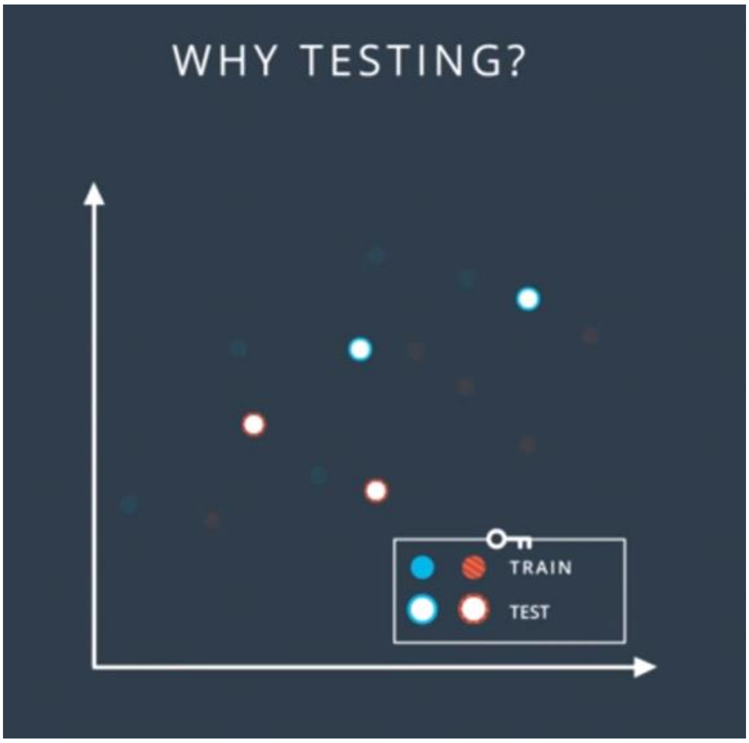
Nanodegree - NN - Lesson 4

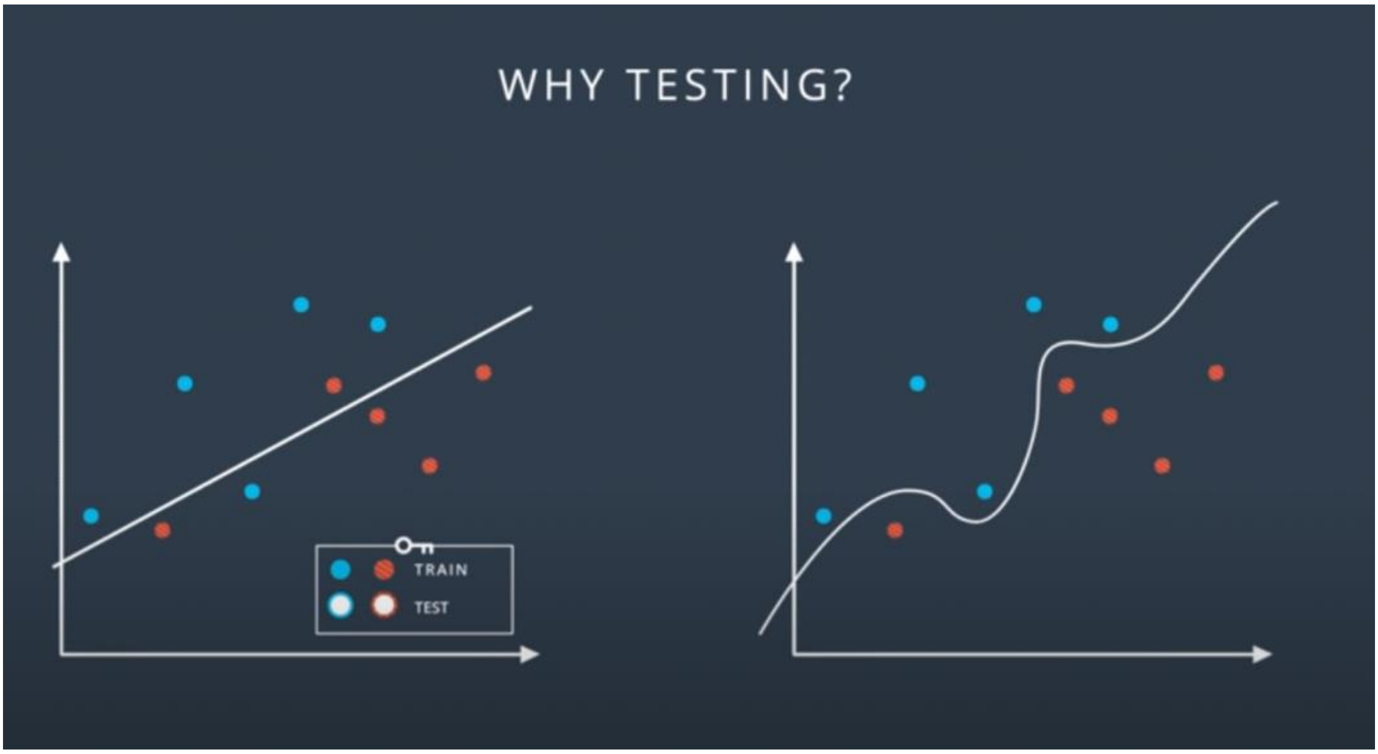
Sunday 24 January 2021 19:35

In this lesson we explored some technical optimisation for the Neural network.

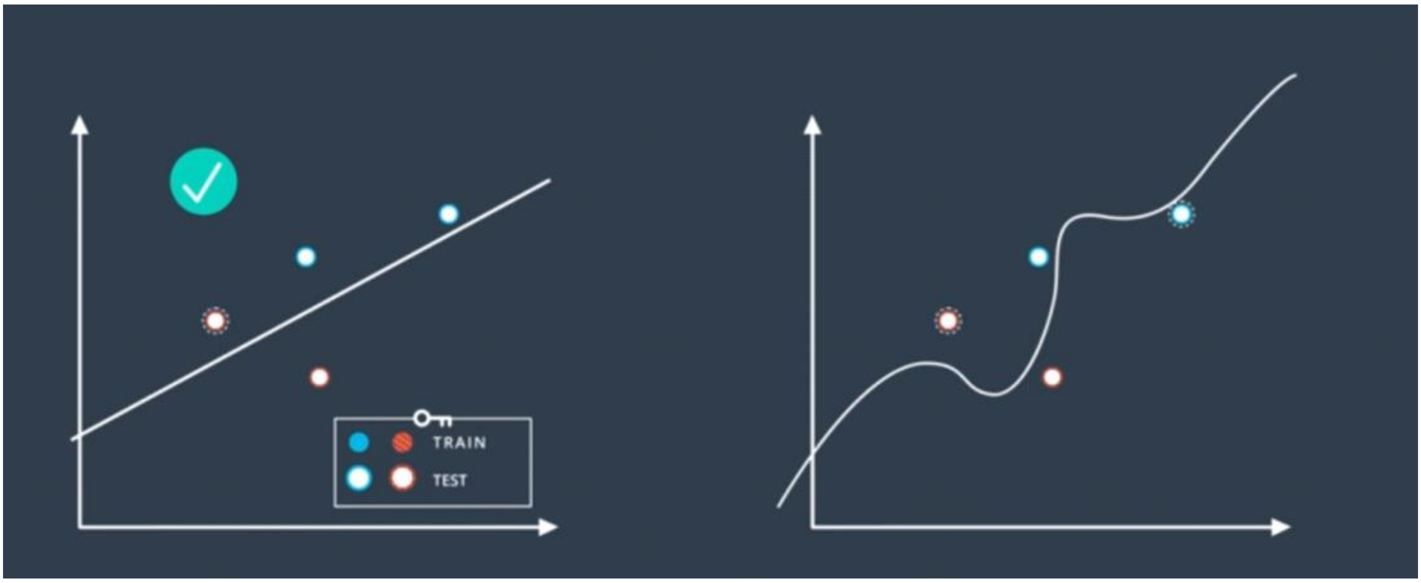
We introduced the concept of underfitting (model too simplistic) and overfitting (model too complicated) and we saw that in order to reduce these problems and allow our model to generalise well we can divide our data in training set and test set and check which model does better on the test set.



Screen clipping taken: 24/01/2021 19:35

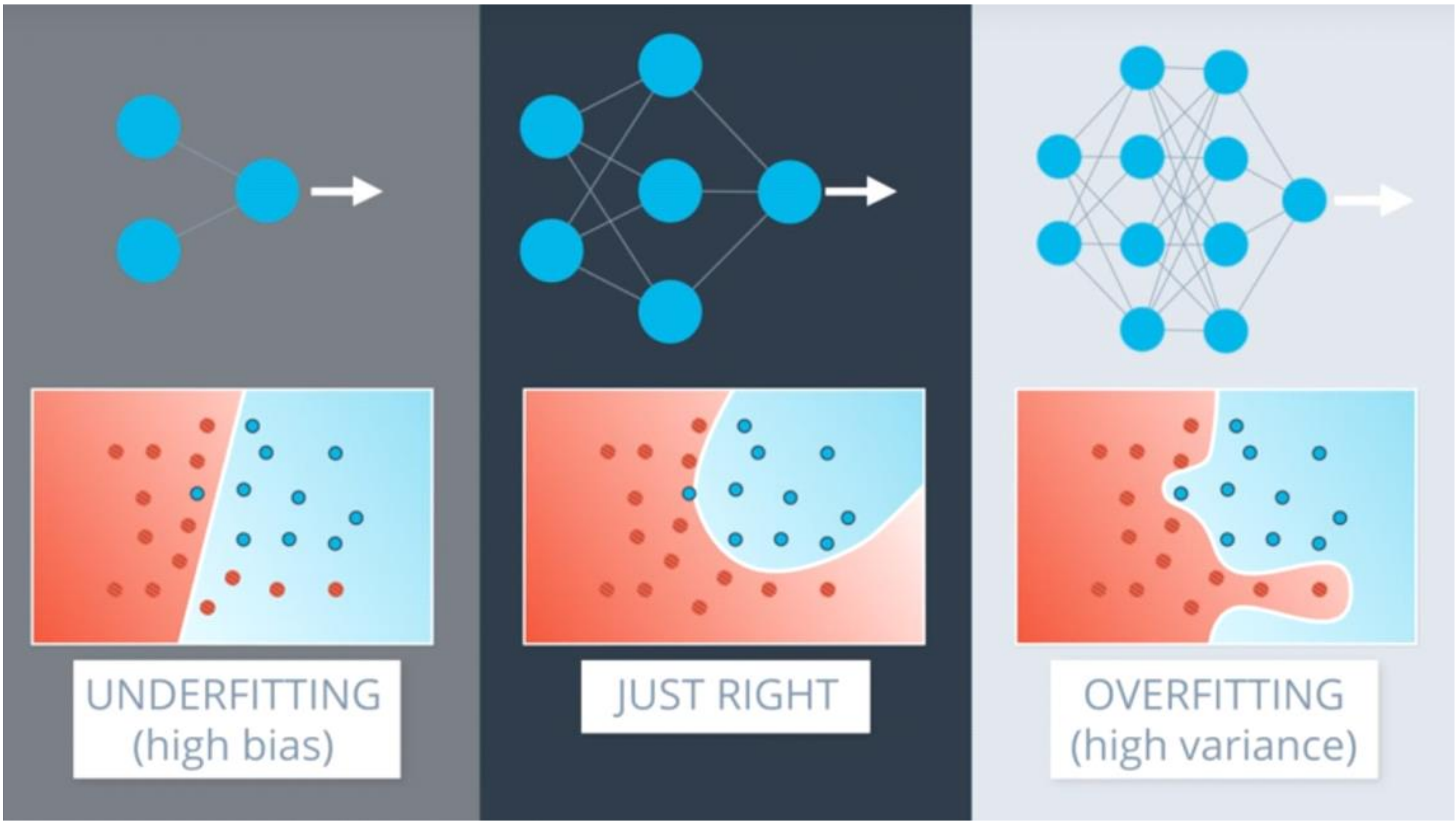


Screen clipping taken: 24/01/2021 19:35

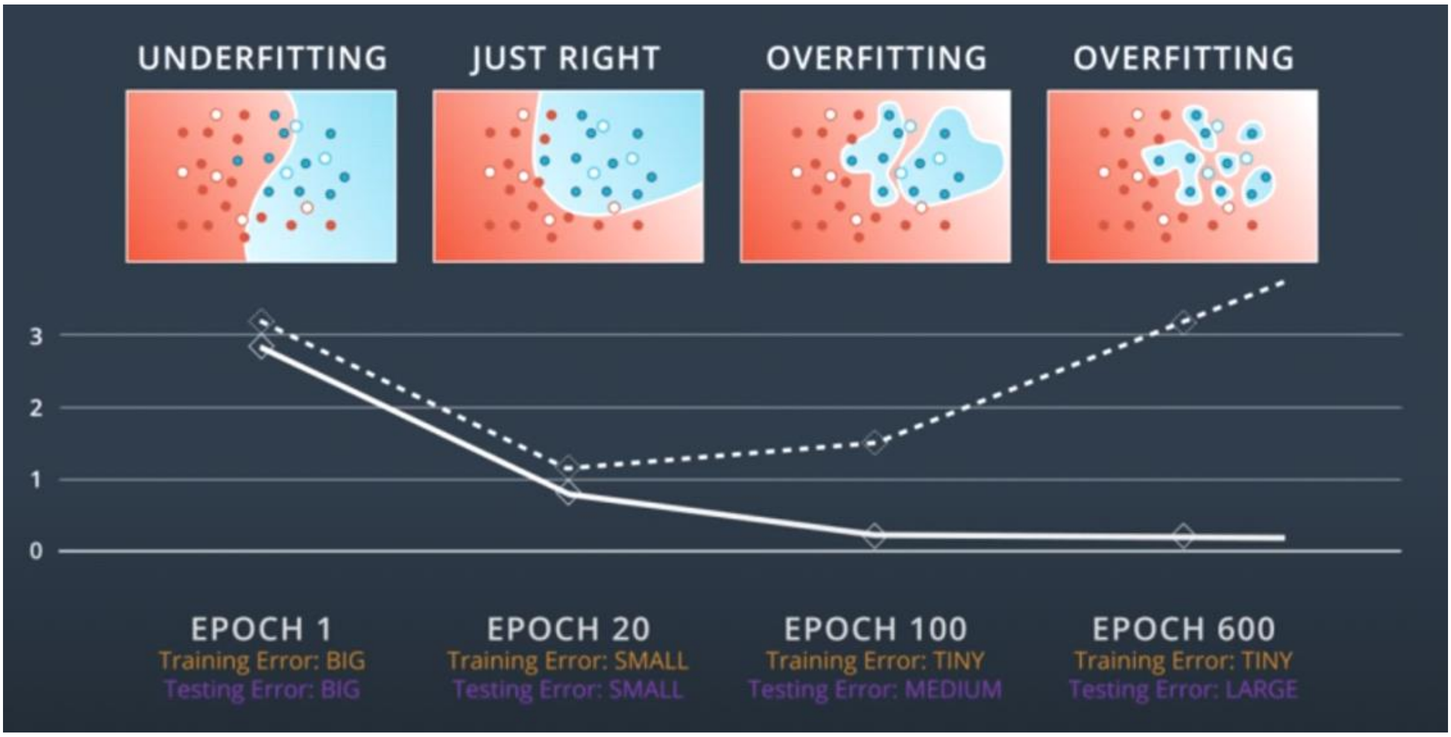


Screen clipping taken: 24/01/2021 19:36

Go for the simpler model - always .

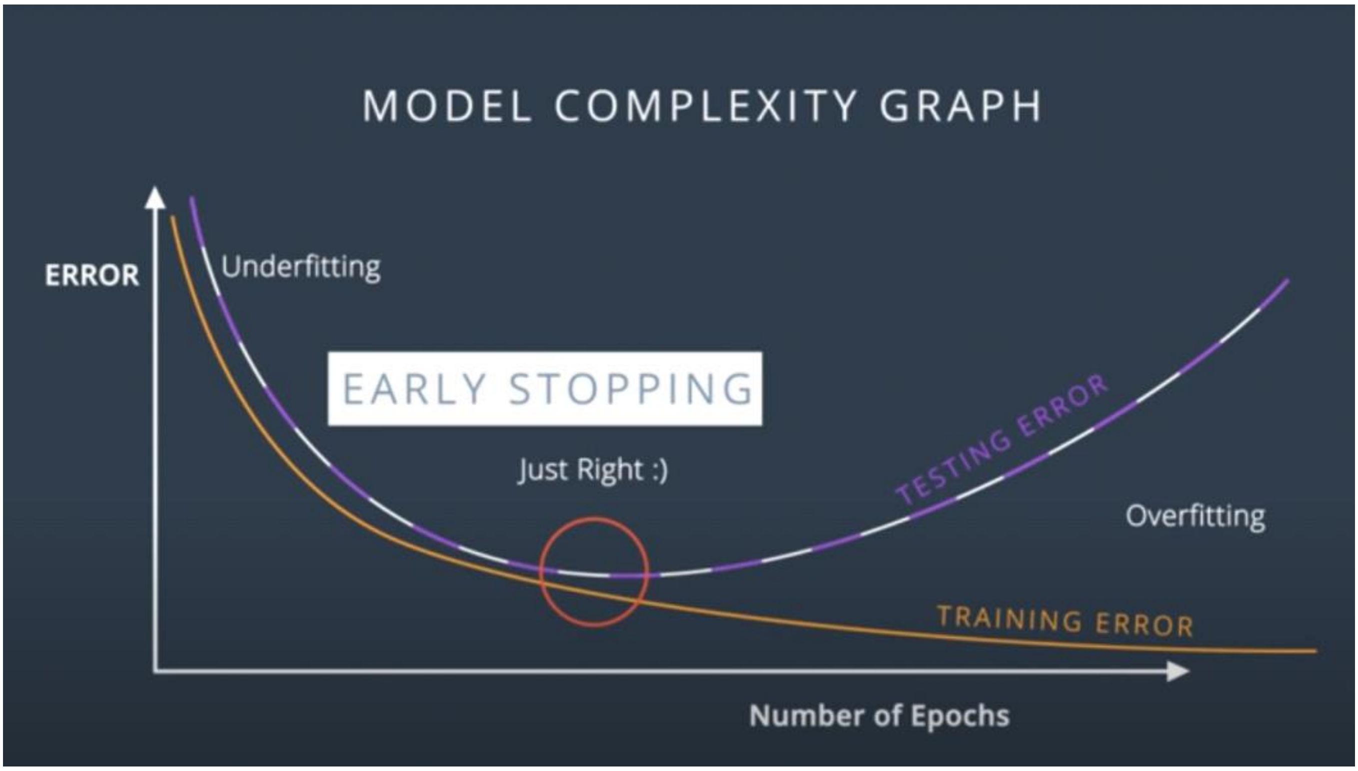


Screen clipping taken: 24/01/2021 19:40



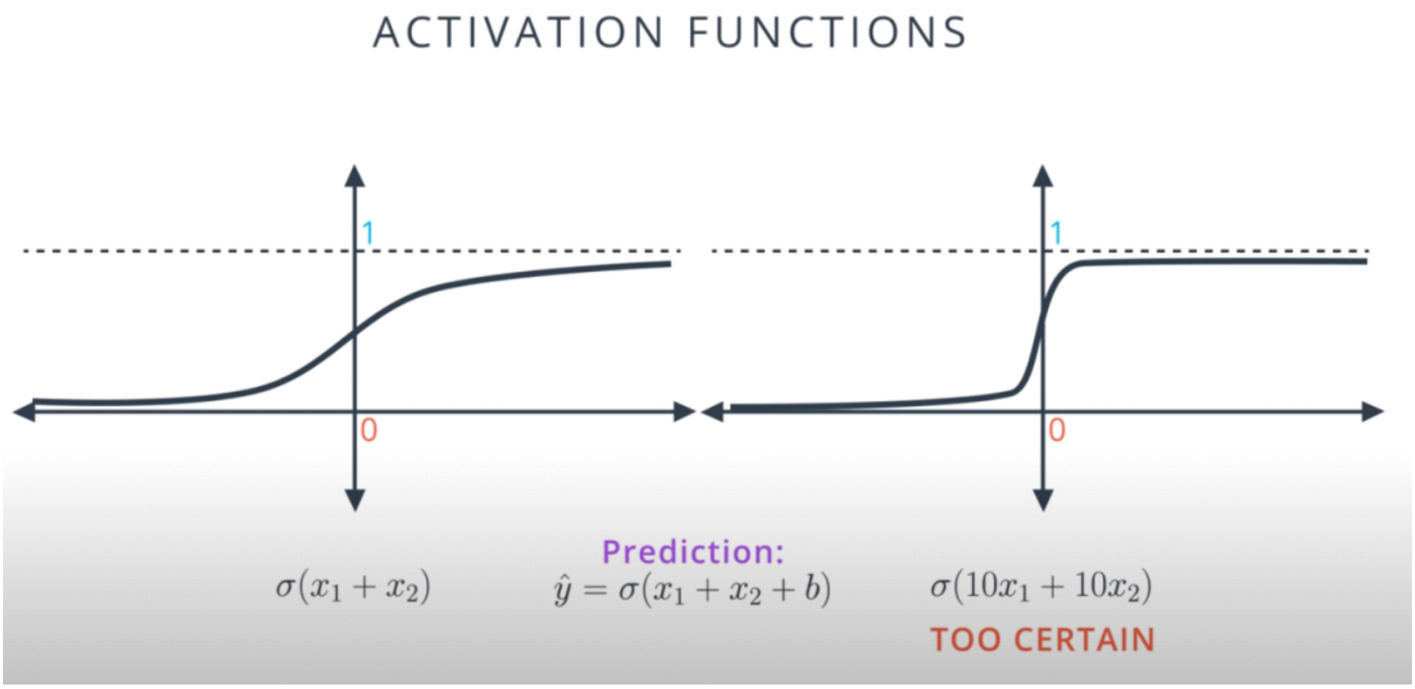
Screen clipping taken: 24/01/2021 19:44

The conclusion is that we stop where we see our test error that does not decrease anymore, so we select the 'just right' model that we hope generalises well.



Screen clipping taken: 24/01/2021 19:45

We have also another problem with the activation functions. There are functions that being more steep allow us to descend more and make the model quicker but the problem with steeper function is that the derivative is smaller, so we end up having products of small numbers that gives us small numbers in return.



ACTIVATION FUNCTIONS

$\sigma(x_1 + x_2)$

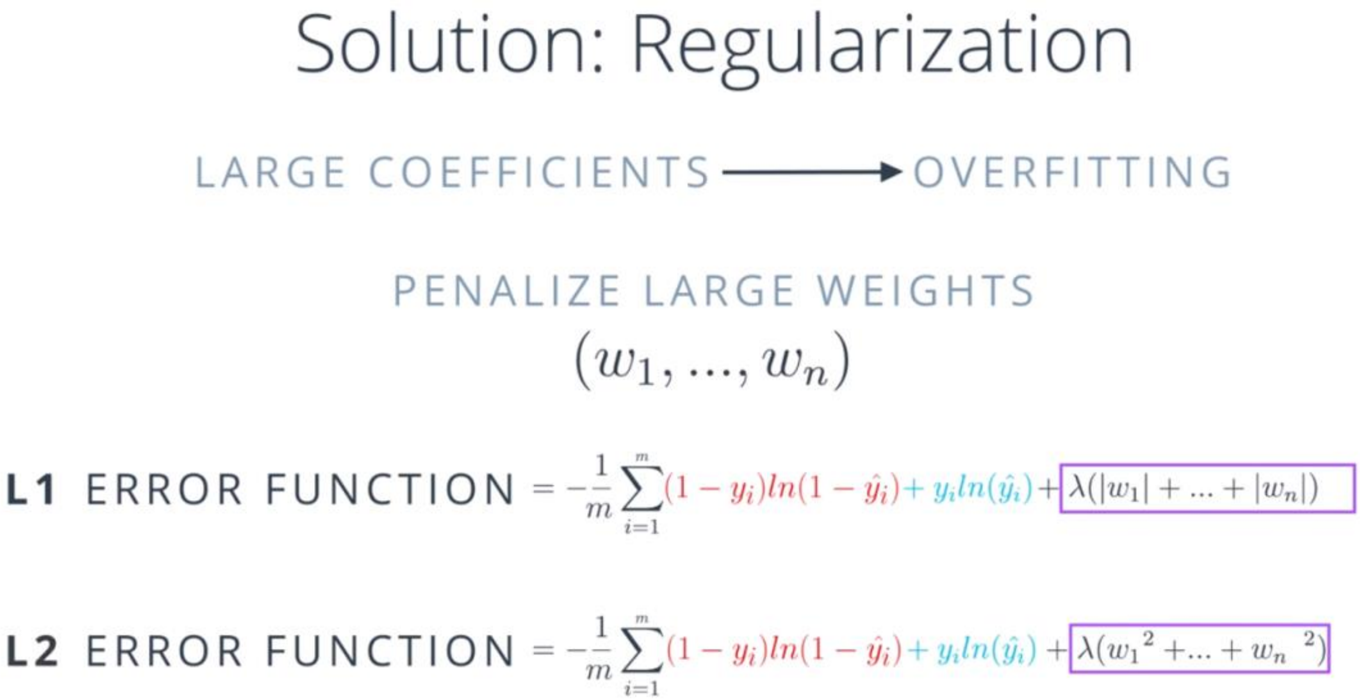
Prediction:
$\hat{y} = \sigma(x_1 + x_2 + b)$

$\sigma(10x_1 + 10x_2)$
TOO CERTAIN

Screen clipping taken: 24/01/2021 19:53

In order to tackle this problem we use regularisation. It is a way to penalise small coefficients that we do not want. We can do this in 2 ways with regularisation L1 and L2.

Regularisation L1 penises small coefficients adding in the optimization function the absolute value of the coefficients that we multiply by a penalisation parameter called lambda. While regularisation L2 instead penalises the small coefficients through the squared coefficients.



Solution: Regularization

LARGE COEFFICIENTS → OVERFITTING

PENALIZE LARGE WEIGHTS

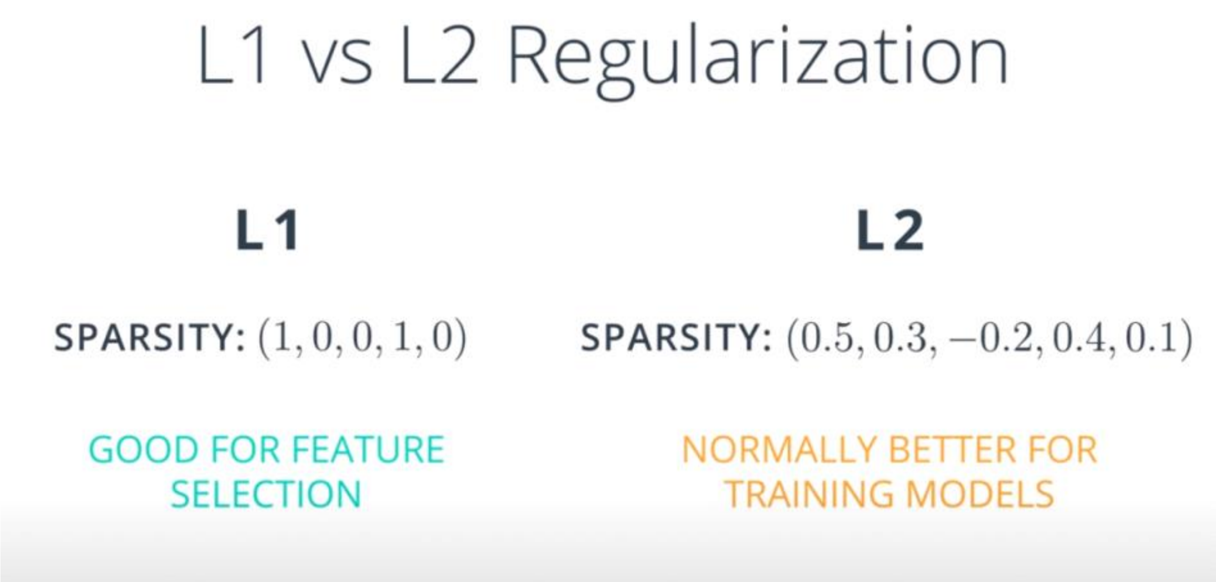
$(w_1, \ldots, w_n)$

L1 ERROR FUNCTION $= -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)ln(1-\hat{y}_i) + y_i ln(\hat{y}_i) + \boxed{\lambda(|w_1| + \ldots + |w_n|)}$

L2 ERROR FUNCTION $= -\frac{1}{m}\sum_{i=1}^{m}(1-y_i)ln(1-\hat{y}_i) + y_i ln(\hat{y}_i) + \boxed{\lambda({w_1}^2 + \ldots + {w_n}^2)}$

Screen clipping taken: 24/01/2021 19:55

There are differences in the solutions provided by these two methods. The solutions out of L1 are sparse, so it means that some coefficients tend to be 0. This is a good way to do feature selection, while the regularization L2 it is better to train models after we identified the input we want to take into account.



L1 vs L2 Regularization

L1

SPARSITY: $(1, 0, 0, 1, 0)$

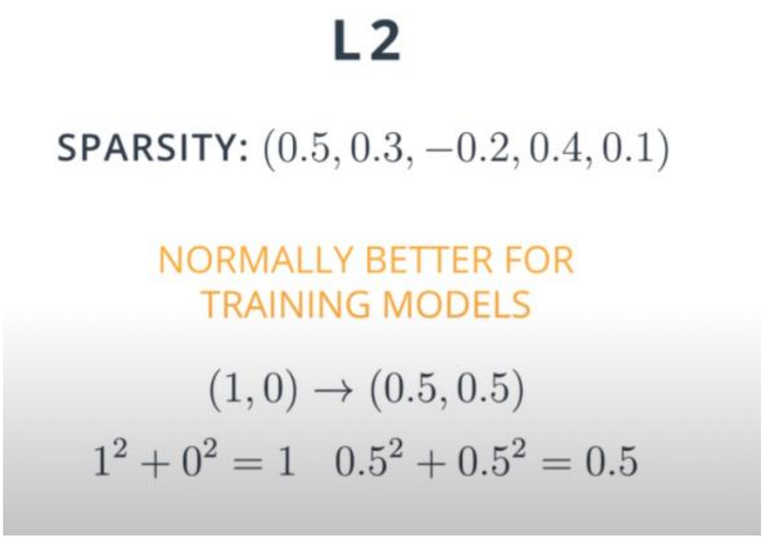
GOOD FOR FEATURE SELECTION

L2

SPARSITY: $(0.5, 0.3, -0.2, 0.4, 0.1)$

NORMALLY BETTER FOR TRAINING MODELS

Screen clipping taken: 24/01/2021 19:55



L2

SPARSITY: $(0.5, 0.3, -0.2, 0.4, 0.1)$

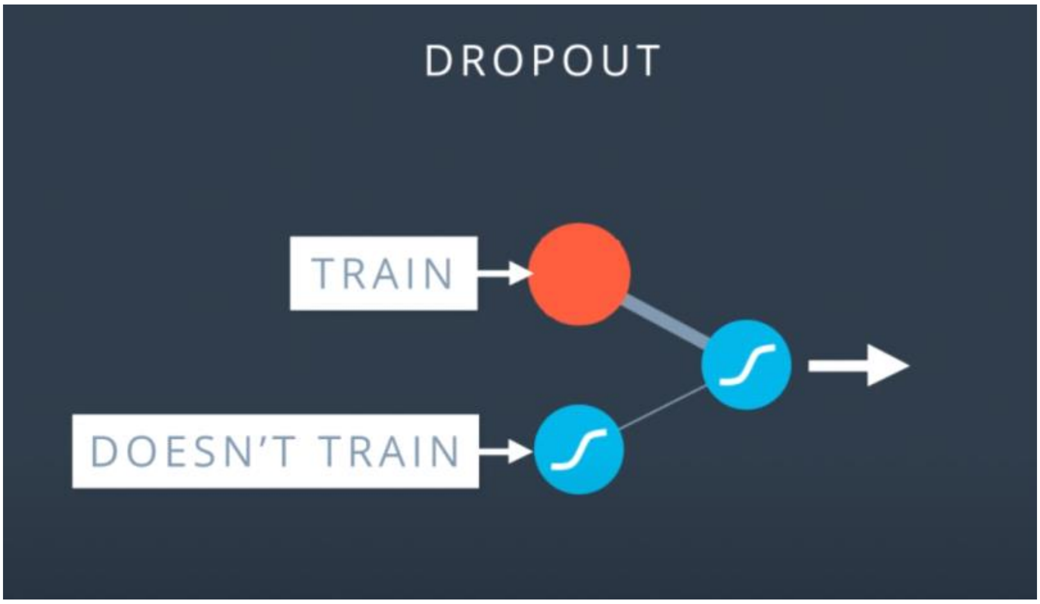
NORMALLY BETTER FOR TRAINING MODELS

$$(1, 0) \rightarrow (0.5, 0.5)$$

$$1^2 + 0^2 = 1 \quad 0.5^2 + 0.5^2 = 0.5$$

Screen clipping taken: 24/01/2021 19:57

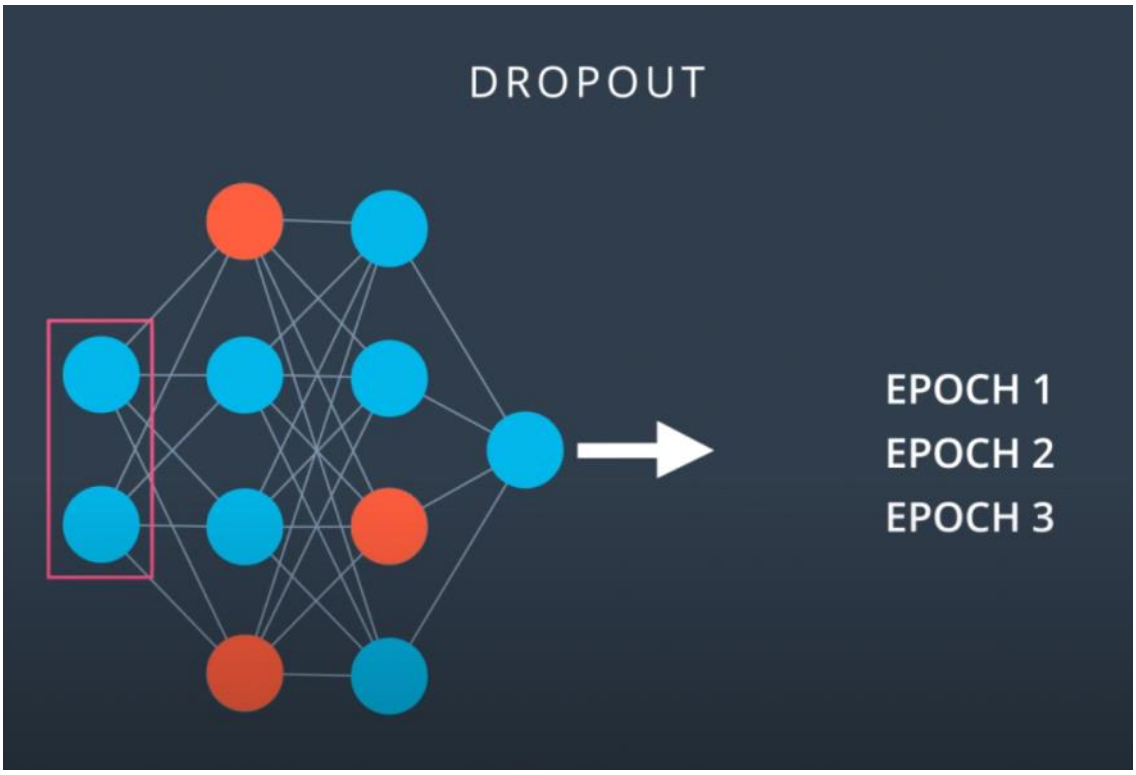
We notice in NN that the output is dominated by certain areas of the network, namely those nodes with higher waits dominates the output (and determines most of the error). In order to tackle this problem we use the DROPOUT. This means to switch off with a certain probability some nodes of the network, and train the model using the left out nodes.



DROPOUT

Screen clipping taken: 24/01/2021 19:59

The way it works is that at each epoch we switch off some of the nodes with a certain probability.

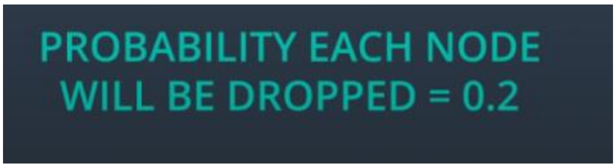


DROPOUT

EPOCH 1
EPOCH 2
EPOCH 3

Screen clipping taken: 24/01/2021 20:00

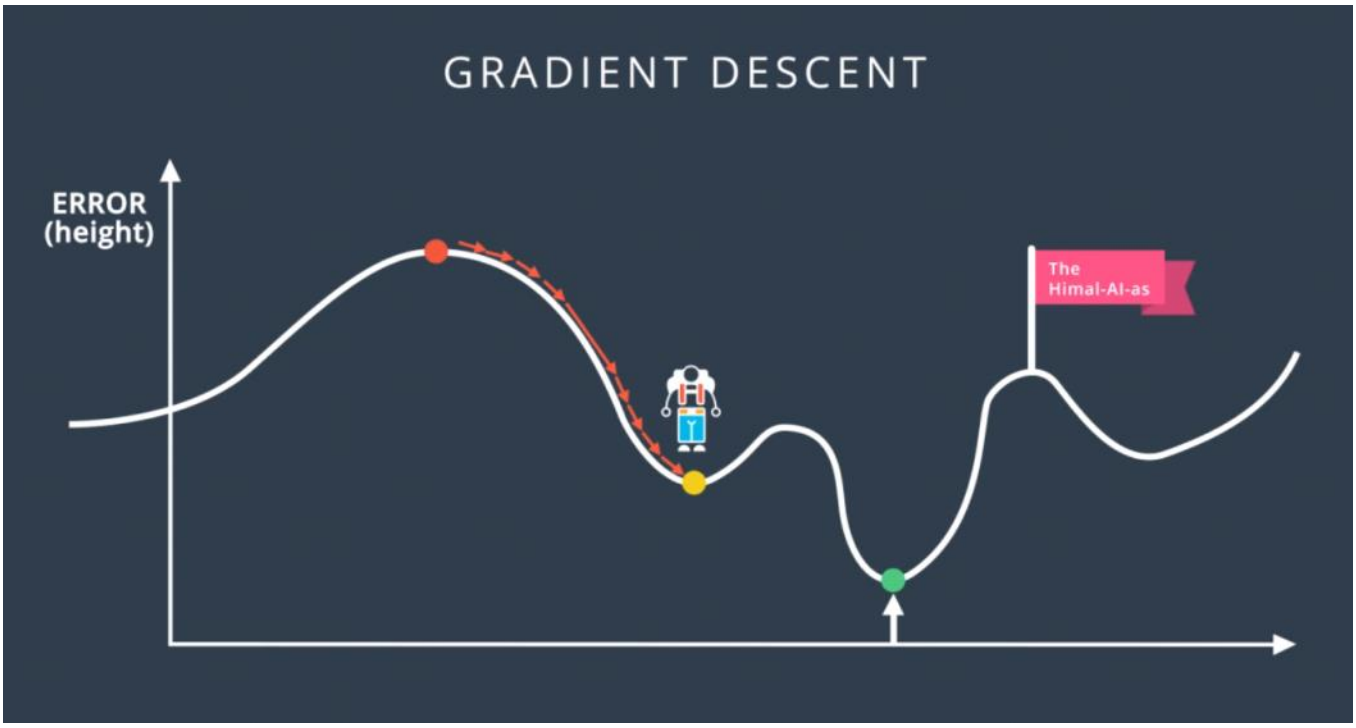
The probability by witch a node is switched off can be .2 (so with 1/5 probability each node can be switched off). Since we repeat the procedures certain number of times, we can be sure that most of the nodes will be switched off.
This should help our model to generalise better and not to have some nodes that take over the rest of the network.



PROBABILITY EACH NODE
WILL BE DROPPED = 0.2

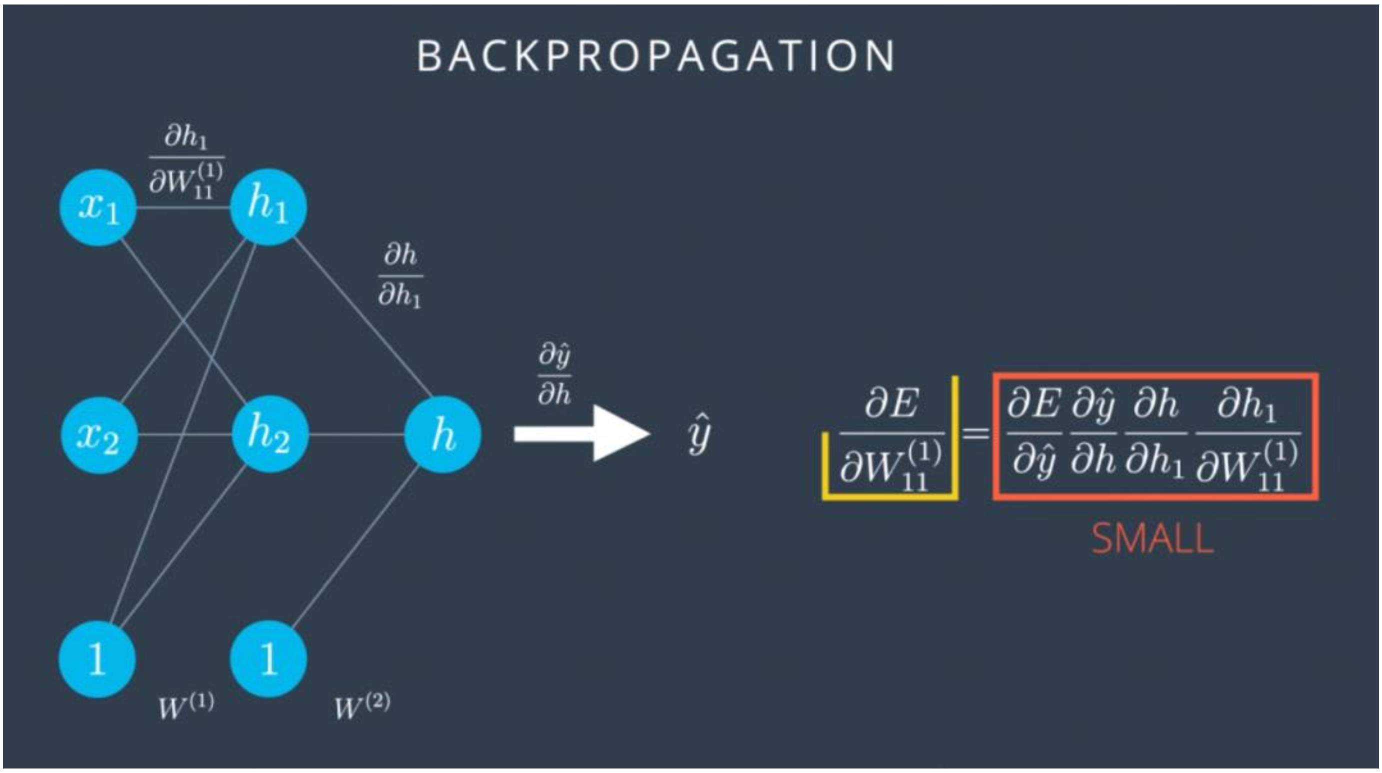
Screen clipping taken: 24/01/2021 20:00

When we apply gradient descend we can reach a local minimum and not the global minimum, so some optimisation space could be left over. When we reach a minimum (either global or local) the gradient will be zero, therefore we will not be able to move around and continue descending.



Screen clipping taken: 25/01/2021 09:10

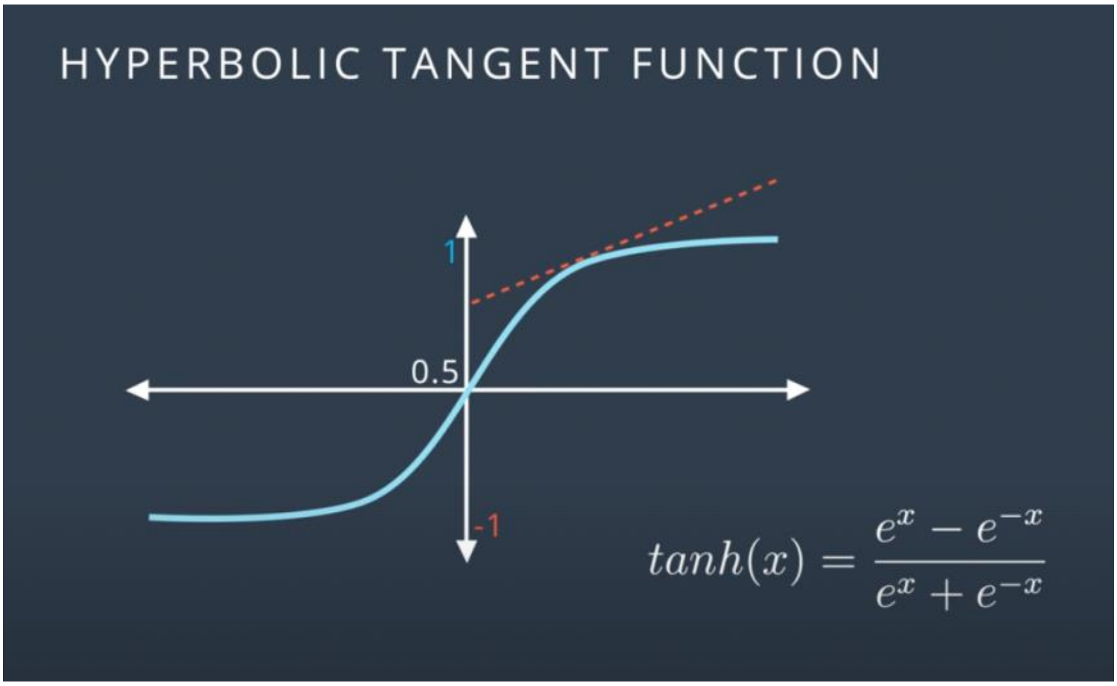
This is the backpropagation stage, when we calculate the error and we backpropagate it to the nodes in order to updates the scored. We notice that the derivative of the error is small because of the small coefficients.


BACKPROPAGATION
$\frac{\partial h_1}{\partial W_{11}^{(1)}}$
x_1
h_1
$\frac{\partial h}{\partial h_1}$
$\frac{\partial \hat{y}}{\partial h}$
x_2
h_2
h
$\hat{y}$
1
1
$W^{(1)}$
$W^{(2)}$
SMALL

$$\boxed{\frac{\partial E}{\partial W_{11}^{(1)}}} = \boxed{\frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} \frac{\partial h}{\partial h_1} \frac{\partial h_1}{\partial W_{11}^{(1)}}}$$

Screen clipping taken: 25/01/2021 09:11

The problem we saw before of the derivative of the error being small is due to the activation function we use. If we use the sigmoid function, for the nature of the function, we have small derivative. Instead of the sigmoid we can use other activation functions, like the tanh that gives us a bigger derivative at each node.

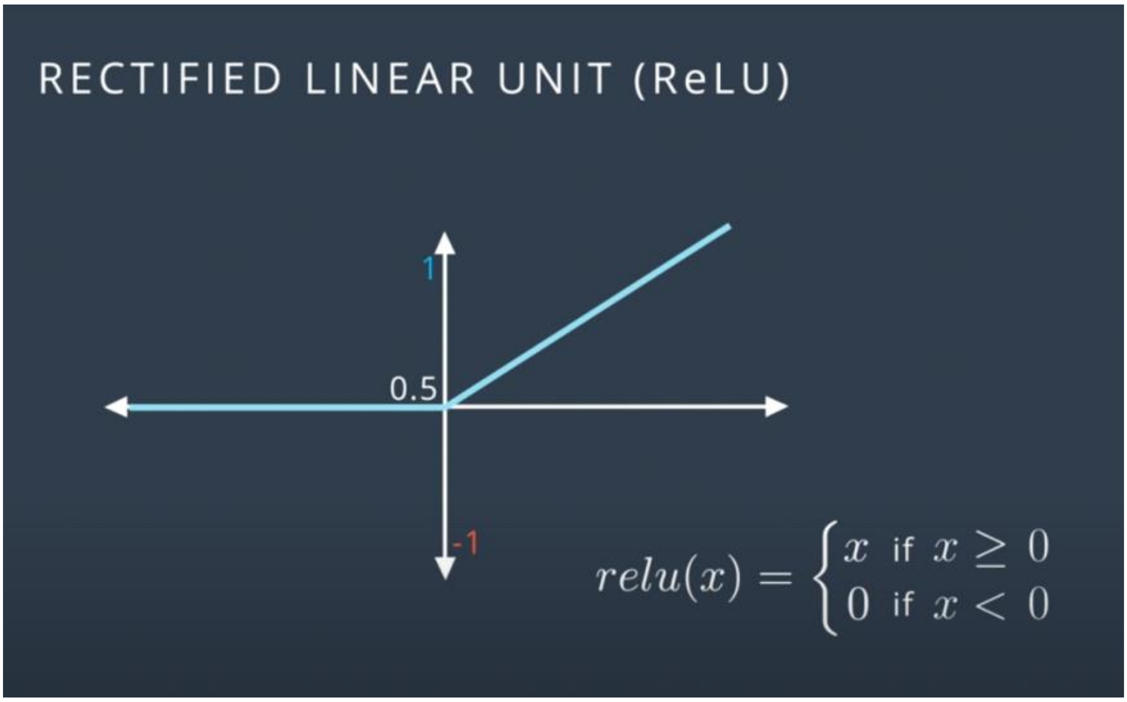


HYPERBOLIC TANGENT FUNCTION

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

Screen clipping taken: 25/01/2021 09:12

Another popular function is the ReLu - Rectified Linear Unit. This can be seen as the MAX function f = Max(0, x). So if the number is positive we have the number itself, otherwise 0. In this way we have big derivative (1) in case of positive numbers.

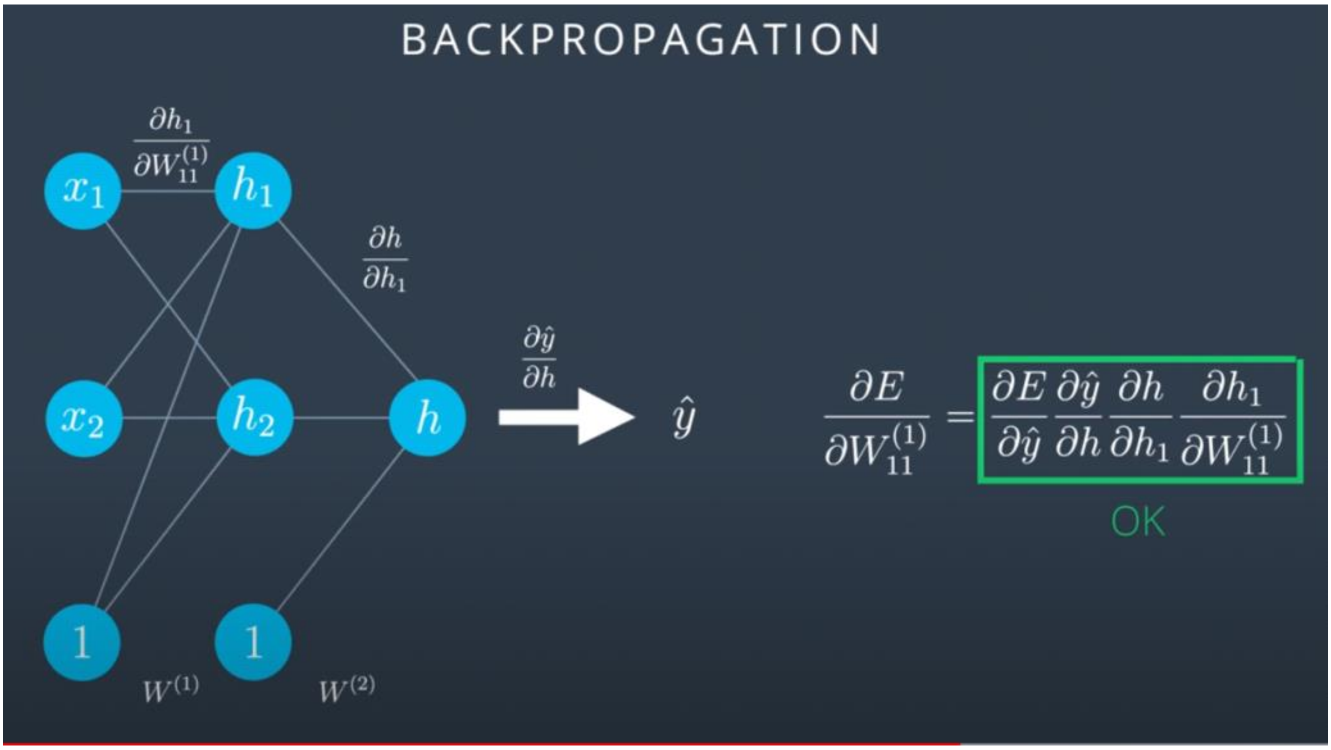


RECTIFIED LINEAR UNIT (ReLU)

$$relu(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

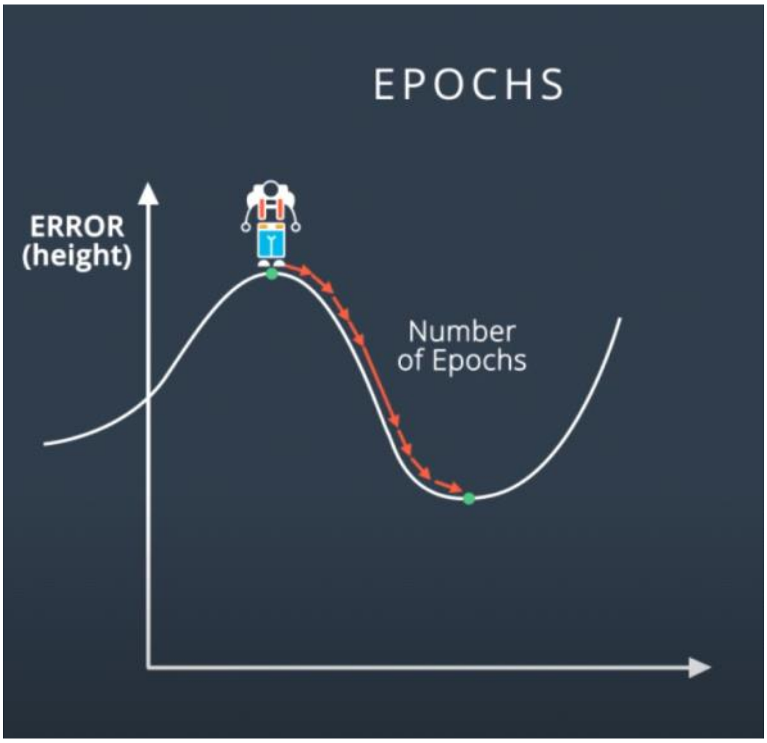
Screen clipping taken: 25/01/2021 09:12

Changing the activation functions we solve the problem of the derivative of the error being small. What we do is changing the activation functions at each node. We do not use sigmoid activation functions for all nodes but different functions, making sure that the last node has the sigmoid activation function since the output we want is a probability between 0 and 1.


BACKPROPAGATION
$\frac{\partial h_1}{\partial W_{11}^{(1)}}$
x_1
h_1
$\frac{\partial h}{\partial h_1}$
$\frac{\partial \hat{y}}{\partial h}$
x_2
h_2
h
$\hat{y}$
1
1
$W^{(1)}$
$W^{(2)}$
$\frac{\partial E}{\partial W_{11}^{(1)}} = \boxed{\frac{\partial E}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial h} \frac{\partial h}{\partial h_1} \frac{\partial h_1}{\partial W_{11}^{(1)}}}$
OK

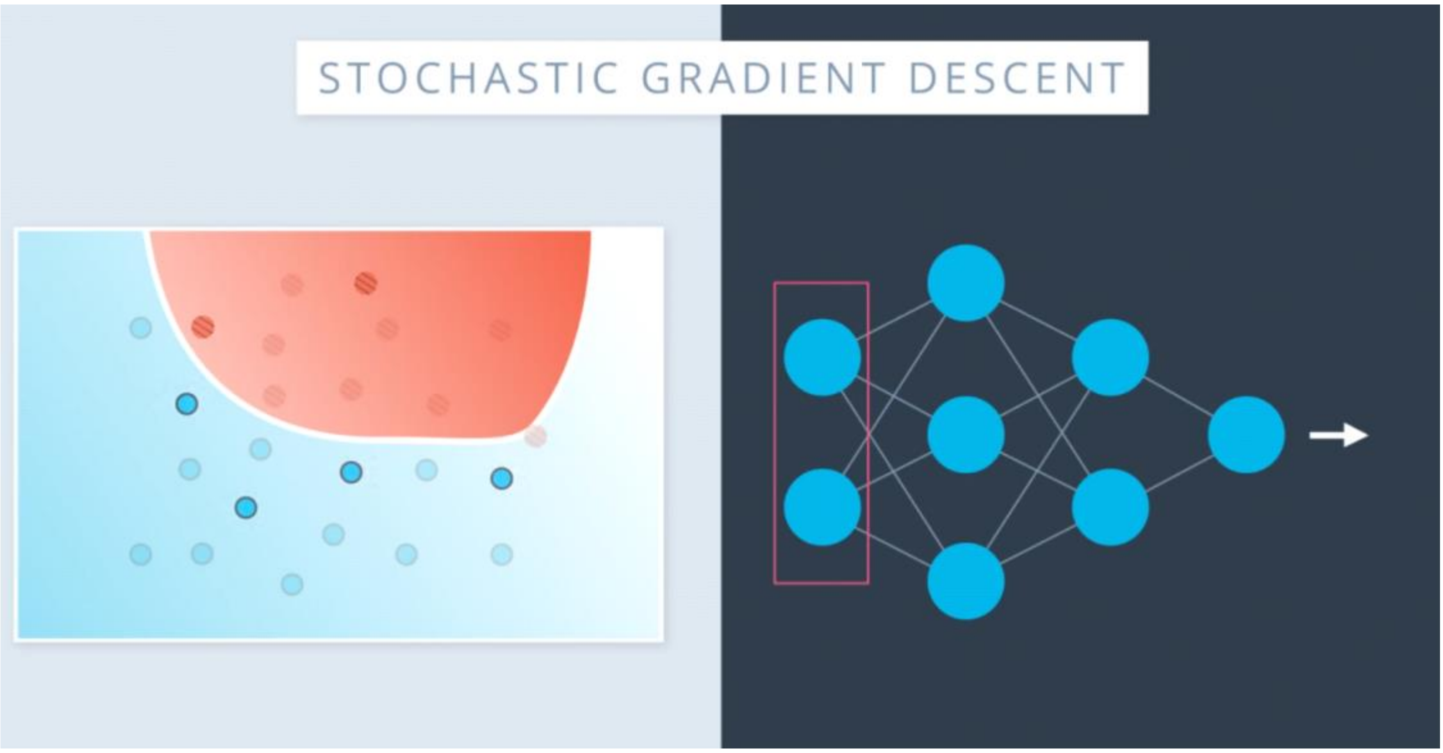
Screen clipping taken: 25/01/2021 09:13

The learning rate is another parameter that needs to be carefully chosen. If it is too big, the descend is faster but it might happen that the local minimum is passed and so the algorithm does not converge, if it is too small it might be slower. The rule of the thumb is when the algorithm does not work reduce the learning rate.



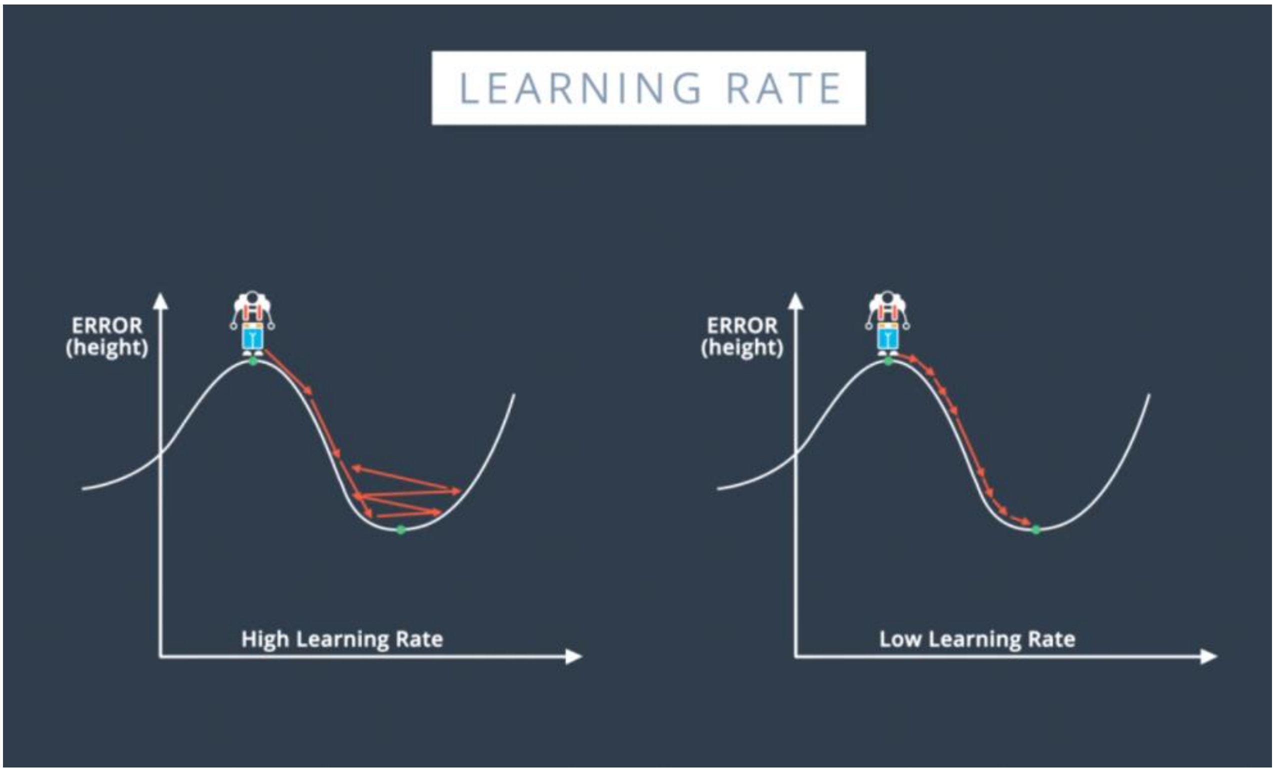
Screen clipping taken: 25/01/2021 09:14

It is not essential to use all the points together to train the algorithm because this might be very slow. So it is better to group the points into batches and run the algorithm on the smaller batches, so at each iteration we will have an adjustment. This might be not accurate but it is faster and works better.



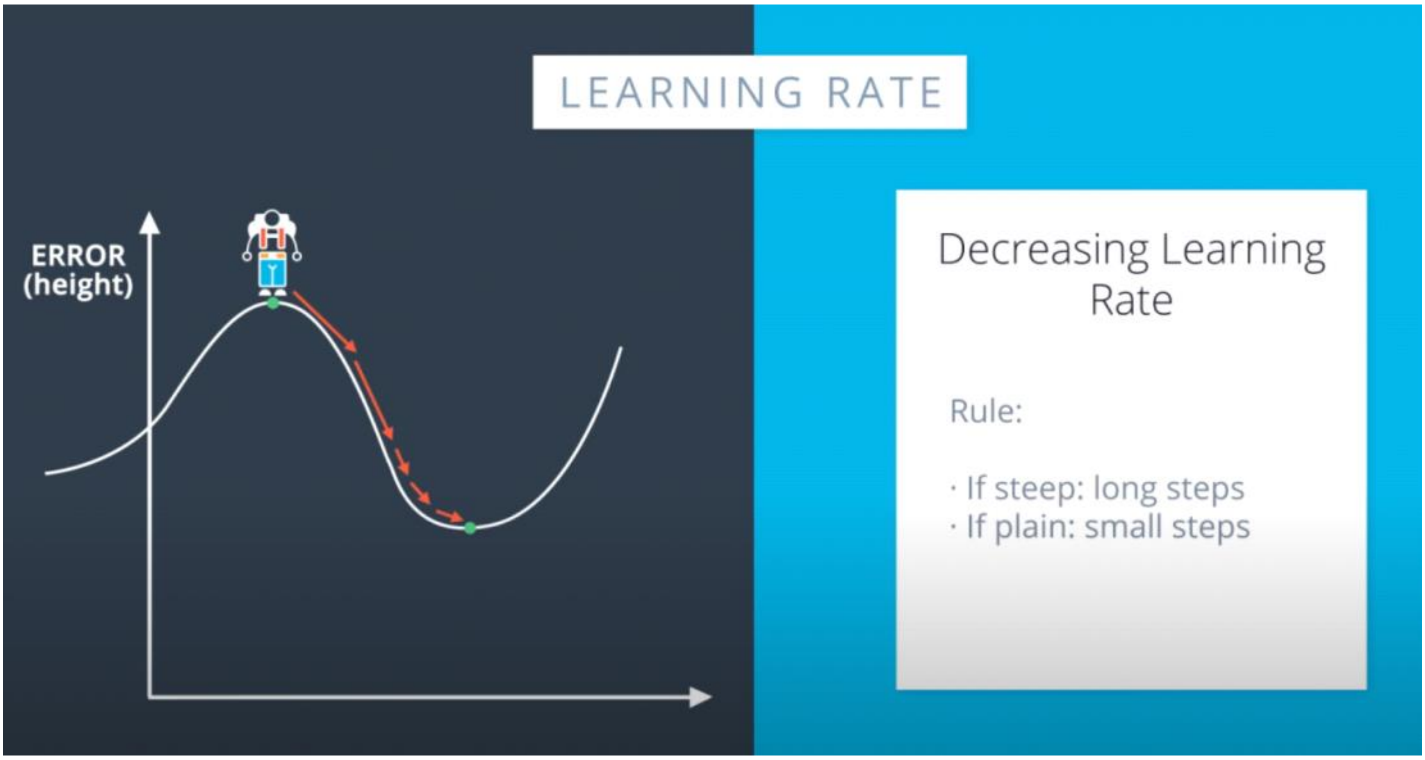
Screen clipping taken: 25/01/2021 09:17

The learning rate is another parameter that needs to be carefully chosen. If it is too big, the descend is faster but it might happen that the local minimum is passed and so the algorithm does not converge, if it is too small it might be slower. The rule of the thumb is when the algorithm does not work reduce the learning rate.



Screen clipping taken: 25/01/2021 09:18

The learning rate can be decreased according to the gradient, the bigger the gradient (steep) the larger the learning rate (so let's take longer steps) when instead the gradient becomes smaller we can decrease the learning rate in order to make sure we do not miss the minimum.



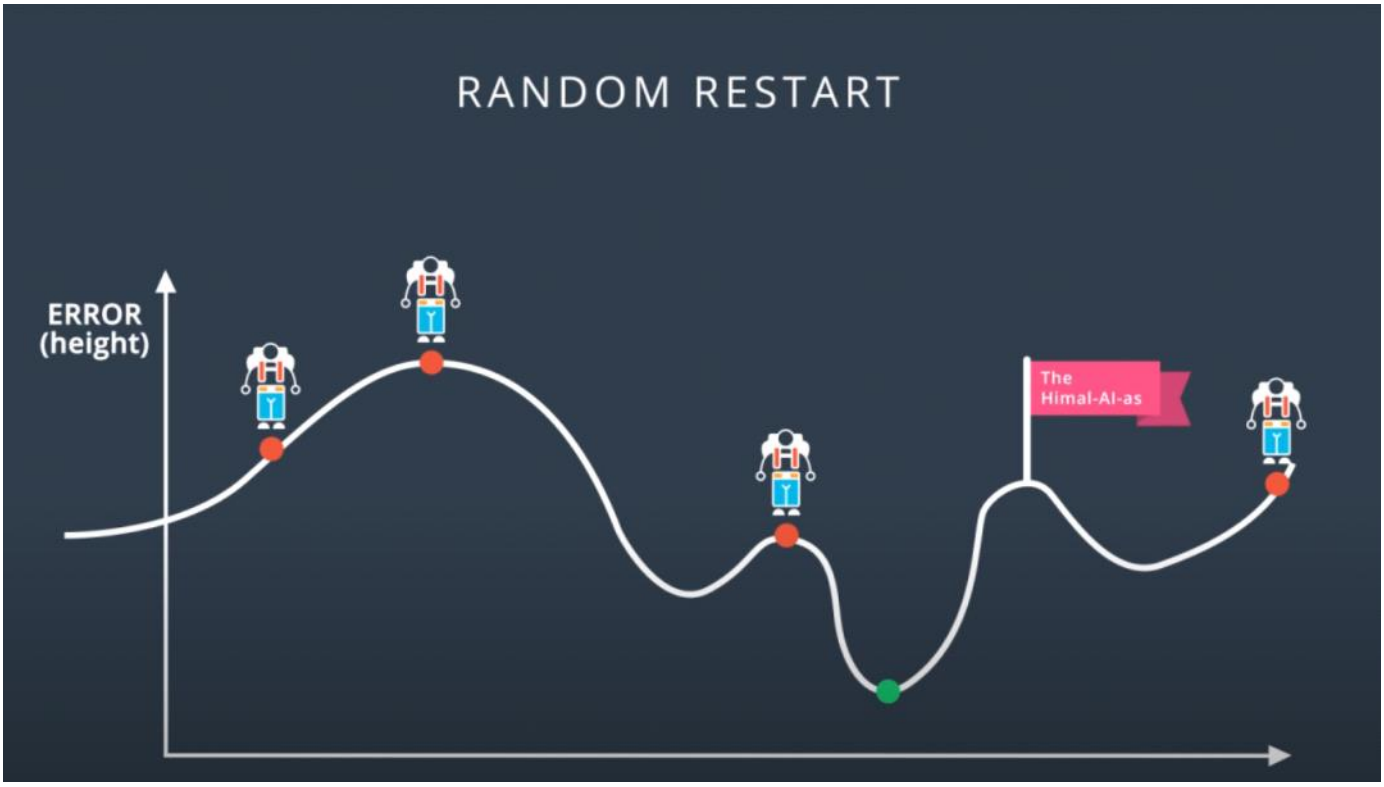
Decreasing Learning Rate

Rule:

- If steep: long steps
- If plain: small steps

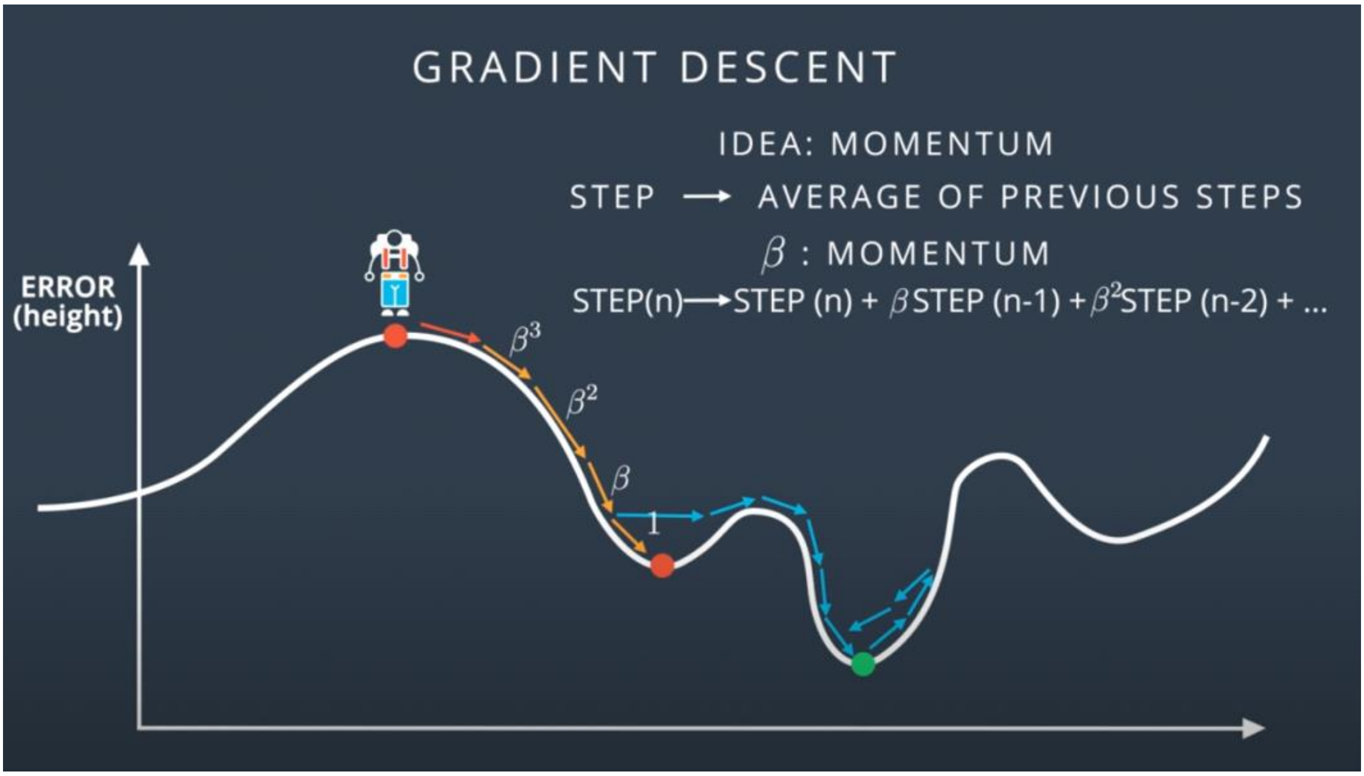
Screen clipping taken: 25/01/2021 09:18

To tackle the problem of local minimum we can restart the algorithm from different points in order to get to several minimums and hopefully one of them will be the global one.



Screen clipping taken: 25/01/2021 09:19

The other solution to tackle the local minimum problems is the momentum. Instead of determining the step at each epoch, we can take the step as an average of all the steps taken. This can be improved considering only the last N steps. The problem is that the most recent steps should have a higher weight, and this can be done with a momentum. A momentum is a coefficient between 0 and 1. Since numbers between 0 and 1 when squared become smaller, we will wait the most recent N steps with the N-powers of the momentum eg Beta * Step(n-1) + Beta^2 * Step(n-2) + … Beta^N * Step(n-N)



GRADIENT DESCENT

IDEA: MOMENTUM

STEP → AVERAGE OF PREVIOUS STEPS

β : MOMENTUM

STEP(n) → STEP (n) + βSTEP (n-1) + β^2STEP (n-2) + ...

Screen clipping taken: 25/01/2021 09:21