NGEE ANN POLYTECHNIC

School of InfoComm Technology

Deep Learning Assignment

Diploma in CSF / FI / IT

April 2020 Semester

ASSIGNMENT 1

(30% of DL Module)

18th May 2020 – 21st Jun 2020

Submission Deadline:
Presentation: 14th Jun 2020 (Sunday), 11:59PM
Report: 21st Jun 2020 (Sunday), 11:59PM

Tutorial Group :	**~~P01~~ / P02 /~~P03 /P04~~**
Student Name :	**NG CHIN TIONG RYAN**
Student Number :	**(removed to protect identity)**

1. Table of Contents

2. Overview

The Problem

The main goal of this assignment is to classify food images according to the 10 different food items assigned to me. This is a time-consuming task if I were to do it myself and it also increases the possibility that I may classify the foods into the wrong categories if I were to do it manually, thus bringing the need for the use of Convolutional Neural Networks which are less likely to make errors in classification.

The problem is of a multi-class, single label type of classification as there are multiple food categories, with the condition that one image can only belong to one category. Data availability is slightly constrained to 1000 images per category, so data augmentation will be used to mitigate this issue of training the models on a relatively small dataset.

The Objective

Convolutional Neural Networks are able to pick up the main features of characteristics of objects in the image through using filters and max pooling, which aid the model in identifying similar features across different images which may have different backgrounds, lightings, angles etc.

The main objective of this assignment is to experiment with various Convolutional Neural Networks (CNNs) to determine which model is the best for classifying new images. In my opinion, a best model should:

- Not overfit too quickly
- Have good accuracy of above 70%
- Be able to classify **MOST** of the images I downloaded from the Internet

The Approach

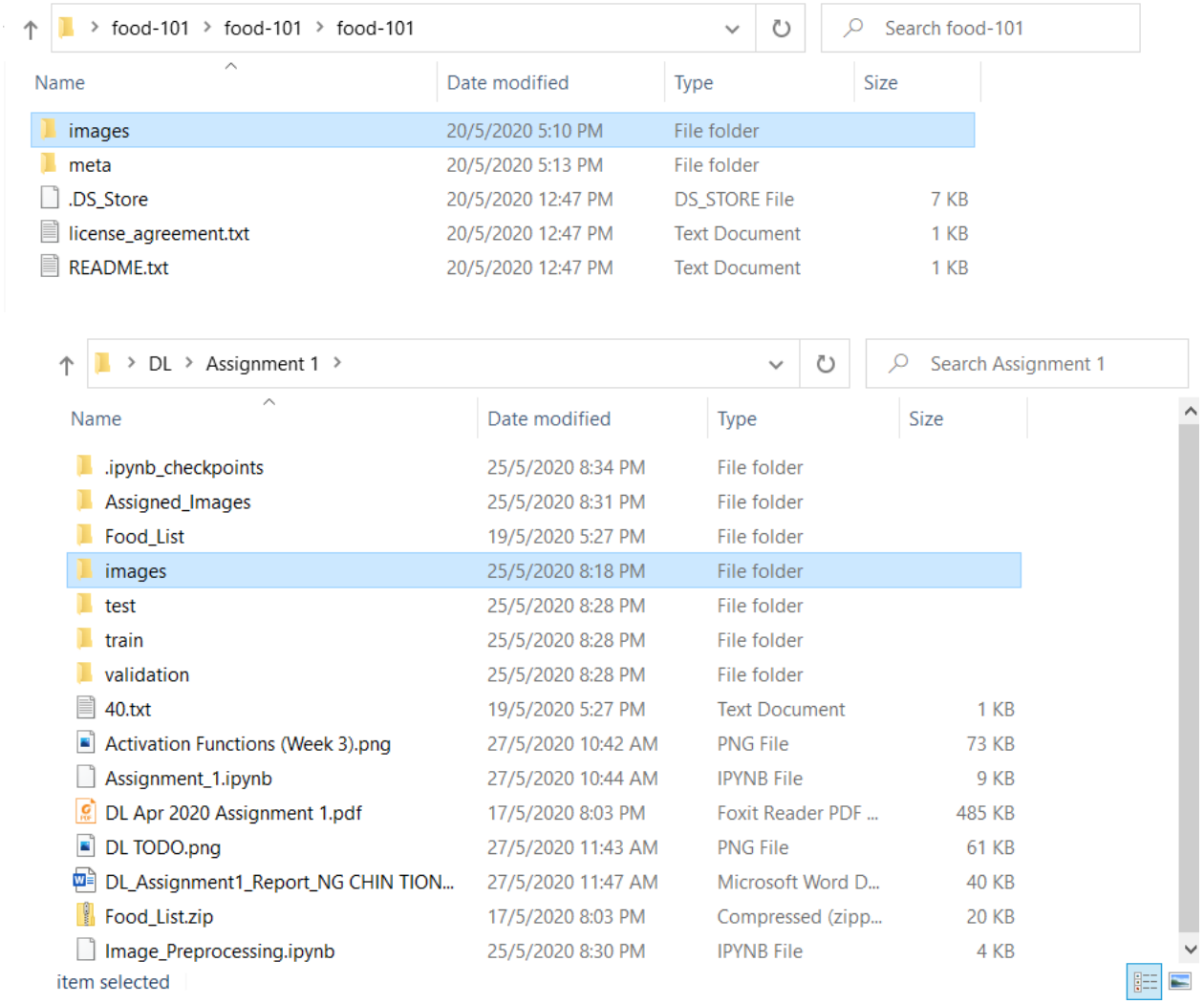
I chose to experiment from non-pretrain models before transitioning to using pretrained models, as from my past experience with pretrained CNNs achieved a higher test and validation accuracies as compared with non-pretrained models. I started off with a base model for each unique model architecture type (Non-Pretrained, VGG16 and ResNet50) and then tweaked the various hyperparameters and tried to achieve the highest possible accuracy for each type of model.

I then used images I had downloaded from the Internet to assess the performance of the best model of different network architectures. Based on the results from the testing, I then went on to find out why the model does well/does not do so well in accurately predicting the corresponding food category and thus determined my best model for food classification. I then used my best model to classify foods downloaded from the Internet to re-evaluate its performance with the use of new images to test if it can perform.

3. Data Preprocessing & Data Loading

The first step in the assignment was data loading and preprocessing to ensure that the images that the models are going to be trained on are in the correct folders and subfolders. To accomplish this, I made use of the provided Image_Preprocessing.ipynb.

Firstly, I downloaded the food-101.zip file from the Kaggle Website. I then unzipped it onto my computer's Desktop folder. After unzipping, the file had multiple subfolders in it. In order to simplify things, I decided to copy the images folder (containing the 101,000 images) into my DL/Assignment 1 directory folder.



Thereafter, I moved my assigned foods list, 40.txt and the Image_Preprocessing.ipynb into the same directory (as required to correctly perform the preprocessing). Then I ran the Image_Preprocessing.ipynb with the path of the base directory and the name of the .txt file specified as shown below.

In [2]:
```
#Set the base directory as the current directory
base_dir = os.getcwd()

#Set the base directory as where you save the downloaded food_images
image_dir = 'C:/Users/maste/Desktop/DL/Assignment 1/images'
```

In [3]:
```
# Directories for your training, validation and test splits
train_dir = os.path.join(base_dir, 'train')
os.mkdir(train_dir)
validation_dir = os.path.join(base_dir, 'validation')
os.mkdir(validation_dir)
test_dir = os.path.join(base_dir, 'test')
os.mkdir(test_dir)
```

In [4]:
```
# Assign the 10 types of food from your .txt file to a list variable 'food_list'

label_file = os.path.join(base_dir, '40.txt')
# Refer to the report Appendix
# Please enter the name of .txt file which contains a list of food assigned to you
# Make sure you save the .txt file in your base_dir

with open(label_file, 'r') as f:
    x = f.readlines()

food_list =[]
for item in x:
    if item == '\n':
        continue
    else:
        food_list.append(item.strip('\n'))
```

The notebook's purpose is to iterate through the list of foods in the .txt file and subsequently create 3 folders—test, train and validation with 10 subfolders each, based on the list of food in the text file and then copy the first 750 images from the images folder to the train subfolders, the next 200 images into the validation subfolders and the last 50 images into the test subfolders, in order to prepare the specific dataset assigned to me.

In [5]:
```
#copy the first 750 images to train folder
for item in food_list:
    train_food_dir = os.path.join(train_dir, item)
    os.mkdir(train_food_dir)
    img_list = os.listdir(os.path.join(image_dir, item))[:750]
    for fname in img_list:
        src = os.path.join(image_dir, item, fname)
        dst = os.path.join(train_food_dir, fname)
        shutil.copyfile(src, dst)
```

In [6]:
```
#copy the following 200 images [750:950] to validation folder
for item in food_list:
    validation_food_dir = os.path.join(validation_dir, item)
    os.mkdir(validation_food_dir)
    img_list = os.listdir(os.path.join(image_dir, item))[750:950]
    for fname in img_list:
        src = os.path.join(image_dir, item, fname)
        dst = os.path.join(validation_food_dir, fname)
        shutil.copyfile(src, dst)
```

In [7]:
```
#copy the remaining 50 images [950:1000] to test folder
for item in food_list:
    test_food_dir = os.path.join(test_dir, item)
    os.mkdir(test_food_dir)
    img_list = os.listdir(os.path.join(image_dir, item))[950:1000]
    for fname in img_list:
        src = os.path.join(image_dir, item, fname)
        dst = os.path.join(test_food_dir, fname)
        shutil.copyfile(src, dst)
```

I decided to copy these 3 folders into another folder to make things easy and specify the path of this folder in the Assignment_1.ipynb, which is required for the images in the respective directories to be loaded later on for various functions (the training images folder for training the model, the validation folder to assess the model's performance during training and the testing images folder to validate the model's performance after training).

```
import os
#base_dir is the directory where you stored the images
base_dir = 'C:/Users/maste/Desktop/DL/Assignment 1/Assigned_Images'

train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')
```

4. Develop the Image Classification Model and Compare models using Test images

For all my models, I chose an image size of 150 x 150 and used data augmentation for all models, which aided in increasing the diversity of the relatively small dataset of 1000 images per food category. For most models, I used a standard set of augmentation parameters. In models where I may have tweaked some of the parameters, I will state which parameters have been changed. Otherwise, all models can be assumed follow the default set of augmentation parameters.

The first category of models that I trained were non-pretrained models.

Base Model: Model 0

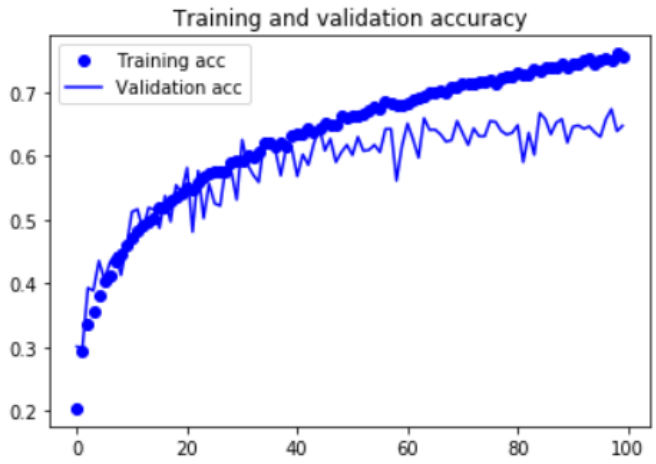
I started off with a base model with 4 Conv2D and max pooling layers, as well as a dense layer with 512 nodes and a final output layer with 10 nodes. I used the Rectified Linear Unit (ReLU) activation function for the input and all the hidden layers, a flatten layer and finally used softmax for the last layer, which ensures that the sum of the probabilities add up to 1. 10 nodes were needed for the last layer to give 10 different probabilities on which type of food category a particular image belonged to when testing the food model.

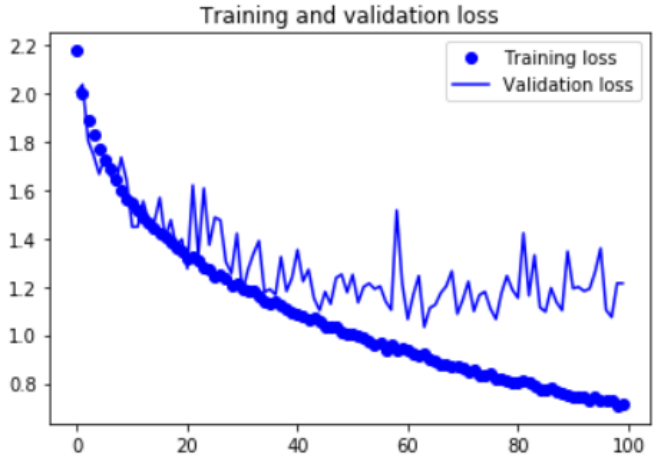
```
# Build the Model
from tensorflow.keras import layers
from tensorflow.keras import models

img_size = 150

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(img_size, img_size, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
model.summary()
```

The loss function is categorical crossentropy since the problem is a multi-class, single label problem. The optimizer I chose to use was RMSprop, with an initial learning rate of 1e-4. The metrics used for model evaluation is accuracy. For fit_generator, I used 250 steps per epoch for 100 epochs, with a batch size of 30 for the train_generator and 8 for the validation generator, since steps per epoch multiplied by the batch size gives the total number of images in the train and validation image folders.

I also used the ImageDataGenerator as shown below for the purpose of data augmentation, to diversify the dataset and to improve the accuracy of the models. Image rescaling is used in the non-pretrained models to normalise the RGB pixel values from the range of 1 to 255 to a range of 0 to 1 instead. In this case, the class mode for both the train and validation generators need to be 'categorical', which corresponds to the model's loss function of categorical crossentropy.

```
from tensorflow.keras import optimizers
from tensorflow.keras.preprocessing.image import ImageDataGenerator
model.compile(loss='categorical_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])

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

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        # This is the target directory
        train_dir,
        target_size=(img_size, img_size),
        batch_size=30,
        class_mode='categorical')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(img_size, img_size),
        batch_size=8,
        class_mode='categorical')

history = model.fit_generator(
      train_generator,
      steps_per_epoch=250,
      epochs=100,
      validation_data=validation_generator,
      validation_steps=250)
```





The model overfits at around 50 epochs as seen from above. I used a test generator of batch size of 10 with 50 steps as shown below. The validation accuracy plateaus at around 65% while the training accuracy continues to increase until about 72% at 100 epochs. This model had a test accuracy of 64.2%.

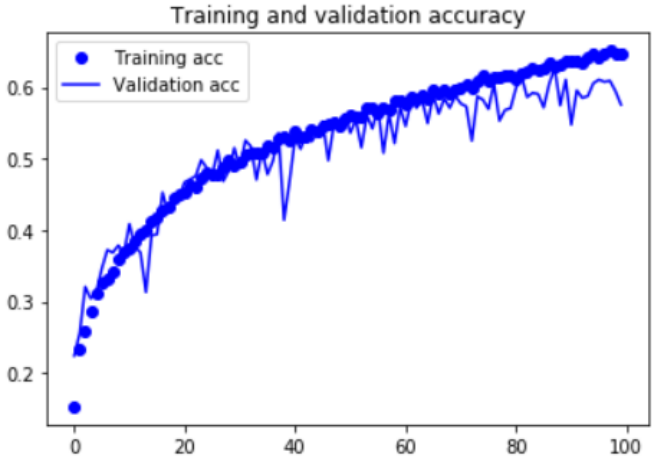
```
#evaluate
test_generator = test_datagen.flow_from_directory(
        test_dir,
        target_size=(img_size, img_size),
        batch_size=10,
        class_mode='categorical')

test_loss, test_acc = model.evaluate_generator(test_generator, steps=50)
print('test acc:', test_acc)
```

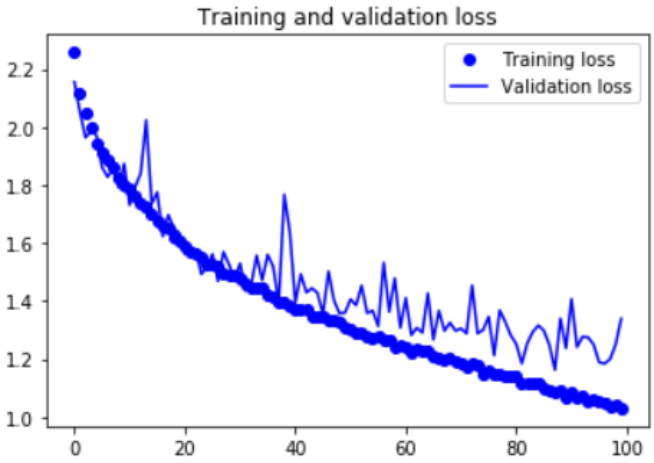
```
  ...
    to
  ['...']
test acc: 0.642
```

Change Steps Per Epoch: Model 2

Then I decided to proceed with changing the steps per epoch to 100 from 250 and also added another Conv2D and a max pooling layer to see if this were to have any impact on the test accuracy of the model. This resulted in the model having a lower test accuracy as compared to the base model, of 57% compared to the initial 64.2%. This model overfits at around 80 epochs, which implies that even if the learning rate stays the same (1e-4), the accuracy can also be affected by adding more layers and changing the batch size of the training and validation generators. After 80 epochs, the validation accuracy of the model falls slightly under the training accuracy and it fluctuates for the last 20 epochs.

```
from tensorflow.keras import layers, models, regularizers
img_size = 150

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(img_size, img_size, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(256, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
model.summary()
```



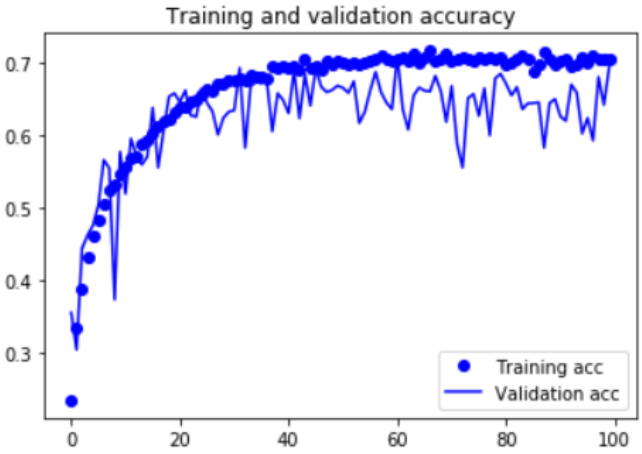


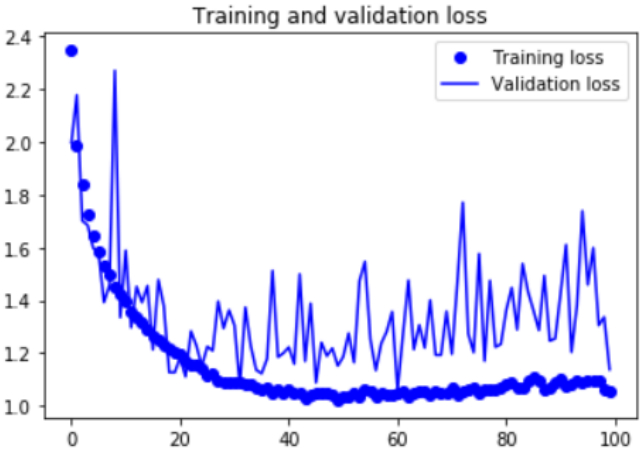

```
...
  to
['...']
test acc: 0.57
```

Add Regularization and Change Learning Rate: Model 1

For the third model, I decided to increase the number of nodes in the last Conv2D layer from 128 to 256, as well as add L2 regularization to two of the layers. I also changed the learning rate to 5e-4, to discover the effects of regularization with a higher learning rate. This model achieved a test accuracy of 68% and now overfits at 70 epochs, thus improvements were made from the base model since Model 1 achieved a higher test accuracy and overfits slightly later. After the model overfits, the validation accuracy continues to fluctuate under the training accuracy, which could be caused by data augmentation.

```
from tensorflow.keras import layers
from tensorflow.keras import models, regularizers

img_size = 150

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(img_size, img_size, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu', kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(256, (3, 3), activation='relu', kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
model.summary()
```



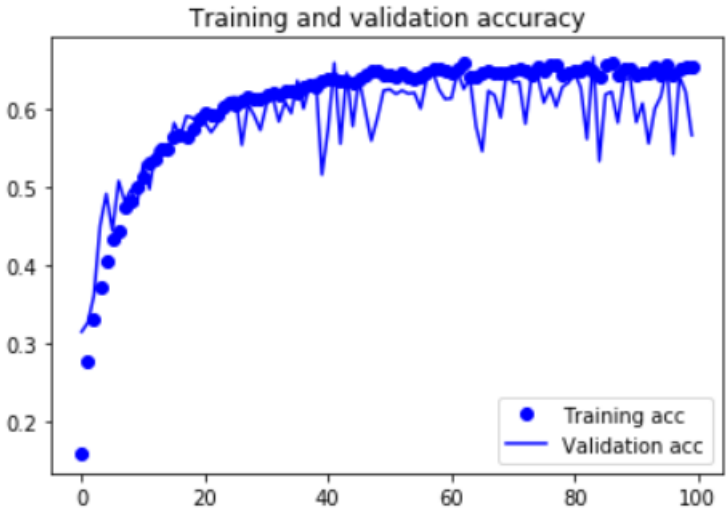


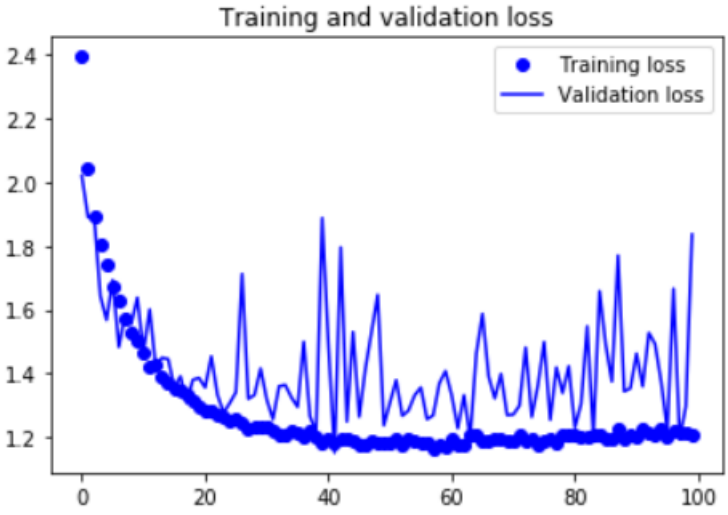

```
  ...
    to
  ['...']
test acc: 0.68
```

Higher Learning Rate: Model 3

In this model, I increased the learning rate of the model from 5e-4 in the previous model to 1e-3 to test the effect of a larger learning rate on the test accuracy of the model. I expected this model to overfit more quickly as compared to the base model as well as model 1. The model becomes slightly overfitted throughout the duration of the 100 epochs as the validation accuracy fluctuates slightly under the training accuracy curve and the model achieves a final test accuracy of 59.8%, which is worse than the base model. Thus, I concluded that the optimal learning rate for RMSprop models is about 5e-4 since the test accuracy is the highest for models with this learning rate.

```
# Build the Model
from tensorflow.keras import layers
from tensorflow.keras import models, regularizers

img_size = 150

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(img_size, img_size, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu', kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(256, (3, 3), activation='relu', kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
model.summary()
```





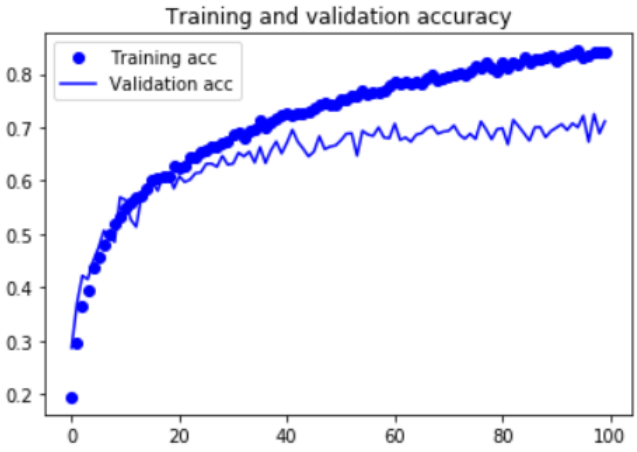

```
  ...
    to
  ['...']
test acc: 0.598
```

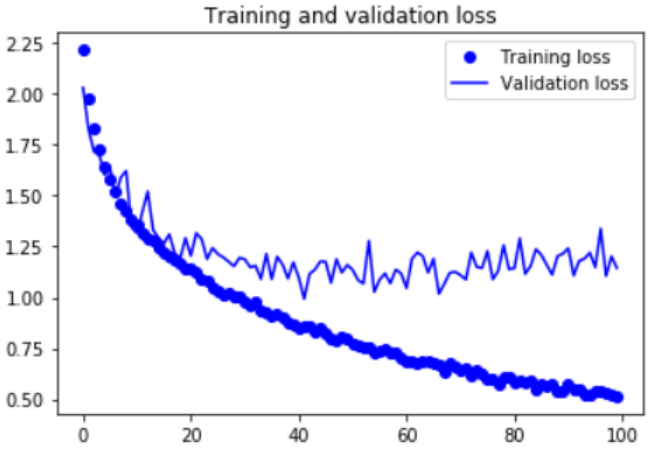
Adam Base Model: Model 5

I then decided to proceed to experiment with the Adam optimiser. I read online that the optimiser achieves good results (accurate predictions for classification) very quickly, meaning it could possibly outperform models with the RMSprop optimiser. I chose to use the architecture as shown below, with one layer of regularization and 2 layers of dropout just in case the model overfits too quickly and a learning rate of 5e-4. This achieved a test accuracy of 71.2% and overfits at about 30 epochs, with the validation accuracy plateauing at around 70% as the training accuracy continues to increase to about 80% at 100 epochs. Based on the findings of my research, this model met my expectations as it overfitted more quickly compared with the RMSprop optimiser and also achieved a higher accuracy.

```
from tensorflow.keras import layers
from tensorflow.keras import models, regularizers

img_size = 150

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(img_size, img_size, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.3))
model.add(layers.Conv2D(128, (3, 3), activation='relu', kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(256, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.3))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
model.summary()
```

```
model.compile(loss='categorical_crossentropy',
              optimizer=optimizers.Adam(lr=5e-4),
              metrics=['acc'])
```





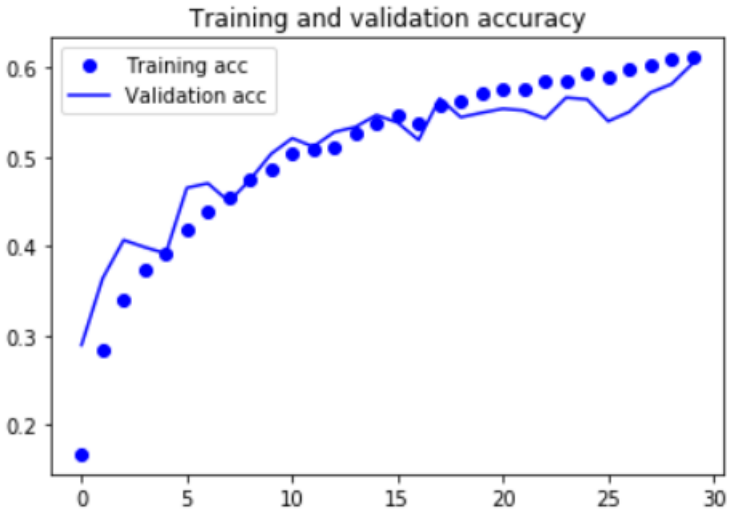

```
  ...
    to
  ['...']
test acc: 0.712
```

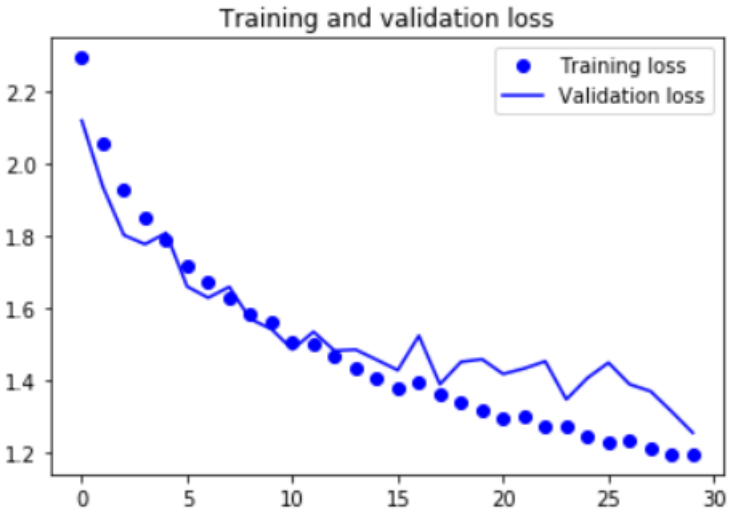
Adam Change Learning Rate: Model 4

I then changed the learning rate of the Adam optimizer to 1e-4 and ran the model for 30 epochs instead, since the previous model overfitted at around 30 epochs in the Adam Base Model. I was expecting this model to overfit less quickly due to the smaller learning rate but it became slightly underfitted since the validation accuracy did not plateau after it dipping slightly at around 25 epochs.

It achieved the worst model test accuracy of 55.8% and did not meet my expectations, so I decided to keep the better performing learning rate (5e-4) for the next Adam model. What I might have done differently for this model is to perhaps run it for 40 epochs instead to check if it overfits then and this might have allowed it to achieve a better test accuracy.

```
from tensorflow.keras import layers
from tensorflow.keras import models, regularizers

img_size = 150

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(img_size, img_size, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.3))
model.add(layers.Conv2D(128, (3, 3), activation='relu', kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(256, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.3))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
model.summary()
```





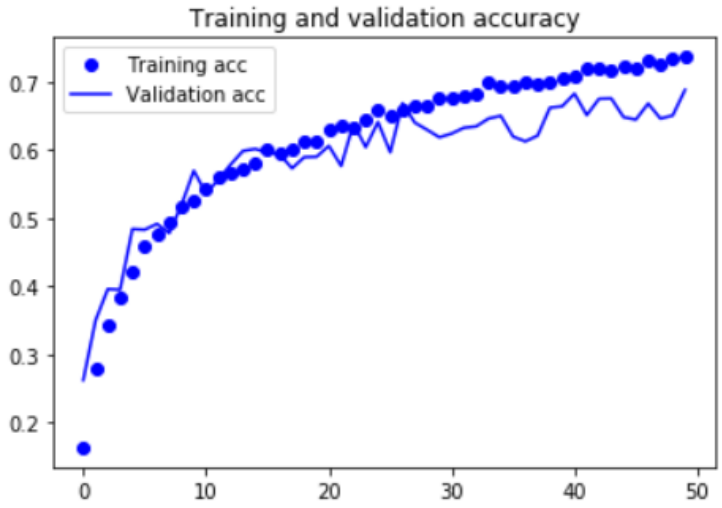

```
  ...
    to
  ['...']
test acc: 0.558
```

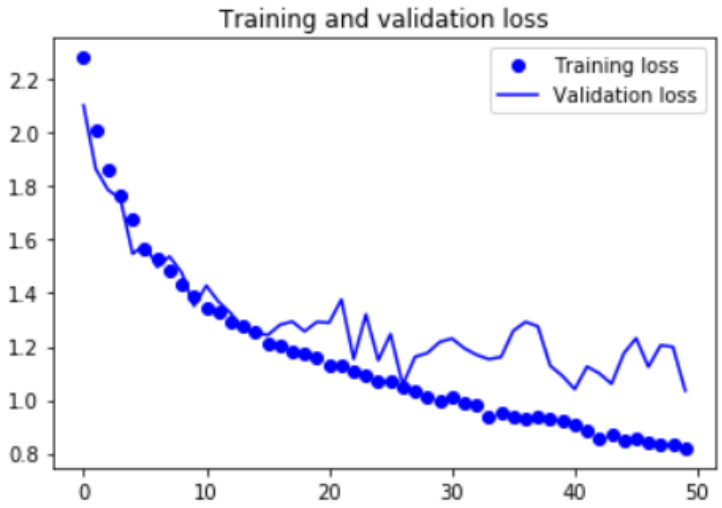
Adam Change Regularization Intensity: Model 8

After I realised using a smaller learning rate resulted in poorer test accuracy scores, I decided to stick with the initial learning rate of 5e-4 for my final Adam model and decided to change the magnitude of the regularizer (from 1e-3 to 2e-3) to see if this would impact the accuracy. I ran this model for 50 epochs instead of 100 since the model was seen to be overfitting quite early in the Adam base model.

This helped to slow down the model overfitting to 30 epochs, similar to the base model, although fluctuations in validation accuracy were still present for the last 5 epochs. However, it has a slightly detrimental effect on the test accuracy, which is reduced from 71.2% in the base model to 69.6% as shown below. After looking at the results of this model, I realised that I should change the magnitude of the regularizer sparingly in future, as this could negatively affect the test accuracy. Since my two attempts at improving the learning rate for Adam was not very successful, I decided to transition to use another optimiser, SGD.

```
from tensorflow.keras import layers
from tensorflow.keras import models, regularizers

img_size = 150

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(img_size, img_size, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.3))
model.add(layers.Conv2D(128, (3, 3), activation='relu', kernel_regularizer=regularizers.l2(0.002)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(256, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Dropout(0.3))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
model.summary()
```






```
  ...
    to
  ['...']
test acc: 0.696
```

SGD Base Model: Model 6

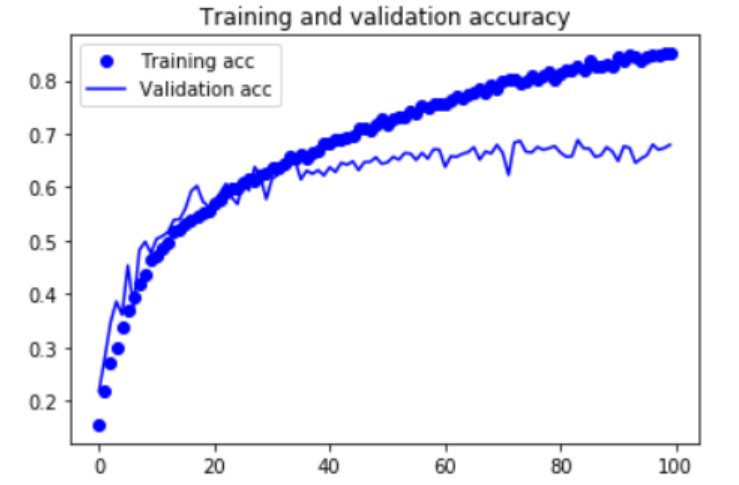
My preliminary research asserted that using the SGD and momentum optimizer combination would allow the model to converge quickly. Some websites also claimed that using momentum also results in less noise in training and thus smoother accuracy and loss curves, as compared with Adam or RMSprop. I was hoping that by doing this, I could achieve a better test accuracy for the models I experimented with.

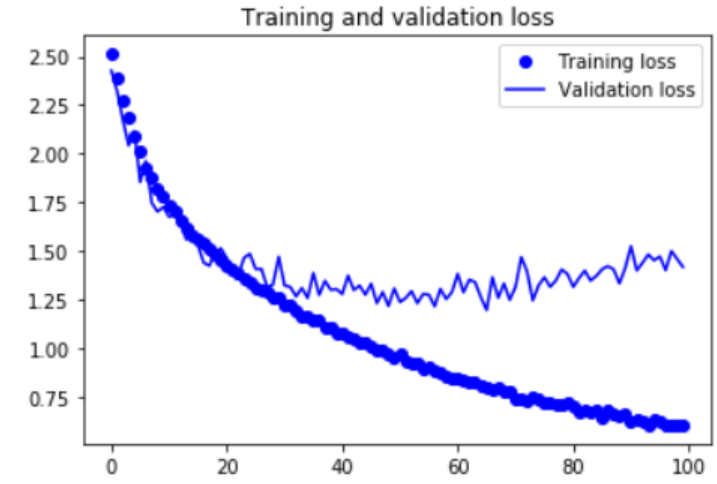
The main goal of me testing out this optimiser combination (SGD with momentum) is to find out if adjusting the momentum can have a large impact on the test accuracy of the model. I started off with the following model architecture as shown below. I used a learning rate of 1e-2 and a momentum of 0.7.

```
from tensorflow.keras import layers
from tensorflow.keras import models, regularizers

img_size = 150

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(img_size, img_size, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu', kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(256, (3, 3), activation='relu', kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
model.summary()
```

```
model.compile(loss='categorical_crossentropy',
              optimizer=optimizers.SGD(lr=1e-2,momentum=0.7),
              metrics=['acc'])
```

This model gave a test accuracy of 67.4%, with it overfitting at around 40 epochs, after running the model for 100 epochs. The validation accuracy starts plateauing at around 50 epochs with an accuracy score of 65%. The noise produced during training is significantly lower than that of RMSprop and Adam which can be seen in the smaller fluctuations in validation accuracy.





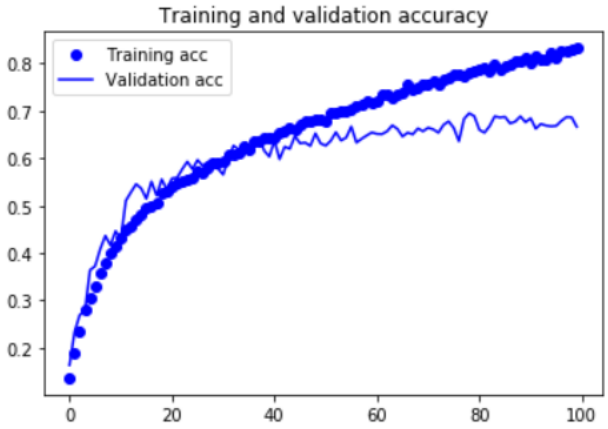

```
  ...
    to
  ['...']
test acc: 0.674
```

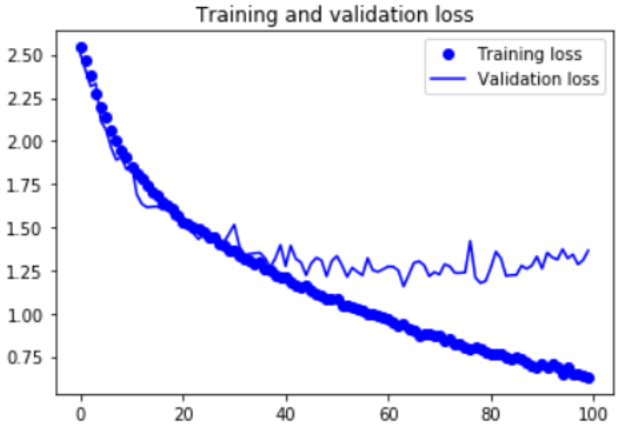
SGD Change Momentum: Model 7

Thereafter, I modified the momentum to 0.5 instead, with the rest of the hyperparameters kept constant. This resulted in a model that overfitted at about 50 epochs, which was largely similar to the previous model. The validation accuracy plateaus at around 80 epochs with small fluctuations of between 64 and 65%. The test accuracy of the model came out to be 64.4%.

Thus, I concluded that using a larger momentum intensity would most likely achieve higher accuracies, although the SGD models' accuracy continues to underperform as compared with the upper bound of the Adam and RMSprop optimisers. I then decided to just use the Adam base model since it has the highest accuracy out of all of my non-pretrained models thus far.

```
from tensorflow.keras import layers
from tensorflow.keras import models, regularizers

img_size = 150

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu',
                        input_shape=(img_size, img_size, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(128, (3, 3), activation='relu', kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(256, (3, 3), activation='relu', kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Flatten())
model.add(layers.Dense(512, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
model.summary()
```






```
...
  to
['...']
test acc: 0.644
```

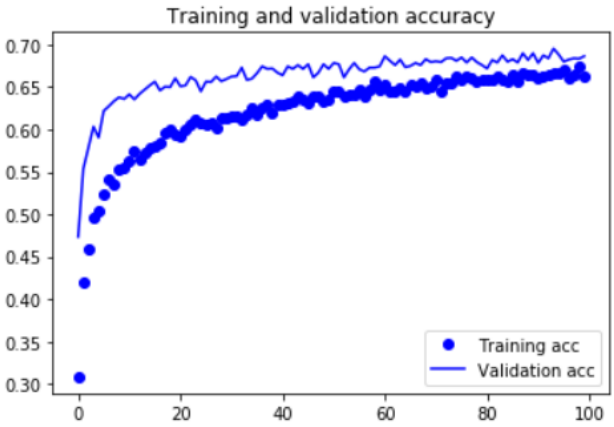
With the decision that the Adam Base Model is my best non-pretrained model, I transitioned to experiment with the pretrained models.

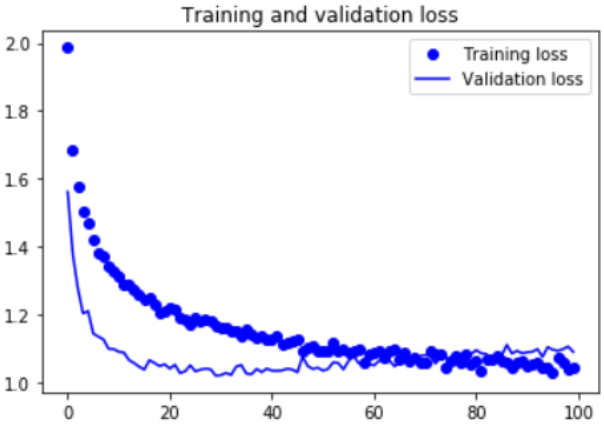
VGG16 Base: Model 10

I started off with VGG16, without unfreezing any of the blocks in the initial architecture. I also added a flatten layer and 1 hidden dense layer in addition to the output dense layer. The model had a learning rate of 1e-4 with a RMSprop optimizer. For this model architecture, data augmentation parameters remained unchanged from the non-pretrained models. After 60 epochs, the validation accuracy started to plateau but continued to remain above the training accuracy at about 68%. Thus, the model is underfitted even at 100 epochs and it achieved a test accuracy of 67.8% upon evaluation of the test generator. Thus, there was a need to modify the learning rate in the first iteration of fine tuning after this to mitigate the issue of underfitting.

```
from tensorflow.keras.applications import VGG16
from tensorflow.keras import models
from tensorflow.keras import layers
img_size = 150

conv_base = VGG16(weights='imagenet',
                  include_top=False,
                  input_shape=(img_size, img_size, 3))
model = models.Sequential()
model.add(conv_base) #VGG16 pretrained
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(10, activation='softmax'))
conv_base.trainable = False
model.summary()
```

```
from tensorflow.keras import optimizers
from tensorflow.keras.preprocessing.image import ImageDataGenerator
model.compile(loss='categorical_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-4),
              metrics=['acc'])

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

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        # This is the target directory
        train_dir,
        target_size=(img_size, img_size),
        batch_size=30,
        class_mode='categorical')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(img_size, img_size),
        batch_size=8,
        class_mode='categorical')

history = model.fit_generator(
      train_generator,
      steps_per_epoch=250,
      epochs=100,
      validation_data=validation_generator,
      validation_steps=250)
```






```
  ...
    to
  ['...']
test acc: 0.678
```

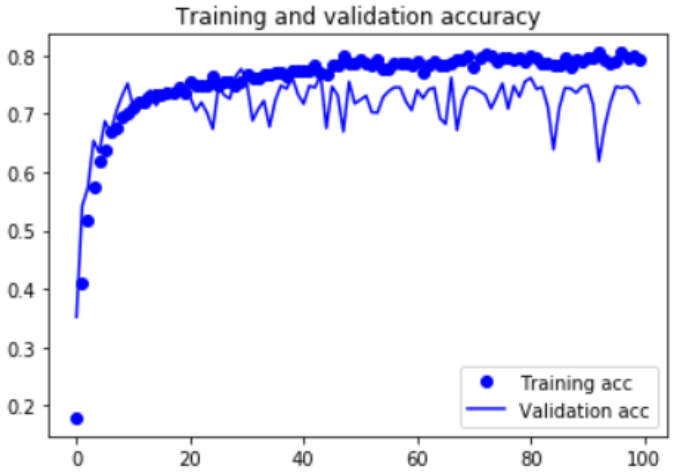
VGG Fine Tuning 1: Model 11

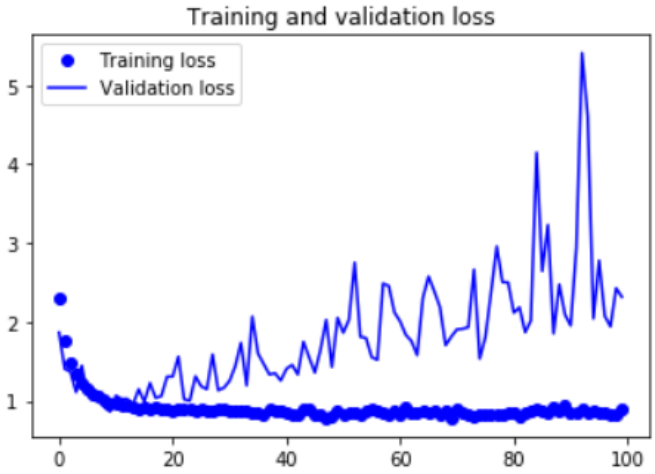
For this model I unfroze some layers of the VGG16 block (from block5_conv1 onwards), as well as added a layer of L2 regularization with a magnitude of 1e-3 to the hidden dense layer and a dropout of 0.2 in case this layer overfitted very quickly. I also changed the learning rate to a higher learning rate from the base model to 5e-4, up from 1e-4 in the base model, to mitigate the underfitting of the model at 100 epochs as in the base model. In addition, I tweaked the zoom range for the data augmentation portion to test if this will lead to better accuracies.

```
from tensorflow.keras.applications import VGG16
from tensorflow.keras import models, regularizers
from tensorflow.keras import layers
img_size = 150

conv_base = VGG16(weights='imagenet',
                  include_top=False,
                  input_shape=(img_size, img_size, 3))


model = models.Sequential()
model.add(conv_base) #VGG16 pretrained
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu',kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.Dropout(0.2))
model.add(layers.Dense(10, activation='softmax'))
conv_base.trainable = True

set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True # after black5_conv1, set_trainable becomes True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
model.summary()
```

The fine-tuned model eventually overfitted at around 50 epochs, with the validation accuracy varying within the range of 60 to 75% after overfitting and the model finally achieved a test accuracy of 73.2%, which was much better as compared with the base model's 67.8%.

```
model.compile(loss='categorical_crossentropy',
              optimizer=optimizers.RMSprop(lr=5e-4),
              metrics=['acc'])

train_datagen = ImageDataGenerator(
    rescale=1./255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.25,
    horizontal_flip=True,)

test_datagen = ImageDataGenerator(rescale=1./255)

train_generator = train_datagen.flow_from_directory(
        # This is the target directory
        train_dir,
        target_size=(img_size, img_size),
        batch_size=30,
        class_mode='categorical')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(img_size, img_size),
        batch_size=8,
        class_mode='categorical')

history = model.fit_generator(
      train_generator,
      steps_per_epoch=250,
      epochs=100,
      validation_data=validation_generator,
      validation_steps=250)
```



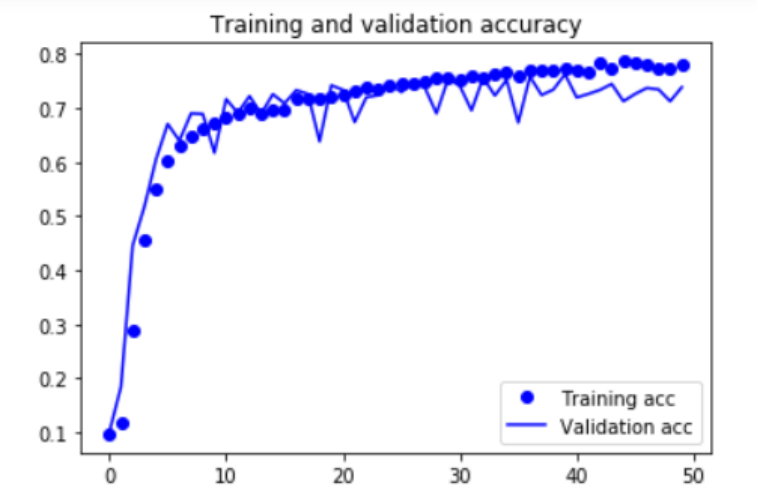


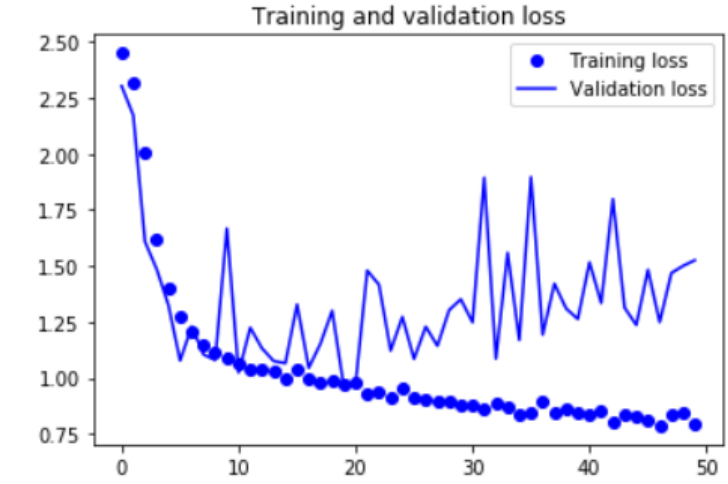

```
  ...
    to
  ['...']
test acc: 0.732
```

VGG Fine Tuning 2: Model 9

I then decided to experiment with changing the magnitude of regularization and dropout intensities from the previous fine-tuning iteration. Note that this model has the same zoom range for data augmentation as the previous fine-tuning iteration of 0.25. I ran this fine-tuned model for 50 epochs since the previous base model already overfitted at around 50 epochs. With this configuration, the new model achieves a test accuracy of 74.6% with it overfitting at around 45 epochs, which is an improvement from both the base model and the first iteration of fine tuning. Although there are no large fluctuations in the validation accuracy, this seems to be the case for the validation loss, which varies between loss scores of 1.0 and 2.0 for this particular model configuration. I was happy with the results of this model and hence decided that this would be the best model for the VGG16 architecture.

```
from tensorflow.keras.applications import VGG16
from tensorflow.keras import models, regularizers
from tensorflow.keras import layers
img_size = 150

conv_base = VGG16(weights='imagenet',
                  include_top=False,
                  input_shape=(img_size, img_size, 3))
model = models.Sequential()
model.add(conv_base) #VGG16 pretrained
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu',kernel_regularizer=regularizers.l2(0.002)))
model.add(layers.Dropout(0.25))
model.add(layers.Dense(10, activation='softmax'))
conv_base.trainable = True

set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
model.summary()
```






```
  ...
    to
  ['...']
test acc: 0.746
```

ResNet50 Base: Model 12

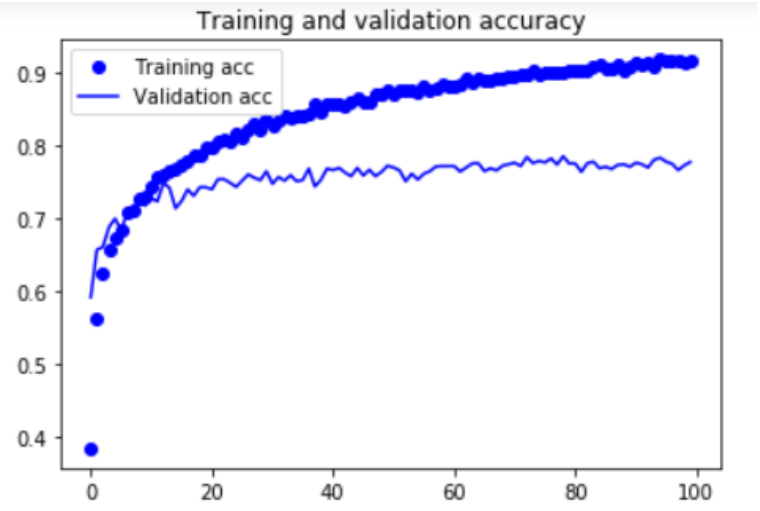
I then proceeded on to train the ResNet50 model. For the first iteration, I used the model without any layers unfrozen. Similar to the VGG16 model, I used a Flatten layer together with a Dense hidden layer with 256 nodes after the VGG blocks. However, I also added regularization and dropout because I read online that ResNet50 overfits rather quickly, thus this hyperparameter tunings were already essential in the base model to prevent it from overfitting too quickly. In addition, I also used a smaller learning rate of 2e-5 for the RMSprop optimizer as compared with the VGG16 model to mitigate the premature overfitting of the model.

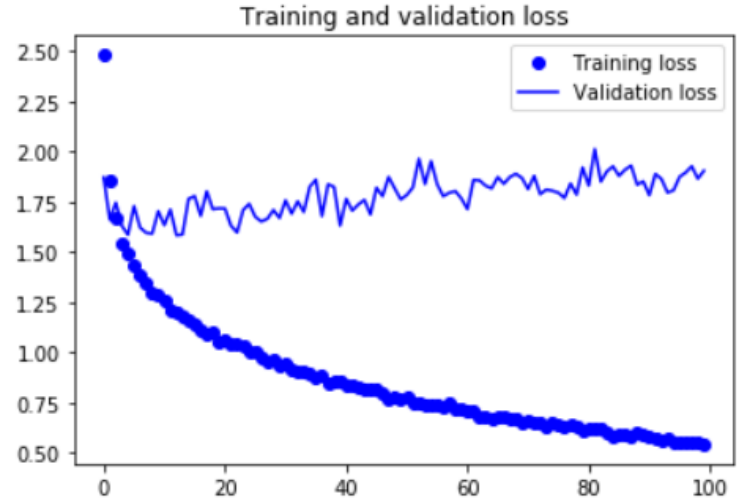
```
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.applications.resnet50 import preprocess_input
from tensorflow.keras import models, regularizers
from tensorflow.keras import layers
img_size = 150

conv_base = ResNet50(weights='imagenet',
                     include_top=False,
                     input_shape=(img_size, img_size, 3))
model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu',kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.Dropout(0.3))
model.add(layers.Dense(10, activation='softmax'))

conv_base.trainable = False
model.summary()
```

For ResNet50's data augmentation portion, another key difference from the non-pretrain models and VGG16 is that image rescaling is not needed but instead, a preprocessing function is used, with the preprocess input indicated as the parameter for data augmentation as shown below.

```
# Train the Model
from tensorflow.keras import optimizers
from tensorflow.keras.preprocessing.image import ImageDataGenerator
model.compile(loss='categorical_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-5),
              metrics=['acc'])

train_datagen = ImageDataGenerator(
    preprocessing_function = preprocess_input,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
    zoom_range=0.25,
    horizontal_flip=True)

test_datagen = ImageDataGenerator(preprocessing_function = preprocess_input)

train_generator = train_datagen.flow_from_directory(
        # This is the target directory
        train_dir,
        target_size=(img_size, img_size),
        batch_size=30,
        class_mode='categorical')

validation_generator = test_datagen.flow_from_directory(
        validation_dir,
        target_size=(img_size, img_size),
        batch_size=8,
        class_mode='categorical')

history = model.fit_generator(
      train_generator,
      steps_per_epoch=250,
      epochs=100,
      validation_data=validation_generator,
      validation_steps=250)
```

The model achieved a final test accuracy of 75.4%, with it overfitting at around 15 epochs despite the hyperparameter tuning I had already put into place, which I thought could slow down overfitting. The validation accuracy plateaus after 50 epochs at about 75% accuracy, while the training accuracy also starts plateauing around 80 epochs, this time with an accuracy reading of about 90%. Validation loss starts increasing gradually after about 10 epochs, which could be due to the overfitting.






```
...
  to
['...']
test acc: 0.754
```

ResNet50 Fine Tuning 1: Model 13

For this first iteration of fine tuning, I changed the learning rate to 1e-5 (from 2e-5), as well as unfroze some of the layers of the model (after the conv5_block1_1_conv layer), in the hope that the model will not overfit so quickly and achieve a better test accuracy.

```
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.applications.resnet50 import preprocess_input
from tensorflow.keras import models, regularizers
from tensorflow.keras import layers
img_size = 150

conv_base = ResNet50(weights='imagenet',
                     include_top=False,
                     input_shape=(img_size, img_size, 3))


model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu',kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.Dropout(0.3))
model.add(layers.Dense(10, activation='softmax'))
conv_base.trainable = True
set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'conv5_block1_1_conv':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
model.summary()
```

```
model.compile(loss='categorical_crossentropy',
              optimizer=optimizers.RMSprop(lr=1e-5),
              metrics=['acc'])
```

This model did not really perform as expected, as it overfitted at around the same time as the base model (at around 15 epochs), although its test accuracy did improve to 79.6%, up from the previous iteration's 75.4%. It can be observed from the loss curve also that the validation accuracy rises sharply from 40 epochs onwards and this may be due to the model getting too optimised towards predicting the training images thus its performance becomes increasingly poor in classifying validation images, resulting in the large increase in validation loss.

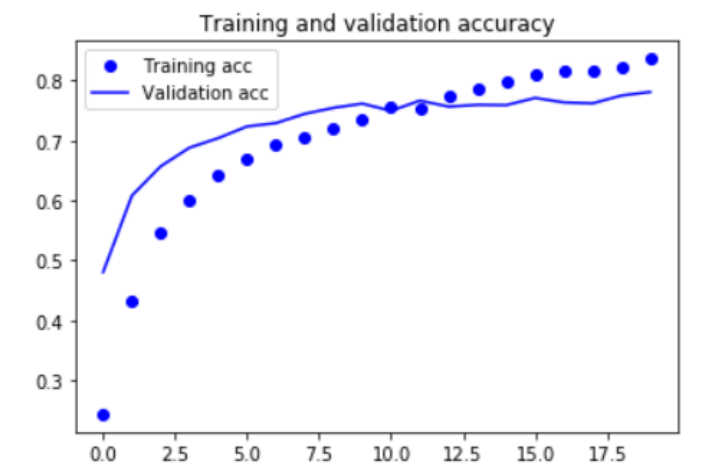

```
  ...
    to
  ['...']
test acc: 0.796
```

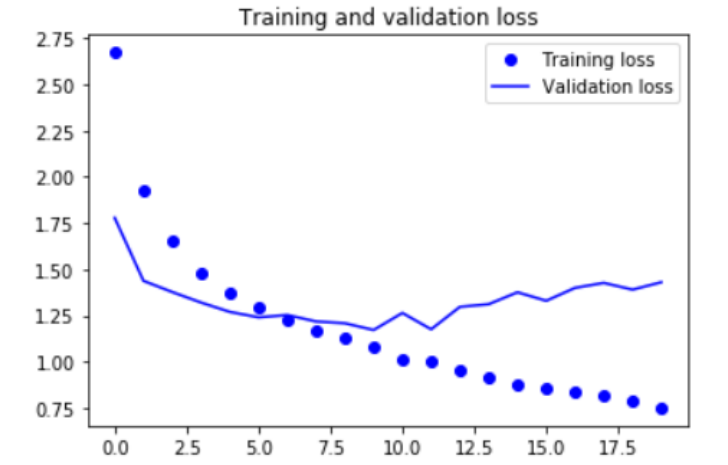
ResNet50 Fine Tuning 2: Model 14

For this model I kept the learning rate the same at 1e-5, although I chose to use less nodes in the dense layer (128 as compared to 256) in order to simplify the network architecture so that the loss score will be further reduced and not rise exponentially like in the previous iteration of fine tuning. I also increased the dropout intensity by 0.05 to 0.35 in order to slow down the rate of overfitting.

```
# Build the Model
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.applications.resnet50 import preprocess_input
from tensorflow.keras import models, regularizers
from tensorflow.keras import layers
img_size = 150

conv_base = ResNet50(weights='imagenet',
                     include_top=False,
                     input_shape=(img_size, img_size, 3))
model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(128, activation='relu',kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.Dropout(0.35))
model.add(layers.Dense(10, activation='softmax'))
conv_base.trainable = True
set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'conv5_block1_1_conv':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
model.summary()
```

Unfortunately, this model did not perform as expected. The loss score still increases after 10 epochs and the model overfits at 15 epochs, after it was run for 20 epochs. This model achieved a worse test accuracy as compared to the first iteration of fine tuning, of 78.8% (down from 79.6%).






```
  ...
    to
  ['...']
test acc: 0.788
```

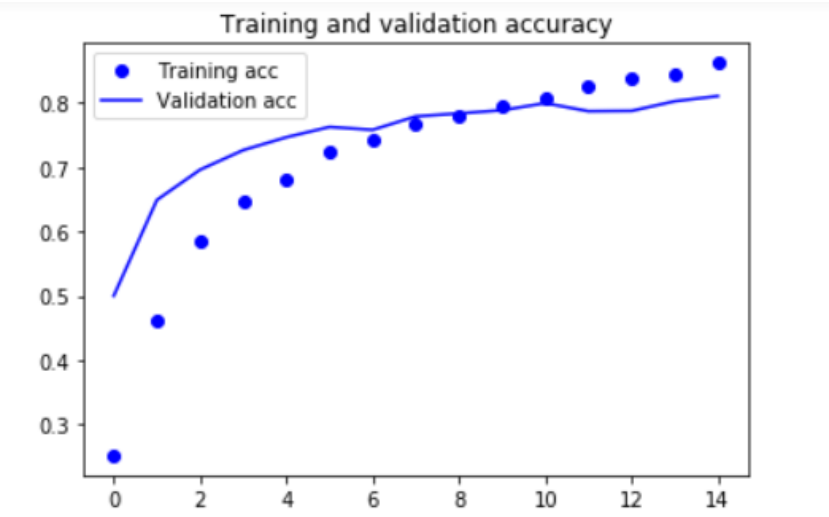
ResNet50 Fine Tuning 3: Model 15

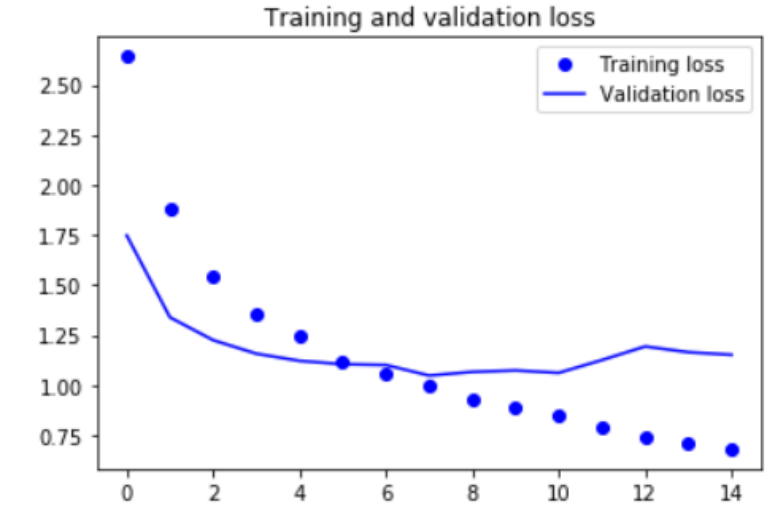
For this last fine-tuning iteration, I decided to test the effect on unfreezing more blocks on the test accuracy of the model. I used the same hyperparameters as the previous model and only increased the number of blocks frozen to hopefully further decrease the loss score which will in turn improve the model's certainty in its prediction of new images.

I also hoped that this model would overfit less quickly, since unfreezing more layers would make more layers and hence more nodes trainable. This would make the model more susceptive of the shared features between images of the same food category due to the fact that pretrained models being biased towards the features of the datasets that they had been previously trained using.

```
# Build the Model
from tensorflow.keras.applications import ResNet50
from tensorflow.keras.applications.resnet50 import preprocess_input
from tensorflow.keras import models, regularizers
from tensorflow.keras import layers
img_size = 150

conv_base = ResNet50(weights='imagenet',
                     include_top=False,
                     input_shape=(img_size, img_size, 3))
model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(128, activation='relu',kernel_regularizer=regularizers.l2(0.001)))
model.add(layers.Dropout(0.35))
model.add(layers.Dense(10, activation='softmax'))
conv_base.trainable = True
set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'conv4_block1_1_conv':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
model.summary()
```

This model achieved the first objective of a smaller increase in the validation loss, but the model overfits faster than the first two iterations, at about 13 epochs as compared with 15 in the previous fine-tuning iteration. The model also achieved a higher test accuracy of 83.4% as compared with the previous two fine tuning iterations and the ResNet50 base model, which had accuracies between 75 and 80%.






```
  ...
    to
  ['...']
test acc: 0.834
```

Summary of changes made that caused improvements in test accuracy

There were improvements for instances where I optimised the learning rate, which is said to be one of the most important hyperparameters that needs to be tweaked when experimenting with Neural Networks in general. In addition, I discovered that the test accuracy could also be improved in pretrained models by unfreezing more layers. Changing the network architecture (the number of layers or the number of nodes in each layer) could also have a direct impact on impacting the final test accuracy of the model.

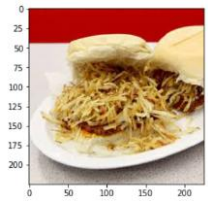
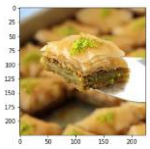
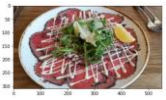
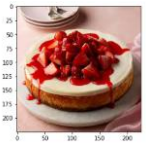
5. Evaluating the best test accuracy models

Based on the three different distinct network architectures I have experimented with (Non-pretrained, VGG16 and ResNet50), I have shortlisted the best model for each architecture type for my consideration towards the best model, which I would ultimately use for the internet image classification. The models are the Adam Base Model (Model 5), VGG Fine Tuning 2 (Model 9) and ResNet50 Fine Tuning 3 (Model 15). In my opinion, the best model:

- Not overfit too quickly
- Have good test accuracy of above 70%
- Be able to classify **MOST** of the images I downloaded from the Internet

To ensure that there is no bias in picking the best model, I decided to use a points system—1 point would be awarded for the best model for each condition, 2 points for the second model and so on. Then I proceeded to calculate the average points and the model with the lowest number of points would be my best model.

Condition 3: Be able to classify **MOST** of the images I downloaded from the Internet
I decided to use six preliminary test images downloaded from the internet to guide my decision of which model I should use, alongside the two other criteria as mentioned above. Here is a table showing the results of my test:

Image						
Model 5	Creme Brulee (75.4%)	Baklava (99.9%)	Beef Carpaccio (99.9%)	Caprese Salad (98.2%)	Cheesecake (96.8%)	Peking Duck (99.4%)
Model 9	Omelette (58.4%)	Baklava (54.8%)	Beef Carpaccio (100%)	Caprese Salad (86.7%)	Cheesecake (73%)	Peking Duck (100%)
Model 15	Hamburger (99.9%)	Hamburger (99.9%)	Hamburger (99.9%)	Hamburger (99.9%)	Hamburger (99.9%)	Hamburger (99.9%)
Correct Label	**Hamburger**	**Baklava**	**Beef Carpaccio**	**Caprese Salad**	**Cheesecake**	**Peking Duck**

Model 5: 1(tie); Model 9: 1(tie); Model 15: 3

Based on this condition, Model 5 and Model 9 do the best in classifying foods correctly. Model 5 however, does have a greater certainty for the 5 foods it correctly labelled as

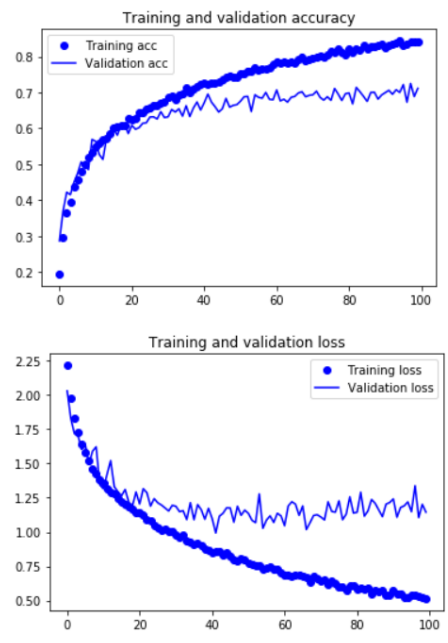
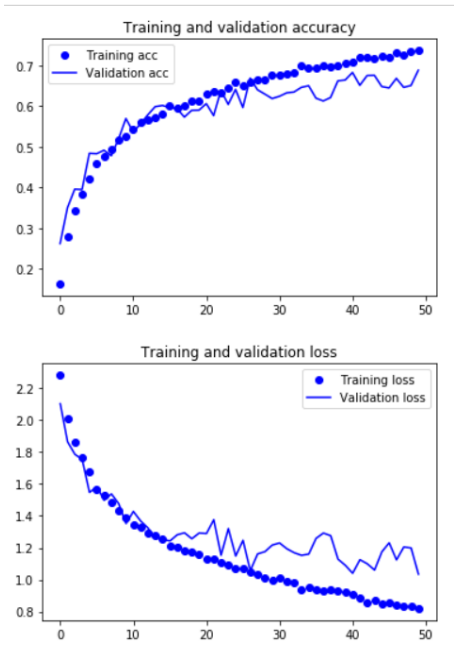
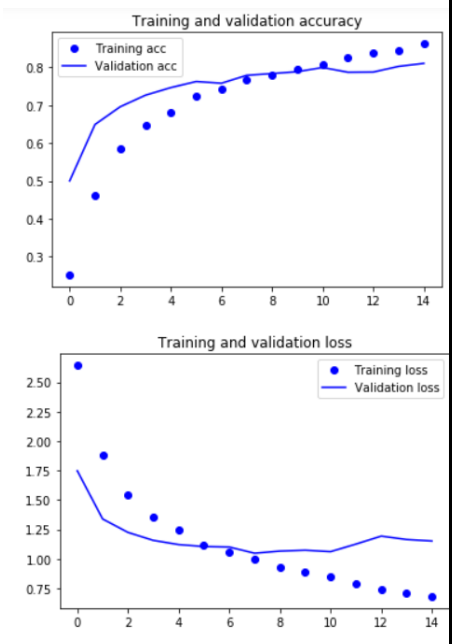
compared with Model 9. Model 15 did the worst here, incorrectly predicting hamburger for 5 out of the 6 images with a high certainty of 99.9%. Based on my criteria, Models 5 and 9 had a tie because I only required them to correctly classify the foods.

Condition 2: Have good test accuracy of above 70%

Since the metrics for all the models built for this assignment is accuracy, it is the main component that I used for evaluation of the models' performance and played a key role in determining my best model. All three models met this condition of having a test accuracy of above 70%. However, the models which did better were favourably considered above those that did not do so well in this aspect. Model 15 took the lead with a test accuracy of 83.4%, followed by Model 9 with an accuracy of 74.6% and lastly, Model 5 with 71.2%.

Model 5: 3; Model 9: 2; Model 15: 1

Condition 1: Should not overfit too quickly

Model 5 (Non-pretrained)	Model 9 (VGG16)	Model 15 (ResNet50)
		

For this condition, Model 9 is the clear winner. Model 9 overfits at 45 epochs which is much later than the 15 epochs in Model 15 as well as 35 epochs in Model 5.

Model 5: 2; Model 9: 1; Model 15: 3

Final Tabulated Points:
Model 5: (1+3+2) /3 = 2
Model 9: (1+2+1) /3 = 1.67 (Best Model)
Model 15: (3+1+3) /3 = 2.33

Based on the points calculated, Model 9 is my best model.

Additional Reasons Why ResNet50 Fine Tuning 3 (Model 15) is not my best model
As mentioned previously, Model 15 has some performance issues with regards to classifying new test images as it predicts the wrong food label with high certainty—hamburger with 99.9% accuracy. It also overfits faster than VGG16 (15 as compared with 35 epochs).

This could be primarily due to the fact that it experiences network degradation, resulting in accuracy saturation, which is not caused by overfitting as described in a Microsoft Research Paper. In addition, a researcher pointed out in his Master's thesis that the maximum test accuracy of ResNet50 is consistently lower than that of VGG16 as shown below (Vasu, 2018).

	AID		UCM	
	Training Accuracy (%)	Test Accuracy (%)	Training Accuracy (%)	Test Accuracy (%)
VGG16	94.3	94.1	92.2	91.8
ResNet50	96.3	92.4	94.2	91.7
DenseNet121	98.74	96.8	95.6	93.87

Table 4.1: Table showing the training and testing accuracy of VGG16, ResNet50 and DenseNet on the AID and UCM dataset.

Based on the experiments that the researcher conducted, the ResNet50 test accuracy is lower than that of VGG16 (92.4% compared to 94.1% and 91.7% compared to 91.8%). This suggests that VGG16 may do better than ResNet50, given the same conditions.

After testing the models against my predictions and further research on the poor performance of ResNet50 on new images, I concluded that the VGG16 Fine Tuning 2 will be my best model, which I would ultimately be using to perform classification on the new images that I downloaded from the Internet.

6. Use the Best Model to perform classification

Test setup

In [1]:
```
# Load the model
from tensorflow.keras import models
model = models.load_model('food_model_9.h5')
```

In [2]:
```
# Load the food list (in alphabetical order)
with open('40.txt', 'r') as f: # the .txt file which contains a list of food assigned to you
    x = f.readlines()
food_list =[]
for item in x:
    if item == '\n':
        continue
    else:
        food_list.append(item.strip('\n'))
food_list = sorted(food_list) # food_list needs to be sorted alphabetically before feed into prediction() function
print(food_list)
```

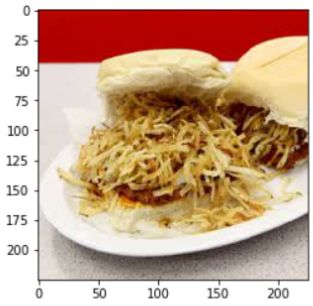
```
['baklava', 'beef_carpaccio', 'caprese_salad', 'cheesecake', 'creme_brulee', 'hamburger', 'omelette', 'paella', 'peking_duck', 'sushi']
```

In [3]:
```
# Define some related functions for image process and model prediction
from keras.preprocessing.image import load_img, img_to_array
img_size = 150
def image_process(img):
    image = load_img(img, target_size =(img_size, img_size))
    image_array = img_to_array(image)/255
    return image_array

import pandas as pd
def prediction(model, img_array, items_l):
    prob = model.predict(img_array.reshape(1,img_size,img_size,3))
    pro_df = pd.DataFrame(prob, columns = items_l)
    result = items_l[np.argmax(prob)]
    return pro_df, result
```

```
Using TensorFlow backend.
```

The saved model is first loaded and saved into the model variable. Thereafter, the contents of 40.txt is open and read, thereafter being stored into a list. Then the image is cut into 150x150 and the RGB pixel values are then normalized from a range of 0 to 255 to a range of 0 to 1. I needed to hard code the img_size if not I would run into an error, saying that it was previously defined. Inside the predict function, the image is reshaped into a 1 by 150 by 150 by 3 array (1 image with dimensions of 150x150 and 3 colour channels—RGB) and thereafter the probability that the image belongs to each food category is calculated and this, together with the corresponding prediction of the image (based on the highest probability) is returned.

```
In [14]: # Make prediction for the image you downloaded from internet
import matplotlib.pyplot as plt
import numpy as np
img = 'hamburger.jpg' # the picture you downloaded from internet, which contains a type of food in your food list
plt.imshow(plt.imread(img))
plt.show()

img_array = image_process(img)
prob_df, result = prediction(model, img_array, food_list)
print('The prediction is: ', result, '\n\n', prob_df)
```

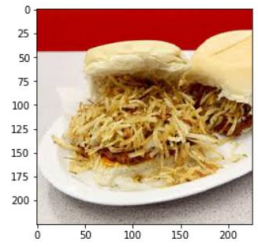


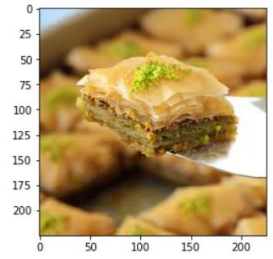
```
The prediction is:  omelette

     baklava  beef_carpaccio  caprese_salad  cheesecake  creme_brulee  \
0  0.015875        0.004082        0.00284    0.008221      0.127775

   hamburger  omelette   paella  peking_duck     sushi
0   0.043469  0.584929  0.19248     0.015595  0.004734
```

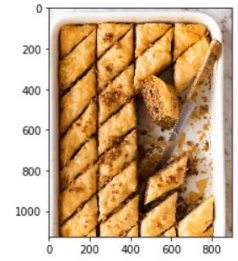
For each image downloaded from the internet, it is first passed through the image_process method to be normalised. Thereafter, the predict_model function is called to predict the most probable food class that particular image belongs to.

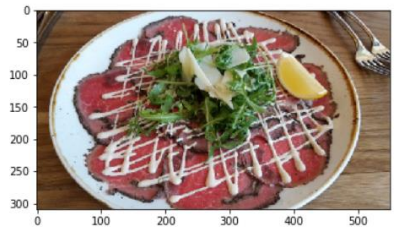
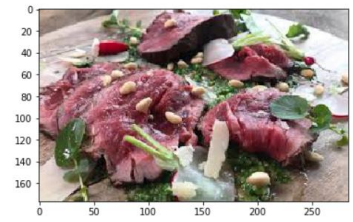
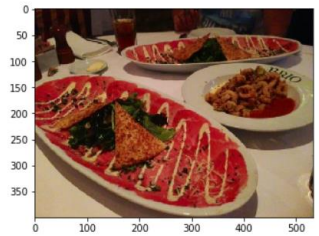
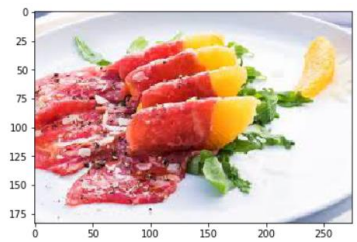
Conducting the testing

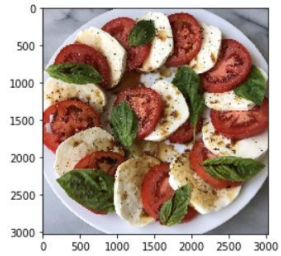
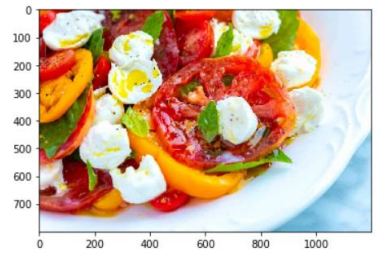
I downloaded 40 testing images from the Internet (4 from each food category). In some cases, I downloaded less obvious examples to assess my best model's (VGG16 Model 9) behaviour on classifying these foods as shown below in the table. I have highlighted the incorrect predictions in red, as well as provided the correct label certainty which would be used later for model evaluation.

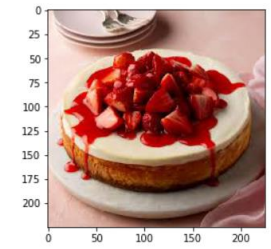
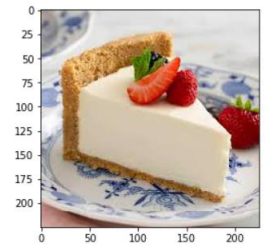
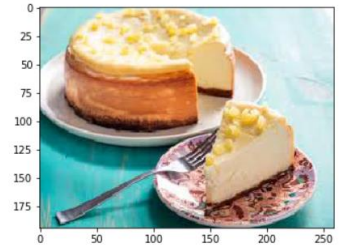
Image	Model Prediction	Correct Label	Correct Label Certainty
 The prediction is: omelette baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.015875 0.004082 0.00284 0.008221 0.127775 hamburger omelette paella peking_duck sushi 0 0.043469 0.584929 0.19248 0.015595 0.004734 hamburger1	Omelette	Hamburger	4.34%

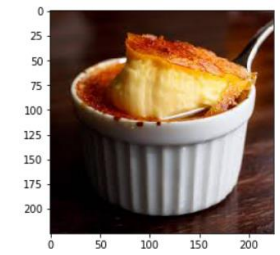
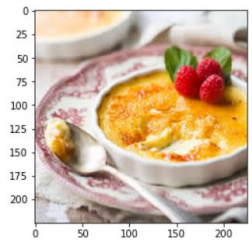
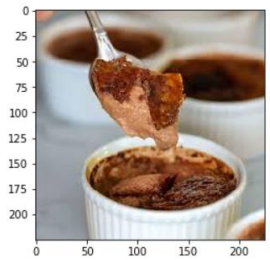
 The prediction is: hamburger baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.000014 0.000001 0.000081 0.000286 0.000004 hamburger omelette paella peking_duck sushi 0 0.996424 0.000861 5.490667e-07 0.00222 0.000108 hamburger2	Hamburger	Hamburger	99.6%
 The prediction is: hamburger baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.004558 1.610952e-08 0.000002 0.006059 0.000066 hamburger omelette paella peking_duck sushi 0 0.981628 0.002805 1.210737e-07 0.004367 0.000517 hamburger3	Hamburger	Hamburger	98.2%
 The prediction is: hamburger baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.0 0.0 0.0 0.0 0.0 hamburger omelette paella peking_duck sushi 0 1.0 0.0 0.0 0.0 0.0 hamburger4	Hamburger	Hamburger	100%
 The prediction is: baklava baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.548714 0.001311 0.001511 0.147168 0.011609 hamburger omelette paella peking_duck sushi 0 0.056231 0.059375 0.001372 0.092663 0.080047	Baklava	Baklava	54%

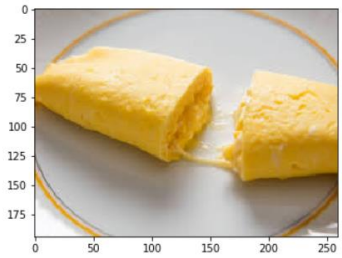
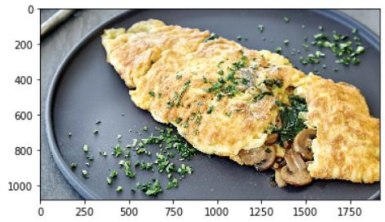
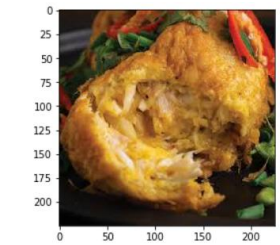
baklava1  The prediction is: baklava baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 1.0 0.0 0.0 4.881796e-24 0.0 hamburger omelette paella peking_duck sushi 0 4.228594e-34 1.040205e-22 0.0 2.015325e-27 5.197965e-20 baklava2	Baklava	Baklava	100%
 The prediction is: baklava baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 1.0 0.0 0.0 0.0 0.0 hamburger omelette paella peking_duck sushi 0 0.0 0.0 0.0 0.0 0.0 baklava3	Baklava	Baklava	100%
 The prediction is: sushi baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.230102 0.000961 0.007983 0.111064 0.003353 hamburger omelette paella peking_duck sushi 0 0.035437 0.028991 0.000115 0.029355 0.552639 baklava4	Sushi	Baklava	23%

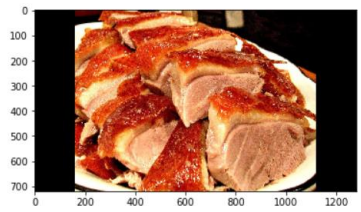
 The prediction is: beef_carpaccio baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.0 1.0 0.0 0.0 0.0 hamburger omelette paella peking_duck sushi 0 0.0 0.0 0.0 0.0 0.0 Beef Carpaccio1	Beef Carpaccio	Beef Carpaccio	100%
 The prediction is: caprese_salad baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 3.095096e-11 0.207873 0.791484 6.970740e-08 5.643580e-12 hamburger omelette paella peking_duck sushi 0 0.000004 0.000005 0.000001 0.000304 0.00033 Beef Carpaccio2	Caprese Salad	Beef Carpaccio	20.8%
 The prediction is: beef_carpaccio baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 1.979605e-15 0.999792 0.000208 4.008597e-14 7.585951e-19 hamburger omelette paella peking_duck sushi 0 2.664311e-12 7.006937e-08 8.688886e-10 7.453129e-08 1.502726e-09 Beef Carpaccio3	Beef Carpaccio	Beef Carpaccio	99.9%
 The prediction is: caprese_salad baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 5.148194e-07 0.334506 0.652488 0.000021 3.986448e-08 hamburger omelette paella peking_duck sushi 0 0.000317 0.000871 0.000842 0.001142 0.009813 Beef Carpaccio4	Caprese Salad	Beef Carpaccio	33.5%

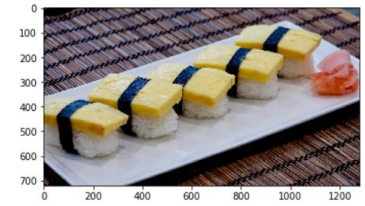
 The prediction is: caprese_salad baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.000002 0.090689 0.867033 0.000091 7.567428e-07 hamburger omelette paella peking_duck sushi 0 0.000542 0.001312 0.000449 0.000958 0.038923 Caprese Salad1	Caprese Salad	Caprese Salad	86.7%
 The prediction is: caprese_salad baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.0 4.758863e-23 1.0 4.851215e-34 0.0 hamburger omelette paella peking_duck sushi 0 2.130727e-35 3.253915e-38 0.0 7.431346e-37 7.617181e-31 Caprese Salad2	Caprese Salad	Caprese Salad	100%
 The prediction is: caprese_salad baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.0 2.133369e-14 1.0 1.844360e-31 0.0 hamburger omelette paella peking_duck sushi 0 4.771700e-29 1.330642e-27 2.617343e-27 1.661267e-28 4.015938e-21 Caprese Salad3	Caprese Salad	Caprese Salad	100%
 The prediction is: sushi baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.020859 0.024422 0.142275 0.071608 0.013322 hamburger omelette paella peking_duck sushi 0 0.081733 0.039325 0.023189 0.097745 0.485521 Caprese Salad4	Sushi	Caprese Salad	14.2%

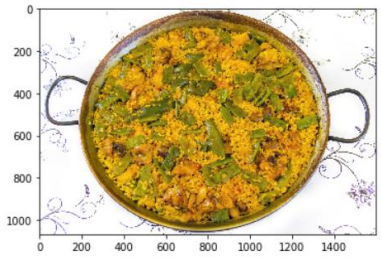
 The prediction is: cheesecake baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.054752 0.013445 0.018785 0.730613 0.023493 hamburger omelette paella peking_duck sushi 0 0.012451 0.013194 0.000213 0.104211 0.028842 Cheesecake1	Cheesecake	Cheesecake	73%
 The prediction is: cheesecake baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.000027 6.464099e-15 1.202621e-11 0.999973 2.234462e-07 hamburger omelette paella peking_duck sushi 0 2.104643e-11 3.157565e-10 1.888419e-18 5.159411e-08 1.053674e-08 Cheesecake2	Cheesecake	Cheesecake	99.9%
 The prediction is: cheesecake baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.000067 1.166572e-12 3.114253e-09 0.999931 7.816072e-07 hamburger omelette paella peking_duck sushi 0 3.189590e-09 6.100622e-09 1.069681e-14 5.222340e-07 2.547084e-07 Cheesecake3	Cheesecake	Cheesecake	99.9%
 The prediction is: creme_brulee baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.000062 0.00001 0.000007 0.002171 0.997503 hamburger omelette paella peking_duck sushi 0 0.000006 0.000142 0.000002 0.000091 0.000005 Cheesecake4	Creme Brulee	Cheesecake	0.2%

 The prediction is: cheesecake baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.015002 0.000017 0.000194 0.919648 0.05643 hamburger omelette paella peking_duck sushi 0 0.000622 0.000643 0.000002 0.005941 0.001502 Creme Brulee1	Cheesecake	Creme Brulee	5.6%
 The prediction is: creme_brulee baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 2.902942e-20 4.706788e-26 5.424137e-26 3.599955e-16 1.0 hamburger omelette paella peking_duck sushi 0 4.922488e-24 1.625576e-16 1.731268e-24 3.580457e-21 1.034322e-28 Creme Brulee2	Creme Brulee	Creme Brulee	100%
 The prediction is: creme_brulee baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.000055 0.000021 0.00001 0.000766 0.998771 hamburger omelette paella peking_duck sushi 0 0.000006 0.000268 0.000007 0.000093 0.000004 Creme Brulee3	Creme Brulee	Creme Brulee	99.8%
 The prediction is: creme_brulee baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 1.758796e-21 2.323400e-25 2.281356e-26 1.095075e-17 1.0 hamburger omelette paella peking_duck sushi 0 2.831422e-23 4.876775e-15 5.362388e-22 1.149160e-21 3.064515e-28 Creme Brulee4	Creme Brulee	Creme Brulee	100%

 The prediction is: omelette baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 2.481082e-16 3.259181e-24 1.580127e-28 2.074812e-29 1.536616e-26 hamburger omelette paella peking_duck sushi 0 1.596708e-17 1.0 2.304225e-13 1.740884e-24 6.627305e-28 Omelette1	Omelette	Omelette	100%
 The prediction is: baklava baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.954244 6.915393e-07 8.834334e-07 0.037834 0.000014 hamburger omelette paella peking_duck sushi 0 0.000318 0.00302 9.378999e-08 0.002498 0.002071 Omelette2	Baklava	Omelette	0.3%
 The prediction is: omelette baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.000016 2.811926e-11 7.696419e-11 8.314249e-10 1.790448e-09 hamburger omelette paella peking_duck sushi 0 0.000012 0.999652 0.00032 3.194324e-08 2.963850e-08 Omelette3	Omelette	Omelette	99.9%
 The prediction is: paella baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.000005 0.000211 0.000695 0.000004 0.000011 hamburger omelette paella peking_duck sushi 0 0.000346 0.013816 0.981313 0.000067 0.003531 Omelette4	Paella	Omelette	13.8%

 The prediction is: peking_duck baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 2.613398e-15 4.770123e-19 1.343646e-23 4.295921e-12 6.730648e-19 hamburger omelette paella peking_duck sushi 0 2.155273e-15 1.851826e-16 1.511041e-24 1.0 3.744553e-20 Peking Duck1	Peking Duck	Peking Duck	100%
 The prediction is: sushi baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.000305 0.01178 0.098725 0.000814 0.000028 hamburger omelette paella peking_duck sushi 0 0.001222 0.005981 0.007506 0.004137 0.869501 Peking Duck2	Sushi	Peking Duck	0.4%
 The prediction is: paella baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 2.660926e-09 0.002515 0.003293 7.758100e-08 0.000001 hamburger omelette paella peking_duck sushi 0 0.000002 0.004973 0.98807 0.000002 0.001144 Peking Duck3	Paella	Peking Duck	0%
 The prediction is: creme_brulee baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.001699 0.000013 0.000014 0.017761 0.953265 hamburger omelette paella peking_duck sushi 0 0.000213 0.000773 0.000044 0.026179 0.000038 Peking Duck4	Creme Brulee	Peking Duck	0.26%

 The prediction is: sushi baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.0 0.0 1.492772e-37 0.0 0.0 hamburger omelette paella peking_duck sushi 0 0.0 0.0 0.0 0.0 1.0 Sushi1	Sushi	Sushi	100%
 The prediction is: sushi baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.0 0.0 0.0 0.0 0.0 hamburger omelette paella peking_duck sushi 0 0.0 0.0 0.0 0.0 1.0 Sushi2	Sushi	Sushi	100%
 The prediction is: sushi baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 8.570119e-29 0.0 2.267303e-24 2.692936e-28 0.0 hamburger omelette paella peking_duck sushi 0 7.659413e-28 2.261598e-28 0.0 2.800073e-27 1.0 Sushi3	Sushi	Sushi	100%
 The prediction is: sushi baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 2.554842e-33 3.112863e-32 2.074459e-18 4.469786e-31 0.0 hamburger omelette paella peking_duck sushi 0 1.453662e-27 9.204025e-26 4.235850e-31 4.117107e-28 1.0 Sushi4	Sushi	Sushi	100%

 The prediction is: paella baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.0 0.0 4.230941e-38 0.0 0.0 hamburger omelette paella peking_duck sushi 0 0.0 1.208795e-31 1.0 0.0 0.0 Paella1	Paella	Paella	100%
 The prediction is: paella baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 0.0 5.049025e-35 4.690048e-33 0.0 1.089279e-27 hamburger omelette paella peking_duck sushi 0 4.268895e-37 2.638846e-13 1.0 0.0 8.355132e-36 Paella2	Paella	Paella	100%
 The prediction is: creme_brulee baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 4.021736e-35 8.645177e-30 1.155579e-27 7.906596e-27 0.999813 hamburger omelette paella peking_duck sushi 0 1.241398e-26 1.470694e-08 0.000187 2.402430e-29 1.437617e-32 Paella3	Creme Brulee	Paella	0%
 The prediction is: paella baklava beef_carpaccio caprese_salad cheesecake creme_brulee \ 0 3.945208e-22 3.686695e-13 2.808527e-10 4.799411e-22 1.815188e-20 hamburger omelette paella peking_duck sushi 0 1.666299e-15 7.585689e-09 1.0 6.807422e-17 5.700715e-10 Paella4	Paella	Paella	100%

The model was able to correctly predict 27 out of 40 images (67.5% accuracy), which was lower than the test accuracy of the model, 74.6%. This could have been due to a few reasons.

The first is got to do with the plating of the dish. In some of the above examples, the plating was not very obvious, like when the food pieces were stacked on top of each other or side by side, they were classified as sushi (for baklava4, pekingduck2 and caprese salad4 examples), perhaps because the positioning of the food items resembled sushi, but not the colour and the shape of the food

In addition, the predominant colours of the foods also mattered in determining whether the model could correctly classify the food. For example, hamburger1 is classified as an omelette as there is cheese in the burger. Likewise, beef carpaccio4 is classified as caprese salad since there are yellow and green vegetables and fruits, which predominantly make up the ingredients of the salad.

The model classifies sushi the best, with all 4 images of sushi labelled correctly with 100% certainty. The model classifies peking duck the worst with 3 incorrect predictions with very low certainties for those classifications (below 5%).

Amongst the 13 incorrect predictions, 10 of them have a correct label certainty of 20% or lower, indicating that the model does very poorly in classifying these foods which may be caused by the model being unable to pick up features from these food categories even after it has been trained on them or the validation and testing images having different features (background lighting, size and position of the food, zoom on the food object etc.) from the training dataset and thus the model is unable to classify the test images correctly. Perhaps this could be resolved by having another step to randomly shuffle the order of the images or to remove and replace the “dirty data”.

Therefore, the VGG Model was not able to yield higher accuracies for all 10 food categories and this is perhaps something that I could work on to improve the certainty of the model.

7. Summary

After wrapping up the testing of my best model, I decided to reflect on some of the improvements I could make when training models, which could ultimately lead to even better models and more findings for this assignment.

Firstly, in the model training phase, perhaps I could have tweaked more model hyperparameters. For example, I could change the learning rate for the SGD with momentum models, instead of just solely changing the learning rate and observe if this would lead to better test accuracies. For the pretrained models, I could have experimented with more pretrain architectures if I had more time, such as DenseNet, Inception and Xception. I could also have tested out changing the optimizer for the pretrain models to Adam or SGD instead of solely using the RMSprop optimizer.

For some models, maybe I could have removed data augmentation and observed how this affects the test accuracy of the models as well as when it overfits, as well as the shape of the accuracy and loss curves. I could also experiment with drastic changes to the data augmentation and perhaps change the hyperparameters one by one to observe if these result significant changes as for some models, I change multiple hyperparameters at once and thus am not really able to tell which parameter made the most significant contribution in that particular change.

Secondly, I could have also paid attention to factors that directly affect the accuracy of the models in general if I had more time or computing resources. Some of my peers discovered that there were some “dirty data” or incorrect images in the food-101 dataset. Perhaps removing such images and replacing them with appropriate ones could improve the accuracy of all my models. I could also choose to train the models with images of a larger size, say 200x200 or 250x250 to achieve better accuracies, granted that I had more time or access to better computing resources. The dataset is also rather small so maybe I could propose that the assignment made use of a larger image dataset, resulting in better accuracies in the models.

Thirdly, after validating that VGG16 can have better accuracies as compared with ResNet50, I could have just focused on improving the test accuracy of VGG16 since it is already proven in external research that it does better than ResNet50.

References

Bushaev, V. (22 October, 2018). Adam — latest trends in deep learning optimization. *Towards Data Science*. Retrieved from https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c

Kaiming He, X. Z. (n.d.). Deep Residual Learning for Image Recognition. (C. V. Foundation, Ed.) *Microsoft Research*. Retrieved from https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/He_Deep_Residual_Learning_CVPR_2016_paper.pdf

Mao, Y. D. (2016). Deep Learning Based Food Recognition. *Stanford University*. Retrieved from http://cs229.stanford.edu/proj2016/report/YuMaoWang-Deep%20Learning%20Based%20Food%20Recognition-report.pdf

Methuku, J. (15 May, 2020). How did the Deep Learning model achieve 100% accuracy? *Towards Data Science*. Retrieved from https://towardsdatascience.com/how-did-the-deep-learning-model-achieve-100-accuracy-6f455283c534

Vasu, B. K. (December, 2018). Visualizing Resiliency Of Deep Convolutional Network Visualizing Resiliency Of Deep Convolutional Network Interpretations For Aerial Imagery Interpretations For Aerial Imagery. *Rochester Institute of Technology (RIT Scholar Works)*. Retrieved from https://scholarworks.rit.edu/cgi/viewcontent.cgi?article=11100&context=theses