Final Project Report for CS 175, Spring 2020

Project Title: State Farm Distracted Driver Detection
Project Number: 42

Student Name(s)
Matthew Mueller, 77136800, mrmuell1@uci.edu
Tyler Reynolds, 48295243, tylermr@uci.edu

1. Introduction and Problem Statement

This project will use a convolutional neural network to predict when a driver is distracted based on the action they are doing in the given image in the State Farm dataset. Our neural network will classify that image into one of the ten classifications given by State Farm.

For this project, we will be using the open-source neural network Keras. This means that our methods and algorithms will be very similar to what we have learned in this class and very similar to what we did on Assignment 3. We will be using Python to create code that can load and arrange the dataset in a way that it can easily feed into Keras. Just like the CIFAR-10 dataset, the State Farm dataset is made up of images that fit into one of ten classifications. Our goal is to classify these images with the highest accuracy possible. To do this we will be experimenting with various combinations of neural network functions to try and find the most optimal result possible. With the similarity of the State Farm dataset to our CIFAR-10 dataset we used in Assignment 3 we will start by employing those strategies and going from there. Once we have found models that we are happy with we will compare the performance of the models that we feel deserve to be highlighted. We will highlight these models through the creation of various plots and text-based analysis.

2. Related Work

To help get an understanding of the model we are trying to create, we will analyze various driver actions depicted by the different pictures given by the State Farm dataset. From these images, our model will decide if the given driver is distracted or not. With the ability to accurately tell what drivers commonly look like while distracted it will be easier to assess, without human intervention, if a person was distracted during an accident. We have already tackled a similar problem to this in our work with the CIFAR-10 dataset on our homework assignments. This dataset similarly breaks images into 10 categories. With this in mind, we can draw from our assignments for some inspiration in solving this problem. This problem has been addressed in the past when State Farm originally released their data. On Kaggle, many people have released their solutions to some of the problems the data set presents. Since we are using Keras we found another user who was also using Keras. His work can be found here. While his

actual model code does not perform well as he mentions, we were not really after any model insight, we just wanted to see how others tackled data preprocessing. After looking at the linked code alongside others code on Kaggle we came to realize that many others utilized the same code for data preprocessing. With this discovery, we felt that this code had good bones and we made modifications to it to handle the preprocessing specific to some of the alterations we had made for our data. Many people on the web have experimented with creating models for this data. One person by the name of Satya Naren Pachigolla on *Medium* used the provided data and analyzed it through the use of a convolutional neural network. He utilized a log loss evaluation metric to get his loss. He utilized a combination of Average Pooling, Convolutional 2D, Dropout, Batch Normalization, and Dense to achieve his results. He also utilized Stochastic Gradient Descent for his optimizer as opposed to using ADAM or RMS Prop as they caused some erratic data, something we can keep in mind while testing. His models are not the end all be all, but it serves as a good seed when getting started with testing different models.

3. Data Sets

To create a model that can decipher between distracted and alert drivers, we are utilizing data provided by State Farm that helps us achieve just that. They provide 102,000 jpeg images. These images are partitioned into training and test sets with the training set having 22,000 images and the test set having 80,000 images. The training data consists of a set of drivers each doing an action that may be considered distracted. We then verify our model's validity by utilizing the test data that consists of various drivers doing various things and classifying their actions as distracted or not.

The first issue with this data set that needed to be addressed was resizing the images as the images were all originally 640x480 resolution. Initially, we solved this issue by creating a python script that could resize these images to any size of our choosing and store them in new folders. This was useful for our initial testing phase and for storing the images on our Github repository. Once we moved into using more complex neural network architectures, we began to use Keras's built-in resizing functionality to resize our images when needed.

Another issue with the data that we found and that many others online had speculated about was that the images are too similar, which caused many of our basic models to overfit. The best solution that we found to solve this overfitting issue was to use the dropout function combined with using complex neural network architectures that are known to perform very well already.

4. Description of Technical Approach

The first thing that we handled was figuring out a way to make the dataset smaller so that we could easily run it and put it on our shared GitHub repository. We accomplished this by writing a python script that can copy and shrink each image in our dataset to a resolution of our choosing. We chose to shrink the images to a resolution 80x60, which preserves the original aspect ratio and cut down the size of our data set from over 4GB to just under 400MB. Next, We worked on writing the code to load in our data. Looking at the Kaggle page for this project, there are a lot of public notebooks with useful code that can read and load data into variables. We used some of this code as a reference but a lot of it was just standard data processing that we have seen in other classes many times before. Lastly, We created a function to store data into files using pickle. Once we had that store function written, we quickly added some functionality that allowed our program to check if the pickle files exist and load the data from those pickle files instead of reading the 400MB dataset each time we wanted to run the program on a new model. This was working well for simple models but we realized we needed a more efficient way of gathering data. It was at this point that we decided to scrap a lot of the previous code we had in favor of Keras's built-in image processing functions. It was a bit hard to start over but the result was worth it as using Keras's built-in functions provided us with very efficient code.

To create the model with the greatest validation accuracy it took much experimentation. To achieve this, we utilized various neural network functions to help find the most optimal one. For each convolutional layer, we use the 2D Convolution function. We experimented with three different optimizers Adam, Stochastic Gradient Descent, and RMSProp. Alongside experimenting with the optimizer we also experimented with different loss functions. I chose to go with Categorical Cross Entropy due to the nature of our data and from my readings it is often the best loss function. We also experimented with adding different amounts of convolutional layers in our convoluted neural network to see the differences in results with the max amount of convolutional layers being 3. As for non required elements of our neural network's functionality, we turned to our work on Assignment 3 to help guide me in building a neural network that would work given the similarity of this data to the CIFAR-10 dataset. For our activation function on each layer, I used ReLU except for at the end of each network where we use softmax. We opted with ReLU due to our experience with it in the past and from our readings and experience it is a relatively good jack of all trades activation function for our purposes Unlike our work on Assignment 3 we added pooling to the neural network. For our purposes we opted with Max Pooling as per our research it is better at detecting small differences in a frame as it does not smooth the frame as much as opposed to average pooling. This ability is important when trying to see small driver actions that may be causing a distraction. After doing pooling we did batch normalization and a dropout as we did in Assignment 3. We experimented with the different ways to create a neural network, to not only figure out the best neural network setup for our data set but also to help generate some plots for visualization for our purposes of understanding as

well as the purposes of the report. With the various models being tested we are not able to feature all of them. So, we were tasked with finding the best 3 models in our testing and generating data and visualizations to make valid comparisons between them and ultimately come away with a relatively optimized model. The block diagram for the final model we chose can be seen below:

Layer (type)	Output Shape	Param #
conv2d_7 (Conv2D)	(None, 58, 78, 64)	1792
batch_normalization_11 (Batc	(None, 58, 78, 64)	256
max_pooling2d_7 (MaxPooling2	(None, 29, 39, 64)	0
dropout_10 (Dropout)	(None, 29, 39, 64)	0
conv2d_8 (Conv2D)	(None, 29, 39, 128)	73856
batch_normalization_12 (Batc	(None, 29, 39, 128)	512
max_pooling2d_8 (MaxPooling2	(None, 15, 20, 128)	0
dropout_11 (Dropout)	(None, 15, 20, 128)	0
conv2d_9 (Conv2D)	(None, 15, 20, 256)	295168
batch_normalization_13 (Batc	(None, 15, 20, 256)	1024
max_pooling2d_9 (MaxPooling2	(None, 8, 10, 256)	0
dropout_12 (Dropout)	(None, 8, 10, 256)	0
flatten_5 (Flatten)	(None, 20480)	0
dense_11 (Dense)	(None, 1024)	20972544
batch_normalization_14 (Batc	(None, 1024)	4096
dense_12 (Dense)	(None, 256)	262400
batch_normalization_15 (Batc	(None, 256)	1024
dropout_13 (Dropout)	(None, 256)	0
dense_13 (Dense)	(None, 10)	2570

Total params: 21,615,242
Trainable params: 21,611,786
Non-trainable params: 3,456

5. Software

a. The first major piece of code that we wrote was our smallerImgs.py file, which contains code to turn the original 640x480 images into any smaller resolution of our choosing using SciPy. Next, we wrote driver code that allowed for us to read in all the necessary images and driver information from the .csv file. After that, we wrote some code to take that loaded information and store it into pickle files so it could be easily retrieved when we needed it. This code was partially written by us using our knowledge of Python data processing and partially adapted from code on a Public Kaggle notebook by user zfturbo. Looking further into the matter, it appears many other participants also used this user's code as a starting point including the eventual winner of the entire original competition. With that being said, I still put this part of our software in this section because the driver code needed to be heavily modified to work on our systems since many of the methods used have become deprecated over the years. All of this took a lot of time and we were getting validation accuracy scores in the 40-60 percentile, but we eventually decided to go a different route with our driver code once we learned that Keras has great built-in image processing functionality. As for the neural network architectures, we first used Keras's model functions to make some custom neural network architecture based on what we knew about CNN.

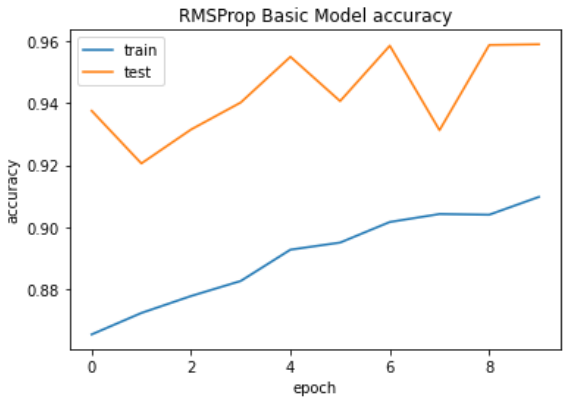
b. At this point, we proceeded to use Keras's built-in image processing functions to grab our needed images and used Keras's fit_generaotr to train our model. These built-in functions

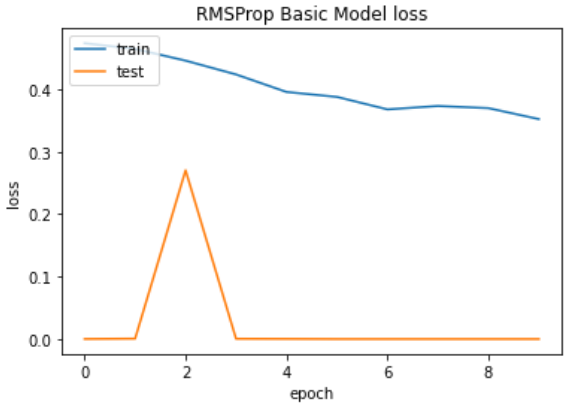
worked and looked much nicer than the code we were using previously mentioned above. After we experimented with our custom architectures as mentioned in section a, we did testing with the premade Xception and InceptionV3 architectures. In the next section, we will explore how our architectures performed. We then made some slight modifications to example plotting code from Keras's website to plot the accuracies and loss values of our models. We also adapted some driver code from the public notebook of Kaggle user bhanotkaran22 to store our model data into a .csv file that could be submitted to Kaggle and give us a final true score for a particular model.

6. Experiments and Evaluation

To get a good idea of the image classification capabilities present within a convoluted neural network we experimented with making our custom models as well as using Keras' pre-trained models. Through doing this we can identify the capabilities of our models in comparison to each other as well as recognize how well some of these pre-trained models work in comparison to our custom ones.

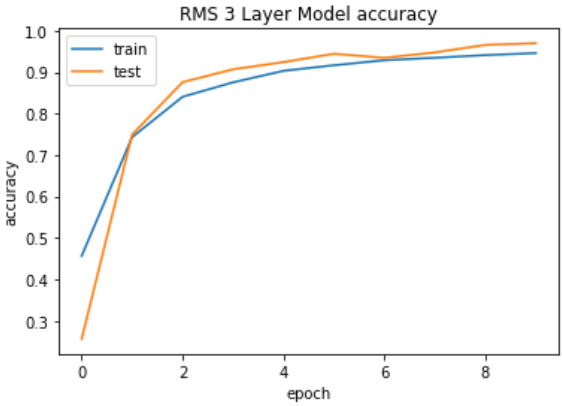
To set up our custom models for experimentation we first decided what elements of our convoluted neural network we wanted to keep static. From our experience with convoluted neural networks and using them with the CIFAR-10 database, as well as the many resources online regarding image classification, we were able to construct multiple models to help us figure out which convoluted neural network architecture is optimal given the structure of our data. We decided that for our custom models we would alter the optimizer(RMSProp, SGD, and Adam) and the number of convolutional layers(1,2,3) to see the differences in test loss and test accuracy.

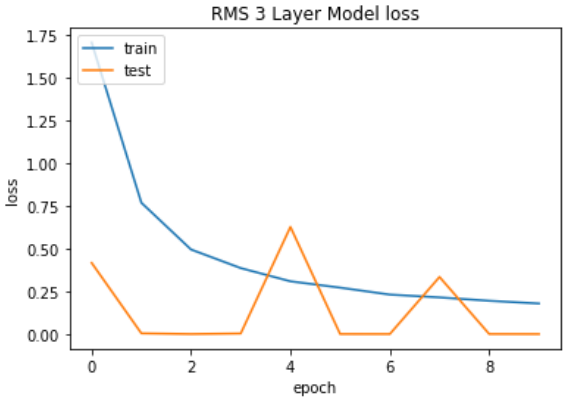




To get an early idea of the performance of our optimizer we trained a very basic model. The architecture of this model is two Convolution2D layers that have a preceding max pooling

and dropout layer. The result of this model can be seen below. Given the differences in training and test accuracy as well as training and test loss, it is obvious that this basic model is experiencing extreme overfitting and this was the best model produced from the basic architecture, so there is room for improvement. One thing to note is that the SGD optimizer failed to produce a meaningful result with the architecture of the basic model.

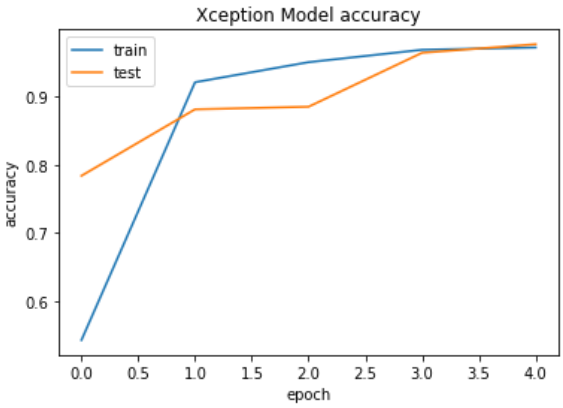


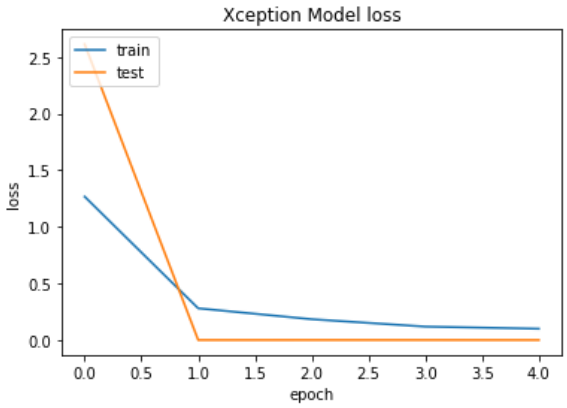


In our testing of the different models' optimizers paired with different amounts of convolutional layers, we utilized the following architecture dependent on the number of convolutional layers (denoted by the number listed on the plot): Convolution2D->ReLU->Batch Normalization->Max Pooling->Dropout. In the end, we also added two extra Dense layers, then utilized batch normalization and added another dropout layer. Unlike the basic architecture, in this model we utilize batch normalization and an activation function to improve our results extensively. Although all models saw improvements to their performance, RMSProp 3 performed significantly better than the other combinations as it produced the highest test accuracy and lowest test loss with the least amount of data volatility. SGD and Adam performed closely, but they saw much more volatility in their data which hindered our confidence in them.

After testing our custom models, we decided to experiment with some of the built-in architectures that Keras has. We experimented with both the Xception and InceptionV3. Xception is the main architecture that we ran experiments with and its model accuracy and loss graphs can be seen below. Xception provided us with an architecture that was much more complex than our custom architecture but it still needed some tweaking to make it run well and score high on our data set. An additional dropout layer was added to help the neural network not overfit the data as much. In addition to this, using the adam optimizer would sometimes cause the training accuracy to become stuck. To fix this, I tried to run the neural network multiple different times and it would usually get itself unstuck. I also believe that the SGD optimizer would help greatly here as well as when I used the SGD for the InceptionV3 architecture, I never experienced the training accuracy becoming stuck. With that being said, the Xception

architecture was by far the best neural network that we experimented with and we were able to achieve a Kaggle score of .63716 with just 5 epochs of training, which is very respectable.



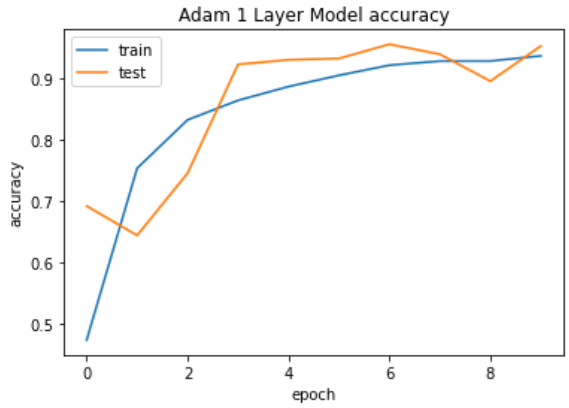


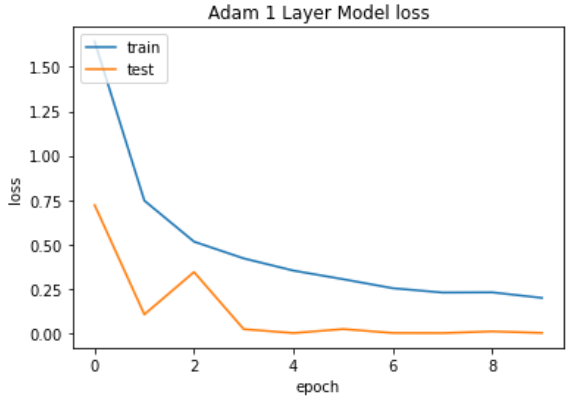
7. Discussion and Conclusion

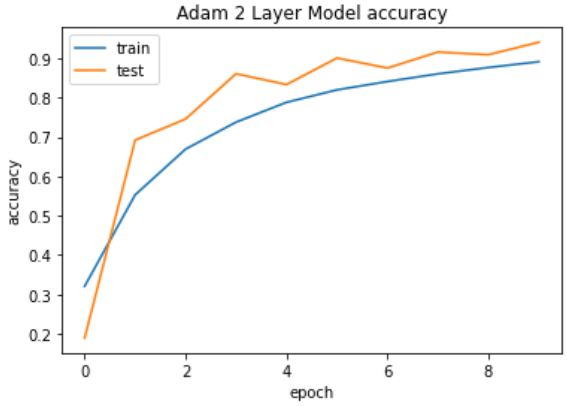
We truly believe that this project offered a lot of insight into neural network architecture and how one should go about creating a good architecture. We learned a lot about Keras and how efficiently it can load data and train a model that can get a nice good accuracy with just a few epochs of training. The sheer amount of built-in functionality for Keras was probably the most surprising part of this project for us. In fact, the functions were so good that we scrapped many hours of work just to switch over to using them but our team believes that a large part of computer science is the willingness to admit that your original solution may not be optimal and you may have to just start over. From our experience and the experience of the other Kaggle participants, our team believes that a major limitation of this was the dataset's similar images that tended to cause our models to overfit very often. This was a big roadblock for us for a while and many of our basic models performed poorly because of this. However, working through this problem was very rewarding and taught us a lot about how a neural network can be customized to fit a specific need or fix a data limitation. If our team was in charge of a research lab, the first thing we would do would be to inspect and try to upgrade the current hardware because our more complex architectures were very time consuming and taxing for our current machines. Even when they were running on our Nvidia 1080 and 1080 ti GPUs, the more complex models were very time consuming to get good data from. We would also prioritize using the best existing architectures or trying to modify them to get better results for the current task our team is handling.

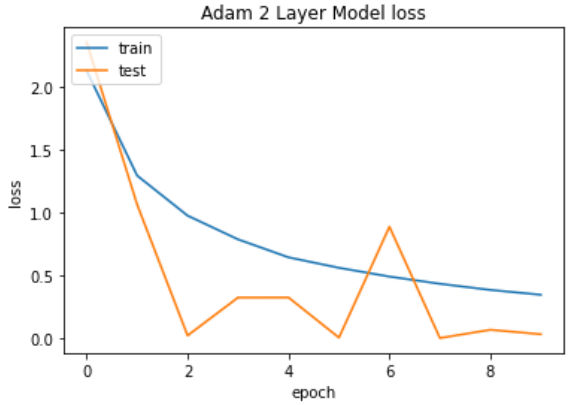
As for some closing thoughts, our team feels that this project was a great time for us to finally tackle a full-on machine learning problem as many of the machine learning problems we faced before in other classes tended to hold our hand too much through the process. We feel like we learned a lot and are very pleased with our final results.

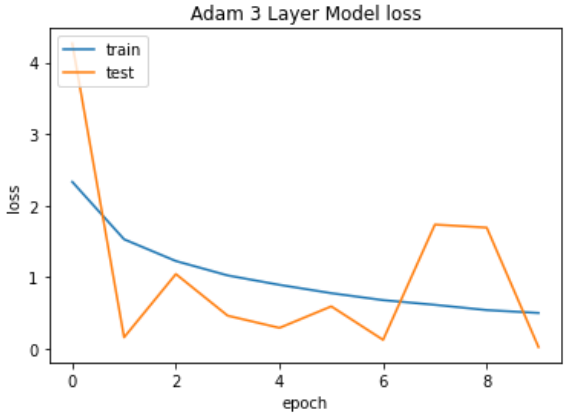
8. Appendix

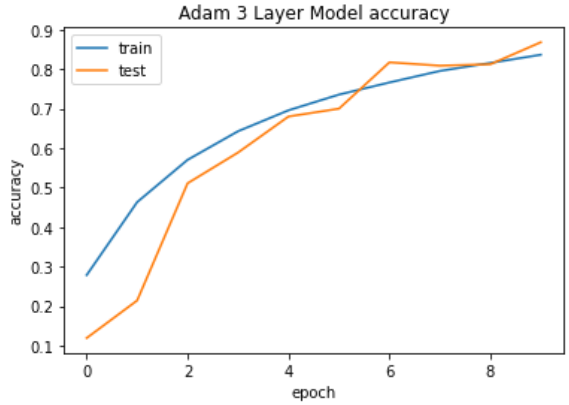

Adam 1 Layer Model accuracy
train
test
accuracy
epoch

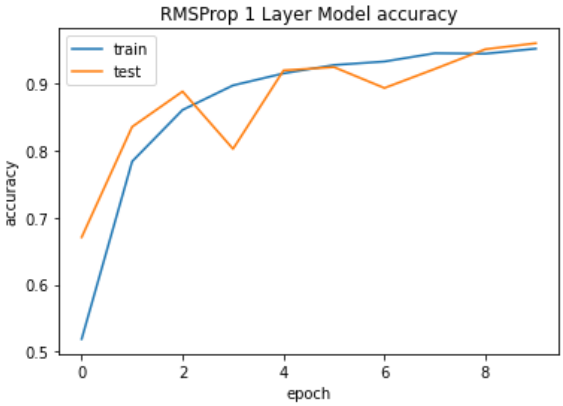

Adam 1 Layer Model loss
train
test
loss
epoch

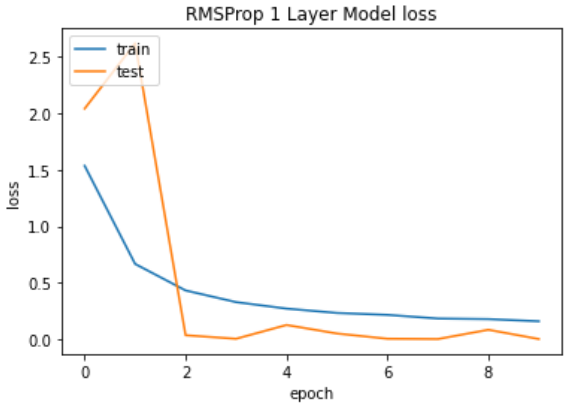

Adam 2 Layer Model accuracy
train
test
accuracy
epoch

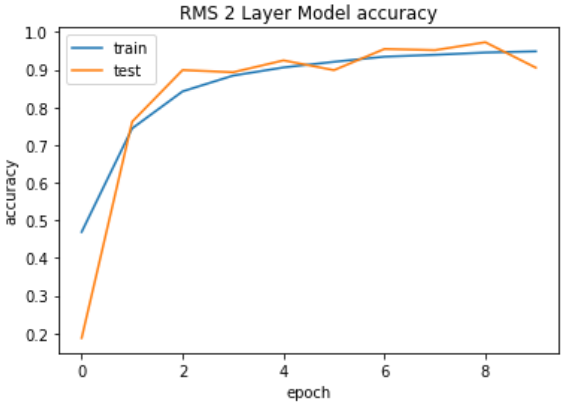

Adam 2 Layer Model loss
train
test
loss
epoch

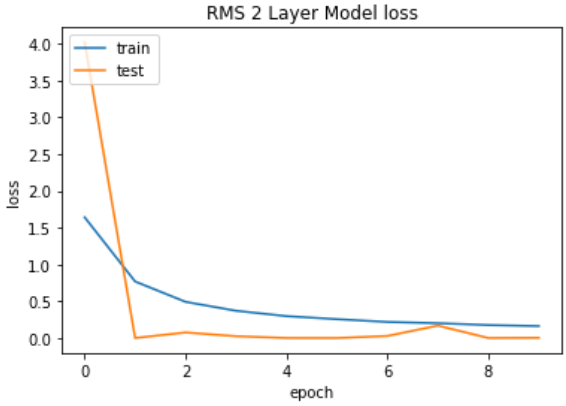

Adam 3 Layer Model loss
train
test
loss
epoch

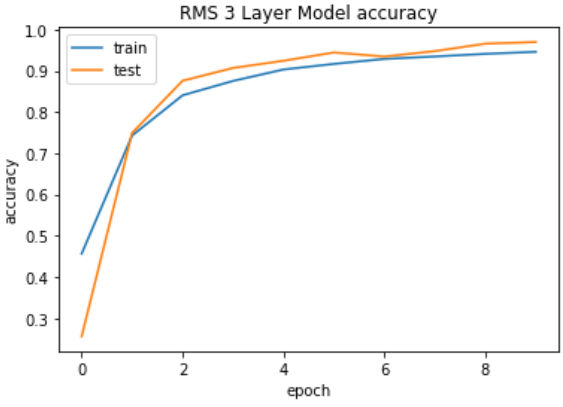

Adam 3 Layer Model accuracy
train
test
accuracy
epoch

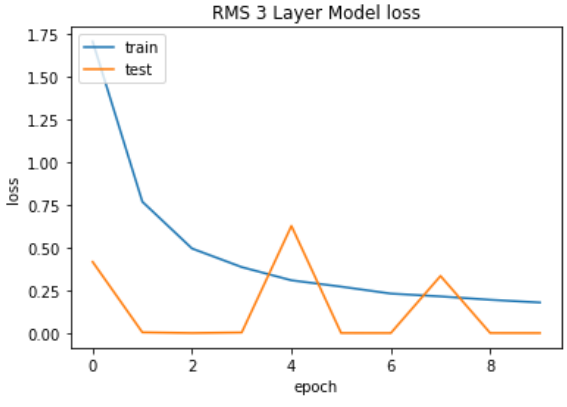

RMSProp 1 Layer Model accuracy
train
test
accuracy
epoch

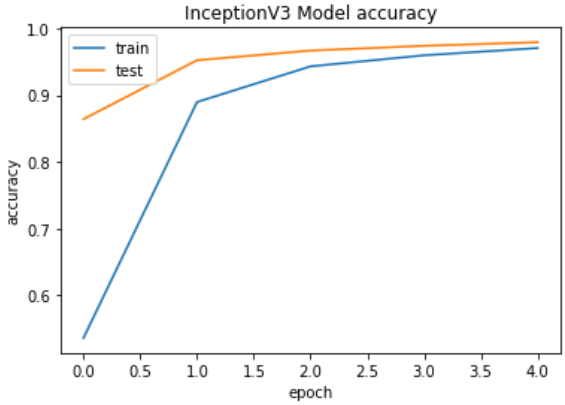

RMSProp 1 Layer Model loss
train
test
loss
epoch

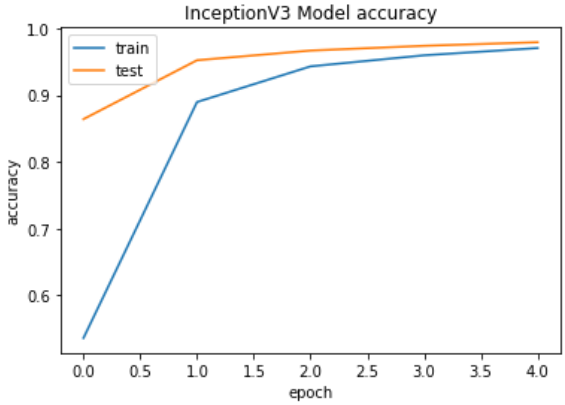

RMS 2 Layer Model accuracy
train
test
accuracy
epoch


RMS 2 Layer Model loss
train
test
loss
epoch


RMS 3 Layer Model accuracy
train
test
accuracy
epoch


RMS 3 Layer Model loss
train
test
loss
epoch


InceptionV3 Model accuracy
train
test
accuracy
epoch


InceptionV3 Model accuracy
train
test
accuracy
epoch