Visual Recognition

Assignment 5

Hyper Parameter Tuning: Resnet

Vasu Bansal
MT2018130
M.Tech CSE - 1st Year

Pavan Bharadiya
MT2018023
M.Tech CSE - 1st Year

April 26, 2019

Tuning Hyper Parameters for a basic RESNET architecture on CIFAR-10 dataset

Since the purpose of this assignment is to tune the hyper parameters we will not be changing the architecture except adding dropout layers where required.

Training, Validation and Test Data Setup

We are using the CIFAR-10 data set which has images of size (32, 32, 3).

Number of training images provided are 50,000 and number of test images provided are 10,000. We keep the training images as it but split the test set into test data set (80%) and validation data set (20%).

We split the test data set and not the training data set since the test and validation data set are supposed to come from the same distribution.

Table below summarizes the data sets:

Data Set		Percentage	Number of Images
Training Data		100 %	50000
Original Test Data	Test Data	80 %	8000
	Validation Data	20 %	2000

Table 0.1: Datasets setup

Model Comparison Criteria

Throughout the entire hyper parameter tuning process we compare the different models using "Categorical Accuracy" since we are doing multi-class classification.

We decided on a fixed accuracy measure since it provides us with a base to compare the different models without worrying about the different accuracy criteriaś.

Parameter Tuning Process

We list the parameter tuning process in the order that we followed without rearranging the steps, we do this to show how we reached a good tuned model that gives a high enough accuracy. We also include the hyper parameters and the approaches that we tried, but which failed to give good results to show how we developed intuition on what works and what not.

Code Files and Platforms

We started with Google Collab and Microsoft Notebooks, Microsoft Notebooks doesnt́ offer any GPU but they are more stable than Google Collab which tends to fail often. Later in the tuning process because of the increased time in training the network we shifted to Kaggle Kernels which provide more stability but fails to show the intermediate results of execution. Just for the Kaggle Kernels we provide links to the committed notebooks.

1. Tuning Depth - Number of Layers

The original RESNET architecture provided to us was 20 layers deep but it had the option to increase the number of layers to any depth of (6*n + 2) for any n. We tried a number of different depth combinations and below table summarizes these results for the test dataset.

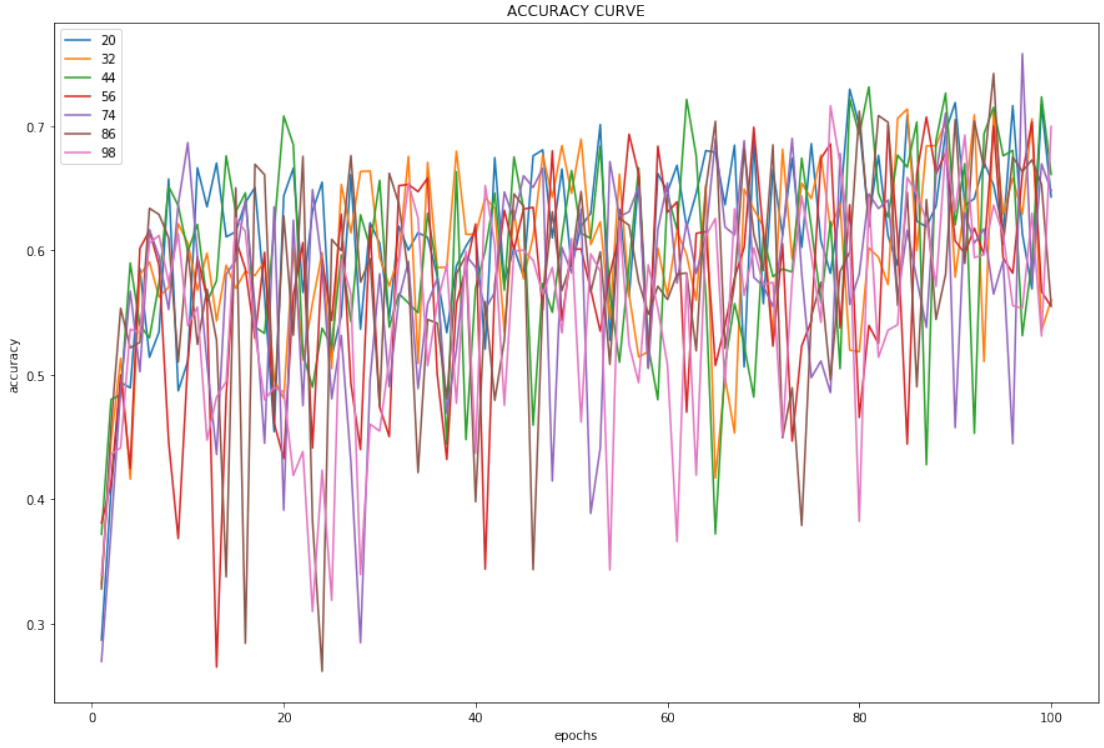
Depth	Loss	Test Accuracy
20	0.4396	0.6291
32	0.3282	0.6726
44	0.3862	0.6696
56	0.6099	0.5521
74	0.5838	0.5495
86	0.5838	0.5495
98	0.3556	0.6962

Table 0.2: Loss and accuracy for test dataset

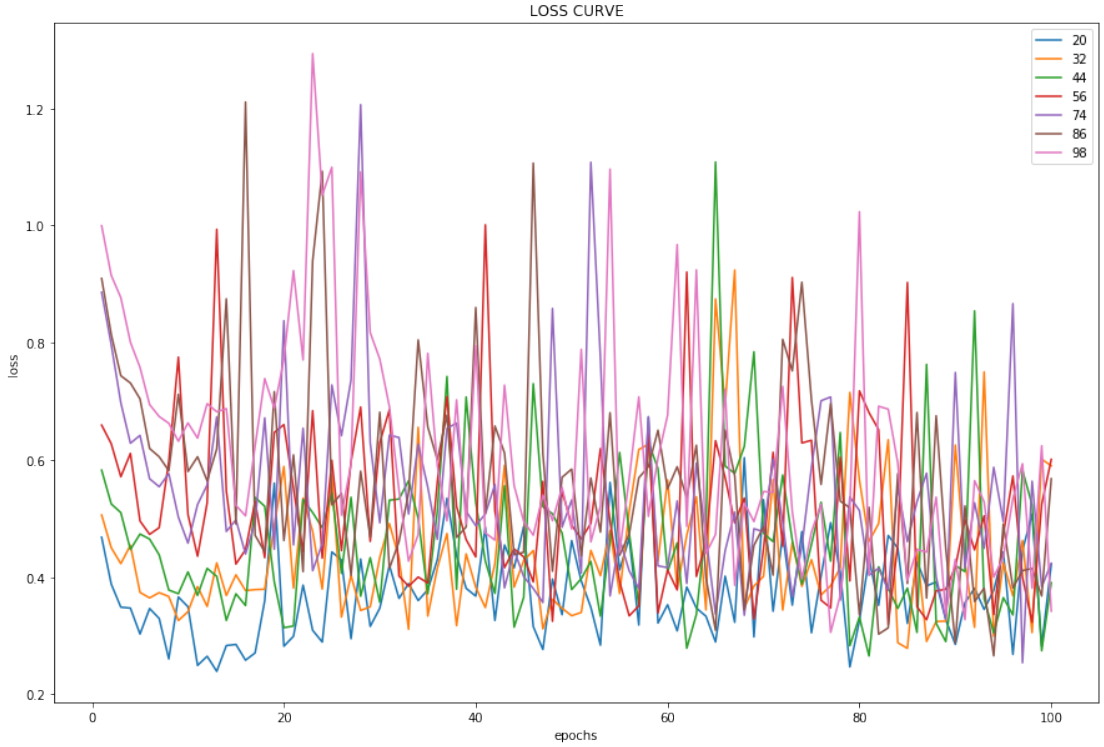
It is easily seen that depth of 98 gives the best accuracy followed by depth 32 and depth 20. So the natural choice will be to go with depth 98 but training a depth 98 model took an insane amount of time. Even training a depth 32 model took too much time which made it in-feasible to use in hyper parameter training process which requires lots and lots of continuous runs.

So we decided to go with depth 20 for the time being, it still takes appx. 90 mins to train for 100 epochs but using Google Collab, Microsoft Azure Notebooks and Kaggle Kernels it was manageable. ***Therefore, we decided to explore the depth hyper parameter a little later in the tuning process.***

Another reason for deferring this hyper parameter to a later stage was the below two figures comparing accuracy and loss for different depths. As much as they don't make sense to you they also didn't made much sense to us :-) The model exhibits too much randomness in both accuracy and loss to make a clear cut selection. Also the above table may have been filled with different values if we had decided to train it for more or less number of epochs. We explore it a little later when our model starts giving us some consistent results.



(a) Accuracy plot for different depthś



(b) Loss plot for different depths

Figure 0.1: Accuracy and loss plots for different depths

2. Optimizer Function

We tried various optimizers to try and find the one which best suites to our problem. The optimizers we tried along with their corresponding loss and categorical accuracy on the test dataset are summarized in the below table:

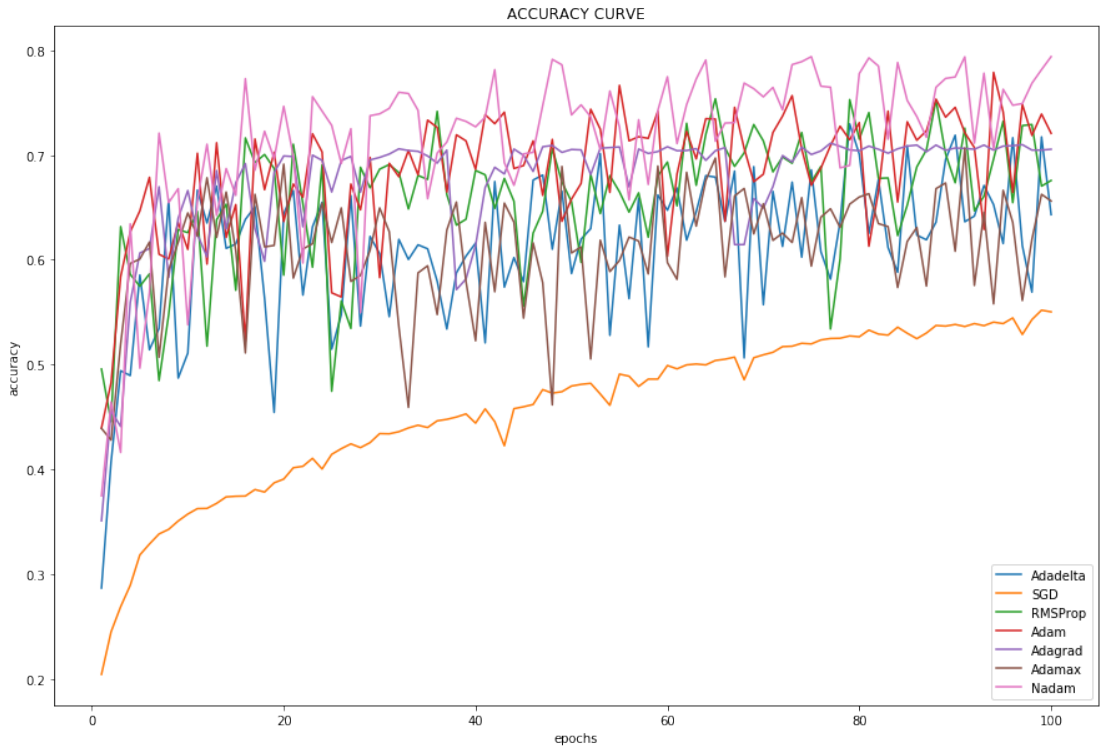
Optimizer	Loss	Accuracy
Adadelta	0.4396	0.6291
SGD	0.3432	0.5426
RMSProp	0.3799	0.6578
Adagrad	0.3140	0.6953
Adam	0.2913	0.7167
Adamax	0.3980	0.6332
Nadam	0.1853	0.7874

Table 0.3: Loss and accuracy on test dataset for different optimizers

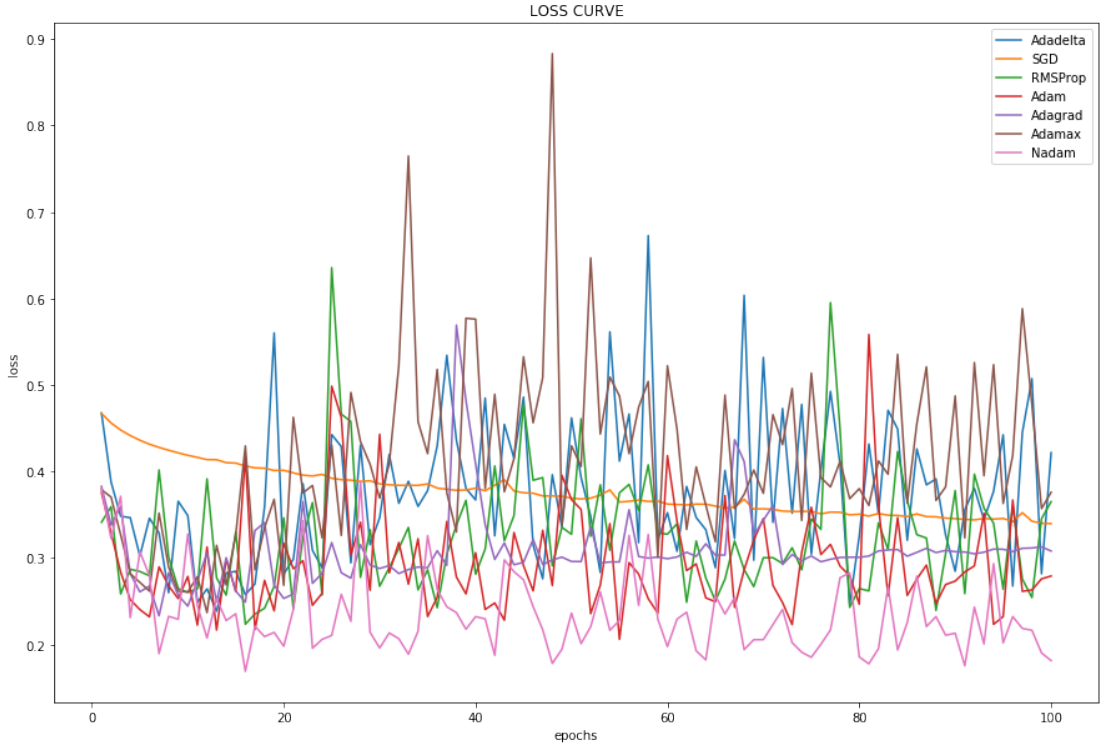
On the first look it looks like "Nadam" is the best optimizer, but if we carefully look at the plots given below for accuracy and loss for the validation dataset we see that "Nadam" still clearly wins but it has a lot of variation thus making the optimizer unreliable since training the model for different number of epochs might lead to different results!

On the other hand if we look at the curves for accuracy and loss for "SGD" (Stochastic Gradient Descent) optimizer we couldn't ask for better looking curves. Agreed it is not the best but on careful examination we can see that the curves are still sloping even after 100 epochs thus indicating the room for improvement. If we apply the same argument to other optimizer curves we can see that even with their random variations they tend to stabilize much before reaching 100 epochs, by stabilize we mean stabilize in a given range for their variations.

Thus, what we tried further was to explore both SGD and Nadam, for SGD we try to improve it further using various learning techniques using which we succeeded and were able to even beat Nadam. We tried tuning Nadam to try and control the variations but it turned out Nadam is too sensitive to any parameter updates, we tried a bunch of different carefully selected variations but even changing them a little caused the accuracy to drop a lot so we went ahead with SGD but later we tried using Nadam again hoping that tuning other hyper parameters would have controlled it variations a little, we list those results later in the tuning process.



(a) Accuracy plot for different optimizerś



(b) Loss plot for different optimizers

Figure 0.2: Accuracy and loss plots for different optimizers

3. Adaptive Learning Rate Methods

We try and improve the SGD optimizer by using different learning rate schedules. Here we explore the different learning rate schedules and their combinations. We tried just step decay and it gave good results and we further tried itś combinations. The following list shows the different combinations that we tried and we use the Step decay schedule as the baseline for comparing the different methods:

1. SGD with Step Decay and Time Decay
2. SBD with Step Decay and Exponential Decay

3A. SGD with Step Decay and Time Decay

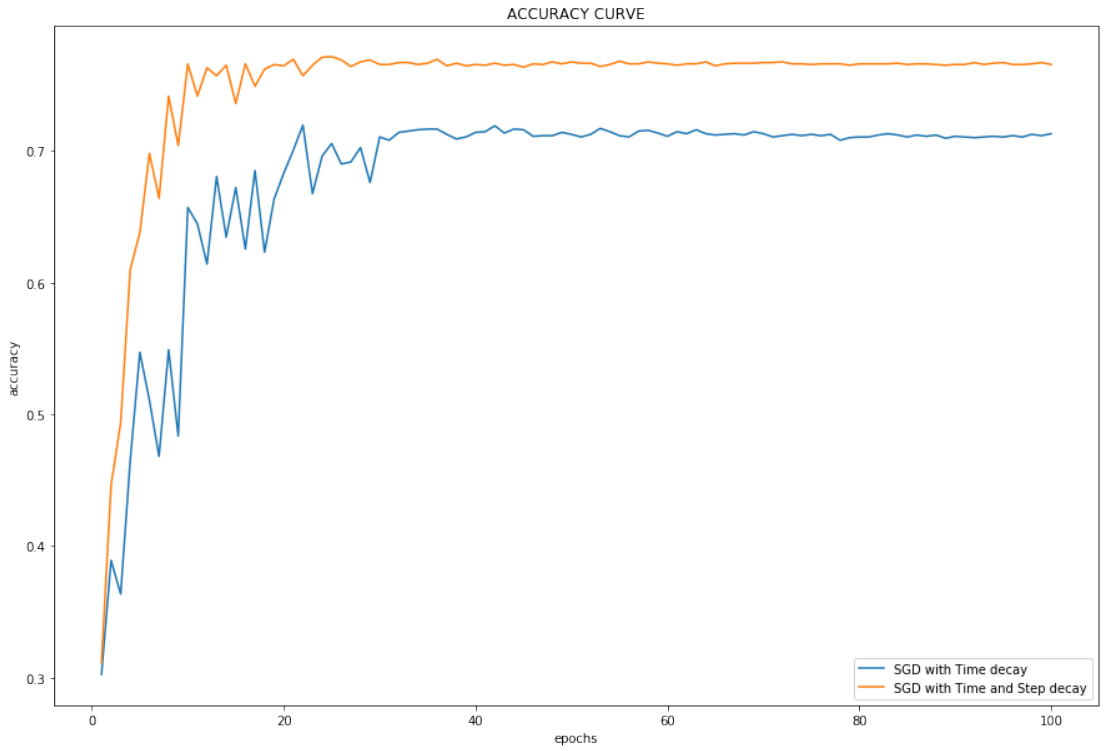
We compare the SGD optimizer with just step decay and the step-time decay combination. The following table summarizes the results:

Learning Rate Schedule	Loss	Categorical Accuracy
Step Decay	1.5455	0.7455
Step Decay with Time Decay	1.3145	0.7528

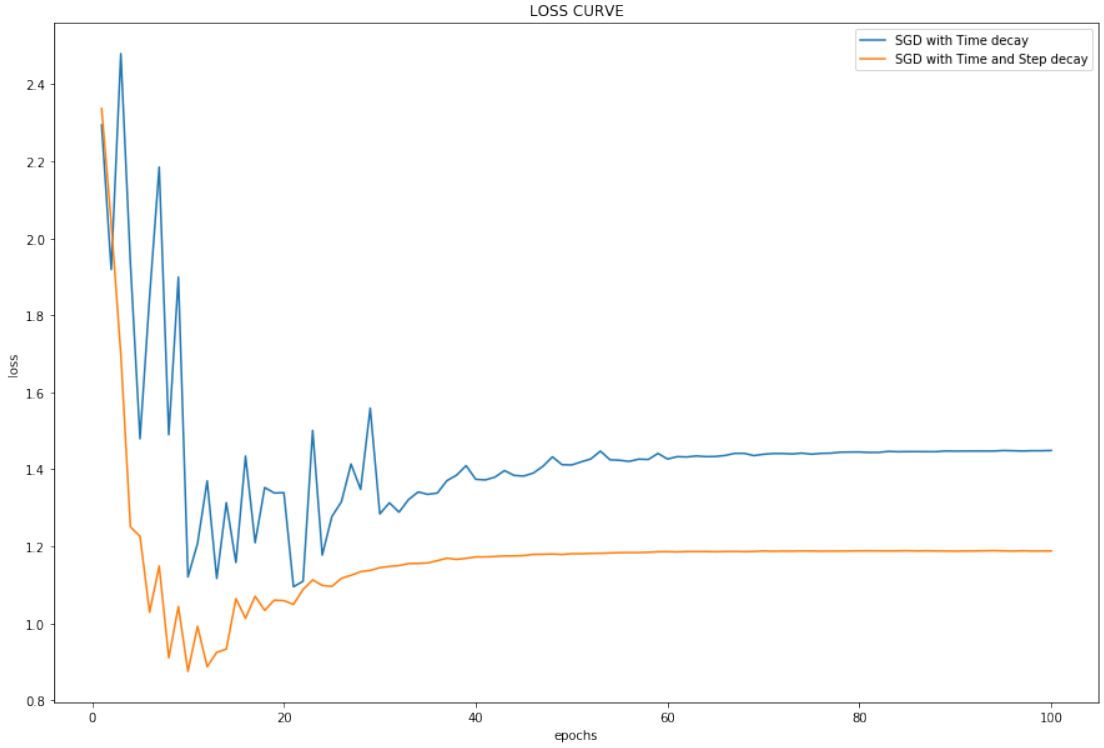
Table 0.4: Loss and accuracy on test dataset for step and step-time decay

The table clearly shows that SGD with step and time decay combination gives great results even comparable with Nadam. Also the below plots strengthen the argument as the differences between these two methods are clearly visible. Also the results are super consistent as per the plots.

Therefore, moving forward we stick with time and step decay adaptive learning rate method.



(a) Accuracy plot for time decay and step decay combinations



(b) Loss plot for time decay and step decay combinations

Figure 0.3: Accuracy and loss plots for time decay and step decay combinations

3B. SGD with Time Decay and Exponential Decay

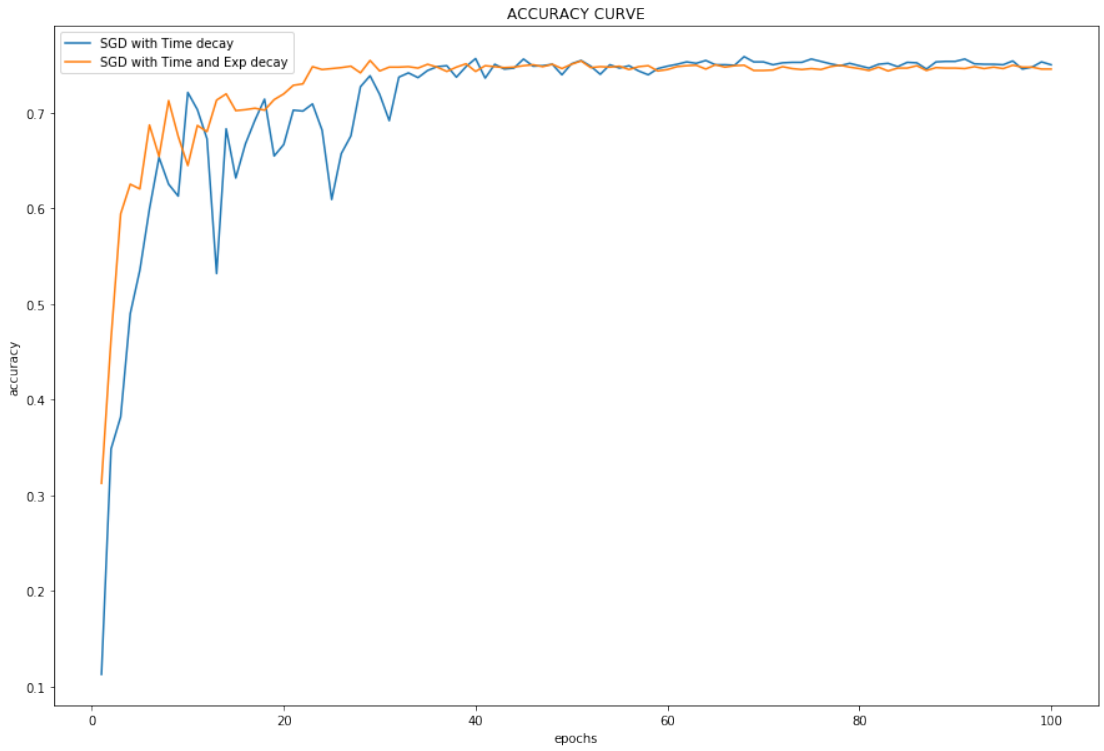
We compare the SGD optimizer with just step decay and the step-exponential decay combination. The following table summarizes the results:

Learning Rate Schedule	Loss	Categorical Accuracy
Step Decay	1.5455	0.7455
Exponential Decay with Time Decay	1.3216	0.75025

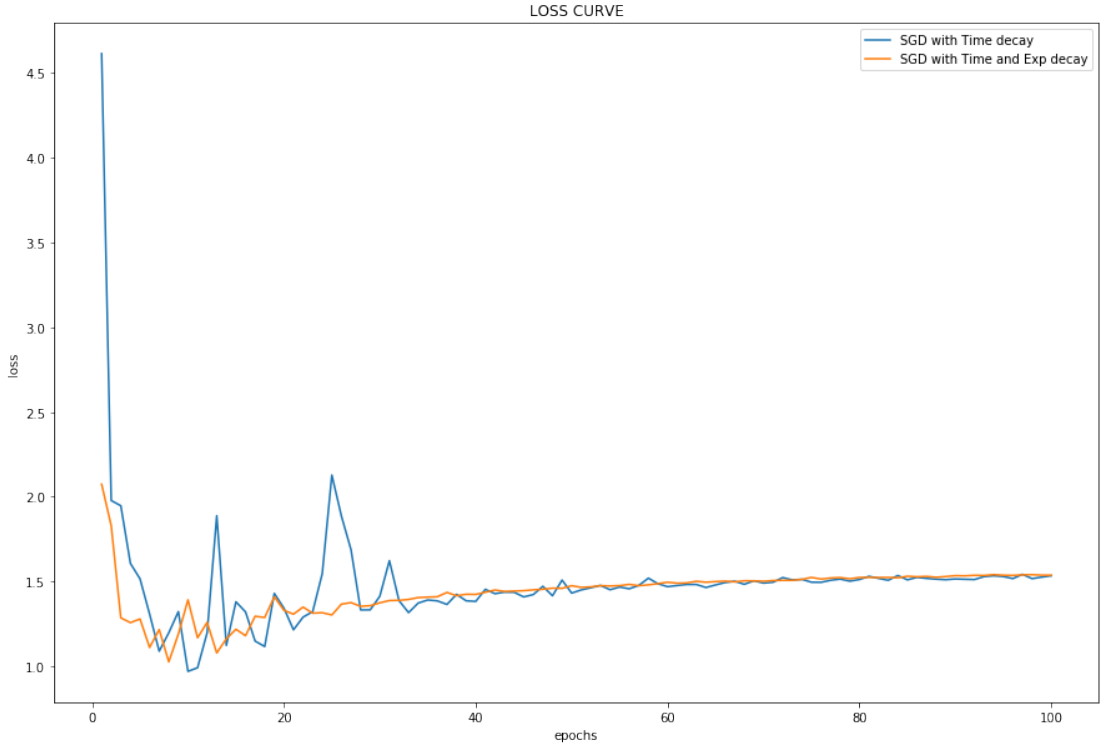
Table 0.5: Loss and accuracy on test dataset for time and exponential-time decay

The table clearly shows that SGD with exponential and time decay combination gives great results even comparable with Nadam. The below plots show that the difference is almost negligible but we still decided to explore this adaptive learning method because it showed improvement on the test data set almost equal to the step-time combination as evident by the tables above. Later we will see that it gives good results when used with normalization.

Therefore, moving forward we also try time and exponential decay adaptive learning rate method along with the time and step decay combination untill we can find a clear winner.



(a) Accuracy plot for time decay and exponential decay combinations



(b) Loss plot for time decay and exponential decay combinations

Figure 0.4: Accuracy and loss plots for time decay and exponential decay combinations

4. Normalization Techniques

Here we try different normalization techniques on the above two adaptive learning methods. The different normalization techniques that we tried are listed below:

- Min-max normalization
- Standard deviation normalization
- Layer-wise standard deviation normalization
- Layer and image wise standard deviation normalization

4A. Step-Time Decay + Normalization Techniques

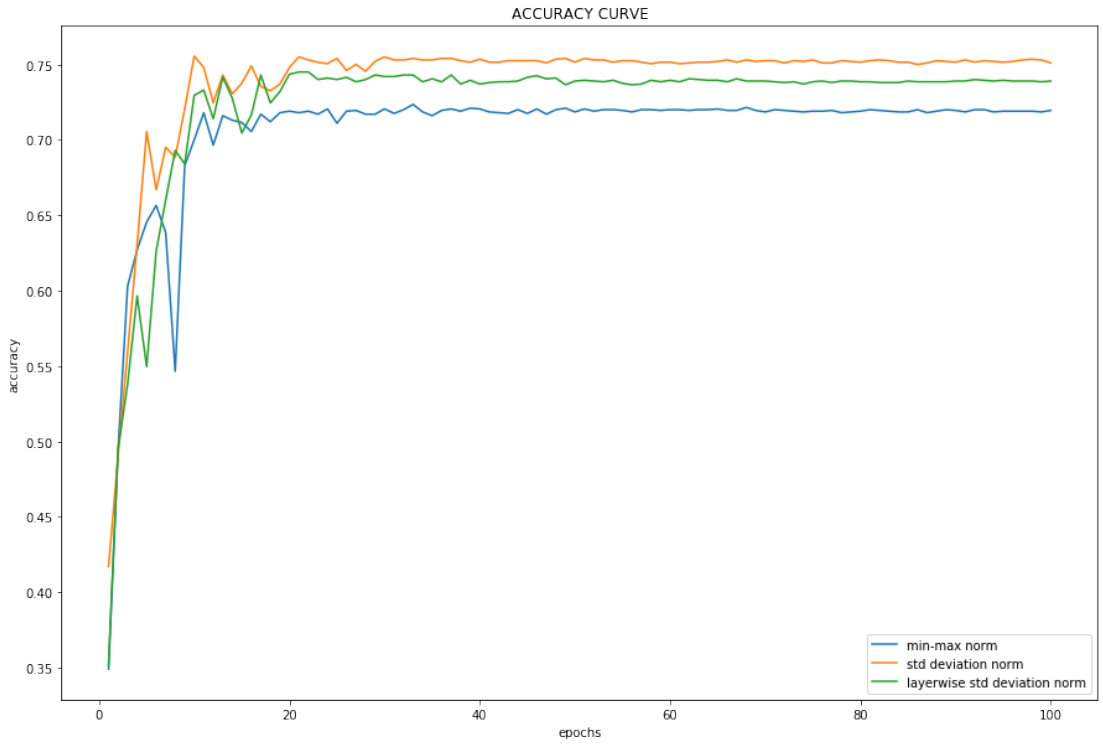
The table below summarizes the results obtained for different normalization techniques:

Normalization Technique	Loss	Categorical Accuracy
min-max	2.591	0.7112
standard dev norm	1.3209	0.7503
layerwise standard dev norm	1.303	0.7308

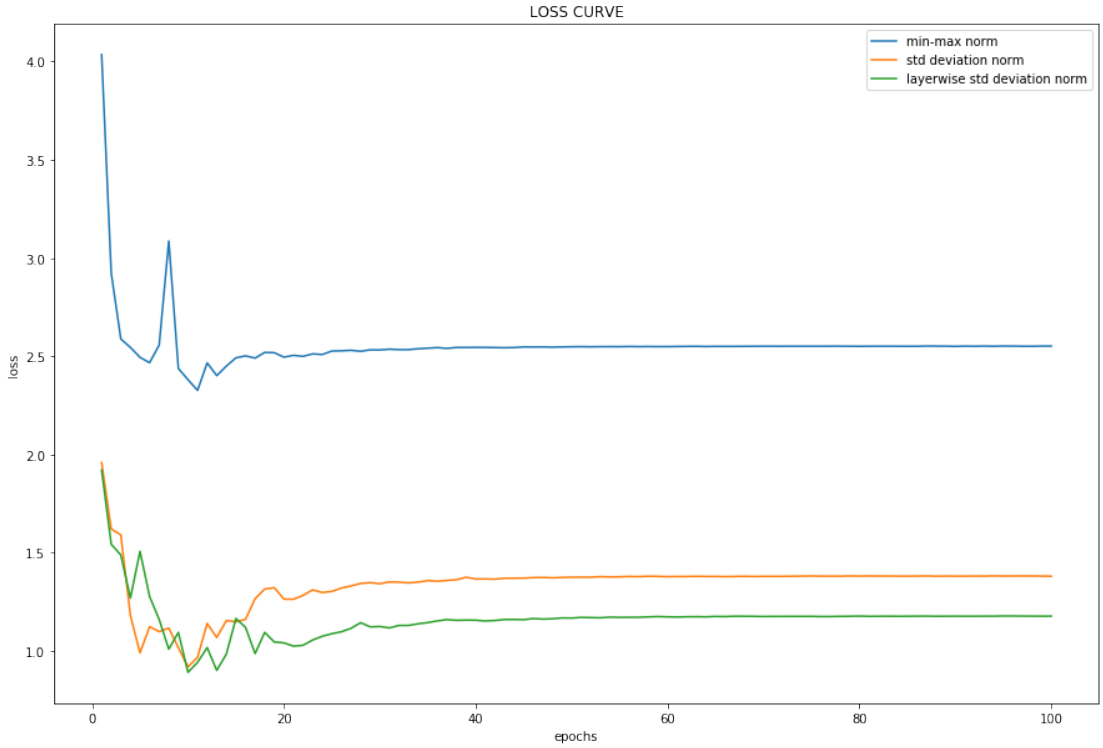
Table 0.6: Loss and accuracy on test dataset for different normalizations

As clearly evident from the table above and also by the accuracy and the loss plots below standard deviation normalization is the clear winner.

Therefore, moving forward we use standard deviation normalization with time-step adaptive learning method.



(a) Accuracy plot for time-step decay using different normalization techniques



(b) Loss plot for time-step decay using different normalization techniques

Figure 0.5: Accuracy and loss plots for time-step decay using different normalization techniques

4B. Step-Exponential Decay + Normalization Techniques

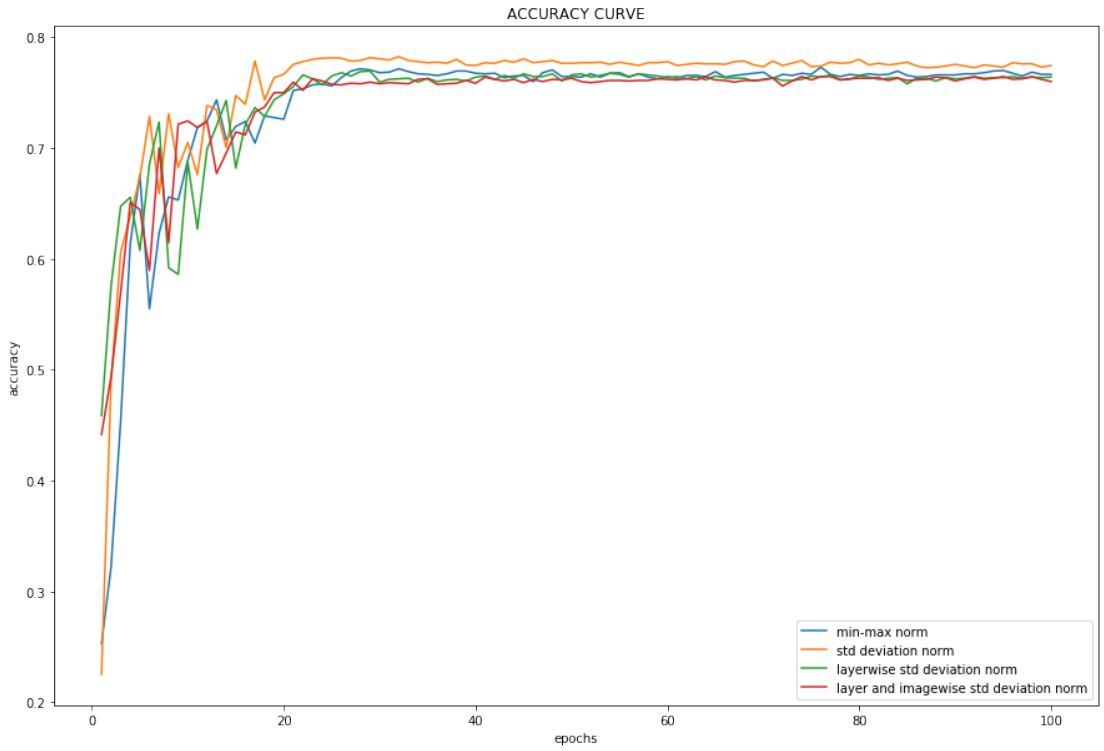
The table below summarizes the results obtained for different normalization techniques:

Normalization Technique	Loss	Categorical Accuracy
min-max	1.4479	0.7611
standard dev norm	1.0693	0.7635
layerwise standard dev norm	1.477	0.7225
layer and image wise standard dev norm	1.4735	0.75375

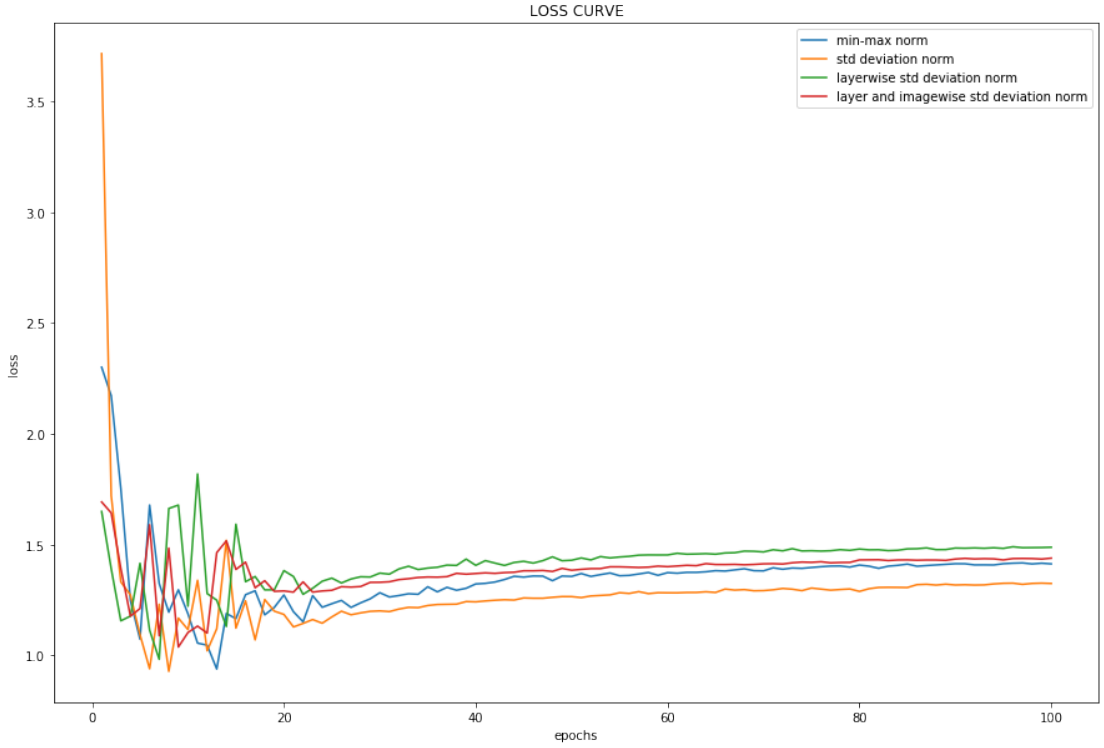
Table 0.7: Loss and accuracy on test dataset for different normalizations

As clearly evident from the table above and also by the accuracy and the loss plots below standard deviation normalization is the clear winner.

Therefore, moving forward we use standard deviation normalization with both time-step and time-exponential adaptive learning methods.



(a) Accuracy plot for time-exponential decay using different normalization techniques



(b) Loss plot for time-exponential decay using different normalization techniques

Figure 0.6: Accuracy and loss plots for time-exponential decay using different normalization techniques

5. Playing around with regularization

After playing around with the model for so long we realized that our training accuracy was touching 100% after appx. 50 epochs and the validation accuracy was becoming kind of constant which made sense since the model was not able to learn more after reaching the 100% accuracy mark.

Thus we decided to play around with the regularization. For regularizing our model we tried the following techniques:

1. Modifying l2 regularizer parameter

   We tried using l1 and l1_l2 but they gave bad results so skipping their description.

2. Dropout Layers with different keep_prob and with different l2 parameters using time-step learning schedule

3. Dropout Layers with different keep_prob and with different l2 parameters using exponential-time learning schedule

5A. Modifying l2 regularizer parameter

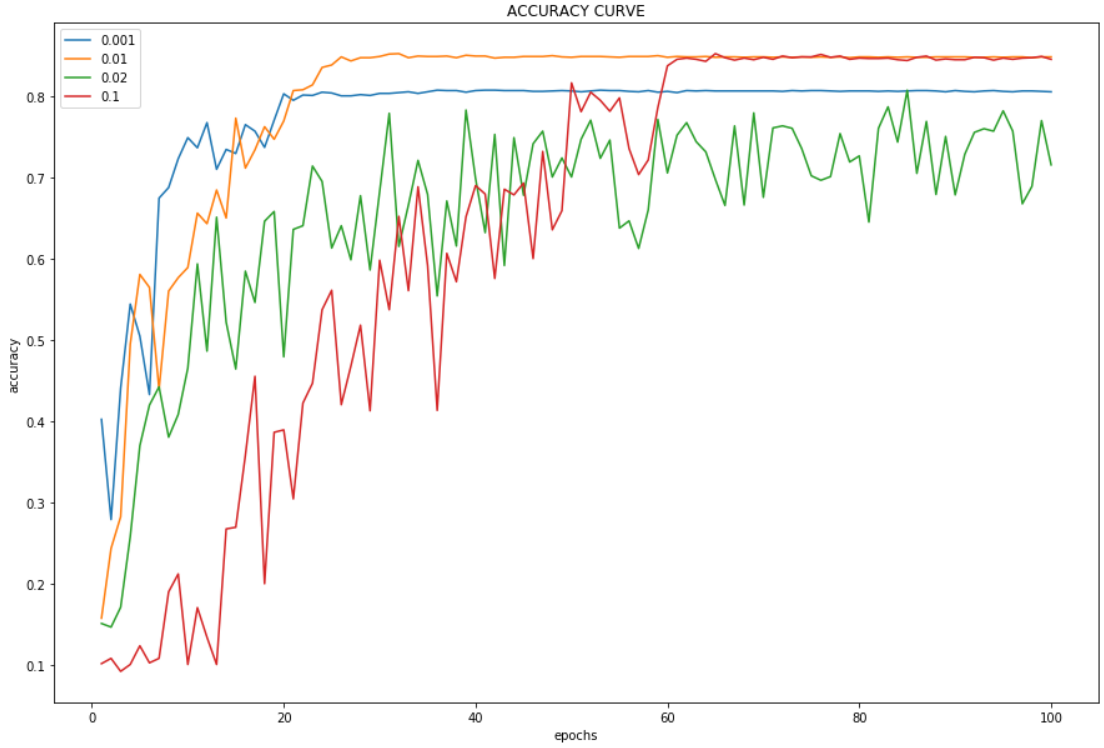
We tried using different values for l2. Below table summarizes the results obtained:

	Loss	**Categorical Accuracy**
0.1	0.5508	0.8541
0.01	0.6214	0.8552
0.001	1.0015	0.8192
0.0001	Default we have been using	

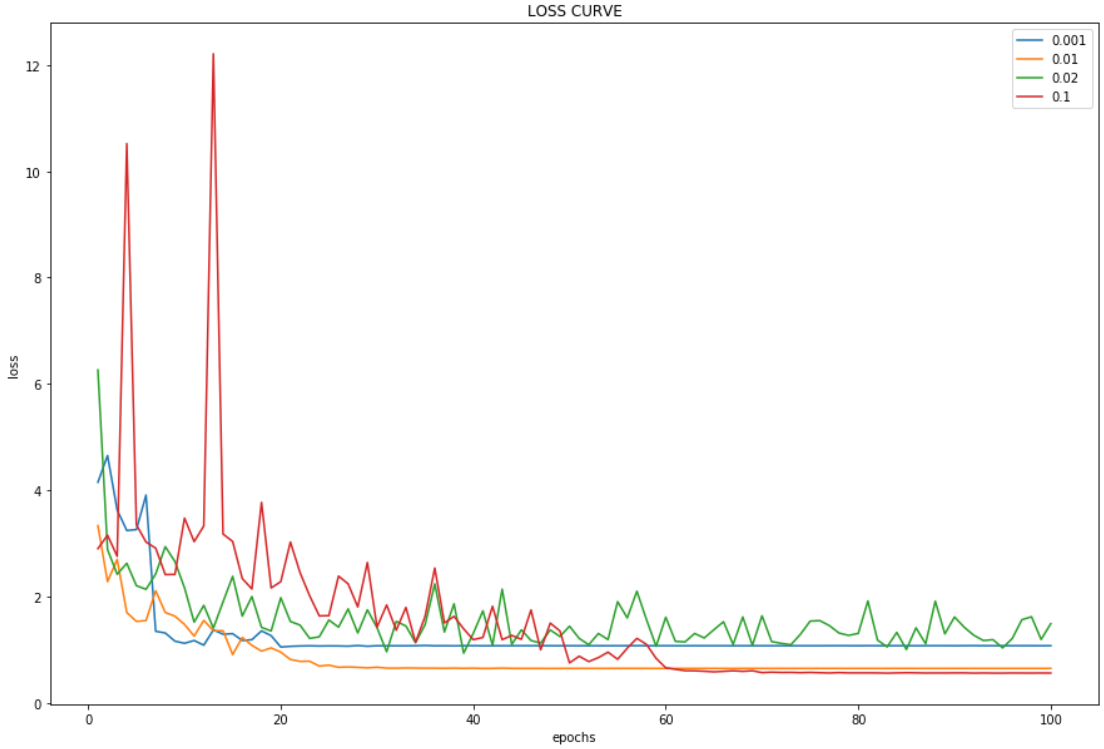
Table 0.8: Loss and accuracy on test dataset for different values of l2 parameter

The above table shows that the values 0.1 and 0.01 give the best results. The plot below shows that 0.1 is not very stable for the initial epochs of training but becomes stable after appx. 50 epochs of training.

We consider both 0.1 and 0.01 but given an option we always go ahead with 0.01 given that it stabilizes early.



(a) Accuracy plot for different values of l2 regularizer parameter



(b) Loss plot for different values of l2 regularizer parameter

Figure 0.7: Accuracy and loss plots different values of l2 regularizer parameter

5B. Dropout Layers with different keep_prob and with different l2 parameters using time-step learning schedule

We tried different combinations for keep_prob and l2 regularizer parameter. Below table summarizes the results obtained:

l2 parameter	dropout (1-keep_prob)	Loss	Categorical Accuracy
0.1	0.1	0.45711	0.882625
0.1	0.08	0.4597	0.8863
0.01	0.1	0.45881	0.891
0.01	0.08	0.512	0.8875

Table 0.9: Loss and accuracy on test dataset for different values of l2 parameter and keep_prob

As seen from the table above l2 parameter as 0.01 with (1 - keep_prob) as 0.1 is the clear winner. The same results can be inferred from the plots below.

Moving forward we use l2 parameter as 0.01 with (1 - keep_prob) as 0.1 in some of our future models. We write some because on further exploration with data augmentation removing the dropout gave better results. We will discuss these results in detail later.

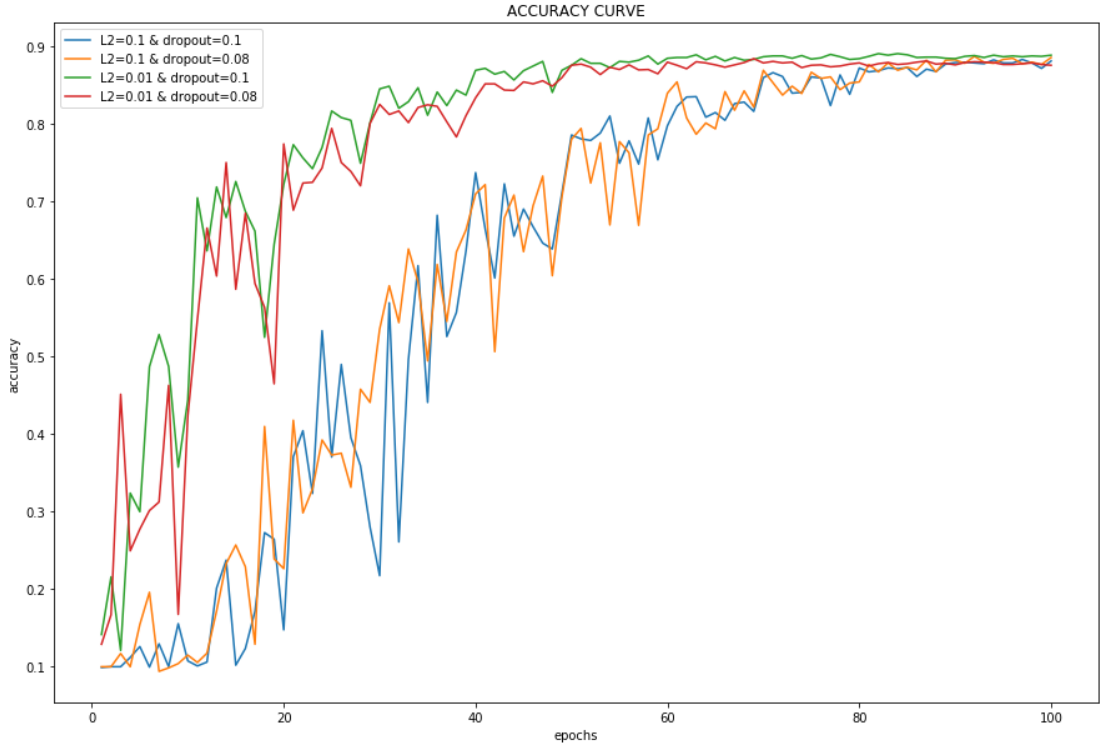
5C. Dropout Layers with different keep_prob and with different l2 parameters using time-exponential learning schedule

We tried different combinations for keep_prob and l2 regularizer parameter. Below table summarizes the results obtained:

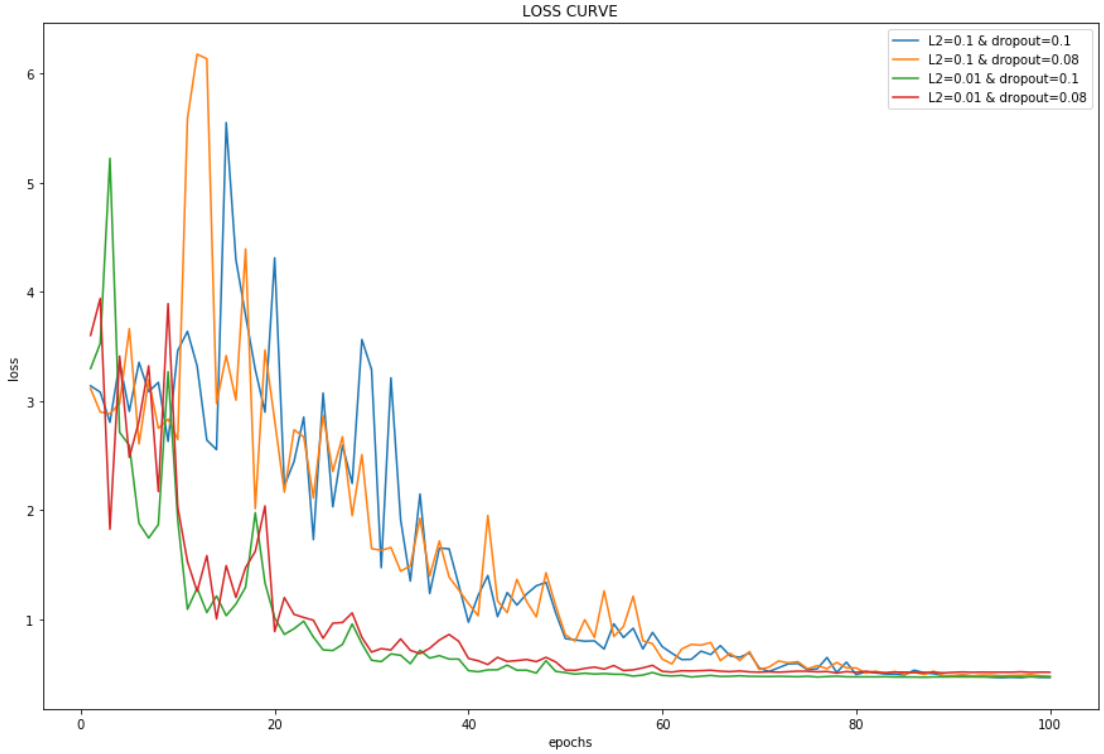
l2 parameter	dropout (1-keep_prob)	Loss	Categorical Accuracy
0.01	0.1	0.4877	0.8775
0.01	0.15	0.4566	0.8855
0.01	0.2	0.5012	0.8125
0.01	0.25	0.5674	0.7913

Table 0.10: Loss and accuracy on test dataset for different values of l2 parameter and keep_prob

Since the results are very much comparable with time-step decay learning schedule and the number of variations to try are getting too large we drop this chain of parameter tuning and continue with the step-decay learning schedule chain.



(a) Accuracy plot for different values of l2 regularizer parameter and keep_prob



(b) Loss plot for different values of l2 regularizer parameter and keep_prob

Figure 0.8: Accuracy and loss plots different values of l2 regularizer parameter and keep_prob

6. Data Augmentation

We try data augmentation techniques to increase out training data size in hope of getting better results. With used the Keras inbuilt data augmenter "ImageDataGenerator".

We tried different variations of data augmentation techniques with different regularization methods since using data augmentation was over-fitting the training data. Of all the many different combinations of values we tried we just report the best ones to keep the report readable.

Following table summarizes the best of the different variations we tried:

Data Augmentation Parameters	Dropout	Loss	Categorical Accuracy
width_shift_range=0.1, height_shift_range=0.1	0.1	0.3834	0.906
horizontal_flip=True	0.08	0.3921	0.9038
fill_mode="nearest"	NA	0.37111	0.913
with vertical_flip=True	NA	0.4809	0.8735
with shear_range=0.01	NA	0.3875	0.9053
with rotation_range=15	NA	0.4012	0.9002

Table 0.11: Loss and accuracy on test dataset for different variations of data augmentation

From the above table it is clearly visible that data augmentation works like a charm and bumps up our accuracy on the test set to 91.3%. There is just a small kickback from our previous tuning process that we had to remove the tuned dropout layers. This suggests that we have to play around more with the regularization.

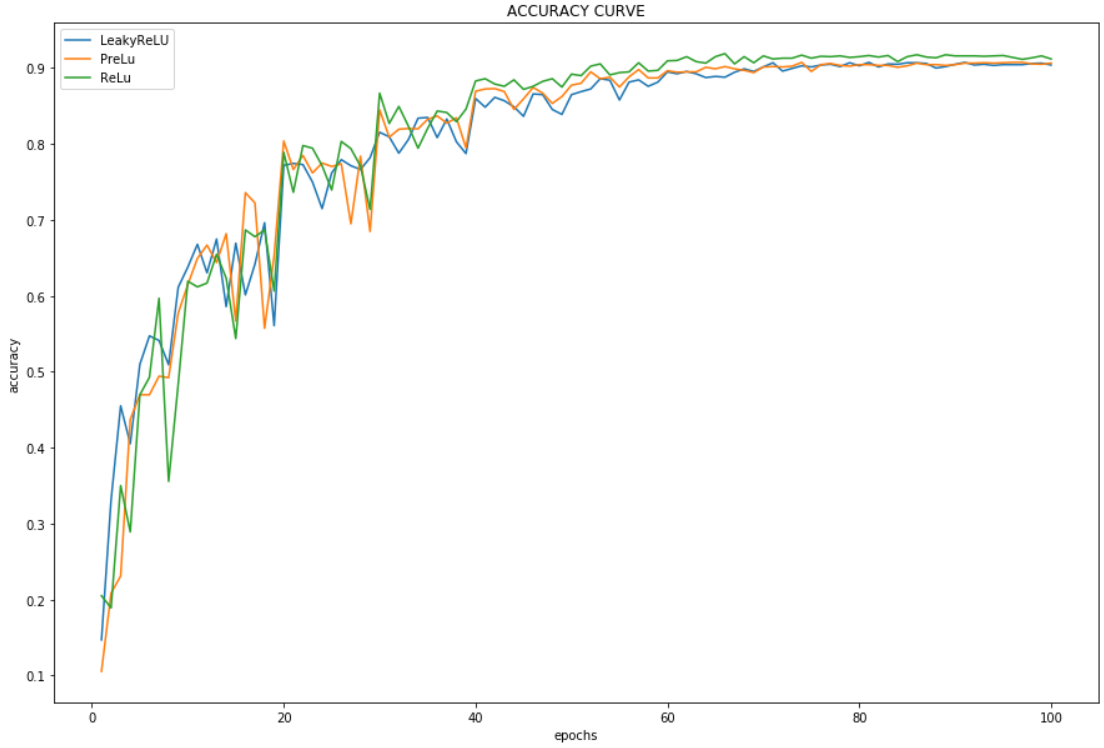
7. Trying out the different activation functions

We try out other activation functions than ReLU. Following table summarizes the results obtained for the different variations:

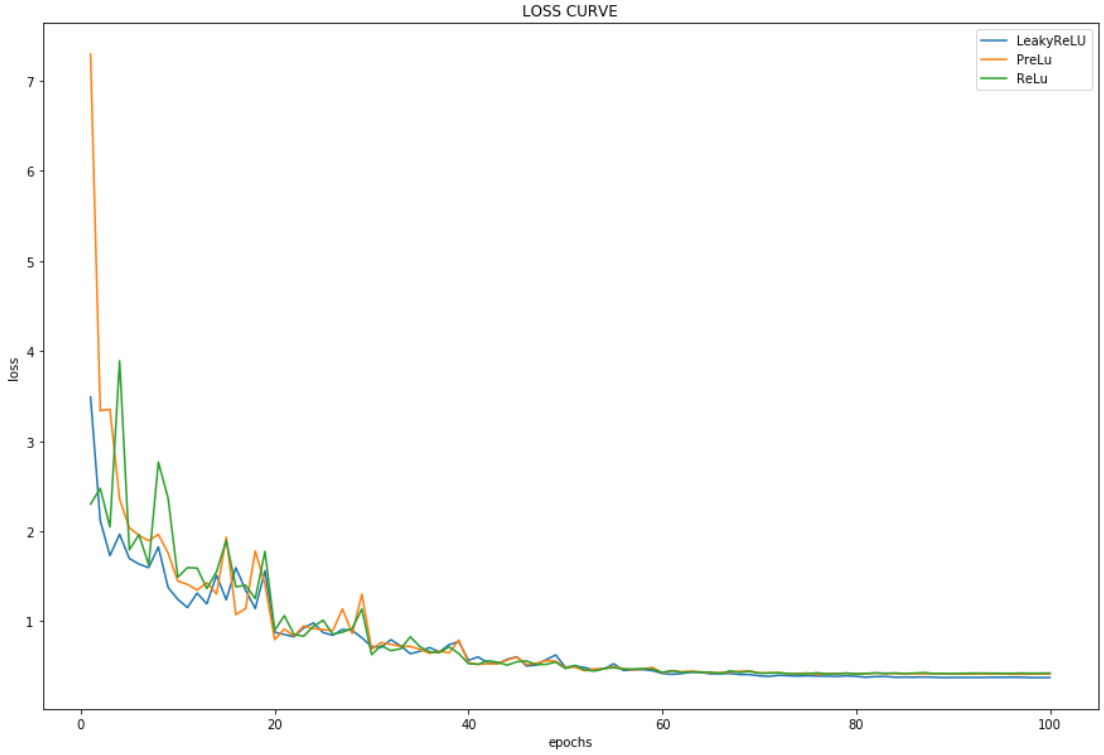
Activation Function	Loss	Accuracy
ReLU	0.37111	0.9130
PReLU	0.913	0.9006
LeakyReLU	0.3803	0.9063

Table 0.12: Loss and accuracy on test dataset for different variations of ReLU

The above table and the below figures clearly shows that ReLU is the winner.



(a) Accuracy plot for different variations of ReLU activation function



(b) Loss plot for different variations of ReLU activation function

Figure 0.9: Accuracy and loss plots different variations of ReLU activation function

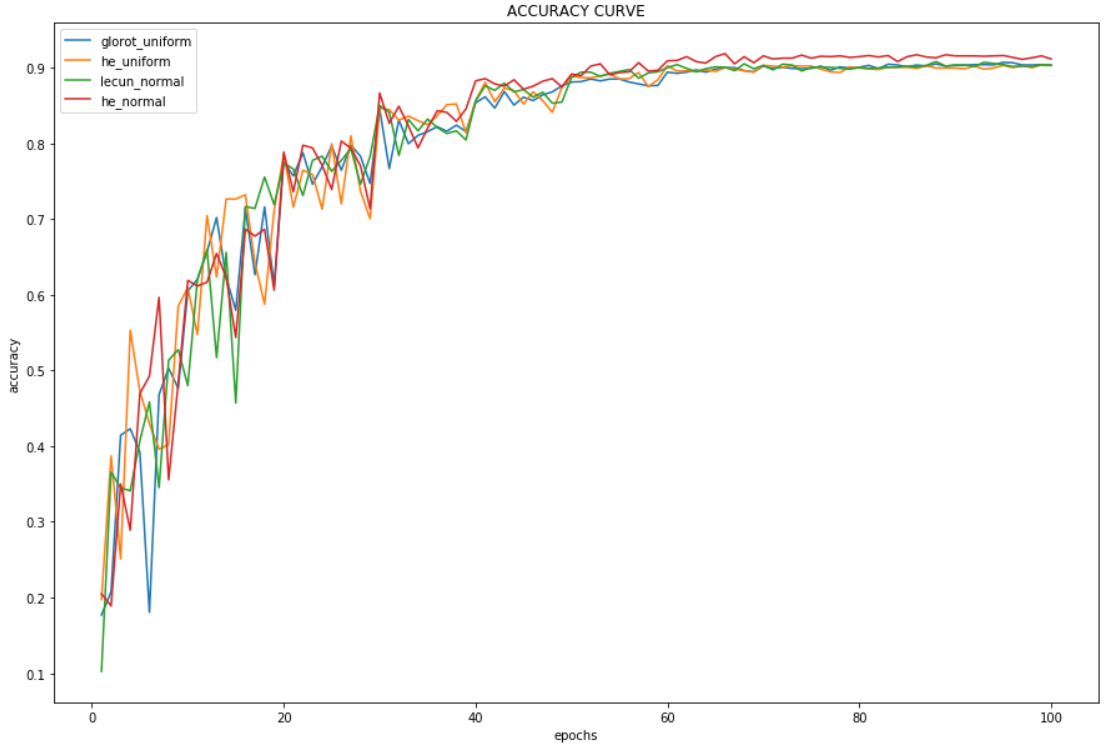
8. Trying out the different weight initialization techniques

We try out different weight initialization techniques other than he_normal. Following table summarizes the results obtained for the different variations:

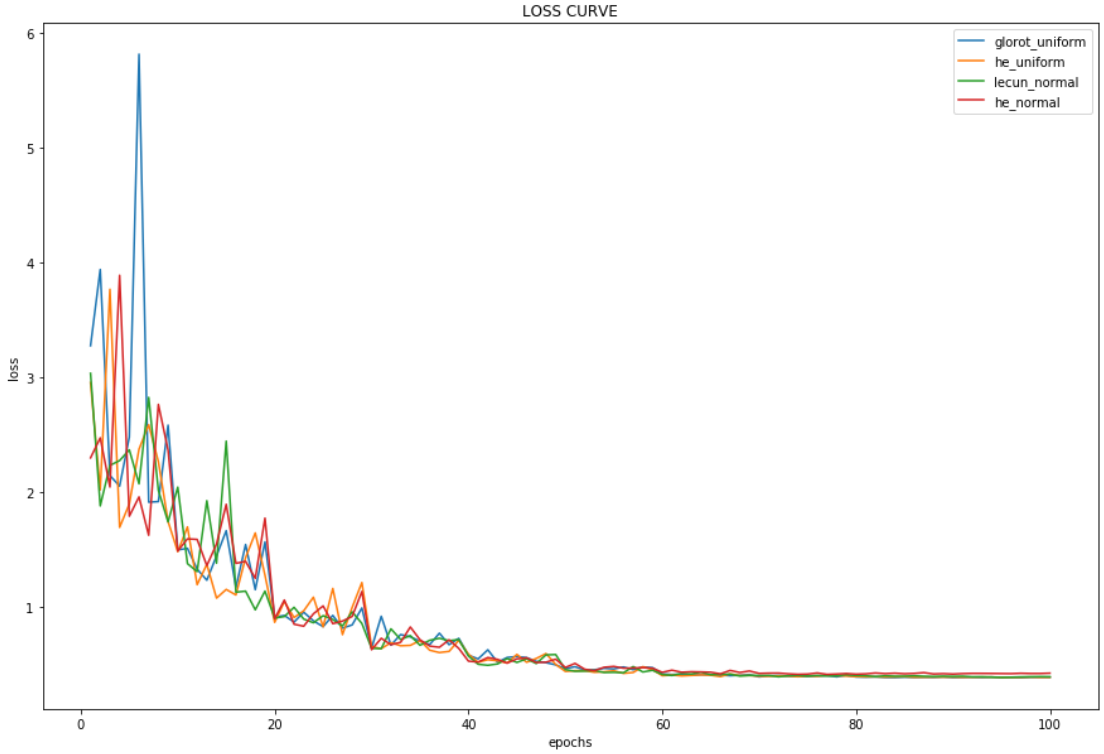
Weight Initializer's	Loss	Accuracy
he_normal	0.3711	0.9130
he_uniform	0.3942	0.9013
glorot_uniform	0.3953	0.9011
lecun_normal	0.3749	0.909

Table 0.13: Loss and accuracy on test dataset for different weight initialization techniques

The above table and the below figures clearly shows that he_normal is the winner followed closely by lecun_normal.



(a) Accuracy plot for different weight initialization techniques



(b) Loss plot for different weight initialization techniques

Figure 0.10: Accuracy and loss plots for different weight initialization techniques

9. Trying out bigger augmented training data size

We tried increasing the augmented training data size and the below table summarizes the results:

Training Data Size	Loss	Accuracy
1562 (one we were using)	0.3711	0.9130
2000	0.3783	0.9025
5000	0.3466	0.9075

Table 0.14: Loss and accuracy on test dataset for different augmented training data size

The above table shows that increasing the training data size doesnt́ helps much this is mainly because the model starts overfitting and we need to train bigger networks with better tuned regularization parameters. We explore this in the next and final step.

10. Trying out bigger tuned models

Since increasing the augmented training data size didńt increase the accuracy much we increase the depth of the model to accommodate for the larger training data. We tried a lot of different combinations but here we summarize just those models for which we got a good enough accuracy:

Depth	Other Variations	Loss	Accuracy
32	NA	0.3933	0.9122
32	Time-exponential LR decay with lecun_normal weight initializations	0.4062	0.9095
44	NA	0.4234	0.9151
44	L2 regularization parameter as 0.2	0.4446	0.9032
44	L2 regularization parameter as 0.2 with time-exponential LR decay	0.4615	0.8947
56	lecun_normal weight initializations with dropout 0.2	0.4344	0.9048
56	lecun_normal weight initializations	0.43503	0.916

Table 0.15: Loss and accuracy on test dataset for different model depths

The above table shows that depth 56 model gives the best results.

SUMMARY

The best model accuracy that we got is 91.6%.

The model can be found at the following Kaggle Kernel: here

Following parameters summarize the datset:

1. Weight Initiatializer: 'lecun_normal'
2. Regularizer: l2(0.01)
3. Depth: 56
4. Epochs: 200
5. Optimizer: SGD with Time-Step Decay
6. Loss: 'categorical_loss'
7. Accuracy Measure: 'categorical_accuracy'
8. Augmentation Techniques: ImageDataGenerator
9. Train Data Size: 1562

FURTHER POSSIBLE IMPROVEMENTS

We wanted to try more bigger networks with large augmented training data with more augmentation techniques and with larger number of epochs. But we are constrained on the platforms that are available to us at the current time. Even Kaggle which we found out to be the most stable platform stops our jobs after 9 hours of training.

We still ran those models for a small number of epochs and the below plots show the potential for further improvement.

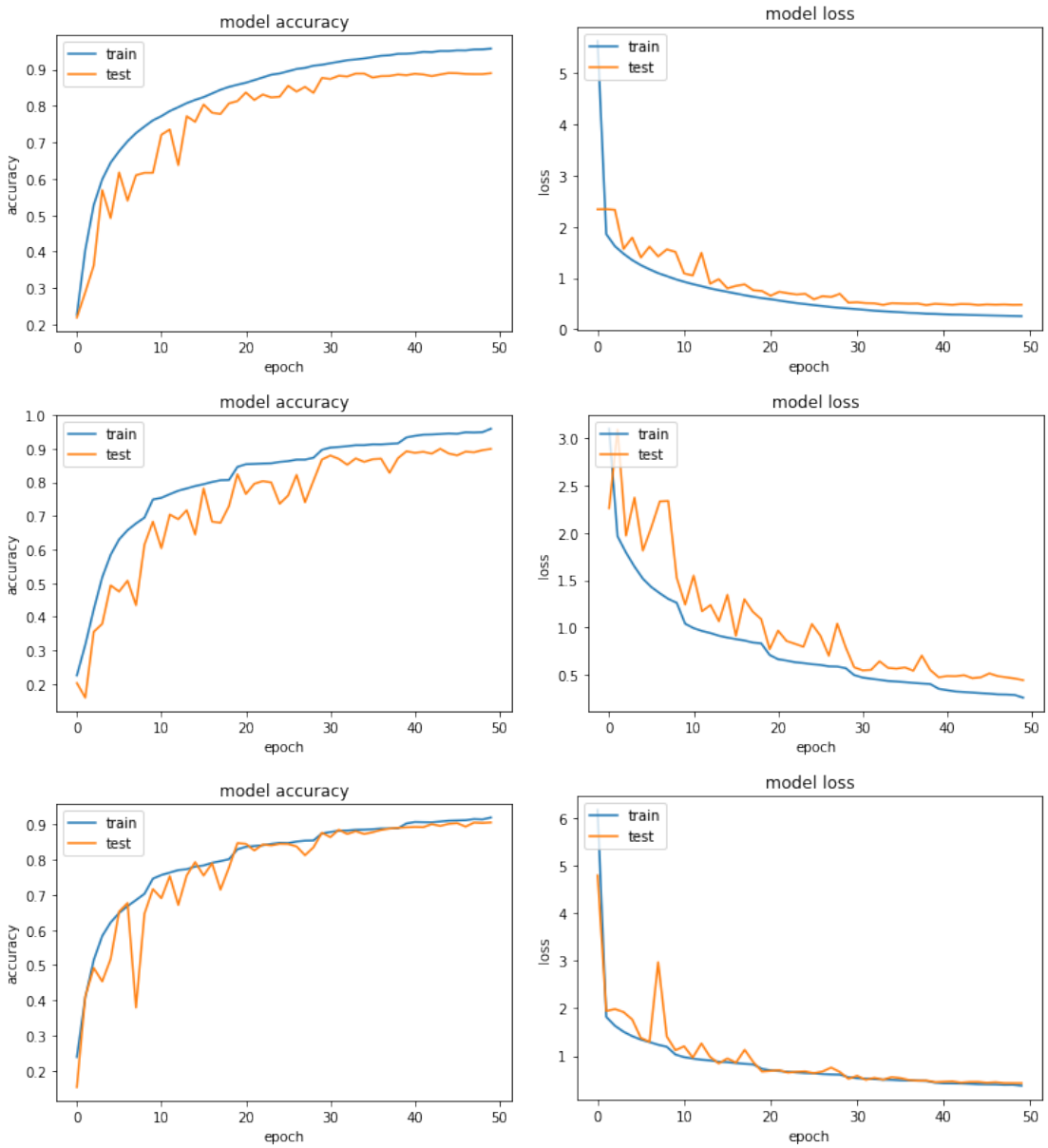


Figure 0.11: Deep models run for 50 epochs of training