Transfer learning and fine-tuning

Sagaf Youssouf

Université Paris-Est Créteil (UPEC)
International Master of Biometrics and Intelligent Vision
https://www.international-master-biometrics-intelligent-vision.org/

1 Introduction

Training a deep learning model for image classification on very specific tasks from scratch does not always result in interesting performances, especially when you are limited in data. We can take advantage of models already trained on large datasets like ImageNet using techniques such as transfer learning and fine-tuning. With transfer learning, learned parameters (weights, bias) of a pre-trained network are used for a new task. The network's layers remain unchanged, and the very last layers are modified. In this way, we can take advantage of learned feature maps without needing to start training from scratch the large model. On the other hand, fine-tuning can be defined as an optimization technique. It is most often applied right after using the transfer learning technique. Here we optimize the network to achieve the optimal results. One can change the number of layers used, numbers of filters, learning rate and we have many parameters of the model to optimize. In the next sections, you will find a critical analysis and how to apply these powerful techniques to take Deep Learning models to a whole new level. We are interested here in the classification of cat and dog images.

2 Data preparation and examination

For this lab, the dataset used consists of several images of cats and dogs. 2000 files are belonging to the two classes. The dataset itself does not contain any test set. By using `tf.data.experimental.cardinality`, the test set can be created knowing the batches of data available. Buffered prefetching method is applied for more performance while loading images from disk.
As we are dealing with a small dataset, data augmentation technique is applied by performing random and realistic transformation such as rotation and horizontal flipping to create new images. In the last lab we learnt about data augmentation. We saw that this method helped us to mitigate over-fitting. Examples of originals dogs and cats images are given in figure 1. The base model which will be used here expects pixels values to be in the range of `[-1,1]`. However, images of the dataset are in the range of `[0-255]`. To rescale them, a `Rescaling` layer is used.

Fig. 1: Examples of images of cats and dogs constituting the dataset.

3 Creating the base model

3.1 Base model definition

To perform transfer learning and fine-tuning, we should have a pre-trained model which will be used as a base model. To create a base model from a pre-trained convnet, one can instantiate a deep learning model available alongside with pre-trained weights using Keras applications API. Here, MobileNet V2 developed at Google and trained on ImageNet dataset is chosen. The model will serve as a feature extractor that converts each 160x160x3 image into a 5x5x1280 block of features. Following code illustrates how to create the base model by loading a network that does n't include the classification layers. This is done by setting up the `include_top` flag to `False`.

```
IMG_SHAPE = IMG_SIZE + (3,)
base_model = tf.keras.applications.MobileNetV2(input_shape=IMG_SHAPE,
                                               include_top=False,
                                               weights='imagenet')
```

After having the base model that will be used as a feature extractor, the next step is to add a classifier on top of it and train the lop-level classifier only. Before that, it's important to freeze the convolutional base by setting `base_model.trainable = False`. The total number of trainable parameters is 2 257 984 and there are no trainable parameters for the base model. Note that BatchNormalization layers should be kept in inference mode in order to do fine-tuning otherwise, the updates applied to the non-trainable parameters will destroy what the model has learned.

3.2 Adding of classification head

Before applying a Dense layer to convert features into a single prediction per image, spatial 5x5 locations should be average to generate predictions from a block of features. This is done using `tf.keras.layers.GlobalAveragePooling2D` layer. The output is a single 1280 element vector per image. Below is the code used to do the classification head :

```
global_average_layer = tf.keras.layers.GlobalAveragePooling2D()
feature_batch_average = global_average_layer(feature_batch)
prediction_layer = tf.keras.layers.Dense(1)
prediction_batch = prediction_layer(feature_batch_average)
```

The whole model is obtained by chaining together the data augmentation layer, rescaling, base model, global_average using the Keras Functional API as illustrated in the following code. A plot of the resulted final model components is given in figure 2 along . Once the model is compiled, the total number of parameters is 2 259 265 and only 1 281 parameters are trainable. This means 2.5 M parameters in MobileNet are frozen and only Dense layer parameters are trainable.

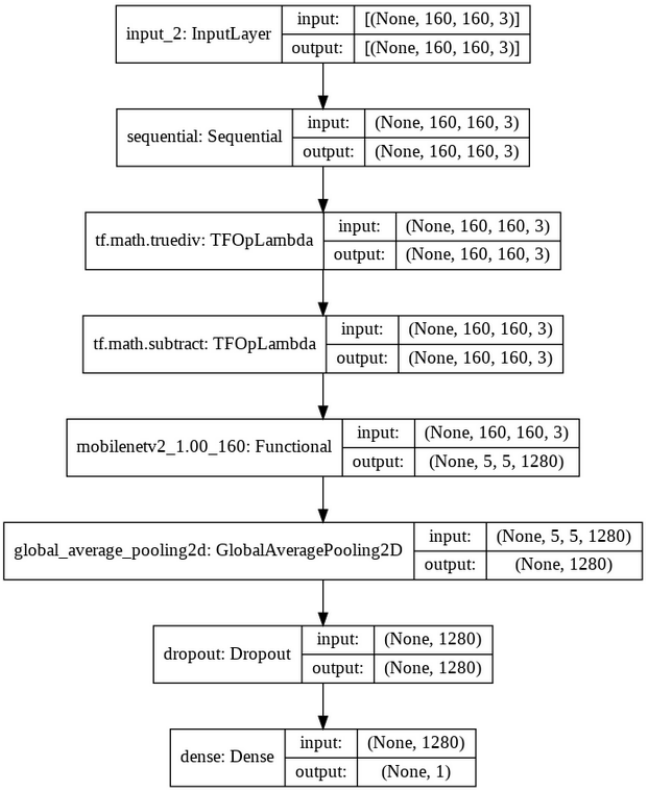

Fig. 2: Base model layers components

```
inputs = tf.keras.Input(shape=(160, 160, 3))
x = data_augmentation(inputs)
x = preprocess_input(x)
x = base_model(x, training=False)
x = global_average_layer(x)
x = tf.keras.layers.Dropout(0.2)(x)
outputs = prediction_layer(x)
model = tf.keras.Model(inputs, outputs)
```

4 Transfer learning

Before training, the model is compiled using a learning rate of 0.0001. Since there are only two classes, binary cross-entropy is used. After training for 10 epochs, the final validation loss is 0.1066 and the final validation accuracy is 97.03 %. The learning curves of loss and accuracy are given in the figure 3 and figure 4 respectively. This is a zoom in the top of the y-axis (0.7 to 1). One can see that the validation metrics are better than the training metrics. There is a gap between the accuracy for the trained data and the validation accuracy despite the fact the validation accuracy is increasing with the number of epochs. There is no sign of over-fitting because both accuracies continue to increase. This is due because of layers like `tf.keras.layers.BatchNormalization` and `tf.keras.layers.Dropout` affect accuracy during training. They are turned off when validations metrics are calculating.

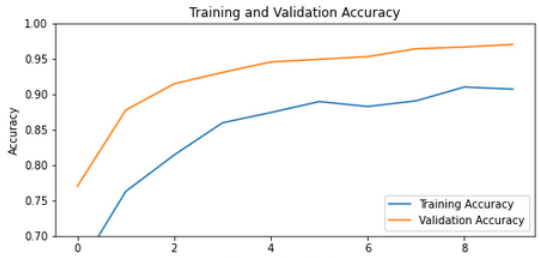

Fig. 3: Training and validation loss after transfer learning

5 Fine-tuning

In the last step with feature extraction, only a few layers on the top of MobileNetV2 base model are trained. And the weights of the pre-trained network were not updated during training. Fine-tuning consists of increasing performance by training the weights of the pre-trained model alongside the head classifier. This is done simply by unfreezing the base model and setting up the bottom layers to be un-trainable. This can be illustrated with the code below.

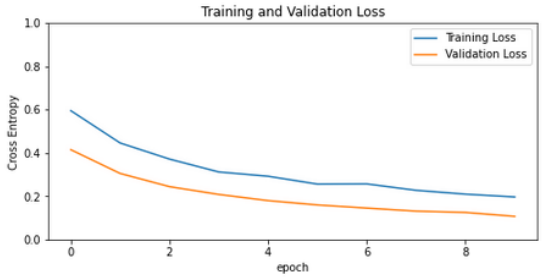

Fig. 4: Training and validation accuracy after transfer learning.

The model is compiled again using a lower learning rate as we are dealing with a much large model. The number of epoch used to train the new model is the initial epoch added with the number of fine-tune epochs which make 20 epochs. The total number of trainable parameters is 1 862 721 and the number of non-trainable parameters is 396 544. Metrics results obtained after applying fine-tuning alongside withe the feature extraction step is illustrated with figures 5 and 6. The accuracy achieved is 98% on the validation data set. With the learning curves obtained when fine-tuned the last few layers, we can see that the gap between validation and training metrics is decreased compared to the first step. We can conclude that this step improves the performance of the model.

```
base_model.trainable = True
# Fine-tune from this layer onwards
fine_tune_at = 100
# Freeze all the layers before the ‘fine_tune_at‘ layer
for layer in base_model.layers[:fine_tune_at]:
    layer.trainable =  False
```

Fig. 5: Training and validation loss after fine-tuning

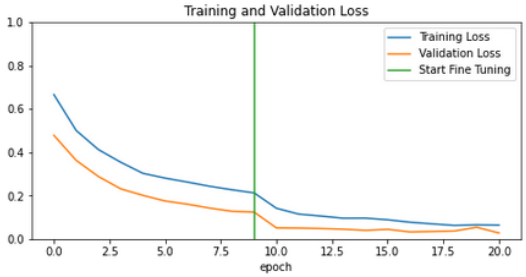

Fig. 6: Training and validation accuracy after fine-tuning

6 Conclusion

In this lab, we expose the idea that when working with a small dataset, one can use features learned by a model trained on a larger dataset like ImageNet. This technique is called transfer learning. To do that, one has to instantiate a pre-trained model and add on top of it a classifier. In this way, only the weights of the classifier will be updated and the pre-trained model is frozen. On the other hand, to increase the performance of the model, the fine-tuning technique can be used. This consists of repurposing the top-level layers of the pre-trained model to the new dataset. Most of the time, this technique follows the transfer learning method and is recommended when the new dataset is similar to the original one.

References

1. Transfer learning and fine-tuning, `https://colab.research.google.com/drive/1QrngNKFbFcAuYrVTcEyO_Z1YWY-4nLz_?usp=sharing` Difference between transfer learning and fine tuning, `https://www.quora.com/What-is-the-difference-between-transfer-learning-and-fine-tuning`