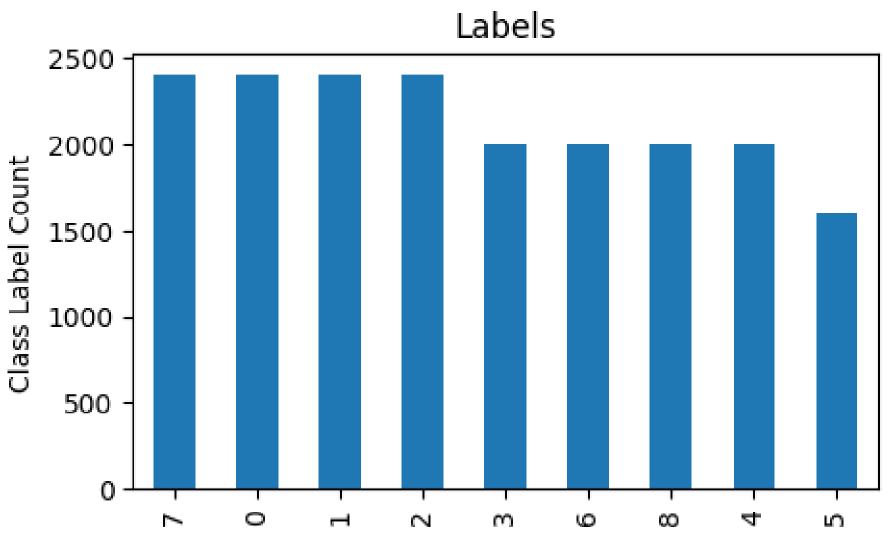
**Name: Melwyn D Souza**
**Student Number: R00209495**
**Date: 1/May/2022**
**Module: Deep Learning**
**Assignment: 2**
**Lecturer: Dr Ted Scully**
**Course: MSc in Artificial Intelligence**

## Dataset

Satellite images of land covers captured by Sentinel-2-Sattellite. The images are RGB 3 channels with varying dimensions. The dimensions are converted to 64x64x3 from the code supplied with the assignment description. There are 19200 images as training data and 4800 images for validation
There is a slight imbalance in the dataset as shown in the figure below which we is not rebalanced in this assignment



## Part A

### Part A (i)

### Data Augmentation

The model architectures used for evaluating the performance of the models on earth data classification is in the table1 below,

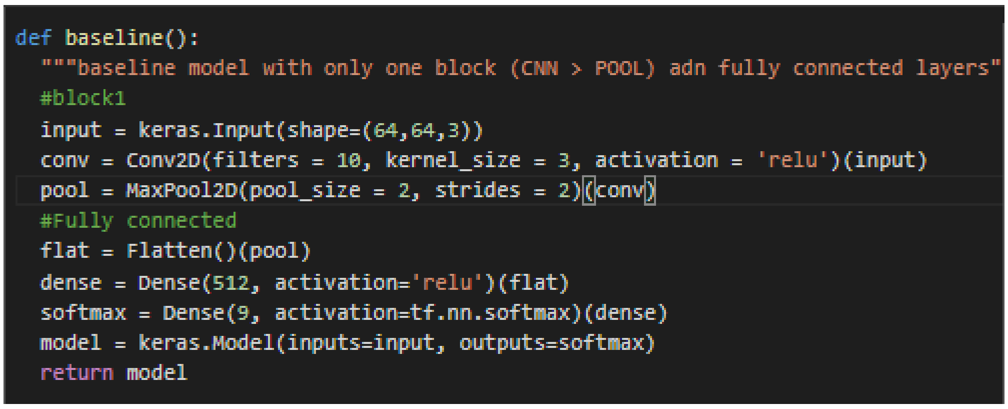
- C: Convolution layer
- P: Pooling layer
- F: Flatten
- D: Dense
- S: SoftMax

| Model Name | CONV Blocks | Architecture | Train Acc | Validation Acc |
|---|---|---|---|---|
| Baseline model | 1 | CPFDS | 0.92 | 0.72 |
| Model 1 | 2 | CPCPFDS | 0.95 | 0.76 |
| Model 2 | 3 | CPCPCPFDS | 0.96 | 0.78 |
| Model 3 | 4 | CPCPCPCPFDS | 0.93 | 0.85 |
| Model 3 - Data Augmentation | 4 | CPCPCPCPFDS | 0.9 | 0.89 |

Table 1: Part 1 model description

From the table the deeper we go, the better the accuracies, but there is overfitting in all models except for when data generator is used to augment data which reduced overfitting

The code snippet is as shown below for baseline model

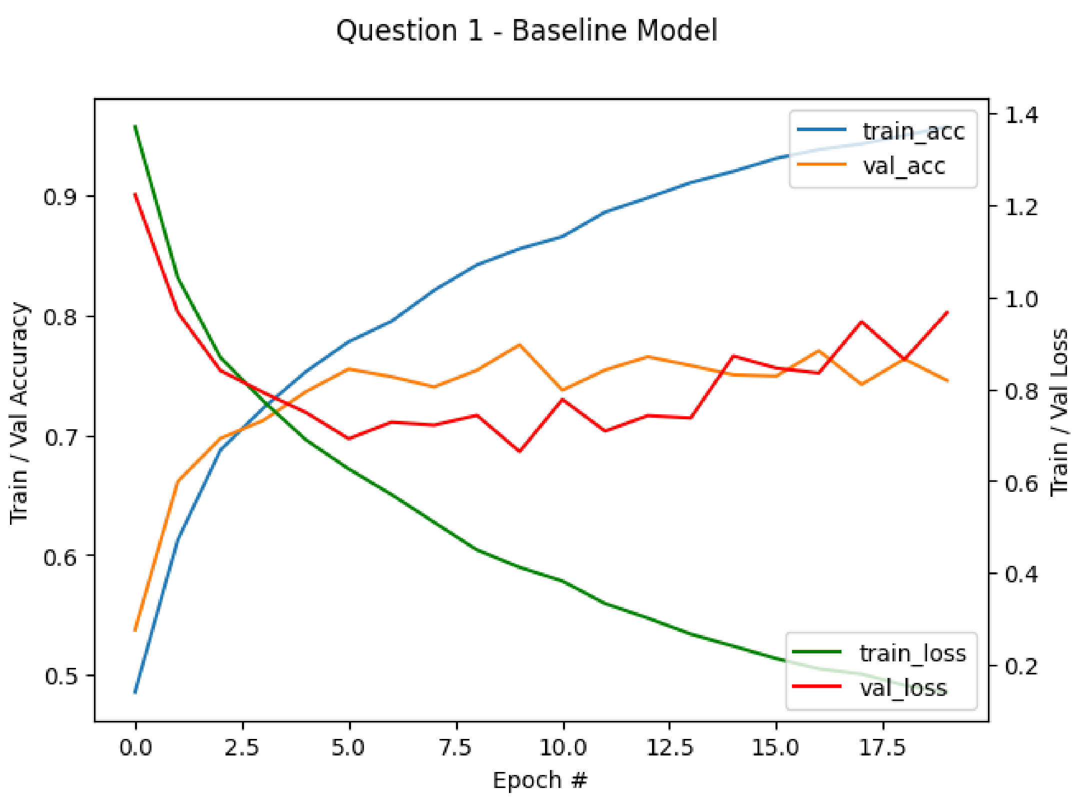


```
def baseline():
  """baseline model with only one block (CNN > POOL) adn fully connected layers"
  #block1
  input = keras.Input(shape=(64,64,3))
  conv = Conv2D(filters = 10, kernel_size = 3, activation = 'relu')(input)
  pool = MaxPool2D(pool_size = 2, strides = 2)(conv)
  #Fully connected
  flat = Flatten()(pool)
  dense = Dense(512, activation='relu')(flat)
  softmax = Dense(9, activation=tf.nn.softmax)(dense)
  model = keras.Model(inputs=input, outputs=softmax)
  return model
```

Input is of shape 64x64x3 which is the layer 0 input dimensions, this is then connected to convolution layer with 10 filters of size 3 and ReLu activations, the default stride 1 is used.

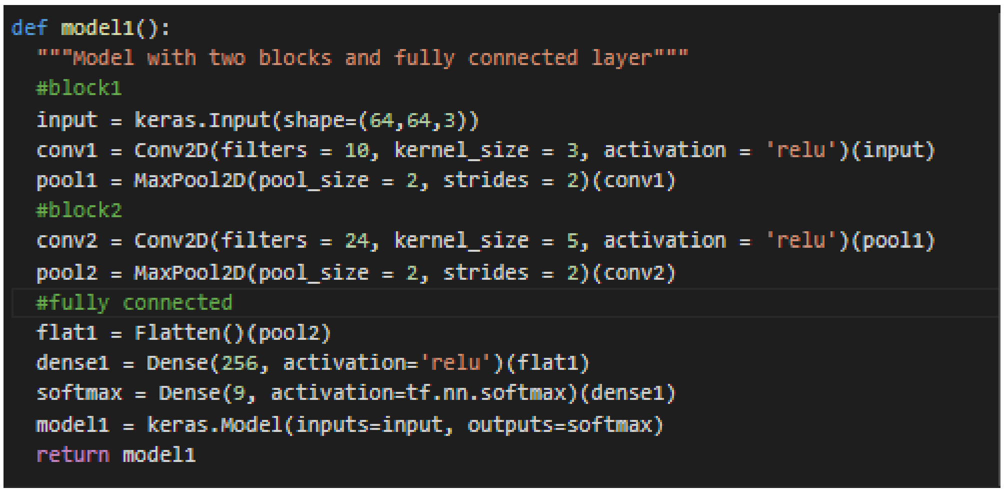
The convolved output is reduced in dimensions using pooling layer with pool size 2 and stride 2. The output from the pooling layer is then flattened using TensorFlow objects, flattened vector is then passed through SoftMax layer with 9 neurons since there are 9 earth land cover classes to be classified, the SoftMax layer will give predicted probability of each class, the maximum probability out of 9 neuron outputs is the predicted class.

This model is trained using 19200 training data images which gives accuracy of 92% but the validation accuracy is 72%, also the validation loss increases s on every epoch which hints towards poor performance of the model, the learning curve for baseline model is as follows



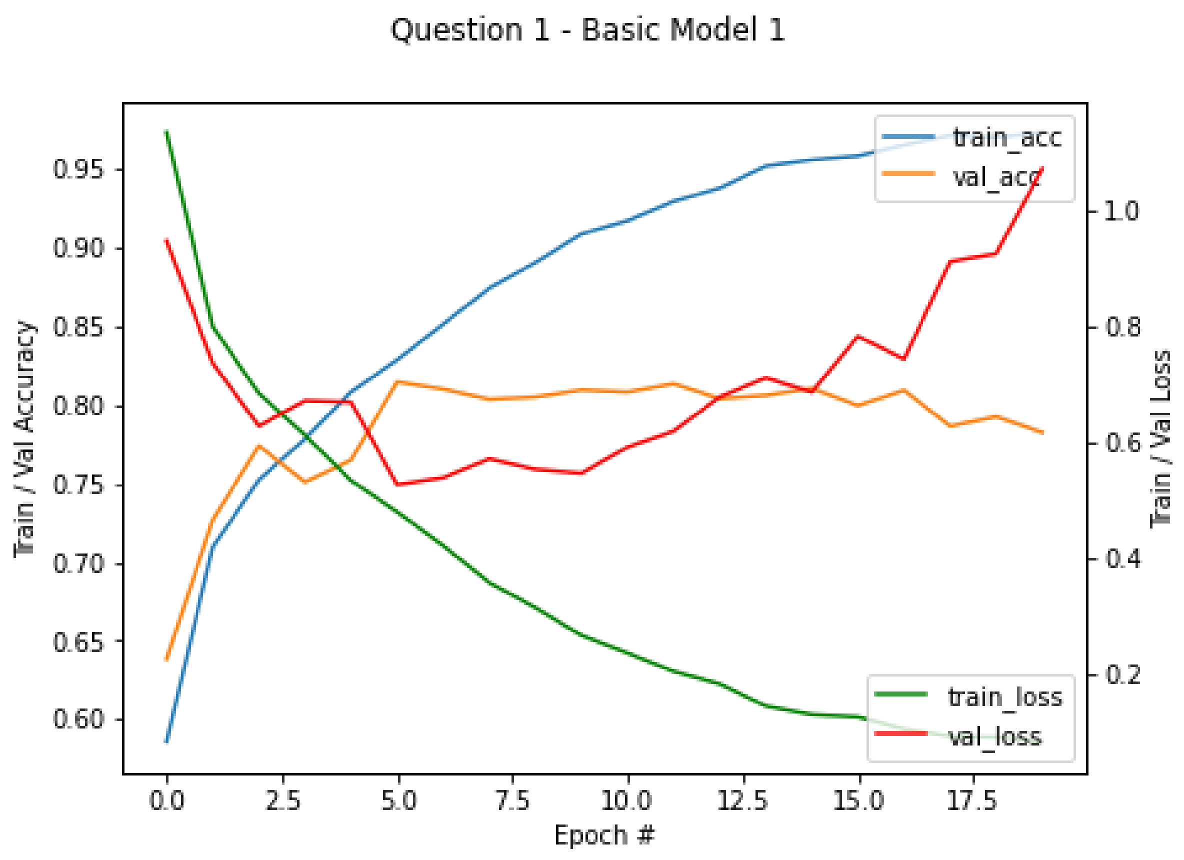
The validation loss diverges from epoch 2 and by the end of 20th epoch, the loss is ~1.0

Model 1 is the extension of baseline model with one extra block as shown in the code below



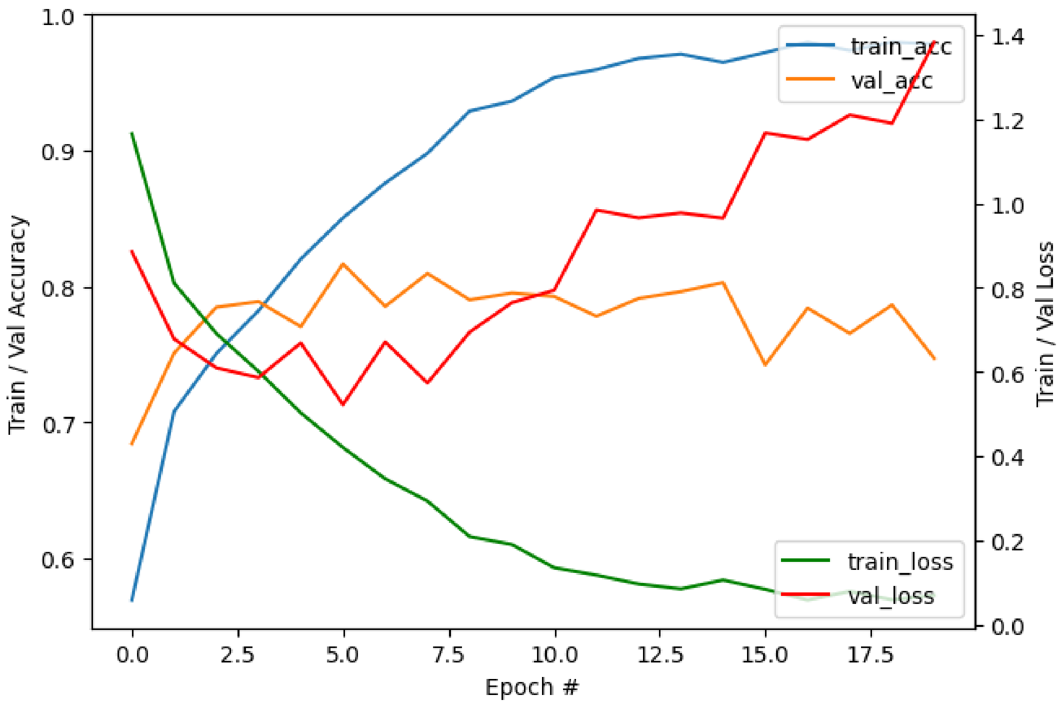
```
def model1():
  """Model with two blocks and fully connected layer"""
  #block1
  input = keras.Input(shape=(64,64,3))
  conv1 = Conv2D(filters = 10, kernel_size = 3, activation = 'relu')(input)
  pool1 = MaxPool2D(pool_size = 2, strides = 2)(conv1)
  #block2
  conv2 = Conv2D(filters = 24, kernel_size = 5, activation = 'relu')(pool1)
  pool2 = MaxPool2D(pool_size = 2, strides = 2)(conv2)
  #fully connected
  flat1 = Flatten()(pool2)
  dense1 = Dense(256, activation='relu')(flat1)
  softmax = Dense(9, activation=tf.nn.softmax)(dense1)
  model1 = keras.Model(inputs=input, outputs=softmax)
  return model1
```

The learning curve for model 1 is as shown in the figure below



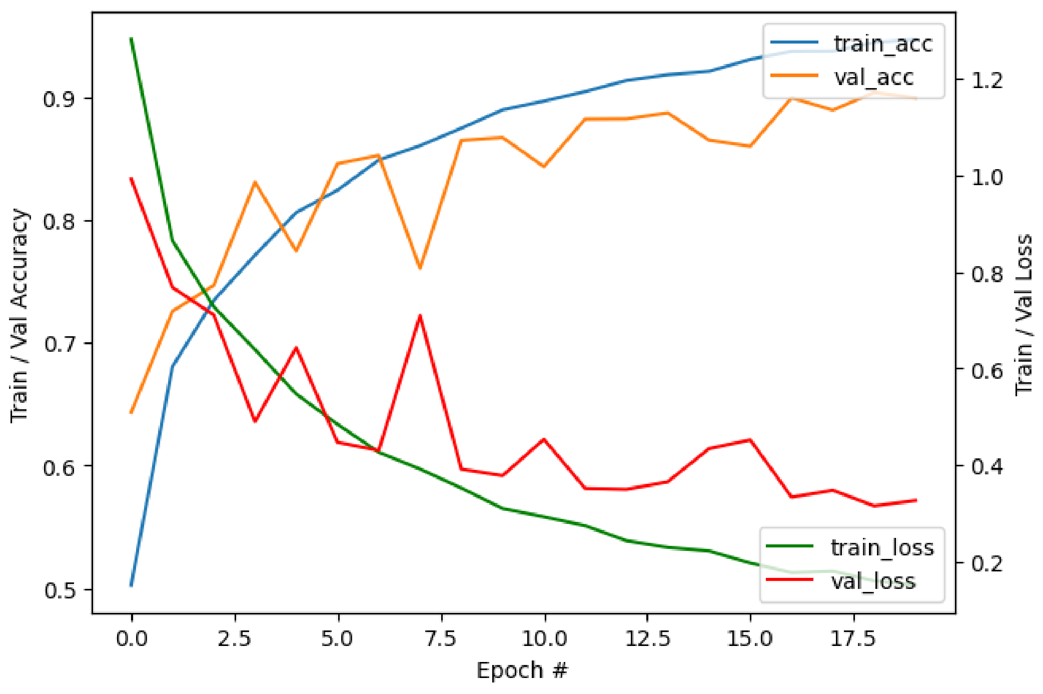
The performance dropped with introduction new block, this is because the input data for any layers was not padded, which results is loss of information on the edges when performing convolution, so I continued to add another block in Model 2 to check if adding another block and making network deeper helps, but the performance of the model is worse as shown in the learning curve below

Question 1 - Basic Model 2



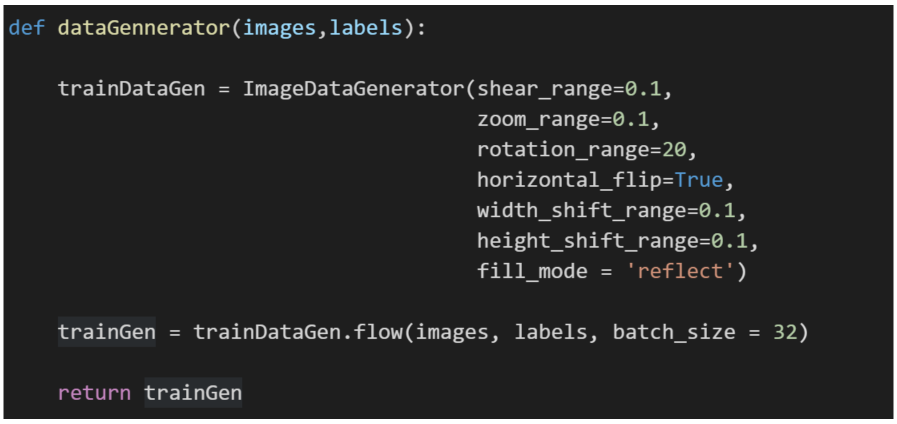
The padding is set to 'same' which saves the dimensions from the previous layers hence retaining the information from the pixels in the edges, this has shown good improvement in accuracy and losses as shown in the figure below, even if there is ripples and divergence in the validation loss with epochs, the results are way better than the other models above, this model is the best performing model and is selected to test augmented data.

Question 1 - Basic Model 3



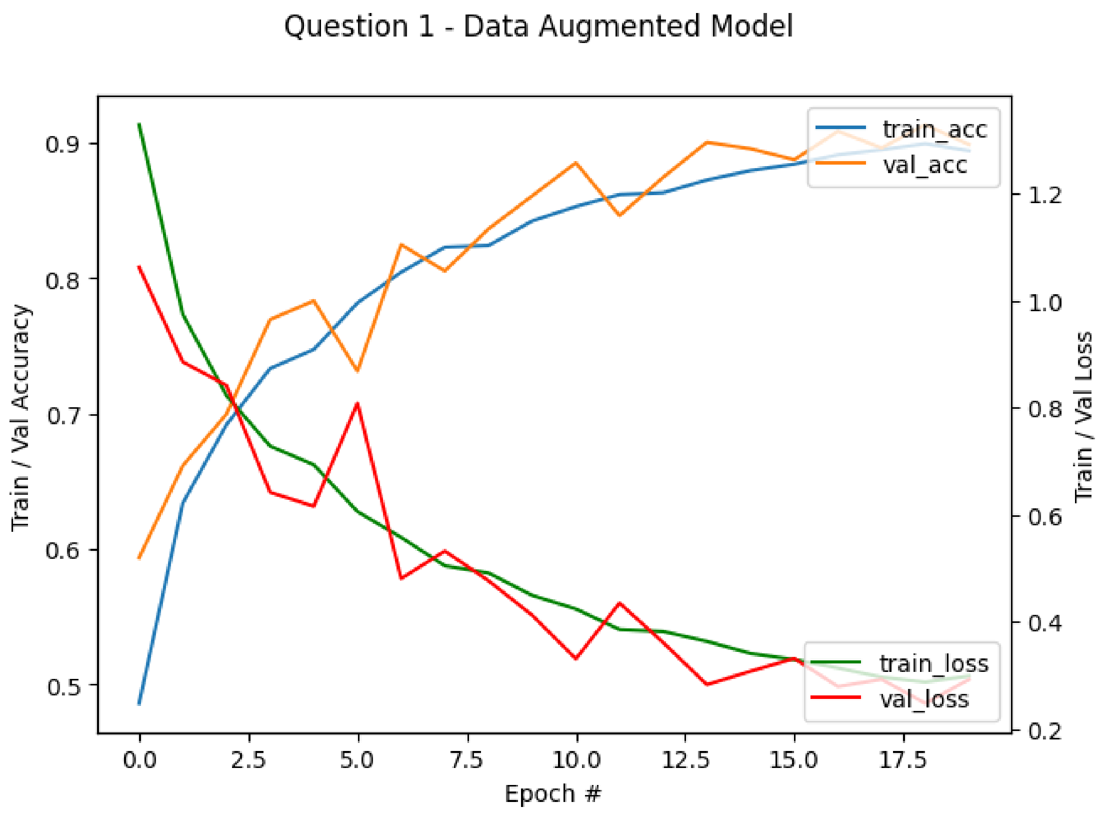
Model 3 is the best performing model, but it should be noted that as we go deeper in the network, the trainable parameters also increase abundantly and will cost execution time and computational power. Therefore, POOL layers are used, they reduce the number of tuneable parameters by reducing the dimensions of data. There are few other ways to mitigate the overfitting we see in all basic models above including the best performing basic model 3, we will mainly focus on Data Augmentation below

Data Augmentation is applied to the raw data for generating augmented data. This can be achieved using several techniques like zooming, rotation of image, mirroring, cropping, shear and blurring, varying brightness etc. I have used the parameters listed below to generate new data from existing earth data set



```
def dataGennerator(images,labels):

    trainDataGen = ImageDataGenerator(shear_range=0.1,
                                      zoom_range=0.1,
                                      rotation_range=20,
                                      horizontal_flip=True,
                                      width_shift_range=0.1,
                                      height_shift_range=0.1,
                                      fill_mode = 'reflect')

    trainGen = trainDataGen.flow(images, labels, batch_size = 32)

    return trainGen
```

The learning curves of model 3 using data augmented images is shown in the figure below



From the above figure, the validation loss has some ripples because each time the training data batch contains different augmented data, but there is no overfitting of the data, model 3 with data augmentation attained accuracies of 90% training accuracy and 89% validation accuracy. Also, the losses were low compared to other models, ~0.25

Therefore, I can conclude that augmenting data using techniques such as zoom, shear, flipping, mirroring, shifting has a positive impact on the performance of my model. Hence, data augmentation is very useful when we have very little data available, also it can reduce overfitting of the model.

Apart from the basic data augmentation techniques, there are deep learning augmentation models which learn the data representation and generate realistic looking fake data which can then be fed into our model to reduce overfitting. This process is also called image synthesis. Main DL networks used for image synthesis are GANs (Generative Adversarial Networks) and its variants. The figure below shows basic GAN flow and an example of GAN used for augmentation of Lungs nodule dataset [1]

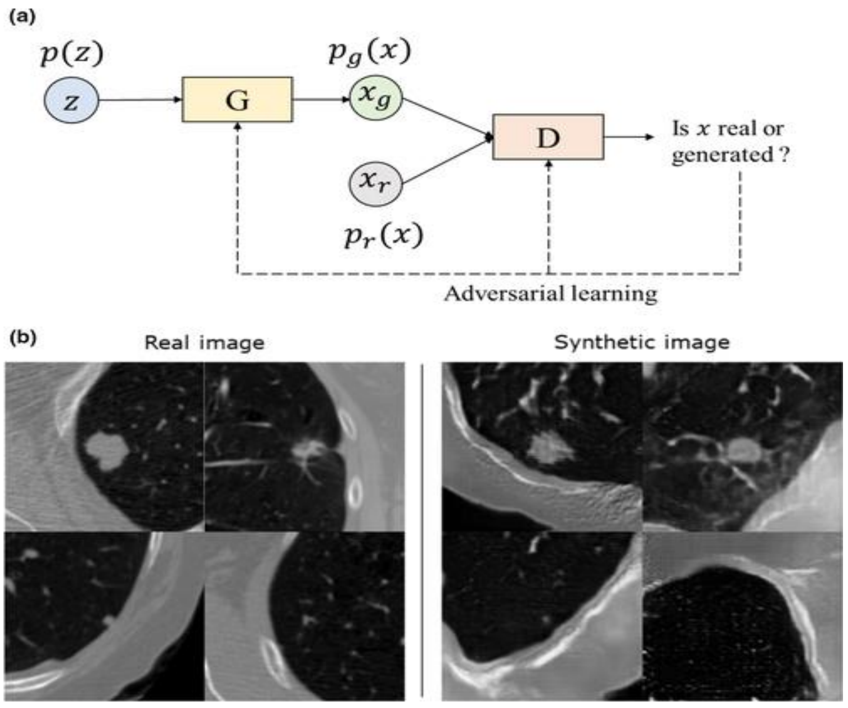


Figure [1]

Another data augmentation technique used in Mixing Images [2], the pixel values are averaged in a counterintuitive approach, the output of this network won’t be useful for a human eye, Inouye [3] demonstrated how mixing of the images randomly from the entire dataset without considering the class it belongs reduce the error rate from 8.22% to 6.9% on the whole of CIFAR-10 dataset. Later the non-linear mixing of images was tested by Summers and Dineen [4] which reduced the error rate from 23% to 19% on CIFAR-100 dataset, an example of the augmented data using Image Mixing is shown in the figure below.

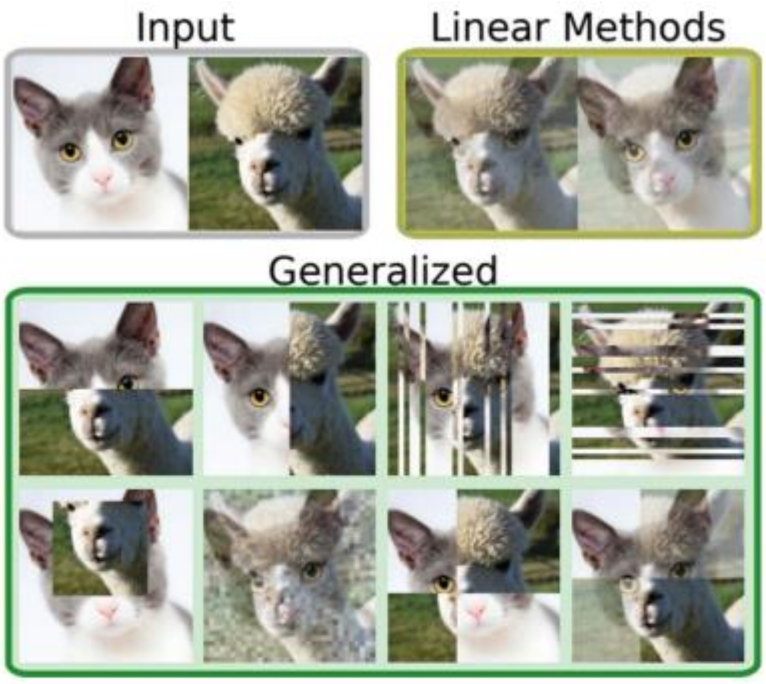


Figure -Non-Linear Mixing of Images [3]

## One Shot Learning

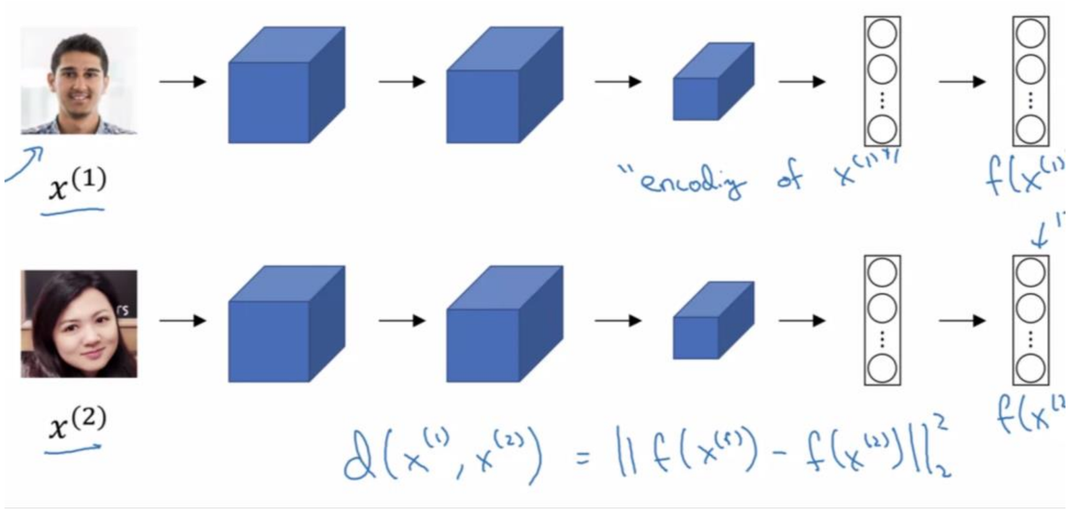
Historically, deep learning networks wont work well with only few or one training example. The model will poorly perform with only one training image. This is because the CNN model must be trained with several images of the same class before applying the model for recognition or prediction. This is a main problem for facial verification and facial recognition models since the data might not contain plenty images of a person in the dataset to train and recognize the person.  So, One Shot Learning is used, this algorithm uses the similarity function as shown in the snippet below

$$d(\text{img1},\text{img2}) = \text{degree of difference between images}$$

The equation above gives degree of difference between input image and the other images in the dataset, so if the input image (ing1) and second image (img2) is of the same person, 'd' should be very small, if they are of two different people, difference should be large

During the predictions, a threshold is set to such that if the difference d is larger than the threshold, img1 and img2 are different, if it is lower than the threshold, img1 and img2 are same person [5]

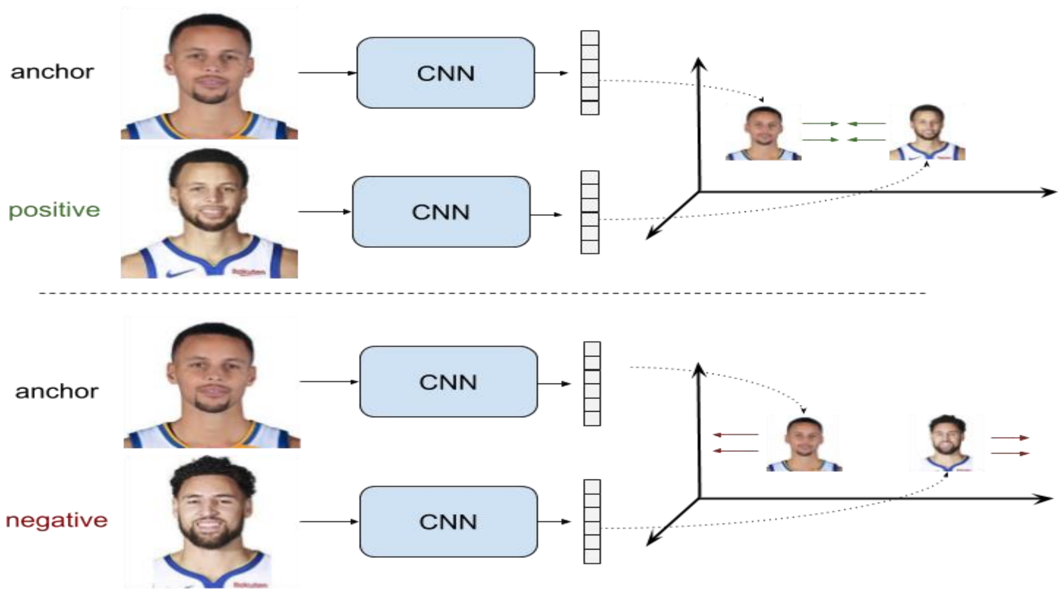
The one-shot learning is applied using Siamese Network as shown below



The image x1 is passed through the neural network and a final layer is used as feature extractor (for example 128 vector in the above image). Likewise, we do the same for the image x2 to check if it is of the same person (it is a different person in the figure above), Siamese network was developed by Taigman et al. in their paper DeepFace.

The encodings above (f(x)) are used in triplet cost function to train the model (A – Anchor or the input image, P – Positive or the image of the same person, N – Negative or the image of a different person), we want the D(a,p) to be small and D(a,n) to be large

$$L(A, P, N) = max(0, d(A, P) - d(A, N) + alpha)$$



So, the network should be trained to reduce the triplet loss which will perform well in face recognizing and verification

## Zero Shot Learning

Unlike DL networks or one-shot learning where the model is given at least 1 training example, zero shot learning is a technique used to recognize patterns with 0 training examples. For an object detection scenario, there are many categories of objects and these can be annotated to a certain degree, but when there are millions of categories (example species of birds) we cannot use conventional supervised learning to these areas

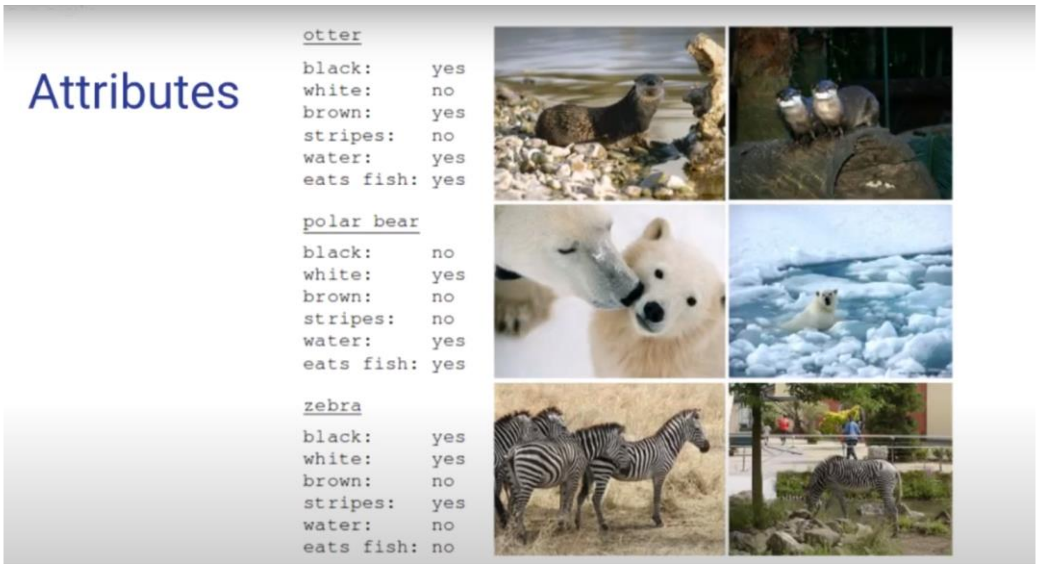
So, if there is a dataset which is annotated by humans to train it on a supervised learning algorithm, and few new examples will be added which do not belong to any categories present in the dataset, these will still be sitting in the same image feature space, but the classifiers won't know what to do with these

Zero shot learning is used to classify the categories that were not seen by the model before, this is done by combining seen and unseen categories through some auxiliary information which gives distinguishing properties of an object, here

- **Seen classes:** labelled data classes available for training model
- **Unseen classes:** data classes which occur during test/inference time which the model has never seen
- **Auxiliary information:** Attributes or information on both seen/unseen data during training of the model

There are two stages, namely training and inference, in the training stage, attribute knowledge is captured, then later in the inference stage the captured attributes are used to categorise new category instances [6]

Examples for attributes that are used in ZSL for animal dataset is shown in the figure below



Attributes

otter

| | |
|---|---|
| black: | yes |
| white: | no |
| brown: | yes |
| stripes: | no |
| water: | yes |
| eats fish: | yes |

polar bear

| | |
|---|---|
| black: | no |
| white: | yes |
| brown: | no |
| stripes: | no |
| water: | yes |
| eats fish: | yes |

zebra

| | |
|---|---|
| black: | yes |
| white: | yes |
| brown: | no |
| stripes: | yes |
| water: | no |
| eats fish: | no |

Figure: attributes example for ZSL [6]

There are two main approaches for ZLS

1. Conventional ZSL – Only unseen data during test
2. Generalized ZSL – Seen and unseen data during test, this is more widely used

In the train phase, the input image is passed through an encoder with its attribute vector, this encoder will learn the hidden information by trying to map the input image vector to its attribute vector. Transformers are being used now to generate sophisticated word embedding attribute vectors. [7]

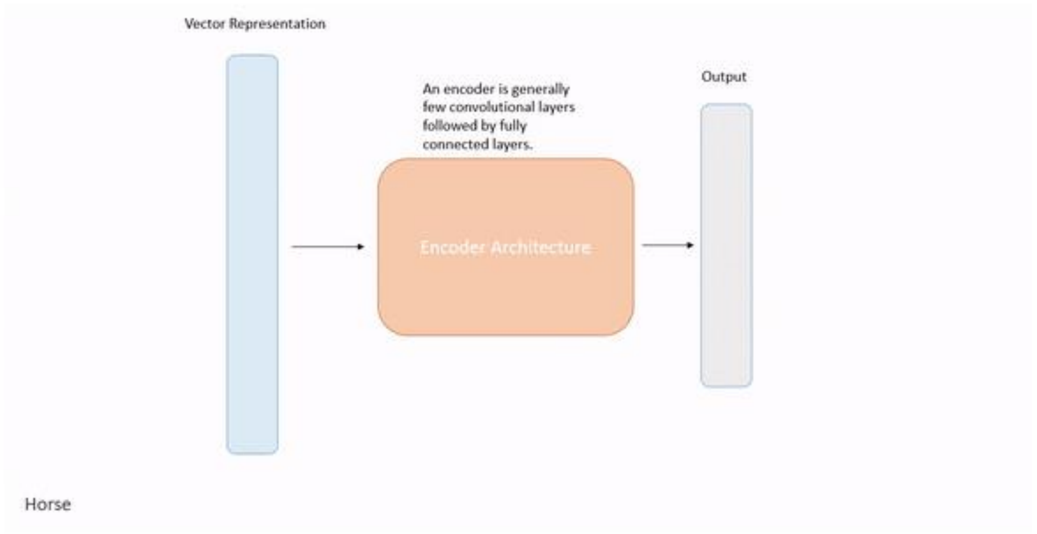

Vector Representation
An encoder is generally few convolutional layers followed by fully connected layers.
Output
Encoder Architecture
Horse

Figure: Simple ZLS representation – train phase [7]

During the test phase, the unseen image is passed through the trained encoder which outputs the attribute vector for the current input image, this vector is compared to the existing attribute vectors to find the similarities between them. The vectors with high similarity score will be the predicted class. The inference phase is as shown in the animation below

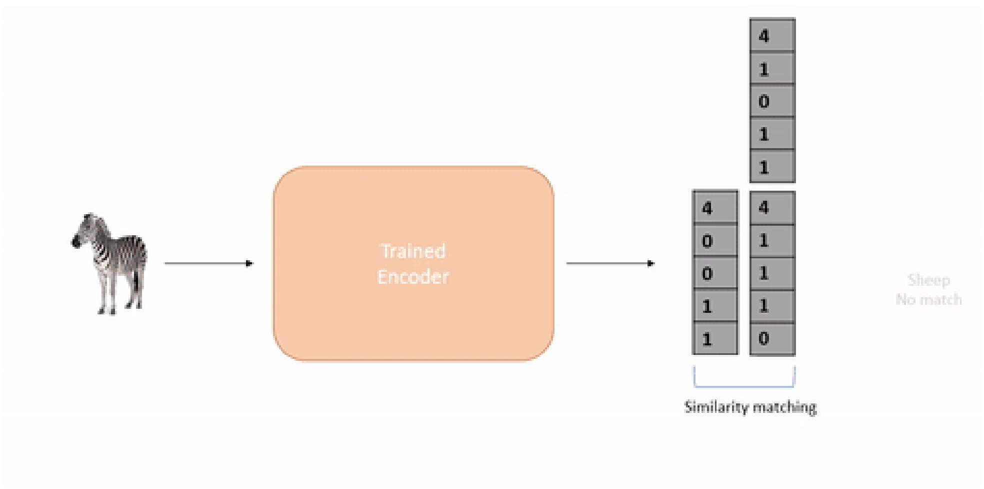


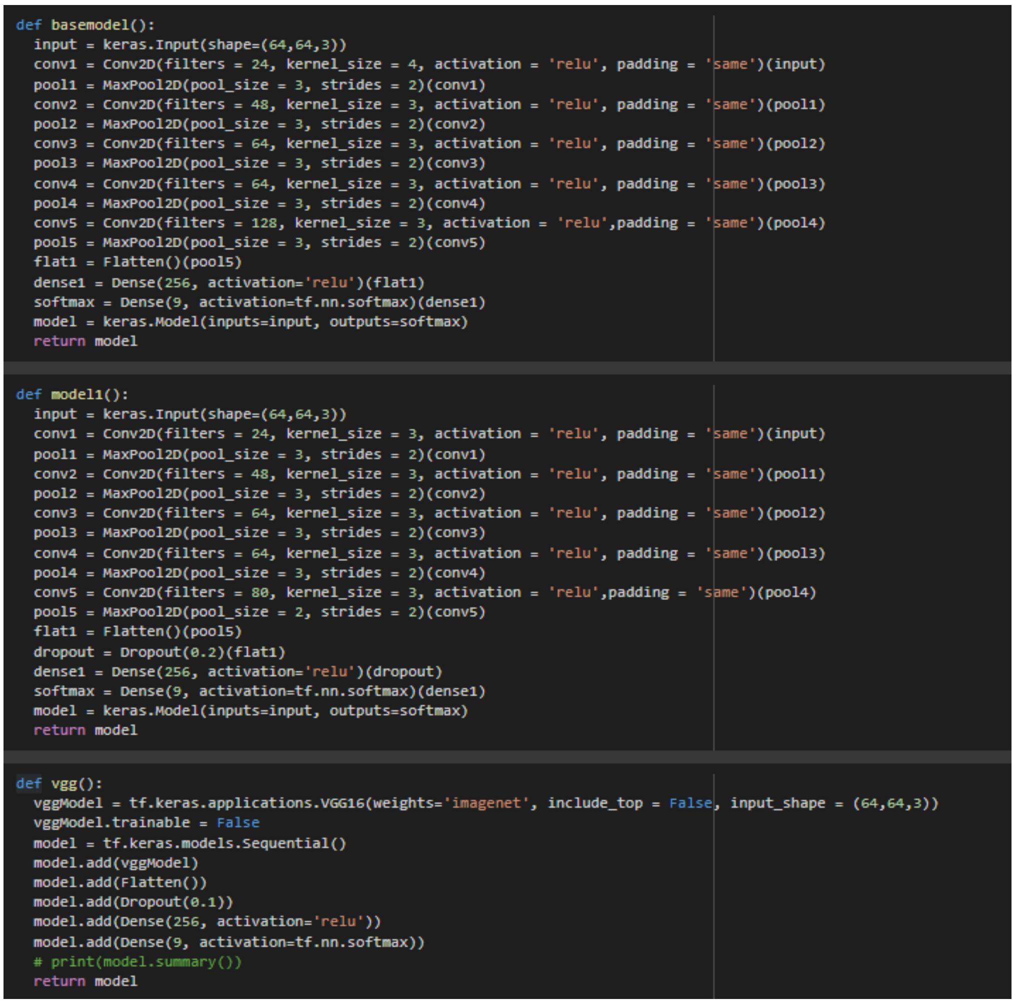
Figure: Inference phase of a simple ZSL [7]

## Part A (ii)

## CNN Ensembles

DL neural networks are nonlinear so each time they see the data they train differently with weights being upgraded to different values at each epoch resulting in slight changes in the prediction even when the data and the dimensions or the architecture of the network is the same. This is referred to as variance and this becomes a pain when we want to come up with a final design of the network. Ensemble technique is used to reduce the variance in the output by training multiple models on the same training data and predicting the classes using all trained models, finally the predictions from all different models are averaged which will be the final prediction of the DL network.

There are different techniques to adapt ensemble, by changing the training data on each iteration which means the same model will get different flavour of inputs on each iteration. In our model we have adapted data augmentation which helps us with this first ensemble method, with data augmentation, the model will get different data on every epoch. The second technique is by using different models on the training data, I have created 3 different base models, two models are basic CNN models and third is transfer learned VGG16 with flattened and SoftMax layer attached to its end for predictions. The models are shown in the figure below

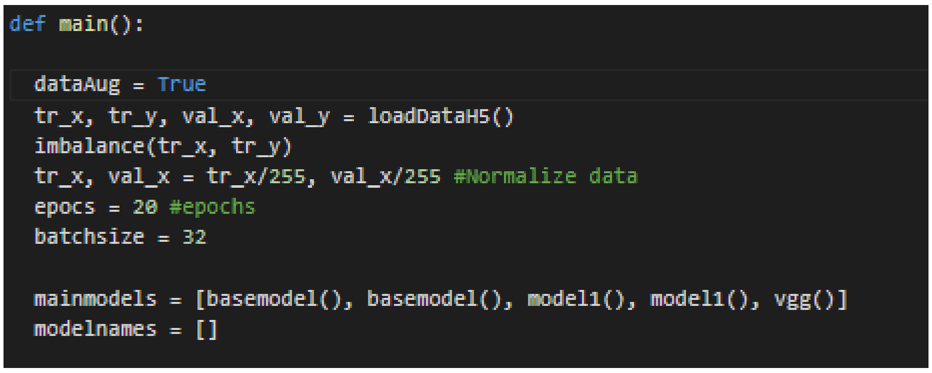


```
def basemodel():
  input = keras.Input(shape=(64,64,3))
  conv1 = Conv2D(filters = 24, kernel_size = 4, activation = 'relu', padding = 'same')(input)
  pool1 = MaxPool2D(pool_size = 3, strides = 2)(conv1)
  conv2 = Conv2D(filters = 48, kernel_size = 3, activation = 'relu', padding = 'same')(pool1)
  pool2 = MaxPool2D(pool_size = 3, strides = 2)(conv2)
  conv3 = Conv2D(filters = 64, kernel_size = 3, activation = 'relu', padding = 'same')(pool2)
  pool3 = MaxPool2D(pool_size = 3, strides = 2)(conv3)
  conv4 = Conv2D(filters = 64, kernel_size = 3, activation = 'relu', padding = 'same')(pool3)
  pool4 = MaxPool2D(pool_size = 3, strides = 2)(conv4)
  conv5 = Conv2D(filters = 128, kernel_size = 3, activation = 'relu',padding = 'same')(pool4)
  pool5 = MaxPool2D(pool_size = 3, strides = 2)(conv5)
  flat1 = Flatten()(pool5)
  dense1 = Dense(256, activation='relu')(flat1)
  softmax = Dense(9, activation=tf.nn.softmax)(dense1)
  model = keras.Model(inputs=input, outputs=softmax)
  return model
```

```
def model1():
  input = keras.Input(shape=(64,64,3))
  conv1 = Conv2D(filters = 24, kernel_size = 3, activation = 'relu', padding = 'same')(input)
  pool1 = MaxPool2D(pool_size = 3, strides = 2)(conv1)
  conv2 = Conv2D(filters = 48, kernel_size = 3, activation = 'relu', padding = 'same')(pool1)
  pool2 = MaxPool2D(pool_size = 3, strides = 2)(conv2)
  conv3 = Conv2D(filters = 64, kernel_size = 3, activation = 'relu', padding = 'same')(pool2)
  pool3 = MaxPool2D(pool_size = 3, strides = 2)(conv3)
  conv4 = Conv2D(filters = 64, kernel_size = 3, activation = 'relu', padding = 'same')(pool3)
  pool4 = MaxPool2D(pool_size = 3, strides = 2)(conv4)
  conv5 = Conv2D(filters = 80, kernel_size = 3, activation = 'relu',padding = 'same')(pool4)
  pool5 = MaxPool2D(pool_size = 2, strides = 2)(conv5)
  flat1 = Flatten()(pool5)
  dropout = Dropout(0.2)(flat1)
  dense1 = Dense(256, activation='relu')(dropout)
  softmax = Dense(9, activation=tf.nn.softmax)(dense1)
  model = keras.Model(inputs=input, outputs=softmax)
  return model
```

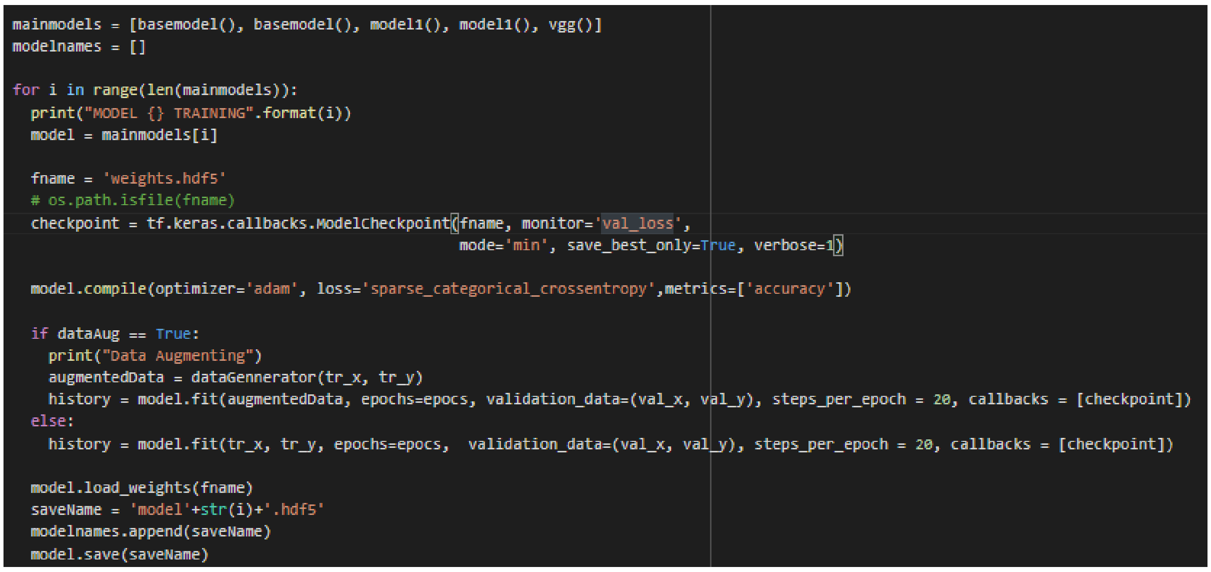
```
def vgg():
  vggModel = tf.keras.applications.VGG16(weights='imagenet', include_top = False, input_shape = (64,64,3))
  vggModel.trainable = False
  model = tf.keras.models.Sequential()
  model.add(vggModel)
  model.add(Flatten())
  model.add(Dropout(0.1))
  model.add(Dense(256, activation='relu'))
  model.add(Dense(9, activation=tf.nn.softmax))
  # print(model.summary())
  return model
```

I then created 5 models, two are from basemodel() and two are from model1() and the last one is from vgg() as shown below in the snippet. This is to introduce a source of variability in the ensembles, the main goal of ensemble is to reduce variance when a single CNN is trained.



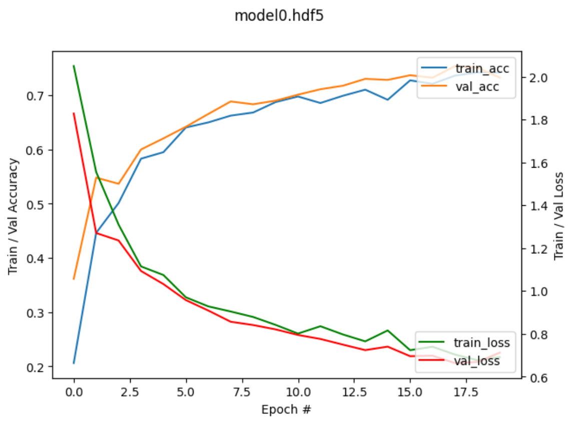
```
def main():

  dataAug = True
  tr_x, tr_y, val_x, val_y = loadDataH5()
  imbalance(tr_x, tr_y)
  tr_x, val_x = tr_x/255, val_x/255 #Normalize data
  epocs = 20 #epochs
  batchsize = 32

  mainmodels = [basemodel(), basemodel(), model1(), model1(), vgg()]
  modelnames = []
```

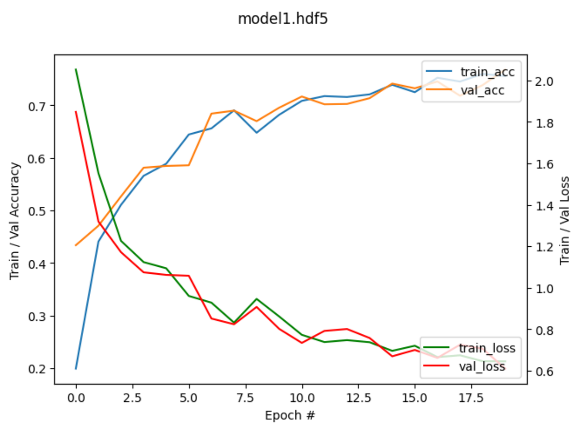
The code loops on each model to train the model for the number of Epochs set by the user, checkpointing is used to save the best performing epoch on each 5 models, the models are saved according to the best weights (minimum validation loss). These best models are then used to predict the test images and the predictions are averaged as the predictions for ensemble technique. The figure below is the snippet from the main() function of my code. It shows everything I described in this paragraph.

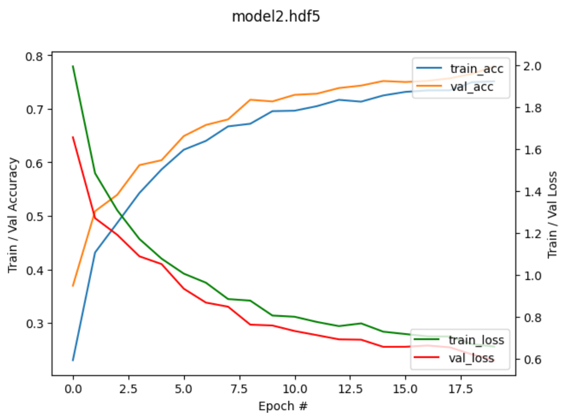


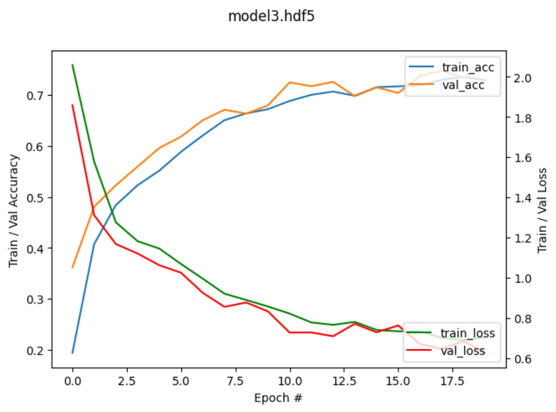
```
mainmodels = [basemodel(), basemodel(), model1(), model1(), vgg()]
modelnames = []

for i in range(len(mainmodels)):
  print("MODEL {} TRAINING".format(i))
  model = mainmodels[i]

  fname = 'weights.hdf5'
  # os.path.isfile(fname)
  checkpoint = tf.keras.callbacks.ModelCheckpoint(fname, monitor='val_loss',
                                                  mode='min', save_best_only=True, verbose=1)

  model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',metrics=['accuracy'])

  if dataAug == True:
    print("Data Augmenting")
    augmentedData = dataGennerator(tr_x, tr_y)
    history = model.fit(augmentedData, epochs=epocs, validation_data=(val_x, val_y), steps_per_epoch = 20, callbacks = [checkpoint])
  else:
    history = model.fit(tr_x, tr_y, epochs=epocs,  validation_data=(val_x, val_y), steps_per_epoch = 20, callbacks = [checkpoint])

  model.load_weights(fname)
  saveName = 'model'+str(i)+'.hdf5'
  modelnames.append(saveName)
  model.save(saveName)
```

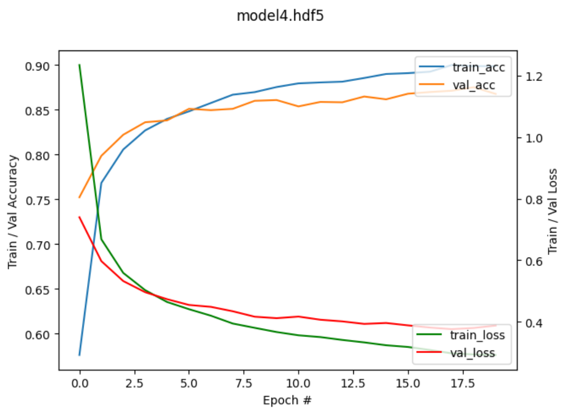
The learning curves for each model is as shown in the figures below







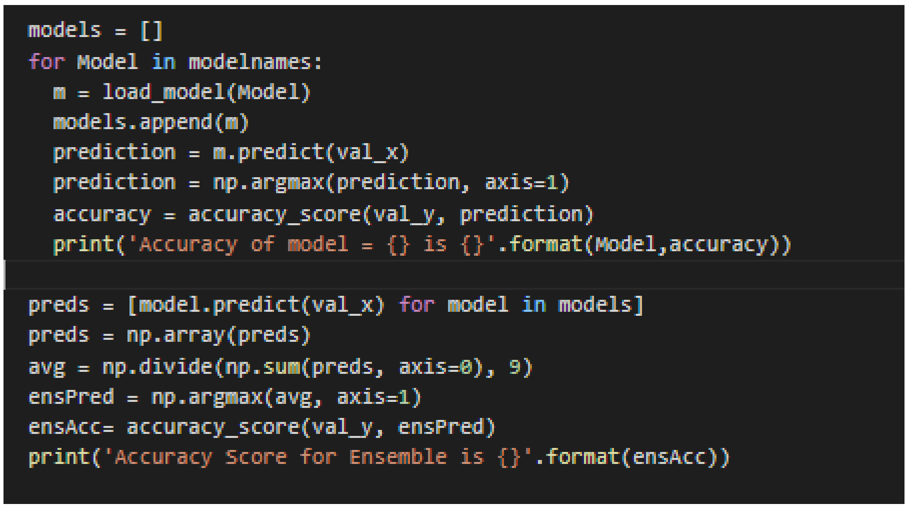




The above figures are learning curves of the 5 model instances

- Model 0, model 1 are created using basemodel()
- Model 2, model 3 are created using model1()
- Model 4 is created using vgg()

The accuracies of the models after training are calculated and the predictions are averaged for ensemble predictions, this is done in the code as shown below


```
models = []
for Model in modelnames:
  m = load_model(Model)
  models.append(m)
  prediction = m.predict(val_x)
  prediction = np.argmax(prediction, axis=1)
  accuracy = accuracy_score(val_y, prediction)
  print('Accuracy of model = {} is {}'.format(Model,accuracy))

preds = [model.predict(val_x) for model in models]
preds = np.array(preds)
avg = np.divide(np.sum(preds, axis=0), 9)
ensPred = np.argmax(avg, axis=1)
ensAcc= accuracy_score(val_y, ensPred)
print('Accuracy Score for Ensemble is {}'.format(ensAcc))
```

were reported as

- Accuracy of model = model0.hdf5 is 0.7535416666666667
- Accuracy of model = model1.hdf5 is 0.768125
- Accuracy of model = model2.hdf5 is 0.7791666666666667
- Accuracy of model = model3.hdf5 is 0.7589583333333333
- Accuracy of model = model4.hdf5 is 0.87125
- Accuracy Score for Ensemble is 0.8410416666666667

As expected, the transfer learned VGG16 with custom flattening and SoftMax layer performed the best, but we can see that the ensemble validation accuracy is 84% which is way better in comparison to the scores of other networks

## Additional ensemble techniques

The future works include implementation of different ensemble techniques and finding out which one will give the best accuracy and least losses while not overfitting or underfitting on the data.

First approach would be varying training data, I would divide the training data by the number of models that are to be trained, if 5 models are used for ensemble, we can divide the training data into 5 batches and use different data for each model. The trained models can then be tested on the validation data to find accuracies. These accuracies can then be averaged to get the ensemble prediction accuracy. This can be done using K-Fold Cross-Validation. K models are trained on K subsets of data.

Bagging or bootstrap aggregation has shown significant improvement in the performance of ensemble models.

The other approach would be using the different models and training all models to find the best performing models. Then using weights while taking averages for ensemble probabilities.

We can assign higher weights to the models which are performing better than the other models. This will help in gaining high accuracy while reducing the variance which is the goal of ensemble [8].

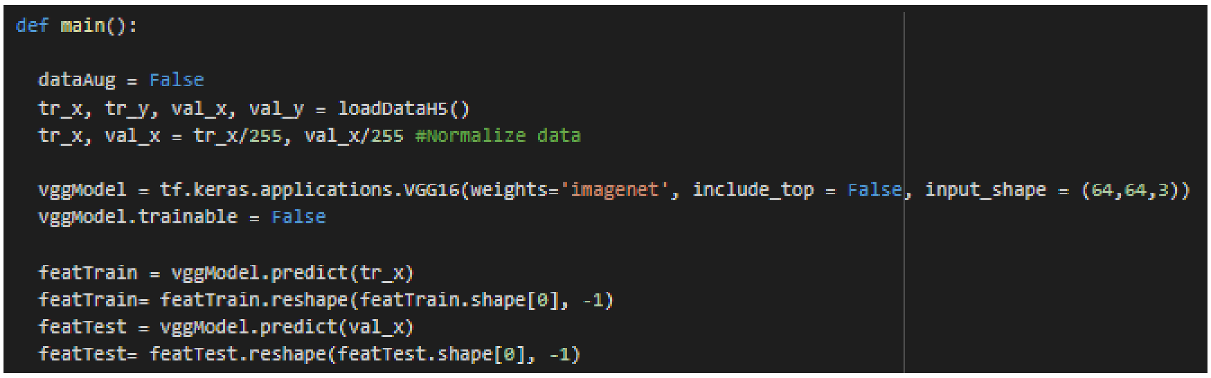
## Part B

### Transfer Learning

### Part B (i)

Training a DNN is very costly on execution time and computational resources, also when we are not sure how the good the network is, it makes everything worse! So, the solution to these issues is to use the networks online which are the best performing networks for our current domain, leverage the existing architecture and train only the final neurons to perform predictions on our data.

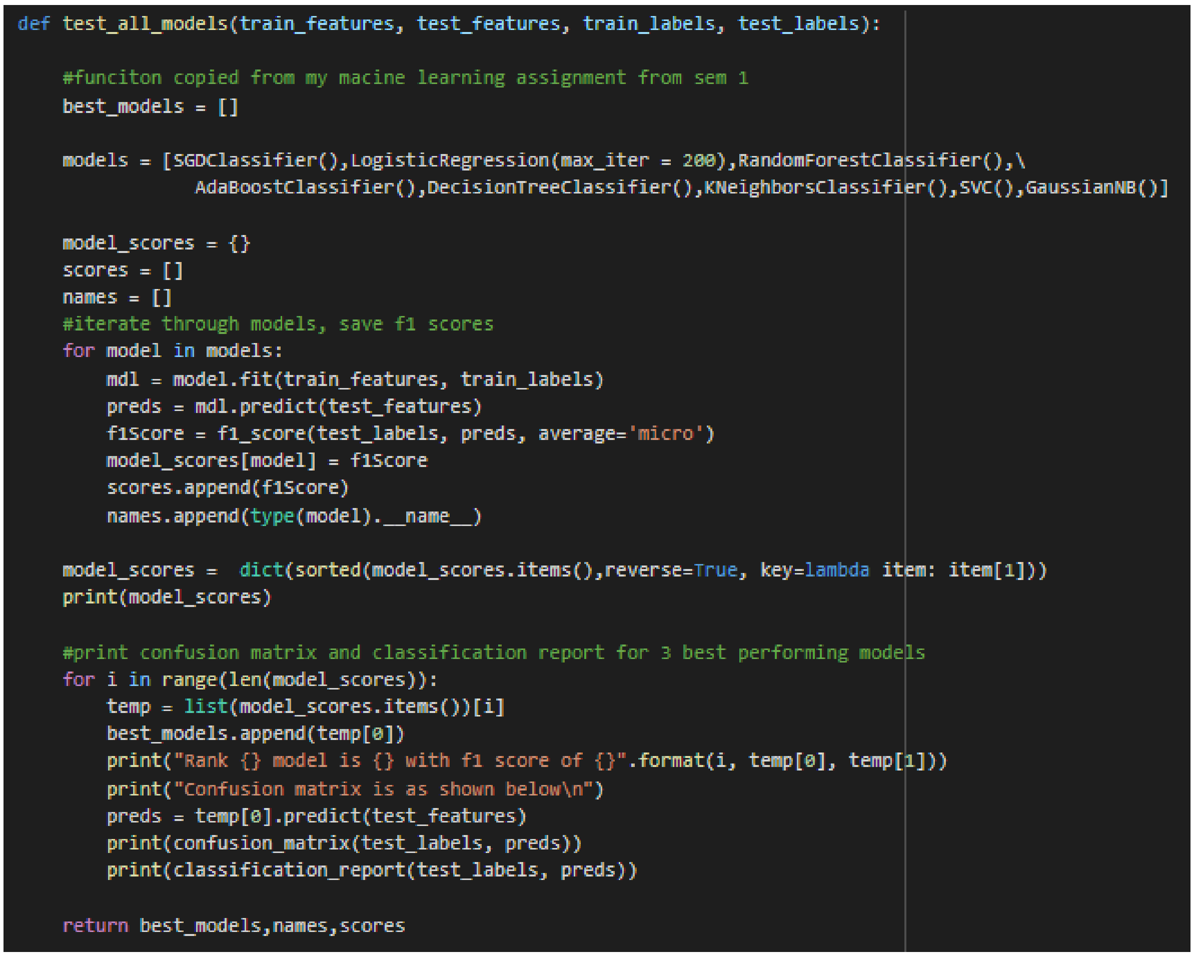
I have selected VGG16 as my model for transfer learning. The code is as shown in the figure below where the final layers of the model are turned off or dropped completely and the featured of the layer before flattening the features is used to extract the featured of our data



```
def main():

  dataAug = False
  tr_x, tr_y, val_x, val_y = loadDataH5()
  tr_x, val_x = tr_x/255, val_x/255 #Normalize data

  vggModel = tf.keras.applications.VGG16(weights='imagenet', include_top = False, input_shape = (64,64,3))
  vggModel.trainable = False

  featTrain = vggModel.predict(tr_x)
  featTrain= featTrain.reshape(featTrain.shape[0], -1)
  featTest = vggModel.predict(val_x)
  featTest= featTest.reshape(featTest.shape[0], -1)
```

In the figure, we use vggModel with top set to False, then the features are extracted by running prediction on our train and validation data, the features are flattened into 2 dimensions, the first dimension denotes the training and validation examples (19200 and 4800) and the second layer is a flattened columns denoting the features of the instance.

I have created a function called test-all-models() to test 8 different models with the features extracted from VGG16, the accuracies are in the range of 48% to 87% with Logistic Regression model giving the best performance compared to all other models



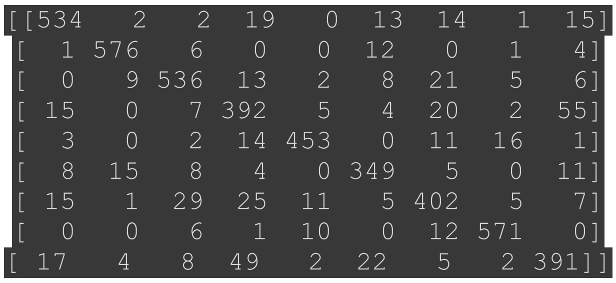
```python
def test_all_models(train_features, test_features, train_labels, test_labels):

    #funciton copied from my macine learning assignment from sem 1
    best_models = []

    models = [SGDClassifier(),LogisticRegression(max_iter = 200),RandomForestClassifier(),\
               AdaBoostClassifier(),DecisionTreeClassifier(),KNeighborsClassifier(),SVC(),GaussianNB()]

    model_scores = {}
    scores = []
    names = []
    #iterate through models, save f1 scores
    for model in models:
        mdl = model.fit(train_features, train_labels)
        preds = mdl.predict(test_features)
        f1Score = f1_score(test_labels, preds, average='micro')
        model_scores[model] = f1Score
        scores.append(f1Score)
        names.append(type(model).__name__)

    model_scores =  dict(sorted(model_scores.items(),reverse=True, key=lambda item: item[1]))
    print(model_scores)

    #print confusion matrix and classification report for 3 best performing models
    for i in range(len(model_scores)):
        temp = list(model_scores.items())[i]
        best_models.append(temp[0])
        print("Rank {} model is {} with f1 score of {}".format(i, temp[0], temp[1]))
        print("Confusion matrix is as shown below\n")
        preds = temp[0].predict(test_features)
        print(confusion_matrix(test_labels, preds))
        print(classification_report(test_labels, preds))

    return best_models,names,scores
```

The accuracies of the models are as listed in the table below

| Model Name | F1 Scores |
|---|---|
| LogisticRegression | 0. 87583 |
| SVC | 0.84604 |
| SGDClassifier | 0.8425 |
| Random Forest Classifier | 0.816044 |
| KNeighborsClassifier | 0.7775 |
| DecisionTreeClassiifier | 0.6408 |
| GaussianNB | 0.5447 |
| AdaBoostClassifier | 0.4897 |

Table: Models attached to VGG16 and their accuracies on earth data

The Confusion matrix for top model Logistic Regression is as follows



```
[[534   2   2  19   0  13  14   1  15]
 [  1 576   6   0   0  12   0   1   4]
 [  0   9 536  13   2   8  21   5   6]
 [ 15   0   7 392   5   4  20   2  55]
 [  3   0   2  14 453   0  11  16   1]
 [  8  15   8   4   0 349   5   0  11]
 [ 15   1  29  25  11   5 402   5   7]
 [  0   0   6   1  10   0  12 571   0]
 [ 17   4   8  49   2  22   5   2 391]]
```

The top 4 models are then selected for hyper parameter optimization, Logistic regression accuracy is improved to 0.887 and the best parameters for the model are 1) Solver: ‘liblinear’, 2) max_iters: 50

This strategy still comes with computational and execution time costs since there are millions of trainable parameters for VGG layers. Also testing 8 variants and then grid searching hyperparameters to tune the model will take a long time on COLAB. Hence, I stopped at this point after finding best parameters for Logistic regression model. The future work could be tuning parameters of other best models such as SVC and Random forest, also using subsets of data to save execution time

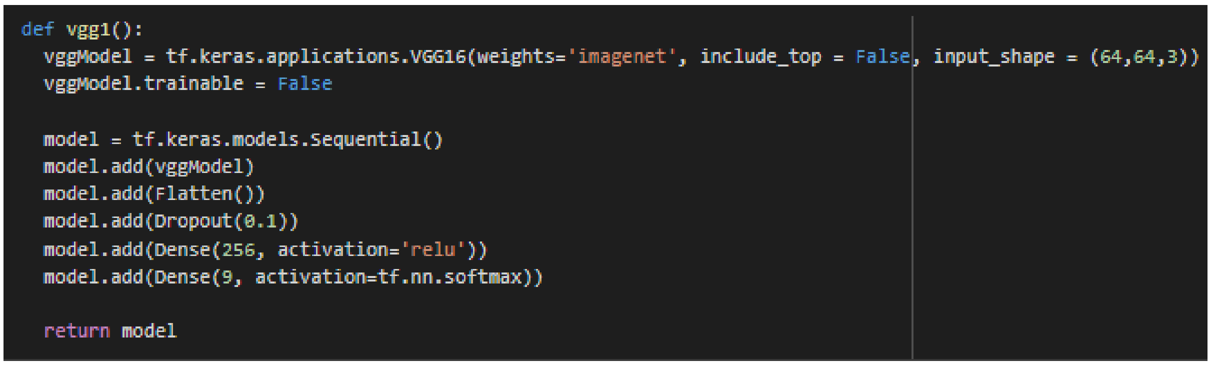
## Part B (ii)

### Fine Tuning

Fine tuning is a part of transfer learning where an existing DNN is leveraged and few or all its layers are re-trained to optimise the weights of the layers suitable for our dataset and predictions.

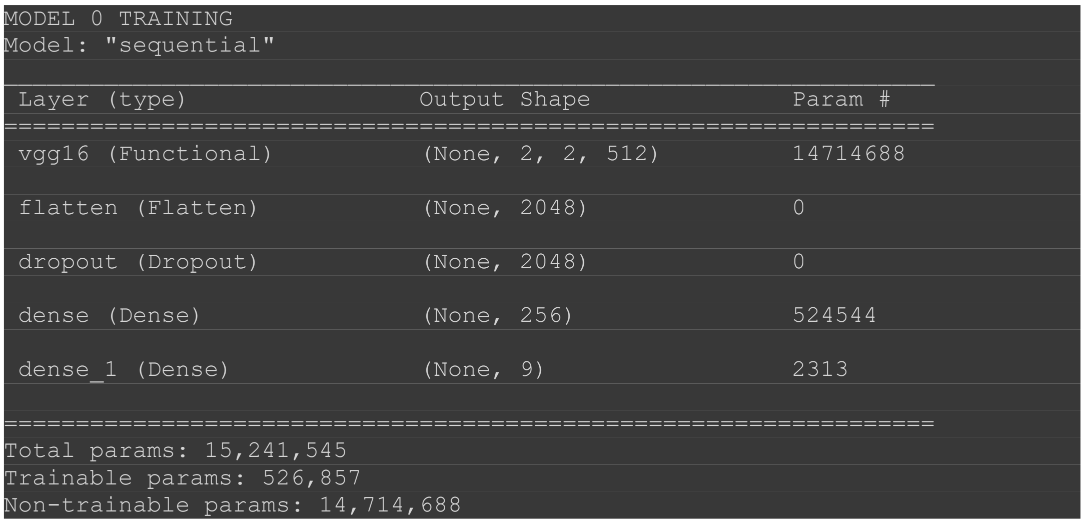
Model 0

I have again used VGG16 model for fine tuning, I created 4 different models for this purpose, the first model is the simple VGG16 model with trainable parameters set to False, which means VGG16 will be running using the weights optimised on ImageNet database, the code snippet of the first model is as shown below.



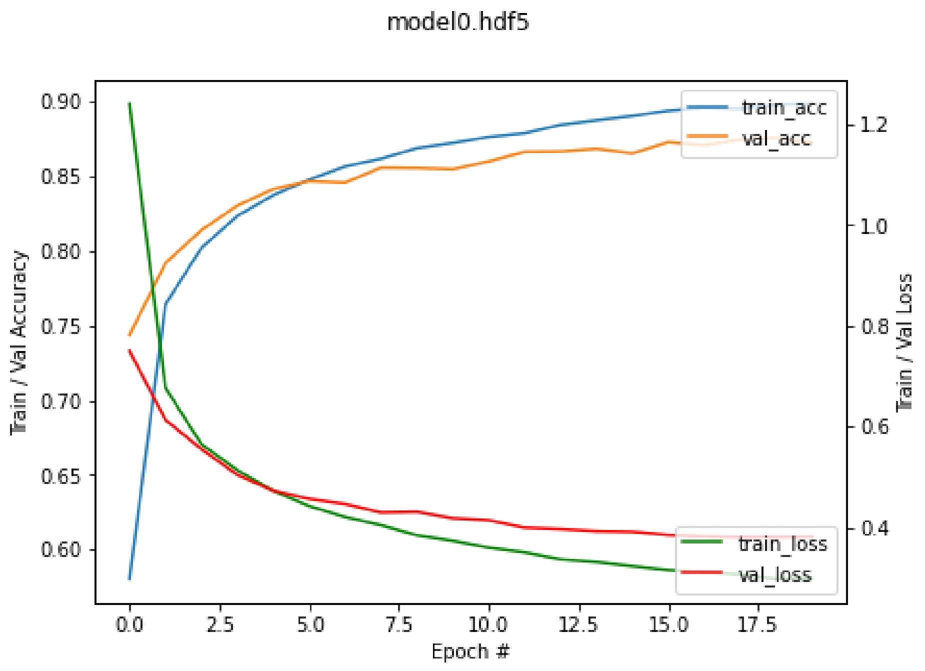
```
def vgg1():
  vggModel = tf.keras.applications.VGG16(weights='imagenet', include_top = False, input_shape = (64,64,3))
  vggModel.trainable = False

  model = tf.keras.models.Sequential()
  model.add(vggModel)
  model.add(Flatten())
  model.add(Dropout(0.1))
  model.add(Dense(256, activation='relu'))
  model.add(Dense(9, activation=tf.nn.softmax))

  return model
```

Since the VGG16 model weight training is set to False, only the additional Flattening and Dense layer weights will be optimised, the summary of the model follows, we can see the trainable parameters are ~526 thousand



```
MODEL 0 TRAINING
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 2, 2, 512)         14714688

 flatten (Flatten)           (None, 2048)              0

 dropout (Dropout)           (None, 2048)              0

 dense (Dense)               (None, 256)               524544

 dense_1 (Dense)             (None, 9)                 2313

=================================================================
Total params: 15,241,545
Trainable params: 526,857
Non-trainable params: 14,714,688
```

The learning curve for this model is as shown below



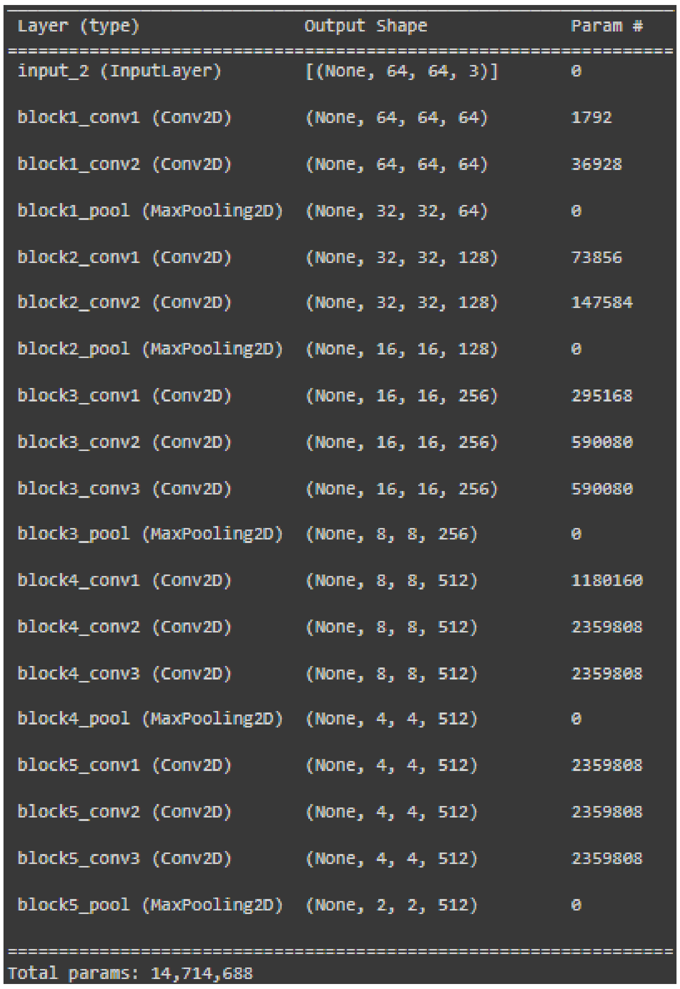
The model is trained for 20 epochs and the final output metrics are as listed below

| Model | Train Loss | Train Accuracy | Val Loss | Val Accuracy |
|---|---|---|---|---|
| VGG model0() | 0.2988 | 0.8986 | 0.3815 | 0.8725 |

The model gives 87% validation accuracy, but the learning curves show that the model overfitting, the losses start diverging after around 4 epochs. This was expected since the weights are not tuned on the VGG model.

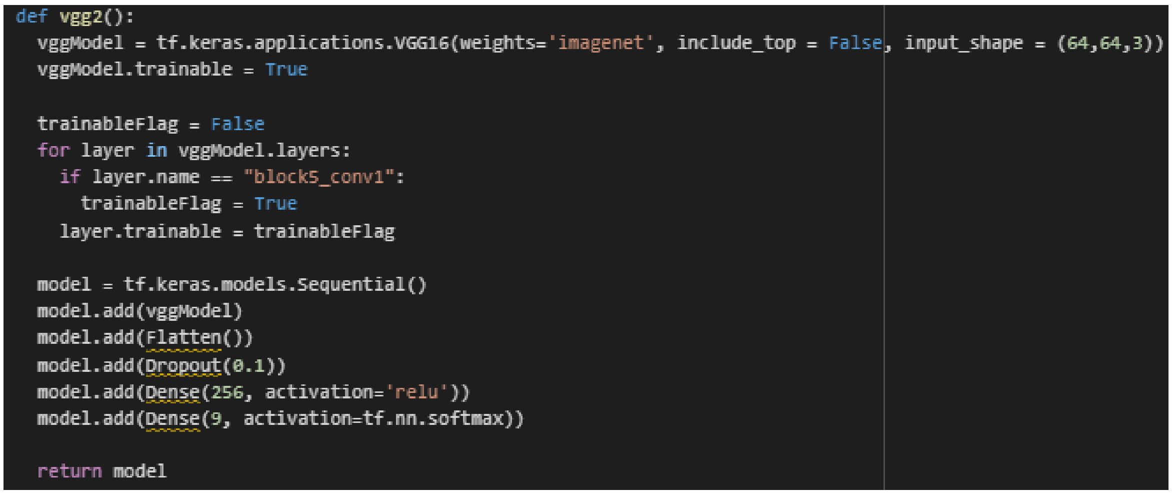
Model 1

The second model is where fine tuning starts, the layers available in VGG is shown in the figure below. I have selected all layers from block 5 and set trainable to True which means the weights and biases of the layers beyond block 5 will be optimised for our data



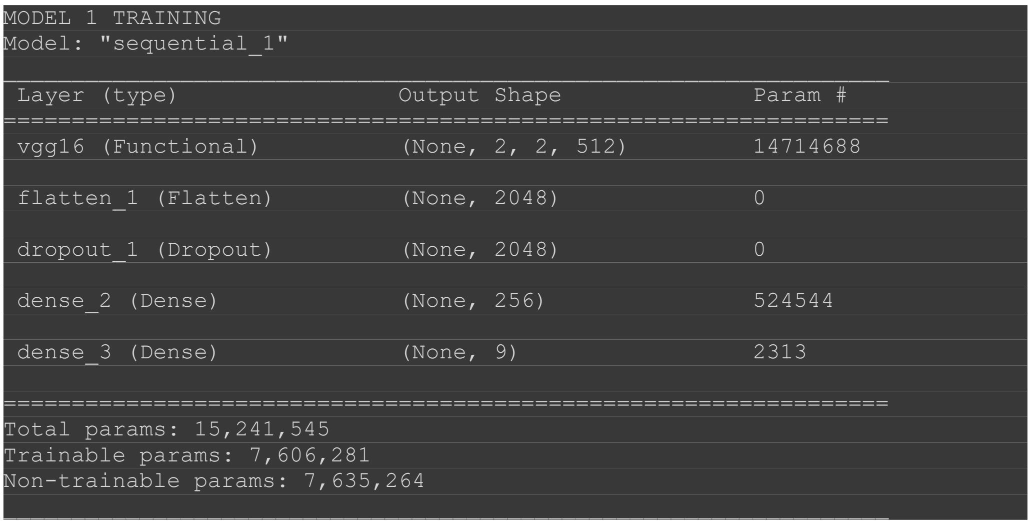
```
 Layer (type)                Output Shape              Param #   
=================================================================
 input_2 (InputLayer)        [(None, 64, 64, 3)]       0         
                                                                 
 block1_conv1 (Conv2D)       (None, 64, 64, 64)        1792      
                                                                 
 block1_conv2 (Conv2D)       (None, 64, 64, 64)        36928     
                                                                 
 block1_pool (MaxPooling2D)  (None, 32, 32, 64)        0         
                                                                 
 block2_conv1 (Conv2D)       (None, 32, 32, 128)       73856     
                                                                 
 block2_conv2 (Conv2D)       (None, 32, 32, 128)       147584    
                                                                 
 block2_pool (MaxPooling2D)  (None, 16, 16, 128)       0         
                                                                 
 block3_conv1 (Conv2D)       (None, 16, 16, 256)       295168    
                                                                 
 block3_conv2 (Conv2D)       (None, 16, 16, 256)       590080    
                                                                 
 block3_conv3 (Conv2D)       (None, 16, 16, 256)       590080    
                                                                 
 block3_pool (MaxPooling2D)  (None, 8, 8, 256)         0         
                                                                 
 block4_conv1 (Conv2D)       (None, 8, 8, 512)         1180160   
                                                                 
 block4_conv2 (Conv2D)       (None, 8, 8, 512)         2359808   
                                                                 
 block4_conv3 (Conv2D)       (None, 8, 8, 512)         2359808   
                                                                 
 block4_pool (MaxPooling2D)  (None, 4, 4, 512)         0         
                                                                 
 block5_conv1 (Conv2D)       (None, 4, 4, 512)         2359808   
                                                                 
 block5_conv2 (Conv2D)       (None, 4, 4, 512)         2359808   
                                                                 
 block5_conv3 (Conv2D)       (None, 4, 4, 512)         2359808   
                                                                 
 block5_pool (MaxPooling2D)  (None, 2, 2, 512)         0         
                                                                 
=================================================================
Total params: 14,714,688
```

The code snippet of the model1() which sets all layers after block5 to trainable is shown below



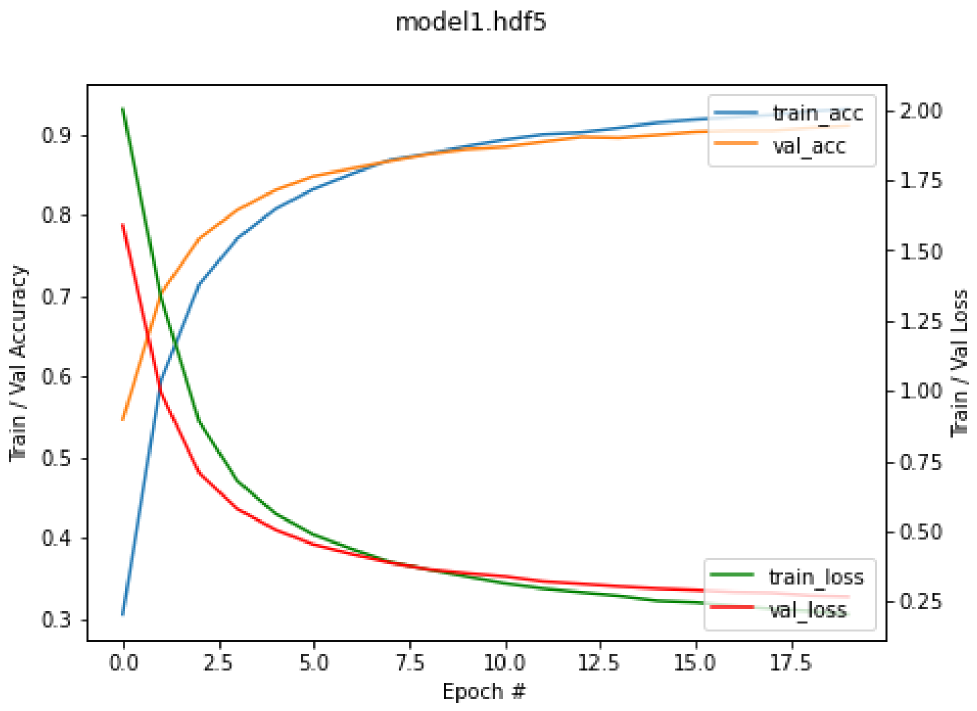
```python
def vgg2():
  vggModel = tf.keras.applications.VGG16(weights='imagenet', include_top = False, input_shape = (64,64,3))
  vggModel.trainable = True

  trainableFlag = False
  for layer in vggModel.layers:
    if layer.name == "block5_conv1":
      trainableFlag = True
    layer.trainable = trainableFlag

  model = tf.keras.models.Sequential()
  model.add(vggModel)
  model.add(Flatten())
  model.add(Dropout(0.1))
  model.add(Dense(256, activation='relu'))
  model.add(Dense(9, activation=tf.nn.softmax))

  return model
```

The summary of my model 1 is as shown below, the trainable parameters are now around 7.5 million, which is too many parameters costing time and resources



```
MODEL 1 TRAINING
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 vgg16 (Functional)          (None, 2, 2, 512)         14714688  
                                                                 
 flatten_1 (Flatten)         (None, 2048)              0         
                                                                 
 dropout_1 (Dropout)         (None, 2048)              0         
                                                                 
 dense_2 (Dense)             (None, 256)               524544    
                                                                 
 dense_3 (Dense)             (None, 9)                 2313      
                                                                 
=================================================================
Total params: 15,241,545
Trainable params: 7,606,281
Non-trainable params: 7,635,264
_________________________________________________________________
```

The learning curve of model1() is shown in the figure below



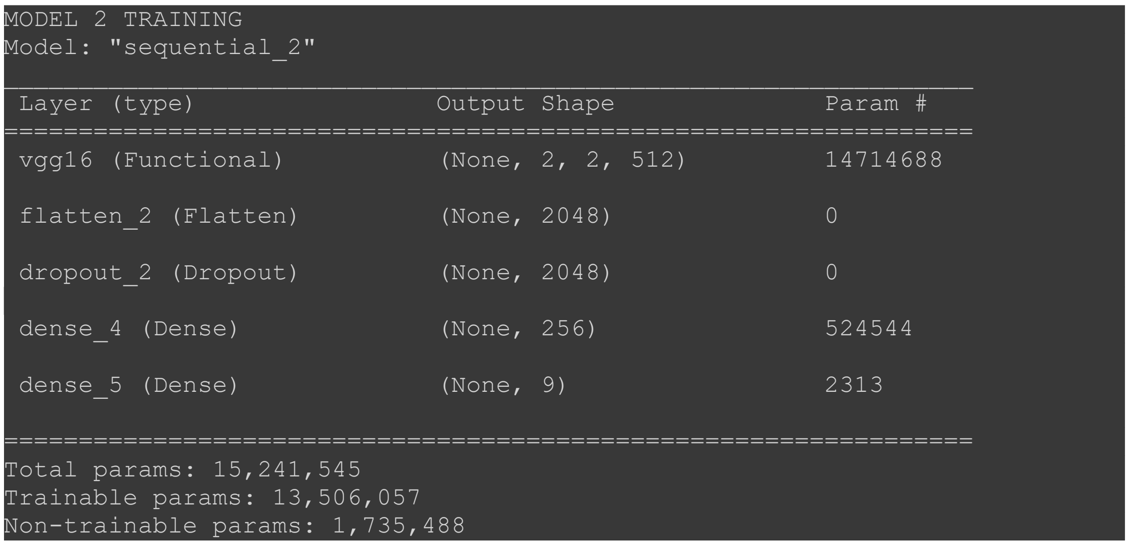
The curves show that the model starts overfitting the training data after 7th epoch, but the curves are getting smoother and better compared to all the previous models with the accuracies and losses of this model as listed below

| Model | Train Loss | Train Accuracy | Val Loss | Val Accuracy |
|---|---|---|---|---|
| VGG model1() | 0.2053 | 0.9317 | 0.22677 | 0.9112 |

Model 2

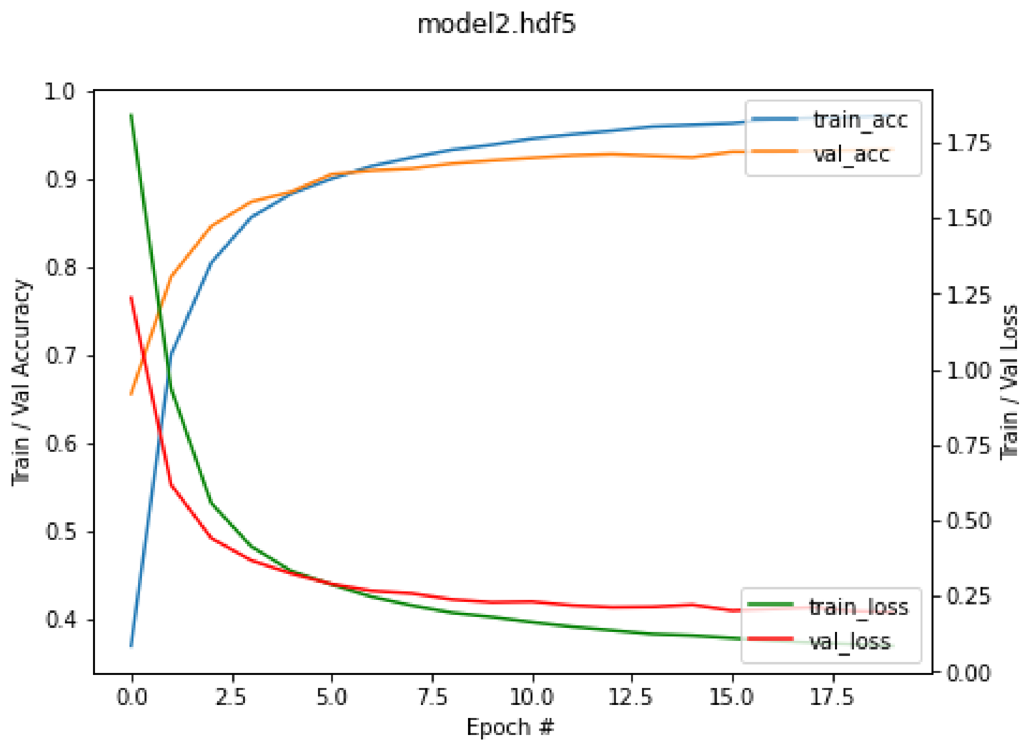
The test and validation accuracies are above 90% which is great, I have created model2() which enables training from block 4 layers, the code is the same as above with trainable parameters set True from block 4

The summary of model2() is shown below, there are ~13 million trainable parameters in model 2



```
MODEL 2 TRAINING
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)           (None, 2, 2, 512)         14714688

 flatten_2 (Flatten)          (None, 2048)              0

 dropout_2 (Dropout)          (None, 2048)              0

 dense_4 (Dense)              (None, 256)               524544

 dense_5 (Dense)              (None, 9)                 2313

=================================================================
Total params: 15,241,545
Trainable params: 13,506,057
Non-trainable params: 1,735,488
```

The learning curve for the model2() is as shown in the figure below



The model accuracies and losses are listed in the table below

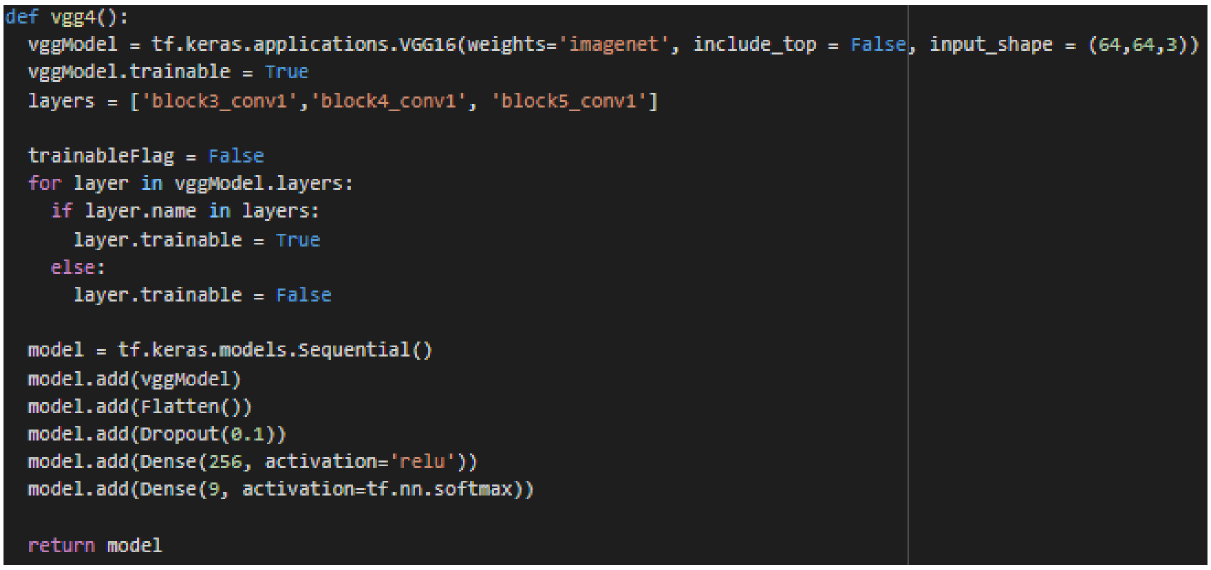
| Model | Train Loss | Train Accuracy | Val Loss | Val Accuracy |
|---|---|---|---|---|
| VGG model2() | 0.0849 | 0.9718 | 0.1966 | 0.9329 |

The losses have dropped significantly and the accuracies for training is 97% which is excellent, the validation loss is very low and the accuracy is 93%. The model seems to start overfitting after 4th epoch, so I checked the accuracies and losses after 4th epoch (checkpoint) but they were not great as shown in the snippet below

```
Epoch 4/20
20/20 [==============================] - ETA: 0s - loss: 0.4142 - accuracy:
0.8561
Epoch 4: val_loss improved from 0.44187 to 0.36793, saving model to
weights.hdf5
```

Model 3

To overcome the overfitting of the models when fine tuning, I experimented making only three different layers of VGG as trainable, the layers selected to be trainable are 'block3_conv1','block4_conv1', 'block5_conv1' as shown in the snippet below

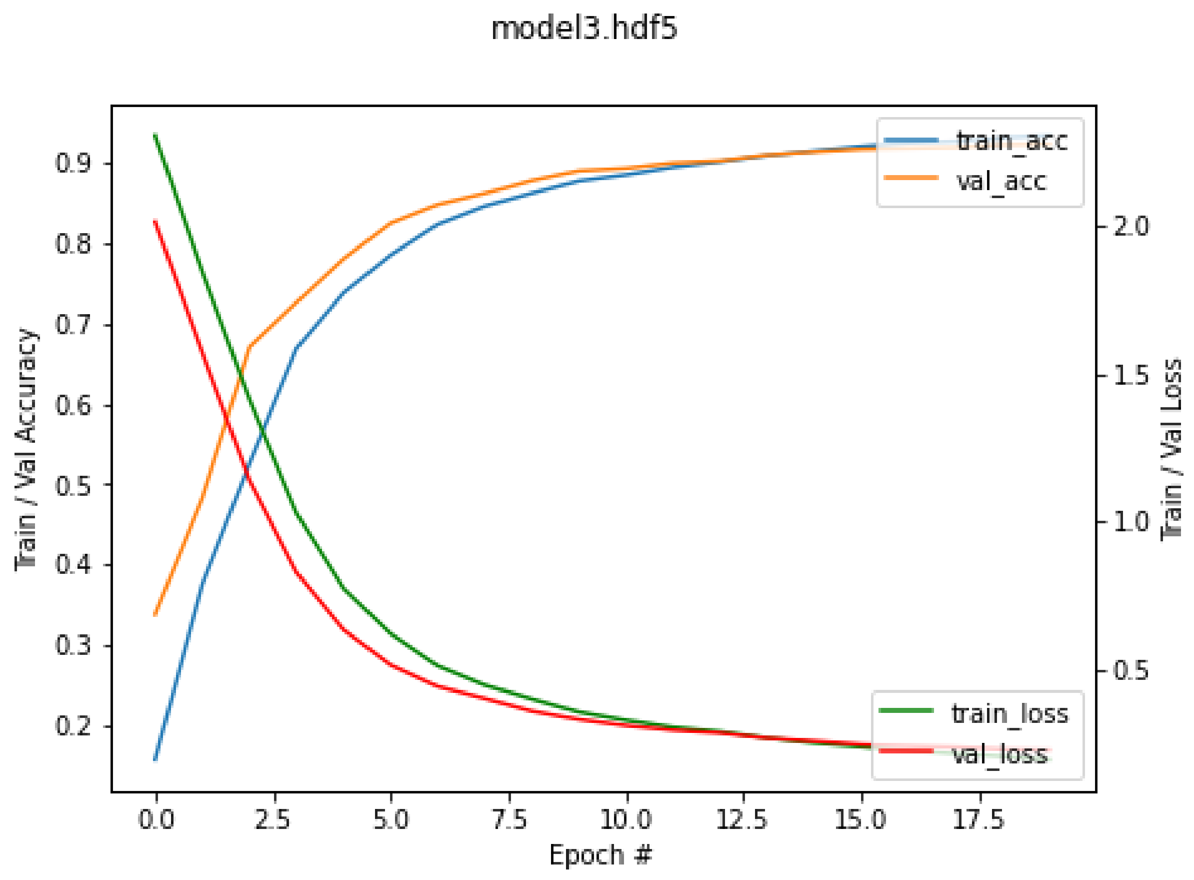


```
def vgg4():
  vggModel = tf.keras.applications.VGG16(weights='imagenet', include_top = False, input_shape = (64,64,3))
  vggModel.trainable = True
  layers = ['block3_conv1','block4_conv1', 'block5_conv1']

  trainableFlag = False
  for layer in vggModel.layers:
    if layer.name in layers:
      layer.trainable = True
    else:
      layer.trainable = False

  model = tf.keras.models.Sequential()
  model.add(vggModel)
  model.add(Flatten())
  model.add(Dropout(0.1))
  model.add(Dense(256, activation='relu'))
  model.add(Dense(9, activation=tf.nn.softmax))

  return model
```

This brings down the trainable parameters from 13 million in model 2() to 4 million in this model as shown in the figure below, which is 1/4th the parameters, this will save execution time and computational cost

```
MODEL 3 TRAINING
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 vgg16 (Functional)          (None, 2, 2, 512)         14714688  
                                                                 
 flatten_3 (Flatten)         (None, 2048)              0         
                                                                 
 dropout_3 (Dropout)         (None, 2048)              0         
                                                                 
 dense_6 (Dense)             (None, 256)               524544    
                                                                 
 dense_7 (Dense)             (None, 9)                 2313      
                                                                 
=================================================================
Total params: 15,241,545
Trainable params: 4,361,993
Non-trainable params: 10,879,552
```

The learning curves for model is shown in the figure below



The model shows excellent result and shown no overfitting even after 20 epochs, the accuracies and losses are listed below, even if they are not as good as the results in model 2, since this model does not overfit on the data, also have 1/4th the trainable parameters, I have assumed this model to be the best performing model compared to all other models

| Model | Train Loss | Train Accuracy | Val Loss | Val Accuracy |
|---|---|---|---|---|
| VGG model3() | 0.1934 | 0.9343 | 0.2265 | 0.9237 |

The Model 3 is the best model in this section of transfer learning and fine tuning. In conclusion, it is good practice to fine tune a model to get good performance out of the network.

# Part C

# Research

## Self-Supervised Learning (SSL)

Yann Lecun, a fellow researcher from Facebook says that the humans learn how the world works by observations as babies (like permanence and gravity) etc. the common sense of a human being is taken for granted, it is somewhat the dark matter of AI. Humans can learn and recognise how a cow looks like after looking at few pics of a cow, also learn to drive with 20-40 hours of driving lessons, whereas AI fails to recognise cows even after training it with millions of images, same with autonomous cars which are trained with thousands of hours of driving data, fail to be as accurate as humans. This is because a human brain can remember stuff, the background knowledge makes it possible for humans to outperform computers in certain tasks. SSL technique is trying to use this background knowledge to match human level performance [9].

SSL is gaining popularity in the recent years with its ability to use the unlabelled data, which is the raw data from any source, this avoids the cost of annotating the data which is very costly and time consuming. It is being used for computer vision, graph learning and natural language processing. SSL is often confused with unsupervised learning since the input data falls in the same domain (no labels), but they are different as discussed below

DNN are data hungry compared to traditional machine learning models. Since the availability of supervised data is low sometimes, the DNN's tend to assume linear relationship to the raw-data and predictions which leads to overfitting since the network gets too confident on small dataset, it fails to generalise. There is large amount of data available since we are living in 2022, big data era. But annotation of this data by humans is becoming very expensive, simple example is scale.ai company that charges ~ 6 dollars per image annotation.

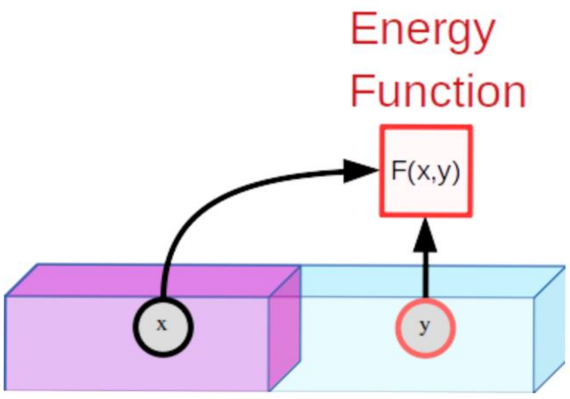
The SSL can be of three types as listed below

- Generative: Input X is encoded to Z by encoder, decoder should decode Z to X
- Contrastive: Encoder encodes X to Z, similarity is measured finally
- Generative-Contrastive: also called adversarial, encoder and decoders come up with fake data and discriminator should differentiate between real and fake data [10].

## Energy Based Model

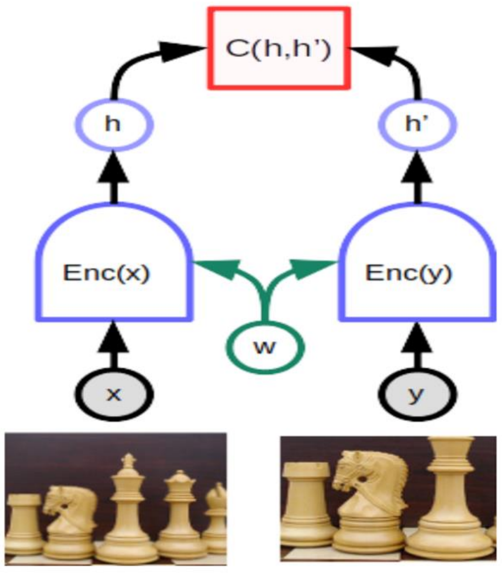
Energy based model (EBM) is a trainable model, given two inputs, it predicts how incompatible they are to each other. The figure below gives a rough idea about an EBM, if the output is low, two inputs are very similar and vice versa. There are two parts

1. Training EBM with x and y which are compatible to produce smaller outputs
2. Ensuring for x, any given y should produce high values when y is not similar and low value when given y is similar



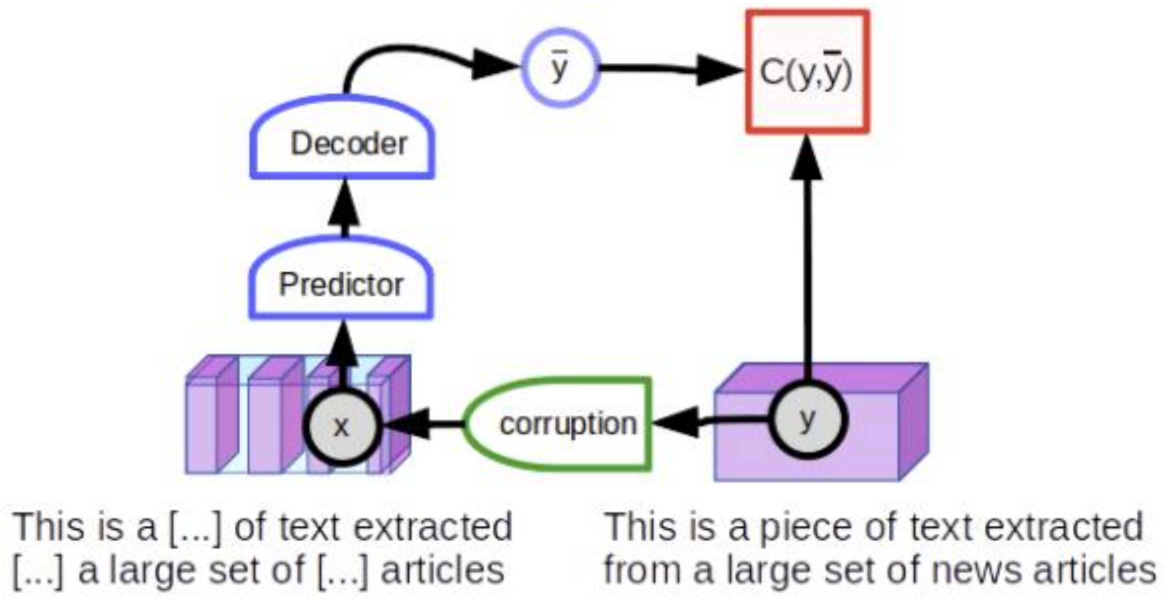
### Siamese Network

This was discussed in the earlier section (one shot learning), basically two identical networks are used, and they are given with x and y each, on the other end, the encoded vectors are collected, the network is trained so that the input x and its distorted version y, when fed into this network, the parameters are adjusted so that the outputs of these networks are expected to move closer (since they are similar). Finally, the similarity score is calculated by the third part or the head of Siamese network (EBM). The simple structure is shown in the figure below



### Contrastive Energy-Based SSL

It is exactly what it sounds like, the x and y inputs given to the networks here are not the same unlike previous techniques and the goal is to tune the output of EBM to be very high since the similarity score should be high for the data which is incompatible. This method is highly useful in natural language processing. The original sentence Y is corrupted first in corruption part of the network, this is our X for the network (also called observation), this corrupted y (x) is fed into DNN to produce the original text y (which is y_predicted), so the original text y will be reconstruct with very low reconstruction error whilst the corrupted text y_ predicted is reconstructed with very high reconstruction error, the simple architecture is as shown below



ȳ
C(y,ȳ)
Decoder
Predictor
x
corruption
y
This is a [...] of text extracted [...] a large set of [...] articles
This is a piece of text extracted from a large set of news articles

This technique cannot be applied for Image and video data since all images cannot be enumerated, so we make use of latent-variable predictive algorithms. These architectures contain an extra variable Z whose value is never observed (hence the name latent). As the latent variable varies over a set, the prediction varies over the possible compatible predictions to input X. An example is GANs, where a discriminator calculated the similarity between X and Y and the generator keeps producing contrastive samples and expects the critic to discriminate with high output energy (since the samples are contrastive, which mean they are incompatible or not similar)

## Non-Contrastive Energy-Based SSL

This is a hot topic of research at present mainly for computer vision tasks. Few successful models are VAE (Variational Auto-Encoders) and sparse modelling, but their uses are currently limited to simple architectures. Devising Non-Contrastive methods for later variable EBMs is a good research area

With this, I conclude my assignment 2 by thanking MTU and Dr. Ted Scully who made it possible for me to gain knowledge in deep learning to a great extent, thank you.

## References

[1] Chlap, P., Min, H., Vandenberg, N., Dowling, J., Holloway, L. and Haworth, A., 2021. A review of medical image data augmentation techniques for deep learning applications. Journal of Medical Imaging and Radiation Oncology, 65(5), pp.545-563.

[2] Shorten, C. and Khoshgoftaar, T.M., 2019. A survey on image data augmentation for deep learning. Journal of big data, 6(1), pp.1-48.

[3] Hiroshi I. Data augmentation by pairing samples for images classifcation. ArXiv e-prints. 2018.

[4] Cecilia S, Michael JD. Improved mixed-example data augmentation. ArXiv preprint. 2018.

[5] Deep Learning Specialization, Andrew Ng, Coursera, https://www.coursera.org/learn/convolutional-neural-networks/lecture/gjckG/one-shot-learning

[6] https://www.youtube.com/watch?v=0iKsimVvfjE

[7] https://www.analyticsvidhya.com/blog/2022/02/classification-without-training-data-zero-shot-learning-approach/

[8] Han, F., Yang, D., Ling, Q.H. and Huang, D.S., 2015, July. A novel diversity-guided ensemble of neural network based on attractive and repulsive particle swarm optimization. In 2015 International Joint Conference on Neural Networks (IJCNN) (pp. 1-7). IEEE.

[9] https://ai.facebook.com/blog/self-supervised-learning-the-dark-matter-of-intelligence/

[10] X. Liu et al., "Self-supervised Learning: Generative or Contrastive," in IEEE Transactions on Knowledge and Data Engineering, doi: 10.1109/TKDE.2021.3090866.