My First Steps in Neuronal Networks (Beginners Guide)

Axel Roth

2021-11-11

Contents

Chapter 1

About

1.1 Me

Hello, my name is Axel Roth and im studding math at a master degree in germany and working half-time in the finance field as something between a data-analyst and a full stack developer. I already got a lot of experience in coding with R and all its features, but i never wrote a line of code in python and i never touched Neuronal Networks bevor. So Why do i want to write a beginners guide in the field of Neuronal Networks in python?
Its simple, at the moment i have a lecture in that we learn how to program a Neuronal Network from scratch with basic packages from python and i want to share my experience. Additionally i learned all i know from free sources of the internet and thats why i want to give something back. Furthermore its a good use-case to write my first things in english and test the fancy Bookdown and GitBook features.

1.2 The Book

This Book will be more or the less the documentation of my lecture "Finance Project" in that we learn to program a simple Perceptron (the simplest Neuronal Network) and then we will continue with a multi layer Perceptron and finish with a slight insight into decision trees. On this journey, we will test the Neuronal Network in different examples that are easy to reproduce. Because these are my first steps in this field, i need to appolagice for my terrible spelling and cant guarantee you the best quality, but maybe this is the best way to educate and attract unexperienced readers to have a look into this field.

1.3 How it works

Im coding this book in the IDE R-Studio with the framework of Bookdown and embed python code that is made possible by the reticulate package. This is the reason, why i need to load the python interpreter in the following R-chunk:

```
library(reticulate)
Sys.setenv(RETICULATE_PYTHON = "D:\\WinPython2\\WPy64-3950\\python-3.9.5.amd64\\")
```

Additionaly im using a fully portable version of R-Studio, R and python. Its nice to have if you want to switch for example between university PCs and your own. R-Studio supports it and python can be downloaded via WinPython to be fully portable. All the Neural Network pictures are handmade by me with draw.io, its a free to use website. If you are new to python and never used R i would recommend to use jupyter lab or PyCharm.

Chapter 2

Single Perceptron

In this chapter i will teach you how to code a single Perceptron in python with only the numpy package. Numpy uses a vectorizible math structure in which you can easily calculate elementwise or do stuff like normal matrix multiplications with just a symbol (i always interpret Vectors as one dimensional matrices!). At the most of the time, its just translating math formulas into python code without changing its structure.
First of all we are starting with the needed parameters, that are explained later:

Number of iterations over all the training dataset := `epochs`
Learning rate := α = `alpha`
Bias value := β = `bias` and the activation function:

$$step(s) = \begin{cases} 1, & s \geq \beta \\ 0, & s < \beta \end{cases}$$

This function is named the heavyside-function and should be the easiest activation function to start with. If the weighted sum is smaller than the bias β, it will send the value zero to the next neuron. Our brain works with the same behavior. If the electricity is too small, the neuron will not activate and the next one dosent get any electricity.

The traing dataset is the following:

$$\left[\begin{array}{cc|c} x_{i,1} & x_{i,2} & y_i \end{array}\right]$$

$$\left[\begin{array}{cc|c} 0 & 0 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 1 \end{array}\right]$$

The provided training dataset contains the `X` matrix with two inputs for each scenario and the `Y` matrix with the correct output (each row contains the input and output of one scenario). If your looking exactly you can see that this is the OR-Gate. Later you will see why these type of problems are the only suitable things to do with a single neuron.

The needed python imports are the following:

```
import numpy as np
import matplotlib.pyplot as pyplot
```

(Do you see more imports than only the numpy package? Yes or No)

Now that we have all the needed parameters and settings, i can give you a quick overview of the algorithm.

2.1 Neural Network Basics

In a NN we are having two basic parts, the forward pass and the backward pass. In the forward pass we are calculating the weighted sum of each input neuron with its weights of the layer and evaluating the activation function with it, to calculate the output. In the backward pass we are analyzing the error to adjust the weights accordingly. This is it! This is all a NN will do. I explained everything to you. Have a good life...
Ahh no no ok we have a deeper look into it :)

Whats exactly is the forward pass in a single Perceptron? Its just the evaluation of the acvtivation function with the weighted sum like i said, so you have for one scenario out of the training dataset, the following:

$$step(W \cdot x_i^T) = y_i$$

That is the normal approach to use this formula to iterate over all scenarios in the training dataset...
But i think its not the right way to describe it, because it gets very confusing to interpret it for all scenarios at the same time.

My next approach is to consider all scenarios in the training dataset in one formula. If your data isnt that huge, its a much faster approach as well. First of all we need to interpret the new dimensions of W and X.
We have X as:

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

each row describes the inputs for each neuron in the scenario i.
For the weights W we have for example:

$$W = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$

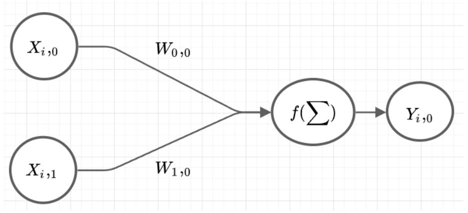
The new formular looks like this:

$$step(X * W) = Y$$

The $*$ symbol defines a matrix to matrix multiplication. For example if you take a look at the i-th row (scenario) of X you will see the following:

$$Y_{i,0} = step([X_{i,0} \cdot W_{0,0} + X_{i,1} \cdot W_{1,0}])$$

and $Y_{i,0}$ is the approximated output of the i-th scenario. Now we can look at the NN and compare the formula with it:



Yes it is the same, its the weighted sum of the inputs and evaluated the activation function with it to calculate the output of the scenario i.

2.2 Forward pass

Now we create the so called `forward()` function in python:

```
def forward(x, w):
  return( step(x @ w) )
```

(Numpy provides us with the `@` symbol to make a matrix to matrix multiplication and the `.T` to transpose)

Because we want to put one dimensional matrices into the `step()` function we need to use numpy for the if-else statement:

```
def step(s):
  return( np.where(s >= bias, 1, 0) )
```

Here an small example for the forward pass:

```
X = np.array([
  [0,0],
  [0,1],
  [1,0],
  [1,1],
])
W = np.array([
  [0.1],
  [0.2]
])
bias = 1
Y_approx = forward(X, W)
print(Y_approx)
```

```
## [[0]
##  [0]
##  [0]
##  [0]]
```

And these are all the generated outputs of our NN over all scenarios. Now we need to calculate the error and adjust the weights accordingly.

2.3 backward pass

To adjust the weights in a single Perceptron, we need the Delta-Rule:

$$W(t+1) = W(t) + \Delta W(t)$$

with

$$\Delta W(t) = \alpha \cdot X^T * (Y - \hat{Y})$$

and $\hat{Y}$ is the output of the NN.

```
def backward(W, X, Y, alpha, Y_approx):
    return(W + alpha * X.T @ (Y - Y_approx))
```

with the result of the forward pass and example data we have the following:

```
Y = np.array([
  [0],
  [1],
  [1],
  [1]
])
alpha = 0.01
W = backward(W, X, Y, alpha, Y_approx)
print(W)
```

```
## [[0.12]
##  [0.22]]
```

and this is the new weight.

2.4 Single Perceptron

Now we want to do the same process multiple times, to train the NN:

```
X = np.array([
  [0,0],
  [0,1],
  [1,0],
  [1,1],
])
Y = np.array([
  [0],
  [1],
  [1],
  [1]
])
W = np.array([
  [0.1],
  [0.2]
])
alpha = 0.01
bias = 1
epochs = 100

errors = []
for i in range(epochs):
  Y_approx = forward(X, W)
  errors.append(Y - Y_approx)
  W = backward(W, X, Y, alpha, Y_approx)
```

The KNN is trained. Now we want to look at the errors of each epoch. We want to measure the mean-square-error with the following formula:

$$Error_i = \frac{1}{2} \cdot \sum (Y - \hat{Y})^2$$

or as python code:

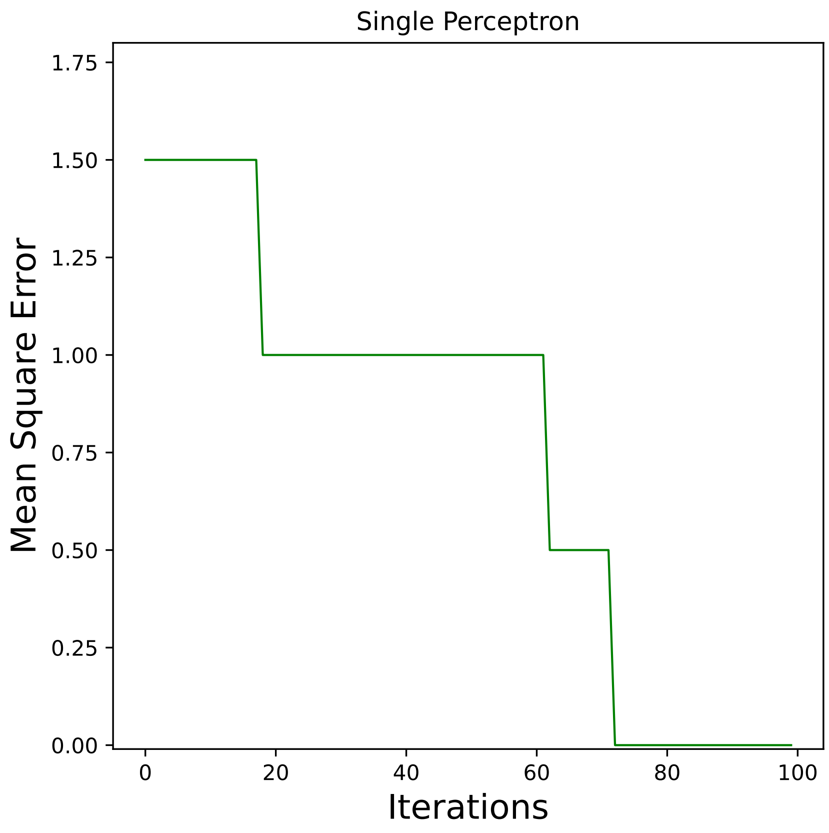
```python
def mean_square_error(error):
  return( 0.5 * np.sum(error ** 2) )
```

Now we need to calculate the mean-square-error for each element in the list `errors` which is made with `map()`:

```python
mean_square_errors = np.array(list(map(mean_square_error, errors)))
```

To plot the errors, im using the following function:

```python
def plot_error(errors, title):
  x = list(range(len(errors)))
  y = np.array(errors)
  pyplot.figure(figsize=(6,6))
  pyplot.plot(x, y, "g", linewidth=1)
  pyplot.xlabel("Iterations", fontsize = 16)
  pyplot.ylabel("Mean Square Error", fontsize = 16)
  pyplot.title(title)
  pyplot.ylim(-0.01,max(errors)*1.2)
  pyplot.show()


plot_error(mean_square_errors, "Single Perceptron")
```



If you survived until now, you have learned how you can program a single Perceptron!

2.5 Appendix (complete code)

```
import numpy as np
import matplotlib.pyplot as pyplot


X = np.array([
  [0,0],
  [0,1],
  [1,0],
  [1,1],
])
Y = np.array([
  [0],
  [1],
  [1],
  [1]
])
W = np.array([
  [0.1],
  [0.2]
])
alpha = 0.01
bias = 1
train_n = 100

def step(s):
  return( np.where(s >= bias, 1, 0) )


def forward(X, W):
  return( step(X @ W) )

def backward(W, X, Y, alpha, Y_approx):
  return(W + alpha * X.T @ (Y - Y_approx))


errors = []
for i in range(train_n):
  Y_approx = forward(X, W)
  errors.append(Y - Y_approx)
  W = backward(W, X, Y, alpha, Y_approx)



def mean_square_error(error):
  return( 0.5 * np.sum(error ** 2) )


mean_square_errors = np.array(list(map(mean_square_error, errors)))


def plot_error(errors, title):
  x = list(range(len(errors)))
  y = np.array(errors)
  pyplot.figure(figsize=(6,6))
  pyplot.plot(x, y, "g", linewidth=1)
  pyplot.xlabel("Iterations", fontsize = 16)
  pyplot.ylabel("Mean Square Error", fontsize = 16)
  pyplot.title(title)
  pyplot.ylim(-0.01,max(errors)*1.2)
  pyplot.show()


plot_error(mean_square_errors, "Single Perceptron")
```

Chapter 3

Adding trainable Bias

The single Perceptron, you saw in the previews chapter had the following activation function:

$$step(s) = \begin{cases} 1, & s \geq \beta \\ 0, & s < \beta \end{cases}$$

with $\beta = 1$ and that is just the right β for the given training dataset. But what happens if you shift the training data for example adding -5 to the X matrix? Now it will never find the correct answer. That is because you need to select the β accordingly. But this wouldnt be intelligent to search for each dataset the optimal β by hand.

3.1 Generalising the Bias

That is why we now generalise the weighted sum step by step. First we generalise the activation function:

$$step(s) = \begin{cases} 1, & s \geq 0 \\ 0, & s < 0 \end{cases}$$

Now we can list it in the weighted sum:

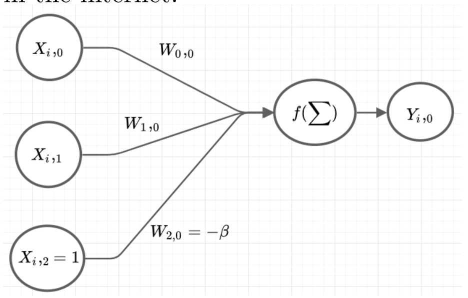
$$step(X * W - \beta) = Y$$

But we have the same problem as previews, because we need to specify the β explicit. To add it to the training process we add a column with ones on the right side of X and add the $-\beta$ to the last row of W. The output of one scenario is now caluclated as the following:

$$Y_{i,0} = step([X_{i,0} \cdot W_{0,0} + X_{i,1} \cdot W_{1,0} + X_{i,2} \cdot W_{2,0}]) = step([X_{i,0} \cdot W_{0,0} + X_{i,1} \cdot W_{1,0} - \beta])$$

Furthermore because the W is holding the β, it now gets re-adjusted in the backward pass so it is involved in the training process. Thats why we now generate a random number for it, because it gets corrected anyway.

Now we have a NN that looks like all the other pictures of a single Perceptron in the internet:



As python code we did the following:

```
X = np.array([
  [0,0],
  [0,1],
  [1,0],
  [1,1],
])-5
X = np.append(X, np.array([np.ones(len(X))]).T, axis=1)

W = np.array([
  [0.1],
  [0.2]
])
W = np.append(W, -np.array([np.random.random(len(W[0]))]).T, axis=0)
```

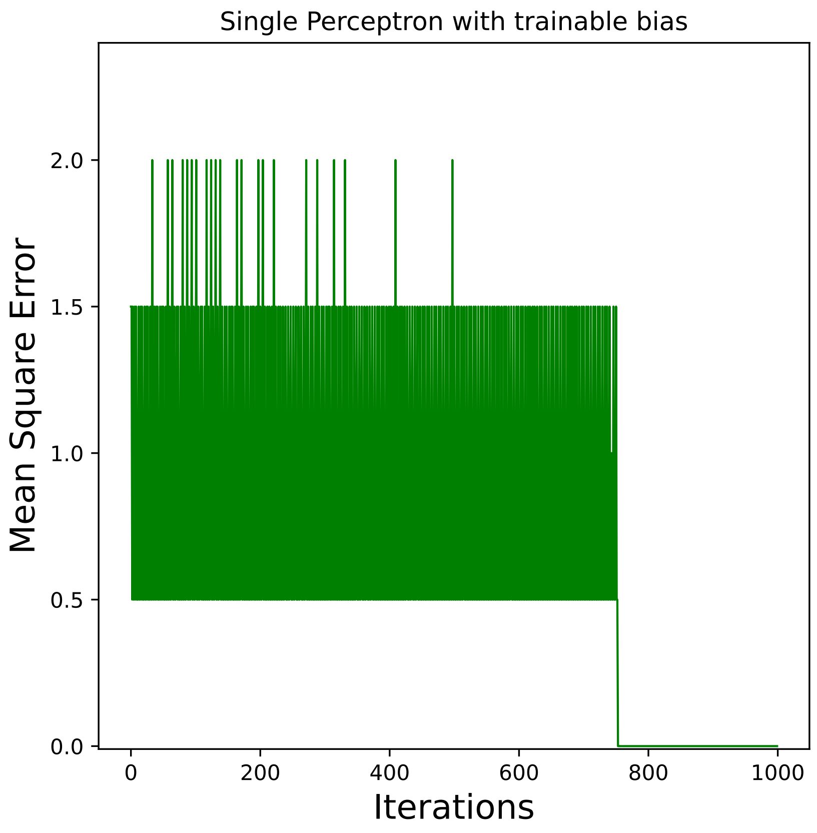
We added -5 to the X matrix to simulate the problem of shifted data, added ones on the left side of X and added negative random numbers between $(0, 1)$ to the weights. Yes, if you would have a clue, what β would be great for the given problem, its better to choose it explicit. The new problem needs more epochs, because it needs to find a good β by it self.

3.2 Appendix (complete code)

The complete code is the following:

```
X = np.array([
  [0,0],
  [0,1],
  [1,0],
  [1,1],
]) - 5
X = np.append(X, np.array([np.ones(len(X))]).T, axis=1)

W = np.array([
  [0.1],
  [0.2]
])
W = np.append(W, -np.array([np.random.random(len(W[0]))]).T, axis=0)

Y = np.array([
  [0],
  [1],
  [1],
  [1]
])

alpha = 0.01
epochs = 1000

def step(s):
  return( np.where(s >= 0, 1, 0) )


def forward(X, W):
  return( step(X @ W) )

def backward(W, X, Y, alpha, Y_approx):
    return(W + alpha * X.T @ (Y - Y_approx))


errors = []
for i in range(epochs):
  Y_approx = forward(X, W)
  errors.append(Y - Y_approx)
  W = backward(W, X, Y, alpha, Y_approx)



def mean_square_error(error):
  return( 0.5 * np.sum(error ** 2) )
```

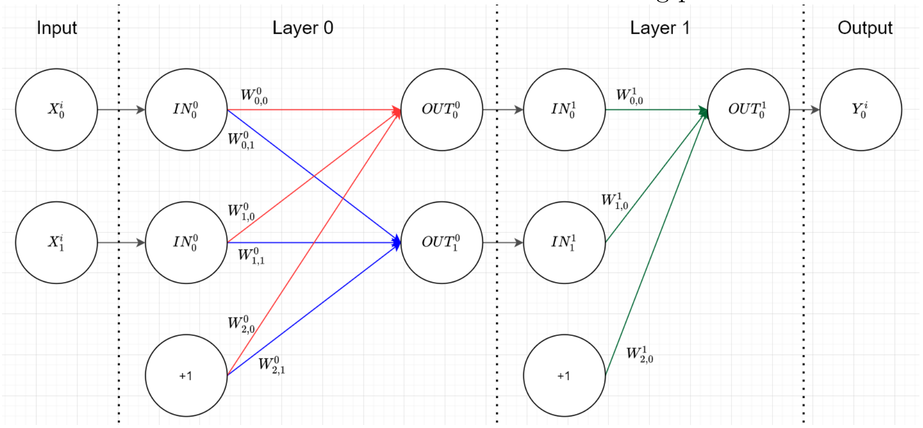
```
mean_square_errors = np.array(list(map(mean_square_error, errors)))


def plot_error(errors, title):
  x = list(range(len(errors)))
  y = np.array(errors)
  pyplot.figure(figsize=(6,6))
  pyplot.plot(x, y, "g", linewidth=1)
  pyplot.xlabel("Iterations", fontsize = 16)
  pyplot.ylabel("Mean Square Error", fontsize = 16)
  pyplot.title(title)
  pyplot.ylim(-0.01,max(errors)*1.2)
  pyplot.show()


plot_error(mean_square_errors, "Single Perceptron with trainable bias")
```


Single Perceptron with trainable bias
Mean Square Error
Iterations
0.0
0.5
1.0
1.5
2.0
0
200
400
600
800
1000

Chapter 4

Multi Layer Perceptrons (MLP)

In a multi layer Perceptron you have multiple layers of neurons. Thats why we need to calculate the forward pass multiple times and the same for the backward pass. First of all, do we need to generalise some definitions, to support this behavior. What we want to do is the NN of the following picture:



It is my own definition of layers, because i thought, it would be better to display layers like shown in the picture, to take the step from n to $n+1$ hidden layers more easy. You can see that each layer has the same process in the forward pass by evaluating $f(IN^{layer} \cdot W^{layer}) = OUT^{layer}$ and just passing the result to the next layer like $OUT^{layer} = IN^{layer+1}$, with f as the chosen activation function. We will choose the sigmoid function as the activation function f, because it has an easy deviation $f^{`}$ for the backward pass and its very close to the behavior of the heavyside function.

```
def sigmoid(x):
  return 1.0 / (1.0 + np.exp(-x))

def deriv_sigmoid(x):
  return x * (1 - x)
```

Additionaly will we choose the XOR-Gate as training dataset and generate the weights in a very generic approach like the following code shows:

```
X = np.array([
  [0,0],
  [0,1],
  [1,0],
  [1,1],
])

Y = np.array([
  [0],
  [1],
  [1],
  [0]
])

n_input = len(X[0]) + 1
n_output = len(Y[0])
hidden_layer_neurons = np.array([2]) # the 2 means that there is one hidden layer with

def generate_weights(n_input, n_output, hidden_layer_neurons):
  W = []
  for i in range(len(hidden_layer_neurons)+1):
    if i == 0: # first layer
      W.append(np.random.random((n_input, hidden_layer_neurons[i])))
    elif i == len(hidden_layer_neurons): # last layer
      W.append(np.random.random((hidden_layer_neurons[i-1]+1, n_output)))
    else: # middle layers
      W.append(np.random.random((hidden_layer_neurons[i-1]+1, hidden_layer_neurons[i]))

  return(W)

W = generate_weights(n_input, n_output, hidden_layer_neurons)

print("W[0]: \n", W[0])
print("W[1]: \n", W[1])
```

```
## W[0]:
```

```
##  [[0.77566019 0.2808405 ]
##  [0.65719859 0.02353656]
##  [0.74029955 0.62351251]]
## W[1]:
##  [[0.94738253]
##  [0.20108027]
##  [0.43594548]]
```

The input and output layer neurons are calculated from the training dataset and the neurons from the hidden layers are generated with the `hidden_layer_neurons`. For example can we generate two hidden layers with 4 and 2 neurons by `hidden_layer_neurons = np.array([4,2])`. I didnt explicitly choose the bias because it gets corrected anyway.
Now we need to define a helper function to add the biases, on the last column of the inputs, with:

```
def add_ones_to_input(x):
  return(np.append(x, np.array([np.ones(len(x))]).T, axis=1))
```

4.1 forward pass

The new forward function looks exactly like in the single Perceptron:

```
def forward(x, w):
  return( sigmoid(x @ w) )
```

Now we have everything to calculate the forward pass of the NN from above with the generated weights step by step:

```
IN = []
OUT = []

# layer 0
i = 0
IN.append( add_ones_to_input(X) )
OUT.append( forward(IN[i], W[i]) )

# layer 1
i = 1
IN.append( add_ones_to_input(OUT[i-1]) )
OUT.append( forward(IN[i], W[i]) )

# error
Y-OUT[-1]
```

```
## array([[-0.76999594],
##        [ 0.20956953],
##        [ 0.20491031],
##        [-0.80699813]])
```

Thats all! We calculated the forward pass in a very generic way for the NN with 2 input neurons, 2 hidden neurons and 1 output neuron for all 4 scenarios at the same time. Sadly is the forward pass the easiest part of the multi layer Perceptron :)

4.2 backward pass

We will adjust the weights with the backpropagation algorithm what is a special case of the descent gradient algorithm. In the output layer is it done by calculating the sensitives of the outputs according to the activation function multiplied with the error that occured. On all other layers its calculated by passing backwards the earlier calculated gradient splitted up on each neuron by the previes weights and multiplied by the sensitivity of the outputs of that layer according to the activation function. The formula is the following:

$$grad^i = \begin{cases} f^{‘}(OUT^i) \cdot (Y - OUT^i), & i = \text{last layer} \\ f^{‘}(OUT^i) \cdot (grad^{i+1} * \widetilde{W}^{i+1\ T}), & \text{else} \end{cases}$$

with $\widetilde{W}$ as the weights of that layer without the connection to the bias neuron, because it has no connection to the previews neurons. In our datastructur its done by removing the last row.
With the example from above is it done by the following lines of code:

```
grad = [None] * 2

# layer 1
i = 1
grad[i] = deriv_sigmoid(OUT[i]) * (Y-OUT[i])

# layer 0
i = 0
grad[i] = deriv_sigmoid(OUT[i]) *(grad[i+1] @ W[i+1][0:len(W[i+1])-1].T) # without bia
```

Now you can look for example at the gradient of the last layer and on the direction it is shown:

```
print("Y: \n",Y)
print("OUT: \n",OUT[1])
print("grad: \n",grad[1])
```

```
## Y:
##  [[0]
##  [1]
##  [1]
##  [0]]
## OUT:
##  [[0.76999594]
##  [0.79043047]
##  [0.79508969]
##  [0.80699813]]
## grad:
##  [[-0.13636797]
##  [ 0.03471522]
##  [ 0.03338441]
##  [-0.12569169]]
```

You can see that the gradient shows in the direction that would drift the output OUT closer to the desired output Y. That is exactly what the gradient descent algorithm is doing.
Now do we need to adjust the weights with the gradients and the learningrate α according to the direction of the gradients with the following formula:

$$W^i_{new} = W^i_{old} + \alpha \cdot (IN^{i\ T} * grad^i)$$

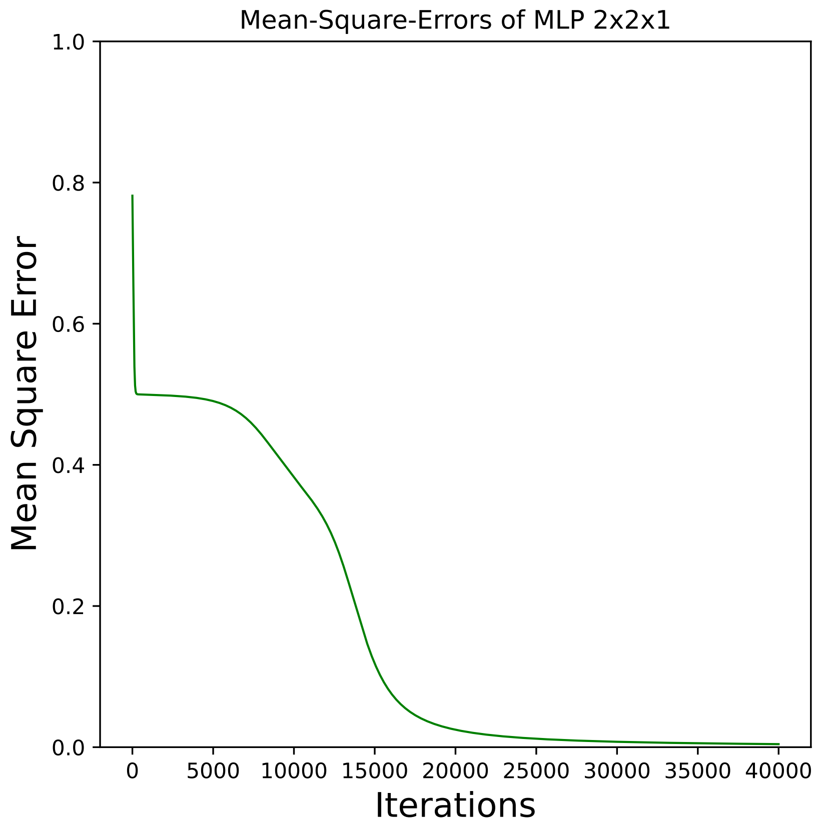
In the example from above we have the following results for the adjusted weights after the first iteration:

```
alpha = 0.03

W[1] = W[1] + alpha * (IN[1].T @ grad[1])
W[0] = W[0] + alpha * (IN[0].T @ grad[0])
```

This was the process of the forward pass and backward pass for one epoch in the multi layer Perceptron with 2 input neurons 2 hidden layer neurons an done output neuron. It is simple to use the given code from above to create a more generic NN for dynamic hidden layers and with dynamic training datasets for the given amount of epochs like in the appendix below.

4.3 Appendix (complete code)

```
import numpy as np
import matplotlib.pyplot as pyplot
np.random.seed(0)

X = np.array([
  [1,1],
  [0,1],
  [1,0],
  [0,0],
])

Y = np.array([
  [0],
  [1],
  [1],
  [0]
])

n_input = len(X[0]) + 1
n_output = len(Y[0])
hidden_layer_neurons = np.array([2])


def generate_weights(n_input, n_output, hidden_layer_neurons):
  W = []
  for i in range(len(hidden_layer_neurons)+1):
    if i == 0: # first layer
      W.append(np.random.random((n_input, hidden_layer_neurons[i])))
    elif i == len(hidden_layer_neurons): # last layer
      W.append(np.random.random((hidden_layer_neurons[i-1]+1, n_output)))
    else: # middle layers
      W.append(np.random.random((hidden_layer_neurons[i-1]+1, hidden_layer_neurons[i]))
  return(W)

def add_ones_to_input(x):
  return(np.append(x, np.array([np.ones(len(x))]).T, axis=1))


W = generate_weights(n_input, n_output, hidden_layer_neurons)


def sigmoid(x):
  return 1.0 / (1.0 + np.exp(-x))

def deriv_sigmoid(x):
  return x * (1 - x)


def forward(x, w):
  return( sigmoid(x @ w) )

def backward(IN, OUT, W, Y, grad, k):
  if k == len(grad)-1:
    grad[k] = deriv_sigmoid(OUT[k]) * (Y-OUT[k])
  else:
    grad[k] = deriv_sigmoid(OUT[k]) *(grad[k+1] @ W[k+1][0:len(W[k+1])-1].T)
  return(grad)

eta = 0.03
errors = []
for i in range(40000):
  IN = []
  OUT = []
  grad = [None]*len(W)
  for k in range(len(W)):
    if k==0:
      IN.append(add_ones_to_input(X))
    else:
      IN.append(add_ones_to_input(OUT[k-1]))
    OUT.append(forward(x=IN[k], w=W[k]))

  errors.append(Y - OUT[-1])

  for k in range(len(W)-1,-1, -1):
    grad = backward(IN, OUT, W, Y, grad, k)

  for k in range(len(W)):
    W[k] = W[k] + eta * (IN[k].T @ grad[k])



def mean_square_error(error):
  return( 0.5 * np.sum(error ** 2) )

mean_square_errors = np.array(list(map(mean_square_error, errors)))

def plot_error(errors, title):
    x = list(range(len(errors)))
    y = np.array(errors)
    pyplot.figure(figsize=(6,6))
    pyplot.plot(x, y, "g", linewidth=1)
    pyplot.xlabel("Iterations", fontsize = 16)
    pyplot.ylabel("Mean Square Error", fontsize = 16)
    pyplot.title(title)
    pyplot.ylim(0,1)
    pyplot.show()

plot_error(mean_square_errors, "Mean-Square-Errors of MLP 2x2x1")
```



Chapter 5

MLP example (Credit Default)

Now we can use our generic MLP model to forecast real life credit defaults.