Classifying Cats and Dogs from images using Deep Learning

(MM803 Course Project)

Name : SHROBON BISWAS
Student ID : 1505851
Course : MM812

Introduction

The aim of this project is to use Deep Learning as a tool to correctly classify images of cats and dogs, using a subset of the Asirra dataset. To foster a good understanding, and appreciate some Deep Learning techniques and models, the project report has been drafted such that, every new experiment leads to an incremental growth in performance, compared to the previous experiment.

Motivation and Background

The subset of the Asirra dataset by Microsoft contains 25,000 labeled images of cats and dogs (12500 cats, 12500 dogs), where height and width of the images are not constant across the dataset. Kaggle hosted a competition in 2013, named Dogs vs. Cats, where the participants aimed at obtaining a high accuracy, using this dataset. The current best classification accuracy obtained in this competition is 98.914%. [4] achieved an accuracy of 94% by training an SVM classifier on the learned features, of first 6 layers of AlexNet. This contribution also pointed out some incorrectly classified images in the dataset, thus explaining their need of using an external training set, which had a positive impact on the accuracy obtained. It is a matter of debate that using extra training set, though beneficial for accuracy, was unfair to the competition. This project aims to achieve an accuracy close to [] , without the need of an external dataset, by effective hyperparameter tuning [] and smart data augmentation techniques.

Setup

The experiments in this project were carried out using the following hardware and software packages.

Software
- Keras 1.1.2
- Theano 0.9.0.dev4
- Python 2.7

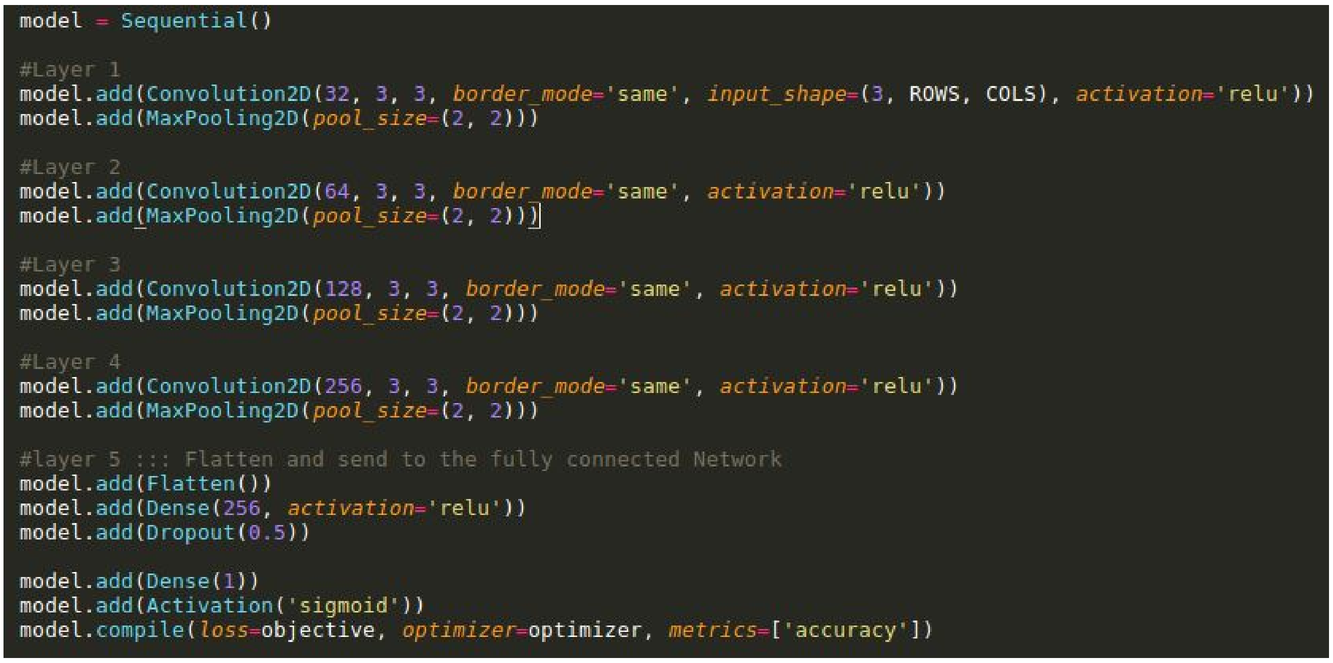
Hardware
- Processor: Intel Core i7-6700HQ (2.6Ghz)
- RAM : 16GB DDR4
- GPU : Nvidia GTX 1060 6GB

Note

The graphs of the Training and Validation loss trend for each experiment has considered, the number of epochs in the X axis. This is the reason why the graphs used in this project look very jittery on first look, as compared to when the graphs are plotted taking the number of iterations in the X axis. To be concise, yet clear, the meaning of iterations and epoch are as follows :

1 epoch = [Number of Training Images / Batch Size] Iterations

Experiment 1



```
model = Sequential()

#Layer 1
model.add(Convolution2D(32, 3, 3, border_mode='same', input_shape=(3, ROWS, COLS), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

#Layer 2
model.add(Convolution2D(64, 3, 3, border_mode='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

#Layer 3
model.add(Convolution2D(128, 3, 3, border_mode='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

#Layer 4
model.add(Convolution2D(256, 3, 3, border_mode='same', activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

#layer 5 ::: Flatten and send to the fully connected Network
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))

model.add(Dense(1))
model.add(Activation('sigmoid'))
model.compile(loss=objective, optimizer=optimizer, metrics=['accuracy'])
```

Figure 1: Model used for experiment 1

Image height	64
Image width	64
Number of Training images	20000
Number of Validation images	5000
Number of epochs	20
Batch size	50
Optimizer	Adadelta
Loss	logloss
Time taken per epoch	12 seconds

Table 1: Parameter used for experiment 1

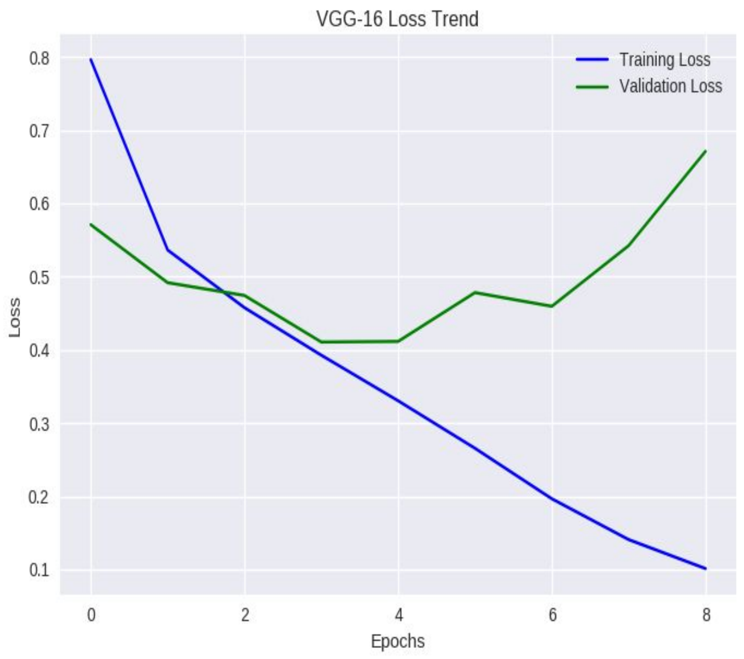


Figure 2: Training and Validation loss trend for experiment 1

Note that figure 2 , shows loss trend till only the 8th epoch, while the number of epochs has been set to 20. This happens due to the use of an early stopping callback function in the experiment, which prematurely stops the experiment when the validation loss is not changing for 3 consecutive epochs. This technique of early stopping is very advantageous in detecting problems in the network, early on during the training.

From the loss trend in figure 2, it can be observed that the model is overfitting as there is a huge gap observed between training loss and validation loss. This is not desired, and implies that the model in figure 1 works well on training images i.e images it has seen many times, but performs miserably on the unseen validation set.

On primary investigation, we observe the amount of data is not sufficient. Thus to mitigate this problem, experiment 2 was conducted, using data augmentation.

Experiment 2

The model used in this experiment is same as in experiment 1 (figure 1)

Image height	64
Image width	64
Number of Training images	20000
Number of Validation images	5000
Number of epochs	10
Batch size	50
Optimizer	Adadelta
Loss	logloss
Augmentation Parameters 1. Rotation Range 2. Fill Mode 3. Horizontal_flip	 90^0 Wrap True
Time taken per epoch	15 seconds

Table 2: Parameter used for experiment 2

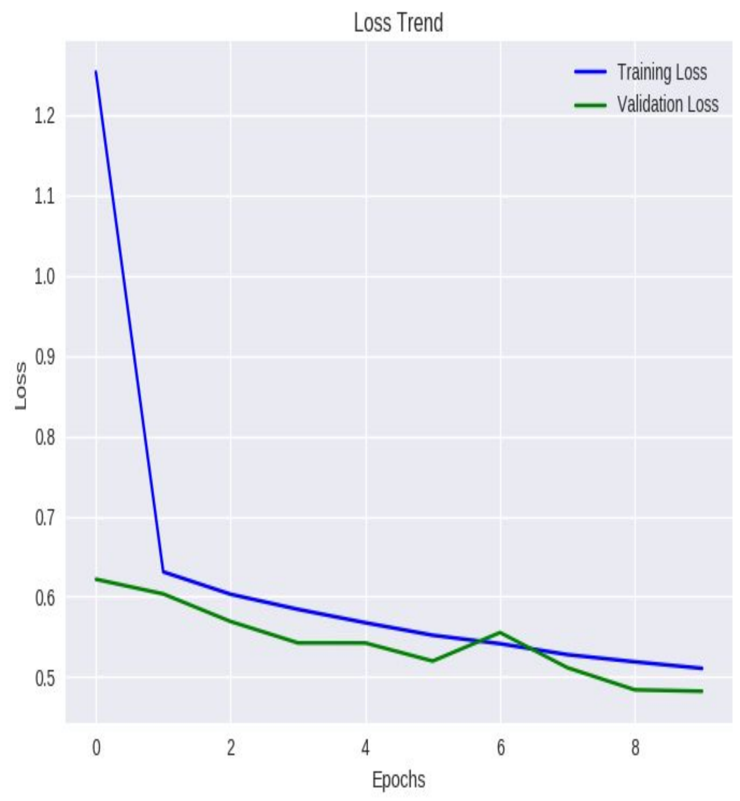


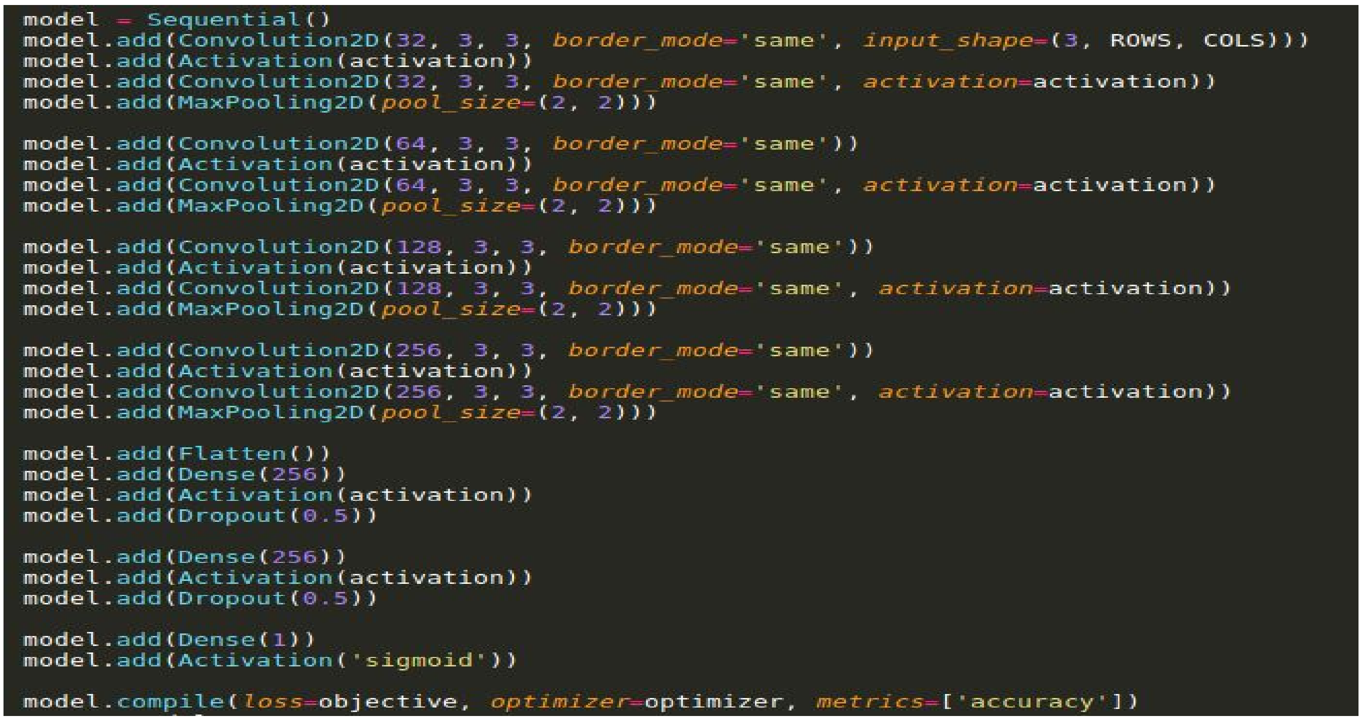
Figure 3: Training and Validation loss trend for experiment 2

In this experiment, rather than creating the augmented dataset and storing them in memory, a just-in-time data augmentation scheme was followed, which reduces the storage overhead but adds to the time taken per epoch during training. Image augmentation parameters are dataset dependent, and results are heavily based on the proper tuning of the parameters. The augmentation parameters applied here were purely based on intuition and thus leaves a large scope of parameter selection and tuning.

One interesting observation in this experiment is that validation loss is most of the times lower than the training loss. This is a special case and falls neither in the class of overfitting and underfitting. One good way of reasoning this scenario is that the experiment model has used dropout regularizations, which drop a percentage of activation maps only during the training, while this does not take place during testing.

Thus we see that the overfitting problem in experiment 1 has been solved in experiment 2 and the model has a classification accuracy of 76%. In the following experiments, the target will be to improve the classification accuracy.

Experiment 3



```
model = Sequential()
model.add(Convolution2D(32, 3, 3, border_mode='same', input_shape=(3, ROWS, COLS)))
model.add(Activation(activation))
model.add(Convolution2D(32, 3, 3, border_mode='same', activation=activation))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Convolution2D(64, 3, 3, border_mode='same'))
model.add(Activation(activation))
model.add(Convolution2D(64, 3, 3, border_mode='same', activation=activation))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Convolution2D(128, 3, 3, border_mode='same'))
model.add(Activation(activation))
model.add(Convolution2D(128, 3, 3, border_mode='same', activation=activation))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Convolution2D(256, 3, 3, border_mode='same'))
model.add(Activation(activation))
model.add(Convolution2D(256, 3, 3, border_mode='same', activation=activation))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(256))
model.add(Activation(activation))
model.add(Dropout(0.5))

model.add(Dense(256))
model.add(Activation(activation))
model.add(Dropout(0.5))

model.add(Dense(1))
model.add(Activation('sigmoid'))

model.compile(loss=objective, optimizer=optimizer, metrics=['accuracy'])
```

Figure 4: Model used for experiment 3

Image height	64
Image width	64
Number of Training images	20000
Number of Validation images	5000
Number of epochs	10
Batch size	50
Optimizer	Adadelta
Loss	logloss
Augmentation Parameters 4. Rotation Range 5. Fill Mode 6. Horizontal_flip	 90^0 Wrap True
Time taken per epoch	17 seconds

Table 3: Parameter used for experiment 3

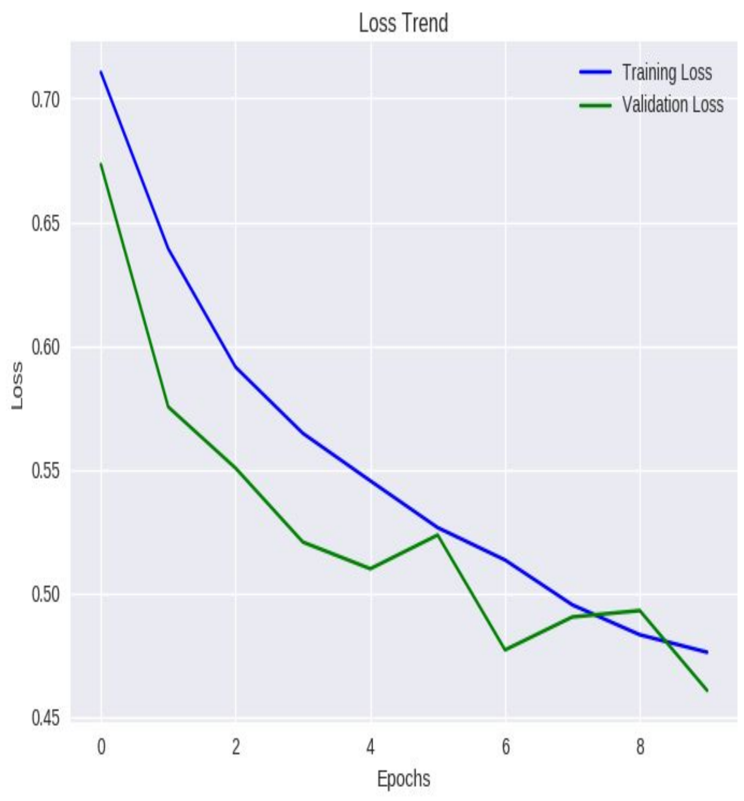


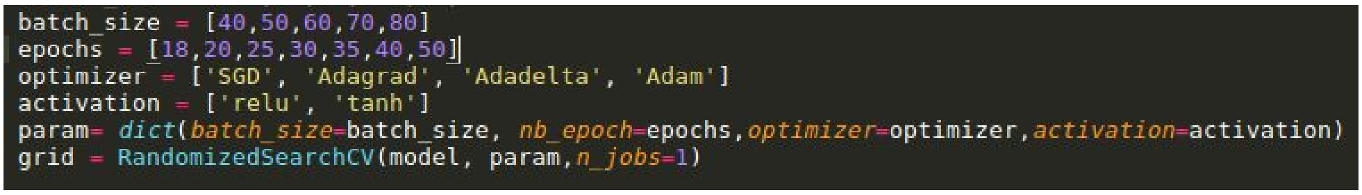
Figure 5: Training and Validation loss trend for experiment 3

This model introduces an extra convolution step of convolution in every layer and thus is a minor change from the model considered in experiment 1 (figure 1). Coupled with data augmentation, this model achieves a classification accuracy of 77.51, which is marginally more than experiment 2.

One could arguably state that perhaps a deeper architecture would help to improve the classification accuracy. Well, it is true to some extent, that deeper networks learn better if fed with lots of data. But one can never deny the fact that a properly tuned shallow network can actually outperform badly tuned deeper networks. To keep the training time within a reasonable limit, the following experiments will consider this model as the base model, and use proper parameter tuning to increase the classification accuracy significantly.

Tuning hyperparameter is an arduous process, and it becomes even more difficult to finetune large networks. For example, the total number of free parameters for the above model (figure 4) is 2287137.
Following a blindfold trial and error method is not feasible and hence other methods need to be considered. There exists a method called Grid Search, which iterates over all the parameters and finds the best suited for this model. This method does find out the best result but is still not feasible since it takes exponential time to compute. A much faster method which finds out reasonably good parameters called Random Search [3] was proposed in 2012.

In this project, Random Search was used initially on the models to find reasonably good base parameters, upon which further tuning was done to achieve good results.


```
batch_size = [40,50,60,70,80]
epochs = [18,20,25,30,35,40,50]
optimizer = ['SGD', 'Adagrad', 'Adadelta', 'Adam']
activation = ['relu', 'tanh']
param= dict(batch_size=batch_size, nb_epoch=epochs,optimizer=optimizer,activation=activation)
grid = RandomizedSearchCV(model, param,n_jobs=1)
```
Figure 6: Implementation of Random Search in Keras

Experiment 4

In this experiment, the model in experiment 3 was kept unchanged, but different weight initializations were added in turn to the dense layers. The parameters decided have been tuned using Random Search Hyperparameter tuning method.

On doing the weight initializations, the classification accuracy of the model increased by leaps and bounds. Empirically it is seen that gloroot_uniform [2] weight initialization gives the best classification accuracy of 88.16%, thus bolstering the fact that proper weight initialization significantly increases its performance. On observation of figure 7, it is seen that, though the number of 80 epochs were set, our network early stopped at nearly 40, thus proving the fact in [2], that weight initializations lead to faster convergence.

Image height	64
Image width	64
Number of Training images	20000
Number of Validation images	5000
Number of epochs	80
Batch size	50
Optimizer	Adadelta
Loss	logloss
Augmentation Parameters 7. Rotation Range 8. Fill Mode 9. Horizontal_flip	 90^0 Wrap True
Time taken per epoch	18 seconds

Table 4: Parameter used for experiment 4

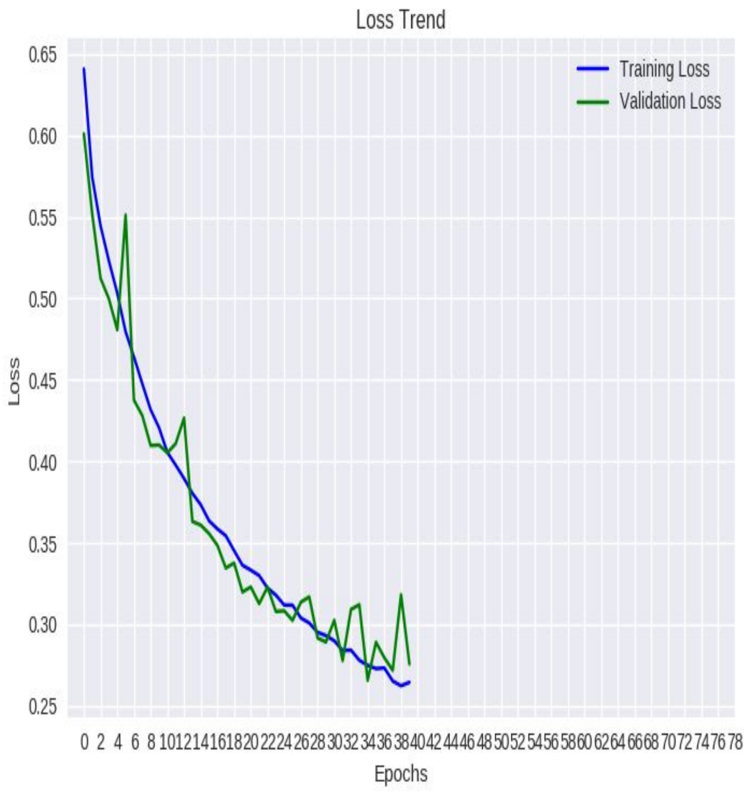
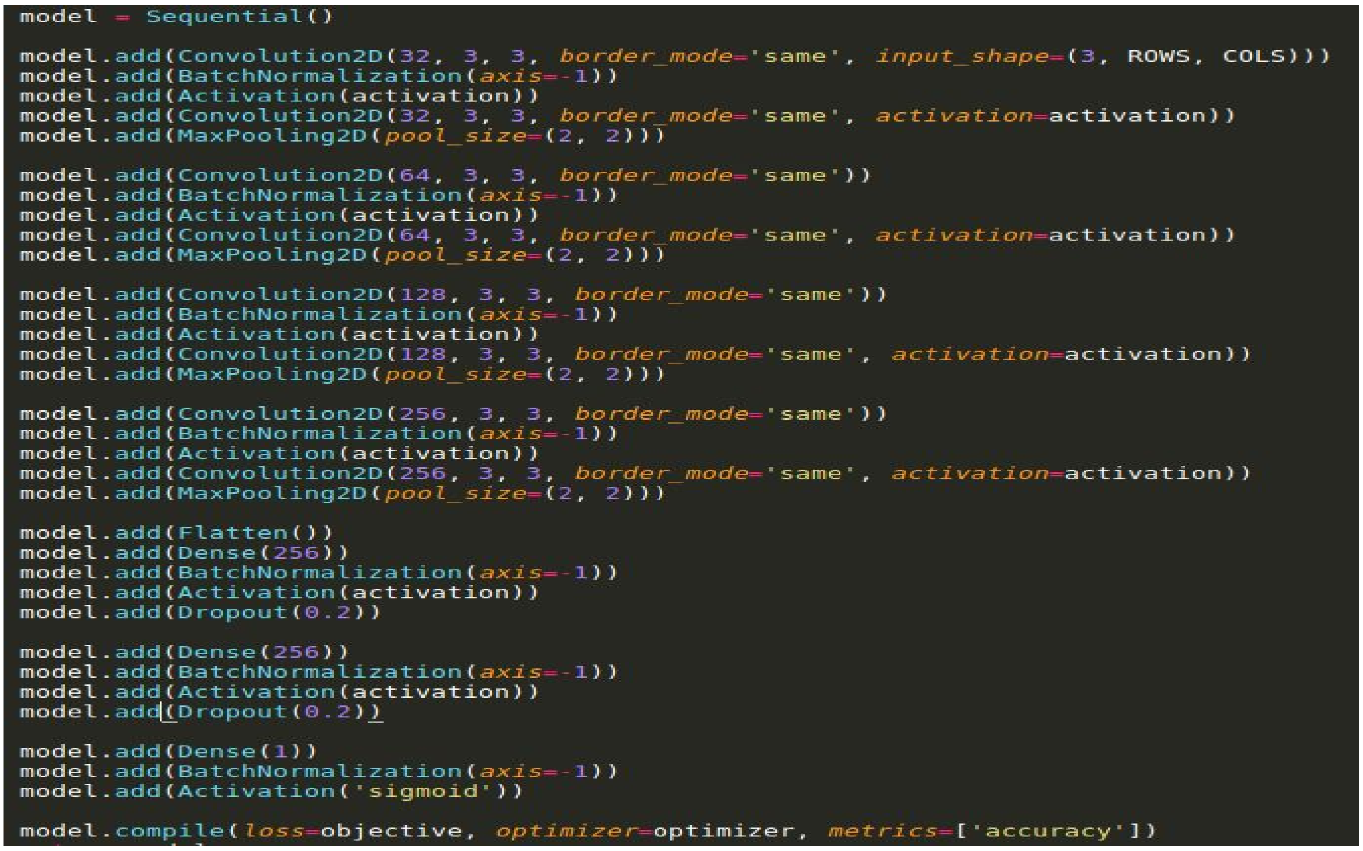


Figure 7: Training and Validation loss trend for experiment 4

Experiment 5

In [1] a technique called batch normalization is proposed, which promises higher learning rates, and reduce the dependence on the network on proper weight initialization. Batch normalization also acts as a regularizer and reduces the need of dropout regularizations. Batch normalization works by explicitly forcing the activations throughout a network to take on a unit gaussian distribution at the beginning of the training. In this experiment, we will be working with batch normalizations and observing its effect on classification accuracy and time taken per epoch.



```
model = Sequential()

model.add(Convolution2D(32, 3, 3, border_mode='same', input_shape=(3, ROWS, COLS)))
model.add(BatchNormalization(axis=-1))
model.add(Activation(activation))
model.add(Convolution2D(32, 3, 3, border_mode='same', activation=activation))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Convolution2D(64, 3, 3, border_mode='same'))
model.add(BatchNormalization(axis=-1))
model.add(Activation(activation))
model.add(Convolution2D(64, 3, 3, border_mode='same', activation=activation))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Convolution2D(128, 3, 3, border_mode='same'))
model.add(BatchNormalization(axis=-1))
model.add(Activation(activation))
model.add(Convolution2D(128, 3, 3, border_mode='same', activation=activation))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Convolution2D(256, 3, 3, border_mode='same'))
model.add(BatchNormalization(axis=-1))
model.add(Activation(activation))
model.add(Convolution2D(256, 3, 3, border_mode='same', activation=activation))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(256))
model.add(BatchNormalization(axis=-1))
model.add(Activation(activation))
model.add(Dropout(0.2))

model.add(Dense(256))
model.add(BatchNormalization(axis=-1))
model.add(Activation(activation))
model.add(Dropout(0.2))

model.add(Dense(1))
model.add(BatchNormalization(axis=-1))
model.add(Activation('sigmoid'))

model.compile(loss=objective, optimizer=optimizer, metrics=['accuracy'])
```

Figure 8: Model used for experiment 5

Image height	64
Image width	64
Number of Training images	20000
Number of Validation images	5000
Number of epochs	60
Batch size	60
Optimizer	Adadelta
Loss	logloss
Augmentation Parameters 10. Rotation Range 11. Fill Mode 12. Horizontal_flip	 90^0 Wrap True
Time taken per epoch	72 seconds

Table 5: Parameter used for experiment 5

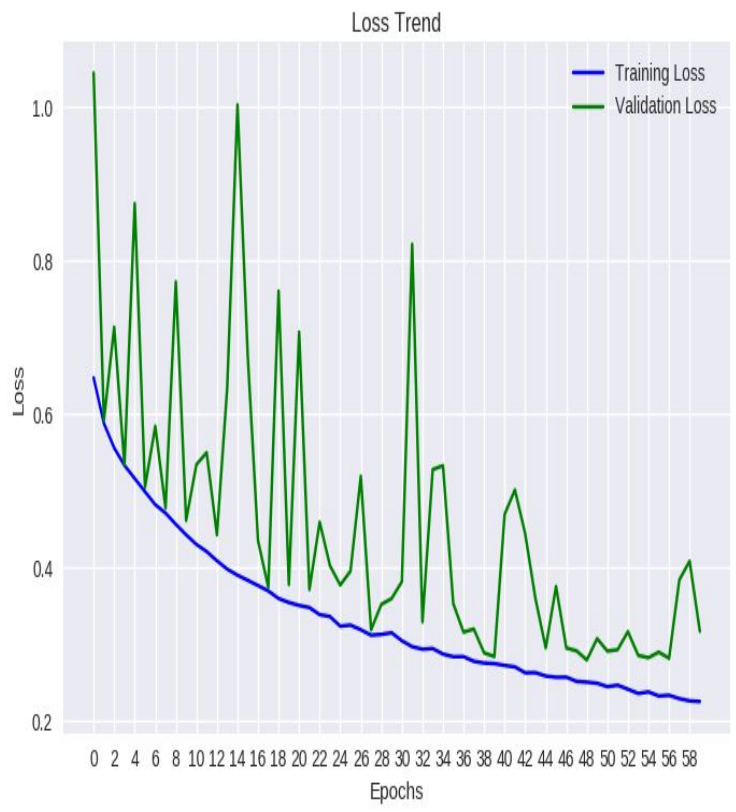
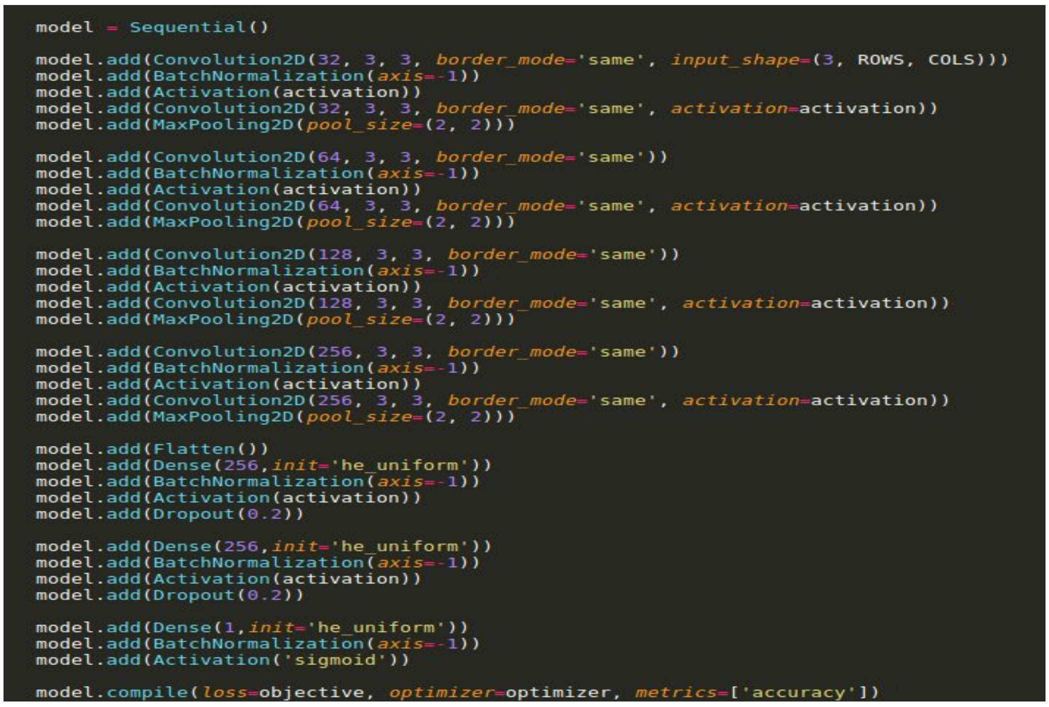


Figure 9: Training and Validation loss trend for experiment 5

The first striking observations from the parameters is that time taken per epoch with batch normalization is 3.5 times more than without batch normalization. Nonetheless, batch normalization gives considerably good results i.e with 60 epochs, it achieves an accuracy of 86%. On running several variations of this model, it was observed that batch normalization in addition to properly calibrated dropout layers and weight initializations gives a good result. The following experiment is a good example of this.

Experiment 6 (Best model)



```
model = Sequential()

model.add(Convolution2D(32, 3, 3, border_mode='same', input_shape=(3, ROWS, COLS)))
model.add(BatchNormalization(axis=-1))
model.add(Activation(activation))
model.add(Convolution2D(32, 3, 3, border_mode='same', activation=activation))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Convolution2D(64, 3, 3, border_mode='same'))
model.add(BatchNormalization(axis=-1))
model.add(Activation(activation))
model.add(Convolution2D(64, 3, 3, border_mode='same', activation=activation))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Convolution2D(128, 3, 3, border_mode='same'))
model.add(BatchNormalization(axis=-1))
model.add(Activation(activation))
model.add(Convolution2D(128, 3, 3, border_mode='same', activation=activation))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Convolution2D(256, 3, 3, border_mode='same'))
model.add(BatchNormalization(axis=-1))
model.add(Activation(activation))
model.add(Convolution2D(256, 3, 3, border_mode='same', activation=activation))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(256,init='he_uniform'))
model.add(BatchNormalization(axis=-1))
model.add(Activation(activation))
model.add(Dropout(0.2))

model.add(Dense(256,init='he_uniform'))
model.add(BatchNormalization(axis=-1))
model.add(Activation(activation))
model.add(Dropout(0.2))

model.add(Dense(1,init='he_uniform'))
model.add(BatchNormalization(axis=-1))
model.add(Activation('sigmoid'))

model.compile(loss=objective, optimizer=optimizer, metrics=['accuracy'])
```

Figure 10: Model used for experiment 6

Image height	64
Image width	64
Number of Training images	20000
Number of Validation images	5000
Number of epochs	200
Batch size	32
Optimizer	Adam
Loss	logloss
Augmentation Parameters 13. Rotation Range 14. Fill Mode 15. Horizontal_flip	 90^0 Wrap True
Time taken per epoch	78 seconds

Table 6: Parameter used for experiment 6.1

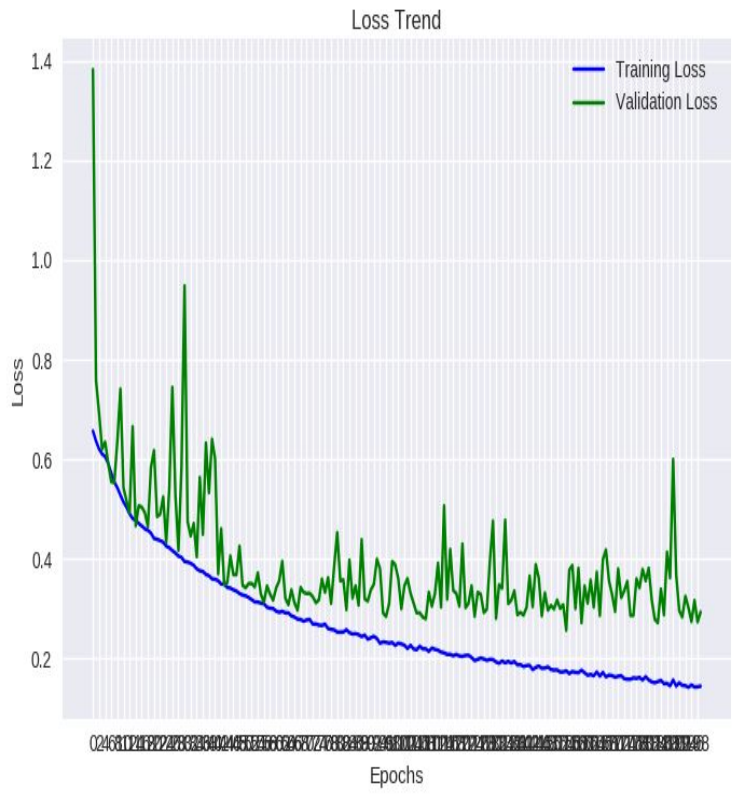
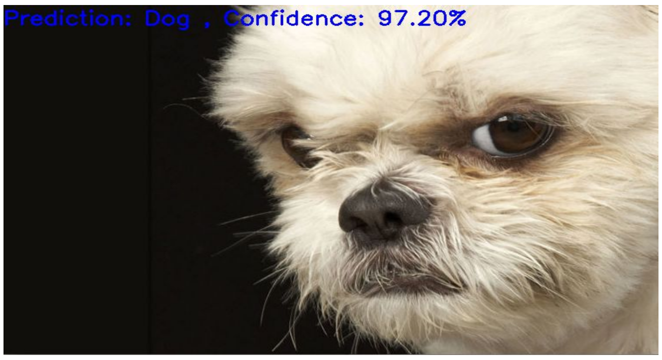


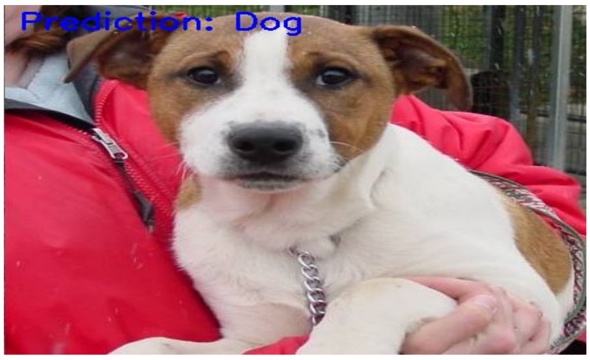
Figure 11: Training and Validation loss trend for experiment 5

Results

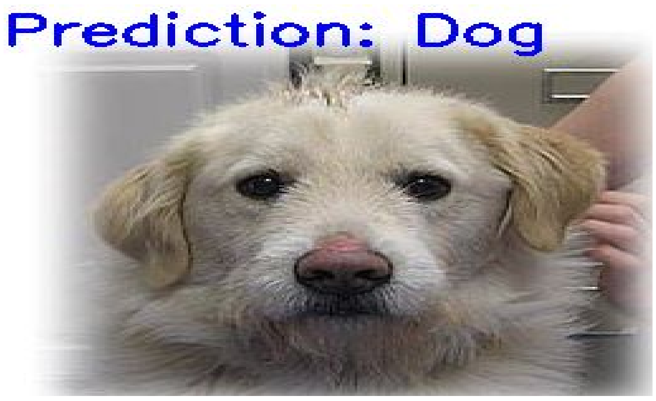
In this experiment 6, the classification accuracy resulted in 90.18% using he_uniform weight initializations and properly tuned dropout layers. Thus experiment 6 provides the best classification results for this project. This model was tested thoroughly on 9 unseen sets of cat and dog images classified as easy, medium and hard, with each set consisting of 30 images, and this model achieved an average of 28/30 right classification result in all the sets. What is more interesting is that, for the correct classification, the model shows a very high prediction confidence value, whereas for a wrongly classified image, the prediction value is poor. A video demonstration during a classification test of this model can been seen here. Apart from this a few results are given in the following table



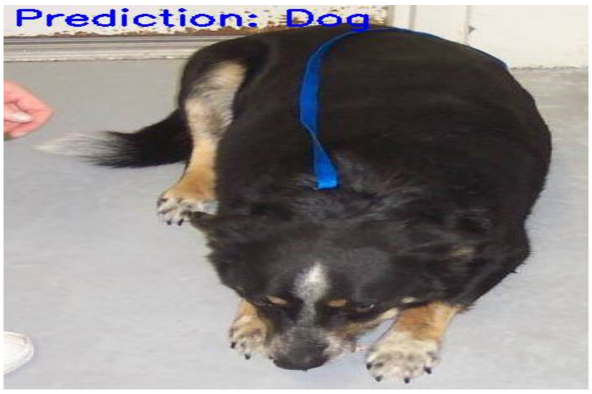
Correctly Predicted
Prediction Confidence: 97.20% Dog



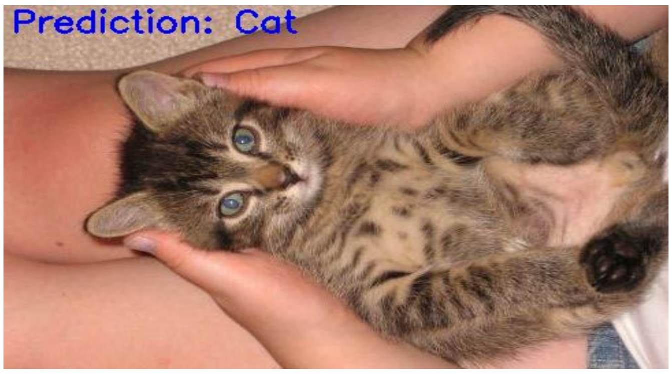
Correctly Predicted
prediction Confidence: 100% Dog



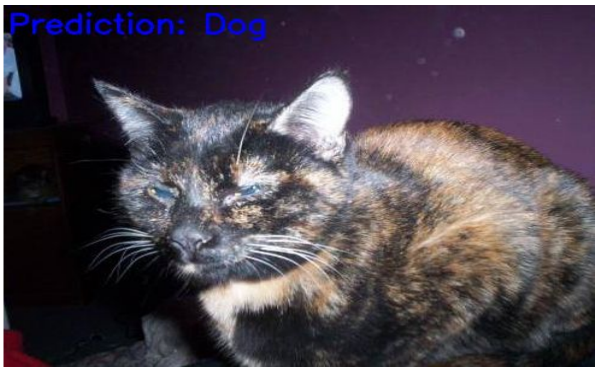
Correctly Predicted
Prediction Confidence: 64.47% Dog



Correctly Predicted
Prediction Confidence: 90.30



Correctly Predicted
Prediction Confidence: 99.97% Cat



Wrongly Predicted
Prediction Confidence: 85.03% Dog

Discussion

Batch normalization as per the experimental results is slow and steady. However, on a personal note, I feel time invested in choosing a good weight initialization would be effectively faster than running batch normalizations. As can be seen in experiment 4, the loss values of the graph comes down quickly as compared to experiment 5 or experiment 6. An important point to look at is the way the batch normalization layers were added in the existing model stack. It has been suggested in [1] that batch normalizations are to be used just after Convolution layers and Fully connected layers and immediately before the activation functions are used. The experiments have been performed keeping this suggestion in mind, however [5] compiles a very nice benchmark performance tests on batch normalization, the result of which is that batch normalization layer placed just after the activation layers provides a better classification accuracy. So effectively the suggestions in [1] and [5] are contradictory.
Throughout all the experiments, the loss function was kept fixed to logloss, the reason being that logloss function on the error provides extreme punishment for wrong predictions, with high confidence. The use of this loss function is very crucial to achieve properly tuned weights, thus leading to a higher classification accuracy. While considering this loss function, it is extremely crucial to consider dropout regularization, so as to reduce the risk of overfitting.

Figure 12 illustrates the logloss function, where the probability is plotted on the X-axis (0 implying a completely wrong prediction, and 1 implying a 100% correct prediction). Thus it can be seen how the value of the logloss function increases rapidly as the probability gets lower.

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} y_{ij}\log(p_{ij})$$

where
N is the number of examples
M is the number of classes
Y_{ij} is variable indicating whether class j is a correct for example i

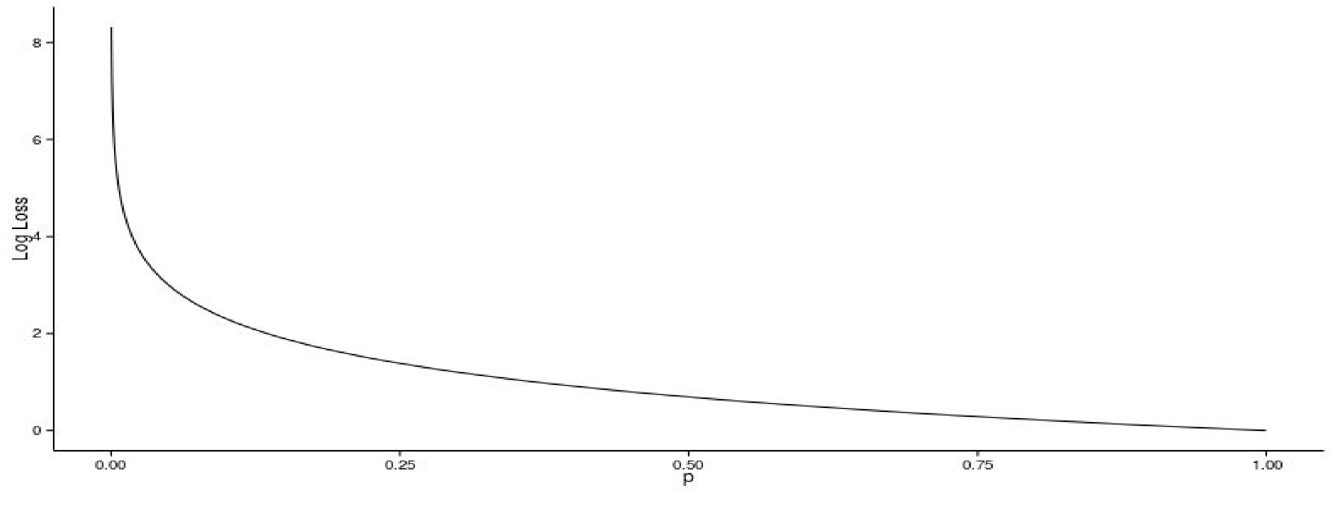


Figure 12: Logloss function

Conclusion

Thus in this project, the different techniques like data augmentation, batch normalization, and weight initialization were studied and their results were compared. I was able to get a classification accuracy of 90.18%, without the use of an external dataset. This accuracy can further be improved by making just slight changes to the existing model by fine tuning the hyperparameters even more. Due to hardware constraints, I had to limit myself to at most of 200 epochs. This leaves a big scope for future work to be done using different activation functions like pRelu and leakyRelu, different models, and benchmarking their performance with respect to suggestions [1] and [5] and thus bolstering, which idea provides a better performance.

Acknowledgement

I am grateful to professor Nilanjan Ray as his lecture notes and classroom discussions have helped me greatly in understanding and framing strategies to get better results in this project. I would also like to acknowledge François Chollet for his contribution to Deep Learning Framework called keras.

References

1. S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”

2. X. Glorot and Yoshua Bengio,"Understanding the difficulty of training deep feedforward neural networks," in International Conference on Artificial Intelligence and Statistics (AISTATS’10), Society for Artificial Intelligence and Statistics, 2010.

3. J. Bergstraand Yoshua Bengio , "Random Search for Hyper-Parameter Optimization," in Journal of Machine Learning Research 13, 2012, pp. 281–305.

4. B. Liu, Y. Liu, and K. Zhou, "Image Classification for Dogs and Cats"

5. D. Mishkin, "Performance evaluation of the batch normalization layer,". [Online]. Available: https://github.com/ducha-aiki/caffenet-benchmark/blob/master/batchnorm.md.

6. F. Chollet, "Keras: Deep Learning library for Theano and TensorFlow,". [Online]. Available: https://keras.io/