Manual Neural Network

After getting to know the process of a neural network and its learning process, it was interesting to think of the possible different things you could try out.

Contents

Basics

Step 1

My first try was to use the basic neural network given in through the lecture notes, use the new weights and biases and compare the loss values. From the new loss value, I could gather, that while it has improved, it only improved from 2.5 down to 2.4254 (true value being 2).

Step 2

So, I went back and increased the learning rate to see how much this impacts the loss improvement. It now jumped from 2.4254 to 2.12, meaning that the learning rate significantly closes the gaps between the predicted number and the true value.

Step 3

To test this even further, I have increased the learning rate once more. This time, however, it predicted 1.96, undershooting the true value. Despite us starting off at a predicted value of 2.5, there is a very real possibility for the neural network to undershoot the true value if it makes sudden adjustments due to a high learning rate.

This shows that it is very important to set the right learning rate for the neural network, because too small values lead to very slow results, meaning that there could be a lot of epochs needed for it to find the true value. But setting the learning rate too high could cause the predictions to overshoot the true value very fast which could lead to fluctuations for the loss.

Increased Node Count

Step 1

Next, I tried to increase the Node count by one, setting the second layer to consist of Node 1,2 and 3 and the third layer to have Node 4. By adding one more Node to the second layer, the neural network had gained three new weight values as well as one new bias value.

On top of that, the number of calculations had to be increased. After having a further look into this, I realized that simply increasing the node count by one, may severely complicate and impact the network.

Looking at the new prediction we can see that it undershot this time, being at 1.75 instead of 2.5 like the 3-node basic network. Ignoring the new weights and biases I had to set randomly; this proves as a sudden change in the results.

Step 2

Again, I set the newly calculated weights and biases and got a value of 1.7875. Comparing this to the old network's improvement, we can see, that it increased by around 0.04, while the other increased by around 0.07.

Step 3

Since we have adjusted the learning rate again, it now jumped to a value of 1.95 (+ ~1.6), which is also a significantly smaller jump than the old network.

Step 4

Interestingly enough, after further increasing the learning rate, the new prediction was 2.009, showing that despite having smaller jumps, it seemed to prove more accurate with the additional node. It's interesting that the last jump was only 0.05, even though I doubled the learning rate since the last prediction increase.

More Nodes could therefore lead to more complex networks but since there exist more variables for adjustments (more weights and biases) it could benefit from larger learning rates.

Different activation functions

I used the previous two files to also create new neural networks with other activation functions.

Leaky ReLU

First, I tried leaky ReLU which changed nothing, all the values were the same. I researched a bit about the activation function and found out that it’s basically just an improved ReLU function.

Leaky ReLU only shows an increase in output when negative values are used, since despite normal ReLU, does not set them as 0.

Sigmoid & Tanh

After trying both, I found it weird how it improved very slowly, despite the learning rate increase. Also, the predicted value never went above 1. After looking further into the two functions, I found out, that they are basically only able to output probabilities, so only values from 0 to 1.

Overall interesting notes

After looking at all the different outputs of these run throughs, one weird thing I noticed is that the networks are pretty biased when it comes to increasing the weight values.

It always adjusted all the values, except weight 3 and 4 for 3-node networks and 3,4,5 and six for 4-node networks respectively. These values are never adjusted during learning.

Advanced Testing

Setting the same values for weights/biases

Setting all values to 1, did not have that much impact. Even though the initial prediction was 5 instead of 2.5, the high jumps in prediction values showed that this has not a high impact on the overall learning of the machine.

Setting all values to 0 however, slowed the learning process. It only adjusted bias3 and will probably do that for a lot of epochs until it can adjust another value.

Switching data

I then switched all the pairs with each other (w1/w2; w3/w4; b1/b2; w5/w6), it changed nothing.

I then switched the values of w1/w2 with w3/w4 and it predicted 2. Since it already predicted the true value exactly, it stopped adjusting in the other epochs.

Change data

Since the machine prefers to adjust the last bias first, I increased the value of bias3 by 0.5. This increased the initial prediction by 0.5 as well.

I then tried to increase all the initial values by 0.01 as well as decrease them by 0.01. As expected, increasing the weights and biases also increases the prediction. It is, however, interesting, that it the prediction increased from the starting point of 2.5 to 2.5704, while it decreased to 2.4304. It increased by 0.0704 but decreased by 0.0696. Meaning it increases faster than decreases.

Changing input/output data

Lastly, I tried changing the inputs and outputs according to the original formula without changing the model. First set of data (*nn_ReLU_basic.ipynb)* has a loss of 0.25. The second set of data (*nn_ReLU_diffData.ipynb*) has a loss of 0. The third set of data (*nn_ReLU_diffData2.ipynb*) has a loss of 1.0.

Even though they all follow the same logic and have the same model, they give different loss values.

Random Number Generator

While creating the random number generator, I found it to be most effective to use random numbers in the ranges of -1 to 1, since these values are good for our small network.

Conclusions

From testing manual neural networks, I think I have an idea how these models work.

First, the network is given inputs and a target value. Based on the network created, it then sets random values for weights and biases.

The first epoch starts. This starts the forward pass, in which the model uses the current (randomly set) weights and biases to calculate a prediction. This will most likely never be true, since the values are random at the beginning. It will then use the predicted value and the target value to calculate the loss. Loss basically tells the model if it should increase or decrease the values for weights and biases.

Based on this information and the learning rate set for the model, adjustments are calculated during backpropagation. In this step, the model walks backwards through the network, solving the partial derivative of the old weights and biases to calculate new values. In our case, this optimizer is called gradient descent.

To access all the layers below the last one, the chain rule is needed to calculate the new values.

After the backpropagation is completed, the new values for weights and biases have been calculated and can now be used to run a new epoch.

If the model is trained on more than one set of data, it will do the forward pass for each set of data, then average the loss values of all of them and use that to do the backpropagation. This way, it creates a generalized picture.

The goal is to have a low loss value to create accurate predictions. After a good model with nice loss values has been created, it can be transformed into a mathematical formula to calculate predictions. If the model is finished, weights and biases will remain and be used to calculate the forward pass. The prediction, that was calculated using the forward pass, should then be the accurate prediction of the finished model.

If we look at the model in the file *nn_ReLU_switcherooTwo.ipynb,* where I accidentally had the correct value on the first try, we can see **one** possible correct solution that could come out of a neural network:

(1 * 1 + 0 * 1 + 0.5) * 1 + (1 * -0.5 + 0 * 0.5 + 0) * 1 + 0.5

This is the forward pass with the weights and biases of the 0-loss model with respect to the inputs.

= 1^2 + 0 + 1

This is the actual formula with respect to the inputs.

= 2

This is the output.

This, however, does not mean it understood the original formula. If I were to use this model to calculate the prediction for the third set of data (*nn_ReLU_diffData2.ipynb*), it will now have a loss of 4. This is called overfitting, since the model now understands the first set of data very well, but not the underlying relationship (this model was creating using only the first set of data, so no wonder).

This means that a lot of data is needed to create a generalized picture of the given data.