My First Steps in Neuronal Networks (Beginners Guide)

Axel Roth

2021-11-07

Contents

Chapter 1

About

1.1 Me

Hello, my name is Axel Roth and im studding math at a master degree in germany. In the same moment im working half-time in the finance field and programming in a wide range of tasks like index replication, factormodels, data visualisation, datamanagement, shiny applications, documantations with rmarkdown, building internal packages and much more. So the most of my experience i gained was around R and all its features. And now the interesting question... Why do i want to write a beginners guide in the field of Neuronal Networks?
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. Additionaly 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. Thats 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.

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 runs over the training data := `n_train`
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` 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 traings dataset is the following:

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

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

The provided training dataset contains the `X` matrix with two inputs for each scenario and the `Y` matrix with the correct output. 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 and default options are the following:

```
import numpy as np
import random as ra
import pandas as pd
import matplotlib.pyplot as pyplot
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 1000)
```

(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 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 look deeper 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 (with the matrix dimensions included) for one scenario out of the training dataset, the following:

$$step(W^{(1,2)} \cdot x^{(2,1)}) = y^{(1,1)}$$

That is the normal approach 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.

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:

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

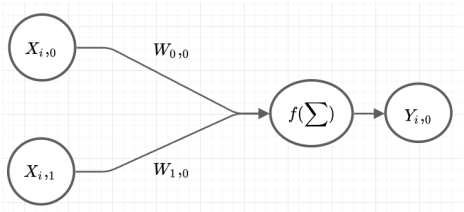
The new formular looks like this:

$$step(X \cdot W) = Y$$

For example if you take a look at the i-th row or scenario of X you will see the following:

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

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



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

2.2 Forward pass

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

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

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

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

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

Here an small example for the forward pass:

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

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

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

2.3 backward pass

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

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

with

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

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

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

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

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

```
])
alpha = 0.01
W = backward(W, X, Y, alpha, Y_approx)
print(W)
```

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

and this is the new weight.

2.4 Single Perceptron

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

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

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)
```

The KNN is trained now. Now we want to look at the error. 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:

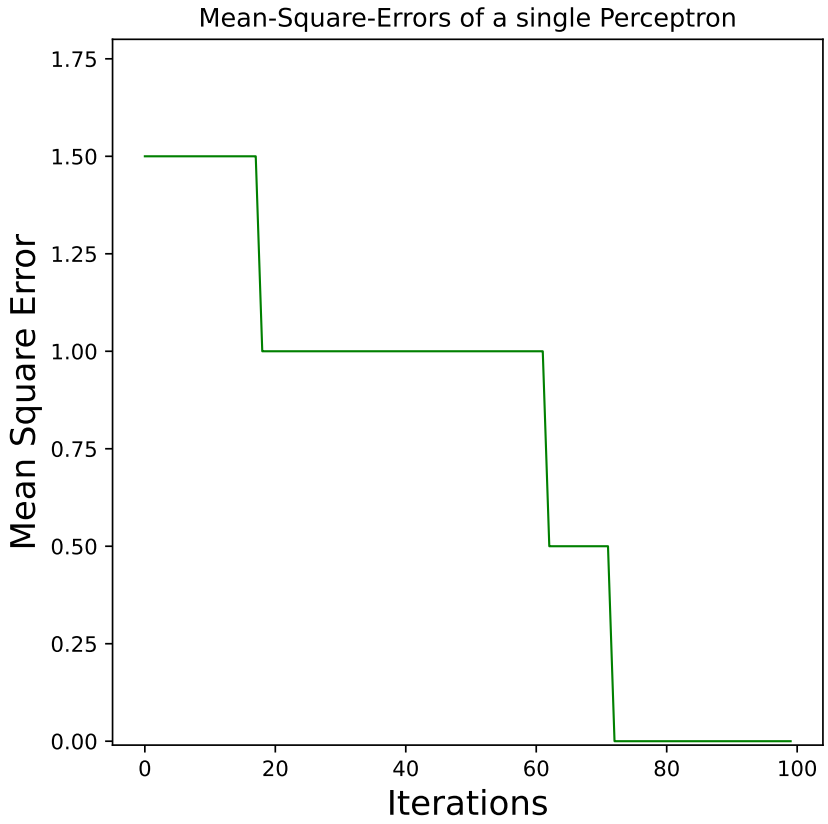
```
def mean_square_error(error):
  return( 0.5 * np.sum(error ** 2) )
```

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

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

To plot the errors, im using the following function:

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


plot_error(mean_square_errors, "Mean-Square-Errors of a single Perceptron")
```



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

2.5 Appendix (complete code)

```
import numpy as np
import random as ra
import pandas as pd
import matplotlib.pyplot as pyplot
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 1000)

X = np.array([
```

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

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


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

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


errors = []
for i in range(train_n*5):
  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, "Mean-Square-Errors of a 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? It is still the OR-Gate but now it will never give you 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 shift 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:

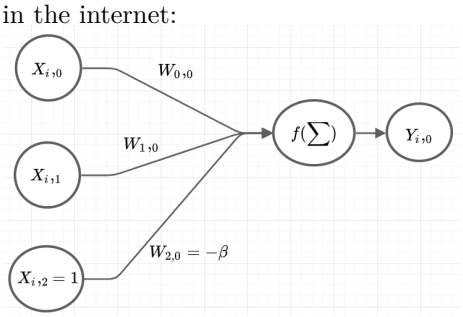
$$step(X \cdot W - \beta) = Y$$

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

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

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

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



As python code we did the following:

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

W = np.array([
  [0.1],
  [0.2]
])
W = np.append(W, -np.array([np.random.random(len(W[0]))]).T, axis=0)
#W = np.append(W, np.array([[-5]]), axis=0)
```

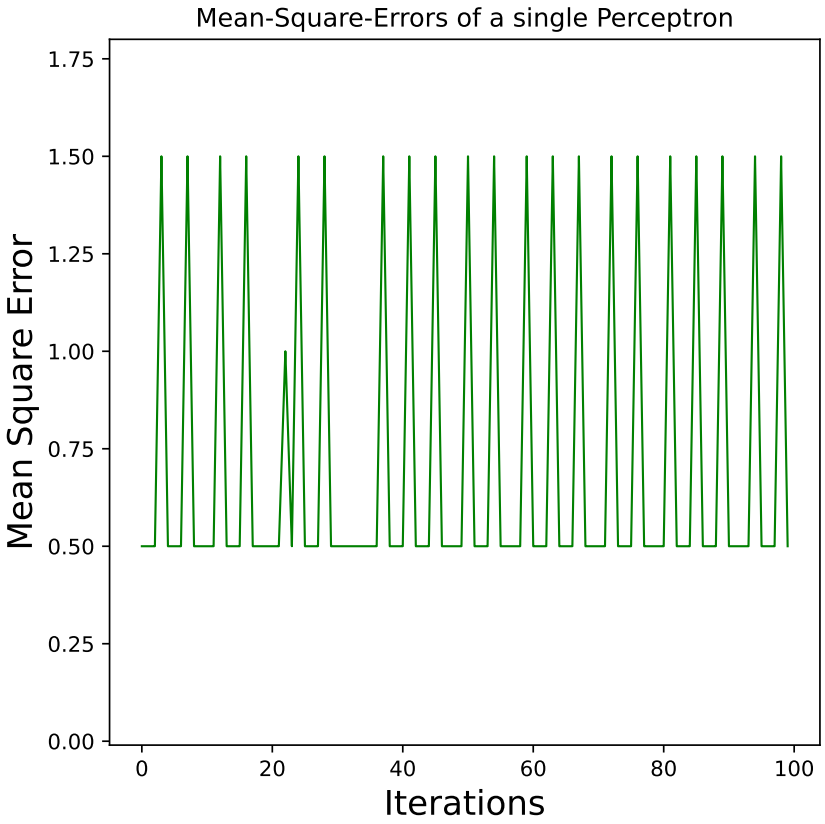
We added $+5$ to the X matrix to simulate the problem of shifted data, added ones on the left side of X and added negative random numbers between $(0, 1)$ to the weights. Yes if you would have a clue, what β would be great for the given problem, its better to choose it explicit. The new problem needs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

3.2 Appendix (complete code)

The complete code is the following:

```
import numpy as np
import random as ra
import pandas as pd
import matplotlib.pyplot as pyplot
pd.set_option('display.max_rows', 500)
pd.set_option('display.max_columns', 500)
pd.set_option('display.width', 1000)

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
train_n = 100

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(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, "Mean-Square-Errors of a single Perceptron")
```


Mean-Square-Errors of a single Perceptron
Mean Square Error
Iterations
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
0
20
40
60
80
100

Chapter 4

Multi Layer Perceptrons

In a multi layer Perceptron you have multiple layers of neurons. Thats why you 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. You can interprete each layer as a single Perceptron.