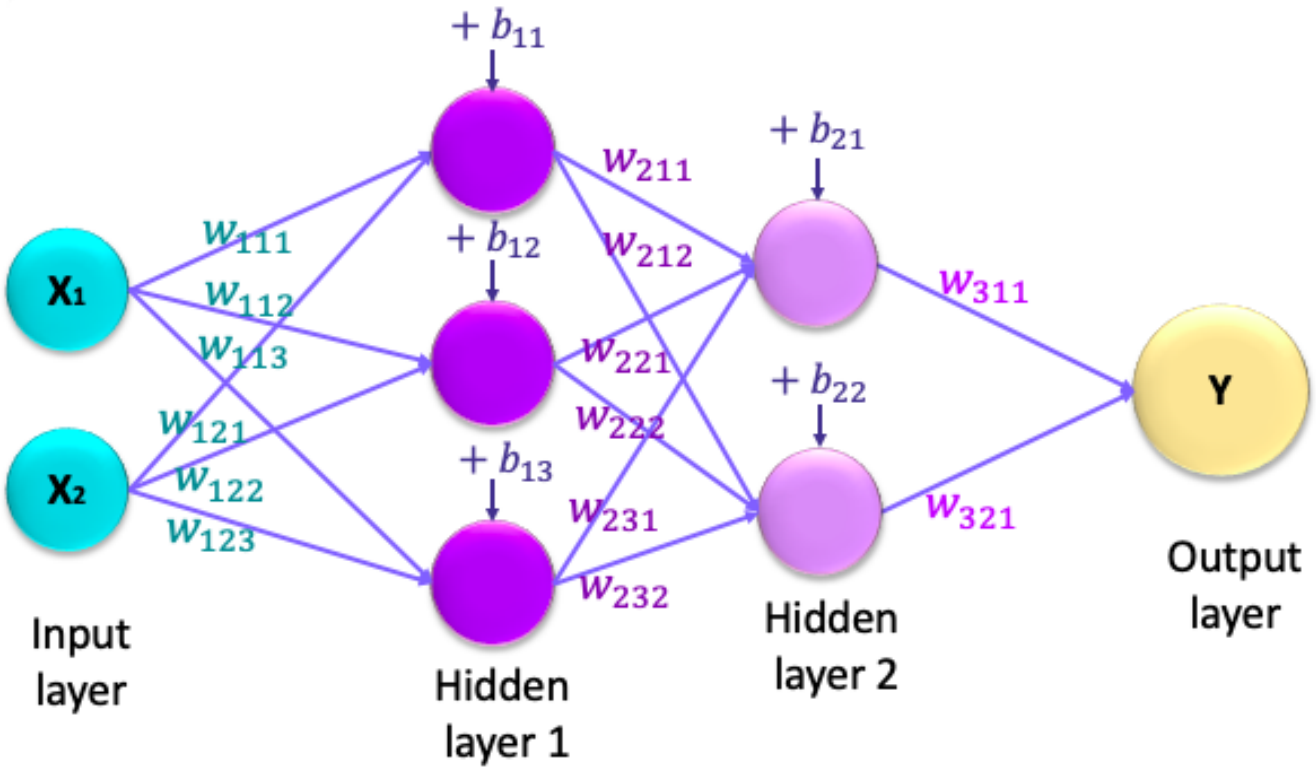
Optimizers

Optimizers

What are optimizers ?

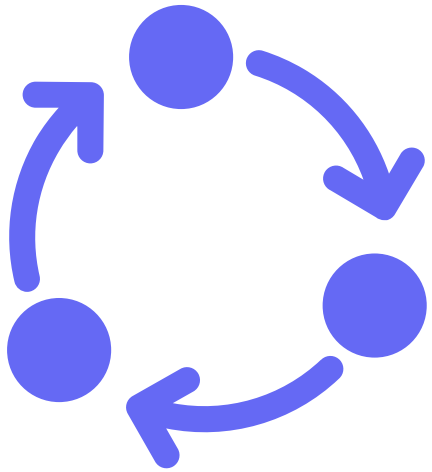
Optimizers (optimization algorithms) minimize the loss function by finding the most accurate model parameters possible



Model parameters correspond to weights and biases

Optimizers

What are optimizers ?



Training a model is an iterative process

It is essential for the optimizers to be :

- → Fast
- → Accurate

Optimizers

We will cover the following optimizers :

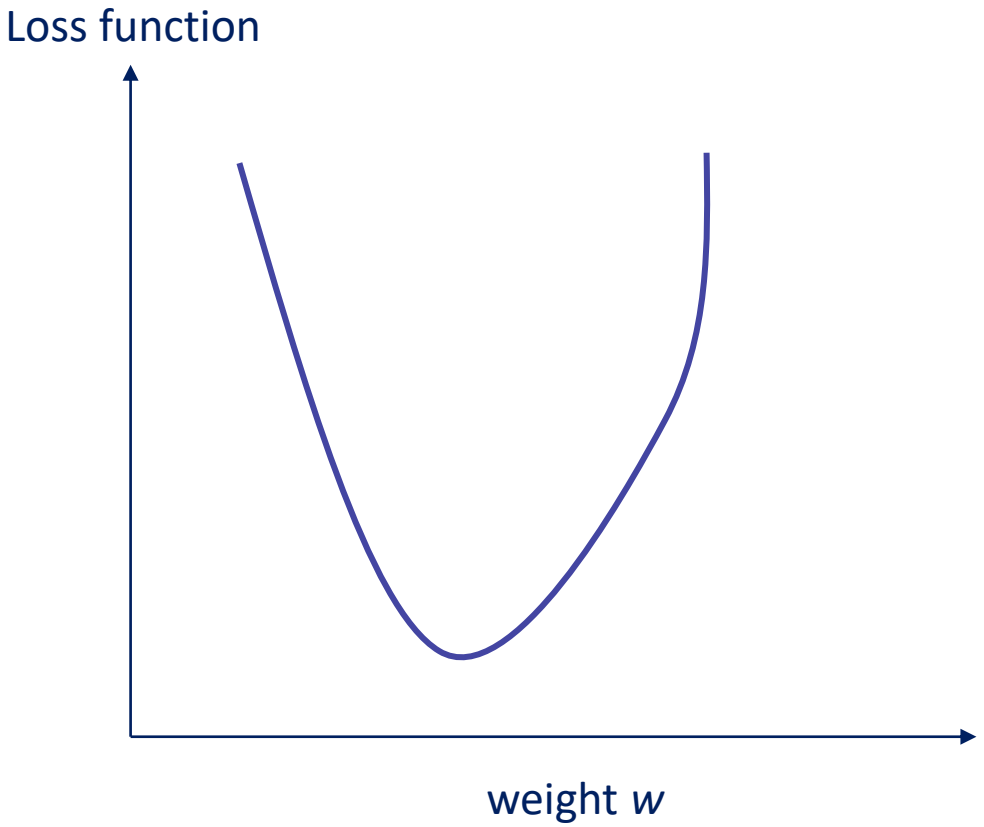
- Gradient Descent
- Stochastic Gradient Descent
- Mini-Batch Gradient Descent
- Gradient Descent with Momentum
- Adagrad
- RMS Prop
- Adam

Optimizers

1

Gradient Descent :

Calculating the loss for every possible value of w is an exhaustive and an inefficient way to find the minimum loss !



A more efficient method is the Gradient Descent

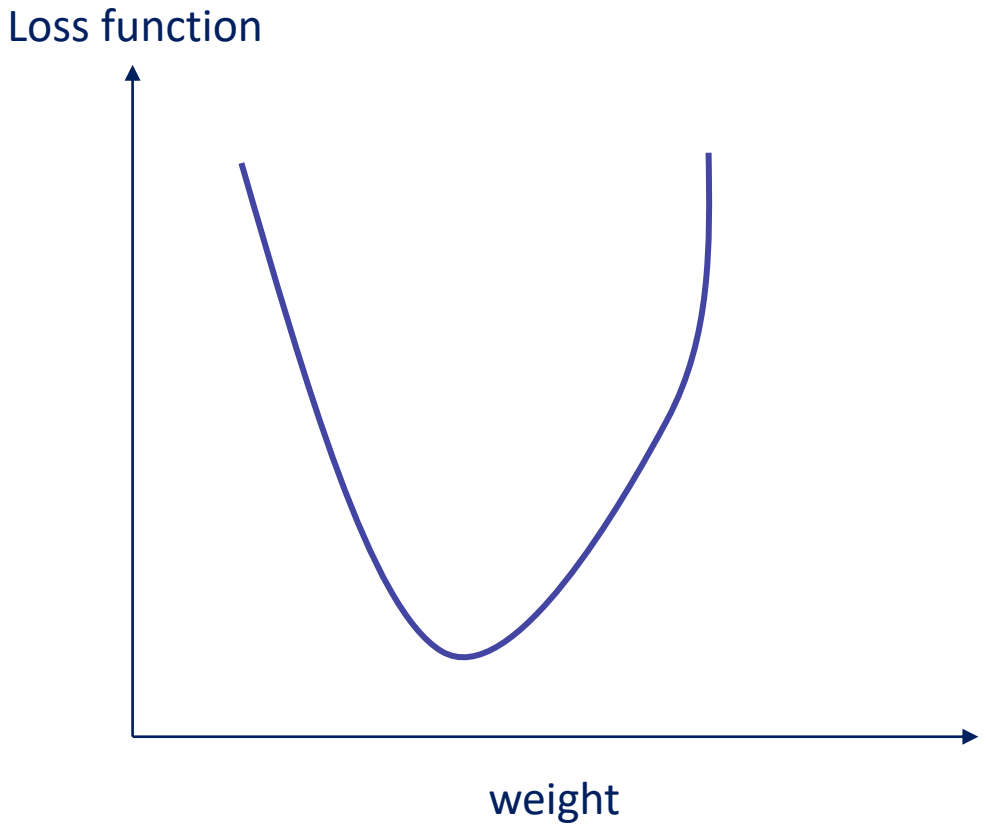
Optimizers

1

Gradient Descent :

- An optimization algorithm that finds the minimum value of a function by taking steps starting from an initial value until it finds the best value

- It takes big steps towards the minimum if far from the optimum value and smaller steps as it gets closer



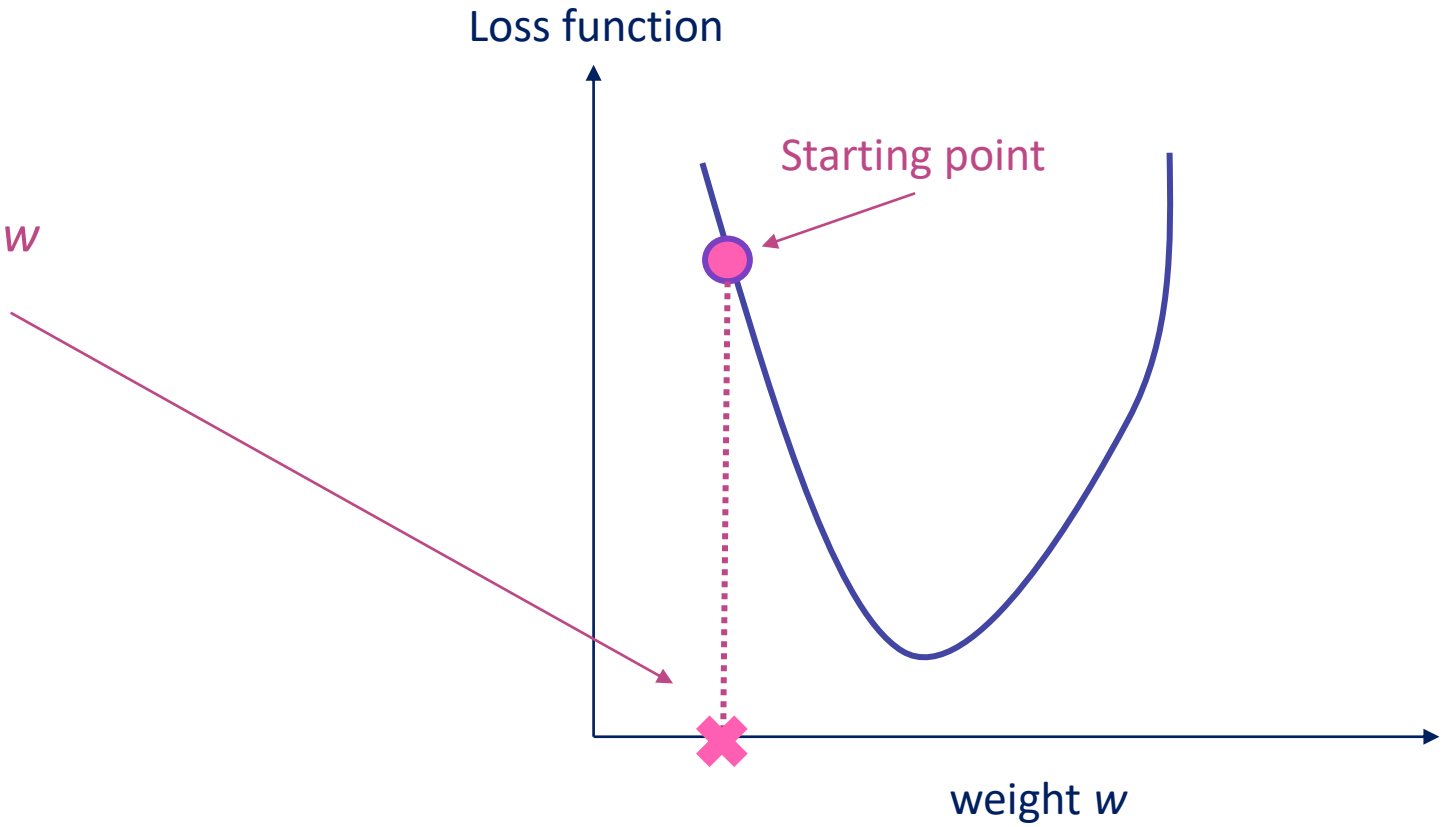
Optimizers

1

Gradient Descent :

Step 1 : We set a starting value for weight *w*

(this value can be randomly chosen)



Optimizers

1

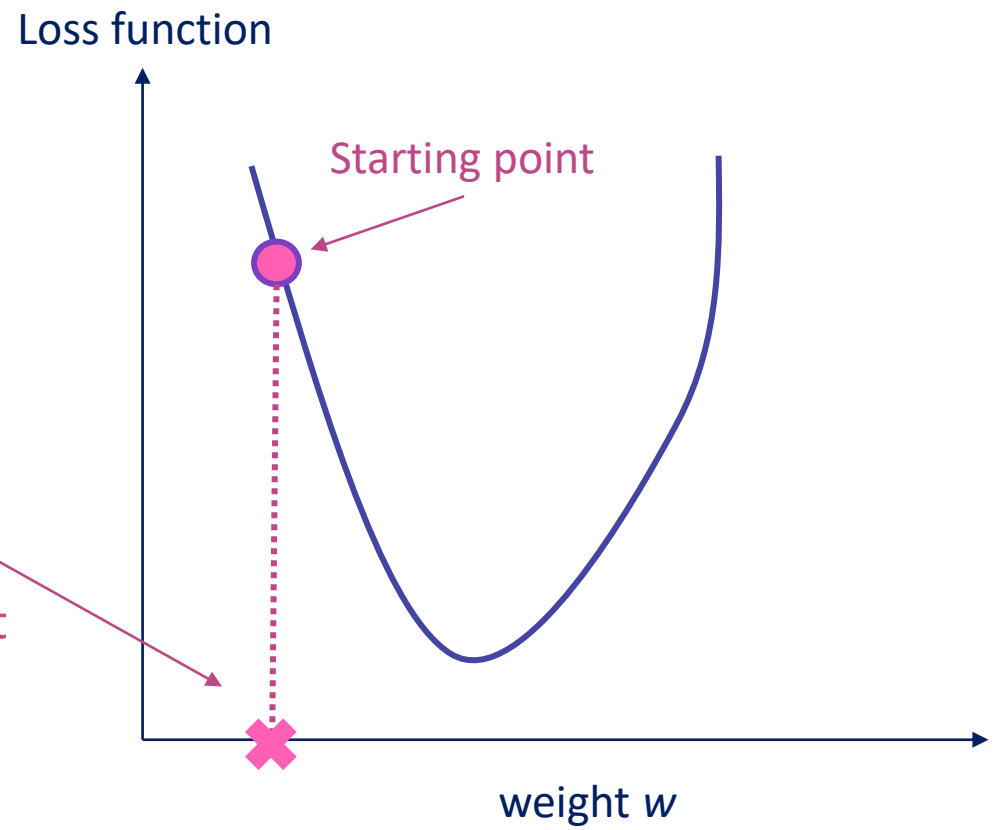
Gradient Descent :

Step 1 : We set a starting value for weight *w*

(this value can be randomly chosen)

Step 2 : The Gradient Descent algorithm calculates the gradient of the loss function at the starting point.

The gradient tells which direction allows the cost function to decrease



Optimizers

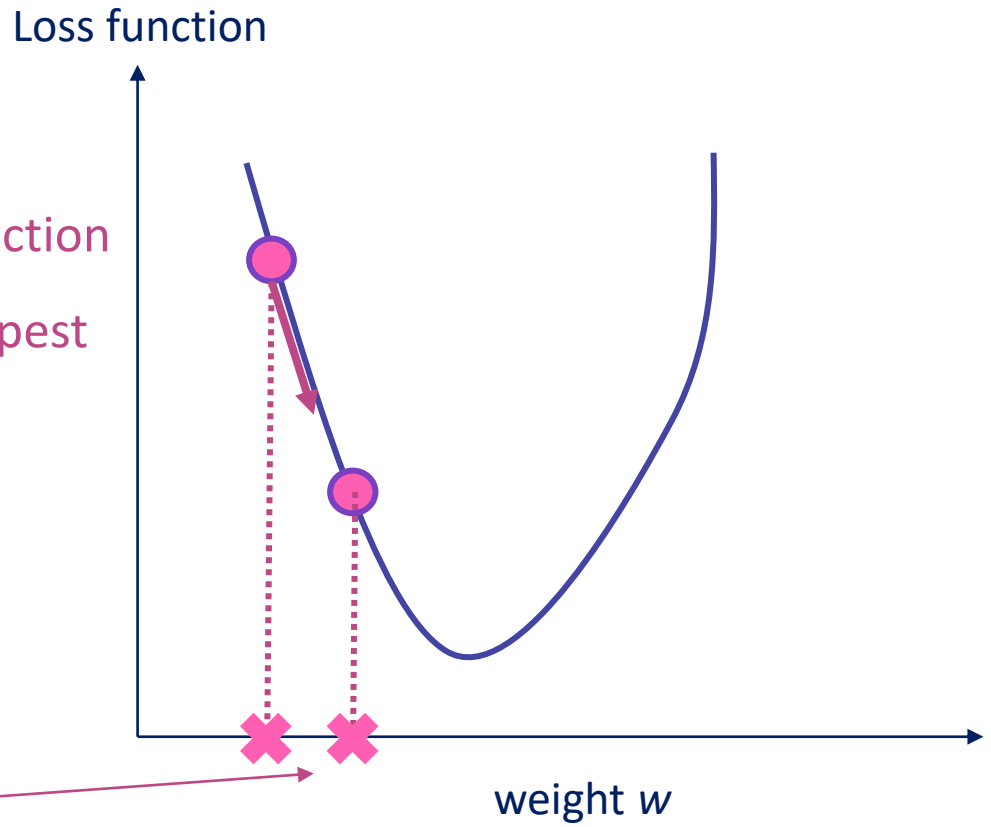
1

Gradient Descent :

Step 3 : The algorithm takes a step in the negative gradient direction (note : the gradient always points towards the direction of steepest increase ➔ we use the negative gradient)

*Step size = gradient magnitude * learning rate*

New weight = old weight – step size



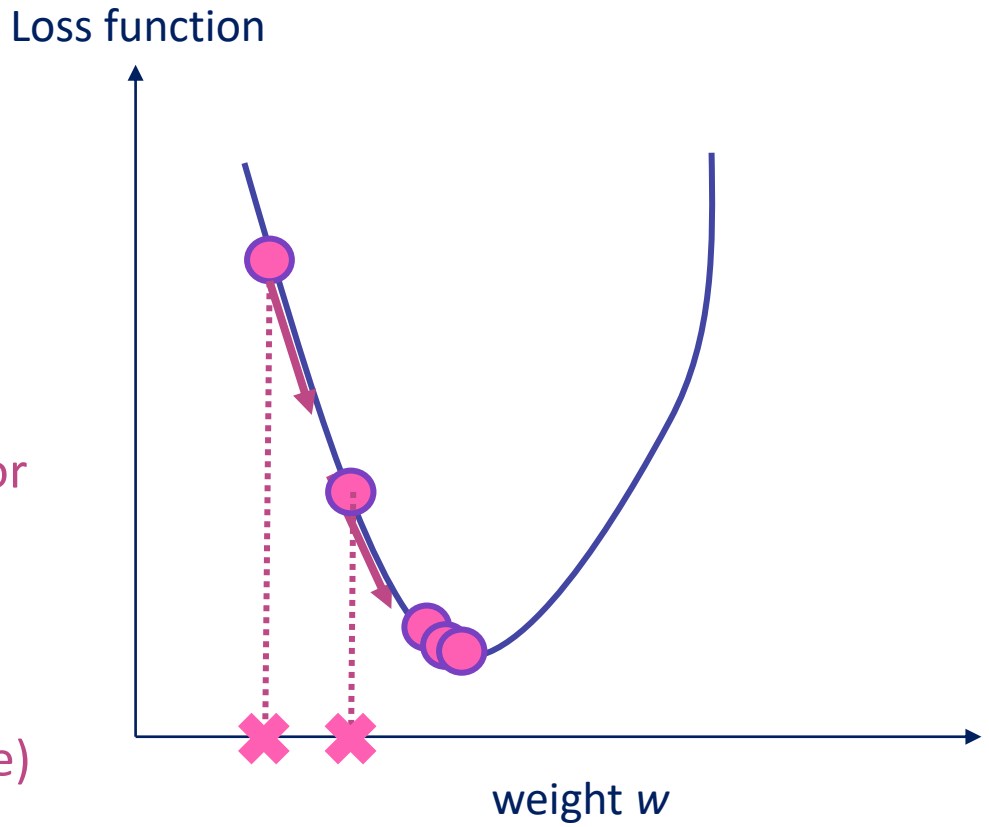
Optimizers

1

Gradient Descent :

The Gradient Descent algorithm repeats these steps until :

- Step size gets very close to 0 (ex. min step size = 0.001 or smaller)
- Maximum number of steps is reached (ex. 1000 or more)



Optimizers

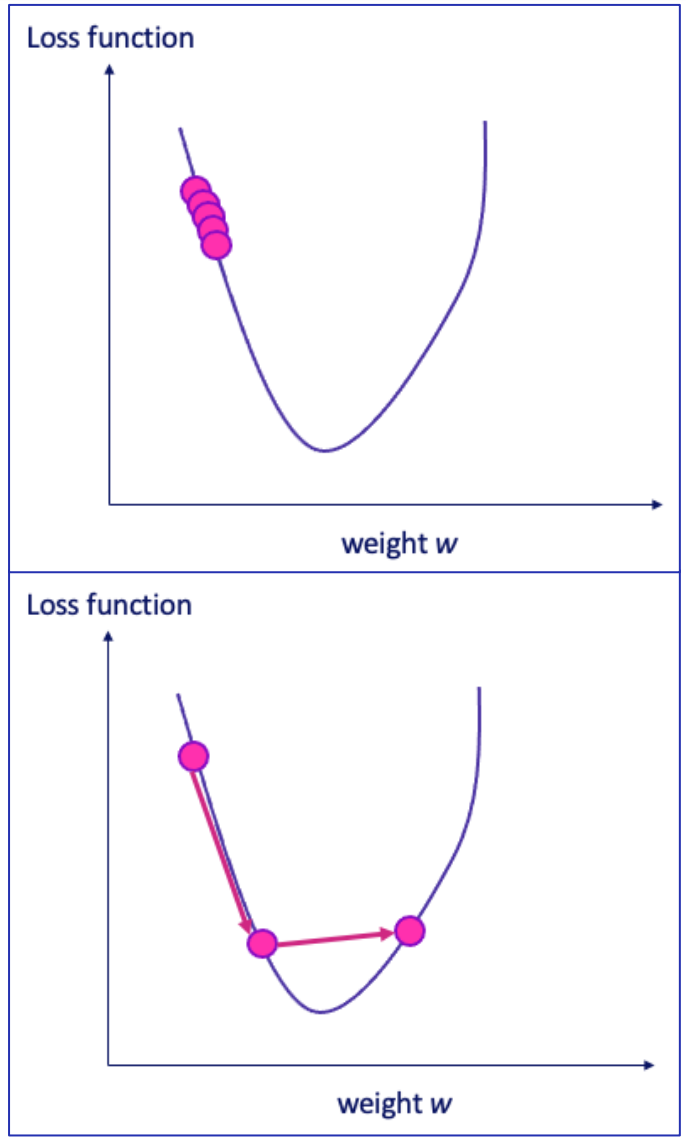
1

Gradient Descent :

Learning rate : A scalar that determines the step size

If learning rate is very small ➔ learning can take too long

If learning is very large ➔ the point will perpetually bounce across the curve minimum



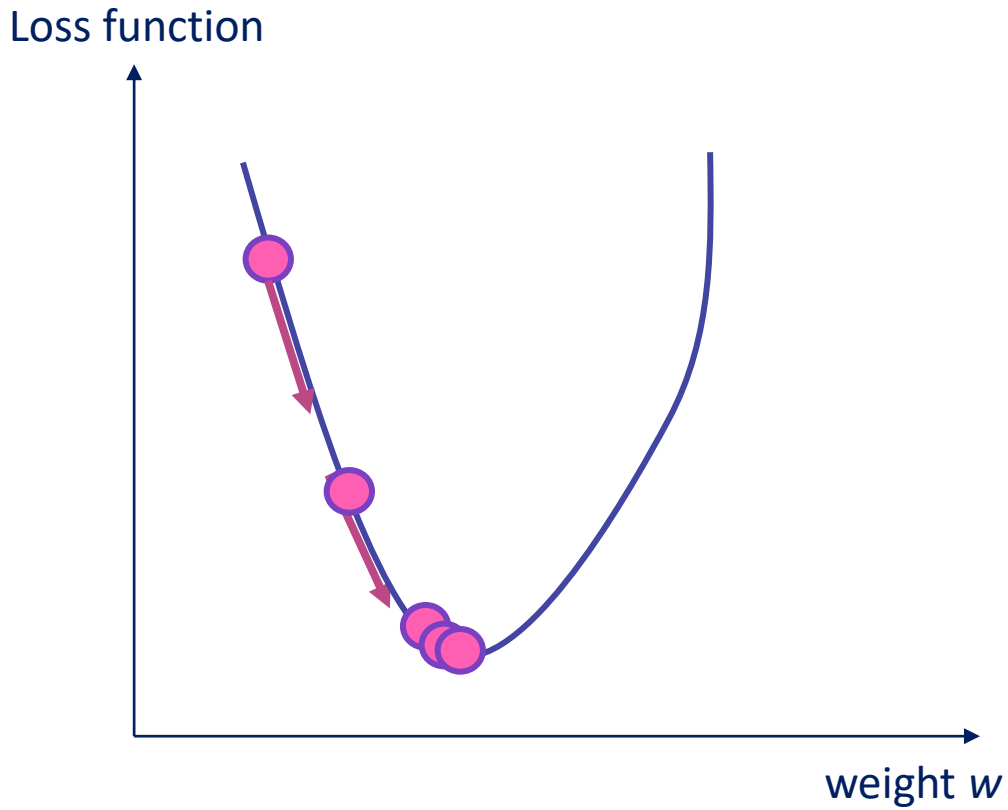
Optimizers

1

Gradient Descent :

Learning rate : A scalar that determines the step size

Ideal learning rate : larger when the point is far from the minimum & smaller as it gets closer



Gradient Descent is very sensitive to learning rate.

In practice, the learning rate can be determined automatically during training (starts large and diminishes gradually).

This strategy is called ‘schedule’.

Optimizers

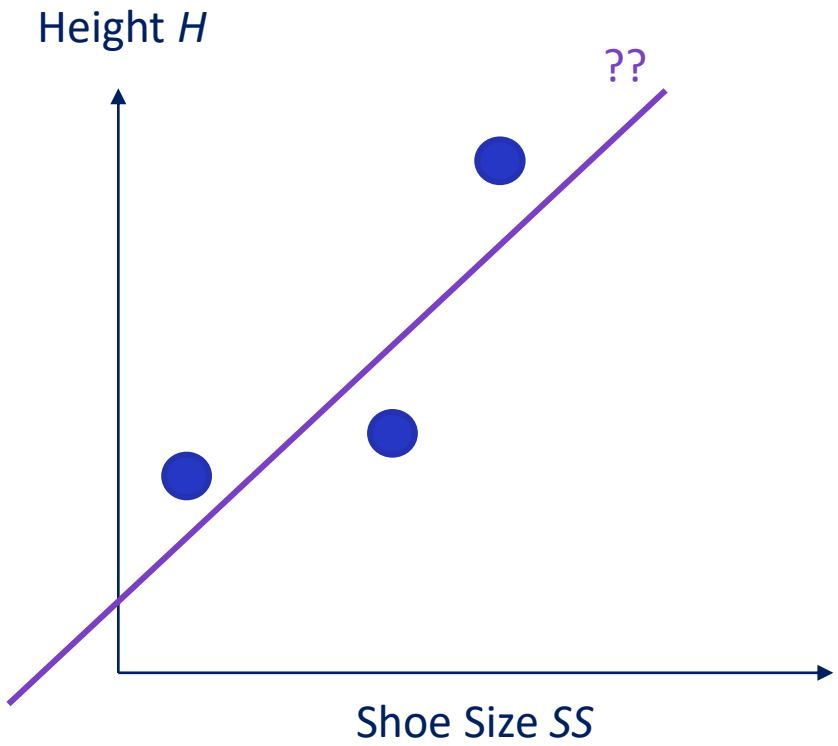
1

Gradient Descent :

Example : How does Gradient Descent fit a line to data ?

Use Gradient Descent to estimate the optimal values for the slope and the intercept :

$$H_{pred} = slope\ SS + intercept$$

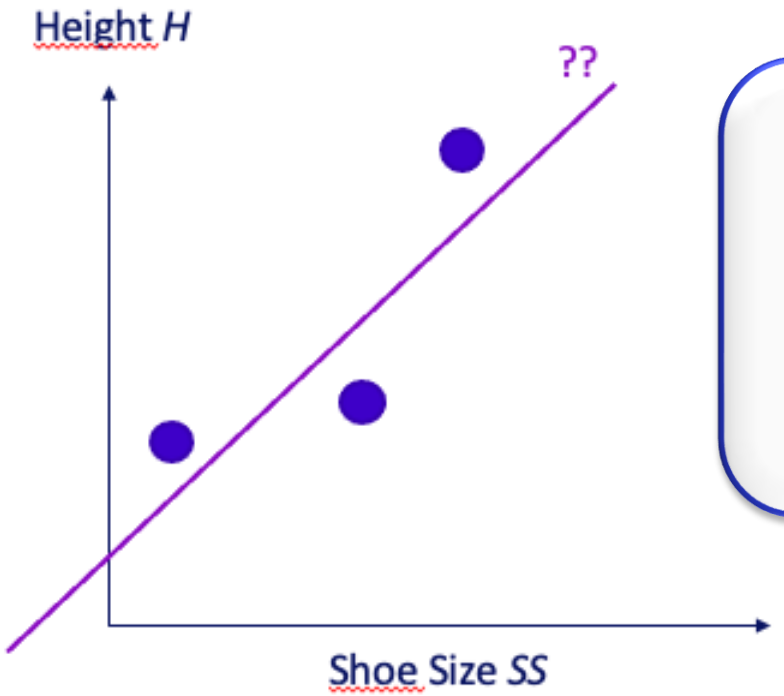


Optimizers

1

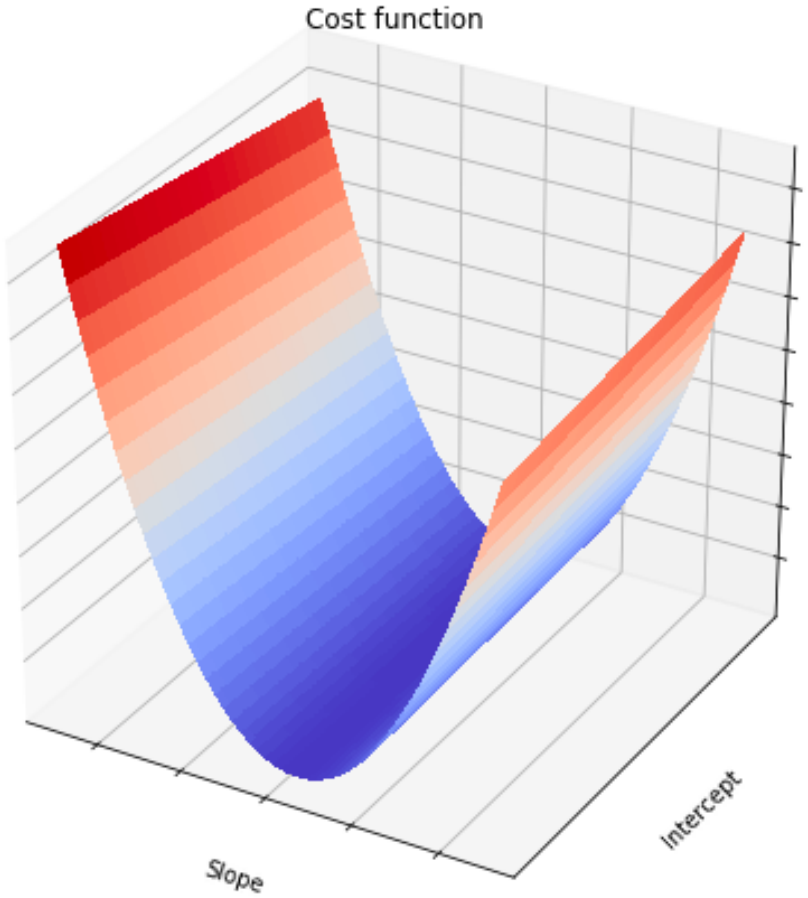
Gradient Descent :

Example : How does Gradient Descent fit a line to data ?



Use GD to estimate the optimal values for the slope and the intercept :

$$H_{pred} = slope\ SS + intercept$$

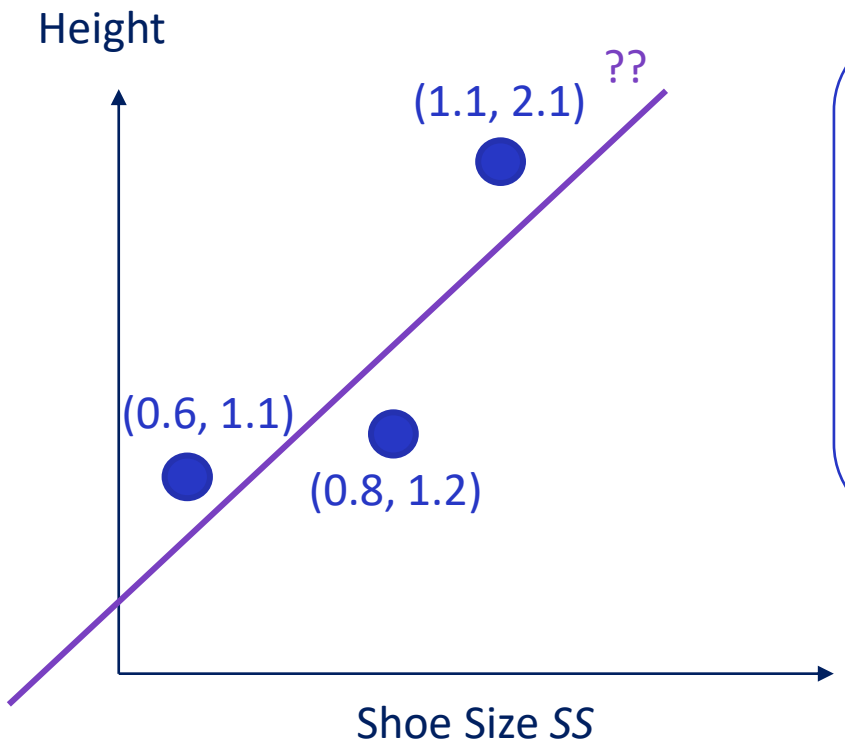


Optimizers

1

Gradient Descent :

$$H_{pred} = slope\ SS + intercept$$



Loss function

= Sum of Squared residuals (SSR)

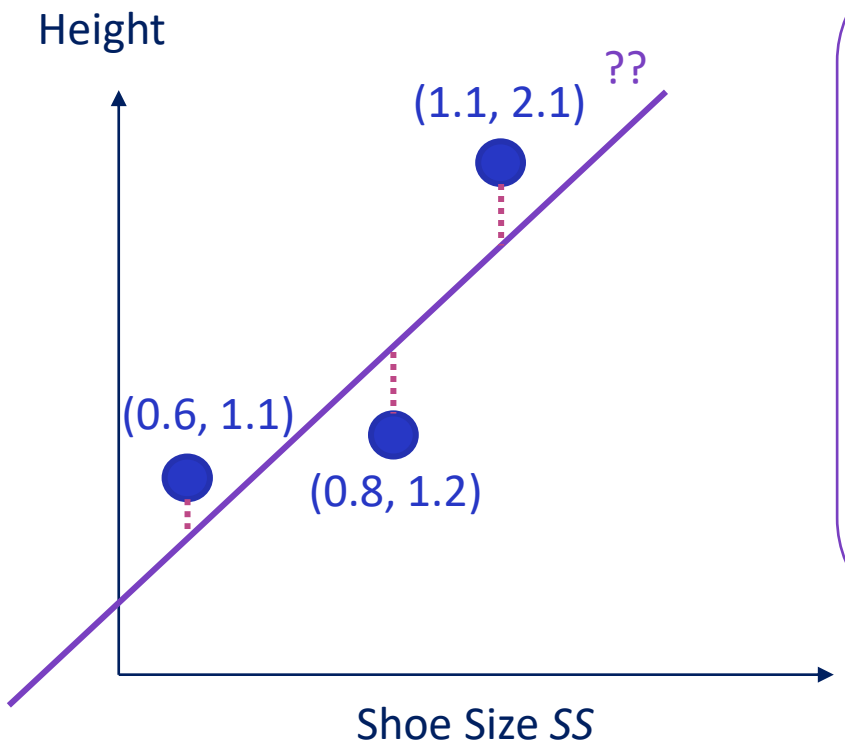
$$= (H_{true_1} - H_{pred_1})^2 + (H_{true_2} - H_{pred_2})^2 + (H_{true_3} - H_{pred_3})^2$$

Optimizers

1

Gradient Descent :

$$H_{pred} = slope\ SS + intercept$$



Loss function

= Sum of Squared residuals (SSR)

$$= (H_{true_1} - H_{pred_1})^2 + (H_{true_2} - H_{pred_2})^2 + (H_{true_3} - H_{pred_3})^2$$

$$= (1.1 - (slope * 0.6 + intercept))^2$$
$$+ (1.2 - (slope * 0.8 + intercept))^2$$
$$+ (2.1 - (slope * 1.1 + intercept))^2$$

Optimizers

1

Gradient Descent :

$$H_{pred} = slope\ SS + intercept$$

Loss function = Sum of Squared residuals (SSR)

$$= (1.1 - (slope * 0.6 + intercept))^2$$
$$+ (1.2 - (slope * 0.8 + intercept))^2$$
$$+ (2.1 - (slope * 1.1 + intercept))^2$$

$$\frac{\text{d(Loss function)}}{d(intercept)}$$

$$\frac{\text{d(Loss function)}}{d(slope)}$$

Optimizers

1

Gradient Descent :

$$H_{pred} = slope\ SS + intercept$$

To calculate the gradient with respect to each of the parameters

$$\frac{\text{d(Loss function)}}{d(intercept)}$$

$$\frac{\text{d(Loss function)}}{d(slope)}$$

we use the Chain Rule

Optimizers

1

Gradient Descent :

$$H_{pred} = slope\ SS + intercept$$

Loss function = Sum of Squared residuals (SSR)

$$= (\overbrace{1.1 - (slope * 0.6 + intercept)}^{U})^2$$

$$+ (1.2 - (slope * 0.8 + intercept))^2$$

$$+ (2.1 - (slope * 1.1 + intercept))^2$$

$$\frac{\text{d(Loss function)}}{d(intercept)}$$

$$= \frac{\text{d(Loss function)}}{d(U)} \times \frac{\text{d(U)}}{d(intercept)}$$

$$\frac{\text{d(Loss function)}}{d(slope)}$$

$$= \frac{\text{d(Loss function)}}{d(U)} \times \frac{\text{d(U)}}{d(slope)}$$

Optimizers

1

Gradient Descent :

$$H_{pred} = slope\ SS + intercept$$

Loss function = Sum of Squared residuals (SSR)

$$= (\overbrace{1.1 - (slope * 0.6 + intercept)}^{U})^2$$

$$+ (1.2 - (slope * 0.8 + intercept))^2$$

$$+ (2.1 - (slope * 1.1 + intercept))^2$$

$$\frac{\text{d(Loss function)}}{d(intercept)}$$

$$= \frac{\text{d(Loss function)}}{d(U)} \times \frac{\text{d(U)}}{d(intercept)}$$

$$= 2U_1 \times (-1) + 2U_2 \times (-1) + 2U_3 \times (-1)$$

$$= -2(1.1 - (0.6 * slope + intercept))$$

$$-2(1.2 - (0.8 * slope + intercept))$$

$$-2(2.1 - (1.1 * slope + intercept))$$

Optimizers

1

Gradient Descent :

$$H_{pred} = slope\ SS + intercept$$

Loss function = Sum of Squared residuals (SSR)

$$= (\overbrace{1.1 - (slope * 0.6 + intercept)}^{U})^2$$

$$+ (1.2 - (slope * 0.8 + intercept))^2$$

$$+ (2.1 - (slope * 1.1 + intercept))^2$$

$$\frac{\text{d(Loss function)}}{d(slope)}$$

$$= \frac{\text{d(Loss function)}}{d(U)} \times \frac{\text{d(U)}}{d(slope)}$$

$$= 2U_1 \times (-0.6) + 2U_2 \times (-0.8) + 2U_3 \times (-1.1)$$

$$= -1.2(1.1 - (0.6 * slope + intercept))$$

$$-1.6(1.2 - (0.8 * slope + intercept))$$

$$-2.2(2.1 - (1.1 * slope + intercept))$$

Optimizers

1

Gradient Descent :

$$\frac{\text{d(Loss function)}}{d(intercept)}$$

$$= -2(1.1 - (0.6 * slope + intercept))$$

$$-2(1.2 - (0.8 * slope + intercept))$$

$$-2(2.1 - (1.1 * slope + intercept))$$

$$\frac{\text{d(Loss function)}}{d(slope)}$$

$$= -1.2(1.1 - (0.6 * slope + intercept))$$

$$-1.6\big(1.2 - (0.8 * slope + intercept)\big)$$

$$-2.2(2.1 - (1.1 * slope + intercept))$$

Optimizers

1

Gradient Descent :

Note : The partial derivatives of a multivariate function

are stored in a vector called the **Gradient** (∇)

$$\nabla \text{ Loss Function} = \begin{bmatrix} \frac{\text{d(Loss function)}}{d(slope)} \\ \frac{\text{d(Loss function)}}{d(intercept)} \end{bmatrix}$$

Optimizers

1

Gradient Descent :

$$\nabla \text{ Loss Function} = \begin{bmatrix} \frac{\text{d(Loss function)}}{d(slope)} \\ \frac{\text{d(Loss function)}}{d(intercept)} \end{bmatrix}$$

We can now use this Gradient to descend to the minimal point in the cost function :

Step 1 : We choose a random intercept (= 0) and a random slope (= 1)

Step 2 : We plug the values in the partial derivative formulas and get 2 values

Step 3 : $stepsize_{slope} = value_1 * learning\ rate$

$stepsize_{intercept} = value_2 * learning\ rate$

Step 4 : $slope_{new} = slope_{old} - stepsize_{slope}$

$intercept_{new} = intercept_{old} - stepsize_{intercept}$

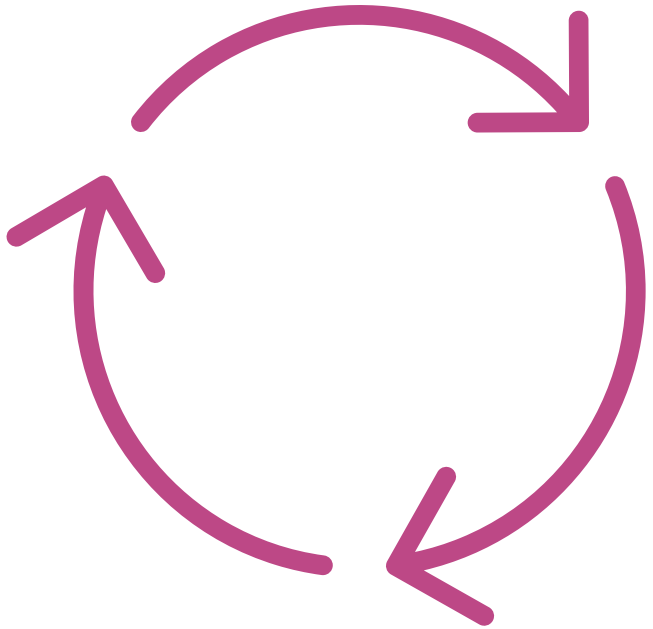
Optimizers

1

Gradient Descent :

$$\nabla \text{ Loss Function} = \begin{bmatrix} \frac{\text{d(Loss function)}}{d(slope)} \\ \frac{\text{d(Loss function)}}{d(intercept)} \end{bmatrix}$$

We can now use this Gradient to descend to the minimal point in the cost function (hence the name Gradient Descent) :



Repeat steps 1 to 4 until :

- ✓ the step sizes are very small
- ✓ the maximum number of steps is reached

Optimizers

1

Gradient Descent :

Note : In the case where we have more parameters to estimate (weights and biases of our ANN), we only use more derivatives, while the whole procedure remains exactly the same

Note : Gradient Descent can be used with any loss function

Optimizers

1

Gradient Descent :

Conclusion :

Area	Number of rooms	Distance from train station	city	Price (label)
32	3	5	Amiens	60 000
25	1	8	Lille	70 000
80	4	10	Versailles	600 000
55	3	10	Nice	450 000

The data is entirely plugged into the Neural Network and the parameters are adjusted

Optimizers

2

Stochastic Gradient Descent :

Gradient Descent considers the whole dataset for each parameter estimation step.

If we have thousands of parameters and millions of data points, repeating all the necessary steps to find the optimal parameters takes a huge amount of time

Optimizers

2

Stochastic Gradient Descent :

Stochastic Gradient Descent :

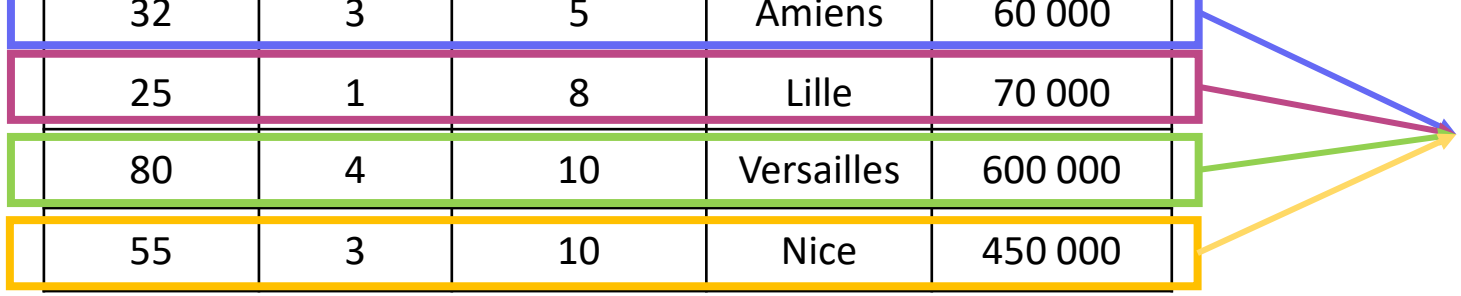
- uses 1 sample per step (calculation of new adjusted parameters)
- reduces the time taken for derivative calculation

Optimizers

2

Stochastic Gradient Descent :

Area	Number of rooms	Distance from train station	city	Price (label)
32	3	5	Amiens	60 000
25	1	8	Lille	70 000
80	4	10	Versailles	600 000
55	3	10	Nice	450 000



The data is plugged in the Neural Network one sample at a time and the parameters are adjusted after every sample

Optimizers

2

Stochastic Gradient Descent :

Notice the low memory requirement as compared to Gradient Descent :

Estimates concerning previous data do not have to be conserved in memory.

We can only store the very last estimates, and use them when new data is added.

Area	Number of rooms	Distance from train station	city	Price (label)
32	3	5	Amiens	60 000
25	1	8	Lille	70 000
80	4	10	Versailles	600 000
55	3	10	Nice	450 000

Optimizers

2

Stochastic Gradient Descent :

Note :

- Sensitivity to learning rate also applies to SGD
- The same ‘schedule’ strategy is used : Starting with large numbers and reducing them gradually

Optimizers

3

Mini-Batch Gradient Descent :

Mini-Batch Gradient Descent :

- uses a small subset of data (mini-batch) for each step

- gives more stable results than using one sample per step (Stochastic Gradient Descent)

- faster than using the whole dataset (Gradient Descent)

Optimizers

3

Mini-Batch Gradient Descent :

Area	Number of rooms	Distance from train station	city	Price (label)
32	3	5	Amiens	60 000
25	1	8	Lille	70 000
80	4	10	Versailles	600 000
55	3	10	Nice	450 000
Etc.	Etc.	Etc.	Etc.	Etc.

mini-batch

Optimizers

4

Gradient Descent with Momentum :

Gradient Descent might make a lot of steps and keep on

oscillating very slowly towards the minimum of the lost function.

Gradient Descent with Momentum :

- uses the exponentially weighted average of the gradients to update the weights
- smoothes out the steps of the Gradient Descent (because it takes into consideration the average of the past parameters)
- is faster than the regular Gradient Descent

Optimizers

4

Gradient Descent with Momentum :

1. On each iteration, we compute the partial derivatives while plugging in the initial slope and intercept values

$$\underbrace{\frac{\text{d(Loss function)}}{d(slope)}}_{value_{slope}} \;\&\; \underbrace{\frac{\text{d(Loss function)}}{d(intercept)}}_{value_{intercept}}$$

Optimizers

4

Gradient Descent with Momentum :

1. On each iteration, we compute the partial derivatives while plugging in the initial slope and intercept values

$$\underbrace{\frac{\text{d(Loss function)}}{d(slope)}}_{value_{slope}} \;\&\; \underbrace{\frac{\text{d(Loss function)}}{d(intercept)}}_{value_{intercept}}$$

2. Compute :

$$weighted\_average_{slope} = \boldsymbol{\beta}\, weighted\_average_{slope} + (1 - \boldsymbol{\beta}) value_{slope}$$

$$weighted\_average_{intercept} = \boldsymbol{\beta}\, weighted\_average_{intercept} + (1 - \boldsymbol{\beta}) value_{intercept}$$

Optimizers

4

Gradient Descent with Momentum :

1. On each iteration, we compute the partial derivatives while plugging in the initial slope and intercept values

$$\underbrace{\frac{\text{d(Loss function)}}{d(slope)}}_{value_{slope}} \;\&\; \underbrace{\frac{\text{d(Loss function)}}{d(intercept)}}_{value_{intercept}}$$

2. Compute :

$$weighted\_average_{slope} = \boldsymbol{\beta}\, weighted\_average_{slope} + (1 - \boldsymbol{\beta}) value_{slope}$$

$$weighted\_average_{intercept} = \boldsymbol{\beta}\, weighted\_average_{intercept} + (1 - \boldsymbol{\beta}) value_{intercept}$$

3. Update :

$$slope_{new} = slope_{old} - (learning\_rate * weighted\_average_{slope})$$

$$intercept_{new} = intercept_{old} - (learning\_rate * weighted\_average_{intercept})$$

Optimizers

4 Gradient Descent with Momentum :

1. On each iteration, we compute the partial derivatives while plugging in the initial slope and intercept values

$$\underbrace{\frac{\text{d(Loss function)}}{d(slope)}}_{value_{slope}} \;\&\; \underbrace{\frac{\text{d(Loss function)}}{d(intercept)}}_{value_{intercept}}$$

2. Compute :

$$weighted\_average_{slope} = \boldsymbol{\beta}\, weighted\_average_{slope} + (1 - \boldsymbol{\beta}) value_{slope}$$

$$weighted\_average_{intercept} = \boldsymbol{\beta}\, weighted\_average_{intercept} + (1 - \boldsymbol{\beta}) value_{intercept}$$

$\boldsymbol{\beta}$ is a parameter that controls the weighted average (common value = 0.9)

Initial $weighted\_averages = 0$

3. Update :

$$slope_{new} = slope_{old} - (learning\_rate * weighted\_average_{slope})$$

$$intercept_{new} = intercept_{old} - (learning\_rate * weighted\_average_{intercept})$$

Optimizers

5

AdaGrad (Adaptive Gradients):

AdaGrad is a technique to change the learning rate over time

$$parameter_{new} = parameter_{old} - stepsize$$

where :

$$stepsize_{parameter_i} = \nabla L(parameter_i) * \frac{learning\ rate}{\sqrt{\epsilon + \sum_{i=1}^{t} (\nabla L(parameter_i))^2}}$$

and

$$\nabla L(parameter_i) = \frac{d(Loss\ function)}{d(parameter_i)}$$

Optimizers

5

AdaGrad (Adaptive Gradients):

AdaGrad is a technique to change the learning rate over time

$$parameter_{new} = parameter_{old} - stepsize$$

where :

$$stepsize_{parameter_i} = \nabla L(parameter_i) * \frac{learning\ rate}{\sqrt{\epsilon + \sum_{i=1}^{t} (\nabla L(parameter_i))^2}}$$

Sum over all the gradients from the first time step until the current one

With every new time step, a new gradient is added which causes the denominator to increase and the step size to decrease

and

$$\nabla L(parameter_i) = \frac{d(Loss\ function)}{d(parameter_i)}$$

Optimizers

5

AdaGrad (Adaptive Gradients):

AdaGrad is a technique to change the learning rate over time

$$parameter_{new} = parameter_{old} - stepsize$$

where :

$$stepsize_{parameter_i} = \nabla L(parameter_i) * \frac{learning\ rate}{\sqrt{\epsilon + \sum_{i=1}^{t} (\nabla L(parameter_i))^2}}$$

and

$$\nabla L(parameter_i) = \frac{d(Loss\ function)}{d(parameter_i)}$$

ϵ A small value to avoid division by 0

Optimizers

6

RMS Prop (Root Mean Squared Propagation) :

RMS Prop is very similar to AdaGrad

However, instead of using the sum of gradients, it uses the exponentially weighted average of the squared gradients.

→ Instead of being concerned about all of the gradients, we are more concerned about the most recent gradients.

Optimizers

6

RMS Prop (Root Mean Squared Propagation) :

RMS Prop is very similar to AdaGrad

AdaGrad : Learning rate decreases monotonously

RMS Prop : Learning rate can adapt to increase or decrease with every step

Optimizers

6

RMS Prop (Root Mean Squared Propagation) :

$$parameter_{new} = parameter_{old} - stepsize$$

where :

$$stepsize_{parameter_i} = \nabla L(parameter_i) * \frac{learning\ rate}{\sqrt{\epsilon + weighted\_average(\nabla L^2(parameter_i))}}$$

and

$$\nabla L(parameter_i) = \frac{d(Loss\ function)}{d(parameter_i)}$$

Default values :
Learning rate = 0.001
β = 0.9

Optimizers

7

Adam Optimizer (Adaptive Moment Estimation) :

Adam is another optimizer with adaptive learning rates for each parameter

Adam :

- uses the exponentially weighted average of the past squared gradients (like RMS Prop)
- uses an exponentially weighted average of past gradients (like GD with momentum)

Optimizers

7

Adam Optimizer (Adaptive Moment Estimation) :

$$parameter_{new} = parameter_{old} - stepsize$$

where :

$$stepsize_{parameter_i} = weighted\_average(\nabla L(parameter_i)) * \frac{learning\ rate}{\sqrt{\epsilon + weighted\_average(\nabla L^2(parameter_i))}}$$

and

$$\nabla L(parameter_i) = \frac{d(Loss\ function)}{d(parameter_i)}$$

Optimizers

7

Adam Optimizer (Adaptive Moment Estimation) :

Default values :

$\beta_1 = 0.9$

$\beta_2 = 0.999$

$\epsilon = 10^{-8}$

However, as the weighted averages are initialized to 0, they are biased towards 0 during the first iterations

➔ bias-corrected moments are calculated and used :

$$\widehat{m_i} = \frac{m_i}{1-\beta_1}$$

$$\widehat{v_i} = \frac{v_i}{1-\beta_2}$$

$$stepsize_{w_i} = \widehat{m}_i * \frac{learning\ rate}{\epsilon + \sqrt{\widehat{v_i}}}$$

$$m_i = \beta_1 m_{i-1} + (1-\beta_1) g_i$$

$$v_i = \beta_2 v_{i-1} + (1-\beta_2) g_i^2$$

$$w_{i+1} = w_i - \frac{\alpha \widehat{m_i}}{\epsilon + \sqrt{\widehat{v_i}}}$$

Optimizers

Bias Correction in Exponentially Weighted Average

- Bias correction makes the computation of the exponentially weighted averages more accurate
- Initialization values are taken = 0

➔ $V_t = \beta V_{t-1} + (1-\beta)\theta_t$

if $V_0 = 0$ & $\beta = 0.9 \Rightarrow V_1 = 0.1\ \theta_1$ This is not a good estimate of the first moving average !

- The effect of these very low values remains for several iterations
- In order to correct this bias :

$\widehat{V}_t = \frac{V_t}{1-\beta^t}$ where t = current iteration

➔ $\widehat{V}_1 = \frac{0.1\ \theta_1}{1-\beta}$

As t increases, β^t approaches 0

➔ The correction has less influence

Optimizers

How does Exponentially Weighted Average give more weight to the most recent observations ?

$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

$$V_{100} = 0.9V_{99} + 0.1\theta_{100}$$

$$V_{99} = 0.9V_{98} + 0.1\theta_{99}$$

$$V_{98} = 0.9V_{97} + 0.1\theta_{98}$$

➔ $V_{100} = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9(0.1\theta_{98} + 0.9V_{97}))$

➔ $V_{100} = 0.1\theta_{100} + 0.09\theta_{99} + 0.081\theta_{98} + 0.729V_{97}$

Optimizers

How does Exponentially Weighted Average give more weight to the most recent observations ?

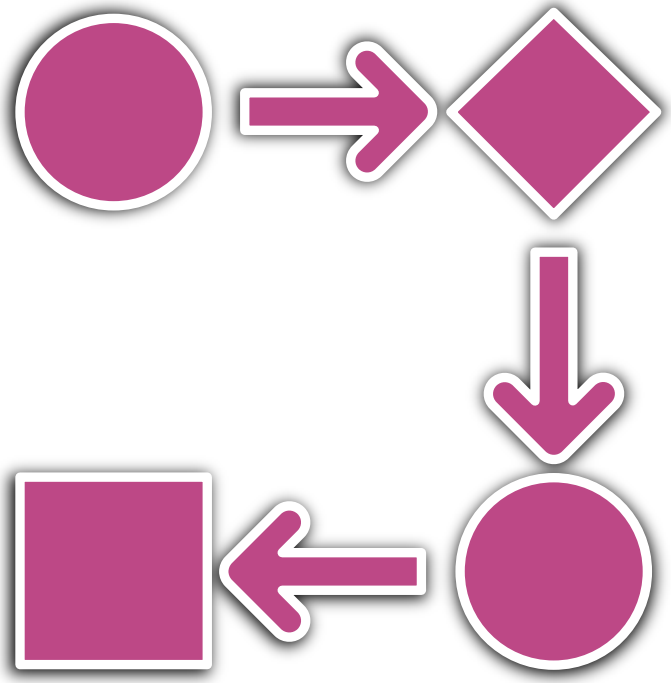
$$V_t = \beta V_{t-1} + (1-\beta)\theta_t$$

$$V_{100} = 0.9V_{99} + 0.1\theta_{100}$$
$$V_{99} = 0.9V_{98} + 0.1\theta_{99}$$
$$V_{98} = 0.9V_{97} + 0.1\theta_{98}$$

- The current observation will always have the highest coefficient
- The coefficient dimishes gradually further backwards

$$\rightarrow V_{100} = 0.1\theta_{100} + 0.9(0.1\theta_{99} + 0.9(0.1\theta_{98} + 0.9V_{97}))$$
$$\rightarrow V_{100} = 0.1\theta_{100} + 0.09\theta_{99} + 0.081\theta_{98} + 0.729V_{97}$$

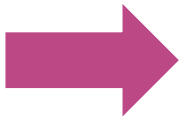
Backpropagation



Backpropagation

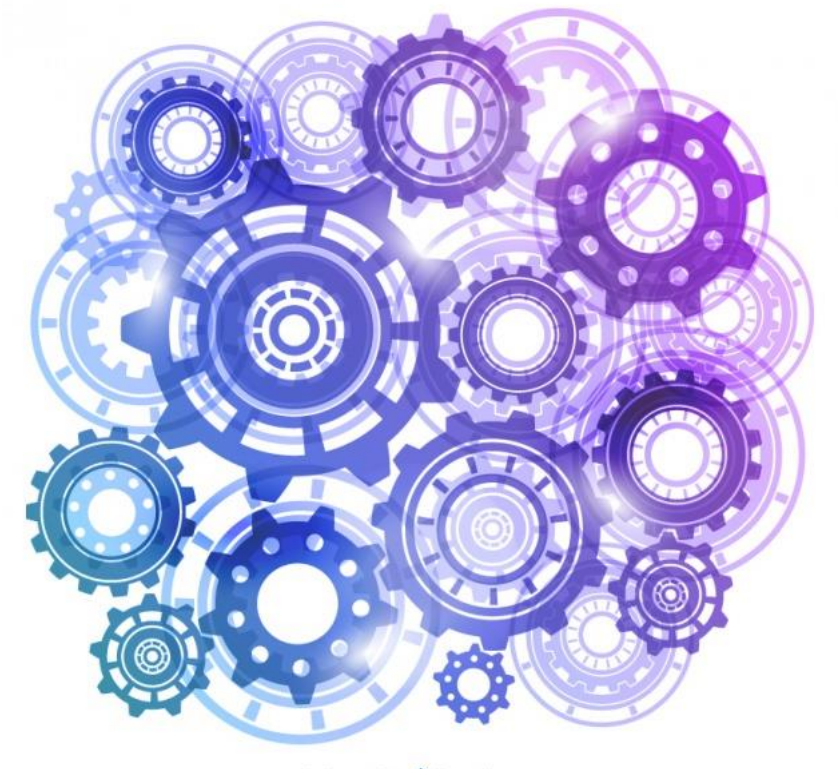
Backpropagation of error :

- An algorithm to calculate the gradient of a loss function relative to the model parameters
- Those gradients are then used by the optimizer to update the model weights
- Gradients are calculated backwards through the network starting at the output layer, one layer at a time



Together, backpropagation and Stochastic Gradient Descent (or variants) can be used to train a neural network.

The Learning Mechanism



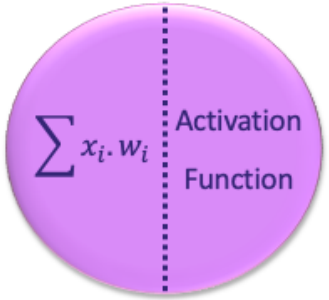
The Learning Mechanism

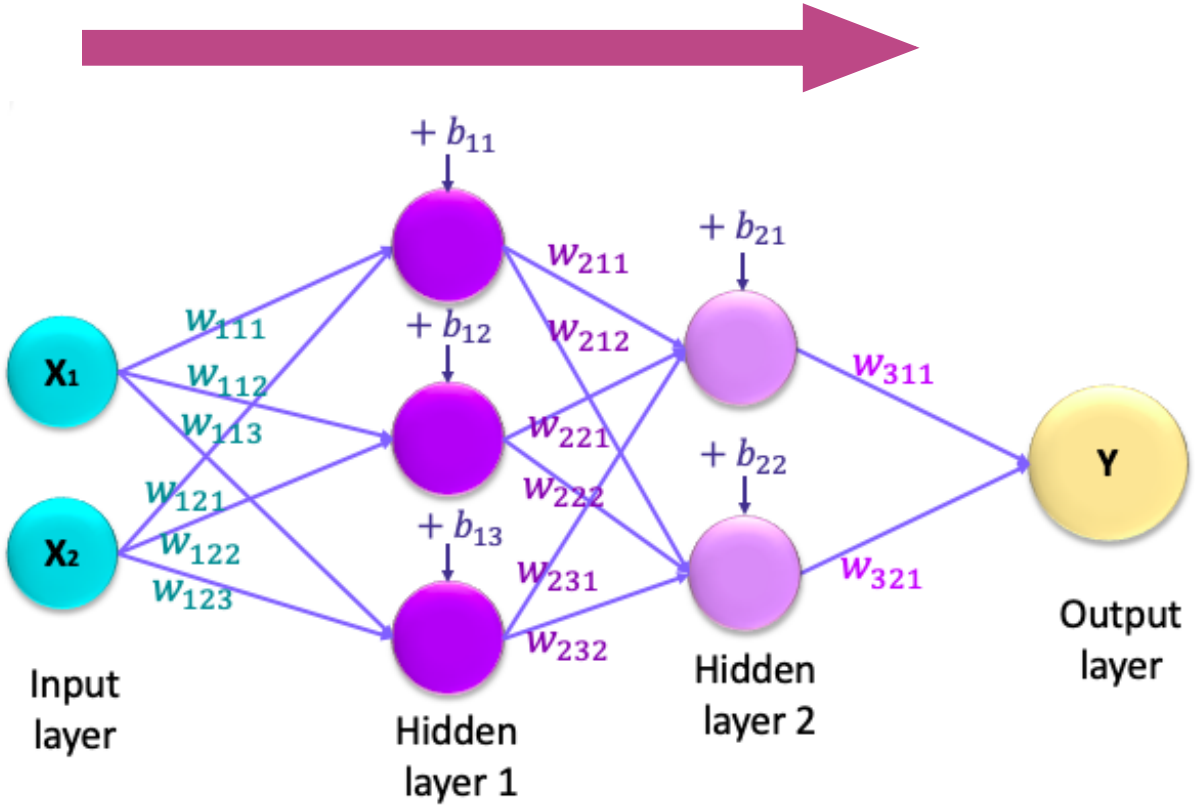
So, how does the network learn ?

Step 1 : Forward propagation:

At each neuron :

- ✦ Preactivation
- ✦ Activation



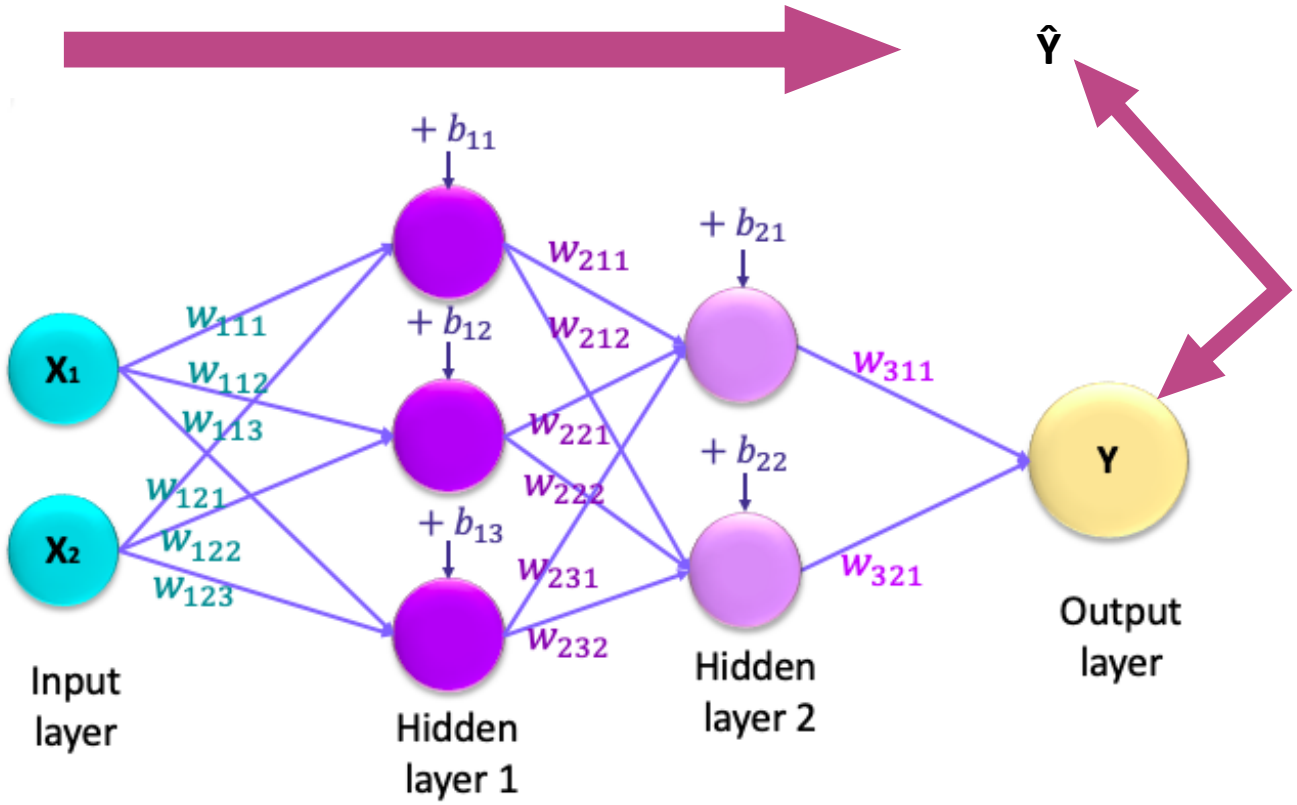


The Learning Mechanism

So, how does the network learn ?

Step 2 : Error calculation by the loss function :

The cost function measures the error between the trues values and the predicted values.



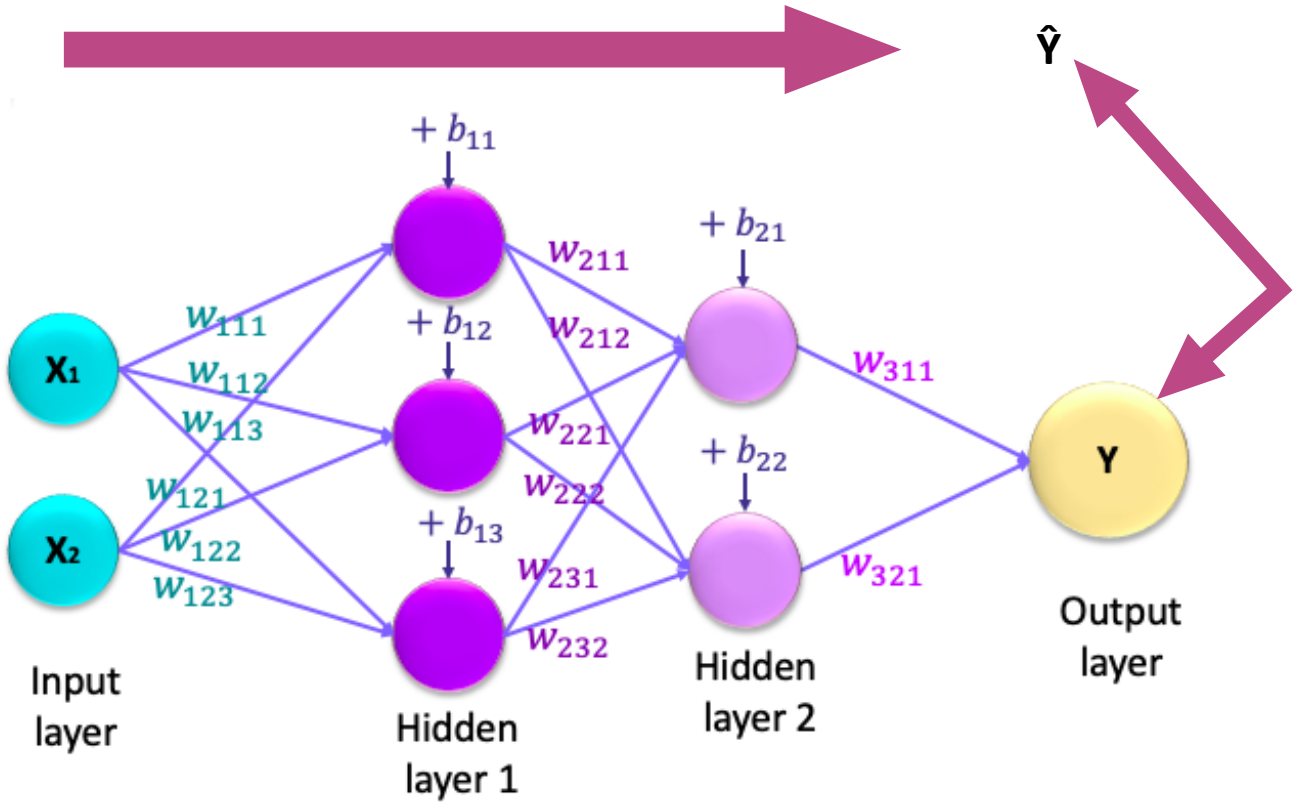
The Learning Mechanism

So, how does the network learn ?

Step 3 : Backpropagation and Optimisation :

✦ Backpropagation calculates the gradients of the loss function with respect to the model weights

✦ The optimizer uses the gradients to find new values for the weights that can reduce the loss



The Learning Mechanism

So, how does the network learn ?

Step 4 : Steps 1 to 3 are repeated until :

- ✦ The assigned number of iterations is reached
- ✦ The defined loss value is reached