My First Steps in Neuronal Networks (Beginners Guide)

Axel Roth

2021-11-14

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 im having 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 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 appologice 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 do we start with the necessary 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 training 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 do we calculate 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 do we 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 :)

What 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 of the training dataset, the following:

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

This 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 do 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 do we have for example:

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

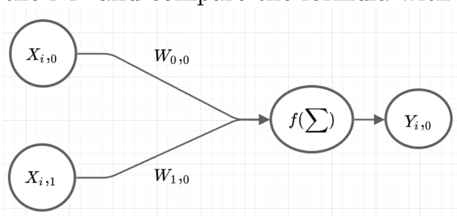
The new formula looks like this:

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

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 can we 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 can 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, its necessary to use numpy for the if-else statement:

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

In the next step will we create 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 do we need to calculate the error and adjust the weights accordingly.

2.3 backward pass

We need the Delta-Rule to adjust the weights in a single Perceptron:

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

with

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

and $\hat{Y}$ is the output of the NN. Translatet to code it is:

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

With the result of the forward pass and and the correct outputs, do 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 these are the new weight.

2.4 Single Perceptron

Now do 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. In the next step do we analyze the errors of each epoch. The best way to consider all the errors of each scenario is to measure the mean-square-error with the following formula:

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

or as python code:

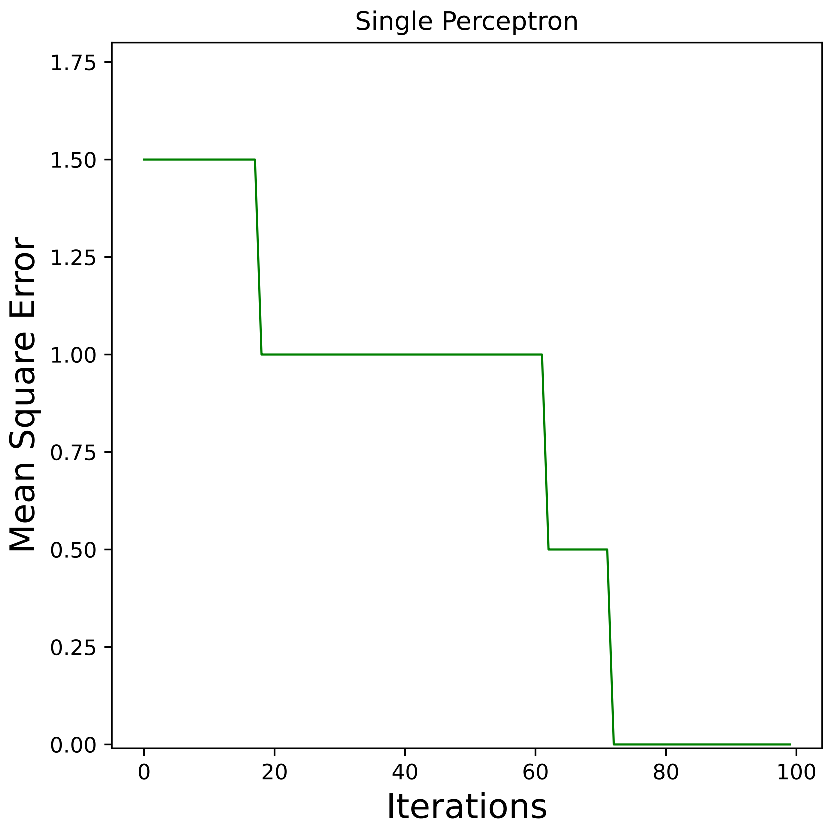
```
def mean_square_error(error):
  return( 0.5 * np.sum(error ** 2) )
```

Now do we need to calculate the mean-square-error for each element in the list `errors` which can be performed with `map()`:

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

To plot the errors, im using the following function:

```
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 to 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

First of all do we need to generalise the use of the bias, stating with the generalisation of the activation function:

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

Now can we list it in the weighted sum:

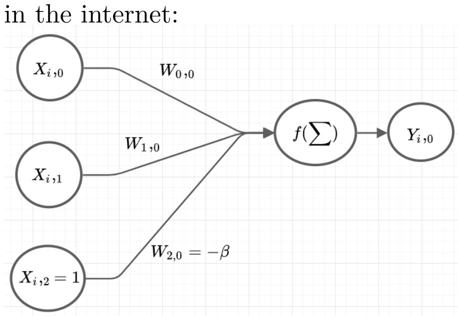
$$step(X * W - \beta) = Y$$

But we have the same problem as previews, because we need to specify the β explicit. Adding the bias to the training process is done by adding ones on the right side of the X matrix and adding the negative bias to the last row of W. The output of one scenario is calculated as the following:

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

The resulting NN includes the bias into the re-adjusting process of the backward pass, because of that we will generate a random number for the bias that will be corrected anyway.

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



The same step can be made with the following python code:

```
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)

print("X: \n", X)
print("W: \n", W)
```

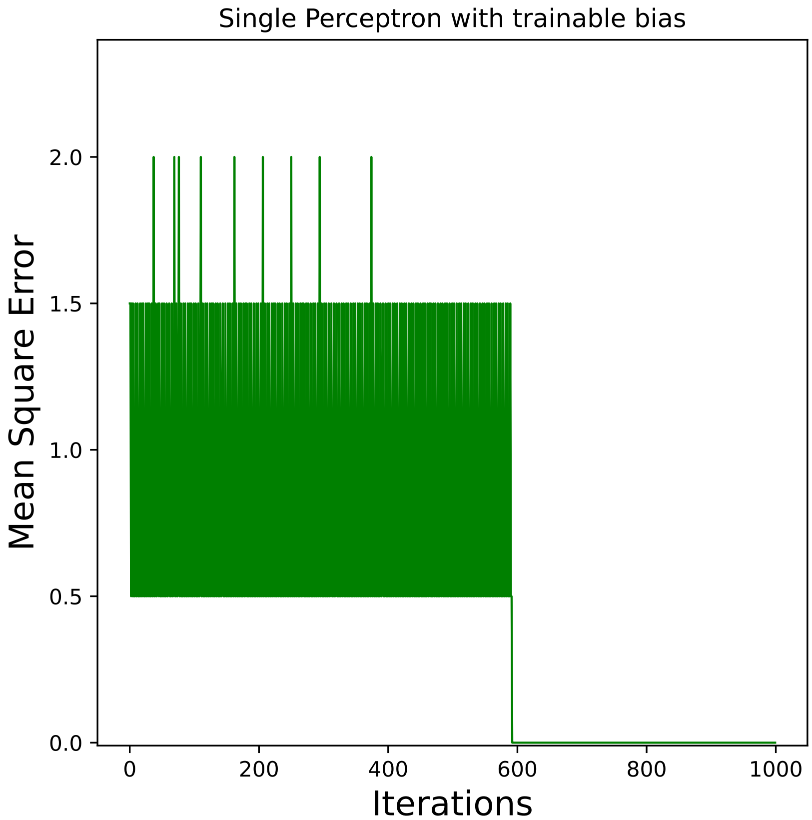
We added -5 to the X matrix to simulate the problem of shifted data, added ones on the left side of X and added negative random numbers in $(0, 1)$ to the weights. Yes, if you would have a clue, what β would be great for the given problem, its better to choose it explicit. The new NN is slower, because 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):
```

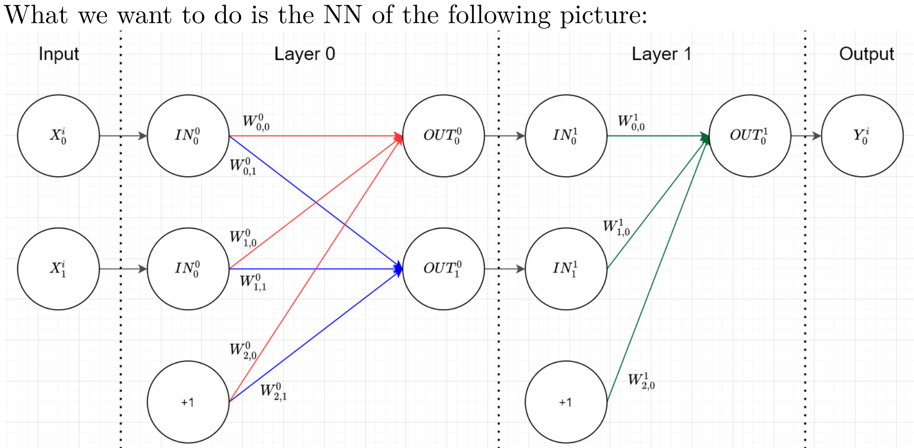
```
  return( 0.5 * np.sum(error ** 2) )


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)

A multi layer Perceptron contains 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 generalize 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 with an output between 0 and 1.

```
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 random weights in a very generic approach:

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

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

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

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+1,
↪   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]:
##  [[0.14641446 0.24580899]
##  [0.36141235 0.50416487]
##  [0.99357406 0.23792218]]
## W[1]:
##  [[0.35047968]
##  [0.46937943]
##  [0.62655433]]
```

The input and output layer neurons are calculated from the training dataset and the neurons for the hidden layers are generated with the `hidden_layer_neurons` array. 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 do 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 structur of 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([[-0.75856162],
##        [ 0.22738294],
##        [ 0.23460722],
##        [-0.77819776]])
```

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 that 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 occurred. On all other layers its calculated by passing backwards the earlier calculated gradient, splitted up on each neuron by the previews 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 the layer i 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, it can be done with:

```
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 bias weights
```

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

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

```
## Y:
##  [[0]
##  [1]
##  [1]
##  [0]]
## OUT:
##  [[0.75856162]
##  [0.77261706]
##  [0.76539278]
##  [0.77819776]]
## grad:
##  [[-0.13892744]
##  [ 0.03994662]
##  [ 0.04212764]
##  [-0.13432161]]
```

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.
In the next step 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:

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

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

```
alpha = 0.03

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

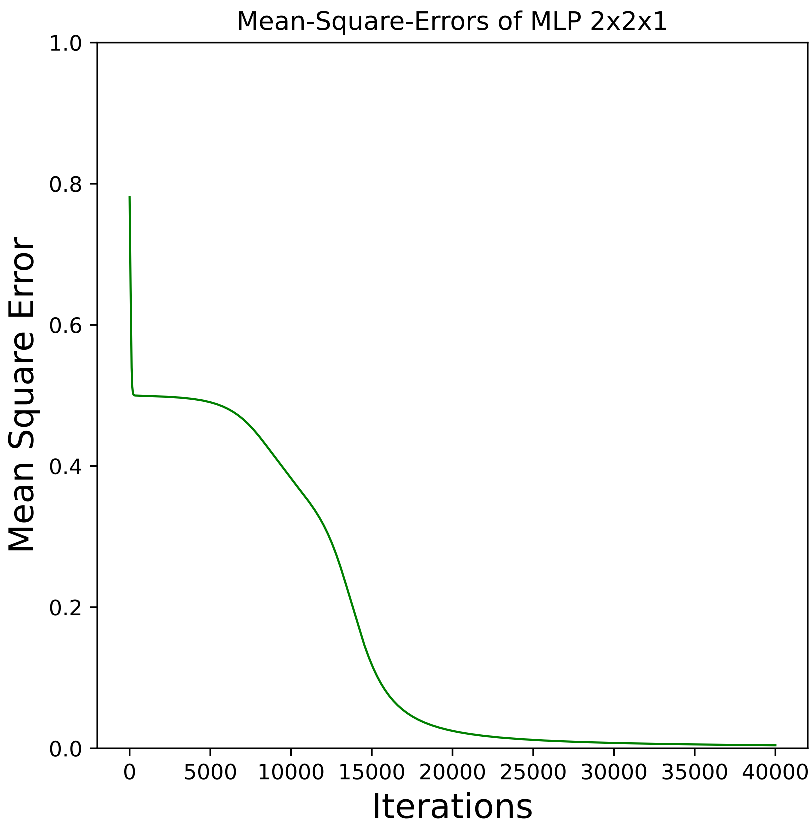
This was the process of the forward pass and backward pass for one epoch in the multi layer Perceptron with 2 input neurons 2 hidden layer neurons and one output neuron. It is simple to use the given code from above to create a more generic NN for dynamic number of hidden layers and with dynamic training 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])
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 + 1,
↪   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)

alpha = 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] + alpha * (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")
```


Mean-Square-Errors of MLP 2x2x1
Mean Square Error
Iterations
1.0
0.8
0.6
0.4
0.2
0.0
0
5000
10000
15000
20000
25000
30000
35000
40000

Chapter 5

MLP example (Credit Default)

Now can we use our generic MLP model from the previews chapter to forecast real life credit defaults. The csv can be downloaded from Kaggle Data Source or from my github repo in the example_data folder.

5.1 Loading and analysing the data

First all do we need to load the csv via pandas and analyse it:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as pyplot
pd.set_option('display.float_format', '{:.2f}'.format)
np.random.seed(0)

data =
↪  pd.read_csv("example_data/credit_risk_dataset.csv").fillna(0)

data.shape
data.head()
```

```
## (32581, 12)
##    person_age  person_income person_home_ownership  person_emp_length loan_intent loan_grade
## 0          22          59000                  RENT             123.00    PERSONAL          D
## 1          21           9600                   OWN               5.00   EDUCATION          B
```

```
## 2          25        9600             MORTGAGE               1.00     MEDICAL
## 3          23       65500                 RENT               4.00     MEDICAL
## 4          24       54400                 RENT               8.00     MEDICAL
```

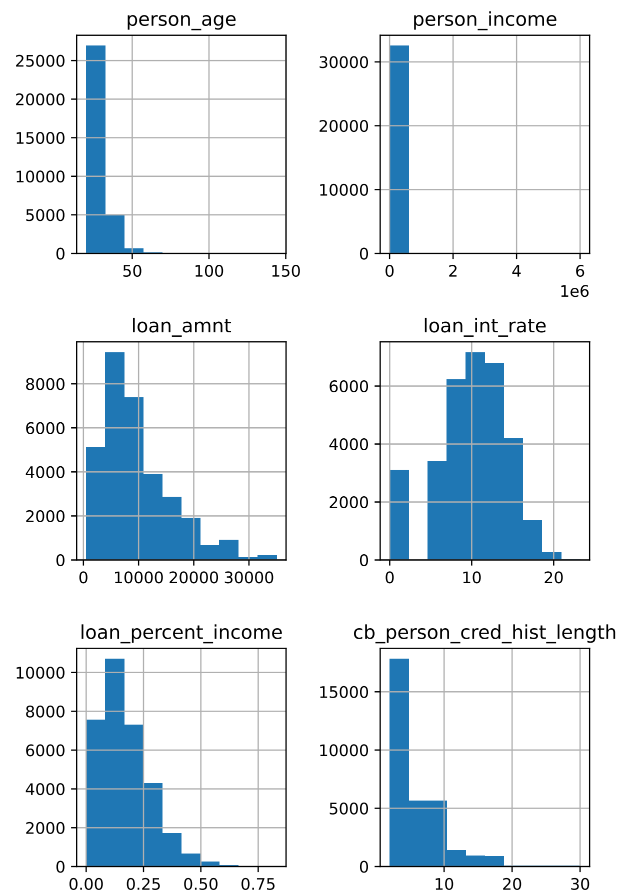
You can find the detailed information about the columns on kaggle, additionaly to the next table:

Feature Name	Description
person_age	Age
person_income	Annual Income
person*home*ownership	Home ownership
person*emp*length	Employment length (in years)
loan_intent	Loan intent
loan_grade	Loan grade
loan_amnt	Loan amount
loan*int*rate	Interest rate
loan_status	Loan status (0 is non default 1 is default)
loan*percent*income	Percent income
cb*person*default*on*file	Historical default
cb*preson*cred*hist*length	Credit history length

The important column is `load_status` that determinates the customers credit default and is used as the correct outputs Y. All other columns are considered as the input matrix X. First of all do we need to analyse the underlying data a little bit more. The columns with numerical data are visualized in the following charts:

```
l = ["person_age", "person_income", "person_emp_length",
↪  "loan_amnt", "loan_int_rate", "loan_status",
↪  "loan_percent_income", "cb_person_cred_hist_length"]
data[l].hist(bins=10,figsize=(8,8))
pyplot.tight_layout()
pyplot.show()
```

```
## array([[<AxesSubplot:title={'center':'person_age'}>,
##         <AxesSubplot:title={'center':'person_income'}>,
##         <AxesSubplot:title={'center':'person_emp_length'}>],
##        [<AxesSubplot:title={'center':'loan_amnt'}>,
##         <AxesSubplot:title={'center':'loan_int_rate'}>,
##         <AxesSubplot:title={'center':'loan_status'}>],
##        [<AxesSubplot:title={'center':'loan_percent_income'}>,
##         <AxesSubplot:title={'center':'cb_person_cred_hist_length'}>,
##         <AxesSubplot:>]], dtype=object)
```



All other input columns are categorical that cant be processed by Neuronal Networks. Luckily there exist some methods to convert the categories to numbers for example with Ordinal Encoding, One hot Encoding or Embedding (more information can be found here). I will choose the Ordinal Encoding for our dataset, because it is the simplest method. Ordinal Encoding just maps numbers to the categories. The best would be to arrange the categories as good as possible and just map numbers to it like in the following code:

```
data = data.replace({"Y": 1, "N":0})
data["person_home_ownership"] =
↪  data["person_home_ownership"].replace({'OWN':1, 'RENT':2,
↪  'MORTGAGE':3, 'OTHER':4})
data["loan_intent"] = data["loan_intent"].replace({'PERSONAL':1,
↪  'EDUCATION':2, 'MEDICAL':3, 'VENTURE':4,
↪  'HOMEIMPROVEMENT':5,'DEBTCONSOLIDATION':6})
data["loan_grade"] = data["loan_grade"].replace({'A':1, 'B':2,
↪  'C':3, 'D':4, 'E':5, 'F':6, 'G':7})
```

```
data.head()
```

```
##    person_age  person_income  person_home_ownership  person_emp_length  loan_intent
## 0          22          59000                      2             123.00            1
## 1          21           9600                      1               5.00            2
## 2          25           9600                      3               1.00            3
## 3          23          65500                      2               4.00            3
## 4          24          54400                      2               8.00            3
```

Its definitely not the best way to encode the categorical data but its the simplest method and it dosent increase the size of the input matrix.

Additionaly its important to normalize the data which leads to more stability and speed for the learning process (for more information). We will normalize the data to the interval $[0, 1]$ because we are using the sigmoid function. The following function will do the trick for the whole numpy array you are putting in:

```
def NormalizeData(np_arr):
  for i in range(np_arr.shape[1]):
    np_arr[:,i] = (np_arr[:,i] - np.min(np_arr[:,i])) /
↪  (np.max(np_arr[:,i]) - np.min(np_arr[:,i]))
  return(np_arr)
```

Now do we need to split the given data in a training dataset and a test dataset, convert the pandas dataframe to a numpy array and normalize it:

```
training_n = 2000
X_train = NormalizeData( data.loc[0:(training_n-1), data.columns
↪  != 'loan_status'].to_numpy() )
Y_train = data.loc[0:(training_n-1), data.columns ==
↪  'loan_status'].to_numpy()

X_test = NormalizeData( data.loc[training_n:, data.columns !=
↪  'loan_status'].to_numpy() )
Y_test = data.loc[training_n:, data.columns ==
↪  'loan_status'].to_numpy()
```

Its time to load the created functions from the previews chapters.

```
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+1,
↪   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))



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)
```

5.2 Train and test phase

We are creating a simple wrapper for the training and the test phase.

```
def train(X, Y, hidden_layer_neurons, alpha, epochs):
  n_input = len(X_train[0])
  n_output = len(Y_train[0])
  W = generate_weights(n_input, n_output, hidden_layer_neurons)
```

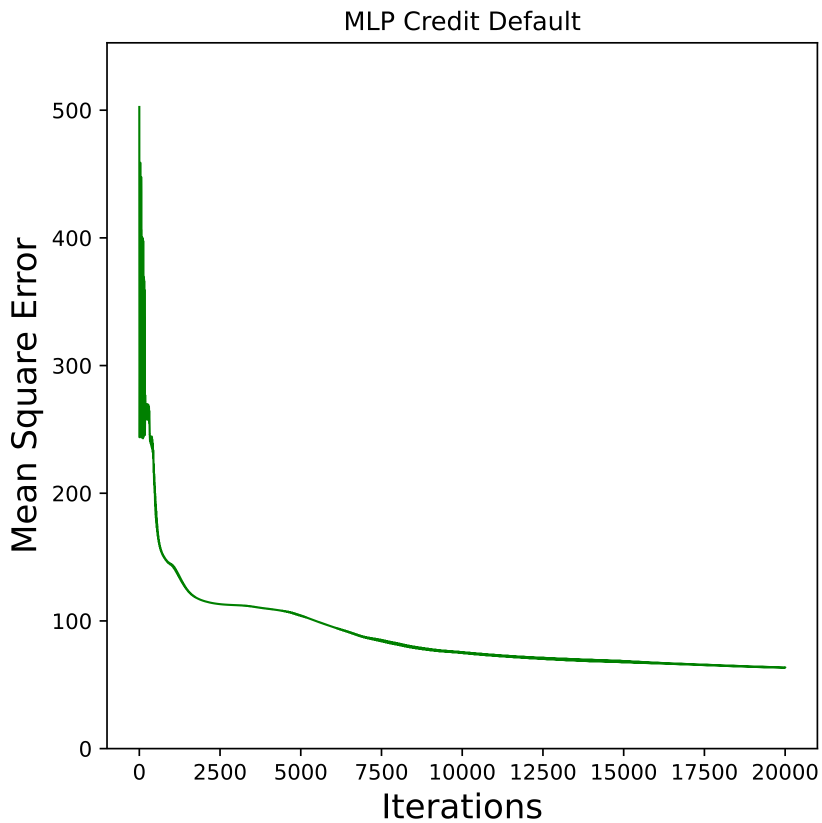
```
  errors = []
  for i in range(epochs):
    IN = []
    OUT = []
    grad = [None]*len(W)
    for k in range(len(W)):
      if k==0:
        IN.append(add_ones_to_input(X))
      else:
        IN.append(add_ones_to_input(OUT[k-1]))
      OUT.append(forward(x=IN[k], w=W[k]))

    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] + alpha * (IN[k].T @ grad[k])

  return W, errors

def test(X_test, W):
  for i in range(len(W)):
    X_test = forward(add_ones_to_input(X_test), W[i])
  return(X_test)
```

The `train()` function is just a simple wrapper around the things done in the last chapter and will fit the weights to the given `X_train` and `Y_train`. The `test()` function only contains the forward pass to calculate the output without adjusting the weights of the NN. Its used to evaluate the quality of the results. Its time to train the NN with the first 2000 rows of the given data, 20000 epochs, alpha of 0.01 and two hidden layers with 11 and 4 neurons:

```
W_train, errors_train = train(X_train, Y_train,
↪   hidden_layer_neurons = np.array([11,4]), alpha = 0.01, epochs
↪   = 20000)
```

The return contains multiple values, that are assigned with the `a, b = fun_that_returns_2_vals()` pattern. We can visualize the learning process by calculating the mean-square-error and plotting it with the familiar line-chart:

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

ms_errors_train = np.array(list(map(mean_square_error,
↪  errors_train)))

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,max(errors)*1.1)
  pyplot.show()

plot_error(ms_errors_train, "MLP Credit Default")
```



In the next step its time to test the NN on the never seen `X_test` and `Y_test` dataset.

```
result_test = test(X_test, W_train)
print("Mean Square error over all testdata: ",
↪  mean_square_error(Y_test - result_test))
```

```
## Mean Square error over all testdata:  2135.358966722297
```

Because the Mean Square Error is hard to interpret, we will classify the output of the NN to be 1 or 0 and analyze the given answer for the credit defaults.

```
def classify(Y_approx):
  return( np.round(Y_approx,0) )
```

```
classified_error = Y_test - classify(result_test)
print("Mean Square error over all classified testdata: ", mean_square_error(classified_error))

print("Probability of a wrong output: ", np.round(np.sum(np.abs(classified_error)) / len(classified_error) * 100, 2), "%" )
print("Probability of a correct output: ", np.round((1 - np.sum(np.abs(classified_error)) / len(classified_error))*100,2),"%" )
```

```
## Mean Square error over all classified testdata:  2458.5
## Probability of a wrong output:  16.08 %
## Probability of a correct output:  83.92 %
```

An incredible tool to qualify the result is the confusion matrix from the sklearn package. It splits the results into 4 categories that can be used to qualify the results with the following table:

Actual class \ Predicted class	P	N
P	TP	FN
N	FP	TN

For example is `TP` short for True-Positiv which means that the prediction was True=1 and the answer Positiv=1 so the prediction was correct. In our example do we have the following confusion matrix for the classified result of the test phase:

```
from sklearn.metrics import confusion_matrix
confusion_matrix(Y_test, classify(result_test))
```

```
## array([[21463,  2843],
##        [ 2074,  4201]], dtype=int64)
```

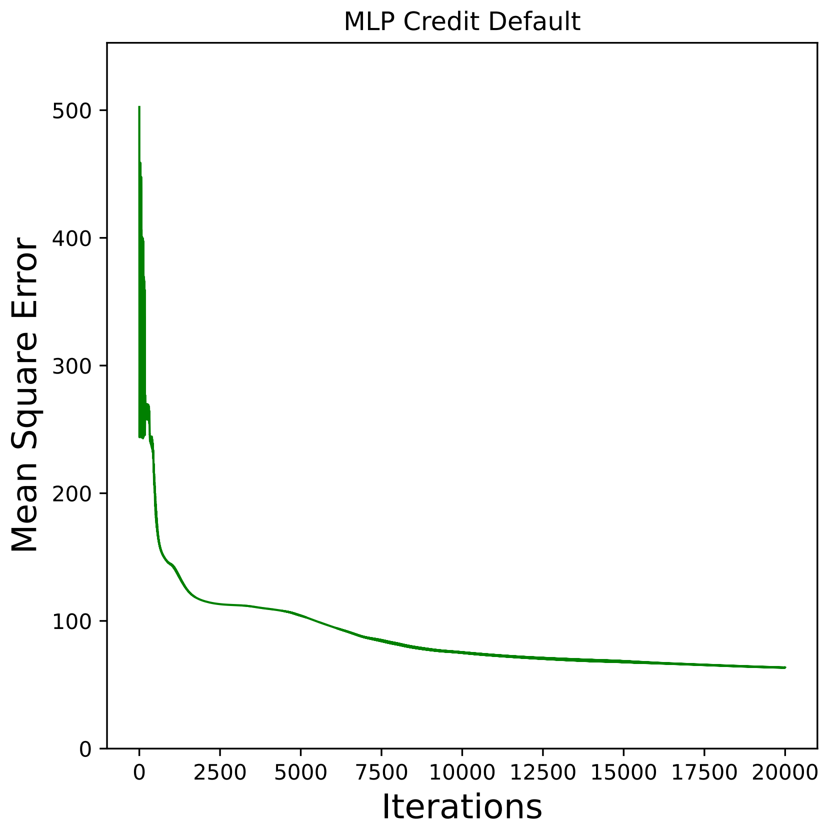
5.3 Appendix (complete code)

```
import numpy as np
import matplotlib.pyplot as pyplot
import pandas as pd
from sklearn.metrics import confusion_matrix
np.random.seed(0)

data =
↪  pd.read_csv("example_data/credit_risk_dataset.csv").fillna(0)
data = data.replace({"Y": 1, "N":0})

data["person_home_ownership"] =
↪  data["person_home_ownership"].replace({'OWN':1, 'RENT':2,
↪  'MORTGAGE':3, 'OTHER':4})
data["loan_intent"] = data["loan_intent"].replace({'PERSONAL':1,
↪  'EDUCATION':2, 'MEDICAL':3, 'VENTURE':4,
↪  'HOMEIMPROVEMENT':5,'DEBTCONSOLIDATION':6})
data["loan_grade"] = data["loan_grade"].replace({'A':1, 'B':2,
↪  'C':3, 'D':4, 'E':5, 'F':6, 'G':7})


def NormalizeData(np_arr):
  for i in range(np_arr.shape[1]):
    np_arr[:,i] = (np_arr[:,i] - np.min(np_arr[:,i])) /
↪   (np.max(np_arr[:,i]) - np.min(np_arr[:,i]))
  return(np_arr)

training_n = 2000
X_train = NormalizeData( data.loc[0:(training_n-1), data.columns
↪  != 'loan_status'].to_numpy() )
Y_train = data.loc[0:(training_n-1), data.columns ==
↪  'loan_status'].to_numpy()

X_test = NormalizeData( data.loc[training_n:, data.columns !=
↪  'loan_status'].to_numpy() )
Y_test = data.loc[training_n:, data.columns ==
↪  'loan_status'].to_numpy()



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+1,
↪  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))



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)

def train(X, Y, hidden_layer_neurons, alpha, epochs):
  n_input = len(X_train[0])
  n_output = len(Y_train[0])
  W = generate_weights(n_input, n_output, hidden_layer_neurons)
  errors = []
  for i in range(epochs):
    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] + alpha * (IN[k].T @ grad[k])

  return W, errors



W_train, errors_train = train(X_train, Y_train,
↪  hidden_layer_neurons = np.array([11,4]), alpha = 0.01, epochs
↪  = 20000)


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

ms_errors_train = np.array(list(map(mean_square_error,
↪  errors_train)))

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,max(errors)*1.1)
  pyplot.show()

plot_error(ms_errors_train, "MLP Credit Default")



def test(X_test, W):
  for i in range(len(W)):
    X_test = forward(add_ones_to_input(X_test), W[i])
  return(X_test)
```

```
result_test = test(X_test, W_train)
print("Mean Square error over all testdata: ",
↪  mean_square_error(Y_test - result_test))


def classify(Y_approx):
  return( np.round(Y_approx,0) )

classified_error = Y_test - classify(result_test)
print("Mean Square error over all classified testdata: ",
↪  mean_square_error(classified_error))

print("Probability of a wrong output: ",
↪  np.round(np.sum(np.abs(classified_error)) /
↪  len(classified_error) * 100, 2), "%" )
print("Probability of a right output: ", np.round((1 -
↪  np.sum(np.abs(classified_error)) /
↪  len(classified_error))*100,2),"%" )


confusion_matrix(Y_test, classify(result_test))
```




```
## Mean Square error over all testdata:  2135.358966722297
## Mean Square error over all classified testdata:  2458.5
## Probability of a wrong output:  16.08 %
## Probability of a right output:  83.92 %
## array([[21463,  2843],
##        [ 2074,  4201]], dtype=int64)
```

Chapter 6

Effect of batch size

First of all do we need to differentiate between the following definitions that i found here:
* one epoch := one forward pass and backward pass of all the training examples.
* batch size := the number of training scenarios in one forward/backward pass. The higher the batch size, the more memory space will be needed.
* number of iterations := number of passes, each pass using [batch size] number of scenarios. To be clear, one pass = forward pass + backward pass (we do not count the forward pass and backward pass as two different passes).

In the first chapter i mentioned that the normal approach to define a NN is to iterate over all training data by selecting a random scenario, until all scenarios are used ones, to create one epoch. I changed it to select all scenarios at the same time to make it simpler (full batch size). But what consequences do we get by setting the batch size to the number of rows in the training dataset?

Sadly there dosent exist a real proof for the differences, but i found a neat post that tries to explain its behavior here. The result is that considering the full batch size will lead to sharper minimaz in the optimization and a smaller batch size leads to flatter minimaz. If you want more information about the problem of speed vs accuracy by selecting the batch size can be found here. He explains very well where the problems are and that switching to a dynamic growing batch size could be a good solution.

All the information i found until now, points out that the underlying data determinates the optimal hyperparameters like bias, alpha, batch size, hidden neurons and so on. That means that changing the underlying dataset will change all the optimal hyperparameters as well. Maybe its better to test some hyperparameters for our Credit Default dataset and visualize the difference in the results.

6.1 Impact of diffrent hyperparameters

First of all do we add a `batch_size` parameter to the preview `train()` function and generate random batches that all contain distinct random integers in each batch. We will use the following function to generate the random batches:

```
import numpy as np
import matplotlib.pyplot as pyplot
import pandas as pd
from sklearn.metrics import confusion_matrix
import math as ma
import time
np.warnings.filterwarnings('ignore',
↪  category=np.VisibleDeprecationWarning)

def generate_random_batches(batch_size, full_batch_size):
  batches = np.arange(full_batch_size)
  np.random.shuffle(batches)
  return(np.array_split(batches,
   ↪  ma.ceil(full_batch_size/batch_size)))

generate_random_batches(3, 10)
```

```
## [array([0, 8, 9]), array([4, 1, 5]), array([6, 2]), array([3, 7])]
```

Now do we need to adjust the `train()` function to iterate over all batches:

```
def train(X, Y, hidden_layer_neurons, alpha, epochs, batch_size):
  n_input = len(X_train[0])
  n_output = len(Y_train[0])
  W = generate_weights(n_input, n_output, hidden_layer_neurons)
  errors = []
  batches = generate_random_batches(batch_size, full_batch_size =
↪  len(X))
  for i in range(epochs):
    error_temp = np.array([])
    for z in range(len(batches)):
      IN = []
      OUT = []
      grad = [None]*len(W)
      for k in range(len(W)):
        if k==0:
          IN.append(add_ones_to_input(X[batches[z],:]))
        else:
```

```
        IN.append(add_ones_to_input(OUT[k-1]))
      OUT.append(forward(x=IN[k], w=W[k]))

    error_temp = np.append(error_temp, Y[batches[z],:] -
↪  OUT[-1])

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

    for k in range(len(W)):
      W[k] = W[k] + alpha * (IN[k].T @ grad[k])
  errors.append(error_temp)

  return W, errors
```

And all the previes created functions, dataloading and transformations are:

```
data =
↪  pd.read_csv("example_data/credit_risk_dataset.csv").fillna(0)
data = data.replace({"Y": 1, "N":0})
data["person_home_ownership"] =
↪  data["person_home_ownership"].replace({'OWN':1, 'RENT':2,
↪  'MORTGAGE':3, 'OTHER':4})
data["loan_intent"] = data["loan_intent"].replace({'PERSONAL':1,
↪  'EDUCATION':2, 'MEDICAL':3, 'VENTURE':4,
↪  'HOMEIMPROVEMENT':5,'DEBTCONSOLIDATION':6})
data["loan_grade"] = data["loan_grade"].replace({'A':1, 'B':2,
↪  'C':3, 'D':4, 'E':5, 'F':6, 'G':7})

def NormalizeData(np_arr):
  for i in range(np_arr.shape[1]):
    np_arr[:,i] = (np_arr[:,i] - np.min(np_arr[:,i])) /
↪  (np.max(np_arr[:,i]) - np.min(np_arr[:,i]))
  return(np_arr)

training_n = 2000
X_train = NormalizeData( data.loc[0:(training_n-1), data.columns
↪  != 'loan_status'].to_numpy() )
Y_train = data.loc[0:(training_n-1), data.columns ==
↪  'loan_status'].to_numpy()

X_test = NormalizeData( data.loc[training_n:, data.columns !=
↪  'loan_status'].to_numpy() )
Y_test = data.loc[training_n:, data.columns ==
↪  'loan_status'].to_numpy()
```

```
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+1,
↪   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))


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)


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

def plot_error(errors, title):
  x = list(range(len(errors)))
  y = np.array(errors)
```

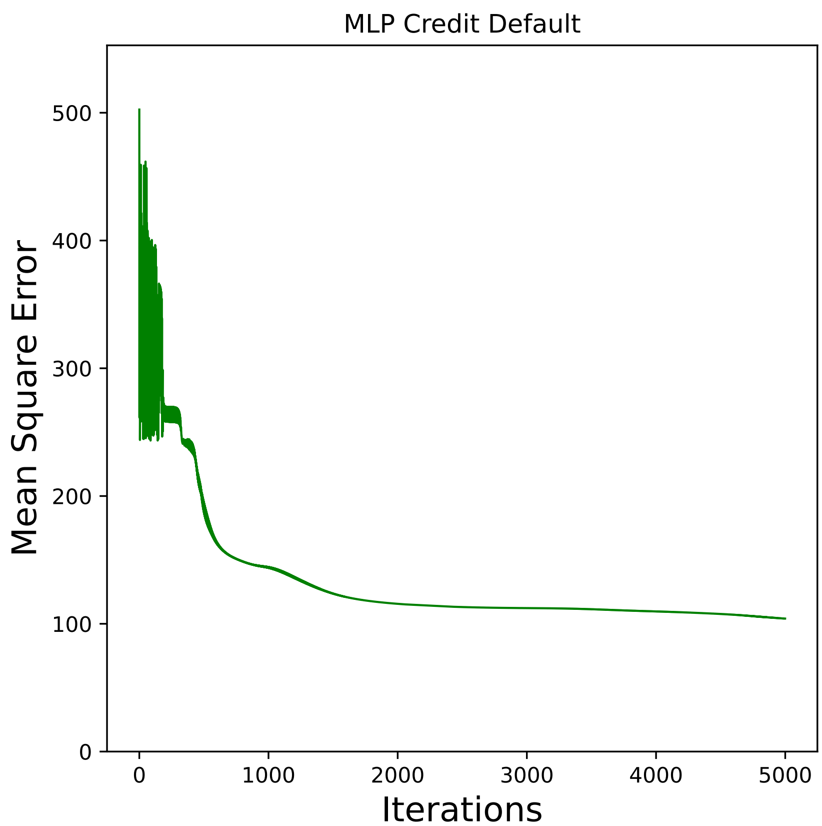
```
  pyplot.figure(figsize=(6,6))
  pyplot.plot(x, y, "g", linewidth=1)
  pyplot.xlabel("Iterations", fontsize = 16)
  pyplot.ylabel("Mean Square Error", fontsize = 16)
  pyplot.title(title)
  pyplot.ylim(0,max(errors)*1.1)
  pyplot.show()

def test(X_test, W):
  for i in range(len(W)):
    X_test = forward(add_ones_to_input(X_test), W[i])
  return(X_test)

def classify(Y_approx):
  return( np.round(Y_approx,0) )
```

Everything is loaded and set up. Now can we compare the time and the error for different `batch_size`:

```
# full batch size
np.random.seed(0)

start = time.time()
W_train, errors_train = train(X = X_train, Y = Y_train,
↪  hidden_layer_neurons = np.array([11,4]), alpha = 0.01, epochs
↪  = 5000, batch_size = 2000)
time_diff = time.time() - start
print("Time to train the NN: ", time_diff)

ms_errors_train = np.array(list(map(mean_square_error,
↪  errors_train)))
plot_error(ms_errors_train, "MLP Credit Default")
result_test = test(X_test, W_train)
print("Mean Square error over all testdata: ",
↪  mean_square_error(Y_test - result_test))

classified_error = Y_test - classify(result_test)
print("Mean Square error over all classified testdata: ",
↪  mean_square_error(classified_error))

print("Probability of a wrong output: ",
↪  np.round(np.sum(np.abs(classified_error)) /
↪  len(classified_error) * 100, 2), "%" )
print("Probability of a right output: ", np.round((1 -
↪  np.sum(np.abs(classified_error)) /
↪  len(classified_error))*100,2),"%" )
```

```
confusion_matrix(Y_test, classify(result_test))
```

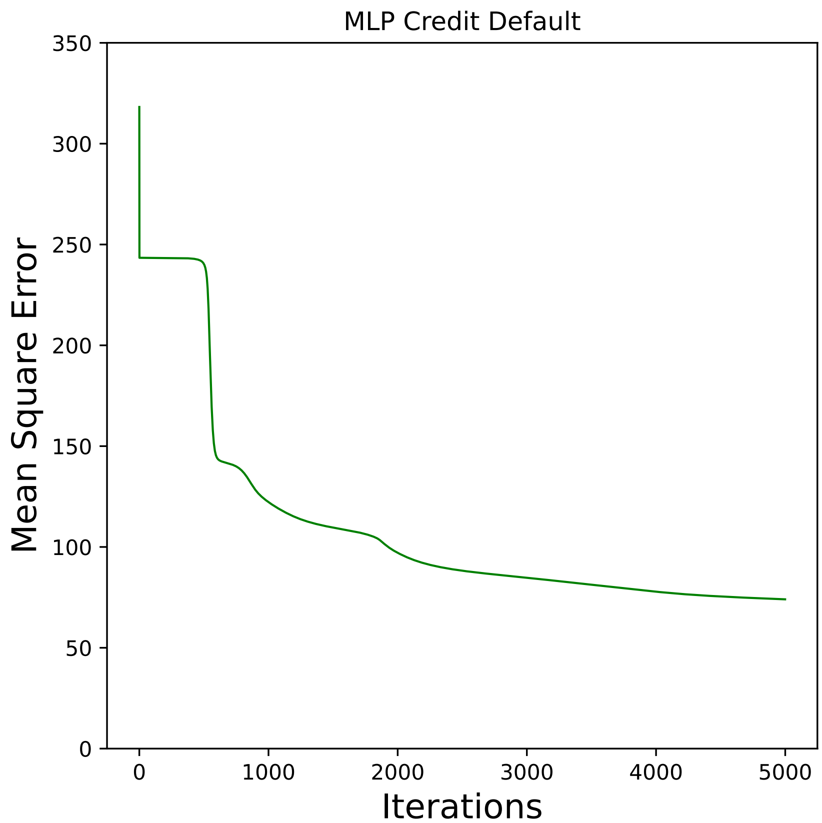
```
## Time to train the NN:  7.66628360748291
```




```
## Mean Square error over all testdata:  2402.2283244767077
## Mean Square error over all classified testdata:  2932.0
## Probability of a wrong output:  19.18 %
## Probability of a right output:  80.82 %
## array([[20193,  4113],
##        [ 1751,  4524]], dtype=int64)
```

```
# batch size = 100
np.random.seed(0)

start = time.time()
W_train, errors_train = train(X = X_train, Y = Y_train,
↪  hidden_layer_neurons = np.array([11,4]), alpha = 0.01, epochs
↪  = 5000, batch_size = 100)
time_diff = time.time() - start
print("Time to train the NN: ", time_diff)

ms_errors_train = np.array(list(map(mean_square_error,
↪  errors_train)))
plot_error(ms_errors_train, "MLP Credit Default")
result_test = test(X_test, W_train)
print("Mean Square error over all testdata: ",
↪  mean_square_error(Y_test - result_test))

classified_error = Y_test - classify(result_test)
print("Mean Square error over all classified testdata: ",
↪  mean_square_error(classified_error))

print("Probability of a wrong output: ",
↪  np.round(np.sum(np.abs(classified_error)) /
↪  len(classified_error) * 100, 2), "%" )
print("Probability of a right output: ", np.round((1 -
↪  np.sum(np.abs(classified_error)) /
↪  len(classified_error))*100,2),"%" )


confusion_matrix(Y_test, classify(result_test))
```

```
## Time to train the NN:  22.562650203704834
```




```
## Mean Square error over all testdata:  2210.3693145823286
## Mean Square error over all classified testdata:  2559.5
## Probability of a wrong output:  16.74 %
## Probability of a right output:  83.26 %
## array([[21396,  2910],
##        [ 2209,  4066]], dtype=int64)
```

We can see that the full batch size is much faster than the smaller one, but the smaller batch size has a smaller error. Maybe the perfect batch size depends on the problem itself. If you have a low dimensional input matrix and need to get a fast solution, its better to go with full batch size. If you have high dimensional input its not possible to use a full batch size, because your ram is jumping off. I think its the best to analyze the underlying data and select a suitable batch size for it. Maybe some situations will need a dynamic batch size, that shrinks over time, to get the best results, like in the blog post i mentioned at the start of the chapter.

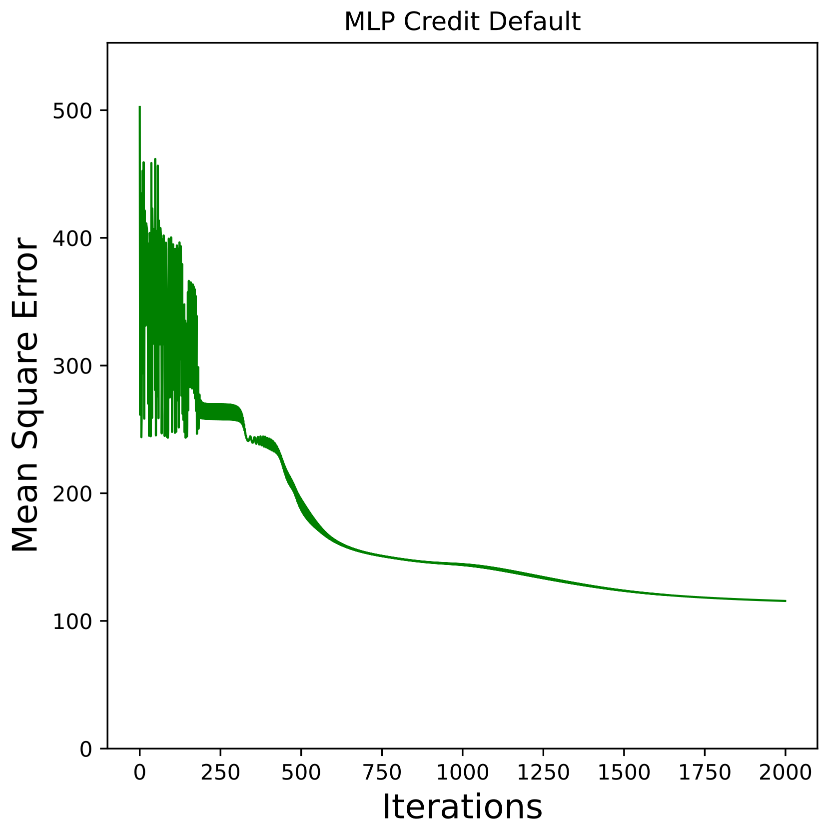
6.2 Appendix (complete code)

```
import numpy as np
import matplotlib.pyplot as pyplot
import pandas as pd
from sklearn.metrics import confusion_matrix
import math as ma
np.random.seed(0)
np.warnings.filterwarnings('ignore',
↪  category=np.VisibleDeprecationWarning)

data =
↪  pd.read_csv("example_data/credit_risk_dataset.csv").fillna(0)
data = data.replace({"Y": 1, "N":0})

data["person_home_ownership"] =
↪  data["person_home_ownership"].replace({'OWN':1, 'RENT':2,
↪  'MORTGAGE':3, 'OTHER':4})
data["loan_intent"] = data["loan_intent"].replace({'PERSONAL':1,
↪  'EDUCATION':2, 'MEDICAL':3, 'VENTURE':4,
↪  'HOMEIMPROVEMENT':5,'DEBTCONSOLIDATION':6})
data["loan_grade"] = data["loan_grade"].replace({'A':1, 'B':2,
↪  'C':3, 'D':4, 'E':5, 'F':6, 'G':7})


def NormalizeData(np_arr):
  for i in range(np_arr.shape[1]):
    np_arr[:,i] = (np_arr[:,i] - np.min(np_arr[:,i])) /
↪   (np.max(np_arr[:,i]) - np.min(np_arr[:,i]))
  return(np_arr)

training_n = 2000
X_train = NormalizeData( data.loc[0:(training_n-1), data.columns
↪  != 'loan_status'].to_numpy() )
Y_train = data.loc[0:(training_n-1), data.columns ==
↪  'loan_status'].to_numpy()

X_test = NormalizeData( data.loc[training_n:, data.columns !=
↪  'loan_status'].to_numpy() )
Y_test = data.loc[training_n:, data.columns ==
↪  'loan_status'].to_numpy()
```

```
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+1,
↪  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))



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)

def generate_random_batches(batch_size, full_batch_size):
  batches = np.arange(full_batch_size)
  np.random.shuffle(batches)
  return(np.array_split(batches,
   ↪  ma.ceil(full_batch_size/batch_size)))

def train(X, Y, hidden_layer_neurons, alpha, epochs, batch_size):
  n_input = len(X_train[0])
```

```
  n_output = len(Y_train[0])
  W = generate_weights(n_input, n_output, hidden_layer_neurons)
  errors = []
  batches = generate_random_batches(batch_size, full_batch_size =
↪  len(X))
  for i in range(epochs):
    error_temp = np.array([])
    for z in range(len(batches)):
      IN = []
      OUT = []
      grad = [None]*len(W)
      for k in range(len(W)):
        if k==0:
          IN.append(add_ones_to_input(X[batches[z],:]))
        else:
          IN.append(add_ones_to_input(OUT[k-1]))
        OUT.append(forward(x=IN[k], w=W[k]))

      error_temp = np.append(error_temp, Y[batches[z],:] -
↪  OUT[-1])

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

      for k in range(len(W)):
        W[k] = W[k] + alpha * (IN[k].T @ grad[k])
    errors.append(error_temp)

  return W, errors



W_train, errors_train = train(X = X_train, Y = Y_train,
↪  hidden_layer_neurons = np.array([11,4]), alpha = 0.01, epochs
↪  = 2000, batch_size = 2000)


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

ms_errors_train = np.array(list(map(mean_square_error,
↪  errors_train)))

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,max(errors)*1.1)
  pyplot.show()

plot_error(ms_errors_train, "MLP Credit Default")



def test(X_test, W):
  for i in range(len(W)):
    X_test = forward(add_ones_to_input(X_test), W[i])
  return(X_test)


result_test = test(X_test, W_train)
print("Mean Square error over all testdata: ",
↪  mean_square_error(Y_test - result_test))


def classify(Y_approx):
  return( np.round(Y_approx,0) )

classified_error = Y_test - classify(result_test)
print("Mean Square error over all classified testdata: ",
↪  mean_square_error(classified_error))

print("Probability of a wrong output: ",
↪  np.round(np.sum(np.abs(classified_error)) /
↪  len(classified_error) * 100, 2), "%" )
print("Probability of a right output: ", np.round((1 -
↪  np.sum(np.abs(classified_error)) /
↪  len(classified_error))*100,2),"%" )


confusion_matrix(Y_test, classify(result_test))
```




```
## Mean Square error over all testdata:  2330.8698549481937
## Mean Square error over all classified testdata:  2940.0
## Probability of a wrong output:  19.23 %
## Probability of a right output:  80.77 %
## array([[20516,  3790],
##        [ 2090,  4185]], dtype=int64)
```

Chapter 7

Class MLP

Finally its time to create a class `MLP` that contains all functions from the previews chapters. You can use this class to play arround with some settings and make your own discoveries about the datasets of your choice.