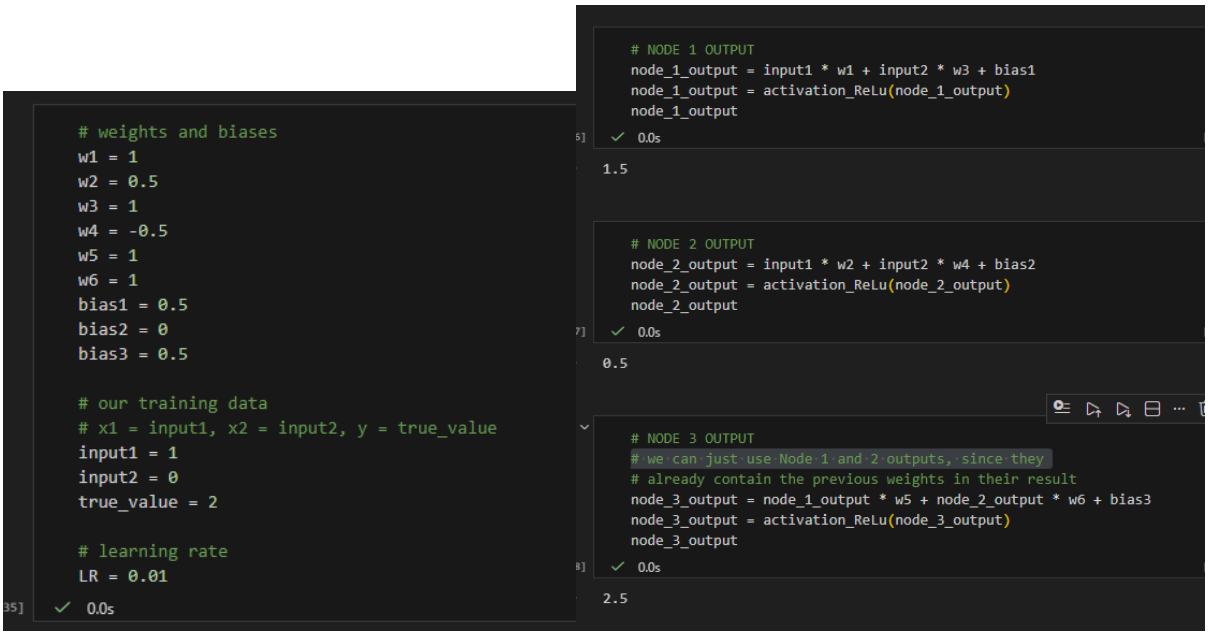
ProjectX

In this project, I have used the neural network math example from my lecturer's github repository, which is called "nn_math_experiment1.ipynb" from the "lecture6_nn_math1" folder. The lecturer implemented a small neural network manually with three nodes, to see an example of how a neural network calculates all outputs, derivatives and weight updates using Python Code. My goal is to do to three experiments, which focuses on adjusting the starting weights and biases to see if there is an effect. Also, I want to see if there is an difference, if some weights and biases have identical values or if all of the weights and biases have unique starting values and lastly I would like to test the learning rate and make a comparison between the range of 0.005 and 0.25 to see the difference and effect on the neural model.



```
# weights and biases
w1 = 1
w2 = 0.5
w3 = 1
w4 = -0.5
w5 = 1
w6 = 1
bias1 = 0.5
bias2 = 0
bias3 = 0.5

# our training data
# x1 = input1, x2 = input2, y = true_value
input1 = 1
input2 = 0
true_value = 2

# learning rate
LR = 0.01
```

0.0s

```
# NODE 1 OUTPUT
node_1_output = input1 * w1 + input2 * w3 + bias1
node_1_output = activation_ReLu(node_1_output)
node_1_output
```

0.0s

1.5

```
# NODE 2 OUTPUT
node_2_output = input1 * w2 + input2 * w4 + bias2
node_2_output = activation_ReLu(node_2_output)
node_2_output
```

0.0s

0.5

```
# NODE 3 OUTPUT
# we can just use Node 1 and 2 outputs, since they
# already contain the previous weights in their result
node_3_output = node_1_output * w5 + node_2_output * w6 + bias3
node_3_output = activation_ReLu(node_3_output)
node_3_output
```

0.0s

2.5

```
# compare predicted value with true value
print(f"Predicted: {node_3_output} --> True value should be: {true_value}")
```

0.0s

```
Predicted: 2.5 --> True value should be: 2
```

```
# LOSS FUNCTION - we are going to use MSE -> mean squared error
# MSE formula for LOSS => (predicted_value - true_value) ^ 2
predicted_value = node_3_output
loss = (predicted_value - true_value) ** 2
loss
```

0.0s

```
0.25
```

Adjusting the starting weights and biases – Is there an effect?

This experiment is divided into 9 small experiments, where I test 9 different combinations with different amounts of weights and biases to see which combination had the biggest effect. But I can already tell, that changing weights and biases in general has an effect, because of the different outputs.

Small weights + small biases

```
# weights and biases
w1 = 0.3
w2 = 0.1
w3 = 0.3
w4 = -0.2
w5 = 0.5
w6 = 0.3
bias1 = 0.2
bias2 = 0
bias3 = 0.2

# our training data
# x1 = input1, x2 = input2, y = true_value
input1 = 1
input2 = 0
true_value = 2

# learning rate
LR = 0.01
```

✓ 0.0s

Forward Pass

```
# NODE 1 OUTPUT
node_1_output = input1 * w1 + input2 * w3 + bias1
node_1_output = activation_ReLu(node_1_output)
node_1_output
```

✓ 0.0s

```
0.5
```

```
# NODE 2 OUTPUT
node_2_output = input1 * w2 + input2 * w4 + bias2
node_2_output = activation_ReLu(node_2_output)
node_2_output
```

✓ 0.0s

```
0.1
```

```
# NODE 3 OUTPUT
# we can just use Node 1 and 2 outputs, since they
# already contain the previous weights in their result
node_3_output = node_1_output * w5 + node_2_output * w6 + bias3
node_3_output = activation_ReLu(node_3_output)
node_3_output
```

✓ 0.0s

```
0.48000000000000004
```

```
# compare predicted value with true value
print(f"Predicted: {node_3_output} --> True value should be: {true_value}")
```

✓ 0.0s Python

```
Predicted: 0.48000000000000004 --> True value should be: 2
```

```
# LOSS FUNCTION - we are going to use MSE -> mean squared error
# MSE formula for LOSS => (predicted_value - true_value) ^ 2
predicted_value = node_3_output
loss = (predicted_value - true_value) ** 2
loss
```

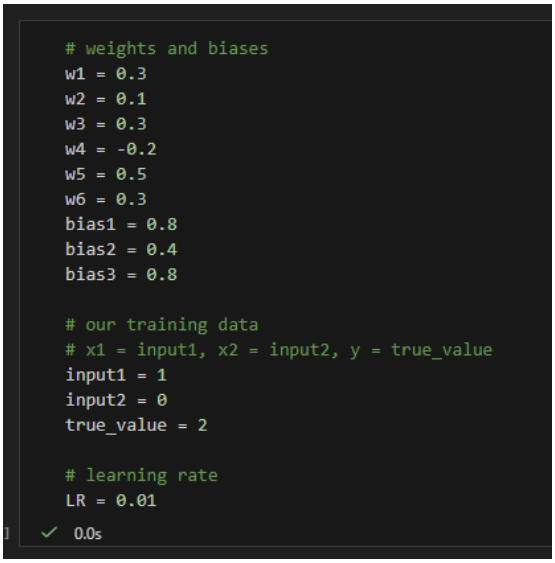
✓ 0.0s Python

```
2.3104
```

In this first test, I have used a combination of small weights and small biases. The network's outputs were extremely low and the final prediction was 0.48, which

was less than 2. I feel like that using too small values does not give the neural network enough strength to work with the nodes properly, which shows that starting with small weights and biases makes the network learn very slowly and also make poor predictions.

Small weights + medium biases

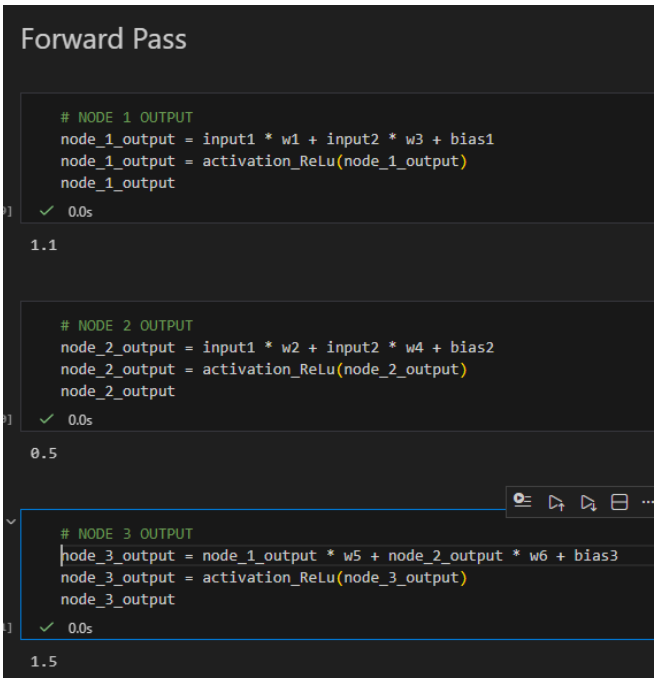


```
# weights and biases
w1 = 0.3
w2 = 0.1
w3 = 0.3
w4 = -0.2
w5 = 0.5
w6 = 0.3
bias1 = 0.8
bias2 = 0.4
bias3 = 0.8

# our training data
# x1 = input1, x2 = input2, y = true_value
input1 = 1
input2 = 0
true_value = 2

# learning rate
LR = 0.01
```

✓ 0.0s



Forward Pass

```
# NODE 1 OUTPUT
node_1_output = input1 * w1 + input2 * w3 + bias1
node_1_output = activation_ReLu(node_1_output)
node_1_output
```

✓ 0.0s

1.1

```
# NODE 2 OUTPUT
node_2_output = input1 * w2 + input2 * w4 + bias2
node_2_output = activation_ReLu(node_2_output)
node_2_output
```

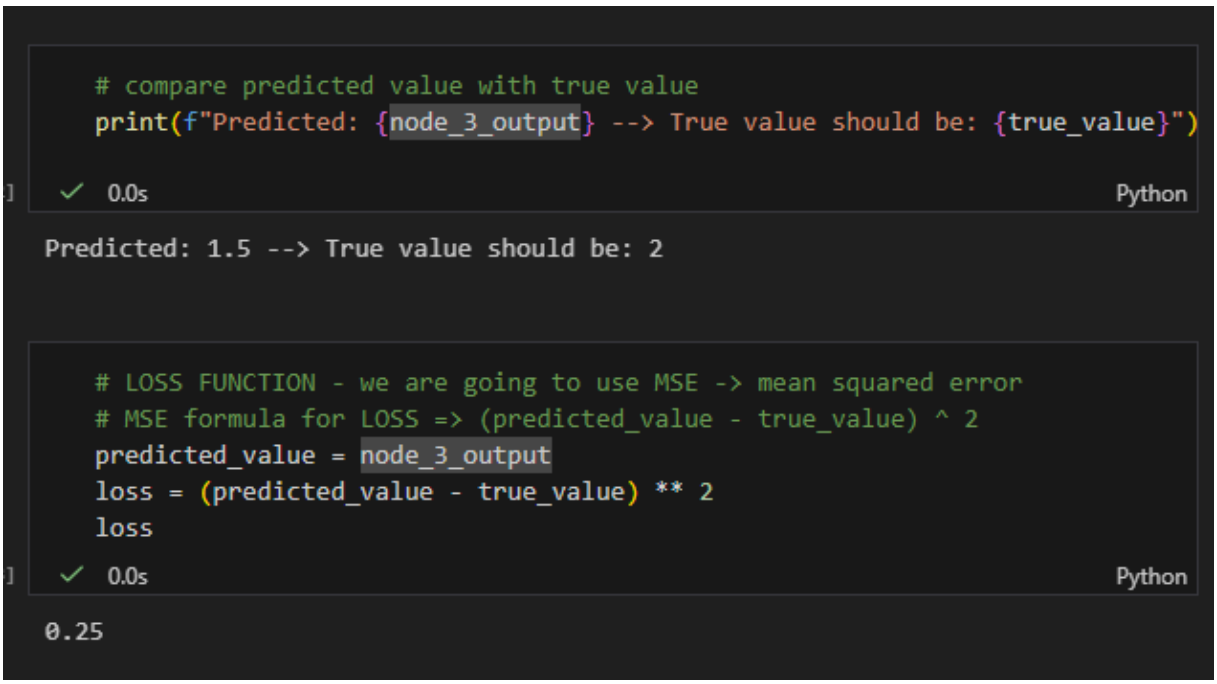
✓ 0.0s

0.5

```
# NODE 3 OUTPUT
node_3_output = node_1_output * w5 + node_2_output * w6 + bias3
node_3_output = activation_ReLu(node_3_output)
node_3_output
```

✓ 0.0s

1.5



```
# compare predicted value with true value
print(f"Predicted: {node_3_output} --> True value should be: {true_value}")
```

✓ 0.0s Python

```
Predicted: 1.5 --> True value should be: 2
```

```
# LOSS FUNCTION - we are going to use MSE -> mean squared error
# MSE formula for LOSS => (predicted_value - true_value) ^ 2
predicted_value = node_3_output
loss = (predicted_value - true_value) ** 2
loss
```

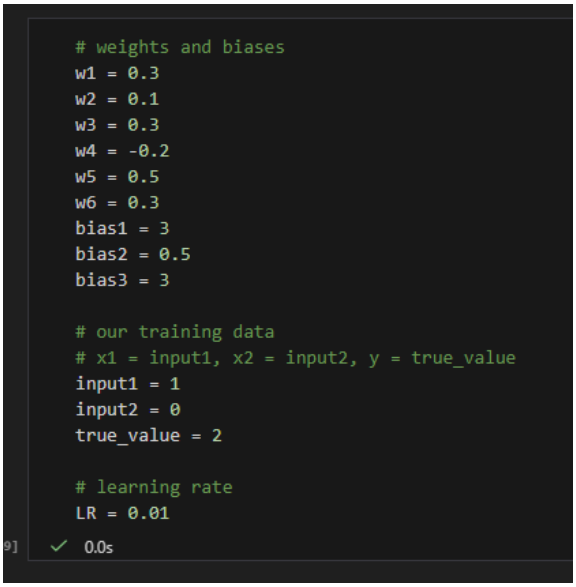
✓ 0.0s Python

```
0.25
```

In this test, I used small weights but medium bias values. This combination had a bigger effect in my opinion than the previous combination, because the final prediction became 1.5 instead of 0.48, which came closer to the true value. I

personally think that if the bias values are higher to a certain amount in general that the neural model performs better and produces better predictions. Also, it works better with the ReLU function.

Small weights + high biases

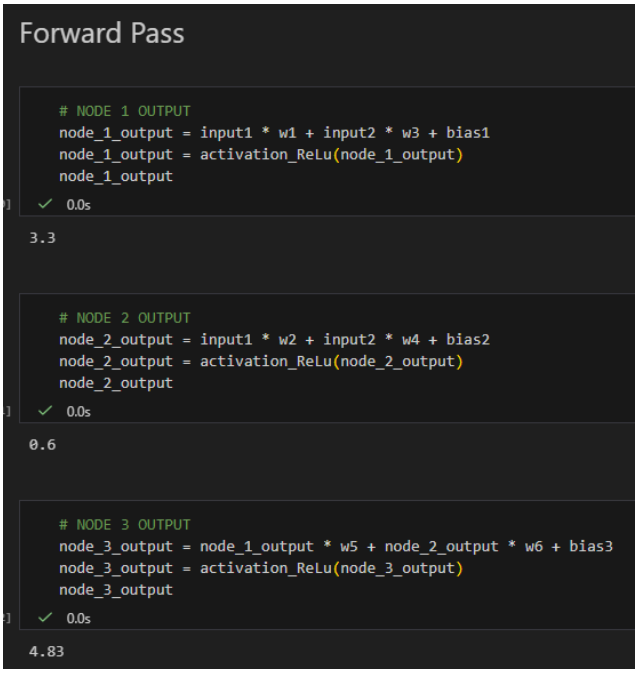


```
# weights and biases
w1 = 0.3
w2 = 0.1
w3 = 0.3
w4 = -0.2
w5 = 0.5
w6 = 0.3
bias1 = 3
bias2 = 0.5
bias3 = 3

# our training data
# x1 = input1, x2 = input2, y = true_value
input1 = 1
input2 = 0
true_value = 2

# learning rate
LR = 0.01
```

✓ 0.0s



Forward Pass

```
# NODE 1 OUTPUT
node_1_output = input1 * w1 + input2 * w3 + bias1
node_1_output = activation_ReLu(node_1_output)
node_1_output
```

✓ 0.0s

3.3

```
# NODE 2 OUTPUT
node_2_output = input1 * w2 + input2 * w4 + bias2
node_2_output = activation_ReLu(node_2_output)
node_2_output
```

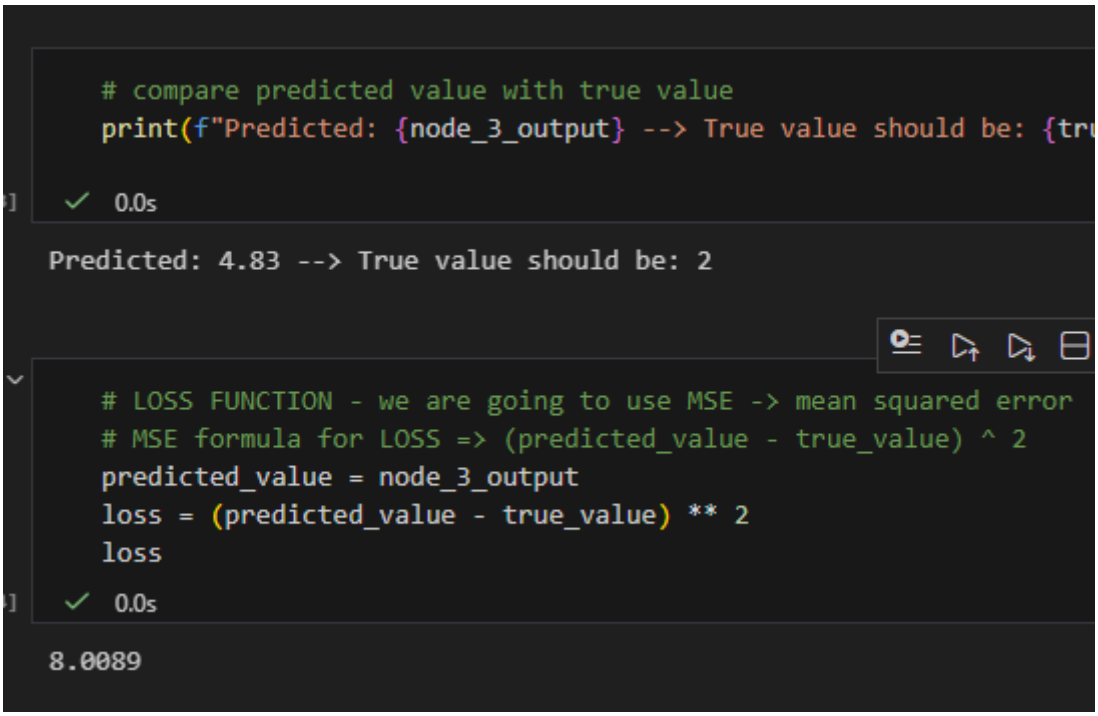
✓ 0.0s

0.6

```
# NODE 3 OUTPUT
node_3_output = node_1_output * w5 + node_2_output * w6 + bias3
node_3_output = activation_ReLu(node_3_output)
node_3_output
```

✓ 0.0s

4.83



```
# compare predicted value with true value
print(f"Predicted: {node_3_output} --> True value should be: {tru
```

✓ 0.0s

```
Predicted: 4.83 --> True value should be: 2
```

```
# LOSS FUNCTION - we are going to use MSE -> mean squared error
# MSE formula for LOSS => (predicted_value - true_value) ^ 2
predicted_value = node_3_output
loss = (predicted_value - true_value) ** 2
loss
```

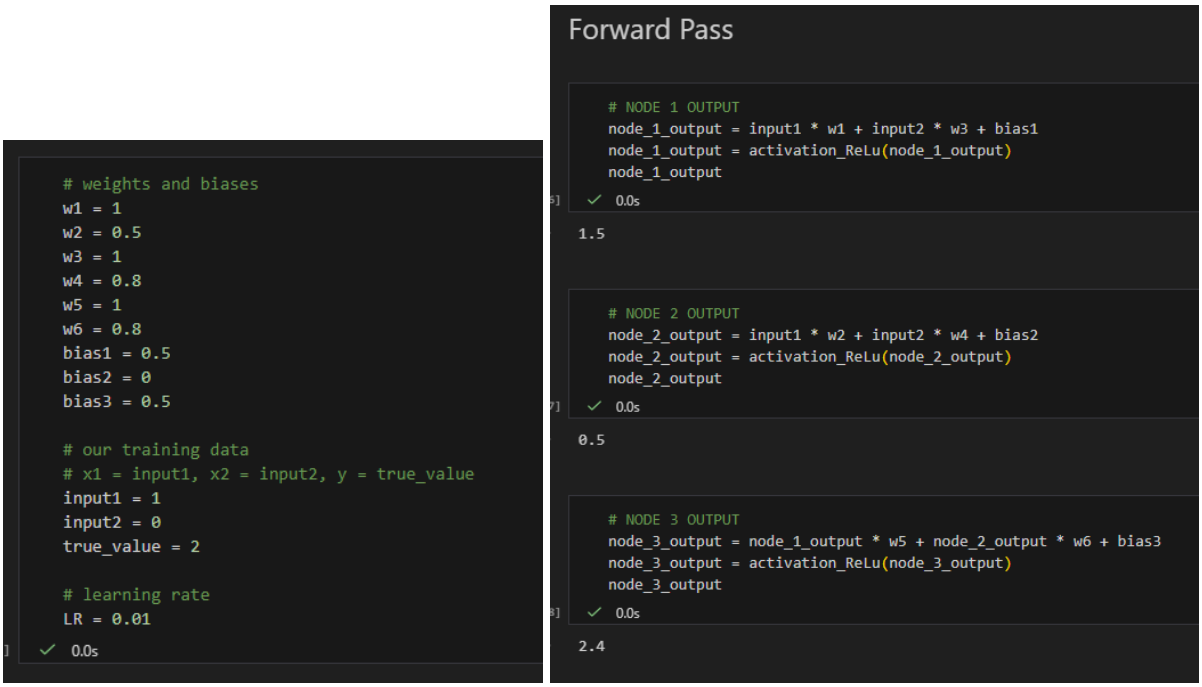
✓ 0.0s

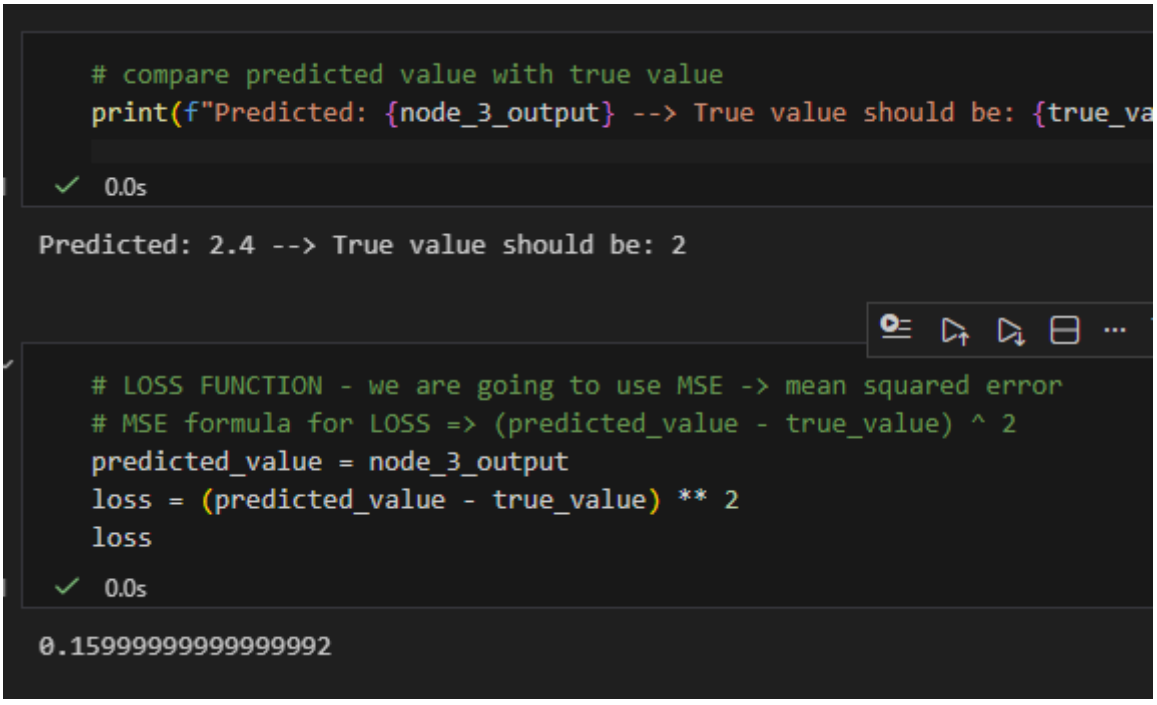
```
8.0089
```

In this experiment, I kept the weights still small, but I increased the bias values even higher than before, to see if the neural model performs better. But unfortunately, the neural network guessed double the amount of the true value

(4.83) and that is the reason the loss became also larger. Even though we used small weights, increasing values of different parameters in Deep Learning, does not always mean that the model performance increases. We have to find the right amount always, for the weights as well as for the biases.

Medium weights + small biases

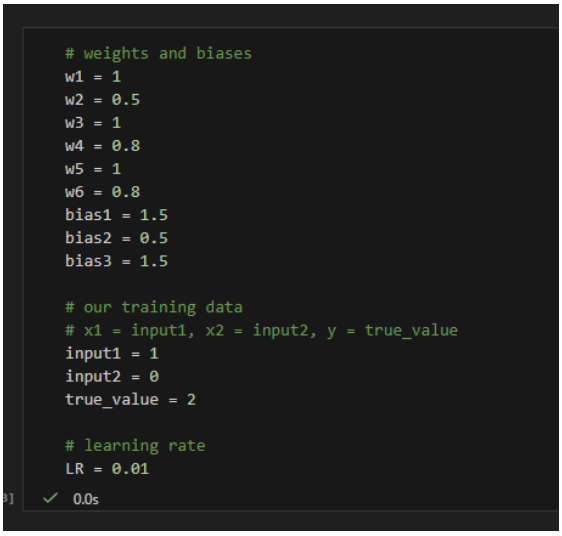




In this experiment, I increased the weights to medium values while keeping the biases small. I think that this combination works well, because we have a final

predicted value of 2.4, which is closer to the true value of 2. And the loss became very low, which is fantastic. For now, the results of this experiment were the best compared to the previous ones.

Medium weights + medium biases

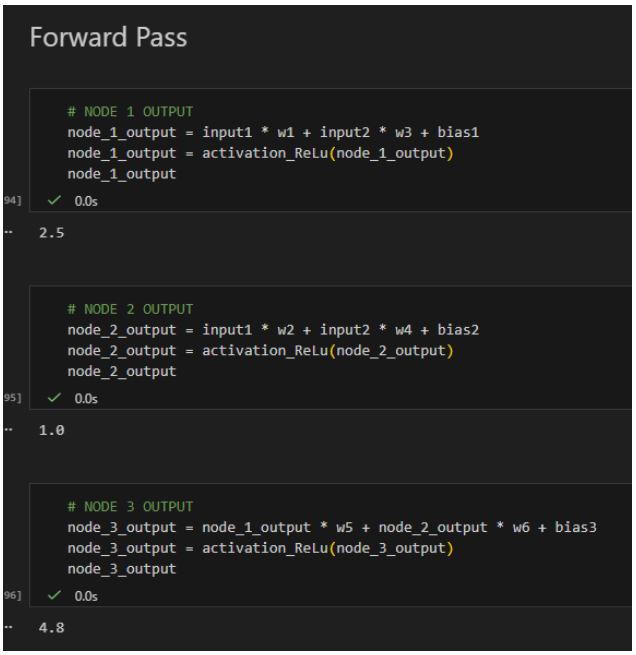


```
# weights and biases
w1 = 1
w2 = 0.5
w3 = 1
w4 = 0.8
w5 = 1
w6 = 0.8
bias1 = 1.5
bias2 = 0.5
bias3 = 1.5

# our training data
# x1 = input1, x2 = input2, y = true_value
input1 = 1
input2 = 0
true_value = 2

# learning rate
LR = 0.01
```

✓ 0.0s



Forward Pass

```
# NODE 1 OUTPUT
node_1_output = input1 * w1 + input2 * w3 + bias1
node_1_output = activation_ReLu(node_1_output)
node_1_output
```

✓ 0.0s

2.5

```
# NODE 2 OUTPUT
node_2_output = input1 * w2 + input2 * w4 + bias2
node_2_output = activation_ReLu(node_2_output)
node_2_output
```

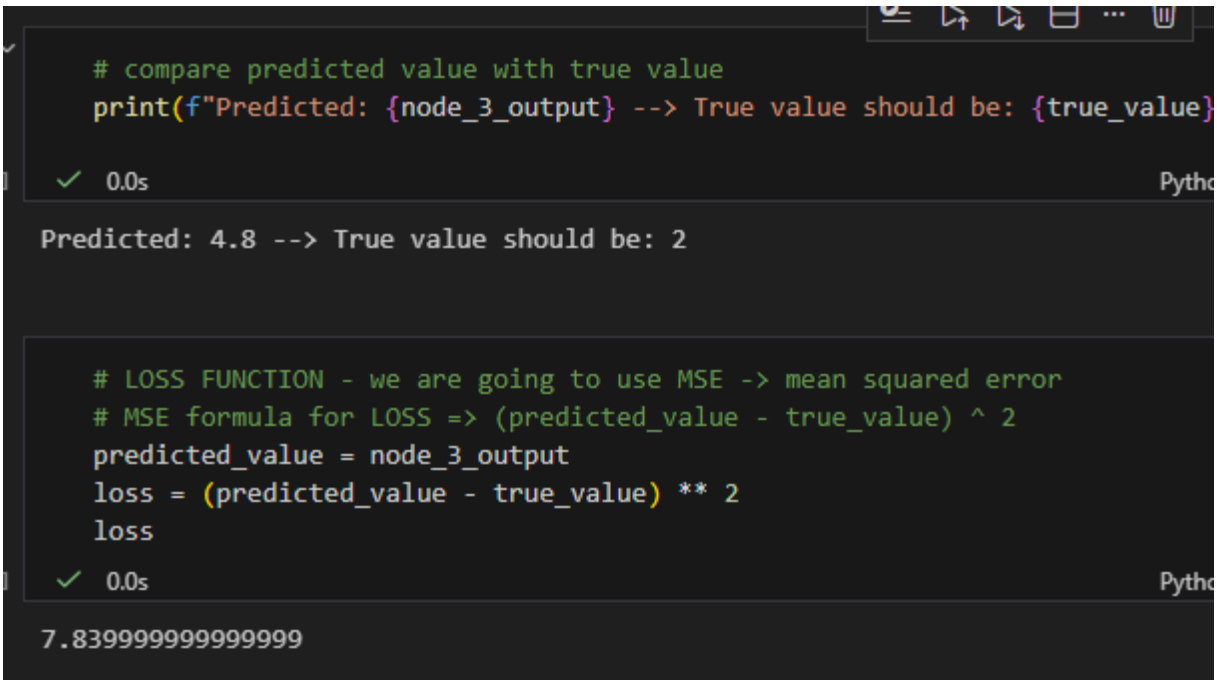
✓ 0.0s

1.0

```
# NODE 3 OUTPUT
node_3_output = node_1_output * w5 + node_2_output * w6 + bias3
node_3_output = activation_ReLu(node_3_output)
node_3_output
```

✓ 0.0s

4.8



```
# compare predicted value with true value
print(f"Predicted: {node_3_output} --> True value should be: {true_value}")
```

✓ 0.0s

```
Predicted: 4.8 --> True value should be: 2
```

```
# LOSS FUNCTION - we are going to use MSE -> mean squared error
# MSE formula for LOSS => (predicted_value - true_value) ^ 2
predicted_value = node_3_output
loss = (predicted_value - true_value) ** 2
loss
```

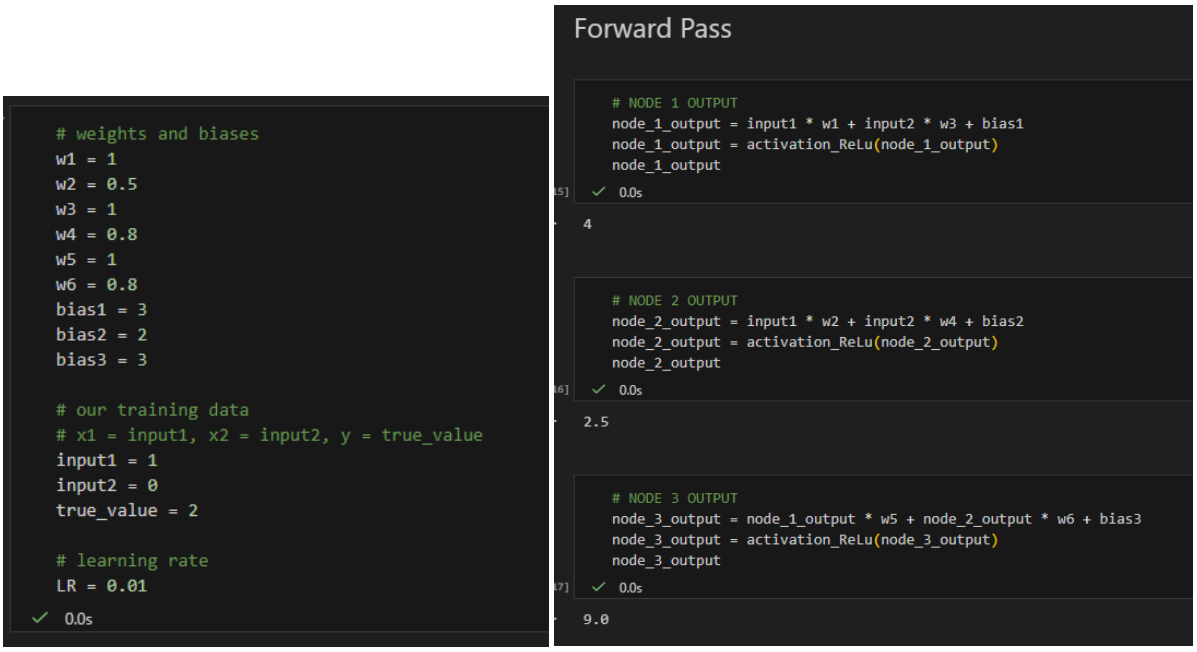
✓ 0.0s

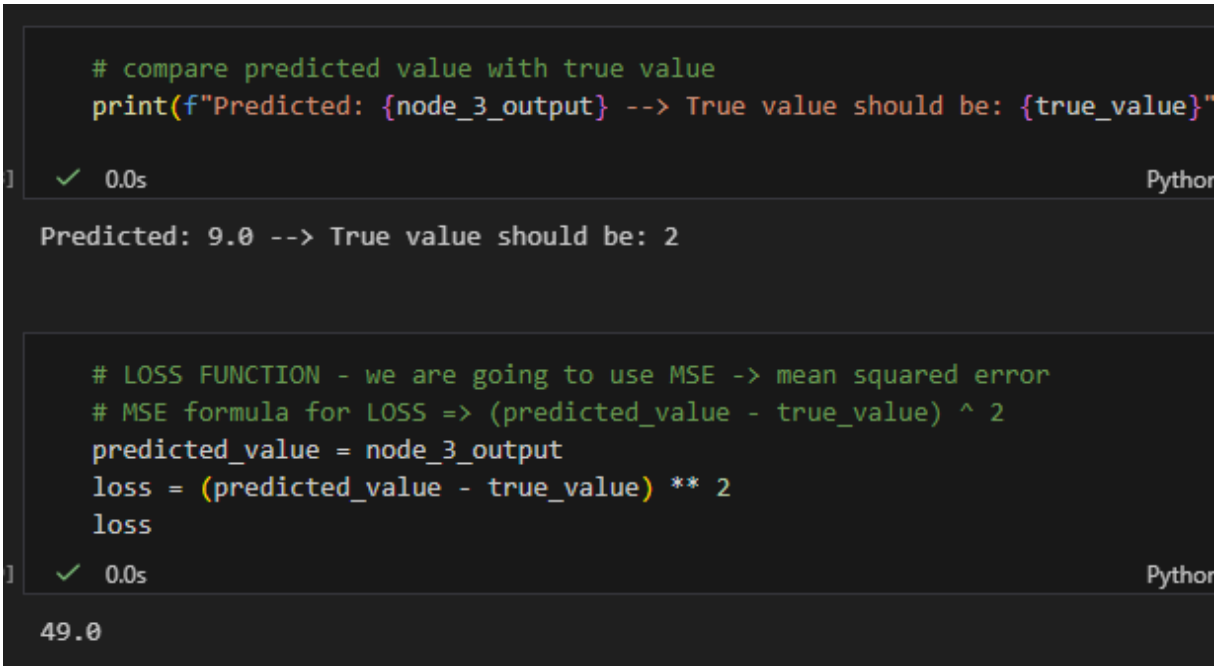
```
7.839999999999999
```

In this test we combined medium values for weights and biases. This combination unfortunately did not work out well, because the neural network predicted the true value too high and the loss value became very large. I assume

from these results that if both weights and biases are strong at the same time, the model becomes unbalanced and produces unrealistic predictions.

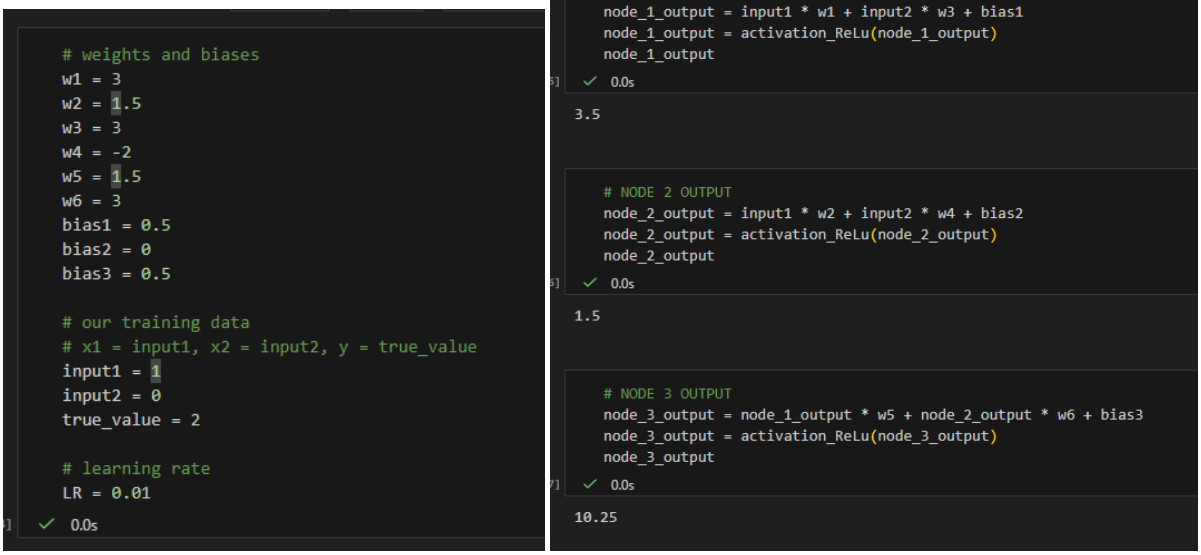
Medium weights + high biases





I have used medium weights together with very high biases. Unfortunately, the results show, that this combination will lead to chaotic outputs. The predicted value became more than four times the amount of the true value, which also increased the loss value enormously. I understand that even with reasonable weights, biases have to be kept at a lower amount, than what I have chosen for this experiment, so that the model becomes more stable.

High weights + small biases



```
# weights and biases
w1 = 3
w2 = 1.5
w3 = 3
w4 = -2
w5 = 1.5
w6 = 3
bias1 = 0.5
bias2 = 0
bias3 = 0.5

# our training data
# x1 = input1, x2 = input2, y = true_value
input1 = 1
input2 = 0
true_value = 2

# learning rate
LR = 0.01
```

0.0s

Forward Pass

```
# NODE 1 OUTPUT
node_1_output = input1 * w1 + input2 * w3 + bias1
node_1_output = activation_ReLu(node_1_output)
node_1_output
```

0.0s

3.5

```
# NODE 2 OUTPUT
node_2_output = input1 * w2 + input2 * w4 + bias2
node_2_output = activation_ReLu(node_2_output)
node_2_output
```

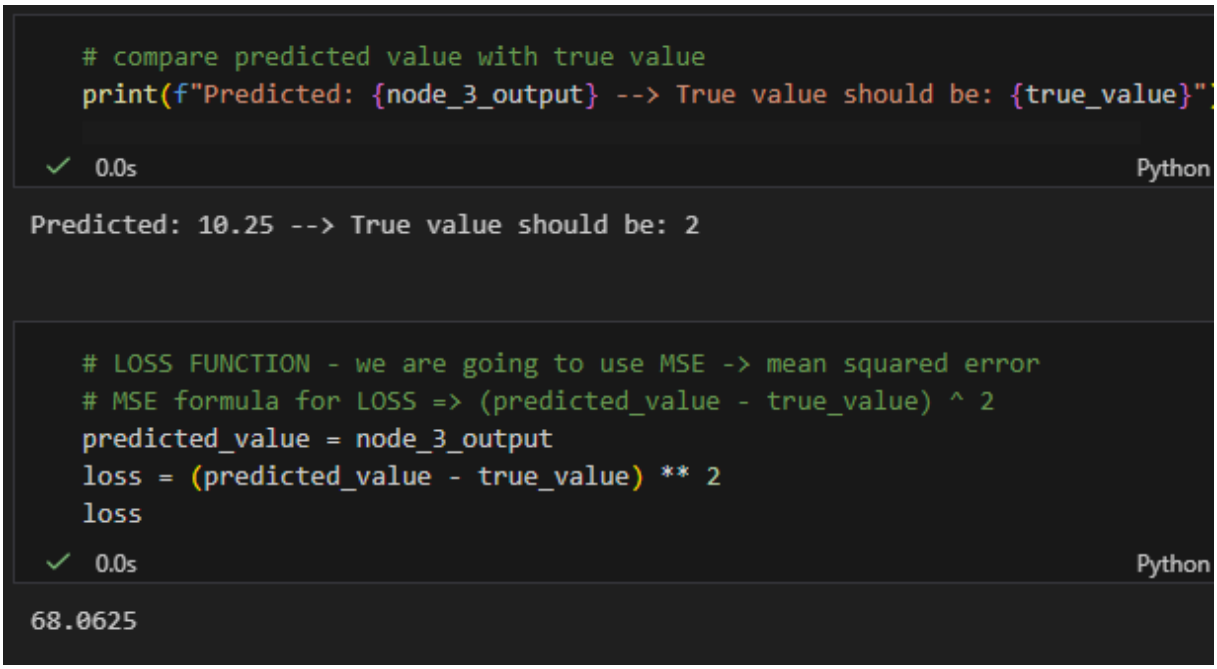
0.0s

1.5

```
# NODE 3 OUTPUT
node_3_output = node_1_output * w5 + node_2_output * w6 + bias3
node_3_output = activation_ReLu(node_3_output)
node_3_output
```

0.0s

10.25



```
# compare predicted value with true value
print(f"Predicted: {node_3_output} --> True value should be: {true_value}")
```

0.0s Python

```
Predicted: 10.25 --> True value should be: 2
```

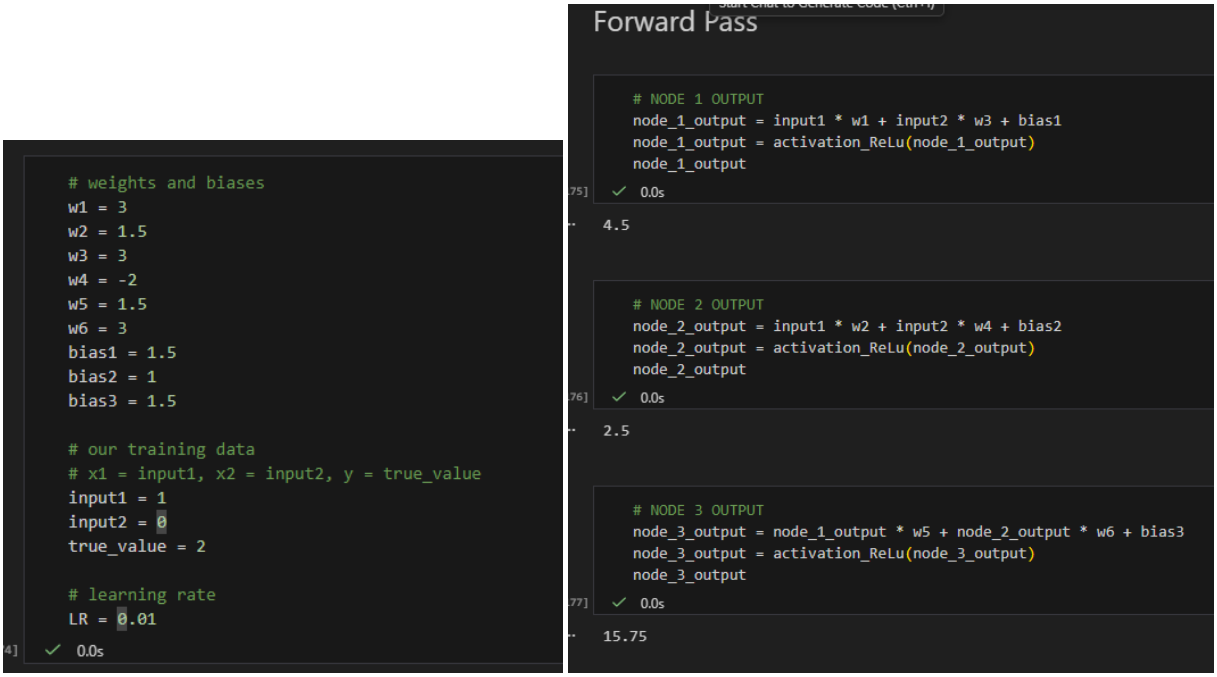
```
# LOSS FUNCTION - we are going to use MSE -> mean squared error
# MSE formula for LOSS => (predicted_value - true_value) ^ 2
predicted_value = node_3_output
loss = (predicted_value - true_value) ** 2
loss
```

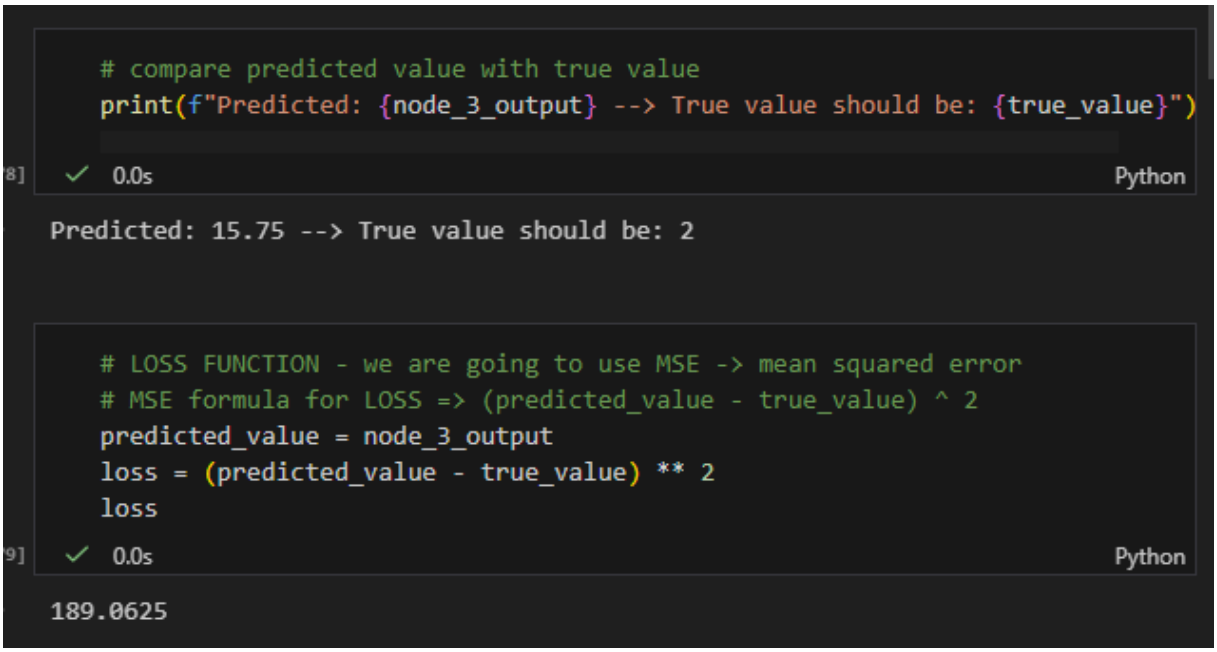
0.0s Python

```
68.0625
```

In this experiment, I used very high weights while I kept the biases small. This combination resulted in the worst results. The neural network predicted a value which is five times more than the true value and the loss value increased. I noticed the higher the deviation with the predicted value of the true value, the higher the loss value gets. For me, this test showed that with too high weights, the model becomes really unstable and produces very inaccurate predictions.

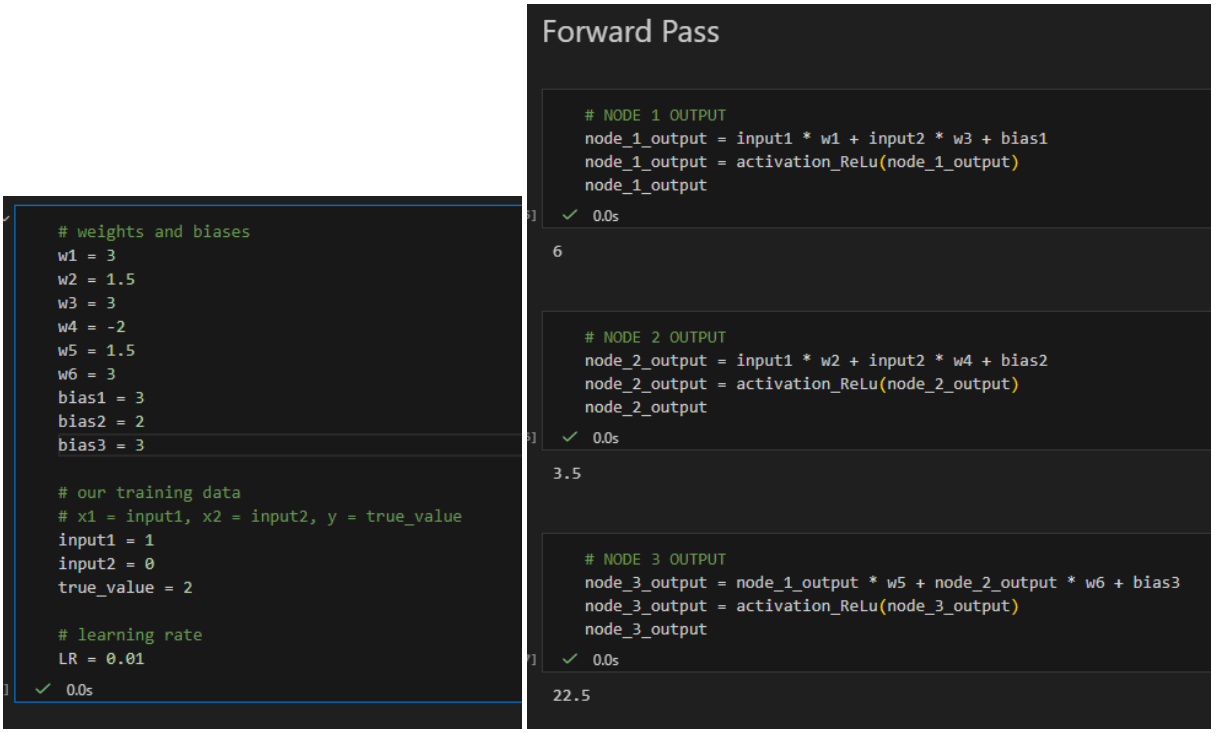
High weights + medium biases

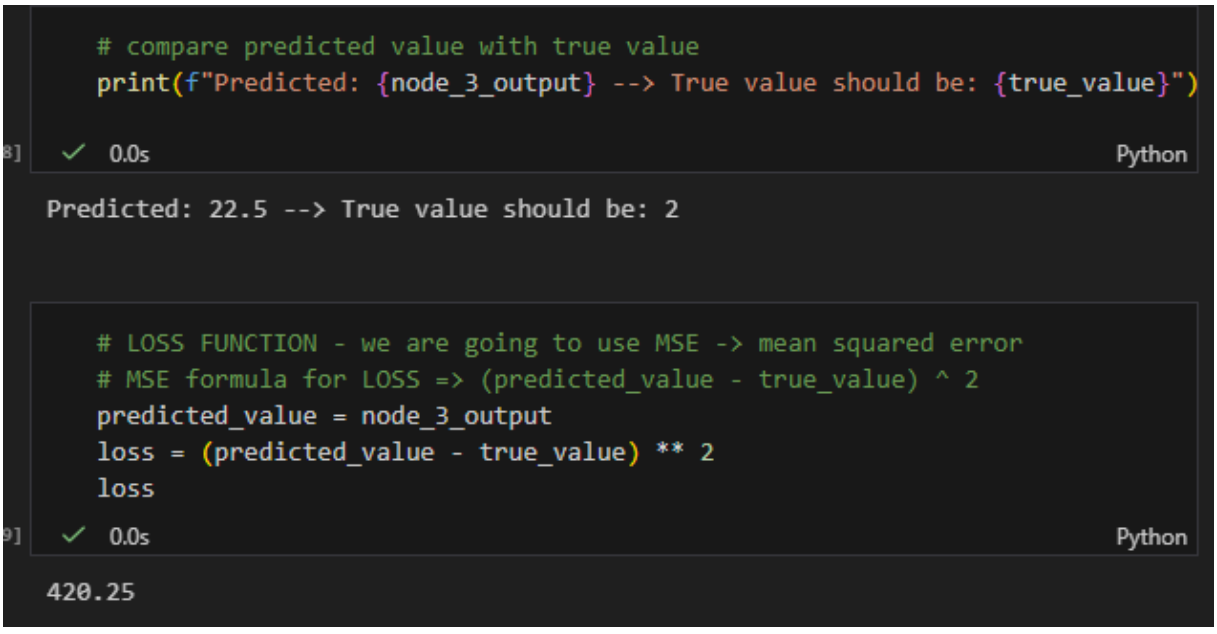




In this experiment, I combined very high weights with medium biases. This combination gave out the worst outputs, with a predicted value that is almost eight times higher than the true value and a very large loss value, this shows that once the weights are too high, even medium and I can already assume also high biases makes the model completely unstable and sensitive. This results in very unrealistic outputs in the forward pass phase.

High weights + high biases





In this experiment we used high weights and high outputs to see the final outputs. And I already knew that this would happen. If we are at too high weight values in general, no low nor high amount can help the model become stable. We had the biggest inaccurate prediction with a predicted value that is 11-times higher than the true value and an enormous loss value.

Results

For all nine experiments, I learned that the best performance came from the combination of medium weights + small biases. This combination produced a

prediction, which was the closest to the true value and also had the lowest loss value. So that is why I think, that medium weights give the network enough power to activate nodes properly, while small biases prevent the model from overfitting, meaning that the low biases keep the model from memorizing data so that they can give the most accurate predictions.

Any difference if some weights and biases have identical values, or if all of the weights and biases have unique starting values?

For this experiment, I wanted to see whether the neural network behaves differently when the weights and biases have either all identical values or unique values or what if weights and biases have some identical and some unique values. For the comparison, I have used the same values for the input1, input2 and true value variables. The goal is to observe how the starting structure of the weights and biases influences the forward pass phase as well as the loss value before training. I have done 3 experiments.

All weights and biases with identical values

```
# weights and biases
w1 = 0.5
w2 = 0.5
w3 = 0.5
w4 = 0.5
w5 = 0.5
w6 = 0.5
bias1 = 0.5
bias2 = 0.5
bias3 = 0.5

# our training data
# x1 = input1, x2 = input2, y = true_value
input1 = 1
input2 = 0
true_value = 2

# learning rate
LR = 0.01
```

0.0s

Forward Pass

```
# NODE 1 OUTPUT
node_1_output = input1 * w1 + input2 * w3 + bias1
node_1_output = activation_ReLu(node_1_output)
node_1_output
```

0.0s

1.0

```
# NODE 2 OUTPUT
node_2_output = input1 * w2 + input2 * w4 + bias2
node_2_output = activation_ReLu(node_2_output)
node_2_output
```

0.0s

1.0

```
# NODE 3 OUTPUT
node_3_output = node_1_output * w5 + node_2_output * w6 + bias3
node_3_output = activation_ReLu(node_3_output)
node_3_output
```

0.0s

1.5

```
    # compare predicted value with true value
    print(f"Predicted: {node_3_output} --> True value should be: {true_value}")
```

✓ 0.0s Python

```
Predicted: 1.5 --> True value should be: 2
```

```
    # LOSS FUNCTION - we are going to use MSE -> mean squared error
    # MSE formula for LOSS => (predicted_value - true_value) ^ 2
    predicted_value = node_3_output
    loss = (predicted_value - true_value) ** 2
    loss
```

✓ 0.0s Python

```
0.25
```

For this experiment, I have used identical values for weights and biases. With everything identical, both hidden nodes produced the same output (1.0) and the final prediction became 1.5, which is below the true value of 2 and the loss value ended up at 0.25.

The results show that identical values make the netowrk behave in a kind of symmetrical way, where both hidden layers learn the same thing. Even though the prediction was reasonable, we want the neural network to use each node independently, instead of all nodes going into one direction.

All weights and biases with unique values

```
    # weights and biases
    w1 = 1.5
    w2 = 0.5
    w3 = 1
    w4 = -1
    w5 = 1.2
    w6 = 0.7
    bias1 = 0.3
    bias2 = 0.4
    bias3 = 0.7

    # our training data
    # x1 = input1, x2 = input2, y = true_value
    input1 = 1
    input2 = 0
    true_value = 2

    # learning rate
    LR = 0.01
```

✓ 0.0s

Forward Pass

```
    # NODE 1 OUTPUT
    node_1_output = input1 * w1 + input2 * w3 + bias1
    node_1_output = activation_ReLu(node_1_output)
    node_1_output
```

✓ 0.0s

```
1.8
```

```
    # NODE 2 OUTPUT
    node_2_output = input1 * w2 + input2 * w4 + bias2
    node_2_output = activation_ReLu(node_2_output)
    node_2_output
```

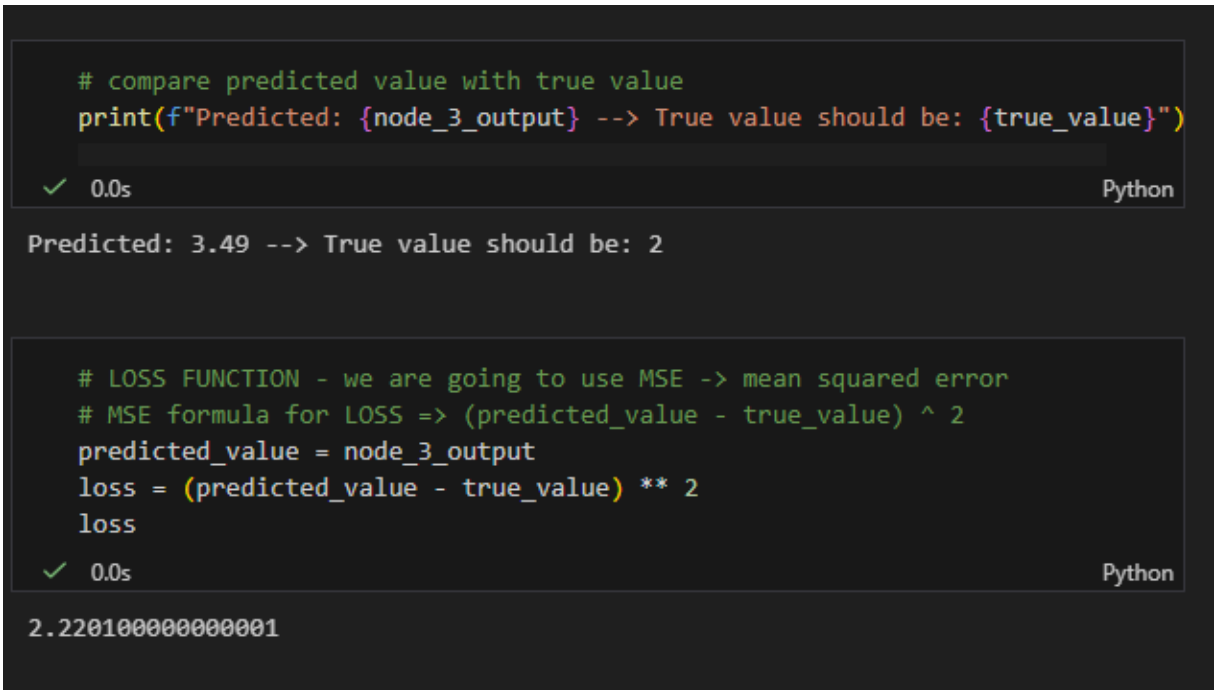
✓ 0.0s

```
0.9
```

```
    # NODE 3 OUTPUT
    node_3_output = node_1_output * w5 + node_2_output * w6 + bias3
    node_3_output = activation_ReLu(node_3_output)
    node_3_output
```

✓ 0.0s

```
3.49
```



In this experiment every weight and bias have a unique value, meaning different from everyone else, there is no weight or bias that has the same value as the others. The two hidden nodes produced different outputs (1.8 and 0.9). However, the final prediction became 3.49, which is much higher than the true value of 2 and also the loss value increased (2.22).

I think that using a unique value approach can be useful, if the values are chosen in the right amount for each weight and bias.

Weights and biases with some identical and some unique values

```
# weights and biases
w1 = 0.5
w2 = 0.5
w3 = 1
w4 = -1
w5 = 1
w6 = 1
bias1 = 0.5
bias2 = 0
bias3 = 0.5

# our training data
# x1 = input1, x2 = input2, y = true_value
input1 = 1
input2 = 0
true_value = 2

# learning rate
LR = 0.01
```

✓ 0.0s

Forward Pass

```
# NODE 1 OUTPUT
node_1_output = input1 * w1 + input2 * w3 + bias1
node_1_output = activation_ReLu(node_1_output)
node_1_output
```

✓ 0.0s

```
1.0
```

```
# NODE 2 OUTPUT
node_2_output = input1 * w2 + input2 * w4 + bias2
node_2_output = activation_ReLu(node_2_output)
node_2_output
```

✓ 0.0s

```
0.5
```

```
# NODE 3 OUTPUT
node_3_output = node_1_output * w5 + node_2_output * w6 + bias3
node_3_output = activation_ReLu(node_3_output)
node_3_output
```

✓ 0.0s

```
2.0
```

```
# compare predicted value with true value
print(f"Predicted: {node_3_output} --> True value should be: {true_value}")
```

✓ 0.0s Python

```
Predicted: 2.0 --> True value should be: 2
```

```
# LOSS FUNCTION - we are going to use MSE -> mean squared error
# MSE formula for LOSS => (predicted_value - true_value) ^ 2
predicted_value = node_3_output
loss = (predicted_value - true_value) ** 2
loss
```

✓ 0.0s Python

```
0.0
```

This experiment was the best one out of all of them, also from the previous experiment. The approach of using some identical values and some unique values for the weights and biases has shown that the neural model reached an accurate prediction of the same number as the true value of 2 and because the prediction was identical, we have a loss value of 0 which is perfect.

Results

Across all the three tests, I have noticed a lot of differences on how the network reacted to the weights and biases. With identical values for all the weights and biases, the neural model produced symmetrical predictions. With all the unique values, I feel like that the neural model became more flexible, but the prediction was not as accurate, whereas the best result came from the approach of mixing identical and unique values for weights as well as biases. We had an identical prediction of the true value and no loss value at all.

Adjust the learning rate to see any difference

In this final experiment, I wanted to see if there is an effect to the weights and biases if the learning rate changes. I have done 6 small experiments ranging with learning rates from 0.005 all the way to 0.25. I have kept the same input, weights and biases and I only changed the learning rate. The goal is to see how effectively the weights and biases are updated during backpropagation and also if the model learns smoothly or becomes more unstable.

Learning rate = 0.005

```
ORIGINAL WEIGHTS AND BIASES
w1: 1
w2: 0.5
w3: 1
w4: -1
w5: 1
w6: 1
b1: 0.5
b2: 0
b3: 0.5


##################################
NEW WEIGHTS AND BIASES
w1: 0.995
w2: 0.495
w3: 1.0
w4: -1.0
w5: 0.9925
w6: 0.9975
b1: 0.495
b2: -0.005
b3: 0.495
```

The Bias updates were very small in this one. Some weights and biases decreased from 1 to 0.995 by 0.005. In my opinion this shows that the model learns very slowly with a low learning rate. The updates actually are really close to the original weights and bias values. This shows that for te model to produce good predictions, it would need a lot of training steps.

Learning rate = 0.01

```
ORIGINAL WEIGHTS AND BIASES
w1: 1
w2: 0.5
w3: 1
w4: -1
w5: 1
w6: 1
b1: 0.5
b2: 0
b3: 0.5


##################################
NEW WEIGHTS AND BIASES
w1: 0.99
w2: 0.49
w3: 1.0
w4: -1.0
w5: 0.985
w6: 0.995
b1: 0.49
b2: -0.01
b3: 0.49
```

With a learning rate of 0.01, the changes of weights and biases became a bit stronger. We have for example w1, moving from 1 to 0.99 and b2 moving from 0 to -0.01. The model learns a bit faster and corrects its values more effectively. What I have noticed is that the weights w5 and w6 decrease a bit more than the weights w1 and w2, whereas w3 and w4 do not decrease at all.

Learning rate = 0.02

```
ORIGINAL WEIGHTS AND BIASES
w1: 1
w2: 0.5
w3: 1
w4: -1
w5: 1
w6: 1
b1: 0.5
b2: 0
b3: 0.5


##################################
NEW WEIGHTS AND BIASES
w1: 0.98
w2: 0.48
w3: 1.0
w4: -1.0
w5: 0.97
w6: 0.99
b1: 0.48
b2: -0.02
b3: 0.48
```

In this experiment we have used a higher learning rate of 0.02 and we see that the updated weights and biases decreased even more, with weights like w1 from 1 to 0.98 and w5 from 1 to 0.97. We can see by increasing the learning rate the difference between original and updated weights and biases gets bigger. An interesting thing I notice is that w1, w2, w5 and w6 decrease, but weights like w3 and w4 stay the same.

Learning rate = 0.05

```
ORIGINAL WEIGHTS AND BIASES
w1: 1
w2: 0.5
w3: 1
w4: -1
w5: 1
w6: 1
b1: 0.5
b2: 0
b3: 0.5


##################################
NEW WEIGHTS AND BIASES
w1: 0.95
w2: 0.45
w3: 1.0
w4: -1.0
w5: 0.925
w6: 0.975
b1: 0.45
b2: -0.05
b3: 0.45
```

With a learning rate of 0.05, the weight and bias updates became even stronger. We have w1 and w2 from 1 to 0.95 and 0.45 and bias like b1 and b3 from 0.5 to 0.45. Weights w3 and w4 did not change at all like usual. This makes me realize that this learning rate can speed up training a lot, but it can also increase the risk of the model becoming more unstable.

Learning rate = 0.10

```
ORIGINAL WEIGHTS AND BIASES
w1: 1
w2: 0.5
w3: 1
w4: -1
w5: 1
w6: 1
b1: 0.5
b2: 0
b3: 0.5


##################################
NEW WEIGHTS AND BIASES
w1: 0.9
w2: 0.4
w3: 1.0
w4: -1.0
w5: 0.85
w6: 0.95
b1: 0.4
b2: -0.1
b3: 0.4
```

With a learning rate of 0.10, the updated weights and biases became large and more like aggressive. This shows that the model is starting to take steps during learning. This model could work but the updated weights and biases are already close to being unstable. So, I think, when choosing the right learning rate, we have to be careful not to increase it too much.

Learning rate = 0.25

```
ORIGINAL WEIGHTS AND BIASES
w1: 1
w2: 0.5
w3: 1
w4: -1
w5: 1
w6: 1
b1: 0.5
b2: 0
b3: 0.5


##################################
NEW WEIGHTS AND BIASES
w1: 0.75
w2: 0.25
w3: 1.0
w4: -1.0
w5: 0.625
w6: 0.875
b1: 0.25
b2: -0.25
b3: 0.25
```

In the last experiment we had a large learning rate of 0.25, the updated weights and biases became extremely large. Weights like w1 dropped from 1 to 0.75, w5 from 1 to 0.625 and biases like b2 from 0 to -0.25. This shows that the model reacts quickly, but it became the most unstable also and it is much more difficult to train the neural model.

Interpretation

From the experiments and their results, it is clear that a higher learning rate in general does not automatically mean that the model learns better. Small learning rates made the updated weights and biases tiny, while very high learning rates caused the neural model to learn faster, but also take more risks of overshooting. That is why I think that the learning rate should not be neither to low nor too high, but at a moderate level. The best learning rate can be found out by experimenting with the neural network.

I also noticed that the updated weights w5 and w6 decreased more than w1 and w2 and I think that happens in general, because the weights w5 and w6 are connected to the output layer, so they receive the strongest gradients from the loss.

How can even a small change cause a drastic change in the performance of the neural model?

Based on my experiments, even very small changes in the starting values of the neural network or also the learning rate can lead to big differences in performance. Even the smallest adjustments to the weights and biases can change the nodes activity, which also changes the final prediction and therefore the loss value as well. This made me realize that neural networks are really sensitive and can drastically change the final output.

Is it possible to find the optimal starting values for weights/biases, LR and data to get loss close to 0? Is the loss meaningful based on the experiments?

```
ORIGINAL WEIGHTS AND BIASES
w1: 0.5
w2: 0.5
w3: 1
w4: -1
w5: 1
w6: 1
b1: 0.5
b2: 0
b3: 0.5


################################
NEW WEIGHTS AND BIASES
w1: 0.5
w2: 0.5
w3: 1.0
w4: -1.0
w5: 1.0
w6: 1.0
b1: 0.5
b2: 0.0
b3: 0.5
```

```
    # compare predicted value with true value
    print(f"Predicted: {node_3_output} --> True value should be: {true_value}")
```

✓ 0.0s Python

```
Predicted: 2.0 --> True value should be: 2
```

```
    # LOSS FUNCTION - we are going to use MSE -> mean squared error
    # MSE formula for LOSS => (predicted_value - true_value) ^ 2
    predicted_value = node_3_output
    loss = (predicted_value - true_value) ** 2
    loss
```

✓ 0.0s Python

```
0.0
```

According to my experiments and the results above to see, I think it is possible to find the optimal values for weights and biases as well as the learning rate, by experimenting with the values. In the pictures above, we can see that I found the optimal values for the weights and biases so that my prediction was identical to the true value of 2 and we can also see that the loss value is at 0, which is the perfect, meaning that the model has nothing left to correct. Also, we can see that the updated weights and biases are identical to the original values. And one interesting thing is, that once we find the optimal values for the weights and biases and use them with different kind of learning rate (low or high), it does not change the output at all, because the updated values of the weights and biases still stays the same.