Model Descriptions

In TensorFlow, there are several types of layers that can be added to a neural network model. In this experiment 2 types of layers were used:

The first layer is called a dense layer. With dense neuron layers, each layer will use its neurons to select features from the supplied feature vector and perform a classification, passing down the classification and criteria to the next layer. In these dense layers, also called fully connected layers, each neuron in the layer is directly connected to every neuron in the preceding layer.

The second type of layer that was used is known as a dropout layer. A dropout layer is created with a dropout rate passed as a parameter. This layer will disconnect a percentage of neurons from the previous layer, preventing those neurons from passing input to the subsequent layers. The percentage of neurons dropped is given by the dropout rate. The purpose of the dropout layer is to prevent overfitting.

Each dense layer must be created with an activation function, which defines what type of calculations and classifications the neurons in that layer will complete. Two different activation functions were used with the dense layers:

The ReLU activation function is an implementation of the rectifier function: where x is the input to the neuron. This was used as the activation function in all layers except the last layer in the model.

The Sigmoid activation function was always used as the last layer in all models. The purpose of using this function as the last layer is to normalize output.

Three different model designs were tested. The first model was the most simple model, containing a single dense layer with a base number of neurons, followed by a dropout layer that would randomly reduce the number of neurons in use at a certain time according to the dropout rate given. A dense sigmoid layer of size 1 was used as the output normalizing last layer. The second model used was a deeper network that utilized 4 dense layers, all with the base number of neurons. A dropout layer was added after each dense layer. A dense sigmoid layer of size 1 was used as the output normalizing last layer. The third model was also four layers deep, however the number of neurons in each layer was decreased. The first layer contained the base number of neurons and the subsequent layers contained the base number divided the depth of the layer neurons (Layer 1: x neurons, Layer 2: x/2 neurons, Layer 3: x/3 neurons, Layer 4: x/4 neurons). A dropout layer was added after each dense layer. A dense sigmoid layer of size 1 was used as the output normalizing last layer.

Grid Search

To find the hyper-parameters that are best suited for this dataset and classification scenario, a grid search is used. The grid search takes as input, a dictionary of parameter names and values, then runs a full test using each combination of parameters. The hyper-parameters that we were concerned with tuning include batch size, epochs, number of neurons per layer, optimizer, dropout rate, and weight constraint and class weight. Each parameter was placed in an individual array containing the desired test values. Each parameter’s array was then placed in a larger dictionary that formed the grid of parameters to be searched. The grid search function would test every possible set of parameters, outputting the results to a csv and determining the best possible set of parameters to use.

Epochs and Batch Size

An epoch is defined as a single pass over an entire training set. More epochs means the model is trained against the same training data more times. Epochs were tested in a range from 1 epoch to 32 epochs.

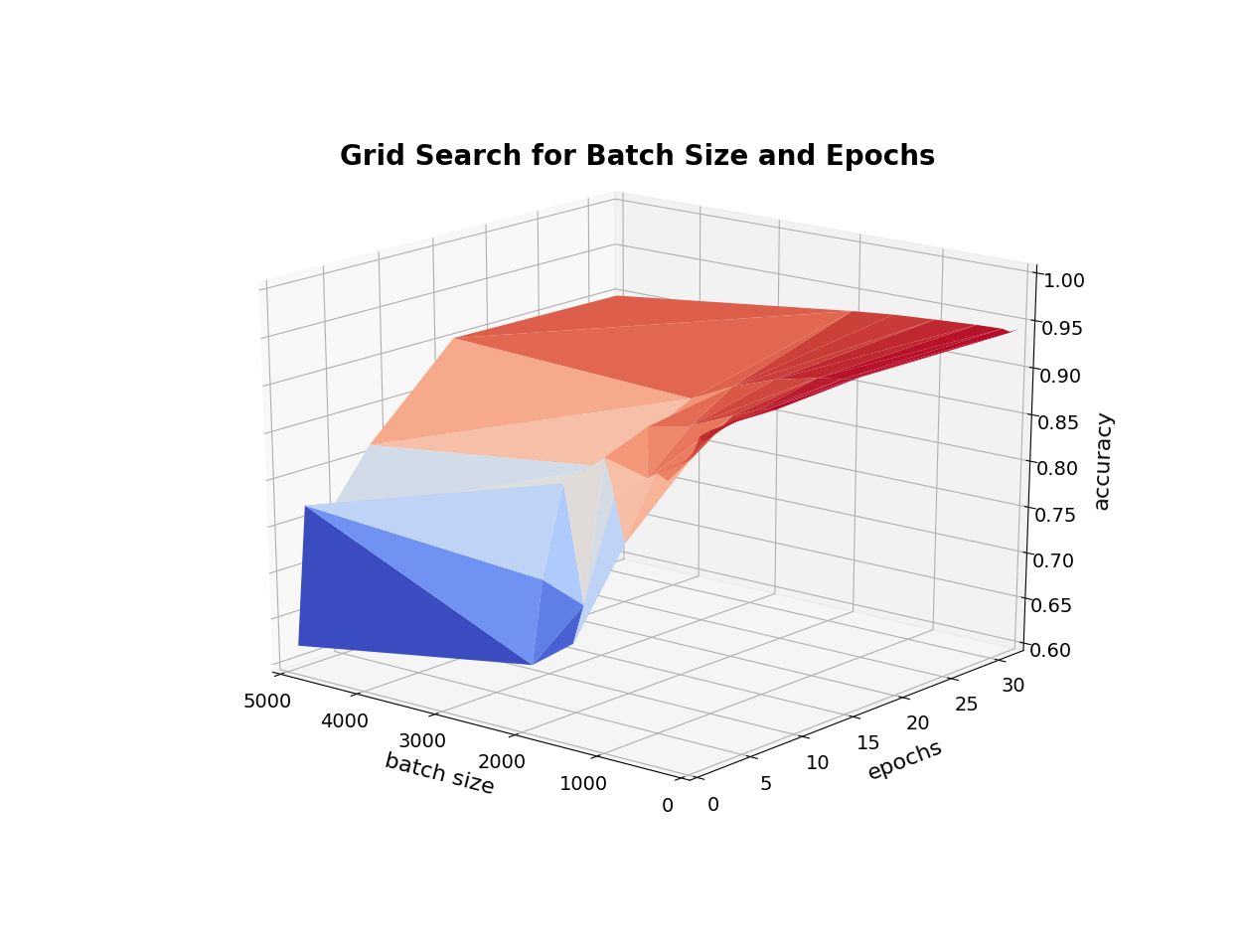
In most cases, as the number of epochs was increased, accuracy increased as well. Although, in some cases it was noticed that accuracy had a slight decline after 16 epochs. This could be due to the model overfitting itself to the specific features of the training data and therefore performing slightly worse on data that it has never seen before. Also, as epochs increase, the time it takes to train the model increases significantly. This is to be expected though, as the model is making multiple passes over the training data. We found that the increase in time was approximately linear with increasing epochs. A double in the number of epochs was approximately equal to a double in the time taken to train the model.

Batch size is defined as the number of data points that are observed at once during training. The model will be fit with the number of data points defined in the batch size repeatedly until the entire data set has been trained on for all epochs. For example, if the batch size is 500, then the model will train based on the first 500 pieces of data, then train further using the next 500, and so on until the training is complete. Batch size in our scenario ranged from a very small batch of size 10 to a large batch of size 5000.

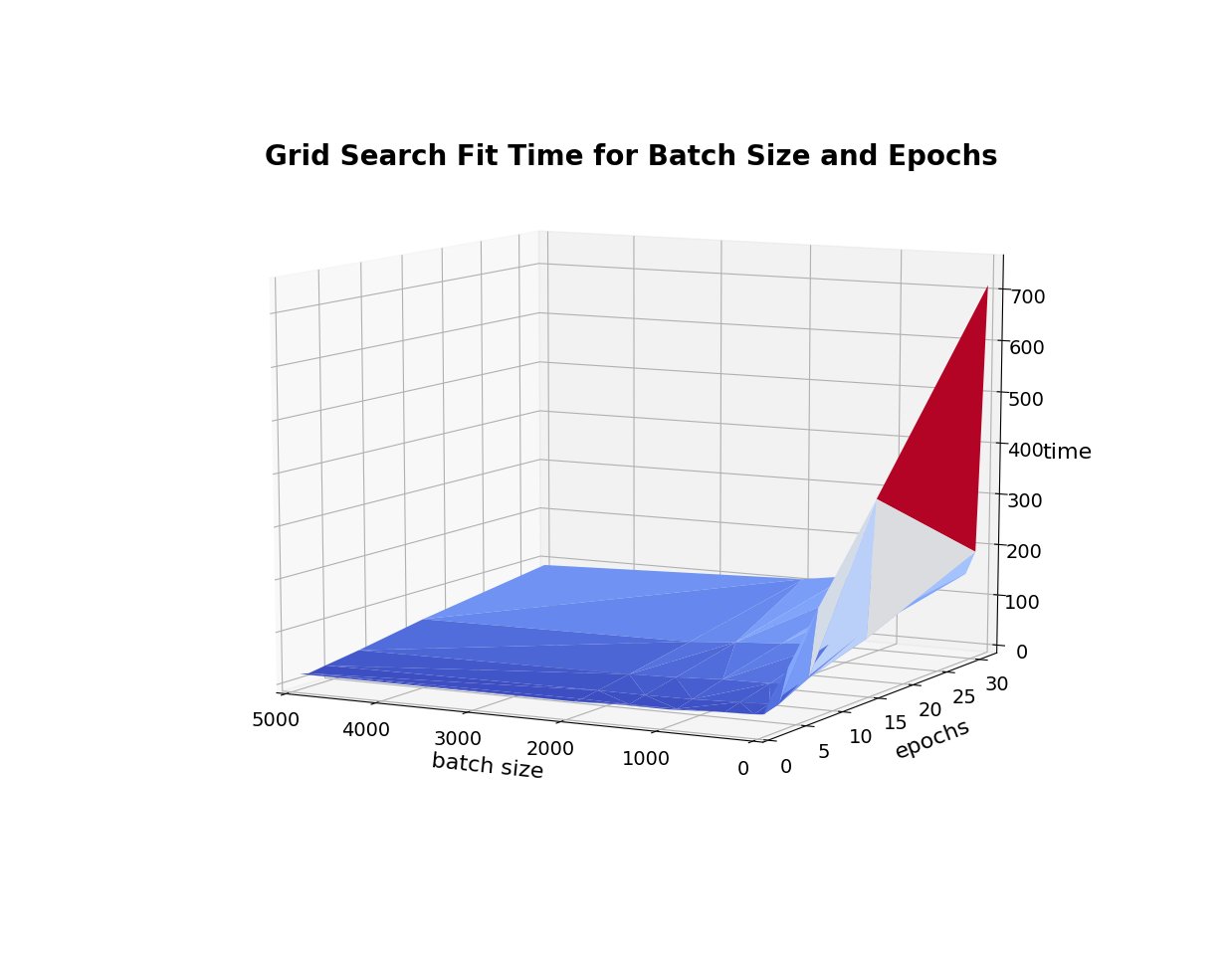
Based on the results, it can be determined that smaller batch sizes lead to the model taking a longer time to train and test. This is the result of the model having to refit itself with each batch. Smaller batches, inherently means that the model will refit itself more times. Additionally, it was seen that smaller batch sizes leads to higher accuracy. There was a 5% difference in accuracy between the highest result at batch size of 10 and lowest result at batch size of 5000.

When examined together, epochs and batch size have an interesting relationship. Increasing batch size and epochs tend to lead to higher accuracy. Though, a small batch size does not see much of an improvement by increasing epochs. A small batch tends to only achieve about 1% better accuracy than a small batch with a high number of epochs. On the contrary, there is a significant difference in the accuracy achieved by a model with a large batch size trained over low and high epochs. At a batch size of 5000, a single epoch training could only achieve approximately 62% accuracy, whereas the same batch size with 16 epochs could achieve approximately 90% accuracy.

In terms of time, it was noticed that batch size had a much larger impact on time than did the number of epochs. The difference in time for the smallest batch size at 32 epochs was over 680 seconds longer than the same batch size at 1 epoch. The difference in time for the largest batch size at 32 epochs was only 70 seconds longer. The training time for a single epoch at the smallest batch size was 10 seconds longer than that at the largest batch size.



<3D EPOCH/BATCH/TIME GRAPH>



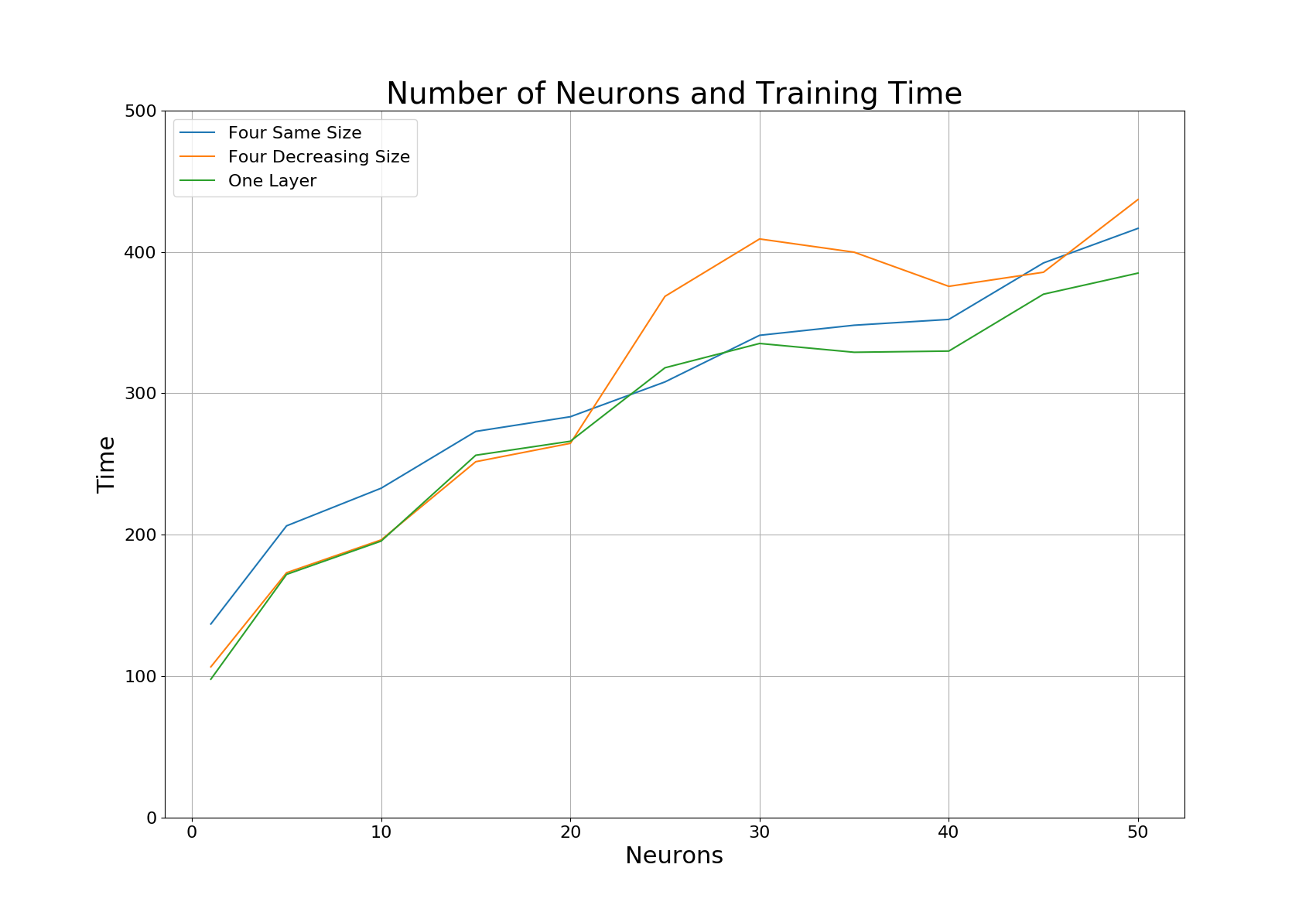
Number of Neurons

The most successful number of neurons per layer was tested in a grid search. A neuron in this sense is the artificial entity making a decision or classification based it’s known criteria. Each model has its own neuron scheme, as defined in the model description section. Values for the number of neurons were tested as multiples of 5 ranging from 1 to 50 neurons per layer (1, 5, 10, …, 50). For the four layers of decreasing size model, integer division was used and the minimum number of neurons in any given layer was 1. Any division resulting in a quotient of 0 was reassigned to 1 in order to preserve the shape/depth of the model.

Through this grid search it was evident that increasing the number of neurons in the model generally led to a higher accuracy, though there was only a slight increase in accuracy. Higher number of neurons can be expected to yield more accuracy due to more features being utilized for classification and more comparisons between layers as a result. In some cases, too many neurons can lead to the network overfitting the model to the training data and therefore underperforming on unseen testing data. This was noticed in our grid search as well. The models utilizing 50 neurons performed worse than the 45 neuron models in most cases, when considering accuracy.

The one layer model showed the least variation in accuracy with additional neurons added. The best result for this model was the 45 neuron test with the other results within 2%. The four layers of the same size model showed improvement with adding neurons. The one neuron test produced an average accuracy of approximately 68%, while the best result produced 94.6% with both 45 and 50 neurons. The four layers with decreasing size model produced an accuracy of approximately 65%, on average, at 1, 5 and 10 neuron tests. The model showed an increase in accuracy at 15 neurons. The best result for this model was the 45 neuron test with 94.6% accuracy.

Additionally, it was noticed in all models that increasing the number of neurons increases the amount of time taken to train the model. This is to be expected because with more nodes in the network, more calculations and classifications must be done in each layer. Because there was only a small increase in accuracy by adding additional neurons, one may consider using a more simple neural network to achieve relatively high accuracy if time is constraining factor.

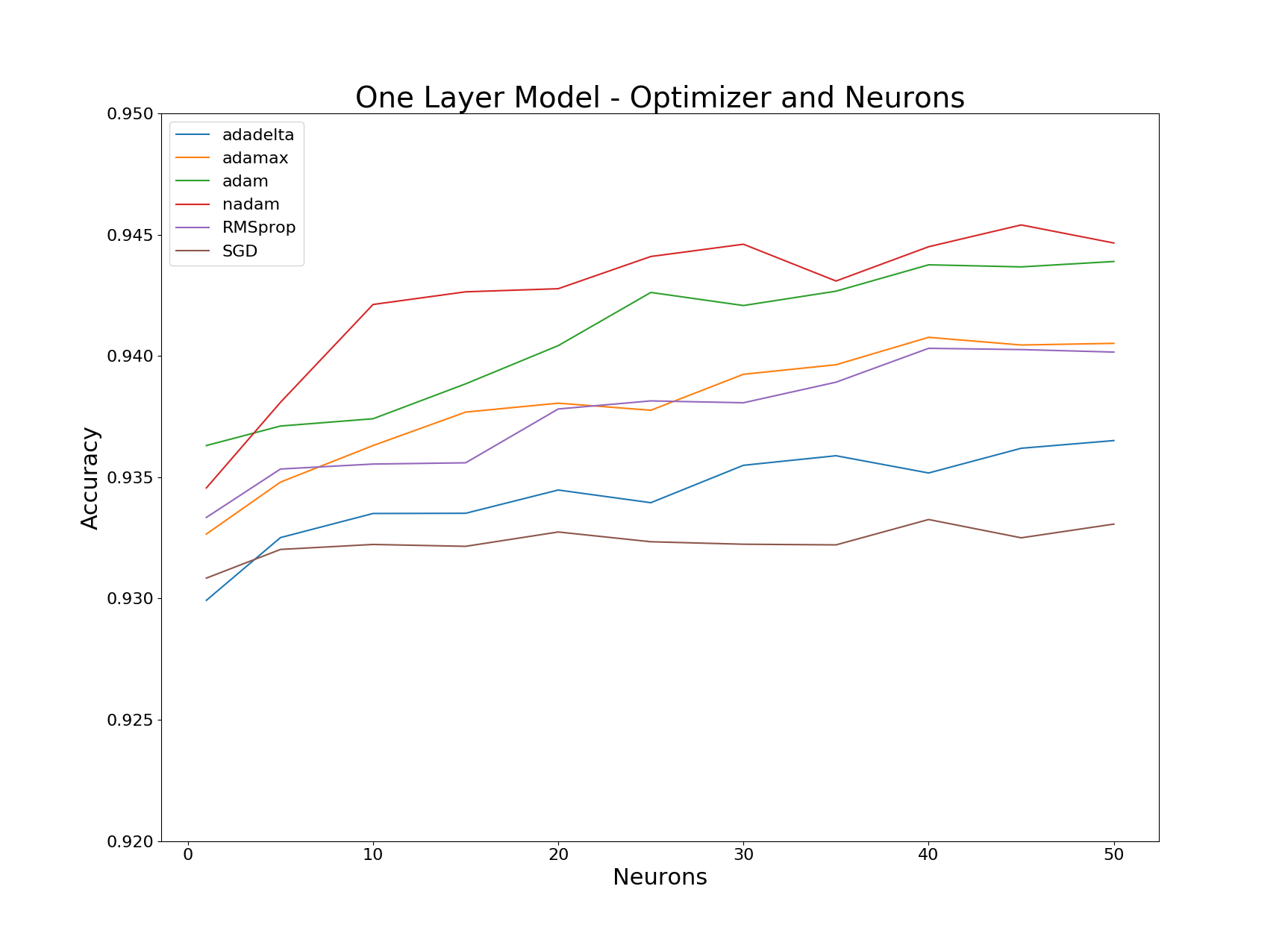


Optimizer

Several optimizers were chosen based on compatibility with the Keras wrapping infrastructure that we are using. Enumerated, these optimizers are: SGD, RMSprop, Adagrad, Adadelta, Adam, Adamax, and Nadam. Each of these optimizers performs slightly different mathematical calculations in order to optimize the results produced by the loss function. Each optimizer has a different set of tunable parameters, which may or may not be unique to that optimizer, that often initialize values in the formulas and calculations. In this grid search, the parameters specific to the optimization algorithms were not tuned and default values were used.

It was evident, through this test that all of the optimizers tested produce similar results, in terms of test accuracy, with this data set. With all three models, all optimizers were within 1% of the best performer, Nadam at 94.5%.

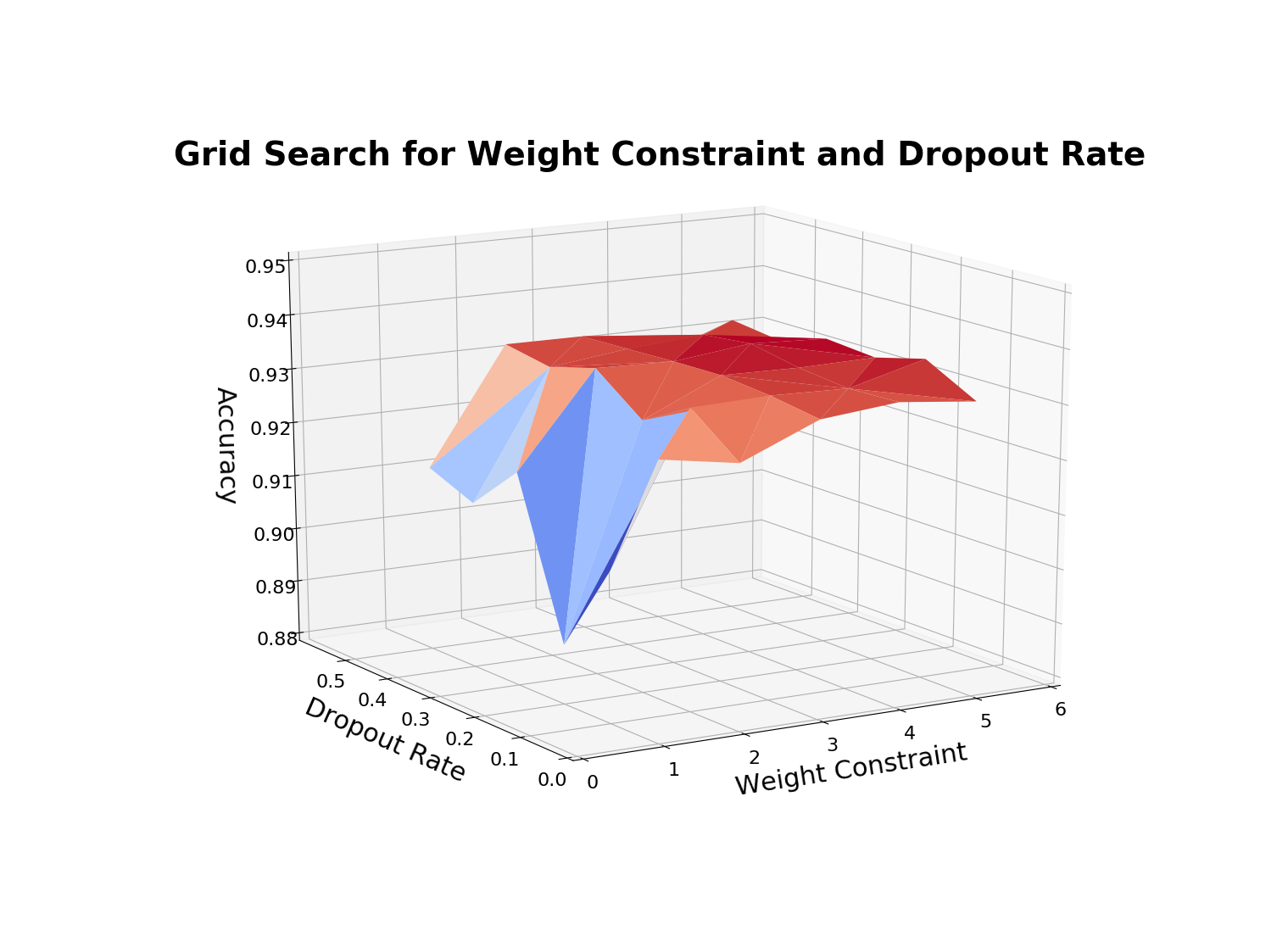
The biggest difference in the optimizers was seen with the time taken to train the model, not the achieved test accuracy. The fastest optimizer was consistently the SGD optimizer and the slowest was consistently the Nadam optimizer. On average, the Nadam optimizer took 109 seconds longer to train than the SGD optimizer. Optimizers besides SGD and Nadam, produced similar accuracies in times between SGD and Nadam.



Dropout Rate and Weight Constraint

The most successful dropout rate and weight constraint were tested together in a grid search. Dropout rate is the percentage that is passed to the dropout layers in each model and designate the rate at which neurons are dropped out of each layer. Weight Constraint is used together with dropout layers to give the neurons that remain after the dropout a certain classification weight. A higher weight means the remaining neurons have more influence on the ultimate classification. Dropout rate was tested at 10% intervals from 0-90%. Weight constraint is always an integer value and was tested at values 1-5.

Based on the results of the grid search, it can be determined that the best values to use for dropout rate and weight constraint are 30% and 3 respectively. Results showed an increase in accuracy as dropout rate was increased up to 30%. This shows that dropout can help reduce overfitting of a model to training data. However, after 30% dropout, there was a decline in accuracy, showing that dropping out too many neurons can restrict the network too much. This is again seen with the weight constraint. An increase in accuracy is seen with increasing weight until the weight of 3. After this point, there is a decline in accuracy, showing that too much weight on the remaining neurons will give too much influence on the classification, possibly resulting in overfitting.



Class Weight

Class weight is a multiplier that is used to produce an equally balanced data set. For example, if a data set is heavily unbalanced, adding the class weight multiplier will make the underrepresented class carry more decisive weight than the overrepresented class. For the purposes of this grid search, the class weight parameter was used as a boolean, tests were completed with or without the calculated class weight to balance the data set.

One grid search was completed without dropout, but using the class weight multiplier. This produced very poor results. The average accuracy for all models with this configuration was approximately 65%. This result encouraged us to repeat the test, using the best dropout, as found in a previous grid search in addition to the class weight. This was done to hopefully reduce any overfitting that could have occurred during the previous test. The average accuracy achieved for the test using class weights and dropout was approximately 92.9%. This result is much better than the 65% result in the previous test without dropout in use, however this result is also lower than results produced by the other grid searches. Because this result is lower than the other observed results, we concluded that it would be preferable to not include class weight in the final parameter list.

Additionally, the data set that we are using has an approximate ratio of 60% benign to 40% malicious data points. This is not heavily imbalanced. Therefore the class weight multiplier will not be a very significant multiplier. A more reasonable use of the class weight parameter would be if a data set is more heavily imbalanced; for example, a 80% to 20% ratio would produce a more significant multiplier.

Another consideration for not using class weight parameter in the final parameter list, is the actual unbalance in benign to malicious applications in a real world environment. Google claims that over 99% of android devices were free of malicious applications as of 2015 (Lawal, 2017). This leaves less than 1% of devices to be infected with malware. In this case, if apps from all the android devices in the world were used as data points, a very small percentage of those applications would be classified as malicious. Our data set adequately represents the worst case scenario of relative equality in representation while also taking into consideration that there are more benign apps than there are malicious.

Final Cross Validation

To perform the best case test, the best parameters from each of the previously described searches were combined. The utilized parameters for this test were: batch size of 10, 16 epochs, 45 neurons, Nadam optimizer, dropout rate of 30%, weight constraint of 3, and no class weight. These parameters were used in tests of training percentages, 20%, 40%, 60%, and 80%. A random percentage, the size of the training percentage, was used for training each time. A five-fold cross validation was performed to ensure the validity of the results. A Stratified Shuffle Split was used to mix the data and ensure that a different portion of the data was used for each fold of cross validation.

For scoring this test, accuracy along with the F1 score were considered. Accuracy is the standard measure of correct classification percentage. F1 score is a standard weighted average of precision and recall given the formula: where and . The best F1 scores are closer to 1 and the worst F1 scores are closer to 0. Though more concerned with accuracy, we can fully determine the effectiveness of the network classification by also considering the F1 metric.

The results of the test showed that at all training ratios, the four layers of decreasing size model performed the best. This model ranged from a low average accuracy of 94.38% and an average F1 score of 0.920 at 20% training to a high average accuracy of 95.04% and an average F1 score of 0.928 at 80% training.

The four layers of the same size model had the worst accuracy, but often the best F1 score. This model ranged from a low average accuracy of 94.15% and an average F1 score of 0.918 at 20% training to a high average accuracy of 94.77% and an average F1 score of 0.926 at 80% training. The accuracies for this model were consistently lower than the other two models. At the 20% training ratio, the F1 score was the lowest of the models. However, the F1 scores for this model were ranked the best out of the three for the 40%, 60% and 80% training ratios.

The model that improved the most by increasing the training ratio was the one layer model. This model ranged from a low average accuracy of 94.22% and an average F1 score of 0.918 at 20% training to a high average accuracy of 95.02% and an average F1 score of 0.930at 80% training. This was the largest increase in accuracy that was seen by increasing the training ratio from 20% to 80%. At the 80% ratio, accuracy was the second best of the models tested and the F1 score was the best of all tested ratios and models.

In terms of time, all models performed similarly. The fastest time to train was seen with the one layer model which took 637 seconds to train at the 20% training percentage. The other models were within 30 seconds of that time at the same percentage. The longest time to train a model was seen with the one layer model which took 2461 seconds to train at the 80% training percentage. The other models were within 50 seconds of that time and the same percentage. Based on these statistics, it can be seen that the training ratio and time increase is directly related in a 1 to 1 manner. A double in the training ratio will lead to a double in the amount of time taken to train the model.

Based on these results, the best model to use for classification with this data set is the four layers of decreasing size model. At all training percentages, this model outperformed the other two when considering accuracy. By also considering the F1 score, this model was the best performer at the 20% training ratio, and the second best performer at all other ratios.

Overall, with all models, it is possible to utilize the Keras Library for the TensorFlow framework in order to do permissions based android malware detection with at least 94% accuracy at a 20% training ratio. By increasing the training ratio, the analysis will be able to achieve higher accuracy, though at the cost of a much higher time taken for evaluation. By tuning hyperparameters of the deep learning infrastructure we were able to optimize the performance.

Lawal, Elijah. “Shielding you from Potentially Harmful Applications.” Google, Google, 16 Feb. 2017, blog.google/topics/safety-security/shielding-you-potentially-harmful-applications/.