

Lecture 12 [Backprop & improving Neural Networks]

Logistic Regression with a Neural Network mindset:

We've seen what a **neural network** is and we started by defining the **logistic regression** from a **neural network** perspective. We said that **logistic regression** can be viewed as a **1-neuron neural network** where there is a **linear** part and an **activation** part, which was **sigmoid** in that case. We've seen that **sigmoid** is a common **activation function** to be used for **classification** tasks because it casts a number between $-\infty$ and $+\infty$ in the $(0, 1)$ interval, which can be interpreted as a probability. Then we introduced the **neural network**. We started to stack some **neurons** inside a **layer** and then stack **layers** on top of each other and said that the more we stack **layers**, the more **parameters** we have, and the more **parameters** we have, the more our **network** is able to copy the complexity of our data, because it becomes more flexible

We stopped at a point where we did a **forward propagation**. We had an example during **training**, we **forward propagated** through the **network**, we got the **output**, then we **computed** the **cost function** which compares the **output** to the **ground truth**, and we were in the process of **backpropagating** the **error** to tell our **parameters** how they should move in order to detect cats more properly

We're going to derive the backpropagation with the chain rule and after that we're going to talk about how to improve our **Neural Networks**. In practice, it's not because you designed a **neural network** that it's going to work. There's a lot of hacks and tricks that you need to know in order to make a **neural network** work

Neural Networks \Rightarrow Backpropagation:

In order to define our **optimization problem** and find the right parameters, we need to define a **cost function**. The letter **J** denotes the cost function. When we talk about **cost function** here, we're talking about the batch of examples. It means we're forward propagating **m** examples at a time

The reason we use a batch instead of a single example is because of **vectorization**. We want to use what our **GPU** can do and **parallelize** the computation

So we have **m** examples that **forward propagate** in the network and each of them has a **loss function** associated with them. The average of the **loss functions** over the batch gives us the **cost function**. And we had defined this loss function

Backpropagation:

$$\text{cost function: } \mathcal{J}(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^m L^{(i)}(\hat{y}, y)$$

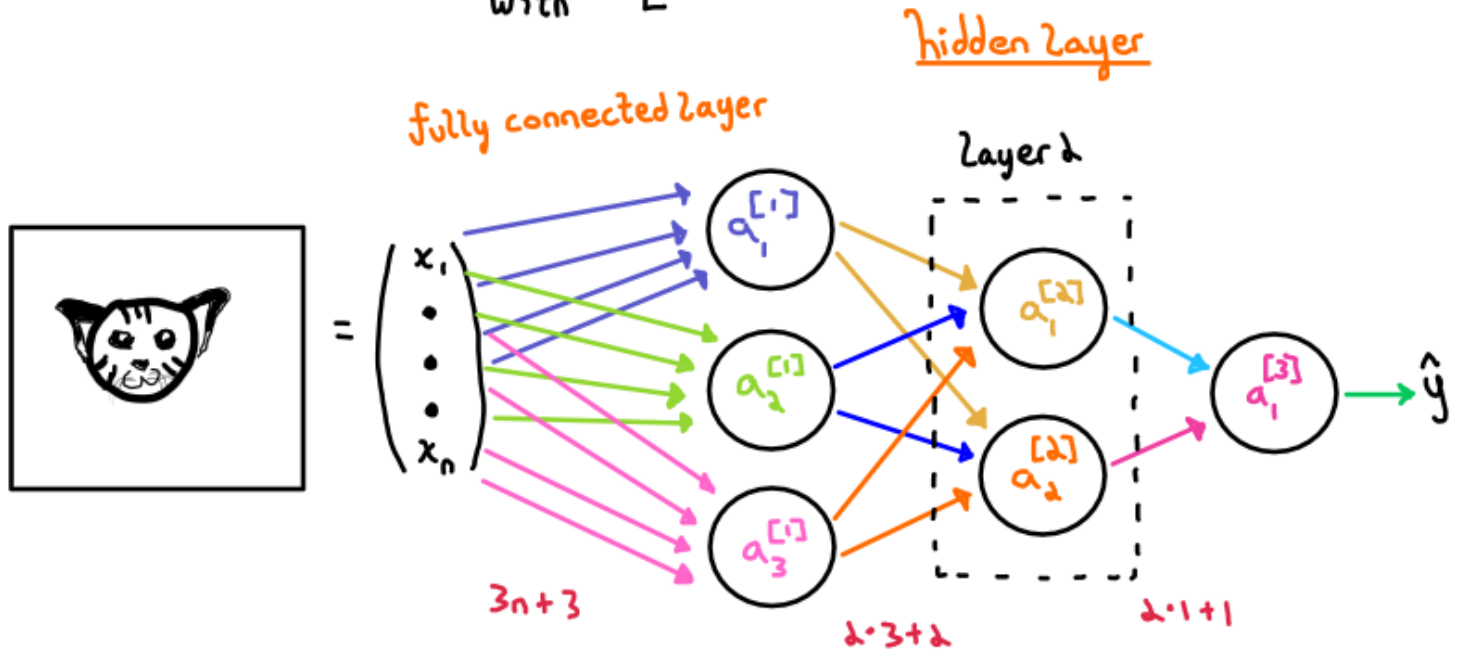
with $L^{(i)}$

We're still in this network where we had a cat:

Backpropagation:

cost function:
$$\mathcal{J}(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^m L^{(i)}(\hat{y}, y)$$

with $L^{(i)}$



The cat was flattened into a vector, **RGB matrix** into one vector, and then there was a **neural network** with **3 neurons**, then **2 neurons**, then **1 neuron**

We take **m** images of cats or non-cats, **forward propagate** everything in the network, compute our **loss function** for each of them, average it, and get the **cost function**

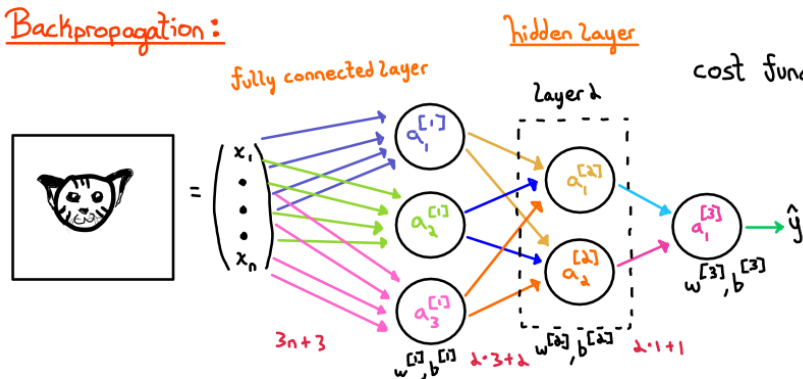
So our **loss function** was the **binary cross-entropy** (also called the **logistic loss function**) and it was the following:

cost function:
$$\mathcal{J}(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^m L^{(i)}(\hat{y}, y)$$

with
$$L^{(i)} = -[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log (1 - \hat{y}^{(i)})]$$

What we said is that this network has many parameters as we said the **first layer** has $w^{[1]}$, $b^{[1]}$, the **second layer** has $w^{[2]}$, $b^{[2]}$, and the **third layer** has $w^{[3]}$, $b^{[3]}$, where the **[]** denotes the **layer**, and we have to train all these parameters

Backpropagation:



cost function:
$$\mathcal{J}(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^m L^{(i)}(\hat{y}, y)$$

with
$$L^{(i)} = -[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log (1 - \hat{y}^{(i)})]$$

One thing we notice is that because we want to make a good use of the **chain rule**, we're going to start by computing the derivative of the $w^{[3]}, b^{[3]}$ parameters and then come back and do $w^{[2]}, b^{[2]}$ and then back again to do $w^{[1]}, b^{[1]}$

In order to use our formulas of the update of the **gradient descent** where w would be equal to:

cost function:
$$\mathcal{J}(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^m L^{(i)}(\hat{y}, y)$$

with
$$L^{(i)} = -[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log (1 - \hat{y}^{(i)})]$$

Update:
$$w^{[2]} = w^{[2]} - d \frac{\partial \mathcal{J}}{\partial w^{[2]}}$$
 "derivative"

For any **layer l** between **1** and **3**. Same for **b**. So now we'll try to do it

This is the first number we want to compute:

$$\frac{\partial \mathcal{J}}{\partial w^{[3]}}$$

The reason we want to compute the **derivative** of the **cost** with respect to $w^{[3]}$ is because the relationship between $w^{[3]}$ and the **cost** is easier than the relationship between $w^{[1]}$ and the **cost** because $w^{[1]}$ had much more connection going through the network before ending up in the **cost computation**

One thing we should notice before starting this calculation is that the **derivative** is **linear**. So if you take the derivative of **J**, you can just take the derivative of **L**, and it's the same thing. You just need to add the summation prior to that because derivative is a **linear** operation. So instead of computing this:

$$\frac{\partial J}{\partial w^{[3]}}$$

We're going to compute that and then we will add the summation which will just make our notation easier

$$\frac{\partial J}{\partial w^{[3]}} =$$

We're taking the derivative of a **loss** of one example propagated through the network with respect to $w^{[3]}$

We remember that \hat{y} was equal to **sigmoid** of $w^{[3]}x + b$ or $w^{[3]}a^{[2]} + b$ because $a^{[2]}$ is the input to the **second layer**

$$\frac{\partial J}{\partial w^{[3]}} = - \left[y^{(i)} \frac{\partial}{\partial w^{[3]}} (\log \sigma(w^{[3]}a^{[2]} + b^{[3]})) + (1 - y^{(i)}) \frac{\partial}{\partial w^{[3]}} (\log(1 - \sigma(w^{[3]}a^{[2]} + b^{[3]}))) \right]$$

The reason we have this:

$$(\cdot (w^{[3]}a^{[2]} + b^{[3]})) \Big]$$

is because we've written the **forward propagation** previously where we had $z^{[3]}$ which took $a^{[2]}$ as inputs and computed the **linear** part, and **sigmoid** is the **activation function** used in the **last neuron** in the network

The derivative of log is:

$$(\log' x) = \frac{1}{x}$$

Now we'll try to compute this derivative. We will take $\mathbf{1} / \sigma(\mathbf{w}^{[3]} \mathbf{a}^{[2]} + \mathbf{b}^{[3]})$ because we know that this can be written as $\mathbf{a}^{[3]}$. So we will just write $\mathbf{a}^{[3]}$ instead of writing the **sigmoid** again

$$\begin{aligned} \frac{\partial \mathcal{J}}{\partial \mathbf{w}^{[3]}} &= - \left[y^{(i)} \frac{\partial}{\partial \mathbf{w}^{[3]}} (\log \sigma(\mathbf{w}^{[3]} \mathbf{a}^{[2]} + \mathbf{b}^{[3]})) + (1 - y^{(i)}) \frac{\partial}{\partial \mathbf{w}^{[3]}} (\log(1 - \sigma(\mathbf{w}^{[3]} \mathbf{a}^{[2]} + \mathbf{b}^{[3]}))) \right] \\ &= - \left[y^{(i)} \frac{1}{\mathbf{a}^{[3]}} \right] \end{aligned}$$

So we have $\mathbf{1} / \mathbf{a}^{[3]}$ times the derivative of $\mathbf{a}^{[3]}$ with respect to $\mathbf{w}^{[3]}$

If we take the derivative of **log** of **sigmoid** of (. . .) over \mathbf{w} , what we have is:

$$\frac{\partial \log \sigma(\dots)}{\partial \mathbf{w}} = \frac{1}{\sigma(\dots)} \frac{\partial \sigma(\dots)}{\mathbf{w}^{[3]}}$$

That's what we're using here

The derivative of **sigmoid** is actually pretty easy to compute. It's:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Taking the derivative, it's going to give us:

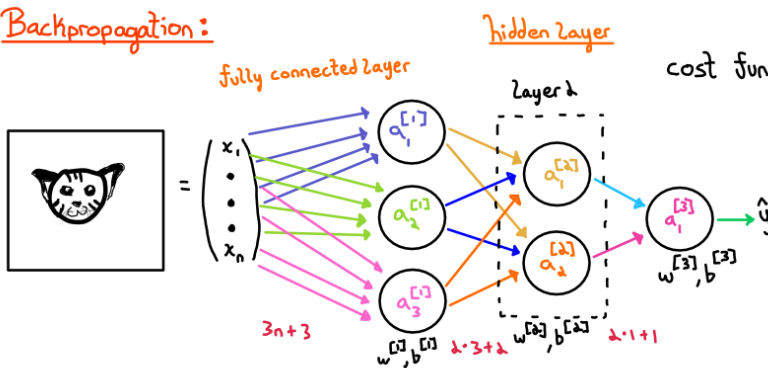
$$\frac{\partial \mathcal{J}}{\partial w^{[3]}} = - \left[y^{(i)} \frac{\partial}{\partial w^{[3]}} (\log \sigma(w^{[3]} a^{[2]} + b^{[3]})) + (1 - y^{(i)}) \frac{\partial}{\partial w^{[3]}} (\log(1 - \sigma(w^{[3]} a^{[2]} + b^{[3]}))) \right]$$

$$= - \left[y^{(i)} \frac{1}{a^{[3]}} \cdot a^{[3]} \cdot (1 - a^{[3]}) \right]$$

There's still one step because there is a composition of three functions here. There is a **logarithm**, there's a **sigmoid**, and there's also a **linear function** ($w\mathbf{x} + \mathbf{b}$ or $\mathbf{w}\mathbf{a}^{[2]} + \mathbf{b}$). So we also need to take the derivative of the **linear** part with respect to $\mathbf{w}^{[3]}$ because we know that to take this derivative, we need to go inside and take the derivative of what's inside:

$$\frac{\partial \sigma(w^{[3]} a^{[2]} + b^{[3]})}{\partial w^{[3]}} = a^{[3]} (1 - a^{[3]}) \frac{\partial}{\partial w^{[3]}} (w^{[3]} a^{[2]} + b^{[3]})$$

Backpropagation:



cost function: $\mathcal{J}(\hat{y}, y) = \frac{1}{n} \sum_{i=1}^n L^{(i)}(\hat{y}^{(i)}, y^{(i)})$
 with $L^{(i)} = -[y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$
 Update: $w^{[1]} = w^{[1]} - \alpha \frac{\partial \mathcal{J}}{\partial w^{[1]}}$ "derivative"

$$\frac{\partial \log \sigma(\dots)}{\partial w} = \frac{1}{\sigma(\dots)} \frac{\partial \sigma(\dots)}{\partial w^{[3]}}$$

$$(\log' x) = \frac{1}{x}$$

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

$$\frac{\partial \sigma(w^{[3]} a^{[2]} + b^{[3]})}{\partial w^{[3]}} = a^{[3]} (1 - a^{[3]}) \frac{\partial}{\partial w^{[3]}} (w^{[3]} a^{[2]} + b^{[3]})$$

Here, we need to take the derivative of the **linear** part with respect to $\mathbf{w}^{[3]}$ which is equal to $\mathbf{a}^{[2]T}$

$$\frac{\partial \mathcal{J}}{\partial w^{[3]}} = - \left[y^{(i)} \frac{\partial}{\partial w^{[3]}} \left(\log \sigma(w^{[3]} a^{[2]} + b^{[3]}) \right) + (1 - y^{(i)}) \frac{\partial}{\partial w^{[3]}} \left(\log(1 - \sigma(w^{[3]} a^{[2]} + b^{[3]})) \right) \right]$$

$$= - \left[y^{(i)} \frac{1}{a^{[3]}} \cdot a^{[3]} \cdot (1 - a^{[3]}) \cdot a^{[2]T} \right]$$

One thing you may want to check is when we're trying to compute this derivative:

$$\frac{\partial}{\partial w^{[3]}} \left(w^{[3]} a^{[2]} + b^{[3]} \right)$$

We're trying to compute this derivative. Why is there a transpose that comes out? How do you come up with that?

You look at the shape of $w^{[3]}$ which is:

$$\frac{\partial}{\partial w^{[3]}} \left(w^{[3]} a^{[2]} + b^{[3]} \right)$$

↑
(1,2)

It is **(1,2)** because it's connecting two **neurons** to one **neuron**, so it has to be **(1,2)**, usually flip it. And in order to come back to that, you can write your **forward propagation**, make the shape analysis, and find out that it's a **(1,2)** matrix

The shape of $\mathbf{w}^{[3]} \mathbf{a}^{[2]} + \mathbf{b}^{[3]}$ is **(1,1)** because it's a **scalar**

$$\frac{\partial}{\partial \mathbf{w}^{[3]}} \left(\underbrace{\mathbf{w}^{[3]} \mathbf{a}^{[2]} + \mathbf{b}^{[3]}}_{(1,1)} \right)$$

$(1,2)$ $(1,1)$

How do you know? It's because this thing is basically $\mathbf{z}^{[3]}$. It's the **linear** part of the last **neuron** and $\mathbf{a}^{[3]}$ we know that it's \mathbf{y}^{\wedge} . So it's a **scalar** between **0** and **1**, so this $(\mathbf{w}^{[3]} \mathbf{a}^{[2]} + \mathbf{b}^{[3]})$ has to be a **scalar** as well. Taking the **sigmoid** should not change the shape

What's the shape of this entire thing?

$$\frac{\partial}{\partial \mathbf{w}^{[3]}} \left(\underbrace{\mathbf{w}^{[3]} \mathbf{a}^{[2]} + \mathbf{b}^{[3]}}_{(1,1)} \right)$$

$(1,2)$ $(1,1)$

The shape of this entire thing should be the shape of $\mathbf{w}^{[3]}$ because you're taking the derivative of a **scalar** with respect to a higher-dimensional matrix or vector (here, called a **row vector**). It means that the shape of this has to be the same shape of $\mathbf{w}^{[3]}$. So **(1,2)**

A handwritten diagram showing the derivative $\frac{\partial}{\partial w^{[3]}} (w^{[3]} a^{[2]} + b^{[3]})$ enclosed in a pink oval. A blue arrow points from the dimension $(1,2)$ to $w^{[3]}$. A blue bracket under $a^{[2]} + b^{[3]}$ is labeled $(1,1)$. A pink arrow points from the top right of the oval to the dimension $(1,2)$.

When you take this simple derivative with **scalars**, not with high-dimensional, you know that this is an easy derivative. It should just give you $\mathbf{a}^{[2]}$. But in higher dimensions, sometimes you have **transpose** that come up

How do you know that the answer is $\mathbf{a}^{[2]T}$? It's because you know that $\mathbf{a}^{[2]}$ is a $(2,1)$ matrix

A handwritten diagram showing the derivative $\frac{\partial}{\partial w^{[3]}} (w^{[3]} a^{[2]} + b^{[3]})$ enclosed in a pink oval. A blue arrow points from the dimension $(1,2)$ to $w^{[3]}$. A blue bracket under $a^{[2]} + b^{[3]}$ is labeled $(1,1)$. A pink arrow points from the top right of the oval to the dimension $(1,2)$. To the right of the oval, the expression $= \underbrace{a^{[2]}}_{(2,1)^T}^T$ is written in pink.

So this is not possible. It's not possible to get $\mathbf{a}^{[2]}$ because otherwise it wouldn't match the derivative that you are calculating. So it has to be $\mathbf{a}^{[2]T}$

So either you learn the formula by heart or you learn how to analyze shapes

So that's why it's $\mathbf{a}^{[2]T}$. Now, for the second term of the derivative:

$$\begin{aligned}\frac{\partial J}{\partial w^{[3]}} &= - \left[y^{(i)} \frac{\partial}{\partial w^{[3]}} (\log \sigma(w^{[3]} a^{[2]} + b^{[3]})) + (1 - y^{(i)}) \frac{\partial}{\partial w^{[3]}} (\log(1 - \sigma(w^{[3]} a^{[2]} + b^{[3]}))) \right] \\ &= - \left[y^{(i)} \frac{1}{a^{[3]}} \cdot a^{[3]} \cdot (1 - a^{[3]}) \cdot a^{[2]T} + (1 - y^{(i)}) \frac{1}{1 - a^{[3]}} \cdot (-1) a^{[3]} (1 - a^{[3]}) \cdot a^{[2]T} \right]\end{aligned}$$

Now we will just simplify

$$\begin{aligned}\frac{\partial J}{\partial w^{[3]}} &= - \left[y^{(i)} \frac{\partial}{\partial w^{[3]}} (\log \sigma(w^{[3]} a^{[2]} + b^{[3]})) + (1 - y^{(i)}) \frac{\partial}{\partial w^{[3]}} (\log(1 - \sigma(w^{[3]} a^{[2]} + b^{[3]}))) \right] \\ &= - \left[y^{(i)} \frac{1}{a^{[3]}} \cdot a^{[3]} \cdot (1 - a^{[3]}) \cdot a^{[2]T} + (1 - y^{(i)}) \frac{1}{1 - a^{[3]}} \cdot (-1) a^{[3]} (1 - a^{[3]}) \cdot a^{[2]T} \right] \\ &= - \left[y^{(i)} (1 - a^{[3]}) a^{[2]T} - (1 - y^{(i)}) a^{[3]} \cdot a^{[2]T} \right] \\ &= - \left[y^{(i)} a^{[2]T} - a^{[3]} a^{[2]T} \right] \\ &= - \left(y^{(i)} - a^{[3]} \right) a^{[2]T}\end{aligned}$$

Once we get this result, we can just write down the costs of the derivative with respect to $w^{[3]}$, taking the summation of it:

$$\begin{aligned}
\frac{\partial \mathcal{J}}{\partial w^{[3]}} &= - \left[y^{(i)} \frac{\partial}{\partial w^{[3]}} (\log \sigma(w^{[3]} a^{[2]} + b^{[3]})) + (1 - y^{(i)}) \frac{\partial}{\partial w^{[3]}} (\log(1 - \sigma(w^{[3]} a^{[2]} + b^{[3]}))) \right] \\
&= - \left[y^{(i)} \frac{1}{a^{[3]}} \cdot a^{[3]} \cdot (1 - a^{[3]}) \cdot a^{[2]T} + (1 - y^{(i)}) \frac{1}{1 - a^{[3]}} \cdot (-1) a^{[3]} (1 - a^{[3]}) \cdot a^{[2]T} \right] \\
&= - \left[y^{(i)} (1 - a^{[3]}) a^{[2]T} - (1 - y^{(i)}) a^{[3]} a^{[2]T} \right] \\
&= - \left[y^{(i)} a^{[2]T} - a^{[3]} a^{[2]T} \right] \\
&= - \left(y^{(i)} - a^{[3]} \right) a^{[2]T} \\
\frac{\partial \mathcal{J}}{\partial w^{[3]}} &= - \frac{1}{m} \sum_{i=1}^m (y^{(i)} - a^{[3]}) a^{[2]T}
\end{aligned}$$

So that's our derivative. We can just take this formula, plugging it back in our **gradient descent update rule**, and update $w^{[3]}$

$$\text{Update: } w^{[2]} = w^{[2]} - d \frac{\partial \mathcal{J}}{\partial w^{[2]}} \quad \text{"derivative"}$$

We can do the same thing we just did, but with $b^{[3]}$. It's going to be the similar difficulty

We're going to do it with $w^{[2]}$ now and think how does that **backpropagate** to $w^{[2]}$. We want to compute the derivative of:

$$\frac{\partial \mathcal{L}}{\partial w^{[2]}}$$

How to get this one without having to do too much work? We're going to use the **chain rule** of calculus. So we're going to try to decompose this derivative into several derivatives

We know that \hat{y} is the first thing that is connected to the **loss function**. The output neuron is directly connected to the **loss function**. So we're going to take the derivative of the **loss function** with respect to \hat{y} , also called $a^{[3]}$. This is the easiest one we can calculate

$$\frac{\partial L}{\partial w^{[2]}} = \frac{\partial L}{\partial a^{[3]}}$$

We also know that $a^{[3]}$, which is the **output activation** of the **last neuron**, is connected with the **linear** part of the **last neuron**, which is $z^{[3]}$. So we can take the derivative of $a^{[3]}$ with respect to $z^{[3]}$

$$\frac{\partial L}{\partial w^{[2]}} = \frac{\partial L}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}}$$

The derivative of $a^{[3]}$ with respect to $z^{[3]}$ is the derivative of **sigmoid**. We know that $a^{[3]} = \sigma(z^{[3]})$

We know that $z^{[3]}$ is equal to $w^{[3]}a^{[2]} + b$. Which path do we need to take in order to **backpropagate**?

We don't want to take the derivative with respect to $w^{[3]}$ because we would get stuck. We don't want to take the derivative with respect to $b^{[3]}$ because we would get stuck. We will take the derivative with respect to $a^{[2]}$ because $a^{[2]}$ will be connected to $z^{[2]}$, $z^{[2]}$ will be connected to $a^{[1]}$, and we can backpropagate from this path

$$\frac{\partial L}{\partial w^{[2]}} = \frac{\partial L}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial w^{[2]}}$$

Why don't we take a derivative with respect to $w^{[3]}$ or $b^{[3]}$? It's because we will get stuck. We want the error to **backpropagate**, and in order for the error to **backpropagate**, we have to go through variables that are connected to each other

Now, how can we use this?

$$\begin{aligned}
\frac{\partial J}{\partial w^{[3]}} &= - \left[y^{(i)} \frac{\partial}{\partial w^{[3]}} (\log \sigma(w^{[3]} a^{[2]} + b^{[3]})) + (1 - y^{(i)}) \frac{\partial}{\partial w^{[3]}} (\log(1 - \sigma(w^{[3]} a^{[2]} + b^{[3]}))) \right] \\
&= - \left[y^{(i)} \frac{1}{a^{[3]}} \cdot a^{[3]} \cdot (1 - a^{[3]}) \cdot a^{[2]T} + (1 - y^{(i)}) \frac{1}{1 - a^{[3]}} \cdot (-1) a^{[3]} (1 - a^{[3]}) \cdot a^{[2]T} \right] \\
&= - \left[y^{(i)} (1 - a^{[3]}) a^{[2]T} - (1 - y^{(i)}) a^{[3]} a^{[2]T} \right] \\
&= - \left[y^{(i)} a^{[2]T} - a^{[3]} a^{[2]T} \right] \\
&= - \left(y^{(i)} - a^{[3]} \right) a^{[2]T} \\
\frac{\partial J}{\partial w^{[3]}} &= - \frac{1}{n} \sum_{i=1}^n \left(y^{(i)} - a^{[3]} \right) a^{[2]T}
\end{aligned}$$

How can we use the derivative we already have in order to compute the derivative with respect to $w^{[2]}$?

This result:

$$= - \left(y^{(i)} - a^{[3]} \right) a^{[2]T}$$

is actually the first two terms here:

$$\frac{\partial L}{\partial w^{[2]}} = \frac{\partial L}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}} \cdot \frac{\partial z^{[2]}}{\partial w^{[2]}}$$

How do we know that? It's not easy to see. One thing we know, based on what we've written here:

$$\frac{\partial}{\partial \mathbf{w}^{[3]}} (\mathbf{w}^{[3]} \mathbf{a}^{[2]} + b^{[3]}) = \mathbf{a}^{[2]T}$$

Is that the derivative of $\mathbf{z}^{[3]}$, because the above is $\mathbf{z}^{[3]}$, with respect to $\mathbf{w}^{[3]}$ is $\mathbf{a}^{[2]T}$

So we could write here that this thing is derivative of $\mathbf{z}^{[3]}$ with respect to $\mathbf{w}^{[3]}$:

$$\begin{aligned} \frac{\partial \mathcal{J}}{\partial \mathbf{w}^{[3]}} &= - \left[y^{(i)} \frac{\partial}{\partial \mathbf{w}^{[3]}} (\log \sigma(\mathbf{w}^{[3]} \mathbf{a}^{[2]} + b^{[3]})) + (1 - y^{(i)}) \frac{\partial}{\partial \mathbf{w}^{[3]}} (\log(1 - \sigma(\mathbf{w}^{[3]} \mathbf{a}^{[2]} + b^{[3]}))) \right] \\ &= - \left[y^{(i)} \frac{1}{\sigma(\mathbf{z}^{[3]})} \cdot \sigma(\mathbf{z}^{[3]}) \cdot (1 - \sigma(\mathbf{z}^{[3]})) \cdot \mathbf{a}^{[2]T} + (1 - y^{(i)}) \frac{1}{1 - \sigma(\mathbf{z}^{[3]})} \cdot (-1) \sigma(\mathbf{z}^{[3]}) (1 - \sigma(\mathbf{z}^{[3]})) \cdot \mathbf{a}^{[2]T} \right] \\ &= - \left[y^{(i)} (1 - \sigma(\mathbf{z}^{[3]})) \mathbf{a}^{[2]T} - (1 - y^{(i)}) \sigma(\mathbf{z}^{[3]}) \mathbf{a}^{[2]T} \right] \\ &= - \left[y^{(i)} \mathbf{a}^{[2]T} - \sigma(\mathbf{z}^{[3]}) \mathbf{a}^{[2]T} \right] \\ &= - (y^{(i)} - \sigma(\mathbf{z}^{[3]})) \mathbf{a}^{[2]T} \end{aligned}$$

$\frac{\partial \mathbf{z}^{[3]}}{\partial \mathbf{w}^{[3]}}$

So we know that because we wanted to compute the derivative of the **loss** to $\mathbf{w}^{[3]}$, we know that we could have written derivative of **loss** with respect to $\mathbf{w}^{[3]}$ as derivative of **loss** with respect to $\mathbf{z}^{[3]}$ times derivative of $\mathbf{z}^{[3]}$ with respect to $\mathbf{w}^{[3]}$

$$\begin{aligned}
\frac{\partial J}{\partial w^{[3]}} &= - \left[y^{(i)} \frac{\partial}{\partial w^{[3]}} \left(\log \sigma(w^{[3]} a^{[2]} + b^{[3]}) \right) + (1 - y^{(i)}) \frac{\partial}{\partial w^{[3]}} \left(\log(1 - \sigma(w^{[3]} a^{[2]} + b^{[3]})) \right) \right] \\
&= - \left[y^{(i)} \frac{1}{a^{[3]}} \cdot a^{[3]} \cdot (1 - a^{[3]}) \cdot a^{[2]T} + (1 - y^{(i)}) \frac{1}{1 - a^{[3]}} \cdot (-1) a^{[3]} (1 - a^{[3]}) \cdot a^{[2]T} \right] \\
&= - \left[y^{(i)} (1 - a^{[3]}) a^{[2]T} - (1 - y^{(i)}) a^{[3]} a^{[2]T} \right] \\
&= - \left[y^{(i)} a^{[2]T} - a^{[3]} a^{[2]T} \right] \\
&= - (y^{(i)} - a^{[3]}) a^{[2]T} \quad \underbrace{\qquad\qquad}_{\frac{\partial z^{[3]}}{\partial w^{[3]}}} \qquad \frac{\partial L}{\partial w^{[3]}} = \frac{\partial L}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial w^{[3]}}
\end{aligned}$$

And we know that this:

$$\frac{\partial L}{\partial w^{[3]}} = \frac{\partial L}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial w^{[3]}}$$

Is $a^{[2]T}$, so it means that this thing is the derivative of the loss with respect to $z^{[3]}$:

$$= - (y^{(i)} - a^{[3]}) a^{[2]T} \quad \underbrace{\qquad\qquad}_{\frac{\partial z^{[3]}}{\partial w^{[3]}}}$$

We got our decomposition of the derivative we had. If we wanted to use the **chain rule** from here on, we could have just separated it into two terms and took the derivatives here:

$$\frac{\partial L}{\partial w^{[3]}} = \frac{\partial L}{\partial z^{[3]}} \cdot \frac{\partial z^{[3]}}{\partial w^{[3]}}$$

We can also use **caching** in order to get this result very quickly

We know the result of this thing is basically:

$$\frac{\partial J}{\partial w^{[3]}} = -\frac{1}{n} \sum_{i=1}^n (y^{(i)} - a^{[3]}) a^{[2]T}$$

$$\frac{\partial L}{\partial w^{[2]}} = \underbrace{\frac{\partial L}{\partial a^{[3]}} \cdot \frac{\partial a^{[3]}}{\partial z^{[3]}}}_{(a^{[3]} - y)} \cdot \underbrace{\frac{\partial z^{[3]}}{\partial a^{[2]}} \cdot \frac{\partial a^{[2]}}{\partial z^{[2]}}}_{a^{[2]}(1 - a^{[2]})} \cdot \underbrace{\frac{\partial z^{[2]}}{\partial w^{[2]}}}_{a^{[1]T}}$$

Now we'll write down our derivative cleanly:

$$\frac{\partial L}{\partial w^{[2]}} = (a^{[3]} - y) \cdot w^{[3]T} \cdot a^{[2]}(1 - a^{[2]}) \cdot a^{[1]T}$$

There's actually two ways to compute derivatives. Either you go very rigorously and do what we did here for $w^{[2]}$ or you try to do a **chain rule** analysis and you try to fit the terms. The problem is, this result is not completely correct. There is a shape problem. It means, when we took our derivatives, we should have flipped some of the terms but we didn't

We're going to see how you can use **chain rule** plus **shape analysis** to come up with the results very quickly

We know that the **first term** is a **scalar**, so it's a **(1,1)**. We know that the **second term** is the transpose of **(1,2)**, so it's **(2,1)**. We know that the **third term** is a **(2,1)** because it's an **element-wise product**. And the **fourth term** is **(3,1)** transpose, so it's **(1,3)**

$$\frac{\partial L}{\partial w^{[2]}} = \underbrace{(a^{[3]} - y)}_{(1,1)} \cdot \underbrace{w^{[3]T}}_{(2,1)} \cdot \underbrace{a^{[2]}(1 - a^{[2]})}_{(2,1)} \cdot \underbrace{a^{[1]T}}_{(1,3)}$$

There seems to be a problem here. There is no match between these two **(2,1)** operations for example. How can we put everything together?

If we do it very rigorously, we know how to put it together. If you're used to doing the **chain rule**, you can quickly do it around. So after experience, you will be able to fit all these together. The important thing to know is that here, there is an **element-wise product**. So every time you will take the derivative of the **sigmoid**, it's going to end up being an **element-wise product**, and it's the case whatever the **activation** that you're using is

The correct result is this:

$$\begin{aligned} \frac{\partial L}{\partial w^{[2]}} &= \underbrace{(a^{[3]} - y)}_{(1,1)} \cdot \underbrace{w^{[3]T}}_{(2,1)} \cdot \underbrace{a^{[2]}(1 - a^{[2]})}_{(2,1)} \cdot \underbrace{a^{[1]T}}_{(1,3)} \\ &\quad \swarrow \\ &\quad (2,3) \\ &= \underbrace{w^{[3]T} \cdot a^{[2]}(1 - a^{[2]})}_{(2,1)} \cdot \underbrace{(a^{[3]} - y) a^{[1]T}}_{(1,1) \cdot (1,3)} \end{aligned}$$

$$\begin{aligned}
 \frac{\partial L}{\partial w^{[2]}} &= \underbrace{(a^{[3]} - y)}_{(1,1)} \cdot \underbrace{w^{[3]T}}_{(2,1)} \cdot \underbrace{a^{[2]}(1 - a^{[2]})}_{(2,1)} \cdot \underbrace{a^{[1]T}}_{(1,3)} \\
 &\quad \swarrow \\
 &\quad (2,3) \\
 &= \underbrace{w^{[3]T}}_{(2,1)} \cdot \underbrace{a^{[2]}(1 - a^{[2]})}_{(2,1)} \cdot \underbrace{(a^{[3]} - y)}_{(1,1)} \cdot \underbrace{a^{[1]T}}_{(1,3)} \\
 &\quad \underbrace{\hspace{10em}}_{(2,1)} \\
 &\quad \underbrace{\hspace{15em}}_{(2,3)}
 \end{aligned}$$

Usually in practice, we don't compute these **chain rules** anymore because programming frameworks do it for us, but it's important to know at least how the **chain rule** decomposes and also how to compute these derivatives, if you read research papers specifically

When you take the derivative of **sigmoid**, you take the derivative with respect to every entry of the matrix which gives you an **element-wise product**

Why is **cache** very important?

One thing is, it seems that during **backpropagation**, there is a lot of terms that appear that were computed during **forward propagation**. All these terms, $a^{[1]T}$, $a^{[2]}$, $a^{[3]}$, all these, we have it from the **forward propagation**. So if we don't **cache** anything, we have to recompute them. It means we're going backwards but then when you need $a^{[2]}$, for example, you'd have to go forward again to get $a^{[2]}$. If you go backwards, you'd need $a^{[1]}$ and you'd need to forward propagate your x again to get $a^{[1]}$. We don't want to do that. So in order to avoid that, we when do our **forward propagation**, we would keep in memory, almost all the values that we're getting including the w 's because as you see, to compute the derivative of **loss** with respect to $w^{[2]}$, we need $w^{[3]}$, but also the **activation** or **linear** variables

So we're going to save them in our **network** during the **forward propagation** in order to use it during the **backward propagation**. It's all for computational efficiency, it has some memory costs

That was **backpropagation**, so now we can use our formula of the **costs** with respect to the **loss function**, and we know that this is going to be our **update**

$$\frac{\partial J}{\partial w^{[2]}} = \frac{1}{E} \sum_{i=1}^E \frac{\partial L}{\partial w^{[2]}}$$

This is going to be used in order to update $w^{[2]}$ and we will do the same for $w^{[1]}$

Improving your Neural Networks:

In practice, when you do this process of training **forward propagation**, **backward propagation**, **updates**, you don't end up having a good network most of the time. In order to get a good network, you need to improve it. You need to use a bunch of techniques that will make your network work in practice

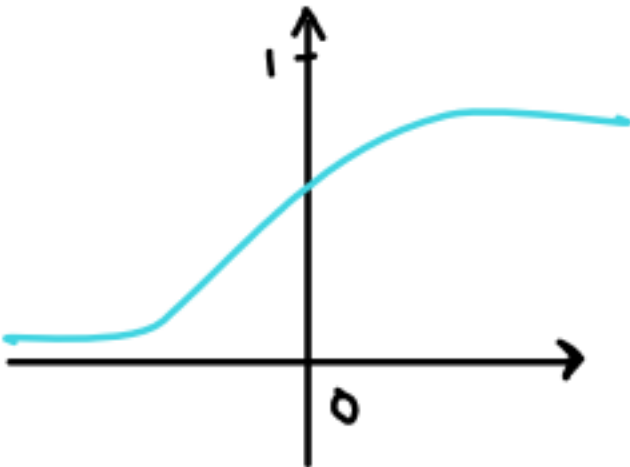
The first trick is to use different **activation functions**. We've seen one **activation function**, which was **sigmoid**, and we remember that the graph of **sigmoid** is getting a number between $-\infty$ and $+\infty$ and **casting** it between **0** and **1**. And we know that the formula is **sigmoid** of z equals $1 / (1 + e^{-z})$. We also know that the derivative of σ is $\sigma(z) (1 - \sigma(z))$

Improving your Neural Networks:

① Activation functions

$$\text{sigmoid } \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$



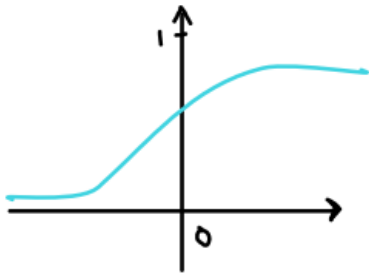
Another very common **activation function** is **ReLU**

Improving your Neural Networks:

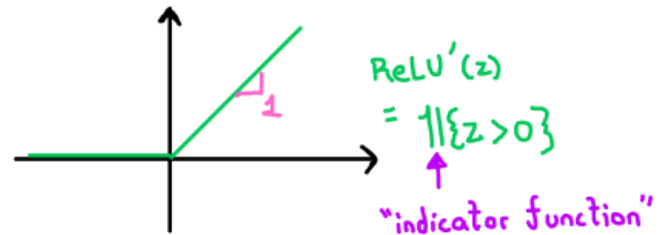
Ⓐ Activation functions

$$\text{sigmoid } \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$



$$\text{ReLU}(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ z & \text{if } z > 0 \end{cases}$$



Another one commonly used as well is **tanh** (*hyperbolic tangents*)

Improving your Neural Networks:

Ⓐ Activation functions

$$\text{sigmoid } \sigma(z) = \frac{1}{1 + e^{-z}}$$

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$

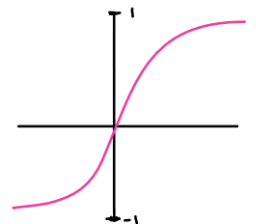


$$\text{ReLU}(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ z & \text{if } z > 0 \end{cases}$$



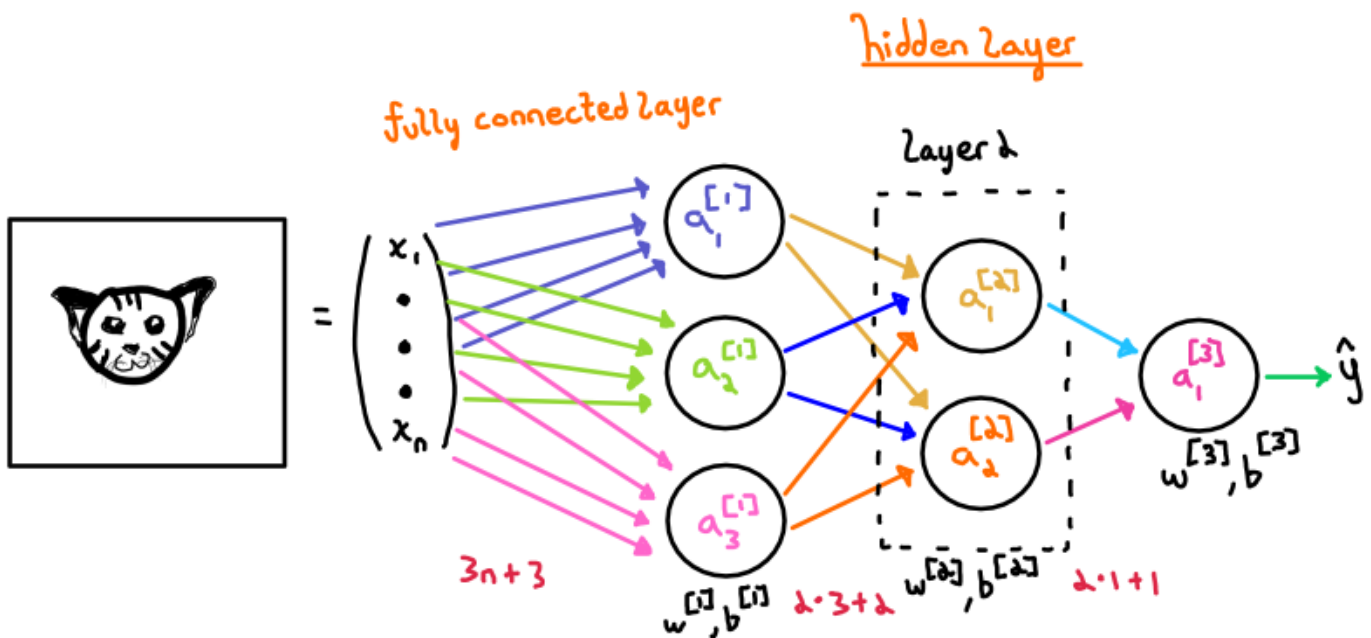
$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\tanh'(z) = 1 - \tanh^2(z)$$



Now that we have 3 **activation functions**, can you guess why we would use one instead of the other and which one has more benefits?

When we talk about **activation functions**, we talk about the functions that you will put in these **neurons** after the **linear** part:



What do you think is the main advantage of **sigmoid**?

You'd use it for **classification** because it gives you a probability

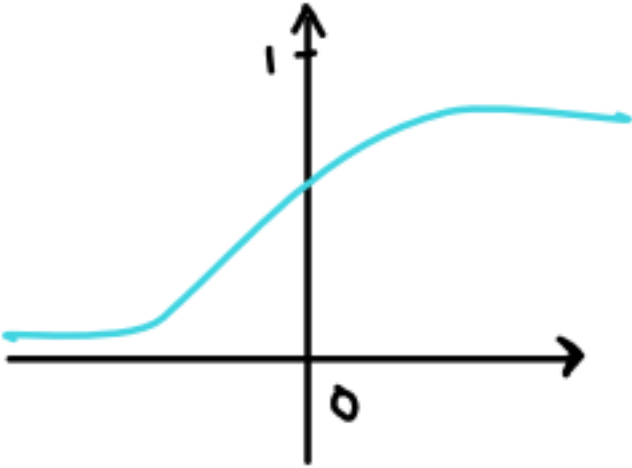
What's the main disadvantage of **sigmoid**?

If you're at **high activation**, if you are at high **z**'s or low **z**'s, your **gradient** is very close to **0**

Based on this graph, we know that if **z** is very big, our **gradient** is going to be very small. The slope of this graph is very very small, it's almost flat. It's the same for the **z**'s that are very low in the negative

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

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$



The problem with having **low gradients** is when you're **backpropagating**. If the **z** you **cached** was big, the **gradient** is going to be very **small** and it will be super hard to **update** your parameters that are early in the **network**, because the **gradient** is just going to vanish

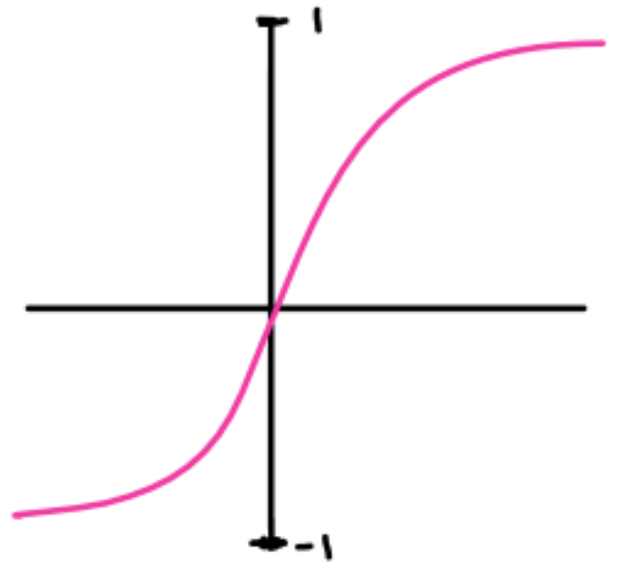
Sigmoid is one of these **activations** which works very well in the **linear regime**, but has trouble working in **saturating regimes** because the **network** doesn't **update** the parameters properly, it goes very very slowly

What about the advantages and disadvantages of **tanh**?

It's similar, similar like high **z**'s and low **z**'s lead to **saturation** of a **tanh activation**

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\tanh'(z) = 1 - \tanh(z)^2$$



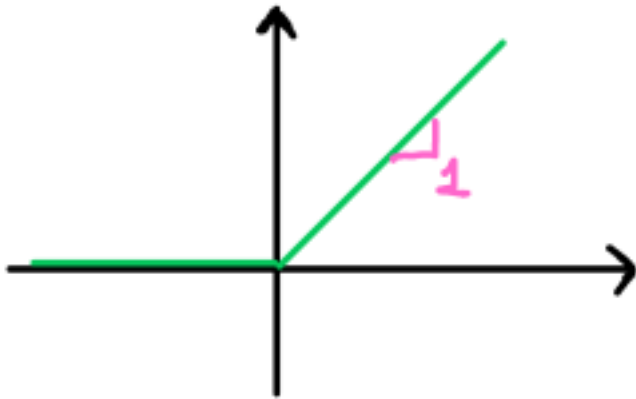
ReLU on the other hand, doesn't have this problem. If z is very big in the positives, there is no **saturation**. The **gradient** just passes and the **gradient** is **1**, the slope is equal to **1**. So it's actually just directing the **gradient** to some entry, it's not multiplying it by anything when you **backpropagate**

This term here, all the $a^{[3]}(1-a^{[3]})$ or $a^{[2]}(1-a^{[2]})$:

$$\frac{\partial \sigma(w^{[3]} a^{[2]} + b^{[3]})}{\partial w^{[3]}} = a^{[3]}(1-a^{[3]}) \frac{\partial}{\partial w^{[3]}} (w^{[3]} a^{[2]} + b^{[3]})$$

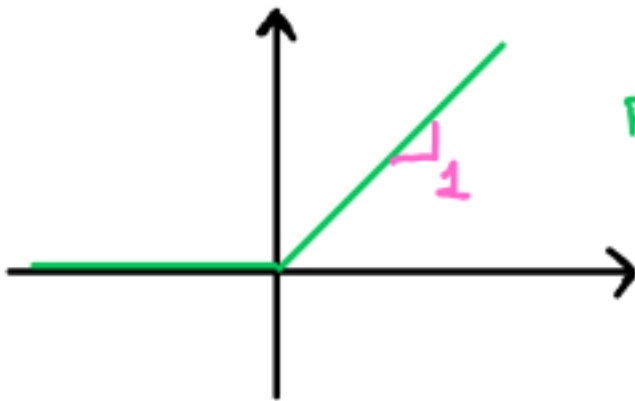
If we use **ReLU activations**, we would change this:

$$\text{ReLU}(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ z & \text{if } z > 0 \end{cases}$$



with the derivative of **ReLU** and the derivative of **ReLU** can be written like this:

$$\text{ReLU}(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ z & \text{if } z > 0 \end{cases}$$



$$\text{ReLU}'(z) = \mathbb{1}_{\{z > 0\}}$$

"indicator function"

For example, with the *house prediction example*, in that case if you want to predict the price of a house based on some features, you would use **ReLU** because you know that the output should be a positive number between **0** and $+\infty$. It doesn't make sense to use one of **tanh** or **sigmoid**

If you want your output to be between **0** and **1**, you would use **sigmoid**. If you want your output to be between **-1** and **1**, you would use **tanh**

There are some tasks where the output is kind of a **reward** or a **minus reward** that you want to get, like in **reinforcement learning**,

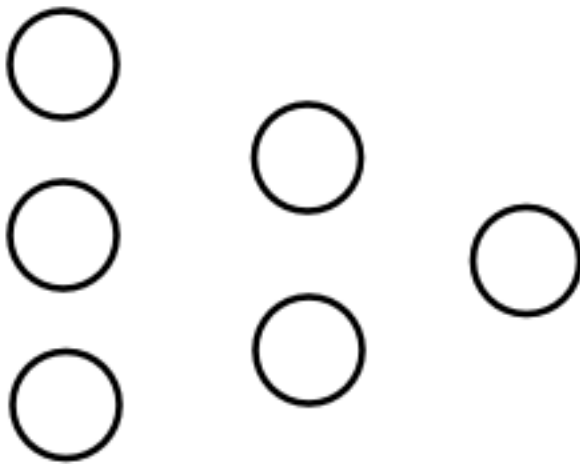
you would use ***tanh*** as an **output activation** which is because **-1** looks like a **negative reward**, **+1** looks like a **positive reward**, and you want to decide what should be the **reward**

Why do we consider these functions?

We can actually consider any functions apart from the **identity function**

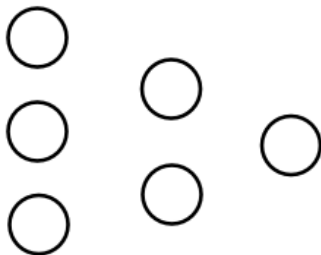
Let's assume that we have a **network** which is the same as before:

Why do we need activation functions? :



So our network is **3 neurons** casting into **2 neurons** casting into **1 neuron**, and we're trying to use **activations** that are equal to **identity functions**

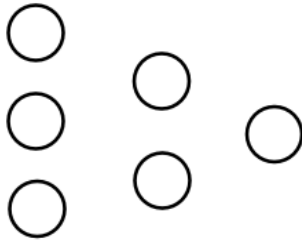
Why do we need activation functions? :



activations = Id $z \mapsto z$ ↪ "identity functions"

We'll try to derive the **forward propagation**

Why do we need activation functions? :

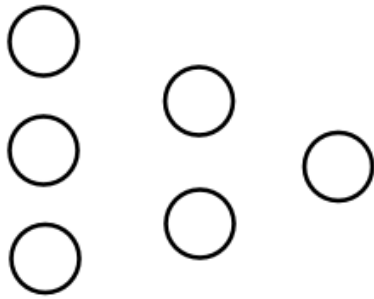


activations = Id $z \mapsto z$ ↪ "identity functions"

$$\hat{y} = a^{[3]} = z^{[3]} = w^{[3]} a^{[2]} + b^{[3]}$$

We know that $a^{[2]}$ is equal to $z^{[2]}$ because there is no activation and $z^{[2]}$ is equal to $w^{[2]} a^{[1]} + b^{[2]}$, so we can **cast** here:

Why do we need activation functions? :



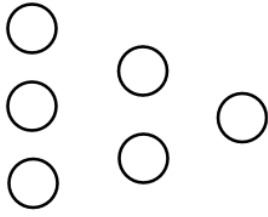
activations = Id $z \mapsto z$ ↪ "identity functions"

$$\hat{y} = a^{[3]} = z^{[3]} = w^{[3]} a^{[2]} + b^{[3]}$$

$$= w^{[3]} (w^{[2]} a^{[1]} + b^{[2]}) + b^{[3]}$$

We know that $a^{[1]}$ is equal to $z^{[1]}$ and we know that $z^{[1]}$ is equal to $w^{[1]} x + b^{[1]}$

Why do we need activation functions?:



activations = Id $z \mapsto z$ "identity functions"

$$\begin{aligned}\hat{y} &= a^{[3]} = z^{[3]} = w^{[3]} a^{[2]} + b^{[3]} \\ &= w^{[3]} (w^{[2]} a^{[1]} + b^{[2]}) + b^{[3]} \\ &= w^{[3]} w^{[2]} (w^{[1]} x + b^{[1]}) + w^{[3]} b^{[2]} + b^{[3]} \\ &= Wx + B \\ \text{with: } W &= w^{[3]} w^{[2]} w^{[1]} \\ B &= w^{[3]} \cdot w^{[2]} \cdot b^{[1]} + w^{[3]} b^{[2]} + b^{[3]}\end{aligned}$$

What's the insight here?

It's that we need **activation functions**. The reason is, if you don't choose **activation functions**, no matter how deep your **network** is, it's going to be equivalent to **linear regression**. So the complexity of the **network** comes from the **activation function**

If we're trying to detect cats, what we're trying to do is to train a **network** that will mimic the formula of detecting cats. We don't know this formula, so we want to mimic it using a lot of parameters. If we just have a **linear regression**, we cannot mimic this because we are going to look **pixel by pixel** and assign every **weight** to a certain **pixel**. If we give a new example it's not going to work anymore

This is why we need **activation functions**

Can we use different **activation functions** and how do we put them inside a **layer** or inside **neurons**?

These (the above three) have been designed with experience, so there are the ones that work better and lets our **networks** train. There are plenty of other **activation functions** that have been tested

Usually you would use the same **activation functions** inside every **layer**. It doesn't have any special reason but when you have a **network** like that, you would call it a fully connected **sigmoid layer** or **ReLU layer**, etc

People have been trying a lot of putting different **activations** in different **neurons** in a **layer** in different **layers**, and the consensus was using one **activation** in the **layer** and also using one of these three **activations**. If someone comes up with a better **activation** that is obviously helping training our **models** on different datasets, people would adopt it but right now these are the ones that work better

These are all **hyperparameters**. So in practice, you're not going to choose these randomly, you're going to try a bunch of them and choose some of them that seem to help your **model** train. There's a lot of experimental results in **deep learning** and we don't really understand fully why certain **activations** work better than others

Another trick that you can use in order to help your **network** train, are **initialization methods** and **normalization methods**

③ Initialization methods

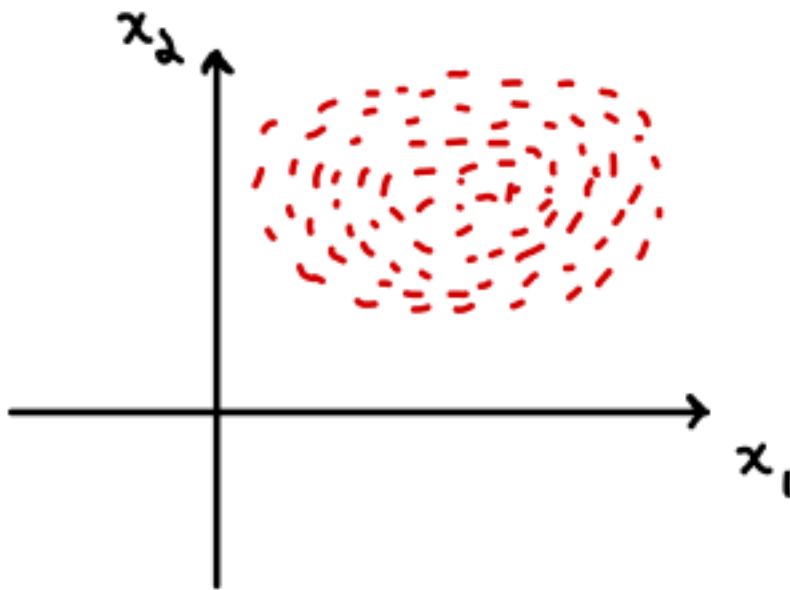
Earlier we talked about the fact that if **z** is too big or **z** is too low in the negative numbers, it will lead to **saturation** of the **network**. So in order to avoid that, you can use **normalization of the input**

Assume that you have a **network** where the data is **2-dimensional** and you can assume that $\mathbf{x}_1, \mathbf{x}_2$ is distributed like this:

⑧ Initialization methods

Normalizing your input :

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$



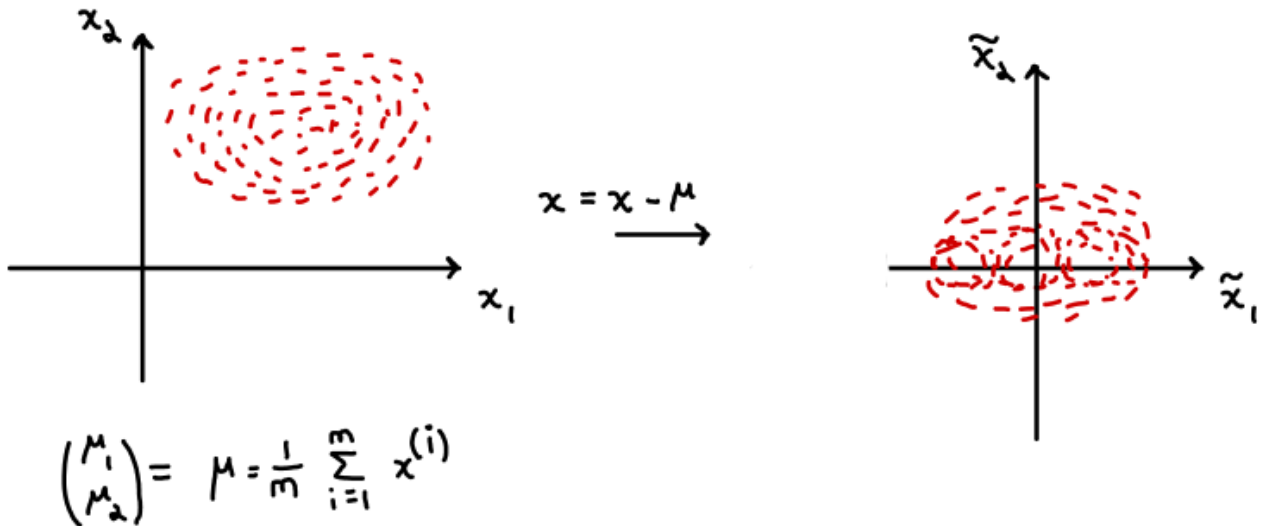
The problem is, if we do our $\mathbf{wx} + \mathbf{b}$ to compute our $\mathbf{z}^{[1]}$, if \mathbf{x} 's are very big, it will lead to very big \mathbf{z} 's which will lead to **saturated activations**

In order to avoid that, one method is to compute the **mean μ** of this data and you would compute the operation $\mathbf{x} = \mathbf{x} - \mu$ to get this type of plot if you replot the transformed data:

⑧ Initialization methods

Normalizing your input :

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

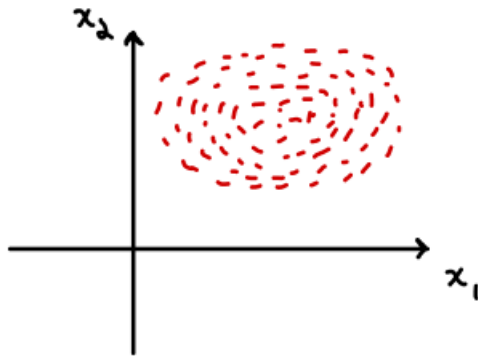


Here it's a little better, but it's still not good. In order to solve the problem fully, we are going to compute σ^2 which is basically the **standard deviation** squared, so the **variance** of the data, and then you will divide by σ^2 . You would do that and make the transformation of \mathbf{x} being equal to \mathbf{x}/σ , and it will give you a graph that is centered

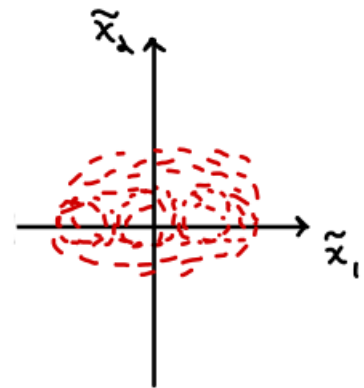
⑧ Initialization methods

Normalizing your input :

$$x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$



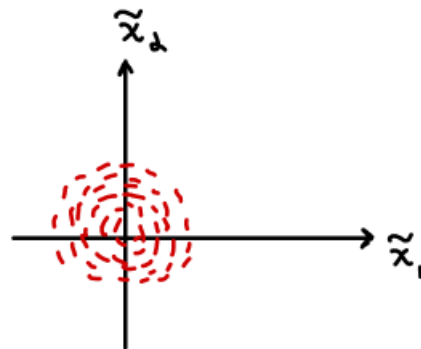
$$x = x - \mu \longrightarrow$$



$$\begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix} = \mu = \frac{1}{n} \sum_{i=1}^n x^{(i)}$$

"sigma" $\rightarrow \sigma^2 = \frac{1}{n} \sum_{i=1}^n (x^{(i)})^2$

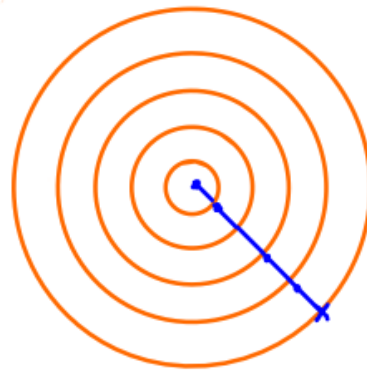
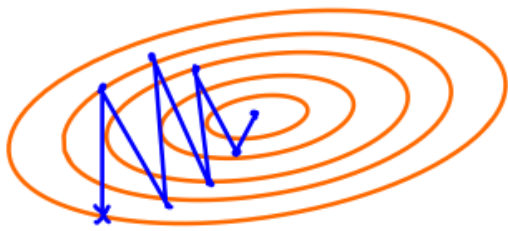
$$x = \frac{x}{\sigma}$$



So you usually prefer to work with a centered data

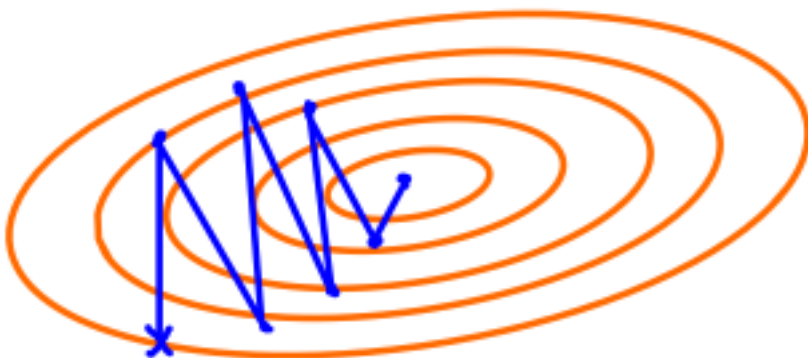
Why is it better?

It's because if you look at your **loss function** before and the **loss function** now, it would look something like this:

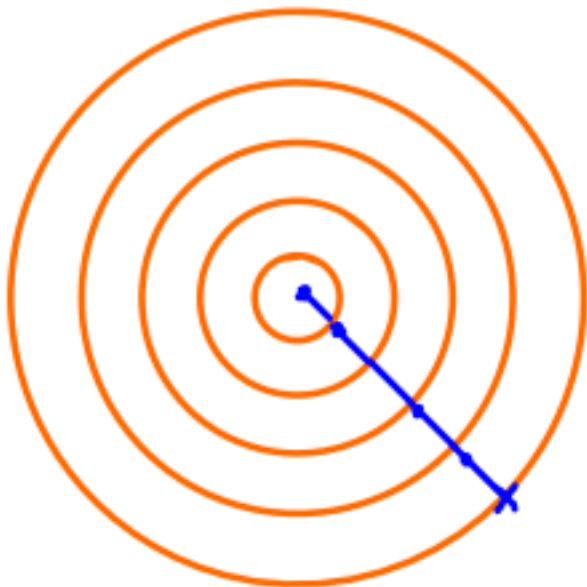


What's the difference between these two **loss functions**? Why is the one on the right easier to train?

It's because, if you have the starting point that is there, for the left **loss function**:



Your **gradient descent** algorithm is going towards approximately the steepest slope until you end up at the right points. But the steepest slope in the **loss function** on the right is always pointing towards the middle



So if you start somewhere, it will directly go towards the **minimum** of your **loss function**. This also leads to fewer iterations

So that's why it's usually helpful to **normalize**

This is one method, and in practice, the way you **initialize** your **weights** is very important

Here, we used a very simple case, but you would divide element-wise by σ :

"sigma" →
$$\sigma^2 = \frac{1}{n} \sum_{i=1}^n (x^{(i)})^2$$

$$x = \frac{x}{\sigma}$$

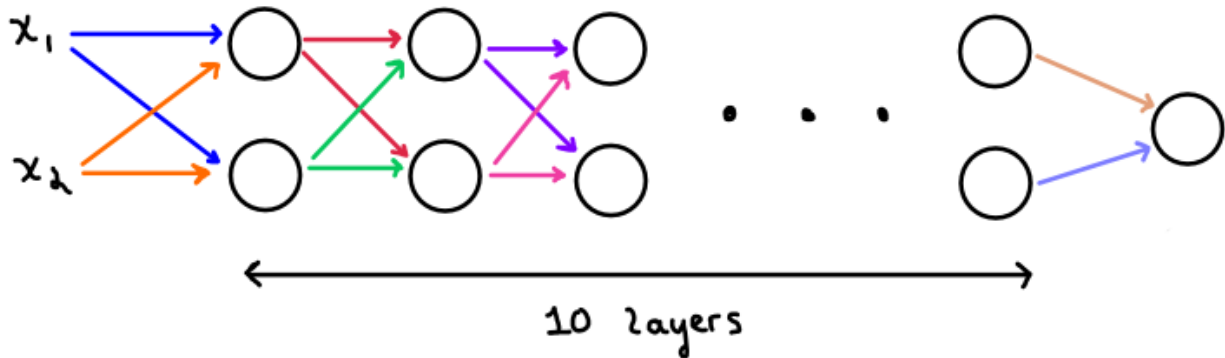
So every entry of your matrix you would divide it by the σ

One other thing that is important to notice is, this σ and μ are computed over the **training set**. You have a **training set**, you compute the **mean** of the **training set**, set the **standard deviation** of the **training set**, and these σ and μ have to be used on the **test set** as well. It means, now that you want to test your algorithm on the **test set**, you should not compute the **mean** of the **test set** and the **standard deviation** of the **test set** and **normalize** your test inputs through the **network**. Instead, you should use the σ and μ that were computed on the **train set** because your **network** is used to seeing this type of transformation as an input. So you'll want the **distribution** of the inputs at the **first neuron** to always be the same, no matter if it's the **train** or the **test set**

Vanishing Gradients and Exploding Gradients:

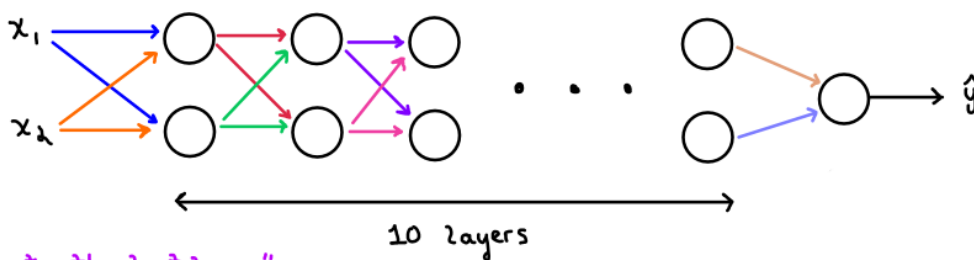
In order to get an intuition of why we have these **vanishing** or **exploding gradient** problems, we can consider a **network** which is very very deep and has a **2-dimensional** input. Let's say we have **10 layers** in total plus an **output layer**

Vanishing Gradients, Exploding Gradients:



Assume all the **activations** are **identity functions** and assume that **biases** are equal to **0**. If you compute \hat{y} 's, the output of the **network** with respect to the input, you know that \hat{y} will be equal to:

Vanishing Gradients, Exploding Gradients:



"denotes the last layer"

$$\begin{aligned}\hat{y} &= w^{[L]} \cdot a^{[L-1]} = w^{[L]} \cdot w^{[L-1]} \cdot a^{[L-2]} \\ &= w^{[L]} \cdot w^{[L-1]} \cdot \dots \cdot w^{[1]} \cdot x\end{aligned}$$

$g : z \mapsto z$
 $b = 0$

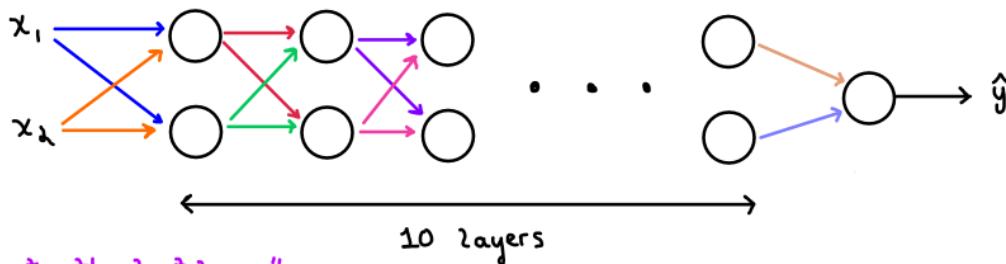
"assume all activations are identity functions"

"assume biases are equal to 0"

You'll get something like that

Now, let's consider two cases. Let's consider the case where the $w^{[l]}$ matrices are a little larger than the **identity function** in terms of values:

Vanishing Gradients, Exploding Gradients:



"denotes the last layer"

$$\hat{y} = w^{[L]} \cdot a^{[L-1]} = w^{[L]} \cdot w^{[L-1]} \cdot a^{[L-2]}$$

$$= w^{[L]} \cdot w^{[L-1]} \cdot \dots \cdot w^{[1]} \cdot x$$

$$g : z \mapsto z$$

$$b = 0$$

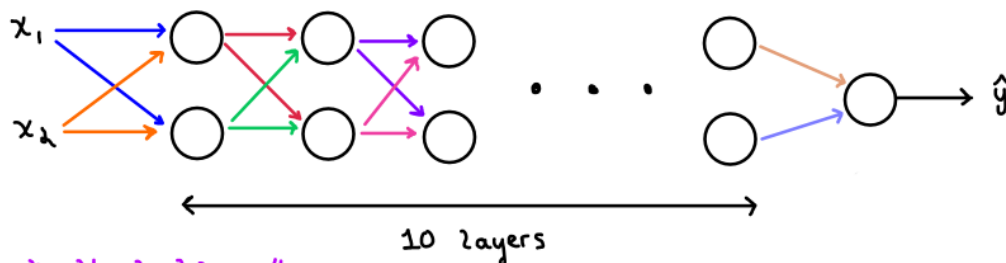
"assume all activations are identity functions"

"assume biases are equal to 0"

$$w^{[1]} = \begin{pmatrix} 1.5 & 0 \\ 0 & 1.5 \end{pmatrix}$$

Let's say $w^{[1]}$, including all these, $(2,2)$ w matrices:

Vanishing Gradients, Exploding Gradients:



"denotes the last layer"

$$\hat{y} = w^{[L]} \cdot a^{[L-1]} = w^{[L]} \cdot w^{[L-1]} \cdot a^{[L-2]}$$

$$= w^{[L]} \cdot w^{[L-1]} \cdot \dots \cdot w^{[1]} \cdot x$$

$$g : z \mapsto z$$

$$b = 0$$

"assume all activations are identity functions"

"assume biases are equal to 0"

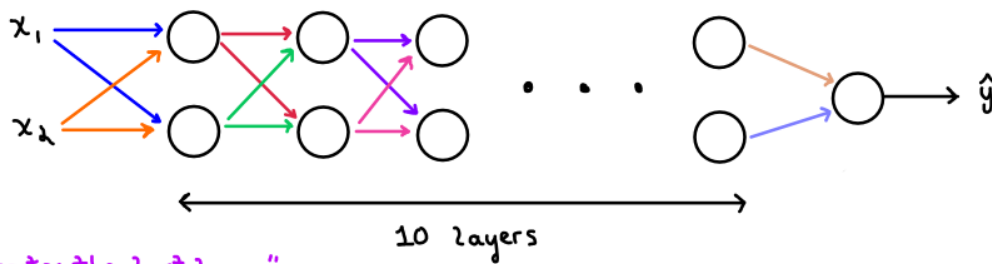
$$w^{[1]} = \begin{pmatrix} 1.5 & 0 \\ 0 & 1.5 \end{pmatrix}$$

(2,2)

What's the consequence?

The consequence is that this whole thing here is going to be equal to:

Vanishing Gradients, Exploding Gradients:



"denotes the last layer"

$$\begin{aligned}\hat{y} &= w^{[L]} \cdot a^{[L-1]} = w^{[L]} \cdot w^{[L-1]} \cdot a^{[L-2]} \\ &= w^{[L]} \cdot w^{[L-1]} \cdot \dots \cdot w^{[1]} \cdot x \\ &\quad \underbrace{\hspace{10em}}_{(2,2)}\end{aligned}$$

$$w^{[1]} = \begin{pmatrix} 1.5 & 0 \\ 0 & 1.5 \end{pmatrix} \rightarrow \begin{pmatrix} 1.5^L & 0 \\ 0 & 1.5^L \end{pmatrix}$$

"assume all activations are identity functions"

$g : z \mapsto z$

$b = 0$

"assume biases are equal to 0"

It will make the value \hat{y} explode just because this number is a tiny little bit more than 1

Same phenomenon, if we had **0.5** instead of **1.5** here, the multiplicative value of all these matrices will be **0.5** to the power **L**, and \hat{y} will always be very close to **0**

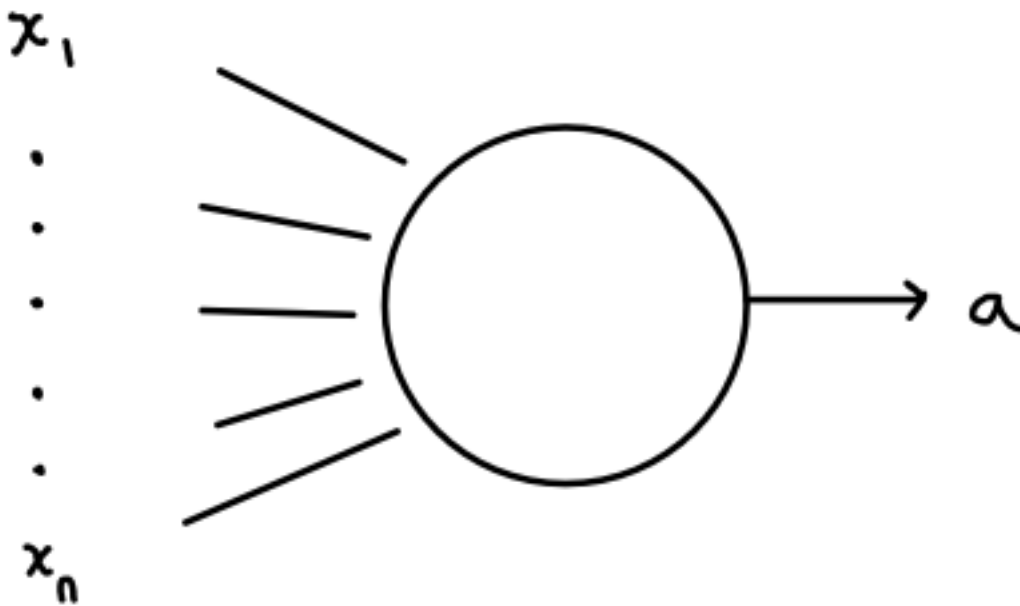
The issue with **vanishing** and **exploding gradients** is that all the errors add up (multiply each other), and if you end up with numbers that are smaller than **1**, you will get a totally **vanished gradient**. When you go back, if you have values that are a little bigger than **1**, you will get **exploding gradients**

We did it as a **forward propagation** equation. We could have done exactly the same analysis we did with derivatives, assuming the derivatives of the **weight** matrices are a little lower than the **identity** or a little higher than the **identity**. So we want to avoid that

One way that is not perfect to avoid this is to **initialize** your **weights** properly. **Initialize** them into the right range of values. So we'd prefer the weight to be around **1**, as close as possible to **1**. If they're very close to **1**, we can probably avoid the **vanishing** and **exploding gradient** problem

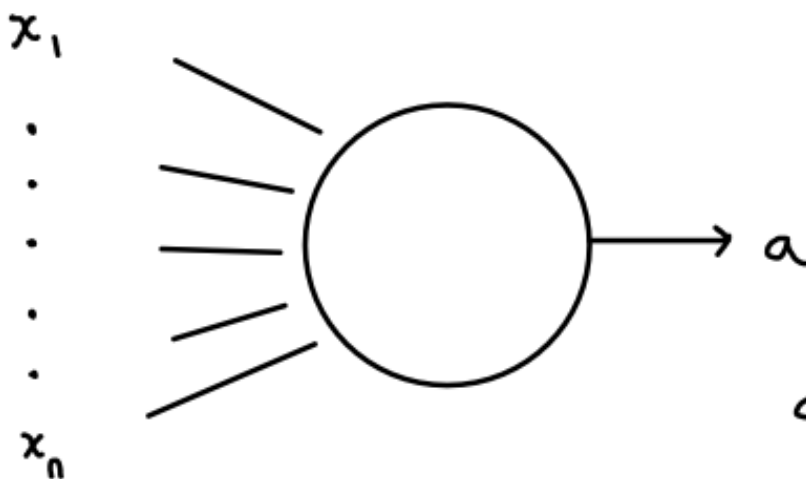
Let's look at the **initialization problem**. The first thing to look at is an example of the **1 neuron**. If you consider this **neuron** here which has a bunch of inputs and outputs an **activation, a**

Example with 1 neuron:



You know that the equation inside the **neuron** is a equals whatever function, let's say $\sigma(z)$, and you know that z is equal to:

Example with 1 neuron:



$$a = \sigma(z)$$

$$z = w_1 x_1 + \dots + w_n x_n$$

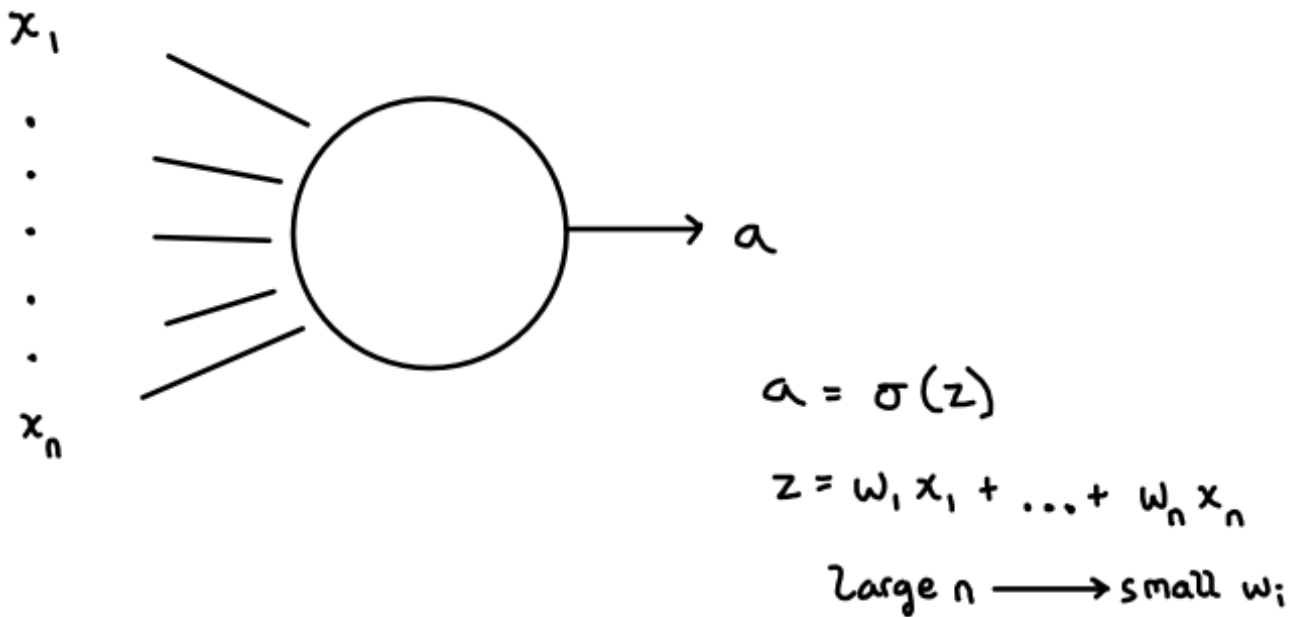
large $n \longrightarrow$ small w_i

It's a **dot product** between the **w**'s and the **x**'s. The interesting thing to notice is that we have **n** terms here. So in order for **z** to not

explode, we would like all of these terms to be small. If w 's are too big, then this term (z) will **explode** with the size of the inputs of the **layer**. So instead, if we have a large n (it means the input is very large), what we want is very small w_i 's. So the larger n , the smaller w_i has to be

Based on this intuition, it seems that it would be a good idea to **initialize** w_i 's with something that is close to $1/n$. We have n terms, the more terms we have the more likely z is going to be big, but if our **initialization** says "the more terms you have, the smaller the value of the **weights**", we should be able to keep z in a certain range that is appropriate to avoid **vanishing** and **exploding gradients**

Example with 1 neuron:



$$w_i \sim \frac{1}{n}$$

"initialize with something that is close to"

So this seems to be a possible **initialization scheme**

We're going to write a few **initialization schemes** (that we're not going to prove). In practice, there are a few **initialization schemes** that are commonly used and again, this is very practical and people have been testing a lot of **initializations** but they ended up using those

One, is to **initialize** the **weights**, written using numpy:

"element-wise times"

$$W_i^{[l]} = \text{np.random.randn}(\text{shape}) * \text{np.sqrt}\left(\frac{1}{n^{[l-1]}}\right)$$

\searrow
 $n^{[l-1]}$

What does that mean? It means that we will look at the number of inputs that are coming into our **layer**, assuming we're at **layer l**, we're going to **initialize** the **weights** of this **layer** proportionally to the number of inputs that are coming in. So the intuition is very similar to what we described previously

This **initialization** has been shown to work very well for **sigmoid activations**

What's interesting, is if you use **ReLU**, it's been observed that putting a **2** in the numerator instead of a **1**, would make the **network** train better. Again, it's very practical

"element-wise times"

$$W_i^{[l]} = \text{np.random.randn}(\text{shape}) * \text{np.sqrt}\left(\frac{1}{n^{[l-1]}}\right)$$

for sigmoid \nearrow

\searrow
 $n^{[l-1]}$

$$W_i^{[l]} = \text{np.random.randn}(\text{shape}) * \text{np.sqrt}\left(\frac{2}{n^{[l-1]}}\right)$$


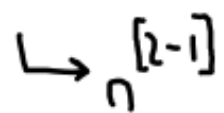
for ReLU \nearrow

\searrow
 $n^{[l-1]}$


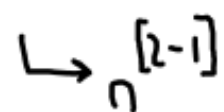
Finally, there is a more common one that is used which is called the **Xavier initialization** which proposes to **update** the **weights**:

"element-wise times"

$$w_i^{[l]} = \text{np.random.randn}(\text{shape}) * \text{np.sqrt}\left(\frac{1}{n^{[l-1]}}\right)$$

for sigmoid  

$$w_i^{[l]} = \text{np.random.randn}(\text{shape}) * \text{np.sqrt}\left(\frac{2}{n^{[l-1]}}\right)$$

for ReLU  

Xavier Initialization:

$$w_i^{[l]} \sim \sqrt{\frac{1}{n^{[l-1]}}} \quad \text{for tanh}$$

This is another one called **He initialization** recommends to **initialize** the **weights** of a **layer** using the following formula:

"element-wise times"

$$w_i^{[l]} = \text{np.random.randn(shape)} * \text{np.sqrt}\left(\frac{1}{n^{[l-1]}}\right)$$

for sigmoid

\searrow $n^{[l-1]}$

$$w_i^{[l]} = \text{np.random.randn(shape)} * \text{np.sqrt}\left(\frac{2}{n^{[l-1]}}\right)$$

for ReLU

\searrow $n^{[l-1]}$

Xavier Initialization:

$$w^{[l]} \sim \sqrt{\frac{1}{n^{[l-1]}}} \text{ for tanh}$$

He Initialization:

$$w^{[l]} \sim \sqrt{\frac{2}{n^{[l]} + n^{[l-1]}}}$$

The quick intuition behind the last one, which is very often used, is that we're doing the same thing but also for the **backpropagated gradients**. So we're saying the **weights** are going to multiply the **backpropagated gradients**, so we also need to look at how many inputs we have during the **backpropagation**, and l is the number of inputs we have during the **backpropagation** and $l-1$ is the number of inputs we have during **forward propagation**, so taking a **geometric average** of those

And the reason we have a **random function** here is because if you don't **initialize** your **weights** randomly, you will end up with some problem called the **symmetry problem**, where every **neuron** is going to learn kind of the same thing. To avoid that, you will make the **neurons** start at different places and let them evolve independently from each other as much as possible

So far we've seen **gradient descent** and **stochastic gradient descent** as two possible **optimization algorithms**. In practice, there is a trade-off between these two which is called **mini-batch gradient descent**

What is the trade-off?

The trade-off is that **batch gradient descent** is cool because you can use **vectorization**. You can give a batch of inputs, **forward propagate** it all at once using **vectorized code**

Stochastic gradient descent's advantage is that the **updates** are very quick

Imagine that you have a dataset with one million images and you want to do **batch gradient descent**. Do you know how long it's going to take to do one **update**? Very long. So we don't want that because maybe we don't need to go over the full dataset in order to have a good **update**. Maybe the **update** based on **1,000** examples might already give us the right direction for the **gradient** of where to go. It's not going to be as good as on the million example, but it's going to be a very good approximation

So that's why most people would use **mini-batch gradient descent**, where you have the trade-off between **stochasticity** and also **vectorization**

In terms of notation, we're going to call **X** the matrix $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots, \mathbf{x}^{(m)}$ and **Y** the same matrix with **y**'s

③ Optimization

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}^{(1)} & \mathbf{x}^{(2)} & \dots & \mathbf{x}^{(m)} \end{pmatrix}$$

$$\mathbf{y} = \begin{pmatrix} y^{(1)} & \cdot & \cdot & \cdot & y^{(m)} \end{pmatrix}$$

We have **m** training examples and we're going to split these into **batches** using the notation **{}** to denote the batch

③ Optimization

$$X = (x^{(1)} \ x^{(2)} \ \dots \ x^{(n)})$$

$$y = (y^{(1)} \ \dots \ y^{(n)})$$

"batch"

$$X = (x^{\{1\}} \ \dots \ x^{\{T\}})$$

$$\downarrow$$
$$(x^{(1)} \ \dots \ x^{(1000)})$$

$$y = (y^{\{1\}} \ \dots \ y^{\{n\}})$$

In terms of algorithm:

© Optimization

$$X = (x^{(1)} \ x^{(2)} \ \dots \ x^{(n)})$$

$$y = (y^{(1)} \ \dots \ y^{(n)})$$

"batch"

$$X = (x^{(1)} \ \dots \ x^{(T)})$$

$$(x^{(1)} \ \dots \ x^{(1000)})$$

$$y = (y^{(1)} \ \dots \ y^{(n)})$$

Algorithm:

For iteration $t=1 \dots$

Select batch $(x^{(t)}, y^{(t)})$

Forward propagate the batch

Backpropagate the batch

Update $w^{[l]}, b^{[l]}$

$$J = \frac{1}{1000} \sum_{i=1}^{1000} L^{(i)}$$

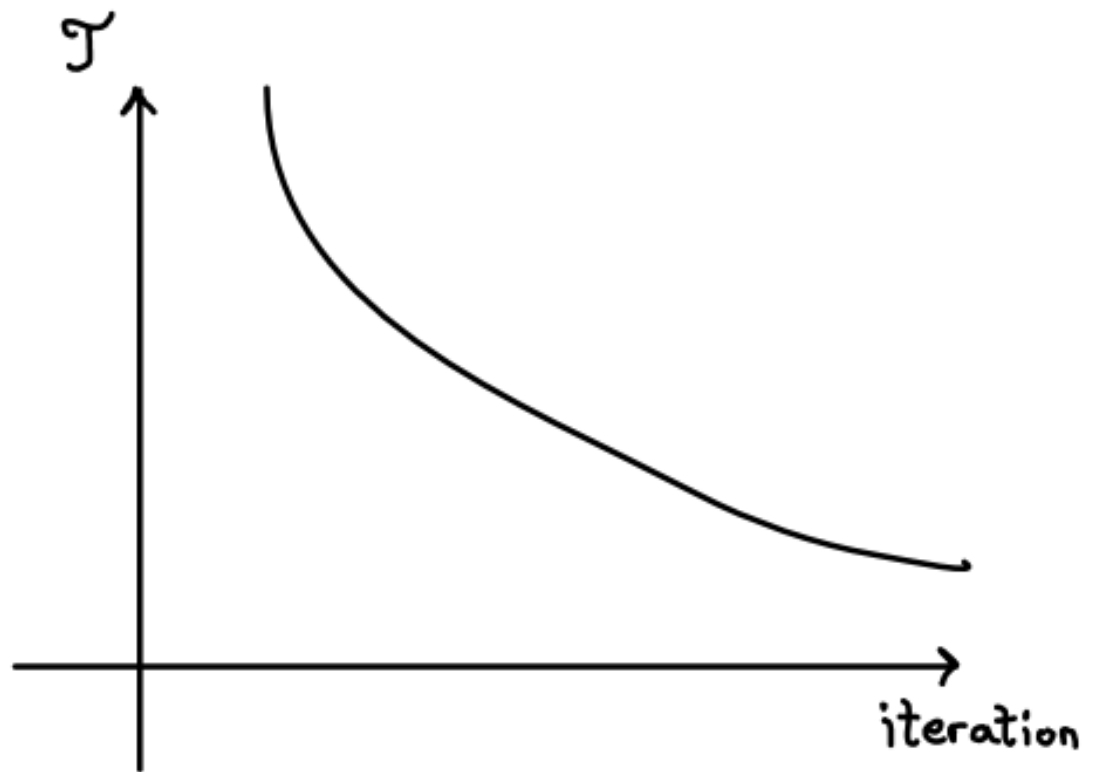
"to how many iterations you want to do"

By **forward propagation**, we mean, you send all the **batch** through the **network** and you compute the **loss functions** for every example of the **batch**, you sum them together and you compute the **cost function** over the entire **batch**, which is the average of the **loss functions**

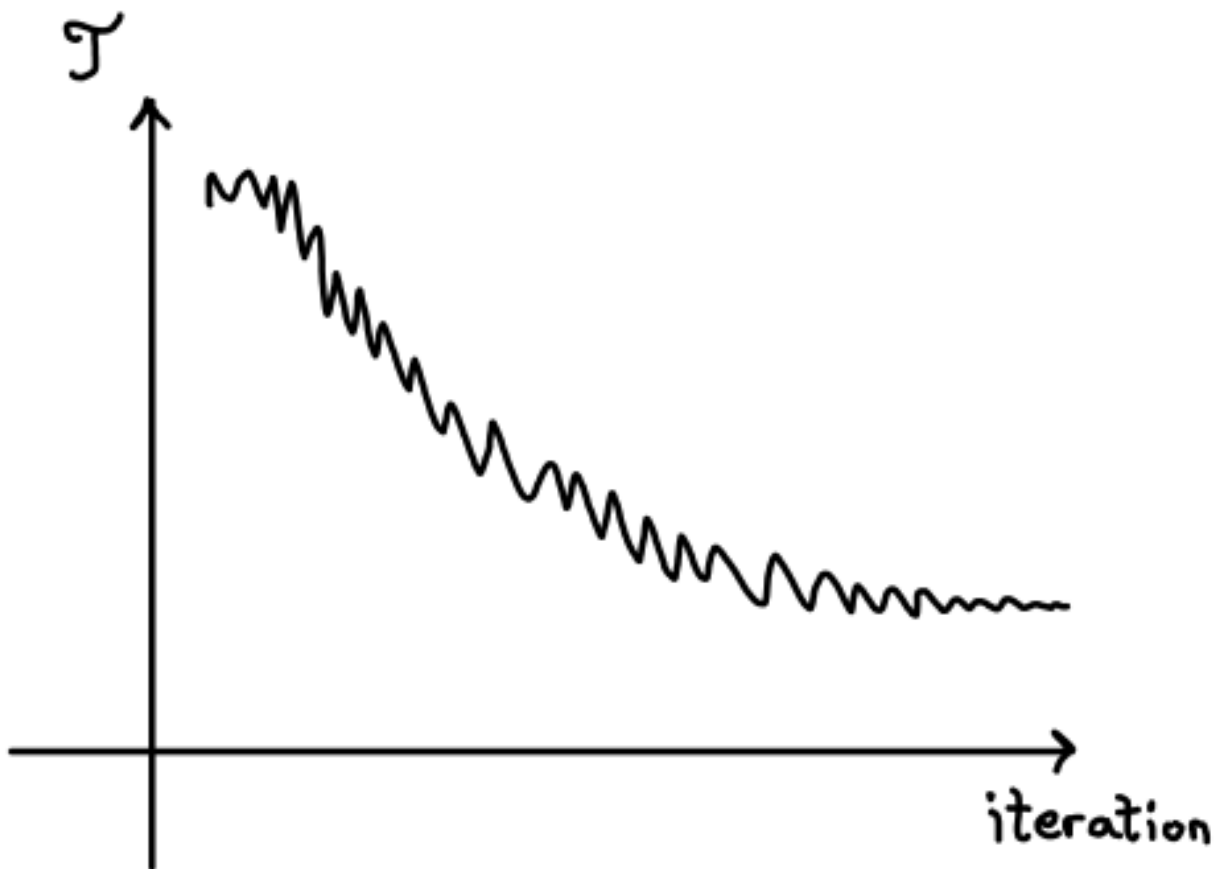
Assuming the batch is of size **1,000**, this would be the formula to compute the **batch** over **1,000** examples. And after the **backpropagation**, update $w^{[l]}$ and $b^{[l]}$ for all the l 's (for all the **layers**)

In terms of graph, what you're likely to see is that for **batch gradient descent**, your **cost function J** would have looked like this if you plot it against the number of iterations:

graph:



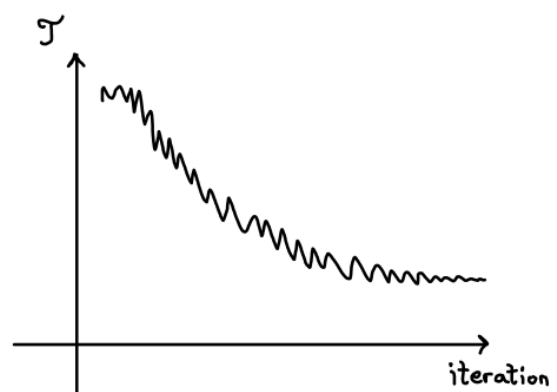
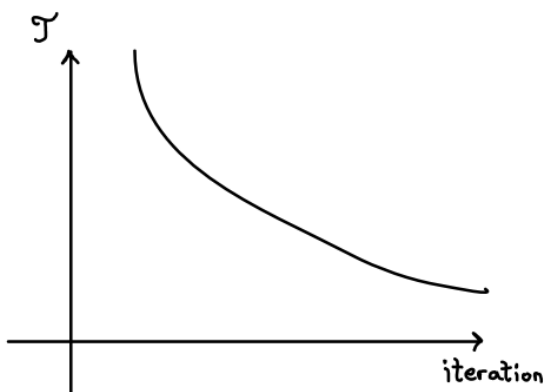
On the other hand, if you use a **mini-batch gradient descent**, you're most likely to see something like this:



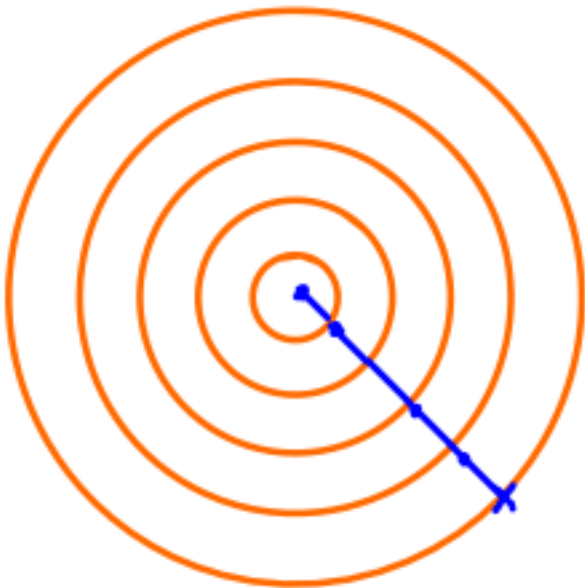
So it's also decreasing as a trend, but because the **gradient** is approximated and doesn't necessarily go straight to the lower point of the **loss function**, you will see the above kind of graph

The smaller the **batch**, the more **stochasticity**, so the more noise you will have on your **cost function** graph

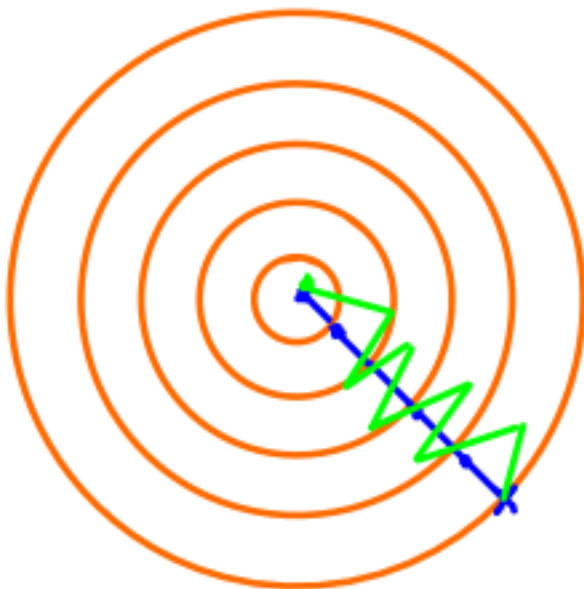
graph:



And if we plot the **loss function** and this was **gradient descent**:



This is the top view of the **loss function**, assuming we're in two dimensions, your **stochastic gradient descent** or **batch gradient descent** would do something like this:



So the difference is there seems to be less iteration with the (blue) algorithm but the iterations are much heavier to compute. Each of the (green) iterations are going to be very quick while the (blue) ones are going to be slow to compute. This is a trade-off

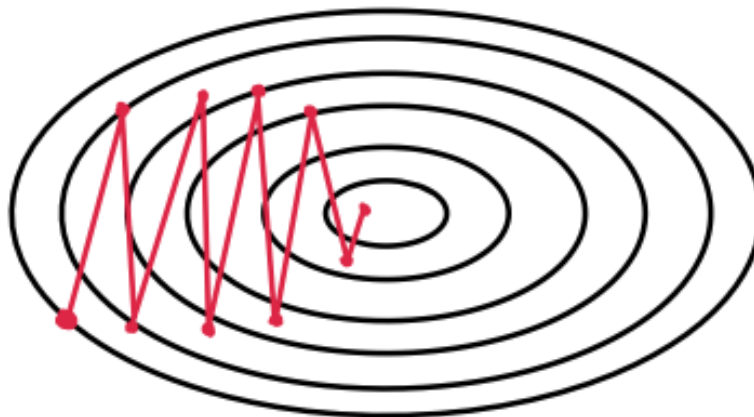
We'll talk about another algorithm which is the **momentum algorithm**, sometimes called **gradient descent + momentum algorithm**

Gradient Descent + Momentum algorithm:

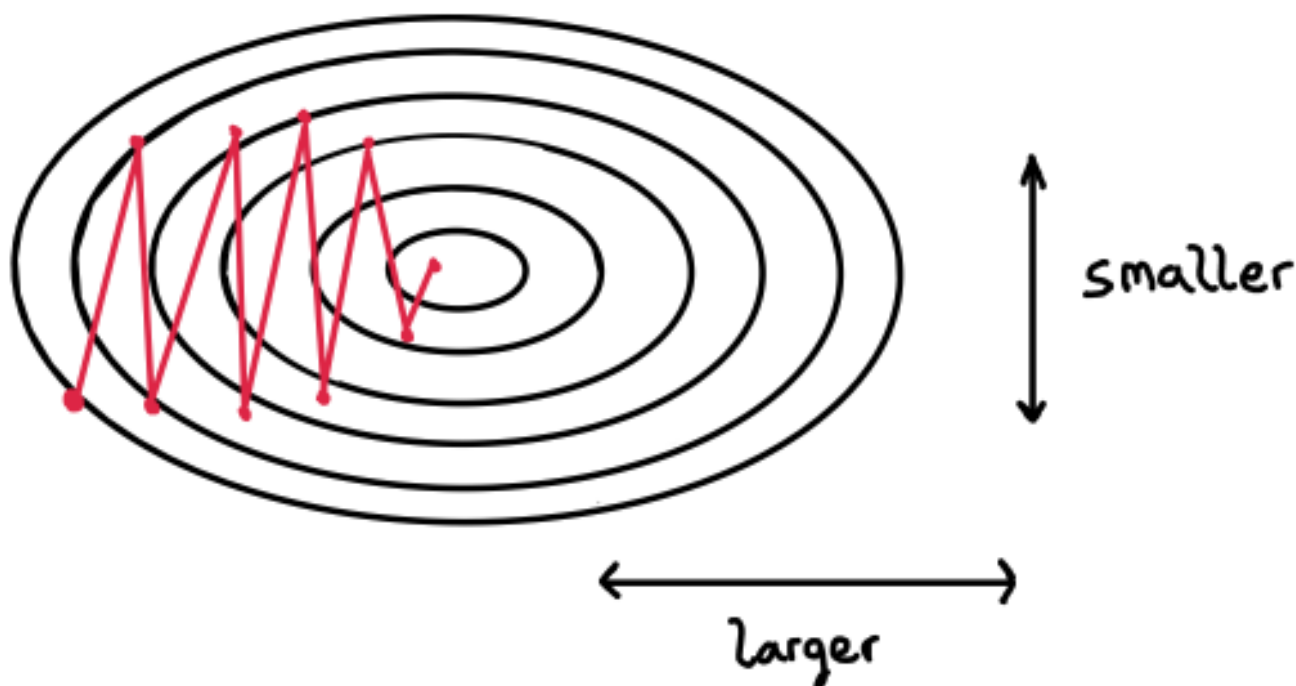
What's the intuition behind **momentum**?

If we look at this **loss contour plot** (we're doing an extreme case just to illustrate the intuition), assume you have the **loss** that is very extended in one direction (horizontal) and the other one is smaller (vertical). You're starting at a point like below. Your **gradient descent** algorithm itself is going to follow the falling bar. It's going to be **orthogonal** to the current **contour loss**

Gradient Descent + Momentum algorithm:



What you would like is to move it **faster** on the **horizontal** line and **slower** on the **vertical** side



So on the **vertical axis**, you would like to move with smaller **updates** and on the **horizontal axis**, you would want to move with larger **updates**. If this happened, we would probably end up in the **minimum** much quicker than we currently are

In order to do that, we're going to use a technique called **momentum** which is going to look at the past **gradients**. **Gradient descent** doesn't look at its past at all, you will just compute the **forward propagation**, compute the **backpropagation**, look at the direction and go to that direction. What **momentum** is going to say is look at the **past updates** that you did and try to consider these **past updates** in order to find the right way to go

If you look at the **past update** and you take an average of the **past update**, you would take an average of this **update** going up:

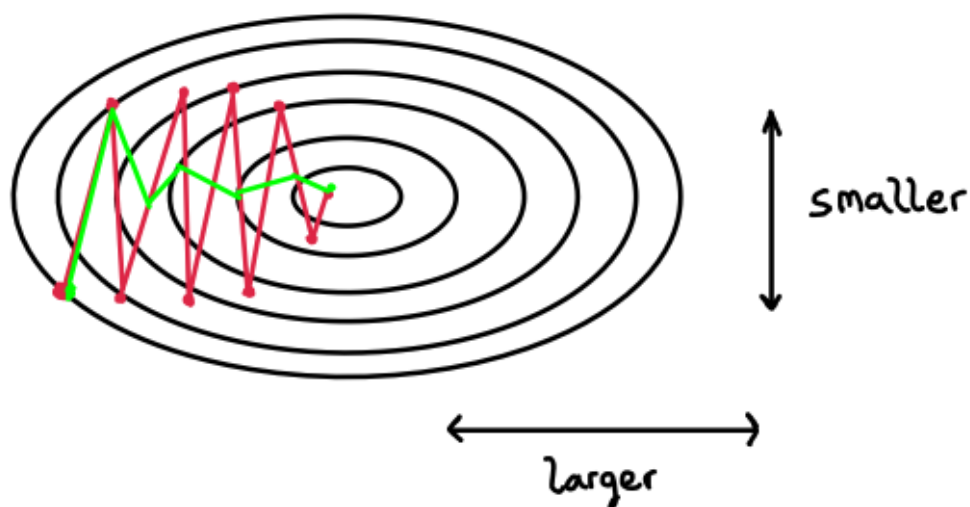


and the **update** after it going down:



The average on the **vertical** side is going to be **small** because one went up and one went down, but on the **horizontal** axis, both went to the same direction, so the **update** will not change too much on the **horizontal** axis. You're most likely to do something like this if you use **momentum**:

Gradient Descent + Momentum algorithm:



That's the intuition on why we want to use **momentum**

You can think of **momentum** as friction. If you launch a rocket and you want to move it quickly around, it's not going to move because the rocket has a certain **weight** and has a certain **momentum**. You cannot change its direction very very noisily

Implementation of momentum gradient descent:

Gradient descent was the following:

$$w = w - d \frac{\partial L}{\partial w}$$

What we're going to do is we're going to use another variable called **velocity**, **v**, which is going to be the average of the **previous velocity** and the **current weight updates**, and instead of the **updates** being the derivative directly, we're going to update the **velocity**

$$\begin{cases} v = \beta v + (1 - \beta) \frac{\partial L}{\partial w} \\ w = w - dv \end{cases}$$

So the velocity is going to be a variable that tracks the direction that we should take regarding the **current update** and also the **past updates**, with a factor **β** that is going to be the **weights**

$$w = w - d \frac{\partial L}{\partial w}$$

$$\begin{cases} v = \beta v + (1 - \beta) \frac{\partial L}{\partial w} \\ w = w - dv \end{cases}$$

The interesting point is that in terms of implementation, it's one more line of code. In terms of memory, it's just one additional variable, and it actually has a big impact on the **optimization**

#!*!# RMSProp and **Adam** are used the most in deep learning. The reason is, if you come up with an **optimization algorithm**, you still have to prove that it works very well on the wide variety of applications before researchers adopt it for their research. **Adam** brings **momentum** to the **deep learning optimization algorithms**