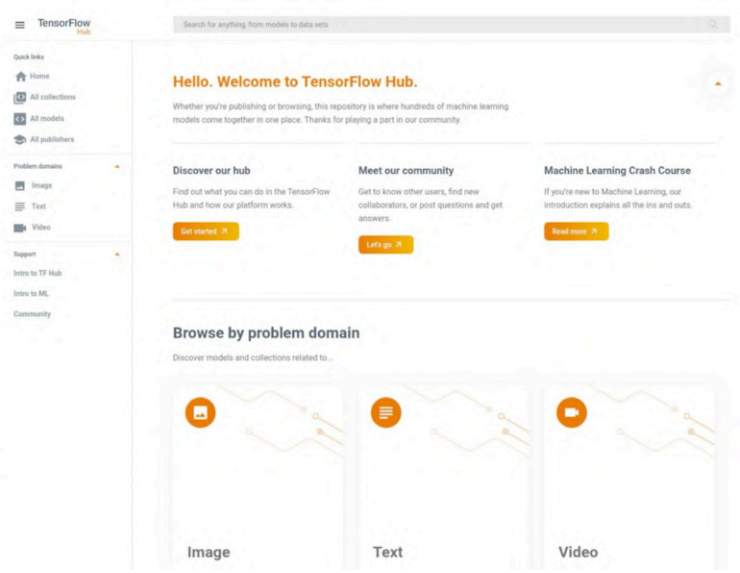
Machine learning is all about building models. Tensorflow hub is an environment where you can share those models with the community. So this week we'll take a look at TensorFlow hub and how you can use models that were deployed to hub for your own projects.

You can find TensorFlow hub at tfhub.de models can be browsed based on a number of axes, including the publisher, the underlying architecture of the problem, domain type.

tfhub.dev



TensorFlow Hub

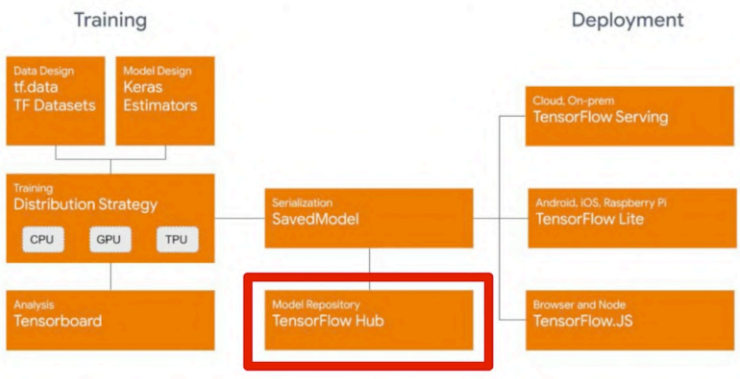
- An extensive library of existing modules
- Well-tested modules designed for better accuracy
- Easy to use and integrate with your model APIs

You start by designing a model with the help of high-level APIs using Keras or estimators. You blend these models with data pipelines using TF data sets, and tf Data TensorFlow supports training on a range of devices like CPU, GPUs, and TPUs and you can visually analyze your training progress using TensorBoard.

Once you're done with training, you can then save your model. And the best way to save them is using the saved model formats. If you're saved model than serve some useful purpose, it could be pushed into a public repository such as TensorFlow hub.

Finally, the saved models can be hosted for deployment through various media like on Cloud or mobile devices or through web browsers, et cetera.

Where does Hub fit in?



Who publishes the modules?

BigGAN

Inflated 3D

Wide ResNets

MobileNet

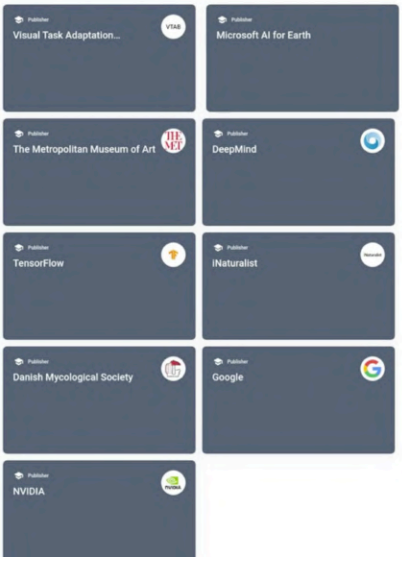
Inception

NASNet

TF-GAN

Image Credits

https://tfhub.dev/s?subtype=publisher



Transfer Learning with TF Hub

- Train models with less data
- Improve generalization
- Speed up training

Perhaps, the most important part of TensorFlow Hub is in how it provides an environment that lets you take advantage of transfer learning.

With transfer learning on top of state of the art modules, the training time of your model can be reduced significantly and you may be able to increase your overall accuracy.

Problem domains

Text
Embedding

Image
Classification
Feature vector
Augmentation
Object Detection
Generator
Style Transfer

Video
Classification

Installation

```
pip install tensorflow_hub
```

Installing Hub is pretty straightforward using a pip install and once it's installed, you can import it like this.

```
import tensorflow_hub as hub
```

Loading a module

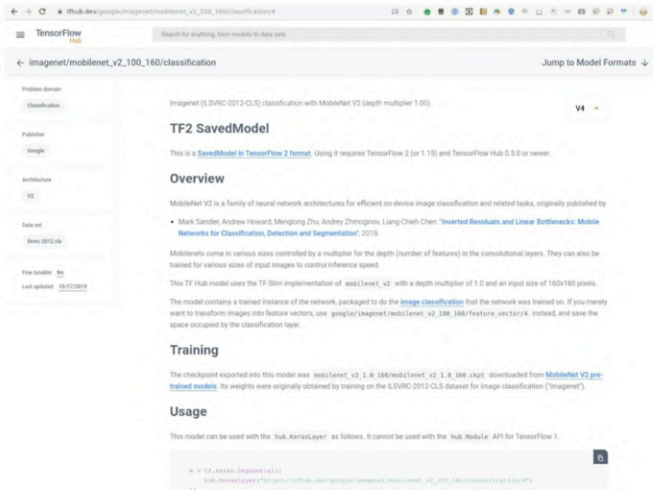
To load a module, you can then use its unique module handle, which is a URL string. To obtain the module handle, the developer has to browse through the catalog of modules in the TensorFlow Hub website and then find it from there.

In this example demonstration, we've displayed the module handle for the mobile net image classification module

Finally, we'll make use of TensorFlow Hub's, load API to load the module into memory.

```
MODULE_HANDLE = 'https://tfhub.dev/google/tf2-preview/mobilenet_v2/classification/4'
module = hub.load(MODULE_HANDLE) # Load a the MobileNet image classification module
```

https://tfhub.dev/google/imagenet/mobilenet_v2_100_160/classification/4

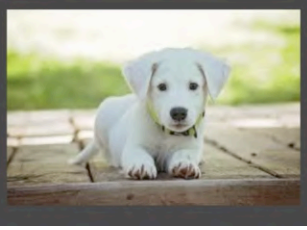


Running inference on the module

```
MODULE_HANDLE = 'https://tfhub.dev/google/tf2-preview/mobilenet_v2/classification/4'
module = hub.load(MODULE_HANDLE)

images = ... # Batches of images
predictions = tf.nn.softmax(module(images))
```

Note however, that the module generates the final layers logits without any activations. Here, we can see we've passed an image of a puppy and these are the top three output classes that we've obtained along with it's probability score.



Label	Probability
Labrador retriever	0.58
kuvasz	0.08
Great Pyrenees	0.07

Using a module with Keras

```
MODULE_HANDLE = 'https://tfhub.dev/google/tf2-preview/mobilenet_v2/classification/4'
OUTPUT_SIZE = 1001
IMAGE_SIZE = (224, 224)

hub.KerasLayer(MODULE_HANDLE,
               output_shape=[OUTPUT_SIZE],
               input_shape=IMAGE_SIZE + (3,))
```

Now, let's see how we can integrate hub modules into high level Keras APIs. We continue with the same procedure of obtaining the module-handle as we'd seen previously. Then, we'll make use of hubs.KerasLayer API to load it.

Using a module with Keras

```
MODULE_HANDLE = 'https://tfhub.dev/google/tf2-preview/mobilenet_v2/classification/4'
OUTPUT_SIZE = 1001
IMAGE_SIZE = (224, 224)

model = tf.keras.Sequential([
    hub.KerasLayer(MODULE_HANDLE,
                   output_shape=[OUTPUT_SIZE], input_shape=IMAGE_SIZE + (3,)),
    tf.keras.layers.Activation('softmax')
])

images = ... # Batches of images
predictions = model.predict(images)
```

Here is the example of how a Keras model is created with the tf hub module. We can use the tf hubs Keras layer and add it using the sequential model along with an activation layer.

Once the model is built, all the Keras model objects can be accessible like you would normally do in Keras.

In this example, we're predicting the classes for a batch of images

Using a Feature Vector

```
MODULE_HANDLE = "https://tfhub.dev/google/tf2-preview/mobilenet_v2/feature_vector/4"
FV_SIZE = 1280
IMAGE_SIZE = (224, 224)

model = tf.keras.Sequential([
    hub.KerasLayer(MODULE_HANDLE,
                   output_shape=[FV_SIZE],
                   input_shape=IMAGE_SIZE + (3,)),
    tf.keras.layers.Dense(NUM_CLASSES, activation='softmax')
])

images = ... # Batches of images
predictions = model.predict(images)
```

You can also use an image feature vector. This is very similar to that of image classification modules, with the only difference being that the final classification had used in the other module is removed. This makes it more comfortable for you to customize your model architecture.

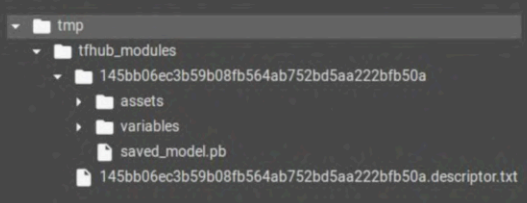
Here's an example of how a feature vector model can be loaded into a Keras sequential model.

Notice, how similar the entire code looks to that of an image classification.

Where do the modules get stored?

At this point in time, you may be wondering what happens in the background when you call the hub.load or the both of them internally initiate the download procedure of the required module if it's not available already. By default, the modules are stored inside the TF Hub modules directory under the temporary folder, which is TMP of your system route. If you take a look inside that directory, you'll find that a TF Hub module is merely a saved model directory.

```
module = hub.load(...)
```



Saving a module for local use

If you to download the Hub module and work on it offline later on, you can do so by explicitly downloading the TF Hub module as a saved model archived as a tarball. To do this you first have to download the Hub module by appending a query parameter to the module handled URL string. This is done by setting the TF Hub format query parameter as shown here. For now, only the compressed option is defined. Next, you need to decompress the tarball and load the on archived save model using tensorflow Hub.

```
MODULE_HANDLE = 'https://tfhub.dev/...?tf-hub-format=compressed'
!wget $MODULE_HANDLE


# Untar the tarball and load it with hub
hub_module = hub.load('/path/to/saved_model')
```

Inspecting the Hub module

```
SAVED_MODEL_DIR = ...
model = tf.saved_model.load(SAVED_MODEL_DIR, tags='serve')

images = ... # Batch of images
predictions = model(images)

>>> print(predictions)
Tensor("StatefulPartitionedCall_2:0", shape=(1, 1001), dtype=float32)
```

Let's now inspect whether the downloaded content from TF Hub is genuinely a saved model or not. Upon experimentation, we'll be able to load the downloaded content using TF saved models load API.

We can make use of serve tag while loading the model. We can then go ahead and call the model with the batch of images that we want to evaluate.

Relocating TF Hub modules

Lastly, if you want to change the download location of modules to a more permanent location, you can do so by setting the environment variable TFHUB cache directory.

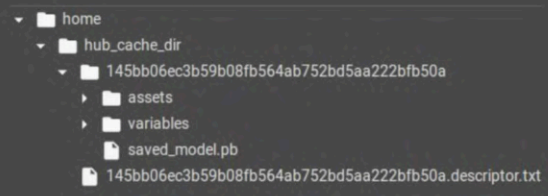
In Python, you can set this environment variable in the environment dictionary that's present in the Pythons OS module as you can see here. If you wish to do this directly in bash a terminal you'll have to set this environment variable by exporting it as shown.

```
import os
os.environ['TFHUB_CACHE_DIR'] = '/home/hub_cache_dir'


export TFHUB_CACHE_DIR='/home/hub_cache_dir'
```

Relocating TF Hub modules

```
module = hub.load(...)
```



Once you set the new location of TF Hub cache directory environment variable all the subsequent models that you quest will get downloaded to that location.

IMDB Reviews

we'll explore some of the model types and TensorFlow hub and dive into how you can reuse them. We'll start with a text-based model. Let's create a text classification model to classify the positive and negative reviews presence in the IMDB reviews dataset. This dataset is comprised of 50,000 samples of reviews given by users on a wide range of movies. We'll download this dataset using TensorFlow datasets, and the dataset has been divided equally into training and test splits. Since we would like to validate the progress of our training for every epoch during training, we would then extract some parts of the dataset from the training sets to create a validation set.

```
train_validation_split = tfds.Split.TRAIN.subsplit([6, 4])


(train_data, validation_data), test_data = tfds.load(
    name="imdb_reviews",
    split=(train_validation_split, tfds.Split.TEST),
    as_supervised=True)
```

Explore the dataset

Let's take a moment to understand the format of the data. Each example is a sentence representing the movie review and a corresponding label. The sentence is not preprocessed in any way, the label is an integer value of either zero or one where zero indicates a negative review and one is a positive review.

```
train_examples_batch, train_labels_batch = next(iter(train_data.batch(10)))
>>> train_examples_batch.numpy()


array([b"As a lifelong fan of Dickens, I have invariably been disappointed by ...",
       b"I absolutely LOVED this movie when I was a kid. I cried every time I ...",
       ...
       dtype=object)
```

Building a model for text classification

Architectural decisions

- How to **represent** the text?
- How many **layers** to use in the model?
- How many **hidden units** to use for each **layer**?

Given data like this, what are the questions that you might ask about how to build a model for this data.? Since we're working with sentences in a text-based dataset, let's think about what are model needs to look like. Neural networks work with numeric values and so there's some way we need to represent our text by transforming it into something numeric.

But what are the other preprocessing steps we might have to do? Is it possible that we could leverage something from TensorFlow hub for these problems? Then what should the width and depth of our models be?

Embeddings to choose from

Typically, texts is represented with word embeddings. Word embeddings, are an efficient way to represent words using dense vectors of floating point values with semantically similar words having similar directions of vector. The length of embedding vectors can be small as an eight-dimensional one all the way up to be as large as 1024 dimensions, depending on the number of words you have in your dataset. When training a model from scratch, a huge part of the training is in learning these embeddings. Being able to use pre-learned ones can actually save you a lot of time. For this purpose, TensorFlow Hub provides us with a collection of pre-trained word embeddings created out of different global languages.

- google/tf2-preview/**gnews-swivel-20dim**(-with-oov)/1
- google/tf2-preview/**nnlm-en-dim50**/1
- google/tf2-preview/**nnlm-en-dim128**/a1

The gnews-swivel embeddings, were learned from a dataset of about 130 gigabytes of English Google News with 20,000 unique words in its vocabulary. It comes in two flavors. There's one with a token for out of vocabulary or OOV and there's another without it.

The second word embedding, is a much larger one with a vocabulary size of approximately 1 million words and it's trained on 50 dimensional word vectors.

The third is an alternate version of this with a 1 million words using 128 dimension embeddings.

We would encourage you to try out all of these and compare each of their impact in accuracy, latency, and memory consumption for your model.

Experimenting with embeddings

We'll use the gnews-swivel embedding which is found at this URL.

```
embedding = "https://tfhub.dev/google/tf2-preview/gnews-swivel-20dim/1"
hub_layer = hub.KerasLayer(embedding, input_shape=[], dtype=tf.string, trainable=True)


>>> hub_layer(train_examples_batch[:3])
<tf.Tensor: id=910, shape=(3, 20), dtype=float32, numpy=
array([[ 3.9819887 , -4.4838037 ,  5.177359  , -2.3643482 , -3.2938678 ,
        -3.5364532 , -2.4786978 ,  2.5525482 ,  6.688532  , -2.3076782 ,
        -1.9807833 ,  1.1315885 , -3.0339816 , -0.7604128 , -5.743445  ,
         3.4242578 ,  4.790099  , -4.03061   , -5.992149  , -1.7297493 ],
         ...
      dtype=float32)>
```

Experimenting with embeddings

```
embedding = "https://tfhub.dev/google/tf2-preview/gnews-swivel-20dim/1"
hub_layer = hub.KerasLayer(embedding, input_shape=[], dtype=tf.string, trainable=True)
```

```
>>> hub_layer(train_examples_batch[:3])
<tf.Tensor: id=910, shape=(3, 20), dtype=float32, numpy=
array([[ 3.9819887 , -4.4838037 ,  5.177359  , -2.3643482 , -3.2938678 ,
        -3.5364532 , -2.4786978 ,  2.5525482 ,  6.688532  , -2.3076782 ,
        -1.9807833 ,  1.1315885 , -3.0339816 , -0.7604128 , -5.743445  ,
         3.4242578 ,  4.790099  , -4.03061   , -5.992149  , -1.7297493 ],
         ...
      dtype=float32)>
```

We can then take a Keras layer out of this embedding with this code. During training, you can further fine-tune the hub module weights by setting the trainable parameter to True.

Experimenting with embeddings

```
embedding = "https://tfhub.dev/google/tf2-preview/gnews-swivel-20dim/1"
hub_layer = hub.KerasLayer(embedding, input_shape=[], dtype=tf.string, trainable=True)
```

```
>>> hub_layer(train_examples_batch[:3])
<tf.Tensor: id=910, shape=(3, 20), dtype=float32, numpy=
array([[ 3.9819887 , -4.4838037 ,  5.177359  , -2.3643482 , -3.2938678 ,
        -3.5364532 , -2.4786978 ,  2.5525482 ,  6.688532  , -2.3076782 ,
        -1.9807833 ,  1.1315885 , -3.0339816 , -0.7604128 , -5.743445  ,
         3.4242578 ,  4.790099  , -4.03061   , -5.992149  , -1.7297493 ],
         ...
      dtype=float32)>
```

We'll display the output for the first three samples in our training sets. Notice one strange thing here. The shape of the output sensor is three by 20, with three as the number of samples passed and 20 is the dimension of the output vector for each sample. Now, how is it that each sample that's comprised of a multiple amount of words only gets converted to a 20 dimensional vector? On top of that, this is when the output of the word embedding for each word is 20 dimensional vector.

It turns out that when you pass sentences to these word embedding modules, they intelligently combine the individual word embeddings into sentence embeddings. The sentence embeddings will then have the same dimension as that of the original word embedding.

Create the model

```
model = tf.keras.Sequential([
                hub_layer,
                tf.keras.layers.Dense(16, activation='relu'),
                tf.keras.layers.Dense(1, activation='sigmoid')])
```

We can now build a model around this hub layer by placing it within a sequential model, adding a dense layer and a final output layer with a single neuron.

Train the model

```
# Compile the model
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Start training
model.fit(train_data.shuffle(10000).batch(512),
          epochs=20,
          validation_data=validation_data.batch(512),
          verbose=1)
```

Time to evaluate our model

```
results = model.evaluate(test_data.batch(512), verbose=2)

>>> for name, value in zip(model.metrics_names, results):
      print("%s: %.3f" % (name, value))
loss: 0.318
accuracy: 0.865
```

Classify Cats and Dogs

```
splits = tfds.Split.ALL.subsplit(weighted=(80, 10, 10))

splits, info = tfds.load('cats_vs_dogs', with_info=True, as_supervised=True, split=splits)

(train_examples, validation_examples, test_examples) = splits

num_examples = info.splits['train'].num_examples
num_classes = info.features['label'].num_classes
```

We'll go to a very common scenario, image classification and how you can use a TensorFlow Hub module, a set of feature vectors that's trained on mobile net, to make training a cats versus dogs classifier very quick and very accurate. Cats and dogs is available in TFDS. So you can download and access the full data set with just a few lines of code.

Here you can see we're getting the data set and splitting it into a training set of 80 percent, a test set of 10 percent and a validation set also of 10 percent.

Prepare datasets

```
def format_image(image, label):
  # Resizes and normalizes the image
  image = tf.image.resize(image, IMAGE_SIZE) / 255.0
  return image, label

# Crate train and validation datasets
train_batches = train_examples.shuffle(SHUFFLE_SIZE)
                              .map(format_image)
                              .batch(BATCH_SIZE)
                              .prefetch(tf.data.experimental.AUTOTUNE)
validation_batches = validation_examples.map(format_image)
                                        .batch(BATCH_SIZE)
                                        .prefetch(tf.data.experimental.AUTOTUNE)
```

Select feature vector module

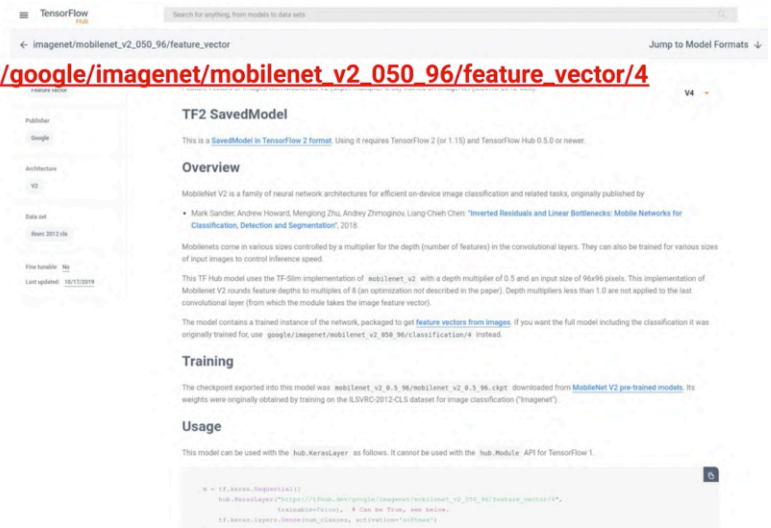
```
# Pick a feature vector module. e.g., InceptionV3, MobileNetV2, ...
handle_base = "mobilenet_v2"
MODULE_HANDLE = "https://tfhub.dev/google/tf2-preview/{}/feature_vector/4"
                .format(handle_base)

# This is size of the feature vector
FV_SIZE = 1280 # For MobileNetV2

IMAGE_SIZE = (224, 224)
```

Hub has lots of feature vector modules. So we'll take a look at using MobileNetV2. We do this by specifying the module handle that we want to use and then we'll then see loading this from hub in a moment. We'll also specify the size of the feature vector and the image size as 224 by 224.

https://tfhub.dev/google/imagenet/mobilenet_v2_050_96/feature_vector/4



You can learn more about each module on their requisite page on Tfhub.dev.

Transfer Learning with TF Hub

```
MODULE_HANDLE = 'https://tfhub.dev/google/tf2-preview/mobilenet_v2/feature_vector/4'

feature_extractor = hub.KerasLayer(MODULE_HANDLE,
                                   input_shape=IMAGE_SIZE + (3,),
                                   output_shape=[FV_SIZE],
                                   trainable=True)
```

```
>>> len(feature_extractor.variables)
260
>>> len(feature_extractor.trainable_variables)
156
>>> len(feature_extractor.non_trainable_variables)
104
```

You can see that we've set up our module handle as a string pointing to it on tfhub.dev, so we can just create a keras layer from this. We'll specify our input shape as the image size we created earlier, but with an additional dimension of three to denote color images. Our output shape will be the feature vector size we created earlier and we'll set it to be trainable.

Build model with TF Hub module

```
# Fine-tuning switch
do_fine_tuning = True

feature_extractor = hub.KerasLayer(MODULE_HANDLE,
                                   input_shape=IMAGE_SIZE + (3,),
                                   output_shape=[FV_SIZE],
                                   trainable=do_fine_tuning)
```

```
# Build model with the chosen module handle
model = tf.keras.Sequential([
    feature_extractor,
    tf.keras.layers.Dense(2, activation='softmax') # (cat, dog)
])
```

Now, it's simple enough to create a keras sequential network starting with the feature extractor we just created and adding a tuner on dense after it to be our classifier.

Model summary

```
>>> model.summary()

Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
keras_layer (KerasLayer)     (None, 1280)              2257984
_________________________________________________________________
dense (Dense)                (None, 2)                 2562
=================================================================
Total params: 2,260,546
Trainable params: 2,226,434
Non-trainable params: 34,112
_________________________________________________________________
```

Choose the right training configuration

We can define our loss function and the desired metrics which in this case is just accuracy. And if we choose to do fine-tuning of the network, we can use an SGD Optimizer to which we'll pass the learning rate parameter. Otherwise, we could just use something like an off-the-shelf adam optimizer.

```
# Define loss and metrics
loss = 'sparse_categorical_crossentropy'
metrics = ['accuracy']

# Choose an appropriate optimizer when fine-tuning
if do_fine_tuning:
  optimizer = tf.keras.optimizers.SGD(lr=0.002, momentum=0.9)
else:
  optimizer = 'adam'
```

Train the model

To train, we compile the model specifying the loss function, the optimizer and the desired metrics and then we call model.fit as usual.

```
# Compile the model and fit
model.compile(optimizer=optimizer, loss=loss, metrics=metrics)
model.fit(train_batches, epochs=5,
          validation_data=validation_batches)
```