Computer Vision

Chennai Mathematical Institute

Project- Part II

Notebook Link: (**Main Model -ResNet101**) (**Extra Credit - InceptionResNetV2**)

Task

Here our task is to perform transfer learning by using a pre-trained network for feature extraction and by fine-tuning a pre-trained network, for a smaller dataset. Specifically,

- To use the ResNet101 model pre-trained on the ImageNet database as feature extractor, add additional layers for classification, train the model and evaluate it.
- To fine-tune the ResNet101 model, unfreezing a portion of the pre-trained network, train the resulting model, evaluate it.

Transfer Learning

- Method for reusing a model trained on a related predictive modeling problem.
- The idea behind it is, if a model is trained on a large and general enough dataset, this model will effectively serve as a generic model of the visual world.
- Reduces computation most of the times
- Increases the performence

Data

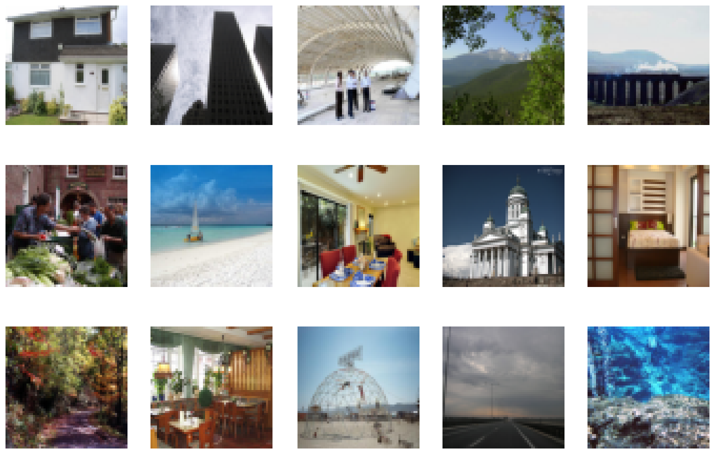
The Dataset Consists of -

- 5400 training Images
- 1800 test Images
- 15 scene categories
- 360 training images for each category
- 120 test images for each category

Here are all 15 categories, 'house', 'skyscraper', 'airport_terminal', 'mountain', 'bridge', 'market_outdoor', 'beach', 'dining_room', 'Cathedral_outdoor', 'bedroom', 'forest_path', 'restaurant', 'playground', 'highway', 'coral_reef'

Visualization:

Here we visualize image from each of the categories in the above category order -



Procedure:

- We use Keras to read images.
- First we call the ImageDataGenerator class of Keras.
- We fix the values of the arguments. For e.g. - rescale = 1.0 /255

  Similarly We can set the values for other arguments like width shift range,height shiftrange, zoom range, horizontal flip etc.
- Here we use a portion of your training set for validation.For that we specify validation split to be 0.30 when we call the ImageDataGenerator Class for training.
- Next using flow from directory of ImageDataGenerator class we read the data. We specify the values of the paramters here.
  - Here we give the resize shape of the image.
  - We specify the class mode which is categorical here.
  - We set the batch size.
  - We set shuffle to be true for random shuffling of the data.
  - In case of training data, to get the train and validation set we define the subset as "training" and "validation" respectively. Here 70 - 30 % split.

    For test data we do the same things except defining any subset.
- We check the categories and visualize an image for each category.
- Next we define our model. We used pre-trained model as feature extractor, added additional layers for classification, trained the model and evaluated it. ‘
- We then fine-tune the pre-trained model, unfreeze a portion of the pre-trained network, train the resulting model, evaluate it.

The model architecture and all related things are described in the Model section.

Model

Used pre-trained Model: **ResNet101**

- The model is trained on Imagenet.
- Trained on about 1.2 million training images with another 50,000 images for validation and 100,000 images for testing.
- Trained to classify an input image into 1,000 separate object categories.

Many pre-trained models inclusing this are available in keras. These pre-trained models can be used for prediction, feature extraction, and fine-tuning. Pre-trained weights are downloaded automatically when instantiating a model.

First we pick which layer of RESNET101 we will use for feature extraction. The output layer of this model is not very helpful us. Number of categories are also different. We have 15 categories while the model is trained to predict images of 1000 categories.

- While instantiating the Resnet Model we set the argument include_top to be false. So the model we load doesn't include the classification layer of that pre-trained model.
- We then depend on the very last layer before flatten operation.

Training was done in two steps -

- First we use the pre-trained weights (removing the output layer of the pretrained model). We only train the added layers. The following layers were added on top of the base model after flattening -
  - A dense layer with 256 nodes.
  - The output layer(also a dense layer) with 15 nodes(as 15 categories)
- After that we unfreeze a part of the pre-trained model and learn the weights of the unfreezed layers also with the same model architecture as before.

So first we define our model. We then set the fix the values of epochs, steps per epochs, validation steps etc. Then we fit the model and learn the weights for the learnable parameters. Same procedure in both training cases, the difference is- when we unfreeze some portion of the network the no of trainable parameters got increased which can take more time to train but can also give better performance.

We define the following which were used somewhere in the model -

Callbacks: A callback is a set of functions to be applied at given stages of the training procedure. We can use callbacks to get a view on internal states and statistics of the model during training. We can pass a list of callbacks (as the keyword argument callbacks) to the .fit() method of the Sequential or Model classes of keras. The following functions were used -

- **EarlyStopping** - It helps to stop training when a monitored quantity(such as accuracy, loss) has stopped improving.
- **ModelCheckpoint** - It saves the model after every epoch. We can use it to save out best model i.e where we got maximum accuracy or minimum loss while running the epochs. We set the parameters accordingly.
- **ReduceLROnPlateau** - Reduce learning rate when a metric has stopped improving. Models often benefit from reducing the learning rate by a factor of 2-10 once learning stagnates. This callback monitors a quantity and if no improvement is seen for a 'patience' number of epochs, the learning rate is reduced.

Optimizer: Several optimizers are tried out. RMSprop, SGD and ADAM all were tried out. Finally ADAM was used.

Batch Normalization: This is a technique for training deep neural networks. It helps us to stabilize the training process and reduce the no of training epochs required to train a model by reducing the problem of internal co-variance shift. It standardizes the inputs to a layer for each mini-batch.
In our case batch normalization was applied to the flatten output and also tried out to the next dense layer. It really worked in making the model learn fast and converge to a high accuracy.

Activation Function: "RELU" activation function is used in added dense layer and "Softmax" activation function is used in the output(dense) layer.

Loss Function: Categorical Cross-entropy is used as Loss function. For one input data the loss can be calculated using the following formula -

$$L(y, p) = -\sum_{i \in C} y_i \log(p_i)$$

Here y is the true label(one hot encoded) and p is the vector of predicted probabilities corresponding to each class.

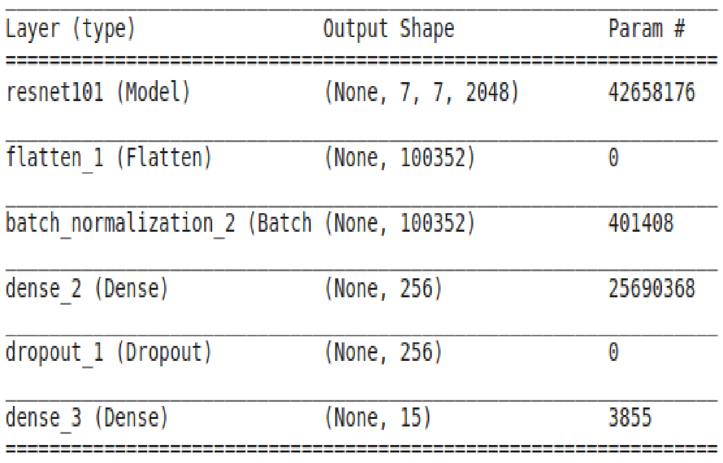
Data Augmentation: A few Data Augmentation techniques, such as rescaling, height and width shift, zooming, horizontal flip were used. [rescale=1.0/255,width_shift_range=0.1,height_shift_range=0.1, zoom_range=0.1, horizontal_flip=True, validation_split=0.30]

Some Hyper-parameters:

- Total Epochs - 50
- No of epochs before fine tune - 20
- No of epochs after fine tune - 30
- Steps per epoch - 50 & Validation Steps - 30
  - For computation purpose taken to be small
  - With increase in these values higher chance of better performance of the model.
- We unfreeze all the layers.

Model Summary :

The following architecture was used-



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
resnet101 (Model)            (None, 7, 7, 2048)        42658176
_________________________________________________________________
flatten_1 (Flatten)          (None, 100352)            0
_________________________________________________________________
batch_normalization_2 (Batch (None, 100352)            401408
_________________________________________________________________
dense_2 (Dense)              (None, 256)               25690368
_________________________________________________________________
dropout_1 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_3 (Dense)              (None, 15)                3855
=================================================================
```

Experiments

- While using the ResNet101 pretrained model, after removing the output layer and flattening the penultimate layer a dense layer were added to the model. Tried out different no of nodes for this dense layer, such as 128, 256, 512. No of parameters got increases with the increase in no of nodes. The accuracy was also effected by that. The convergence was relatively faster with increase in no of nodes in the dense layer.
- Batch normalization is also tried out. It had a significant effect in increasing the accuracy and faster convergence.
- Different optimizers were tried out, SGD with different momentums, ADAM, RMSProp. All worked almost similarly in this case.

- Tried changing the learning rate. Also used callback **ReduceLROnPlateau** which helps to keep changing the learning rate when the specified metric(accuracy, loss) stops changing much. It reduces the learning rate by specified factor while running the epochs, thus helping it to converge faster.
- Tried adding more than one dense layer before the output layer. There wasn't significant change in this case.
- Different probability values for dropout were tried out.
- Tried out callback function early stopping which helped sometimes when a model was not learning much for a long time. It wasn't used later as the we need to fix the patience(no of epochs to wait) and in our case the resnet101 was taking too much time and got stuck around some accuracy for a long time and to check whether it really learns or not after some time.
- Few Data Augmentation techniques like height and width shift, zooming, horizontal flip were tried out. There wasn't much improvement.
- Using model check point we saved the best model i.e where the highest validation accuracy is obtained. Using load model from keras we load it and use for predicting and evaluating. It was not used in all the cases as highest accuracy and final validation accuarcy wasn't too far some times.
- Tried different numbers for unfreeze layers but finally we unfreezed all the layers and fit the model.
- Other pre-trained models like ResNet101V2, InceptionResNetV2, VGG were tried out. InceptionResNetV2 worked best with a training for very less no of epochs without fine tuning.

Results

The training was done in two steps. First we use the pre-trained model weights with the additional layers whose weights were learned. In the 2nd step we unfreeze some fraction(here whole) of the layers of the pre-trained model. Here all the weights corresponding to the specified layers were learned.

- Pre-trained Model: **ResNet101**

  Before fine-tune -

  - Here steps per epoch and validation steps were set to be 50 and 30 for computing purposes. Here we have a scope to increase the accuracy by taking steps equals to no of batches or greater than whatever taken.
  - Train:
    - Loss: 0.3339
    - Accuracy: 93.38 %
  - Validation:
    - Loss: 1.5605762004852295
    - Accuracy: 52.083 %
  - Test:
    - Loss: 1.5241038799285889
    - Accuracy: 54.75 %

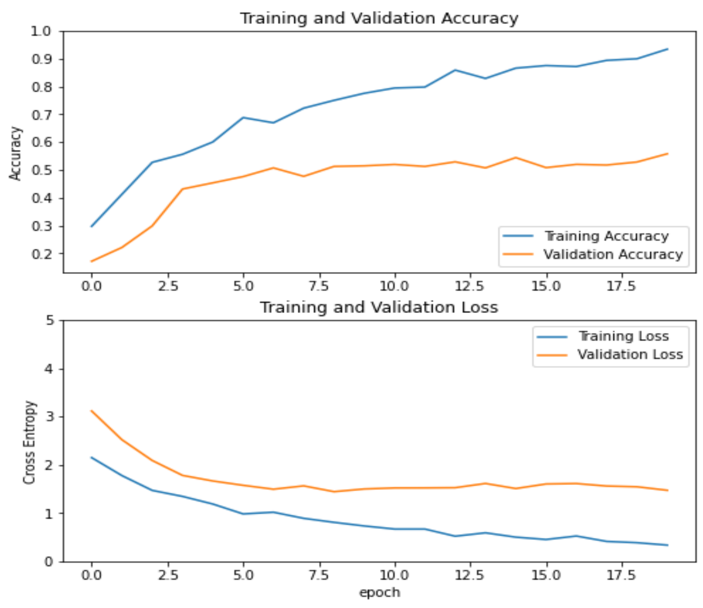
  After fine-tune:

  - Here steps during training are same as before. While evaluating the steps are equal to the no of batches in each case.
  - Train:
    - Loss: 0.0072
    - Accuracy: 99.75 %
  - Validation:
    - Loss: 0.3852
    - Accuracy: 91.604 %

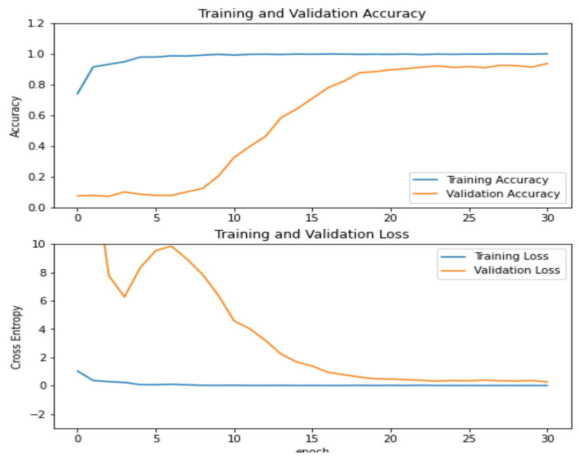
- Test:
    - Loss: 0.3329
    - Accuracy: 92.222 %

Plot of accuracy and loss over epoch:

Before fine tune:



After Fine tune:



Pre-trained Model: **InceptionResNetV2**

Before fine-tune -

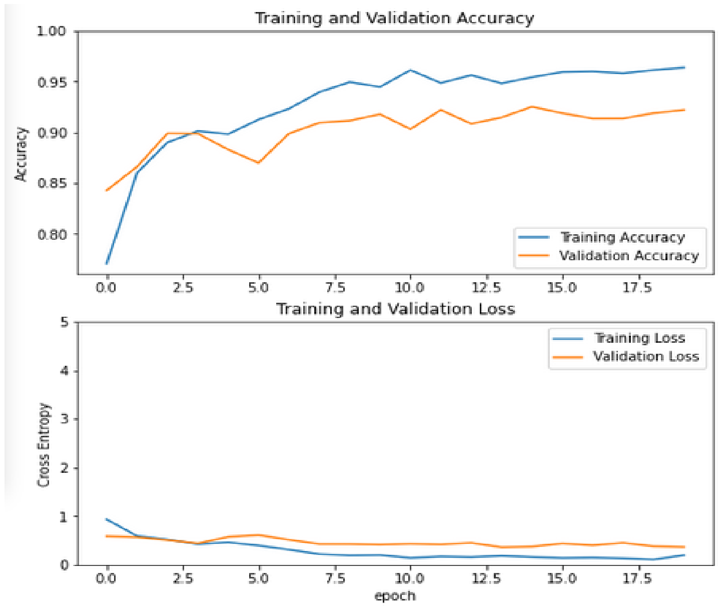
- Train:
  - Loss: 0.1923
  - Accuracy: 96.37 %
- Validation:
  - Loss: 0.4453
  - Accuracy: 91.458 %
- Test:
  - Loss: 0.348
  - Accuracy: 92.833 %

After fine-tune -

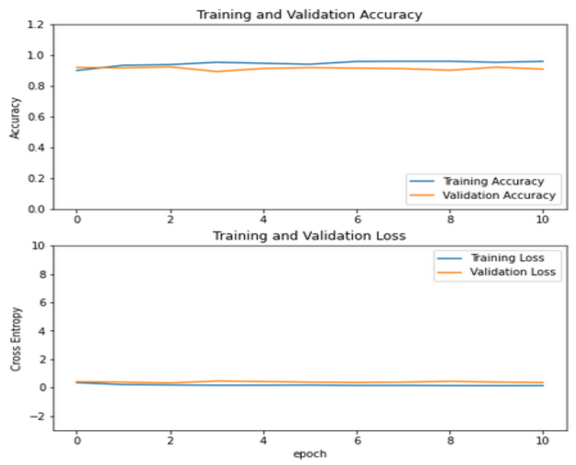
- Unfreezing only the last 60 layers
- Train for only 10 epochs
- Validation:
  - Loss: 0.4021
  - Accuracy: 90.925 %
- Test:
  - Loss: 0.36047
  - Accuracy: 92.388 %

Plot of accuracy and loss over epoch:

Before fine tune:



After Fine tune:



We see that after unfreezing all the layers of the ResNet101 model accuracy increased to 92.3 % while before fine tuning it was not increasing much more than 55 %. So here unfreezing the layers helped. We can see the plots and get an idea about that. We trained for only 50 epochs with less no of steps per epochs and validation steps. So there is some chance of increasing the accuracy. With high computing power it can be done more efficiently and lot more experiments can be done.It seems there is over-fitting as training accuracy was almost 100 %. Data Augmentation was also tried out which didn't help to a larger extent. The InceptionResnetV2 worked better than this without fine tuning but with very less no of epochs. After the evaluation on the validation data if the model seems to work well, we can train a model with both the train and validation data together. More data generalizes the model better and also helps in achieving better performance. But in general Transfer Learning is very helpful for these tasks.

Extra Credit

Pre-trained models tried out -

- InceptionResnetV2
- ResNet101V2
- VGG

Both InceptionResnetV2 and ResNet100V2 worked very well. InceptionResnetV2 was the best choice. Within 5 epochs the validation accuracy rises to 93% and it increased further in the next epochs. The accuracy and loss converged very fast for both these models. VGG didn't performe well. Notebook for the best model is given.

For this pre-trained InceptionResnetV2 model -

All the results were shown in result section.

Same architecture as that of using ResNet101 as pre-trained model.

Total Epochs - 30

No of epochs before fine tune - 20

No of epochs after fine tune - 10

Steps per epoch - 50 & Validation Steps - 30

- For computation purpose taken to be small
- With increase in these values higher chance of better performance of the model.

We unfreeze the last 60 layers.

Not much increase in accuracy after unfreezing only last 60 layers for 10 epochs.

Comparison:

- InceptionResnetV2 consists of more than double no of layers than that of ResNet101.
- Performance: InceptionResnetV2 performed way better than ResNet101 without fine tuning. The accuracy achieved by ResNet101 after fine tune is nearly same as that of InceptionResnetV2 without fine tuning.
- InceptionResnetV2 achevied higher accuracy way earlier than ResNet101. So with a very less no of epochs we achieve very good accuracy with InceptionResnetV2.