# Exercise project X – Experimentation with neural network mathematics

## Introduction

In this project, I will look at the `nn_math_tool_2_scaled_realdata` notebook from one of the lectures and will experiment with neural network mathematics and what they consist of.

## Experiments

### Input data

I changed the data source to `generate_train_data` to be able to experiment with the input data, as well. The original training loss at the end of the 300$^{th}$ epoch was around 0.0109. To see how the model handles a bigger range of numbers and bigger noise, I changed the n1 and n2 ranges to 0-20 and 0-50, and the noise to 0-10. Interestingly, it resulted in a smaller loss, around 0.0047. Probably, the wider range was what helped, because this way, ReLU can provide more useful gradients. Scaling may also work better on a larger range than a simple 0-5 range.

```python
def generate_train_data():
    result = []

    # create 100 numbers
    for x in range(100):
        n1 = np.random.randint(0, 20)
        n2 = np.random.randint(0, 50)


        n3 = n1 ** 2 + n2 + np.random.randint(0, 10)
        n3 = int(n3)

        result.append([n1, n2, n3])


    return result
```

*1Modified generate_train_data function*

## Initial weights and biases

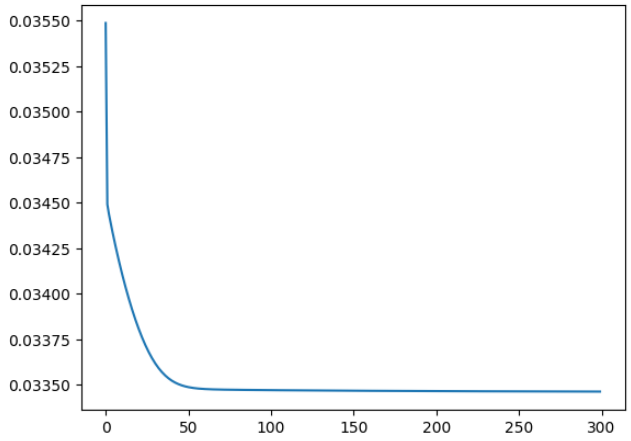
For the remaining experiments, I used the medical insurance dataset, so that results can be more appropriate with real data. Our initial training loss is 0.0335 before changing anything, and the loss curve is stable. At first, I increased the weights and biases to values at around 2-3. It didn't result in a big change; the training loss was almost the same and the plot also looked stable.

```
w1 = 2
w2 = 2.5
w3 = 3
w4 = -2
w5 = -2.5
w6 = 3.5
bias1 = -3
bias2 = 2
bias3 = 3
```

Then I changed them to very small values close to zero. The training loss remained the same, but the learning process became a bit slower.

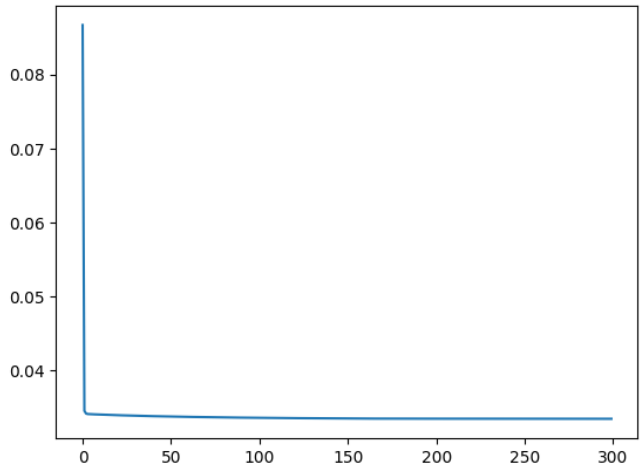
```
w1 = 0.002
w2 = 0.0025
w3 = 0.003
w4 = -0.002
w5 = -0.0025
w6 = 0.0035
bias1 = -0.003
bias2 = 0.002
bias3 = 0.003
```

*2 The smaller weights and biases*




*3 Slower learning with smaller values*

I also tried seeing what happens if all the weights and biases are equal (exactly 1). The outcomes were also very similar, training loss being 0.0335, and again a faster learning process.



```
w1 = np.random.normal(0, 1.5)
```

Then I tried using a random number using normal distribution for all weights and biases, which produced almost the same numbers and plot. Moreover, the new weights the model produced were on the same scale as the original numbers, but the proportions between the weights and the biases changed.

```
NEW WEIGHTS AND BIASES
w1: 0.0015999953524905685
w2: 0.0003583725664477503
w3: 0.0026746691254171703
w4: -0.002273805599556278
w5: -0.0025179859765810694
w6: 0.28204324674653597
b1: -0.0035252967985168375
b2: 0.004256923974092913
b3: 0.1930068345923557
```

*4 Final weights and biases with low starting values*

```
NEW WEIGHTS AND BIASES
w1: 1.3802343059295634
w2: 2.3681755068844033
w3: 2.777629588934456
w4: -2.085303777286159
w5: 0.7910183091247122
w6: 0.048173001125924333
b1: -3.303619674237175
b2: 1.8664258441503485
b3: 0.09945349181662441
```

*5 Weights and biases with random starting values*

## Learning rate

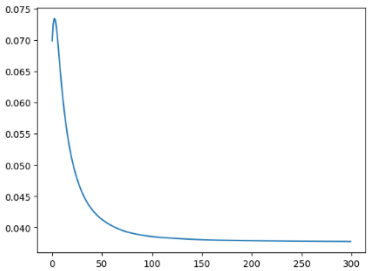
It is very important to find the optimal learning rate. Too big learning rate can cause unstable learning, while too small needs more resources and time to learn. Therefore, I tried increasing it from a smaller to a larger number.

### 0.00005

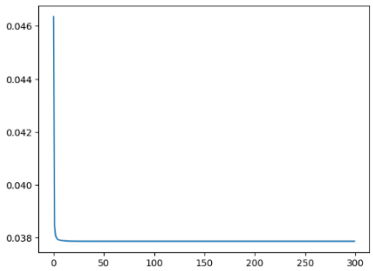
This caused a slower, but more stable learning process, with a small increase in loss in the beginning.

### 0.005

This was the original one, with a faster learning, but remaining stable.




*6 Learning rate = 0.00005*




*7 Learning rate = 0.005*

### 0.01

This was still stable with a slightly higher (0.038) training loss.

### 0.1

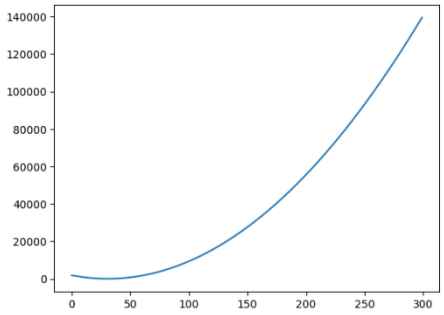
Although the plot still looked to be stable, the loss increased again to 0.041 and almost all the new weights and biases became negative numbers.

### 1

I tried it just out of curiosity. Of course, the outcome was horrible, with an increasing loss.

```
NEW WEIGHTS AND BIASES
w1: 1.441533915123739
w2: -3.095671238998893
w3: -1.0402300867750265
w4: -4.644639032227141
w5: -0.1935066532006305
w6: -1.4670865591264255
b1: -1.2507838954561243
b2: -2.1457967046981667
b3: 0.2082893543362356
```

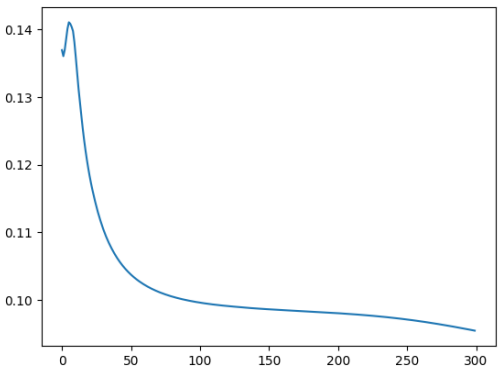
*8 Negative weights and biases*




*9 Learning rate = 1*

## Dataset size

To test with the dataset size, I started with a very small number and increased it to see how it improves according to the amount of training data. With 30 rows, the model learns slowly, with a higher loss. I was assuming that this low amount of data with 300 epochs would probably overfit the model, so I asked GitHub Copilot to generate a code for evaluating the model.




*10 Change of loss with 30 rows of training data*

And yes, $R^2$ was 0.05 and MAPE 189%. By increasing the dataset size, the MAPE slowly decreased, although the best result was still 125% (testing with the original example). The $R^2$ always stayed close to 0. So even though I got worse metrics with a smaller dataset, the difference was not significant.

## Last thoughts and Keras / TensorFlow comparison

This exercise helped me a lot to understand how exactly neural networks work. I already had some knowledge about deep learning, but even though I knew the basics, it was hard to understand how these many steps build up to form a whole. Backpropagation seemed to be black magic for me before seeing an exact example.

Although we created a very simple neural network, it became a long code. We also had to do the derivation part manually, which means if we make any changes in the structure, the whole backpropagation part must be recalculated. That's one of the reasons why we are using frameworks like Keras, because it is easier to code and to scale. These frameworks are well optimized and stable, and they have numerous built-in layers, activation functions, optimizers etc. All in all, they help a lot when it comes to building deep learning models, because machine learning is all about experimenting, so it saves time and resources.

*For this exercise, I used generative AI to help create the model evaluation code, to provide explanations for the results I obtained, and finally to verify that my conclusions were both technically accurate and grammatically correct.*