Nanodegree - NN - Lesson 3

Friday 22 January 2021 19:19

Gradient Descent with Squared Errors

We want to find the weights for our neural networks. Let's start by thinking about the goal. The network needs to make predictions as close as possible to the real values. To measure this, we use a metric of how wrong the predictions are, the **error**. A common metric is the sum of the squared errors (SSE):

$$E = \frac{1}{2}\sum_{\mu}\sum_{j}\left[y_j^{\mu} - \hat{y}_j^{\mu}\right]^2$$

where $\hat{y}$ is the prediction and y is the true value, and you take the sum over all output units j and another sum over all data points μ. This might seem like a really complicated equation at first, but it's fairly simple once you understand the symbols and can say what's going on in words.

First, the inside sum over j. This variable j represents the output units of the network. So this inside sum is saying for each output unit, find the difference between the true value y and the predicted value from the network $\hat{y}$, then square the difference, then sum up all those squares.

Then the other sum over μ is a sum over all the data points. So, for each data point you calculate the inner sum of the squared differences for each output unit. Then you sum up those squared differences for each data point. That gives you the overall error for all the output predictions for all the data points.

The SSE is a good choice for a few reasons. The square ensures the error is always positive and larger errors are penalized more than smaller errors. Also, it makes the math nice, always a plus.

Remember that the output of a neural network, the prediction, depends on the weights

$$\hat{y}_j^{\mu} = f\left(\sum_i w_{ij} x_i^{\mu}\right)$$

and accordingly the error depends on the weights

$$E = \frac{1}{2}\sum_{\mu}\sum_{j}\left[y_j^{\mu} - f\left(\sum_i w_{ij} x_i^{\mu}\right)\right]^2$$

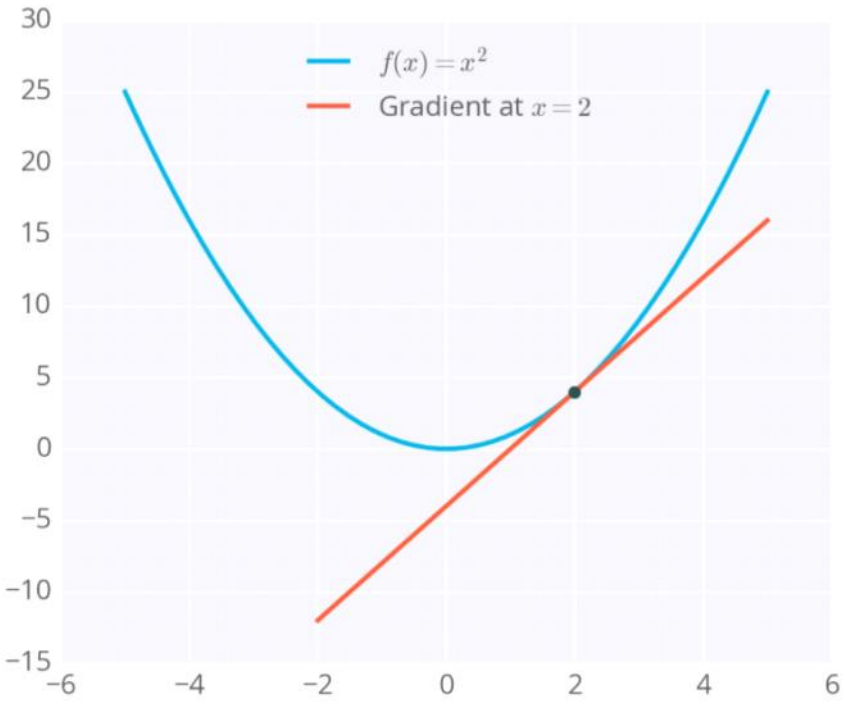
We want the network's prediction error to be as small as possible and the weights are the knobs we can use to make that happen. Our goal is to find weights w_{ij} that minimize the squared error E. To do this with a neural network, typically you'd use **gradient descent**.

As Luis said, with gradient descent, we take multiple small steps towards our goal. In this case, we want to change the weights in steps that reduce the error. Continuing the analogy, the error is our mountain and we want to get to the bottom. Since the fastest way down a mountain is in the steepest direction, the steps taken should be in the direction that minimizes the error the most. We can find this direction by calculating the *gradient* of the squared error.

Gradient is another term for rate of change or slope. If you need to brush up on this concept, check out Khan Academy's **great lectures** on the topic.

To calculate a rate of change, we turn to calculus, specifically derivatives. A derivative of a function $f(x)$ gives you another function $f'(x)$ that returns the slope of $f(x)$ at point x. For example, consider $f(x) = x^2$. The derivative of x^2 is $f'(x) = 2x$. So, at $x = 2$, the slope is $f'(2) = 4$. Plotting this out, it looks like:

Link:
https://www.khanacademy.org/math/multivariable-calculus/multivariable-derivatives/gradient-and-directional-derivatives/v/gradient

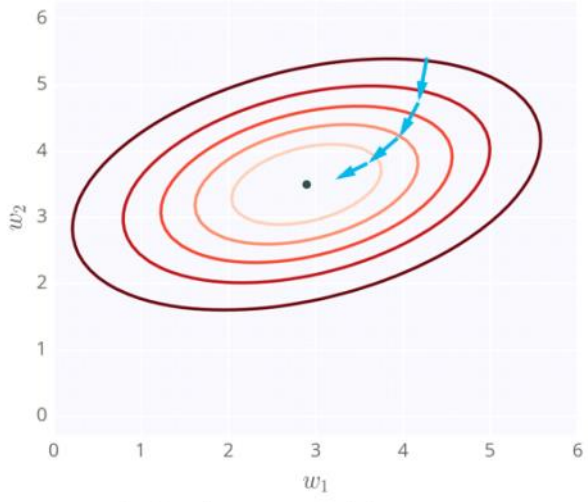


Example of a gradient

The gradient is just a derivative generalized to functions with more than one variable. We can use calculus to find the gradient at any point in our error function, which depends on the input weights. You'll see how the gradient descent step is derived on the next page.

Below I've plotted an example of the error of a neural network with two inputs, and accordingly, two weights. You can read this like a topographical map where points on a contour line have the same error and darker contour lines correspond to larger errors.

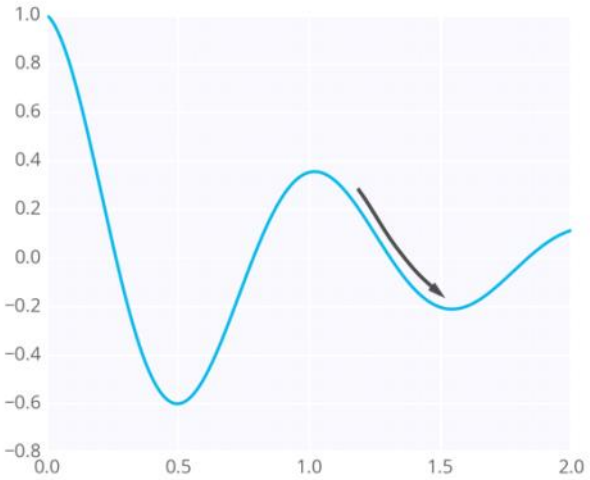
At each step, you calculate the error and the gradient, then use those to determine how much to change each weight. Repeating this process will eventually find weights that are close to the minimum of the error function, the black dot in the middle.



Gradient descent steps to the lowest error

Caveats

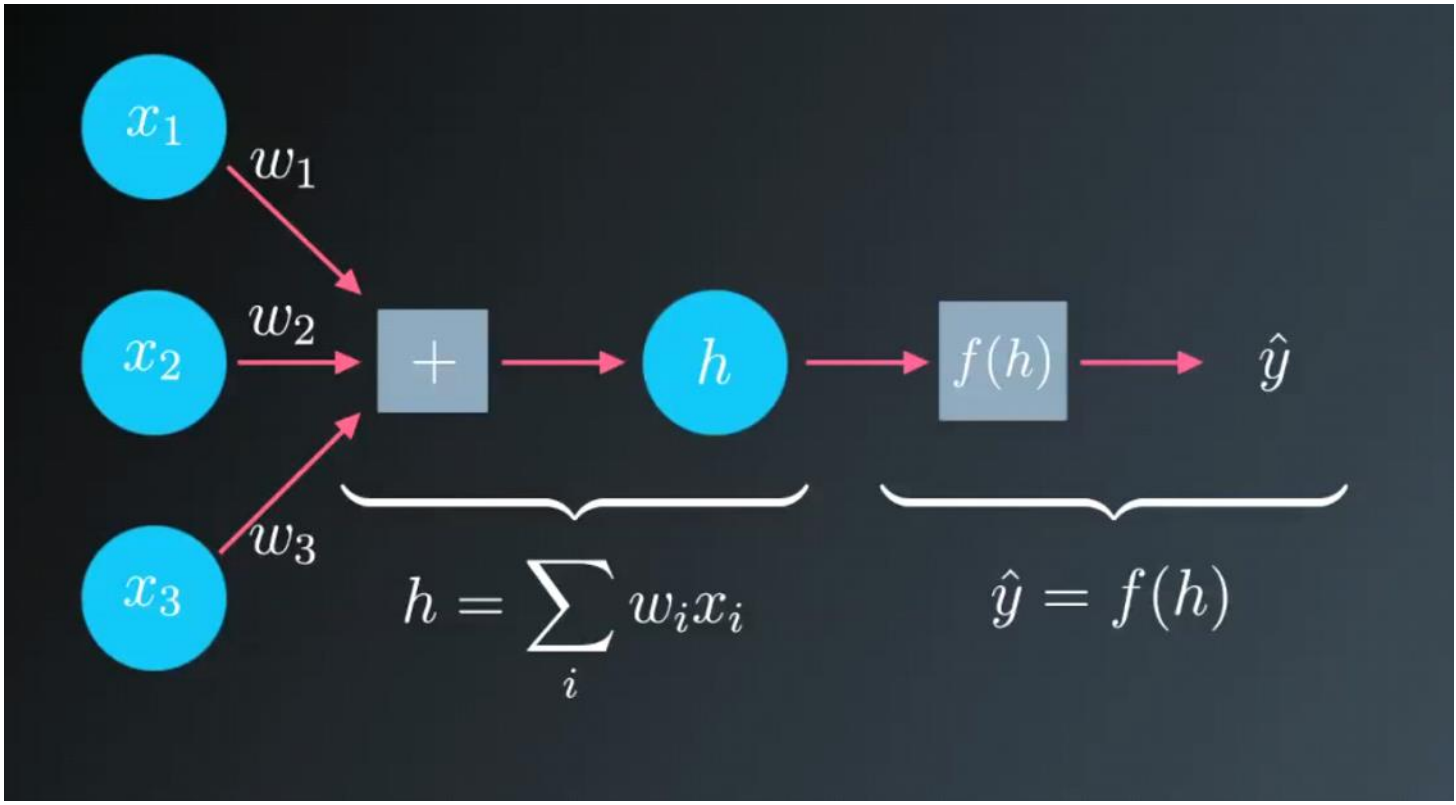
Since the weights will just go wherever the gradient takes them, they can end up where the error is low, but not the lowest. These spots are called local minima. If the weights are initialized with the wrong values, gradient descent could lead the weights into a local minimum, illustrated below.



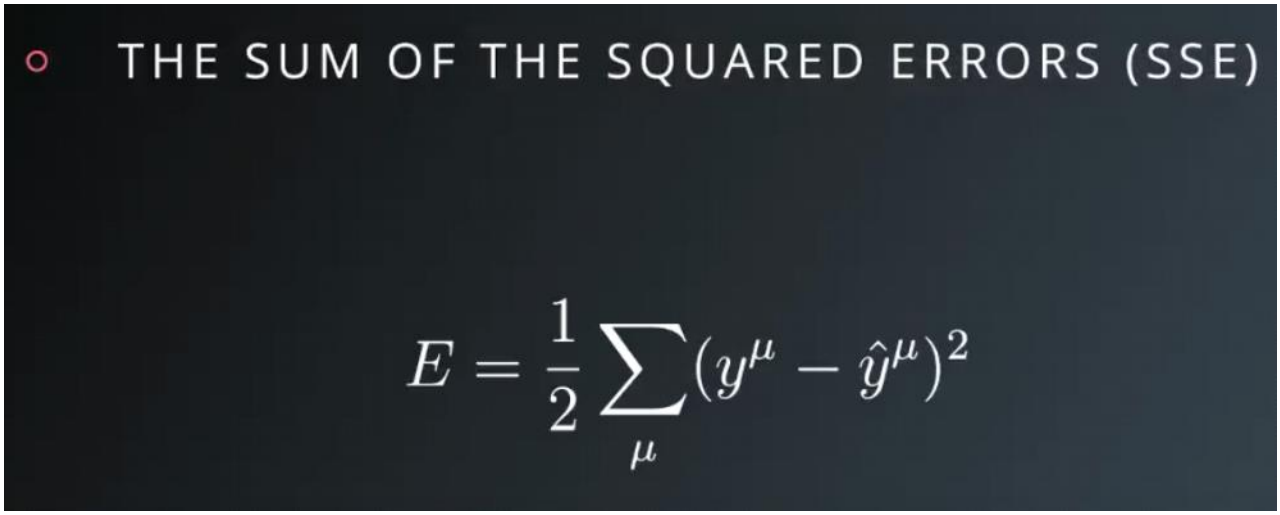
Gradient descent leading into a local minimum

There are methods to avoid this, such as using **momentum**.

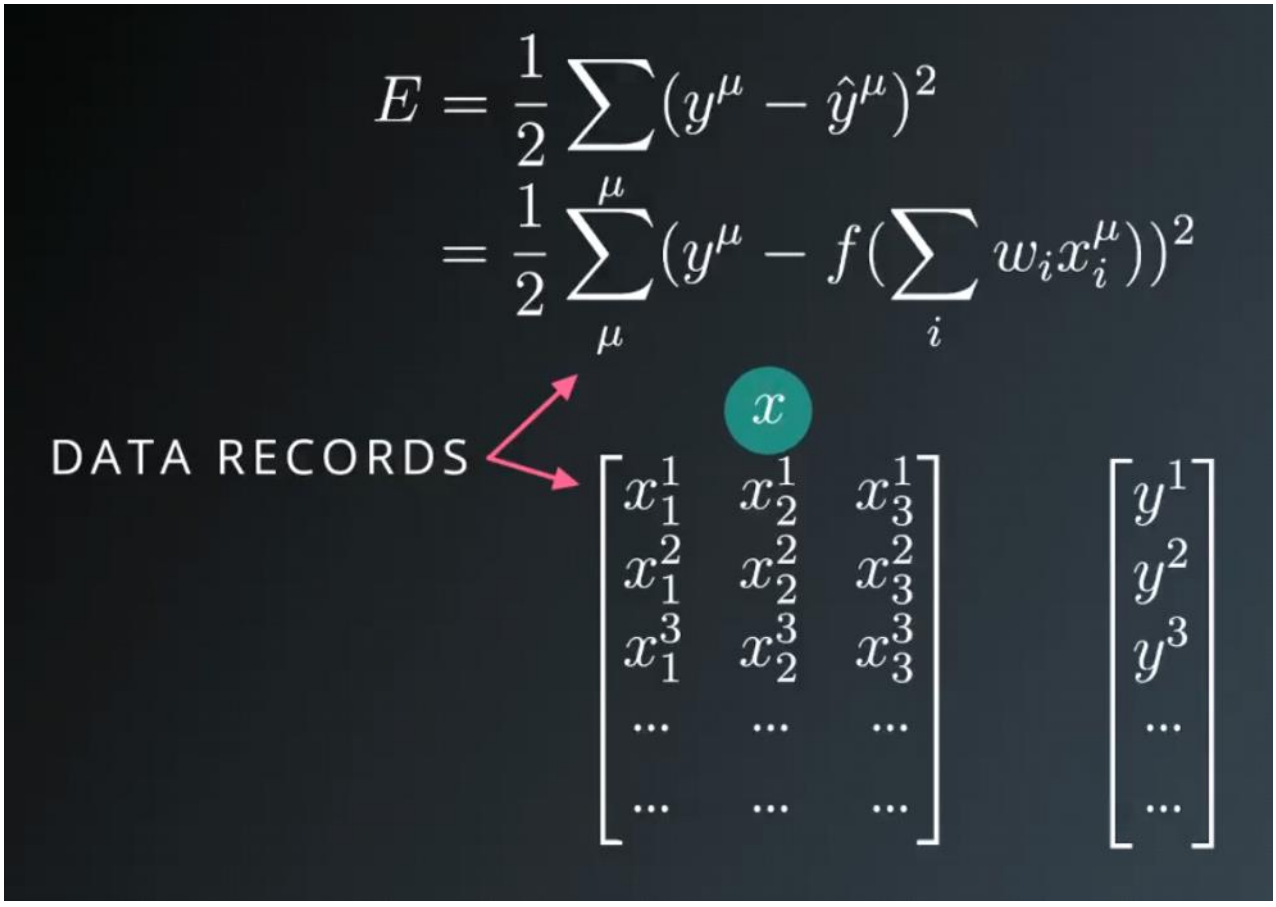
https://distill.pub/2017/momentum/



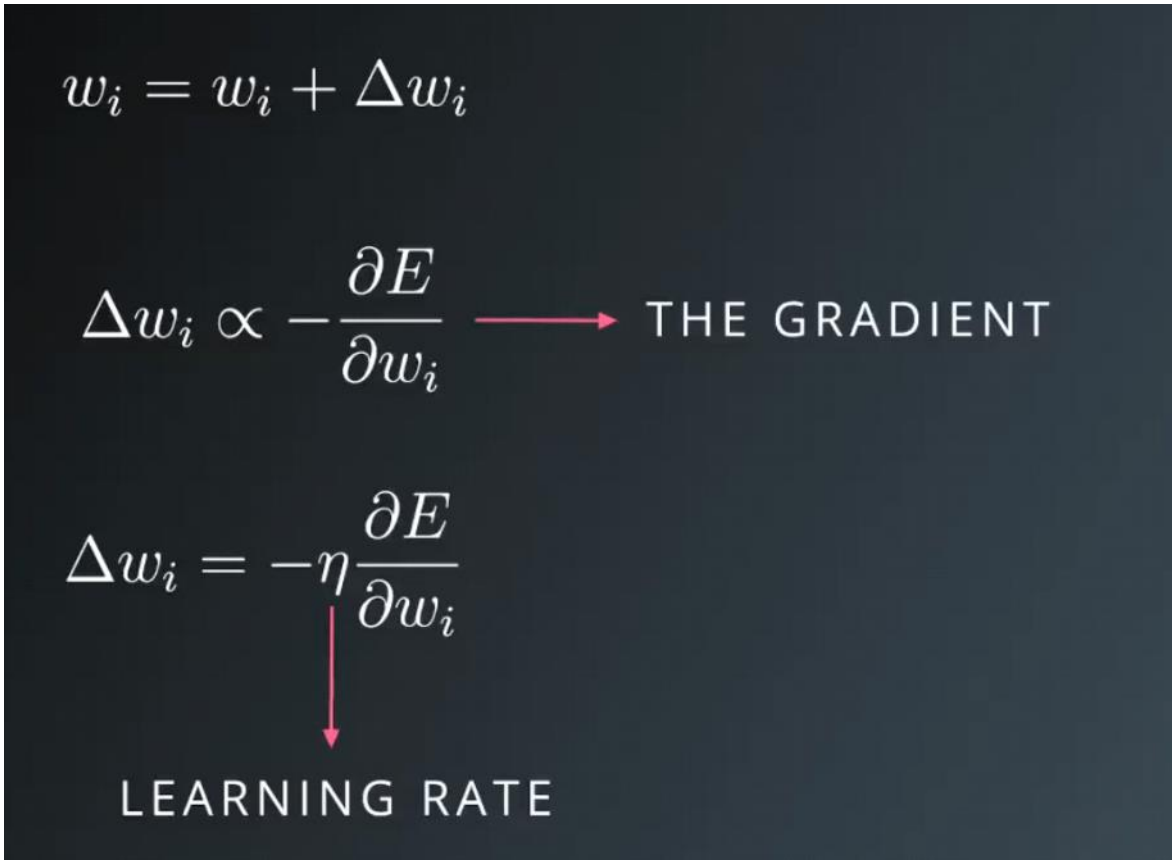
Screen clipping taken: 22/01/2021 19:31



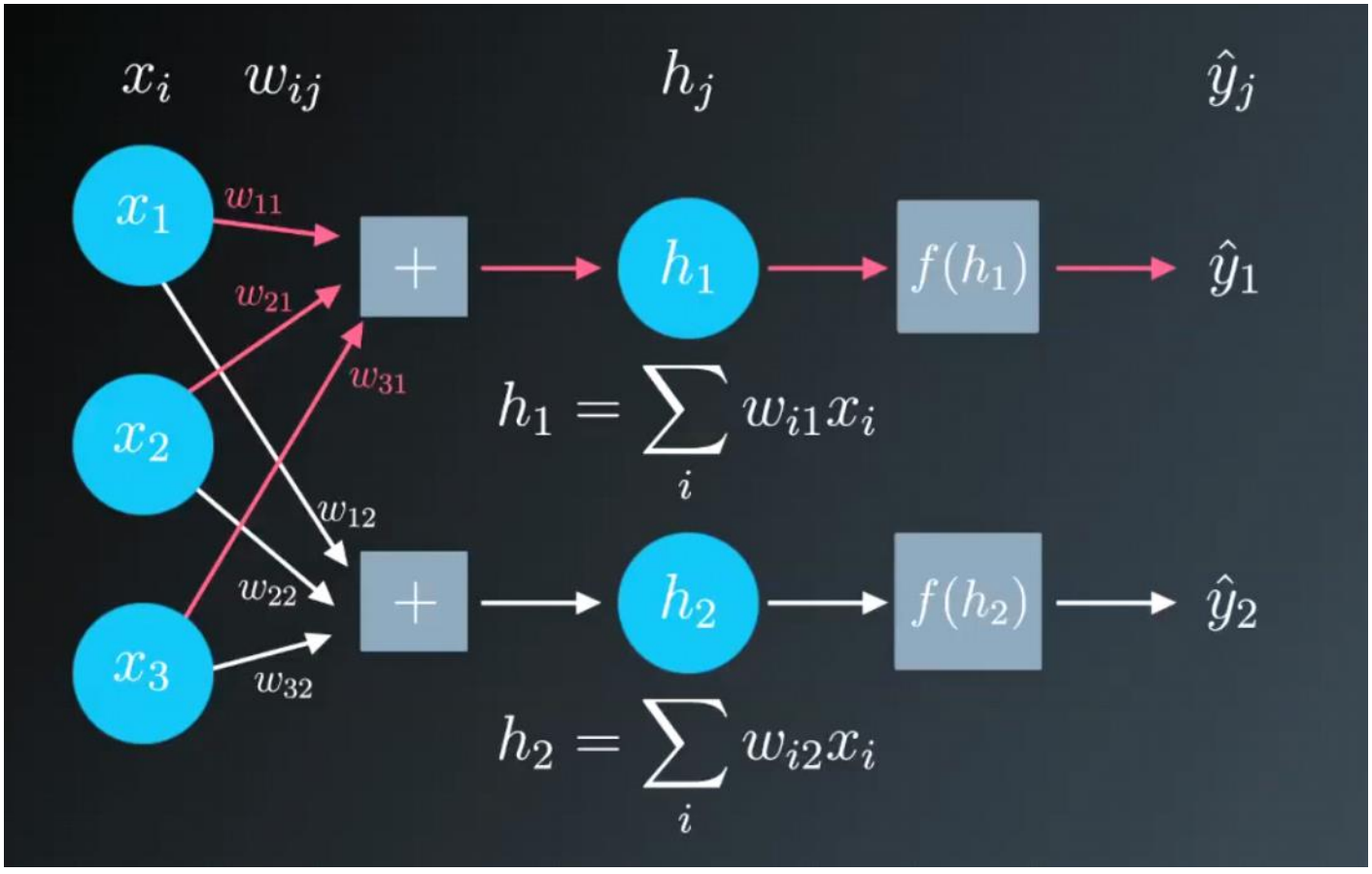
Screen clipping taken: 22/01/2021 19:33



Screen clipping taken: 22/01/2021 19:33



Screen clipping taken: 22/01/2021 19:35



Screen clipping taken: 22/01/2021 19:38

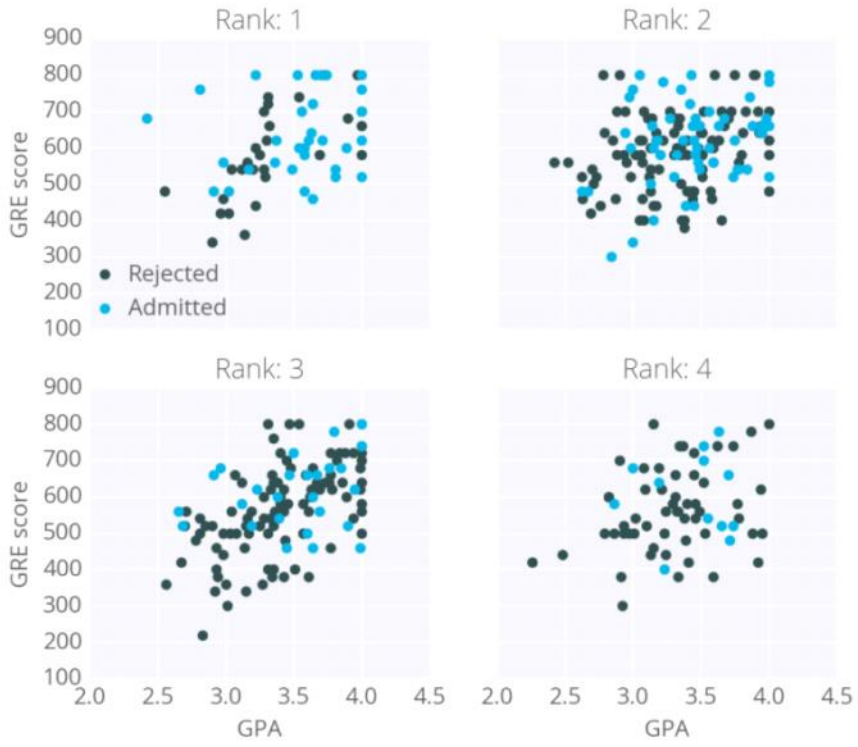
Implementing gradient descent

Okay, now we know how to update our weights:

$$\Delta w_{ij} = \eta * \delta_j * x_i,$$

You've seen how to implement that for a single update, but how do we translate that code to calculate many weight updates so our network will learn?

As an example, I'm going to have you use gradient descent to train a network on graduate school admissions data (found at http://www.ats.ucla.edu/stat/data/binary.csv). This dataset has three input features: GRE score, GPA, and the rank of the undergraduate school (numbered 1 through 4). Institutions with rank 1 have the highest prestige, those with rank 4 have the lowest.



The goal here is to predict if a student will be admitted to a graduate program based on these features. For this, we'll use a network with one output layer with one unit. We'll use a sigmoid function for the output unit activation.

Data cleanup

You might think there will be three input units, but we actually need to transform the data first. The `rank` feature is categorical, the numbers don't encode any sort of relative values. Rank 2 is not twice as much as rank 1, rank 3 is not 1.5 more than rank 2. Instead, we need to use dummy variables to encode `rank`, splitting the data into four new columns encoded with ones or zeros. Rows with rank 1 have one in the rank 1 dummy column, and zeros in all other columns. Rows with rank 2 have one in the rank 2 dummy column, and zeros in all other columns. And so on.

We'll also need to standardize the GRE and GPA data, which means to scale the values such that they have zero mean and a standard deviation of 1. This is necessary because the sigmoid function squashes really small and really large inputs. The gradient of really small and large inputs is zero, which means that the gradient descent step will go to zero too. Since the GRE and GPA values are fairly large, we have to be really careful about how we initialize the weights or the gradient descent steps will die off and the network won't train. Instead, if we standardize the data, we can initialize the weights easily and everyone is happy.

This is just a brief run-through, you'll learn more about preparing data later. If you're interested in how I did this, check out the `data_prep.py` file in the programming exercise below.

	admit	gre	gpa	rank_1	rank_2	rank_3	rank_4
15	0	-0.932334	0.131646	0	0	1	0
115	0	0.279614	1.576859	0	0	1	0
55	1	1.318426	1.603135	0	0	1	0
175	1	0.279614	-0.052290	0	1	0	0
63	1	0.799020	1.208986	0	0	1	0
67	0	0.279614	-0.236227	1	0	0	0
216	0	-2.144282	-1.287291	1	0	0	0
145	0	-1.798011	0.105369	0	0	1	0
286	1	1.837832	-0.446439	1	0	0	0
339	1	0.625884	0.210476	0	0	1	0

Ten rows of the data after transformations.

Now that the data is ready, we see that there are six input features: `gre`, `gpa`, and the four `rank` dummy variables.

Mean Square Error

We're going to make a small change to how we calculate the error here. Instead of the SSE, we're going to use the **mean** of the square errors (MSE). Now that we're using a lot of data, summing up all the weight steps can lead to really large updates that make the gradient descent diverge. To compensate for this, you'd need to use a quite small learning rate. Instead, we can just divide by the number of records in our data, m to take the average. This way, no matter how much data we use, our learning rates will typically be in the range of 0.01 to 0.001. Then, we can use the MSE (shown below) to calculate the gradient and the result is the same as before, just averaged instead of summed.

$$E = \frac{1}{2m}\sum_{\mu}(y^{\mu} - \hat{y}^{\mu})^2$$

Here's the general algorithm for updating the weights with gradient descent:

- Set the weight step to zero: $\Delta w_i = 0$
- For each record in the training data:
  - Make a forward pass through the network, calculating the output $\hat{y} = f(\sum_i w_i x_i)$
  - Calculate the error term for the output unit, $\delta = (y - \hat{y}) * f'(\sum_i w_i x_i)$
  - Update the weight step $\Delta w_i = \Delta w_i + \delta x_i$
- Update the weights $w_i = w_i + \eta \Delta w_i / m$ where η is the learning rate and m is the number of records. Here we're averaging the weight steps to help reduce any large variations in the training data.
- Repeat for e epochs.

You can also update the weights on each record instead of averaging the weight steps after going through all the records.

Remember that we're using the sigmoid for the activation function, $f(h) = 1/(1 + e^{-h})$

And the gradient of the sigmoid is $f'(h) = f(h)(1 - f(h))$

where h is the input to the output unit,

$h = \sum_i w_i x_i$

Implementing with NumPy

For the most part, this is pretty straightforward with NumPy.

First, you'll need to initialize the weights. We want these to be small such that the input to the sigmoid is in the linear region near 0 and not squashed at the high and low ends. It's also important to initialize them randomly so that they all have different starting values and diverge, breaking symmetry. So, we'll initialize the weights from a normal distribution centered at 0. A good value for the scale is $1/\sqrt{n}$ where n is the number of input units. This keeps the input to the sigmoid low for increasing numbers of input units.

```
weights = np.random.normal(scale=1/n_features**.5, size=n_features)
```

NumPy provides a function `np.dot()` that calculates the dot product of two arrays, which conveniently calculates h for us. The dot product multiplies two arrays element-wise, the first element in array 1 is multiplied by the first element in array 2, and so on. Then, each product is summed.

```
# input to the output layer
output_in = np.dot(weights, inputs)
```

And finally, we can update Δw_i and w_i by incrementing them with `weights += ...` which is shorthand for `weights = weights + ...`.

Efficiency tip!

You can save some calculations since we're using a sigmoid here. For the sigmoid function, $f'(h) = f(h)(1 - f(h))$. That means that once you calculate $f(h)$, the activation of the output unit, you can use it to calculate the gradient for the error gradient.

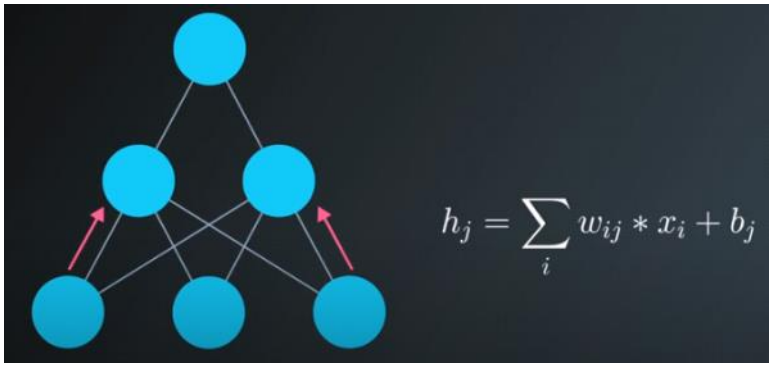
Programming exercise

Below, you'll implement gradient descent and train the network on the admissions data. Your goal here is to train the network until you reach a minimum in the mean square error (MSE) on the training set. You need to implement:

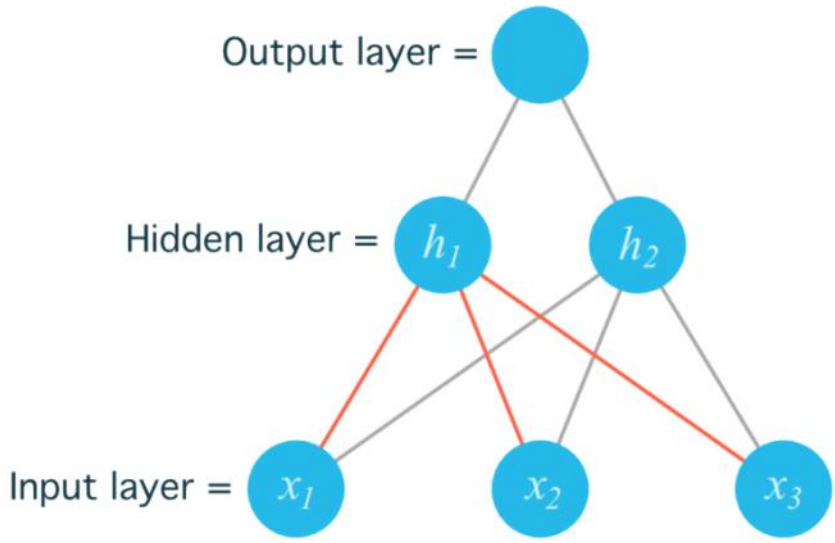
- The network output: `output`.
- The output error: `error`.
- The error term: `error_term`.
- Update the weight step: `del_w +=`.
- Update the weights: `weights +=`.

After you've written these parts, run the training by pressing "Test Run". The MSE will print out, as well as the accuracy on a test set, the fraction of correctly predicted admissions.

Feel free to play with the hyperparameters and see how it changes the MSE.



Screen clipping taken: 23/01/2021 11:38



The lines indicating the weights leading to h_1 have been colored differently from those leading to h_2 just to make it easier to read.

Now to index the weights, we take the input unit number for the $_i$ and the hidden unit number for the $_j$. That gives us

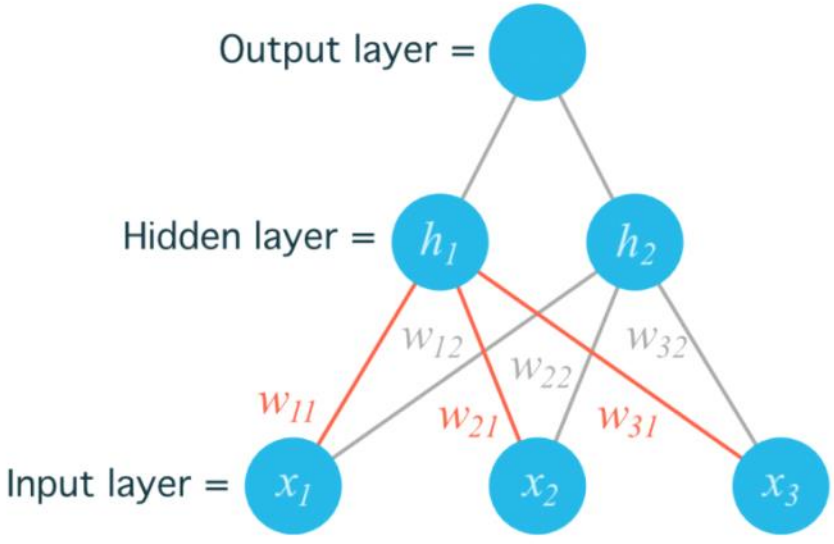
w_{11}

for the weight leading from x_1 to h_1, and

w_{12}

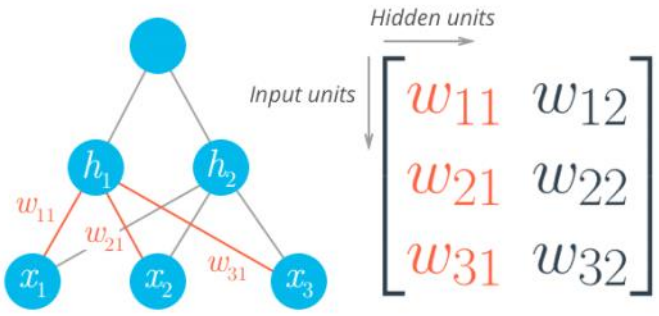
for the weight leading from x_1 to h_2.

The following image includes all of the weights between the input layer and the hidden layer, labeled with their appropriate w_{ij} indices:



Before, we were able to write the weights as an array, indexed as w_i.

But now, the weights need to be stored in a **matrix**, indexed as w_{ij}. Each **row** in the matrix will correspond to the weights **leading out** of a **single input unit**, and each **column** will correspond to the weights **leading in** to a **single hidden unit**. For our three input units and two hidden units, the weights matrix looks like this:



Weights matrix for 3 input units and 2 hidden units

Be sure to compare the matrix above with the diagram shown before it so you can see where the different weights in the network end up in the matrix.

To initialize these weights in NumPy, we have to provide the shape of the matrix. If `features` is a 2D array containing the input data:

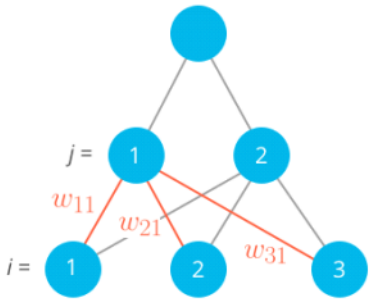
```
# Number of records and input units
n_records, n_inputs = features.shape
# Number of hidden units
n_hidden = 2
weights_input_to_hidden = np.random.normal(0, n_inputs**-0.5, size=(n_inputs, n_hidden))
```

This creates a 2D array (i.e. a matrix) named `weights_input_to_hidden` with dimensions `n_inputs` by `n_hidden`. Remember how the input to a hidden unit is the sum of all the inputs multiplied by the hidden unit's weights. So for each hidden layer unit, h_j, we need to calculate the following:

$$h_j = \sum_i w_{ij} x_i$$

To do that, we now need to use **matrix multiplication**. If your linear algebra is rusty, I suggest taking a look at the suggested resources in the prerequisites section. For this part though, you'll only need to know how to multiply a matrix with a vector.

In this case, we're multiplying the inputs (a row vector here) by the weights. To do this, you take the dot (inner) product of the inputs with each column in the weights matrix. For example, to calculate the input to the first hidden unit, $j = 1$, you'd take the dot product of the inputs with the first column of the weights matrix, like so:



$$\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix}$$

Calculating the input to the first hidden unit with the first column of the weights matrix.

$$h_1 = x_1 w_{11} + x_2 w_{21} + x_3 w_{31}$$

And for the second hidden layer input, you calculate the dot product of the inputs with the second column. And so on and so forth.

In NumPy, you can do this for all the inputs and all the outputs at once using `np.dot`

```
hidden_inputs = np.dot(inputs, weights_input_to_hidden)
```

You could also define your weights matrix such that it has dimensions `n_hidden` by `n_inputs` then multiply like so where the inputs form a *column vector*:

$$h_j = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

Note: The weight indices have changed in the above image and no longer match up with the labels used in the earlier diagrams. That's because, in matrix notation, the row index always precedes the column index, so it would be misleading to label them the way we did in the neural net diagram. Just keep in mind that this is the same weight matrix as before, but rotated so the first column is now the first row, and the second column is now the second row. If we *were* to use the labels from the earlier diagram, the weights would fit into the matrix in the following locations:

$$\begin{bmatrix} w_{11} & w_{21} & w_{31} \\ w_{12} & w_{22} & w_{32} \end{bmatrix}$$

Weight matrix shown with labels matching earlier diagrams.

Remember, the above is **not** a correct view of the **indices**, but it uses the labels from the earlier neural net diagrams to show you where each weight ends up in the matrix.

Remember, the above is **not** a correct view of the **indices**, but it uses the labels from the earlier neural net diagrams to show you where each weight ends up in the matrix.

The important thing with matrix multiplication is that *the dimensions match*. For matrix multiplication to work, there has to be the same number of elements in the dot products. In the first example, there are three columns in the input vector, and three rows in the weights matrix. In the second example, there are three columns in the weights matrix and three rows in the input vector. If the dimensions don't match, you'll get this:

```
# Same weights and features as above, but swapped the order
hidden_inputs = np.dot(weights_input_to_hidden, features)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-11-1bfa0f615c45> in <module>()
----> 1 hidden_in = np.dot(weights_input_to_hidden, X)

ValueError: shapes (3,2) and (3,) not aligned: 2 (dim 1) != 3 (dim 0)
```

The dot product can't be computed for a 3x2 matrix and 3-element array. That's because the 2 columns in the matrix don't match the number of elements in the array. Some of the dimensions that could work would be the following:

$$\begin{bmatrix} x_1 & x_2 \end{bmatrix} \times \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}$$

$$2 \qquad\qquad 2 \times 3$$

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

$$2 \times 3 \qquad\qquad 3$$

The rule is that if you're multiplying an array from the left, the array must have the same number of elements as there are rows in the matrix. And if you're multiplying the *matrix* from the left, the number of columns in the matrix must equal the number of elements in the array on the right.

Making a column vector

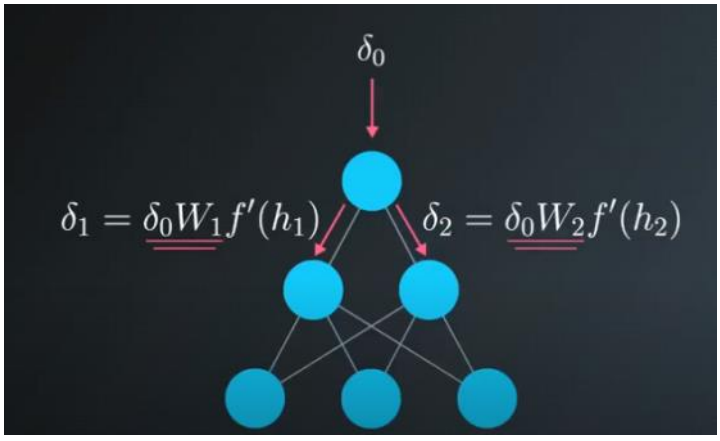
You see above that sometimes you'll want a column vector, even though by default NumPy arrays work like row vectors. It's possible to get the transpose of an array like so `arr.T`, but for a 1D array, the transpose will return a row vector. Instead, use `arr[:,None]` to create a column vector:

```
print(features)
> array([ 0.49671415, -0.1382643 ,  0.64768854])

print(features.T)
> array([ 0.49671415, -0.1382643 ,  0.64768854])

print(features[:, None])
> array([[ 0.49671415],
       [-0.1382643 ],
       [ 0.64768854]])
```

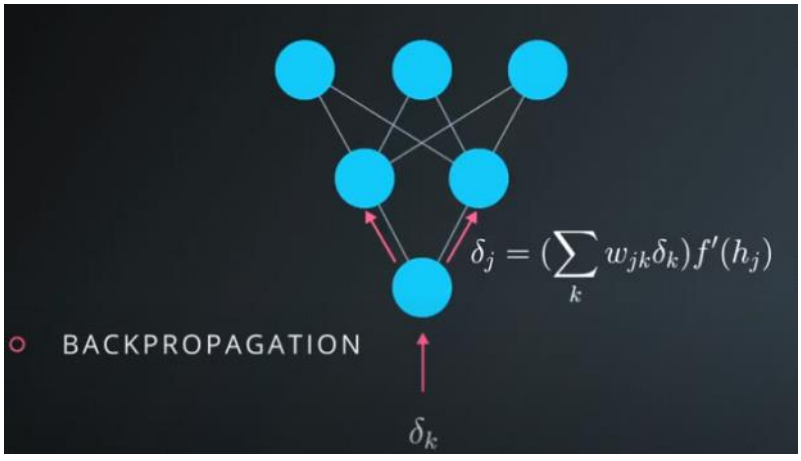
Alternatively, you can create arrays with two dimensions. Then, you can use `arr.T` to get the column vector.

```
np.array(features, ndmin=2)
> array([[ 0.49671415, -0.1382643 ,  0.64768854]])

np.array(features, ndmin=2).T
> array([[ 0.49671415],
       [-0.1382643 ],
       [ 0.64768854]])
```

I personally prefer keeping all vectors as 1D arrays, it just works better in my head.



Screen clipping taken: 23/01/2021 12:22



Screen clipping taken: 23/01/2021 12:23

Backpropagation

Now we've come to the problem of how to make a multilayer neural network *learn*. Before, we saw how to update weights with gradient descent. The backpropagation algorithm is just an extension of that, using the chain rule to find the error with the respect to the weights connecting the input layer to the hidden layer (for a two layer network).

To update the weights to hidden layers using gradient descent, you need to know how much error each of the hidden units contributed to the final output. Since the output of a layer is determined by the weights between layers, the error resulting from units is scaled by the weights going forward through the network. Since we know the error at the output, we can use the weights to work backwards to hidden layers.

For example, in the output layer, you have errors δ_k^o attributed to each output unit k. Then, the error attributed to hidden unit j is the output errors, scaled by the weights between the output and hidden layers (and the gradient):

$$\delta_j^h = \sum W_{jk} \delta_k^o f'(h_j)$$

Then, the gradient descent step is the same as before, just with the new errors:

$$\Delta w_{ij} = \eta \delta_j^h x_i$$

where w_{ij} are the weights between the inputs and hidden layer and x_i are input unit values. This form holds for however many layers there are. The weight steps are equal to the step size times the output error of the layer times the values of the inputs to that layer

$$\Delta w_{pq} = \eta \delta_{output} V_{in}$$

Here, you get the output error, δ_{output}, by propagating the errors backwards from higher layers. And the input values, V_{in} are the inputs to the layer, the hidden layer activations to the output unit for example.

Working through an example

Let's walk through the steps of calculating the weight updates for a simple two layer network. Suppose there are two input values, one hidden unit, and one output unit, with sigmoid activations on the hidden and output units. The following image depicts this network. (**Note:** the input values are shown as nodes at the bottom of the image, while the network's output value is shown as $\hat{y}$ at the top. The inputs themselves do not count as a layer, which is why this is considered a two layer network.)

Assume we're trying to fit some binary data and the target is $y = 1$. We'll start with the forward pass, first calculating the input to the hidden unit

$$h = \sum_i w_i x_i = 0.1 \times 0.4 - 0.2 \times 0.3 = -0.02$$

and the output of the hidden unit

$$a = f(h) = \text{sigmoid}(-0.02) = 0.495.$$

Using this as the input to the output unit, the output of the network is

$$\hat{y} = f(W \cdot a) = \text{sigmoid}(0.1 \times 0.495) = 0.512.$$

With the network output, we can start the backwards pass to calculate the weight updates for both layers. Using the fact that for the sigmoid function $f'(W \cdot a) = f(W \cdot a)(1 - f(W \cdot a))$, the error term for the output unit is

$$\delta^o = (y - \hat{y}) f'(W \cdot a) = (1 - 0.512) \times 0.512 \times (1 - 0.512) = 0.122.$$

Now we need to calculate the error term for the hidden unit with backpropagation. Here we'll scale the error term from the output unit by the weight W connecting it to the hidden unit. For the hidden unit error term, $\delta_j^h = \sum_k W_{jk} \delta_k^o f'(h_j)$, but since we have one hidden unit and one output unit, this is much simpler.

$$\delta^h = W \delta^o f'(h) = 0.1 \times 0.122 \times 0.495 \times (1 - 0.495) = 0.003$$

Now that we have the errors, we can calculate the gradient descent steps. The hidden to output weight step is the learning rate, times the output unit error, times the hidden unit activation value.

$$\Delta W = \eta \delta^o a = 0.5 \times 0.122 \times 0.495 = 0.0302$$

Then, for the input to hidden weights w_i, it's the learning rate times the hidden unit error, times the input values.

$$\Delta w_i = \eta \delta^h x_i = (0.5 \times 0.003 \times 0.1, 0.5 \times 0.003 \times 0.3) = (0.00015, 0.00045)$$

From this example, you can see one of the effects of using the sigmoid function for the activations. The maximum derivative of the sigmoid function is 0.25, so the errors in the output layer get reduced by at least 75%, and errors in the hidden layer are scaled down by at least 93.75%! You can see that if you have a lot of layers, using a sigmoid activation function will quickly reduce the weight steps to tiny values in layers near the input. This is known as the **vanishing gradient** problem. Later in the course you'll learn about other activation functions that perform better in this regard and are more commonly used in modern network architectures.

Implementing in NumPy

For the most part you have everything you need to implement backpropagation with NumPy.

However, previously we were only dealing with error terms from one unit. Now, in the weight update, we have to consider the error for *each unit* in the hidden layer, δ_j:

$$\Delta w_{ij} = \eta \delta_j x_i$$

Firstly, there will likely be a different number of input and hidden units, so trying to multiply the errors and the inputs as row vectors will throw an error:

```
hidden_error*inputs
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-22-3b59121cb809> in <module>()
----> 1 hidden_error*x

ValueError: operands could not be broadcast together with shapes (3,) (6,)
```

Also, w_{ij} is a matrix now, so the right side of the assignment must have the same shape as the left side. Luckily, NumPy takes care of this for us. If you multiply a row vector array with a column vector array, it will multiply the first element in the column by each element in the row vector and set that as the first row in a new 2D array. This continues for each element in the column vector, so you get a 2D array that has shape `(len(column_vector), len(row_vector))`.

```
hidden_error*inputs[:,None]
array([[ -8.24195994e-04,  -2.71771975e-04,   1.29713395e-03],
       [ -2.87777394e-04,  -9.48922722e-05,   4.52909055e-04],
       [  6.44605731e-04,   2.12553536e-04,  -1.01449168e-03],
       [  0.00000000e+00,   0.00000000e+00,  -0.00000000e+00],
       [  0.00000000e+00,   0.00000000e+00,  -0.00000000e+00],
       [  0.00000000e+00,   0.00000000e+00,  -0.00000000e+00]])
```

It turns out this is exactly how we want to calculate the weight update step. As before, if you have your inputs as a 2D array with one row, you can also do `hidden_error*inputs.T`, but that won't work if `inputs` is a 1D array.

Implementing backpropagation

Now we've seen that the error term for the output layer is

$$\delta_k = (y_k - \hat{y}_k) f'(a_k)$$

and the error term for the hidden layer is

$$\delta_j = \sum [w_{jk} \delta_k] f'(h_j)$$

For now we'll only consider a simple network with one hidden layer and one output unit. Here's the general algorithm for updating the weights with backpropagation:

- Set the weight steps for each layer to zero
  - The input to hidden weights $\Delta w_{ij} = 0$
  - The hidden to output weights $\Delta W_j = 0$
- For each record in the training data:
  - Make a forward pass through the network, calculating the output $\hat{y}$
  - Calculate the error gradient in the output unit, $\delta^o = (y - \hat{y}) f'(z)$ where $z = \sum_j W_j a_j$, the input to the output unit.
  - Propagate the errors to the hidden layer $\delta_j^h = \delta^o W_j f'(h_j)$
  - Update the weight steps:
    - $\Delta W_j = \Delta W_j + \delta^o a_j$
    - $\Delta w_{ij} = \Delta w_{ij} + \delta_j^h a_i$
- Update the weights, where η is the learning rate and m is the number of records:
  - $W_j = W_j + \eta \Delta W_j / m$
  - $w_{ij} = w_{ij} + \eta \Delta w_{ij} / m$
- Repeat for e epochs.

Screen clipping taken: 23/01/2021 20:00

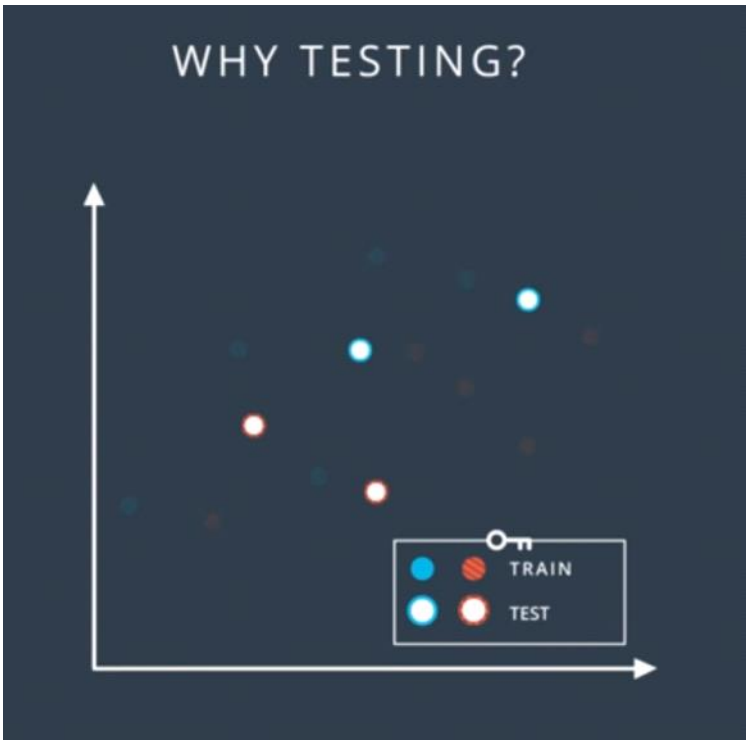
https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Nanodegree - NN - Lesson 4

Sunday 24 January 2021 19:35

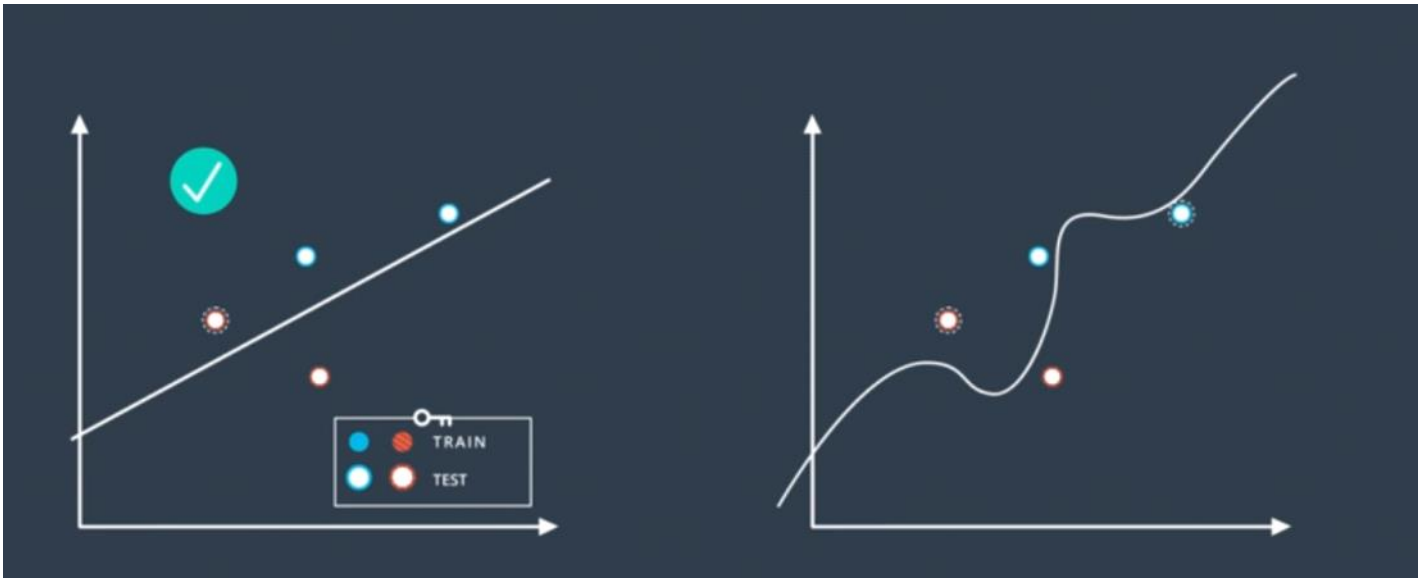
In this lesson we explored some technical optimisation for the Neural network.

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.



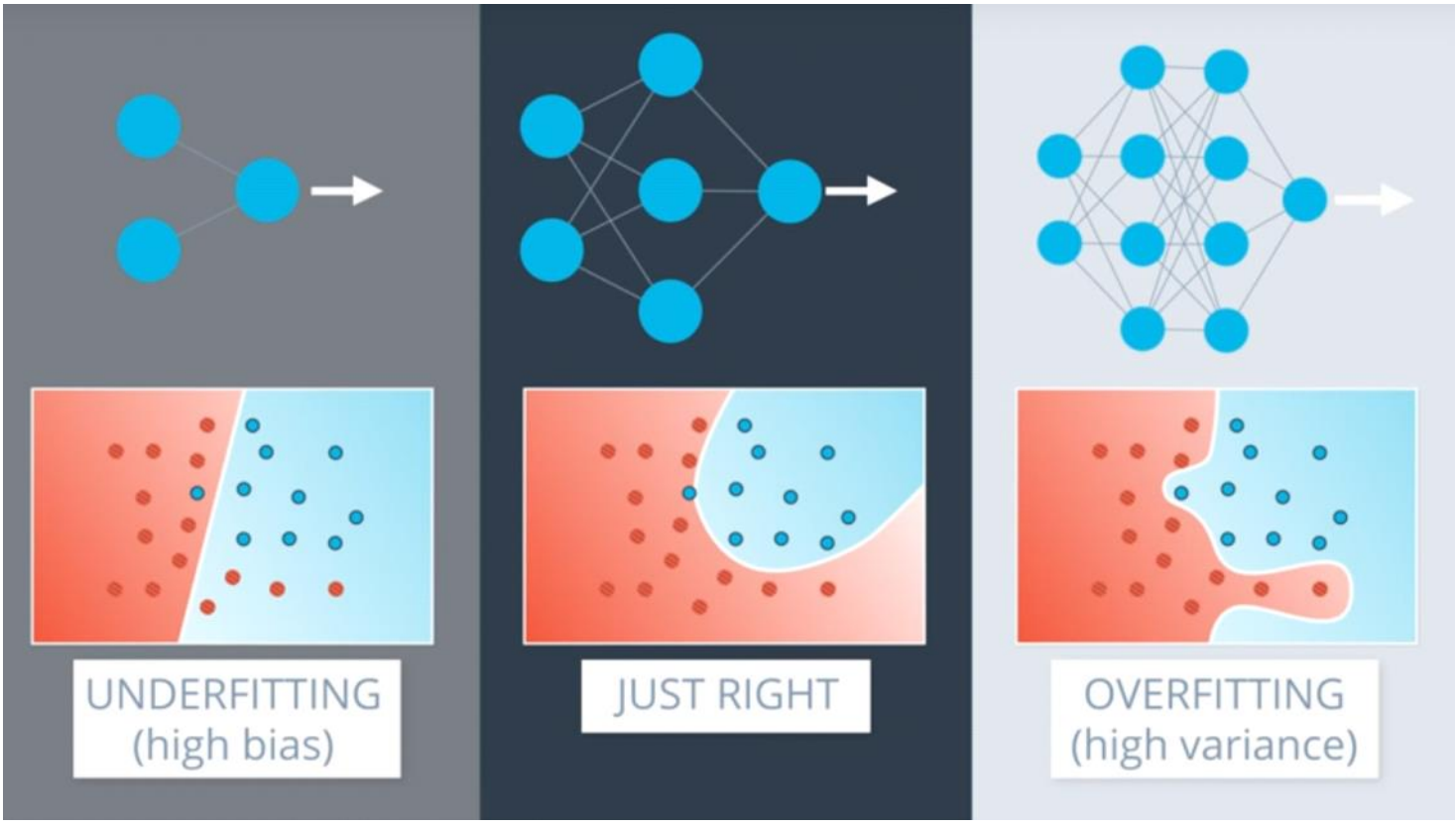
Screen clipping taken: 24/01/2021 19:35

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

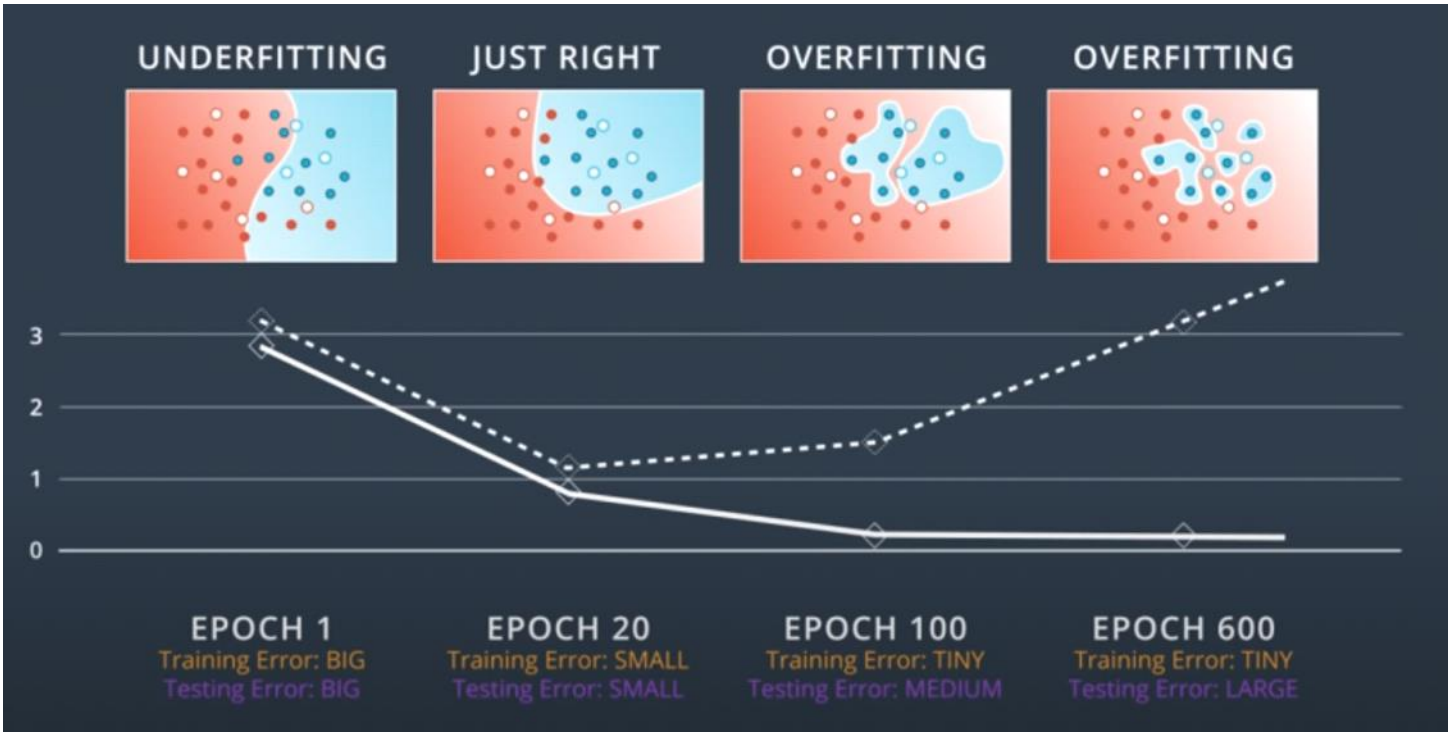


Screen clipping taken: 24/01/2021 19:36

Go for the simpler model - always .

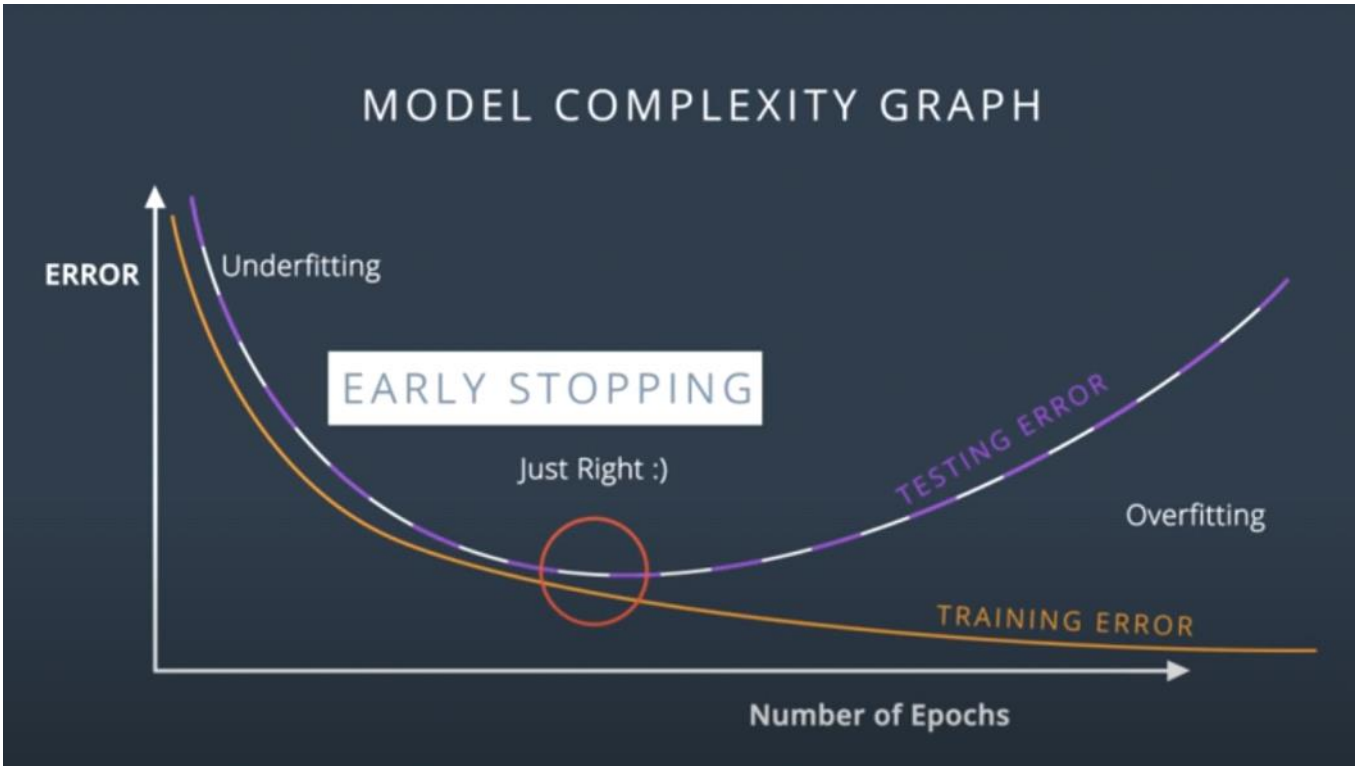


Screen clipping taken: 24/01/2021 19:40



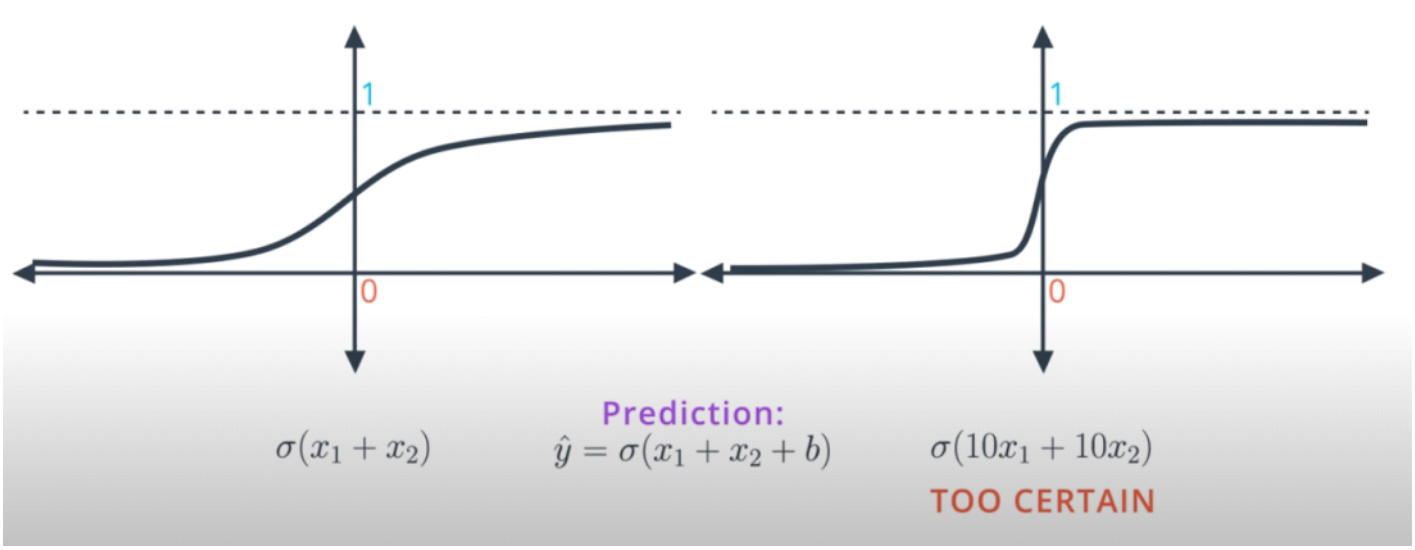
Screen clipping taken: 24/01/2021 19:44

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

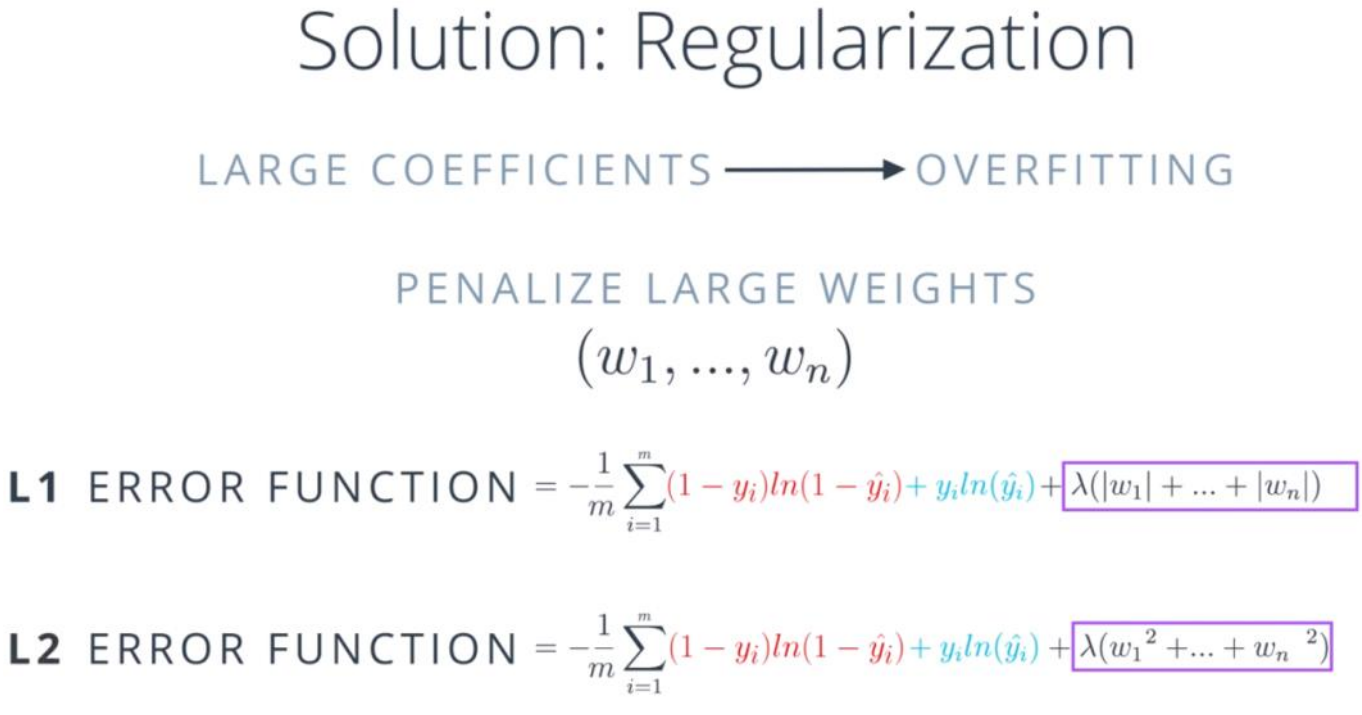
We have also another problem with the activation functions. There are functions that being more steep allow us to descend more and make the model quicker but the problem with steeper function is that the derivative is smaller, so we end up having products of small numbers that gives us small numbers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



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.



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

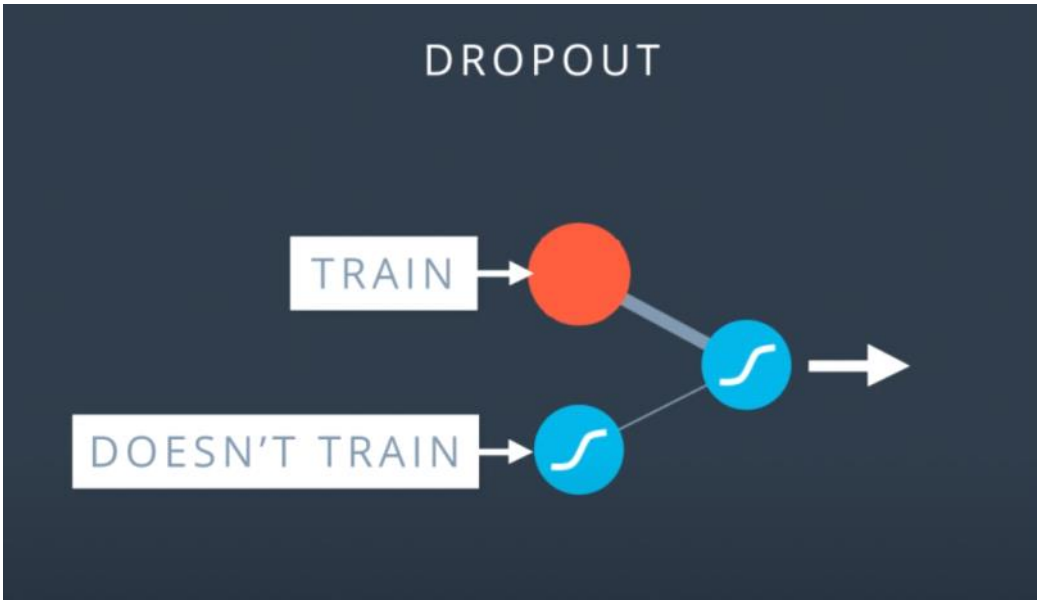
NORMALLY BETTER FOR TRAINING MODELS

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

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

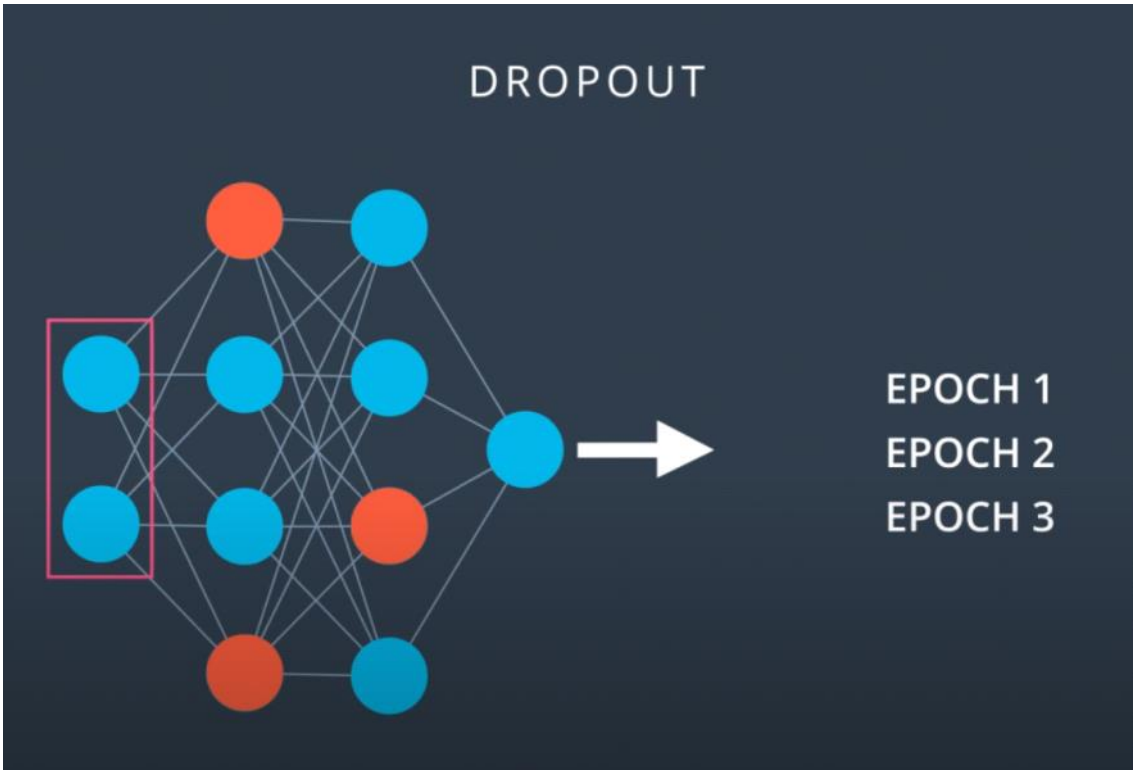
Screen clipping taken: 24/01/2021 19:57

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.



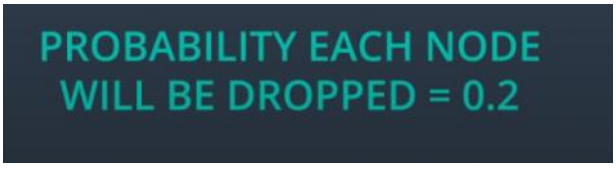
Screen clipping taken: 24/01/2021 19:59

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



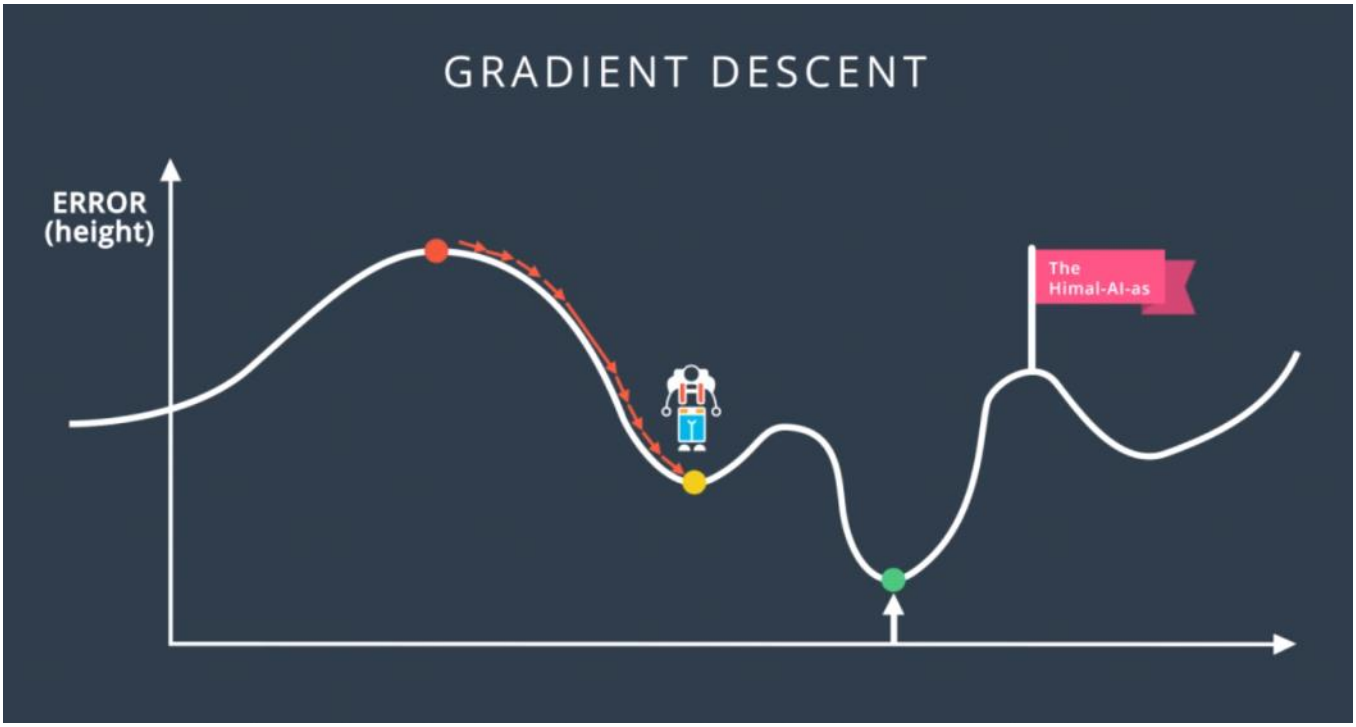
Screen clipping taken: 24/01/2021 20:00

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.



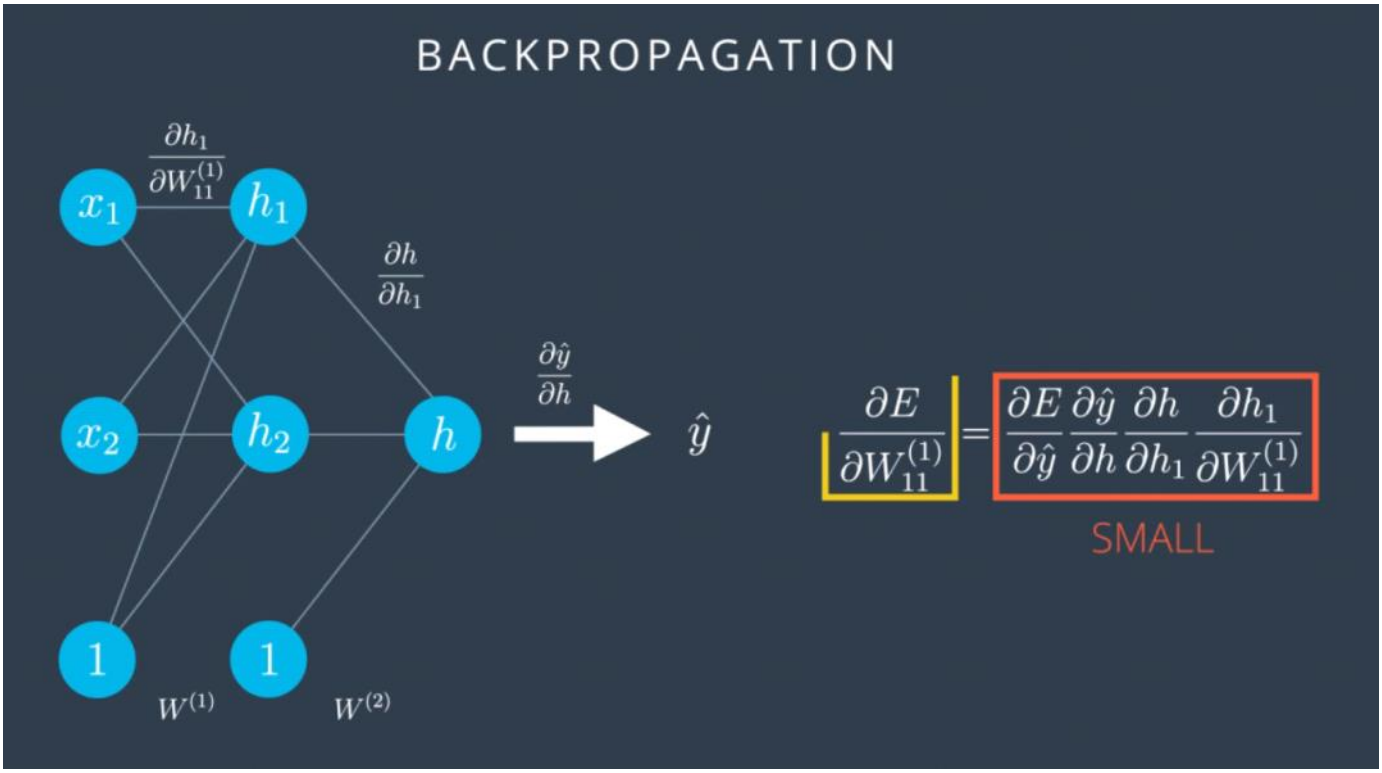
Screen clipping taken: 24/01/2021 20:00

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.



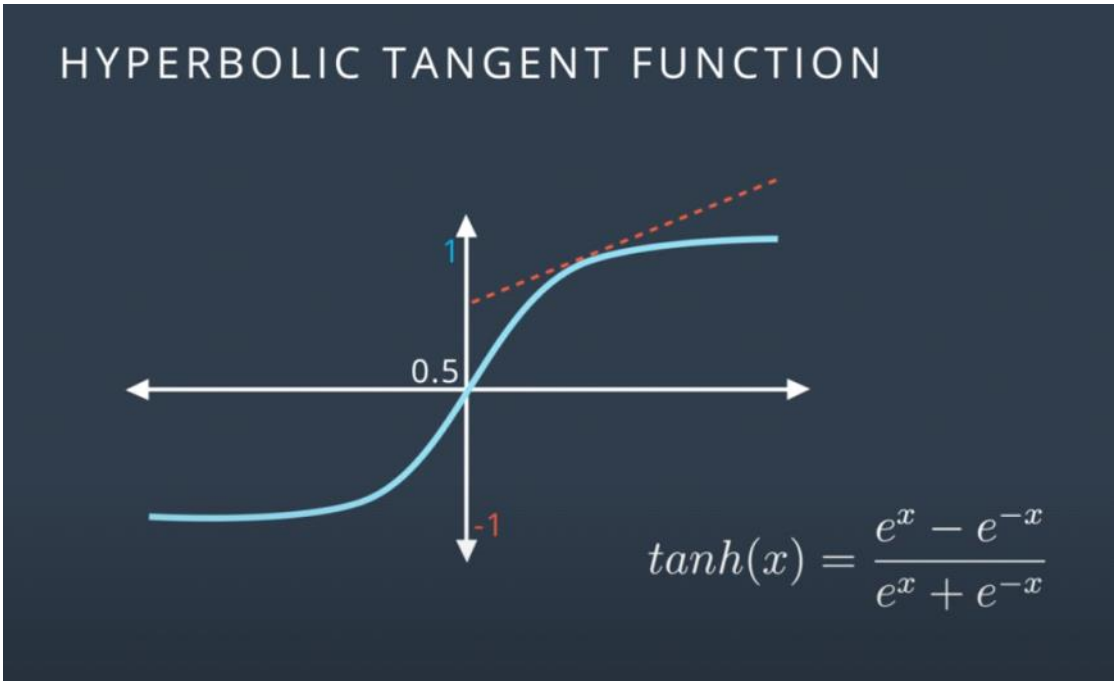
Screen clipping taken: 25/01/2021 09:10

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.



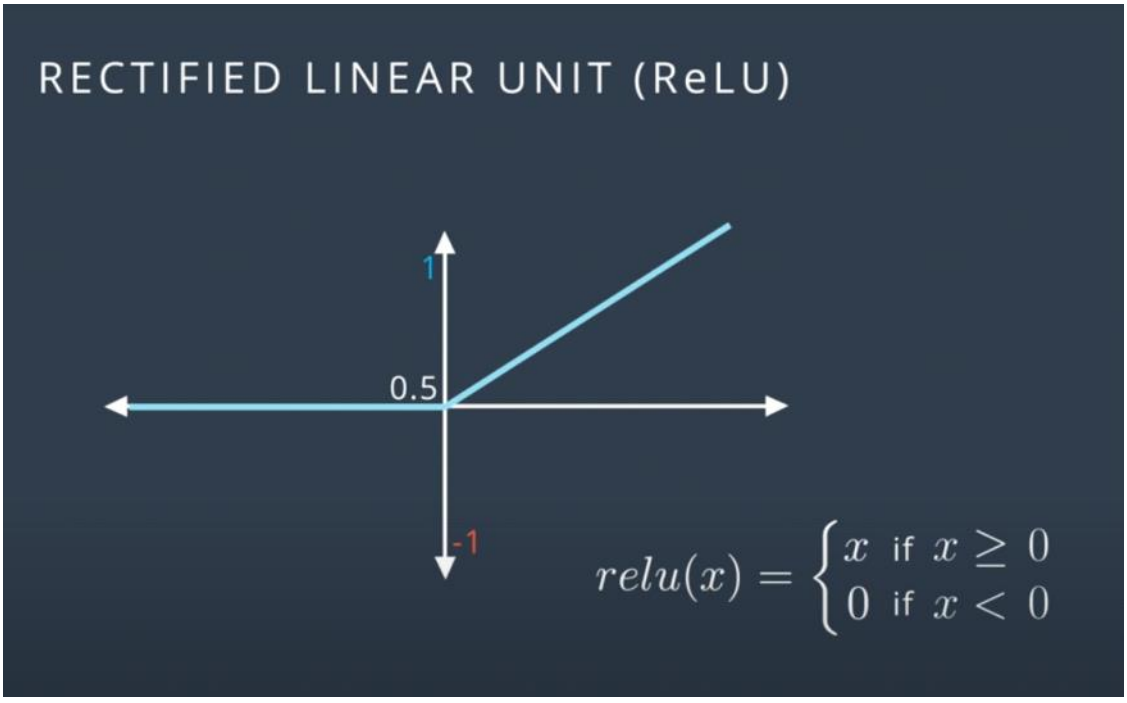
Screen clipping taken: 25/01/2021 09:11

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.



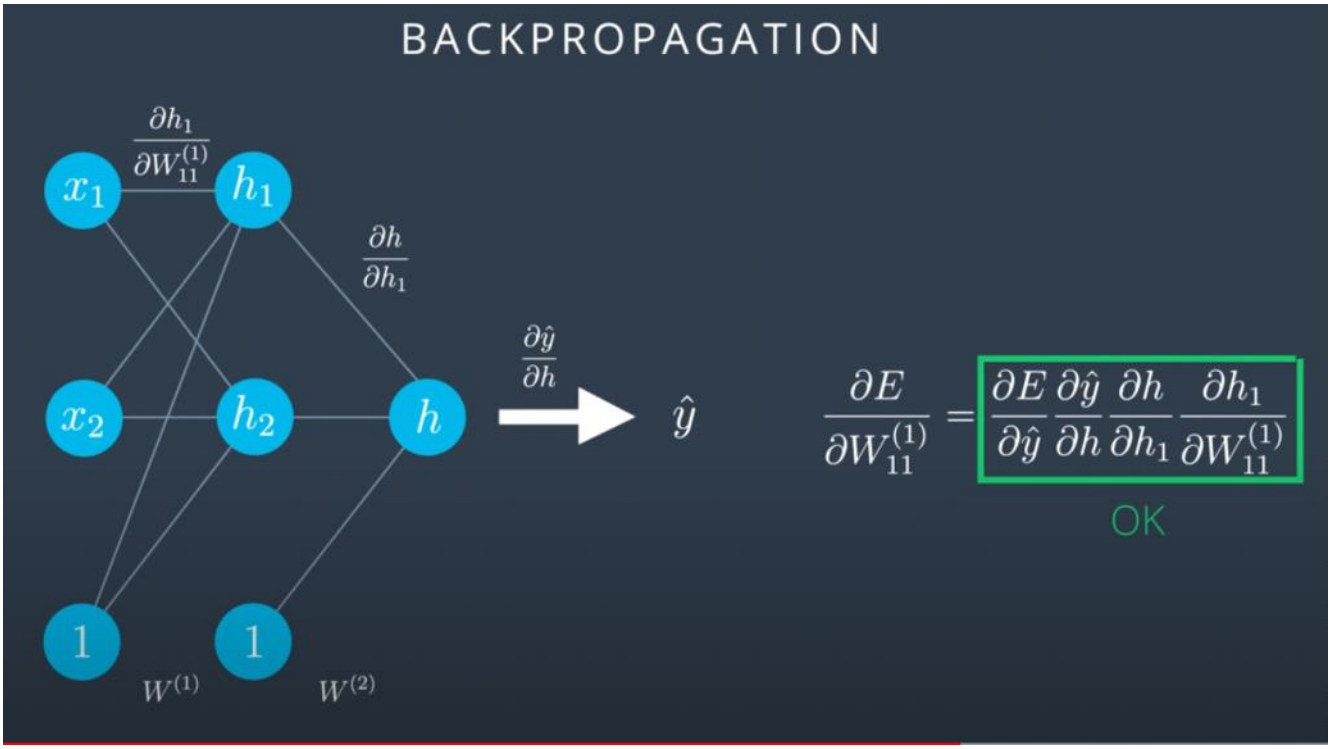
Screen clipping taken: 25/01/2021 09:12

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.



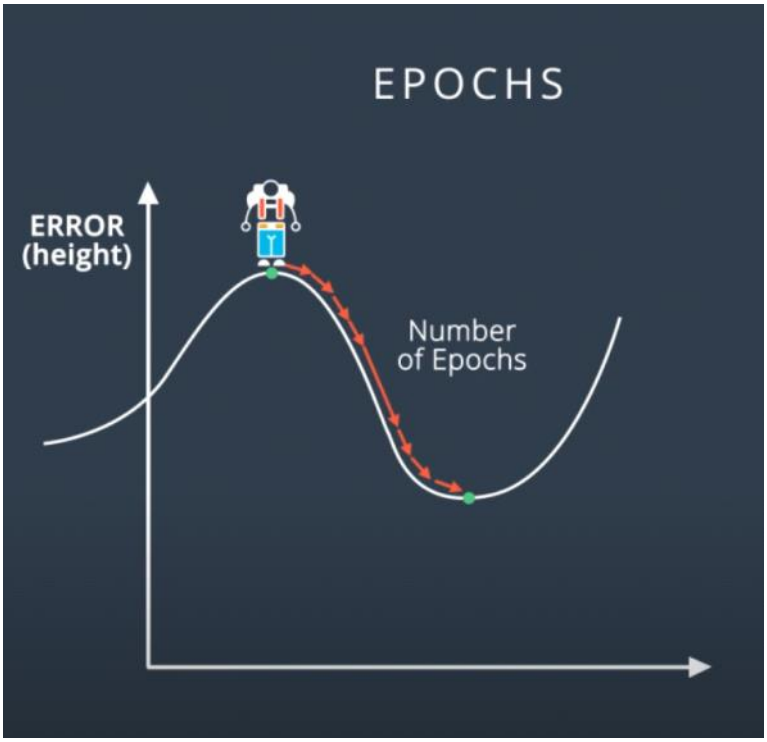
Screen clipping taken: 25/01/2021 09:12

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.



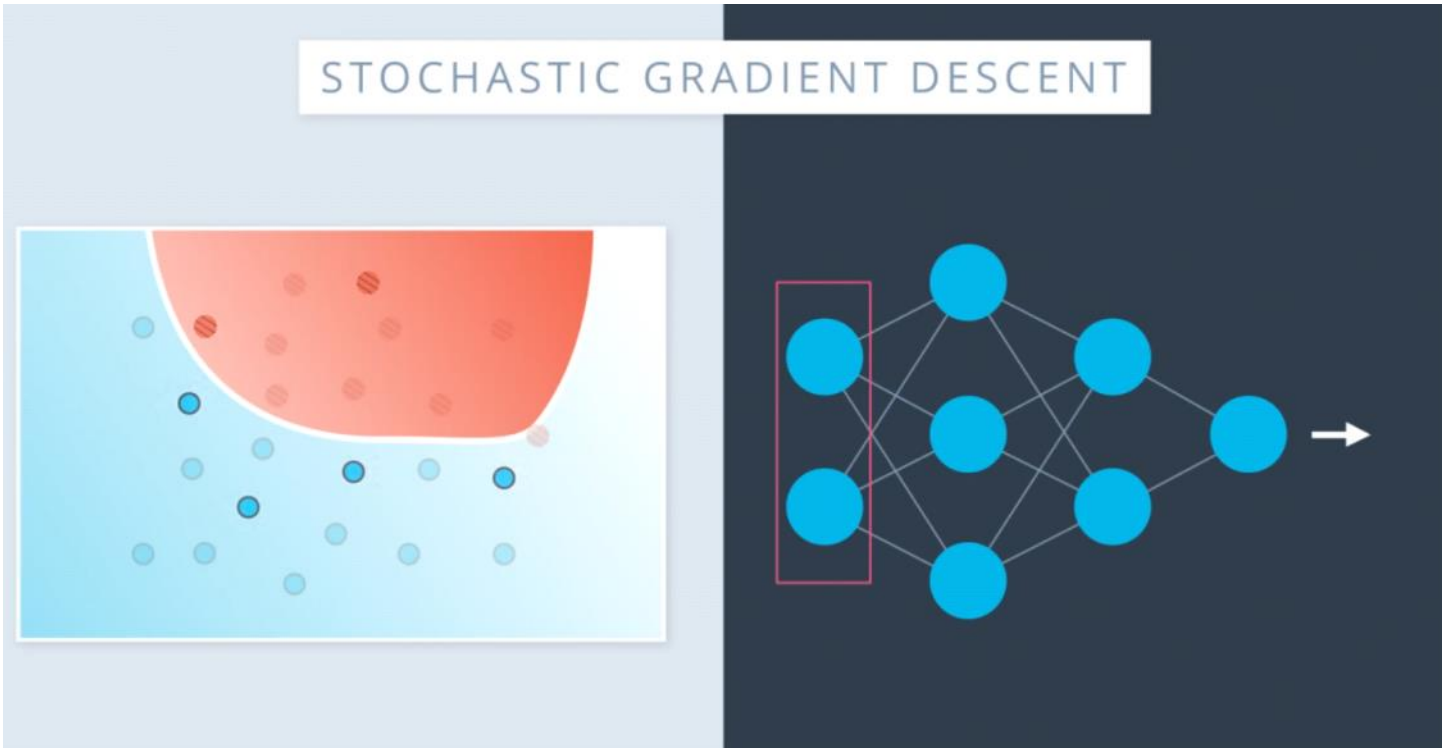
Screen clipping taken: 25/01/2021 09:13

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.



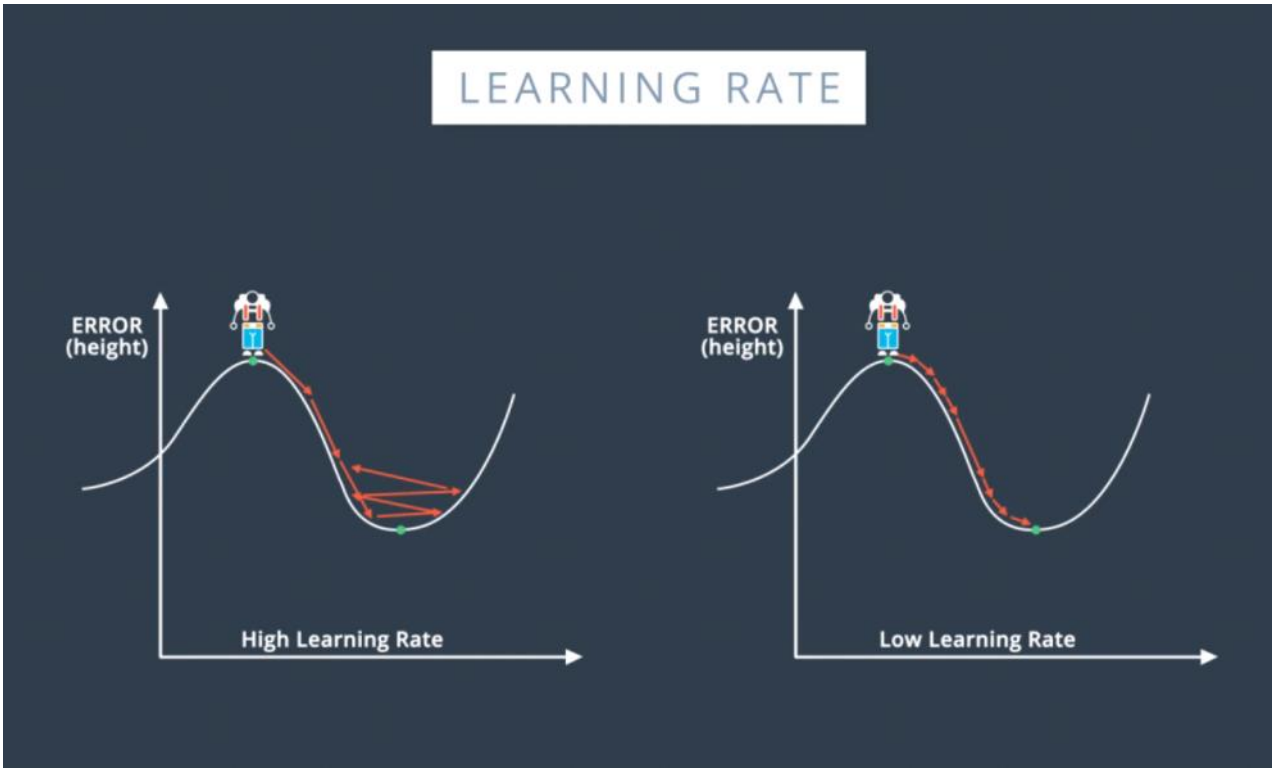
Screen clipping taken: 25/01/2021 09:14

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.



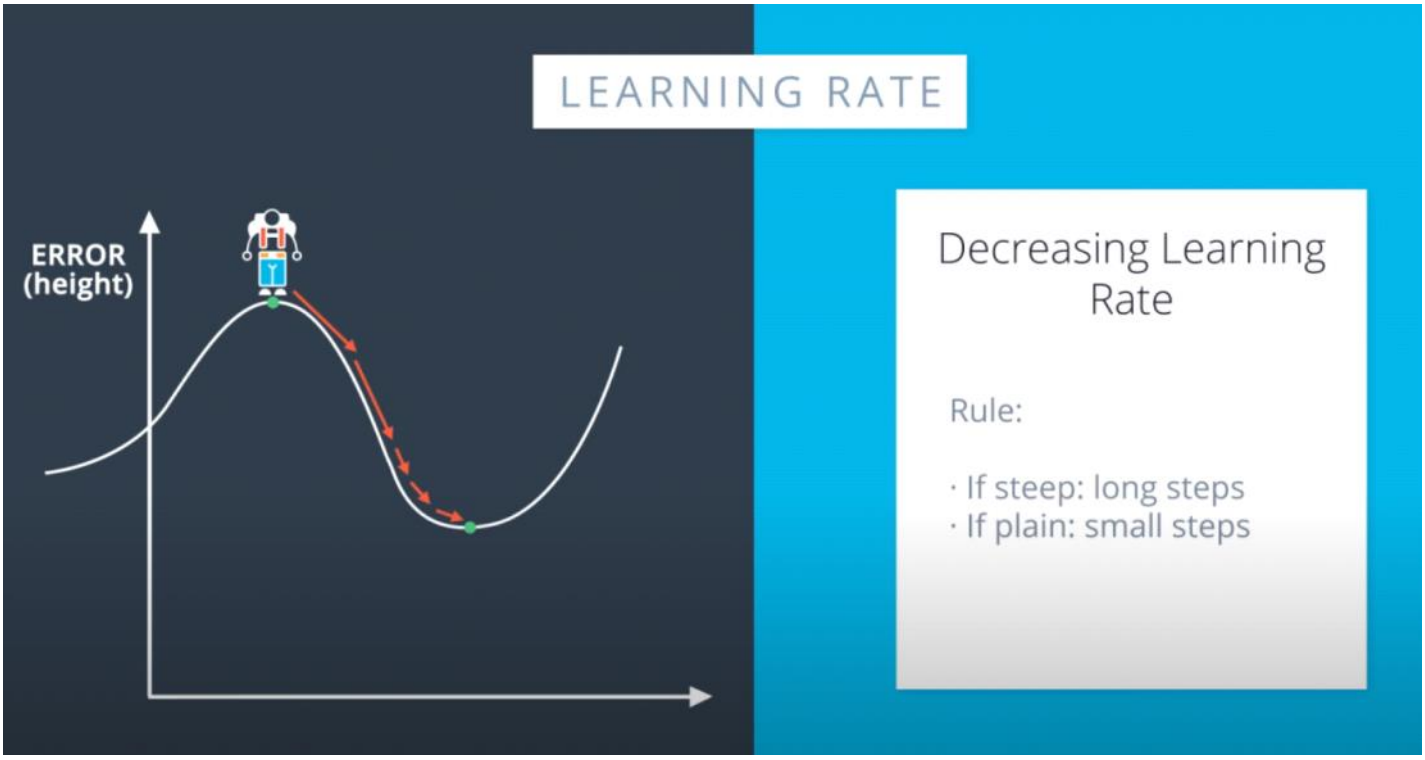
Screen clipping taken: 25/01/2021 09:17

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.



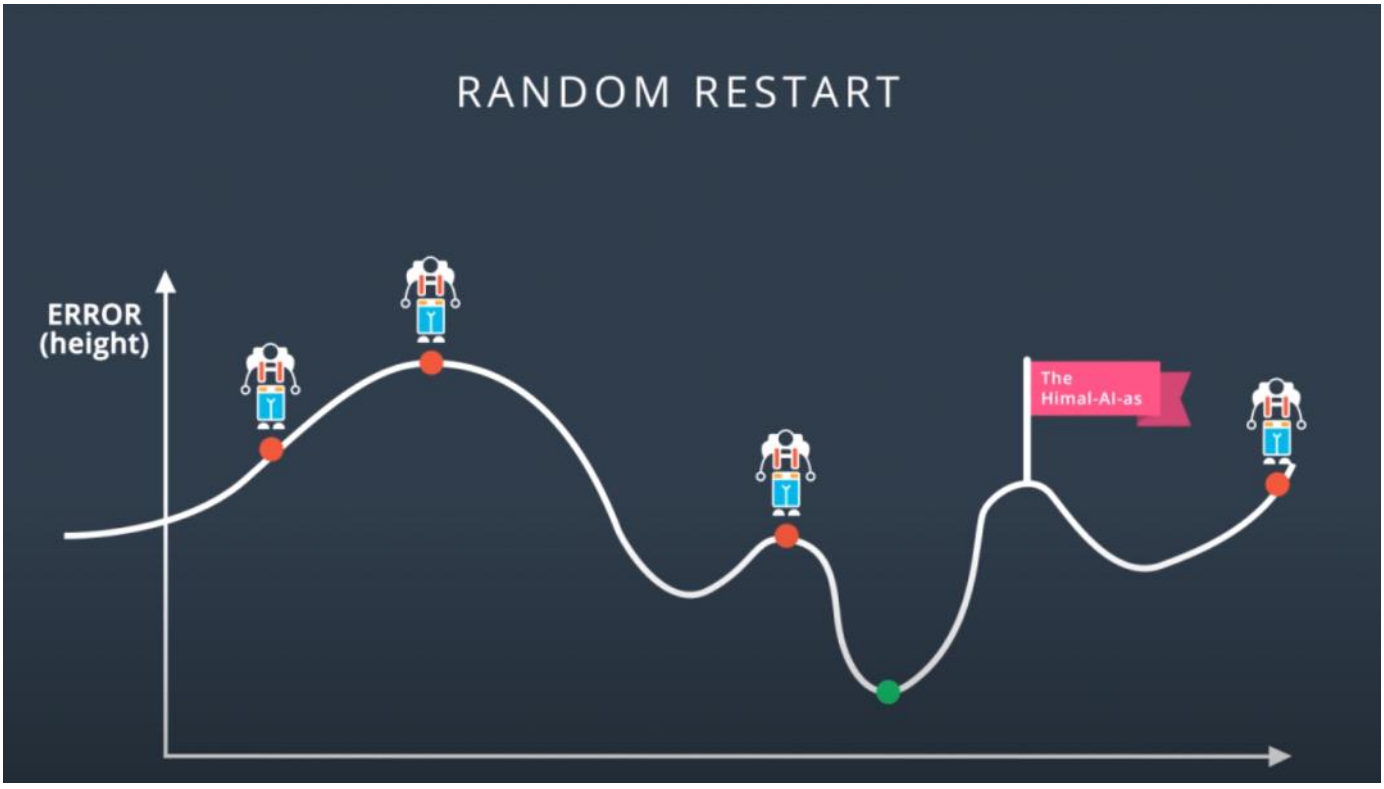
Screen clipping taken: 25/01/2021 09:18

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.



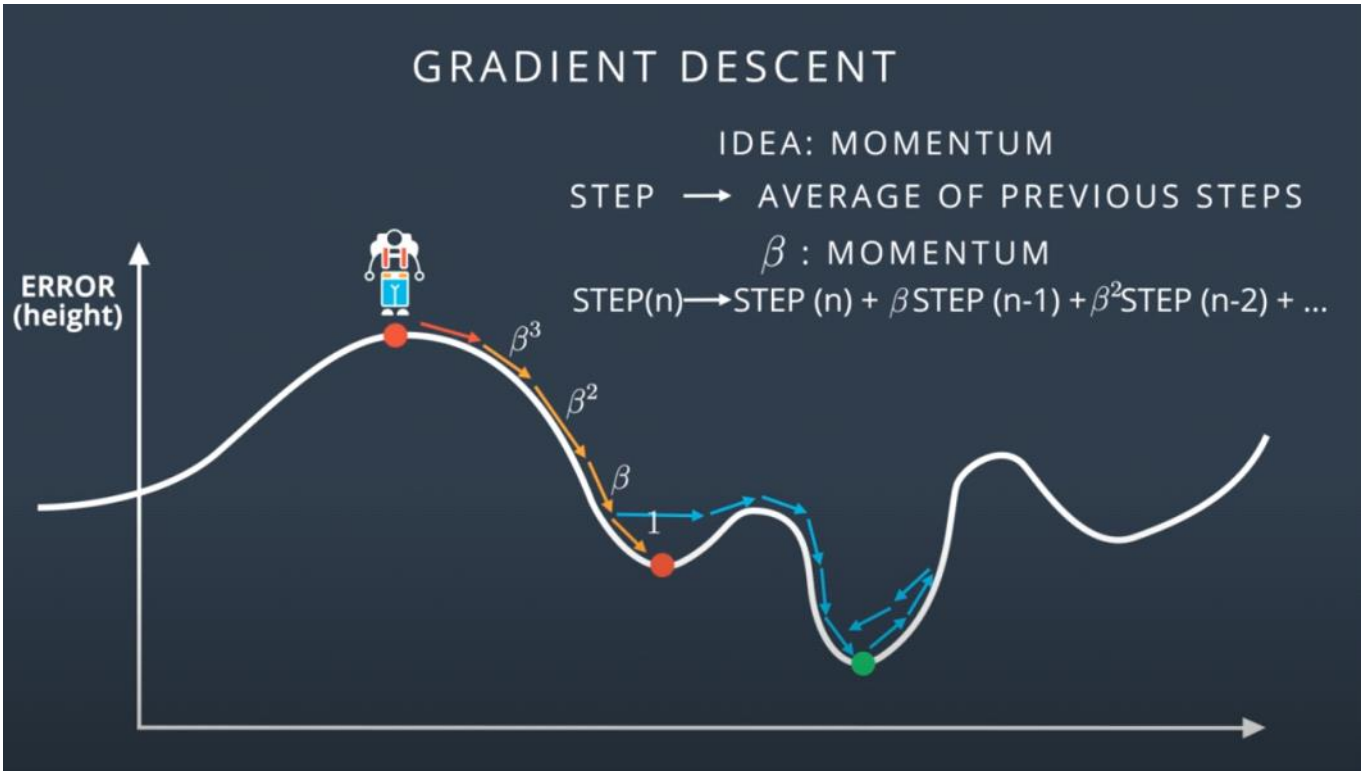
Screen clipping taken: 25/01/2021 09:18

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)



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