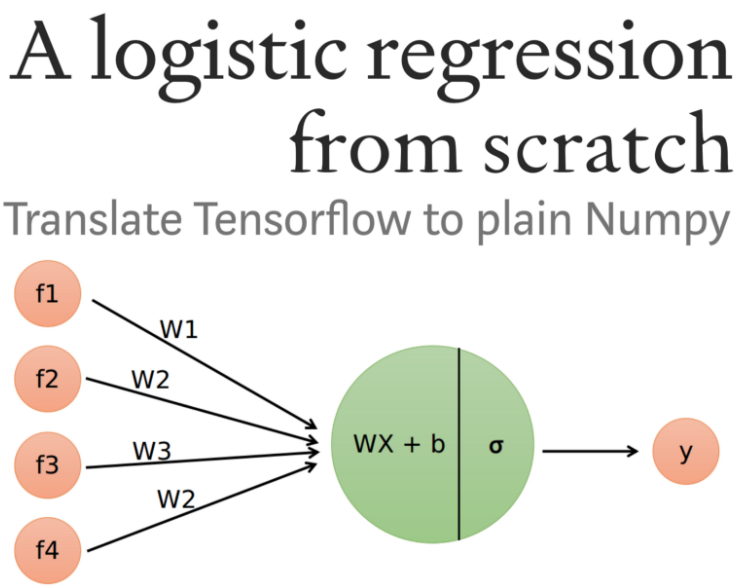


A logistic regression from scratch

Translate Tensorflow to plain Numpy

Dennis Bakhuis May 17, 2020 · 17 min read★

In my previous blog post we worked on artificial neural networks and developed a class to build networks with arbitrary numbers of layers and neurons. While the blog referenced to the previous notebook explaining the prerequisites, there was not yet the accompanying article, which is this blog post.

This post is also available as a Jupyter Notebook on my Github, so you can code along while reading.

If you are new to Python and Jupyter, here is a short explanation on how I manage my Python environment and packages.

A short overview of the topics we will be discussing:

1. Link between neural networks and logistic regression
2. One step back: linear regression

3. From linear to (binary) logistic regression

4. Round up

Link between neural network and logistic regression

When we hear or read about *deep learning* we generally mean the sub-field of machine learning using *artificial neural networks* (ANN). These computing systems are quite successful in solving complex problems in various fields, examples are, image recognition, language modelling, and speech recognition. While the name ANN implies that they are related to the inner workings of our brain, the truth is that they mainly share some terminology. An ANN generally consists of multiple interconnected layers, which on itself are build using neurons (also called nodes). An example is shown in figure 1.

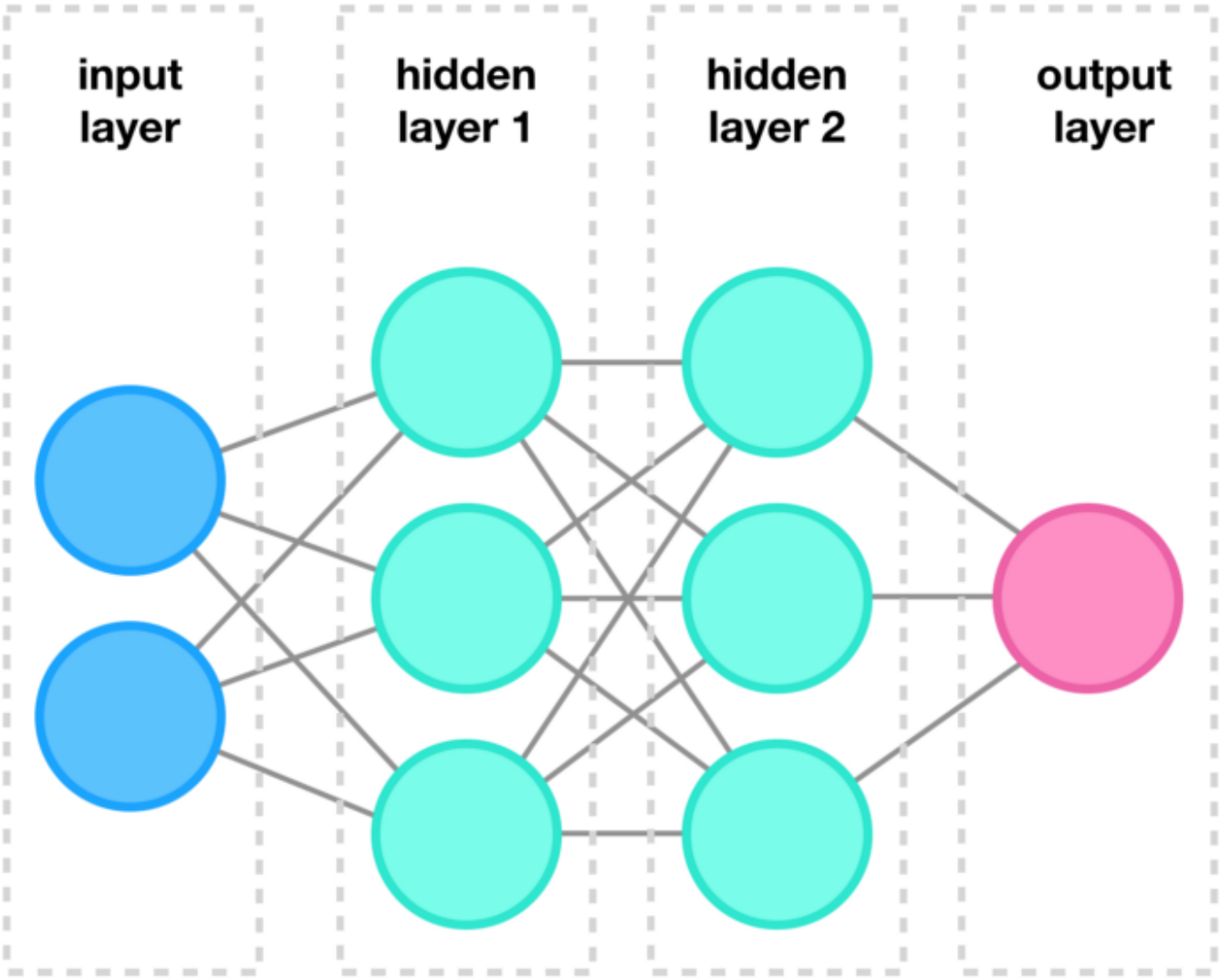


Figure 1: Example of a typical neural network, borrowed from my follow up article on neural networks.

In this example, we have one input layer, consisting of four individual inputs nodes. This input layer is 'fully connected' to the first hidden layer, i.e. Fully connected means that each input is connected to each node. The first hidden layer is again fully connected to another 'hidden' layer. The term hidden indicates that we are not directly interact with these layers and these are kind of obscured to the user. The second hidden layer is on its turn fully connected two the final output layer, which consists of two nodes. So in this example we feed the model four inputs and we will receive two outputs.

Let's now focus on a single neuron from our previous example. This neuron still is connected to all inputs, also called features. Using these features, the neuron calculates a single response (or output). A diagram of such a system is depicted in figure 2.

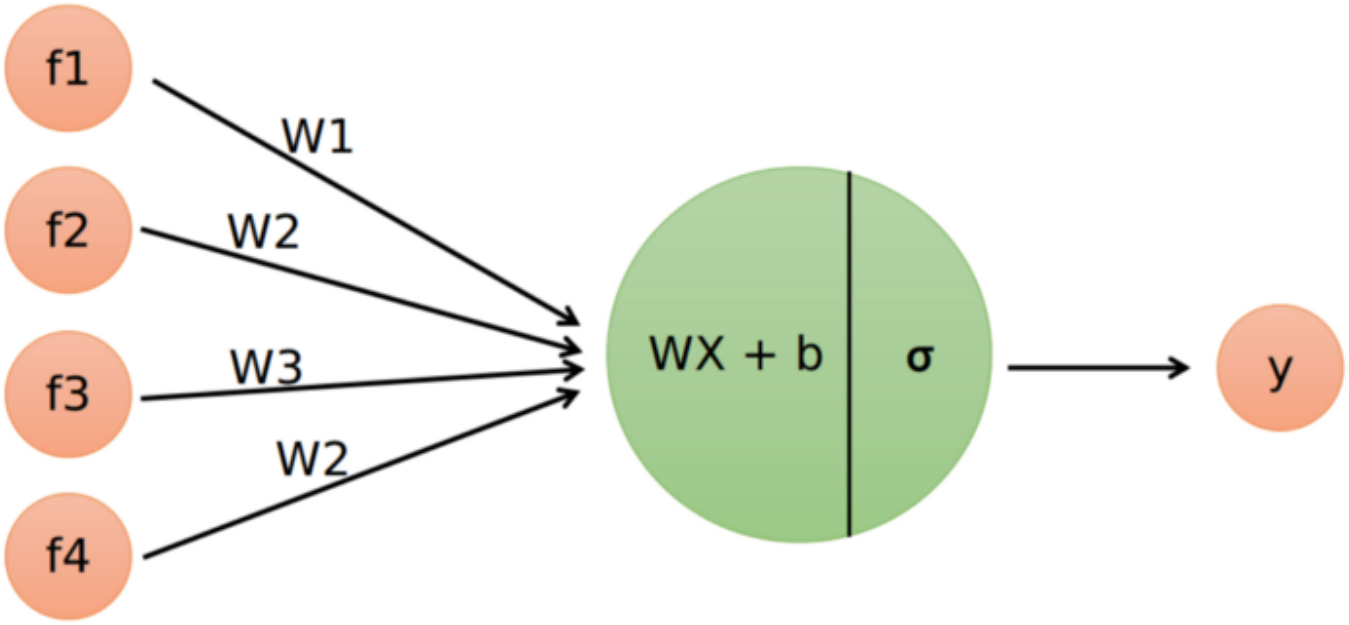


Figure 2: single neuron with four input features. The neuron has two operations: a linear part and the activation function.

The input features are named f^1, f^2, f^3, and f^4 and are all connected to the single neuron. This neuron executes two operations. The first operation is a multiplication of the features with a weight *vector* W and adding the bias term b. The second operation is a so called *activation function*, indicated here by σ. The output of the neuron is a probability between zero and one. The single neuron acts like a small logistic regression model and therefore, an ANN can be seen as a bunch of interconnected logistic regression models stacked together. While this idea is pretty neat, the underlying truth is a bit more subtle. There are many different architectures for ANNs and they can use various building blocks that act quite different than in this example.

The linear operation in our single neuron is nothing more than a linear regression. Therefore, to understand logistic regression, the first step is to have an idea how linear regression works. The next section will show a step by step example as a recap.

One step back: linear regression

What is a linear regression again?

Linear regression in its simplest form (also called simple linear regression), models a single dependent variable y using a single independent variable x. This may sound daunting, but was this means is that we want to solve the following equation:

$$y = ax + b$$

In the context of *machine learning*, x represents our input data, y represents our output data, and by solving we mean to find the best weights (or parameters), represented by w and b in the linear regression equation. A computer can help us find the best values for w and b, to have the closest match *for* y using the input variable x.

For the next examples, let us define the following values for x and y:

$$x = [-2,-1,0,1,2,3,4,5]$$
$$y = [-3,-1,1,3,5,7,9,11]$$

The values for x and y have a linear relation so we can use linear regression to find (or fit) the best weights and solve the problem. Maybe, by staring long enough at these values, we can discover the relation, however it is much easier to use a computer to find the answer.

If you have stared long enough or just want to know the answer, the relation between x and y is the following:

$$y = 2x + 1$$

In the next section we will use Tensorflow to create our single neuron model and try to 'solve' the equation.

An implementation in Tensorflow

Before we start with Tensorflow, we should first organize our input data (x) and output data (y). For this we are going to use Numpy:

In the next step, we will import Tensorflow. It is always a good practice to check which version we are using:

Now we can create a model using Keras, which is now a part of Tensorflow. To do this, we will use the Sequential class, which can stack various layers 'sequentially' after each other. We use the Dense class from Keras to create a 'fully connected' layer, which consists of a single neuron (unit).

The Dense function is used to create layers of many fully connected neurons (logistic units). The parameter units is used to set the amount of neurons. We only use a single unit and therefore, we will set it to one. As this is the first 'layer' of our model, we need to tell Tensorflow what shape it can expect as an input. This is only necessary for the first layer.

Now that we have defined the model, we need to use the Compile() method to configure the model for training. The method requires at least two parameters, a loss function and an optimizer. The loss function is a measure for how well the model predicts the actual value. For this example we will use the mean squared error (average of the squared difference between the predicted and the actual value of y). The '*learning algorithm*' will try to minimize the loss my adjusting (optimizing) the parameters (weights and bias) for each step. The optimizer defines a method to perform this Optimizing step and a common method is Gradient Descent, or in our case Stochastic Gradient Descent (SGD). This method will become more clear in the next section where we will implement it in plain Numpy.

Next, we use the Fit() method to let the algorithm learn the best parameters. While Tensorflow has many smart ways to address this problem, the recipe is more or less like this:

1. calculate the prediction $\hat{y}$ using the current parameters (W and b)
2. calculate the loss of the current values
3. calculate the gradients of the loss function with respect to the parameters
4. adjust the weights (optimize) using the gradients.
5. repeat for the number of epochs, i.e. the number of times to go through the provided examples (dataset).

Do not worry to much if this is not completely clear. We will code each step in plain Numpy, with a thorough derivation of all the math used in the next section.

We have trained the model in Tensorflow using our training data and we are ready to give it a spin. Now we can use our model to 'predict' values it has never seen. This is sometimes also called inference. Let's try the value of 12. We know that it should be 25.

Why is the value not exactly 25?

The model calculates the difference between the actual value and the predicted value and creeps slowly towards the actual value. Running the fit method for a longer time will get you closer to 25.

This was not that hard, but it might feel like some dark Jedi power. Therefore, in the next section we will implement this algorithm in plain Python (with the help of Numpy).

What actually is happening 'under the hood'

In the previous section we gave a rough overview what Tensorflow is doing under the hood:

1. calculate the prediction $\hat{y}$ using the current parameters (W and b)
2. calculate the loss of the current values
3. calculate the gradients of the loss function with respect to the parameters
4. adjust the weights (optimize) using the gradients.
5. repeat for the number of epochs, i.e. the number of times to go through the provided examples (dataset).

We will now implement exactly this in plain Python and hopefully come to a similar result as Tensorflow.

First we define the model parameters. These are the weights W, which is just a single value, because we only have a single input. Also we need to define the bias term b.

This are all the trainable parameters in our model, a single scalar for the weight and a single scalar for the bias.

Next we will define a function that makes a prediction using our current model parameters. In deep learning terms, this is called a forward pass. The variable of the predicted value is generally name $\hat{y}$ (or A but you can learn this in my next article).

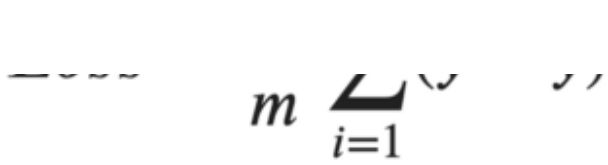
The function is named forward and uses the input vector X and multiplies it with the weight parameter W and adds the bias term b. Exactly as we described in the before-mentioned equation.

We can now test the function with the current parameters. Again, we will input a value of 12.0 but of course, it will return gibberish as the weights are randomly initialized.

```
-0.2122614846691141
```

Indeed, -0.212261… is not really what we were looking for, but we still have to train our model before it can make proper predictions. Before we can do that we need to calculate the current loss. As a loss we used the mean squared error of the predicted value $\hat{y}$ and the actual value y.

$$Loss = \frac{1}{m} \sum^{m} (y - \hat{y})^2$$



Hopefully the math does not scare you, but if you take the time, it is not that hard. The variable m here is the amount of examples (points in the dataset). Our X holds eight values and therefore, $m=8$. When we have a $1/m$ followed by as sum ($\sum$) over all m values, it is nothing more than an average of the values that are *inside* the summation. Here, we take the average over all $(y-\hat{y})^2$ which is the difference between the true value y and the predicted value $\hat{y}$ squared. The square is important because a negative difference and a positive difference would cancel each other out if we would not square the difference. Now that we fully understand the *mean squared error*, we can implement it in code:

To see what the loss is between our previously calculated value, we can do this:

```
149.13933056993267
```

As we can see, the loss is pretty large and our weights are quite off. Therefore, we need to update our trainable parameters to make better predictions. To do so, we need to first calculate the gradients $\delta\text{Loss}/\delta W$ and $\delta\text{Loss}/\delta b$. Maybe your differential skills are a bit rusty. The trick is to apply the chain rule. Another benefit is that the average operator (the sum) is linear and therefore, we can ignore it in the differential, and put it back later.

$$\frac{\delta}{\delta W} Loss = \frac{1}{m} \sum_{i=1}^{m} \frac{\delta}{\delta W} (y - \hat{y})^2$$

$$U = (y - \hat{y}) = (y - (Wx + b))$$

$$\frac{\delta}{\delta W} Loss = \frac{1}{m} \sum_{i=1}^{m} \frac{\delta}{\delta U} U^2 \frac{\delta}{\delta W} - (Wx + b)$$

$$\frac{\delta}{\delta W} Loss = \frac{1}{m} \sum_{i=1}^{m} -2x(y - \hat{y})$$

To calculate $\delta \text{Loss}/\delta b$ we only need to repeat the last step, with respect to b:

$$\frac{\delta}{\delta b} Loss = \frac{1}{m} \sum_{i=1}^{m} \frac{\delta}{\delta U} U^2 \frac{\delta}{\delta b} - (Wx + b)$$

$$\frac{\delta}{\delta b} Loss = \frac{1}{m} \sum_{i=1}^{m} -2(y - \hat{y})$$

We will implement this in the ‘backward pass’ function. As variable names get a bit long and we are a bit lazy, we will just call them dW and db.

So now we can get the gradient of our previous test example:

```
(-605.0942756320587, -50.42452296933823)
```

The last function we need, before we can compose our training loop is the update function. This function will 'optimize' our weights one step. This is the actual gradient descent in which we subtract (descent) the gradient from our current weights. Gradient descent is defined as follows:

$$W = W - \alpha \delta W$$
$$b = b - \alpha \delta b$$

Here we have a new parameter α which is called the learning rate. This value will set the speed in which we try to converge to the minimum loss. For this example we will set it to 0.01. There is no golden value for the learning rate and this value needs to be fine-tuned for each problem. Our code for the update function is as follows:

To update our current model parameters we simply do:

```
(array([6.0332543]), 0.5042452296933823)
```

Alright, we have updated our weights for the first time. To improve the weights, we have to repeat this process many times. For this we will write a loop:

```
129.7037237445872
0.06819476739792682
0.00458155318441586
0.0003078050915820409
2.067944441333496e-05
```

We are in luck and the loss, i.e. the difference between our model prediction and the actual value, is decreasing. How would we now predict, when we input a value of 12.0?

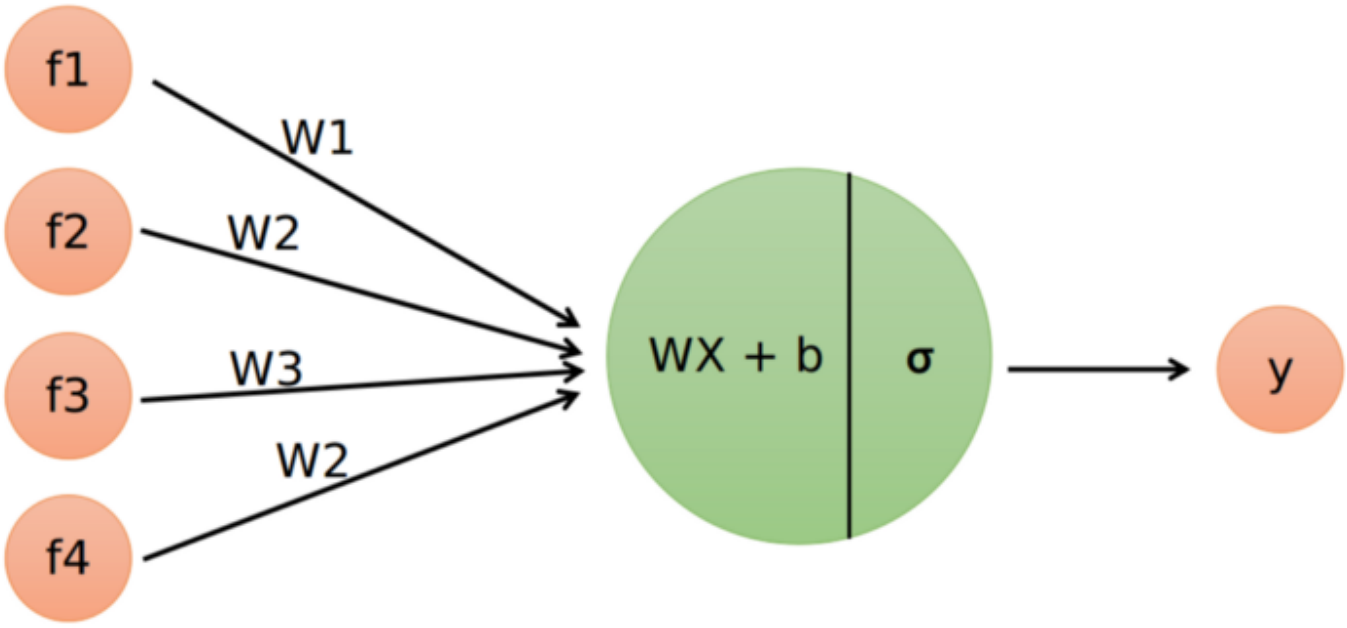
```
array([25.00101652])
```

I hope that this dark magic in Tensorflow is now a bit more clear. In the next section we will use our newly acquired knowledge to solve a binary logistic regression problem.

From linear to (binary) logistic regression

What is the difference exactly?

The differences between a linear regression and a logistic regression are not that major. There are two differences from the previous code we created. First, our linear regression model only had a single feature, which we inputted with x, meaning that we only had a single weight. In logistic regression, you generally input more than one feature, and each will have its own weight. Technically, you could have a single feature, but it would be nothing more than an if-statement (think about it). Increasing the number of features will change the previous *simple multiplication* to a *matrix multiplication* (dot product). Secondly, we will add a so called *activation function* to map this value between 0 or 1. Let's remind ourselves again of our simple model:



By convention in Tensorflow (from what I have understood), the input vector has columns for features, and rows for examples. If we would have 2 datapoints the input matrix would

look like this:

$$X = \begin{pmatrix} f_1^1 & f_1^2 & f_1^3 & f_1^4 \\ f_2^1 & f_2^2 & f_2^3 & f_2^4 \end{pmatrix}$$

The superscript shows the feature number, the subscript indicates the example.

Each of these inputs are associated with their own weights. The node itself has two distinct operations. The first is the dot product between the weights vector and the input vector. The second is the Sigmoid function. The weight vector W in this example has four weights:

$$X = \begin{pmatrix} W^1 \\ W^2 \\ W^3 \\ W^4 \end{pmatrix}$$

In the node we first compute the linear part:

$$Z = WX + b$$

For our example this will look like this:

$$Z = \begin{pmatrix} W^1 \\ W^2 \\ W^3 \end{pmatrix} \begin{pmatrix} f_1^1 & f_1^2 & f_1^3 & f_1^4 \\ f_2^1 & f_2^2 & f_2^3 & f_2^4 \end{pmatrix} + b$$

$$\begin{pmatrix} W^4 \end{pmatrix}$$

$$= \begin{pmatrix} W^1 f_1^1 + W^2 f_1^2 + W^3 f_1^3 + W^4 f_1^4 + b \\ W^1 f_1^1 + W^2 f_1^2 + W^3 f_1^3 + W^4 f_1^4 + b \end{pmatrix} = \begin{pmatrix} z_1 \\ z_2 \end{pmatrix}$$

Notice that the result is only a single value for each example.

For all values of x, which can be from $-\infty$ to $+\infty$ (all real numbers), the Sigmoid function maps x between 0 and 1. Values for x close to zero have the largest effect, as these are in the 'linear regime'. Very large, or very small are only clipped to 1 and 0 respectively. The mathematical definition of the Sigmoid function is:

$$A = \frac{1}{1 + e^{-z}}$$

This is all there is to the logistic unit. The Sigmoid function gives the node its non-linear character. Many of these units together can do almost magical things. In the next section we will first make a logistic regression model in Tensorflow.

An implementation in Tensorflow

Before we can start we first need some data to do a logistic regression. I downloaded the titantic dataset from Azeem Bootwala from Kaggle to have a play for this example. It can be downloaded from here:
https://www.kaggle.com/azeembootwala/titanic

```
(792, 17)
```

First, always inspect the columns and data types:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 792 entries, 0 to 791
Data columns (total 17 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   Unnamed: 0   792 non-null    int64  
 1   PassengerId  792 non-null    int64  
 2   Survived     792 non-null    int64  
 3   Sex          792 non-null    int64  
 4   Age          792 non-null    float64
 5   Fare         792 non-null    float64
 6   Pclass_1     792 non-null    int64  
 7   Pclass_2     792 non-null    int64  
 8   Pclass_3     792 non-null    int64  
 9   Family_size  792 non-null    float64
 10  Title_1      792 non-null    int64  
 11  Title_2      792 non-null    int64  
 12  Title_3      792 non-null    int64  
 13  Title_4      792 non-null    int64  
 14  Emb_1        792 non-null    int64  
 15  Emb_2        792 non-null    int64  
 16  Emb_3        792 non-null    int64  
dtypes: float64(3), int64(14)
memory usage: 105.3 KB
```

I expect that Azeem did not save the set using index=False option and therefore we have a ‘Unnamed: 0’ column. This one is redundant with our current index so we can remove it. Also the PassengerId is not very useful for our model, let us remove that column too. After that, let us sample the dataset to get an idea:

	Survived	Sex	Age	Fare	Pclass_1	Pclass_2	Pclass_3	Family_size	Title_1	Title_2	Title_3	Title_4	Emb_1	Emb_2	Emb_3
207	1	1	0.3250	0.036671	0	0	1	0.0	1	0	0	0	1	0	0
229	0	0	0.3500	0.049708	0	0	1	0.4	0	0	0	1	0	0	1
753	0	1	0.2875	0.015412	0	0	1	0.0	1	0	0	0	0	0	1
650	0	1	0.3500	0.015412	0	0	1	0.0	1	0	0	0	0	0	1
380	1	0	0.5250	0.444099	1	0	0	0.0	0	0	0	1	1	0	0

Azeem already did some preprocessing. The target variable Y is 'Survived', all other columns are features. A short description of the features:

- Sex: 0 or 1 -> male or female
- age: value rescaled between 0 and 1
- fare: ticket price rescaled between 0 and 1
- Pclass_1 .. Pclass_3: One-hot encoded Passenger class
- family size: rescaled value between 0 and 1 of family size.
- title_1 .. title_4: mr, mrs, master, miss one-hot encoded
- emb_1 .. emb_3: Embark location one-hot encoded.

In total we will have 14 features.

For this example, the data will suffice and I will not go into detail on how this data has become what it is. Honestly, I do not know myself and have just downloaded it from Kaggle ;-).

Lets put these variables in the format we defined before (X and Y). Here Y corresponds to the label if a person has survived. We have a total of 792 examples. Therefore, the shape for Y is $(m,1)$ where $m = 792$. For X we expect $(m, 14)$, where the columns are the features.

```
((792, 14), (792,))
```

Now that we have prepared the data we can create a model in Tensorflow:

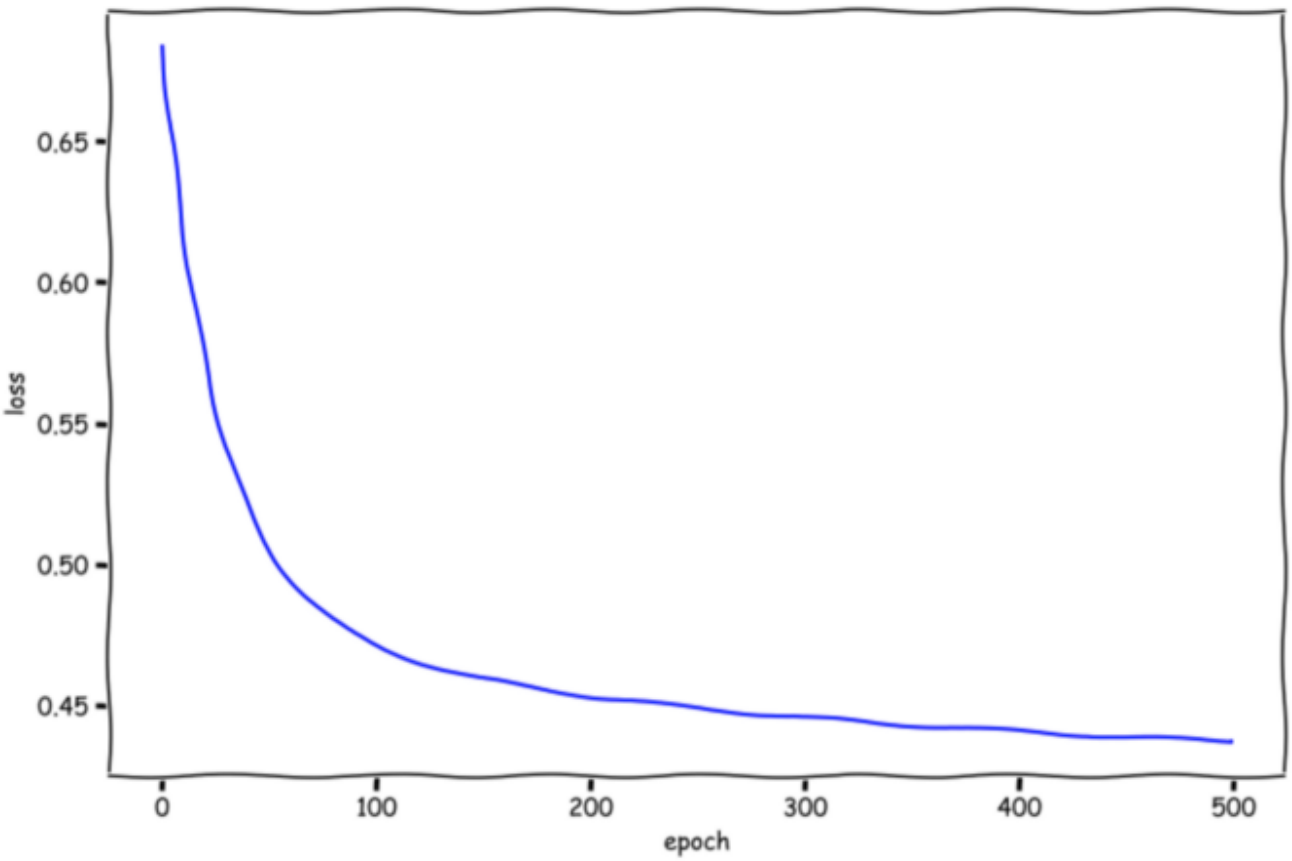
The model in Tensorflow is very similar to our linear regression model. The input has changed from 1 to 14 features and we added the Sigmoid activation function. Next, we must again compile our model:

In this example I changed the loss into 'binary_cross_entropy'. This is another loss function which works better for binary logistic regression problems. If you are interested in the inner workings I recommend wikipedia.
We added an additional metric called 'accuracy' which is now calculated for each epoch.
Now we are ready to train our model:

Well, these were all steps required for training a binary logistic classifier in Tensorflow. We achieved an accuracy of ~80% which is not too bad for the effort we put in.

We can extract the weights W and the bias b from our model. These values we can later compare to our own implementation of the logistic regression:

Plotting the loss is also quite easy:



Don't mind the nice XKCD wobble :-). It is quite impressive how few steps are required to get to such a result. In the final section we will unveil the Dark Jedi arts being performed by Tensorflow.

What is actually happening?

Well, the general recipe has not changed:

1. forward pass

2. calculate loss
3. backward pass
4. update weights
5. repeat

The steps itself, need minor modification. The forward pass will be the more general dot product and we need to add the activation function. The loss function is binary cross entropy, which is of course different from the mean square error. The backward pass will calculate the gradient of the new loss function with respect to W and b. As we now also have an activation function, we will have an additional step. The update-weights function is unchanged, the final loop is also very similar.

In Numpy, and in math in general as far as I know, the dot product needs the shapes of the vectors (and matrices) to be compatible:

$$XY = X_{i,j}Y_{j,k}$$

This means that the number of columns of X must be equal to the number of rows in Y. To make ourselves a bit easier, we will Transpose our input vector X and flip the vector. This will result that the rows will be features and the columns will be examples.

```
(14, 792)
```

For testing, let us just select two examples, to make it a bit more readable:

```
(14, 2)
```

Lets define our weights. As we have 14 features, our vector W will have 14 values. The bias is a constant for the whole node, and is only a single value.

```
(14, 1)
```

Next, we need to define the Sigmoid function:

```
array([3.72007598e-44, 5.00000000e-01, 5.24979187e-01, 1.00000000e+00])
```

Now lets redefine our forward function, and make it use the dot product and the activation function. We can split these in two steps:

$$Z = WX + b$$
$$A = \sigma(Z)$$

Note that WX is a dot product.

```
array([[0.48951622, 0.50149394]])
```

Now that we have actual predictions, we can write our loss function to measure how well our predictions are. This is done with the binary cross entropy, for which I will simply give the equation:

$$loss = -\frac{1}{m}\sum_{i=1}^{m} y\log(A) + (1-y)\log(1-A)$$

It might happen we try to calculate a log(0) which is of course not defined. To avoid the warning, we will add a tiny value to our loss. As it is super small, the difference is not noticeable, but does help suppress the warning. One less warning a day keeps the ….

```
1.3625601508159457
```

Next is the backwards pass. For this, We would need to differentiate the Loss function with W and b. Not to bore you guys, I have provided these functions, but I will not stop you from calculating the differentials yourself:

$$\frac{\partial loss}{\partial W} = \frac{1}{m}\sum_{i=1}^{m} X(A-Y)^T$$

$$\frac{\partial loss}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} (A-y)$$

In Python, this looks like this:

```
(array([[ 0.06118953],
        [-0.01277168],
        [-0.0078041 ],
```

```
        [-0.06231326],
        [ 0.        ],
        [ 0.06118953],
        [-0.00011237],
        [-0.00112373],
        [ 0.        ],
        [ 0.        ],
        [ 0.        ],
        [-0.06231326],
        [ 0.        ],
        [ 0.06118953]]),
 -0.0011237299781017493)
```

Almost there, next we need to update the weights. The function has not changed from our previous example:

```
(array([[-0.01830035],
        [ 0.00088324],
        [-0.01122826],
        [-0.00589117],
        [-0.00893116],
        [-0.01335291],
        [-0.00061042],
        [ 0.00065638],
        [ 0.00410113],
        [-0.00572882],
        [-0.00801334],
        [ 0.01374348],
        [ 0.01274699],
        [-0.01275547]]),
 1.1237299781017494e-05)
```

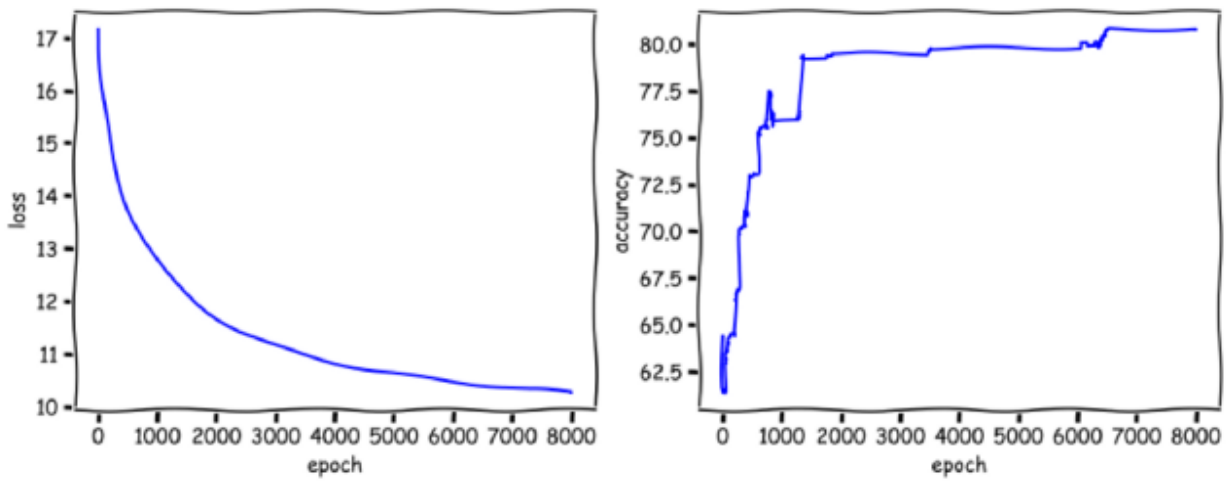
To compare the results, we can calculate the accuracy. However our activation function returns a probability between 0 and 1. By definition, values <= 0.5 are rounded to 0 and values > 0.5 are rounded to 1. This is slightly different from the regular round function so we will make our own function for this:

```
array([[0, 1]], dtype=uint8)
```

Now we have everything to create our training loop. We will store some metrics to plot afterwards:

```
loss: 17.16646271874957  accuracy: 64.4%
loss: 12.757526841623513         accuracy: 75.9%
loss: 11.688361064742441         accuracy: 79.5%
loss: 11.150776615503466         accuracy: 79.5%
loss: 10.83231532525657  accuracy: 79.8%
loss: 10.622380143288218         accuracy: 79.8%
loss: 10.473049466548982         accuracy: 79.8%
loss: 10.36043263717339  accuracy: 80.8%
```

We trained the network and got a final accuracy of just above 80%, very similar results as when using Tensorflow. Of course, with a bit more effort, but also with a bit more fun. Let's plot our metrics:



Of course, we have calculated the metrics on the training data. To get a proper idea on the model, this should be done using the test data. This, I leave open to the reader ;-).