Project 2
Multi Layer Perceptron

CS457 - Computational Intelligence
Bryce Harmsen - Eric Engman - Luis Torres - Nisser Aldossary

Introduction

Project 2 is based around the Multi Layer Perceptron. The perceptron is implemented thrice: all through the WEKA interface, through TensorFlow, and through a mixture of WEKA's Java API and Python code written by our team. For the reader who is purely seeking our answers to CS 457 Project 2, only the Analysis section should be read. Though for the reader who wishes to see a more complete picture of our process, we have provided a basis and background for our empirical analysis.

Implementation

WEKA

The multilayer perceptron was first configured through the WEKA GUI, then implemented through a Java application. The benefit of first configuring through the GUI is that all options available for the multilayer perceptron are easily viewable. This allowed for a more informed, full utilization of the WEKA API in the Java application. While the WEKA GUI has the benefit of exploration and education of the WEKA capabilities, the Java implementation allows for faster tuning. The Java application builds random network structures, with either one or two hidden layers and up to nineteen neurons per layer. The Java application also allows for tuning of other factors such as number of epochs, the learning rate, and momentum, but by the time research was shifted from the GUI to the Java application, the main tuning that remained was the network configuration.

TensorFlow

The TensorFlow (TF) implementation was built in Python using the Keras library. The network used sigmoid activation functions within the hidden layers and a linear activation function for the output. The network architecture chosen was TF's sequential model, which creates a linear stack of layers, as stated in the TF documentation. While this library allows for succinct code, there is still enough flexibility to allow hyperparameter tuning and to choose accuracy reporting that makes sense for various discrete and continuous solution spaces.

Results

WEKA

The WEKA model allows the user to tune the learning rate, momentum, number of epochs, number of hidden layers, and number of neurons in each hidden layer. Other hyperparameters are available, like number of folds, but for this experiment the validation step was excluded. Instead, all data was split between training and testing phases.

WEKA GUI

Below are the test results using the WEKA GUI. We did six configurations in the single hidden layer architecture, and nine configurations in the two hidden layer architecture. The testing was done using a 50% split, with 500 total instances. From these results, we found using one hidden layer to be optimal with four or more neurons.

1 Hidden Layer	**2 Neurons**	**4 Neurons**	**6 Neurons**	**8 Neurons**	**12 Neurons**	**14 Neurons**
Correlation coefficient	0.8006	0.8006	0.8154	0.8156	0.8167	0.8158
Mean absolute error	0.7431	0.7459	0.6996	0.6969	0.6962	0.6971
Root mean squared error	0.9255	0.9298	0.8547	0.849	0.848	0.8492
Relative absolute error	73.8679 %	74.1505 %	69.5479 %	69.2774 %	69.2118 %	69.2936 %
Root relative squared error	72.8899 %	73.2319 %	67.3193 %	66.8709 %	66.7857 %	66.886 %
Number of instances	250	250	250	250	250	250

2 Hidden Layers	1x1	1x5	2x2	2x4	4x4	4x8	8x4	15x15	18x18
Correlation coefficient	0.7978	0.8007	0.8028	0.8049	0.8032	0.8042	0.8061	0.8068	0.8056
Mean absolute error	0.7401	0.7264	0.7386	0.732	0.7291	0.7245	0.7269	0.7241	0.7239
Root mean squared error	0.9121	0.8901	0.9211	0.9114	0.9028	0.894	0.9014	0.8965	0.8956
Relative absolute error	73.5716%	72.2068%	73.42%	72.7682%	72.4834%	72.024%	72.264%	71.9785%	71.9573%
Root relative squared error	71.8357%	70.1033%	72.5437%	71.7826%	71.1053%	70.4147%	70.9928%	70.6074%	70.5409
Number of instances	250	250	250	250	250	250	250	250	250

Ex: 1x2 means 1 neuron on first layer and 2 neurons on 2nd

Java WEKA

The main focus of the Java implementation was to tune the network configuration. As stated earlier, this experiment was limited to either one or two hidden layers and up to nineteen neurons per hidden layer. Specifically for this implementation, network configurations were randomly chosen to reduce experimentation time. Below is a table showing the best configuration found for each training session. Training sessions are set apart by the chosen number of epochs. Success of a configuration is based on the Root Mean Squared Error (RMSE).

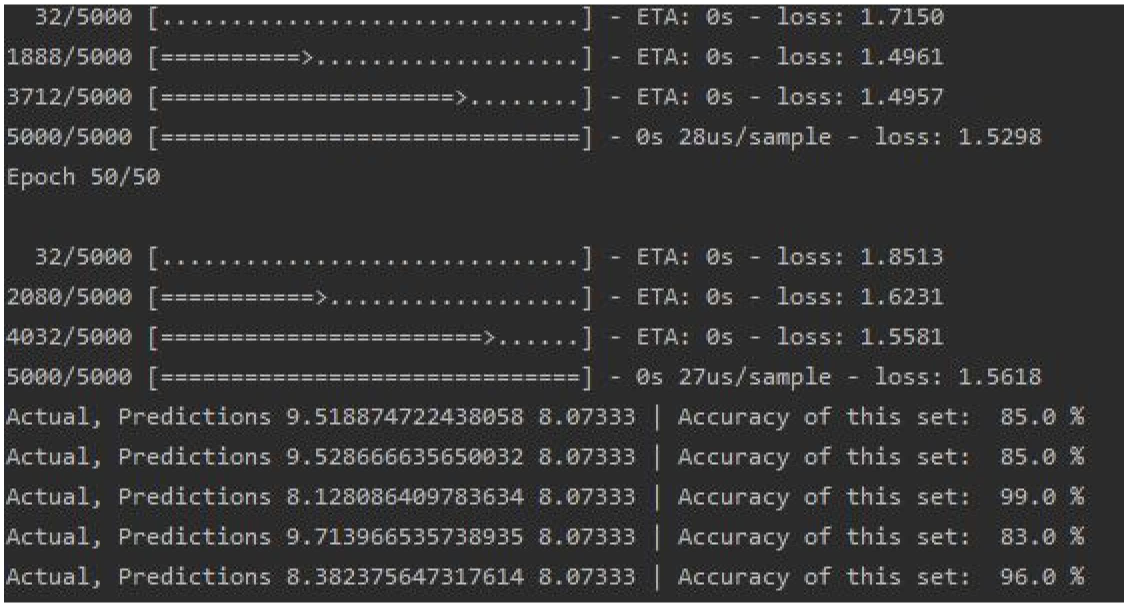
Number of Epochs	Best Network Configuration	RMSE
5	Single 9-neuron layer	0.7359
10	Single 11-neuron layer	0.7337
25	Single 15-neuron layer	0.7332
100	Single 19-neuron layer	0.7287
300	Single 18-neuron layer	0.7264
5000	Single 19-neuron layer	0.7263

Notice here that though two hidden layers were included in the experiment, all of the optimal configurations found only had one hidden layer. The other hyperparameters in the Java implementation were a learning rate of 0.3, momentum of 0.2, and a 50% even split of the sample data between training and testing sets. 10,000 samples were used in total.

TensorFlow

In tensorflow training with a size 10,000, 2 hidden layers, each with 19 neurons, 5 epochs, and a learning rate of 0.2 the average means squared error for each epoch was on average of 1.5. And in testing we get an average accuracy of 92 percent. With one less hidden layer, our accuracy stays around the same, with a drop of 0.100 drop in accuracy.

Using a size of 5000, 2 hidden layers one with 4 and 3 neurons respectively, 500 epochs, and a learning rate of 0.2 we got similar training data, but with a lower average of 89.74% accuracy during testing. Below is a small example of what my data looked like.



```
  32/5000 [..............................] - ETA: 0s - loss: 1.7150
1888/5000 [==========>...................] - ETA: 0s - loss: 1.4961
3712/5000 [=====================>........] - ETA: 0s - loss: 1.4957
5000/5000 [==============================] - 0s 28us/sample - loss: 1.5298
Epoch 50/50

  32/5000 [..............................] - ETA: 0s - loss: 1.8513
2080/5000 [===========>..................] - ETA: 0s - loss: 1.6231
4032/5000 [=======================>......] - ETA: 0s - loss: 1.5581
5000/5000 [==============================] - 0s 27us/sample - loss: 1.5618
Actual, Predictions 9.518874722438058 8.07333 | Accuracy of this set:  85.0 %
Actual, Predictions 9.528666635650032 8.07333 | Accuracy of this set:  85.0 %
Actual, Predictions 8.128086409783634 8.07333 | Accuracy of this set:  99.0 %
Actual, Predictions 9.713966535738935 8.07333 | Accuracy of this set:  83.0 %
Actual, Predictions 8.382375647317614 8.07333 | Accuracy of this set:  96.0 %
```

Analysis

Measuring accuracy

In this experiment accuracy was determined using root mean square error (RMSE). RMSE is a useful measure for regression, since there is no binary value comparison to determine if an output from the model matches the target. Rather, RMSE is the standard deviation of the residuals. In this context, a residual is the difference between the network output value and the target for a given sample. In general, a lower RMSE is better than a higher one. Note in the results section above that as epochs increase the RMSE decreases. This represents the perceptron building a regression model that is a better fit for the solution space. However, because the measure is dependent on the scale of the numbers used, comparisons across different types of data are invalid. For some context within this solution space, z-values range from roughly 0 to 10.

Neural Network Explained

The number of nodes in each hidden layer and the number of layers are main components of the neural network architecture. So, we must specify values for these hyperparameters when configuring our network.The best way to configure these hyperparameters for our specific modeling problem is by systematic experimentation with a strong and useful test.

An Optimal Network Configuration

The main focus of this experiment was to determine if there was a neural network configuration that provided optimal results for the given solution space.Based on a random search optimization of the hidden layer hyperparameters, a single layer with a relatively large number of neurons is the optimal configuration. Note, the solution space that the multilayer perceptron was approximating was nonlinear, as seen in the function defined below.

$$f(x,y) = \sin\left(10\pi x + \frac{10}{1+y^2}\right) + \ln\left(x^2+y^2\right),\ 1 \le x \le 100,\ 1 \le y \le 100$$

As explained by Stephen Marsland, author of *Machine Learning: an Algorithmic Perspective*, two neurons are ample to provide nonlinear regression of any continuous function. He also mentions that in some cases, even one layer is sufficient to approximate nonlinear functions, though the number of neurons required might become quite large.

Works Cited

Marsland, Stephen. Machine Learning: an Algorithmic Perspective. CRC Press, Second Edition, 2015.