My First Steps in Neuronal Networks (Beginners Guide)

Axel Roth

2021-11-13

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:

$$\begin{bmatrix} x_{i,1} & x_{i,2} & | & y_i \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & | & 0 \\ 0 & 1 & | & 1 \\ 1 & 0 & | & 1 \\ 1 & 1 & | & 1 \end{bmatrix}$$

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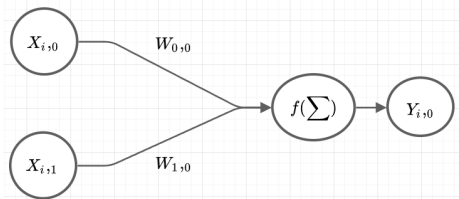
$$W = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$

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:

```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:

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

Here an small example for the forward pass:

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

```python
])
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.

```python
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:

```python
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:

```python
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")
```
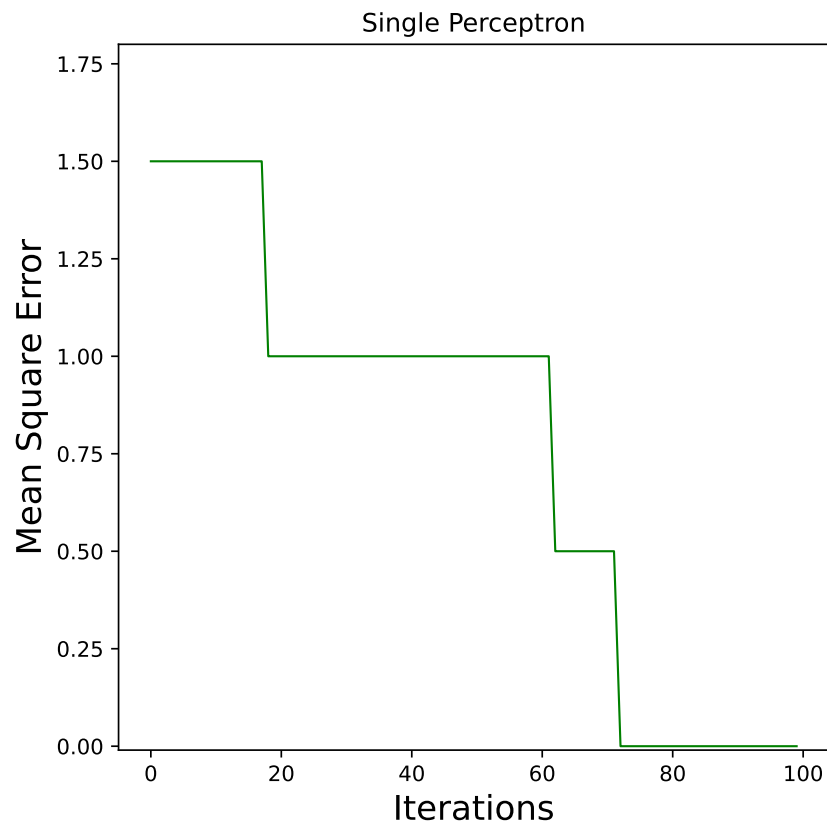
Single Perceptron

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

2.5   Appendix (complete code)

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


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

```python
])
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))
```

```python
    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:
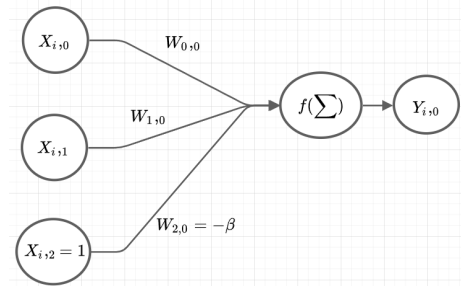
As python code we did the following:

```python
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:

```python
X = np.array([
  [0,0],
```

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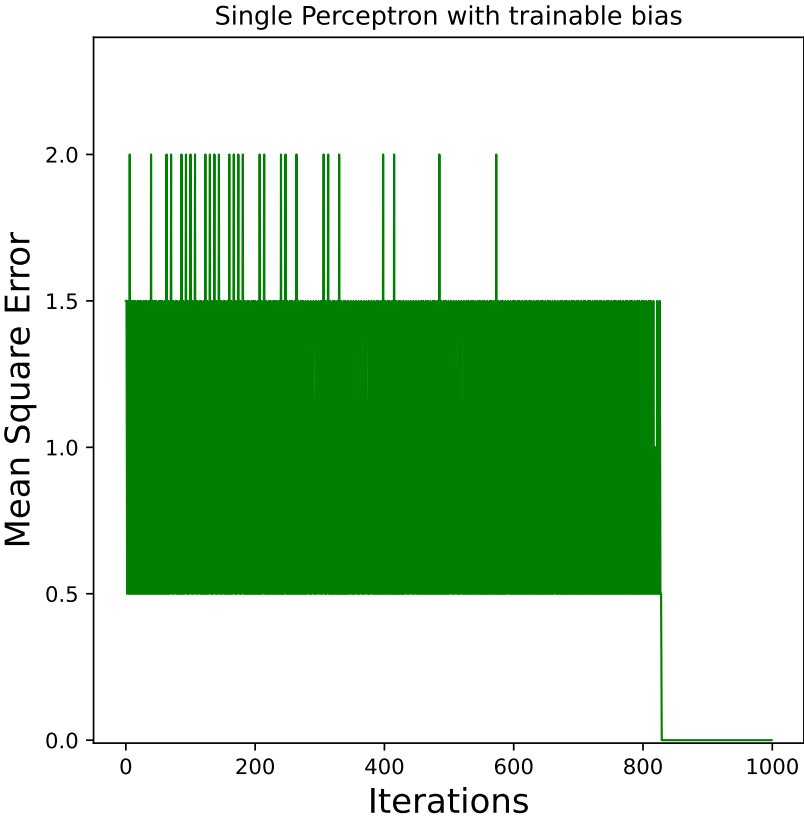
```python
mean_square_errors = np.array(list(map(mean_square_error, errors)))


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

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:
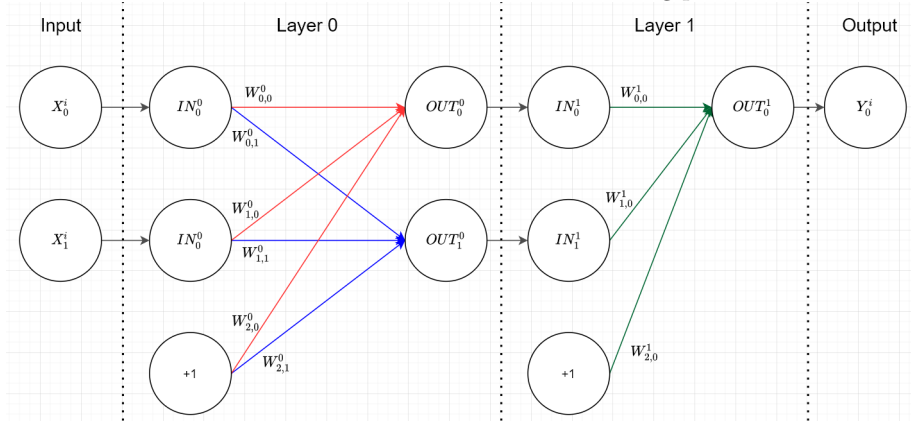
It is my own definition of layers, because i thought, it would be better to display layers like shown in the picture, to take the step from n to $n+1$ hidden layers more easy. You can see that each layer has the same process in the forward pass by evaluating $f(IN^{layer} \cdot W^{layer}) = OUT^{layer}$ 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.

```python
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:

```python
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

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]:

```
##   [[0.68823775 0.65188743]
##   [0.54435202 0.62033879]
##   [0.09792154 0.37793845]]
## W[1]:
##   [[0.93457527]
##   [0.89860947]
##   [0.73682695]]
```

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:

```python
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:

```python
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:

```python
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([[-0.85323323],
##         [ 0.11858954],
##         [ 0.11495934],
##         [-0.90289922]])
```

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:
##   [[0.85323323]
##    [0.88141046]
##    [0.88504066]
##    [0.90289922]]
## grad:
##   [[-0.10684723]
##    [ 0.0123957 ]
##    [ 0.01169639]
##    [-0.07915918]]
```

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)

```python
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):
```

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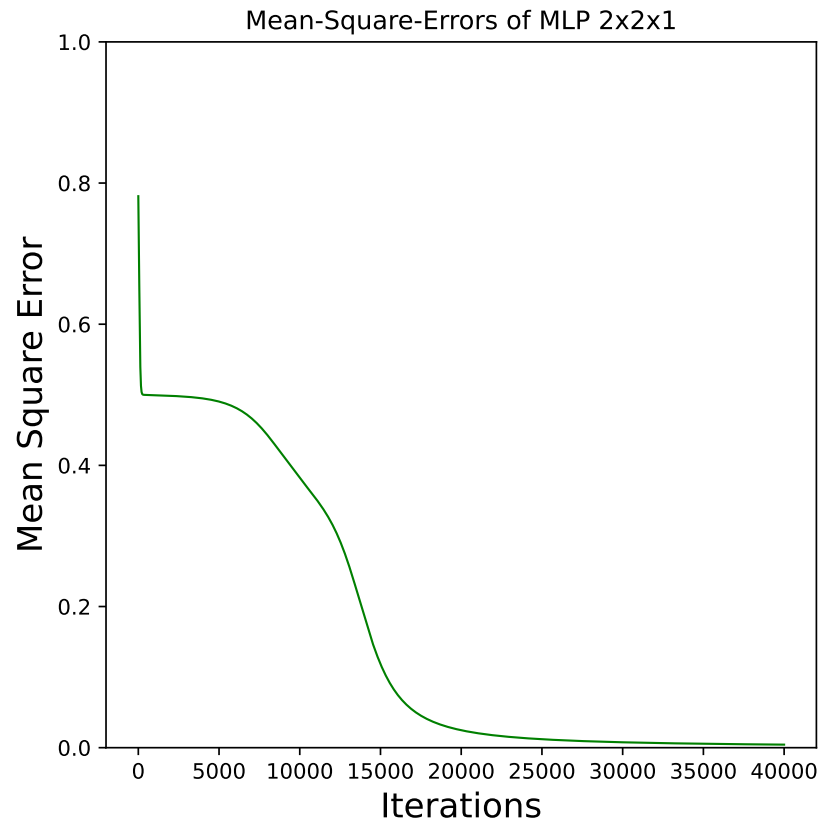
```python
    x = list(range(len(errors)))
    y = np.array(errors)
    pyplot.figure(figsize=(6,6))
    pyplot.plot(x, y, "g", linewidth=1)
    pyplot.xlabel("Iterations", fontsize = 16)
    pyplot.ylabel("Mean Square Error", fontsize = 16)
    pyplot.title(title)
    pyplot.ylim(0,1)
    pyplot.show()

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

Chapter 5

MLP example (Credit Default)

Now 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:

```python
pd.set_option('display.float_format', '{:.2f}'.format)
import pandas as pd
import matplotlib.pyplot as pyplot

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          C
## 3          23          65500                  RENT               4.00     MEDICAL          C
## 4          24          54400                  RENT               8.00     MEDICAL          C
```
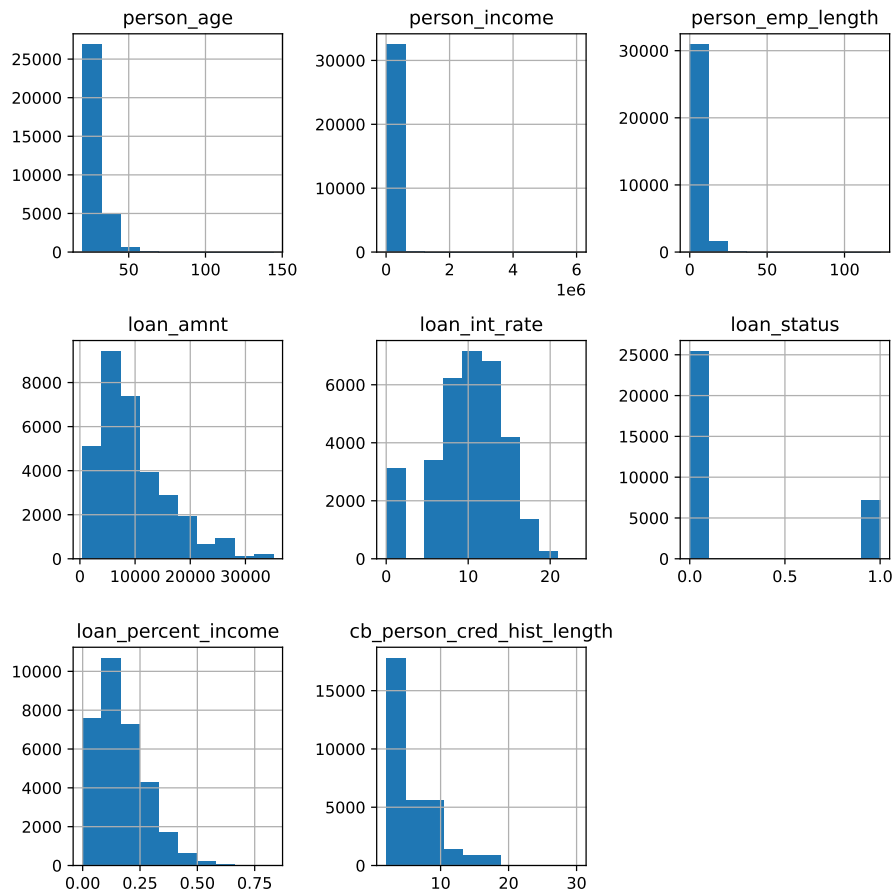
You can find the detailed information about the columns on kaggle, additionaly to the next table:

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",
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 and Embedding (more information can be found here). I will choose the simplest, the Ordinal Encoding, for our dataset. 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, 'MORTGA
data["loan_intent"] = data["loan_intent"].replace({'PERSONAL':1, 'EDUCATION':2, 'MEDICAL':3, 'VEN
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  loan_grad
## 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:

```python
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(
  return(np_arr)
```

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

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

Now its time to load the created functions from the previews chapters.

```python
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))
```

```python
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.

```python
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_train - OUT[-1])

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

```python
    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 get the given output of the NN to evaluate the quality.
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:

```python
W_train, errors_train = train(X_train, Y_train, hidden_layer_neurons = np.array([11,4]
```

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 with the familiar line-chart:

```python
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")
```
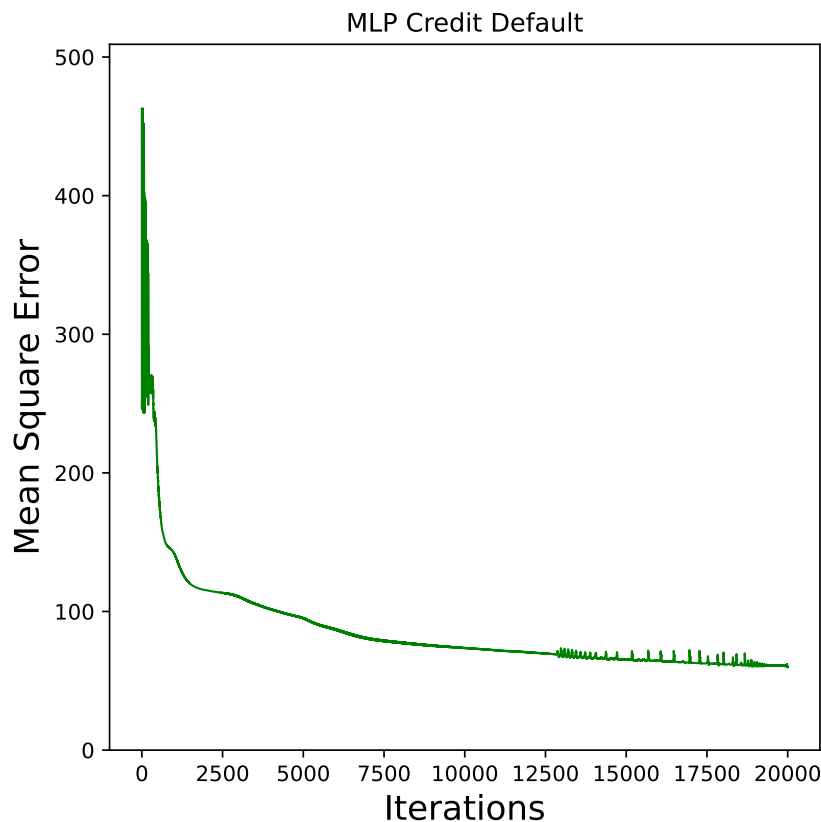
Now its time to test the NN on the never seen `X_test` and `Y_test` dataset.

```
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:   2944.4912492971584
```

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)) / l
print("Probability of a right output: ", np.round((1 - np.sum(np.abs(classified_error)
```

```
## Mean Square error over all classified testdata:  3361.0
## Probability of a wrong output:  21.98 %
## Probability of a right output:  78.02 %
```

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

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

results in to the following table:

For example `TP` mean 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([[19968,  4338],
##        [ 2384,  3891]], 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'
data["loan_intent"] = data["loan_intent"].replace({'PERSONAL':1, 'EDUCATION':2, 'MEDIC
```

```python
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:
```

```python
      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_train - 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])


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))
```

```python
  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(classifi
print("Probability of a right output: ", np.round((1 - np.sum(np.abs(classified_error)) / len(cla


confusion_matrix(Y_test, classify(result_test))
```
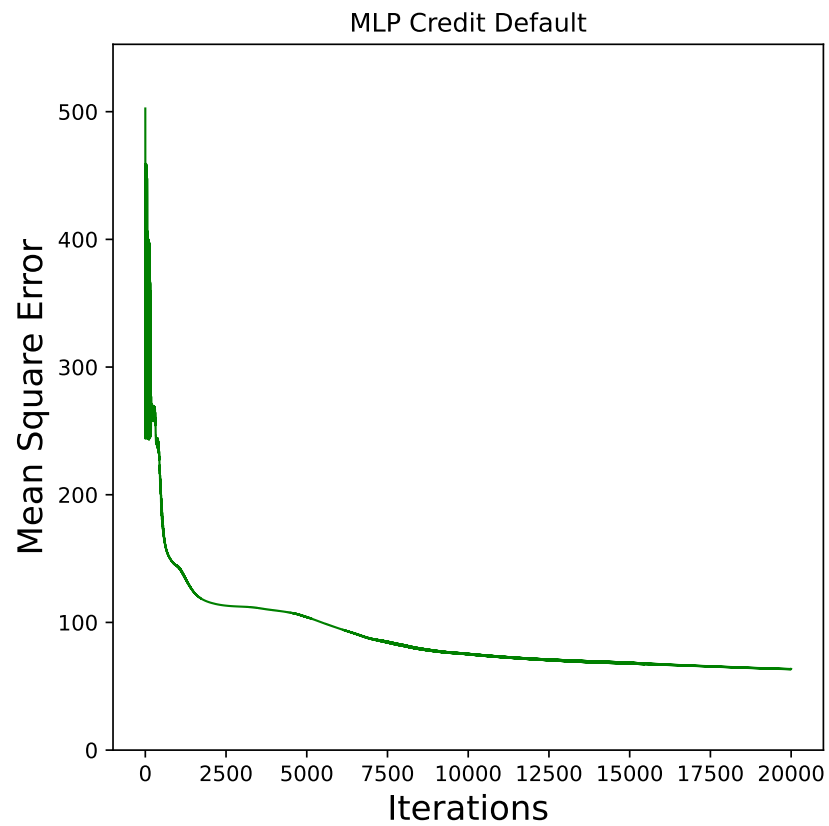
MLP Credit Default


```
## 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)
```