My First Steps in Neuronal Networks

Axel Roth

2021-11-06

Contents

Chapter 1

About

1.1 Me

Hello, my name is Axel Roth and at the moment im studieing math at a master degree. 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 experiece i gained was around R and all its features. And now the iterresting 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 programm 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 programm 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.

Chapter 2

A single Perceptron

In this chapter i will teach you how to code a single Perceptron in Python with more or the less 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 interprete Vectors as one dimensional matrixes!). At the most of the times its just translating math formulars 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 := `train_n`
Learning rate := α = `alpha`
and the activation function:

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

This function is named the heavyside-function and should be the easiest activation function to start with.

The traings dataset is the following:

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

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

The provided traings dataset contains the `X_train` matrix with two inputs each scenario and the `Y_train` 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.

For a wider use we will now change the `X_train` matrix a little bit by adding a

column with ones on the left side of it. Now you will not understand but later you will. In short, its a dummy value to make it possible to shift the NN, so it can be better fitted. The new traings dataset looks like this:

$$\begin{bmatrix} x_{i,0} & x_{i,1} & x_{i,2} & \big| & y_i \end{bmatrix}$$

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

The needed python imports and default options are the following:

```python
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 backward pass. In the forward pass we are calculating the weighted sum of each input with its weights of the layer to get the output. In the backward pass we are analysing the error to adjust the weights accordingly. This is it! This is all a NN will do. Now that you know, see you. 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 weighted sum like i said, so you have with the dimensions included for one scenario out of the traingdata the folowing:

$$step(W^{(1,3)} \cdot x^{(3,1)}) = y^{(1,1)}$$

That is the normal approach to iterate over all scenarios in the trainingdata... 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 trainingdata in one formular. If your data isnt that huge, its a much faster approach aswell. First of all we

need to interpret the new dimensions of W and X.
We have X as:

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

each row describes the inputs for each neuron in the scenario i.
For the weights W we have for example:
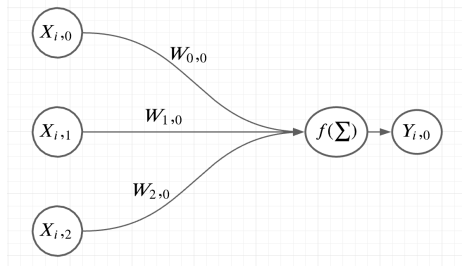
$$W = \begin{bmatrix} 0.1 \\ 0.2 \\ 0.3 \end{bmatrix}$$

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:

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

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

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

2.2 Forward pass

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

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

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

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

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

Here an small example for the forward pass:

```python
X = np.array([
  [1,0,0],
  [1,0,1],
  [1,1,0],
  [1,1,1],
])
W = np.array([
  [0.1],
  [0.2],
  [0.3]
])
Y_approx = forward(X, W)
print(Y_approx)
```

```
## [[1]
##  [1]
##  [1]
##  [1]]
```

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 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.

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

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

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

and this is the new weight.

2.4 Single Perceptron

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

```python
X = np.array([
  [1,0,0],
  [1,0,1],
  [1,1,0],
  [1,1,1],
])
Y = np.array([
  [0],
  [1],
  [1],
  [1]
])
W = np.array([
  [0.1],
  [0.2],
  [0.3]
])
alpha = 1

errors = []
for i in range(20):
  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:

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

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

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

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

Mean-Square-Errors of a single Perceptron
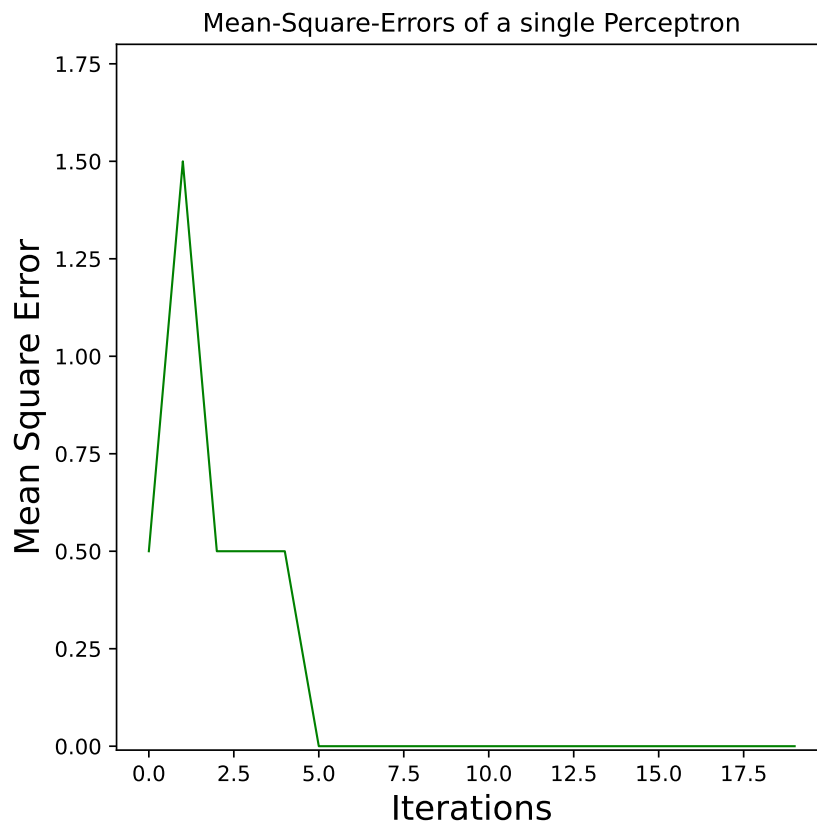
If you survived until now, you have learned how you can programm a single Perceptron!

2.5   Appendix (full code)

```python
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([
```

```python
    [1,0,0],
    [1,0,1],
    [1,1,0],
    [1,1,1],
])
Y = np.array([
    [0],
    [1],
    [1],
    [1]
])
W = np.array([
    [0.1],
    [0.2],
    [0.3]
])
alpha = 1


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(20):
  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)))
```

```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.xticks(x)
  pyplot.show()


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