \text{ matches } m_{?i}$$

This matrix has the transpose of the shape of M.

7.X.15 Derivative: scalar/matrix

We'll get our derivative of the form

$$\frac{\partial(\text{Scalar})}{\partial(\text{Matrix})} = \frac{\partial s}{\partial M} \tag{7.106}$$

We have a matrix M in the shape $(r \times k)$ and a scalar s. Our **input** is a **matrix**, and our **output** is a **scalar**.

Let's do what we did last time: break it into **row vectors**.

$$M = \begin{bmatrix} M_1 \\ M_2 \\ \vdots \\ M_k \end{bmatrix} \tag{7.107}$$

The gradient for this "vector" gives us a **column vector**:

$$\frac{\partial s}{\partial M} = \begin{bmatrix} \dfrac{\partial s}{\partial M_1} \\[2ex] \dfrac{\partial s}{\partial M_2} \\[2ex] \vdots \\[2ex] \dfrac{\partial s}{\partial M_k} \end{bmatrix} \tag{7.108}$$

This time, each derivative is a **row vector**. Let's **expand**:

$$\frac{\partial s}{\partial M} = \begin{bmatrix} \begin{bmatrix} \dfrac{\partial s}{\partial m_{11}} & \dfrac{\partial s}{\partial m_{12}} & \cdots & \dfrac{\partial s}{\partial m_{1r}} \end{bmatrix} \\[2ex] \begin{bmatrix} \dfrac{\partial s}{\partial m_{21}} & \dfrac{\partial s}{\partial m_{22}} & \cdots & \dfrac{\partial s}{\partial m_{2r}} \end{bmatrix} \\[2ex] \vdots \\[2ex] \begin{bmatrix} \dfrac{\partial s}{\partial m_{k1}} & \dfrac{\partial s}{\partial m_{k2}} & \cdots & \dfrac{\partial s}{\partial m_{kr}} \end{bmatrix} \end{bmatrix} \tag{7.109}$$

Definition 35

If M is a matrix in the shape $(r \times k)$ and s is a scalar,

Then we define the **matrix derivative** $\partial s/\partial M$ as the $(r \times k)$ matrix:

$$\frac{\partial s}{\partial M} = \begin{array}{cl} \overbrace{}^{\text{Column } j \text{ matches } m_{?j}} & \\ \begin{bmatrix} \dfrac{\partial s}{\partial m_{11}} & \dfrac{\partial s}{\partial m_{12}} & \cdots & \dfrac{\partial s}{\partial m_{1r}} \\[2ex] \dfrac{\partial s}{\partial m_{21}} & \dfrac{\partial s}{\partial m_{22}} & \cdots & \dfrac{\partial s}{\partial m_{2r}} \\[2ex] \vdots & \vdots & \ddots & \vdots \\[2ex] \dfrac{\partial s}{\partial m_{k1}} & \dfrac{\partial s}{\partial m_{k2}} & \cdots & \dfrac{\partial s}{\partial m_{kr}} \end{bmatrix} & \left.\vphantom{\begin{bmatrix} 1 \\[2ex] 1 \\[2ex] 1 \\[2ex] 1 \\[2ex] 1 \\[2ex] 1 \end{bmatrix}}\right\} \text{Row } i \text{ matches } m_{i?} \end{array}$$

This matrix has the same shape as M.

7.X.16 Other Derivatives

After these, you might ask yourself, what about other derivative combinations?

$$\frac{\partial v}{\partial M}? \qquad \frac{\partial M}{\partial v}? \qquad \frac{\partial M}{\partial M^2}? \tag{7.110}$$

There's a problem with all of these: the total number of axes is **too large**.

What do we mean by an **axis**?

Definition 36

An **axis** is one of the **indices** we can adjust to get a different scalar in our array: each index is a "direction" we can move along our object to **store** numbers.

- A **scalar** has **0 axes**: we only have one scalar, so we have no indices to adjust.

- A **vector** has **1 axis**: we can get different scalars by moving **vertically** (for column vectors): v_1, v_2, v_3...

$$\left.\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_m \end{bmatrix}\right\} \text{Axis 1}$$

- A **matrix** has **2 axes**: we can move **horizontally** or **vertically**.

$$\overbrace{\begin{bmatrix} m_{11} & m_{12} & \cdots & m_{1r} \\ m_{21} & m_{22} & \cdots & m_{2r} \\ \vdots & \vdots & \ddots & \vdots \\ m_{k1} & m_{k2} & \cdots & m_{kr} \end{bmatrix}}^{\text{Axis 2: Columns}} \left.\vphantom{\begin{bmatrix} m \\ m \\ \vdots \\ m \end{bmatrix}}\right\} \text{Axis 1: Rows}$$

These can also be called **dimensions**.

Why does the number of **axes** matter? Remember that, so far, for our derivatives, each axis of the output represented an axis of the **input** or **output**.

Note that last bit: we're saying a vector has one dimension. Can't a vector have **multiple** dimensions? Jump to 7.X.17 for a clarification.

$$\frac{\partial w}{\partial v} = \begin{array}{c} \overbrace{\hphantom{\dfrac{\partial w_1}{\partial v_1} \quad \dfrac{\partial w_2}{\partial v_1} \quad \cdots \quad \dfrac{\partial w_n}{\partial v_1}}}^{\text{Column j: vertical axis of } w} \\ \left.\begin{bmatrix} \dfrac{\partial w_1}{\partial v_1} & \dfrac{\partial w_2}{\partial v_1} & \cdots & \dfrac{\partial w_n}{\partial v_1} \\ \dfrac{\partial w_1}{\partial v_2} & \dfrac{\partial w_2}{\partial v_2} & \cdots & \dfrac{\partial w_n}{\partial v_2} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial w_1}{\partial v_m} & \dfrac{\partial w_2}{\partial v_m} & \cdots & \dfrac{\partial w_n}{\partial v_m} \end{bmatrix}\right\} \text{Row i: vertical axis of } v \end{array}$$

The way we currently build derivatives, we try to get **every pair** of input-output variables: we use **one** axis for each **axis** of either the **input** or **output**.

Take some examples:

- $\partial s/\partial v$: we need one axis to represent each term v_i.
  - 0 axis + 1 axis $\rightarrow$ 1 axis: the output is a (column) **vector**.
- $\partial v/\partial s$: we need one axis to represent each term w_j.
  - 1 axis + 0 axis $\rightarrow$ 1 axis: the output is a (row) **vector**.
- $\partial w/\partial v$: we need one axis to represent each term v_i, and another to represent each term w_j.
  - 1 axis + 1 axis $\rightarrow$ 2 axes: the output is a **matrix**.
- $\partial M/\partial s$: we need one axis to represent the rows of M, and another to represent the columns of M.
  - 2 axis + 0 axis $\rightarrow$ 2 axes: the output is a **matrix**.
- $\partial s/\partial M$: we need one axis to represent the rows of M, and another to represent the columns of M.
  - 0 axis + 2 axis $\rightarrow$ 2 axes: the output is a **matrix**.

Notice the pattern!

Concept 37

A **matrix derivative** needs to be able to account for each type/**index** of variable in the input **and** the output.

So, if the **input** x has m axes, and the **output** y has n axes, then the derivative needs to have the same **total** number:

$$\text{Axes}\left(\frac{\partial y}{\partial x}\right) = \text{Axes}(y) + \text{Axes}(x) \tag{7.111}$$

This is where our problem comes in: if we have a vector and a matrix, we need **3 axes**! That's more than a matrix.

7.X.17 Dimensions (Optional)

Here's a quick aside to clear up possible confusion from the last section: our definition of axes and "dimensions".

We said a vector has 1 axis, or "dimension" of movement. But, can't a vector have **multiple** dimensions?

Clarification 38

We have two competing definition of **dimension**: this explains why we can say seemingly conflicting things about derivatives.

So far, by "**dimension**", we mean, "a separate **value** we can **adjust**".

- Under this definition, a $(k \times 1)$ column **vector** has k dimensions: it contains k different scalars we can **adjust**.

$$\left.\begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_k \end{bmatrix}\right\} \text{We can adjust each of our } k \text{ scalars.}$$

- You might say a $(k \times r)$ **matrix** has k dimensions, too: based on the **dimensionality** of its column vectors.
  - Since we prioritize the size of the vectors, we could say this is a very "vector-centric" definition.

In this section, by "dimension", we mean, "an **index** we can **adjust** (move along) to find another scalar.

- Under this definition, a $(k \times 1)$ column **vector** has 1 dimension: we only have 1 axis of **movement**.
- You might say a $(k \times r)$ **matrix** has 2 dimensions: a **horizontal** one, and a **vertical** one.
  - This **definition** is the kind we use in the following sections.

If you jumped here from 7.X.16, feel free to follow this link back. Otherwise, continue on.

7.X.18 Dealing with Tensors

If a vector looks like a "**line**" of numbers, and a matrix looks like a "**rectangle**" of numbers, then a **3-axis** version would look like a "**box**" of numbers. How do we make sense of this?

First, what is this kind of object we've been working with? Vectors, matrices, etc. This collection of numbers, organized neatly, is an **array**.

Definition 39

An **array** of objects is an **ordered sequence** of them, stored together.

The most typical example is a **vector**: an ordered sequence of **scalars**.

A **matrix** can be thought of as a **vector** of **vectors**. For example: it could be a row vector, where every column is a column vector.

So, we think of a matrix as a "two-dimensional array".

We can extend this to any number of dimensions. We call this kind of generalization a **tensor**.

Definition 40

In **machine learning**, we think of a **tensor** as a "**multidimensional array**" of numbers.

Each "dimension" is what we have been calling an "**axis**".

A tensor with c axes is called a **c-Tensor**.

Note that what we call a tensor is **not** a mathematical (or physics) tensor: we do not often use the "tensor product", or other tensor properties.

Our tensor can be better thought of as a "**generalized matrix**".

Example: The 3-D box we are talking about above is called a 3-Tensor. We can simply think of it as a stack of matrices.

How do we handle **tensors**? Simply, we convert them into regular **matrices** in some way, and then do our usual math on them:

These examples aren't especially important, but you'll see different variations in different softwares!

- If a tensor has a pattern of zeroes, we might be able to flatten it into a matrix.
  - For example, if we wanted to flatten a matrix into a vector (which we sometimes do!), we could do

$$\begin{bmatrix} 3 & 0 & 0 \\ 0 & 9 & 0 \\ 0 & 0 & 4 \end{bmatrix} \rightarrow \begin{bmatrix} 3 \\ 9 \\ 4 \end{bmatrix} \tag{7.112}$$

- We can also flatten it into a matrix or vector by placing the layers next to each other.
- We cleverly do regular matrix multiplication in a way that's compatible with our tensors.
  - Note that tensors do not have a matrix multiplication-like multiplication by default: several have been designed, however.

- We ignore the structure of the tensor, and just look at the individual elements: we take the scalar chain rule for each of them, without respecting the overall tensor.

Clarification 41

If you look into **derivatives** that would result in a **3-tensor** or higher, you'll find that there's no consistent **notation** for what these derivatives look like.

These techniques are part of why: there are **different** approaches for how to approach these objects.

As we will see in the next chapter, tensors are **very** important to machine learning.

However, because they don't have a natural matrix multiplication, we'll try to convert it into a matrix in most cases.

7.X.19 The loss derivative

Finally, we apply this to our common derivatives in section 7.5.

$$\overbrace{\frac{\partial \mathcal{L}}{\partial A^L}}^{(n^L \times 1)} \tag{7.113}$$

Loss is not given, so we can't compute it. But, we can get the shape: we have a scalar/vector derivative, so the shape matches A^L.

Notation 42

Our derivative

$$\frac{\partial \mathcal{L}}{\partial A^L} \tag{7.114}$$

Is a scalar/vector derivative, and thus the shape $(n^L \times 1)$.

7.X.20 The weight derivative

$$\overbrace{\frac{\partial Z^\ell}{\partial W^\ell}}^{(m^\ell \times 1)?} \tag{7.115}$$

This derivative is difficult - it's a derivative in the form vector/matrix. With **three** axes, we might imagine representing as a 3-tensor.

In fact, this can be manipulated into multiple different interesting **shapes** based on your **interpretation**: as we mentioned, there's no consistent rule for these variables.

But, our goal is to use this for the **chain rule**: so, we need to make the shapes **match**. This is why we do that strange transposing for our complete derivative.

$$\frac{\partial \mathcal{L}}{\partial W^\ell} = \overbrace{\frac{\partial Z^\ell}{\partial W^\ell}}^{\text{Weight link}} \cdot \overbrace{\left(\frac{\partial \mathcal{L}}{\partial Z^\ell}\right)^\mathsf{T}}^{\text{Other layers}} \tag{7.116}$$

Our problem is we have **too many axes**: the easiest way to resolve this to **break up** our matrix. So, for now, we focus on only **one neuron** at a time: it has a column vector W_i.

For simplicity, we're gonna ignore the ℓ notation: just be careful, because Z and A are from two different layers!

$$W = \begin{bmatrix} W_1 & W_2 & \cdots & W_n \end{bmatrix} \tag{7.117}$$

Notice that, this time, we broke it into **column vectors**, rather than row vectors: each neuron's **weights** are represented by a column vector.

We'll ignore everything except W_i.

$$W_i = \begin{bmatrix} w_{1i} \\ w_{2i} \\ \vdots \\ w_{mi} \end{bmatrix} \tag{7.118}$$

Finally, we get into our equation: notice that a **single** neuron has only **one** pre-activation z_i, so we don't need the whole vector.

$$z_i = W_i^\mathsf{T} A \tag{7.119}$$

Wait: there's something to notice, right off the bat. z_i is **only** a function of W_i: that means the derivative for every other term $\partial/\partial W_k$ is **zero**!

For example, changing W_2 would have **no** effect on z_1.

Concept 43

The i^{th} neuron's **weights**, W_i, have **no effect** on a different neuron's **pre-activation** z_j.

So, if the **neurons** don't match, then our derivative is zero:

- i is the neuron for pre-activation z_i
- j is the j^{th} weight in a neuron.
- k is the neuron for weight vector W_k

$$\frac{\partial z_i}{\partial W_{jk}} = 0 \qquad \text{if } i \neq k$$

So, our only nonzero derivatives are

$$\frac{\partial z_i}{\partial W_{ji}}$$

With that done, let's substitute in our values:

$$z_i = \begin{bmatrix} w_{1i} & w_{2i} & \cdots & w_{mi} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \tag{7.120}$$

And we'll do our **matrix multiplication**:

$$z_i = \sum_{j=1}^{n} W_{ji} a_j \tag{7.121}$$

Finally, we can get our derivatives:

$$\frac{\partial z_i}{\partial W_{ji}} = a_j \tag{7.122}$$

So, if we combine that into a vector, we get:

$$\frac{\partial z_i}{\partial W_i} = \begin{bmatrix} \dfrac{\partial z_i}{\partial W_{1i}} \\ \\ \dfrac{\partial z_i}{\partial W_{2i}} \\ \\ \vdots \\ \\ \dfrac{\partial z_i}{\partial W_{mi}} \end{bmatrix} \tag{7.123}$$

We can use our equation:

$$\frac{\partial z_i}{\partial W_i} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} = A \tag{7.124}$$

We get a result!

What if the pre-activation z_i and weights W_k don't match? We've already seen: the derivative is 0: weights don't affect different neurons.

$$\frac{\partial z_i}{\partial W_{jk}} = 0 \qquad \text{if } i \neq k \tag{7.125}$$

We can combine these into a **zero vector**:

$$\frac{\partial z_i}{\partial W_k} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \vec{0} \qquad \text{if } i \neq k \tag{7.126}$$

So, now, we can describe all of our vector components:

$$\frac{\partial z_i}{\partial W_k} = \begin{cases} A & \text{if } i = k \\ \vec{0} & \text{if } i \neq k \end{cases} \tag{7.127}$$

These are all the elements of our matrix $\partial z_i / \partial W_k$: so, we can get our result.

$$\frac{\partial Z}{\partial W} = \begin{bmatrix} A & \vec{0} & \cdots & \vec{0} \\ \vec{0} & A & \cdots & \vec{0} \\ \vdots & \vdots & \ddots & \vec{0} \\ \vec{0} & \vec{0} & \vec{0} & A \end{bmatrix} \tag{7.128}$$

We have our result: it turns out, despite being stored in a **matrix**-like format, this is actually a **3-tensor**! Each entry of our **matrix** is a **vector**: 3 axes.

But, we don't really... *want* a tensor. It doesn't have the right shape, and we can't do matrix multiplication.

We'll solve this by **simplifying**, without losing key information.

Concept 44

For many of our "tensors" resulting from matrix derivatives, they contain **empty** rows or **redundant** information.

Based on this, we can **simplify** our tensor into a fewer-dimensional (fewer axes) object.

We can see two types of **redundancy** above:

- Every element **off** the diagonal is 0.
- Every element **on** the diagonal is the same.

Let's fix the first one: we'll go from a diagonal matrix to a column vector.

$$\begin{bmatrix} A & \vec{0} & \cdots & \vec{0} \\ \vec{0} & A & \cdots & \vec{0} \\ \vdots & \vdots & \ddots & \vec{0} \\ \vec{0} & \vec{0} & \vec{0} & A \end{bmatrix} \longrightarrow \begin{bmatrix} A \\ A \\ \vdots \\ A \end{bmatrix} \tag{7.129}$$

Then, we'll combine all of our redundant A values.

$$\begin{bmatrix} A \\ A \\ \vdots \\ A \end{bmatrix} \longrightarrow A \tag{7.130}$$

We have our big result!

Notation 45

Our derivative

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

Is a vector/matrix derivative, and thus should be a 3-tensor.

But, we have turned it into the shape $(m^\ell \times 1)$.

This is as **condensed** as we can get our information: if we compress to a scalar, we lose some of our elements.

Even with this derivative, we still have to do some clever **reshaping** to get the result we need (transposing, changing derivative order, etc.)

However, at the end, we get the right shape for our chain rule!

7.X.21 Linking Layers

$$\frac{\partial Z^\ell}{\partial A^{\ell-1}} \tag{7.131}$$

This derivative is much more manageable: it's just the derivative between a vector and a vector. Let's look at our equation again:

Ignoring superscripts ℓ, as before.

$$Z = W^T A \tag{7.132}$$

We'll use the same approach we did last time: W is a vector, and we'll focus on W_i. This will allow us to break it up **element-wise**, and get all of our **derivatives**.

We could treat W as a whole matrix, but this will give us our results without as much clutter: the only **difference** is that we would have to depict every W_i at **once**.

$$W = \begin{bmatrix} W_1 & W_2 & \cdots & W_n \end{bmatrix} \qquad W_i = \begin{bmatrix} w_{1i} \\ w_{2i} \\ \vdots \\ w_{mi} \end{bmatrix} \tag{7.133}$$

Here's our equation:

$$z_i = \begin{bmatrix} w_{1i} & w_{2i} & \cdots & w_{mi} \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix} \tag{7.134}$$

We matrix multiply:

$$z_i = \sum_{j=1}^{n} W_{ji} a_j \tag{7.135}$$

The derivative can be gotten from here -

$$\frac{\partial z_i}{\partial a_j} = W_{ji} \tag{7.136}$$

We look at our whole matrix derivative:

This notation looks a bit weird, but it's just a way to represent that all of our elements follow this pattern.

$$\frac{\partial Z}{\partial A} = \left.\overbrace{\begin{bmatrix} \ddots & \vdots & \iddots \\ \cdots & \frac{\partial z_i}{\partial a_j} & \cdots \\ \iddots & \vdots & \ddots \end{bmatrix}}^{\text{Column } j \text{ matches } z_i}\right\} \text{Row } i \text{ matches } a_j \tag{7.137}$$

Wait.

- The derivative $\partial z_i / \partial a_j$ is in the j^{th} row, i^{th} column.
- W_{ji} represents the element in the j^{th} row, i^{th} column.

They're the same matrix!

If two matrices have exactly the same shape and elements, they're the same matrix.

We get our final result:

Notation 46

Our derivative

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

Is a vector/vector derivative, and thus a matrix.

But, we have turned it into the shape $(m^\ell \times n^\ell)$.

7.X.22 Activation Function

$$\frac{\partial A^\ell}{\partial Z^\ell} \tag{7.138}$$

The last derivative is less unusual than it looks.

$$A^\ell = f(Z^\ell) \longrightarrow \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = f\left(\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix} \right) \tag{7.139}$$

We can apply our function element-wise:

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \begin{bmatrix} f(z_1) \\ f(z_2) \\ \vdots \\ f(z_n) \end{bmatrix} \tag{7.140}$$

As we can see, each activation is a function of only **one** pre-activation.

Concept 47

Each **activation** is only affected by the **pre-activation** in the **same neuron**.

So, if the **neurons** don't match, then our derivative is zero:

- i is the neuron for pre-activation z_i
- j is the neuron for activation a_j

$$\frac{\partial a_j}{\partial z_i} = 0 \qquad \text{if } i \neq j$$

So, our only nonzero derivatives are

$$\frac{\partial a_j}{\partial z_i}$$

As for our remaining term, we'll describe any row of the above vectors:

$$a_i = f(z_i) \tag{7.141}$$

Our derivative is:

$$\frac{\partial a_i}{\partial z_i} = f'(z_i) \tag{7.142}$$

In general, including the non-diagonals:

$$\frac{\partial a_i}{\partial z_i} = \begin{cases} f'(z_i) & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{7.143}$$

This gives us our result:

Notation 48

Our derivative

$$\overbrace{\frac{\partial A^\ell}{\partial Z^\ell}}^{(n^\ell \times n^\ell)} = \left.\overbrace{\begin{bmatrix} f'(z_1^\ell) & 0 & 0 & \cdots & 0 \\ 0 & f'(z_2^\ell) & 0 & \cdots & 0 \\ 0 & 0 & f'(z_3^\ell) & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & f'(z_n^\ell) \end{bmatrix}}^{\text{Column } j \text{ matches } a_j}\right\} \text{Row } i \text{ matches } z_i \tag{7.144}$$

Is a vector/vector derivative, and thus a matrix.

But, we have turned it into the shape $(n^\ell \times n^\ell)$.

7.X.23 Element-wise multiplication

Notice that, in the previous section, we would've compressed this matrix down to remove the unnecessary 0's:

$$\begin{bmatrix} f'(z_1^\ell) \\ f'(z_2^\ell) \\ \vdots \\ f'(z_n^\ell) \end{bmatrix} \tag{7.145}$$

This is a valid way to interpret this matrix! The only thing we need to be careful of: if we were to use this in a chain rule, we couldn't do normal matrix multiplication.

However, because of how this matrix works, you can just do **element-wise** multiplication instead!

You can check it for yourself: each index is separately scaled.

Concept 49

When multiplying two vectors R and Q, if they take the form

$$R = \begin{bmatrix} r_1 & 0 & 0 & \cdots & 0 \\ 0 & r_2 & 0 & \cdots & 0 \\ 0 & 0 & r_3 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & r_n \end{bmatrix} \qquad Q = \begin{bmatrix} q_1 \\ q_2 \\ q_3 \\ \vdots \\ q_n \end{bmatrix}$$

Then we can write their product each of these ways:

$$RQ = \overbrace{R * Q}^{\text{Element-wise multiplication}} = \begin{bmatrix} r_1 q_1 \\ r_2 q_2 \\ r_3 q_3 \\ \vdots \\ r_n q_n \end{bmatrix} \tag{7.146}$$

So, we can substitute the chain rule this way.

7.6 Training (WIP)

7.6.1 Comments

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

Concept 50

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 51

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.

7.6.2 Pseudocode

Our training algorithm for backprop can follow smoothly from what we've laid out.

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

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**:
                    $W^\ell = W^\ell - \Big(\eta(t) * (\partial\mathcal{L}/\partial W^\ell)\Big)$

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

Return final neural network with weights and biases

7.7 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
- Matrix/scalar derivative
- Scalar/Matrix derivative
- Axis
- Dimension (vector)
- Dimension (array)
- Array
- "Tensor" (Generalized matrix)
- c-Tensor
- Gaussian/Normal Distribution (Optional)