Computer Vision

Subhasish Basak (MDS201803)

Final Project 2 : Image Classification using Transfer Learning

April 29, 2020

1 Introduction

In this Assignment we implement **Transfer Learning** by using a pre-trained network for feature extraction and by fine-tuning a pre-trained network, for a smaller dataset. The main steps are as follows:

- We use the **ResNet101** model pre-trained on the *ImageNet* database as feature extractor and train the model by adding additional layers for classification.
- In the next step we fine-tune the **ResNet101** model by unfreezing a portion of the pre-trained network.

The link to my **Colab Notebook** is as follows: Click to view Colab Notebook

2 Data Loading & Preprocessing

We are using the **ResNet101** model which is already pre-trained on the *ImageNet* database. For validation, fine tuning the hyperparameters and testing purpose we use a smaller database. This dataset contains images from 15 categories and 480 images from each category. We split the data into *Train : Validation : Test* in the ratio 56.25 : 18.75 : 25. In other words corresponding to each category there are 270 Train images, 90 Validation images and 120 Test images. The 15 categories are as follows:

- 'playground', 'bedroom', 'restaurant', 'highway', 'bridge', 'airport_terminal', 'dining_room', 'Cathedral_outdoor', 'coral_reef', 'skyscraper', 'forest_path', 'house', 'market_outdoor', 'beach', 'mountain'

We pre-process the data using the following steps :

- **Transforming into gray-scale** : Since each pixel value in the images are in the range $[0, 255]$. We transform the images into Gray-scale by dividing each pixel value by 255.
- **Resizing** : We also need to reshape the training images in order to feed it into the network. We reshape the images using *ImageDataGenerator* library of Keras and store them in batches. We use the following configuration:
  - **Batch size** : 32
  - **Image height** : 224
  - **Image width** : 224

3 The Model Architecture

We call the pretrained ResNet101 model and print its model summary as follows:

In [1]:
```
base_model = ResNet101(include_top=False
                        , input_shape=IMG_SHAPE
                        , pooling='avg'
                        , weights="imagenet")
base_model.summary()
```

```
Model: "resnet101"
__________________________________________________________________________
Layer (type)                     Output Shape          Param #
Connected to
==========================================================================
input_1 (InputLayer)             (None, 224, 224, 3)   0

__________________________________________________________________________
conv1_pad (ZeroPadding2D)        (None, 230, 230, 3)   0
input_1[0][0]
__________________________________________________________________________
conv1_conv (Conv2D)              (None, 112, 112, 64) 9472
conv1_pad[0][0]

....
....

__________________________________________________________________________
conv5_block3_out (Activation)    (None, 7, 7, 2048)    0
conv5_block3_add[0][0]
__________________________________________________________________________
avg_pool (GlobalAveragePooling2 (None, 2048)           0
conv5_block3_out[0][0]
==========================================================================
Total params: 42,658,176
Trainable params: 0
Non-trainable params: 42,658,176
```

We notice that, since we are using a ***pre-trained*** model, the total number of trainable parameters is 0 compared to total number of parameters in the model which is $42,658,176$.

3.1 Adding Classification Layer

Next we add a classification layer on the top of the network. Here we add a 128 node dense layer followed by a Dense layer with the **Softmax** activation function, with the following configuration:

- **Optimizer**: RMSprop
- **Loss**: Categorical Cross-Entropy
- **Learning Rate**: 0.001

After adding this layer we have the model summary as follows:

In [2]:
```
model = keras.Sequential([base_model])
model.add(Dense(128))
model.add(Dropout(0.3))
model.add(Dense(15, activation = "softmax"))
base_learning_rate = 0.001
model.compile(optimizer=keras.optimizers.RMSprop(lr=base_learning_rate),
              loss = "categorical_crossentropy",
              metrics=['accuracy'])
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
resnet101 (Model)            (None, 2048)              42658176

_________________________________________________________________
dense_1 (Dense)              (None, 128)               262272

_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0

_________________________________________________________________
dense_2 (Dense)              (None, 15)                1935
=================================================================
Total params: 42,922,383
Trainable params: 264,207
Non-trainable params: 42,658,176
_________________________________________________________________
```

As we can see, due to the addition of the new dense classification layer the total number parameters increased to $42,922,383$ out of which $264,207$ are trainable.

3.2 Fine tuning

After running the model with some basic choices of parameters we observed that, training the newly added layers only has performed very poorly on the validation side. To improve the validation accuracy we try **Unfreezing** the top layers of the model and train top ***k*** layers of the model along with the newly added layers.
In our base model there are 346 layers out of which we unfreeze the top 146 layers. We keep the bottom 200 layers non-trainable. After partially unfreezing the layers our model architecture is as follows:

In [3]:
```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
resnet101 (Model)            (None, 2048)              42658176
_________________________________________________________________
dense_1 (Dense)              (None, 128)               262272
_________________________________________________________________
dropout_1 (Dropout)          (None, 128)               0
_________________________________________________________________
dense_2 (Dense)              (None, 15)                1935
=================================================================
Total params: 42,922,383
Trainable params: 27,548,175
Non-trainable params: 15,374,208
_________________________________________________________________
```

As we can see due to partial unfreezing the total number of trainable parameters increased to $27,548,175$.

4 Results

We First present the results obtained from the simple feature extraction model. Next we fine tune our model based on the validation accuracy and then present the result obtained from the fine tuned model. For the simple feature extraction model we have found the following configuration well performing:

- **No. of epochs** : 20
- **Validation steps** : 30
- **Steps per epoch** : 50

4.1 Base model evaluation

In this part we present the results obtained by evaluating the Base model. As described above with **base_model.trainable** = ***False*** i.e we train only the parameters of the newly added layers we get the following results after 20 epochs. Here are the outputs:

```
Epoch 19/20
50/50 [==============================] - 43s 864ms/step -
loss: 0.1758 - accuracy: 0.9469 - val_loss: 4.7894 - val_accuracy: 0.0675
Epoch 20/20
50/50 [==============================] - 46s 927ms/step -
loss: 0.1918 - accuracy: 0.9356 - val_loss: 5.1663 - val_accuracy: 0.0688
```

The Test loss for this case is 4.716. We summarize the results as follows:

- ⋆ **Training accuracy** : 93.56
- ⋆ **Validation accuracy** : 0.068
- ⋆ **Test accuracy** : 0.064

4.2 Fine-tuned network evaluation

We continue with the base trained model as above. Next we unfreeze top 146 layers of it and train the model on the whole trained data for another 30 epochs. Here are the results:

```
Epoch 49/50
50/50 [==============================] - 49s 983ms/step -
loss: 0.0358 - accuracy: 0.9924 - val_loss: 6.0550 - val_accuracy: 0.7615
Epoch 50/50
50/50 [==============================] - 48s 968ms/step -
loss: 0.0162 - accuracy: 0.9962 - val_loss: 1.6856 - val_accuracy: 0.8790
```

The Test loss for this case is 0.543. We summarize the results as follows:

⋆ **Training accuracy** : 99.62

⋆ **Validation accuracy** : 87.9

⋆ **Test accuracy** : 88.95

This is the **best model** obtained through the experiments.

4.3 Full trained model evaluation

We also tried training the whole model with our training data. By setting **base_model.trainable = *True*** and we can learn the parameters for all the layers. In other words we unfreeze all the layers of the network. We get the following results after 50 epochs.

```
Epoch 49/50
50/50 [==============================] - 51s 1s/step -
loss: 0.0344 - accuracy: 0.9937 - val_loss: 2.7089 - val_accuracy: 0.8340
Epoch 50/50
50/50 [==============================] - 50s 1s/step -
loss: 0.0330 - accuracy: 0.9919 - val_loss: 0.8462 - val_accuracy: 0.8896
```

The Test loss for this case is 0.356. The model yields the following results:

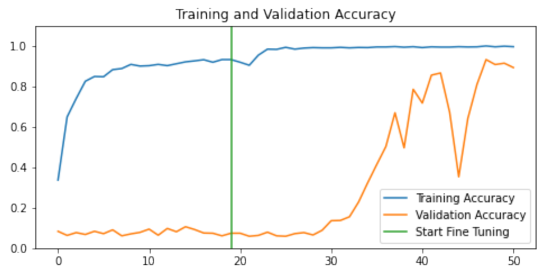
⋆ **Training accuracy** : 99.19

⋆ **Validation accuracy** : 88.96

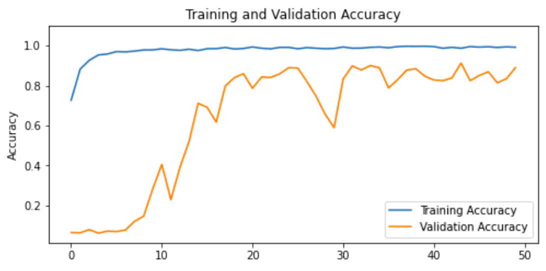
⋆ **Test accuracy** : 89.1

5 Plots & Visualization

We plot the **Training accuracy** vs the **validation accuracy** for the partially and fully unfreezed network as follows: As we can see the validation accuracy has increased after fine tuning the network.



(a) Fine tuned network



(b) Fully trained network

The vertical green line denotes the point from where we started training the fine tuned network.

6 Extra Credit : Implementation with VGG

In this section we illustrate the same problem with the implementation of a different network architechture, for e.g. here we have used the **VGG** network architecture imported from keras library.

6.1 Configurations

With the same data and the same additional layers as described in Section 3.1, we apply the following configuration to our network to run it :

- **Optimizer**: RMSprop
- **Loss**: Categorical Cross-Entropy
- **Learning Rate**: 0.001
- **No. of epochs** : 20
- **Validation steps** : 30
- **Steps per epoch** : 50

6.2 Results & outputs

The outputs corresponding the **base model** are here:

```
Epoch 19/20
50/50 [==============================] - 44s 870ms/step -
loss: 1.3124 - accuracy: 0.6856 - val_loss: 1.6620 - val_accuracy: 0.7388
Epoch 20/20
50/50 [==============================] - 42s 832ms/step -
loss: 1.2854 - accuracy: 0.6956 - val_loss: 1.3306 - val_accuracy: 0.7094
```

The Test loss for this case is 1.21. We summarize the results as follows:

⋆ **Training accuracy** : 69.56

⋆ **Validation accuracy** : 70.94

⋆ **Test accuracy** : 72.39

Now we fine tune the network by unfreezing the top 10 layers of the network and training the model for another 30 epochs. The results are as follows:

```
Epoch 49/50
50/50 [==============================] - 45s 892ms/step -
loss: 0.3308 - accuracy: 0.9250 - val_loss: 0.2242 - val_accuracy: 0.8302
Epoch 50/50
50/50 [==============================] - 50s 997ms/step -
loss: 0.2553 - accuracy: 0.9425 - val_loss: 0.4708 - val_accuracy: 0.8662
```

The Test loss for this case is 0.823. We summarize the results as follows:

⋆ **Training accuracy** : 94.25

⋆ **Validation accuracy** : 86.62

⋆ **Test accuracy** : 88.02

6.3 Conclusion

Thus we can see that in case of the **VGG** model also the network gains significant accuracy for validation and test part. The performance of the network was remarkably well before fine-tuning both in case of training and testing.

6.4 Plots & visualization

The Plots corresponding the Training and validation accuracy (before & after fine-tuning):

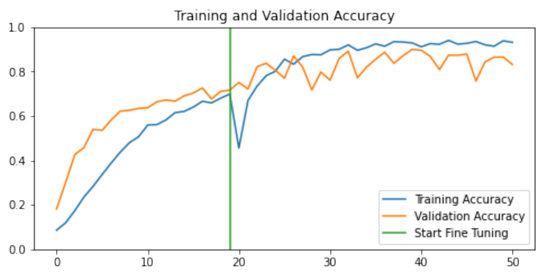


Figure 1: Fine tuned VGG network

6.5 Comparison b/w ResNet101 & VGG

As we can see from the plots and the results, the overall performance of the **VGG** network was better than the **ResNet** network as a base model. Although the training accuracy for the **VGG** was 67.02% which is much lower than the training accuracy of the **ResNet** model which is 93.25%, but the **VGG** model performed far better than its peer for validation and testing.
In case of the fine-tuned model both the networks gained significant accuracies for both validation and testing. The overall performance after fine tuning was better for **ResNet** compared to **VGG**.

7 References

- https://www.tensorflow.org/tutorials/images/transfer_learning
- https://www.tensorflow.org/tutorials/load_data/images