# CHAPTER 8: Pre-Trained Networks

## Overview

In this chapter, you will analyze pre-trained models. You will get hands-on experience using the different state-of-the-art model architectures available on TensorFlow. You will explore concepts such as transfer learning and fine-tuning and look at TensorFlow Hub and its published deep learning resources.

By the end of the chapter, you will be able to use pre-trained models directly from TensorFlow and TensorFlow Hub.

## Introduction

In the previous chapter, you learned how **convolution neural networks** (**CNNs**) analyze images and learn relevant patterns to classify their main subjects or identify objects within them. You also saw the different types of layers used for such models.

But rather than training a model from scratch, it would be more efficient if you could reuse existing models with pre-calculated weights. This is exactly what **transfer learning** and **fine-tuning** are about. You will learn how to apply these techniques to your own projects and datasets in this chapter.

You will also look at the ImageNet competition and the corresponding dataset that is used by deep learning researchers to benchmark their models against state-of-the-art algorithms. Finally, you will learn how to use TensorFlow Hub's resources to build your own model.

## ImageNet

ImageNet is a large dataset containing more than 14 million images annotated for image classification or object detection. It was first consolidated by Fei-Fei Li and her team in 2007. The goal was to build a dataset that computer vision researchers could benefit from.

The dataset was presented for the first time in 2009, and every year since 2010, an annual competition called the **ImageNet Large-Scale Visual Recognition Challenge** (**ILSVRC**) has been organized for image classification and object detection tasks.

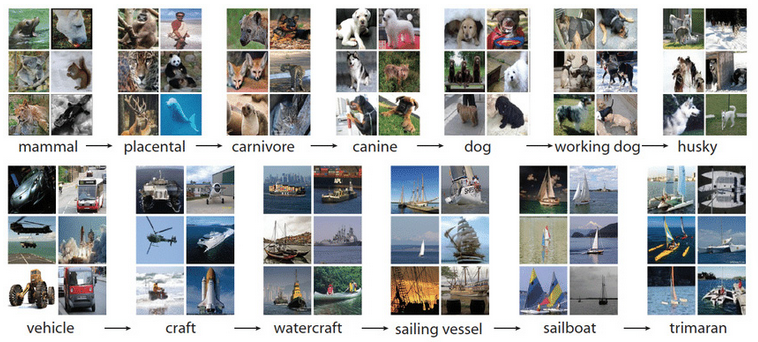


Figure 8.1: Examples of images from ImageNet

Over the years, some of the most famous CNN architectures (such as AlexNet, Inception, VGG, and ResNet) have achieved amazing results in this ILSVRC competition. In the following graph, you can see how some of the most famous CNN architectures performed in this competition. In less than 10 years, performance increased from 50% accuracy to almost 90%.

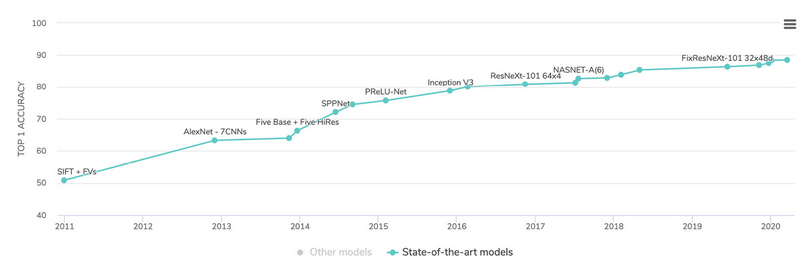


Figure 8.2: Model benchmarking from paperswithcode.com

You will see in the next section how you can use transfer learning with these models.

## Transfer Learning

In the previous chapter, you got hands-on practice training different CNN models for image classification purposes. Even though you achieved good results, the models took quite some time to learn the relevant parameters. If you kept training the models, you could have achieved even better results. Using **graphical processing units** (**GPUs**) can shorten the training time, but it will still take a bit of time, especially for bigger or more complex datasets.

Deep learning researchers have published their work for the benefit of the community. Everyone can benefit by taking existing model architectures and customizing them, rather than designing architectures from scratch. More than this though, researchers also share the weights of their models. You can then not only reuse an architecture but also leverage all the training performed on it. This is what transfer learning is about. By reusing pre-trained models, you don't have to start from scratch. These models are trained on a large dataset such as ImageNet and have learned how to recognize thousands of different categories of objects. You can reuse these state-of-the-art models straight out of the box without having to train them. Isn't that amazing? Rather than training a model for weeks, you can now just use an existing model.

TensorFlow provides a list of state-of-the-art models pre-trained on the ImageNet dataset for transfer learning in its Keras API.

Note

You can find the full list of pre-trained models available in TensorFlow at the following link: <https://www.tensorflow.org/api_docs/python/tf/keras/applications>.

Importing a pre-trained model is quite simple in TensorFlow, as shown with the following example, where you load the InceptionV3 model:

*import tensorflow as tf*

*from tensorflow.keras.applications import InceptionV3*

Now that you have imported the class for the pre-trained model, you need to instantiate it by specifying the dimensions of the input image and imagenet as the pre-trained weights to be loaded:

*model = InceptionV3(input\_shape=(224, 224, 3), \*

*weights='imagenet', include\_top=True)*

The include\_top=True parameter specifies that you will be re-using the exact same top layer (which is the final layer) as for the original model trained on ImageNet. This means that the last layer is designed to predict the 1,000 classes that are in this dataset.

Now that you have instantiated your pre-trained model, you can make predictions from it:

*model.predict(input\_image)*

If you want to use this pre-trained model to predict different categories than the ones from ImageNet, you will need to replace the top layer with another one that will be trained to recognize the specific categories of the input dataset.

First, you need to remove this layer by specifying include\_top=False:

*model = InceptionV3(input\_shape=(224, 224, 3), \*

*weights='imagenet', include\_top=False)*

In the preceding example, you have loaded an InceptionV3 model. The next step will be to freeze all the layers from this model so that their weights will not be updated:

*model.trainable = False*

After this, you will instantiate a new fully connected layer with the number of units and activation function of your choice. In the following example, you want to predict 50 different classes. To do this, you create a dense layer with 20 units and use softmax as the activation function:

*top\_layer = tf.keras.layers.Dense(20, activation='softmax')*

Then you need to add this fully connected layer to your base model with the Sequential API from Keras:

*new\_model = tf.keras.Sequential([model, top\_layer])*

Now, you can train this model and only the top-layer weights will be updated. All the other layers have been frozen:

*new\_model.compile(loss='sparse\_categorical\_crossentropy',\*

*optimizer=tf.keras.optimizers.Adam(0.001))*

*new\_model.fit(X\_train, t\_train, epochs=50)*

In just a few lines of code, you have loaded the Inception V3 model, which is a state-of-the-art model that won the ILSVRC competition in 2016. You learned how to adapt it to your own project and dataset.

In the next exercise, you will have hands-on practice on transfer learning.

## Exercise 8.01: Classifying pizza and steak with Transfer Learning

In this exercise, you will use transfer learning to correctly classify images as either pizzas or steak. You will use a pre-trained model, NASNet-Mobile, that is already available in TensorFlow. This model comes with pre-trained weights on ImageNet.

Note

The original dataset used in this exercise has been provided by Google. It contains 2000 images of pizzas and steaks. It can be found here:

<https://storage.googleapis.com/ztm_tf_course/food_vision/pizza_steak.zip>

1. Open a new Jupyter notebook.
2. Import the TensorFlow library:

*import tensorflow as tf*

1. Create a variable called file\_url containing a link to the dataset:

*file\_url='https://storage.googleapis.com/ztm\_tf\_course/food\_visio n/pizza\_steak.zip'*

1. Dowload the dataset using tf.keras.get\_file with ‘pizza\_and\_steak.zip’, origin=file\_url, and extract=True as parameters, and save the result to a variable called zip dir.

*zip\_dir = tf.keras.utils.get\_file('pizza\_and\_streak.zip',*

*origin=file\_url, extract=True)*

1. Import the pathlib library:

*import pathlib*

1. Create a variable called path containing the full path to the pizza\_and\_steak directory using pathlib.path(zip\_dir).parent:

*path = pathlib.Path(zip\_dir).parent / 'pizza\_steak'*

1. Create two variable called train\_dir and validation\_dir that take the full path to the train and validation folders, respectively:

*train\_dir = path / 'train'*

*validation\_dir = path / 'validation'*

1. Create four variables called train\_pizza\_dir, train\_steak\_dir, validation\_pizza\_dir, validation\_steak\_dir that take the full path to the pizza and steak folders for the train and validation sets, respectively.

*train\_pizza\_dir = train\_dir / 'pizza'*

*train\_steak\_dir = train\_dir /'steak'*

*validation\_pizza\_dir = validation\_dir / 'pizza'*

*validation\_steak\_dir = validation\_dir / 'steak'*

1. Import the os package. In the next step, you will need to count the number of images from a folder:

*Import os*

1. Create two variables called total\_train and total\_val that get the number of images for the training and validation sets:

*total\_train = len(os.listdir(train\_pizza\_dir)) + len(os.listdir(train\_steak\_dir))*

*total\_val = len(os.listdir(validation\_pizza\_dir)) + len(os.listdir(validation\_steak\_dir))*

1. Impor ImageDataGenerator from tensorflow.keras.preprocessing:

*from tensorflow.keras.preprocessing.image import ImageDataGenerator*

1. Instantiate two ImageDataGenerator classes and call them train\_image\_generator and validation\_image\_generator. These will rescale images by dividing by 255:

*train\_image\_generator = ImageDataGenerator(rescale=1./255)*

*validation\_image\_generator = ImageDataGenerator(rescale=1./255)*

1. Create three variables called batch\_size, img\_height, and img\_width that takes values 16, 224, and 224, respectively:

*batch\_size = 16*

*img\_height = 224*

*img\_width = 224*

1. Create a data generator called train\_data\_gen using flow\_from\_directory() method, and specify the batch size, the path to the training folder, the size of the target, and the mode of the class:

*train\_data\_gen = train\_image\_generator.flow\_from\_directory\*

*(batch\_size = batch\_size, \*

*directory = train\_dir, \*

*shuffle=True, \*

*target\_size = (img\_height, img\_width), \*

*class\_mode='binary')*

1. Create a data generator called val\_data\_gen using flow\_from\_directory() method and specify the batch size, the path to the validation folder, the size of the target, and the mode of the class:

*val\_data\_gen = validation\_image\_generator.flow\_from\_directory\*

*(batch\_size = batch\_size, \*

*directory = validation\_dir, \*

*target\_size=(img\_height, img\_width), \*

*class\_mode='binary')*

1. Import numpy as np, tensorflow as tf, and layers from tensorflow.keras:

*import numpy as np*

*import tensorflow as tf*

*from tensorflow.keras import layers*

1. Set 8 (this is totally arbitrary) as seed for NumPy and TensorFlow:

*np.random.seed(8)*

*tf.random.set\_seed(8)*

1. Import the NASNETMobile model from tensorflow.keras.applications:

*from tensorflow.keras.applications import NASNetMobile*

1. Instantiate the model with the ImageNet weights, remove the top layer, and specify the correct input dimensions:

*base\_model = NASNetMobile(include\_top=False, \*

*input\_shape=(img\_height, img\_width, 3),\*

*weights='imagenet')*

1. Freeze all the layers of this model:

*base\_model.trainable = False*

1. Print a summary of the model using the summary() method:

*base\_model.trainable = False*

The expected output will be as follows:

Table

Description automatically generated

Figure 8.3: Summary of the model

1. Create a new model that combines the NASNETMobile model with two new layers 500 and 1 unit(s) and ReLu and sigmoid as the activation functions:

*model = tf.keras.Sequential([base\_model, layers.Flatten(),*

*layers.Dense(500, activation='relu'),*

*layers.Dense(1, activation='sigmoid')])*

1. Compile the model by providing binary\_crossentropy as the loss function, an Adam optimizer with a learning rate of 0.001, and accuracy as the metric to be displayed:

*model.compile(loss='binary\_crossentropy', \*

*optimizer=tf.keras.optimizers.Adam(0.001), \*

*metrics=['accuracy'])*

1. Fit the model, provide the train and validation data generators, and run it for five epochs:

*model.fit(train\_data\_gen, \*

*steps\_per\_epoch = total\_train // batch\_size, \*

*epochs=5, \*

*validation\_data = val\_data\_gen, \*

*validation\_steps = total\_val // batch\_size)*

The expected output is as follows:

A picture containing calendar

Description automatically generated

Figure 8.4: Model training output

You can observe that the model achieved an accuracy score of 0.99 on the training set and 0.98 on the validation set. This is quite a remarkable result given that you only trained the last two layers, and it took less than a minute. This is the benefit of applying transfer learning and using pre-trained state-of-the-art models.

In the next section, you will see how you can apply fine-tuning to a pre-trained model.

## Fine-Tuning

Previously, you used transfer learning to leverage pre-trained models on your own dataset. You used the weights of state-of-the-art models that have been trained on large datasets such as ImageNet. These models learned the relevant parameters to recognize different patterns from images and helped you to achieve amazing results on different datasets.

But there is a catch with this approach. Transfer learning works well in general if the classes you are trying to predict belong to the same list as that of ImageNet. If this is the case, the weight learned from ImageNet will also be relevant to your dataset. For example, the pizzas and streaks classes from the preceding exercise are present in ImageNet, so its weights will also be relevant for this dataset.

However, if your dataset is very different from ImageNet, then the weights from these pre-trained models may not all be relevant. For example, if your dataset contains satellite images, and you are trying to determine whether a house has solar panels installed on its roof, this will be very different compared to ImageNet. The weights from the last layers will be very specific to the classes from ImageNet, such as pizzas or car wheels (which are not very useful for the satellite image dataset case), while the ones from earlier layers will be more generic, such as for detecting shapes, colors, or texture (which can be applied to the satellite image dataset).

So, it will be great to still leverage some of the weights from earlier layers but train the final layers so that your models can learn the specific patterns relevant to your dataset and improve its performance.

This technique is called fine-tuning. The idea behind it is quite simple: you freeze early layers and update the weights of the final layers only. Let's see how you can achieve this in TensorFlow:

1. First, instantiate a pre-trained MobileNetV2 model without the top layer:

*from tensorflow.keras.applications import MobileNetV2*

*base\_model = MobileNetV2(input\_shape=(224, 224, 3), \*

*weights='imagenet', include\_top=False)*

1. Next, iterate through the first layers and freeze them by setting them as non-trainable. In the following example, you will freeze only the first 100 layers:

*for layer in base\_model.layers[:100]:*

*layer.trainable = False*

1. Now you need to add your custom top layer to your base model. In the following example, you will be predicting 20 different classes, so you need to add a fully connected layer of 20 units with the softmax activation function:

*prediction\_layer = tf.keras.layers.Dense(20, activation='softmax')*

*model = tf.keras.Sequential([base\_model, prediction\_layer])*

1. Finally, you will compile and then train this model:

*model.compile(loss='sparse\_categorical\_crossentropy', \*

*optimizer = tf.keras.optimizers.Adam(0.001))*

*model.fit(features\_train, label\_train, epochs=5)*

This will display a number of logs, as seen in the following screenshot:



Figure 8.5: Fine-tuning results on a pre-trained MobileNetV2 model

That’s it.You have just performed fine-tuning on a pre-trained MobileNetV2 model. You have used the first 100 pre-trained weights from ImageNet and only updated the weights from layer 100 onward according to your dataset.

In the next activity, you will put into practice what you have just learned and apply fine-tuning to a pre-trained model.

## Activity 8.01: Food Classification with Fine-Tuning

The “**10\_food\_classes\_all\_data”** dataset

<https://storage.googleapis.com/ztm_tf_course/food_vision/10_food_classes_all_data.zip>

This data set consists of 10 food categories, with 10,000 images. For each class, 250 manually reviewed test images are provided as well as 750 training images. On purpose, the training images were not cleaned, and thus still contain some amount of noise. This comes mostly in the form of intense colors and sometimes wrong labels. All images were rescaled to have a maximum side length of 512 pixels. Our aim is to train a deep learning model which can successfully classify food images.

In this activity, you are tasked with training a NASNetMobile model to recognize images of different varieties of foods (classification into 10 different classes). You will use fine-tuning to train the final layers of this model.

Follow these steps will help you to complete this activity:

1. Import the dataset and unzip the file using TensorFlow.
2. Create a data generator with the following data augmentation:

*Rescale = 1./255,*

*rotation\_range = 40,*

*width\_shift\_range = 0.1,*

*height\_shift\_range = 0.1,*

*shear\_range = 0.2,*

*zoom\_range = 0.2,*

*horizontal\_flip = True,*

*fill\_mode = 'nearest*

1. Load a pre-trained NASNETMobile model from TensorFlow.
2. Freeze the first 600 layers of the model.
3. Add two fully connected layers on top of NASNETMobile

* A fully connected layer with Dense(1000, activation=relu)
* A fully connected layer with Dense(10, activation=’softmax’)

1. Specify an Adam Optimizer with a learning rate 0.001
2. Train the model.
3. Evaluate the model on the test set.

The expected output is as follows:

A picture containing text

Description automatically generated

Figure 8.6: Expected output of the activity

Note

The solution to this activity can be found via this link

Now that you know how to use pre-trained models from TensorFlow, you will learn how models can be accessed from TensorFlow Hub in the following section.

## TensorFlow Hub

TensorFlow Hub is a repository of TensorFlow modules shared by publishers such as Google, NVIDIA, and Kaggle. TensorFlow modules are self-contained models built on TensorFlow that can be reused for different tasks. Put simply, it is an external collection of published TensorFlow modules for transfer learning and fine-tuning. With TensorFlow Hub, you can access different deep learning models or weights than the ones provided directly from TensorFlow's core API.

Note

You can find more information about TensorFlow Hub here: <https://tfhub.dev/>.

In order to use it, you first need to install it:

*Pip install tensorflow-hub*

Once it's installed, you can load available classification models with the load() method by specifying the link to a module:

*import tensorflow\_hub as hub*

*MODULE\_HANDLE = 'https://tfhub.dev/tensorflow/efficientnet'\*

*'/b0/classification/1'*

*module = hub.load(MODULE\_HANDLE)*

In the preceding example, you have loaded the **EfficientNet B0** model, which was trained on ImageNet. You can find more details on this at the TensorFlow Hub page: <https://tfhub.dev/tensorflow/efficientnet/b0/classification/1>.

Note

TensorFlow Hub provides a search engine to find a specific module: <https://tfhub.dev/s?subtype=module,placeholder>.

By default, modules loaded from TensorFlow Hub contain the final layer of a model without an activation function. For classification purposes, you need to add an activation layer of your choice. To do so, you can use the Sequential API from Keras. You just need to convert your model into a Keras layer with the KerasLayer class:

*import tensorflow as tf*

*model = tf.keras.Sequential([*

*hub.KerasLayer(MODULE\_HANDLE,input\_shape=(224, 224, 3)),*

*tf.keras.layers.Activation('softmax')*

*])*

Then, you can use your final model to perform predictions:

*model.predict(data)*

You just performed transfer learning with a model from TensorFlow Hub. This is very similar to what you learned previously using the Keras API, where you loaded an entire model with include\_top=True. With TensorFlow Hub, you can access a library of pre-trained models for object detection or image segmentation.

In the next section, you will learn how to extract features from TensorFlow Hub pre-trained modules.

## Feature Extraction

TensorFlow Hub provides the option of downloading a model without the final layer. In this case, you will be using a TensorFlow module as a feature extractor; you can design your custom final layers on top of it. In TensorFlow Hub, a module used for feature extraction is known as a feature vector:

*import tensorflow\_hub as hub*

*MODULE\_HANDLE = 'https://tfhub.dev/google/efficientnet/b0'\*

*'/feature-vector/1'*

*module = hub.load(MODULE\_HANDLE)*

Note

To find all the available feature vectors on TensorFlow Hub, you can use its search engine: <https://tfhub.dev/s?module-type=image-feature-vector&tf-version=tf2>.

Once loaded, you can add your own final layer to the feature vector with the Sequential API:

*model = tf.keras.Sequential([*

*hub.KerasLayer(MODULE\_HANDLE, input\_shape=(224, 224, 3)),*

*tf.keras.layers.Dense(20, activation='softmax')*

*])*

In the preceding example, you added a fully connected layer of 20 units with the softmax activation function. Next, you need to compile and train your model:

*model = tf.keras.Sequential([*

*hub.KerasLayer(MODULE\_HANDLE, input\_shape=(224, 224, 3)),*

*tf.keras.layers.Dense(20, activation='softmax')*

*])*

And with that, you just used a feature vector from TensorFlow Hub and added your custom final layer to train the final model on your dataset.

Now, test the knowledge you have gained so far in the next activity.

## Activity 8.02: Transfer Learning with TensorFlow Hub

In this activity, you are required to correctly classify images of pizzas and steaks using transfer learning. Rather than training a model from scratch, you will benefit from the **EfficientNet B0** feature vector from TensorFlow Hub, which contains pre-computed weights that can recognize different types of objects.

You can find the dataset here

<https://storage.googleapis.com/ztm_tf_course/food_vision/pizza_steak.zip>

The following steps will help you to complete this activity:

1. Import the dataset and unzip the file using TensorFlow.
2. Create a data generator that will perform rescaling.
3. Load a pre-trained **EfficientNet B0** feature vector from TensorFlow Hub.
4. Add two fully connected layers on top of the feature vector:

– A fully connected layer with Dense(500, activation=relu)

– A fully connected layer with Dense(1, activation='sigmoid')

1. Specify an Adam optimizer with a learning rate of 0.001.
2. Train the model.
3. Evaluate the model on the test set.

The expected output is as follows:

A picture containing table

Description automatically generated

Figure 8.7: Model training output

The expected accuracy scores should be around 1.0 for the training and validation sets.

Note

The solution to this activity can be found via [this link](https://subscription.packtpub.com/book/data/9781800205253/8/ch08lvl1sec62/B16341_Solution_ePub.xhtml#_idTextAnchor277).

## Summary

In this chapter, you learned two very important concepts: transfer learning and fine-tuning. Both help deep learning practitioners to leverage existing pre-trained models and adapt them to their own projects and datasets.

Transfer learning is the re-use of models that have been trained on large datasets such as ImageNet (which contains more than 14 million images). TensorFlow provides a list of such pre-trained models in its core API. You can also access other models from renowned publishers such as Google and NVIDIA through TensorFlow Hub.

Finally, you got some hands-on practice fine-tuning a pre-trained model. You learned how to freeze the early layers of a model and only train the last layers according to the specificities of the input dataset.

These two techniques were a major breakthrough for the community as they facilitated access to state-of-the-art models for anyone interested in applying deep learning models.

In the next chapter, you will look at another type of model architecture, **recurrent neural networks** (**RNNs**). This type of architecture is well suited for sequential data such as time series or text.