Tom Cordruwisch
R504D80 Deep Learning
23.03.2024

Exercise project X - Experimentation with neural network mathematics

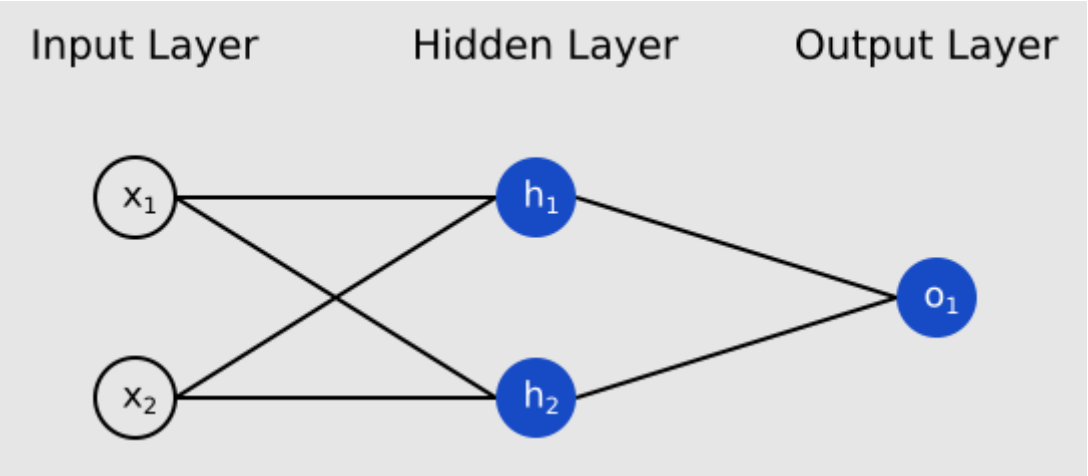
To further my understanding of the inner workings of neural networks I experimented with a simple example in a jupyter notebook that did not use any external libraries for the neural network itself.
Numpy was used for the generation of dummy data and matplotlib for visualising the training process.
The structure was as following:

- 1 hidden layer with 2 neurons (or nodes)
- 1 output layer
- 2 numeric inputs to predict one numeric target

The function to generate test data was a simple quadratic equation in the form of:

$$x_1^2+x_2=y$$

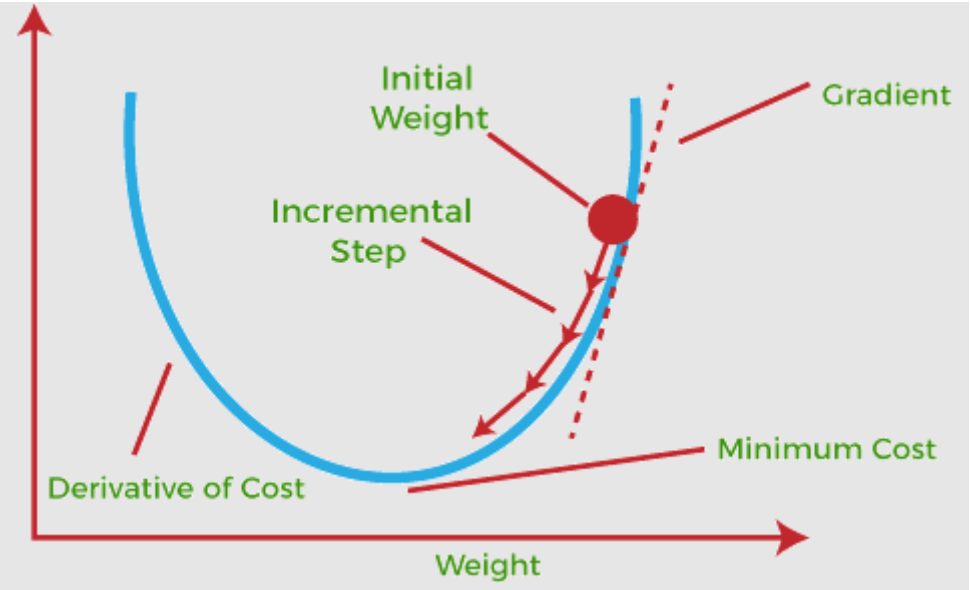
Where x_1 and x_2 are the inputs to the neural network aiming to predict y.
A bit of noise was added to make the problem less trivial.
Apart from that two functions were defined, the ReLu activation function which only forwards a value if it's positive and otherwise returns 0, and its partial derivative for the backpropagation step, where it returns 1 for an positive input and 0 in all other cases.
The network with one hidden layer containing two neurons and one output layer with one neuron necessitated six weights and 3 biases.



As seen in the illustration, the neurons are connected with each other, which is where the weights are applied to strengthen/weaken the connection in order to fit to the problem.
Because there are six connections, there are also six weights. The biases on the other hand are applied at each neuron, irrespective of the connection, and thus there are three.
Further things defined in the code are a few hyperparameters, learning rate and number of epochs, but that's as far as the requirements go for such a simple neural network.
When training the network, the first preparatory step is the initialisation of weights and biases (set to some random starting values), the manually set hyperparameters, and generation and scaling of the dataset.
Afterwards the training starts and is done in epochs (basically rounds of training) where the same code is run for each epoch.
In a single epoch, the input data is passed to the hidden layer nodes, multiplied by the respective weight of the connection and with the bias added to it. The result of this mathematical operation is passed through the activation function to decide if it should be passed on or if the neuron instead returns 0.
The outputs of the two neurons are then passed to the output node in just the same way, and the output of the final node is what the neural network predicts for the target value y.
Of course the prediction from the first epoch is likely very far off from the true value, as all the weights and biases were randomly chosen. In order to improve, the difference between the prediction and the true value, the loss, is recorded and serves as the base for adjusting the weights and biases. To do that, there is the so-called "backpropagation" where the loss is used alongside the learning rate to tune both the weights and biases starting from the last layer and working its way backwards to the beginning. This is where derivations are necessary because the outputs of the later nodes are connected to and dependent on the previous nodes. Thus, when adjusting the weights and biases for the early layers, the impact on the subsequent layers has to be accounted for.
Finally, the newly calculated weights and biases are updated and the next epoch starts.

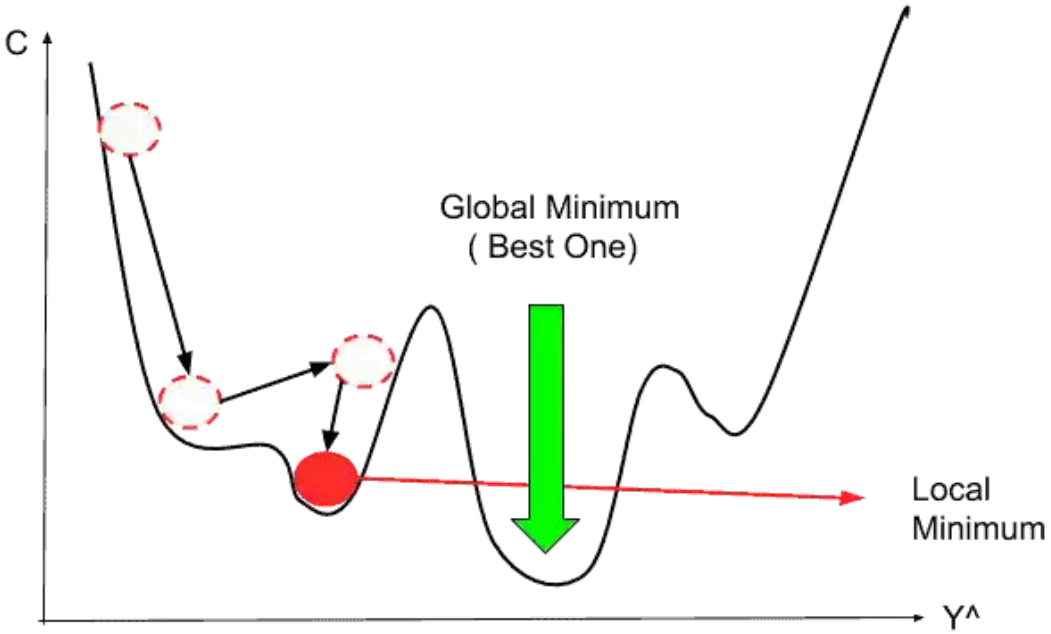
In principle, the neural network improves by changing (adjusting) the node connections in a way that minimises its error.
The activation function allows neurons to selectively activate and effectively become a dead-end which ignores patterns in the data that are irrelevant to the problem.
The weights and biases determine how important or impactful a pattern is for the prediction.

To give an explanation on loss, learning rate and gradient descent, I found a helpful image:



Assuming that the ideal value for a weight is the one where the error of the prediction, the loss, is the lowest, the gradient descent constitutes the mechanism for how the weights should be changed in order to arrive at the optimum values. Each epoch the weights are updated, the amount of which is scaled via the learning rate. Because of this a higher learning rate can cause the values to be updated too much and overshoot the optimal value.

However, there are a lot of cases where a minimum is not actually the best value, but instead a local minimum. This is why it can be good to have a higher learning rate and “overshoot” the local minima, as the model might not be able to find the global minimum with a low learning rate.



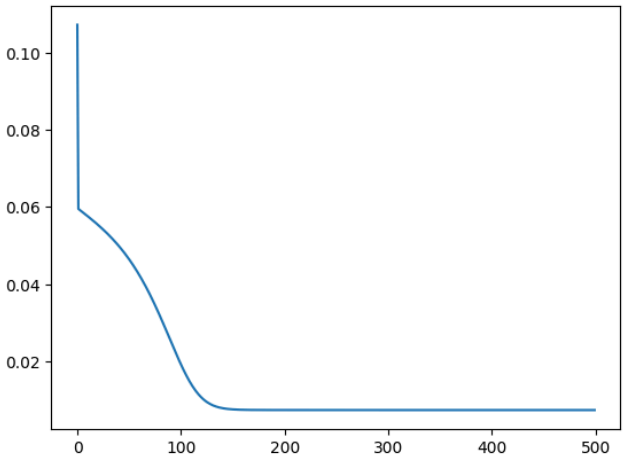
A common addition in machine learning is “reduce learning rate on plateau” which slows down the learning rate as it approaches the ideal value, allowing to both set a higher learning rate for more easily getting to the global minimum and then slowing down to find the precise bottom of it.

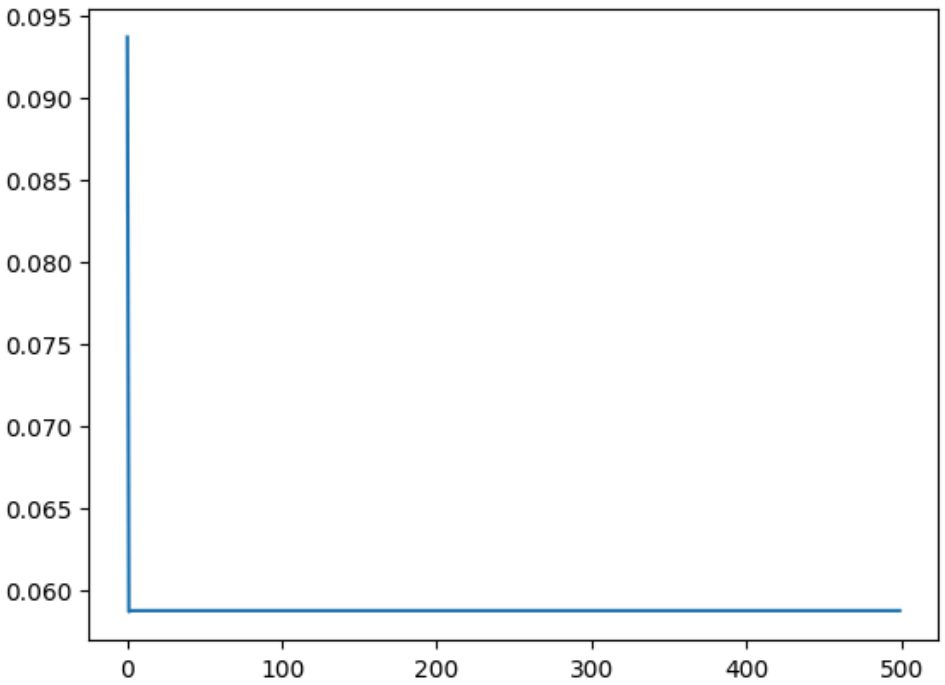
Moving on to the experimentation with the neural network, I first tested different learning rates and my understanding is that a high learning rate causes more erratic behaviour, the model might be able to approach a good solution more quickly - but only if it’s able to find it, which gets less likely the higher the learning rate is.
A lower learning rate on the other hand causes the model to learn more consistently, but it will approach the solution more slowly.
Also it might be more likely to find solutions that are easier to get to from the starting state, but not necessarily the best solution overall, similar to how there can be local and global maxima in a mathematical function.

Tom Cordruwisch
R504D80 Deep Learning

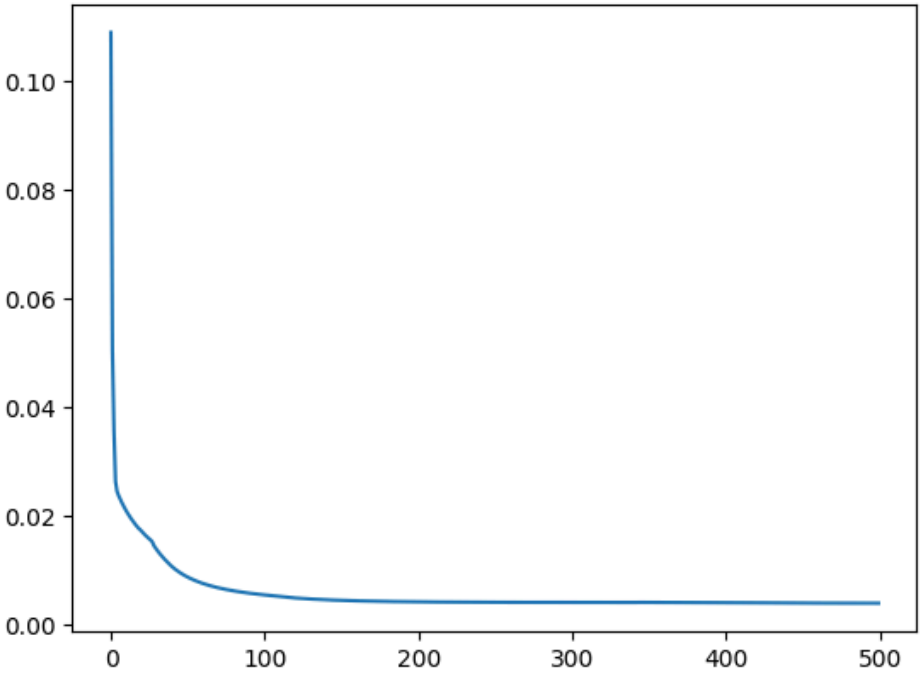
23.03.2024

Changing the starting weights and biases slightly doesn't have too much of an effect, after all the point is that the neural network repeatedly changes them to adapt and fit to the problem. There is some variation with how quickly the training progresses in the early stage, but at least for the weights even setting them to ridiculous values (in the millions) didn't stop the model from learning well.
When setting the biases very high, the loss-curve becomes very abrupt, which I think is because there is insanely high loss at the beginning, dwarfing the remaining losses in the training.
I did see more of a difference when actually changing weights and biases to very similar values, this was the outcome with all weights = 0 and biases = 2:
When setting everything to 0, the learning went really quickly:





And on the other hand, setting weights = 2 and biases = 0, the learning curve looked more like a normal (gradual) learning curve for neural networks:



This was very interesting, based on this it seems like the initial biases in particular can have a surprisingly high impact on the training.
That might be down to the nature of them being additive as opposed to multiplicative as the weights are.

Something else I noticed when inspecting weights and biases after training, is that more often than not the weights and biases will be quite close to the original values. Usually only some of them get changed considerably, so I think that the way it works is that (based on the learning rate) the model tries to find improvements that are close to the starting weights/biases and modifies mostly those that have a bigger impact for reducing loss.

I made the testing a bit easier by randomising the values like this:

The weights are values between -10 and 10 and the biases between -5 and 5.

```python
# Initialising weights and biases
# Randomizing weights and biases
# Weights: floats between -10 and 10 (2 decimals)
# Biases: floats between -5 and 5 (2 decimals)
weights = []
biases = []

for i in range(6):
    weight = round(random.uniform(-10,10), 2)
    weights.append(weight)

w1 = weights[0]
w2 = weights[1]
w3 = weights[2]
w4 = weights[3]
w5 = weights[4]
w6 = weights[5]

# Biases
for i in range(3):
    bias = round(random.uniform(-5,5), 2)
    biases.append(bias)

bias1 = biases[0]
bias2 = biases[1]
bias3 = biases[2]
```

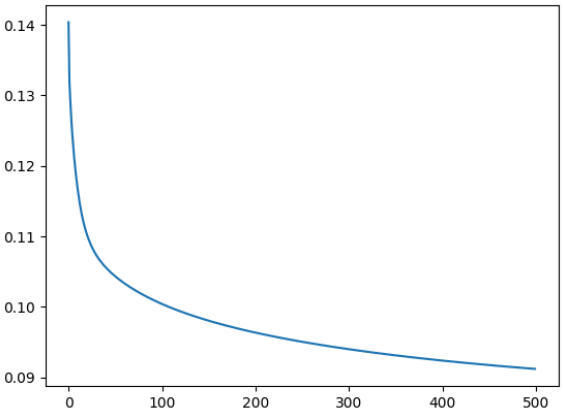
The example output below shows how they change from before to after the training.

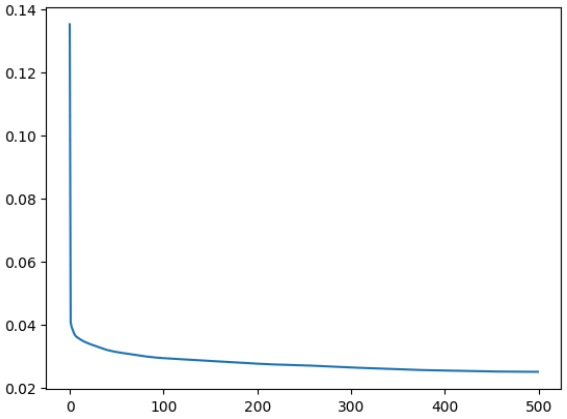
```
------------------
Original weights/biases
W1: -0.46
W2: -6.59
W3: 8.47
W4: -9.91
W5: -8.92
W6: 2.06
B1: -4.44
B2: 4.01
B3: -2.5
------------------
Final weights/biases
W1: -0.46
W2: -8.01584467645139
W3: 8.47
W4: -7.860690565992684
W5: -8.92
W6: -0.3476833457527351
B1: -4.44
B2: 4.9182789818978145
B3: 1.50080667340648
```

To me it stands out that W1, W3, W5 and B1 did not really change in this case. W6 changed the most, from 2.06 to -0.35, so it likely was the biggest factor for lowering the loss.

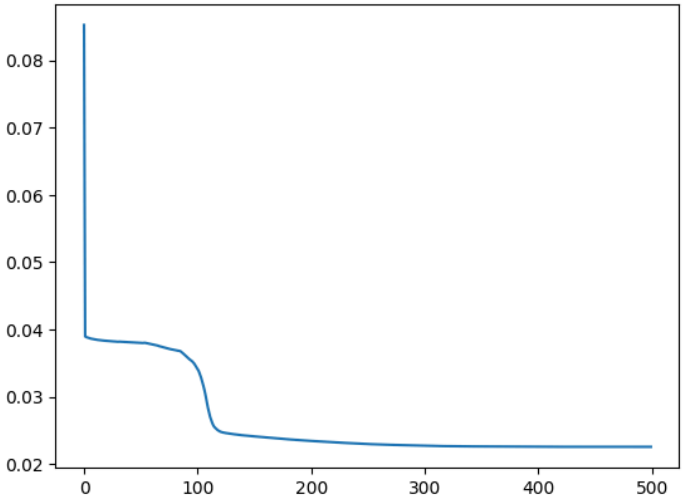
Which specific weights and biases changed and stayed unchanged was dependent on the training run, there didn't seem to be a clear pattern.
For the loss, the final loss was usually around 0.05 with this data.
Data was generated from a function $f(x1, x2) = x1^2 + x2 + b$
where b is a random integer to add a bit of noise to the generated data.

When increasing the range of possible inputs (e.g. x1 from -50 to 50 and x2 from -100 to 100), the model would more often than not take more epochs to fit (image on the left below), compared to the original value ranges (x1 from -5 to 5 and x2 from -10 to 10) with training visualisation shown below on the right.





Very interesting was a curve I got when giving 3000 input data points for training (instead of 300):



A big caveat is that a lot of these results are not always repeatable and I realise that one of the sometimes frustrating, but unavoidable limitations for neural networks is the randomness that affects not only the input data, but also the way a neural network trains.

There are tools that can help, such as setting a random seed, but even then that mostly helps with reproducibility and does not change the fact there typically is a range of different possible outcomes for the same training process, with no way of knowing how large that range is without trying out different random seeds.
However, I think the better designed and the larger the neural network is, the lower the discrepancy between possible outcomes for trained models gets. In other words, I doubt that in most real use cases, someone could train a model 10 times and get an accuracy of ~90% each time, but then training it for an 11th time would result in a model with 99% accuracy.

In general, I think building a neural network from scratch was really valuable to understand how it works on a lower level, and what aspects of it are simple or challenging to create.
One of the benefits of studying the low-level architecture is to see what mathematical operations are done and how often they occur, and this understanding is especially valuable for people who design hardware for machine learning and for researchers looking for new and more efficient machine learning algorithms and methods.
The two aspects of efficiency and effectiveness are vital for advancements in the field. I personally really enjoy the process of learning to understand challenging concepts, and experimentation is the best way to achieve that.
Apart from being good for learning purposes and for designing new types of neural networks, algorithms or custom layer types, I see little reason to code them from scratch like this every time. Especially the backpropagation makes it difficult to read and the number of lines and defined weights and biases will quickly balloon to huge amounts of code that are hard to keep track of. There is a reason why programmers define functions and create libraries for these kinds of repetitive structures, which is why TensorFlow, Keras, etc. are so commonly used and it would be an enormous waste of time to completely avoid them in production. Depending on the requirements of a project it can make sense to avoid Keras to allow for more customisation, but in many cases it is unnecessary to go more low-level than TensorFlow.