


EE 5841 - Project 5 - Transfer Learning & Augmentation

Submitted by : Ponkrshnan Thiagarajan, Vrushaketu Mali, Ashwini Nikumbh 

Date : 24-Apr-2020

1. Introduction

This Project aims at developing a tutorial for transfer learning and augmentation using keras. This project walks us through the methodology on how augmentation is applied to machine learning data set and shows an example. This project also demonstrates the approach for transfer learning where we start with pre-trained network as feature layer which we use with the output layer of our classification problem. We will be using two different pre-trained networks and compare the performance of both in terms of the results. This project will serve as a tutorial for someone who wants to make use of transfer learning and augmentation in similar project work of image classification. In this project, we have chosen the data set of bird species available on kaggle as Image classification problem.

For the purpose of this project, we have selected first 10 classes out of the 180 classes available in this dataset.

Data can be found here

<https://www.kaggle.com/gpiosenka/100-bird-species> (<https://www.kaggle.com/gpiosenka/100-bird-species>)

```
In [0]: %%capture
import tensorflow as tf          # Import tensorflow open source library
from keras.utils import np_utils # Import np_utils library
import pandas as pd             # Import pandas library
import numpy as np              # Import numpy library
import seaborn as sns           # Import seaborn library
from keras import applications
from keras.models import Sequential # Import sequential model input
from keras.layers import Conv2D, MaxPooling2D # Import 2D convolution layer
from keras.layers import Activation, Dropout, Flatten, Dense # Import activations, dense layers
from keras import optimizers     # Import optimizers available in keras
from google.colab import drive   # For mounting the google drive
drive.mount('/content/drive')
```

1.1 Generate batches of image data (and their labels)

```
In [0]: %%capture
# Creating generators for data processing
from keras.preprocessing.image import ImageDataGenerator, array_to_img, img_to_array, load_img
batch_size = 10 # Specify the batch size for data generator

datagen = ImageDataGenerator(rescale=1./255) # Generate the data by rescaling

train_generator = datagen.flow_from_directory(
    '/content/drive/Shared drives/Machine Learning Project/Project-5/100-bird-species/train',
    target_size=(224, 224), # Generate training data batches with size 224,224 from specieid path
    batch_size=batch_size, # as per specified batch size
    shuffle = 0, # do not shuffle the data
    class_mode='categorical') # make data type as categorical

validation_generator = datagen.flow_from_directory(
    '/content/drive/Shared drives/Machine Learning Project/Project-5/100-bird-species/valid',
    target_size=(224, 224), # Generate validation data batches with size 224,224 from specieid path
    batch_size=batch_size, # as per specified batch size
    shuffle = 0, # do not shuffle the data
    class_mode='categorical') # make data type as categorical

test_generator = datagen.flow_from_directory(
    '/content/drive/Shared drives/Machine Learning Project/Project-5/100-bird-species/test',
    target_size=(224, 224), # Generate testing data batches with size 224,224 from specieid path
    shuffle = 0, # do not shuffle the data
    batch_size=1) # Generate single batch for whole test data
```

2. Data Augmentation

2.1 What is data Augmentation?

Data augmentation is a method in which we artificially increase the size of our training data set by modifying the images in our training data set with different random transformation such as rotation, zooming, whitening, flipping and so on.

2.2 Why data Augmentation?

The training accuracy of the deep neural network varies greatly with the variations of the images in our dataset and this is where the data augmentation helps to create variety of images through random modification. Thus, given plethora of the data, we can get our neural network model as more skillful.

2.3 Different transformations for data augmentation

Now, let's discuss what are different transformation techniques for augmentation. >

Rotation : To rotate the image in the given rotation range.

Shift : To move the image in horizontal and vertical direction as per width and height shift parameters.

Rescale : Scaling the image as per the input scaling factor.

Flip : Flip the image either vertically or horizontally.

Featurewise_std_normalization : This divides the input by standard deviation of entire data set

Samplewise_std_normalization : This divides the input by its own standard deviation.

2.4 Augmenting the data set

Now, we will show an example of how to augment the data set, Thus, we will develop a function for augmentation which will take different image manipulations as its input arguments. Different image manipulations can be rotating, shifting, flipping, rescaling and so on as discussed above and finally we will see some results of augmented data set.

We will also code small function for displaying the augmented image so that we can use that function to display each manipulated image.

```

In [0]: def Augment(Input, rotation=False, rescale = False, shear = False, shifts = False, zoom = False, flip = False): # define augmentation function
        if rotation: # if rotation is true
            aug_datagen = ImageDataGenerator(rotation_range = -30) # rotate the image by specified angle
            x = img_to_array(Input) # convert image to array
            x = x.reshape((1,) + x.shape) # reshape the array to (1, 3, 224, 224)
            i = 0
            for batch in aug_datagen.flow(x, batch_size=1, save_to_dir='preview/rotate', save_format='jpeg'): # Save to this directory with jpeg image format
                i += 1
                if i > 1: # if number of the images is 2
                    break # stop the generator

        if shifts: # if shifts is true
            aug_datagen = ImageDataGenerator(width_shift_range=-0.3, height_shift_range=-0.3) # Shift the image by specified dimensions along width and height
            x = img_to_array(Input) # convert image to array
            x = x.reshape((1,) + x.shape) # reshape the array to (1, 3, 224, 224)
            i = 0
            for batch in aug_datagen.flow(x, batch_size=1, save_to_dir='preview/shifts', save_format='jpeg'): # Save to this directory with jpeg image format
                i += 1
                if i > 1: # if number of the images is 2
                    break # stop the generator

        if zoom: # if zoom is true
            aug_datagen = ImageDataGenerator(zoom_range=0.5) # zoom the image by specified zoom factor
            x = img_to_array(Input) # convert image to array
            x = x.reshape((1,) + x.shape) # reshape the array to (1, 3, 224, 224)
            i = 0
            for batch in aug_datagen.flow(x, batch_size=1, save_to_dir='preview/zoom', save_format='jpeg'): # Save to this directory with jpeg image format
                i += 1
                if i > 1: # if number of the images is 2
                    break # stop the generator

        if flip: # if flip is true
            aug_datagen = ImageDataGenerator(horizontal_flip=True) # flip the image horizontally
            x = img_to_array(Input) # convert image to array
            x = x.reshape((1,) + x.shape) # reshape the array to (1, 3, 224, 224)
            i = 0
            for batch in aug_datagen.flow(x, batch_size=1, save_to_dir='preview/flip', save_format='jpeg'):
                i += 1
                if i > 1: # if number of the images is 2
                    break # stop the generator

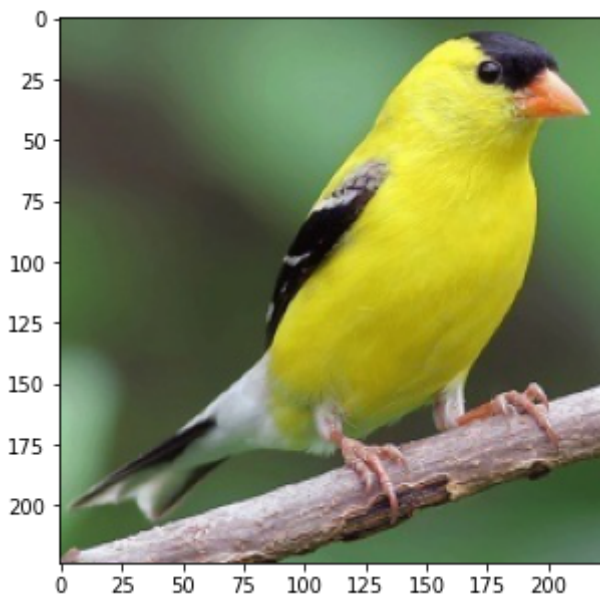
def display(Input): # define function for displaying the image
    import matplotlib.pyplot as plt
    f = plt.figure(figsize=(5,5)) # plot figure with size 5 x 5
    plt.imshow(Input) # plot the image
    plt.show # show the plot

```

2.4.1 Original Image

In below code, we have extracted a random image from the training data set and it is plotted so that we can compare it with different manipulation of its own.

```
In [0]: # Load the original image
image = load_img('/content/drive/Shared drives/Machine Learning Project/Project-5/100-bird-species/train/AMERICAN GOLD FINCH/072.jpg')
display(image) # display the original image
```



2.4.2 Augment the original Image

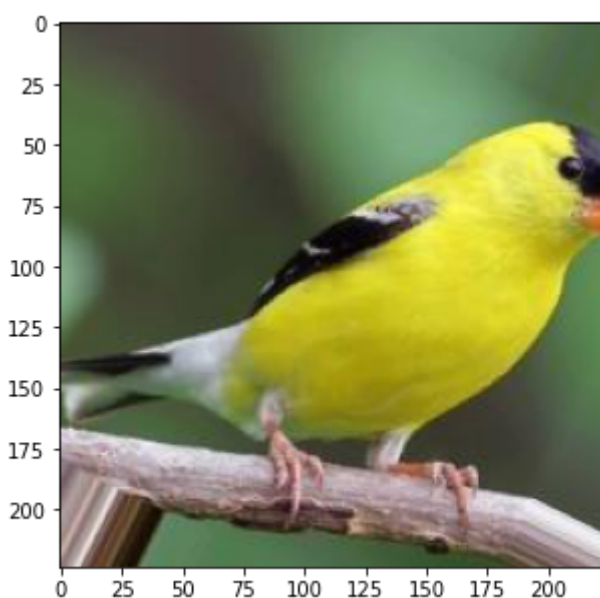
In below code, we will just run the augmentation function with specific augmentation input arguments such as rotation, flipping, shifting and so on.

```
In [0]: Augment(image, rotation=True) # Rotate the image
Augment(image, rescale=True) # Rescale the image
Augment(image, shifts=True) # Shift the image
Augment(image, zoom=True) # Zoom the image
Augment(image, flip=True) # Flip the image
```

2.4.2 Image rotation

Now, we will load the rotated image and display it.

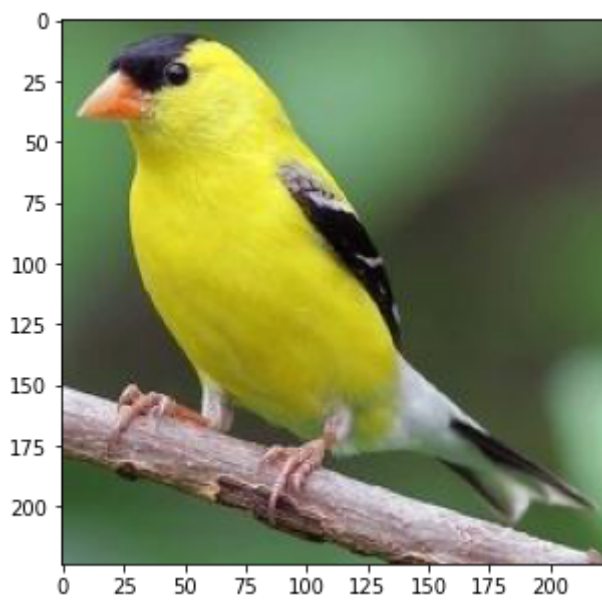
```
In [0]: aug_image = load_img('/content/preview/rotate/_0_9653.jpeg') # Load the rotated image
display(aug_image) # display the rotated image
```



2.4.3 Image flipping

Now, we will load the flipped image and display it.

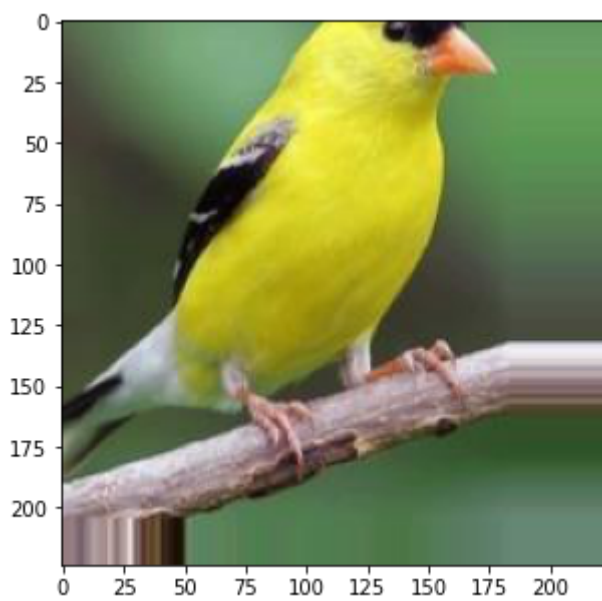
```
In [0]: aug_image = load_img('/content/preview/flip/_0_2037.jpeg') # Load the flipped image
display(aug_image) # display the flipped image
```



2.4.4 Image shifting

Now, we will load the shifted image and display it.

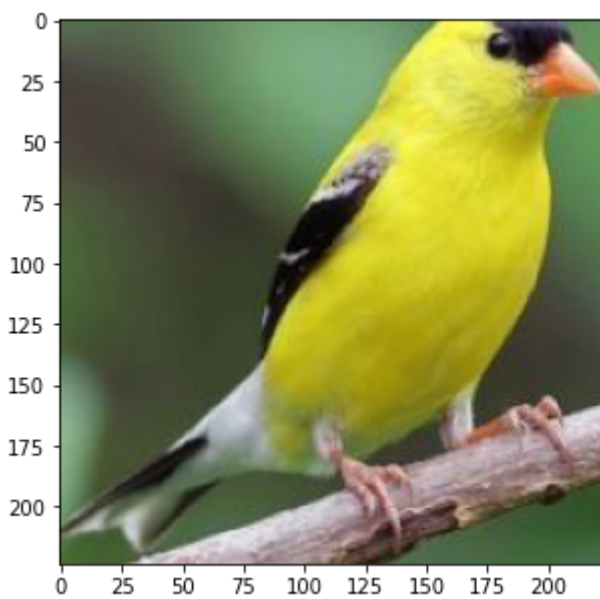
```
In [0]: aug_image = load_img('/content/preview/shifts/_0_2961.jpeg') # Load the shifted image
display(aug_image) # display the shifted image
```



2.4.5 Image zooming

Now, we will load the zoomed image and display it.

```
In [0]: aug_image = load_img('/content/preview/zoom/_0_5712.jpeg') # Load the zoomed image
display(aug_image) # display the zoomed image
```



2.5 Conclusion

Thus, above code and results show us how to perform the augmentation on images. There is no unique way to write the code for augmentation. We demonstrated one way of doing the augmentation.

The intention of showing this technique is to learn how can we artificially expand the size of the data set as it is common knowledge that the more data an ML algorithm has access to, the more effective it can be.

We will make the use of augmented data set in further transfer learning technique and will see how it helps in improving the image classification accuracy.

3. Transfer learning

3.1 What is Transfer learning?

Transfer learning is a method where we re-use a pre-trained model on a new related problem. The knowledge of already trained machine learning model is applied to different related problem. So, basically we use the knowledge that is gained by model during training is used and with that it means that we transfer the weights which network has learned at "task A" to a new "task B."

3.2 Why Transfer learning?

Instead of starting the learning process from scratch, we start with patterns learned by a network while solving a related task in past. Often, a lot of data is needed to train a neural network from scratch and most of the times, it is not possible to get access to large amount of data. Additionally, training a neural network from scratch takes lot of training time. With transfer learning, a machine learning model can be built with comparatively lesser training data because the model is already pre-trained.

3.3 Pre-trained models for transfer learning

Now, let's take a look at what different pre-trained models are available in keras which can be potentially used for fine tuning our neural network. Below mentioned are the some of the available pre-trained models in Keras for image classification.

Xception, VGG16, VGG19, MobileNet, DenseNet, NasNet

In our project, we will experiment transfer learning on the data set of bird species with two pre-trained models.

A) VGG16

B) VGG19

3.4 Using VGG16 pre-trained model

3.4.1 Task 1

3.4.1.1 Running training data through feature layers of VGG 16 pre-trained Network Architecture

Below mentioned code shows how we run the training data from our data set on the VGG16 pre-trained model. We are implementing this so that we can use the output of this code to train our own output layers which we are going to create from scratch.

```
In [0]: %%capture
# Pretrained VGG-16 model
vgg_model = applications.VGG16(include_top=False, weights='imagenet', input_shape = (224,224,3))
# Running pre trained VGG-16 on training data - Extracting features
vgg_train = vgg_model.predict_generator(train_generator, verbose=1)
# Changing Labels into categorical labels
vgg_train_labels = pd.get_dummies(pd.Series(train_generator.classes))
# Extracting features for validation data
vgg_validation = vgg_model.predict_generator(validation_generator, verbose=1)
vgg_validation_labels = pd.get_dummies(pd.Series(validation_generator.classes))
```

3.4.1.2 Training output layers from scratch using saved feature data of VGG-16

Below mentioned code shows how we have created our own output layer from the scratch. The reason it is said as output layer because it will be connected to VGG16 pretrained network on its top side in next task. We are adding two hidden dense layers and an output dense layers of 10 neurons since we have selected 10 categories of bird species for image classification. The activation function selected in the output layer is the sigmoid which determines the maximum probability of the feature belonging to a particular class.

We then train these layers using the VGG16 features we get in previous step. After training, our neural network model will learn certain parameters and generate the weights of each feature which we are going to save for later use.

```
In [0]: # Training dense Layer with the output of VGG-16 (without top Layer)
model = Sequential() # Use the sequential API to add the Layers
model.add(Flatten(input_shape=vgg_train.shape[1:])) # Flatten the VGG training Layer
model.add(Dense(300)) # add hidden dense Layer of 300 neurons
model.add(Activation('relu')) # make the activation function as rectified Linear Unit
model.add(Dense(200)) # add another hidden dense Layer of 200 neurons
model.add(Activation('relu')) # make the activation function as rectified Linear Unit
model.add(Dropout(0.3)) # add a new drop out Layer with drop out as 30%
model.add(Dense(10)) # add output dense Layer of 10 neurons
model.add(Activation('sigmoid')) # output activation function as sigmoid

# Compiling the dense layer
# use RMSprop optimizer
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# Training the dense layer
# fit the model on output layers using feature data of VGG 16
# set iteration as 8
# Use validation feature data of VGG16
Network_1_1=model.fit(vgg_train, vgg_train_labels,
                      epochs=8,
                      batch_size=batch_size,
                      validation_data=(vgg_validation, vgg_validation_labels))

model.save_weights('weights.h5') # save the weights after training
```

Train on 1504 samples, validate on 50 samples

```
Epoch 1/8
1504/1504 [=====] - 1s 792us/step - loss: 2.5573 - accuracy: 0.1722 - val_loss: 1.9951 - val
_accuracy: 0.4400
Epoch 2/8
1504/1504 [=====] - 1s 737us/step - loss: 1.6689 - accuracy: 0.4900 - val_loss: 0.5461 - val
_accuracy: 0.8400
Epoch 3/8
1504/1504 [=====] - 1s 700us/step - loss: 0.7173 - accuracy: 0.7819 - val_loss: 0.4196 - val
_accuracy: 0.9600
Epoch 4/8
1504/1504 [=====] - 1s 680us/step - loss: 0.2938 - accuracy: 0.9082 - val_loss: 0.9950 - val
_accuracy: 0.7800
Epoch 5/8
1504/1504 [=====] - 1s 690us/step - loss: 0.1945 - accuracy: 0.9448 - val_loss: 0.7955 - val
_accuracy: 0.8600
Epoch 6/8
1504/1504 [=====] - 1s 661us/step - loss: 0.2063 - accuracy: 0.9555 - val_loss: 0.6236 - val
_accuracy: 0.8800
Epoch 7/8
1504/1504 [=====] - 1s 668us/step - loss: 0.0889 - accuracy: 0.9787 - val_loss: 0.6308 - val
_accuracy: 0.9000
Epoch 8/8
1504/1504 [=====] - 1s 665us/step - loss: 0.1305 - accuracy: 0.9747 - val_loss: 0.2434 - val
_accuracy: 0.9600
```

Comments

Looking at the results of iterations above, we can say that after 8 iterations our training and validation accuracy has pretty much good and is converged. We will then plot these accuracies vs iterations.

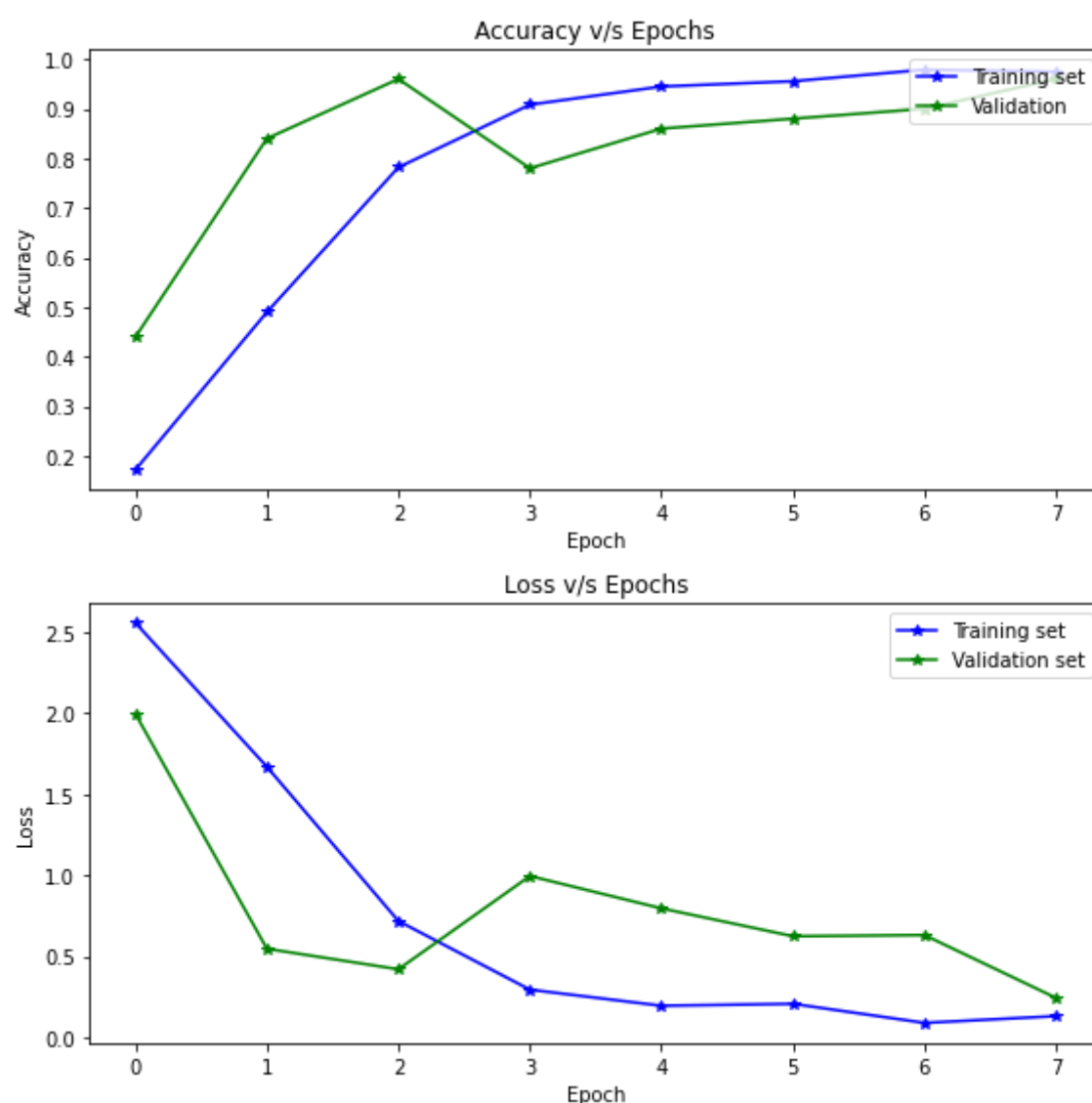
Below mentioned code is developed for plotting those accuracies.


```
In [0]: import numpy as np
import matplotlib.pyplot as plt

# Code for plotting the training and validation accuracy variation with Iterations
fig = plt.figure(figsize=(8,8)) # Figure size
plt.subplot(2,1,1) #First plot in subplot of 2
plt.plot(Network_1_1.history['accuracy'],'b*-') # Plot the training accuracy of network
plt.plot(Network_1_1.history['val_accuracy'],'g*-') # Plot the validation accuracy of network
plt.title('Accuracy v/s Epochs') # Title
plt.ylabel('Accuracy') #Y-Label
plt.xlabel('Epoch') # X-Label
plt.legend(['Training set', 'Validation'], loc='upper right') # Legend

# Code for plotting the training and validation Loss variation with Iterations
plt.subplot(2,1,2) #second plot in subplot of 2
plt.plot(Network_1_1.history['loss'],'b*-') # Plot the training Loss of network
plt.plot(Network_1_1.history['val_loss'],'g*-') # Plot the validation Loss of network
plt.title('Loss v/s Epochs') # Title
plt.ylabel('Loss') #Y-Label
plt.xlabel('Epoch') # X-Label
plt.legend(['Training set', 'Validation set'], loc='upper right') # Legend

plt.tight_layout()
```



Comments

The first figure in above plots show the training and validation accuracy vs number of iterations. The second figure shows the training and validation loss vs number of iterations. From the figure, it can be inferred that the solution has been converged and thus the model is trained appropriately. Now, let's move to next task.

3.4.2 Task 2

In this task, we are going to work on three steps. These three steps are as below:

1. Augment the training and validation data
2. Combine VGG16 feature layers and our own output layer.
3. Training the output layer only by passing augmented data set

Let's go through each step by developing required codes.

3.4.2.1 Augmenting the features data

Below mentioned code augments the training and validation data which we are going to use for training our output layer. The different image manipulations used here are the rescaling, rotating, shearing, zooming & flipping. These all manipulations are combined and then our dataset is augmented. Please note that we are not going to load the augmented data any particular directory therefore we have not made the use of earlier developed augmentation function.

```
In [0]: %%capture
# Augmenting data with a new datagenerator
aug_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range = 5,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=True) # Augment the data with given input arguments

# Augmenting training data which is stored in specified path
train_generator_aug = aug_datagen.flow_from_directory(
    '/content/drive/Shared drives/Machine Learning Project/Project-5/100-bird-species/train',
    target_size=(224, 224),
    batch_size=batch_size,
    shuffle = 0,
    class_mode='categorical') # Augment the training data

# Augmenting validation data which is stored in specified path
validation_generator_aug = aug_datagen.flow_from_directory(
    '/content/drive/Shared drives/Machine Learning Project/Project-5/100-bird-species/valid',
    target_size=(224, 224),
    batch_size=batch_size,
    shuffle = 0,
    class_mode='categorical') # Augment the validation data
```

3.4.2.2 Combining feature layers and output layers into one network & freezing the feature layers

Below code explains the procedure of creating our own output layer. The procedure is same as we used in previous task for creating the output layer. We are creating same layers and we are loading the weights which we have generated by running the VGG16 features data in previous task.

After that, we will be freezing the feature layers of VGG16 because we want to train our model only on the output layers. So the number of trainable layers will be equal to the number of layers in the network which we have created from the scratch.

```
In [0]: # Comibning pretrained VGG-16 with the dense Layer
tune_vgg = applications.VGG16(include_top=False, weights='imagenet',input_shape = (224,224,3))
# Include VGG16 layer by cutting its top Layer

dense_model = Sequential() # Use the sequential API to add the layers
dense_model.add(Flatten(input_shape=tune_vgg.output_shape[1:])) # Flatten the VGG Layers
dense_model.add(Dense(300)) # add hidden dense Layer of 300 neurons
dense_model.add(Activation('relu')) # make the activation function as rectified Linear unit
dense_model.add(Dense(200)) # add hidden dense Layer of 200 neurons
dense_model.add(Activation('relu')) # make the activation function as rectified Linear unit
dense_model.add(Dropout(0.3)) # add a new drop out Layer with drop out as 30%
dense_model.add(Dense(10)) # add output dense Layer of 10 neurons
dense_model.add(Activation('sigmoid')) # Use sigmoid as the output activation function

# Loading the saved weights of dense Layer from previous task
dense_model.load_weights('weights.h5')

tune_model = Sequential() # Use the sequential API to add the layers
tune_model.add(tune_vgg) # add complete VGG 16 pre-trained network
tune_model.add(dense_model) # add the output layers created from scratch

# Freezing the layers of VGG-16
for layer in tune_vgg.layers[:]:
    layer.trainable = False # freeze all the layers of VGG 16 network
tune_model.summary() # show the summary of the model
```

Model: "sequential_10"

Layer (type)	Output Shape	Param #
vgg16 (Model)	(None, 7, 7, 512)	14714688
sequential_9 (Sequential)	(None, 10)	7588910

=====
Total params: 22,303,598
Trainable params: 7,588,910
Non-trainable params: 14,714,688
=====

Comments

If we look at the model summary above, it shows that out of total 22,303,598 parameters, only 7,588,910 parameters are trainable which are the layers we have created from scratch. This means that we have successfully freezed the feature layers of VGG16.

3.4.2.3 Training the output layers using augmented data set

In last step now, we have combined both VGG feature layers and our output layers. Nowm we will compile the model and train our output layers by passing the augmented data.

```
In [0]: # Compiling and training the network
tune_model.compile(loss='categorical_crossentropy', # Compile the full network model
                  optimizer=optimizers.Adam(learning_rate=1e-5,clipnorm=1), # use Adam optimizer
                  metrics=['accuracy'])
Network_2_1=tune_model.fit_generator(train_generator_aug, # Fit the model on augmented training and validation data
                                   epochs=1,
                                   verbose=1,
                                   validation_data=validation_generator_aug)
```

Epoch 1/1
151/151 [=====] - 21s 142ms/step - loss: 0.1427 - accuracy: 0.9674 - val_loss: 0.0339 - val_accuracy: 0.9600

Comments

Looking at the training and validation accuracy, we can say that the model is trained with almost same training and validation accuracy as in task 1. Now, let's move to task 3.

3.4.3 Task 3

In this task, we are going to fine tune the full network and use that network to establish the training, testing and validation accuracy.

3.4.3.1 Fine tuning the full network

Fine tuning full network means we will make all feature layers of VGG16 as trainable. There won't be any non-trainable parameters. After that, we will pass the augmented data set on full connected network in order to fine tune the weights.

```
In [0]: # Unfreezing the VGG-16 layers
for layer in tune_vgg.layers[:]:
    layer.trainable = True # Make all layers of VGG 16 as trainable (Unfreeze)
tune_model.summary() # show the summary of model

# Compiling and training the full network
# use adam optimizer
tune_model.compile(loss='categorical_crossentropy',
                   optimizer=optimizers.Adam(learning_rate=1e-5,clipnorm=1),
                   metrics=['accuracy'])

# Fit the model on augmented training and validation data
Network_3_1=tune_model.fit_generator(train_generator_aug,
                                     epochs=1,
                                     verbose=1,
                                     validation_data=validation_generator_aug)
```

Model: "sequential_10"

Layer (type)	Output Shape	Param #
=====	=====	=====
vgg16 (Model)	(None, 7, 7, 512)	14714688
sequential_9 (Sequential)	(None, 10)	7588910
=====	=====	=====
Total params: 22,303,598		
Trainable params: 22,303,598		
Non-trainable params: 0		

Epoch 1/1

151/151 [=====] - 30s 201ms/step - loss: 0.1253 - accuracy: 0.9787 - val_loss: 0.2073 - val_accuracy: 0.9400

Comments

If we look at the model summary above, it shows that out of total 22,303,598 parameters, none of the parameters are non-trainable This means that we have successfully unfreezed the feature layers of VGG16.

3.4.3.2 Evaluate the model on testing data

Now, as we have fine tuned the weights and parameters by training the full connected network, lets use this network to establish the test accuracy. That means, we will evaluate this model on test data where it will generate the predictions of test features, compare those predicted labels with actual labels of test data and report the accuracy.

```
In [0]: test_loss_16, test_acc_16 = tune_model.evaluate(test_generator) # Evaluate the model performance on test data set
print('The testing accuracy is ', (test_acc_16*100).round(2), '%') # Print the testing accuracy of built model
```

```
50/50 [=====] - 1s 23ms/step
The testing accuracy is 96.0 %
```

Comments

The testing accuracy we have got with fine tuning of full network is 96% which is good. By this, we can infer that the optimization parameters, number of neurons in dense layers we have selected are appropriate enough for good image classification.

3.5 Using VGG19 pre-trained model**3.5.1 Task 1****3.5.1.1 Running training data through feature layers of VGG 16 pre-trained Network Architecture**

Below mentioned code shows how we run the training data from our data set on the VGG19 pre-trained model. We are implementing this so that we can use the output of this code to train our own output layers which we are going to create from scratch.

```
In [0]: %%capture
# Pretrained VGG-19 model
VGG19_model = applications.vgg19.VGG19(include_top=False, weights='imagenet', input_shape = (224,224,3))
# Running pre trained VGG-19 on training data - Extracting features
VGG19_train = VGG19_model.predict_generator(train_generator, verbose=1)
# Changing Labels into categorical labels
VGG19_train_labels = pd.get_dummies(pd.Series(train_generator.classes))
# Extracting features for Validation data
VGG19_validation = VGG19_model.predict_generator(validation_generator, verbose=1)
VGG19_validation_labels = pd.get_dummies(pd.Series(validation_generator.classes))
```

3.5.1.2 Training output layers from scratch using saved feature data of VGG19

Below mentioned code shows how we have created our own output layer from the scratch. The reason it is said as output layer because it will be connected to VGG16 pretrained network on its top side in next task. We are adding two hidden dense layers and an output dense layers of 10 neurons since we have selected 10 categories of bird species for image classification. The activation function selected in the output layer is the sigmoid which determines the maximum probability of the feature belonging to a particular class.

We then train these layers using the VGG19 features we get in previous step. After training, our neural network model will learn certain parameters and generate the weights of each feature which we are going to save for later use.

```
In [0]: # Training dense layer with the output of VGG-19 (without top layer)
model_2 = Sequential() # Use the sequential API to add the layers
model_2.add(Flatten(input_shape=VGG19_train.shape[1:])) # Flatten the VGG training layer
model_2.add(Dense(300)) # add hidden dense layer of 300 neurons
model_2.add(Activation('relu')) # make the activation function as rectified linear Unit
model_2.add(Dense(200)) # add another hidden dense layer of 200 neurons
model_2.add(Activation('relu')) # make the activation function as rectified linear Unit
model_2.add(Dropout(0.3)) # add a new drop out layer with drop out as 30%
model_2.add(Dense(10)) # add output dense layer of 10 neurons
model_2.add(Activation('sigmoid')) # output activation function as sigmoid

# Compiling the dense layer
# use RMSprop optimizer
model_2.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])

# Training the dense layer
# fit the model on output layers using feature data of VGG19
# set iteration as 8
Network_1_2=model_2.fit(VGG19_train, VGG19_train_labels,
                        epochs=8,
                        batch_size=batch_size,
                        validation_data=(VGG19_validation, VGG19_validation_labels)) # Use validation feature data of VGG19

model_2.save_weights('weights_2.h5') # save the weights after training
```

Train on 1504 samples, validate on 50 samples

```
Epoch 1/8
1504/1504 [=====] - 1s 764us/step - loss: 2.7098 - accuracy: 0.1190 - val_loss: 2.3363 - val
_accuracy: 0.1400
Epoch 2/8
1504/1504 [=====] - 1s 671us/step - loss: 2.4021 - accuracy: 0.2234 - val_loss: 2.0014 - val
_accuracy: 0.3200
Epoch 3/8
1504/1504 [=====] - 1s 675us/step - loss: 1.7074 - accuracy: 0.4781 - val_loss: 1.0395 - val
_accuracy: 0.6600
Epoch 4/8
1504/1504 [=====] - 1s 670us/step - loss: 0.8137 - accuracy: 0.7606 - val_loss: 1.0492 - val
_accuracy: 0.6600
Epoch 5/8
1504/1504 [=====] - 1s 674us/step - loss: 0.4160 - accuracy: 0.8783 - val_loss: 1.2450 - val
_accuracy: 0.7800
Epoch 6/8
1504/1504 [=====] - 1s 672us/step - loss: 0.2599 - accuracy: 0.9249 - val_loss: 0.4804 - val
_accuracy: 0.9000
Epoch 7/8
1504/1504 [=====] - 1s 672us/step - loss: 0.1699 - accuracy: 0.9468 - val_loss: 0.5670 - val
_accuracy: 0.9000
Epoch 8/8
1504/1504 [=====] - 1s 670us/step - loss: 0.1214 - accuracy: 0.9688 - val_loss: 0.6909 - val
_accuracy: 0.9000
```


Comments

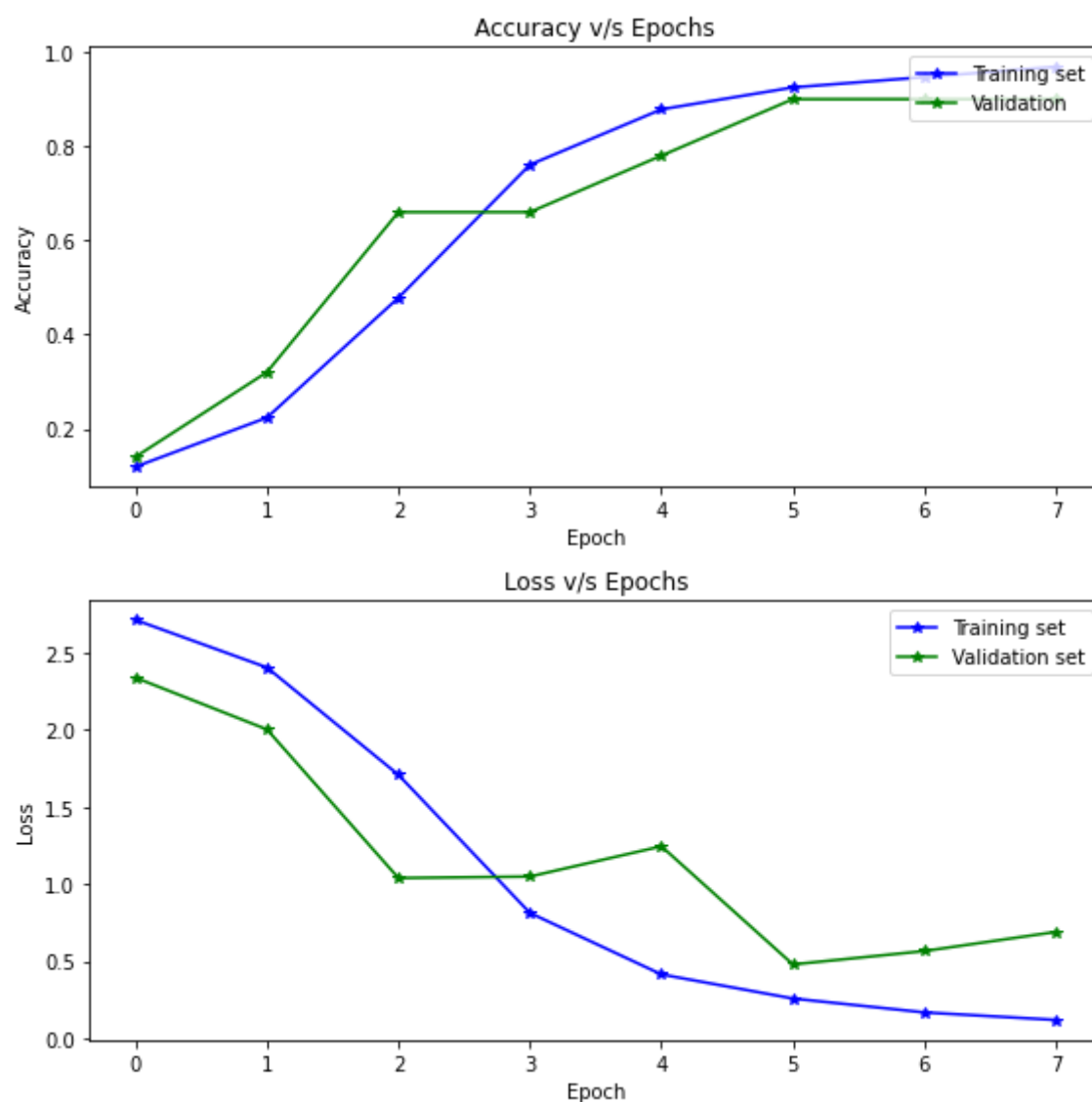
Looking at the results of iterations above, we can say that after 8 iterations our training and validation accuracy has pretty much good and is converged. We will then plot these accuracies vs iterations.

Below mentioned code is developed for plotting those accuracies.

```
In [0]: # Code for plotting the training and validation accuracy variation with Iterations
fig = plt.figure(figsize=(8,8)) # Figure size
plt.subplot(2,1,1) #First plot in subplot of 2
plt.plot(Network_1_2.history['accuracy'],'b*-') # Plot the training accuracy of network
plt.plot(Network_1_2.history['val_accuracy'],'g*-') # Plot the validation accuracy of network
plt.title('Accuracy v/s Epochs') # Title
plt.ylabel('Accuracy') #Y-Label
plt.xlabel('Epoch') # X-Label
plt.legend(['Training set', 'Validation'], loc='upper right') # Legend

# Code for plotting the training and validation Loss variation with Iterations
plt.subplot(2,1,2) #second plot in subplot of 2
plt.plot(Network_1_2.history['loss'],'b*-') # Plot the training Loss of network
plt.plot(Network_1_2.history['val_loss'],'g*-') # Plot the validation Loss of network
plt.title('Loss v/s Epochs') # Title
plt.ylabel('Loss') #Y-Label
plt.xlabel('Epoch') # X-Label
plt.legend(['Training set', 'Validation set'], loc='upper right') # Legend

plt.tight_layout()
```

**Comments**

The first figure in above plots show the training and validation accuracy vs number of iterations. The second figure shows the training and validation loss vs number of iterations. From the figure, it can be inferred that the solution has been converged and thus the model is trained appropriately. Now, let's move to next task.

3.5.2 Task 2

In this task, we are going to work on three steps. These three steps are as below:

1. Combining feature layers and output layers into one network
2. freezing the feature layers
3. Training the output layer only by passing augmented data set

Let's go through each step by developing required codes.

3.5.2.1 Combining feature layers and output layers into one network

Below code explains the procedure of creating our own output layer. The procedure is same as we used in previous task for creating the output layer. We are creating same layers and we are loading the weights which we have generated by running the VGG19 features data in previous task.

```
In [0]: # Combining pretrained VGG-19 with the dense layer
tune_VGG19 = applications.vgg19.VGG19(include_top=False, weights='imagenet', input_shape = (224,224,3))
# Include VGG19 layer by cutting its top layer

# Dense Layer
dense_model_2 = Sequential() # Use the sequential API to add the layers
dense_model_2.add(Flatten(input_shape=tune_VGG19.output_shape[1:])) # Flatten the VGG layers
dense_model_2.add(Dense(300)) # add hidden dense layer of 300 neurons
dense_model_2.add(Activation('relu')) # make the activation function as rectified linear unit
dense_model_2.add(Dense(200)) # add hidden dense layer of 200 neurons
dense_model_2.add(Activation('relu')) # make the activation function as rectified linear unit
dense_model_2.add(Dropout(0.6)) # add a new drop out layer with drop out as 60%
dense_model_2.add(Dense(10)) # add output dense layer of 10 neurons
dense_model_2.add(Activation('sigmoid')) # Use sigmoid as the output activation function

# Loading the saved weights of dense layer from previous task
dense_model_2.load_weights('weights_2.h5')

# Combining VGG-19 and the dense layer
tune_model_2 = Sequential() # Use the sequential API to add the layers
tune_model_2.add(tune_VGG19) # add complete VGG 19 pre-trained network
tune_model_2.add(dense_model_2) # add the output layers created from scratch
```

3.5.2.2 Freezing the feature layers of VGG19

After that, we will be freezing the feature layers of VGG19 because we want to train our model only on the output layers. So the number of trainable layers will be equal to the number of layers in the network which we have created from the scratch.

```
In [0]: # Freezing VGG-19 Layers.
for layer in tune_VGG19.layers[:]:
    layer.trainable = False # freeze all the layers of VGG19 network
tune_model_2.summary() # show the summary of the model
```

Model: "sequential_26"

Layer (type)	Output Shape	Param #
=====	=====	=====
vgg19 (Model)	(None, 7, 7, 512)	20024384
sequential_25 (Sequential)	(None, 10)	7588910
=====	=====	=====
Total params: 27,613,294		
Trainable params: 7,588,910		
Non-trainable params: 20,024,384		

Comments

If we look at the model summary above, it shows that out of total 27,613,294 parameters, only 7,588,910 parameters are trainable which are the layers we have created from scratch. This means that we have successfully frozen the feature layers of VGG19.

3.5.2.3 Training the output layers using augmented data set

In last step now, we have combined both VGG feature layers and our output layers. Now we will compile the model and train our output layers by passing the augmented data. We are not augmenting the data again as the data augmented during use of VGG16 model can be used again for this task.

```
In [0]: # Compiling and training the network
# use Adam optimizer
tune_model_2.compile(loss='categorical_crossentropy',
                    optimizer=optimizers.Adam(learning_rate=1e-5, clipnorm=1),
                    metrics=['accuracy'])

# Fit the model on augmented training and validation data
Network_2_2=tune_model_2.fit_generator(train_generator_aug,
    epochs=1,
    verbose=1,
    validation_data=validation_generator_aug)
```

Epoch 1/1

151/151 [=====] - 22s 147ms/step - loss: 0.5280 - accuracy: 0.8930 - val_loss: 1.6623 - val_accuracy: 0.9200

Comments

Looking at the training and validation accuracy, we can say that the model is trained with almost lesser training accuracy and better validation accuracy as in task 1. We can infer that we might have encountered the overfitting in the task 1. Now, let's move to task 3.

3.5.3 Task 3

In this task, we are going to fine tune the full network and use that network to establish the training, testing and validation accuracy.

3.5.3.1 Fine tuning the full network

Fine tuning full network means we will make all feature layers of VGG19 as trainable. There won't be any non-trainable parameters. After that, we will pass the augmented data set on full connected network in order to fine tune the weights.

```
In [0]: # Unfreezing the VGG19 Layers
for layer in tune_VGG19.layers[:]:
    layer.trainable = True # Make all layers of VGG19 as trainable (Unfreeze)
tune_model_2.summary() # show the summary of model

# Compiling and training the full network
# use adam optimizer
# Fit the model on augmented training and validation data
tune_model_2.compile(loss='categorical_crossentropy',
                    optimizer=optimizers.Adam(learning_rate=1e-5,clipnorm=1),
                    metrics=['accuracy'])

# Fit the model by running the augmented data on full network
Network_3_2=tune_model_2.fit_generator(train_generator_aug,
                                     epochs=1,
                                     verbose=1,
                                     validation_data=validation_generator_aug)

Model: "sequential_26"

Layer (type)                Output Shape                Param #
=====
vgg19 (Model)                (None, 7, 7, 512)          20024384
=====
sequential_25 (Sequential)  (None, 10)                  7588910
=====
Total params: 27,613,294
Trainable params: 27,613,294
Non-trainable params: 0

Epoch 1/1
151/151 [=====] - 37s 245ms/step - loss: 0.2413 - accuracy: 0.9461 - val_loss: 0.1330 - val_
accuracy: 0.9600
```

Comments

If we look at the model summary above, it shows that out of total 27,613,294 parameters, none of the parameters are non-trainable This means that we have successfully unfreezed the feature layers of VGG19.

3.5.3.2 Evaluate the model on testing data

Now, as we have fine tuned the weights and parameters by training the full connected network, lets use this network to establish the test accuracy. That means, we will evaluate this model on test data where it will generate the predictions of test features, compare those predicted labels with actual labels of test data and report the accuracy.

```
In [0]: test_loss_19,test_acc_19=tune_model_2.evaluate(test_generator) # Evaluate the model performance on test data set
print('The testing accuracy is ',(test_acc_19*100).round(2),'%') # Print the testing accuracy of built model
```

```
50/50 [=====] - 1s 15ms/step
The testing accuracy is  98.0 %
```

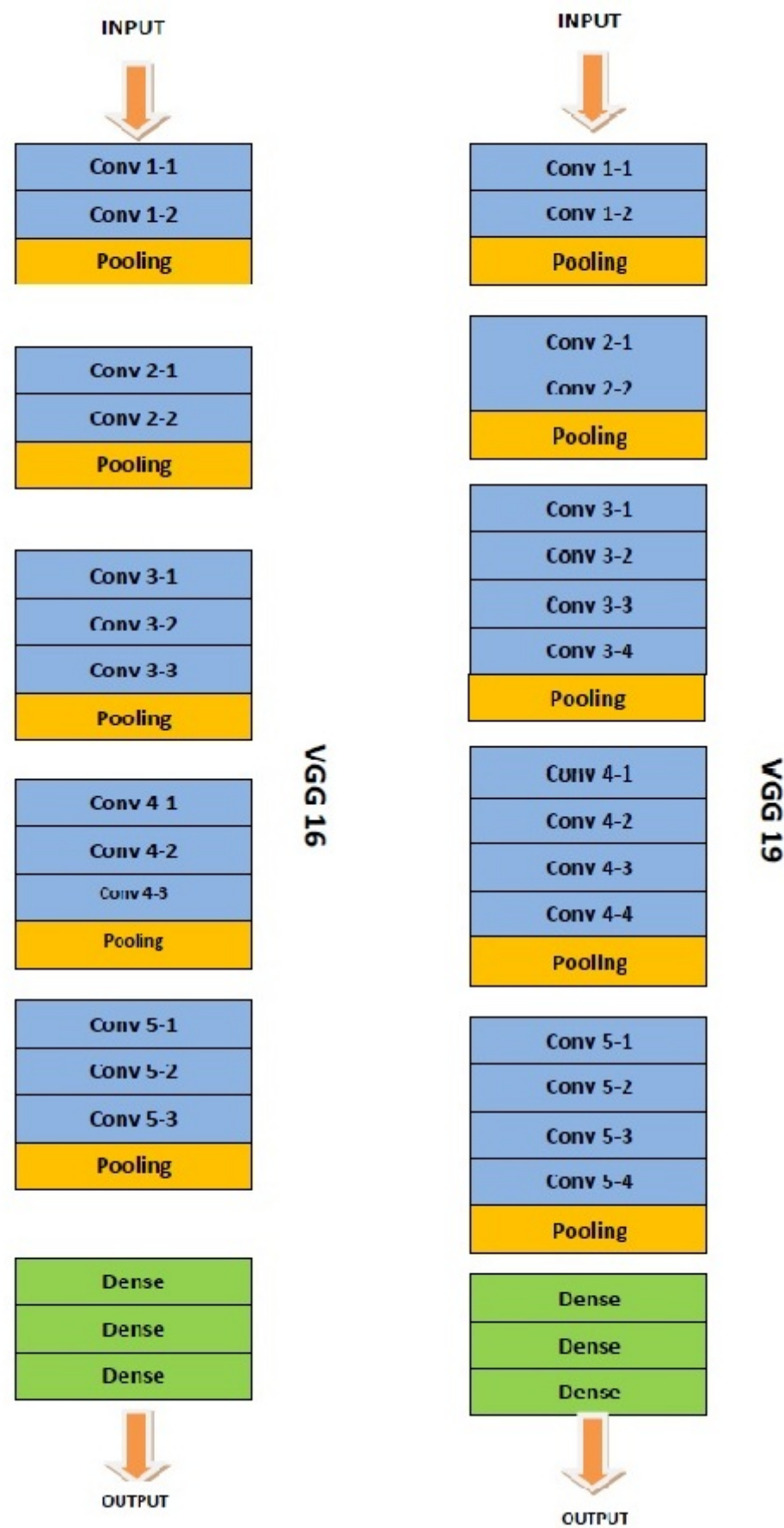
Comments

The testing accuracy we have got with fine tuning of full network is 98% which is good. By this, we can infer that the optimization parameters, number of neurons in dense layers we have selected are appropriate enough for good image classification.

3.6 Comparison of VGG16 & VGG19

Since we have experimented the transfer learning with two different pre-defined networks, we would compare them in terms of structure and their performance and would draw some inferences.

3.6.1 Comparison of structure



Looking at the figure above, we can definitely comment the basic difference between VGG16 and VGG19 as the number of layers of deep neural networks. VGG19 has additional 3 layers as compare to VGG16. These 3 additional layers are present in Conv 3, Conv 4 and Conv 5. Finally, both models have same number of dense and pooling layers.

Number of parameters in VGG16 - 14714688

Number of parameters in VGG19 - 20024384

3.6.2 Comparison of performance

Now, we will compare the performance achieved in terms of training, validation and testing accuracy by using both pre-defined networks. Below code is developed to collect the trainin, validation and testing accuracy after we use both pre-trained models on our data set in an array and use it to show the bar plot for comparison. Let's have a look at the code and the results.

```

In [0]: # Gather the training accuracy of both models
train_acc=np.array([Network_3_1.history['accuracy']][0]*100,Network_3_2.history['accuracy']][0]*100)).round(4)
# Gather the validation accuracy of both models
val_acc=np.array([Network_3_1.history['val_accuracy']][0]*100,Network_3_2.history['val_accuracy']][0]*100)).round(4)
# Gather the testing accuracy of both models
test_acc=np.array([test_acc_16.round(2)*100,test_acc_19.round(2)*100])

labels = ['Training', 'Validation', 'Testing'] # Generate array of these labels
VGG16 = [train_acc[0], val_acc[0], test_acc[0]] # Generate the array of all three accuracies for VGG16
VGG19 = [train_acc[1], val_acc[1], test_acc[1]] # Generate the array of all three accuracies for VGG19

my = np.arange(len(labels))
width = 0.3 # specify the width of bars

fig, ax = plt.subplots(figsize=(8,10)) # figure size

rects1 = ax.bar(my-width/2 , VGG16, width, label='VGG16') # bars of VGG16
rects2 = ax.bar(my + width/2, VGG19, width, label='VGG19') # bars of VGG19

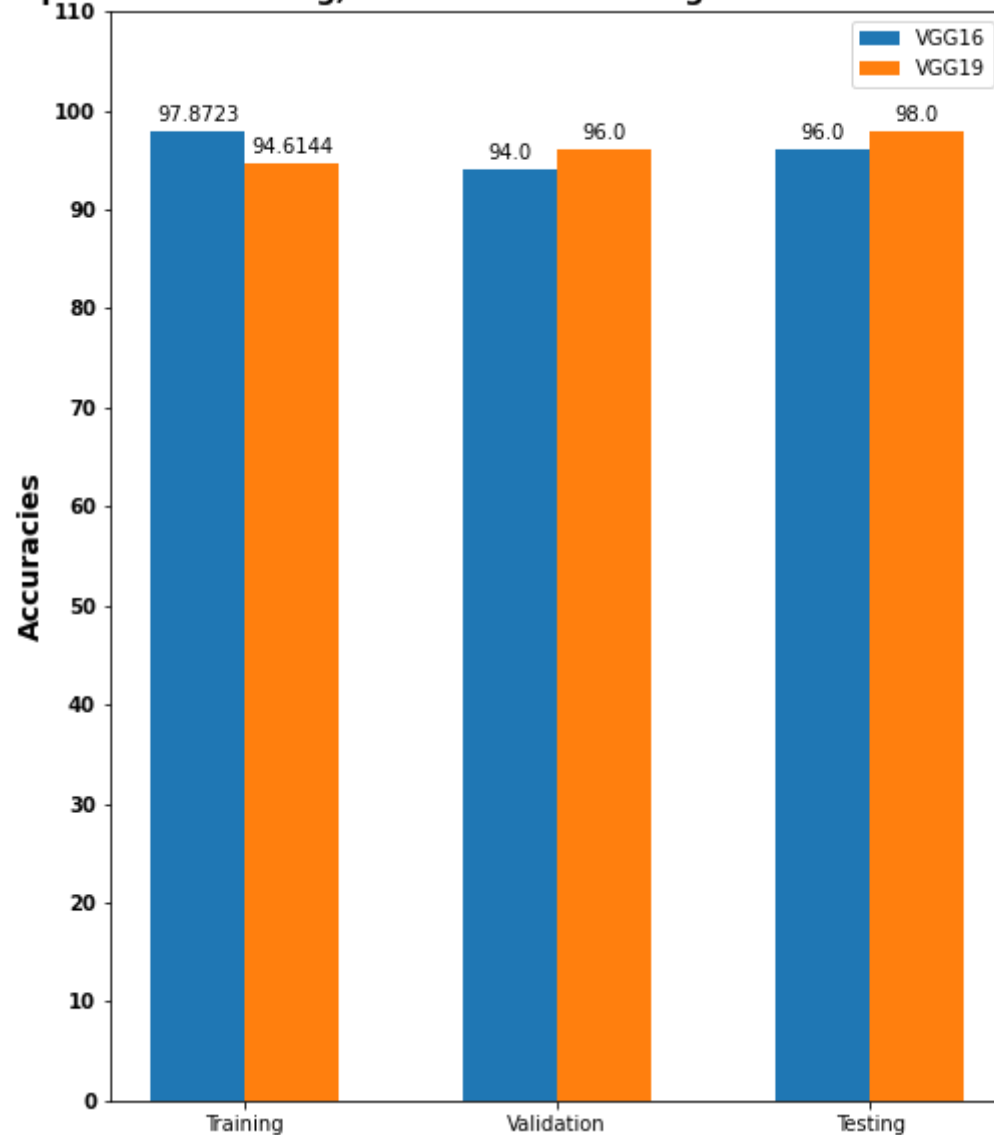
ax.set_ylabel('Accuracies',fontweight = 'bold', fontsize = '14') # set y axis label
ax.set_title('Comparison of training, validation and testing accuracies for both models',fontweight = 'bold', fontsize = '14') # title
ax.set_xticks(my) # set x axis ticks
plt.yticks(np.arange(0.0,120,10),fontweight='bold',fontsize='10') # Give y-ticks
ax.set_xticklabels(labels) # x-axis tick labels
ax.legend() # Give the legend

autolabel(rects1) # auto label the numerical value for VGG16 bars
autolabel(rects2) # auto label the numerical value for VGG19 bars

plt.show() # show the bar plot

```

Comparison of training, validation and testing accuracies for both models



Comments

The above bar plot compares the training accuracy of VGG16 with that of VGG19 and same for validation and testing accuracy. From the plot, it can be said that the training accuracy on VGG16 is more as compared to VGG19 however we got more validation and testing accuracy in VGG19 as compared to VGG16. Now, this can be attributed to two facts. Either we have slight overfitting of the data during training on VGG16 or because of additional three layers and more number of parameters the testing and validation accuracy of VGG19 model is better as compared to VGG16.

4. Conclusion

With the scope of this project, we have learnt how we can perform the augmentation on machine learning data set. We also learnt that there is no unique way to develop the code for augmentation. In our tutorial, we have defined the function of augmentation and used it to create different image manipulations. However, in transfer learning part, we have not used same function as we do not want to retrieve the augmented images for display purpose and by that it also shows another way of performing the augmentation experiment without developing a function.

In second part of the project scope, we learnt how we can experiment the transfer learning using available pre-defined networks in Keras for image classification. The overall intention of our experiment was to learn how can we train the neural network in different ways such as training only output layers by freezing the pre-defined network layers, training fully connected network and so on. As a part of this project, we evaluated the performance on two different networks which gives the insights on the differences between two networks in terms of structure and performance.

In conclusion, pre-trained networks have high applicability for image classification even if they are trained on different data set. It can be said that features learnt by these models are transferrable to image classification problem with high accuracy. The required number of less training samples and fast convergence makes pre-trained network a favorable option for implementing CNN for image classification.