October 19, 2021

Predicting Anomalies in Network Traffic

Phase-1 : Data Analysis and Preparation
Phase-2 : Build a model to overfit the dataset
Phase-3 : Model selection and evaluation

Reshma Priya, Manne Muddu
AI 5300
University of Missouri, St Louis, Fall 2021

Contents

1 Introduction

1.1 Intrusion Detection Systems (IDS)

Intrusion Detection Systems (IDS) provide network security software applications by continuously monitoring network traffic and classifying the connections as normal or malicious. IDS are categorized into two types based on the responsive nature - Passive IDS and Active IDS. A passive IDS is designed to identify and block the malicious and malware attacks manually by human experts whereas active IDS is designed to identify and block the malware attacks using a software automatically.

IDS are also categorized into Signature based IDS and Anomaly Based IDS. In the Signature based IDS, there exists a database which contains details about all known malware attacks against which each network traffic connection is validated to identify the malicious nature if it exists. This type of IDS is costly and has to keep updating new types of attacks frequently. Anomaly based IDS is a behavior-based system where any deviation from the normal network traffic patterns will be reported using pattern-recognition techniques. In this research paper, a proof-of-concept for a machine learning based Anomaly based intrusion detection will be evaluated on a benchmark intrusion detection dataset.

1.2 Motivation

The Internet of Things (IoT) has changed the way devices communicate across a network. There are over 30 billion devices connected to each other in today's world where their communication happens over the network. Due to the variety, velocity and vast nature of this traffic, the traditional intrusion detection systems are not sufficient to identify the malware patterns [1]. With the help of data science, machine learning and neural network techniques, one can leverage the historic network traffic patterns and can build an intelligent artificial intelligence model to solve this intrusion detection problem. This project exactly tries to provide a proof-of-concept for this problem by applying machine learning and neural network techniques on a benchmark network intrusion detection dataset.

1.3 Problem Statement

In this research, KDD Cup 1999 dataset will be used to build a machine learning based intrusion detection problem to predict (classify) whether a network connection is "normal" or "abnormal". As this is a categorical prediction, this is a binary classification problem

Github Link to this project:click here
Jupyter notebook html: click here

2 Dataset Description and Exploration

In this research, the KDD Cup 1999 dataset has been chosen for the data analysis and model building. This dataset was used in the fifth international conference for the data mining and knowledge discovery competition and contains a huge variety of network intrusion data that is simulated in an environment equipped with a military network setting.

The original source of the dataset is from the official KDD website [2] and can also be found in the kaggle website [3]. The dataset contains about 494,000 records and 41 features (columns) which contains network traffic details like source bytes, destination connection information, type of attacks and so on. Below are the list of features (columns) and their data types in the dataset.

2.1 Target Variable "label" distribution

The target variable "label" contains all different types of malware attacks and also the value "normal" (i.e., not an attack) as shown below.

'normal', 'buffer overflow', 'loadmodule', 'perl', 'neptune', 'smurf', 'guess passwd', 'pod', 'teardrop', 'portsweep', 'ipsweep', 'land', 'ftp write', 'back', 'imap', 'satan', 'phf', 'nmap', 'multihop', 'warezmaster', 'warezclient', 'spy', 'rootkit'

All malware attacks are grouped (transformed) to the value "abnormal" to make this problem a binary classification problem instead of a multi-class classification. The target variable distribution can be found below (see **Figure 1**).

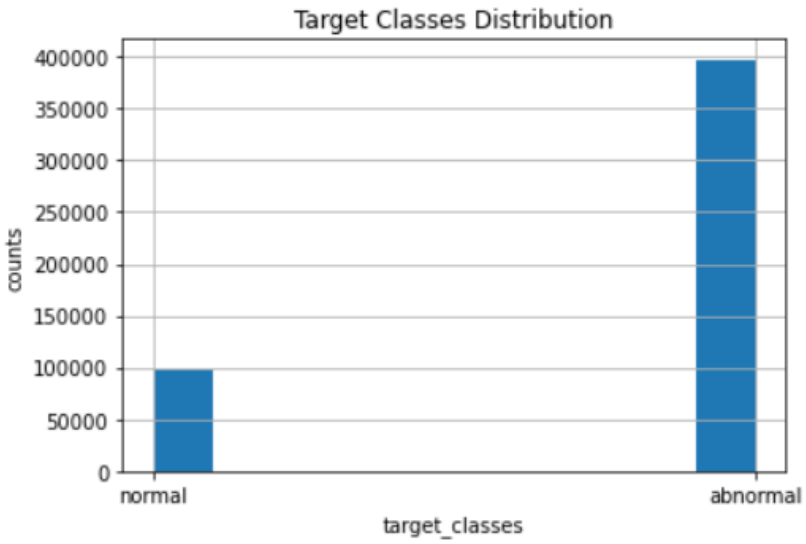


Figure 1: Target Variable "label" Distribution.

2.2 Categorical Columns Distributions

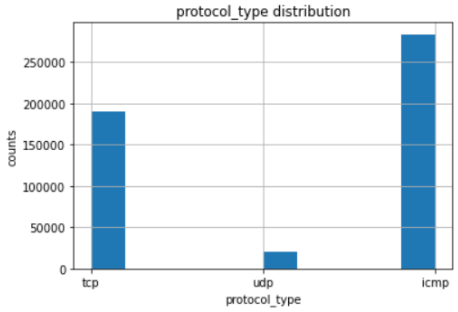
Below are the categorical columns in this dataset and their corresponding distribution plots.

'protocol type', 'service', 'flag', 'land', 'logged in', 'is host login', 'is guest login'

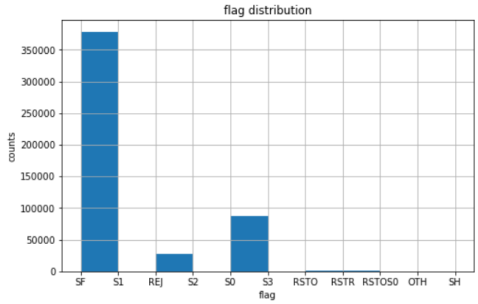
Categorical variables statistics can be found in the below table:

Table 1: Categorical Variables Statistics.

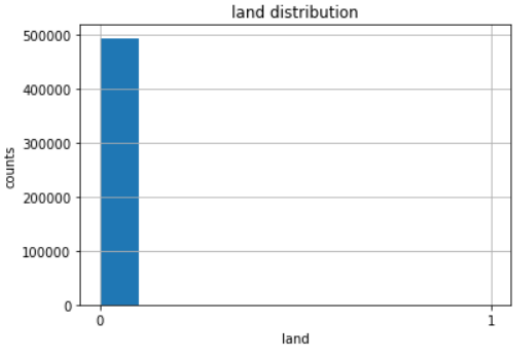
Stats	protocol_type	service	flag	land	logged_in	is_host_login	is_guest_login
count	494020	494020	494020	494020	494020	494020	494020
unique	3	66	11	2	2	1	2
top	icmp	ecr_i	SF	0	0	0	0
freq	283602	281400	378439	493998	420784	494020	493335



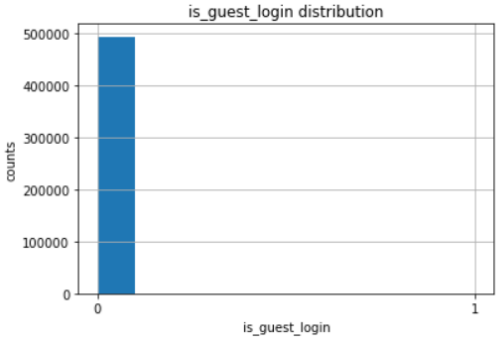
(a) protocol type



(b) Flag

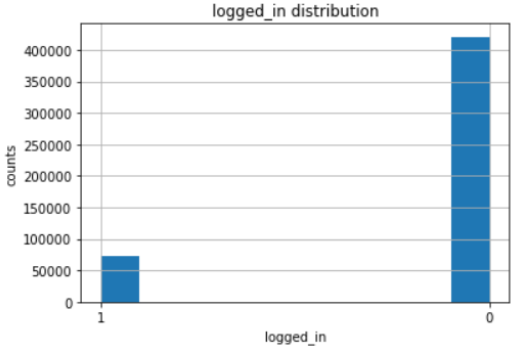


(c) Land



(d) Is Guest Login

(e) Is Host Login



(f) Logged In

Figure 2: Categorical Columns Distributions

2.3 Continuous Columns

Below are the continuous columns in this dataset and their corresponding distribution plots.

'duration', 'src bytes', 'dst bytes', 'wrong fragment', 'urgent', 'hot', 'num failed logins', 'num compromised', 'root shell', 'su attempted', 'num root', 'num file creations', 'num shells', 'num access files', 'num outbound cmds', 'count', 'srv count', 'serror rate', 'srv serror rate', 'rerror rate', 'srv rerror rate', 'same srv rate', 'diff srv rate', 'srv diff host rate', 'dst host count', 'dst host srv count', 'dst host same srv rate', 'dst host diff srv rate', 'dst host same src port rate', 'dst host srv diff host rate', 'dst host serror rate', 'dst host srv serror rate', 'dst host rerror rate', 'dst host srv rerror rate'

Continuous variables statistics can be found in the below table:

Table 2: Categorical Variables Statistics.

Stats	count	mean	std	min	25%	50%	75%	max
duration	494020	47.9794	707.7472	0	0	0	0	58329
src_bytes	494020	3025.616	988219.1	0	45	520	1032	6.93E+08
dst_bytes	494020	868.5308	33040.03	0	0	0	0	5155468
wrong_fragment	494020	0.006433	0.134805	0	0	0	0	3
urgent	494020	1.42E-05	0.00551	0	0	0	0	3
hot	494020	0.034519	0.782103	0	0	0	0	30
num_failed_logins	494020	0.000152	0.01552	0	0	0	0	5
num_compromised	494020	0.010212	1.798328	0	0	0	0	884
root_shell	494020	0.000111	0.010551	0	0	0	0	1
su_attempted	494020	3.64E-05	0.007793	0	0	0	0	2
num_root	494020	0.011352	2.01272	0	0	0	0	993
num_file_creations	494020	0.001083	0.096416	0	0	0	0	28
num_shells	494020	0.000109	0.01102	0	0	0	0	2
num_access_files	494020	0.001008	0.036482	0	0	0	0	8
num_outbound_cmds	494020	0	0	0	0	0	0	0
count	494020	332.2864	213.1471	0	117	510	511	511
srv_count	494020	292.9071	246.3227	0	10	510	511	511
serror_rate	494020	0.176687	0.380717	0	0	0	0	1
srv_serror_rate	494020	0.176609	0.381017	0	0	0	0	1
rerror_rate	494020	0.057434	0.231624	0	0	0	0	1
srv_rerror_rate	494020	0.057719	0.232147	0	0	0	0	1
same_srv_rate	494020	0.791547	0.38819	0	1	1	1	1
diff_srv_rate	494020	0.020982	0.082206	0	0	0	0	1
srv_diff_host_rate	494020	0.028996	0.142397	0	0	0	0	1
dst_host_count	494020	232.4712	64.7446	0	255	255	255	255
dst_host_srv_count	494020	188.6661	106.0402	0	46	255	255	255
dst_host_same_srv_rate	494020	0.753781	0.41078	0	0.41	1	1	1
dst_host_diff_srv_rate	494020	0.030906	0.109259	0	0	0	0.04	1
dst_host_same_src_port_rate	494020	0.601936	0.481309	0	0	1	1	1
dst_host_srv_diff_host_rate	494020	0.006684	0.042133	0	0	0	0	1
dst_host_serror_rate	494020	0.176754	0.380593	0	0	0	0	1
dst_host_srv_serror_rate	494020	0.176443	0.38092	0	0	0	0	1
dst_host_rerror_rate	494020	0.058118	0.23059	0	0	0	0	1
dst_host_srv_rerror_rate	494020	0.057412	0.230141	0	0	0	0	1

3 Data Normalization

As seen from the above continuous variables distribution in the above section the scale of some of the features (columns) values are not in same range as others. This irregularity will make the machine learning model train poorly. Hence the features need to be normalized so that the optimization during model training will happen in a better way and the model will not be sensitive to the features. This research paper uses the existing normalization function that exists in python's scikit-learn library which can be found here

After applying normalization transformation to each continuous feature, below are the new distribution plots.

4 Modeling

4.1 Model that over-fits the entire dataset

To understand the dataset and to estimate the model size (architecture and hyper-parameters) , we decided to build a model that over-fits the entire dataset. This is an experiment step to understand what model size would be required to overfit entire data. We performed below steps:

- Step-1: Using ENTIRE dataset to "OVERFIT" the model using vannila (single neuron) logistic regression model to reach accuracy 100 percent or close to 100 percent

- Step-2: If the accuracy did not reach 100 percent, then a bigger architecture model will be designed and modeled to achieve accuracy of 100 percent or close to 100 percent.

4.1.1 Model-1 Architecture Summary Diagram

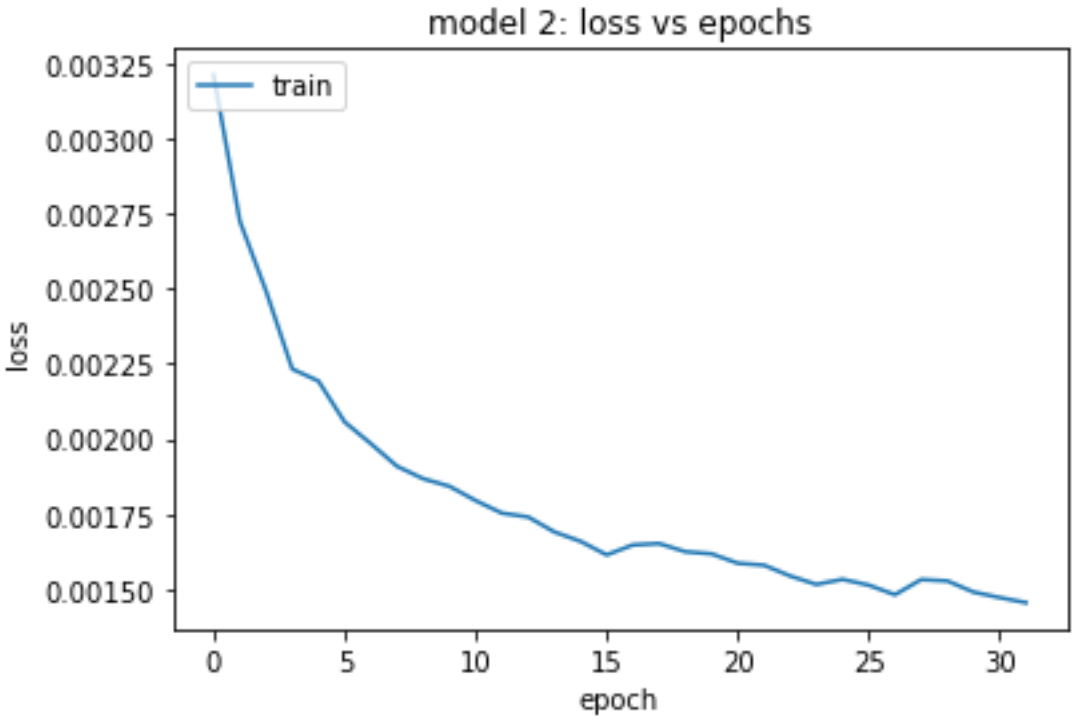
4.1.2 Inference

As seen from the above model, the basic logistic regression model has achieved an accuracy of 99.81 with 256 epochs.A seen from the model-2 performance we have achieved 99.96 percent accuracy (close to 100 percent) which means this model did over-fit the dataset. Therefore, a 4-layer neural network model architecture with 8 neurons, 4 neurons and 2 neurons in internal layers is sufficient enough to overfit the entire dataset.

5 Generalized Model building, selection and evaluation

In the previous section, after exploring the architectures that would result in an over-fitting model, in this section, the focus will be on using a feed-forward neural network (and its architecture variants) to build a best model that would perform well on the validation dataset. The data has been shuffled before performing the experiments and the same dataset has been used for all the models for fair comparison. The code did not implement seeding (which is used for ability to reproduce the model), hence a new run might result in a slight difference in the performances of the models.

The path towards building the best models involves performing experiments with different hyper-parameter settings from a baseline model configuration to the best model (iterations of



experiments). The notebooks/code related to this section can be found in the below github link.

Github Link to this project:click here

5.1 Iteration-0: Baseline Modeling

The model built (and selected) in the first iteration of the experiments acts as a baseline model (as a control) to the next step of experiment models.

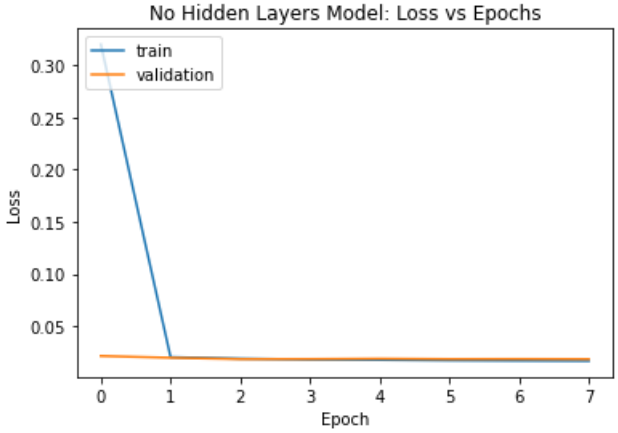
5.1.1 Baseline Model Performance

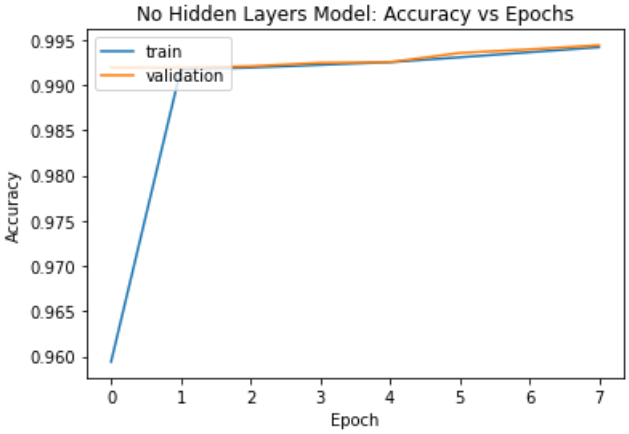
Table 3: Baseline Model Performance.

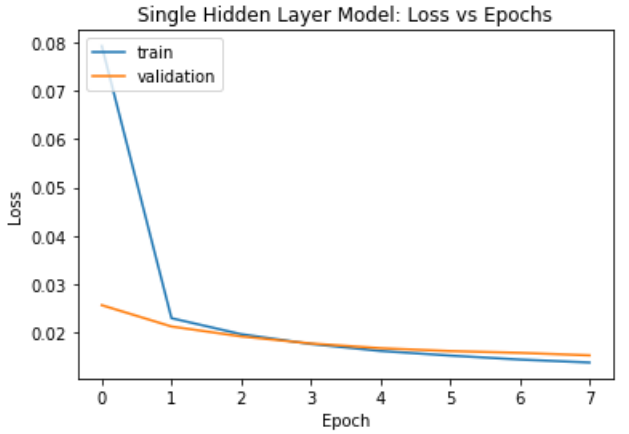
Num of Hidden Layers	Validation Set Loss	Validation Set Accuracy
0	0.0183	99.44
1	0.0153	99.61
2	0.0265	99.43

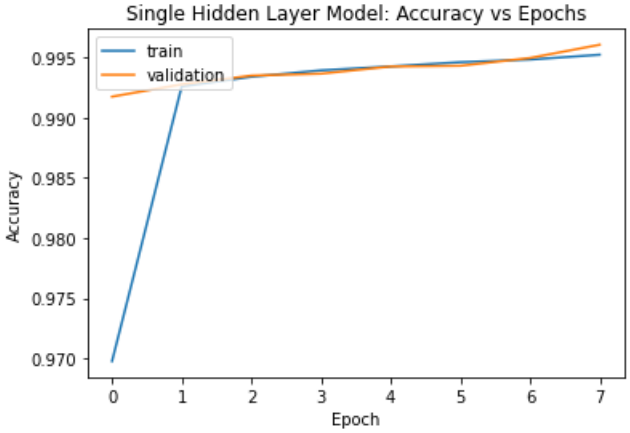
As seen from the above table, the model with 1-hidden-layer outperformed model with no hidden layers and the model with 2 hidden layers. Therefore, 1-hidden-layer will be selected and will be used as a baseline model architecture for the next step of experiments.

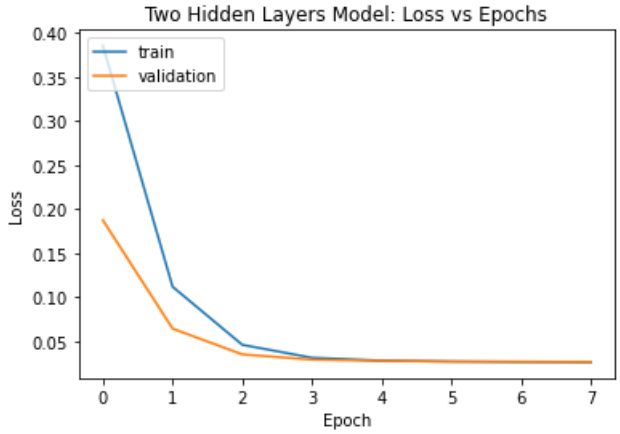
5.1.2 Learning Curves of Iteration-0 Architectures

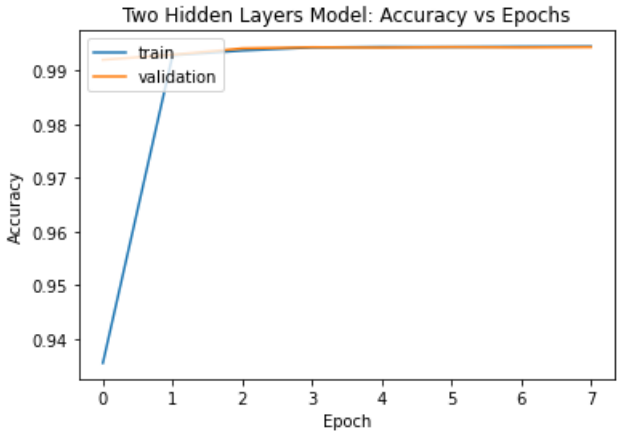

No Hidden Layers Model: Loss vs Epochs
train
validation
Loss
Epoch
0.30
0.25
0.20
0.15
0.10
0.05
0
1
2
3
4
5
6
7


No Hidden Layers Model: Accuracy vs Epochs
train
validation
Accuracy
Epoch
0.995
0.990
0.985
0.980
0.975
0.970
0.965
0.960
0
1
2
3
4
5
6
7


Single Hidden Layer Model: Loss vs Epochs
train
validation
Loss
Epoch
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0
1
2
3
4
5
6
7


Single Hidden Layer Model: Accuracy vs Epochs
train
validation
Accuracy
Epoch
0.995
0.990
0.985
0.980
0.975
0.970
0
1
2
3
4
5
6
7


Two Hidden Layers Model: Loss vs Epochs
train
validation
Loss
Epoch
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0
1
2
3
4
5
6
7


Two Hidden Layers Model: Accuracy vs Epochs
train
validation
Accuracy
Epoch
0.99
0.98
0.97
0.96
0.95
0.94
0
1
2
3
4
5
6
7

5.2 Iteration-1: Modifying activation functions

The binary classification requires 'sigmoid' activation function in the last layer, however, other activations functions like ReLU and Softmax can be used in interior layers. We built two models with 'ReLU' and 'Softmax' activations functions each. We have also changed the loss function to 'categorical-cross-entropy' when 'Softmax' is used. We deliberately did not use MSE loss function as this is a classification problem. Also, MSE makes a presumption that the underlying data distribution is normally distributed. Below are the modeling performing results in iteration-1.

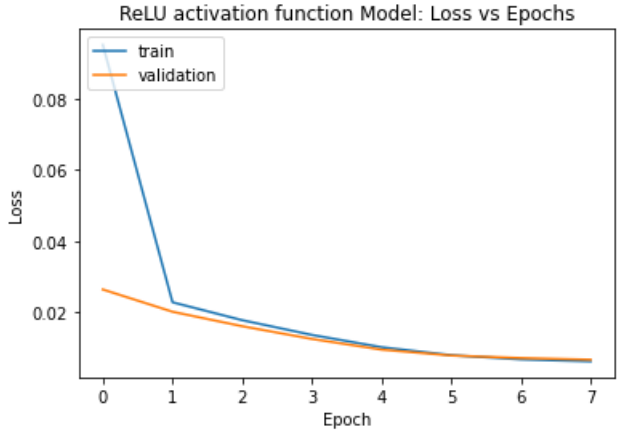
5.2.1 Iteration-1 Model Performances

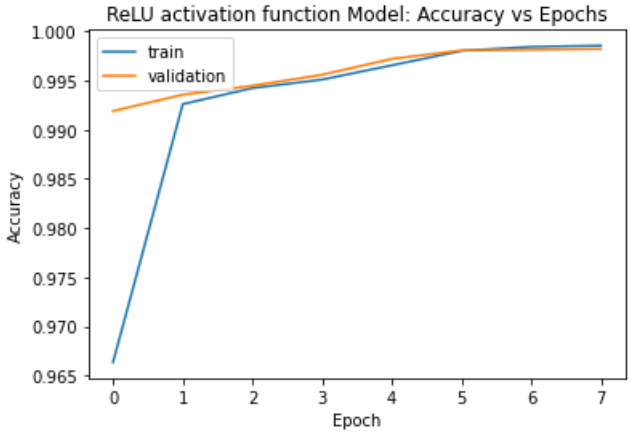
Table 4: Iteration-1: Model Performances with different activation functions.

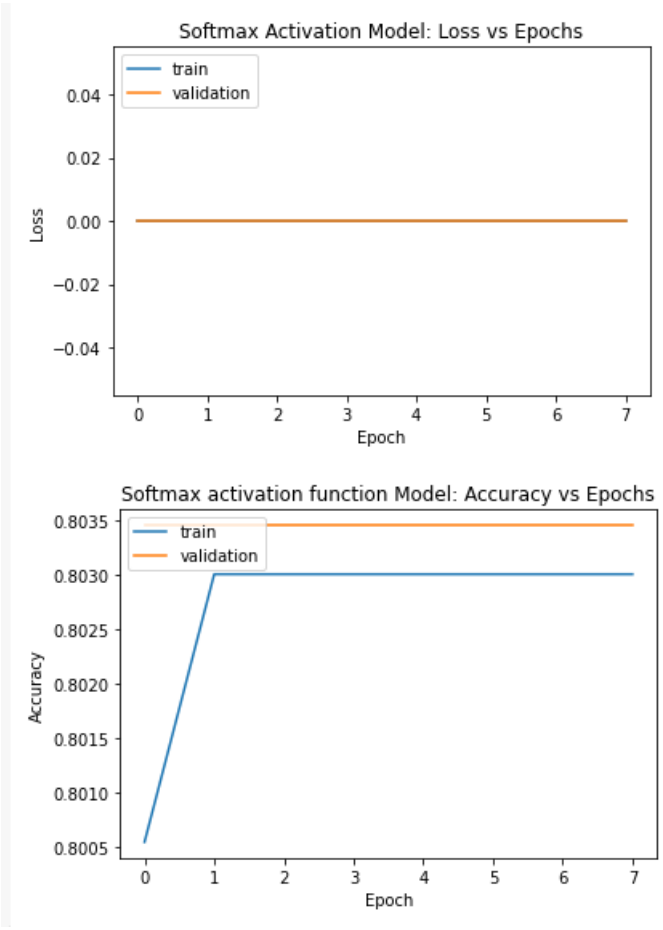
Activation Function Input Layer	Loss function	Validation Set Loss	Validation Set Accura
ReLU	Binary Cross Entropy	0.0066	99.82
Softmax	Categorical Cross Entropy	0.000001	80.34

As seen from above table, ReLU activation function in the input layer of single hidden layer architecture outperformed with accuracy of 99.61. The next set of experiments will include ReLU activation function in the input layer.

5.2.2 Learning Curves of Iteration-1 Architectures


ReLU activation function Model: Loss vs Epochs
train
validation
Loss
0.08
0.06
0.04
0.02
0
1
2
3
4
5
6
7
Epoch


ReLU activation function Model: Accuracy vs Epochs
train
validation
Accuracy
1.000
0.995
0.990
0.985
0.980
0.975
0.970
0.965
0
1
2
3
4
5
6
7
Epoch


Softmax Activation Model: Loss vs Epochs
train
validation
0.04
0.02
0.00
−0.02
−0.04
Loss
0
1
2
3
4
5
6
7
Epoch
Softmax activation function Model: Accuracy vs Epochs
train
validation
0.8035
0.8030
0.8025
0.8020
0.8015
0.8010
0.8005
Accuracy
0
1
2
3
4
5
6
7
Epoch

5.3 Iteration-2: Modifying Optimizer functions

The baseline architecture had "rmsprop" optimizer. Therefore, in this iteration, we will be trying out SGD, Adam and Adagrad optimizer functions to understand which optimizer helped the model to achieve optimum minimum in an effective way.

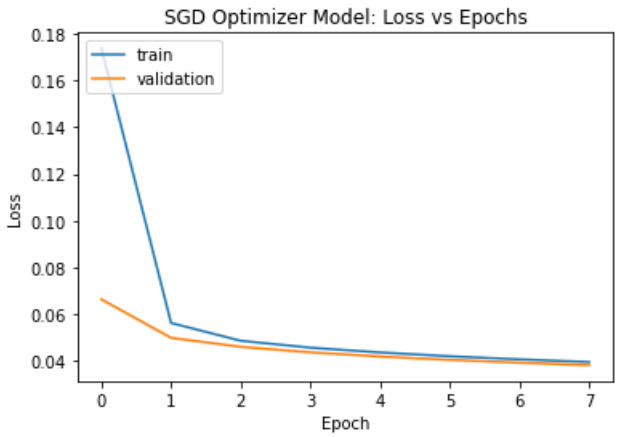
5.3.1 Iteration-2 Model Performances

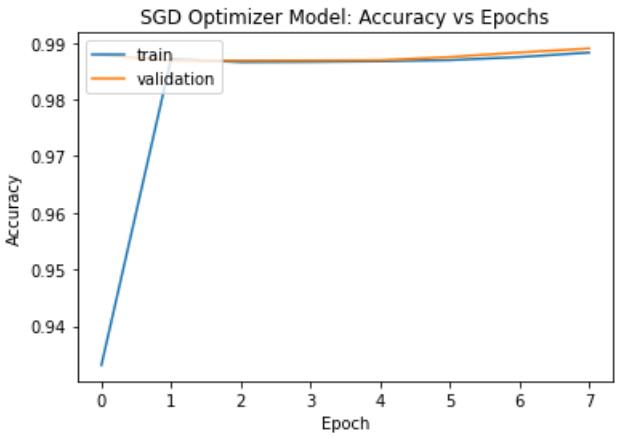
Table 5: Iteration-2: Model Performances with different optimizer functions.

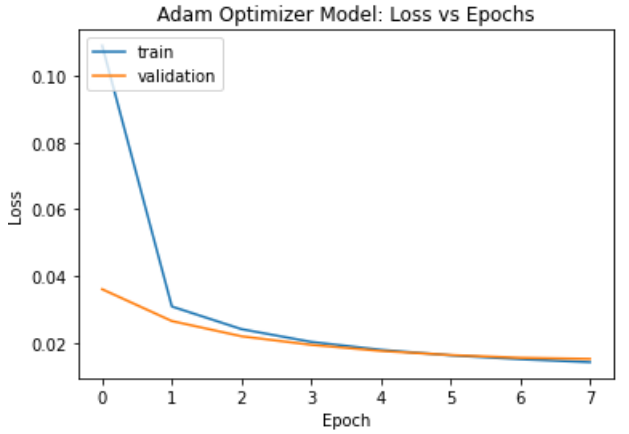
Optimizer	Validation Set Loss	Validation Set Accuracy
rmsprop (current best)	0.0066	99.82
SGD	0.038	98.9
Adam	0.0151	99.48
Adagrad	0.417	98.42

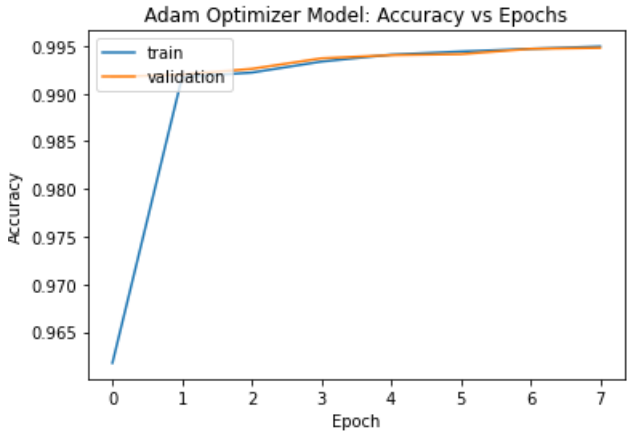
As seen from above table, the current best model with "RMSPROP" model optimizer function still outperforms the models with other optimizers. Hence, the next set of experiments will stick to "RMSPROP" optimizer function in the model architectures.

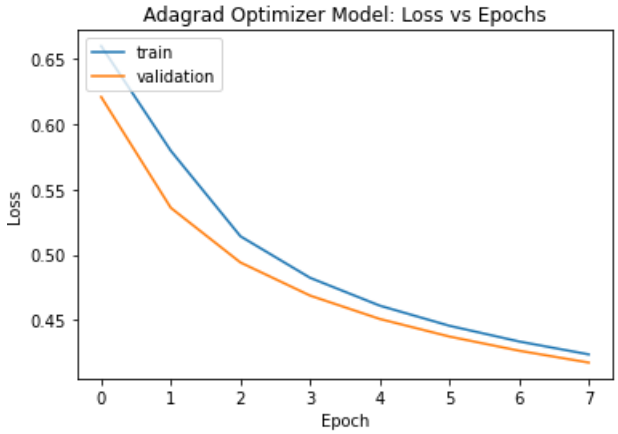
5.3.2 Learning Curves of Iteration-2 Architectures

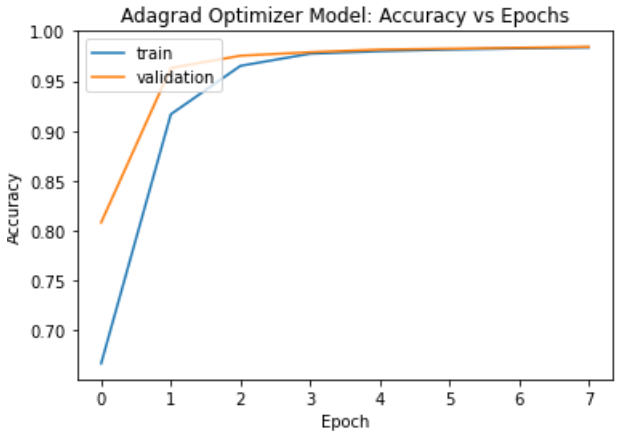





Adam Optimizer Model: Loss vs Epochs
train
validation
Loss
0.10
0.08
0.06
0.04
0.02
0
1
2
3
4
5
6
7
Epoch


Adam Optimizer Model: Accuracy vs Epochs
train
validation
Accuracy
0.995
0.990
0.985
0.980
0.975
0.970
0.965
0
1
2
3
4
5
6
7
Epoch


Adagrad Optimizer Model: Loss vs Epochs
train
validation
Loss
Epoch
0.65
0.60
0.55
0.50
0.45
0
1
2
3
4
5
6
7


Adagrad Optimizer Model: Accuracy vs Epochs
train
validation
Accuracy
Epoch
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0
1
2
3
4
5
6
7

5.4 Iteration-3: Modifying Batch Size

The baseline architecture used a batch size of 256 for training. Different batch sizes would result in different models as the batch size effect the way gradient is being computed in the optimizer functions. Hence, we tried batch sizes 128, 64 and 32 in addition to understand what batch-size would result in the best validation set performance.

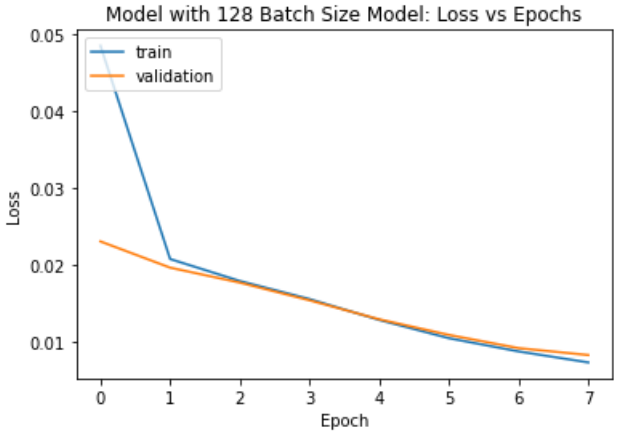
5.4.1 Iteration-3 Model Performances

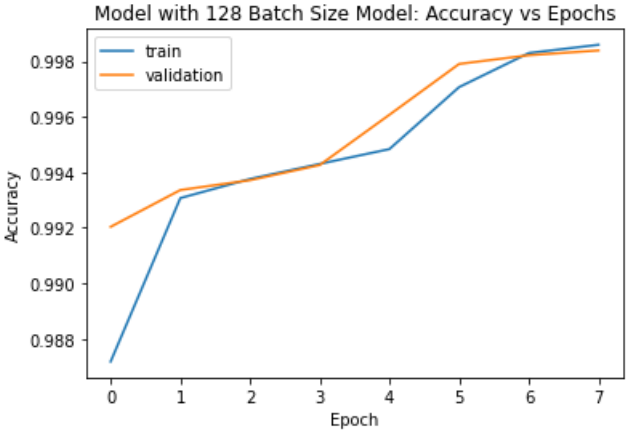
Table 6: Iteration-3: Model Performances with different batch sizes.

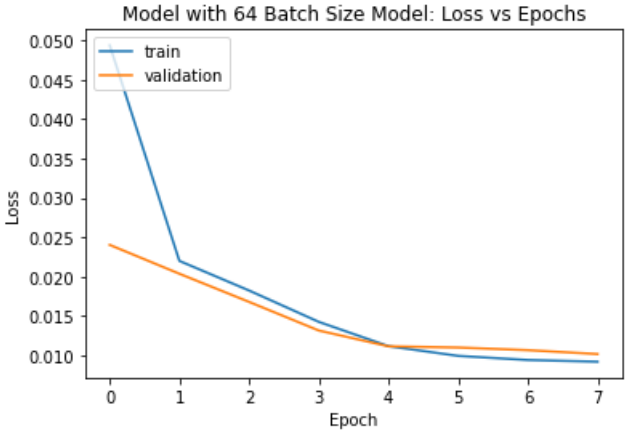
Batch Size	Validation Set Loss	Validation Set Accuracy
256 (current best)	0.0066	99.82
128	0.0083	99.84
64	0.0101	99.75
32	0.0126	99.83

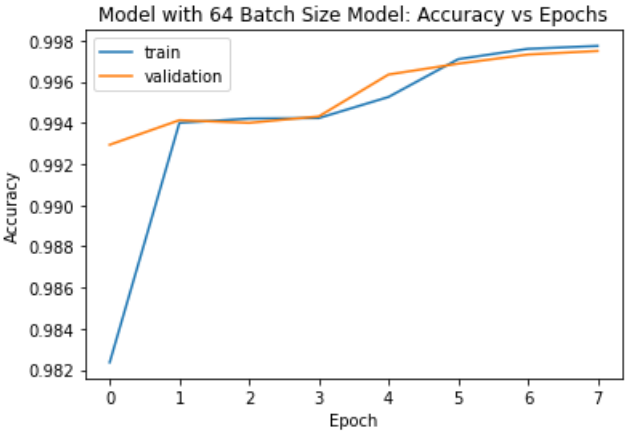
As seen from above table, the model that is trained with batch-size of 128 outperformed the current best (used 256 batch size) and the other batch-sizes. Therefore, the next set of experiments will be using a batch size of 128.

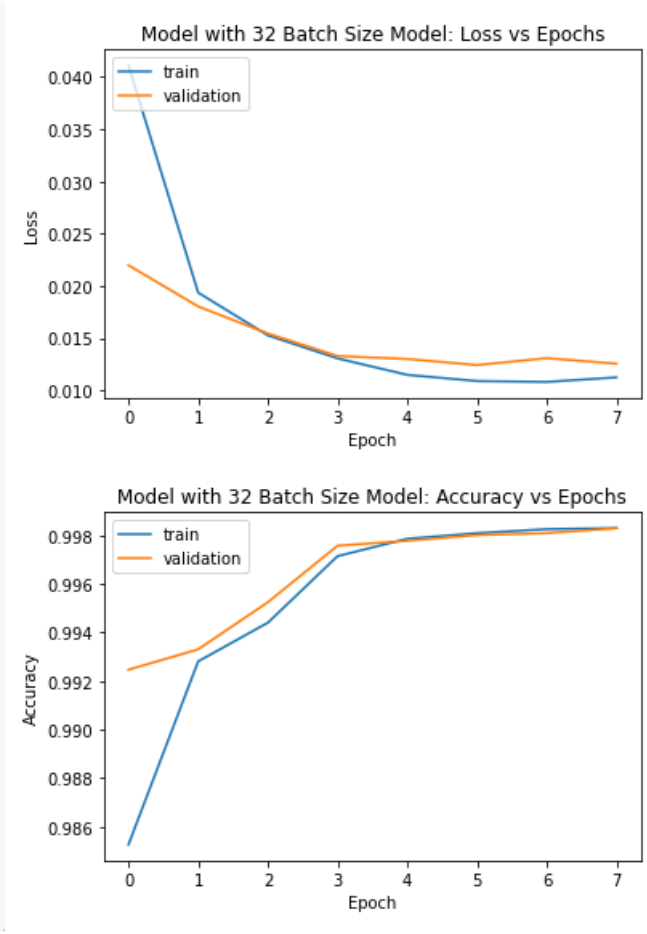
5.4.2 Learning Curves of Iteration-3 Architectures


Model with 128 Batch Size Model: Loss vs Epochs
train
validation
Loss
Epoch
0.05
0.04
0.03
0.02
0.01
0
1
2
3
4
5
6
7


Model with 128 Batch Size Model: Accuracy vs Epochs
train
validation
Accuracy
Epoch
0.998
0.996
0.994
0.992
0.990
0.988
0
1
2
3
4
5
6
7


Model with 64 Batch Size Model: Loss vs Epochs
train
validation
Loss
Epoch
0.050
0.045
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0
1
2
3
4
5
6
7


Model with 64 Batch Size Model: Accuracy vs Epochs
train
validation
Accuracy
Epoch
0.998
0.996
0.994
0.992
0.990
0.988
0.986
0.984
0.982
0
1
2
3
4
5
6
7


Model with 32 Batch Size Model: Loss vs Epochs
train
validation
Loss
0.040
0.035
0.030
0.025
0.020
0.015
0.010
0
1
2
3
4
5
6
7
Epoch
Model with 32 Batch Size Model: Accuracy vs Epochs
train
validation
Accuracy
0.998
0.996
0.994
0.992
0.990
0.988
0.986
0
1
2
3
4
5
6
7
Epoch

5.5 Iteration-4: Modifying Number of Neurons

The baseline architecture used 4 neurons for the computations. Hence, hidden layer of 2 neurons and hidden layer with 6 neurons has been tried to find if change in neurons resulted in model performance improvements as seen below

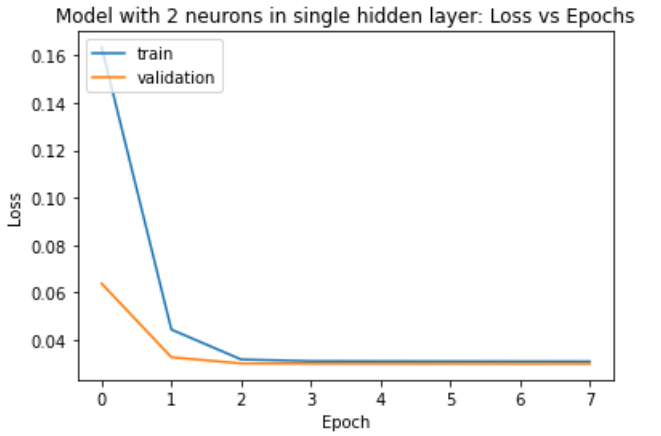
5.5.1 Iteration-4 Model Performances

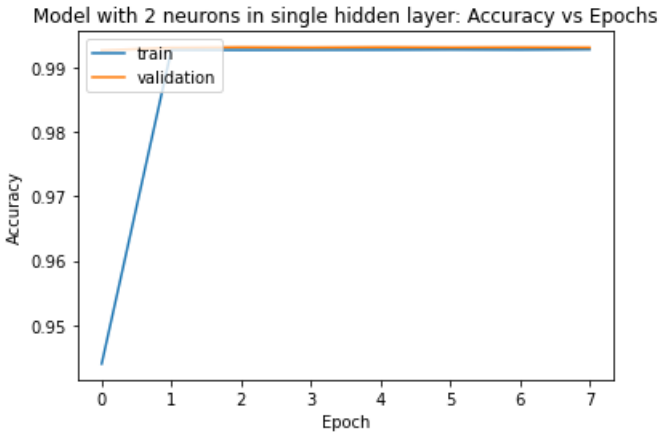
Table 7: Iteration-4: Model Performances with different number of neurons in hidden layer.

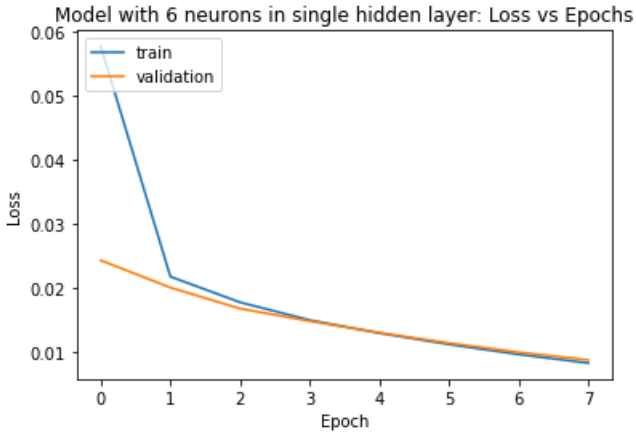
Number of Neurons	Validation Set Loss	Validation Set Accuracy
4 (current best)	0.0083	99.84
2	0.0299	99.31
6	0.0087	99.84

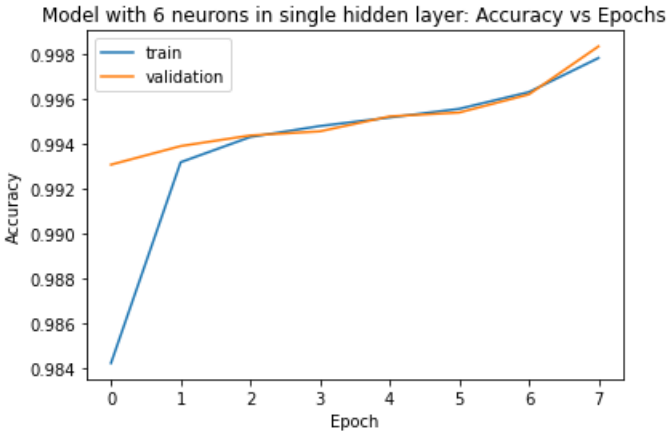
As seen from above table, the model that is trained with 6 neurons performed same as the current best model which has 4 neurons. We will be sticking to 4 neurons for next set of iterations so that the computation cost will be lower with the lesser number of neurons.

5.5.2 Learning Curves of Iteration-4 Architectures









5.6 Iteration-5: Modifying Number of Epochs

The baseline architecture used 8 epochs for the computations. We will be trying out different epoch sizes (16,32 and 64) to find out the best epoch number for this model architecture

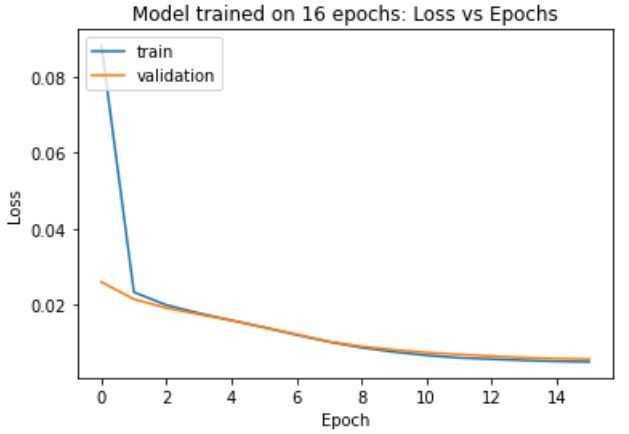
5.6.1 Iteration-5 Model Performances

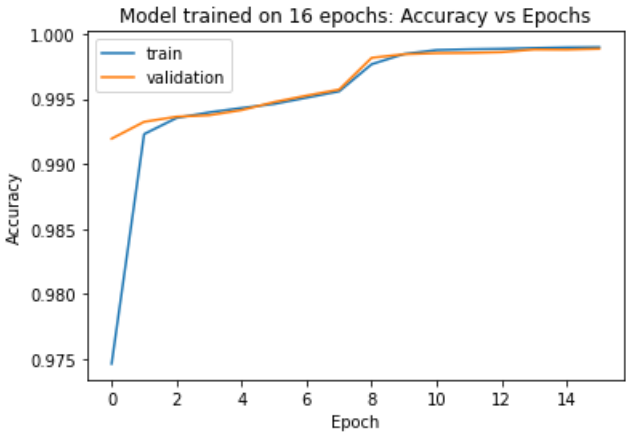
Table 8: Iteration-5: Model Performances with different number of epochs.

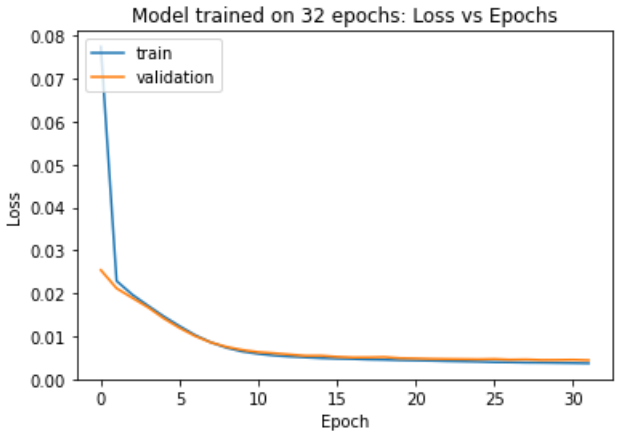
Number of Epochs	Validation Set Loss	Validation Set Accuracy
8 (current best)	0.0083	99.84
16	0.0056	99.88
32	0.0044	99.9
64	0.0047	99.91
64(Early Stopping)	0.0048	99.91

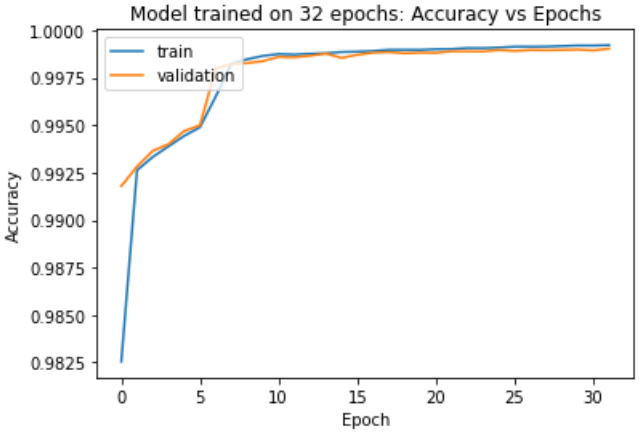
As seen from above table, the model that is trained with 6 neurons performed same as the current best model which has 4 neurons. We will be sticking to 4 neurons for next set of iterations so that the computation cost will be lower with the lesser number of neurons.

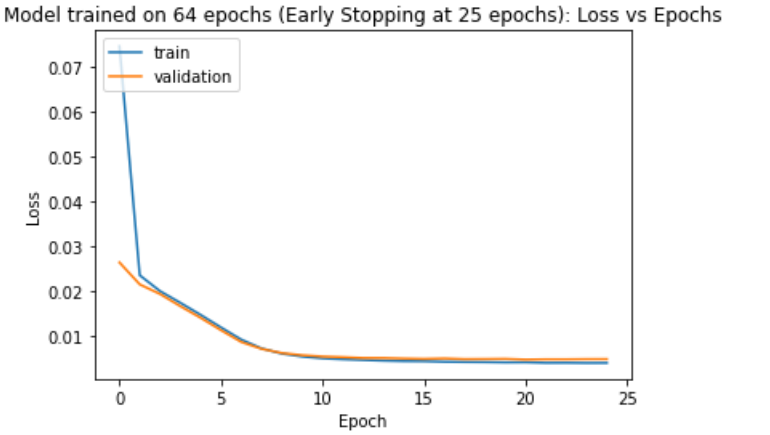
5.6.2 Learning Curves of Iteration-5 Architectures

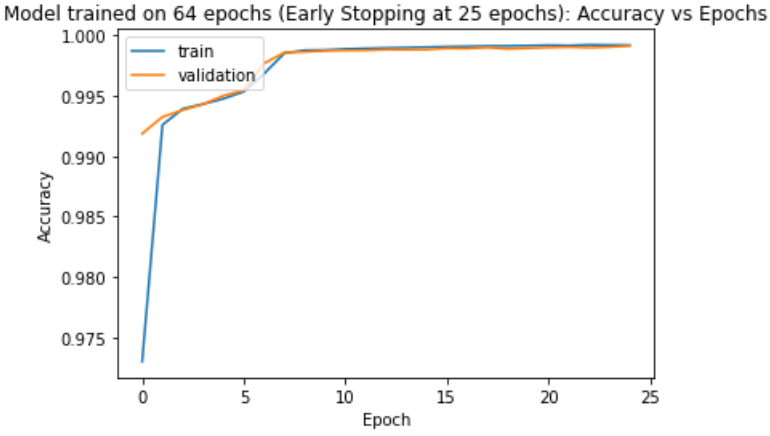

Model trained on 16 epochs: Loss vs Epochs
train
validation
Loss
0.08
0.06
0.04
0.02
0
2
4
6
8
10
12
14
Epoch

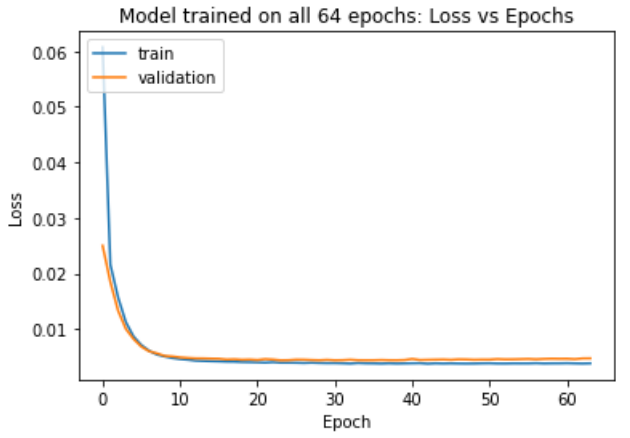

Model trained on 16 epochs: Accuracy vs Epochs
train
validation
Accuracy
1.000
0.995
0.990
0.985
0.980
0.975
0
2
4
6
8
10
12
14
Epoch

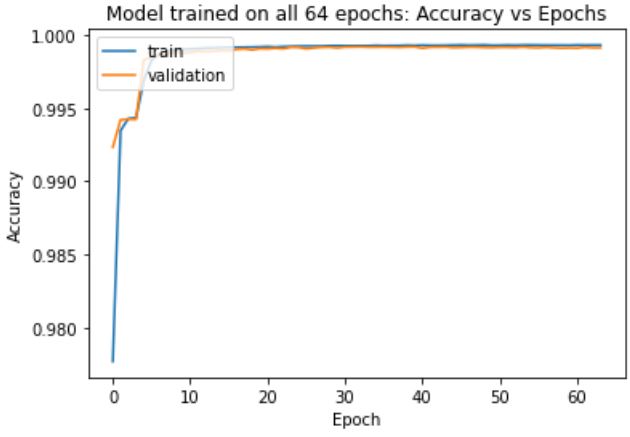

Model trained on 32 epochs: Loss vs Epochs
train
validation
Loss
Epoch
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0.00
0
5
10
15
20
25
30


Model trained on 32 epochs: Accuracy vs Epochs
train
validation
Accuracy
Epoch
1.0000
0.9975
0.9950
0.9925
0.9900
0.9875
0.9850
0.9825
0
5
10
15
20
25
30


Model trained on 64 epochs (Early Stopping at 25 epochs): Loss vs Epochs
train
validation
Loss
Epoch
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
5
10
15
20
25

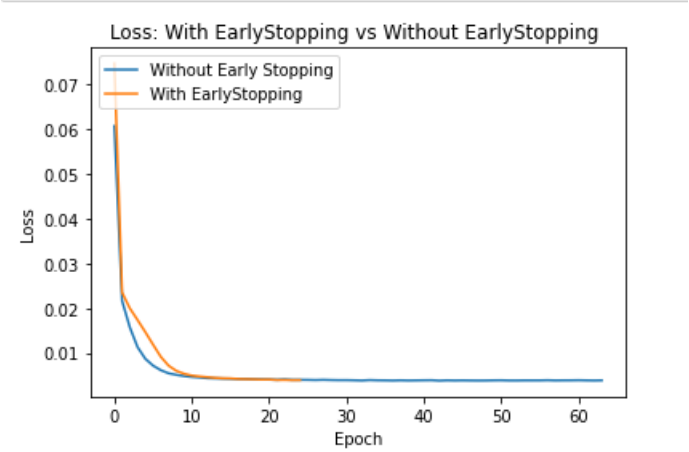

Model trained on 64 epochs (Early Stopping at 25 epochs): Accuracy vs Epochs
train
validation
Accuracy
Epoch
1.000
0.995
0.990
0.985
0.980
0.975
0
5
10
15
20
25


Model trained on all 64 epochs: Loss vs Epochs
train
validation
Loss
0.06
0.05
0.04
0.03
0.02
0.01
0
10
20
30
40
50
60
Epoch


Model trained on all 64 epochs: Accuracy vs Epochs
train
validation
Accuracy
1.000
0.995
0.990
0.985
0.980
0
10
20
30
40
50
60
Epoch

5.6.3 EarlyStopping

The concept of early-stopping in model training saves computation time and cost when the model is not learning. As seen in the iteration-5, the model did not learn after epoch 25. The experiment did set the patience to 4 in the keras callback function which means the model loss did not improve continuously for 4 epochs before 25th epoch (including 25).



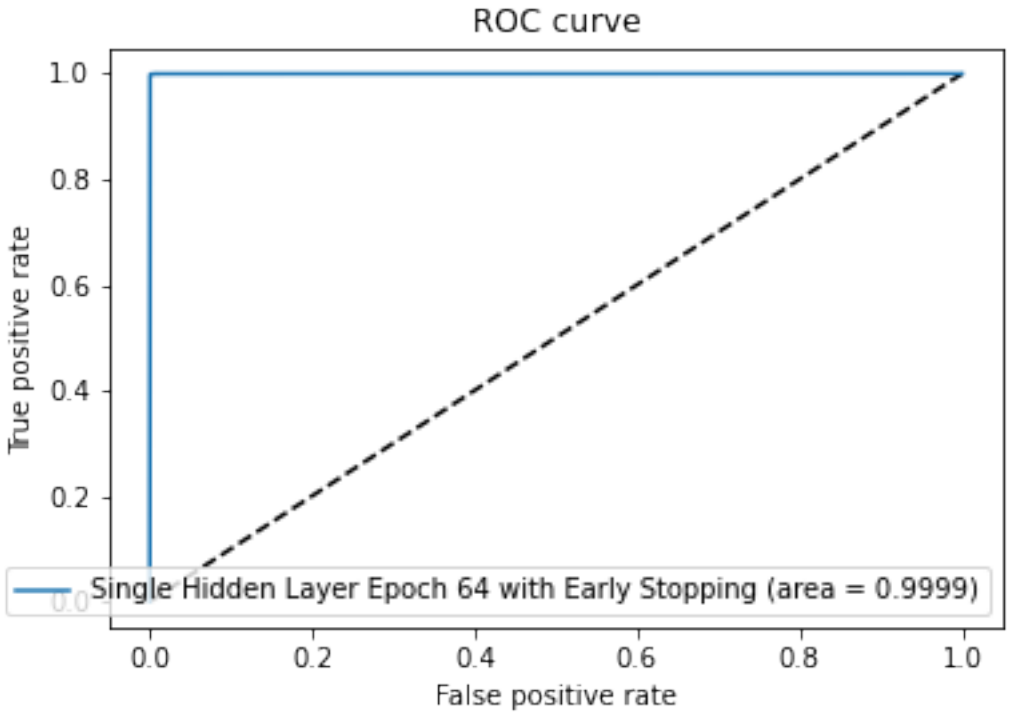
5.7 Model Selection

We have obtained various metrics like precision, recall, F1, and Area-Under-Curve (AOC) for each model to select the best model.

Model	N_Params	Accuracy	Precision	Recall	F1-Score	auc_score
No Hidden Layers	120	0.9944	0.9932	0.9787	0.9859	0.9984
Single Hidden Layer	485	0.9961	0.994	0.9863	0.9901	0.9992
Two Hidden Layers	493	0.9943	0.9972	0.9746	0.9858	0.9965
Single Hidden Layer ReLU Activation	485	0.9982	0.9942	0.9966	0.9954	0.9999
Single Hidden Layer Softmax Activation	485	0.8034	0	0	0	0.5321
Single Hidden Layer SGD optimizer	485	0.989	0.9886	0.9569	0.9725	0.9955
Single Hidden Layer Adam optimizer	485	0.9948	0.9959	0.978	0.9869	0.9994
Single Hidden Layer Adagrad optimizer	485	0.9842	0.9804	0.9414	0.9605	0.9873
Single Hidden Layer Batch Size 128	485	0.9984	0.9955	0.9963	0.9959	0.9997
Single Hidden Layer Batch Size 64	485	0.9975	0.991	0.9962	0.9936	0.9998
Single Hidden Layer Batch Size 32	485	0.9983	0.9953	0.996	0.9957	0.9998
Single Hidden Layer 2 Neurons	243	0.9931	0.9993	0.9667	0.9828	0.9959
Single Hidden Layer 6 Neurons	727	0.9984	0.9957	0.9959	0.9958	0.9999
Single Hidden Layer Epoch 16	485	0.9988	0.9972	0.9969	0.9971	0.9999
Single Hidden Layer Epoch 32	485	0.999	0.9976	0.9975	0.9975	0.9999
Single Hidden Layer Epoch 64 with Early Stopping	485	0.9991	0.9984	0.9973	0.9978	0.9999
Single Hidden Layer Epoch 64 without Early Stopping	485	0.9991	0.9978	0.9977	0.9978	0.9999

5.7.1 ROC Curves for all Models

Below is the ROC curve for the best model with epoch size = 25 along with best parameters. ROC curves for all models can be found in this notebook



5.8 Model Selection Inference

As seen from the classification metrics and ROC curves, the single layer model with 4 neurons, rmsprop optimizer and ReLU activation function along with batch-size=128, epochs=25 achieved higher accuracy, precision, recall, f1-score and auc values. This proves that this model is more generalized across both the target classes. Hence, the above highlightened model is being selected.

6 Overfitting with target variable

Another way of understanding the overfitting problem is to understand how big the neural network architecture should be when output variable is being used as additional feature in the input feature set. We have performed an experiment with the output feature in the input dataset which can be found in the github link -
As seen in the below figure, it required only a basic architecture with no hidden layer to overfit the entire dataset in this case. This is because, the correlation between the output variable feature and output variable itself will be 1 which could be the reason for this easy overfitting.

```
no_hidden_layers_model = Sequential()
no_hidden_layers_model.add(Dense(1, input_dim = len(XTRAIN[0, :])))
no_hidden_layers_model.compile(loss='binary_crossentropy', optimizer = 'rmsprop', metrics=[ 'accuracy'])
```

```
no_hidden_layers_model_history = no_hidden_layers_model.fit(XTRAIN, YTRAIN, validation_data = (XVALID, YVALID
                                        batch_size=256, epochs = 8, verbose = 1)
```

```
Epoch 1/8
1544/1544 [==============================] - 1s 959us/step - loss: 2.5568e-06 - accuracy: 1.0000 - val_loss:
08 - val_accuracy: 1.0000
Epoch 2/8
1544/1544 [==============================] - 1s 809us/step - loss: 2.1143e-06 - accuracy: 1.0000 - val_loss:
00 - val_accuracy: 1.0000
Epoch 3/8
1544/1544 [==============================] - 1s 735us/step - loss: 1.8307e-06 - accuracy: 1.0000 - val_loss:
08 - val_accuracy: 1.0000
Epoch 4/8
1544/1544 [==============================] - 1s 840us/step - loss: 1.6096e-06 - accuracy: 1.0000 - val_loss:
07 - val_accuracy: 1.0000
Epoch 5/8
1544/1544 [==============================] - 1s 739us/step - loss: 1.4003e-06 - accuracy: 1.0000 - val_loss:
08 - val_accuracy: 1.0000
Epoch 6/8
1544/1544 [==============================] - 1s 637us/step - loss: 1.3117e-06 - accuracy: 1.0000 - val_loss:
00 - val_accuracy: 1.0000
Epoch 7/8
1544/1544 [==============================] - 1s 665us/step - loss: 9.9634e-07 - accuracy: 1.0000 - val_loss:
00 - val_accuracy: 1.0000
Epoch 8/8
1544/1544 [==============================] - 1s 634us/step - loss: 8.8477e-07 - accuracy: 1.0000 - val_loss:
00 - val_accuracy: 1.0000
```

7 Custom predict function

After selecting and saving the best model, we have also implemented a custom predict function manually by using the trained model parameters (weights). We performed the computations manually as per the network architecture. As seen in below notebook cell outputs, the predictions (and metrics) are same as compared to the keras predict function.

Manual Compute prediction

In [157]:
```
def manual_custom_predict(best_model_selected, XVALID):

    # Parameters layer 1
    W1 = best_model_selected.get_weights()[0]
    b1 = best_model_selected.get_weights()[1]

    # Parameters layer 2
    W2 = best_model_selected.get_weights()[2]
    b2 = best_model_selected.get_weights()[3]

    # Input
    #X1 = np.array([[0,0],[0,1],[1,0],[1,1]], "float32")
    # Use the following X1 for single input instead of all at once
    #X1 = np.array([[0,0]])

    # First layer calculation
    L1 = np.dot(XVALID,W1)+b1
    # Relu activation function
    X2 = np.maximum(L1,0)
    # Second layer calculation
    L2 = np.dot(X2,W2)+b2
    # Sigmoid
    output = 1/(1+np.exp(-L2))
    return output

output = manual_custom_predict(best_model_selected, XVALID)
output[:5]
```

Out[157]:
```
array([[5.74528108e-05],
       [5.74528108e-05],
       [1.00000000e+00],
       [5.74528108e-05],
       [5.74528108e-05]])
```

Compare predictions

In [156]:
```
keras_model_predictions = best_model_selected.predict(XVALID)
keras_model_predictions[:5]
```

Out[156]:
```
array([[5.7456113e-05],
       [5.7456113e-05],
       [1.0000000e+00],
       [5.7456113e-05],
       [5.7456113e-05]], dtype=float32)
```

Compare metrics

```
y_pred_output = [1 if each[0] >=0.5 else 0 for each in output]
acc = accuracy_score(y_pred_output, y_true)
prec = precision_score(y_pred_output, y_true)
rec = recall_score(y_pred_output, y_true)
f1 = f1_score(y_pred_output, y_true)

acc, prec, rec, f1
```

```
(0.9990688636087608,
 0.9973223480947477,
 0.9979389942291839,
 0.9976305758730812)
```

```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_pred_output, y_true)
```

```
array([[79344,    52],
       [   40, 19368]], dtype=int64)
```

```
y_pred_model = [1 if each[0] >=0.5 else 0 for each in keras_model_predictions]
acc = accuracy_score(y_pred_model, y_true)
prec = precision_score(y_pred_model, y_true)
rec = recall_score(y_pred_model, y_true)
f1 = f1_score(y_pred_model, y_true)

acc, prec, rec, f1
```

```
(0.9990688636087608,
 0.9973223480947477,
 0.9979389942291839,
 0.9976305758730812)
```

```
confusion_matrix(y_pred_model, y_true)
```

```
array([[79344,    52],
       [   40, 19368]], dtype=int64)
```

8 Conclusion

Will be updated in next phase

References

[1] Manoj Kumar Putchala. Deep learning approach for intrusion detection system (ids) in the internet of things (iot) network using gated recurrent neural networks (gru). 2017.

[2] Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali A Ghorbani. A detailed analysis of the kdd cup 99 data set. In *2009 IEEE symposium on computational intelligence for security and defense applications*, pages 1–6. IEEE, 2009.

[3] Steven Huang. Kdd cup 1999 data, computer network intrusion detection (version 1). 2018. Available from https://www.kaggle.com/galaxyh/kdd-cup-1999-data.