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

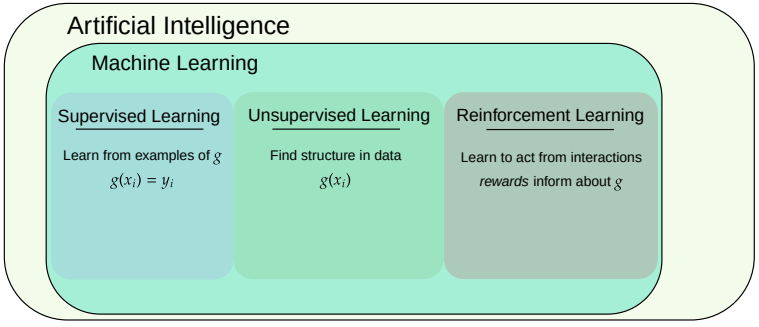
- Deep Learning book by Goodfellow, Bengio and Courville
- Denis Coquenet, Université Rennes
- Mathieu Lefort, Université Rennes and Lyon
- Michael Nielsen's book on Neural Networks and Deep Learning
- 3blue1brown (Grant Sanderson) video series on Deep Learning
- Welsch Labs videos on Deep Learning

Objective

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

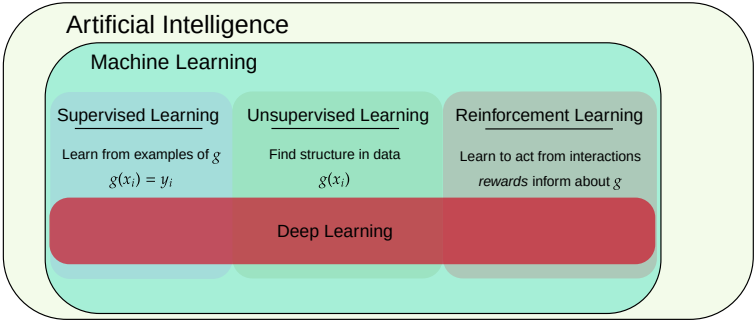
Objective

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



Objective

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



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

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

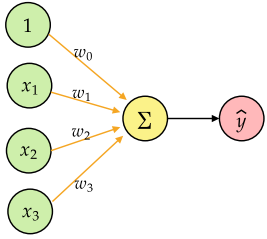
Text	**Image**	**Audio** & **Signal**	**Decision**
document analysis	image recognition	audio recognition	chess
sentiment analysis	segmentation	forecasting	starcraft
translation	style transfer	translation	minecraft
text generation	image generation	audio generation	robotics

From Linear Regression to Deep Learning

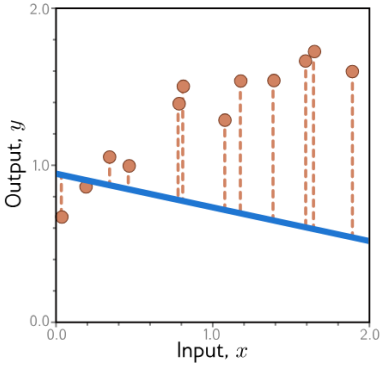
We want a strong approximator $f_{\boldsymbol{\theta}} \approx g : \mathcal{X} \to \mathcal{Y}$.

Start from linear regression/classification:

$$
\begin{aligned}
f_{\boldsymbol{\theta}}(\boldsymbol{x}) &= w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_d x_d \\
&= \boldsymbol{\theta}^\top \boldsymbol{x} \approx g(\boldsymbol{x})
\end{aligned}
$$



Linear Regression intuition



Model: $y = \phi_0 + \phi_1 x$

Intercept, ϕ_0

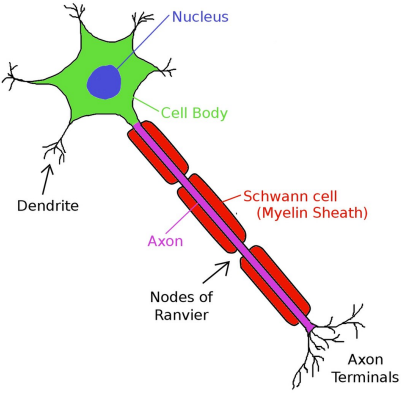
Slope, ϕ_1

Loss: 6.61

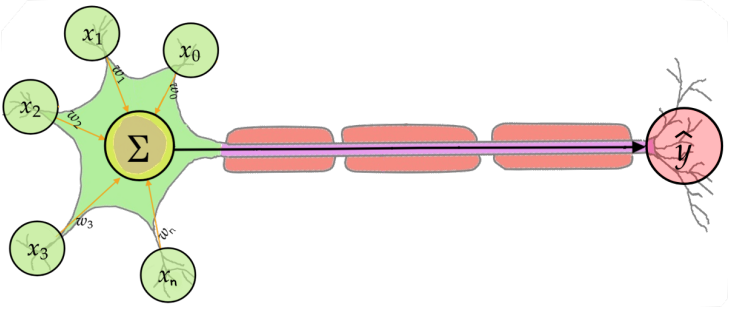
Find the parameters that make the best line through the data

Try it yourself! (figure 2.2, press the right arrow once)

Concept of Perceptron: inspiration from neurons



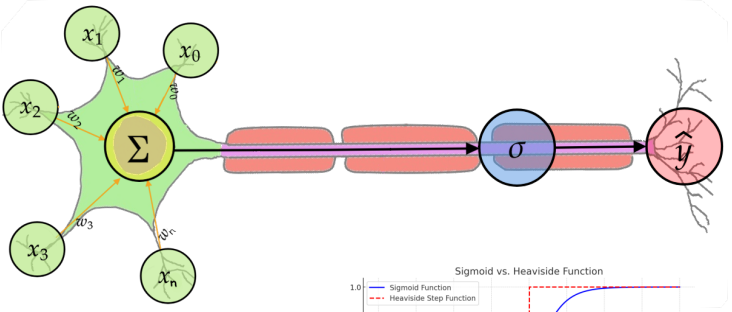
Concept of Perceptron: inspiration from neurons



this looks like a linear regressor!

...but a neuron output is `on/off`

Concept of Perceptron: inspiration from neurons

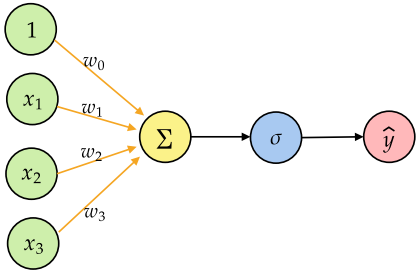


Concept of Perceptron

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

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

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



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

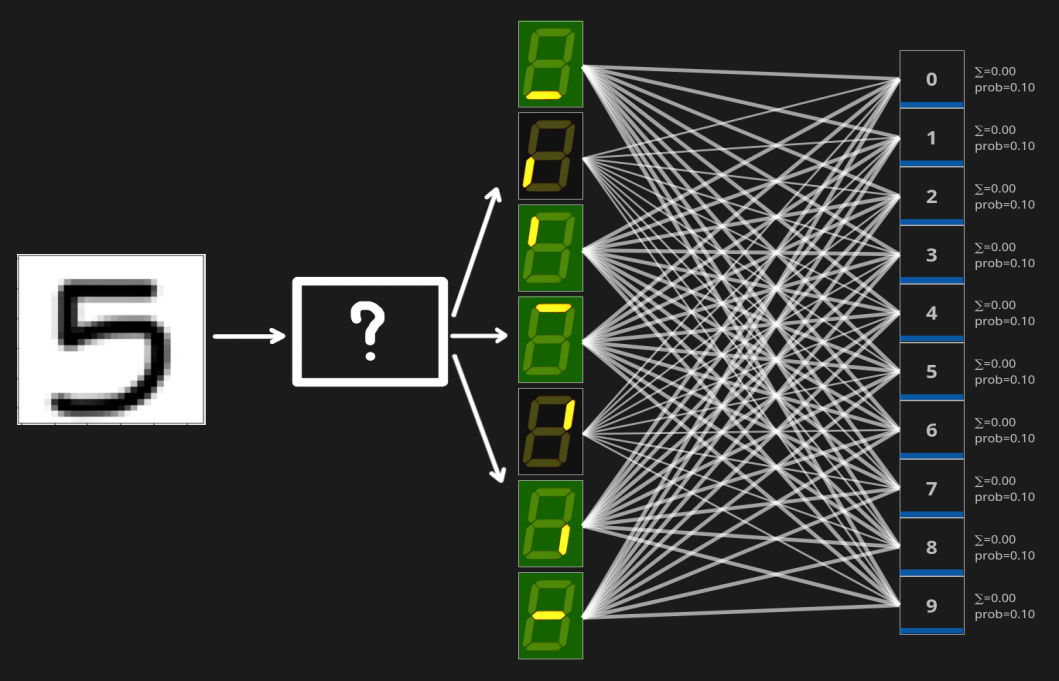
Extracting Features

"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!"

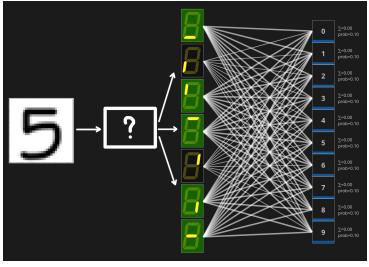
Feature: something that describes the input $\boldsymbol{x}$ in a "useful" way, $\phi(\boldsymbol{x})$.
$\rightarrow$ even better when we can find multiple $\phi_i(\boldsymbol{x})$ coherent with each other!


?
0
∑=0.00
prob=0.10
1
∑=0.00
prob=0.10
2
∑=0.00
prob=0.10
3
∑=0.00
prob=0.10
4
∑=0.00
prob=0.10
5
∑=0.00
prob=0.10
6
∑=0.00
prob=0.10
7
∑=0.00
prob=0.10
8
∑=0.00
prob=0.10
9
∑=0.00
prob=0.10

Extracting Features

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

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

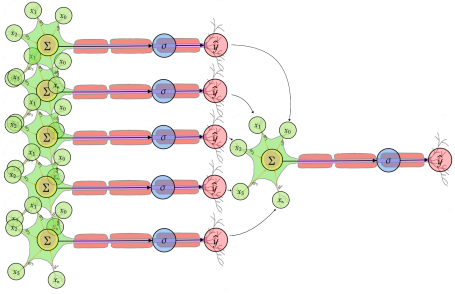


[?] = decades of feature extraction research

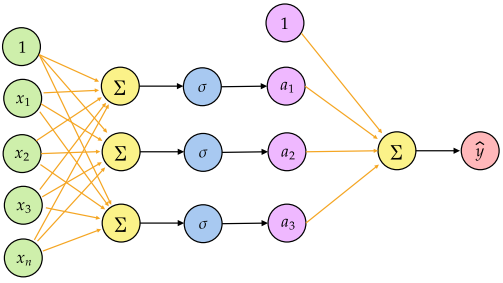
The first hidden layer

Key intuition: **Chain** perceptrons/logistic regressions:

$\rightarrow$ **learn** the features as well!



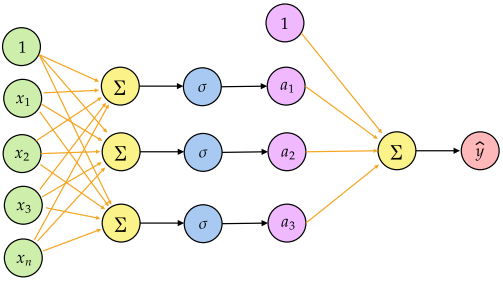
The first hidden layer



$\rightarrow$ Now we can talk about a “Neural **Network**”!

aka “Multilayer Perceptron”, MLP

The first hidden layer

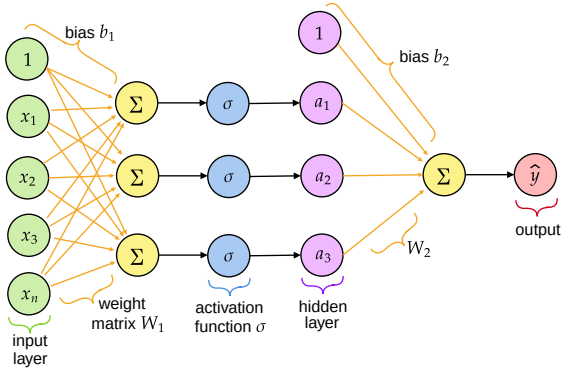


$\rightarrow$ Now we can talk about a "Neural **Network**"!

aka "Multilayer Perceptron", MLP

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

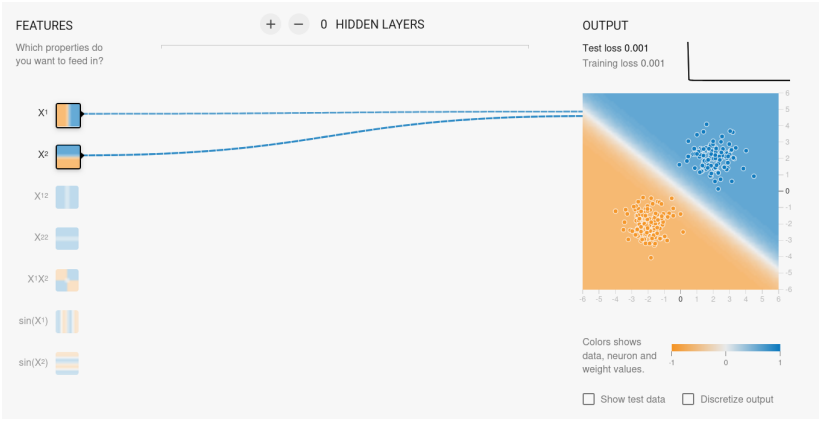
Definitions in the neural network



$$f_{\boldsymbol{\theta}}(\boldsymbol{x}) = W_2\sigma(W_1\boldsymbol{x} + \boldsymbol{b_1}) + \boldsymbol{b_2}$$

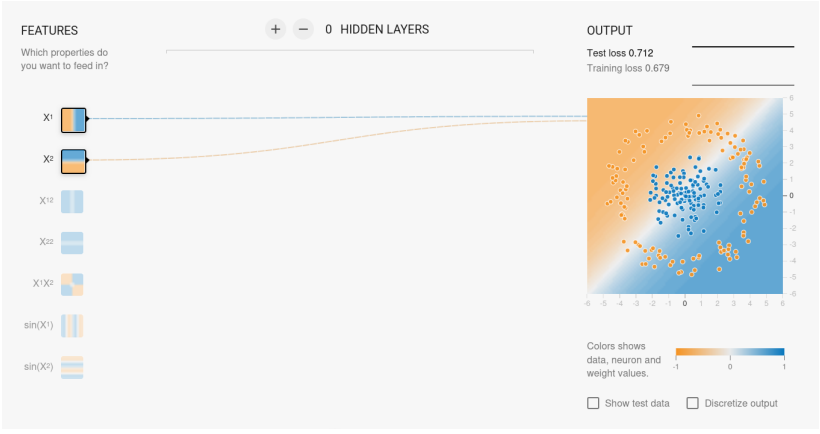
Let's see how this behaves!

Visualizing the benefits of a hidden layer with Playground



Linear regression/classification: easy!

Visualizing the benefits of a hidden layer with Playground

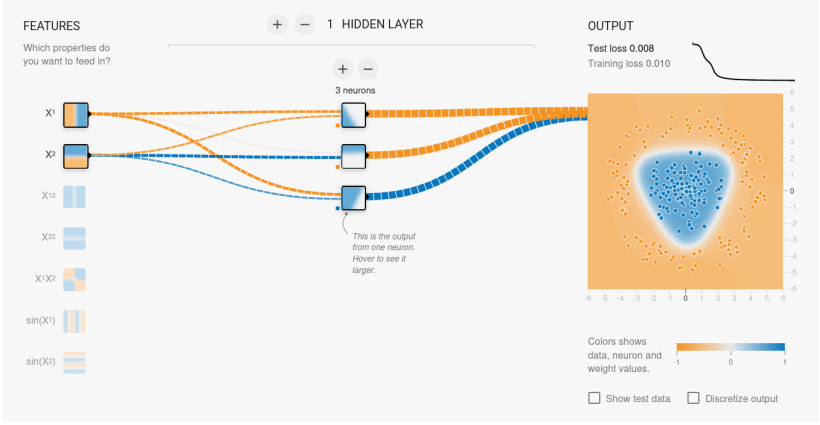


With a linear model, we can never model non-linear relationships...

Visualizing the benefits of a hidden layer with Playground

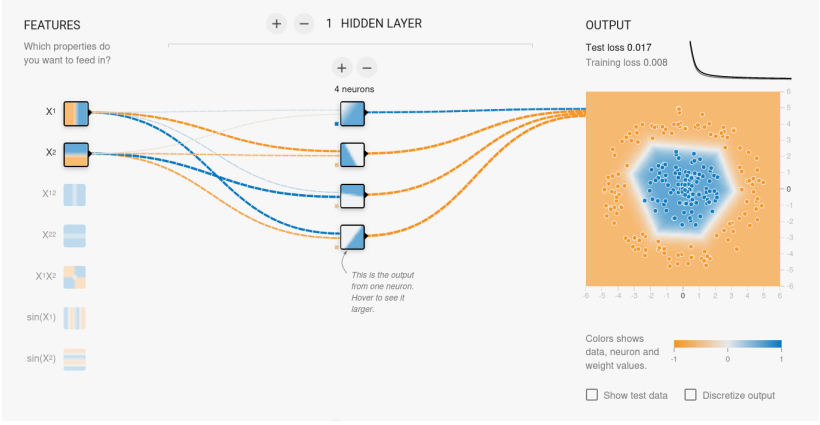
It would be so nice if we just had the right features!

Visualizing the benefits of a hidden layer with Playground



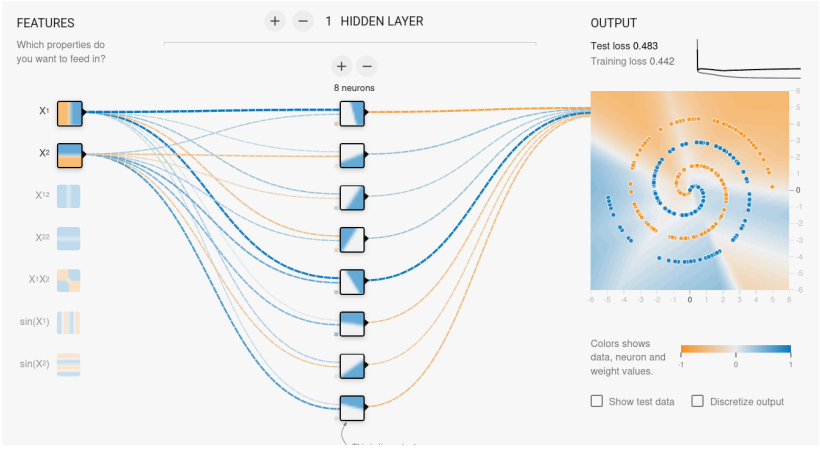
Simply add a hidden layer, learn useful features!

Visualizing the benefits of a hidden layer with Playground



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

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* σ is not a polynomial

THEN

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

  $\rightarrow$ 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** $\boldsymbol{\theta}^*$ with any method

  $\rightarrow$ certainly not with gradient descent
- a bunch of other universal approximators exist (polynomials, Fourier)

but still, **it's something**

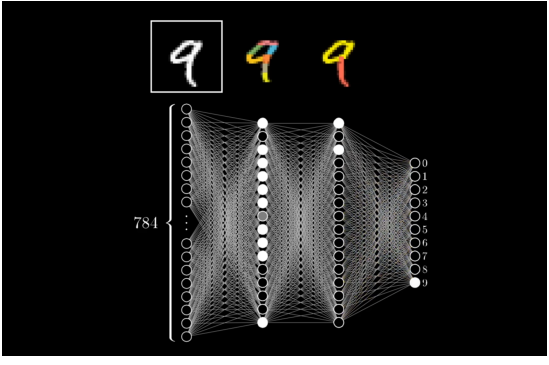
Features to your features

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

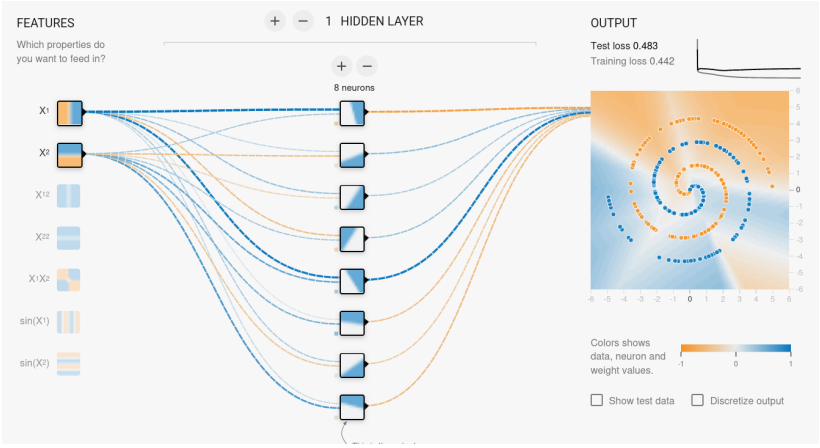
Features to your features

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:

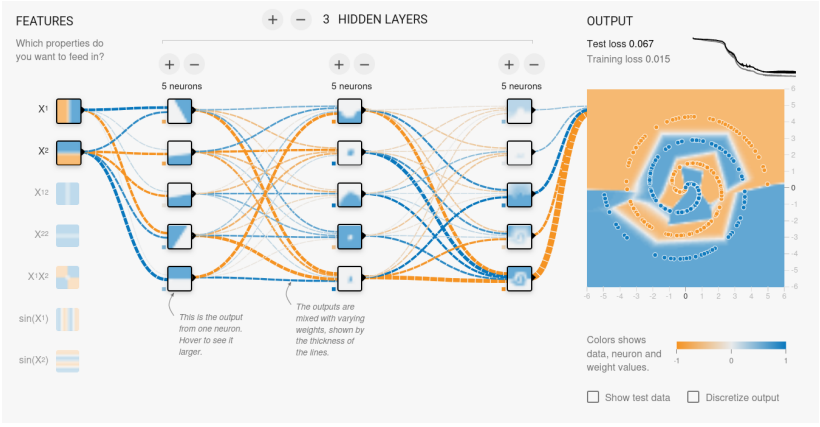


Visualizing the benefits of *multiple* layers with Playground



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

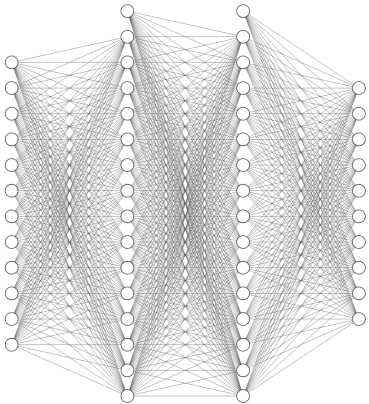


Figure: Feedforward DNNs are often represented only with input, hidden and output layers, the sum and activation function steps are implicit.

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 $\boldsymbol{\theta} = \{W_1, \boldsymbol{b_1}, W_2, \boldsymbol{b_2}, \cdots\}$ that best approximate some function g.

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

$$g(\boldsymbol{x_i}) = \boldsymbol{y_i} \text{ for } i \in [1, N]$$

$$g\left(\text{  }\right) = \text{“cat”}$$

$$\rightarrow \text{Dataset } \mathcal{D} = \{\boldsymbol{x_i}, \boldsymbol{y_i}\}_{i=1}^{N}$$

What do we want to achieve?

We want $f_{\boldsymbol{\theta}}(\boldsymbol{x_i})$ as close as possible to $g(\boldsymbol{x_i}) = \boldsymbol{y_i}$.
How do I measure this? If $y_i \in \mathbb{R}$:

$$\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^{N} |y_i - f_{\boldsymbol{\theta}}(\boldsymbol{x_i})|$$

For $\boldsymbol{y_i} \in \mathcal{Y} \subseteq \mathbb{R}^m$, use

$$\|\boldsymbol{y_i} - f_{\boldsymbol{\theta}}(\boldsymbol{x_i})\|_1 \doteq \sum_{j=1}^{m} \left| (\boldsymbol{y_i})_j - f_{\boldsymbol{\theta}}(\boldsymbol{x_i})_j \right|$$

What do we want to achieve?

We want $f_{\boldsymbol{\theta}}(\boldsymbol{x_i})$ as close as possible to $g(\boldsymbol{x_i}) = \boldsymbol{y_i}$.
How do I measure this? If $y_i \in \mathbb{R}$:

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} |y_i - f_{\boldsymbol{\theta}}(\boldsymbol{x_i})|$$

For $\boldsymbol{y_i} \in \mathcal{Y} \subseteq \mathbb{R}^m$, use

$$\|\boldsymbol{y_i} - f_{\theta}(\boldsymbol{x_i})\|_1 \doteq \sum_{j=1}^{m} \left|(\boldsymbol{y_i})_j - f_{\theta}(\boldsymbol{x_i})_j\right|$$

What do we want to achieve?

We want $f_{\boldsymbol{\theta}}(\boldsymbol{x_i})$ as close as possible to $g(\boldsymbol{x_i}) = \boldsymbol{y_i}$.
How do I measure this? If $y_i \in \mathbb{R}$:

$$\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^{N} |y_i - f_{\boldsymbol{\theta}}(\boldsymbol{x_i})|$$

For $\boldsymbol{y_i} \in \mathcal{Y} \subseteq \mathbb{R}^m$, use

$$\|\boldsymbol{y_i} - f_\theta(\boldsymbol{x_i})\|_1 \doteq \sum_{j=1}^{m} \left|(\boldsymbol{y_i})_j - f_\theta(\boldsymbol{x_i})_j\right|$$

What do we want to achieve?

We want $f_{\boldsymbol{\theta}}(\boldsymbol{x_i})$ as close as possible to $g(\boldsymbol{x_i}) = \boldsymbol{y_i}$.
How do I measure this? If $y_i \in \mathbb{R}$:

$$\min_{\boldsymbol{\theta}} \frac{1}{N} \sum_{i=1}^{N} |y_i - f_{\boldsymbol{\theta}}(\boldsymbol{x_i})|$$

For $\boldsymbol{y_i} \in \mathcal{Y} \subseteq \mathbb{R}^m$, use

$$\|\boldsymbol{y_i} - f_{\boldsymbol{\theta}}(\boldsymbol{x_i})\|_1 \doteq \sum_{j=1}^{m} \left| (\boldsymbol{y_i})_j - f_{\boldsymbol{\theta}}(\boldsymbol{x_i})_j \right|$$

Minimizing the Loss

Lots of possible choices for a “distance function” between y_i and $f_{\boldsymbol{\theta}}(\boldsymbol{x_i})$

$\rightarrow$ lots of possible choices for a **loss** $\mathcal{L}$

Minimizing the Loss

Lots of possible choices for a "distance function" between y_i and $f_{\boldsymbol{\theta}}(\boldsymbol{x_i})$

$\rightarrow$ lots of possible choices for a **loss** $\mathcal{L}$

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

Minimizing the Loss

Lots of possible choices for a "distance function" between y_i and $f_{\boldsymbol{\theta}}(\boldsymbol{x_i})$

$\rightarrow$ lots of possible choices for a **loss** $\mathcal{L}$

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

Minimizing the Loss

Lots of possible choices for a "distance function" between y_i and $f_{\boldsymbol{\theta}}(\boldsymbol{x_i})$

$\rightarrow$ lots of possible choices for a **loss** $\mathcal{L}$

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

Minimizing the Loss

Lots of possible choices for a "distance function" between y_i and $f_{\boldsymbol{\theta}}(\boldsymbol{x_i})$

$\rightarrow$ lots of possible choices for a **loss** $\mathcal{L}$

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

Minimizing the Loss

Lots of possible choices for a "distance function" between y_i and $f_{\boldsymbol{\theta}}(\boldsymbol{x_i})$
$\rightarrow$ lots of possible choices for a **loss** $\mathcal{L}$

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

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

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

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 θ 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 θ 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?

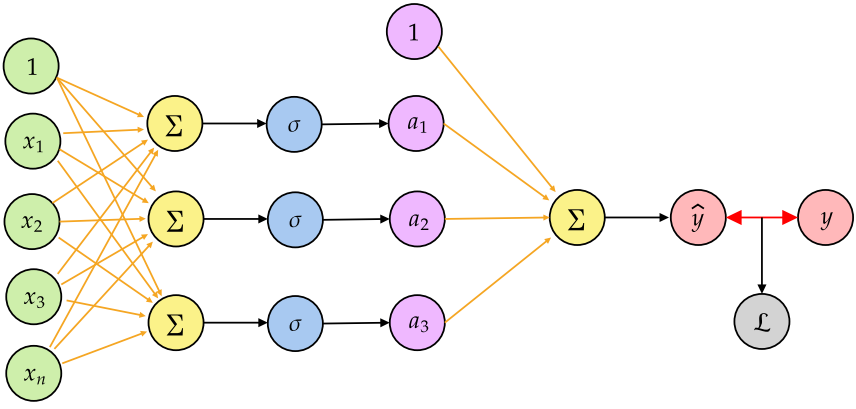
Options:

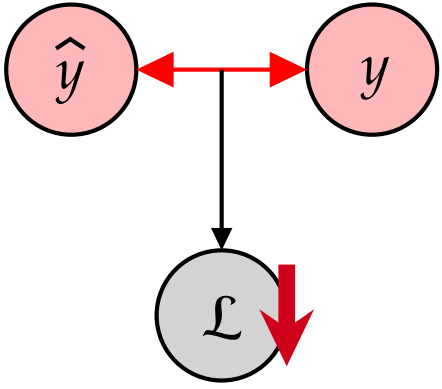
- pick each weight by hand (thousands, millions, billions)
- sample a bunch of θ and pick the best?
- keep taking weights around the best-performing ones (genetic algorithms)
- **Gradient Descent**!

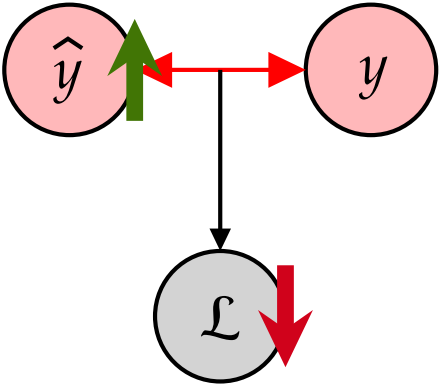
Gradient Descent

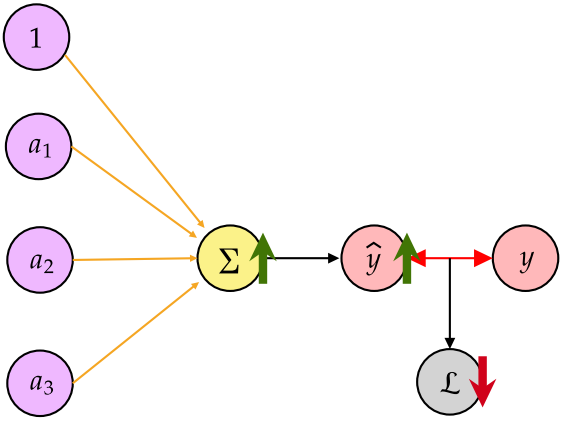
General idea:

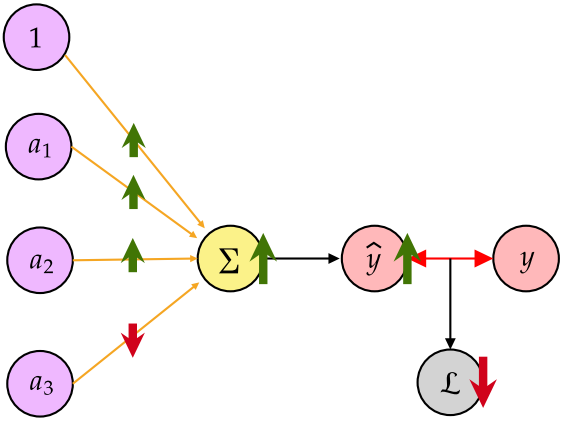
How should I change $w_{ij}^{(l)}$ so that $\mathcal{L}$ gets a bit smaller?

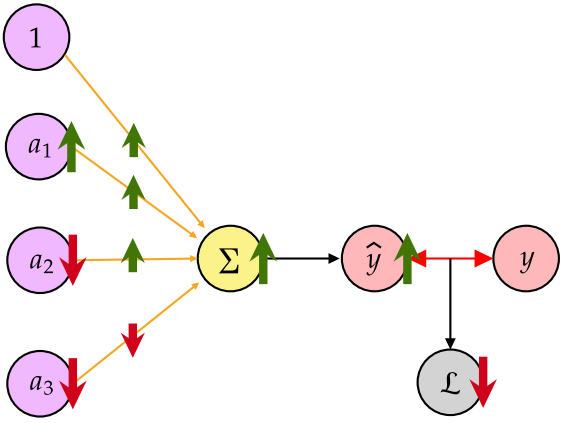

1
x_1
x_2
x_3
x_n
Σ
Σ
Σ
σ
σ
σ
1
a_1
a_2
a_3
Σ
$\hat{y}$
y
$\mathcal{L}$

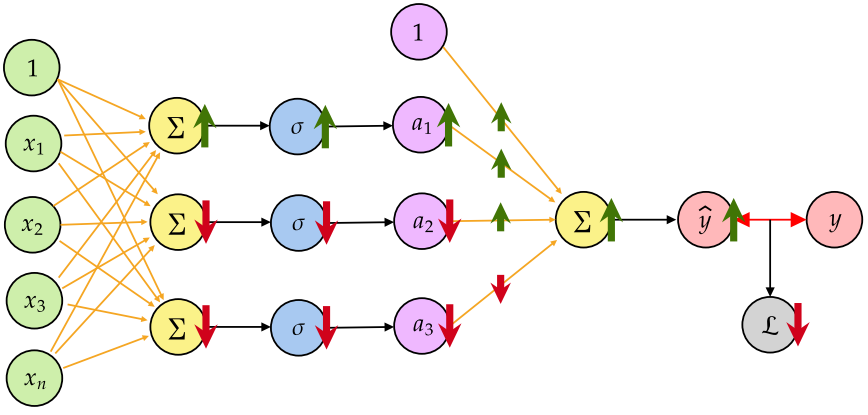

$\hat{y}$
y
$\mathcal{L}$

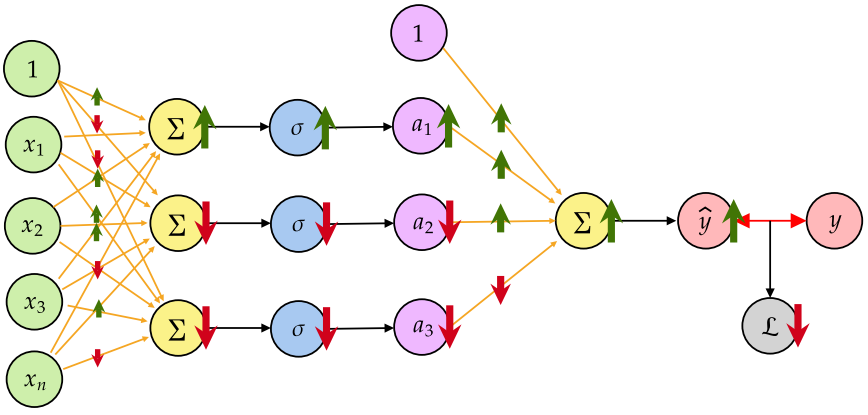

$\hat{y}$
y
$\mathcal{L}$


1
a_1
a_2
a_3
Σ
$\hat{y}$
y
$\mathcal{L}$


1
a_1
a_2
a_3
Σ
$\hat{y}$
y
$\mathcal{L}$


1
a_1
a_2
a_3
Σ
$\hat{y}$
y
$\mathcal{L}$


1
x_1
x_2
x_3
x_n
Σ
Σ
Σ
σ
σ
σ
1
a_1
a_2
a_3
Σ
$\hat{y}$
y
$\mathcal{L}$


1
x_1
x_2
x_3
x_n
Σ
Σ
Σ
σ
σ
σ
1
a_1
a_2
a_3
Σ
$\hat{y}$
y
$\mathcal{L}$

Gradient Descent

General idea:

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

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

Gradient Descent

General idea:

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

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

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

Gradient Descent

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

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

Reminder 1D:

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

Gradient Descent

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

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

Reminder 1D:

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

- 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

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

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

$\rightarrow$ in general, the gradient $\nabla_{\boldsymbol{\theta}} \mathcal{L}$ is the direction of **steepest ascent**

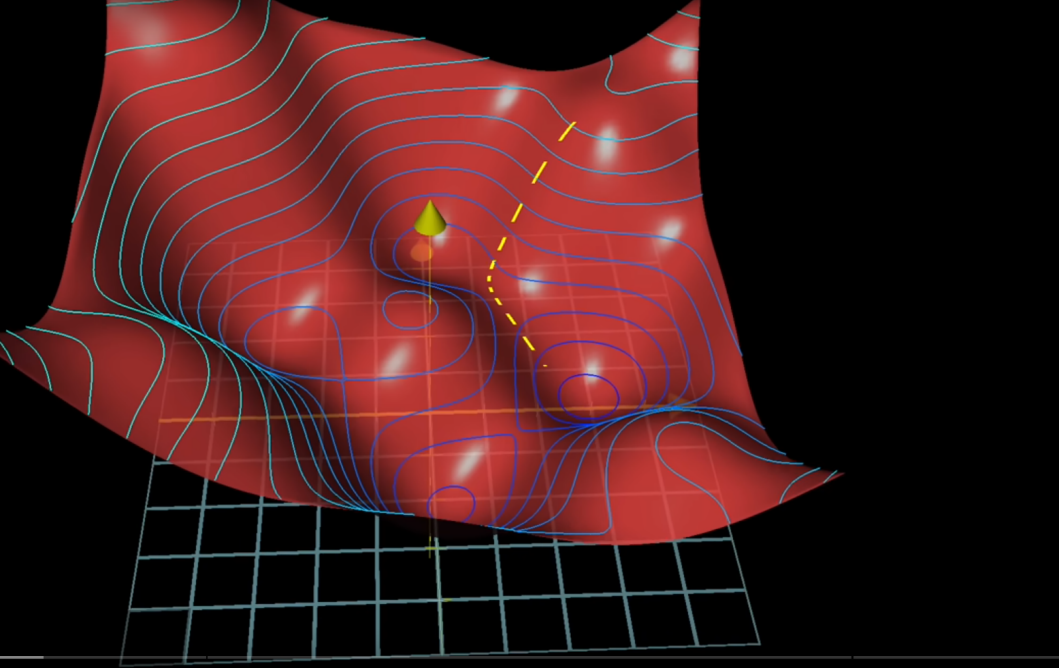
$\rightarrow$ simply follow $-\nabla_{\boldsymbol{\theta}} \mathcal{L}$ to get **steepest descent**: best local change to each $w_{ij}^{(l)}$

Gradient Descent

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

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

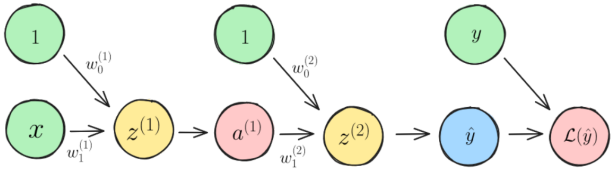
$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \frac{\partial \mathcal{L}}{\partial w_{ij}^{(l)}}$$



/ 12:46 • Récapitulation ›

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**.



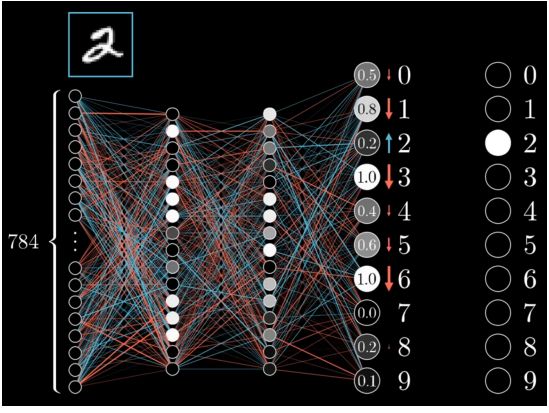
$$\frac{\partial \mathcal{L}}{\partial w_1^{(1)}} = \underbrace{\frac{\partial \mathcal{L}}{\partial \hat{y}}}_{-2(y-\hat{y})} \cdot \underbrace{\frac{\partial \hat{y}}{\partial z^{(2)}}}_{1} \cdot \underbrace{\frac{\partial z^{(2)}}{\partial a^{(1)}}}_{w_1^{(2)}} \cdot \underbrace{\frac{\partial a^{(1)}}{\partial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

Figure: Slide from the DL course of Timon Deschamps

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

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

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

Mini-batch Gradient Descent

We are looking for

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

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

$\rightarrow$ compute only $\mathcal{L}$ over a small *batch* of data

Mini-batch Gradient Descent

We are looking for

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

Computing this over the whole dataset: **impractical** to **impossible**!
$\rightarrow$ 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**.

Training of a Deep Neural Network (simplified)

1. Sample data $\boldsymbol{x_i}, \boldsymbol{y_i}$ from dataset $\mathcal{D}$
2. Compute "forward" pass $f_{\boldsymbol{\theta}}(\boldsymbol{x_i}) = \hat{y}_i$
3. Compute loss function $\mathcal{L}(\hat{y}_i, \boldsymbol{y_i})$
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 mutiplication
- 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	σ
Loss function	$\mathcal{L}$
Learning rate	η

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	σ
Loss function	$\mathcal{L}$
Learning rate	η
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	σ
Loss function	$\mathcal{L}$
Learning rate	η
LOTS more to come	optim, layers, batch size, norms,...

$\boldsymbol{\theta}$ 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	σ
Loss function	$\mathcal{L}$
Learning rate	η
LOTS more to come	optim, layers, batch size, norms,...

$\boldsymbol{\theta}$ are the *parameters* of the model, above are *"hyper"parameters*

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

$\rightarrow$ follow whatever people have done in similar contexts

Let’s implement this!