Applied Machine Learning - Week 6 - Transfer Learning

Bui Minh Thuan

In week 6's lab, we learn how to apply Transfer Learning to utilize a good pre-trained model for specific tasks. This lab involves explaining theory questions as well as analyzing this notebook: https://www.tensorflow.org/tutorials/images/transfer_learning

1. Explain the concept of transfer learning. How does it differ from training a model from scratch?

Transfer learning refers to where a pre-trained model (with a large enough dataset) is utilized for a new, smaller but related task. Instead of training a model from scratch, we leverage the knowledge the model has already learned and fine-tune it for the target task.

Compared to training a model from scratch, transfer learning is faster, requires less data, and is more computationally efficient. It leverages pre-trained models, making it ideal when data is limited, whereas training from scratch is better for large datasets but more resource-intensive.

2. What is fine-tuning in the context of transfer learning, and why is it useful?

Fine-tuning in transfer learning refers to the process of further training a pre-trained model on a new dataset by adjusting some or all of its layers. Initially, the model's earlier layers (which capture general features) are frozen, while the later layers are retrained to adapt to the specific task. In some cases, all layers are fine-tuned for better optimization.

Fine-tuning is useful as it helps adapt the Model to a specific task, which can lead the model to learn domain-specific features better. It also improves overall performance, with less data required and saves more time and resources for training.

3. Why is it important to freeze the convolutional base during feature extraction?

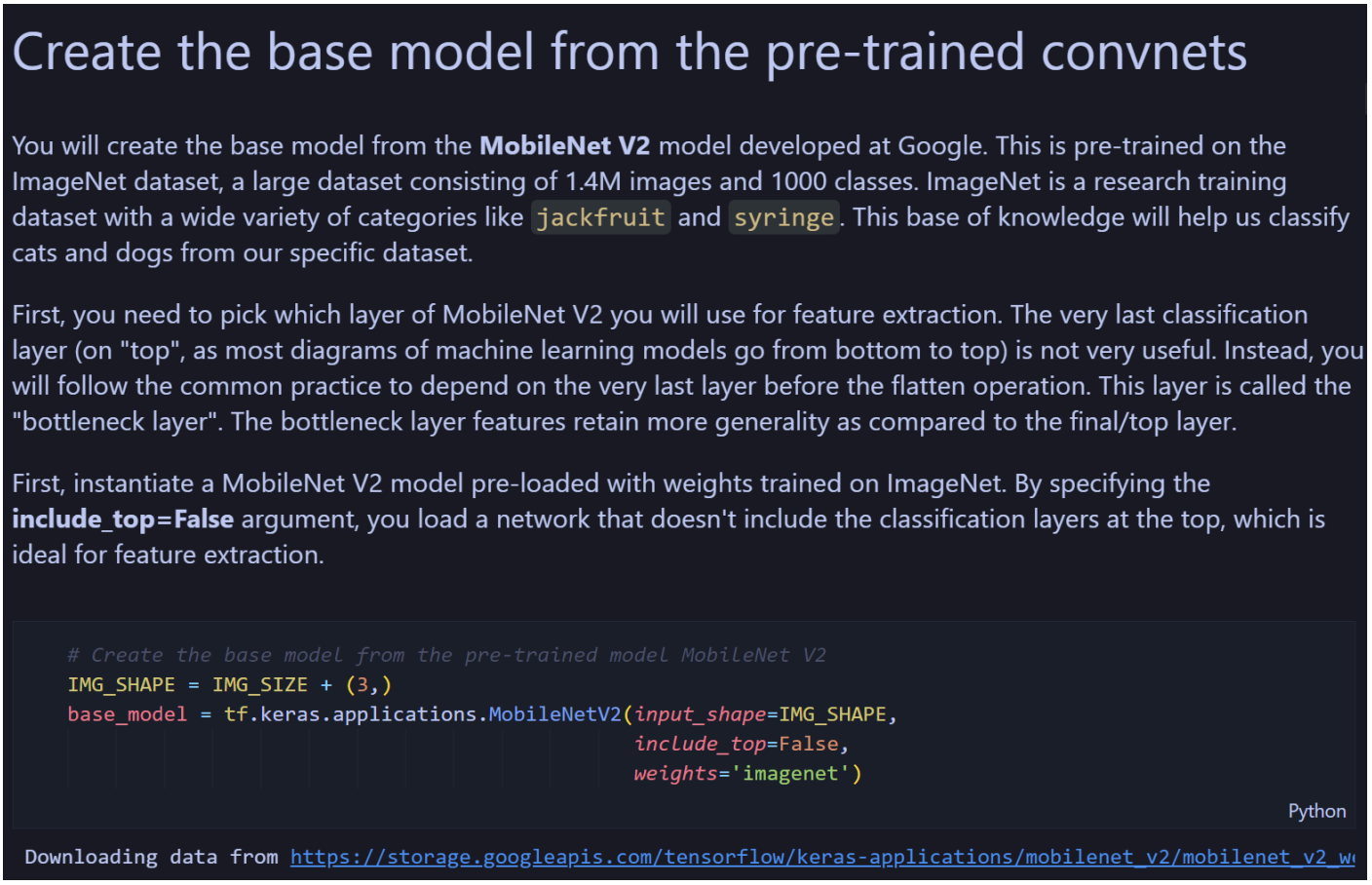
It is important to freeze the convolutional base before you compile and train the model. Freezing (by setting layer.trainable = False) prevents the weights in a given layer from being updated during training. By freezing them, we ensure that these learned features remain intact and avoid unnecessary computational costs. This also helps prevent overfitting, especially when working with small datasets, as it reduces the number of trainable parameters.

4. Why use data augmentation?

Data augmentation is essential for improving model generalization, especially when working with limited datasets.. When we don't have a large image dataset, it's a good practice to artificially introduce sample diversity by applying random, yet realistic, transformations to the training images, such as rotation and horizontal flipping. This helps expose the model to different aspects of the training data and reduce overfitting.

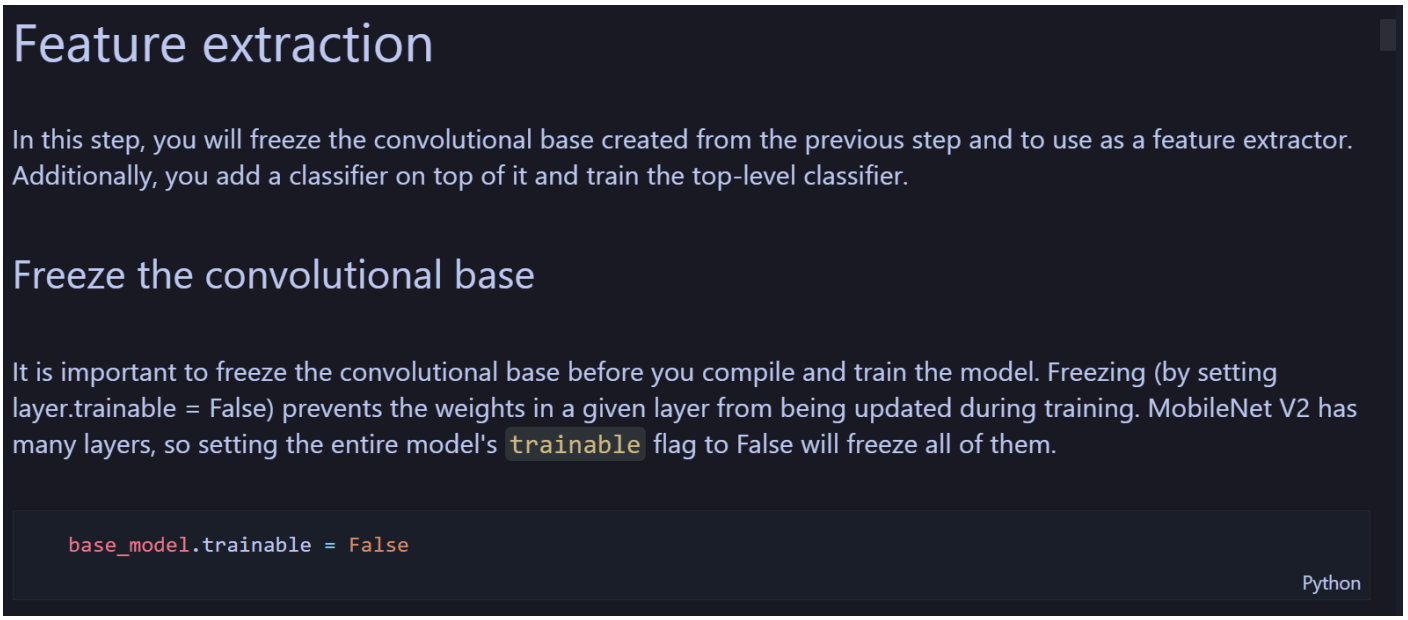
5. Take a screenshot of the code snippet where the pre-trained MobileNetV2 model is loaded without the top classification layers.

Here, pre-trained MobileNetV2 is loaded without the top classification layers because this is not very useful for the transfer learning process.



6. Take a screenshot of the portion of code where the pre-trained model is set to be non-trainable for feature extraction purposes.

Here is the code snippet where the model is set to be non-trainable during feature extraction in order to prevent trained weights being updated during training.



7. Take a screenshot of the data augmentation layers defined in the model.

Here is the code snippet of the data augmentation layers in the model, used to improve data quality by introducing sample diversity.

Use data augmentation

When you don't have a large image dataset, it's a good practice to artificially introduce sample diversity by applying random, yet realistic, transformations to the training images, such as rotation and horizontal flipping. This helps expose the model to different aspects of the training data and reduce overfitting. You can learn more about data augmentation in this tutorial.

```
data_augmentation = tf.keras.Sequential([
  tf.keras.layers.RandomFlip('horizontal'),
  tf.keras.layers.RandomRotation(0.2),
])
```

Python

Note: These layers are active only during training, when you call `Model.fit`. They are inactive when the model is used in inference mode in `Model.evaluate`, `Model.predict`, or `Model.call`.

Let's repeatedly apply these layers to the same image and see the result.

```
for image, _ in train_dataset.take(1):
  plt.figure(figsize=(10, 10))
  first_image = image[0]
  for i in range(9):
    ax = plt.subplot(3, 3, i + 1)
    augmented_image = data_augmentation(tf.expand_dims(first_image, 0))
    plt.imshow(augmented_image[0] / 255)
    plt.axis('off')
```

Accept Review

8. Take a screenshot of the code that shows the addition of the new classifier layers on top of the base model

Here is the code snippet that shows the addition of the new classifier layers on top of the base model.

Add a classification head

To generate predictions from the block of features, average over the spatial `5x5` spatial locations, using a `tf.keras.layers.GlobalAveragePooling2D` layer to convert the features to a single 1280-element vector per image.

```
global_average_layer = tf.keras.layers.GlobalAveragePooling2D()
feature_batch_average = global_average_layer(feature_batch)
print(feature_batch_average.shape)
```

Python

```
(32, 1280)
```

Apply a `tf.keras.layers.Dense` layer to convert these features into a single prediction per image. You don't need an activation function here because this prediction will be treated as a `logit`, or a raw prediction value. Positive numbers predict class 1, negative numbers predict class 0.

```
prediction_layer = tf.keras.layers.Dense(1, activation='sigmoid')
prediction_batch = prediction_layer(feature_batch_average)
print(prediction_batch.shape)
```

Python

```
(32, 1)
```