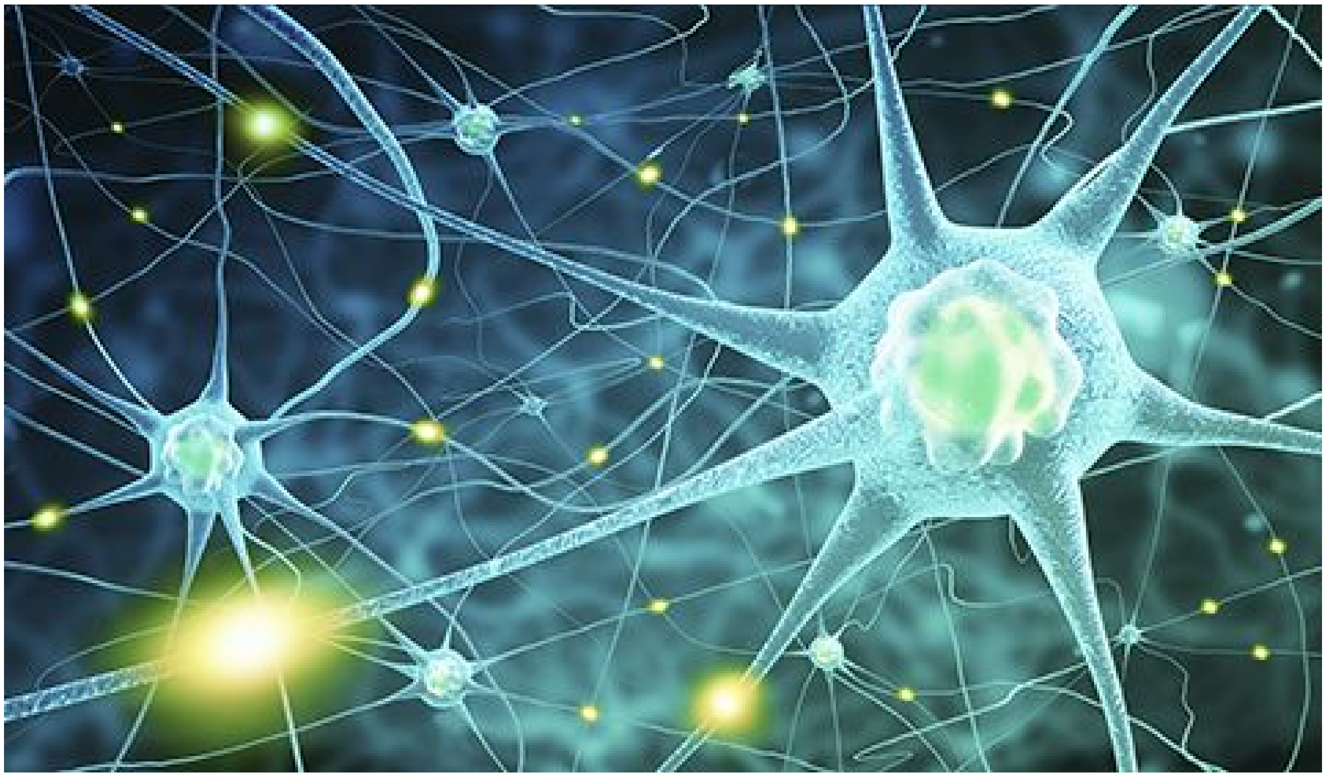
Assignment 1

Deep Learning - CS577



Akshay R

A20442409

Task 1:

Problem Statement:

Multi-Class classification of images from CIFAR10 dataset using a fully connected neural network. Need to perform classification on a subclass of the dataset, 3 precisely and compute the model accuracy.

Proposed Solution:

The CIFAR10 dataset has 60,000 images,60,000 labels distributed over 10 classes, 11,000 images and labels in each class. This data must be sampled to obtain a dataset consisting of only three sub classes. The data obtained must be normalized and further divided into train, validation and test data sets. The labels must be encoded into a categorical variable with three values.

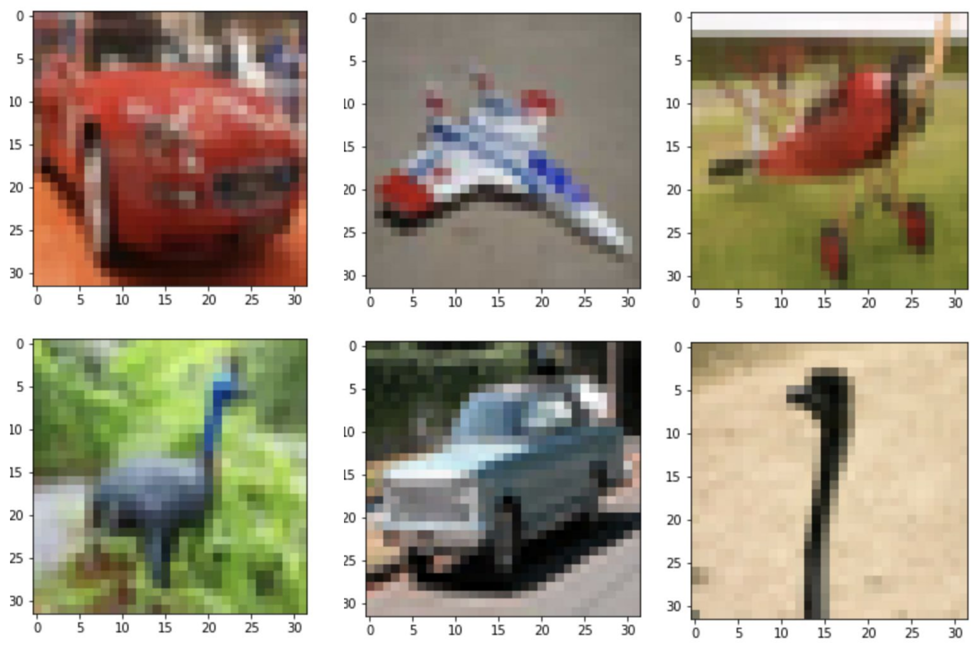
A fully connected neural network must be built that takes in input in the shape of a 1-D array and provides an output out of three neurons. The model must be tweaked to obtain the best accuracy and minimal loss.

Implementation and methodology:

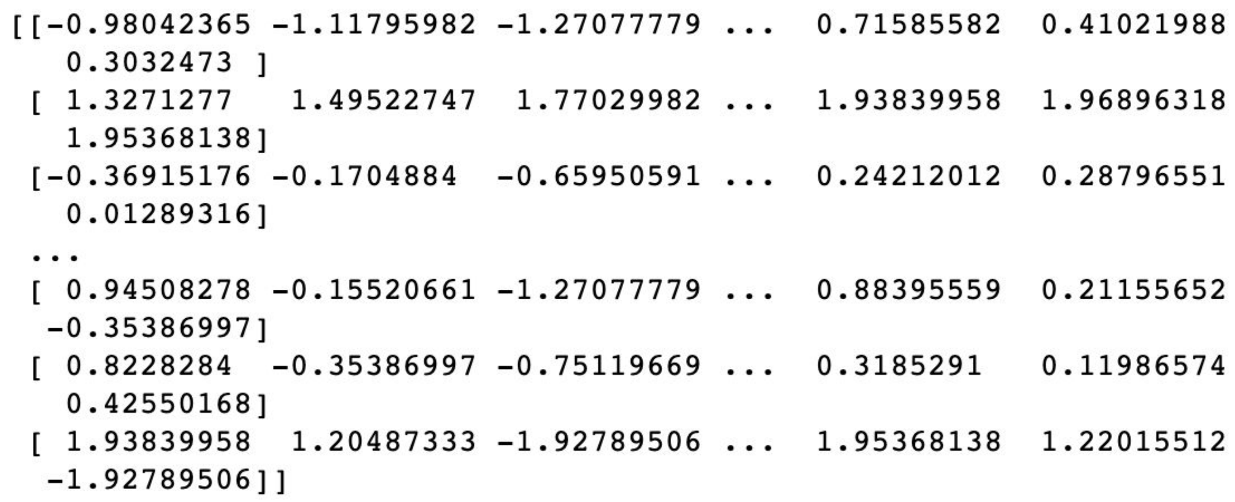
The given dataset is first divided into test and train data. Further only three classes of this dataset are extracted using a simple loop. The classes extracted in this case are 0, 1 and 2 which are 0 - Aeroplanes, 1- Automobiles and 2 - Birds.

Before performing subclass division the first few images looks as follows:

After selecting only first three subclasses the dataset looks as follows:



The data is normalized by subtracting the mean and dividing the standard deviation from pixel images so that processing of data is done right. Normalized data is reshaped into a single dimension array from a 3-D array. After normalizing and reshaping the images are processed in the following way:

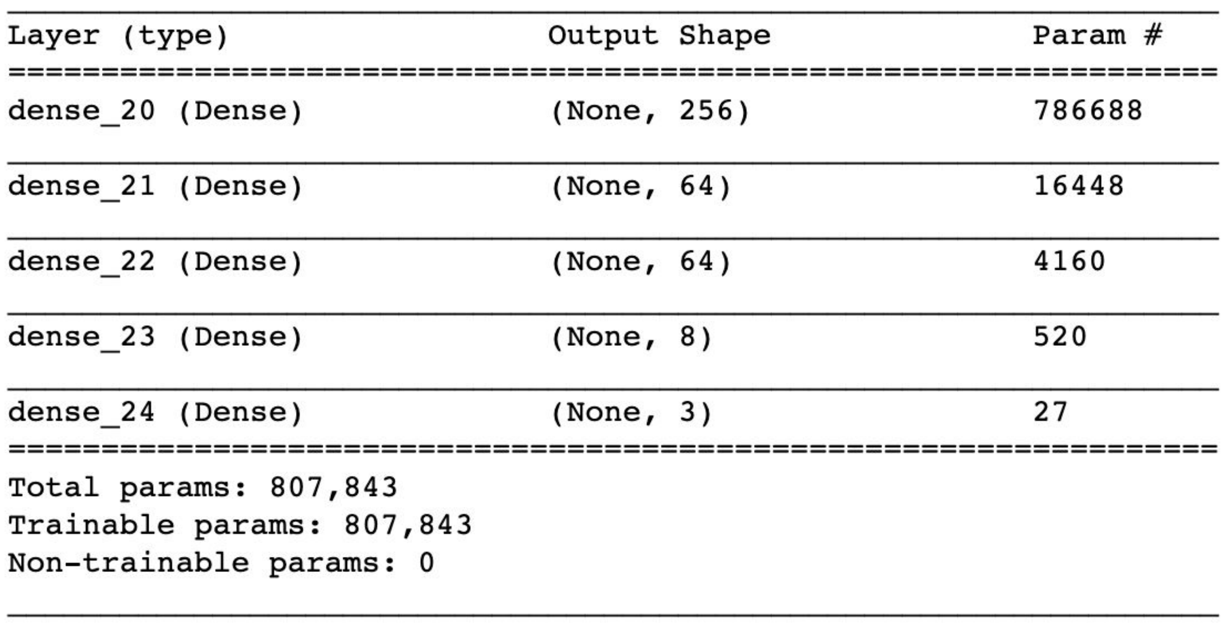


```
[[-0.98042365 -1.11795982 -1.27077779 ...  0.71585582  0.41021988
   0.3032473 ]
 [ 1.3271277   1.49522747  1.77029982 ...  1.93839958  1.96896318
   1.95368138]
 [-0.36915176 -0.1704884  -0.65950591 ...  0.24212012  0.28796551
   0.01289316]
 ...
 [ 0.94508278 -0.15520661 -1.27077779 ...  0.88395559  0.21155652
  -0.35386997]
 [ 0.8228284  -0.35386997 -0.75119669 ...  0.3185291   0.11986574
   0.42550168]
 [ 1.93839958  1.20487333 -1.92789506 ...  1.95368138  1.22015512
  -1.92789506]]
```

Now it's time to create a model and run our data through it.

Model 1:

The first model created consisted of 4 hidden layers with the structure as below:



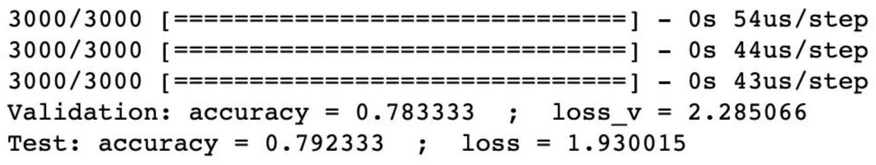
```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_20 (Dense)             (None, 256)               786688
_________________________________________________________________
dense_21 (Dense)             (None, 64)                16448
_________________________________________________________________
dense_22 (Dense)             (None, 64)                4160
_________________________________________________________________
dense_23 (Dense)             (None, 8)                 520
_________________________________________________________________
dense_24 (Dense)             (None, 3)                 27
=================================================================
Total params: 807,843
Trainable params: 807,843
Non-trainable params: 0
_________________________________________________________________
```

The activation layer at each layer was ‘relu’ except the final layer which was ‘softmax’ due to the reason that the output layer has to be robust to the output of the hidden layers. A relu activation function would not suffice due to the reason that we are performing a multiclass classification and the result will be decided based on the probability of the current image belonging to a certain class, which is decided by the model.

This model is the fed with the image 1-D vector and its labels to perform classification

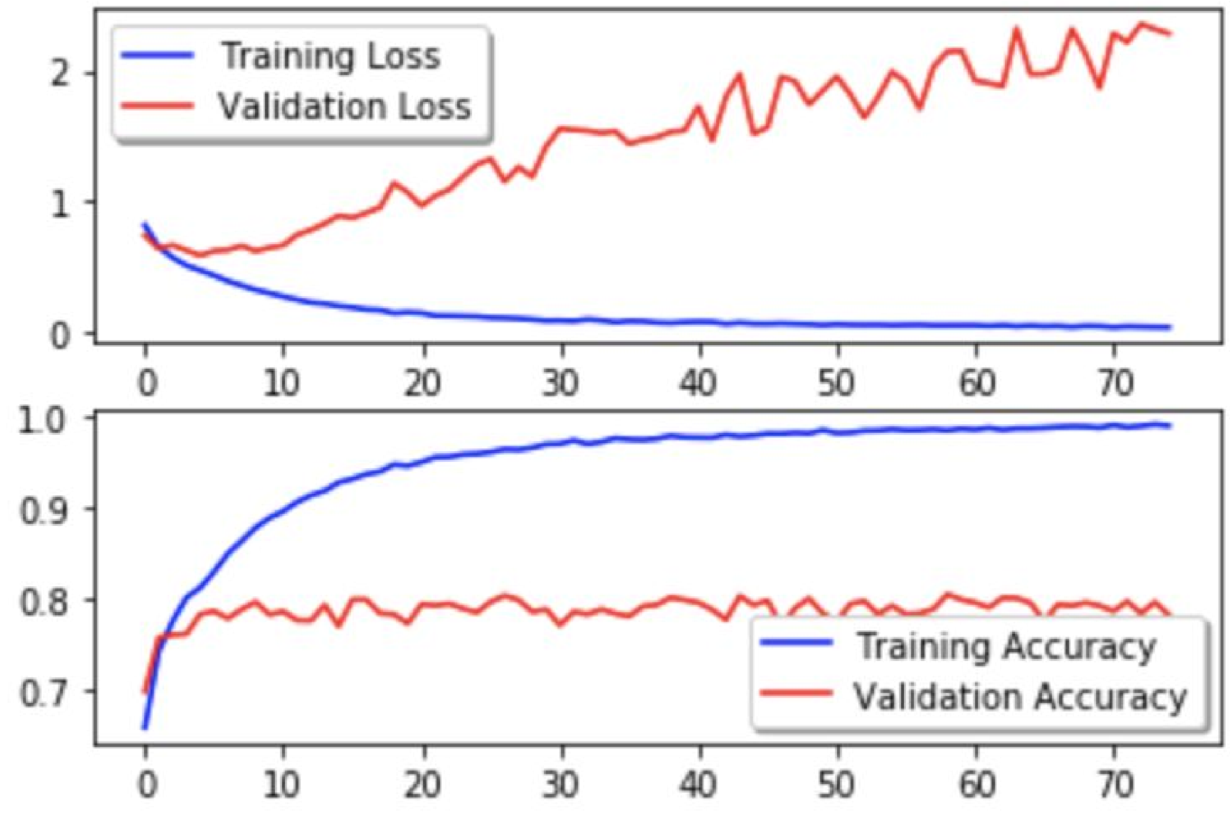
The optimizer used here is RMSprop with its default learning rate. We are measuring the accuracy of classification and hence accuracy is the metric used. This setup runs for 75 epochs with a batch size of 100.

The results of the model are as below:



```
3000/3000 [==============================] - 0s 54us/step
3000/3000 [==============================] - 0s 44us/step
3000/3000 [==============================] - 0s 43us/step
Validation: accuracy = 0.783333  ;  loss_v = 2.285066
Test: accuracy = 0.792333  ;  loss = 1.930015
```

Training and validation loss are plotted against the number of epochs to obatin the below graph:

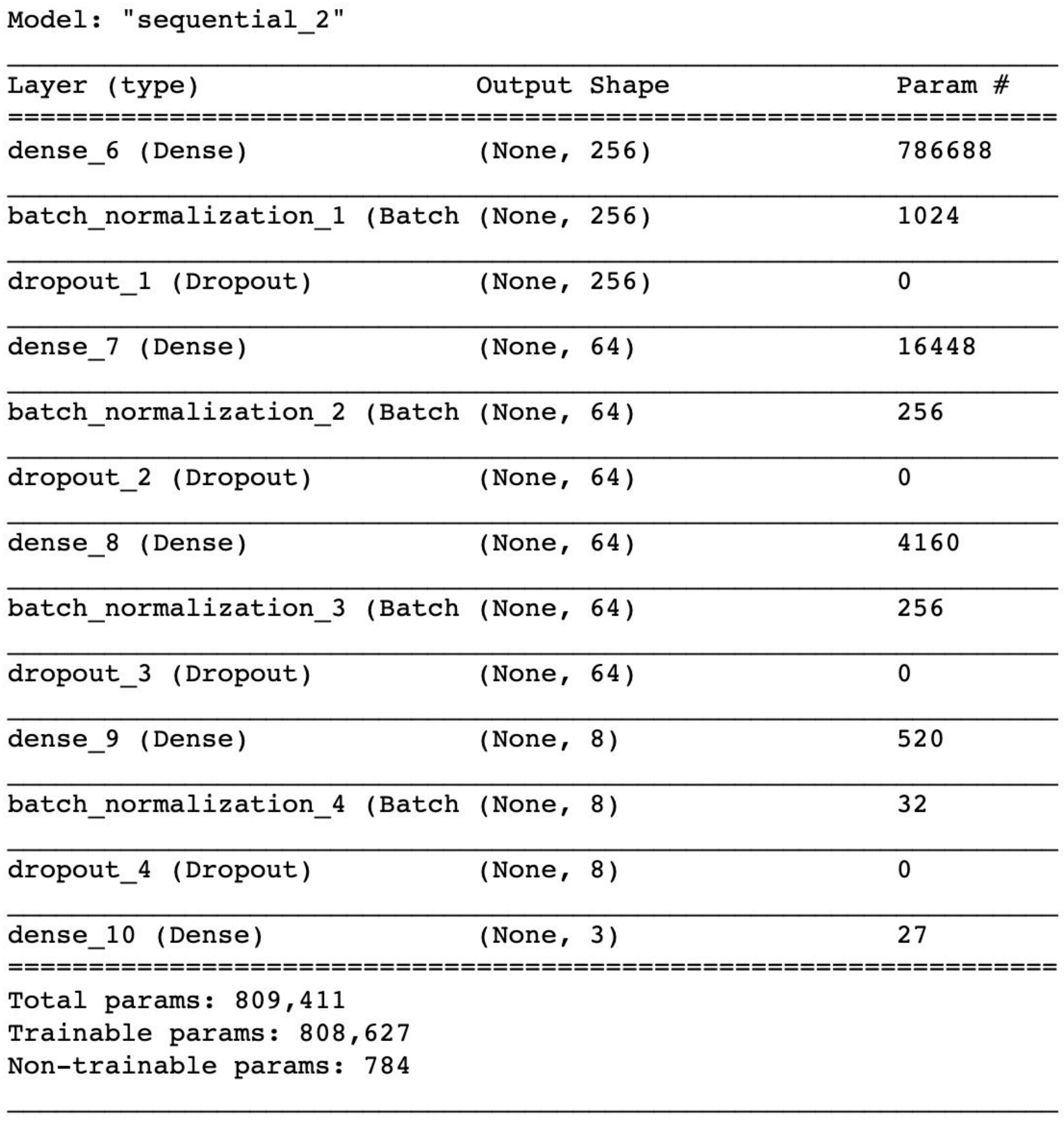


The output obtained looks distorted upon visual inspection.The reason for such distorted output can be for the reason that the weights might vary more than it is supposed to. A general measure to ensure the prevention of this is to normalize the inputs before every stage. Hence in the next model created a Batch_normalization layer is added before each layer.

Another measure to reduce overfitting and to generalize the output provided by the model we use Dropout layers after the Batch_normalization layer. This induces randomness to the data fed to each other layer and hence a less biased model is obtained.

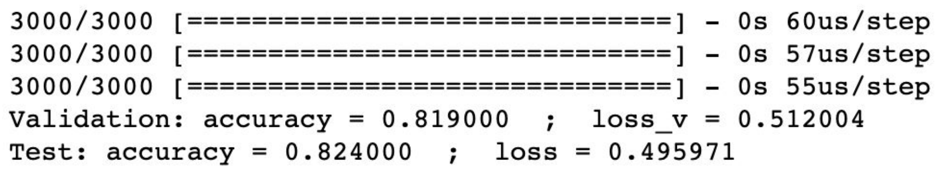
Model 2:

After adding the Batch_normalization and dropout layers the model obtained looks as follows:



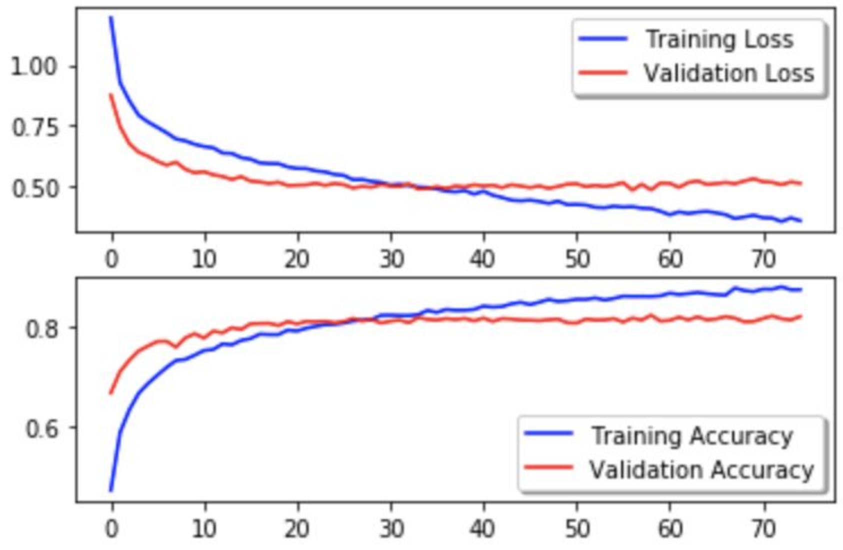
```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_6 (Dense)              (None, 256)               786688
_________________________________________________________________
batch_normalization_1 (Batch (None, 256)               1024
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_7 (Dense)              (None, 64)                16448
_________________________________________________________________
batch_normalization_2 (Batch (None, 64)                256
_________________________________________________________________
dropout_2 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_8 (Dense)              (None, 64)                4160
_________________________________________________________________
batch_normalization_3 (Batch (None, 64)                256
_________________________________________________________________
dropout_3 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_9 (Dense)              (None, 8)                 520
_________________________________________________________________
batch_normalization_4 (Batch (None, 8)                 32
_________________________________________________________________
dropout_4 (Dropout)          (None, 8)                 0
_________________________________________________________________
dense_10 (Dense)             (None, 3)                 27
=================================================================
Total params: 809,411
Trainable params: 808,627
Non-trainable params: 784
_________________________________________________________________
```

The activation function and optimizer are unchanged and the model is run again. The output of this model is as follows:



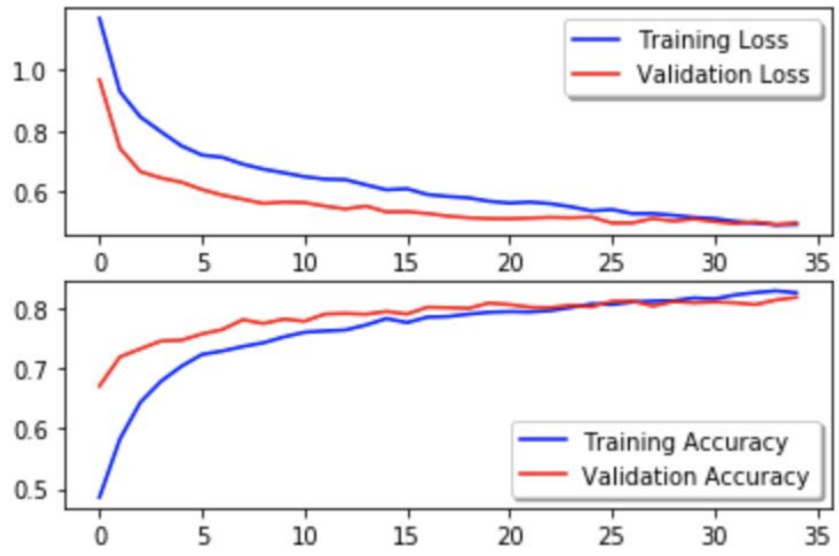
```
3000/3000 [==============================] - 0s 60us/step
3000/3000 [==============================] - 0s 57us/step
3000/3000 [==============================] - 0s 55us/step
Validation: accuracy = 0.819000  ;  loss_v = 0.512004
Test: accuracy = 0.824000  ;  loss = 0.495971
```

Clearly an increase in accuracy. The graph of model accuracy and loss vs epochs for model 2 is as shown below:



It is seen that after 35 epochs the difference in validation accuracy doesn’t vary much and hence the hyperparameter are further tuned and epoch count iset to 35 to obtain our final result:

```
3000/3000 [==============================] - 0s 60us/step
3000/3000 [==============================] - 0s 58us/step
3000/3000 [==============================] - 0s 55us/step
Validation: accuracy = 0.817333  ;  loss_v = 0.498753
Test: accuracy = 0.818667  ;  loss = 0.474176
```



Results and discussion:

After observing the performance of model 1 and model 2 we can conclude that model 2 has a better performance. This is based on the fact that model 2 produces a better accuracy and lower loss than model 1.

What can be learnt from the transition from model 1 to model 2 is that the data was being overfit and the model had some bias in it. Model 2 addressed these issues resulting in a better accuracy and lower loss.

References:

- Batch Normalization and Dropout in Neural Networks with Pytorch
- Understanding RMSprop — faster neural network learning

Task 2:

Problem Statement:

To perform Binary classification of spam emails from spambase dataset using a fully connected neural network.

Proposed Solution:

The Spambase dataset has 57 columns and 4,601 rows of data. The dataset consists of 56 columns of attributes that should be fed to the neural network and the last column of the dataset is the categorical attribute that provides the label for the spam or not spam classification.

Implementation and methodology:

The spambase dataset is downloaded from the UCL Machine Learning Repository and is pre processed in the load_spam_data function. First the columns are separated into frequencies and labels with a division of 56 columns as frequencies and the last column as the last column as labels.

Further this is divided into train and test data with a ratio of 80% to 20% of the data.Once we have the train test data this data is normalized by subtracting the mean and dividing by the standard deviation of each datasets, similar to the task performed in Task 1. Once ve have the normalized data we further divide the training data into train and validate datasets.

The resulting data sets obtained are already in a 1-D array and hence there is no need to reshape the data for modeling.

The input data obtained from the above process is as follows :

```
[[-6.90213404e-02 -6.90213404e-02 -5.70189287e-02 ...  8.53593955e-01
   3.24005480e+00  3.62143984e+00]
 [-6.90213404e-02 -6.90213404e-02 -6.63292107e-02 ... -4.95033998e-02
   4.69404607e-01  1.51260488e+00]
 [-6.73387594e-02 -6.66657269e-02 -6.25153602e-02 ... -1.48534467e-02
   1.23217470e+00  1.77551210e+01]
 ...
 [-6.90213404e-02 -6.90213404e-02 -5.12981530e-02 ... -5.23974393e-02
   9.49911026e-03  4.13318571e-01]
 [-6.90213404e-02 -6.90213404e-02 -6.90213404e-02 ... -5.01203462e-02
   9.49911026e-03  2.21928894e+00]
 [-6.90213404e-02 -6.90213404e-02 -6.90213404e-02 ... -4.68000529e-02
   6.55851465e-02  1.10878542e+00]]
```

The labels for these data points are as follows:

```
[1. 1. 1. ... 0. 0. 0.]
```

Time to create a model!

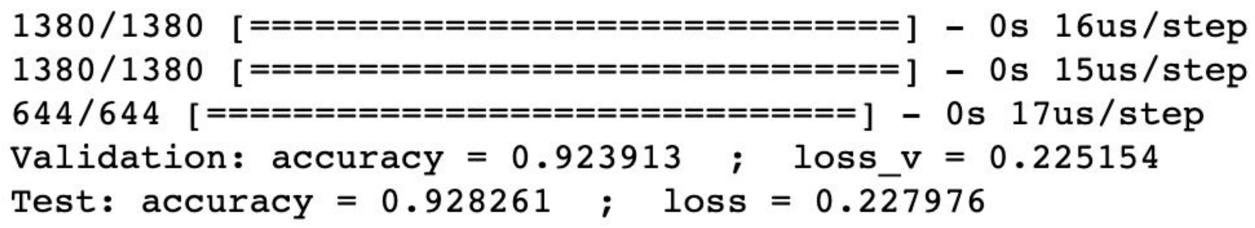
The model created has three hidden layers and is structured in the following way:

```
Model: "sequential_12"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
dense_33 (Dense)             (None, 256)               14848     
_________________________________________________________________
dense_34 (Dense)             (None, 128)               32896     
_________________________________________________________________
dense_35 (Dense)             (None, 64)                8256      
_________________________________________________________________
dense_36 (Dense)             (None, 1)                 65        
=================================================================
Total params: 56,065
Trainable params: 56,065
Non-trainable params: 0
_________________________________________________________________
```

Since we have a binary classification we use relu units as activation function and the output has a sigmoid as the activation function. Relu is not reliable as the activation function for the output layer because we do not need a discrete value to decide the classification but a probability function to do so. We set a batch size of 5 since we have very few data points to work with and test this on 200 epochs.

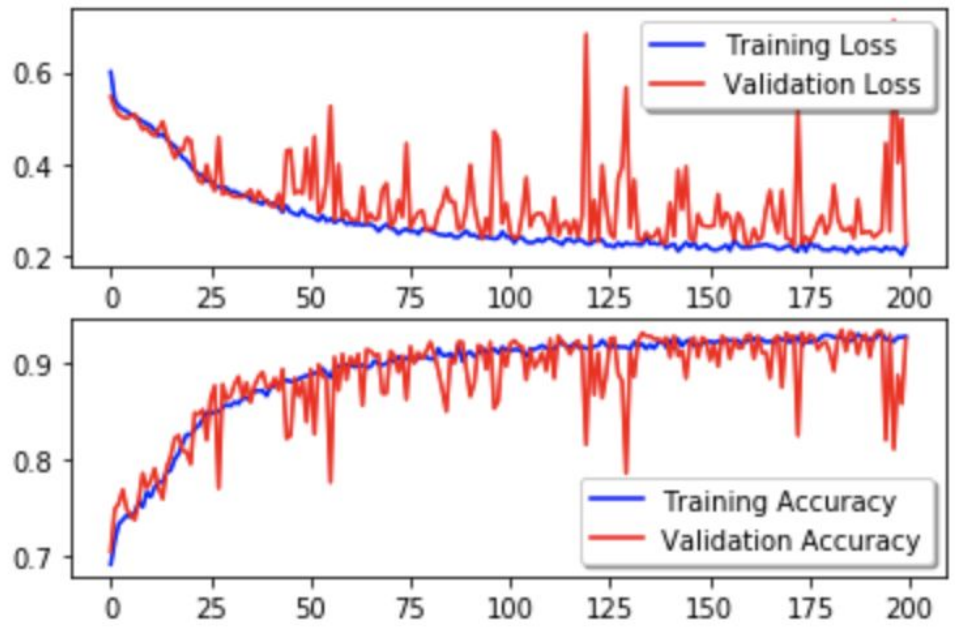
We use RMSprop as the optimizer here. The resulting graph of epochs vs loss or accuracy is very spiky and hence the learning rate is decreased from a default of 0.001 to 0.0001 and the results are tabulated.

The resulting output of this model is as shown below:

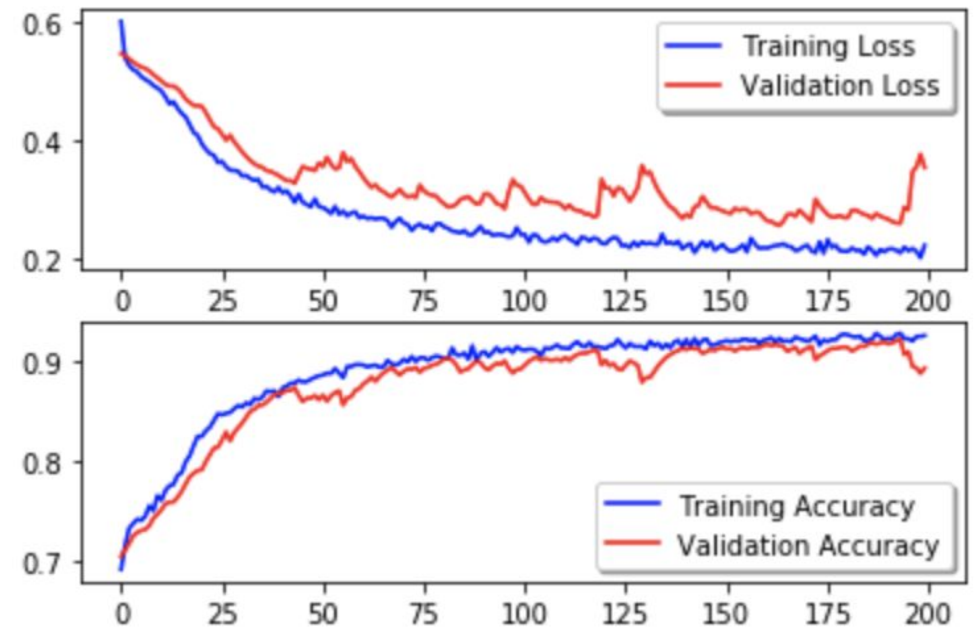


```
1380/1380 [==============================] - 0s 16us/step
1380/1380 [==============================] - 0s 15us/step
644/644 [==============================] - 0s 17us/step
Validation: accuracy = 0.923913  ;  loss_v = 0.225154
Test: accuracy = 0.928261  ;  loss = 0.227976
```

Even though we obtain a desirable result we obtain a spiky graph for this model as shown below:



Hence a smoothing function is written to better visualize the variation of loss and accuracy with respect to epochs. The resulting graph is as shown below:



Results and discussion:

After observing the result of the model designed above it is noted that reducing the learning rate substantially reduces the spikes in the graph but doesn't quite do the job.

The spikes in the graph can be due to various reasons some of them are:

- Not enough data points
- Certain data points are biased which causes the model to learn something new when it sees this data point.
- Large learning rate.
- Feature scaling and normalization is not done.

The first two reasons can not be addressed because we do not have more data. If the data present is already biased and is causing the spikes, resampling or bootstrapping may result in expanding the dataset consisting of more of these biased datasets.

References:

- https://archive.ics.uci.edu/ml/datasets/spambase
- http://www.cs.iit.edu/~agam/cs577/share/ker.pdf
- https://medium.com/aidevnepal/for-sigmoid-funcion-f7a5da78fec2

Task 3:

Problem Statement:

To perform regression on a UCL repository called Communities and Crime and to predict the violent crimes per population column.

Proposed solution:

The dataset given contains 127 columns as attributes with 1,994 data points. This dataset has an awful lot of missing values to be treated. After data preparation and normalization is completed a fully connected network is built. The validation set is created by k-fold cross validation.

Mse and Mae are calculated from the model.evaluate function and tabulated.

Implementation and methodology:

The data is loaded into a python pandas dataframe. This data contains a lot of missing values in the form of '?' and NaN. First things first columns that have data types different from float of integers are dropped (Column community name is dropped). Attributes like state , country, community have a lot of missing data and hence it can not be included in our data set. Fold attribute has mostly the same kind of data = 1and wouldn't affect the output of regression and hence is removed.

Manual inspection of data is time consuming and hence the rest of the data is filtered based on the number of missing values. Since we have ,missing values in the form of '?' and Nan we convert all the missing values into one format which is Nan.

It is observed that one of the columns OtherPerCap has only one missing value and is hence replaced with the median of the column.

Finally all columns which have missing values are dropped from the dataframe. The data frame obtained is divided into test and train datasets.

Normalization is performed by calculating the mean and standard deviation of the test data and performing the mathematical operation (x- mean) / standard deviation. The resulting data obtained is then fed to our model.

Model:

Since we have a limited number of datapoints we create a model that is as simple as possible. We create a model with two hidden layers of 64 neurons each and an output layer of 1 neuron. The activation function for these layers is a simple relu function except that the output layer doesn't have any activation layer since we are performing regression.

K for cross validation is performed with k = 4, and the resulting mean average error is calculated. These are the following results:

```
processing fold # 0
processing fold # 1
processing fold # 2
processing fold # 3
```

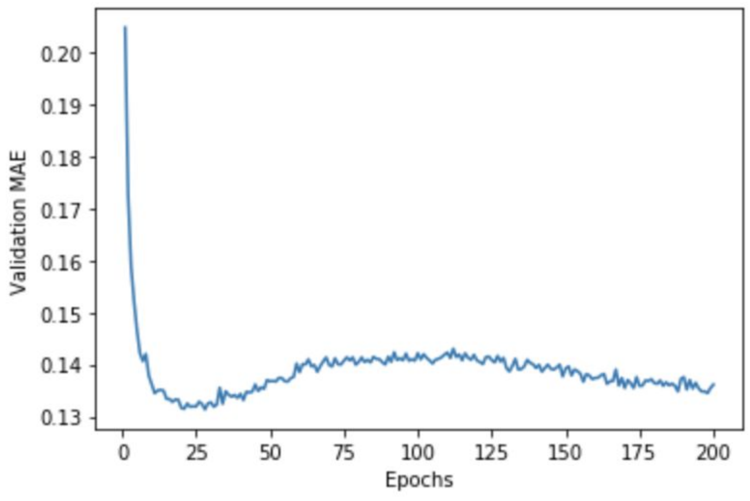
```
all_scores
```

```
[0.13888193666934967,
 0.130196675658226,
 0.14208023250102997,
 0.1334351897239685]
```

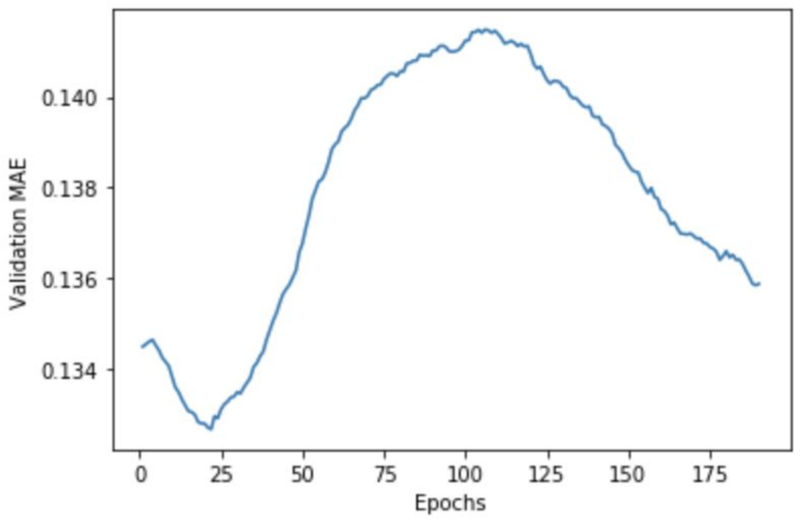
```
np.mean(all_scores)
```

```
0.13614850863814354
```

This is the validation mean squared error. The graph of MAE vs epochs are as follows :

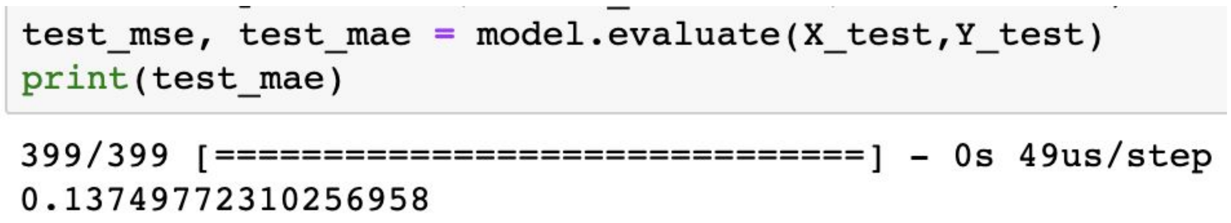


Since the graph is grainy we smoothen the graph with a smoothen function and the resulting graph obtained is as follows:



From this we create a tuned model and run our test data through it. We set the epochs to 20 as we can see an increase in the MAE score after epoch 25.

The resulting test mean squared error is tabulated and is as follows:



```
test_mse, test_mae = model.evaluate(X_test,Y_test)
print(test_mae)
```

```
399/399 [==============================] - 0s 49us/step
0.13749772310256958
```

Result and discussion:

From the model we see that performing k fold cross validation over the data is necessary due to fact the we have very few data points. Tuning the model is essential as we tend to overfit the data as we train more and more epochs. The resulting Mean average error states that we are 14,000 violent crimes away for a 100k population.

References:

- https://machinelearningmastery.com/custom-metrics-deep-learning-keras-python/
- http://www.cs.iit.edu/~agam/cs577/share/ker.pdf