My First Steps in Neuronal Networks
(Beginners Guide)

Axel Roth

2021-11-25

Contents

Chapter 1

About

1.1 Me

Hello, my name is Axel Roth, and I'm pursuing a master's degree in mathematics in Germany while working part-time in finance as something between a data analyst and a full stack developer. I have a lot of experience with R and all of its features, but I have never written a line of Python code or worked with Neuronal Networks before. So, why am I writing a Python Beginner's Guide to Neuronal Networks in the first place? It's simple. I'm currently attending a lecture in which we're learning how to program a Neuronal Network from scratch using basic Python packages, and I'd like to share my experience. Furthermore, I learned everything I know from free internet sources, which is why I want to give something back. It's also a good use-case for me to write my first things in English.

1.2 The Book

This book will essentially be a transcript of my lecture, in which we will learn to program a simple Perceptron (the most basic Neuronal Network), then progress to a multilayer Perceptron, and finally to a brief overview of decision trees. On this journey, we'll put the Neuronal Network to the test in a variety of scenarios that are simple to replicate. Because these are my first steps in this field, I apologize for my terrible spelling and cannot guarantee the highest quality, but this may be the best way to educate and attract uninitiated readers to take a look.

1.3   How it works

I'm writing this book in the R-Studio IDE, using the Bookdown framework, and using the reticulate package to embed python code. This is why I need to load the Python interpreter in the R-chunk below:

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

In addition, I use R-Studio, R, and Python in a portable manner. It comes in handy when you need to switch between university and home computers, for example. Python can be downloaded through WinPython to be fully portable, and R-Studio supports it. I created all of the Neural Network images using the free website draw.io and im using QuillBot to compensate for my poor spelling skills. I would recommend jupyter lab or PyCharm if you are new to Python and have never used R.

Chapter 2

Single Perceptron

Throughout this chapter, I will show you how to program a single Perceptron in Python using only the numpy package. Numpy uses a vectorizable math structure, which allows you to easily perform normal matrix multiplications with just an expression (i always interpret vectors as one dimensional matrices!). In most cases, it is just a matter of translating mathematical formulas into python code without altering their structure. Before we begin, let's identify the necessary parameters, which will be 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 understand. If the weighted sum is smaller than the bias β, it will send the value zero to the next neuron. It is the same in the brain. If there is not enough electricity, the neuron will not activate, and the next does not receive electricity.

The training dataset is the following:

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

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

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 closely, you can see that this is the OR-Gate. Later you will understand, 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:
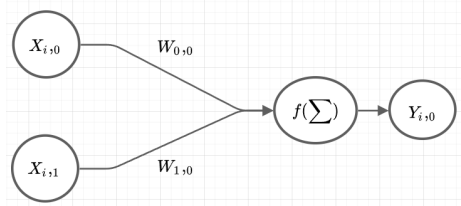
$$W = \begin{bmatrix} 0.1 \\ 0.2 \end{bmatrix}$$

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:

```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:

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

In the next step will we create an small example for the forward pass:

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

```python
  [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:

```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 and the correct outputs, do 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 these are the new weight.

2.4 Single Perceptron

Now do 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. In the next step, we will analyze the errors of each epoch. The best way to do so 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:

```python
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()`:

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

To plot the errors, im using the following function:

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


plot_error(mean_square_errors, "Single Perceptron")
```
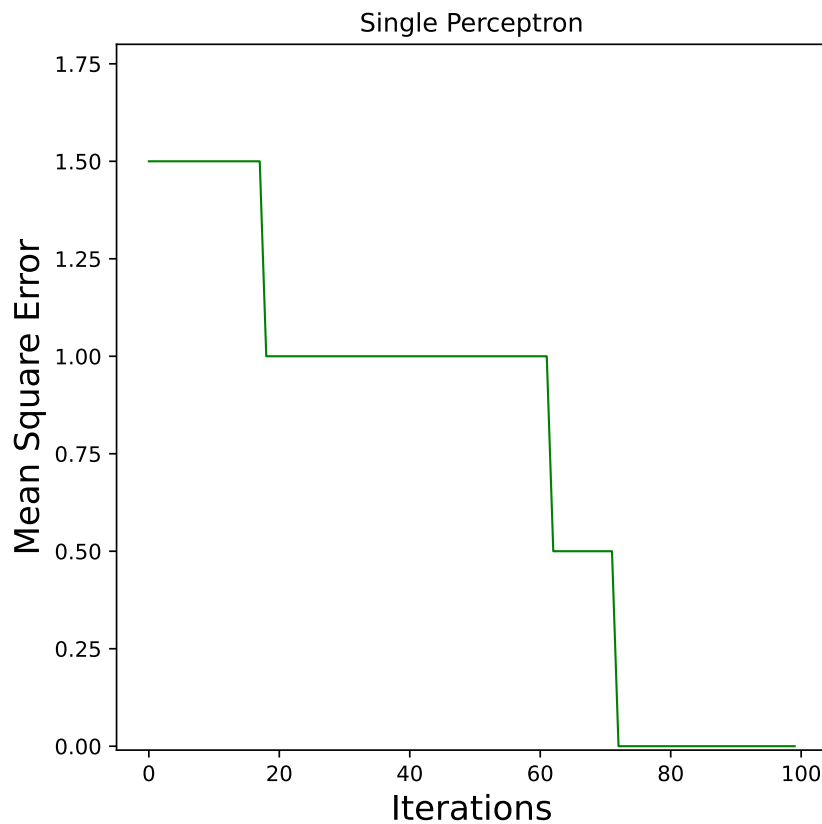
If you survived until now, you have learned how to program a single Perceptron!

2.5   Why does it work?

A single Perceptron with the heavyside activationfunction defines a classifier with the outputs 0 and 1. To find the correct solution, it needs to define a combination of weights and bias so that the inputs can be transferred to the groups $X \cdot W \geq \beta$ or $X \cdot W < \beta$. The single Perceptron only converges to the given results, if the inputs could be splitted into groups by a straight line in the graph. For example the OR-Gate:
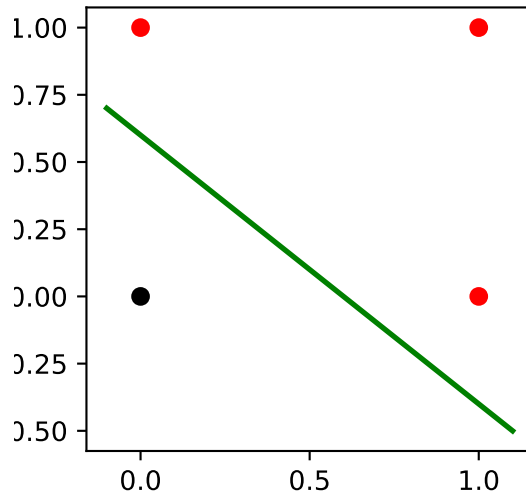
The red points (equals $Y_i = 1$) and the black points (equals $Y_i = 0$) can be split up with a straight line. If this isn't possible, as in the XOR-Gate, the single Perceptron will never find a combination of bias and weights to perform well. This explains why we need atleast multiple layers, as described in the chapter on multilayer Perceptrons. Here can you find a good source with more explanation.

2.6   Appendix (complete code)

```python
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]
])
```

```python
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 $\beta = 1$. It is the perfect 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 wouldn't 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 generalization 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:
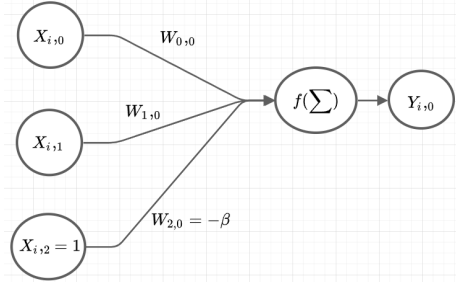
The same step can be made with the following python code:

```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)

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:

```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)

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):
```

```python
  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")
```
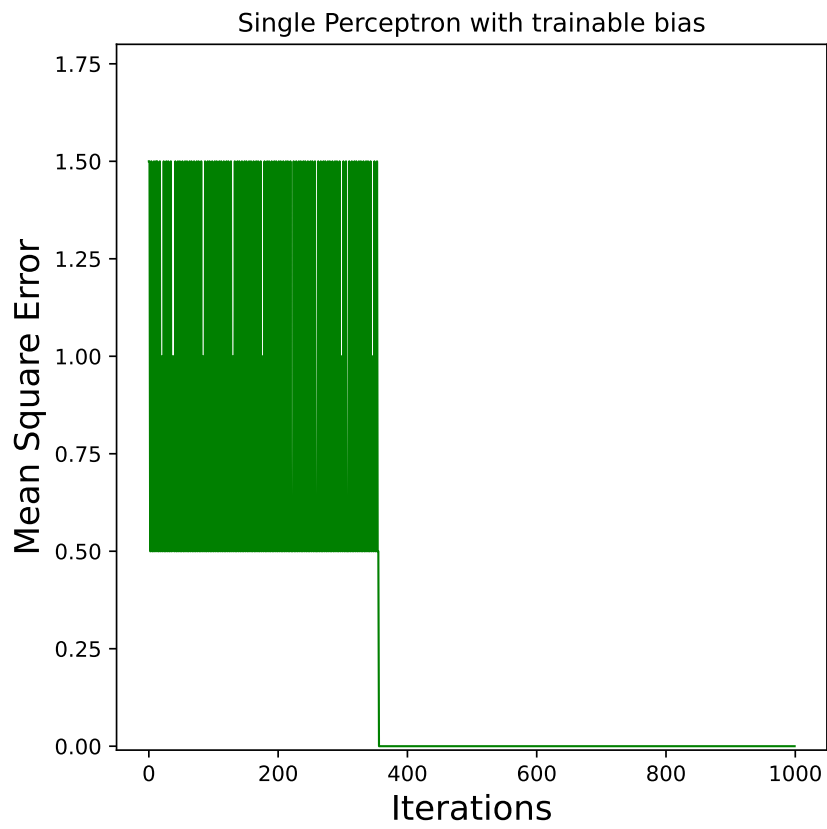
Chapter 4

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:
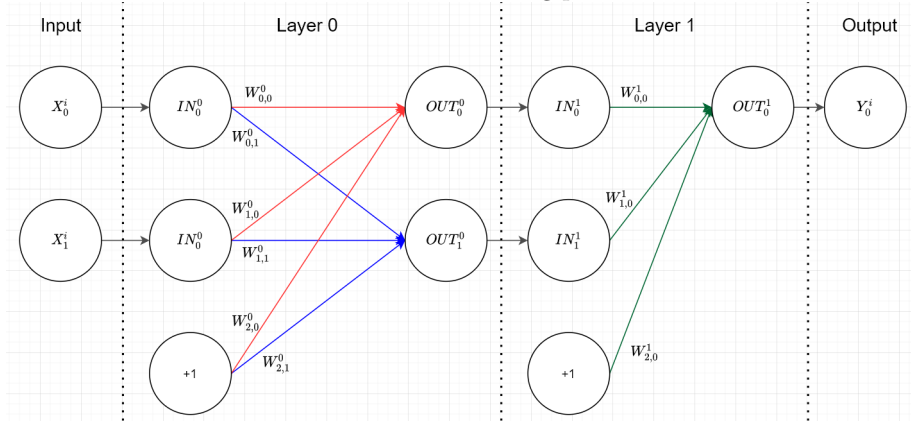
It's my own definition of layers since I believed it would be easier to transition from n to $n+1$ hidden layers if they were displayed as indicated in the image. By evaluating $f(IN^{layer} \cdot W^{layer}) = OUT^{layer}$ and just transferring the result to the next layer like $OUT^{layer} = IN^{layer+1}$, with f as the chosen activation function, you can see that each layer has the identical procedure in the forward pass. We'll choose the sigmoid function as the activation function f because it has a simple deviation f' which is used for the backward pass and behaves similarly to the heavyside function with an output between 0 and 1.

```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 random weights in a very generic approach:

```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 = [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]:
##  [[0.60136064 0.43288031]
##   [0.33505774 0.7272638 ]
##   [0.8422646  0.34953117]]
## W[1]:
##  [[0.45000163]
##   [0.54777406]
##   [0.72498472]]
```

The neurons for the hidden layers are generated using the `hidden_layer_neurons` list, while the input and output layer neurons are calculated from the training dataset. `hidden_layer_neurons = [4,2]`, for example, can construct two hidden layers with 4 and 2 neurons. I didn't choose the bias because it is automatically rectified. Is it now necessary to construct a helper function to add the biases to the last column of the inputs, like follows:

```
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]) )
```

```python
# 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([[-0.79588666],
##        [ 0.18579425],
##        [ 0.18771517],
##        [-0.82615389]])
```

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:

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

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:

```python
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:
##   [[0.79588666]
##   [0.81420575]
##   [0.81228483]
##   [0.82615389]]
## grad:
##   [[-0.12929265]
##   [ 0.02810598]
##   [ 0.02862247]
##   [-0.11865523]]
```

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:

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

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)

```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 = [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])))
```

```python
  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)
```

```python
  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
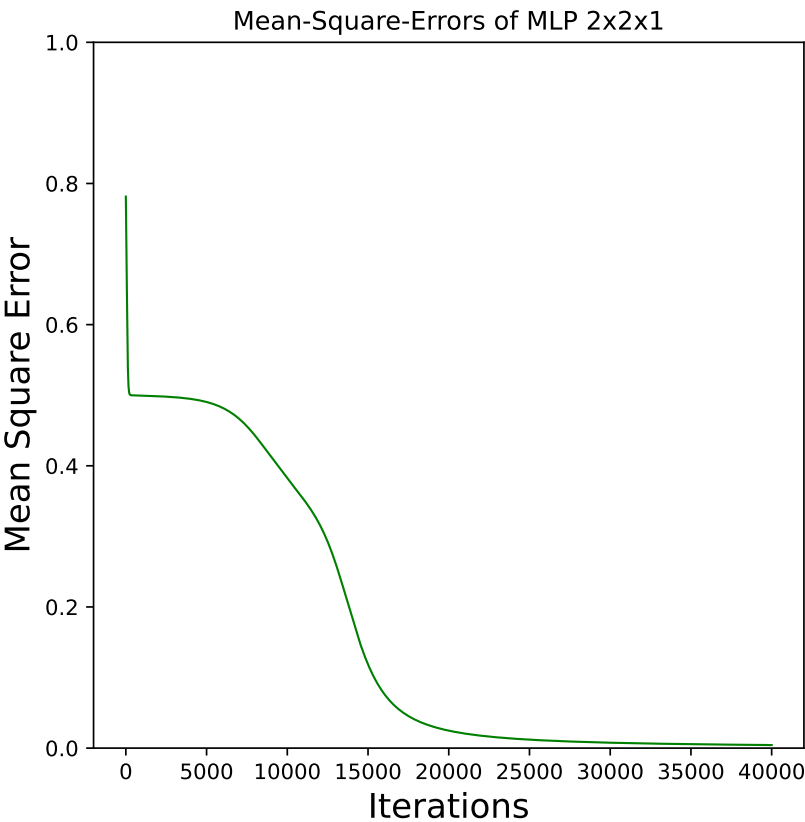
Chapter 5

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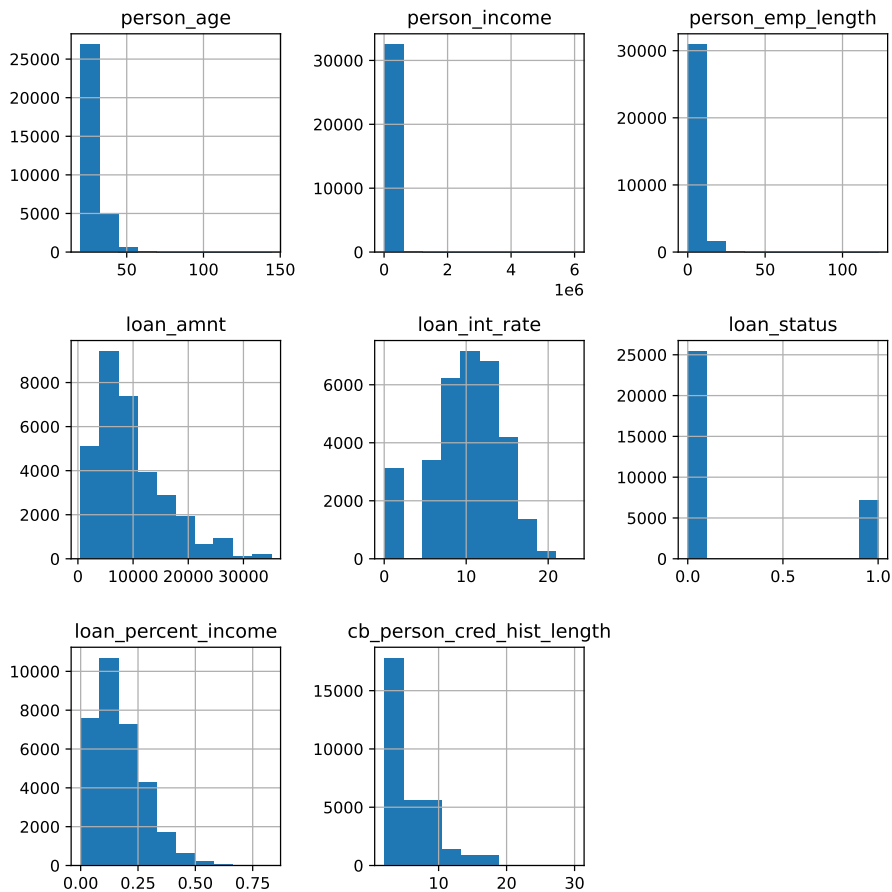
You can find the detailed information about the columns on kaggle, additionaly to the next table:

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*pres*on*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:

```python
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:

```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(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:

```python
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.

```python
def generate_weights(n_input, n_output, hidden_layer_neurons):
  W = []
```

```python
  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.

```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)
```

```python
  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:

```python
W_train, errors_train = train(X_train, Y_train,
↪   hidden_layer_neurons = [11,4], alpha = 0.01, epochs = 10000)
```

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:

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

```python
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")
```

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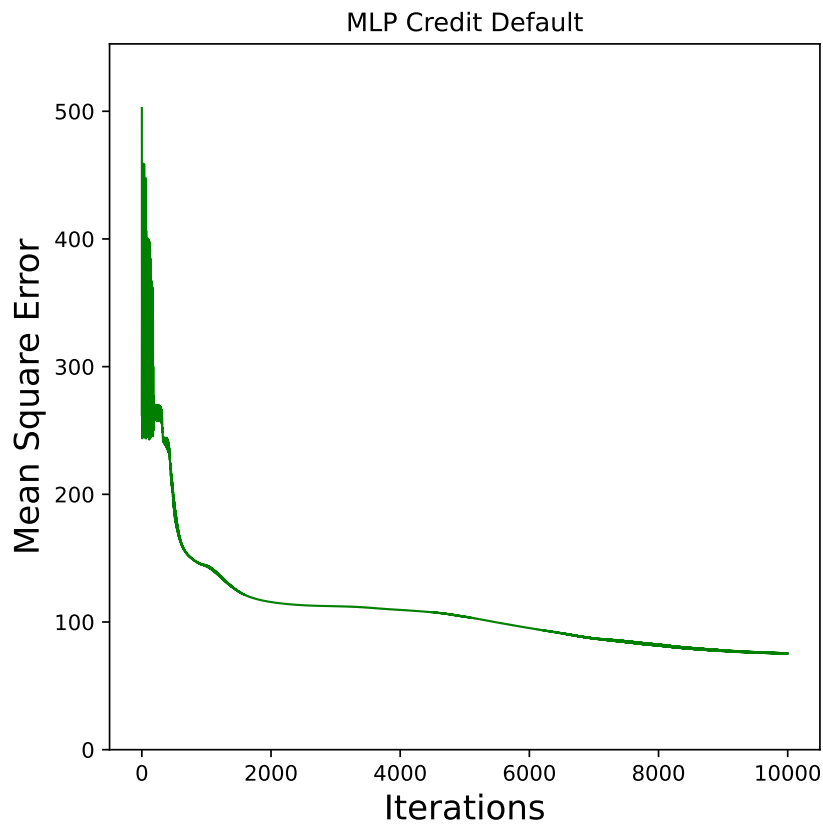
In the next step its time to test the NN on the never seen `X_test` and `Y_test` dataset.

```
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:   2012.1713984629632
```

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:   2457.0
## Probability of a wrong output:   16.07 %
## Probability of a correct output:   83.93 %
```

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 with the following table:

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([[21164,  3142],
##        [ 1772,  4503]], dtype=int64)
```

5.3 Appendix (complete code)

```python
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() )
```

```python
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[0])
  n_output = len(Y[0])
```

```python
  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 = [11,4], alpha = 0.01, epochs = 10000)


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()
```

```python
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))
```

MLP Credit Default


```
## Mean Square error over all testdata:   2012.1713984629632
## Mean Square error over all classified testdata:   2457.0
## Probability of a wrong output:   16.07 %
## Probability of a right output:   83.93 %
## array([[21164,   3142],
##        [ 1772,   4503]], 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:

```python
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:

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

```python
            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:

```python
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()
```

```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))


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)
```

```python
  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`:

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

start = time.time()
W_train, errors_train = train(X = X_train, Y = Y_train,
↪  hidden_layer_neurons = [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),"%" )
```
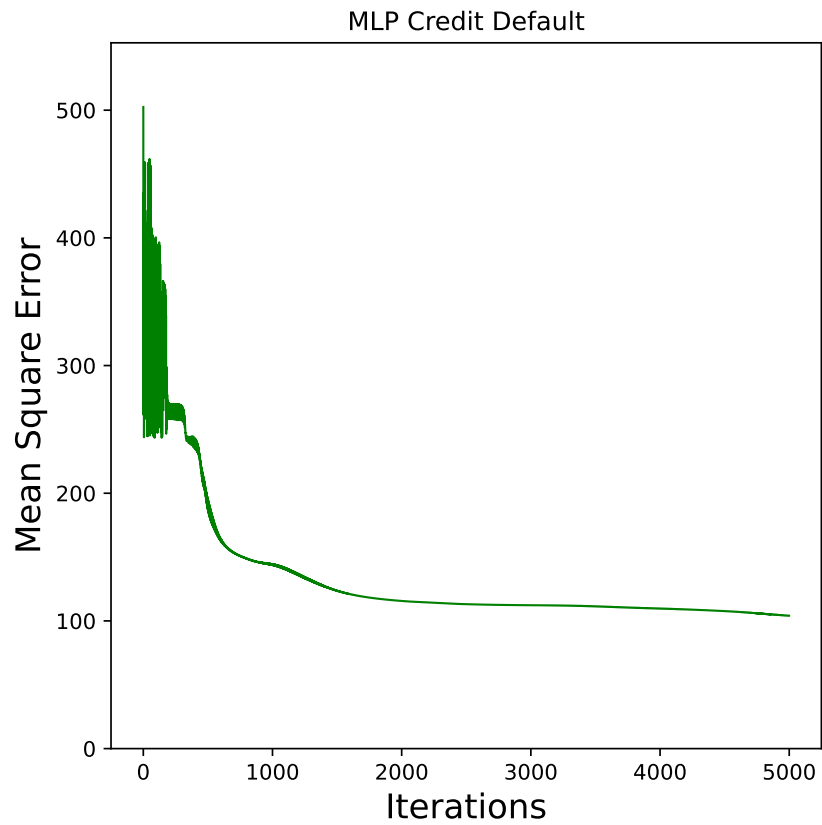
```
confusion_matrix(Y_test, classify(result_test))
```

```
## Time to train the NN:   10.723796129226685
```



```
## 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)
```

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

start = time.time()
W_train, errors_train = train(X = X_train, Y = Y_train,
↪  hidden_layer_neurons = [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:  23.007314682006836
```
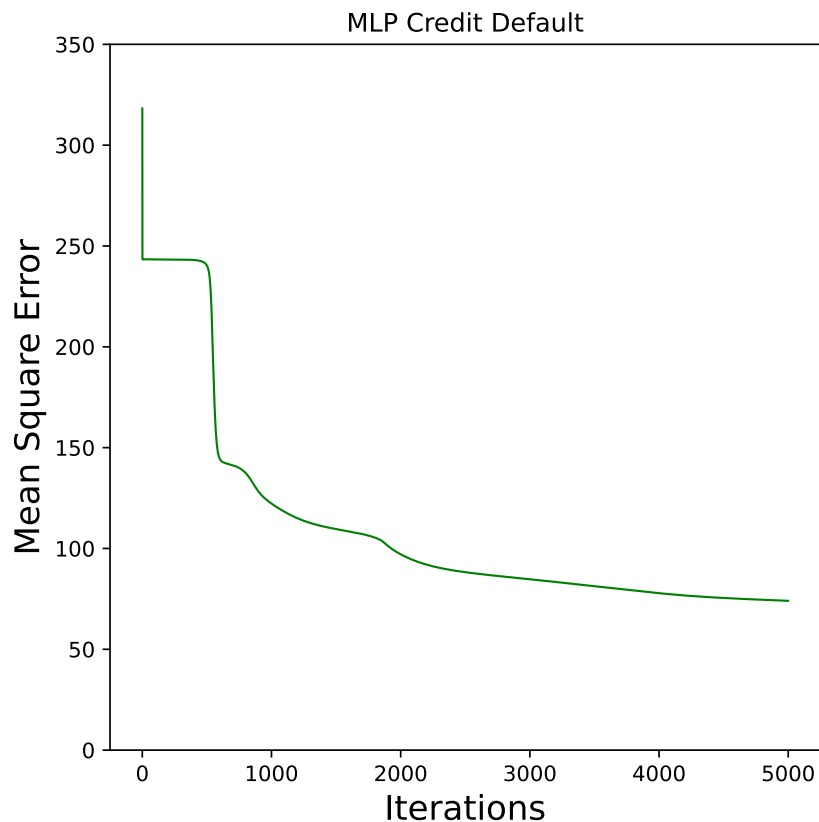
```
## Mean Square error over all testdata:   2210.3693145823286
## Mean Square error over all classified testdata:   2559.5
## Probability of a wrong output:   16.74 %
## Probability of a right output:   83.26 %
## array([[21396,   2910],
##        [ 2209,   4066]], dtype=int64)
```

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)

```python
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()
```

```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))



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[0])
```

```python
  n_output = len(Y[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 = [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)))
```

```python
  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))
```

MLP Credit Default


```
## 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.

```python
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)

class MLP:
  def __init__(self, X_train, Y_train, X_test, Y_test,
  ↪  hidden_layer_neurons, alpha, epochs, batch_size):
    self.X_train = X_train
    self.Y_train = Y_train
    self.X_test = X_test
    self.Y_test = Y_test
    self.hidden_layer_neurons = hidden_layer_neurons
    self.alpha = alpha
    self.epochs = epochs
    self.batch_size = batch_size

  def generate_weights(self):
    W = []
    for i in range(len(self.hidden_layer_neurons)+1):
      if i == 0: # first layer
```

```python
      W.append(np.random.random((len(self.X_train)+1,
↪  self.hidden_layer_neurons[i])))
    elif i == len(self.hidden_layer_neurons): # last layer

↪  W.append(np.random.random((self.hidden_layer_neurons[i-1]+1,
↪  len(self.Y_train))))
    else: # middle layers

↪  W.append(np.random.random((self.hidden_layer_neurons[i-1]+1,
↪  self.hidden_layer_neurons[i])))
  return(W)

@staticmethod
def add_ones_to_input(x):
  return(np.append(x, np.array([np.ones(len(x))]).T, axis=1))
@staticmethod
def sigmoid(x):
  return 1.0 / (1.0 + np.exp(-x))
@staticmethod
def deriv_sigmoid(x):
  return x * (1 - x)

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

@staticmethod
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(self):
  batches = np.arange(len(self.X_train))
  np.random.shuffle(batches)
  return(np.array_split(batches,
    ↪  ma.ceil(len(self.X_train)/self.batch_size)))

def train(self):
  W = self.generate_weights()
  errors = []
  batches = self.generate_random_batches()
```

```python
    for i in range(self.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(self.add_ones_to_input(x =
→   self.X_train[batches[z],:]))
          else:
            IN.append(self.add_ones_to_input(x = OUT[k-1]))
          OUT.append(self.forward(IN[k], W[k]))

        error_temp = np.append(error_temp,
→   self.Y_train[batches[z],:] - OUT[-1])

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

        for k in range(len(W)):
          W[k] = W[k] + self.alpha * (IN[k].T @ grad[k])
      errors.append(error_temp)
    self.W = W
    self.errors = errors
  @staticmethod
  def mean_square_error(error):
    return( 0.5 * np.sum(error ** 2) )
  @staticmethod
  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(self):
    X_test = self.X_test
    for i in range(len(self.W)):
      X_test = self.forward(self.add_ones_to_input(X_test),
→   self.W[i])
```

```python
    return(X_test)
  @staticmethod
  def classify(Y_approx):
    return( np.round(Y_approx,0) )
```

```python
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()


# mlp = MLP(X_train, Y_train, X_test, Y_test,
↪  hidden_layer_neurons = [11,4], alpha = 0.01, epochs = 2000,
↪  batch_size = 2000)
# mlp.train()
```

Chapter 8

Decision Tree

The Decision Trees (DT) and Neuronal Netwoks (NN) aim for the same goal, analyzing the data and giving answers to never seen data. Nevertheless do they have huge differences in working with each of these methods. For this reason do i link here an very good comparison of these two methods. The result is that Decision Trees are much simpler than Neuronal Networks with the additional fact that its easier to interpret the way of giving a certain answer. In comparison to this, can Neuronal Networks process very huge datasets with ease and you can adjust the behavior of learning strongly by the selected hyper-parameter, which can lead to very high accuracy. If you take all in consideration, its useful to think about Decision Trees, before you create a Neuronal Network, because its a simpler approach. If your data is more complex or large you should definitely go with the Neuronal Networks. In very complex situation they often will use hybrids of Decision Trees and Neuronal Networks to archive understandable results with high accuracy (for example Neural-Backed Decision Trees).

8.1 Entropy

Entropy measures the impurity of the given data. Low impurity leads to better classification and higher accuracy. The goal of Decision Trees is to split the data at each node with the highest gain of purity. The purity of the given dataset can be calculated with the concave Entropy-Formula:

$$E = -p \cdot log_2(p) - (1 - p) \cdot log_2(1 - p)$$

with p as the probability of having no default in the credit default dataset. The Entropy increases to 1 if the dataset contains 50% of defaults and 50% of no default and decreases to 0 if the dataset contains only defaults or only no defaults. The python function is the following:

```python
def calc_entropy(df, decision_on = "loan_status"):
  if len(df)==0:
    return(0)
  p = np.sum(df[decision_on] == 0)/len(df)
  if p == 0 or p == 1:
    return(0)
  result = -p * ma.log(p,2) - (1-p)*ma.log(1-p,2)
  return result
```

Now can we calculate the initial Entropy of the credit default dataset as shown in the next code snippet:

```python
import numpy as np
import math as ma
import pandas as pd

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

entropy_of_data = calc_entropy(df=data)
print("Initial Inpurity/Entropy of data: ", entropy_of_data)
```

```
## Initial Inpurity/Entropy of data:  0.7568007498467375
```

Sadly do we have the same problem with categorical data as the NN. We could transform categorical data to numerical as shown in the Credit Default chapter, but i want to drop these columns to make it easier to understand. Additional will we select only 4 columns plus the answer, to make it more interpretable and manageable.

```python
data = data.loc[:, ["person_age", "loan_percent_income",
↪ "loan_int_rate", "cb_person_default_on_file" ,"loan_status"]]
```

Now do we need to create the nodes of the tree based on decisions that lead to the lowest Entropy. The resulting purity by splitting the data in 2 subsets by a condition of one column (value z), can be calculated as:

$p_1(z) =$ proportion of column-value $> z p_2(z) =$ proportion of no default and column-value $> z p_3(z) =$ p

and in python can we use the `calc_entropy` function to make it even simpler:

```python
def calc_splitted_entropy(df, col, val, decision_on =
↪  "loan_status"):
  w = np.sum(df[col] > val)/len(df)
  result = w * calc_entropy(df.loc[df[col] > val], decision_on) +
↪  (1-w) * calc_entropy(df.loc[df[col] <= val], decision_on)
  return result
```

For example can we split the dataset by column `loan_percent_income` and value $z = 0.3$ to archive an decrease in Entropy:

```python
entropy_of_splitted_data = calc_splitted_entropy(df=data,
↪  col="loan_percent_income", val=0.3, decision_on =
↪  "loan_status")
print("Splitted Inpurity/Entropy of data: ",
↪  entropy_of_splitted_data)
```

```
## Splitted Inpurity/Entropy of data:  0.648938734265563
```

This split results in a decrease of `python round(entropy_of_data-entropy_of_splitted_data,5)` in the overall Entropy.

8.2 Constructing the Tree

First of all do we need to find the column and the value that results in the highest decrease of Entropy and splitt the dataset by it. Afterwards do we pass the resulting subsets into the same function (recursion). We will save the given conditions for the splitting. If the given subset contains less than `min_size` of rows or reached the `max_depth` it will turn into a leaf. We need to analyze the function properties to find the minimal value for the optimal splitting. The `calc_splitted_entropy` function is concave ("The Entropy is concave in the probability mass function"). We can use this property to write a simple minimiza that checks if the next evaluation is smaller than the previous and steps along the input value. If the evaluation is bigger, it will change the direction and decrease the step distance. It repeats this process until the change in the evaluation is stagnating.

```python
def find_minima(df, col, decision_on = "loan_status", round_at =
↪  5):
  direction = 1
  step = (df[col].max()-df[col].min()) * 0.1
  val = df[col].min() + step
  best_entropy = 1
```

```
    stagnation = 0

    while stagnation <= 15:
      temp = calc_splitted_entropy(df, col, val)
      if temp > best_entropy:
        direction = -direction
        step = 0.5 * step
        stagnation += 1
      elif round(temp,round_at) < round(best_entropy,round_at):
        stagnation = 0
      else:
        stagnation += 1
      best_entropy = temp
      val = val + direction * step

    return best_entropy, val
```

This minimizer is written by my self and it only works for convex functions. I dont know if there do exist better approaches, but this one works.

Now do we need to find the best decrase in Entropy of all columns with the next function:

```
def find_best_col(df, decision_on = "loan_status", round_at = 5):
  cols = list(df.columns[df.columns != decision_on])
  entropys = np.ones(len(cols))
  vals = np.ones(len(cols))

  for i in range(len(cols)):
    entropys[i], vals[i] = find_minima(df, col=cols[i],
↪   decision_on = "loan_status", round_at = 5)

  best_i = int(np.where(entropys == min(entropys))[0][0])
  return cols[best_i], entropys[best_i], vals[best_i]
```

For example would the output for the initial dataste be:

```
find_best_col(data)
```

```
## ('loan_percent_income', 0.6564119496913492, 0.29990234374999997)
```

Now its time to construct the tree and save all data that ends in a leaf:

```python
def make_node_and_leafs(df, decision_on = "loan_status", round_at
↪  = 5, path = "I", condition = "", min_size = 1000, max_depth =
↪  4, leafs = pd.DataFrame(columns=["path", "condition", "rows",
↪  "P_of_no_default", "entropy"])):
  if len(df) < min_size or (path.count("-")-1) >= max_depth or
  ↪  len(df.columns) <= 1:
    leafs = leafs.append({"path":path+"}",
↪  "condition":condition[0:(len(condition)-5)], "rows":len(df),
↪  "P_of_no_default":np.sum(df[decision_on] == 0)/len(df),
↪  "entropy":calc_entropy(df)}, ignore_index=True)
  else:
    col, entropy, val = find_best_col(df, decision_on, round_at)
    print("path:", path, "   entropy:", entropy, "  col:", col, "
    ↪  val:", val, "  rows:", len(df))
    leafs = make_node_and_leafs( df.loc[df[col] > val, df.columns
↪  != col], decision_on, round_at, path+"-R", condition+col+" >
↪  "+str(float(round(val,5)))+" and ", min_size, max_depth,
↪  leafs)
    leafs = make_node_and_leafs( df.loc[df[col] <= val,
↪  df.columns != col], decision_on, round_at, path + "-L",
↪  condition+col+" <= "+str(float(round(val,5)))+" and ",
↪  min_size, max_depth, leafs)
  return(leafs)

leafs = make_node_and_leafs(df=data, decision_on = "loan_status",
↪  round_at = 5, min_size = 1000, max_depth = 4)
leafs["entropy"] = (leafs["entropy"]*leafs["rows"])/len(data)

print("Entropy in data: ", calc_entropy(data))
print("Entropy in all leafs: ", np.sum(leafs["entropy"]))
```

```
## path: I     entropy: 0.6564119496913492   col: loan_percent_income    val: 0.29990234374999997
## path: I-R    entropy: 0.9103981172376336   col: loan_int_rate    val: 14.01635546875   rows: 4
## path: I-R-L   entropy: 0.9607335396537932   col: cb_person_default_on_file    val: 0.14999999
## path: I-R-L-L   entropy: 0.9648046914552115   col: person_age    val: 34.296875   rows: 2977
## path: I-L    entropy: 0.531844917392527   col: loan_int_rate    val: 14.280073242187505   rows
## path: I-L-R    entropy: 0.998669387657678   col: person_age    val: 22.3046875   rows: 4057
## path: I-L-R-R    entropy: 0.9992350724472542   col: cb_person_default_on_file    val: 0.149999
## path: I-L-L    entropy: 0.44498607709157834   col: cb_person_default_on_file    val: 0.1499999
## path: I-L-L-R    entropy: 0.704963233218175   col: person_age    val: 33.9921875   rows: 2838
## path: I-L-L-L    entropy: 0.40973705382045866   col: person_age    val: 21.840624999999967   r
## Entropy in data:   0.7568007498467375
## Entropy in all leafs:  0.574488597974446
```

We can see that the Entropy of all leafs is significantly smaller than the initial

Entropy.

8.3   Forcast Credit Defaults with DT

We need to define a limit that splits all leafs in default and none default. Here do we choose the limit 0.65, that means that leafs with a probability of none default smaller than 65% will be classified as defaults. We can get this rows by using the conditions column in the `leafs`. The resulting analysis shows the quality of the forecast:

```
data_temp = data.copy()
data_temp["ID"] = list(range(len(data_temp)))
conditions = "("+ ") |
↪   (".join(list(leafs.loc[leafs["P_of_no_default"] < 0.65,
↪   leafs.columns ==
↪   "condition"]["condition"].replace("and","&")))+")"
data_temp = data_temp.query(conditions)
X = np.zeros(len(data))
X[list(data_temp["ID"])] = 1

Y = data.loc[:, data.columns == 'loan_status'].to_numpy()[:,0]

print("Wrong answers of the decission tree:
↪   ",np.sum(np.abs(Y-X))/len(Y) * 100, "%")
confusion_matrix(Y,X)
```

```
## Wrong answers of the decission tree:  17.94911144532089 %
## array([[21932,  3541],
##        [ 2307,  4801]], dtype=int64)
```

We can compare the results to the previews created NN (with `hidden_layer_neurons = [4,4]`, `alpha = 0.01`, `epochs = 500`, `batch_size = 2000`):

8.4   Extern Packages

There do exist some packages to create DTs for example sklearn, but i wasnt able to get the data like in my own code:

```
from sklearn.tree import DecisionTreeClassifier, plot_tree,
↪   export_graphviz, export_text
X = data.drop('loan_status',axis=1)
```

```python
y = data['loan_status']
clf =
↪ DecisionTreeClassifier(criterion='entropy',max_depth=4,min_samples_split=1000,min_samples_lea
clf = clf.fit(X,y)
pyplot.figure(figsize=(16,8))
plot_tree(clf, filled=True, feature_names=X.columns,
↪ proportion=False, fontsize=6)
pyplot.show()

r = export_text(clf, feature_names=list(X.columns))
print(r)
```
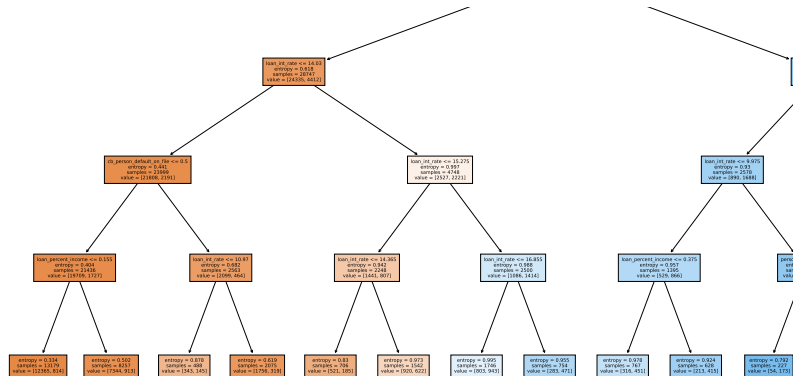


```
## |--- loan_percent_income <= 0.31
## |    |--- loan_int_rate <= 14.03
## |    |    |--- cb_person_default_on_file <= 0.50
## |    |    |    |--- loan_percent_income <= 0.16
## |    |    |    |    |--- class: 0
## |    |    |    |--- loan_percent_income >  0.16
## |    |    |    |    |--- class: 0
## |    |    |--- cb_person_default_on_file >  0.50
## |    |    |    |--- loan_int_rate <= 10.97
## |    |    |    |    |--- class: 0
## |    |    |    |--- loan_int_rate >  10.97
## |    |    |    |    |--- class: 0
## |    |--- loan_int_rate >  14.03
## |    |    |--- loan_int_rate <= 15.28
## |    |    |    |--- loan_int_rate <= 14.37
## |    |    |    |    |--- class: 0
## |    |    |    |--- loan_int_rate >  14.37
```

```
## |   |   |   |   |--- class: 0
## |   |   |--- loan_int_rate >  15.28
## |   |   |   |--- loan_int_rate <= 16.85
## |   |   |   |   |--- class: 1
## |   |   |   |--- loan_int_rate >  16.85
## |   |   |   |   |--- class: 1
## |--- loan_percent_income >  0.31
## |   |--- loan_int_rate <= 12.76
## |   |   |--- loan_int_rate <= 9.97
## |   |   |   |--- loan_percent_income <= 0.38
## |   |   |   |   |--- class: 1
## |   |   |   |--- loan_percent_income >  0.38
## |   |   |   |   |--- class: 1
## |   |   |--- loan_int_rate >  9.97
## |   |   |   |--- person_age <= 22.50
## |   |   |   |   |--- class: 1
## |   |   |   |--- person_age >  22.50
## |   |   |   |   |--- class: 1
## |   |--- loan_int_rate >  12.76
## |   |   |--- loan_int_rate <= 14.31
## |   |   |   |--- class: 1
## |   |   |--- loan_int_rate >  14.31
## |   |   |   |--- class: 1
```

8.5   Appendix (complete code)

```python
import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as pyplot
import math as ma


data =
→   pd.read_csv("example_data/credit_risk_dataset.csv").fillna(0)
data = data.replace({"Y": 1, "N":0})
data = data.loc[:, ["person_age", "loan_percent_income",
→   "loan_int_rate", "cb_person_default_on_file" ,"loan_status"]]



def calc_entropy(df, decision_on = "loan_status"):
  if len(df)==0:
```

```python
    return(0)
  p = np.sum(df[decision_on] == 0)/len(df)
  if p == 0 or p == 1:
    return(0)
  result = -p * ma.log(p,2) - (1-p)*ma.log(1-p,2)
  return result

def calc_splitted_entropy(df, col, val, decision_on =
↪  "loan_status"):
  w = np.sum(df[col] > val)/len(df)
  result = w * calc_entropy(df.loc[df[col] > val], decision_on) +
↪  (1-w) * calc_entropy(df.loc[df[col] <= val], decision_on)
  return result

def find_minima(df, col, decision_on = "loan_status", round_at =
↪  5):
  direction = 1
  step = (df[col].max()-df[col].min()) * 0.1
  val = df[col].min() + step
  best_entropy = 1
  stagnation = 0

  while stagnation <= 15:
    temp = calc_splitted_entropy(df, col, val)
    if temp > best_entropy:
      direction = -direction
      step = 0.5 * step
      stagnation += 1
    elif round(temp,round_at) < round(best_entropy,round_at):
      stagnation = 0
    else:
      stagnation += 1
    best_entropy = temp
    val = val + direction * step

  return best_entropy, val


def find_best_col(df, decision_on = "loan_status", round_at = 5):
  cols = list(df.columns[df.columns != decision_on])
  entropys = np.ones(len(cols))
  vals = np.ones(len(cols))

  for i in range(len(cols)):
    entropys[i], vals[i] = find_minima(df, col=cols[i],
↪  decision_on = "loan_status", round_at = 5)
```

```python
  best_i = int(np.where(entropys == min(entropys))[0][0])
  return cols[best_i], entropys[best_i], vals[best_i]




def make_node_and_leafs(df, decision_on = "loan_status", round_at
↪  = 5, path = "I", condition = "", min_size = 1000, max_depth =
↪  4, leafs = pd.DataFrame(columns=["path", "condition", "rows",
↪  "P_of_no_default", "entropy"])):
  if len(df) < min_size or (path.count("-")-1) >= max_depth or
  ↪  len(df.columns) <= 1:
    leafs = leafs.append({"path":path+"}",
↪  "condition":condition[0:(len(condition)-5)], "rows":len(df),
↪  "P_of_no_default":np.sum(df[decision_on] == 0)/len(df),
↪  "entropy":calc_entropy(df)}, ignore_index=True)
  else:
    col, entropy, val = find_best_col(df, decision_on, round_at)
    print("path:", path, "  entropy:", entropy, "  col:", col, "
    ↪  val:", val, "  rows:", len(df))
    leafs = make_node_and_leafs( df.loc[df[col] > val, df.columns
↪  != col], decision_on, round_at, path+"-R", condition+col+" >
↪  "+str(float(round(val,5)))+" and ", min_size, max_depth,
↪  leafs)
    leafs = make_node_and_leafs( df.loc[df[col] <= val,
↪  df.columns != col], decision_on, round_at, path + "-L",
↪  condition+col+" <= "+str(float(round(val,5)))+" and ",
↪  min_size, max_depth, leafs)
  return(leafs)




leafs = make_node_and_leafs(df=data, decision_on = "loan_status",
↪  round_at = 5, min_size = 1000, max_depth = 4)
leafs["entropy"] = (leafs["entropy"]*leafs["rows"])/len(data)

print("Entropy in data: ", calc_entropy(data))
print("Entropy in all leafs: ", np.sum(leafs["entropy"]))


data_temp = data.copy()
data_temp["ID"] = list(range(len(data_temp)))
```

```python
conditions = "("+ ") |
↪  (".join(list(leafs.loc[leafs["P_of_no_default"] < 0.65,
↪  leafs.columns ==
↪  "condition"]["condition"].replace("and","&")))+")"
data_temp = data_temp.query(conditions)
X = np.zeros(len(data))
X[list(data_temp["ID"])] = 1

Y = data.loc[:, data.columns == 'loan_status'].to_numpy()[:,0]

print("Wrong answers of the decission tree:
↪  ",np.sum(np.abs(Y-X))/len(Y) * 100, "%")
confusion_matrix(Y,X)
```

```
## path: I    entropy: 0.6564119496913492   col: loan_percent_income    val: 0.29990234374999997
## path: I-R    entropy: 0.9103981172376336   col: loan_int_rate    val: 14.01635546875   rows: 4
## path: I-R-L    entropy: 0.9607335396537932   col: cb_person_default_on_file    val: 0.14999999
## path: I-R-L-L    entropy: 0.9648046914552115   col: person_age    val: 34.296875   rows: 2977
## path: I-L    entropy: 0.531844917392527   col: loan_int_rate    val: 14.280073242187505   rows
## path: I-L-R    entropy: 0.998669387657678   col: person_age    val: 22.3046875   rows: 4057
## path: I-L-R-R    entropy: 0.9992350724472542   col: cb_person_default_on_file    val: 0.149999
## path: I-L-L    entropy: 0.44498607709157834   col: cb_person_default_on_file    val: 0.1499999
## path: I-L-L-R    entropy: 0.704963233218175   col: person_age    val: 33.9921875   rows: 2838
## path: I-L-L-L    entropy: 0.40973705382045866   col: person_age    val: 21.840624999999967   r
## Entropy in data:  0.7568007498467375
## Entropy in all leafs:  0.574488597974446
## Wrong answers of the decission tree:  17.94911144532089 %
## array([[21932,  3541],
##        [ 2307,  4801]], dtype=int64)
```