Utilization of Numerous Neural Networks to Classify and Regress

Affan Dhankwala

Abstract

Neural Networks are data structures capable of regression and classification by incorporating multiple layers and nodes within each layer to alter the input data to match the value of the expected value. In other words, given any set of inputs, a neural network, if properly trained, will perform calculations on said input values and translate them through the various layers to get a predicted output value within reasonable error rates. This paper explores the training, testing and tuning of three different neural network architectures.

1 Problem Statement

Throughout the course of module eight through eleven, we have been tasked to create, train, and tune three different neural network architectures that are compatible with the six provided datasets. These neural networks must be trained sufficiently to both classify and regress any example based on the input values passed in. The performance of these modules shall be measured based on the type of module. For regression networks, MSE shall be the dominant form of error measurement. For classification networks, cross entropy loss will be employed.

We believe that once properly trained our modules will be able to reasonably determine the example's predicted value. We also believe that the simple neural network will have the lowest performance as it merely employs a simple linear/logistic regression for regression/classification respectively. Secondly, we expect the neural network with two hidden layers to have a lower loss function than the simple NN. This is based on the reasoning that the two hidden layers will act as noise reducing agents and fit the function better to the dataset. Finally, we expect the neural network trained on the autoencoder to have higher performance because it endures a two-fold training process for encoding and training. After being encoded, it is expected that the first encoding layer will be fine-tuned to handle the current training data. Then a second layer is added to this encoding layer—locking the first layer's weights and trained via feed-forward back-propagation (FFBP) logic. Due to the intensity of this algorithm, it is expected that the auto-encoded neural network will display a higher performance than the other two algorithms.

With regards to the actual functions of the neural networks, we believe that classification networks will perform better than regression networks because most of our regression datasets have such large variance that it would be difficult to accurately predict any value. In addition classification networks are expected to have a lower loss because we build off the implicit bias that examples with similar features must be classified similarly.

2 Experimental Approach

To efficiently discuss the experimental approach to the creation of this model, it is critical to examine the major files and sections of this model. We shall first examine the loading and preprocessing stage.

Load and Pre-process

The loading phase consisted of capturing all the feature data from its respective dataset and translating it into a dictionary within the 'features' key. Each element of this list is a Feature object which contains the name, type, possible values, and data of the feature. Examples can be extracted by pulling all the feature values at any particular index. Class labels are likewise input into the dataset under the 'label' key which is one Feature object. The 'label' Feature object's indexes are aligned with the 'feature' Feature object's indexes.

Pre-processing the data involved confirming that the dataset contains no null values. If a null value is detected, it is replaced. For nominal features, the null value is replaced by the majority label—mode—of the feature. For continuous or discrete features, the null value is replaced by the median of the feature. The second step of pre-processing is to verify that all the data is in the correct format. A string should remain a string, but floats written in a string format should be converted to floats. We also normalize all continuous and discrete data. The final step of pre-preprocessing involves numerically encoding both the feature and the label nominal values. This is done by storing all values that any nominal feature or label could hold within a member variable called 'values'. Then, after storing all possible values, we iterate through each nominal feature and replace the value with its index within the values list. This resulted in a numerical encoding of all nominal values.

Training

Training of the neural networks was done after all the data was loaded and preprocessed properly. We employed a 5x2 cross validation algorithm to train and test our data. This was done by first shuffling the overall dataset to eliminate any unintentional ordering biases. We then split the datasets into five feature and label pairs. Each of these feature and label pairs were split at an 80/20 proportion for training and testing. Once the splitting was complete, we shuffled the split data again and trained all three models based on the 80 portion. Once each respective neural network model was trained, we tested the performance by measuring the loss on the 20 proportion data. Finally, we reported the loss data. This way we trained each network five times and validated them five times—concluding in a total of ten runs per neural network per dataset.

The training for all three of the models is a bit different. For the simple NN model, we pass in all the values as input nodes. We then make a prediction based on the neural network. For all networks, we calculate the Z combination of the network with the following equation:

$$Z = X \cdot W + \vartheta$$

X is a matrix of all the examples and their respective features while W is a weight matrix from each input to the output node. Once this Z value is calculated, we need to extract the prediction values from it. For a regression problem, the predicted values are the Z values and no further calculations is necessary. For a classification, we need to determine the softmax of Z. The softmax of Z is a matrix of size example count * label count (row * col) which stores the probability of each class within each element at each row.

Once these predictions are made, we determine the loss of our current model. For regression, this is simply the MSE between the true labels and the predicted labels. For classification, we

determine the mean cross entropy loss. This is done by first altering the previously calculated matrix by subtracting 1 from each correct class label. In other words, if a particular row was valued at [2, 3, 1], and the correct class index for this example was 1 then we would subtract 1 from index 1 and result with the new row of [2, 2, 1]. This new matrix was known as 'true_class_probabilities'. To compute the mean cross entropy loss, we utilized the below equation

$$Mean\ Cross\ Entropy\ Loss = -\frac{\sum_{i=1}^{example\ count}\left(\sum_{j=1}^{label\ count}(\log(true\ class\ probability[i][j]))\right)}{example\ count}$$

After the calculation of the loss functions, we entered into the back propagation portion. The error was then dotted with the input values (X) to return the output layer's weight gradient. Similarly, the bias gradient was a sum of the error matrix. These gradients were multiplied with the learning rate, eta, and subtracted from the current weights and gradients. This whole sequence was considered one epoch. For the second epoch, the saved weights and gradients were used. Over the course of a hypertuned number of epochs, we aimed to reduce the overall MSE or mean cross entropy loss.

For a neural network with two hidden layers, we employ similar tactics to the calculation procedure of the simple neural network. When calculating the Z value, instead of just determining the output, we calculate the Z value at each hidden node and then activate it via the sigmoid function. Once activated, we take these values and treat them as inputs into the second hidden layer. Dot product and activate these values. Treat these activated values as input nodes into the final layer. For the error and loss functions, we utilize MSE or mean cross entropy depending on the neural network's task. When backpropagating, we must factor in the error propagation to each layer and this requires us to utilize the sigmoid_derivative equation. From this propagated error matrix, we calculate the weight and bias gradients for each layer. Similar to the simple NN, we alter all the weights by the gradients and aim to reduce the overall error over a set number of epochs.

For the final network consisting of an autoencoder, we first set up a network with all the inputs pointing to an encoding layer. This layer consists of a count of nodes that is strictly less than the number of nodes in the input layer. The feedforward and backpropagation are similar to previously discussed with weights and bias gradients through the layers. Once we iterate over a certain number of epochs, we lock the weights from the input layer to the encoding layer and save the encoding layer values. Then we add a hidden layer to the encoded layer that points to the prediction layer. This new network is then trained via FFBP. Keep in mind that we have locked the weights between the input and encoding layer so the only values being fine tuned are the weights between encoding and hidden layer and weights between hidden layer and outputs layer.

Testing

Once each model is given a set of data, we retrieve the weights and bias matrices. For a simple NN and a NN with hidden layers, we use the test_nn method that calculates the loss from a neural network initialized with the given weights. At this point, we use the 0.20 portion of the data and

determine the overall loss of the entire dataset. Keep in mind that, although we employ feedforward logic, we do not conduct any backpropagation. This is because we do not aim to alter the weights of the network.

For the auto encoded NN, we conduct a similar approach but the model consists of the weights between the encoding layer and the hidden layer as separate from the weights between the hidden layer and the output layer. Similarly to the other testing sequence, we conduct feedforward algorithm to calculate the values at each hidden nodes and the value at the output but we do not employ any sort of backpropagation.

Once all models are run, we print out the loss for each of the models over all the datasets. This results in a list of five losses per model per dataset. These values are stored and shared within this document.

Tuning

A critical portion of this assignment included the tuning the hyperparameters. The many hyperparameters required the need to conduct some tuning process to verify that we were utilizing the appropriate values per hyperparameter. The hyperparameters of this project included the following:

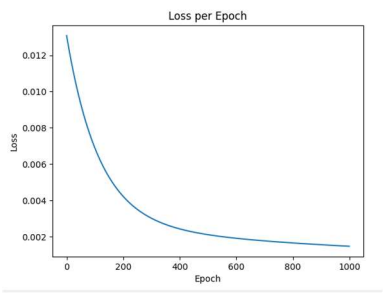
- Epochs: The number of times we had to iterate through all the data in any given dataset for the construction of the neural network. If this parameter is too small, our model may be underfit and produce high errors. If this parameter is too large, our model may either be overfit and produce high errors, or simply conduct an excessive number of calculations.
- Eta: The learning rate of each model is a generic hyperparameter that most models require. Since we employed a gradient descent deterministic approach, we needed to verify that this value was appropriate for the weight and bias adjustment. If this value is too small, our model will take too long to converge. If this value is too large, our model may diverge.
- Hidden Nodes: Since we were allowed to choose the number of hidden nodes in each layer, this becomes a hyperparameter that needs to be adjusted. The number of nodes per hidden layer drastically affects the calculated weight matrices and can alter the performance of the model.
- Auto Encoder Nodes: Similarly to the hidden nodes, we must determine how many nodes to place within the encoding layer and predicting layers when building an autoencoder trained neural network. The encoded layer must have less nodes than the input features as the purpose of an encoder is to reduce the number of nodes.

To combat the tuning of all these parameters, we conducted the following steps:

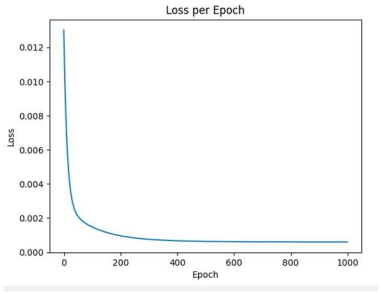
First, we built a simple NN with a large number of epochs and retrieved the loss of each epoch. From here, we determined that if the loss was decreasing by anything less than 0.1%, we did not care. Meaning that we kept running epochs until any epoch failed to drop the current loss by 0.1%. This was the cutoff performance. We also drew a graph of all the losses per epoch in case

our cutoff was unreasonable. If we determined that the cutoff epoch was not on the optimum part of the graph, we disregarded it and manually entered the visually determined optimum epoch.

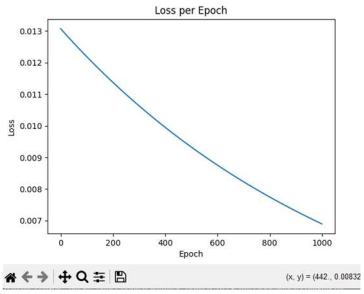
After training the epoch, we embarked on training the eta value. The eta value had a twist to it as it required us to visually determine whether a certain model was trained enough via a graph. See the graphs below that were trained on the Abalone dataset:



Eta: 0.001



Eta: 0.01



Eta: 0.0001

The left most graph shows the optimum eta value and this is because the graph begins to converge near the end of the 1000 epochs. Although not depicted, our model predicted that the optimum epoch count for the abalone dataset is 542. At 542, our graph is almost converged, and this is a good place for model prediction as we have balance between speed and accuracy. The middle graph depicts an eta value that is too large and therefore overfitting our data. At epoch = 542, our model is already overfitted and converged and therefore it is not a good model. For the final right graph, the eta value is too small and therefore we are underfitting the data. At epoch = 542, our model is too inaccurate to be utilized. We could increase our epochs and increase the accuracy but that would come at a cost to our computational speed.

After tuning the epochs and eta, we embarked on tuning the hidden node values. This was done by setting up an $O(N^2)$ algorithm that would build a hidden layer NN with the first layer consisting of n number of nodes where $1 \leq n \leq 15$ and a second layer consisting of m nodes where $1 \leq m \leq 15$. All possible combinations had their losses determined and the lowest loss combination was stored.

Similar to the tuning of the hidden layer nodes, the autoencoder nodes were likewise trained in an $O(N^2)$ approach but the first encoding layer was size n where $1 \leq n \leq input\ feature\ count$. This is in accordance with the autoencoder principles of reducing the number of nodes required for calculation. The second layer was identical to the hidden layer NN's second hidden layer with size m where $1 \leq m \leq 15$.

After all these hyperparameters were tuned, we created a method to simply pull the values into the cross validation method. This method was called the get_tuned_values() and it contained the following information:

Dataset	Label type	Epochs	Eta	Hidden Nodes	Autoencoder Nodes
Abalone	Regression	542	0.001	[1, 8]	[4, 5]
Breast-cancer-wisonsin	Classification	408	0.005	[15, 15]	[4, 5]
Car	Classification	500	0.001	[3, 15]	[4, 5]
ForestFires	Regression	200	0.01	[7, 4]	[4, 5]
House-votes-84	Classification	293	0.1	[15, 4]	[4, 5]
machine	Regression	482	0.0005	[1, 2]	[4, 5]

Presentation of Results

As part of the testing phase, all six datasets were considered for the training and testing of all neural networks. The performance of each of the networks is shown below:

DATASET	LOSS						TIME (S)
	Simple NN					Mean	
	0.0093	0.0093	0.0093	0.0093	0.0093	0.00930	
	Hidden Layer NN					Mean	
	0.0114	0.0115	0.0115	0.0114	0.0115	0.01146	
	Auto Encoded NN					Mean	
Abalone	88.512	91.4258	90.1006	90.3533	90.6263	90.20360	28.12
	Simple NN					Mean	
	0.136	0.1357	0.1363	0.1373	0.1348	0.13602	
	Hidden Layer NN					Mean	
	0.6884	0.6884	0.6884	0.6884	0.6884	0.68840	
Breast-cancer-wisconsin	Auto Encoded NN					Mean	
	0.6166	0.2767	0.5813	0.6754	0.6227	0.55454	22.69
	Simple NN					Mean	
	1.0038	1.002	1.0039	1.0023	0.9989	1.00218	
	Hidden Layer NN					Mean	
	0.9905	0.9943	0.9955	0.9956	0.9979	0.99476	
	Auto Encoded NN					Mean	
Car	0.7758	0.8641	0.8108	0.8446	0.7878	0.81662	80.72
	Simple NN					Mean	
	4.5008	4.5008	4.501	4.5008	4.5009	4.50086	
	Hidden Layer NN					Mean	
	4.4344	4.4359	4.4356	4.4354	4.4378	4.43582	
	Auto Encoded NN					Mean	
Forest fires	1481.864	6233.18	588.814	12226.2	399.266	4185.85406	1.46

House-votes-84	Simple NN					Mean	
	0.0989	0.0986	0.0987	0.0987	0.099	0.09878	
	Hidden Layer NN					Mean	
	0.6542	0.6542	0.6542	0.6542	0.6542	0.65420	
	Auto Encoded NN					Mean	
	0.6232	0.7859	0.4764	0.6145	1.1358	0.72716	9.02
Machine	Simple NN					Mean	
	56.0256	56.012	56.0103	56.0233	56.0123	56.01670	
	Hidden Layer NN					Mean	
	395.2085	394.005	394.063	395.078	395.396	394.75006	
						Mean	
	16549.9	44571.6	456334	29557.4	47384.5	36739.4594	1.21

Discussion and Conclusion

From the results shown above, we can draw many statements and conclusions and tie them back to the original hypothesis:

It is evident that datasets that were classified had an overall lower loss than those that were regressed. This agrees with the original hypothesis that we would see a higher performance in our classification models over our regression models. This may be because our classification models employed a softmax activation function while the regression models simply had a linear model. This limits the number of distinctions that we are expected to see within our output. It is also evident that there are distinct classes between different labels for classification. This allows the model to focus on the probability of certain or one class label and disregard the rest. For regression, there aren't as well-defined boundaries between different output values and this can increase the loss between the predicted and true values.

Contrary to our hypothesis, most of the NN with hidden layers performed worse than their simple counterparts. We had assumed that, with the additional layers, our model would have been able to foresee more complex behaviors and account for them. However, it is possible that our data may be too small for these parameters to help us. These extra layers could be damaging the performance of our model by overfitting them. This theory is supported by the difference between the performance of the simple NN and the hidden layer NN being almost identical within the Abalone dataset—which consists of more than 4000 examples. On the contrary, the difference between the loss of the simple NN and the hidden layer NN on the machine dataset—which consists of a little over 200 examples—is drastically high. Due to the dataset size limitation, it is also possible that we were unable to optimize all values within the given training set which hints at the possibility that our models were underfit.

Another critical observation is that the loss for our auto encoded NN is far higher than the error of any other model. This issue may be more of an operator issue than a model inefficiency. When

creating the autoencoder, there was some doubt on how the encoding layer would be linked to the second hidden layer and how much to encode it. This could have led to some mismatch between the encoded and predicted layers where the alignment of the two layers is off and therefore, even if it is minimizing the reconstruction error, it may be producing worse downstream predictions and therefore skyrocketing the loss metrics.