

# **Deep Learning, Introduction and Advanced Concepts**

---

## **1. From Linear Regression to Deep Neural Networks**

Louis Bagot  
Université Claude Bernard Lyon 1

November 23, 2025

## Course info

Course written by Louis Bagot.

Inspired by courses from

- Deep Learning book by Goodfellow, Bengio and Courville
- Denis Coquenet, Université Rennes
- Mathieu Lefort, Université Rennes and Lyon
- Michael Nielsen's book on Neural Networks and Deep Learning
- 3blue1brown (Grant Sanderson) video series on Deep Learning
- Welsch Labs videos on Deep Learning

## Objective

Continuation of Machine Learning:  
Match a desired function,  $f_{\theta} \approx g : \mathcal{X} \rightarrow \mathcal{Y}$

# Objective

Continuation of Machine Learning:

Match a desired function,  $f_{\theta} \approx g : \mathcal{X} \rightarrow \mathcal{Y}$

## Artificial Intelligence

### Machine Learning

#### Supervised Learning

Learn from examples of  $g$

$$g(x_i) = y_i$$

#### Unsupervised Learning

Find structure in data

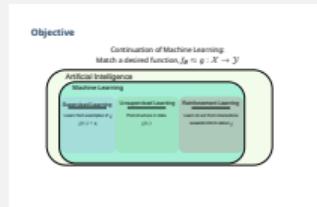
$$g(x_i)$$

#### Reinforcement Learning

Learn to act from interactions

*rewards* inform about  $g$

## Notes for this slide

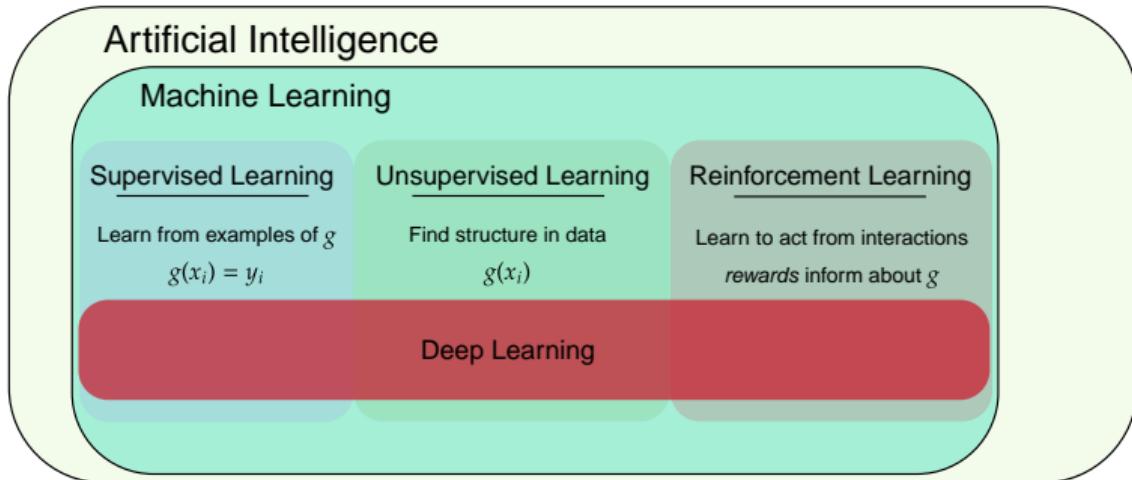


- *Supervised Learning*: learn to match inputs  $\mathbf{x}_i$  and outputs  $\mathbf{y}_i$ .  
This typically requires a big dataset of labelled examples, e.g., pictures of animals and the name of the corresponding animal.  
→  $g$  is the “perfect” function with optimal outputs (matching the dataset)
- *Unsupervised Learning*: extracting information about a dataset of unlabelled data.  
This can take many forms: identifying clusters/groups, relationships between datapoints, key identifying features...  
→  $g$  contains this information (e.g. good clustering  $g(\mathbf{x}_i) = \text{cluster } 3$ )
- *Self-supervised* learning exists in-between supervised and unsupervised to describe problems where the model learns a task that did not require explicit labelling (image reconstruction, next token prediction)
- (RL on next slide note)

# Objective

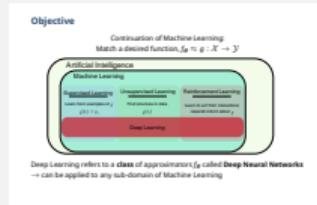
Continuation of Machine Learning:

Match a desired function,  $f_{\theta} \approx g : \mathcal{X} \rightarrow \mathcal{Y}$



Deep Learning refers to a **class** of approximators  $f_{\theta}$  called **Deep Neural Networks**  
→ can be applied to any sub-domain of Machine Learning

## Notes for this slide



- *Reinforcement Learning*: problem of sequential decision making: the model (agent) needs to make a good sequence of decisions (games, robotics).  
As opposed to supervised learning, we do not know the answer (output label) to the question (input): for example in chess, i do not know the optimal move, i cannot teach a supervised model. Instead, the agent learns from interacting with the world, through trial and error.  
The objective is formulated as a reward function (e.g. +1 if I win, -1 if I lose). →  $g$  is the optimal policy, the behavior that gathers the most possible reward in all situations

## Limitations of “Shallow” Machine Learning

Why do we need Deep Neural Nets?

- Huge input and output spaces
- Huge amounts of data
- Highly non-linear problems
- Unstructured data types (image, audio, video)

## Limitations of “Shallow” Machine Learning

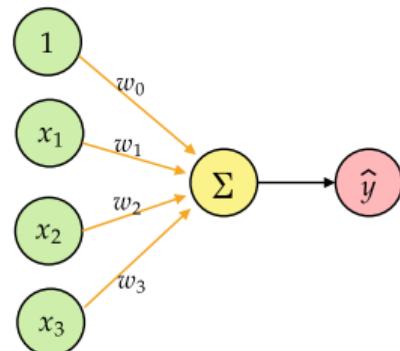
<b>Text</b>	<b>Image</b>	<b>Audio &amp; Signal</b>	<b>Decision</b>
document analysis	image recognition	audio recognition	chess
sentiment analysis	segmentation	forecasting	starcraft
translation	style transfer	translation	minecraft
text generation	image generation	audio generation	robotics

## From Linear Regression to Deep Learning

We want a strong approximator  $f_{\theta} \approx g : \mathcal{X} \rightarrow \mathcal{Y}$ .

Start from linear regression/classification:

$$\begin{aligned}f_{\theta}(\mathbf{x}) &= w_0 + w_1x_1 + w_2x_2 + \cdots + w_dx_d \\&= \boldsymbol{\theta}^\top \mathbf{x} \approx g(\mathbf{x})\end{aligned}$$



## Notes for this slide

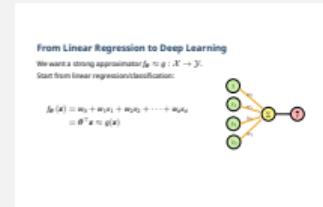
For linear regression,

$$\begin{aligned}\mathbf{x} &\in \mathcal{X} \subseteq \mathbb{R}^d, \\ \boldsymbol{\theta} &= \{w_i\} \in \mathbb{R}^{d+1} \\ g(\mathbf{x}) &\in \mathcal{Y} \subseteq \mathbb{R}\end{aligned}$$

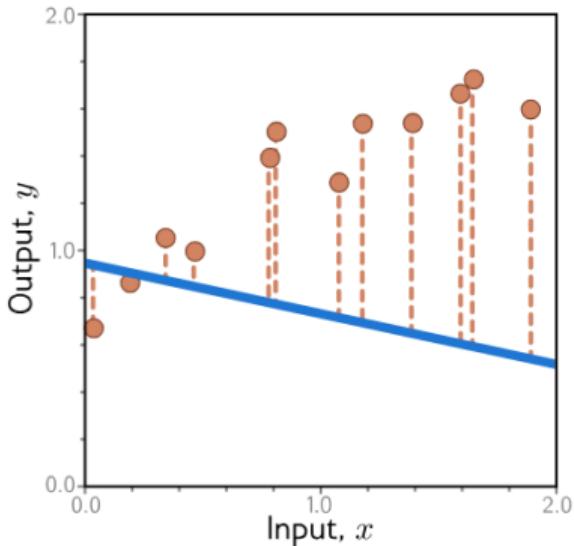
If you have a linear relationship, raising the variable 1 by some value  $\delta x$  always raises the variable 2 by some  $\delta y$ .

Some simple examples of linear relationships:

- predicting the price of a house/apartment w.r.t the number of  $m^2$ 
  - Paris:  $\approx 10000\text{€}/m^2$ , Lyon:  $\approx 5000\text{€}/m^2$ .
- calories burnt vs effort duration
- displacement vs time at fixed speed
  - regression finds speed



# Linear Regression intuition



$$\text{Model: } y = \phi_0 + \phi_1 x$$



Intercept,  $\phi_0$

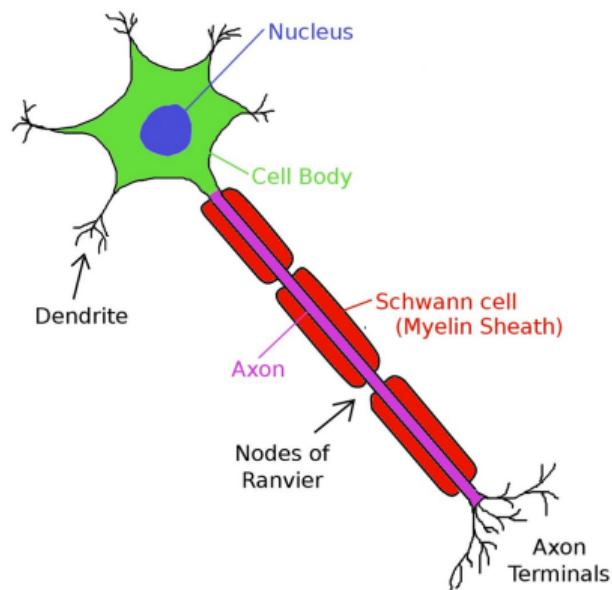


Slope,  $\phi_1$

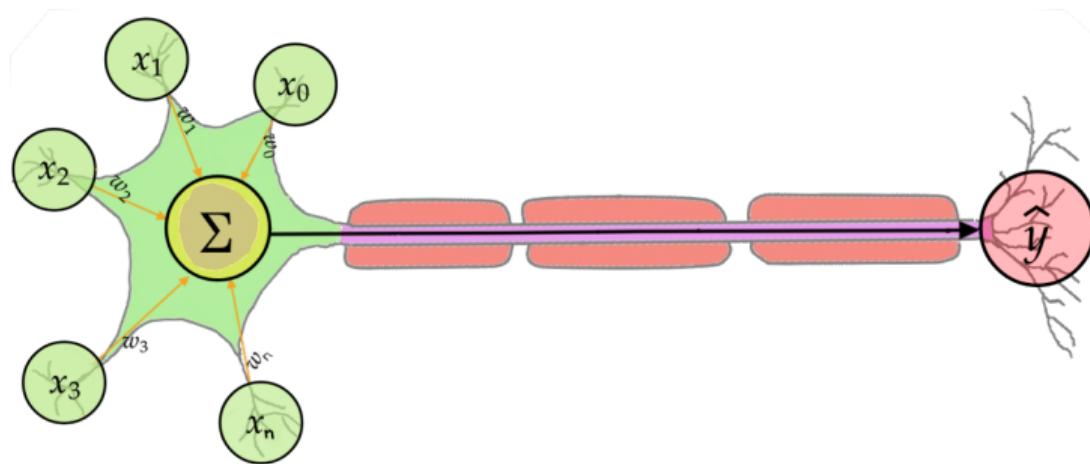
Loss: 6.61

Find the parameters that make the best line through the data  
Try it yourself! (figure 2.2, press the right arrow once)

## Concept of Perceptron: inspiration from neurons

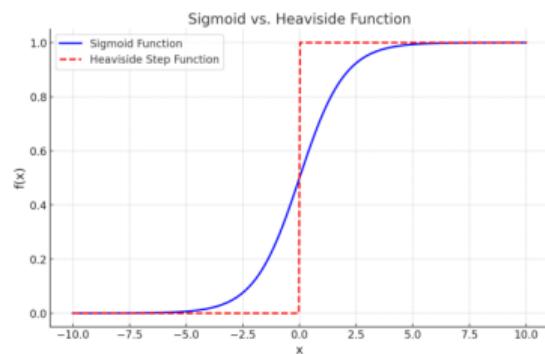
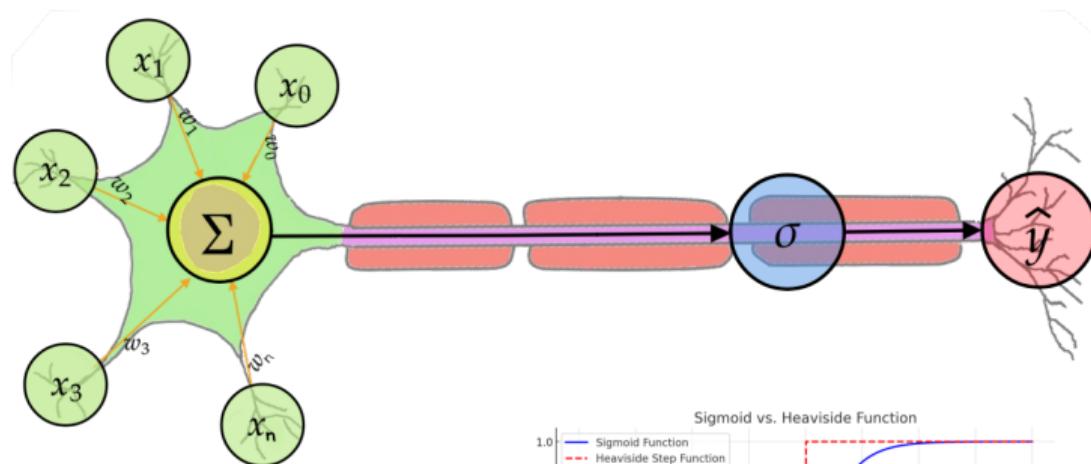


## Concept of Perceptron: inspiration from neurons



this looks like a linear regressor!  
...but a neuron output is on/off

## Concept of Perceptron: inspiration from neurons



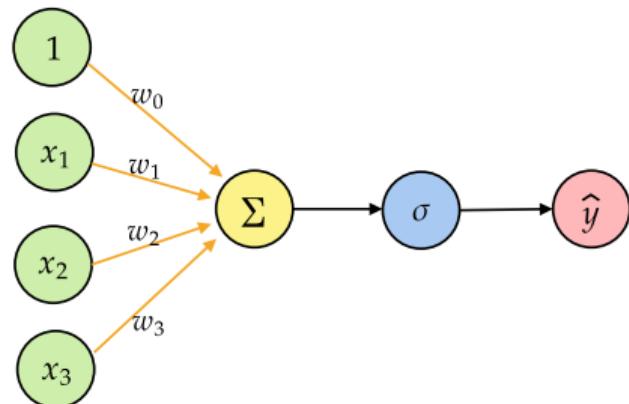
## Concept of Perceptron

If we need  $f_{\theta}(\mathbf{x}) \in [0, 1]$  (true/false, prob), reshape the output (function  $\sigma$ )

Heaviside (H):  $\sigma(x) = 1_{x>0}$

Sigmoid:  $\sigma(x) = \frac{1}{1 + e^{-x}}$

- Perceptron = using Heaviside  
→ non differentiable! weaker learning algo
- Logistic Regression = using Sigmoid  
→ differentiable! gradient descent



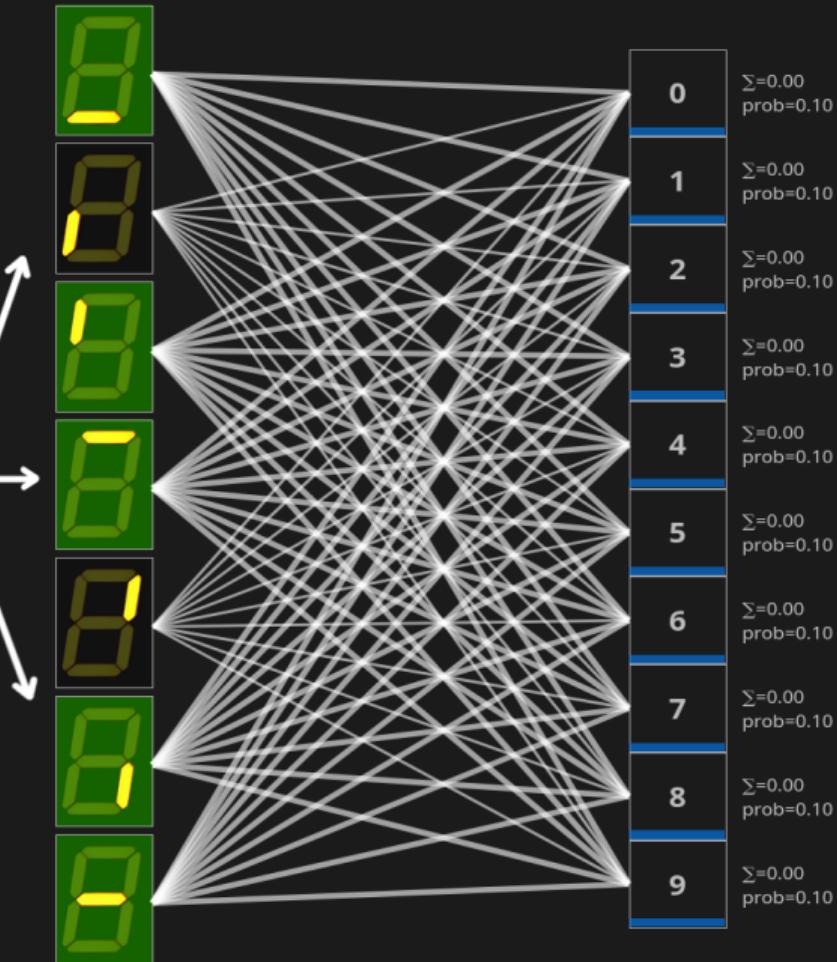
## Extracting Features

"If I could **extract good features**,  
linear/logistic regression would be enough!"

## Extracting Features

"If I could **extract good features**,  
linear/logistic regression would be enough!"

*Feature*: something that describes the input  $\mathbf{x}$  in a "useful" way,  $\phi(\mathbf{x})$ .  
→ even better when we can find multiple  $\phi_i(\mathbf{x})$  coherent with each other!



## **Notes for this slide**

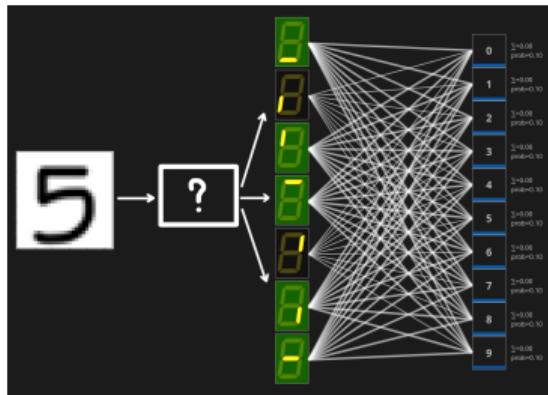
If we could extract, from the handwritten digit (here, 5), the segments that make it up (like on a 7-segment clock), then it would be possible to guess the right number with just a linear model!

This will be done in the lab.

# Extracting Features

"If I could **extract good features**,  
linear/logistic regression would be enough!"

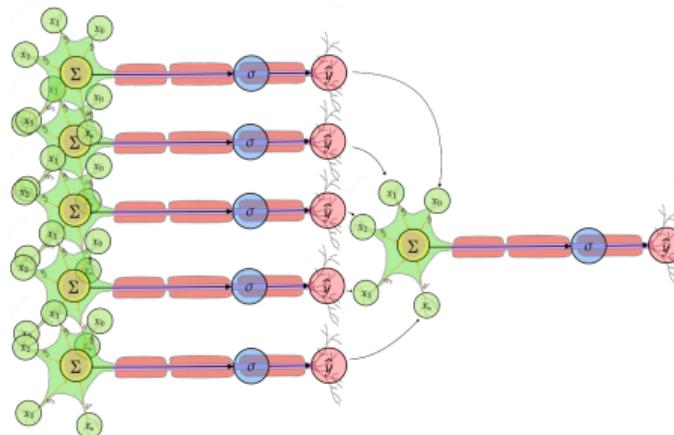
*Feature*: something that describes the input  $\mathbf{x}$  in a "useful" way,  $\phi(\mathbf{x})$ .  
→ even better when we can find multiple  $\phi_i(\mathbf{x})$  coherent with each other!



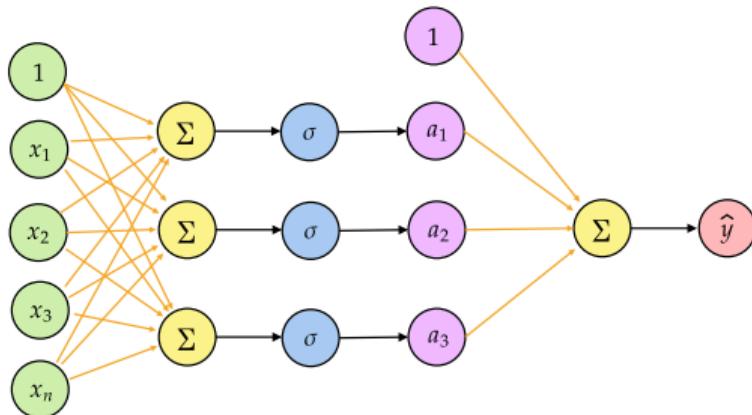
[?] = decades of feature extraction research

# The first hidden layer

Key intuition: **Chain** perceptrons/logistic regressions:  
→ learn the features as well!

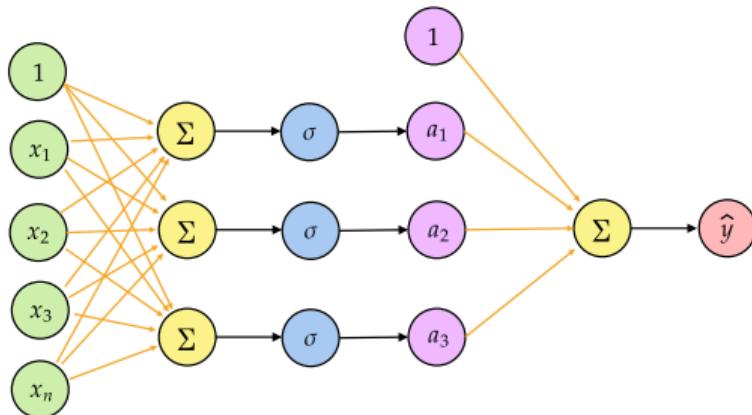


## The first hidden layer



→ Now we can talk about a “**Neural Network**”!  
aka “Multilayer Perceptron”, MLP

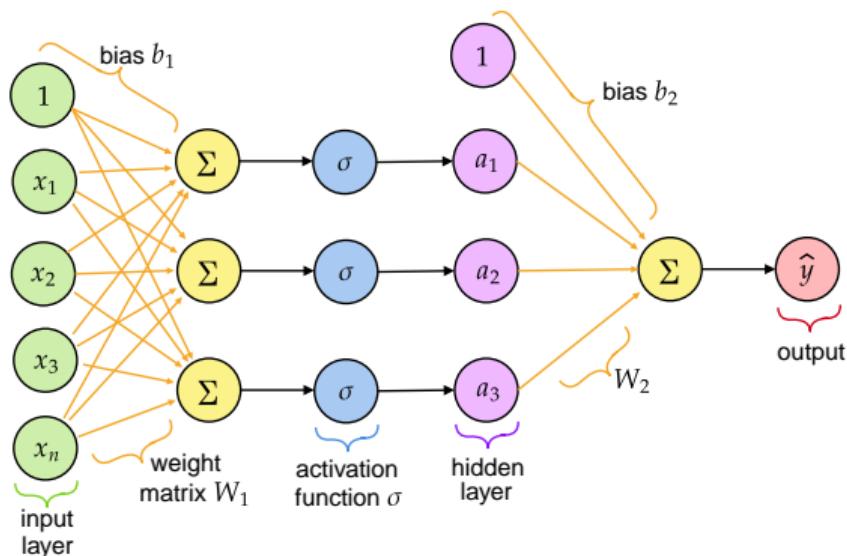
## The first hidden layer



→ Now we can talk about a “**Neural Network**”!  
aka “Multilayer Perceptron”, MLP

Info flows from input to output (no loops): aka “Feedforward” neural network

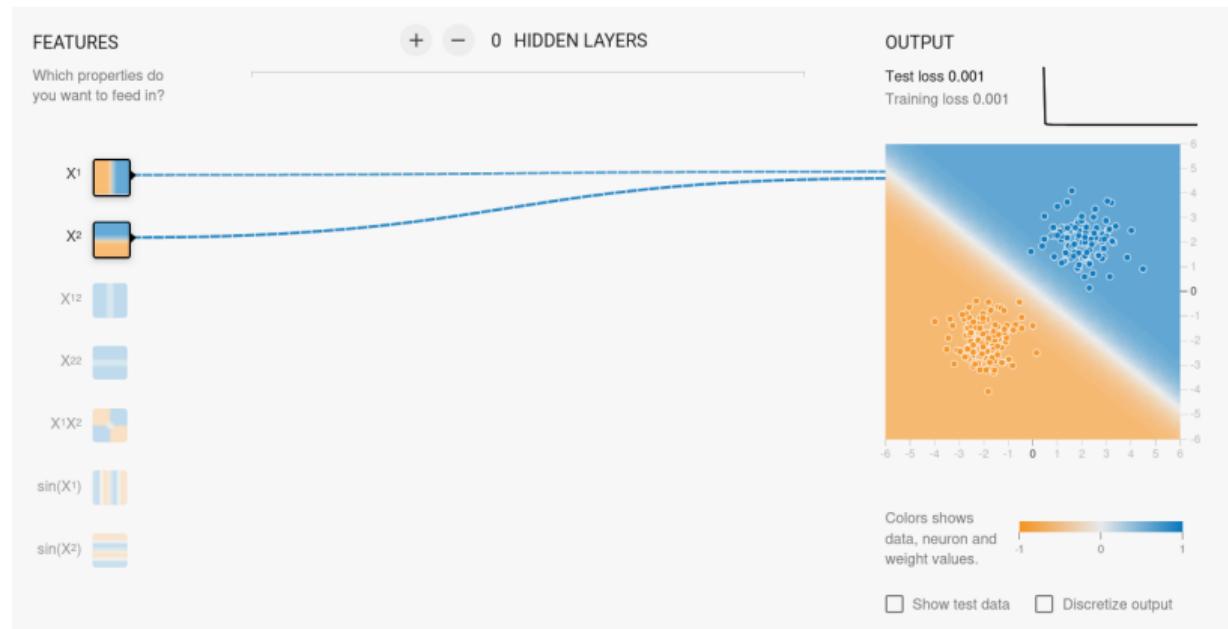
# Definitions in the neural network



$$f_{\theta}(\mathbf{x}) = W_2 \sigma(W_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2$$

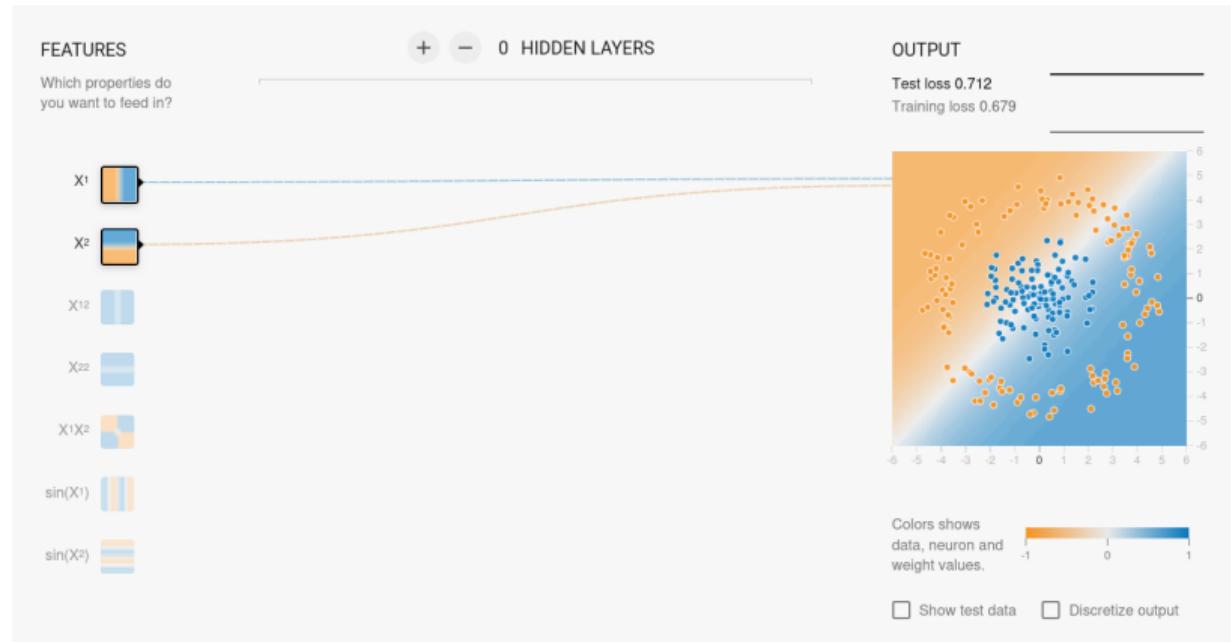
Let's see how this behaves!

# Visualizing the benefits of a hidden layer with Playground



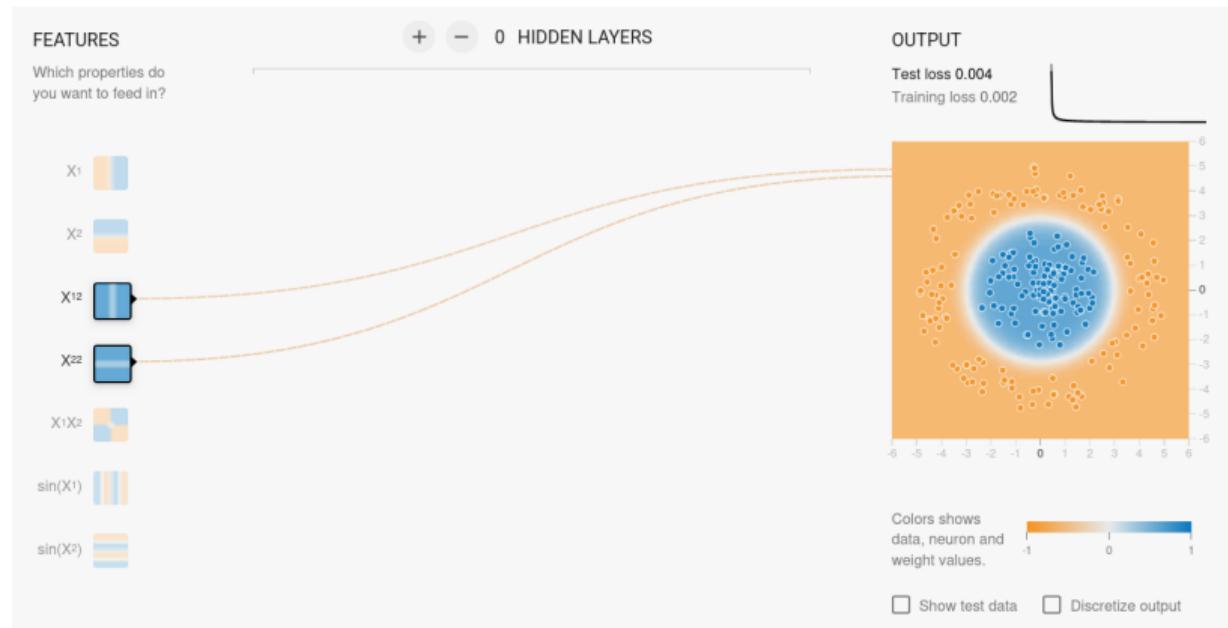
Linear regression/classification: easy!

# Visualizing the benefits of a hidden layer with Playground



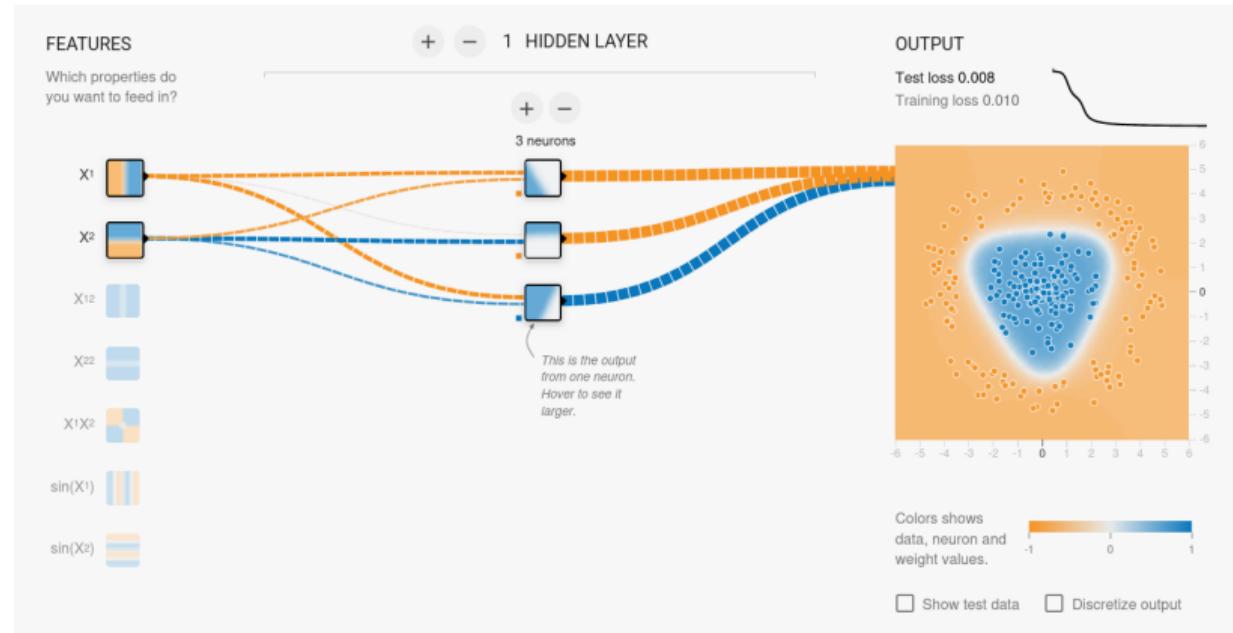
With a linear model, we can never model non-linear relationships...

# Visualizing the benefits of a hidden layer with Playground



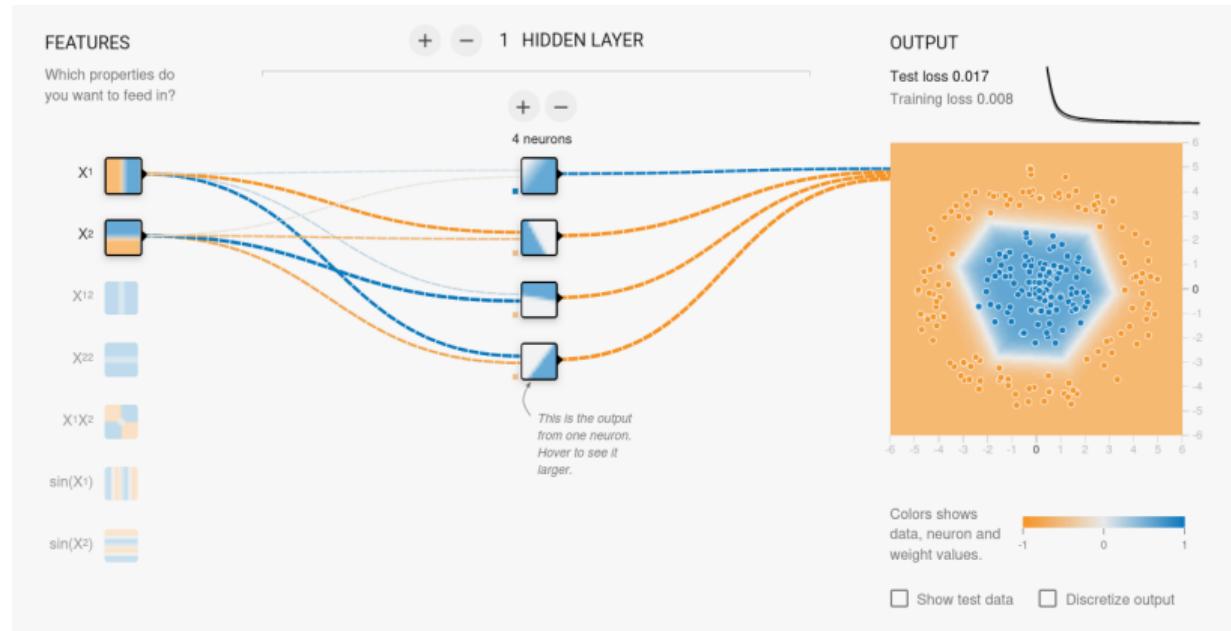
It would be so nice if we just had the right features!

# Visualizing the benefits of a hidden layer with Playground



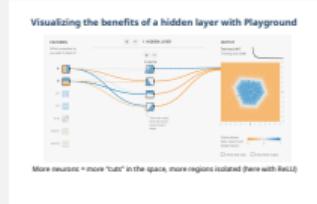
Simply add a hidden layer, learn useful features!

# Visualizing the benefits of a hidden layer with Playground



More neurons = more “cuts” in the space, more regions isolated (here with ReLU)

## Notes for this slide



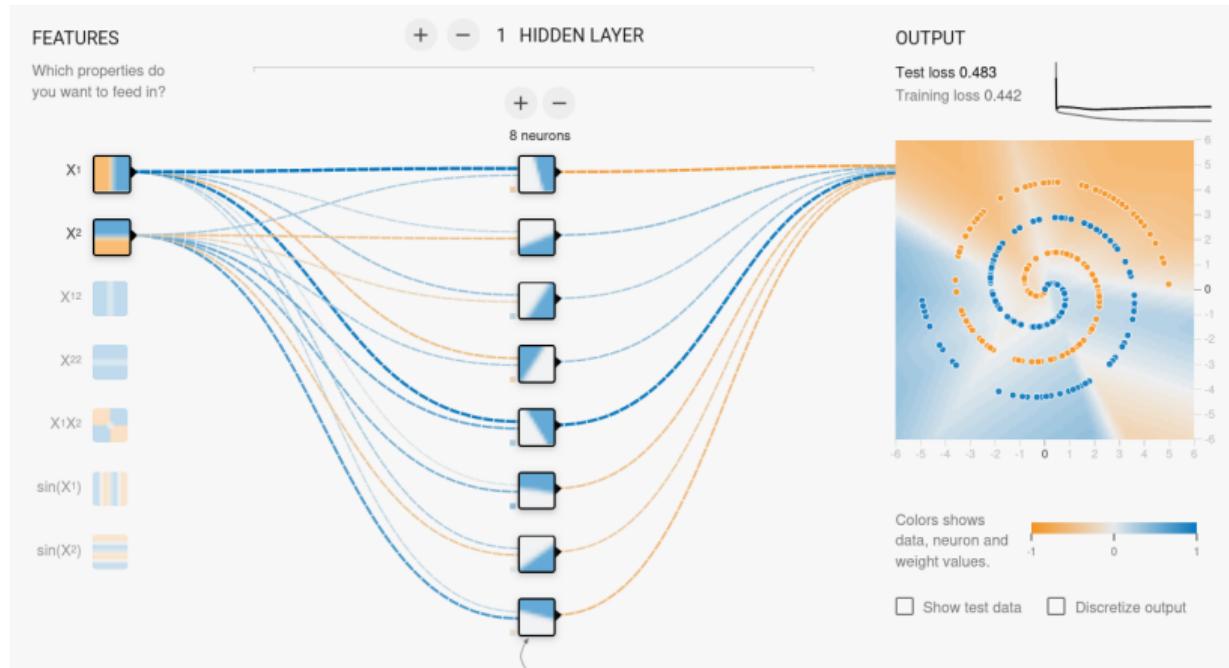
ReLU:

$$\text{ReLU}(x) = \max(0, x)$$

Extremely popular activation function, more neural nets today use either ReLU or variants of it. → Simply “erase” negative inputs onto 0.

→ leads to clear “cuts” in the space

# Visualizing the benefits of a hidden layer with Playground



We would need a lot more neurons to cut this shape!

## Universal Approximation Theorem

IF

- I have enough neurons in the hidden layer
- The *activation function*  $\sigma$  is not a polynomial

THEN

- A one-layer neural network can approximate any  $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$
- (i.e., for any  $g$ , there exists a  $\theta^*$  with which  $f_{\theta^*}$  approximates  $g$ )

# Universal Approximation Theorem

We're done! 🎉

# Universal Approximation Theorem

We're done! 🎉

HOWEVER

- number of hidden neurons  $k$  can be absurdly high
- there is no guarantee we can **find  $\theta^*$**  with any method
  - certainly not with gradient descent
- a bunch of other universal approximators exist (polynomials, Fourier)

# Universal Approximation Theorem

We're done! 🎉

HOWEVER

- number of hidden neurons  $k$  can be absurdly high
- there is no guarantee we can **find  $\theta^*$**  with any method
  - certainly not with gradient descent
- a bunch of other universal approximators exist (polynomials, Fourier)



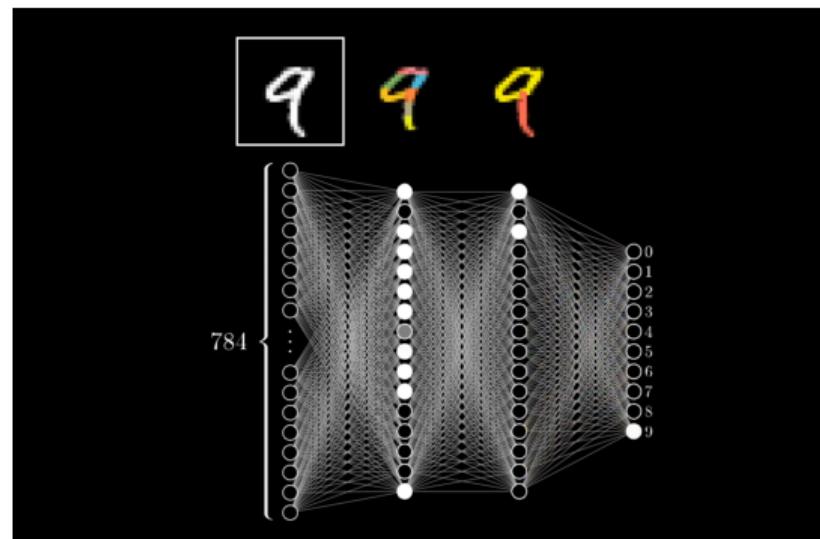
## Features to your features

If your problem is really hard, you might need *basic features*  
to extract *more complex features*

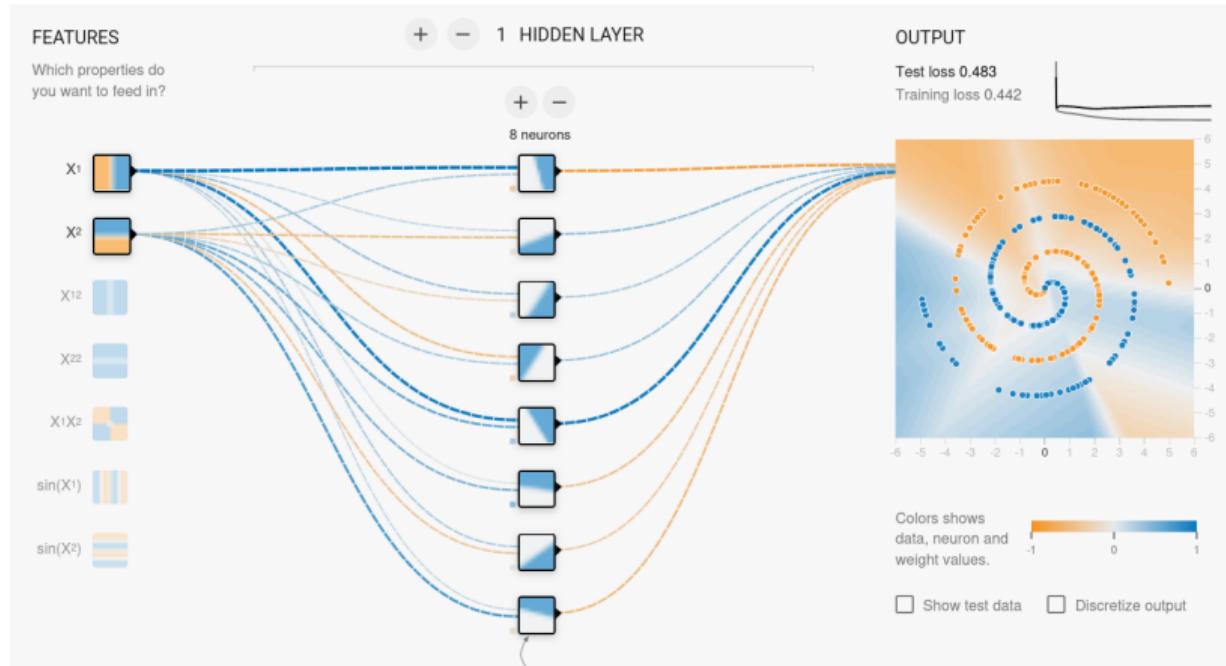
## Features to your features

If your problem is really hard, you might need *basic features* to extract *more complex features*

Check out the incredible [intro to Deep Learning video](#) from 3blue1brown:

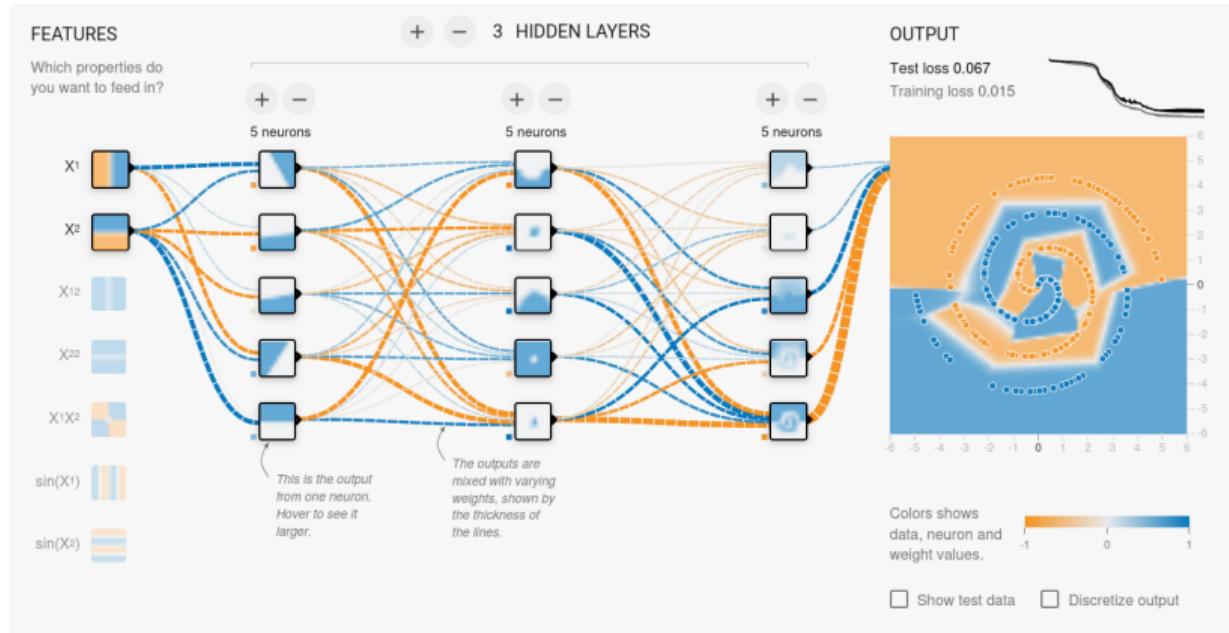


# Visualizing the benefits of *multiple* layers with Playground



We would need a lot more neurons to cut this shape!

# Visualizing the benefits of *multiple* layers with Playground



More layers allow for much more cuts, even with the same total number of neurons.

See [this video from Welsch Labs](#) for more details on this insight.

Now we can talk about a **Deep** Neural Network!

# Generic representation of Deep Neural Networks

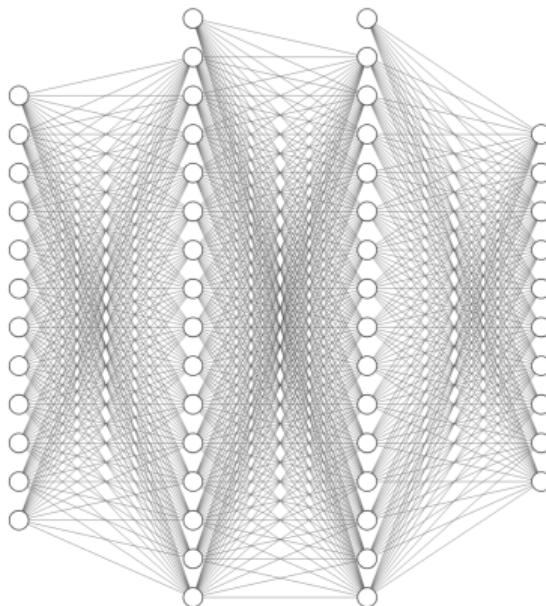


Figure: Feedforward DNNs are often represented only with input, hidden and output layers, the sum and activation function steps are implicit.

## How do I train this thing?

Before asking about training... What do I even *want*?

# How do I train this thing?

Before asking about training... What do I even *want*?

We are looking for the weights  $\theta = \{W_1, \mathbf{b}_1, W_2, \mathbf{b}_2, \dots\}$  that best approximate some function  $g$ .

For now, let's assume Supervised Learning and classification:  
we have examples of data

$$g(\mathbf{x}_i) = \mathbf{y}_i \text{ for } i \in [1, N]$$

$$g \left( \begin{array}{c} \text{orange cat sitting on grass} \end{array} \right) = \text{"cat"}$$

$$\rightarrow \text{Dataset } \mathcal{D} = \{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$$

## What do we want to achieve?

We want  $f_{\theta}(\mathbf{x}_i)$  as close as possible to  $g(\mathbf{x}_i) = \mathbf{y}_i$ .

How do I measure this? If  $y_i \in \mathbb{R}$ :

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^N |y_i - f_{\theta}(\mathbf{x}_i)|$$

For  $\mathbf{y}_i \in \mathcal{Y} \subseteq \mathbb{R}^m$ , use

$$\|\mathbf{y}_i - f_{\theta}(\mathbf{x}_i)\|_1 \doteq \sum_{j=1}^m |(\mathbf{y}_i)_j - f_{\theta}(\mathbf{x}_i)_j|$$

## What do we want to achieve?

We want  $f_{\theta}(\mathbf{x}_i)$  as close as possible to  $g(\mathbf{x}_i) = \mathbf{y}_i$ .

How do I measure this? If  $y_i \in \mathbb{R}$ :

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^N |y_i - f_{\theta}(\mathbf{x}_i)|$$

For  $\mathbf{y}_i \in \mathcal{Y} \subseteq \mathbb{R}^m$ , use

$$\|\mathbf{y}_i - f_{\theta}(\mathbf{x}_i)\|_1 \doteq \sum_{j=1}^m |(\mathbf{y}_i)_j - f_{\theta}(\mathbf{x}_i)_j|$$

## What do we want to achieve?

We want  $f_{\theta}(\mathbf{x}_i)$  as close as possible to  $g(\mathbf{x}_i) = \mathbf{y}_i$ .

How do I measure this? If  $y_i \in \mathbb{R}$ :

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^N |y_i - f_{\theta}(\mathbf{x}_i)|$$

For  $\mathbf{y}_i \in \mathcal{Y} \subseteq \mathbb{R}^m$ , use

$$\|\mathbf{y}_i - f_{\theta}(\mathbf{x}_i)\|_1 \doteq \sum_{j=1}^m |(\mathbf{y}_i)_j - f_{\theta}(\mathbf{x}_i)_j|$$

## What do we want to achieve?

We want  $f_{\theta}(\mathbf{x}_i)$  as close as possible to  $g(\mathbf{x}_i) = \mathbf{y}_i$ .

How do I measure this? If  $y_i \in \mathbb{R}$ :

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^N |y_i - f_{\theta}(\mathbf{x}_i)|$$

For  $\mathbf{y}_i \in \mathcal{Y} \subseteq \mathbb{R}^m$ , use

$$\|\mathbf{y}_i - f_{\theta}(\mathbf{x}_i)\|_1 \doteq \sum_{j=1}^m |(\mathbf{y}_i)_j - f_{\theta}(\mathbf{x}_i)_j|$$

## Minimizing the Loss

Lots of possible choices for a “distance function” between  $y_i$  and  $f_{\theta}(\mathbf{x}_i)$   
→ lots of possible choices for a **loss**  $\mathcal{L}$

## Minimizing the Loss

Lots of possible choices for a “distance function” between  $y_i$  and  $f_{\theta}(\mathbf{x}_i)$   
→ lots of possible choices for a **loss**  $\mathcal{L}$

<i>Loss name</i>	<i>Formula</i>
Mean Absolute Error (aka L1 Loss)	$\mathcal{L}_{\text{MAE}} = 1/N \sum_i  y_i - f_{\theta}(\mathbf{x}_i) $
Mean Squared Error (aka L2 Loss)	$\mathcal{L}_{\text{MSE}} = 1/N \sum_i (y_i - f_{\theta}(\mathbf{x}_i))^2$
Cross Entropy Loss (classification)	$\mathcal{L}_{\text{CE}} = - \sum_i \sum_{\text{class } c} y_{i,c} \log f_{\theta}(\mathbf{x}_i)_c$
...	...

## Minimizing the Loss

Lots of possible choices for a “distance function” between  $y_i$  and  $f_{\theta}(\mathbf{x}_i)$   
→ lots of possible choices for a **loss**  $\mathcal{L}$

<i>Loss name</i>	<i>Formula</i>
Mean Absolute Error (aka L1 Loss)	$\mathcal{L}_{\text{MAE}} = 1/N \sum_i  \mathbf{y}_i - f_{\theta}(\mathbf{x}_i) $
Mean Squared Error (aka L2 Loss)	$\mathcal{L}_{\text{MSE}} = 1/N \sum_i (\mathbf{y}_i - f_{\theta}(\mathbf{x}_i))^2$
Cross Entropy Loss (classification)	$\mathcal{L}_{\text{CE}} = - \sum_i \sum_{\text{class } c} y_{i,c} \log f_{\theta}(\mathbf{x}_i)_c$
...	...

## Minimizing the Loss

Lots of possible choices for a “distance function” between  $y_i$  and  $f_{\theta}(\mathbf{x}_i)$   
→ lots of possible choices for a **loss**  $\mathcal{L}$

<i>Loss name</i>	<i>Formula</i>
Mean Absolute Error (aka L1 Loss)	$\mathcal{L}_{\text{MAE}} = 1/N \sum_i  \mathbf{y}_i - f_{\theta}(\mathbf{x}_i) $
Mean Squared Error (aka L2 Loss)	$\mathcal{L}_{\text{MSE}} = 1/N \sum_i (\mathbf{y}_i - f_{\theta}(\mathbf{x}_i))^2$
Cross Entropy Loss (classification)	$\mathcal{L}_{\text{CE}} = - \sum_i \sum_{\text{class } c} y_{i,c} \log f_{\theta}(\mathbf{x}_i)_c$
...	...

## Minimizing the Loss

Lots of possible choices for a “distance function” between  $y_i$  and  $f_{\theta}(\mathbf{x}_i)$   
→ lots of possible choices for a **loss**  $\mathcal{L}$

<i>Loss name</i>	<i>Formula</i>
Mean Absolute Error (aka L1 Loss)	$\mathcal{L}_{\text{MAE}} = 1/N \sum_i  \mathbf{y}_i - f_{\theta}(\mathbf{x}_i) $
Mean Squared Error (aka L2 Loss)	$\mathcal{L}_{\text{MSE}} = 1/N \sum_i (\mathbf{y}_i - f_{\theta}(\mathbf{x}_i))^2$
Cross Entropy Loss (classification)	$\mathcal{L}_{\text{CE}} = - \sum_i \sum_{\text{class } c} y_{i,c} \log f_{\theta}(\mathbf{x}_i)_c$
...	...

## Minimizing the Loss

Lots of possible choices for a “distance function” between  $y_i$  and  $f_{\theta}(\mathbf{x}_i)$   
→ lots of possible choices for a **loss**  $\mathcal{L}$

<i>Loss name</i>	<i>Formula</i>
Mean Absolute Error (aka L1 Loss)	$\mathcal{L}_{\text{MAE}} = 1/N \sum_i  \mathbf{y}_i - f_{\theta}(\mathbf{x}_i) $
Mean Squared Error (aka L2 Loss)	$\mathcal{L}_{\text{MSE}} = 1/N \sum_i (\mathbf{y}_i - f_{\theta}(\mathbf{x}_i))^2$
Cross Entropy Loss (classification)	$\mathcal{L}_{\text{CE}} = - \sum_i \sum_{\text{class } c} y_{i,c} \log f_{\theta}(\mathbf{x}_i)_c$
...	...

In general we choose  $\mathcal{L}(f_{\theta}, \mathcal{D})$  and look for  $\theta$  to minimize

$$\min_{\theta} \mathcal{L}(f_{\theta}, \mathcal{D})$$

**I know what I want! ...How do I train this thing?**

## I know what I want! ...How do I train this thing?

Options:

- pick each weight by hand (thousands, millions, billions)

## I know what I want! ...How do I train this thing?

Options:

- pick each weight by hand (thousands, millions, billions)
- sample a bunch of  $\theta$  and pick the best?

## I know what I want! ...How do I train this thing?

Options:

- pick each weight by hand (thousands, millions, billions)
- sample a bunch of  $\theta$  and pick the best?
- keep taking weights around the best-performing ones (genetic algorithms)

## I know what I want! ...How do I train this thing?

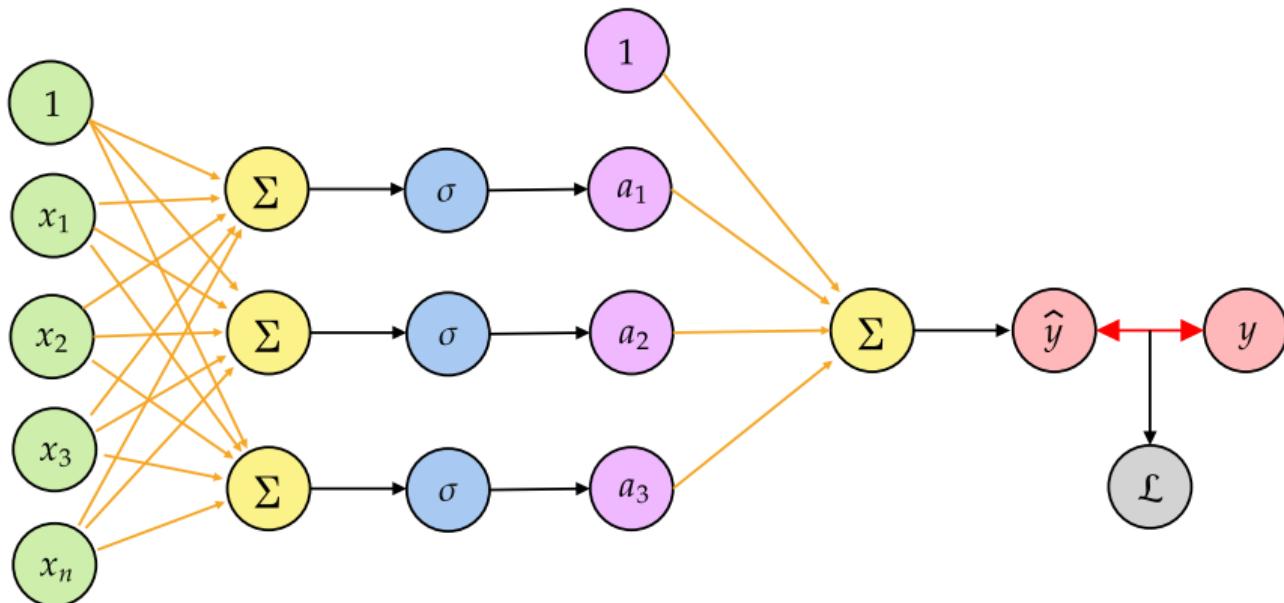
Options:

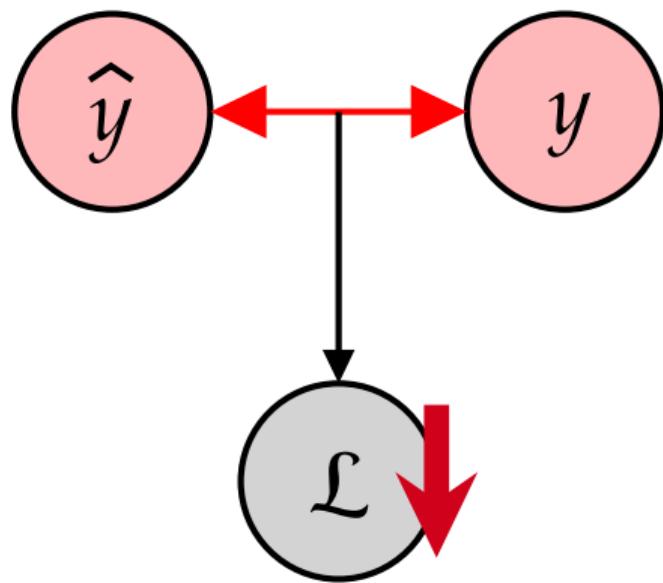
- pick each weight by hand (thousands, millions, billions)
- sample a bunch of  $\theta$  and pick the best?
- keep taking weights around the best-performing ones (genetic algorithms)
- **Gradient Descent!**

## Gradient Descent

General idea:

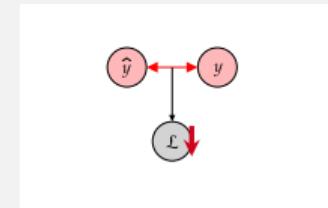
How should I change  $w_{ij}^{(l)}$  so that  $\mathcal{L}$  gets a bit smaller?

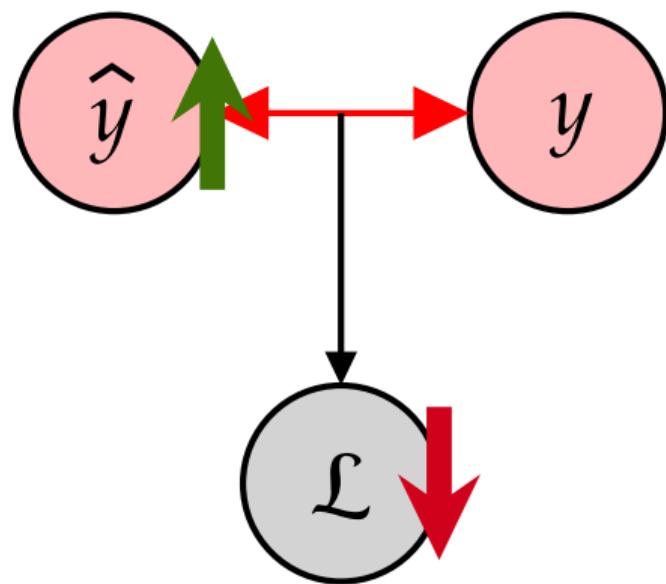




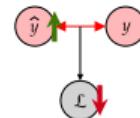
## Notes for this slide

we want to decrease the loss  $\mathcal{L}$ ...

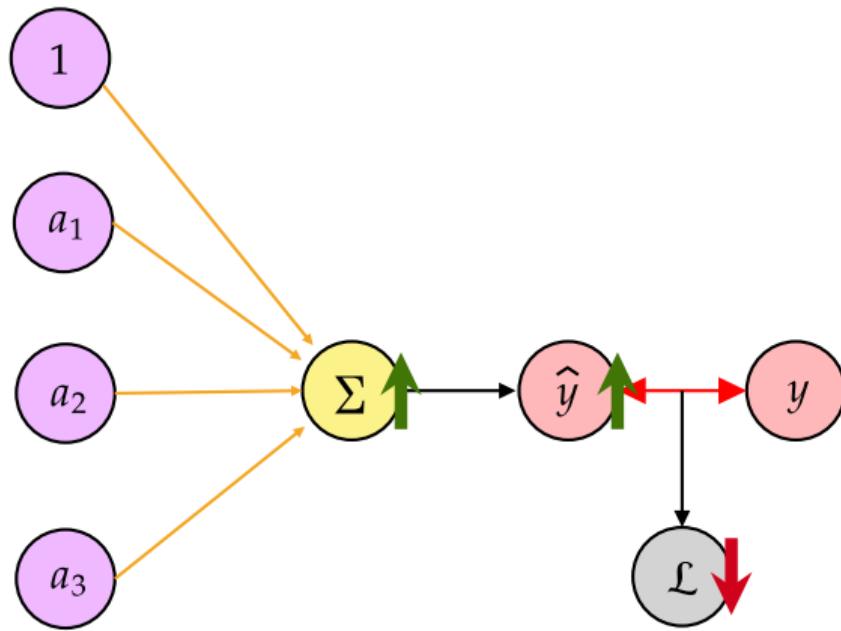




## Notes for this slide

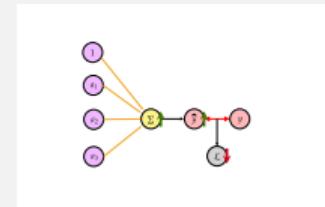


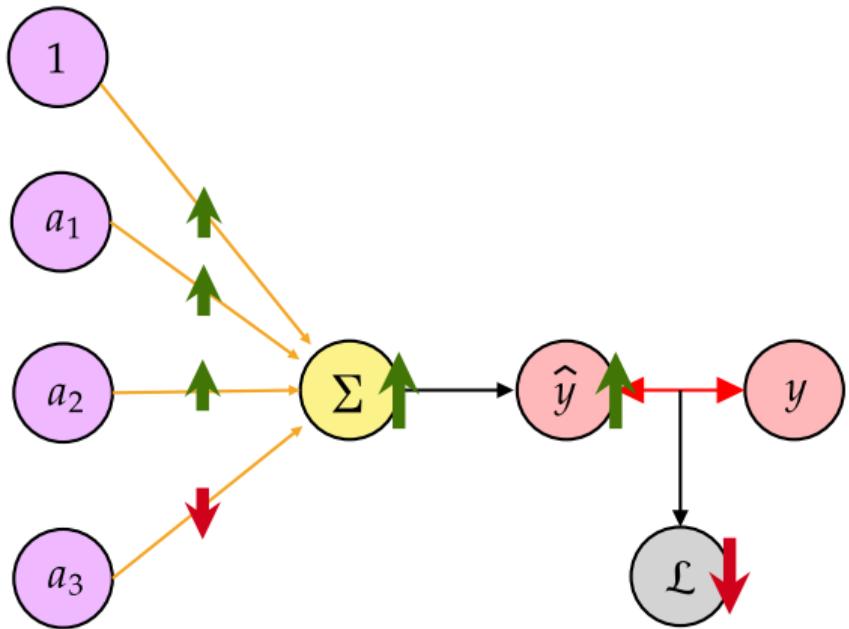
imagine that  $\hat{y}$  was too low (compared to  $y$ ): we want to raise it



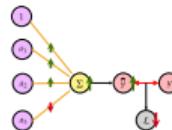
## Notes for this slide

$\hat{y} = \sum w_i^{(2)} a_i$  so the sum should go up





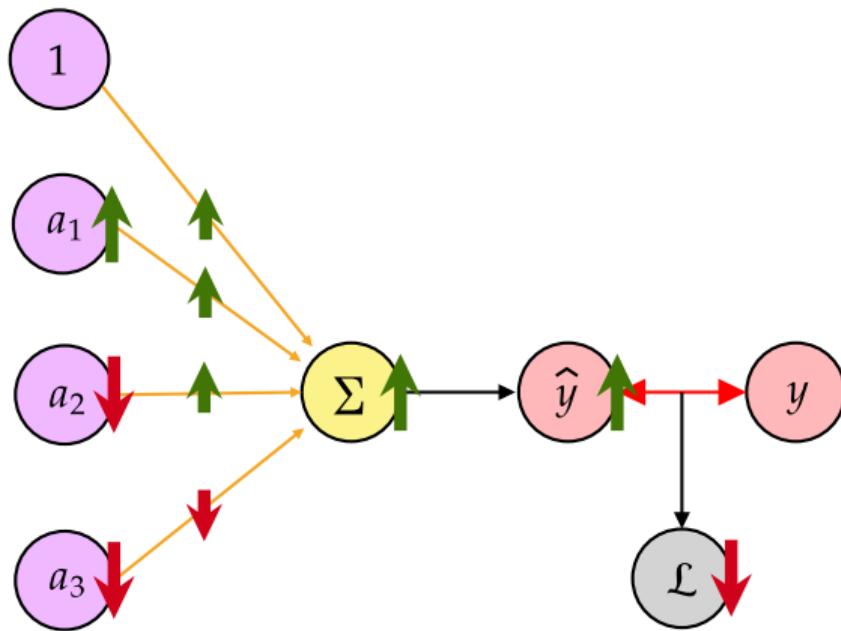
## Notes for this slide



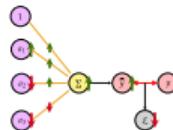
to get the sum up, the bias weight  $w_0^{(2)}$  should go up.

For the other weights, it depends on the value of  $a_i$ :

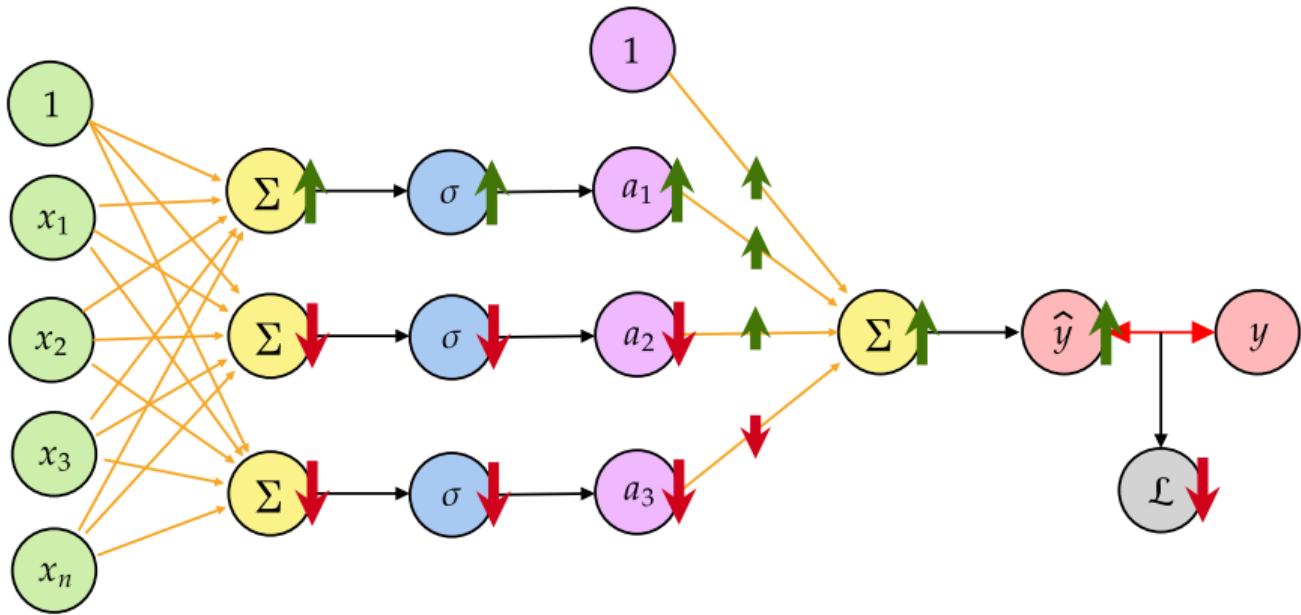
- if both  $a_i$  and  $w_i^{(2)} > 0$ :  $w_i^{(2)}$  should go up
- if both  $a_i$  and  $w_i^{(2)} < 0$ :  $w_i^{(2)}$  should go down (more negative)
- if  $a_i > 0$  and  $w_i^{(2)} < 0$ :  $w_i^{(2)}$  should go up (less negative)
- if  $a_i < 0$  and  $w_i^{(2)} > 0$ :  $w_i^{(2)}$  should go down



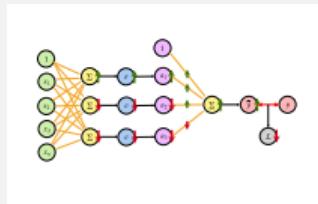
## Notes for this slide



but we don't need to stop there! we also control  $a_i$ , so we can do the same thing



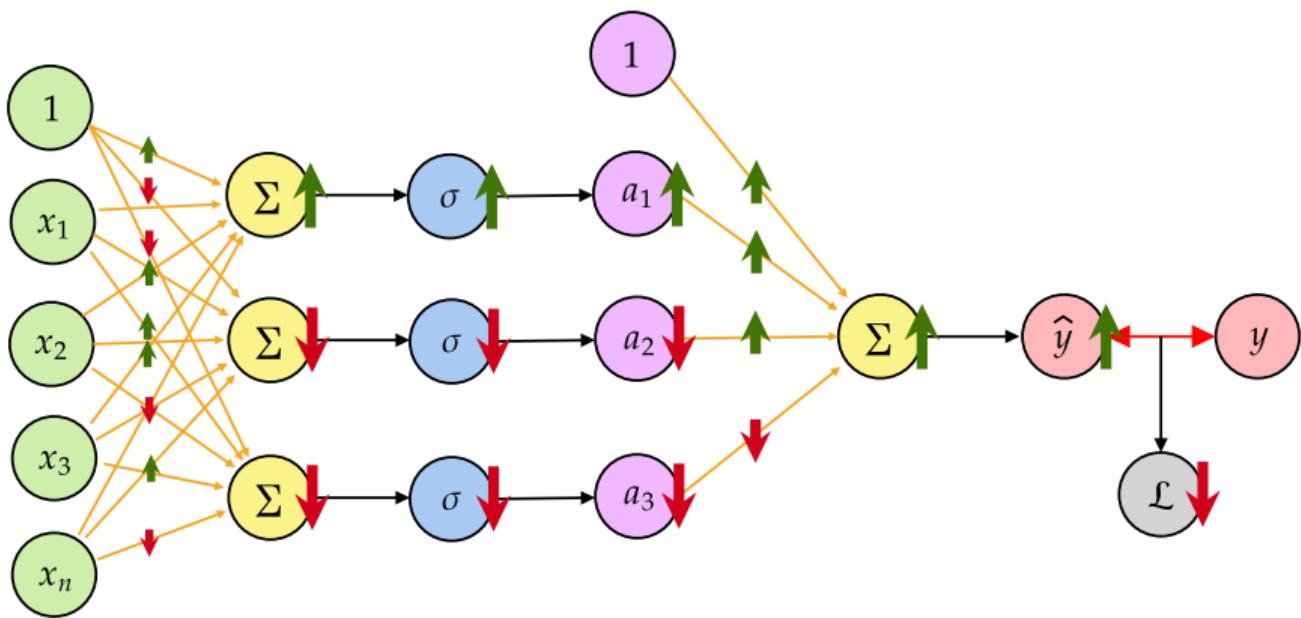
## Notes for this slide



$\sigma$  is generally a monotonous increasing function, so if  $a_i$  should go one way, the sum should go the same way.

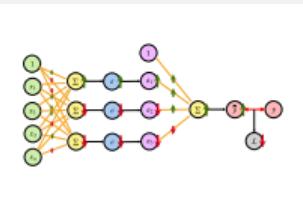
If  $\sigma$  is strictly positive (sigmoid, ReLU), then an order to raise the sum raises all weights, but proportionally to the corresponding activation:

- stronger activation = stronger associated weight increase
- activation close to 0 = nearly no increase in weight.



## Notes for this slide

where  $w_{ij}^{(1)}$  should go depends on the sign of  $x_i$ , same as for  $a_i$



## Gradient Descent

General idea:

How should I change  $w_{ij}^{(l)}$  so that  $\mathcal{L}$  gets a bit smaller?

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} + \eta \uparrow$$

## Gradient Descent

General idea:

How should I change  $w_{ij}^{(l)}$  so that  $\mathcal{L}$  gets a bit smaller?

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} + \eta \uparrow$$

We achieve this mathematically by computing the *gradient*  $\nabla_{\theta} \mathcal{L} = \left[ \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} \right]_{\forall ijl}$ .

## Notes for this slide

Why do we need  $\eta$ ?

Consider the case of  $\hat{y} < y$ : we wanted  $\mathcal{L}$  to go down, so we wanted  $\hat{y}$  to go up to catch up to  $y$ . Too big of a swing up might overshoot past  $y$ !

### Gradient Descent

General idea:

How should I change  $w_j^{(l)}$  so that  $\mathcal{L}$  gets a bit smaller?

$$w_j^{(l)} \leftarrow w_j^{(l)} + \eta \uparrow$$

We achieve this mathematically by computing the gradient:  $\nabla_{\theta} \mathcal{L} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \theta_i} \end{bmatrix}_{i \in \theta}$

## Gradient Descent

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} + \eta \uparrow$$

We achieve this mathematically by computing the *gradient*  $\nabla_{\theta} \mathcal{L} = \left[ \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} \right]_{\forall ijl}$ .

Reminder 1D:

$$\frac{d\mathcal{L}}{dw} \doteq \lim_{h \rightarrow 0} \frac{\mathcal{L}(w + h) - \mathcal{L}(w)}{h}$$

## Gradient Descent

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} + \eta \uparrow$$

We achieve this mathematically by computing the *gradient*  $\nabla_{\theta} \mathcal{L} = \left[ \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} \right]_{\forall ijl}$ .

Reminder 1D:

$$\frac{d\mathcal{L}}{dw} \doteq \lim_{h \rightarrow 0} \frac{\mathcal{L}(w + h) - \mathcal{L}(w)}{h}$$

- if  $\mathcal{L}(w + h) > \mathcal{L}(w)$ ,  $\frac{d\mathcal{L}}{dw}$  is positive (change of  $w$  where  $\mathcal{L}$  is higher)
- otherwise  $\frac{d\mathcal{L}}{dw}$  is negative (change of  $w$  where  $\mathcal{L}$  is higher)

## Gradient Descent

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} + \eta \uparrow$$

We achieve this mathematically by computing the *gradient*  $\nabla_{\theta} \mathcal{L} = \left[ \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} \right]_{\forall ijl}$ .

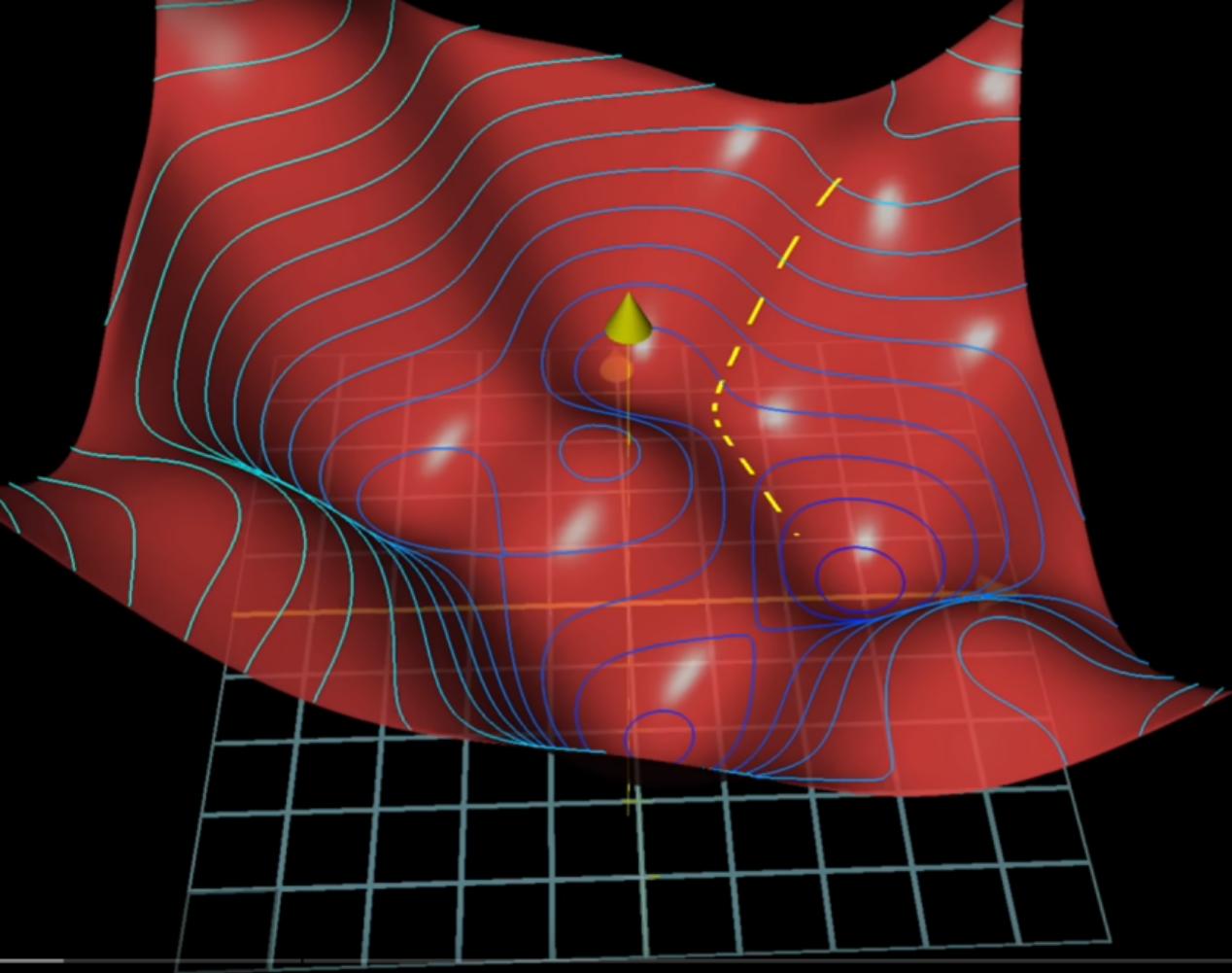
- in general, the gradient  $\nabla_{\theta} \mathcal{L}$  is the direction of **steepest ascent**
  - simply follow  $-\nabla_{\theta} \mathcal{L}$  to get **steepest descent**: best local change to each  $w_{ij}^{(l)}$

## Gradient Descent

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} + \eta \uparrow$$

We achieve this mathematically by computing the *gradient*  $\nabla_{\theta} \mathcal{L} = \left[ \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}} \right]_{\forall ijl}$ .

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}}$$



## Notes for this slide

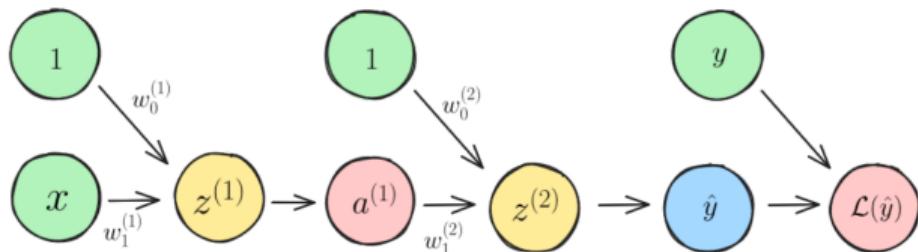
Gradient Descent is often visualized with a hilly landscape, representing the loss  $\mathcal{L}$  as a function of the parameters (here, two, e.g.  $w_1$  and  $w_2$ ). As the parameters change, the value of the loss changes and our objective is to find a minimum of the function (global if possible, more likely local).

Gradient descent computes the direction of steepest descent and simply follows it with consecutive little steps, eventually reaching a local minimum (yellow dashed path).

Picture from [the 3blue1brow lecture](#).

# Backpropagation

The algorithm to compute the partial derivatives  $\frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}}$  by going back layer by layer is called **Backpropagation**.



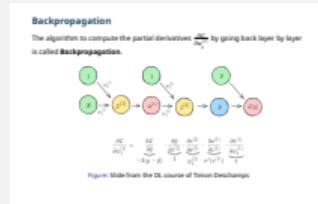
$$\frac{\partial \mathcal{L}}{\partial w_1^{(1)}} = \underbrace{\frac{\partial \mathcal{L}}{\partial \hat{y}}}_{-2(y - \hat{y})} \cdot \underbrace{\frac{\partial \hat{y}}{\partial z^{(2)}}}_1 \cdot \underbrace{\frac{\partial z^{(2)}}{\partial a^{(1)}}}_{w_1^{(2)}} \cdot \underbrace{\frac{\partial a^{(1)}}{\partial z^{(1)}}}_{\sigma'(z^{(1)})} \cdot \underbrace{\frac{\partial z^{(1)}}{\partial w_1^{(1)}}}_x$$

Figure: Slide from the DL course of Timon Deschamps

## Notes for this slide

Read the figure from right (Loss) to left ( $w_1^{(1)}$ ), but the formula from left to right. Each  $\frac{\partial v}{\partial u}$  indicates “how does a small change in  $u$  affect  $v$ ?”, we use this concept to go up the network (towards the left/input layer).

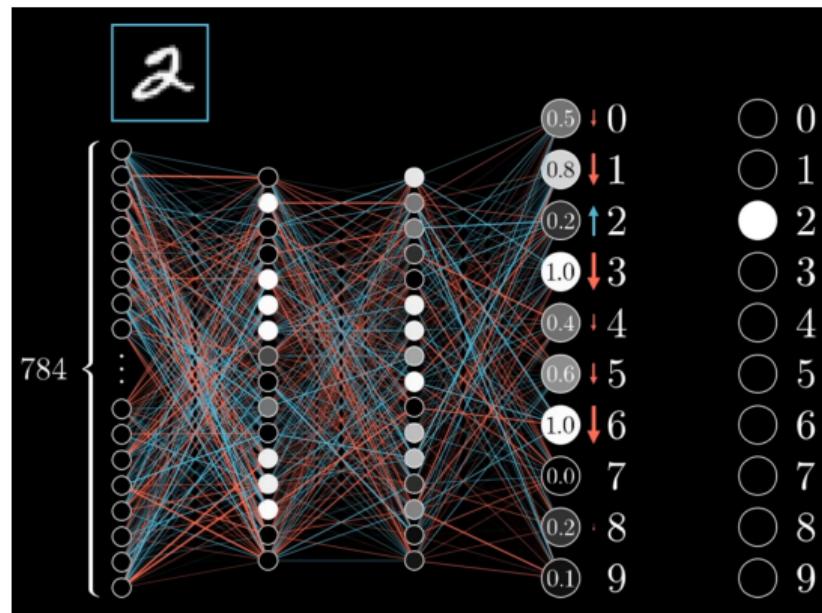
This layer-by-layer approach allows us to efficiently compute the partial derivatives for all weights. With a small learning rate, moving all parameters in these directions should slightly decrease the loss.



# Backpropagation, animated

For an animated version of the arrows visuals

Check out the [lecture on Backpropagation](#) from 3blue1brown.



## Mini-batch Gradient Descent

We are looking for

$$\arg \min_{\theta} \mathcal{L}(f_{\theta}, \mathcal{D})$$

Computing this over the whole dataset: **impractical** to **impossible!**

## Notes for this slide

Why is it impractical/impossible? → If the dataset has thousands, millions or billions of points, computing  $\mathcal{L}$  is longer and longer and eventually impossible.

What are we doing? → Imagine the negative gradient as an arrow pointing perfectly down in the loss landscape. The approximate gradient we get with stochastic gradient descent is an arrow generally in the same direction, but with some noise (*stochastic*) due to only sampling a few datapoints. Imagine a drunk old man going downhill! It's not perfect, but much faster to compute.

We need to take even smaller steps. Techniques like momentum (summing up the gradients) will very much help make sure we don't wander too much.



## Mini-batch Gradient Descent

We are looking for

$$\arg \min_{\theta} \mathcal{L}(f_{\theta}, \mathcal{D})$$

Computing this over the whole dataset: **impractical** to **impossible!** → compute only  $\mathcal{L}$  over a small *batch* of data

## Notes for this slide

Why is it impractical/impossible? → If the dataset has thousands, millions or billions of points, computing  $\mathcal{L}$  is longer and longer and eventually impossible.

What are we doing? → Imagine the negative gradient as an arrow pointing perfectly down in the loss landscape. The approximate gradient we get with stochastic gradient descent is an arrow generally in the same direction, but with some noise (*stochastic*) due to only sampling a few datapoints. Imagine a drunk old man going downhill! It's not perfect, but much faster to compute.

We need to take even smaller steps. Techniques like momentum (summing up the gradients) will very much help make sure we don't wander too much.



## Mini-batch Gradient Descent

We are looking for

$$\arg \min_{\theta} \mathcal{L}(f_{\theta}, \mathcal{D})$$

Computing this over the whole dataset: **impractical** to **impossible!** → compute only  $\mathcal{L}$  over a small *batch* of data

- we get an *approximate* gradient instead of the full one
- the bigger the batch *size*, the closer to the real gradient
- this is called **Stochastic Gradient Descent**.

## Notes for this slide

Why is it impractical/impossible? → If the dataset has thousands, millions or billions of points, computing  $\mathcal{L}$  is longer and longer and eventually impossible.

What are we doing? → Imagine the negative gradient as an arrow pointing perfectly down in the loss landscape. The approximate gradient we get with stochastic gradient descent is an arrow generally in the same direction, but with some noise (*stochastic*) due to only sampling a few datapoints. Imagine a drunk old man going downhill! It's not perfect, but much faster to compute.

We need to take even smaller steps. Techniques like momentum (summing up the gradients) will very much help make sure we don't wander too much.



# Training of a Deep Neural Network (simplified)

1. Sample data  $\mathbf{x}_i, \mathbf{y}_i$  from dataset  $\mathcal{D}$
2. Compute “forward” pass  $f_{\theta}(\mathbf{x}_i) = \hat{\mathbf{y}}_i$
3. Compute loss function  $\mathcal{L}(\hat{\mathbf{y}}_i, \mathbf{y}_i)$
4. Compute gradients with Backprop

$$\partial \mathcal{L} / \partial w_{ij}^{(l)}$$

5. Perform Gradient Descent,

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \partial \mathcal{L} / \partial w_{ij}^{(l)}$$

6. Repeat (go to 1.)

```
1  for x, y in dataset:  
2      y_hat = net(x)  
3      loss = mse_loss(y_hat, y)  
4      loss.backward()  
5      optimizer.step()
```

Figure: Core training loop in PyTorch

## Why did it work so well?

Deep Neural Nets are everywhere now. Why? Lots of reasons...

- Conceptual: learning multiple *layers* of abstraction
- Practical: access to ridiculous amounts of *data* with the internet
- Computational: advances in chips, compute efficiency & GPUs for matrix multiplication
- Coincidental: backprop scales extremely well in these conditions
- Feedback loop: more research, more development & tools, more research...

# Big pile of hyperparameters

With just our basic deep neural net, tons of choices to make:

Number of hidden layers       $L$

Size of hidden layers       $k$

Activation function       $\sigma$

Loss function       $\mathcal{L}$

Learning rate       $\eta$

## Big pile of hyperparameters

With just our basic deep neural net, tons of choices to make:

Number of hidden layers       $L$

Size of hidden layers       $k$

Activation function       $\sigma$

Loss function       $\mathcal{L}$

Learning rate       $\eta$

LOTS more to come      optim, layers, batch size, norms,...

## Big pile of hyperparameters

With just our basic deep neural net, tons of choices to make:

Number of hidden layers       $L$

Size of hidden layers       $k$

Activation function       $\sigma$

Loss function       $\mathcal{L}$

Learning rate       $\eta$

LOTS more to come      optim, layers, batch size, norms,...

$\theta$  are the *parameters* of the model, above are “*hyper*”*parameters*

## Big pile of hyperparameters

With just our basic deep neural net, tons of choices to make:

Number of hidden layers       $L$

Size of hidden layers       $k$

Activation function       $\sigma$

Loss function       $\mathcal{L}$

Learning rate       $\eta$

LOTS more to come      optim, layers, batch size, norms,...

$\theta$  are the *parameters* of the model, above are “*hyper*”*parameters*

→ often Deep Learning feels more like cooking than science - lots of trial & error!

→ follow whatever people have done in similar contexts

**Let's implement this!**