Backpropagation Coursework Report

# Data Pre-processing

To pre-process my data I initially copied all the data into a text file and constructed a python script to determine outliers and missing data, and then convert the raw data into usable lists of floats. Once I had determined which data was missing, I interpolated to replace missing data and outliers which I deemed to be excessive within excel.

I have plotted the raw data into figure 1 below:

Chart

Description automatically generated

Figure

After replacing missing data and interpolating outliers (Fig.2) which I deemed to be excessive, the dataset became much more usable.

Chart, line chart, histogram

Description automatically generated

Figure

I then standardised the data within the range 0.1-0.9 to ensure that data which naturally has a high value doesn’t mask the other data in the set.

Chart, bar chart, histogram

Description automatically generated

Figure

Finally, I split my data into training and test sets with an 80:20 split. The training data is used to train the model, and the test data is used to test the effectiveness of my implementation.

The script I created to handle data processing is below:







# Implementation



# Training and Network Selection

## Forward Pass

I started by initialising nested lists containing weights and biases for each neuron in the hidden layer and the output layer. I then worked through the data set, calculating weighted sums for each neuron in the hidden layer by iterating through the nested list of weights, applying each weight associated to the hidden layer to each input, and adding the bias for that hidden neuron to the resulting value. I stored all these values in an array called “weightedSums”. I calculated the activations for these values by applying the sigmoid function to each and stored these in an array called “activations”. I calculated the weighted sum of the output layer by multiplying the activations for each neuron in the hidden layer by the weight applied to that neuron, then applying the bias for the output layer. I calculated the activation for the output layer by passing this value through the sigmoid function.

## Backward Pass

I started by calculating the delta value for the output layer, multiplying the error between the expected output and the output activation by the value given by applying the derivative of the sigmoid function by the output activation. I then calculated the delta values for the hidden layer, iterating through each neuron in the hidden layer, calculating the delta values by multiplying the delta for the hidden layer with the weight of the neuron and the value given by applying the sigmoid derivative function to the activation for the neuron, appending these to a hiddenDelta array.

## Updating Weights and Biases

I updated the weights from hidden to output layer by multiplying the learning rate with the delta of the output layer and the activation of the output layer and adding this value to the old weight value. I then updated the biases for the hidden layer by multiplying the deltas for the hidden layer to the learning rate and adding this to the existing biases. Then I iterate through the weights applied to the input layer, using the same method as before to update them. Finally, I update the output bias using the learning rate and delta of the output layer.

## Network Selection

To avoid local minima, I train my network multiple times. After each training cycle, I test that set of weights and biases’ effectiveness by finding the mean error when that model is applied to the test set. I store the weights and biases which resulted in the lowest mean error upon testing, and that is the network which I select to use after training.

## Other information

My code is fully customisable, with the learning parameters, number of inputs, hidden and output nodes able to be changed and customised to suit any dataset. My standardisation and data splitting algorithms will also work with any dataset. I used python to implement my MLP with numpy, matplotlib, random and math imported. I opted not to use an OOP approach, instead using arrays, nested arrays and splitting my code into functions. Each time my trainModel function is run is an epoch, so to train the algorithm this needs to be run as many times as there are epochs that it’s training for, which is set in the “if \_\_name\_\_ == “\_\_main\_\_:” in the “epochs” variable.

# Evaluation

## Base Model

To test my implementation, I used my trained model to predict the predictand for each piece of test data and recorded the mean error of these predictions. The first test which I did was to train my model with 1, 10, 100, 1000 and 10000 epochs. For each number of epochs I trained the algorithm 5 times and selected the best performing set of weights and biases, based on the mean error produced when predicting the test data. The results of this are in the table below:

|  |  |
| --- | --- |
| Number of Epochs | Mean Error |
| 1 | 0.10990237210214156 |
| 10 | 0.1074126243224174 |
| 100 | 0.01621058616022838 |
| 1000 | 0.012702017361175332 |
| 10000 | 0.02018308558401532 |

I repeated this test using one set of weights and biases for each of the epoch cycles to show how the accuracy of the algorithm gradually improves with each increase in the number of epochs:

|  |  |
| --- | --- |
| Number of Epochs | Mean Error |
| 1 | 0.12514793363165122 |
| 10 | 0.11998875209236642 |
| 100 | 0.018120941015058973 |
| 1000 | 0.01261083700607197 |
| 10000 | 0.02427507336911702 |

## Momentum

I implemented the “momentum” improvement method by storing the pastWeightChange from previous changes to the weights and biases and adding 0.9 \* pastWeightChange to the weights when changing them. I then tested the effectiveness of my algorithm as shown below:

This test incorporated the “Momentum” method of improvement, with random initial weights and biases for each training sequence:

|  |  |
| --- | --- |
| Number of Epochs | Mean Error |
| 1 | 0.13269743081448637 |
| 10 | 0.12248877243723441 |
| 100 | 0.014380047174487987 |
| 1000 | 0.012771655787776514 |
| 10000 | 0.021758688292582498 |

This test showcases overtraining of my model as training for 10,000 epochs resulted in the model being overtrained and less accurate than training for both 100 and 1000 epochs.

I then repeated this without resetting the initial weights and biases between each training loop, showcasing the ability of the algorithm to improve with time.

|  |  |
| --- | --- |
| Number of Epochs | Mean Error |
| 1 | 0.13535810519208208 |
| 10 | 0.12194116034515536 |
| 100 | 0.01535510325338471 |
| 1000 | 0.012684681790307035 |
| 10000 | 0.02326283177085362 |

## Base vs Momentum

This graph shows the impact on the error function of the number of epochs which the algorithm is trained for of the base algorithm without any improvements made.

Chart

Description automatically generated with low confidence

This graph shows the impact on the error function of the number of epochs which the algorithm is trained for, with **Momentum** implemented into my algorithm. You can clearly see the difference in training speed when using momentum.

Chart

Description automatically generated with low confidence

This graph shows the above, but with **Annealing** implemented upon the base algorithm. I decided to start with an initial learning rate of 0.2, which had a great effect on the error, which is also slightly reduced compared to results which did not use annealing as it allowed the model to find a slightly improved set of weights and biases.

Chart

Description automatically generated

This graph shows both **annealing** and **momentum** implemented upon the base algorithm:  
Chart

Description automatically generated with medium confidence

From these tests, I found that implementing momentum had a more drastic effect on the speed of training MLP’s than annealing did. This is due to the increased size of steps along the error function where the gradient is steeper, allowing the algorithm to traverse to minima faster than if momentum wasn’t implemented.

Annealing had a less dramatic impact on the speed of training my perceptron, although it did train faster with annealing implemented. Where annealing had an impact, however, was in improving the quality of the network which was produced after training. Both tests which incorporated annealing had less error after 2000 epochs than the tests which didn’t have annealing implemented, which can be attributed to annealing allowing the model to avoid local minima with a greater learning rate step, and then reducing the learning rate over time to home in on a minimum which is more likely to be global.

The best results were seen in the case which incorporated both momentum and annealing into the algorithm. The MLP trained at a rapid rate, and found an average error which was lower than the base algorithm.

# Comparison

To get a benchmark for the performance of my algorithm, I have chosen to compare it to a simple multiple linear regression model, LINEST, in excel.