NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

EE4483 Artificial Intelligence and Data Mining

Mini Project (Option 2)

Student: Agarwal Mehal

Matric Number: U1823940E

TA Group: GP08

2021-2022

This project has been implemented using TensorFlow.

The answers to the questions are as follows:

Question (a): State the amount of image data you used to form your training and testing set. Describe the data pre-processing procedures and image augmentations (if any).

Answer:

Dataset Distribution

The distribution of the dataset is as follows:

- The training set being used in this project comprises of 20,000 images in total, among which it contains 10,000 images for each of the two classes: cat and dog.
- The validation set contains 5,000 images in total, among which it comprises of 2,500 images for each of the two classes.
- The test set comprises of 500 images.

Data exploration was performed.

In order to understand the dataset distribution better, a bar plot is used to visualize the number of images in the training, validation and test sets. The plot is shown in Fig. 1. Further, in order to visualize the class distribution, a bar plot was created to present the number of images for each class (cat and dog) in the training and validation sets and is shown in Fig. 2.

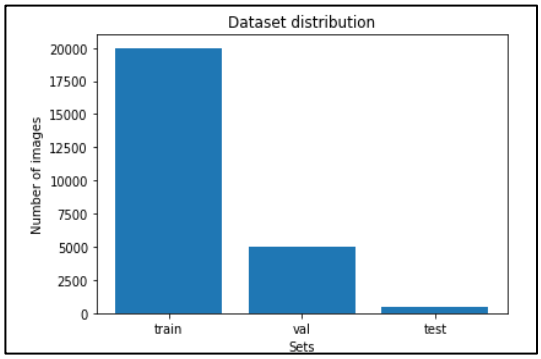


Figure 1

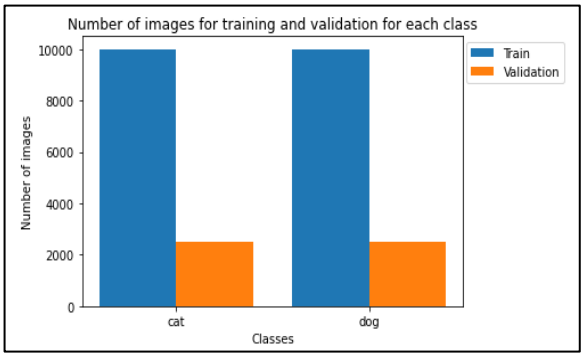


Figure 2

From the above, we can see that the samples in the training and validation sets are balanced. This is because both training and validation sets contain equal number of images for each of the two classes.

Data Pre-processing

While loading the data using Keras image processing API, the following data preprocessing procedures were used:

1. Resize the images to 224x224x3 pixels

   The images are resized before modeling so that all images have the same shape when being passed through the model for classification. The size of the image affects the time taken by the model to train as small inputs imply that the model trains faster.

In this project, we choose a fixed size of 224x224x3 pixels and all images are resized to this shape when being loaded.

2. Rescale the pixel values in an image (which are between 0 to 255) to the interval of 0 to 1.

Rescaling is done to normalize the pixel values to the range of 0 to 1. It is done by multiplying the pixel values with a rescale value of 1/255. The RGB values in an image range from 0 to 255 which would be too high for the model to process. Hence, the pixel values are rescaled to be in the range of 0 to 1 for ease of computation.

After data loading, we explore the data by visualizing some images from the training and validation sets by plotting some sample images and their corresponding labels. The corresponding code is included in the attached notebook.

Some samples visualized from the raining set are as follows:

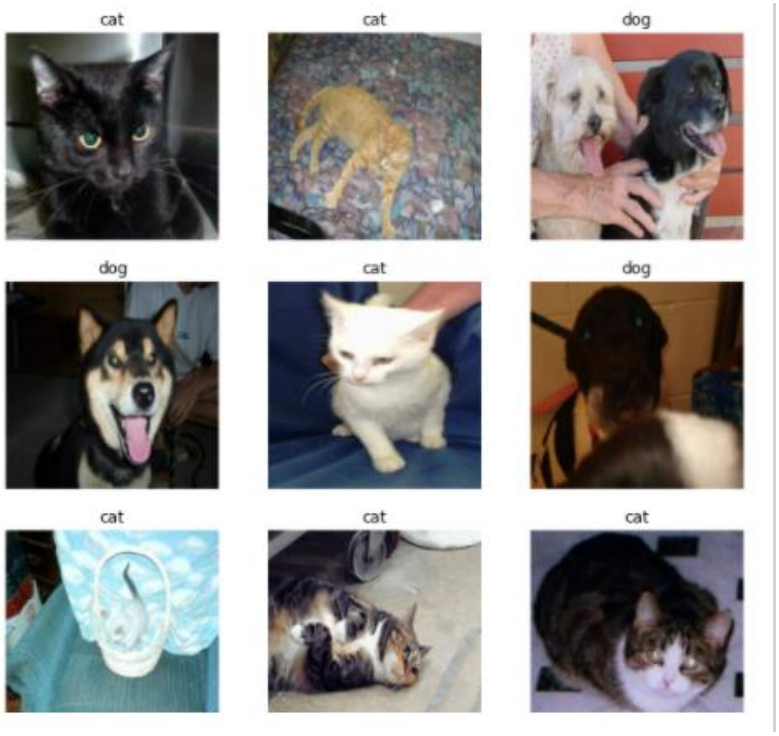


Figure 3: Some examples of visualized samples from training set

Some samples visualized from validation set are as follows:

Figure 4: Some examples of visualized samples from validation set

In figure 3, we can see that in one of the images (bottom left corner), the cat has very less visibility whereas in Figure 4, we can see that one of the images (bottom right corner) has two dogs. This implies that the classification algorithm to be used for this dataset needs to be robust.

Image Augmentation

Data augmentation is a method to modify the data available in a realistic but randomized way to increase the variety of data seen during training. It's used to increase the data complexity and helps in preventing overfitting. Augmentation techniques create variations in images which improves the generalizability of the model.

The image augmentation techniques used in this project are:

- **Rotation_range = 40**: In this method of augmentation, the images are rotated by an angle selected at random from the range of 0 to 40 degrees.
  This augmentation technique is used to help the model become robust to changes in the orientation of objects. In this case for our problem, even if we rotate an image, the information of the image remains the same. For example, a cat from a different angle remains a cat and the same is applicable to a dog.
- **width_shift_range = 0.2**: This method of augmentation shifts the image left or right by a value selected at random from the range of (-0.2*224) to (0.2* 224) where 224 pixels is the width of the image. Here, the value of 0.2 indicates the percentage of the width of the image to shift.

- **height_shift_range = 0.2:** This method of augmentation shifts the image up or down by a value selected at random from the range of (-0.2*224) to (0.2*224) where 224 pixels is the height of the image. Here, the value of 0.2 indicates the percentage of the height of the image to shift.

  There may be instances where the objects in the image are not centered in the frame and can be off-center in different ways. This augmentation technique of horizontal and vertical random shifting helps in training the network to expect and currently handle the off-center objects by artificially creating shifted versions of the training data.

- **shear_range = 0.2:** The shear transformation is a process of slanting the image by fixing one axis and stretching the image at an angle of 20 degrees (in this case) in the counter-clockwise direction.

  In this process, the image will be distorted along an axis to create or rectify the perception angles.

- **zoom_range = 0.2:** This augmentation technique is used for randomly zooming in or out of the image. We are using a zoom_range of [1-0.2, 1+0.2], i.e., [0.8, 1.2] which indicates zooming between 80% (zoom in) and 140% (zoom out). This is because a value<1 in the range will result in zooming in, making the object bigger whereas a value>1 will result in zooming out, making the object smaller.

  This method generates samples with a random zoom that is different on the width and height dimensions and changes the aspect ratio of the object in an image.

- **horizontal_flip = True:** This method of augmentation randomly flips images with respect to the vertical axis.

  This is useful as the objects in a sample can have varied orientation. Since we are using images of cats and dogs, vertical flipping of images with respect to the horizontal axis will not make any sense and hence is not used in this case.

- **fill_mode = 'nearest'**: This technique is used to fill the empty area after some operations, e.g., rotation. 'nearest' fill mode replaces the empty area with the nearest pixel values.

These augmentations are specified as arguments to ImageDataGenerator for training set. Data augmentation is not applied to validation set which is used to evaluate the trained model as the model should be evaluated using unmodified validation data. Hence separate ImageDataGenerator instances are created for training and validation sets and the respective datasets are loaded using the corresponding data generators.

A screenshot of the code to perform data loading and image augmentation is shown in Fig 5. The corresponding code is included in the attached notebook.

```
def load_data():
    train_datagen = ImageDataGenerator(rescale = 1./255.,
                                       rotation_range = 40,
                                       width_shift_range = 0.2,
                                       height_shift_range = 0.2,
                                       shear_range = 0.2,
                                       zoom_range = 0.2,
                                       horizontal_flip = True)

    validation_datagen = ImageDataGenerator(rescale = 1.0/255.)

    train_dataset = train_datagen.flow_from_directory(train_dir,
                                                      target_size=(224, 224),
                                                      batch_size=32,
                                                      class_mode='binary',
                                                      seed =123)
    validation_dataset = validation_datagen.flow_from_directory(val_dir,
                                                         target_size=(224, 224),
                                                         batch_size=32,
                                                         class_mode='binary',
                                                         seed =123)
    return train_dataset, validation_dataset
```

Figure 5: Data Loading and Augmentation

After performing data augmentation, we plot some samples from the train dataset to visualize the effect of data augmentation. A few visualized samples are shown in Fig 6.

Figure 6: Some samples from training set after data augmentation

Question (b). Select or build at least one machine learning model (e.g., CNN + linear layer, etc.) to construct your classifier. Clearly describe the model you use, including a figure of model architecture, the input and output dimensions, structure of the model, loss function(s), training strategy, etc. Include your code and instructions on how to run the code. If non-deterministic method is used, ensure reproducibility of your results by averaging the results over multiple runs, cross-validation, fixing random seed, etc.

Answer:

In this project we use two machine learning models to classify dog and cat images. The first is a Convolutional Neural Network (CNN) model [1] which is built from scratch and the second model is built using transfer learning which uses the VGG-16 pretrained feature extraction backbone.

For both the models, the input is an image with a fixed size of 224x224x3 pixels.

The two models used to construct the classifier are described as follows:

1. **Convolutional Neural Network (CNN) Model**

The first model is a convolutional neural network model which has been built from scratch. It comprises of 3 pairs of alternating convolutional and max pooling layers, followed by a flatten layer and 2 dense layers. These layers use He Uniform weight initialization and Relu activation function except for the last layer which uses sigmoid activation function. These layers are stacked to form the CNN architecture. The description of these layers are as follows:

- **Convolutional Layers**: These layers are used to extract various features from input images by performing convolution between the input and a filter of a given size. The output is feature maps which give information about the input image. These feature maps are passed to the other layers. In this project, the properties of the convolutional layers used in this model are as follows:
  - Filter size = 3x3.
  - Stride = 1. The convolutional stride is fixed to 1 pixel.
  - Padding = 'same'. The spatial padding of input is such that the spatial resolution is preserved after convolution. Hence the padding is 1 pixel for 3x3 convolutional layers.
  - Activation function = Relu. Relu activation function is used in these layers It adds non-linearity to the network.
  - Weight initializer = He Uniform

  These layers are added using ***Conv2D()*** function from Keras.

- **Pooling layers**: These layers perform down sampling and are used to reduce the size of the feature maps which reduces the number of parameters, which in turn reduces the computational costs.
  In this project, Max Pooling is used. The properties of the max pooling layers in the model constructed are as follows:
  - Filter size = 3x3
  - Stride = 2

  These layers are added using ***MaxPooling2D()*** function from Keras

- **Flatten:** This layer is used to flatten the feature maps obtained from the previous layers into a single vector which can be used as input for the following dense layers.
  This layer is added using the ***Flatten()*** function from Keras

- **Dense Layers**: We construct one fully connected/ dense layer. This forms the classifier part of the model.
    - Number of nodes/units = 128
    - Activation function = Relu

  This layer is added using the ***Dense()*** function from Keras.

- **Output Layer**: This layer is constructed using a dense layer with one node/unit which gives the predicted label as output.
    - Number of nodes/units = 1
    - Activation function=Sigmoid. This is because we are handling the given problem as a binary classification problem where the output can either be 0 (cat) or 1 (dog).

The architecture of the model constructed is shown in Fig 7.

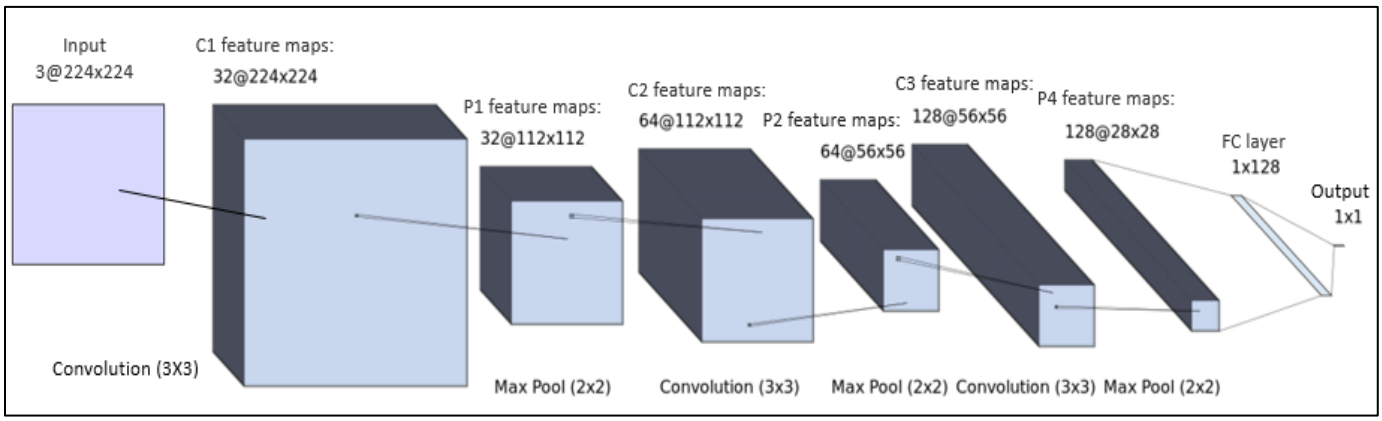


Figure 7: CNN Architecture

The model is built using the Sequential API of Keras which allows the creation of the network by simply defining the list of layers sequentially [2]. A screenshot of the code used to build the model is shown in Fig 8. The complete code is attached in the notebook.

```
def build_baseline_CNN():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=(224, 224, 3)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dense(1, activation='sigmoid'))

    return model
```

Figure 8: Code to build the CNN model

The CNN model has 10 layers and the input and output shape for each layer have been summarized in Table 1:

Layer Type	Kernel size	Input shape	Output shape
Input layer	-	224x224x3	224x224x3
Conv_1	3x3	224x224x3	224x224x32
Max Pooling	2x2	224x224x32	112x112x32
Conv_2	3x3	112x112x32	112x112x64
Max Pooling	2x2	112x112x64	56x56x64
Conv_3	3x3	56x56x64	56x56x128
Max Pooling	2x2	56x56x128	28x28x128
Flatten	-	28x28x128	1x100352
Dense_1	1x1	1x100352	1x128
Dense_2	1x1	1x128	1x1

Table 1: Input and output dimensions for each layer

The corresponding structure of the model was constructed using the tf.keras.utils.plot_model() function from TensorFlow and is shown in the first column of Table2.

Dropout Regularization: In order to add regularization to the model, the dropout regularization method is used. Dropout randomly ignores some number of layer outputs during training and makes the network more robust. It limits the model complexity and helps in reducing overfitting.

Hence, the CNN model shown above is updated. A dropout of 20% is applied after each pair of convolutional and max pooling layers and a larger dropout of 50% is applied after the dense layer in the classifier part of the model. The dropout layers are added to the model using the *Dropout()* function from Keras. The updated model with dropout regularization forms the CNN model used in this project. The screenshot of the code to build the CNN model is shown in Fig. 9.

```
def build_model_CNN():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=(224, 224, 3)))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dropout(0.5))
    model.add(Dense(1, activation='sigmoid'))
    return model
```

Figure 9: Code to build the CNN model with dropout regularization

The architecture of the model after adding dropout regularization is shown in Fig 10.

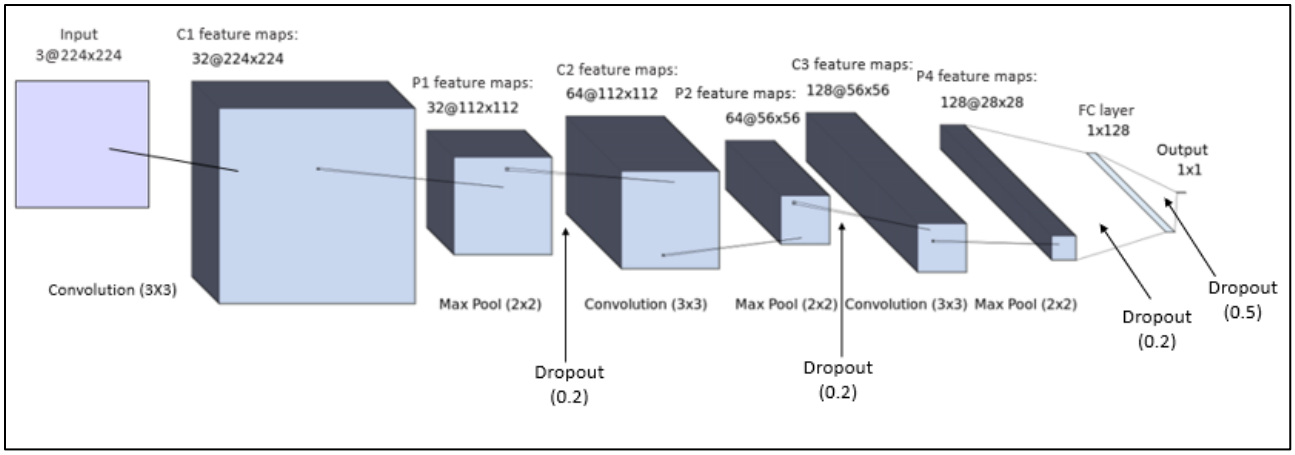


Figure 10: CNN Architecture after adding dropout regularization

The structure of the corresponding model was constructed using the tf.keras.utils.plot_model() function from TensorFlow and is shown in the second column of Table2.

The summary of the CNN model is shown in Fig 11. which gives an overview of the layers and presents the number of parameters in the different layers.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 224, 224, 32)      896
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 112, 112, 32)      0
_________________________________________________________________
dropout (Dropout)            (None, 112, 112, 32)      0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 112, 112, 64)      18496
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 56, 56, 64)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 56, 56, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 56, 56, 128)       73856
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 28, 28, 128)       0
_________________________________________________________________
dropout_2 (Dropout)          (None, 28, 28, 128)       0
_________________________________________________________________
flatten (Flatten)            (None, 100352)            0
_________________________________________________________________
dense (Dense)                (None, 128)               12845184
_________________________________________________________________
dropout_3 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 129
=================================================================
Total params: 12,938,561
Trainable params: 12,938,561
Non-trainable params: 0
_________________________________________________________________
```

Figure 11: CNN model summary

The CNN model without dropout regularization is used as the baseline model which will be used to analyze the effect of pre-processing on model performance which is presented in answer (f).

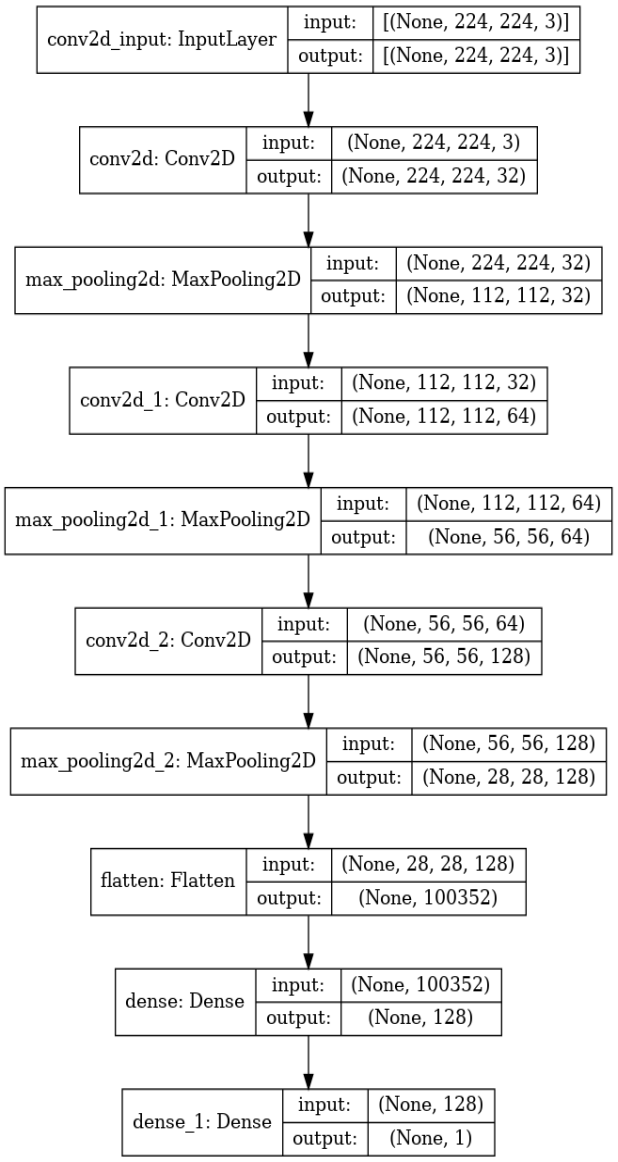
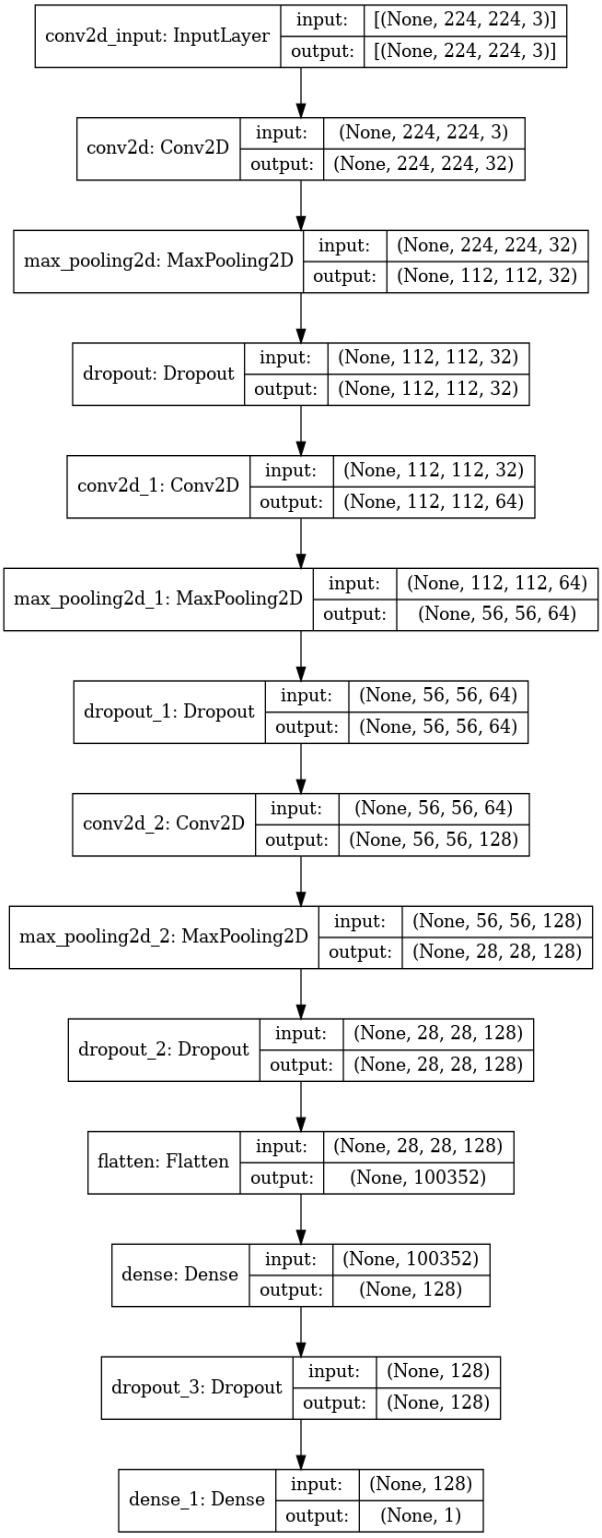
Baseline CNN Model Structure without Regularization	CNN Model Structure with Regularization
	

Table 2: Structure of the CNN models

2. Model with Transfer Learning (VGG-16)

Transfer learning involves using all or parts of a model which was trained on a related task.

In his project, we utilize the VGG-16 [3] model as the feature extraction backbone and the pretrained model is loaded using the Keras Applications API. The VGG-16 model is one of the most popular pretrained models for image classification and it achieved the top results in the ImageNet image classification challenge.

The VGG-16 model has 16 layers and comprises of two parts: (1) feature extractor made up of VGG blocks and (2) classifier part made up of fully connected layers and the output layer.

In this project, we use the feature extraction part and add a new classifier part to the model related to the cats and dogs dataset being used. The weights of the convolutional layers are frozen during training and are made non-trainable. The new fully connected layer will be trained which learns to interpret the features extracted from the model and performs binary classification.

The architecture of the VGG-16 model is shown in Fig 12.

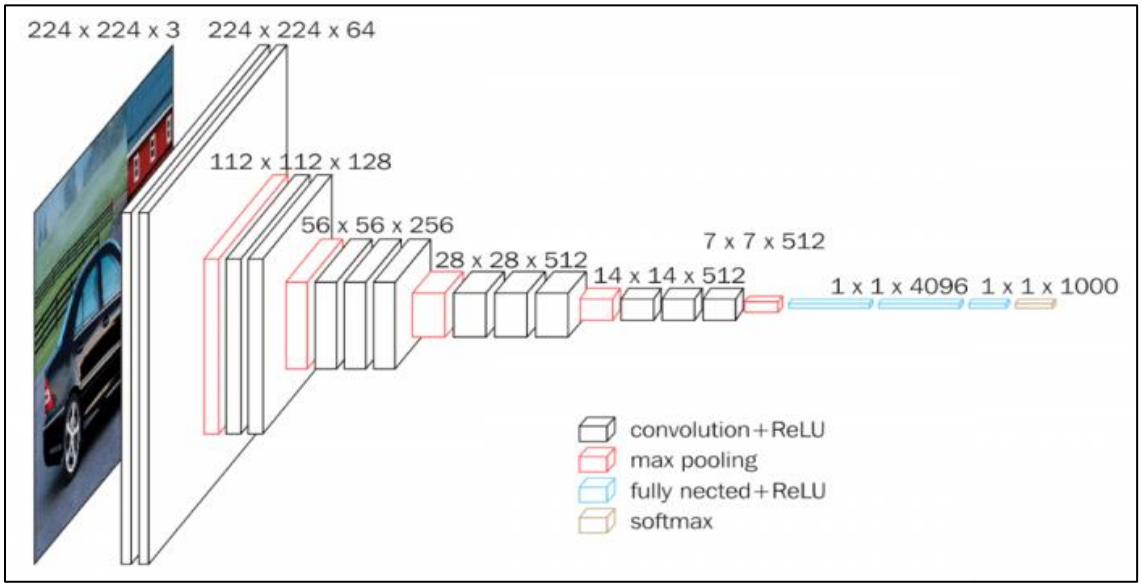


Figure 12: VGG-16 Architecture

The structure of the VGG-16 model is shown in Fig 13.

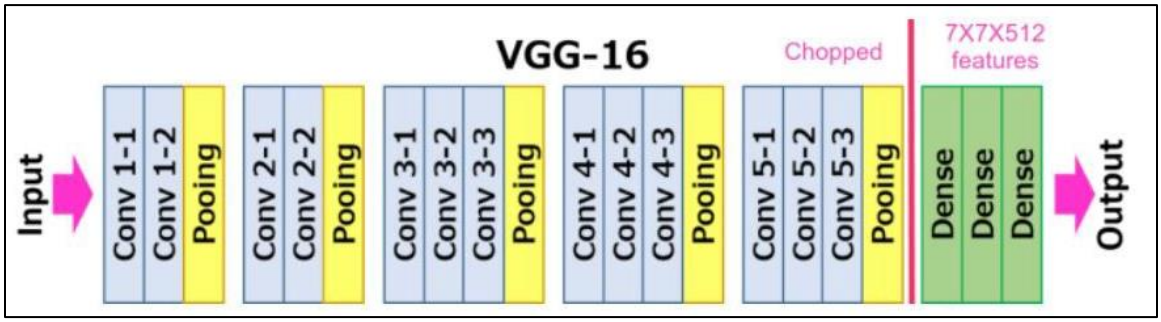


Figure 13: VGG-16 Structure

The model implemented in this project, removes the fully-connected/dense layers from the output end of the model shown above. Hence, we utilize the VGG-16 model till the last max pooling layer, after which a flatten layer is added followed by a dense layer with 128 units and Relu activation function. The output layer is formed by a dense layer consisting of 1 unit with the sigmoid activation function. This forms the new classifier part. The structure of only the new classifier implemented in this project as a continuation of the VGG-16 feature extractor part is shown in Fig 14. The complete structure of the VGG-16 model implemented is attached in the appendix.

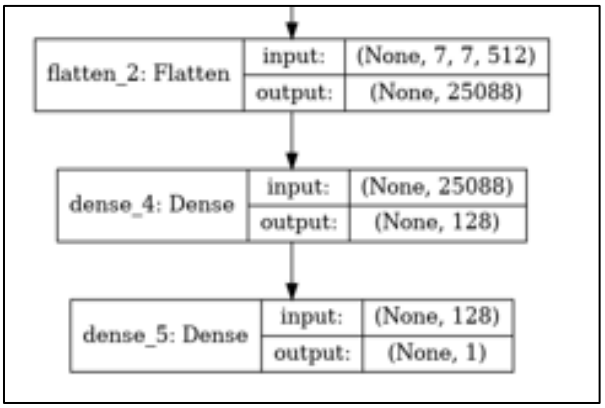


Figure 14: New classifier part of the VGG-16 model implemented

The screenshot of the code used to build the VGG-16 model is shown in Fig. 15.

```
def build_model_VGG16():
    model = VGG16(include_top=False, input_shape=(224, 224, 3))
    # loaded layers set to be not trainable
    for layer in model.layers:
        layer.trainable = False
    # addition of new classifier layers
    flat1 = Flatten()(model.layers[-1].output)
    class1 = Dense(128, activation='relu', kernel_initializer='he_uniform')(flat1)
    output = Dense(1, activation='sigmoid')(class1)
    # define new model
    model = Model(inputs=model.inputs, outputs=output)

    return model
```

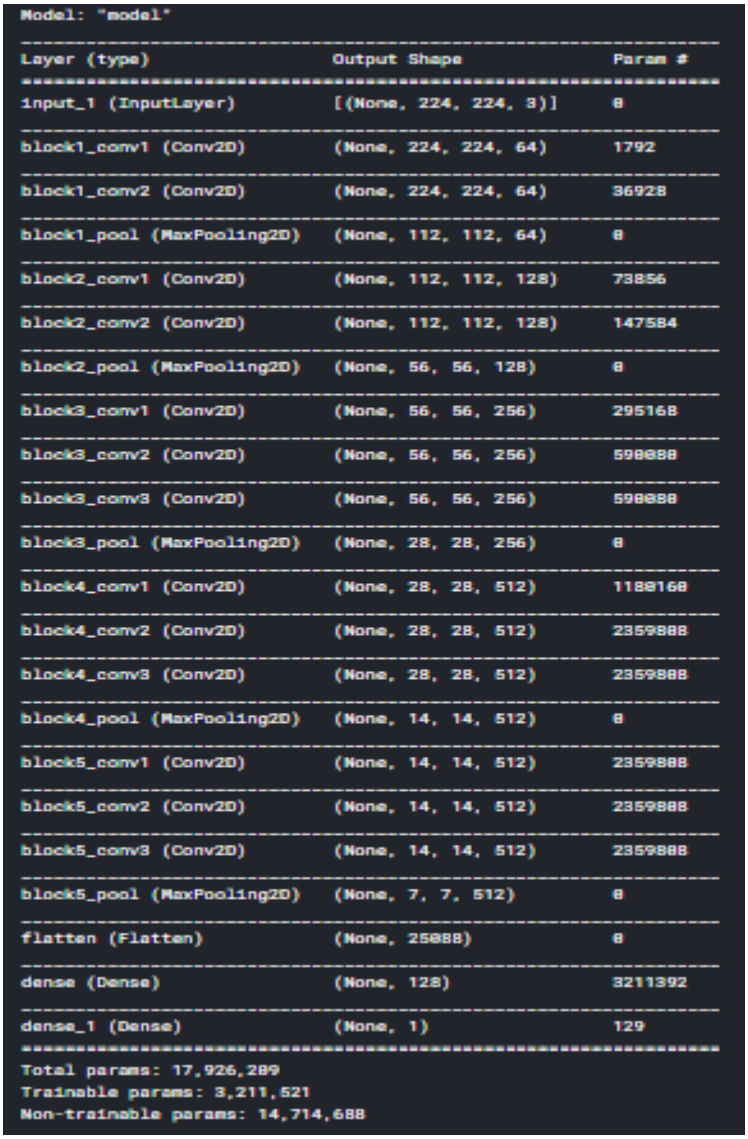
Figure 15: Screenshot of the code to build VGG-16 model

As shown in Figure 15, while loading the VGG-16 model, we set include_top = False, so that the classifier part of the model is not loaded.

The model comprises of 13 convolutional layers, 5 max pooling layers and one dense layer. The convolutional layers use 3x3 filters with a stride of 1 and same padding. The max pooling layers use a 2x2 pixel window with a stride of 2. The input to the model is an image of size 224x224x3 pixels which is also the size of the image using which the VGG-16 pretrained model was configured. The details and the dimensions of input and output shape for each component of the model are as follows:

1. **Input layer**: This layer is the image of dimension 224x224x3 pixels.
2. **Convolutional Layer (Conv1)**: This block comprises of two convolution layers with 64 filters each of size 3x3 and 1 max pooling layer. It uses ReLU activation function.
   Input shape: 224x224x3 and output shape: 112x112x64
3. **Convolutional Layer (Conv2)**: This block comprises of two convolution layers with 128 filters each of size 3x3 and 1 max pooling layer. It uses ReLU activation function.
   Input shape: 112x112x64 and output shape: 56x56x128
4. **Convolutional Layer (Conv3):** This block comprises of three convolution layers with 256 filters each of size 3x3 and 1 max pooling layer. It uses ReLU activation function.
   Input shape: 56x56x128 and output shape: 28x28x256
5. **Convolutional Layer (Conv4)**: This block comprises of three convolution layers with 512 filters each of size 3x3 and 1 max pooling layer. It uses ReLU activation function.
   Input shape: 28x28x256 and output shape: 14x14x512
6. **Convolutional Layer (Conv5):** This block comprises of three convolution layers with 512 filters each of size 3x3 and 1 max pooling layer. It uses ReLU activation function.
   Input shape: 14x14x512 and output shape: 7x7x512
7. **Flatten Layer**: The 7x7x512 output is flattened to generate a feature vector of size (1,25088)
8. **Dense Layer:** This layer has 128 nodes/units and uses ReLU activation function It takes the feature vector from the flatten layer of dimension (1,25088) as input and generates a feature vector of dimension (1,128) as output.
9. **Output Layer**: This layer is constructed using a dense layer with one node/unit which gives the predicted label as output. It uses the sigmoid activation function as we are handling a binary classification problem where the output is either 0 (cat) or 1 (dog).

The summary of the VGG-16 model is shown in Fig 16. which gives an overview of the layers and presents the number of parameters in the different layers.



```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 224, 224, 3)]     0
_________________________________________________________________
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
_________________________________________________________________
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
_________________________________________________________________
flatten (Flatten)            (None, 25088)             0
_________________________________________________________________
dense (Dense)                (None, 128)               3211392
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 129
=================================================================
Total params: 17,926,289
Trainable params: 3,211,521
Non-trainable params: 14,714,688
```

Figure 16: Summary of the VGG-16 model

Training Strategy

After constructing both the models, the models are compiled using the *compile()* method where the loss, optimizer and metrics parameters are specified.

The parameters for compiling the model are as follows:

- **Loss function**: binary cross-entropy loss
  The models are optimized using the binary cross-entropy loss function since we are training a classification model with two possible outputs. Binary cross entropy loss computes the cross-entropy loss between the true and predicted labels.
- **Optimizer**: We experiment with three different choices of optimizers which are: Adam, Stochastic Gradient Descent (SGD) and Root Mean Squared Propagation (RMSProp) [4]. The learning rate is controlled by the optimizer.
- **Metrics**: Accuracy

The accuracy metric is used to determine the accuracy score on the validation set when training the model.

The screenshot of the code used to compile the model is shown in Fig 17.

```
#compile model
model.compile(optimizer=opt,
              loss=tf.keras.losses.binary_crossentropy,
              metrics=['accuracy'])
```

Figure 17: Screenshot of the code to compile the model

After compiling, the model is trained using the *fit()* function which takes the training data, validation data and number of epochs as arguments. The batch size used for loading the training and validation data is 32. Hence, the number of steps per epoch for training data = 20000/32 = 625 and that for validation data = 5000/32 = 157.

The *fit()* function returns a history object containing all the training information once the training is finished. This object is used to visualize the metrics and analyze the performance of a model. The screenshot of the code used to fit the model is shown in Fig 18. and the complete code is in the attached notebook.

```
history = model.fit(train_dataset,
                    validation_data=validation_dataset,
                    epochs=hp['epoch'])
```

Figure 18: Screenshot of the code to fit the model

The training is tried with 10 epochs and then with 20 epochs and their performances are compared. Different sets of learning rates are used to train the model with different optimizers.

In order to automate the process of compiling and training the model with different parameters, we define a list of nested dictionaries with different sets of parameter configurations. Each dictionary has the keys: ‘epoch’, ‘optimizer’ and ‘lr’ (learning rate) to define each configuration’s parameters. The corresponding code for the different parameter configurations that have been tried are included in the attached notebook.

After the training is completed, we visualize the training results by plotting the loss and accuracy for the training and validation sets as a function of the number of epochs. The code for plotting the training results is included in the attached notebook.

Code & Instructions:

The complete code has been attached as a notebook in the .ipynb format. The code has been modularized for ease of implementation by writing different components of the project in different functions. For this project, I used the Kaggle platform to run the code. In order to ensure reproducibility of results, the method of fixing a random seed is used while loading batches of training and validation data to train the model. This is shown in Fig 5.

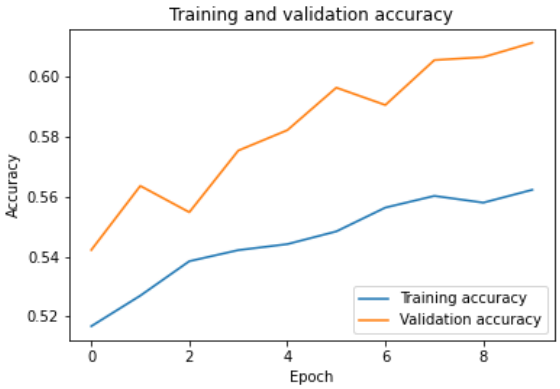
Question (c): Discuss how you consider and determine the parameters (e.g., learning rate, etc.) / settings of your model as well as your reasons of doing so.

Answer:

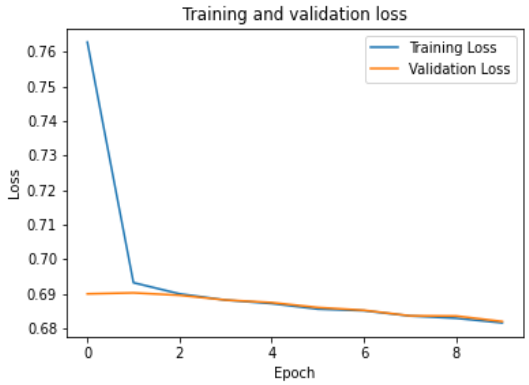
Using the **CNN** model (with dropout regularization), we optimize the model using three different optimizers which are Adam, Stochastic Gradient Descent (SGD) and RMSProp. The initial learning rates used with these optimizers are 0.00001, 0.001, 0.00001 respectively and we train the model for 10 and 20 epochs. We analyze the performance of the CNN model with different sets of parameters by plotting the training and validation accuracy and loss as a function of the number of epochs.

1. We use Adam optimizer with a learning rate of 0.00001 and train the CNN model for 10 and 20 epochs. The following plots are obtained after training.

Optimizer: Adam
Learning Rate: 0.00001
No. of Epochs: 10

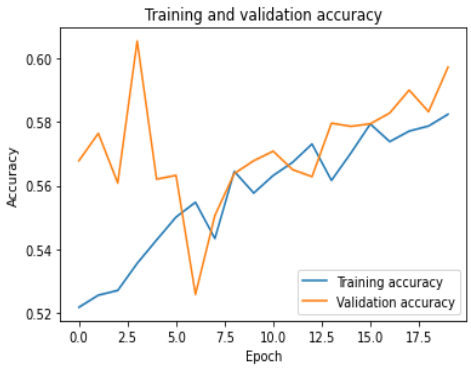


Training Accuracy = 52.35%
Validation accuracy = 55.18%

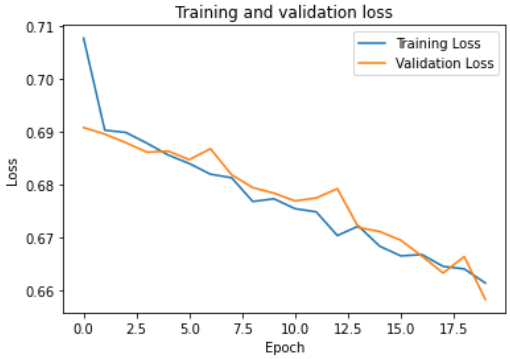


Training Loss = 0.68707
Validation Loss = 0.68164

Optimizer: Adam
Learning Rate: 0.00001
No. of Epochs: 20



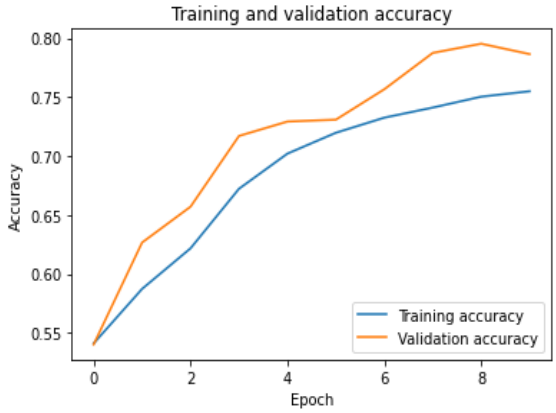
Training Accuracy = 55.01%
Validation accuracy = 59.72%



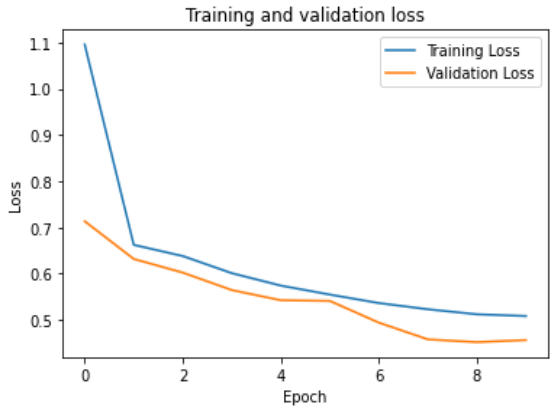
Training Loss = 0.68251
Validation Loss = 0.65830

On analyzing the training and validation loss graphs shown above for both 10 and 20 epochs, we observe that the training loss is converging very slowly. This usually happens if the learning rate is small. Further, the performance of the model in terms of accuracy is around 60% which implies that the model is not performing well with these parameters. Hence, we try to increase the learning rate to 0.001 and train the CNN model while using Adam optimizer for both 10 and 20 epochs. The following plots are obtained after training with these parameters:

Optimizer: Adam
Learning Rate: 0.001
No. of Epochs: 10

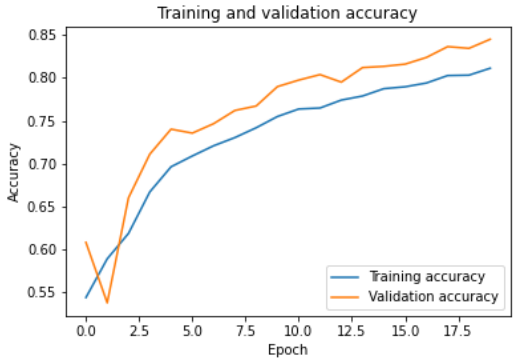


Training Accuracy = 77.58%
Validation accuracy = 78.64%

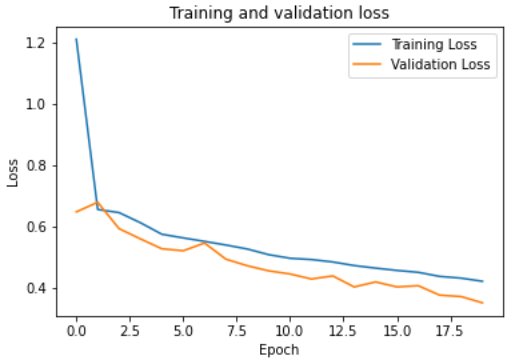


Training Loss = 0.46836
Validation Loss = 0.45627

Optimizer: Adam
Learning Rate: 0.001
No. of Epochs: 20



Training Accuracy = 83.25%
Validation accuracy = 84.5%



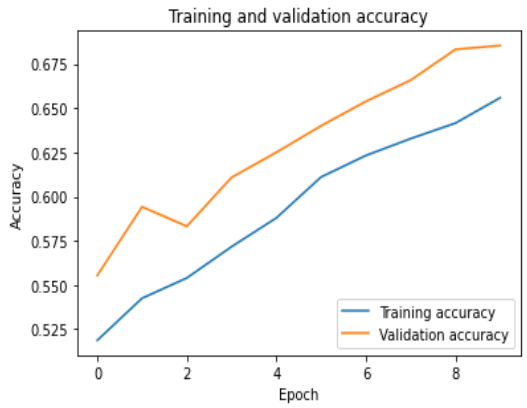
Training Loss = 0.3798
Validation Loss = 0.35387

On increasing the learning rate from 0.00001 to 0.01 when using Adam optimizer, we see a significant improvement in model performance (for 20 epochs, the model performance improves by around 25%). Further, the model performs better when trained for 20 epochs compared to 10 epochs as the

loss converges further. By analyzing the loss curves for different sets of parameters, we observe that there is almost no indication of overfitting which implies that this model is robust and generalizable.

2. We now analyze the model performance by using the Stochastic Gradient Descent (SGD) optimizer with a learning rate of 0.001 and momentum of 0.9 for 10 and 20 epochs. The following plots are obtained after training.

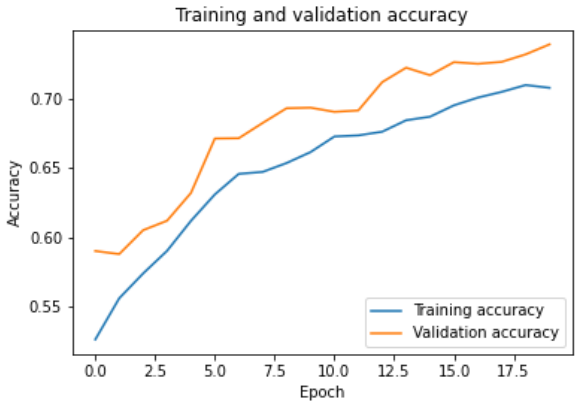
Optimizer: SGD
Learning Rate: 0.001
No. of Epochs: 10



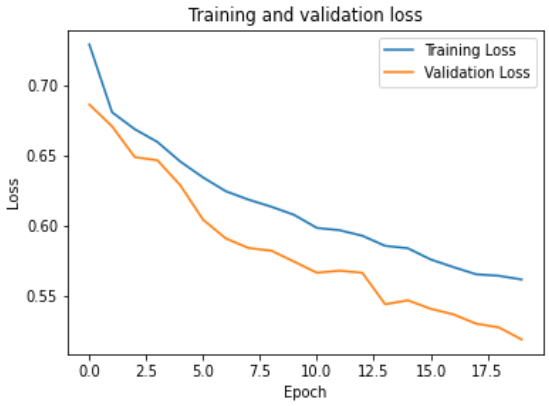
Training Accuracy = 66.02%
Validation accuracy = 68.54%

Training Loss = 0.60831
Validation Loss = 0.57757

Optimizer: SGD
Learning Rate: 0.001
No. of Epochs: 20



Training Accuracy = 73.34%
Validation accuracy = 73.88%

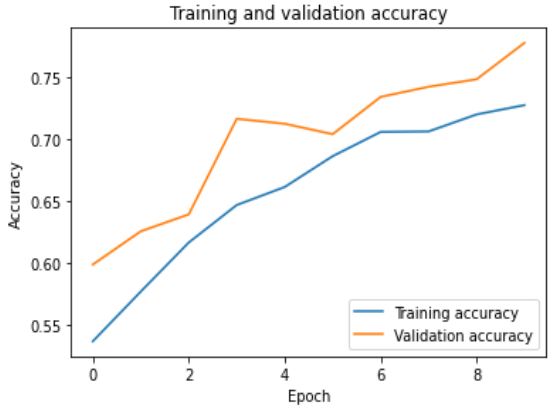


Training Loss = 0.54467
Validation Loss = 0.51892

We observe that the validation accuracy improves on training the model for 20 epochs compared to 10 epochs as the loss converges further. On analyzing the training and validation loss graphs shown above for both 10 and 20 epochs, we observe that the training loss is converging very slowly. This

usually happens if the learning rate is small. Hence, we try to increase the learning rate to 0.01 and train the CNN model using SGD optimizer for both 10 and 20 epochs. The following plots are obtained after training with these parameters:

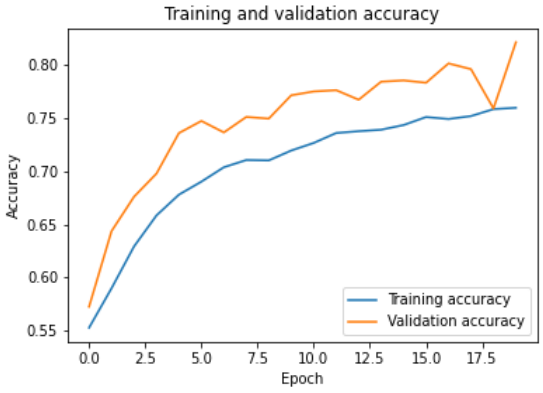
Optimizer: SGD
Learning Rate: 0.01
No. of Epochs: 10



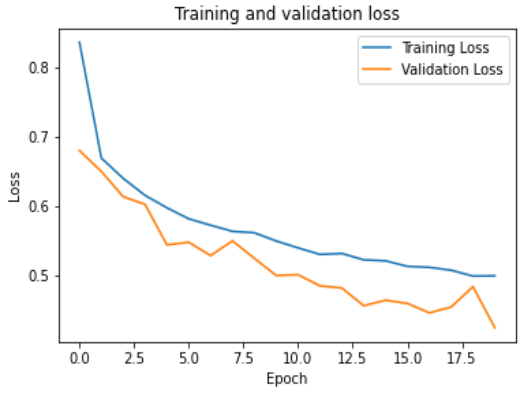
Training Accuracy = 73.05%
Validation accuracy = 77.74%

Training Loss = 0.53481
Validation Loss = 0.47709

Optimizer: SGD
Learning Rate: 0.01
No. of Epochs: 20



Training Accuracy = 76.99%
Validation accuracy = 82.18%

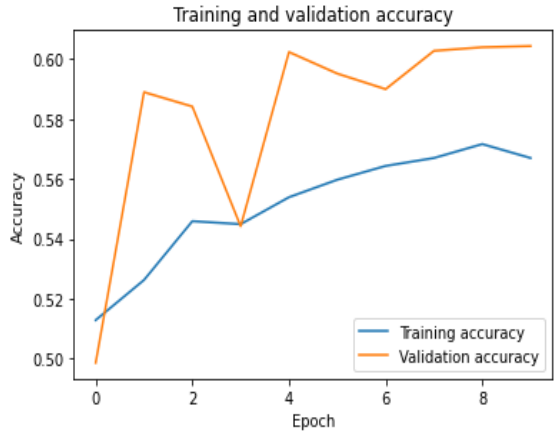


Training Loss = 0.48624
Validation Loss = 0.42551

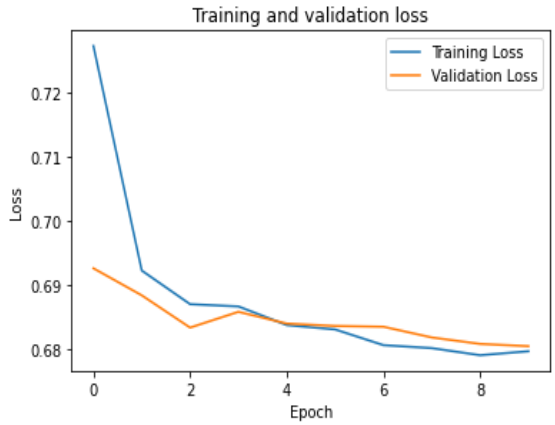
On increasing the learning rate from 0.001 to 0.01 when using SGD optimizer, we see an improvement in model performance (for 20 epochs, the model performance improves by around 10%) and a better convergence in training loss. Further, the model performs better when trained for 20 epochs compared to 10 epochs. By analyzing the loss curves for different sets of parameters, we observe that there is almost no indication of overfitting which implies that the model is robust and generalizable.

3. We now analyze the model performance by using the RMSProp optimizer with a learning rate of 0.00001 for 10 and 20 epochs. The following plots are obtained after training.

Optimizer: RMSProp
Learning Rate: 0.00001
No. of Epochs: 10

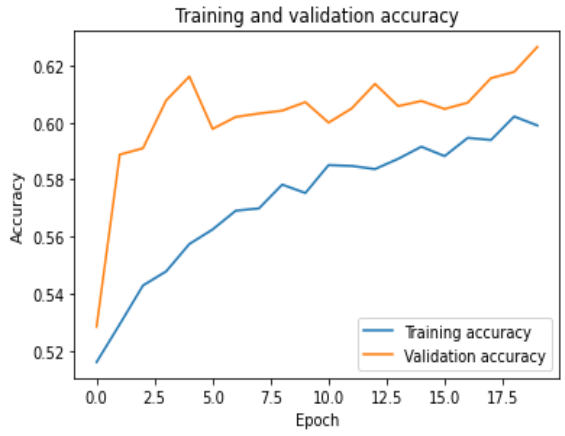


Training Accuracy = 58.18%
Validation accuracy = 60.44%



Training Loss = 0.6380
Validation Loss = 0.68055

Optimizer: RMSProp
Learning Rate: 0.00001
No. of Epochs: 20



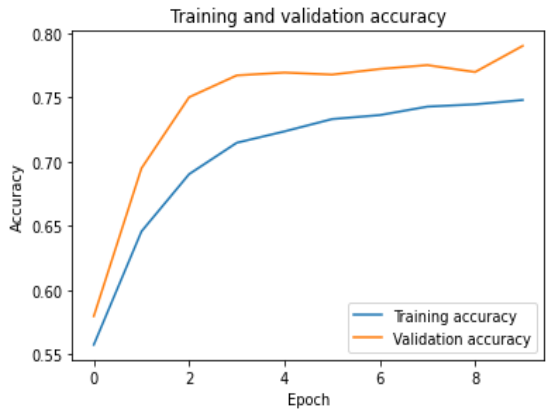
Training Accuracy = 60.57%
Validation accuracy = 62.66%

Training Loss = 0.66918
Validation Loss = 0.65687

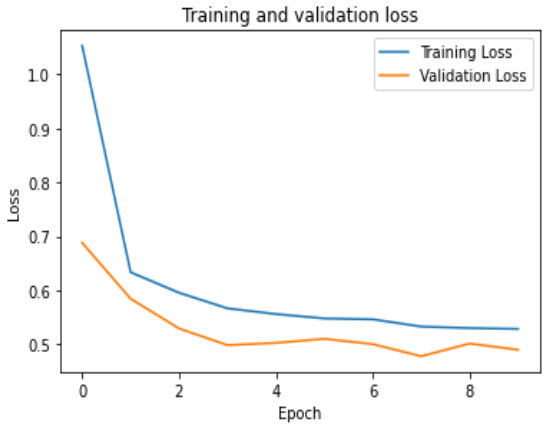
On analyzing the training and validation loss graphs shown above for both 10 and 20 epochs, we observe that there is some gap between the training and validation loss curves which indicates slight overfitting. The performance of the model improves from training for 10 epochs to 20 epochs. However, the performance of the model in terms of accuracy is around 62% which implies that the model is not performing well with these parameters. Further, on analyzing the training loss curves, it is evident that the loss is converging slowly which indicates that the learning rate is low. Hence, we

try to increase the learning rate to 0.001 and train the CNN model while using RMSProp optimizer for both 10 and 20 epochs. The following plots are obtained after training with these parameters:

Optimizer: RMSProp
Learning Rate: 0.001
No. of Epochs: 10

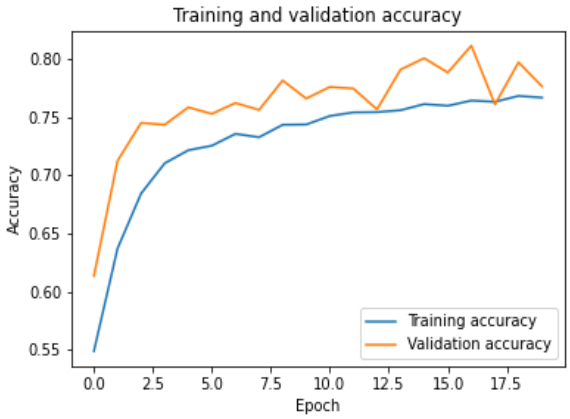


Training Accuracy = 76.84%
Validation accuracy = 79.02%

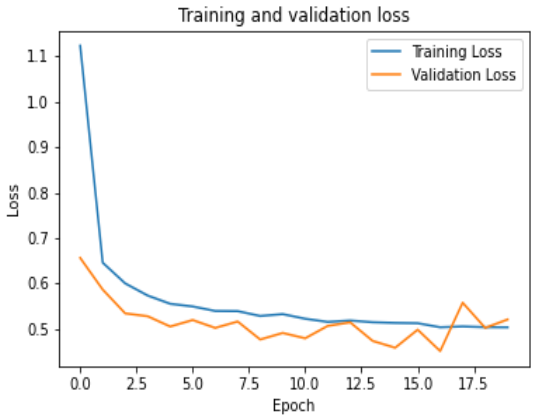


Training Loss = 0.53395
Validation Loss = 0.4901

Optimizer: RMSProp
Learning Rate: 0.001
No. of Epochs: 20



Training Accuracy = 75.74%
Validation accuracy = 77.68%



Training Loss = 0.53857
Validation Loss = 0.52106

On increasing the learning rate from 0.00001 to 0.001 when using RMSProp optimizer, we see an improvement in model performance (for 10 epochs, the model performance improves by around 20%) and a better convergence in training loss. Further, the model performs better when trained for 10 epochs compared to 20 epochs. By analyzing the loss curves for different sets of parameters, we observe that there is almost no indication of overfitting which implies that the model is robust and generalizable.

After training the CNN model with different sets of parameters, we obtain the best performance when we use Adam optimizer with a learning rate of 0.01 for 20 epochs. With these parameters we obtain a validation accuracy of 84.5 % for the CNN model (with dropout regularization).

The different sets of parameters used to train the CNN model and their corresponding training and validation loss and accuracy have been summarized in the table below:

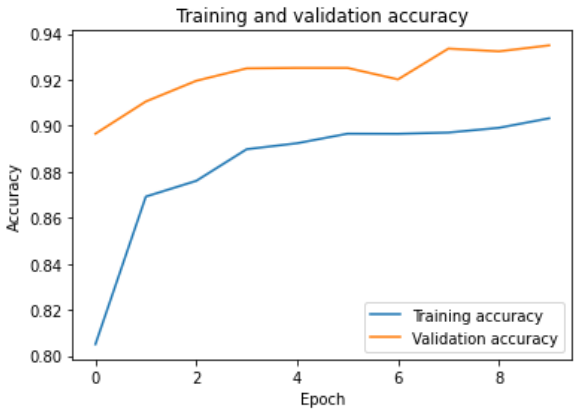
Optimizer	**Learning Rate**	**Training Loss**	**Training Accuracy (%)**	**Validation Loss**	**Validation Accuracy (%)**
Adam	0.001	0.46836	77.58	0.45627	78.64
SGD	0.001	0.60831	66.02	0.57757	68.54
RMSProp	0.001	0.53395	76.84	0.4901	79.02
Adam	0.00001	0.68707	52.35	0.68164	55.18
SGD	0.01	0.53481	73.05	0.47709	77.74
RMSProp	0.00001	0.68233	55.13	0.67179	59.54
Adam	0.001	0.3798	83.25	0.35387	84.5
SGD	0.001	0.54467	73.34	0.51892	73.88
RMSProp	0.001	0.53857	75.74	0.52106	77.68
Adam	0.00001	0.68665	54.74	0.66909	58.62
SGD	0.01	0.48624	76.99	0.42551	82.18
RMSProp	0.00001	0.6727	58.61	0.65727	62.16

Table 3: Training results of CNN model with different parameters

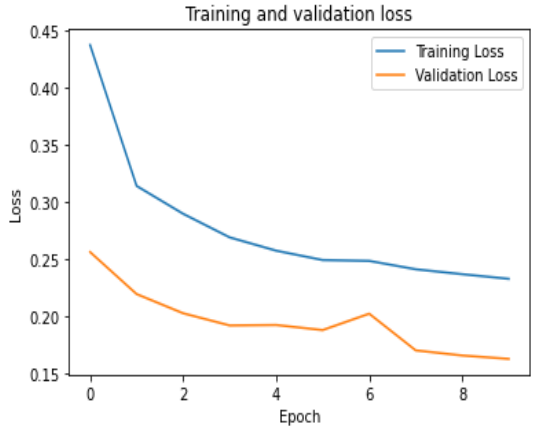
Using the **VGG-16** transfer learning model, we train the model using three different optimizers which are Adam, Stochastic Gradient Descent (SGD) and RMSProp. The initial learning rates used with these optimizers are 0.00001, 0.001, 0.00001 respectively and we train the model for 10 and 20 epochs. We analyze the performance of the model with different sets of parameters by plotting the training and validation accuracy and loss as a function of the number of epochs.

1. We use Adam optimizer with a learning rate of 0.00001 and train the VGG-16 transfer learning model for 10 and 20 epochs. The following plots are obtained after training.

Optimizer: Adam
Learning Rate: 0.00001
No. of Epochs: 10

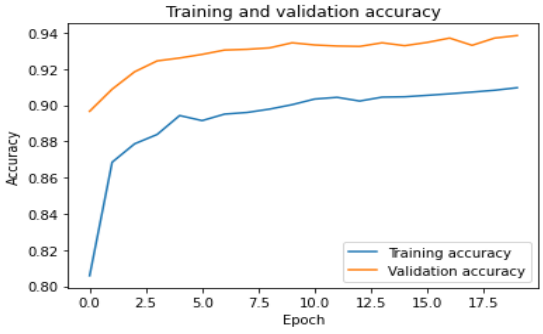


Training Accuracy = 89.73%
Validation accuracy = 93.5%

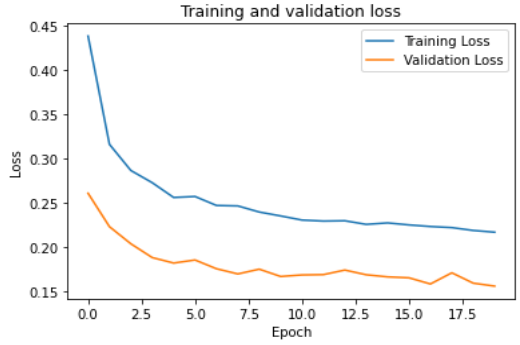


Training Loss = 0.23716
Validation Loss = 0.16319

Optimizer: Adam
Learning Rate: 0.00001
No. of Epochs: 20



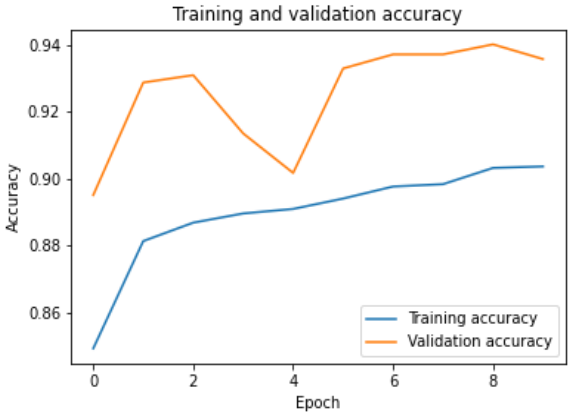
Training Accuracy = 90.84%
Validation accuracy = 93.84%

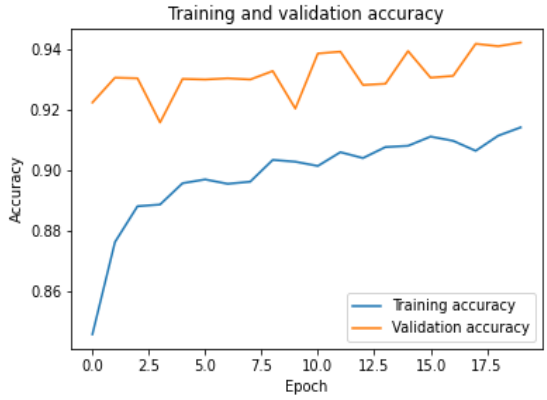


Training Loss = 0.21856
Validation Loss = 0.15602

We observe that the performance of the model is almost the same for both 10 and 20 epochs, indicating that not much training is needed. This is because we are using pre-trained weights from the VGG-16 model. On analyzing the training and validation loss graphs shown above for both 10

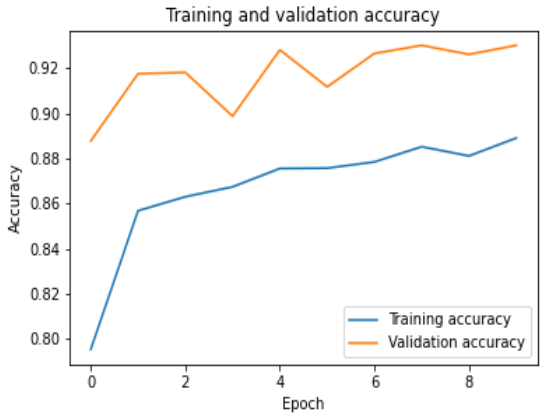
and 20 epochs, we observe that the training loss is converging very slowly. This indicates that the learning rate is small. Although the performance of the model in terms of accuracy is around 93.8% which implies that the model is performing well with these parameters, but we also try to increase the learning rate to 0.001 and train the model while using Adam optimizer for both 10 and 20 epochs. The following plots are obtained after training with these parameters:

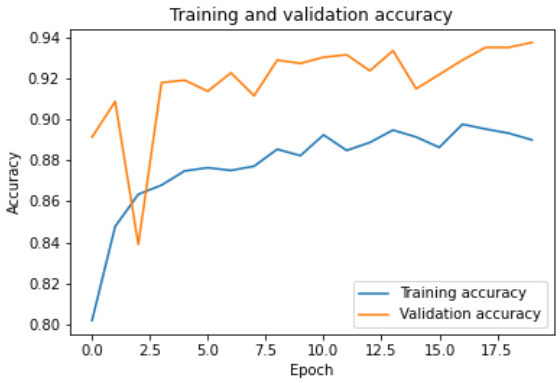
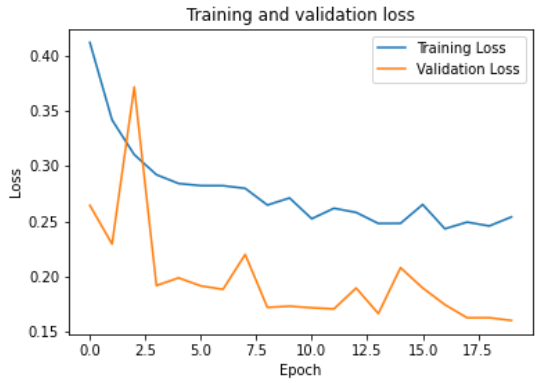
Optimizer: Adam Learning Rate: 0.001 No. of Epochs: 10	
 Training Accuracy = 90.76% Validation accuracy = 93.58%	 Training Loss = 0.22255 Validation Loss = 0.16911

Optimizer: Adam Learning Rate: 0.001 No. of Epochs: 20	
 Training Accuracy = 91.68% Validation accuracy = 94.24%	 Training Loss = 0.19662 Validation Loss = 0.14352

On increasing the learning rate from 0.00001 to 0.01 when using Adam optimizer, we see an improvement in model performance.

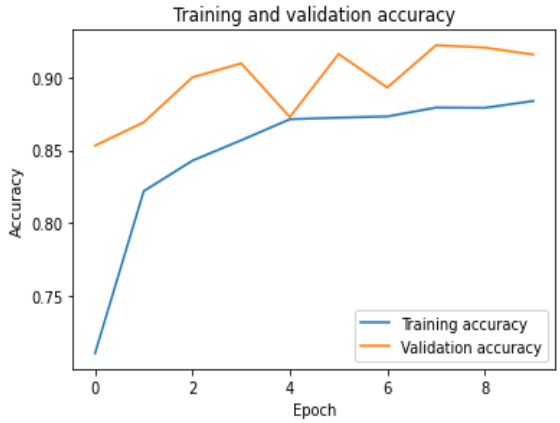
2. We now analyze the model performance by using the Stochastic Gradient Descent (SGD) optimizer with a learning rate of 0.001 and momentum of 0.9 for 10 and 20 epochs. The following plots are obtained after training.

Optimizer: SGD Learning Rate: 0.001 No. of Epochs: 10	
 Training Accuracy = 89.78% Validation accuracy = 93.02%	 Training Loss = 0.23872 Validation Loss = 0.17268

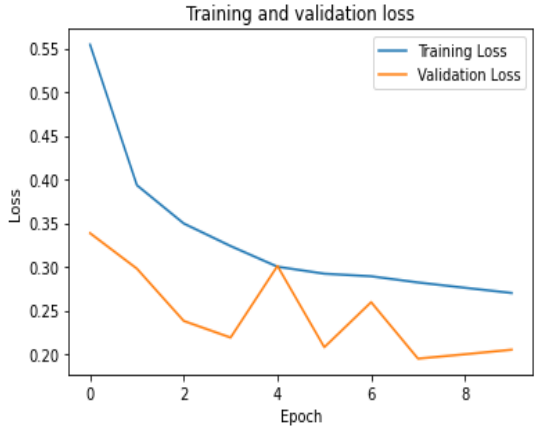
Optimizer: SGD Learning Rate: 0.001 No. of Epochs: 20	
 Training Accuracy = 90.15% Validation accuracy = 93.76%	 Training Loss = 0.23064 Validation Loss = 0.16012

We observe that the validation accuracy is almost the same for both 10 and 20 epochs, indicating not much training is needed because of the pretrained model being used. On analyzing the training and validation loss graphs shown above for both 10 and 20 epochs, we observe that the training loss is converging very slowly implying that the learning rate is small. Hence, we try to increase the learning rate to 0.01 and train the model using SGD optimizer for both 10 and 20 epochs. The following plots are obtained after training with these parameters:

Optimizer: SGD
Learning Rate: 0.01
No. of Epochs: 10

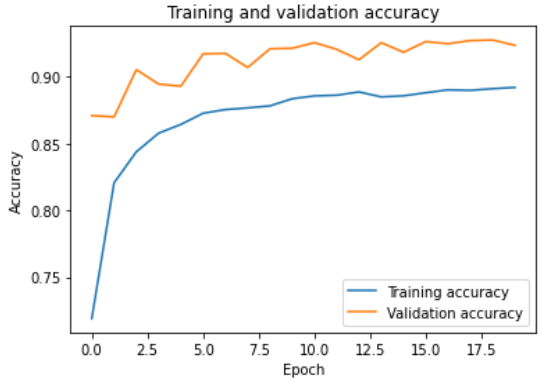


Training Accuracy = 88.52%
Validation accuracy = 91.56%

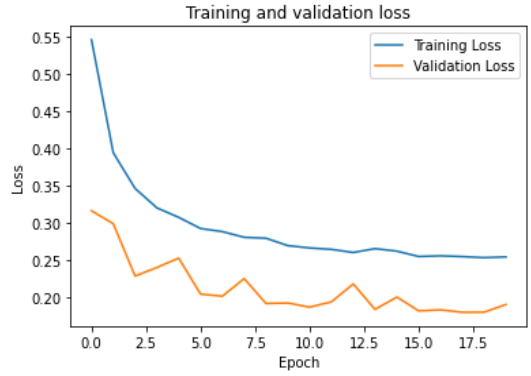


Training Loss = 0.26899
Validation Loss = 0.20501

Optimizer: SGD
Learning Rate: 0.01
No. of Epochs: 20



Training Accuracy = 89.17%
Validation accuracy = 92.36%

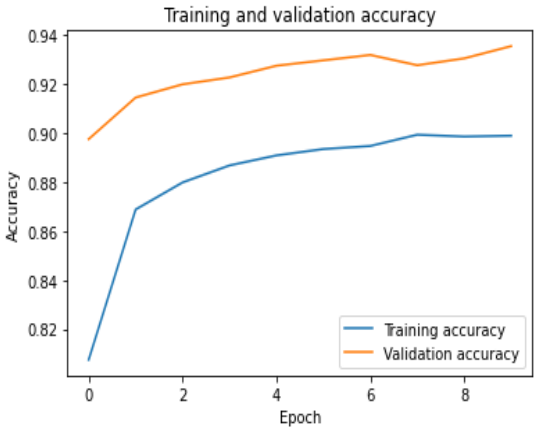


Training Loss = 0.25394
Validation Loss = 0.19073

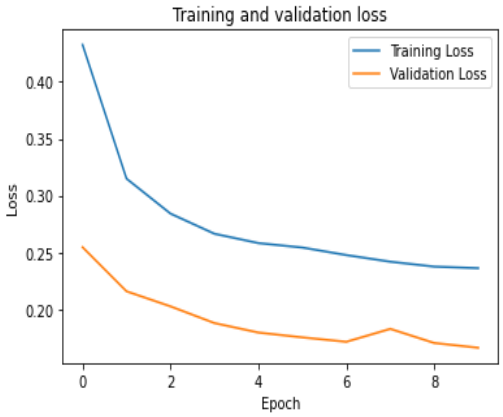
On increasing the learning rate from 0.001 to 0.01 when using SGD optimizer, we do not see an improvement in model performance, so in this case the learning rate of 0.001 performs better. By analyzing the loss curves for different sets of parameters, we observe that there is almost no indication of overfitting which implies that the model is robust and generalizable.

3. We now analyze the model performance by using the RMSProp optimizer with a learning rate of 0.00001 for 10 and 20 epochs. The following plots are obtained after training.

Optimizer: RMSProp
Learning Rate: 0.00001
No. of Epochs: 10

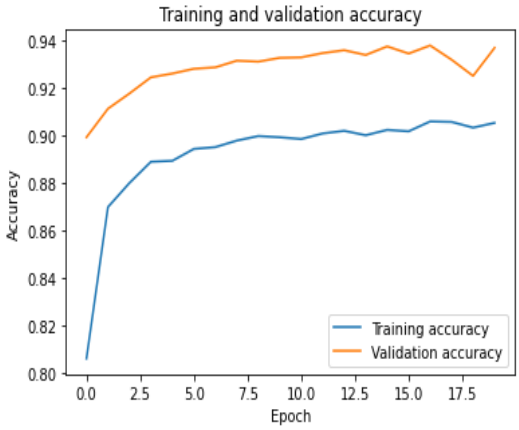


Training Accuracy = 89.89%
Validation accuracy = 93.56%

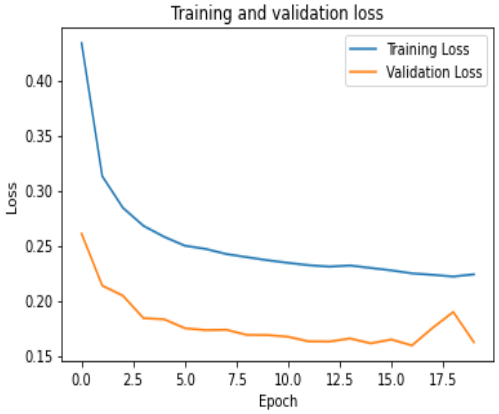


Training Loss = 0.23519
Validation Loss = 0.16612

Optimizer: RMSProp
Learning Rate: 0.00001
No. of Epochs: 20



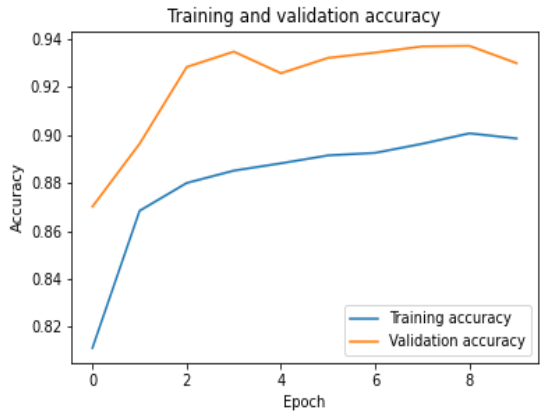
Training Accuracy = 90.87%
Validation accuracy = 93.7%



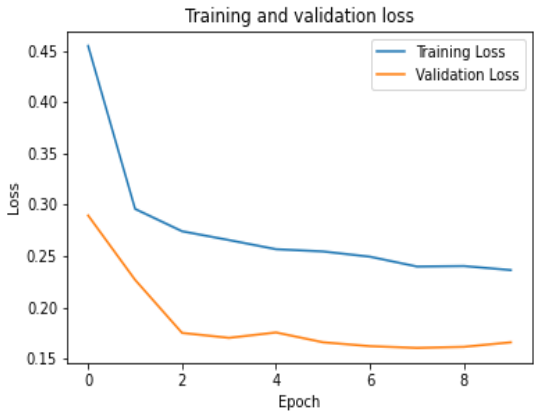
Training Loss = 0.21994
Validation Loss = 0.16213

We again observe that there is no significant improvement in training for 20 epochs compared to 10 epochs. On analyzing the training loss curves, it is evident that the loss is converging slowly which indicates that the learning rate is low. Hence, we try to increase the learning rate to 0.001 and train the model while using RMSProp optimizer for both 10 and 20 epochs. The following plots are obtained after training with these parameters:

Optimizer: RMSProp
Learning Rate: 0.001
No. of Epochs: 10

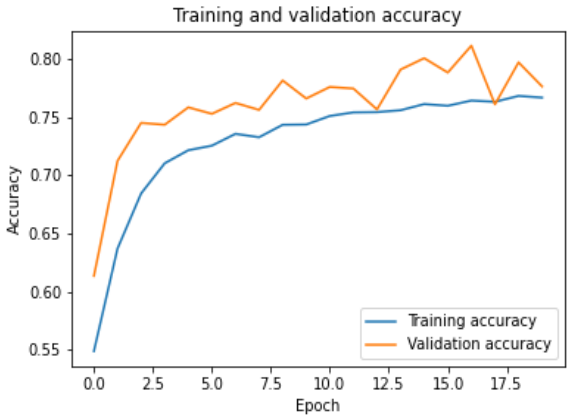


Training Accuracy = 90.48%
Validation accuracy = 93%

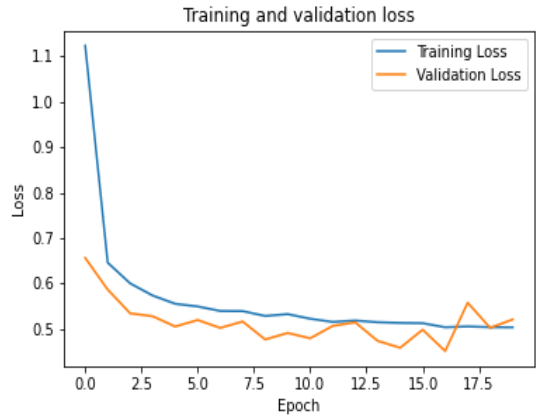


Training Loss = 0.22948
Validation Loss = 0.16585

Optimizer: RMSProp
Learning Rate: 0.001
No. of Epochs: 20



Training Accuracy = 91.18%
Validation accuracy = 93.33%



Training Loss = 0.21782
Validation Loss = 0.17103

On increasing the learning rate from 0.00001 to 0.001 when using RMSProp optimizer, we see a slight improvement in model performance

After training the model using pretrained weights from VGG-16 with different sets of parameters, we obtain the best performance when we use Adam optimizer with a learning rate of 0.01 for 20 epochs. With these parameters we obtain a validation accuracy of 94.24 % for the model.

The different sets of parameters used to train the VGG-16 model and their corresponding training and validation loss and accuracy have been summarized in the table below:

Optimizer	Learning Rate	Training Loss	Training Accuracy (%)	Validation Loss	Validation Accuracy (%)
Adam	0.001	0.22255	90.76	0.16911	93.58
SGD	0.001	0.23872	89.78	0.17268	93.02
RMSProp	0.001	0.22948	90.48	0.16585	93
Adam	0.00001	0.23716	89.73	0.16319	93.5
SGD	0.01	0.26899	88.52	0.20501	91.56
RMSProp	0.00001	0.23519	89.89	0.16612	93.56
Adam	0.001	0.19662	91.68	0.14352	94.24
SGD	0.001	0.23064	90.15	0.16012	93.76
RMSProp	0.001	0.21782	91.18	0.17103	93.33
Adam	0.00001	0.21856	90.84	0.15602	93.84
SGD	0.01	0.25394	89.17	0.19073	92.36
RMSProp	0.00001	0.21994	90.87	0.16213	93.7

Table 4: Training results of transfer learning (VGG-16) model with different parameters

On comparing the performance of the two models, both models give their best results when using the Adam optimizer with a learning rate of 0.001 and when trained for 20 epochs. However, the model with transfer learning performs better than the CNN model. This is because in the case of transfer learning we are using pretrained weights and the network is more complex.

The corresponding code has been included in the attached .ipynb notebook.

Question (d): Report the classification accuracy on validation set. Apply the classifier(s) built to the test set. Submit the "submission.csv" with the results you obtained.

Answer:

The highest classification accuracy obtained on the validation set is 94.24%. It is obtained on using the model with transfer learning (VGG-16). The model is optimized using the Adam optimizer with a learning rate of 0.001 and is trained for 20 epochs. A batch size of 32 was used.

The corresponding classifier was applied to the test set and the classification results of the testing data are saved to the 'submission.csv' file which includes two columns, i.e., ID of the test images and the corresponding predicted label (1=dog, 0=cat). The csv file has been included in the zip file submission in NTULearn. A count plot was made to visualize the number of predicted dog and cat images in the test dataset and is shown in the figure below:

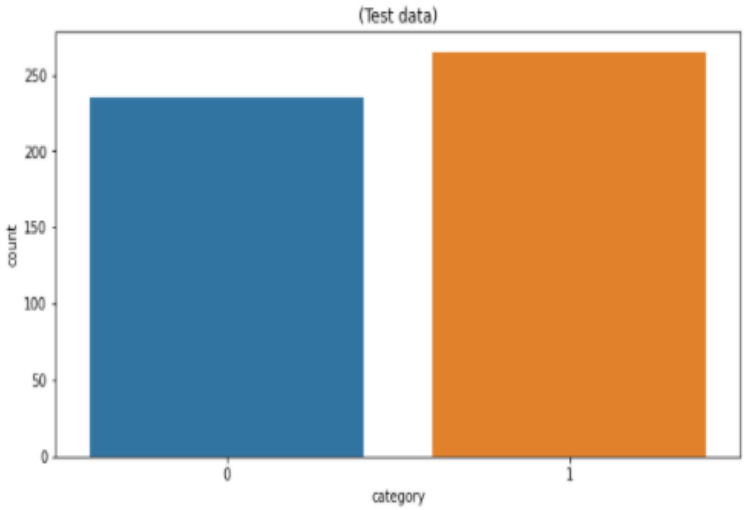


Figure 19: Count plot for the testing set predictions

We also visualize some of the testing prediction results which are shown below. The predicted label is enclosed in curly braces beside the testing image filename.

Figure 20: Visualization of predictions on testing set

From Fig 20. we observe that out of 15 images only 1 image has been misclassified. The complete code for this has been included in the .ipynb notebook attached.

Question (e): Analyze some correctly and incorrectly classified (if any) samples in the test set. Select 1-2 cases to discuss the strength and weakness of the model.

Answer:
We use the model to predict the labels of the images in the validation set. The predicted labels are compared with the ground truth labels and we obtain examples of some correctly and incorrectly classified samples in the validation set. The corresponding code is included in the attached notebook.

(i) Some examples of correctly classified samples which are used to portray the strengths of the model are as follows:

Figure (a)

Figure (b)

Figure (c)

Figure(d)

Figures (a) – (d) are examples of correctly classified samples from the validation set.

In Figure (a), the model can currently identify the dog in the image despite the presence of the cage which reduces the visibility of the dog making the classification challenging. However, the model is able to correctly classify the image which shows that the model is robust and it has learned by seeing similar images in the training set as well.

In Figures(b) and (c) we see pictures of multiple dogs and cats respectively and the classifier was able to make the correct predictions despite the number of cats or dogs in the image. Hence, another strength of the model is to make correct predictions irrespective of the number of objects of interest present in the image.

In figure (d) we observe that the cat is printed on a cloth and is not a real cat. However, the model was able to make the correct prediction. This also signifies the generalizability and robustness of the model, contributing to its strengths.

(ii) Some examples of misclassified samples which are used to portray the weaknesses of the model are as follows:

Figure (e)

Figure (f)

Figure (g)

Figure(h)

Figures (e) – (h) are examples of incorrectly classified samples from the validation set

In figure (e), we observe that the image is that of a cat held by a human being, however the model misclassified this image as a dog. This misclassification can be attributed to the fact that the face of the cat has a dark spot in the middle which makes it similar to the face of dogs. Moreover, the proportion of the cat visible in the image is less because of the presence of a human being. This could be the possible reason for misclassification in this case.

In figure (f), we observe that the image is a picture of a dog which has been misclassified as a cat. This misclassification can be attributed to the fact that the forehead of the dog if orangish white in color making it very similar to that of cats. Moreover, the posture in which the dog is sitting is very similar to that of cats. This could be the possible reason for this misclassification.

In figure (g), we observe that a dog has been misclassified as a cat. The dog in this image indeed looks very similar to cat. Its features are not distinctive enough and match more to the features of a cat. This could be a possible reason why this image was misclassified and the model was not able to recognize that in reality the image contains a dog.

In figures (h) we observe that the dog has been misclassified as a cat. This image is very noisy and unclear which could possibly be the reason which led to this misclassification. This indicates that there are instances where the model might make a misclassification in the presence of excess noise in the image.

We also observe some sample cases of correct and incorrect classifications from the test set and the examples are as follows:

(i) Some examples of correctly classified samples in the test set are as follows:

Figure (a)

Figure (b)

Figure (c)

In Figure (a), the image is very zoomed out, which reduces the proportion of the dog visible in the image. However, the model was able to correctly identify the dog. This can be attributed to the zooming in and out data augmentation used before training, which makes the model robust against these situations.

In figure (b), only the face of the dog is visible, portions of which are also covered by the human hand. However, the model was able to make the correct prediction by only looking at the face of the dog.

In figure (c), we observe that there are multiple cats behind a cage and the model makes the correct prediction. Hence, the model is robust against the number of objects of interest.

(ii) Some examples of incorrectly identified samples from the test set are as follows:

Figure (d)

Figure (e)

In figure (d), the dog in the image has been incorrectly classified as a cat. This can be because the face of the dog is not completely visible and the forehead has a combination of colors which are very similar to those of cats. This became a challenging situation for the model leading to a misclassification.

In figure (e), the image in reality contains a dog, however, the model misclassified it as a cat. One of the reasons for this could be because the image has a little noise and the features of the dog are not very distinctive and are similar to those of cats. Hence, the model misclassified the image as a cat.

Question (f): Discuss how different choice of models and data processing may affect the project in terms of accuracy on validation set.

Answer:

In order to analyze the effect of different choices of models and data preprocessing in terms of accuracy on validation dataset we compare the performance of the following models:

1) CNN Baseline model without data augmentation and regularization
2) CNN model with data augmentation and regularization
3) Transfer Learning using VGG-16

The above models are optimized using the Adam optimizer with a learning rate of 0.001 and the models are trained for 20 epochs. Batch size of 32 is used. The training results of these models are as follows:

Metric	CNN Baseline Model	CNN Model with Preprocessing and Regularization	Transfer Learning using VGG-16
Training Loss	0.01284	0.3798	0.19662
Training Accuracy (%)	99.67	83.25	91.68
Validation Loss	2.04812	0.35387	0.14352
Validation Accuracy (%)	76.62	84.5	94.24

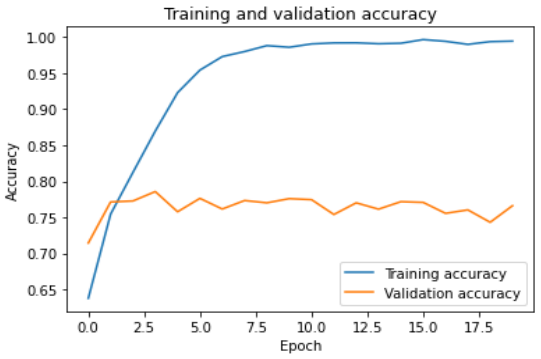
Table 5: Comparison of different models

With the CNN Baseline model, which does not include data augmentation or regularization, we obtain a validation accuracy of 76.62% but a very high training accuracy of 99.67%. This indicates that the model performs well on the training set but is not generalizable which leads to a poor performance on the validation dataset. Hence, this indicates that the model is overfitting to the noise in the training set. In order to prevent overfitting, we apply data augmentation and add dropout regularization to the CNN model. After training the CNN model with these modifications, we get a validation accuracy of 84.5% and training accuracy of 83.25%. There is no more overfitting of the model and the validation accuracy improves from 76.62% to 84.5%. In order to further improve the accuracy, we use VGG-16 pretrained feature extraction backbone and implement the model using transfer learning. The validation accuracy improves to 94.24%.

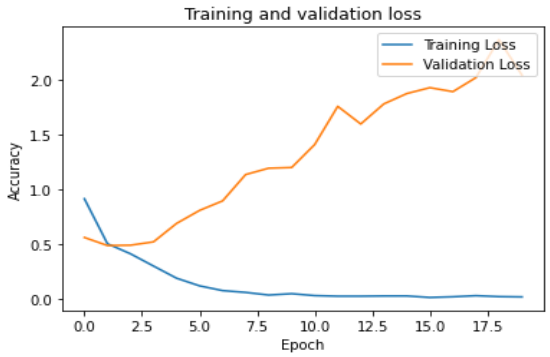
In terms of accuracy on the validation set, we obtain the best performance when transfer learning is used. This is because we utilize a pretrained feature extraction backbone of VGG-16 and the number of trainable parameters is less. The model has a low bias and a low variance and is suitable to give optimal results with an accuracy of 94.24% on the validation set.

In order to analyze the effect of data preprocessing on the accuracy of validation dataset we compare the performance of the CNN model with and without data augmentation and regularization.

We train the baseline CNN model which does not include data preprocessing and regularization. In order to visualize the model performance, we plot the training and validation accuracy and loss curves which are shown below:



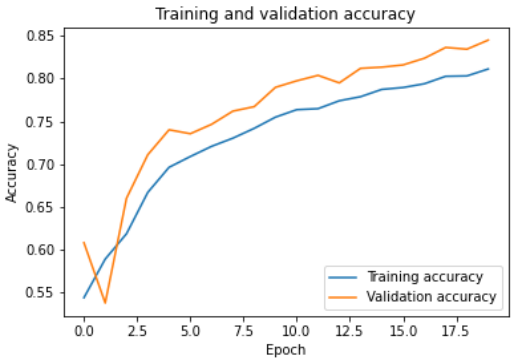
Training Accuracy = 99.67%
Validation Accuracy = 76.62%



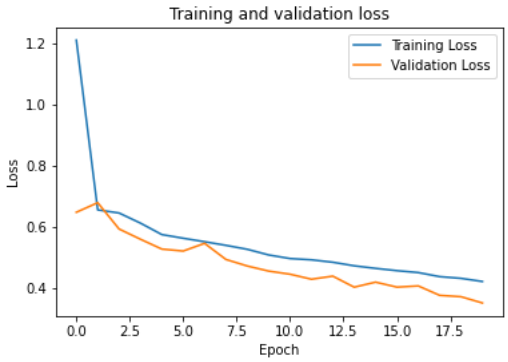
Training Loss = 0.01284
Validation Loss = 2.04812

On analyzing the training and validation loss curves, we observe that the baseline model shows strong overfitting which can be seen as the large gap between the training and validation errors. We observe that the training error keeps decreasing while the validation error starts increasing. This indicates that the model is fitting to the noise in the training set and is not generalizable, because of which it is performing poorly on the validation set. This is the reason why overfitting triggers a large gap between training and validation errors. Further, due to overfitting, the model performs extremely well on the training set giving a high training accuracy of 99.6% but performs poorly on the validation set, giving a validation accuracy of 76.62%.

In order to reduce overfitting, we explore two approaches: (1) data augmentation and (2) dropout regularization. We perform data preprocessing using the techniques outlined in answer (a) and add regularization. After training the CNN model, we plot the training and validation accuracy and loss curves and obtain the following results:



Training Accuracy = 83.25%
Validation accuracy = 84.5%



Training Loss = 0.3798
Validation Loss = 0.35387

On analyzing the training and validation loss curves, we observe that after data preprocessing and addition of regularization we are able to prevent overfitting which results in an improvement in the accuracy on the validation set.

Question (g): Apply and improve your classification algorithm to a multi-category image classification problem for Cifar-10 dataset (https://www.cs.toronto.edu/~kriz/cifar.html). Describe details about the dataset, classification problem that you are solving, and how your algorithm can tackle this problem. Report your results for the testing set of Cifar-10

Answer:

Dataset:

The CIFAR-10 dataset was developed by researchers at the CIFAR institute [5]. The dataset comprises of 60,000 color images of 32x32 pixels in 3 channels of objects from 10 classes. The 10 class labels and their associate integer values are:

0: airplane

1: automobile

2: bird

3: cat

4: deer

5: dog

6: frog

7: horse

8: ship

9: truck

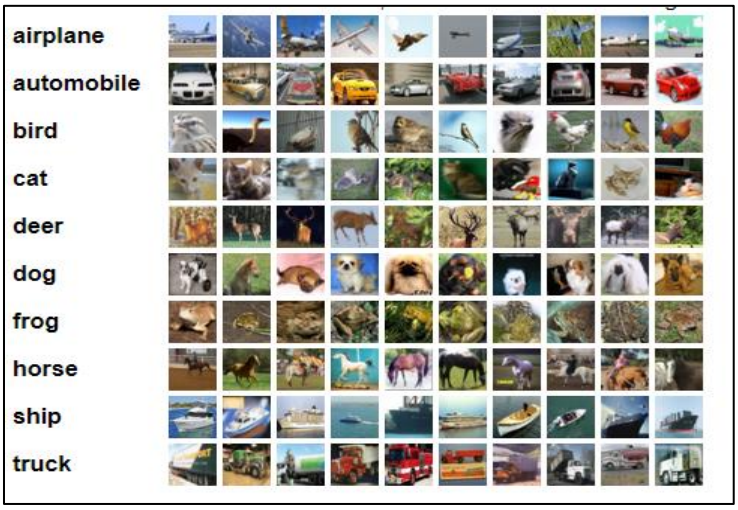


Figure 21: CIFAR-10 dataset samples

Figure 21. can be used to comprehend the dataset better. Each class contains 6,000 images. The training set contains 50,000 images while the test set contains 10,000 images.

Here, we are solving a multi-class classification problem.

In order to perform data exploration, we visualize some images from the training set and are shown below:

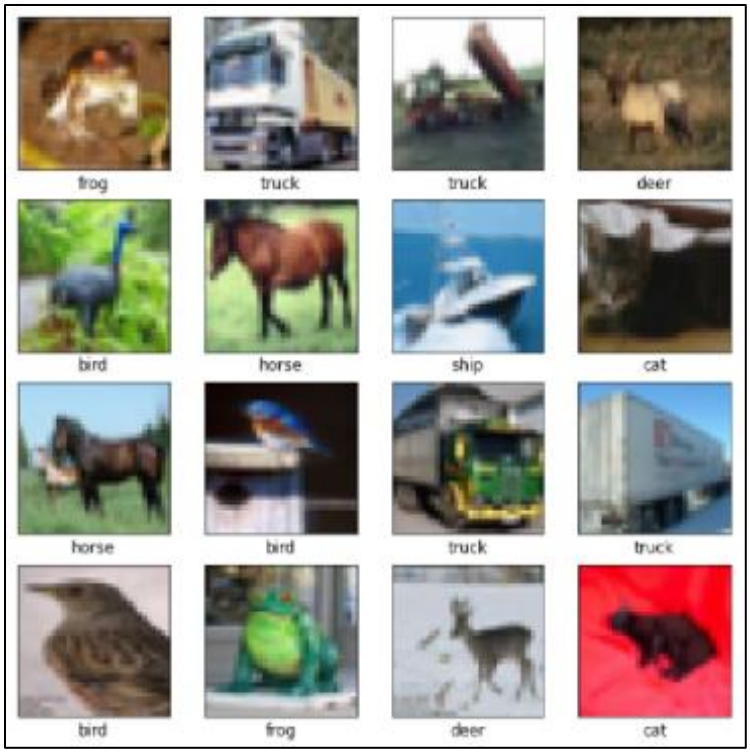


Figure 22: Some examples of visualized samples from CIFAR-10 training set

It can be seen from the figure above that these images are very small compared to a typical photograph.

Data preprocessing is performed by normalizing the pixel values to be in the range of 0 to 1. The data augmentation techniques applied to the training data include:

- width_shift_range = 0.1
- height_shift_range=0.1
- horizontal_flip=True

These techniques have been explained in details in answer (a).

Algorithm

The model used to perform multi-label classification using CIFAR-10 dataset, is a CNN model developed by improving the CNN model which was used earlier to classify dog and cat images as described in answer (b).

We modify the existing CNN model by repeating each convolutional layer in each block once. This part forms the feature detector of the model. The classifier part interprets these features and outputs a predicted class label. The feature maps output from the feature extraction part are flattened to form a single vector which is passed through a fully-connected/dense layer with 128 units. This is followed by the output layer which now has 10 nodes corresponding to the 10 classes being predicted and SoftMax activation function is used in the output layer. ReLU activation function and He uniform weight initialization is used in all the other layers.

The model is optimized using Adam optimizer with a learning rate of 0.001. The loss function is changed from binary cross-entropy loss to categorical cross-entropy loss function as we are handling a multi-class classification problem. The metrics used is accuracy. Batch size of 64 is used.

The screenshot of the code used to build the model is shown in Figure 23.

```
def define_model():
    model = Sequential()
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same', input_shape=(32, 32, 3)))
    model.add(Conv2D(32, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(Conv2D(128, (3, 3), activation='relu', kernel_initializer='he_uniform', padding='same'))
    model.add(MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(Dropout(0.2))
    model.add(Dense(10, activation='softmax'))
    # compile model
    opt = 'adam'
    model.compile(optimizer=opt, loss='categorical_crossentropy', metrics=['accuracy'])
    return model
```

Figure 23: Screenshot of the code to build CNN model for multi-class classification

The structure of the CNN model used for cats and dogs dataset and the modified CNN model used to perform multi-class classification for the CIFAR-10 dataset are shown in Table 6. The summary of the model is shown in Figure 24. which gives an overview of the different layers and outlines the number of parameters in each layer.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
conv2d_1 (Conv2D)	(None, 32, 32, 32)	9248
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
dropout (Dropout)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18496
conv2d_3 (Conv2D)	(None, 16, 16, 64)	36928
max_pooling2d_1 (MaxPooling2	(None, 8, 8, 64)	0
dropout_1 (Dropout)	(None, 8, 8, 64)	0
conv2d_4 (Conv2D)	(None, 8, 8, 128)	73856
conv2d_5 (Conv2D)	(None, 8, 8, 128)	147584
max_pooling2d_2 (MaxPooling2	(None, 4, 4, 128)	0
dropout_2 (Dropout)	(None, 4, 4, 128)	0
flatten (Flatten)	(None, 2048)	0
dense (Dense)	(None, 128)	262272
dropout_3 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1290

Total params: 550,570
Trainable params: 550,570
Non-trainable params: 0

Figure 24: Summary of CNN model in multi-class classification

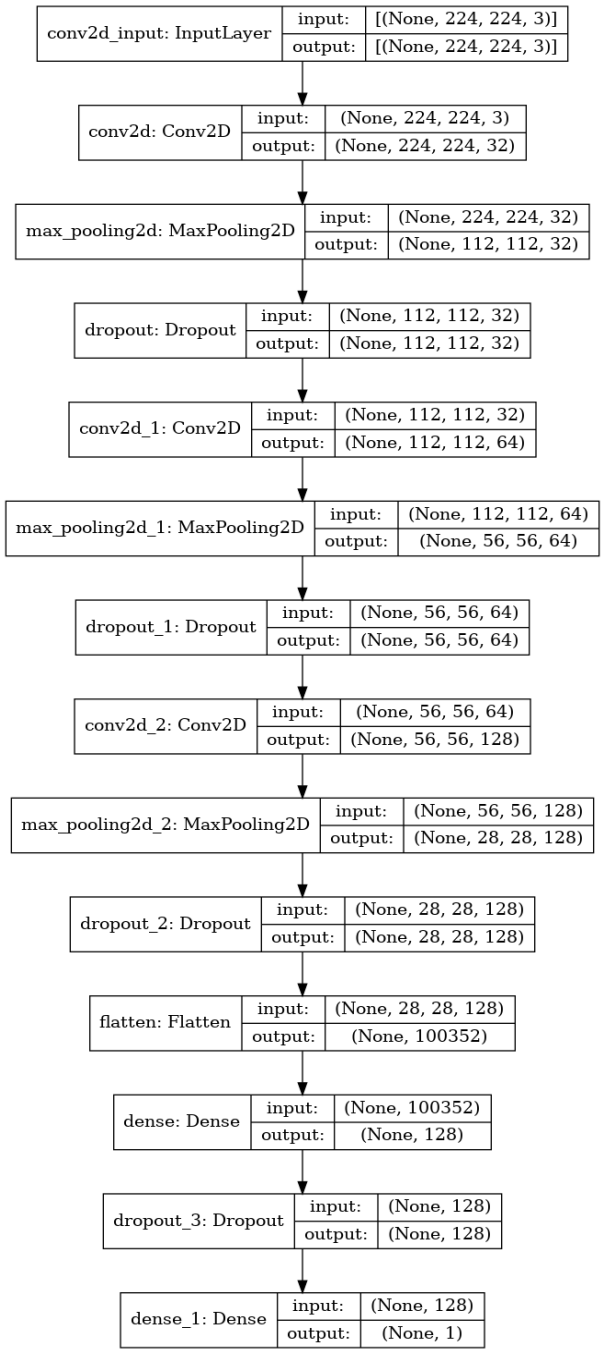
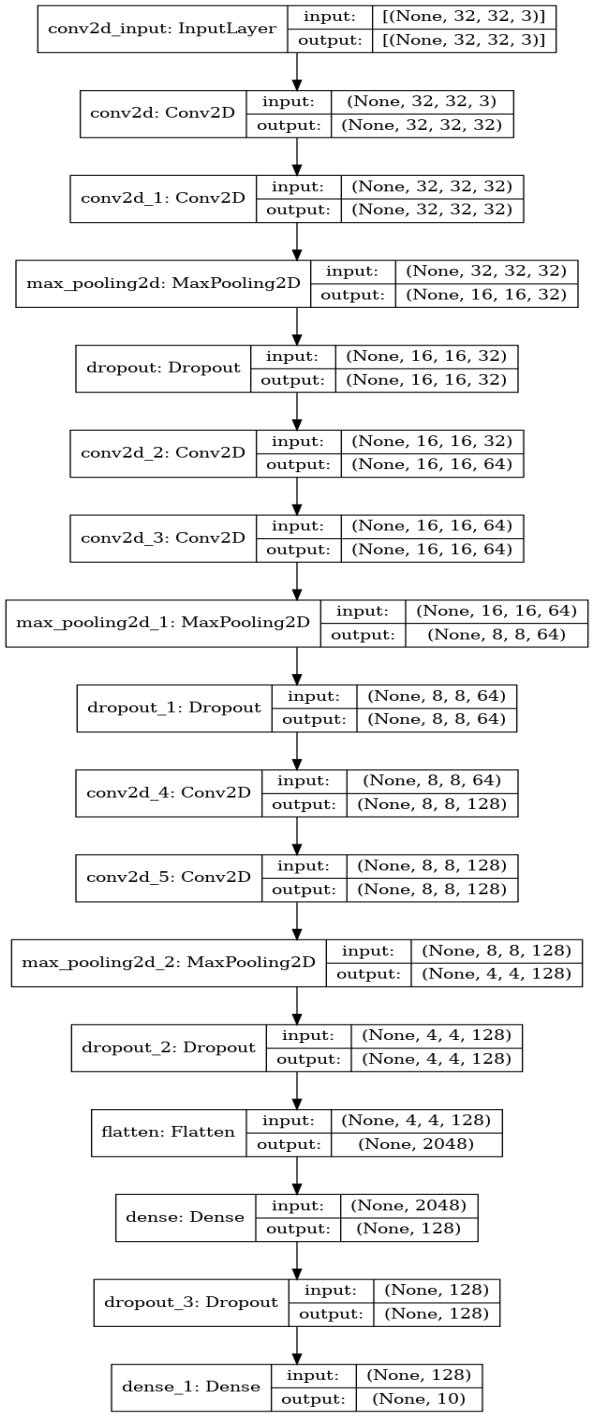
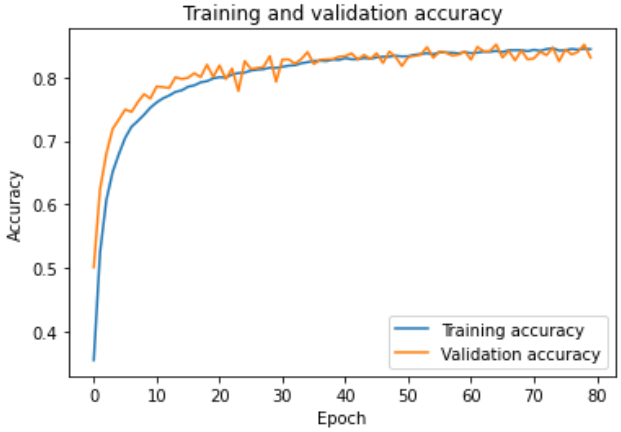
CNN model structure used for binary classification (cats and dogs)	CNN model structure used for multi-class classification (CIFAR-10)
	

Table 6: Structure of CNN models for binary and multi-class classification

Training and Evaluation

The model was trained for 80 epochs and the corresponding training and validation loss and accuracy curves were plotted as shown below:



After training the model we obtain a training loss = 0.34629 and training accuracy = 88.12%

On evaluating the model, we obtain a test loss = 0.5618 and the testing accuracy obtained on the CIFAR-10 dataset is 83.12%.

```
313/313 [==============================] - 1s 3ms/step - loss: 0.5618 - accuracy: 0.8311
1563/1563 [==============================] - 5s 3ms/step - loss: 0.3463 - accuracy: 0.8812
Test loss: 0.5617572665214539
Test accuracy: 0.8310999870300293
Training loss: 0.3462858200073242
Training accuracy: 0.8812000155448914
```

The complete code has been included in the attached .ipynb notebook.

We also plot the heat map and the corresponding classification report for a better understanding and analysis of model performance [6]. The classification report is shown in Figure 25.

	precision	recall	f1-score	support
0	0.83	0.86	0.84	1000
1	0.90	0.96	0.93	1000
2	0.87	0.69	0.77	1000
3	0.72	0.70	0.71	1000
4	0.84	0.77	0.81	1000
5	0.83	0.71	0.77	1000
6	0.68	0.95	0.79	1000
7	0.90	0.86	0.88	1000
8	0.92	0.90	0.91	1000
9	0.90	0.91	0.90	1000
accuracy			0.83	10000
macro avg	0.84	0.83	0.83	10000
weighted avg	0.84	0.83	0.83	10000

Figure 25: Classification Report

The heat map is shown in Figure 26.

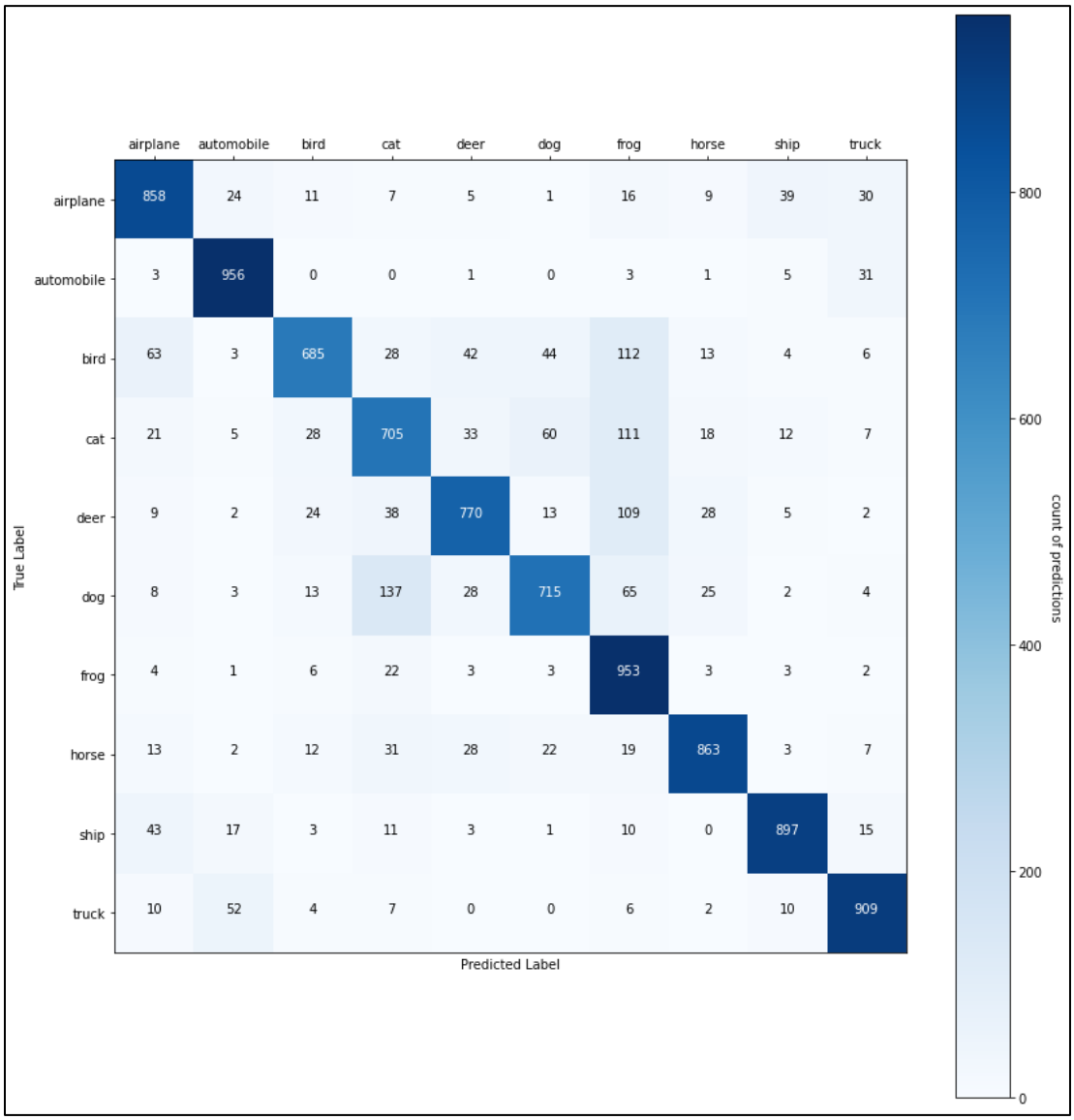


Figure 26: Confusion Matrix

On analyzing the heat map shown above, we observe that the model makes the best predictions for the class ‘automobile’. However, its weakness lies in identifying dogs which are misclassified as cats.

Some incorrect predictions of the model are visualized to have a better understanding of the model’s strengths and weaknesses. Some incorrect predictions are shown in Figure 27.

Figure 27: Misclassifications

From Fig. 27, the picture in the 1st column, 2nd row is a bird which has been misclassified as a ship. This can be possibly attributed to the fact that the image is zoomed into the sharp beak of the bird which also looks similar to the edge of a ship from the side view. These are some examples of misclassifications.

Some general prediction visualizations were also made and the corresponding code has been included in the attached notebook.

References

[1] Basic CNN architecture: Explaining 5 layers of convolutional neural network," *Upgrad.com*, 07-Dec-2020. [Online]. Available: https://www.upgrad.com/blog/basic-cnn-architecture/. [Accessed: 10-Nov-2021].

[2] uysimty, "Keras CNN dog or cat classification," *Kaggle.com*, 16-Jun-2019. [Online]. Available: https://www.kaggle.com/uysimty/keras-cnn-dog-or-cat-classification. [Accessed: 10-Nov-2021].

[3] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv [cs.CV]*, 2014.

[4] J. Brownlee, "Gentle introduction to the adam optimization algorithm for deep learning," *Machine Learning Mastery*, 12-Jan-2021. [Online]. Available: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/. [Accessed: 10-Nov-2021].

[5] CIFAR-10 and CIFAR-100 datasets. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html. [Accessed: 10-Nov-2021].

[6] roblexnana, "cifar10 with CNN for beginer," *Kaggle.com*, 25-May-2020. [Online]. Available: https://www.kaggle.com/roblexnana/cifar10-with-cnn-for-beginer. [Accessed: 12-Nov-2021].

Appendix

Structure of the model using VGG-16 pretrained feature extraction backbone