Programming Project #4

Chris Wells

# Introduction

The purpose of this project is to implement three different types of networks: A logistic regression and linear network where the inputs feed directly into the output layer, a feedforward neural network with two hidden layers, and an autoencoder network where the autoencoder and prediction network each have one hidden layer.

I applied these networks to each of the datasets. For each dataset, I first split the dataset into five partitions and performed five-fold cross validation. For each fold, I built the network and trained it on four of the partitions, testing it on the fifth partition.

In theory, while I anticipate that the feedforward neural network and autoencoder network will outperform the simple linear networks (the former due to the ability to model non-linear relationships and the latter due to its ability to perform feature extraction) for some datasets, the additional complexity of these networks will make them more difficult to tune and lead to less reliable results.

# Algorithm

All of the network types were implemented as Python classes: LinearRegressor, LinearClassifier, LinearMultiClassifier, FeedForwardNetwork, and AutoEncodedNetwork.

## Linear Regressor

This network uses a single set of weights, in this case a one-dimensional array representing the weight between each input and the sole output node. Given an input Xt, with *j* features, I calculate the output as:

I then find the delta between this output and the expected output *r* for this training row and calculate the update for each of the *j* weights as follows:

I then update the weights with the weight delta multiplied by the model learning rate and the regularization value of the model.

After performing the above on the entire training set, I find the mean squared error (MSE) of the outputs and repeat the process of weight modification until convergence – when the MSE stops decreasing.

## Linear Classifier

Like the linear regressor, this network uses a single output node to classify input into one of two classes. This network uses a single set of weights, in this case a one-dimensional array representing the weight between each input and the sole output node which outputs 0 or 1 depending on its activation. Output is calculated the same as the linear regressor, but after calculating the sum, it is passed through a sigmoid function:

Weight deltas and updating are calculated the same as the linear regressor. After performing the above on the entire training set, I find the accuracy score of the predictions and repeat the process of weight modification until convergence – when the accuracy score stops increasing.

## Linear Multi Classifier

Unlike the above linear networks, this network uses multiple output nodes with each node representing a different class. The weights here are a two-dimensional array, with rows representing each output node and columns each feature. Given an input Xt, with *j* features and *k* output classes, I calculate the output for class *k* as:

The out value, then is an array with a length equal to the number of classes. I next pass this array through the following function to produce a y array of the same length:

Weight deltas and updating are calculated the same as the above linear networks, only operating in two dimensions like so:

Finally, when making a final prediction, I use the node corresponding to the highest output in the y array – the softmax function. After performing the above on the entire training set, I find the accuracy score of the predictions and repeat the process of weight modification until convergence – when the accuracy score stops increasing.

## Feed Forward Network (2 Hidden Layers)

In addition to one or more output nodes, this network has two hidden layers and so will have three sets of weights. Each is a two-dimensional array with the following dimensions:

wH1 = [ # of features, # of nodes in hidden layer 1 ]

wH2 = [ # of nodes in hidden layer 1, # of nodes in hidden layer 2 ]

wOut = [ # of nodes in hidden layer 2, # of outputs ]

For regression there will only be a single output node, but classification datasets will create an output node for each type of output class.

Next, I shuffle the rows of the training set to ensure that the rows Xt and yt are passed through the algorithm in a random order.

Next for each row:

I pass forward using the dot product and the sigmoid function for each hidden layer in succession to generate the output, which will be an array the length of the number of output nodes. To make the process of calculations more intuitive I have used the dot product in my calculations (Salloum, 2019):

Where the sigmoid function performs the following on each element of the output:

Next, I back-propagate from the output to the second hidden layer, calculating the error and sigmoid derivative. For each weight *i* going from hidden layer 2 to the output, I multiply these by output value of the hidden node the weight originates from:

Next, I back-propagate from the second hidden layer to the first by calculating the error for each of *i* nodes in the second hidden layer then taking the derivative of the sigmoid outputs of the second hidden layer, and using these to calculate the change in the first hidden layer weights using these:

Next, I back-propagate from the first hidden layer to the inputs by calculating the error for each node *i* in the first hidden layer, then taking the derivative of the sigmoid outputs of the first hidden layer, and using these to calculate the change in the input weights using these:

Finally, after all the weight deltas are calculated I modify the weight matrices using the deltas and the learn rate:

After this, I repeat for the next row in the training set until the network has trained on all rows. After performing the above on the entire training set, I find the accuracy score using softmax (for classification) or the MSE of the predictions and repeat the process of weight modification until convergence – when the accuracy score stops increasing or the MSE stops decreasing. In addition, to speed up training time I implemented a training “cutoff” – a minimum difference between the current and last scores in order to continue training.

## Autoencoder

The autoencoder can be seen as two single layer, feed forward neural networks set in sequence, with the first network being used to re-create the input X via its hidden layer, or “encoding” layer. This layer is smaller than the input layer and ideally will be able to perfectly recreate the input in fewer dimensions. The output of this encoding layer is then used as the input to another single layer feed forward network which will then make predictions based on this input.

The following sets of weights will be used - each is a two-dimensional array with the following dimensions:

wEnc = [ # of features, # of nodes in encoding layer ]

wDec = [ # of nodes in encoding layer, # of features ]

wH1 = [ # of nodes in encoding layer, # of nodes in hidden layer 1 ]

wOut = [ # of nodes in hidden layer 1, # of outputs ]

To begin, I shuffle the input and pass each row to the autoencoder network, which calculates the output like so:

Like with the feed forward network above, I back propagate from the decode layer through the encode layer and the inputs, modifying both associated weight sets and repeat until the network has trained all the rows. After performing the above on the entire training set, I find Euclidean distance between the vector produced by the decoder and the input vector and repeat the process of weight modification until convergence – when the distance stops decreasing. In order to prevent the system from “thrashing” with increasingly small distances I include a cutoff distance which automatically stops the loop when the distance falls below the cutoff.

Next, I feed the outEnc vector into the prediction network, treating it as the “new” X data. Like the feed forward network above, I back propagate through the hidden layer and repeat until all the rows are trained. I continue training until the classification score or MSE reach their optimum or the training cutoff.

# Results

## Pre-Processing

During pre-processing I performed standard normalization on all numerical values and one-hot encoded all categorical values. In addition, since the feed forward network and autoencoder used multiple output nodes I one-hot encoded the class labels for classification datasets when running those models in order for the model to more easily see how many nodes to create.

## Hyperparameter Tuning

Hyperparameter tuning was simple for the linear networks, and after testing with a learning rate of 0.1, 0.01, and 0.001, the middle value was found to give the best results in a reasonable time frame. Testing regularization values of 0.5, 0.1, and 0.01 on all datasets, the highest value gave the best results.

For the feed forward networks tuning became more complicated, but in general a lower learning rate improved accuracy at the cost of processing time. Likewise, the number of hidden rows in each layer. For the sake of performance, I kept the number of hidden nodes small (three in each layer) for all datasets and set the learning rate from experimentation, using the rate that performed the best in a reasonable amount of time for each dataset. The only exception to this was the machine dataset, which ran so slowly even with a high learning rate that I opted for two hidden nodes per layer.

For the autoencoder tuning became still more complicated. Using a relatively high learning rate of 0.1 sped up the encoding process without impacting performance on any datasets. I wanted the encoding layer to have minimal loss so I set the number of nodes in it to be one less than the number of features in the original dataset. Finally, I used 6 hidden layers in the processing network in order to make up for its lack of depth. Again, learning rate in the processing network was determined by experimentation on each dataset.

## Performance

Results are shown below compared to the Null Model from Project 1:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **REGRESSION** | **MSE** |  |  |  |  |
| **Dataset/Fold** | **Linear** | **Feed Forward** | **Autoencoder Loss** | **Autoencoder** | **Null Model** |
| **Abalone** |  |  |  |  |  |
| **Fold 1** | 6.65 | 8.26 | 0.07 | 33.64 | 11.50 |
| **Fold 2** | 6.61 | 7.28 | 0.08 | 37.36 | 9.39 |
| **Fold 3** | 6.69 | 8.09 | 0.07 | 29.40 | 10.13 |
| **Fold 4** | 7.97 | 8.25 | 0.07 | 27.82 | 10.31 |
| **Fold 5** | 6.07 | 8.40 | 0.08 | 33.47 | 9.90 |
| **Avg** | **6.80** | **8.06** | **0.074** | **32.34** | **10.25** |
| **Forest** |  |  |  |  |  |
| **Fold 1** | 268.11 | 563.53 | 0.71 | 825.56 | 495.00 |
| **Fold 2** | 7,000.14 | 2,121.35 | 0.70 | 968.21 | 1,522.41 |
| **Fold 3** | 353.75 | 632.46 | 0.71 | 11,806.5 | 7,159.40 |
| **Fold 4** | 11,743.10 | 16,567.08 | 0.71 | 711.54 | 14,423.31 |
| **Fold 5** | 669.99 | 341.47 | 0.71 | 6,812.56 | 1,338.95 |
| **Avg** | **4,007.02** | **4,045.18** | **0.71** | **4,224.78** | **4,987.82** |
| **Machine** |  |  |  |  |  |
| **Fold 1** | 9,732.35 | 18,644.57 | 0.13 | 31,943.23 | 11,583.53 |
| **Fold 2** | 3,947.27 | 2,007.61 | 0.20 | 15,068.44 | 17,743.86 |
| **Fold 3** | 12,267.79 | 25,155.19 | 0.33 | 32,917.85 | 15,501.21 |
| **Fold 4** | 3,855.35 | 40,545.28 | 0.27 | 88,075.44 | 77,886.21 |
| **Fold 5** | 14,347.18 | 7,847.63 | 0.24 | 36,613.63 | 26,855.64 |
| **Avg** | **8,829.99** | **18,840.06** | **0.23** | **40,923.72** | **29,914.09** |
| **CLASSIFICATION** | **% Acc** |  |  |  |  |
| **Dataset/Fold** | **Logistic** | **Feed Forward** | **Autoencoder Loss** | **Autoencoder** | **Null Model** |
| **Breast** |  |  |  |  |  |
| **Fold 1** | 94.26% | 61.43% | 0.31 | 42.86% | 65.18% |
| **Fold 2** | 96.43% | 42.86% | 0.25 | 43.57% | 65.77% |
| **Fold 3** | 97.84% | 54.68% | 0.31 | 37.41% | 65.18% |
| **Fold 4** | 94.29% | 43.57% | 0.24 | 37.86% | 66.07% |
| **Fold 5** | 95.71% | 42.14% | 0.27 | 45.00% | 65.18% |
| **Avg** | **95.71%** | **48.94%** | **0.28** | **41.34%** | **65.47%** |
| **Car** |  |  |  |  |  |
| **Fold 1** | 69.94% | 69.94% | 0.62 | 16.18% | 70.29% |
| **Fold 2** | 69.94% | 69.94% | 0.62 | 2.02% | 70.04% |
| **Fold 3** | 69.94% | 69.94% | 0.63 | 15.03% | 69.93% |
| **Fold 4** | 69.86% | 70.14% | 0.62 | 9.57% | 70.03% |
| **Fold 5** | 70.14% | 70.14% | 0.62 | 1.74% | 69.93% |
| **Avg** | **69.97%** | **70.02%** | **0.62** | **8.91%** | **70.04%** |
| **House** |  |  |  |  |  |
| **Fold 1** | 93.10% | 91.95% | 0.62 | 0.00% | 61.43% |
| **Fold 2** | 97.73% | 80.68% | 0.70 | 0.00% | 61.43% |
| **Fold 3** | 96.51% | 94.19% | 0.60 | 0.00% | 61.43% |
| **Fold 4** | 96.59% | 92.04% | 0.59 | 0.00% | 61.76% |
| **Fold 5** | 89.53% | 86.05% | 0.63 | 1.16% | 61.43% |
| **Avg** | **94.69%** | **88.98%** | **0.63** | **0.23%** | **61.50%** |

# Discussion

Across all the datasets, the performance of the linear/logistic and feed forward networks was comparable, though there was more variance in feed forward. The performance suffered significantly for the autoencoder. The low loss - distances between the decoded values and the original input vectors - indicate that the original input was being reconstructed with a reasonable degree of fidelity, so I believe the issue with this is a combination of some of the models being poorly tuned and bugs within the back propagation calculation that might also be the source of some of the lower scores in the feed forward network.

A major problem with getting the most performance out of the feed forward and autoencoder models were the significantly longer time taken for them which precluded testing with a wide range of hyperparameter settings.

The use of standardization on all numerical features contributed greatly to the increased performance of the Abalone, House Votes, and Machine datasets compared to the results from the previous programming project. Going forward, I will be continuing this practice.

# Conclusion

In this project three types of neural networks were successfully modeled and run on all datasets. Contradicting my hypothesis, the additional functionality of the feed forward and autoencoder did not outperform linear/logistic regression in general. I believe this is due to a combination of errors in the implementation of these more complex models or poor tuning of the models.

In my opinion, this project confirms the usefulness of relatively transparent, fast-processing models which allow for easy implementation and debugging and multiple, rapid runs to establish the best hyperparameter settings from a wide range. Both of these are true of linear/logistic regression and less true of the other two networks and their poorer performance speaks to the importance of these features.

# Works Cited

Salloum, Z. (2019, January 5). *Back Propagation, the Easy Way*. Retrieved from Towards Data Science: https://towardsdatascience.com/back-propagation-the-easy-way-part-1-6a8cde653f65