# DUBLIN CITY UNIVERSITY

## ELECTRONIC AND COMPUTER ENGINEERING

## EE544 - Computer Vision

### Assignment 1

*Author*

Michael Lenehan    michael.lenehan4@mail.dcu.ie

Student Number:    15410402

08/04/2020

# Declaration

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others, save and to the extent that such work has been cited and acknowledged within the text of my work. I understand that plagiarism, collusion, and copying are grave and serious offences in the university and accept the penalties that would be imposed should I engage in plagiarism, collusion or copying. I have read and understood the Assignment Regulations set out in the module documentation. I have identified and included the source of all facts, ideas, opinions, and viewpoints of others in the assignment references. Direct quotations from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the source cited are identified in the assignment references. This assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study.

I have read and understood the DCU Academic Integrity and Plagiarism at `https://www4.dcu.ie/sites/default/files/policy/1%20-%20integrity_and_plagiarism_ovpaa_v3.pdf` and IEEE referencing guidelines found at `https://loop.dcu.ie/mod/url/view.php?id=448779`.

Signed: _____          Date: 08/04/2020

Michael Lenehan

# Title

Subtitle

**Michael Lenehan**

**Abstract**

# Contents

# 1   Introduction

Using Keras, two separate image classification problems will be solved. The first of these problems involves developing a classifier for a subset of the ImageNet dataset. This classifier is based on the VGG-16D neural network.

The second problem involves performing transfer learning on the ResNet50 CNN. The lower 141 layers of the base model must not be modified, with only the upper 33 layers being modified. This is done using the Tensorflow Keras ResNet50 model, and the freeze functionality which is built in.

Google Colab is used as the development environment for this assignment. With access to a GPU, training can be accelerated. Colab allows for the importing of data from a mounted Google Drive volume, the importing of Python libraries, and the saving and exporting of the trained models.

# 2   Question 1

The aim of section 1 of this assignment is to create an "ImageNet" style classifier. The provided dataset contains pictures divided into 6 classes, which are "Church", "English Springer", "Garbage Truck", "Gas Pump", "Parachute", and "Tench". The data has been provided in three subsets, training, validation, and test subsets. This removes the need to do test-train splitting. The dataset is relatively small, with a size of 1350 images per class, totaling 8100 images, compared to the 14 million images in the original "ImageNet" dataset.

The provided CNN structure is based on the "VGG" neural network. The base network has a total of 9 layers. The input images are of size 224x224, and the CNN must output a classification decision based on the the image classes. This section of the assignment is divided into five sections. The first involves building the base CNN. Subsequent sections build on top of this, adding dropout, data augmentation, and batch

normalization to the nerual network in order to assess the performance impact of these features. The final section involves creating a final version of the neural network, which has the highest achievable performance.
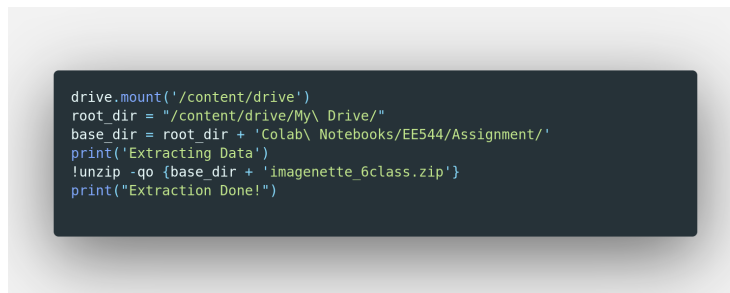
## 2.1 Part a

### 2.1.1 Introduction

For Part a, the given "VGG-lite" CNN structure (Figure 1) must be implemented. This model will act as a base for each of the subsequent parts in section 1. The given structure has 9 layers, takes the input images, which are of size 224x224, and outputs a classification based on one of the six classes.

| Layer | Number of output filters |
|---|---|
| 2D (3x3) convolution layer | 32 |
| 2D (3x3) convolution layer | 32 |
| 2D (2x2) Pooling | |
| | |
| 2D (3x3) convolution layer | 64 |
| 2D (3x3) convolution layer | 64 |
| 2D (2x2) Pooling | |
| | |
| Flatten | |
| Fully-connected NN layer | 512 |
| Fully-connected NN layer (Prediction) | Number of classes |
| | |

**Convolution layers** should use Xavier (glorot) uniform kernel initialization, a dilation rate of 1, 'same' padding, strides of (height,width)= (1,1) and 'relu' activation. Note the first layer will need to account for the input shape of the data been examined.

**Pooling layers** will use (2x2) max pooling with (ie a pool size of (vertical, horizontal) = (2,2)).

**Fully-connected layers** will also use Xavier (glorot) uniform kernel initialization and 'relu' activations.

Figure 1: Given VGG-lite Structure

### 2.1.2 Design

The first steps in implementing this system is loading the data into Google Colab. This functionality is available through the "google.colab" "drive" library. The required directory must be mounted to the session, and the zipped data must be extracted. The code implementation in Figure 2 shows this functionality.

```
drive.mount('/content/drive')
root_dir = "/content/drive/My\ Drive/"
base_dir = root_dir + 'Colab\ Notebooks/EE544/Assignment/'
print('Extracting Data')
!unzip -qo {base_dir + 'imagenette_6class.zip'}
print("Extraction Done!")
```

Figure 2: Extract Data from ZIP File

To load data from a directory, Tensorflow Keras provides the "flow_from_directory" method. This method takes a directory, image size, and class mode. For the given dataset, the directories are "train", "validation", and "test". The target size for each of the images is 224x224, and the class mode is "categorical", as there are multiple classes of output. For the test data, the class mode must be set to "None", and the data must not be shuffled, as specified by the arguement "shuffle=False".

```
trainGenerator = trainDataGenerator.flow_from_directory(
    directory=r"./train/",
    target_size=(224,224),
    batch_size=50,
    class_mode="categorical",
)

validGenerator = validDataGenerator.flow_from_directory(
    directory=r"./validation/",
    target_size=(224,224),
    batch_size=50,
    class_mode="categorical",
)

testGenerator = testDataGenerator.flow_from_directory(
    directory=r"./test/",
    target_size=(224,224),
    batch_size=1,
    class_mode=None,
    shuffle=False
)
```

Figure 3: Load in Data from Directory

To construct the neural network, the "model.add" method is called to add layers. Layers are added as previously specified. The 2D convolutional layers have 32 output filters, a 3x3 kernel, a stride of 1, padding "same", "relu" activation, and "glorot_uniform" kernel initialization. There is a difference with the input layer, which must specify the input shape in the form (rows, columns, channels). In this case, the input shape is 244x244x3. These layers are added as Keras "Conv2D" layers.

Following the 2D convolutional layer is a 2D pooling layer. This is implemented with the "MaxPooling2D" function, which takes a pool size as an input. In this case, the specified pool size is 2x2. There are two additional 2D convolutional layers, with 64 output filters, followed by another 2D pooling layer.

A Flatten layer is added before the fully connected layers of the model. The Keras "Flatten" function takes no input arguements.

Each of the final Dense layers use the "glorot_uniform" kernel initializer. The first Dense layer has 512 output units and "relu" activation. The final layer must have 6

output units, as there are 6 total classification classes. It uses "softmax" activation. This must also be used, as ReLU has very poor performance as an output layer.

The finalized neural network is shown in Figure 4.

```python
model = tf.keras.Sequential()
model.add(layers.Conv2D(
    filters=32, kernel_size=(3,3), input_shape=(224,224,3),
    strides=(1,1), padding='same',
    dilation_rate=(1,1), activation='relu',
    kernel_initializer='glorot_uniform',
    kernel_regularizer=regularizers.l2(0.01)
))
model.add(layers.Conv2D(
    filters=32, kernel_size=(3,3), strides=(1,1), padding='same',
    dilation_rate=(1,1), activation='relu',
    kernel_initializer='glorot_uniform',
    kernel_regularizer=regularizers.l2(0.01)
))
model.add(layers.MaxPooling2D(
    pool_size=(2,2)
))
model.add(layers.Conv2D(
    filters=64, kernel_size=(3,3), strides=(1,1), padding='same',
    dilation_rate=(1,1), activation='relu',
    kernel_initializer='glorot_uniform',
    kernel_regularizer=regularizers.l2(0.01)
))
model.add(layers.Conv2D(
    filters=64, kernel_size=(3,3), strides=(1,1), padding='same',
    dilation_rate=(1,1), activation='relu',
    kernel_initializer='glorot_uniform',
    kernel_regularizer=regularizers.l2(0.01)
))
model.add(layers.MaxPooling2D(
    pool_size=(2,2), strides=None
))
model.add(layers.Flatten())
model.add(layers.Dense(
    units=512,
    activation='relu',
    kernel_initializer='glorot_uniform',
    kernel_regularizer=regularizers.l2(1e-5)
))
model.add(layers.Dense(
    units=6,
    activation='softmax',
    kernel_initializer='glorot_uniform'
))
```

Figure 4: Question 1 Base Model

### 2.1.3  Testing

There are a number of hyperparameters which may be tuned for this model. Each "Conv2D" and "Dense" layer have activity regularizers, kernel regularizers, and bias regularizers which can be modified in order to tune the performance of the neural network.

Manual tuning was performed for these hyperparameters for various input values. This was done due to a lack of understanding of the "kerastuner" tool, which allows for automated hyperparameter selection.In order to shorten the amount of time taken for testing, early stopping was implemented, which stops training when there has been no change in the validation loss in a set number of epochs. 5 epochs was chosen as the limit for this.

Kernel regularizers were the hyperparameters which were utilised for the purpose of tuning. A number of values for the hyperparameters were tested, with the validation accuracy used as the metric by which they were compared.

First, the kernel regularizer for the final "relu" activated Dense layer was tested, this is labelled as "Regularizer Value" within the following table. The regularizer chosen for the purpose of this section was the "l2" "weight decay" regularizer. The kernel regularizer for each other layers was set to a value of 0.01, while the Dense layer regularizer was modified in order to obtain the highest possible accuracy.

Table 1: Kernel Regularizer Hyperparameter Tuning (Final "ReLU" Dense Layer)

| Regularizer Value | Validation Accuracy |
|:---:|:---:|
| 0.00005 | 69.17 |
| 0.00001 | 71.83 |
| 0.0005 | 76 |
| 0.00025 | 77 |
| 0.0001 | 70 |

As given by the output listed above, the regularizer value of 0.00025 gave the highest validation accuracy, and as such was used moving forward. The kernel regularizer value of the other layers were then tested, as shown in the table below.

Table 2: Kernel Regularizer Hyperparameter Tuning (Other Layers)

| Regularizer Value | Validation Accuracy |
|-------------------|---------------------|
| 0.001             | 74                  |
| 0.005             | 75                  |
| 0.0005            | 73                  |
| "None"            | 77                  |

As the optimal value is shown to be "None", i.e. no regularizer is used, this is the option used for the model. While these options are the optimum arrived at using manual hyperparameter tuning, there may be more optimal choices.

Another hyperparameter which must be chosen is the learning rate of the chosen optimizer. For this section, the "Adam" optimizer was chosen, as the hyperparameter tuning is relatively easy, especially when being done manually.

Table 3: Learning Rate Hyperparameter Tuning

| Learning Rate | Validation Accuracy |
|---------------|---------------------|
| 0.01          | 16.67               |
| 0.001         | 71.33               |
| 0.0001        | 74.83               |
| 0.00001       | 74.33               |

The learning rate value of 0.0001 was selected, as it gave the highest value for validation accuracy. With this value, the final model could be tested.

For the purposes of training, the model was set to train for 30 epochs, however, the value for early stopping patience was increased from 5 to 7, in order to prevent overfit-

ting without stopping too early to train correctly.

### 2.1.4   Results

The correct output for the neural network structure was printed using the "model.summary" function in keras, as seen in Figure 5. This shows the output shape of the final layer as 6, which is the number of possible classifications, along with all of the specified outputs in all other layers.

```
Layer (type)                    Output Shape              Param #
=================================================================
conv2d_8 (Conv2D)               (None, 224, 224, 32)      896
_____
conv2d_9 (Conv2D)               (None, 224, 224, 32)      9248
_____
max_pooling2d_4 (MaxPooling2    (None, 112, 112, 32)      0
_____
conv2d_10 (Conv2D)              (None, 112, 112, 64)      18496
_____
conv2d_11 (Conv2D)              (None, 112, 112, 64)      36928
_____
max_pooling2d_5 (MaxPooling2    (None, 56, 56, 64)        0
_____
flatten_2 (Flatten)             (None, 200704)            0
_____
dense_4 (Dense)                 (None, 512)               102760960
_____
dense_5 (Dense)                 (None, 6)                 3078
=================================================================
Total params: 102,829,606
Trainable params: 102,829,606
Non-trainable params: 0
```

Figure 5: Model Summary

It is clear from both Figures 6 and 7 that overfitting does not occur within this model. The validation accuracy of the model reaches approximately 80%, however, the loss value reaches approximately 2.00. The final output, as shown in Figure 8 shows a final validation accuracy of 79.50%.
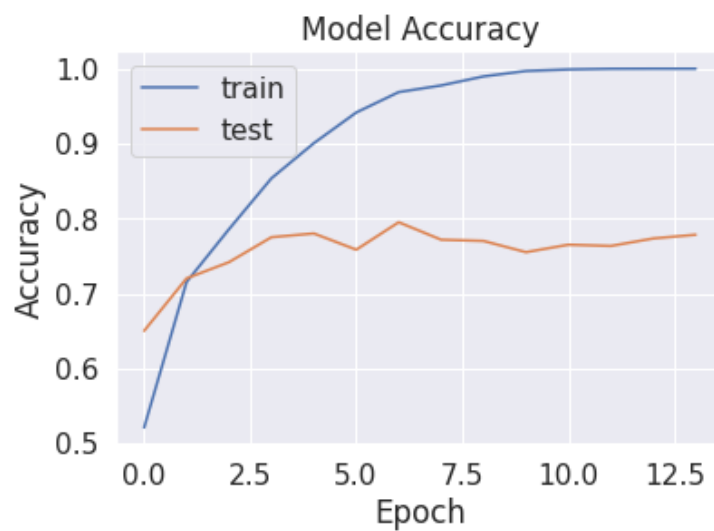
Figure 6: Validation and Test Acccuracy

While the validation accuracy is trending upwards, it is possible that better tuning of the hyperparameters would give a higher validation accuracy.
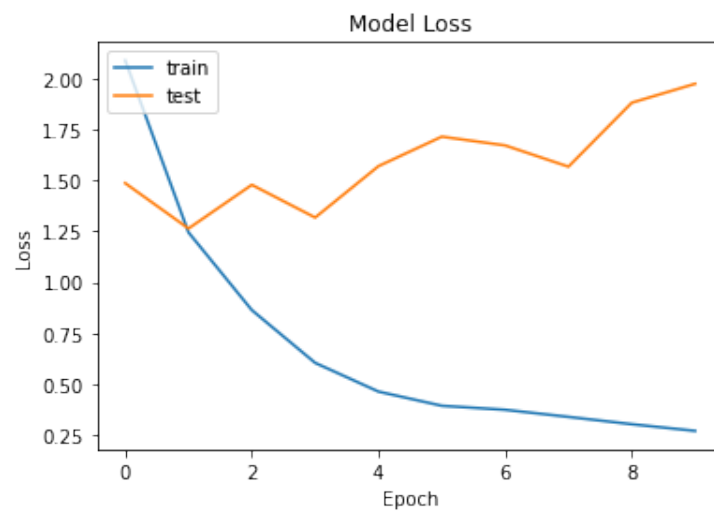


Figure 7: Validation and Test Loss

The validation loss continues to rise. It is again possible that this could be attributed to incorrect hyperparameter tuning.

The final output shows an average precision and recall of 81%, with the highest values attributed to the "Parachute" class, and the lowest values attributed to the "English Springer" class.



```
12/12 [==============================] - 2s 157ms/step - loss: 0.8675 - accuracy: 0.7950
CNN Accuracy: 79.50%
CNN Error: 20.50%
300/300 [==============================] - 2s 6ms/step
                  precision    recall  f1-score   support

         Church       0.84      0.82      0.83        51
English Springer       0.66      0.75      0.70        44
  Garbage Truck       0.86      0.77      0.81        56
       Gas Pump       0.72      0.77      0.74        47
      Parachute       0.92      0.94      0.93        49
          Tench       0.88      0.83      0.85        53

       accuracy                           0.81       300
      macro avg       0.81      0.81      0.81       300
   weighted avg       0.82      0.81      0.81       300
```

Figure 8: Model Testing Results

The final confustion matrix, as shown in Figure 9, shows the number of correctly identified samples in testing, and as previously noted, the class with the highest precision and recall is "Parachute" labelled as "4" within the matrix.
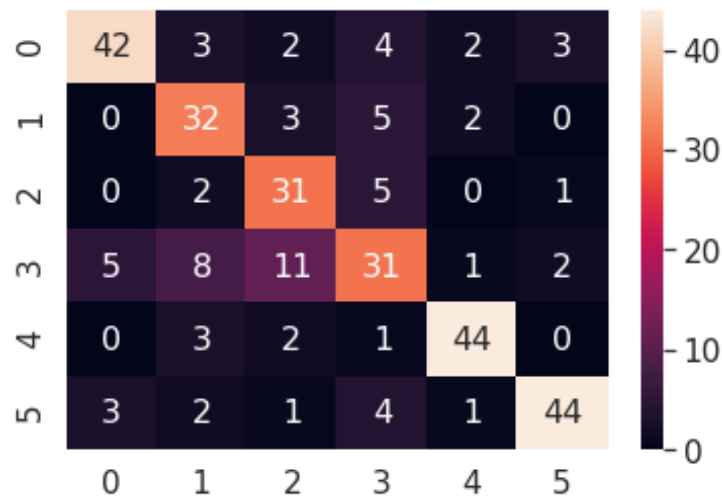


Figure 9: Confusion Matrix

14

Using early stopping, the model ran for a total of 14 epochs. The training time for the model was 436.109 seconds. Given that 1 hour of training on an Nvidia GPU produces approximately 0.25lbs of $CO_2$ equivalent emissions, this training has an emission output of 0.03lbs $CO_2$ equivalent emissions.

## 2.2 Part b

### 2.2.1 Introduction

The aim of Part b is to apply "Dropout" to the neural network designed in Part A. The hyperperameters, i.e. the dropout rate, needs to be tuned in order to obtain an optimal performance from the network.

### 2.2.2 Rational

Dropout is a technique used in Neural Networks in order to prevent over-fitting. By dropping certain neurons at random during testing, the network must learn features which are considered "robust". This will lead to a greater test accuracy.

### 2.2.3 Design

### 2.2.4 Testing

For the purposes of testing, the dropout rate was the hyperparameter being tuned. The dropout rate of the first two dropout layers was set to 0.1 as a baseline measurement, while the rate of the dropout rate before the final "Dense" layer was modified. This resulted in an optimum value of XX given for the final dropout layer, as shown in the table below.

Table 4: Dropout Hyperparameter Tuning

| Dropout Rate | Validation Accuracy |
|:---:|:---:|
| 0.55 | |
| 0.5 | 78.5 |
| 0.25 | 74.7 |
| 0.1 | 76.17 |

Setting this dropout rate, the dropout rates of the other two dropout layers are changed simultaneously.

Table 5: Dropout Hyperparameter Tuning

| Dropout Rate | Validation Accuracy |
|:---:|:---:|

### 2.2.5 Results

## 2.3 Part c

### 2.3.1 Introduction

The aim of this section is to demonstrate the effects of data augmentation on the performance of the neural network.

### 2.3.2 Rational

Data augmentation is a technique typically used in order to increase the size of small datasets. Having more images, with unique features allows for the training of a more robust neural network. By rotating, mirroring, and scaling images the neural network can become more robust in real use cases to these variances in input image.

### 2.3.3 Design

### 2.3.4 Testing

### 2.3.5 Results

## 2.4 Part d

### 2.4.1 Introduction

This section aims to add batch normalization to the neural network designed in Part A. The effect on the neural network should be investigated in order to determine if it increases the performance of the network.

### 2.4.2 Rational

Batch normalization works to alleviate the effects of covariate shift within a CNN. Batch normalization allows for the use of higher training rates, and also the use of less dropout within the network, due to the regularization effects it has.

### 2.4.3 Design

### 2.4.4 Testing

### 2.4.5 Results

## 2.5 Part e

### 2.5.1 Introduction

Part e aims to combine the previous sections, applying a combination of dropout, data augmentation and batch normalization to the baseline model from Part a. The aim is to develop a CNN with the optimal accuracy.

### 2.5.2 Rational

### 2.5.3 Design

### 2.5.4 Testing

### 2.5.5 Results

# 3 Question 2

The aim of section 2 of this assignment is to implement "Transfer Learning" using a Resnet-50 model. The model must be adapted to the "Food-101" classification task, classifying images of foods which fall into the following classes; Chicken Curry, Hamburger, Omelette, and Waffles. The dataset is yet again pre-divided into training, validation, and test sets, therefore requiring no pre-processing in order to be divided.

The Keras "ResNet50" model is used for this section of the assignment. This model has 174 layers, as is specified in the assignment brief.

## 3.1 Part a

### 3.1.1 Introduction

The aim of part A is to implement the optimisation of the classification task using the pretrained ResNet50 model. Fine-tuning based transfer learning is to be utilised in order to apply the pretrained model to the required task, i.e. classifying the Food-101 dataset.

**3.1.2 Rational**

**3.1.3 Design**

**3.1.4 Testing**

**3.1.5 Results**

## 3.2 Part b

**3.2.1 Introduction**

**3.2.2 Rational**

**3.2.3 Design**

**3.2.4 Testing**

**3.2.5 Results**

# 4 Conclusion