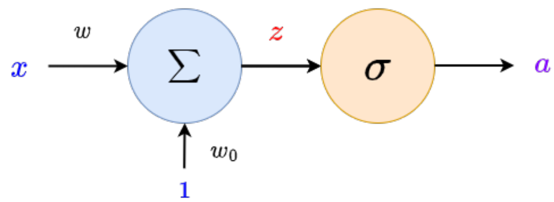
Explanatory Notes for 6.390

Shaunticlair Ruiz (Current TA)

Fall 2022

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**!



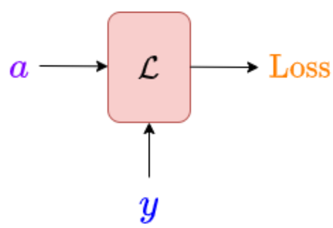
As usual, our first unit $\sum$ is our **linear** component. The output is z, nothing different from before with LLC.

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.

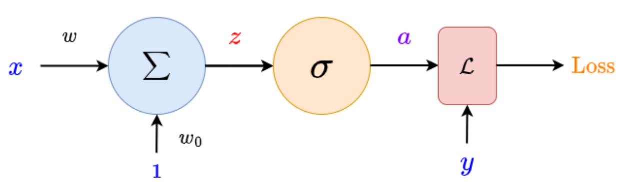
Something we neglected before: this diagram is **missing** the **loss function**. Let's create a small unit for that.

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

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 1

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:

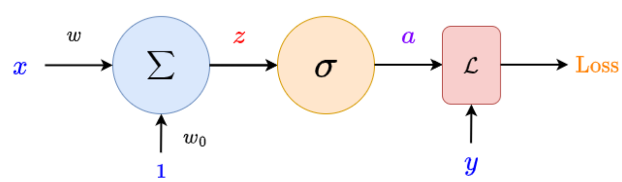
- $f(10) = 10 + 2$
- $g(f) = 3 \cdot 12$
- $h(g) = \sin(36)$

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

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

LLC Back-propagation

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

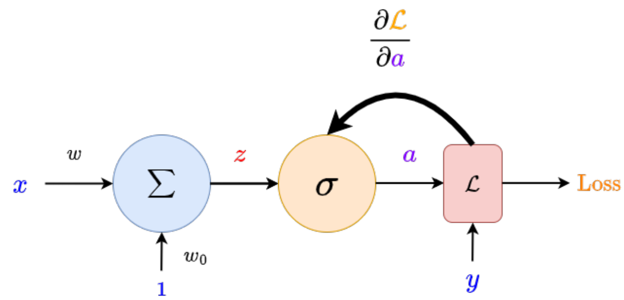


We want to link $\mathcal{L}$ to w. In order to do that, we need to **connect** each thing in between.

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

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

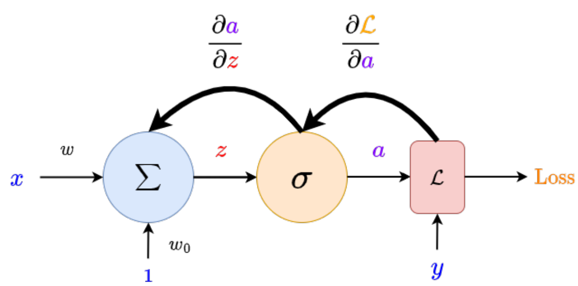
Loss is what we really care about. So, what is the loss directly **connected** to? The **activation**, a.



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

$$\overbrace{\frac{\partial \mathcal{L}}{\partial a}}^{\text{Loss unit}} \tag{1}$$

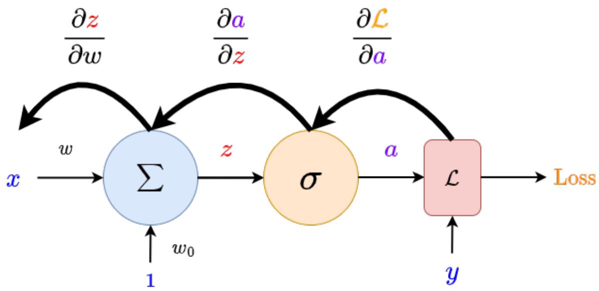
And what is that connected to? The **pre-activation** z:



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

$$\overbrace{\frac{\partial \mathcal{L}}{\partial a}}^{\text{Loss unit}} \cdot \overbrace{\frac{\partial a}{\partial z}}^{\text{Activation function}} \tag{2}$$

And finally, we've reached w:



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

$$\frac{\partial \mathcal{L}}{\partial w} = \overbrace{\frac{\partial \mathcal{L}}{\partial a}}^{\text{Loss unit}} \cdot \overbrace{\frac{\partial a}{\partial z}}^{\text{Activation}} \cdot \overbrace{\frac{\partial z}{\partial w}}^{\text{Linear subunit}} \tag{3}$$

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

Definition 2

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.

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 3

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**.