DEEP LEARNING PROJECT 1

ROHIT MURALI

Project Assignment #1: Convolutional Neural Networks (CNNs)

OBJECTIVE:

In this assignment, we have to implement and train a convolutional neural network (CNN) (using Keras) to classify cats and dogs. We use Google's Colaboratory (Colab) for accessing free GPUs (the data set needs to be pre-processed into binary files for training) to speed up the time taken for training the CNN.

PREPARING THE DATASET:

The dataset which we are using for this project is the "Cats and Dogs" dataset. This dataset consists of a total of 25,000 images. The number of images for dogs and cats are equally split such that there are 12,500 images of both classes.

We create two folders **train** and **validation** (each with two subfolders: dogs and cats) for training and validation data. We split the 'Cats and Dogs' data set into a training data set (of 10,000 dog images & 10,000 cat images) and a validation data set (of 2,499 dog images & 2,499 cat images).

The resultant file structure looks like this:

The images in these data sets are not all of the same size, which means that we have to resize them so that they are all of the same size before training. Also, we need to generate a target label for each image(0 for cat, 1 for dog) , so that the CNN can have an output label to recognize each image based on its features.

During the pre-processing part we take the images from the training and validation data set and resize them as 150 × 150 × 3, so that the time taken for training these images is reduced. Apart from resizing the images we also generate labels for each image based on the names of the images.

We save the pre-processed training and validation data sets as binary pickle files which we then upload on google drive.

ACCESSING FILES FROM GOOGLE COLAB:

In order to access the files which we upload on our drive, we need to include the following statements in our colab file:

```
from google.colab import drive
drive.mount('/content/gdrive')
```

Once, we execute this we can access our pickle files by navigating to the appropriate directory.

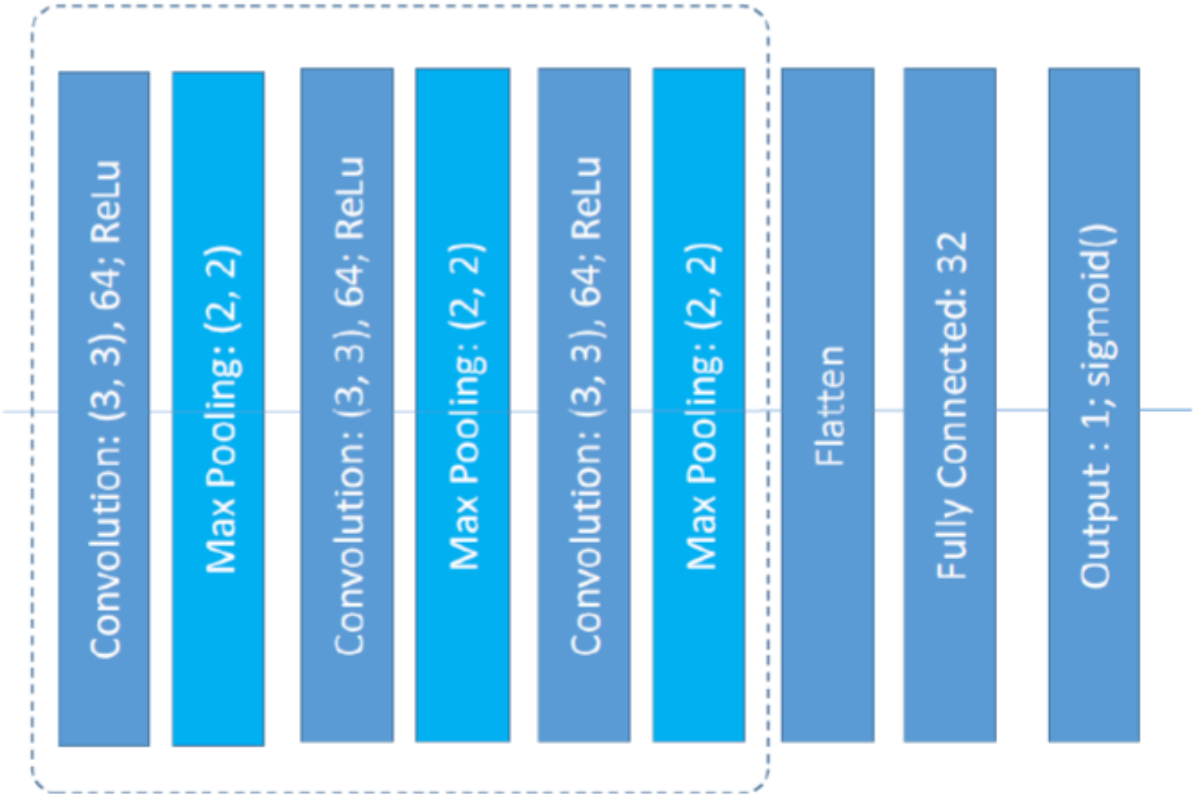
NORMALIZING THE DATA:

Once we load the training data set into colab we need to normalize the data. This is done so that the CNN doesn't have to work with very high pixels values which will take up comparatively more time for processing through each epoch.

We normalize the data by dividing each image in the training data set by 255, as 255 is the highest value which would be present in each image.

EXPERIMENTAL RESULTS:

We will build our CNN with the following layers.



We have 3 convolution layers which each have 64, 3 × 3 filters. The dimension of the image reduces as it passes through the CNN. After each convolutional layer we have a 2 × 2 Max Pooling layer which reduces the width and height of the image after each convolution layer by 2. At the end of the last max pooling layer we have a flatten layer along with a fully connected layer and output layer to form a deep neural network. The number of filters remain constant in each layer of the model. The summary of the number of parameters involved in each layer is shown in the image below:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 148, 148, 64)      1792
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 74, 74, 64)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 72, 72, 64)        36928
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 36, 36, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 34, 34, 64)        36928
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 17, 17, 64)        0
_________________________________________________________________
flatten (Flatten)            (None, 18496)             0
_________________________________________________________________
dense (Dense)                (None, 32)                591904
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 33
=================================================================
Total params: 667,585
Trainable params: 667,585
Non-trainable params: 0
_________________________________________________________________
```

For training the CNN, the activation function used in all of the convolutional and fully connected layers is the “ReLu” activation function. The output layer uses the “Sigmoid” function as this is a classification problem and sigmoid gives us the probability of the image if being of a particular class.

For compiling the model we have used the ADAM optimizer, along with the binary_crossentropy loss function and the number of epochs being 10.

We load the validation data set from google drive. Similar to the training data set, the validation data set is also normalized and then used in the model.fit function for hyperparameter tuning.

The accuracy of the model during the training dataset is shown in the image below.

Using ADAM optimizer

```
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [==============================] - 608s 30ms/sample - loss: 0.1977 - acc: 0.9187 - val_loss: 0.4176 - val_acc: 0.8302
Epoch 2/10
20000/20000 [==============================] - 601s 30ms/sample - loss: 0.1689 - acc: 0.9318 - val_loss: 0.4390 - val_acc: 0.8400
Epoch 3/10
20000/20000 [==============================] - 600s 30ms/sample - loss: 0.1471 - acc: 0.9413 - val_loss: 0.4932 - val_acc: 0.8298
Epoch 4/10
20000/20000 [==============================] - 600s 30ms/sample - loss: 0.1222 - acc: 0.9517 - val_loss: 0.5078 - val_acc: 0.8236
Epoch 5/10
20000/20000 [==============================] - 598s 30ms/sample - loss: 0.0974 - acc: 0.9627 - val_loss: 0.5429 - val_acc: 0.8284
Epoch 6/10
20000/20000 [==============================] - 603s 30ms/sample - loss: 0.0826 - acc: 0.9689 - val_loss: 0.7353 - val_acc: 0.8012
Epoch 7/10
20000/20000 [==============================] - 602s 30ms/sample - loss: 0.0764 - acc: 0.9704 - val_loss: 0.6321 - val_acc: 0.8300
Epoch 8/10
20000/20000 [==============================] - 628s 31ms/sample - loss: 0.0475 - acc: 0.9823 - val_loss: 0.7813 - val_acc: 0.8250
Epoch 9/10
20000/20000 [==============================] - 621s 31ms/sample - loss: 0.0439 - acc: 0.9845 - val_loss: 0.7986 - val_acc: 0.8250
Epoch 10/10
20000/20000 [==============================] - 605s 30ms/sample - loss: 0.0280 - acc: 0.9900 - val_loss: 0.8059 - val_acc: 0.8228
```

Using RMSProp optimizer

```
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [==============================] - 609s 30ms/sample - loss: 0.6943 - acc: 0.5742 - val_loss: 0.6708 - val_acc: 0.6026
Epoch 2/10
20000/20000 [==============================] - 599s 30ms/sample - loss: 0.5744 - acc: 0.6953 - val_loss: 0.5439 - val_acc: 0.7420
Epoch 3/10
20000/20000 [==============================] - 614s 31ms/sample - loss: 0.4973 - acc: 0.7576 - val_loss: 0.4751 - val_acc: 0.7752
Epoch 4/10
20000/20000 [==============================] - 616s 31ms/sample - loss: 0.4482 - acc: 0.7903 - val_loss: 0.4441 - val_acc: 0.7952
Epoch 5/10
20000/20000 [==============================] - 594s 30ms/sample - loss: 0.4103 - acc: 0.8117 - val_loss: 0.4245 - val_acc: 0.8052
Epoch 6/10
20000/20000 [==============================] - 605s 30ms/sample - loss: 0.3682 - acc: 0.8347 - val_loss: 0.4668 - val_acc: 0.7862
Epoch 7/10
20000/20000 [==============================] - 594s 30ms/sample - loss: 0.3348 - acc: 0.8526 - val_loss: 0.6965 - val_acc: 0.7258
Epoch 8/10
20000/20000 [==============================] - 595s 30ms/sample - loss: 0.2986 - acc: 0.8711 - val_loss: 0.4554 - val_acc: 0.7964
Epoch 9/10
20000/20000 [==============================] - 594s 30ms/sample - loss: 0.2611 - acc: 0.8890 - val_loss: 0.4834 - val_acc: 0.8048
Epoch 10/10
20000/20000 [==============================] - 595s 30ms/sample - loss: 0.2245 - acc: 0.9064 - val_loss: 0.4254 - val_acc: 0.8234
```

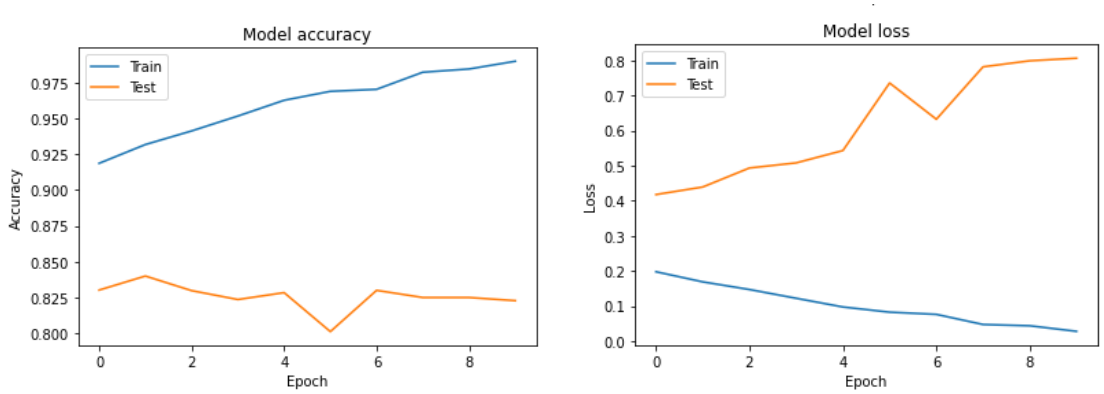
Using ADADelta optimizer

```
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [==============================] - 617s 31ms/sample - loss: 0.6410 - acc: 0.6303 - val_loss: 0.5718 - val_acc: 0.7172
Epoch 2/10
20000/20000 [==============================] - 610s 30ms/sample - loss: 0.5590 - acc: 0.7167 - val_loss: 0.5413 - val_acc: 0.7258
Epoch 3/10
20000/20000 [==============================] - 619s 31ms/sample - loss: 0.5275 - acc: 0.7385 - val_loss: 0.5212 - val_acc: 0.7494
Epoch 4/10
20000/20000 [==============================] - 620s 31ms/sample - loss: 0.5086 - acc: 0.7538 - val_loss: 0.5369 - val_acc: 0.7252
Epoch 5/10
20000/20000 [==============================] - 615s 31ms/sample - loss: 0.4954 - acc: 0.7640 - val_loss: 0.5203 - val_acc: 0.7330
Epoch 6/10
20000/20000 [==============================] - 613s 31ms/sample - loss: 0.4846 - acc: 0.7690 - val_loss: 0.4904 - val_acc: 0.7640
Epoch 7/10
20000/20000 [==============================] - 625s 31ms/sample - loss: 0.4738 - acc: 0.7800 - val_loss: 0.4735 - val_acc: 0.7732
Epoch 8/10
20000/20000 [==============================] - 630s 32ms/sample - loss: 0.4648 - acc: 0.7836 - val_loss: 0.4638 - val_acc: 0.7840
Epoch 9/10
20000/20000 [==============================] - 618s 31ms/sample - loss: 0.4572 - acc: 0.7880 - val_loss: 0.4767 - val_acc: 0.7764
Epoch 10/10
20000/20000 [==============================] - 622s 31ms/sample - loss: 0.4492 - acc: 0.7939 - val_loss: 0.4570 - val_acc: 0.7880
```

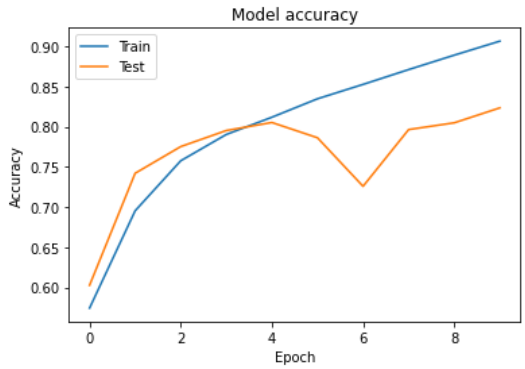
After the model is trained its weight are saved into an h5 file in google drive and loaded any time we would want to use the trained model.

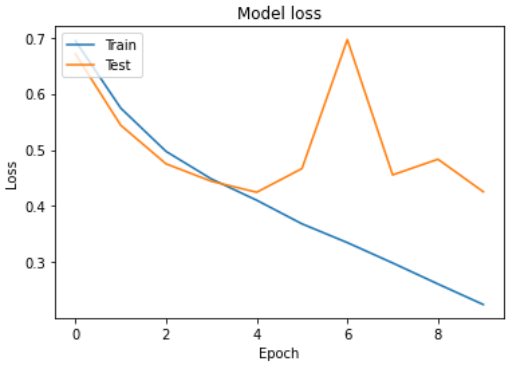
PLOTS:

Using ADAM optimizer

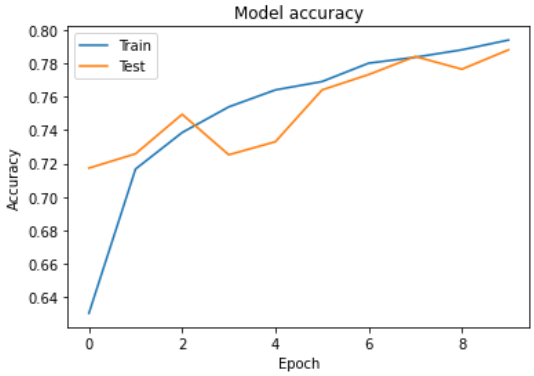


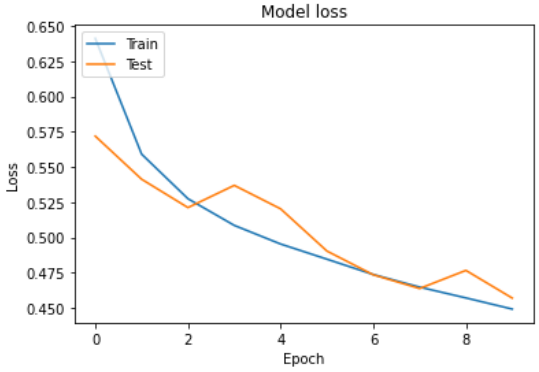
Using RMSProp optimizer





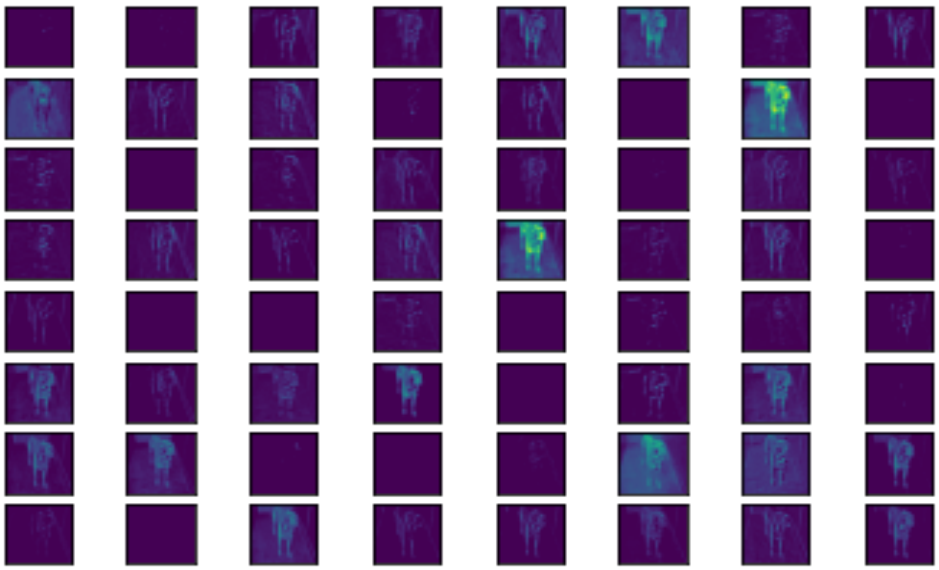
Using ADADelta optimizer



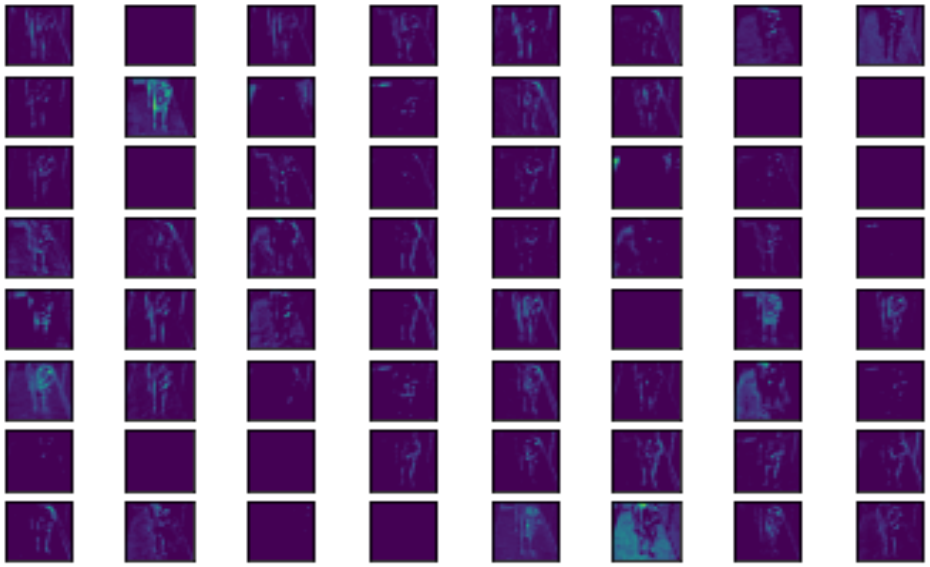


The feature maps obtained for each convolutional layer is plotted below as a subplot of 8 × 8 images.

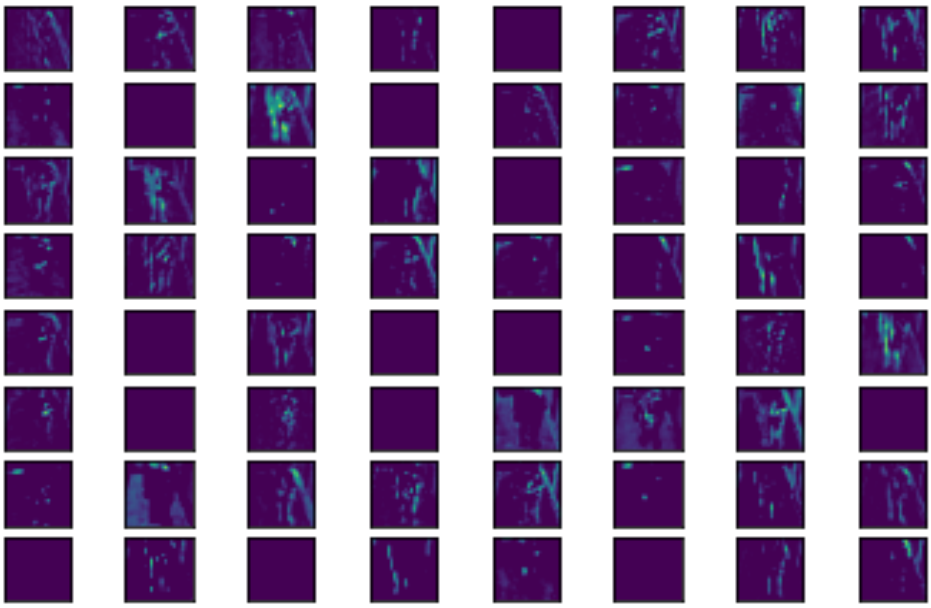
The feature map of the first convolutional layer is shown below. Each image in this feature map has a size of 148 × 148 pixels and the total size of the feature map is 1 × 148 × 148 × 64.



The feature map of the second convolutional layer is shown below. Each image in this feature map has a size of 72 × 72 pixels and the total size of the feature map is 1 × 72 × 72 × 64.



The feature map of the second convolutional layer is shown below. Each image in this feature map has a size of 34 × 34 pixels and the total size of the feature map is 1 × 34 × 34 × 64.



DISCUSSIONS AND CONCLUSION:

As we can see in the case of **ADAM** optimizer the training data has been overfit because of which we get 99% accuracy in the training data set. But in the validation data set, we get the final accuracy to be around approximately 83%. Since the data is being overfit, we can conclude that the model beings to remember the input image patterns instead of learning from the input images. Hence, because of this we get a constant decrease in training loss and an increase in the validation loss while executing the "model.fit" function.

In case of the **RMSProp** optimizer, there is a steady increase in the training accuracy but the model does overfit the data since the training accuracy is not very high. The validation accuracy increases in the start and saturates after a certain time. While observing the training and validation loss plots, we can see that both of the losses decrease steadily with some increasing spikes in the validation loss.

While using the **ADADelta** optimizer we can see that the training and validation accuracies and losses are almost the same. Although this is a good sign for predictions on unseen data, the model as whole would not be very accurate in classifying images as the accuracy obtained during the training is just approximately 80%.

To reduce the overfitting problem we can use one of the following methods:

1 – Regularization

2 – Decreasing the learning rate during training

3 – Dropout

4 – Data Augmentation

5 – Batch normalization

For the case of predicting labels for images, we test it with images from the validation data set. The target labels are either 0 or 1, representing a cat or a dog respectively. Since, our neural network model has only 1 output node, the way it predicts images is for a 0 label image, the output prediction is a very small number and the predicted output for an image with label 1 is very close to 1.

For the feature map outputs, we see that the convolution layers deep inside the CNN detect patterns of edges in each of the images. This is because as we go deep into the CNN the image becomes smaller and smaller and only the most dominant and unique features remain. This helps the filter applied to the images to detect the features which will differentiate between a cat and a dog as well as to identify features that represent images of the same class.