Deep Learning for Android Malware Detection:

A Grid Search Approach

Jarrett Booz, Josh McGiff, Chao Lu, Wei Yu

Towson University

### Abstract

As concerns around mobile malware continue to grow, new systems for malicious application detection must be implemented. These systems must be tuned to the optimal settings to achieve the highest performance at all times. Maximum performance is needed to ensure the highest detection accuracy and fastest detection speeds, thus offering the best possible protection against malicious applications that may already be in use. In our study, we implement a deep learning environment on a single node and fine tune many parameters to determine the optimal settings to use in Android malware classification based on permissions data. By determining the optimal settings, we are able to demonstrate the potential performance of a deep learning environment for Android malware detection.

# 1. Introduction

Machine learning has been helping to solve computational problems for some time now. As advancements are continuing, deep learning is beginning to emerge as a way to process highly complex data [lecun2015deep]. Many current uses of deep learning are for image recognition, however, deep learning can be used in cyber security situations as well.

The initial question this paper asked was if deep learning would be a viable method for analyzing permissions data from Android applications in order to classify apps as malicious or benign. To solve this, we used several different python libraries to build a deep learning infrastructure that would extract and vectorize features from a text file, then use these dense vectors as matrices for deep learning analysis. This proved to be possible, and yielded results of about 90% accuracy.

Once this initial question was answered affirmatively, we began asking how we could optimize the results. We began using a grid search technique, a sort of brute force method, to test many combinations of tunable parameters for the deep learning environment. We also experimented with several model shapes, making some networks deeper or wider than others to determine the best shape and size of the model. The results showed that by tuning 6 different parameters, we were able to increase the accuracy of the classification network by 5%, for a maximum accuracy of approximately 95% correct classifications.

The remainder of this paper is as follows: Section 2 describes background information pertaining to deep learning, mobile malware, and our testing environment. Section 3 describes our specific approach to the research questions including the basic idea, the system workflow and the parameters and scenarios that were tested. Section 4 provides an overall performance evaluation with the evaluation setup and results. Section 5 discusses future considerations for our work. Section 6 describes related research in similar areas. Section 7 provides a summary and conclusion.

# 2. Background

In this section we will provide some background details on the project. Included information provides details about deep learning, mobile malware, and our specific testbed environment.

## 2.1 Deep Learning

Deep learning is often the term used to describe the function of neural networks that feature multiple hidden layers to process their data. Though many differing definitions have been given to describe deep learning, two common elements are: i) the presence of multiple, non-linear, layers for data processing. ii) The notion of processing data at an increasingly higher level and abstract manner throughout the network [deng2014deep]. Neural networks, and more specifically, deep neural networks, have emerged in recent years as highly effective tools when approaching machine learning problems.

The mechanics behind neural networks and deep learning techniques have their roots in mathematics as far back as the 1940’s, technological advances have led to a resurgence in the field, yielding great success when applying deep learning techniques to complex problems such as image recognition and natural language processing [schmidhuber2015deep].

## 2.2 Malware in Mobile

Android continues to dominate the global market for mobile operating systems, holding what is estimated to be as high as 87% of the global market share [SA2016global]. This dominance has made Android a priority target for malware developers. According to McAfee’s quarterly threats report for Q1 2017 mobile malware injection rates have risen to a recent high of nearly 13% of mobile users affected [cruz2014mcafee]. Sophos notes a trend towards malicious adware as well as Android ransomware in their 2017 forecast, based on trends noted during 2016 [sophos2017].

Current trends in Android malware development include increasingly nuanced means of evading current malware detection tools on both devices and in the Android marketplace. One such technique is the repackaging of legitimate applications with malicious code or by utilizing dynamically loaded code to load in malware [lindorfer2014andrubis]

As malware strains continue to evolve and develop invasive techniques there is a clear and present need to keep pace in malware analysis and identification.

## 2.3 Testbed Overview

This section will describe the testbed environment that was used in our scenario. The testbed included a TensorFlow backend, a Keras wrapper for TensorFlow, Scikit-Learn utilities, and a feature extraction phase that was carried out with Scikit-Learn.

***TensorFlow***

TensorFlow is a machine learning system that was designed by Google for experimenting with learning models and training on large datasets. TensorFlow is able to be run on a large scale with multiple, often GPU-enabled, servers, or on a small scale, such as a mobile device [abadi2016tensorflow]. In our testbed, TensorFlow was run on a single server using only CPU power. The server had 8 cores available for TensorFlow computation.

TensorFlow was compiled and installed from source, as described in the documentation [abadi2016tensorflow]. Compiling and installing from source allowed the software to identify the make and model of the CPU in use and optimize performance based on the CPU specs.

***Keras***

Keras is a python library that serves as a sort of wrapper for various deep learning infrastructures. According to the Keras website, “Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. Being able to go from idea to result with the least possible delay is key to doing good research.”[chollet2017keras].

Keras was used to make writing the code to build various models faster and easier to implement. In many cases, building a model in Keras can be done with very few lines of code, whereas in a native TensorFlow environment, much more code would be needed. This library helped to simplify the development process.

***Scikit-Learn***

Scikit-Learn is a python library that allows for easy use of well-known machine learning algorithms and other machine learning related tools. According to the creators, “Scikit-learn is a Python module integrating a wide range of state-of-the-art machine learning algorithms for medium-scale supervised and unsupervised problems. This package focuses on bringing machine learning to non-specialists using a general-purpose high-level language.” [SKL2011]. This library was used to provide a well-known implementation of several machine learning components.

***Feature Extraction***

Feature extraction deals with the process of transforming raw data into a format compatible with machine learning algorithms. From our raw dataset of android APK files the permissions information was extracted through the use of the *aapt* tool. The raw text output of this process contains information on declared, used, and optional permissions for each application.

Android permissions can exist in different contexts, namely, used and optional permissions. Permissions identified in the manifest with the string *uses-permission* are required permissions that must be granted for the application to function. In some contexts a permission will feature the *optional-permission* string indicating that the permission is optional and the application can function in its absence. For our purposes we recognize the distinction between these two contexts and subsequently view the same permission under each context as a unique feature.

The Scikit-learn library was utilized to transform this text data into feature matrices for use in training our neural networks. To perform the extraction we used the CountVectorizer module which can be used to build a matrix of token counts from a text input source. The vectorizer parses the input data according to a token defined to recognize android permissions strings, as defined through the following regular expression:

*(\\b(:?uses-|optional-)?permission:\sname=\'android.permission[^\s]\*)*

The above expression distinguishes between optional and used permissions, as well as those that are defined within the manifest. As the vectorizer parses through the input data it will build a vocabulary of unique tokens as they are encountered, appending a new column to the matrix each time a new token is identified. Each input source, in our case each line of data, is represented in a row of the token matrix.

Using the above expression on our entire dataset yields a sparse mxn matrix, denoting that across a data set of m android applications the vectorizer identified a vocabulary of n permissions and permission contexts. This matrix must then be converted into a dense matrix before being supplied to the neural network for training. For our particular dataset this yielded a 48643x22300 matrix. This matrix indicates that there was a total of 22,300 unique permissions in the 48,643 APK files that were parsed.

# 3. Our Approach

In this section we will overview the approach we took to optimizing deep learning results for mobile malware detection. We will discuss the basic idea and general approach, the system workflow, and the specific parameters and scenarios that were tested.

## 3.1 Basic Idea

***Grid Search***

Neural networks often have many hyper-parameters that are able to be initialized by the programmer or left at a default value. To find the hyper-parameter values that are best suited for this dataset and classification scenario, a grid search was used. The Scikit-Learn library implements a grid search function that was utilized here. 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.

Another consideration when working with neural networks is the network shape. One must ask how deep and now wide the network should be in order to produce the best results. To answer this question, three shapes of networks were tested in addition to the hyper-parameter testing via the grid search described earlier.

## 3.2 System Workflow

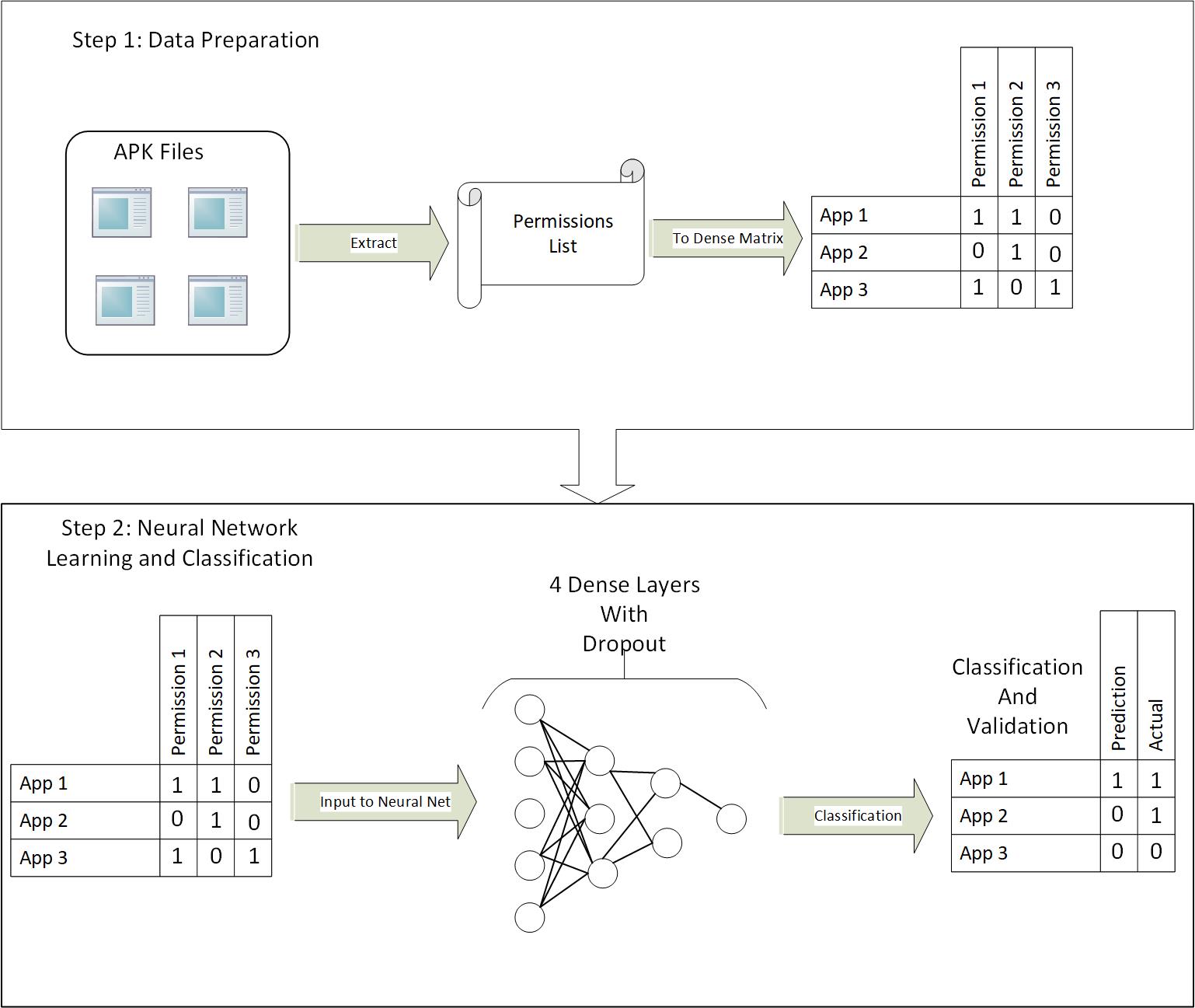
Because of the number of hyper-parameters that we wished to tune and the number of different models we wished to test, a traditional brute-force approach could have taken months of computation time. In order to circumvent this, hyper-parameters were tested in pairs. Each pair of hyper-parameters was used to train then test one network shape at a time at a 20% training data ratio. The 20% training ratio was used to reduce the amount of time taken to train the models as well as to provide the worst case scenario, where not a significant amount of data is available for training. After the completion of a grid search on one network design, the same grid search was completed on the remaining network designs.

The pairs of hyper-parameters that were tested together were as follows: epochs and batch size, number of neurons and optimizer, dropout rate and weight constraint, and class weight was tested alone.

As the result of each grid search was received, the next grid search was executed using the best performing parameters from the previous search. This was done to ensure that adding and tuning new parameters did, in fact, improve the accuracy of the system alongside the other parameters.

Each grid search was cross validated within the Scikit-Learn grid search implementation. For the final test of the best parameters a 5 fold cross validation was completed using a stratified shuffle split for choosing random data points to be included in the training set. This stratified shuffle split allowed for each fold of cross validation to be performed using a different subset of the data as the training sample portion. This is significant to note because, based on this method of cross validation, the training data will not always have the same data points, nor will it always have the same ratio of malicious to benign samples. By implementing cross validation in this manner, we can be sure that the results are consistent and not based simply on random chance or careful selection of training/testing samples.

Figure 1 displays the overall workflow of the system. In Step 1 we must prepare the data for input to the neural network. In this phase we utilize feature extraction to extract the permissions data from the APK files. This leaves us with a general list of permissions in each application. Next, we must convert this list to a dense matrix using Scikit-Learn as described in Section 2.3 of this paper. Step 2 of the process is where we use the dense matrix as input the neural network for learning and classification. In Figure 1 a 4 dense layer neural network with dropout. In a traditional dense layer configuration, all nodes in a previous layer are connected to all nodes in the next layer, however, because the network displayed is utilizing dropout, some nodes are disconnected from the next layer. This is further explained in Section 3.3 of this paper. After completion of the neural network training and testing, we are left with a matrix of all the applications with their predicted and actual classifications. This can then be used to determine accuracy and F1 score as described in Section 4.1 of this paper.



#### Figure 1: System Workflow

## 3.3 System Parameters and Scenarios

***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.

A sample neural network utilizing dense layers with dropout between can be seen in Figure 1.

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 of neurons divided the depth of the layer (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.

***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.

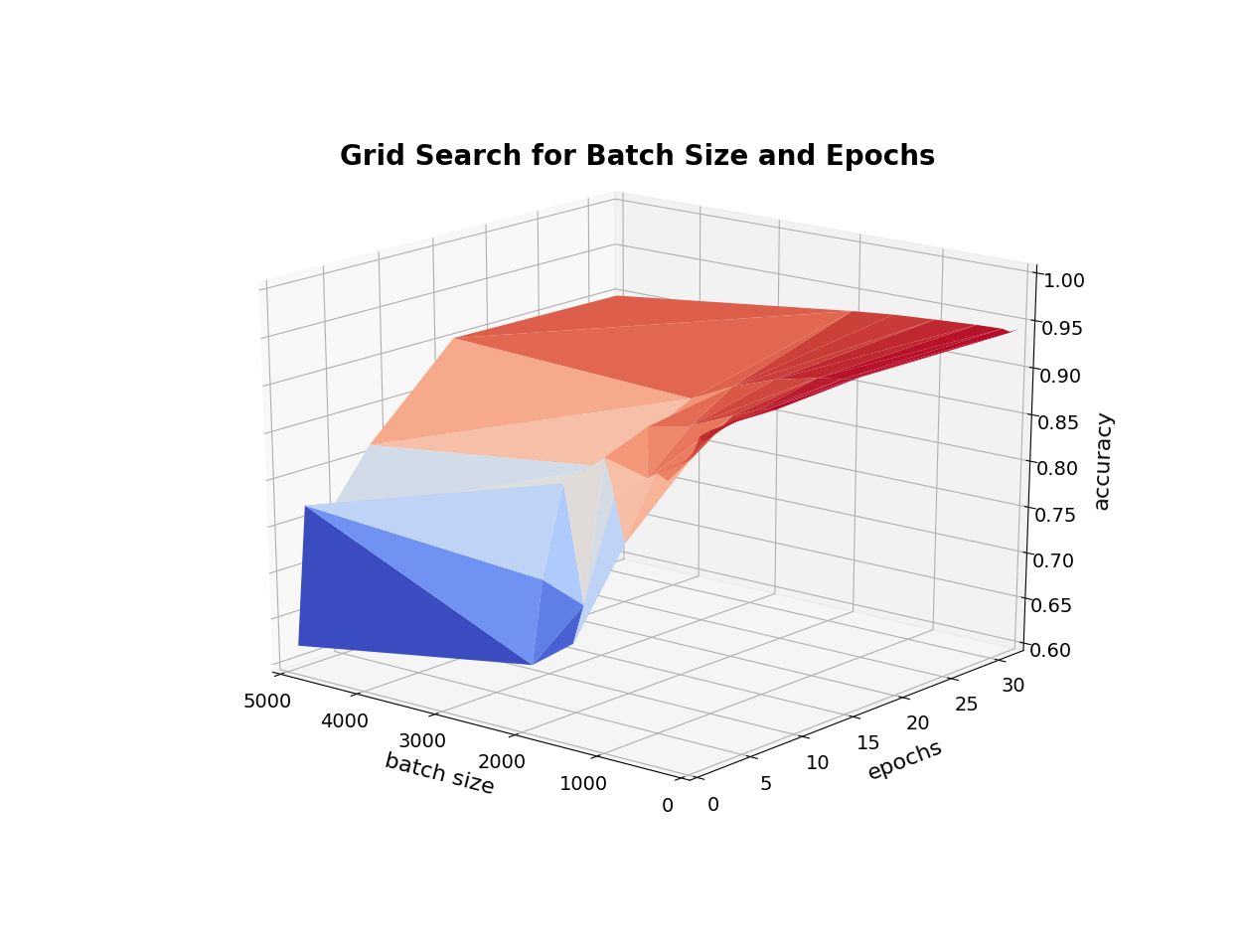
In most cases, as the number of epochs was increased, accuracy increased as well. Although, in some cases it was noticed that accuracy had a slight decline after 16 epochs. This could be due to the model overfitting itself to the specific features of the training data and therefore performing slightly 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. Figure 2 displays that increasing batch size and epochs tend to lead to higher accuracy. Though, a small batch size does not see as significant of an improvement by increasing epochs when comparing to a larger batch size. It can be seen in Figure 2 that a small batch tends to achieve slightly 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, as displayed by Figure 2.

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.



#### Figure 2: Batch Size & Epochs v. Accuracy

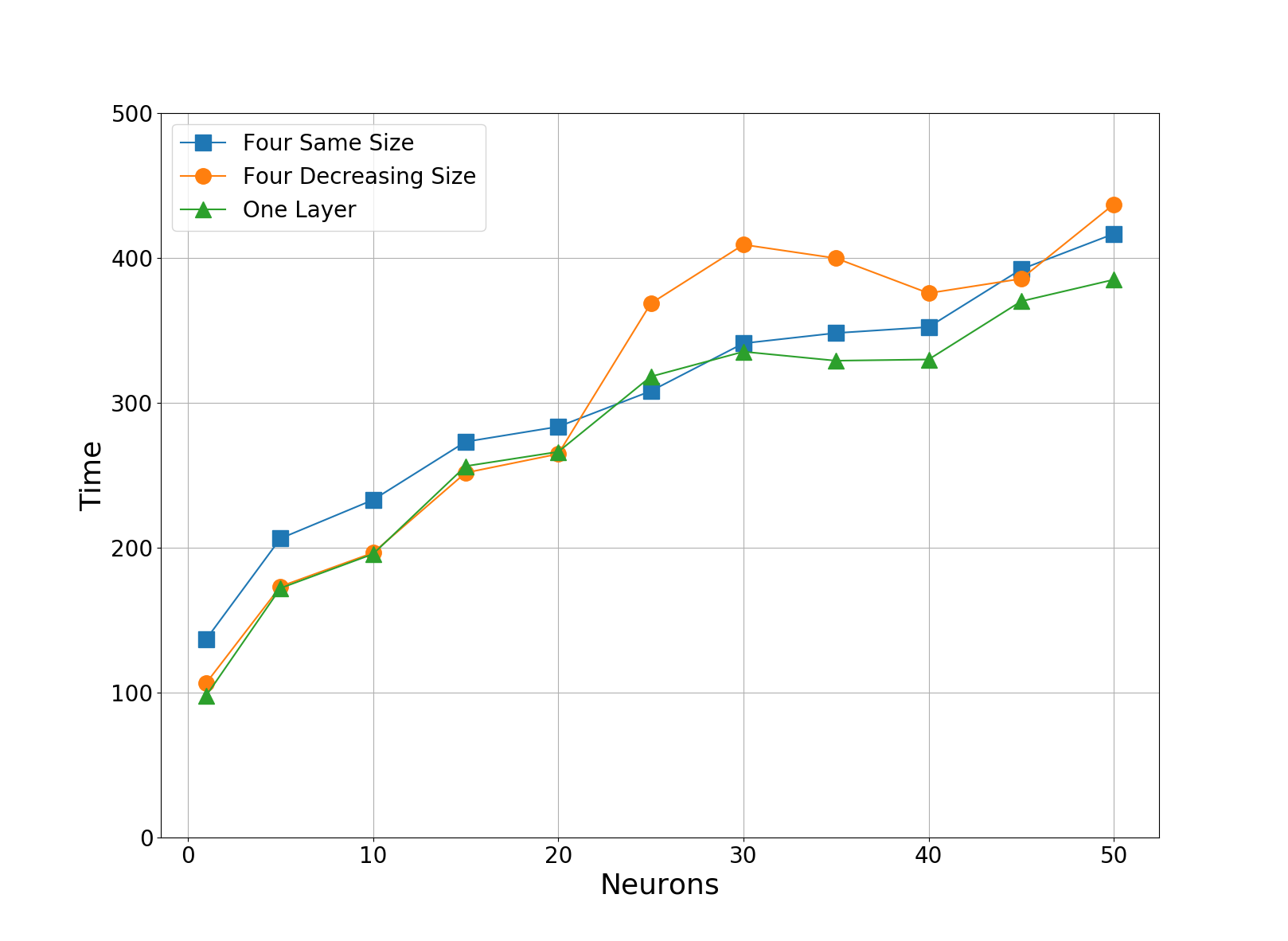
***Number of Neurons***

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 assigned 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 and also took a longer amount of time to train.

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, as displayed in Figure 3, 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.



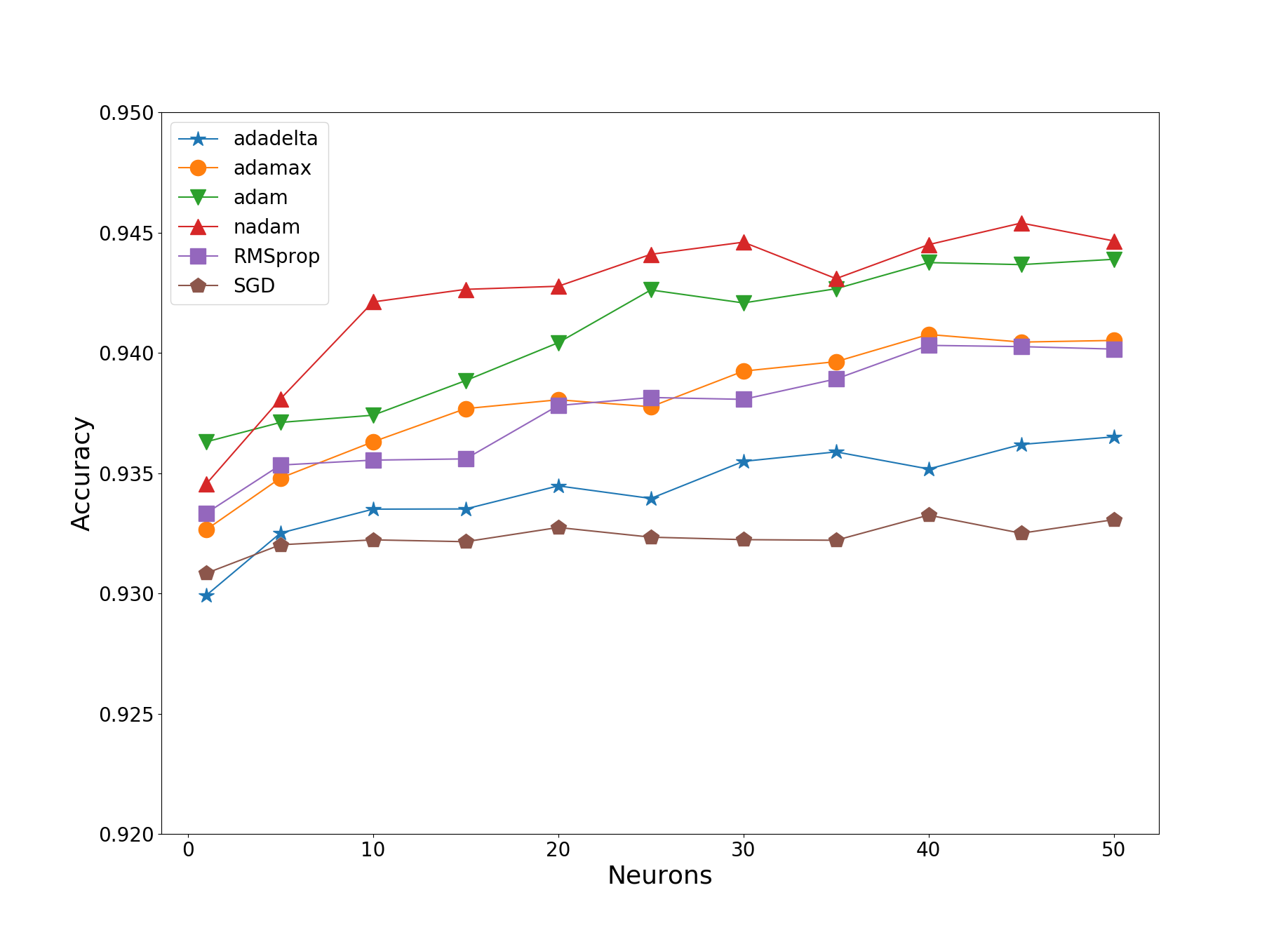
#### Figure 3: Number of Neurons v. Time

***Optimizer***

Several optimizers were chosen based on compatibility with the Keras wrapping infrastructure that we were 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 approximately 2% of the best performer. Figure 4 shows the performance of many optimizers with tests using different numbers of neurons. The best result, as displayed by Figure 4, was the Nadam optimizer when using 45 neurons.

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.

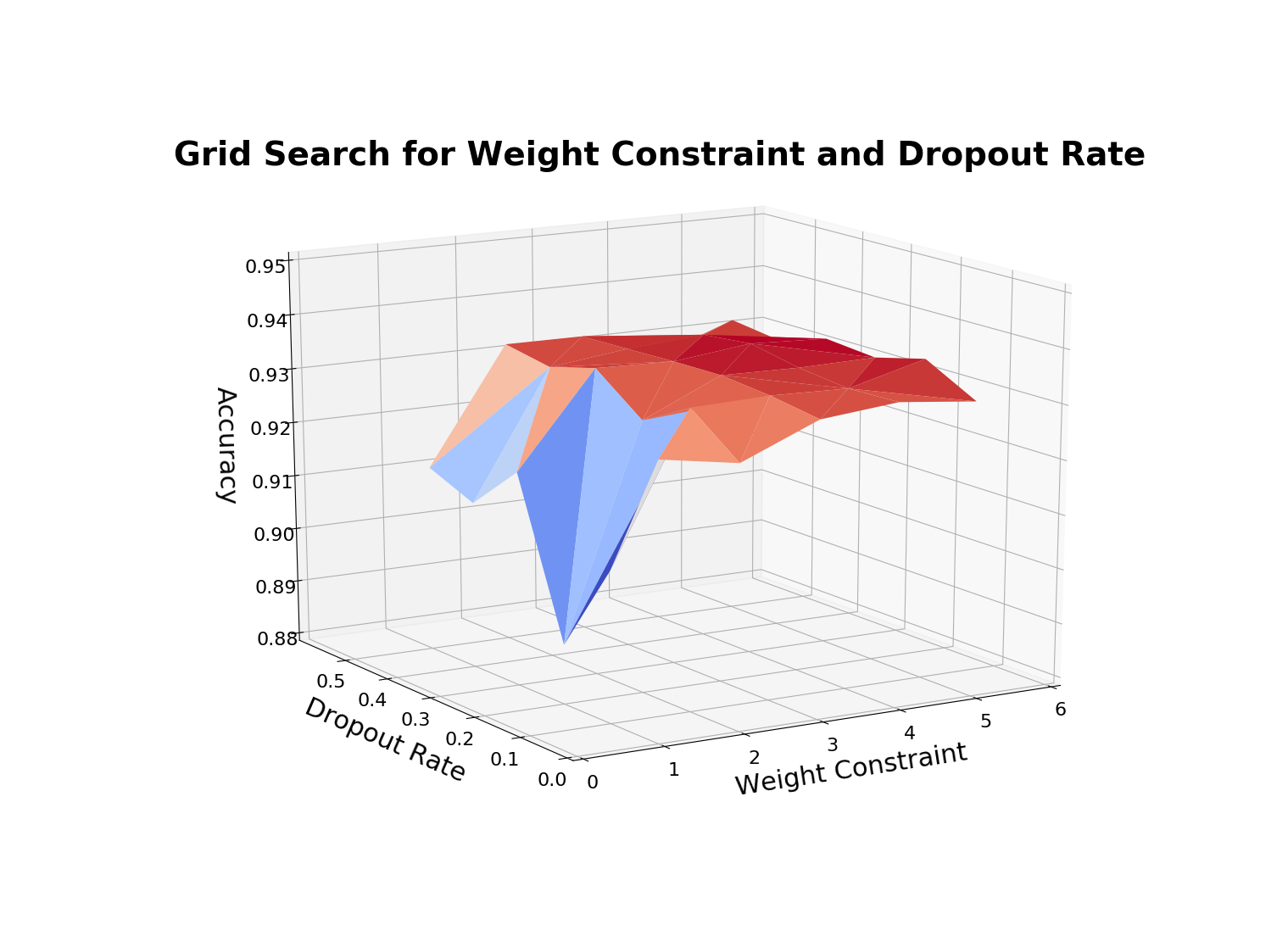


#### Figure 4: Number of Neurons v. Accuracy for Many Optimizers

***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 can be seen by the dark coloring of the graph in Figure 5 at the 0.3 dropout rate marker. It is seen in the figure that the surrounding dropout rates are lighter in color, indicating that they are less accurate. 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. Figure 5 displays that utilizing appropriate numbers for both weight constraint and dropout rate can lead to significant increases in accuracy.



#### Figure 5: Dropout and Weight Constraint v. Accuracy

***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 [googleBlog]. 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.

# 4. Performance Evaluation

This section will detail the evaluation of the experiment as a whole. We explain the evaluation setup as well as the final evaluation results.

## 4.1 Evaluation Setup

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.

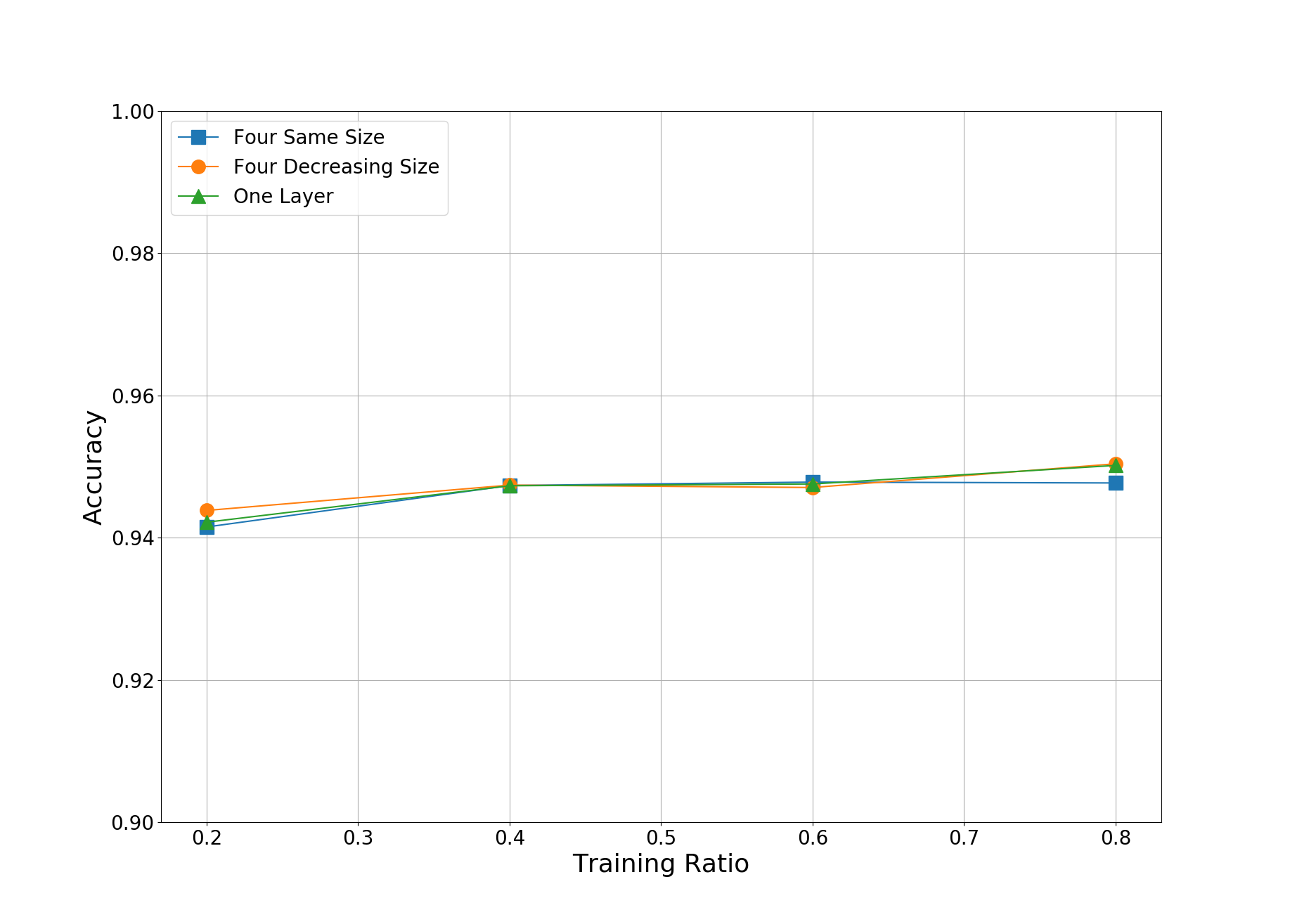
For scoring this test, accuracy along with the F1 score were considered. Accuracy is the standard measure of correct classification percentage. F1 score is a standard weighted average of precision and recall given the formula: where and . The best F1 scores are closer to 1 and the worst F1 scores are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

## 4.2 Evaluation Results

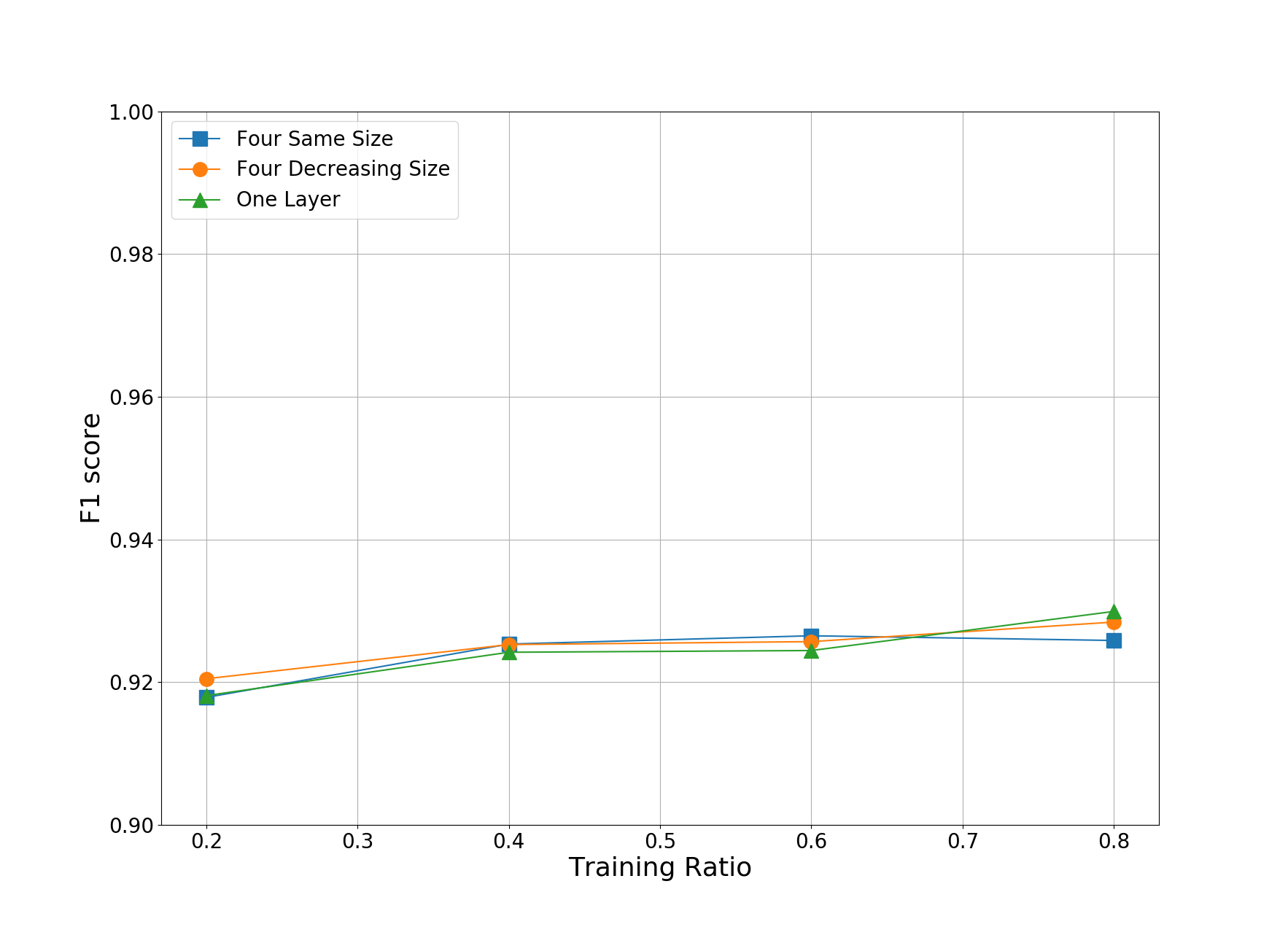
The results of the test showed that at all training ratios, the four layers of decreasing size model performed the best. This model ranged from a low average accuracy of 94.38% and an average F1 score of 0.920 at 20% training to a high average accuracy of 95.04% and an average F1 score of 0.928 at 80% training. Figure 6 and Figure 7 can be used together to display these results.

The four layers of the same size model had the worst accuracy, but often the best F1 score, as shown by Figures 6 and 7. 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, as seen in Figure 7.

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.

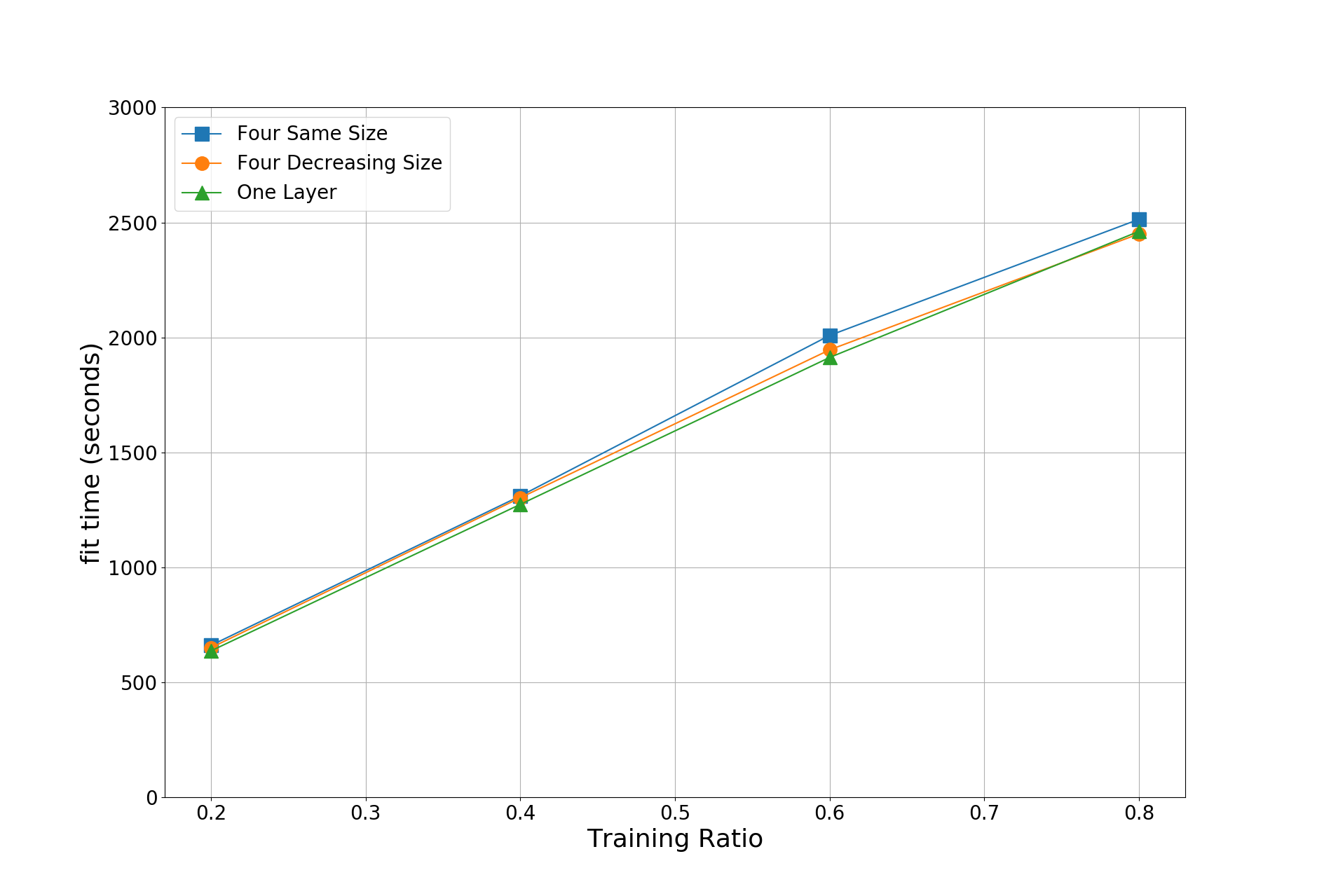


#### Figure 6: Training Ratio v. Accuracy for 3 Models



#### Figure 7: Training Ratio v. F1 Score for 3 Models

In terms of time, all models performed similarly. The fastest time to train was seen with the one layer model which took an average of 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 an average of 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 training ratio and time increase is probably linearly related. Therefore, 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. By tuning hyperparameters of the deep learning infrastructure we were able to optimize the performance.

# 5. Discussion

Some considerations should be made in order to further this research. Considerations can include adversarial training, scalability, other frameworks, and other types of data.

***Adversarial Training:*** One way to train data for machine learning is adversarial training. In a cyber security environment, adversaries may attempt to deliberately circumvent detection systems. Machine learning must provide accurate detection despite the adversaries’ attempts at avoidance [laskov2010machine]. Some strategies exist to employ methods of adversarial learning. Some ways to introduce this tactic are data manipulation to intentionally alter supervised learning classifications and to provide a worst-case scenario in terms of ratio of malicious to benign samples [laskov2010machine]. In relation to this, we plan to implement an adversarial environment by intentionally altering supervised classifications. By having some malicious applications labeled as benign during the supervised training phase, we will be simulating an environment where the applications generally assumed to be benign are actually malicious in nature.

***System Scalability:*** Another consideration is the scalability of the neural network approach. In this paper, we examined neural networks operating on a single server with only CPU power. To provide faster computation, one may consider using a GPU instead of a CPU, which can boost performance, in many cases, by as much as 50% [schmidhuber2015deep]. Though this increase in speed is known, it is also known that computation can slow down if the model does not fit in GPU memory. This can be avoided by using a cluster of GPU enabled machines [dean2012large]. To address the scalability problem, we plan to implement a cloud environment to utilize distributed computing. By utilizing the cloud environment built in a previous project [hatcher2017], we will continue investigation in distributed computing for an application in deep learning malware detection.

***Other toolkits:***  Additionally, other neural network tools may be implemented to solve a similar problem. For example, the Keras framework utilized in this paper, can also be used with CNTK and Theano in the backend [chollet2017keras]. It has been shown that various open-source products may produce different results resulting in higher or lower accuracy and/or more or less time efficiency [gao2016comparison], therefore, using one of these different projects in the backend may change the results. For further results we will install other deep learning architectures that are interchangeable with the current system workflow, as described prior. By making a comparison between several deep learning technologies, we will be able to determine the be performer for our data and detection scenario.

***Additional Data Sets:*** Other types of data could be used in this network environment as well. For example, network traffic data can be used in malware classification [wang2017malware]. Additionally, techniques can be employed to visualize malicious files as images, leading to high accuracy in classification (Singh, 2017). Additional other features, data sets, or ways of interpreting data can be used to achieve different results. To further consider this, we plan to utilize a data set consisting of PCAP files to analyze network related information in relation to malicious software. By adding a new data set with additional features, we may be able to improve accuracy results and get some further insight into the behavior of mobile malware from a network perspective.

# 6. Related Work

There has been other research related to utilizing a deep learning approach to Android malware detection. Yuan, Zhenlong, et al. examined both static and dynamic features, for a total feature vector size of about 200 features, using a deep learning approach. Yuan, Zhenlong, et al. used various depths of neural networks with various amounts of neurons in each layer [yuan2014droid]

One drawback to the approach described in Yuan, Zhenlong, et al. is the limited size of the feature vector. With our approach, we were able to create a much larger feature vector by allowing Scikit-Learn to parse out all features, rather than select only certain features. This allows for more data to be used in classification calculations. This therefore lets the deep learning algorithms decide which features are most important rather than the programmer deciding which features to consider.

Yuan, Zhenlong, et al. used a much larger number of neurons in the tests they completed when compared to the number of neurons we used in our testing. It is possible that in our testing, we have reached a local maximum in performance at 45 neurons. Therefore, it is possible that we could have found higher absolute maximum had we increased the number of neurons further.

# 7. Final Remarks

In this paper we were able to successfully demonstrate the application of deep neural networks to permissions based Android malware classification. Distinct from similar research efforts, examine both core Android and user defined permissions, and distinguish between optional and required permissions, allowing us to establish a much larger feature vocabulary for classification. Utilizing the grid search method of hyper-parameter tuning we were able to optimize our networks for best possible performance in terms of accuracy, precision, and recall.

@article{googleBlog,  
 title={Shielding you from Potentially Harmful Applications},  
 author={Lawal, Elijah},  
 journal={URL blog.google/topics/safety-security/shielding-you-potentially-harmful-applications/.},  
 year={2017}  
}

@article{chollet2017keras,  
 title={Keras (2015)},  
 author={Chollet, Fran{\c{c}}ois},  
 journal={URL http://keras. io},  
 year={2017}  
}

@article{SKL2011,

title={Scikit-learn: Machine Learning in {P}ython},

author={Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V.

and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P.

and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and

Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E.},

journal={Journal of Machine Learning Research},

volume={12},

pages={2825--2830},

year={2011}

}

@inproceedings{yuan2014droid,  
 title={Droid-Sec: deep learning in android malware detection},  
 author={Yuan, Zhenlong and Lu, Yongqiang and Wang, Zhaoguo and Xue, Yibo},  
 booktitle={ACM SIGCOMM Computer Communication Review},  
 volume={44},  
 number={4},  
 pages={371--372},  
 year={2014},  
 organization={ACM}  
}

@article{schmidhuber2015deep,  
 title={Deep learning in neural networks: An overview},  
 author={Schmidhuber, J{\"u}rgen},  
 journal={Neural networks},  
 volume={61},  
 pages={85--117},  
 year={2015},  
 publisher={Elsevier}  
}

@article{lecun2015deep,  
 title={Deep learning},  
 author={LeCun, Yann and Bengio, Yoshua and Hinton, Geoffrey},  
 journal={Nature},  
 volume={521},  
 number={7553},  
 pages={436--444},  
 year={2015},  
 publisher={Nature Research}  
}

@article{laskov2010machine,  
 title={Machine learning in adversarial environments},  
 author={Laskov, Pavel and Lippmann, Richard},  
 journal={Machine learning},  
 volume={81},  
 number={2},  
 pages={115--119},  
 year={2010},  
 publisher={Springer}  
}

@inproceedings{dean2012large,  
 title={Large scale distributed deep networks},  
 author={Dean, Jeffrey and Corrado, Greg and Monga, Rajat and Chen, Kai and Devin, Matthieu and Mao, Mark and Senior, Andrew and Tucker, Paul and Yang, Ke and Le, Quoc V and others},  
 booktitle={Advances in neural information processing systems},  
 pages={1223--1231},  
 year={2012}  
}

article{gao2016comparison,  
 title={Comparison and Analysis of the Open-Source Frameworks for Deep Learning},  
 author={GAO, Dong-sheng and ZHAO, Yan-rong and Jing, GAO and Hao, WANG},  
 journal={DEStech Transactions on Computer Science and Engineering},  
 number={mcsse},  
 year={2016}  
}

@inproceedings{wang2017malware,  
 title={Malware traffic classification using convolutional neural network for representation learning},  
 author={Wang, Wei and Zhu, Ming and Zeng, Xuewen and Ye, Xiaozhou and Sheng, Yiqiang},  
 booktitle={Information Networking (ICOIN), 2017 International Conference on},  
 pages={712--717},  
 year={2017},  
 organization={IEEE}  
}

@phdthesis{singh2017malware,  
 title={Malware Classification using Image Representation},  
 author={Singh, Ajay},  
 year={2017},  
 school={INDIAN INSTITUTE OF TECHNOLOGY KANPUR}  
}

@inproceedings{hatcher2017,  
 title={Edge Computing Based Machine Learning Mobile Malware Detection},  
 author={William G. Hatcher, Jarrett Booz, Josh McGiff, Chao Lu, and Wei Yu},  
 year={2017},  
 organization={National Cyber Summit}  
}

@inproceedings{lindorfer2014andrubis,  
 title={Andrubis--1,000,000 apps later: A view on current Android malware behaviors},  
 author={Lindorfer, Martina and Neugschwandtner, Matthias and Weichselbaum, Lukas and Fratantonio, Yanick and Van Der Veen, Victor and Platzer, Christian},  
 booktitle={Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS), 2014 Third International Workshop on},  
 pages={3--17},  
 year={2014},  
 organization={IEEE}  
}

@article{cruz2014mcafee,  
 title={McAfee Labs threats report},  
 author={Cruz, B and Gupta, D and Kapoor, A and Haifei, L and McLean, D and Moreno, F and others},  
 journal={McAfee Inc., Santa Clara, CA. Available: http://www. mcafee. com/us/resources/reports/rp-quarterlythreat-q1-2014. pdf},  
 year={2014}  
}

@techreport{sophos2017,

title = {Looking ahead: SophosLabs 2017 malware forecast},

author = {SophosLabs},

group = {Sophos},

year = {2017},

month = {02},

}

@article{cruz2014mcafee,  
 title={McAfee Labs threats report},  
 author={Cruz, B and Gupta, D and Kapoor, A and Haifei, L and McLean, D and Moreno, F and others},  
 journal={McAfee Inc., Santa Clara, CA. Available: http://www. mcafee. com/us/resources/reports/rp-quarterlythreat-q1-2014. pdf},  
 year={2014}  
}

@article{deng2014deep,  
 title={Deep learning: methods and applications},  
 author={Deng, Li and Yu, Dong and others},  
 journal={Foundations and Trends{\textregistered} in Signal Processing},  
 volume={7},  
 number={3--4},  
 pages={197--387},  
 year={2014},  
 publisher={Now Publishers, Inc.}  
}

@article{abadi2016tensorflow,  
 title={Tensorflow: Large-scale machine learning on heterogeneous distributed systems},  
 author={Abadi, Mart{\'\i}n and Agarwal, Ashish and Barham, Paul and Brevdo, Eugene and Chen, Zhifeng and Citro, Craig and Corrado, Greg S and Davis, Andy and Dean, Jeffrey and Devin, Matthieu and others},  
 journal={arXiv preprint arXiv:1603.04467},  
 year={2016}  
}

@misc{SA2016global,

title={Global smartphone OS market share by region: Q3 2016},

author={Linda Sui},

year={2016}

}

***MLA Citations (For Reference)***

Chollet, François. "Keras (2015)." *URL http://keras. io* (2017).

Scikit-learn: Machine Learning in Python, JMLR 12, pp. 2825-2830, 2011.

Yuan, Zhenlong, et al. "Droid-Sec: deep learning in android malware detection." *ACM SIGCOMM Computer Communication Review*. Vol. 44. No. 4. ACM, 2014.

Schmidhuber, Jürgen. "Deep learning in neural networks: An overview." *Neural networks* 61 (2015): 85-117.

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

LeCun, Yann, Yoshua Bengio, and Geoffrey Hinton. "Deep learning." *Nature* 521.7553 (2015): 436-444.

Laskov, Pavel, and Richard Lippmann. "Machine learning in adversarial environments." *Machine learning* 81.2 (2010): 115-119.

Dean, Jeffrey, et al. "Large scale distributed deep networks." *Advances in neural information processing systems*. 2012.

GAO, Dong-sheng, et al. "Comparison and Analysis of the Open-Source Frameworks for Deep Learning." *DEStech Transactions on Computer Science and Engineering* mcsse (2016).

Wang, Wei, et al. "Malware traffic classification using convolutional neural network for representation learning." *Information Networking (ICOIN), 2017 International Conference on*. IEEE, 2017.

Singh, Ajay. *Malware Classification using Image Representation*. Diss. INDIAN INSTITUTE OF TECHNOLOGY KANPUR, 2017

William G. Hatcher, Jarrett Booz, Josh McGiff, Chao Lu, and Wei Yu, “Edge Computing Based Machine Learning Mobile Malware Detection,” in Proceedings of National Cyber Summit (NCS), June 2017, AL, USA.

Lindorfer, Martina, et al. “ANDRUBIS - 1,000,000 Apps Later: A View on Current Android Malware Behaviors.” 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security, Sept. 2014, doi:10.1109/BADGERS.2014.7 .

McAfee Labs, “McAfee Labs Threats Report” https://www.mcafee.com/us/resources/reports/rp-quarterly-threats-jun-2017.pdf, June 2017.

Li Deng and Dong Yu (2014), "Deep Learning: Methods and Applications", Foundations and Trends® in Signal Processing: Vol. 7: No. 3–4, pp 197-387. <http://dx.doi.org/10.1561/2000000039>

“Looking ahead: SophosLabs 2017 malware forecast .” sophos.com, Feb. 2017, [www.sophos.com/en-us/medialibrary/PDFs/technical-papers/sophoslabs-2017-malware-forecast-report.pdf?la=en](http://www.sophos.com/en-us/medialibrary/PDFs/technical-papers/sophoslabs-2017-malware-forecast-report.pdf?la=en).

Abadi, Martın, et al. "TensorFlow: large-scale machine learning on heterogeneous systems, Software available from tensorflow.org, 2015." *Available: http://tensorflow. Org*.