APPENDIX A

Introduction to Keras

Keras is an API designed to make developing neural network easy for humans. It is also the API that we use throughout this book. The goal of this appendix is not to cover all aspects of Keras (surely the space would not be enough), but to give you the minimum amount of information that you need to understand the code in this book and then point you to resources where you can find more. If you want to learn everything about Keras, the most efficient way is to study the book *Deep Learning with Python* by François Chollet and to read the official documentation at `https://keras.io`.

In a few words, citing F. Chollet,[1] "TensorFlow is an infrastructure layer for differentiable programming, dealing with tensors, variables, and gradients. Keras is a user interface for deep learning, dealing with layers, models, optimizers, loss functions, metrics, and more."

If you are looking for custom training loops, custom layers, and so on, check out Appendix B, where we cover those topics, although briefly.

Some History

Keras was developed by F. Chollet, and its first version was made available on March 27, 2015. Up to and including version 2.3, Keras needed a backend, in other words a system that performed the low-level operations needed by your code. At the very beginning Keras did not work with TensorFlow (the first supported backend was Theano). The `tf.keras` package was introduced in TensorFlow 1.10.0. Note that these two imports are very different things:

```
import keras
```

and

```
from tensorflow import keras
```

[1] `https://keras.io/getting_started/intro_to_keras_for_researchers/`

U. Michelucci, *Applied Deep Learning with TensorFlow 2*, https://doi.org/10.1007/978-1-4842-8020-1

After Keras 2.3.0, F. Chollet declared that this release will be in sync with `tf.keras` and that practitioners should use `tf.keras` and not `keras` anymore. After this release, `keras` will not support multiple backends.

Note You should always use `from tensorflow import keras` in your code.

Understanding the Sequential Model

A sequential model is simply a plain stack of layers, where each has one input and one output tensor. The easiest way to create a sequential model is by providing a list of layers to the `keras.Sequential` call. For example

```
model = keras.Sequential(
    [
        layers.Dense(2, activation="relu"),
        layers.Dense(2, activation="relu")
    ]
)
```

In this code, it is assumed you have imported

```
from tensorflow.keras import layers
```

There is an alternative way to create a sequential model and that is by using the `add()` method. The network defined above could be also created with

```
model = keras.Sequential()
model.add(layers.Dense(2, activation="relu"))
model.add(layers.Dense(2, activation="relu"))
```

The two versions of the code are completely equivalent. This second version is a bit more readable than the first when the number of layers is large.

There are situations when the sequential model is not appropriate. For example:[2]

- Your model has multiple inputs or multiple outputs
- Any of your layers have multiple inputs or multiple outputs

[2] As taken from the official documentation at https://keras.io/guides/sequential_model/.

- You need to do layer sharing
- You want a non-linear topology (e.g., a residual connection, a multi-branch model, etc.)

For those cases, you need the functional APIs, as described later in this appendix.

Understanding Keras Layers

Layers are a fundamental part of Keras. A *layer* includes a state (the weights of the neurons for example) and some computation (implemented in the `call` method). Keras offers a lot of layers that you can use without having to develop your own. The most commonly used (and probably the ones you may have seen so far) are the following:[3]

- Dense: Like the ones we saw in the FFNN discussion
- Conv1D, Conv2D, and Conv3D: Convolutional layers in multiple dimensions
- MaxPooling1D, MaxPooling2D and MaxPooling3D: Max-pooling layers
- AveragePooling1D, AveragePooling2D and AveragePooling3D: Average-pooling layers
- LSTM layers
- Regularization layers as `Dropout`

And many more. Remember that any operation that takes a tensor as input and gives a tensor as output is a layer in the Keras language. For example, flattening a 2D image into a 1D vector is also a layer (see the `Flatten` layer). Also, reshaping an input can be done with a layer (see the `Reshape` layer). Even applying an activation function can be done with a layer (see the `ReLU` layer for example).

Note Remember that any operation that takes a tensor as input and gives a tensor as output is a layer in the Keras language.

---

[3] If you want to see the complete list, consult the official documentation at https://keras.io/api/layers/.

In Appendix B, we briefly discuss how to develop your own layers. Note that you can also easily do the following things with layers:

- Retrieve the gradients (see Appendix B)
- Retrieve the weights (see Appendix B)
- Add regularization losses (as discussed in the main part of the book)
- Set the weights to values of your choosing (see Appendix B)
- Use initializers for the weights (for example, He, Glorot, etc.)

Setting the Activation Function

To set the activation function in layers, use the `activation` property. For example, the code

```
layers.Dense(2, activation="relu")
```

creates a layer with two neurons and the `ReLU` activation function. Note that if you don't specify any activation, `none` is used (or, in other words, the identity function is used as the activation function). As usual, Keras offers many activation functions:[4] `relu`, `sigmoid`, `softmax`, `softplus`, `softsign`, `tanh`, `selu`, `elu`, and `exponential function`.

Using Functional APIs

The functional Keras APIs offer a way to create models that are not linear, with shared layers or with multi-input or multi-output layers (or both). The idea behind it is that neural networks are normally a directed acyclic graph, so with the functional APIs you can build graphs of layers (therefore, you can build non-linear architectures). For example, the previous network would be built as follows

```
inputs = keras.Input(shape=(...,))
x = layers.Dense(2, activation="relu")(inputs)
x = layers.Dense(2, activation="relu")(x)
```

[4] As of November 2021.

where we have added an input layer since it is needed when using the functional APIs. As you can see, each layer is "applied" to another one. Or in graph language, it's like drawing a connection between two layers. This model can be graphically plotted with Keras,[5] as shown in Figure A-1.

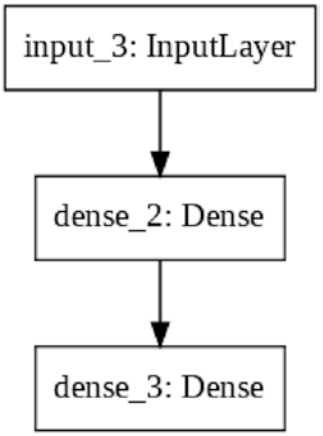

Figure A-1. *A graphical representation of the small network defined in the text as a graph*

For example, the code

```
inputs = keras.Input(shape=(784,))
x1 = layers.Dense(2, activation="relu")(inputs)
x2 = layers.Dense(2, activation="relu")(x1)
y = layers.Dense(2, activation="relu")(x1)
model = keras.Model(inputs, outputs = [x2,y])
```

would give you the architecture shown in Figure A-2.

[5] To plot a model you can use the useful call `keras.utils.plot_model(model, "...")`. Swap the three dots with the filename that you want to use.

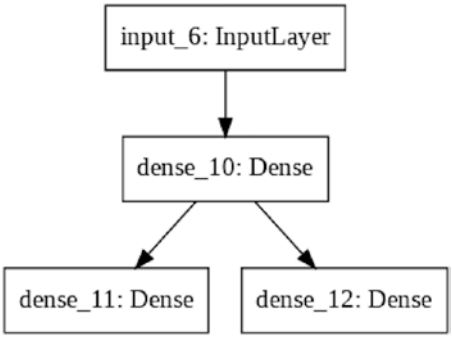

Figure A-2. *A graphical representation of a network with multiple inputs built with the Keras Functional API*

You can get really creative with the architectures you can create. You will find lots of examples in the official documentation at https://keras.io/guides/functional_api/.

Specifying Loss Functions and Metrics

To train any neural network, you need of course to specify which loss function you want to minimize and which optimizer you want to use. In Keras, this is achieved by calling the `compile()` method. For example, if you want to use Adam as the optimizer and the MSE as the loss function, you use

```
model.compile(optimizer='Adam', loss='mse')
```

Putting It All Together and Training

The easiest way to train a model in Keras involves three steps:

1. Create the network by specifying the architecture (the number of layers, number of neurons, types of layers, activation functions, etc.). You would do this in the examples with, for example, `keras.Sequential()`.
2. Compile the model with the `compile()` method. This step specifies which loss function and which metrics Keras should use.

3. Train the model by using the `fit()` method. In the `fit()` method you can specify the number of epochs, batch size, and many other parameters.

Here is a minimal example

```
model = keras.Sequential(
    [
        layers.Dense(2, activation="relu"),
        layers.Dense(3, activation="relu"),
        layers.Dense(1),
    ]
)
```

This specifies the network architecture. After that, you compile the model with the `compile()` method.

```
model.compile(optimizer=Adam, loss='mse')
```

where you specify the optimizer Adam and the MSE loss. After that, you can train the model

```
model.fit(x, y, batch_size=32, epochs=10)
```

where `x` indicates the inputs, `y` indicates the labels, the `batch_size` is specified as `32`, and you want to train for ten epochs.

Note The easiest approach to creating and training a model in Keras involves three steps: 1) Create the architecture, 2) Compile the model with `compile()`, and 3) Train the model with the `fit()` method.

The `fit()` method accepts lots of parameters. You can specify:

- How much output you want by specifying the `verbose` parameter (`0` for no output, `1` for a progress bar, and `2` for one line for each epoch).
- Actions at different points during the training by specifying which `callbacks` functions the `fit()` method should use. For more information on callbacks, see the next sections.

- A validation dataset by simply giving a `validation_split` parameter (that is, the fraction of the data that you want to use as the validation dataset). The `fit()` method will give you the metrics for this dataset.

You can specify many more options. As usual, to get a complete overview, check out the official documentation at https://keras.io/api/models/model_training_apis/.

Modeling *evaluate*() and *predict*()

Once you have trained your model, you can use the `evaluate()` method. It will return the loss and the metrics applied to the inputs in test mode.[6] For example, a call would look like

```
model.evaluate(x,y)
```

And finally you can use the `predict()` method to generate predictions for the input samples. A call would look like

```
model.predict(x)
```

Using Callback Functions

Callback functions are a powerful way to customize training of a model. They can be used with the `fit()`, `evaluate()`, and `predict()` functions.

It is instructive to understand a bit better what Keras callback functions are, since they are used quite often when developing models. From the official documentation[7]

> *A callback is a set of functions to be applied at given stages of the training procedure.*

The idea is that you can pass a list of callback functions to the `.fit()` method of the `Sequential` or `Model` classes. Relevant methods of the callbacks will then be called at each stage of the training. Their use is rather easy. For the `fit()` function, you would use them as

[6] This is relevant when, for example, dealing with a dropout that has a different behavior during training or during testing.

[7] https://keras.io/callbacks/

```
model.fit(
    ...,
    callbacks=[Callback()],
)
```

Where `Callback()` is a placeholder name for a callback (you need to change it to the callback function name you want to use). There are callbacks functions that perform many tasks, such as

- `ModelCheckPoint` saves weights and models at specific frequencies
- `LearningRateScheduler` changes the learning rate according to some schedule
- `TerminateOnNaN` stops the training process if NaN appears (so you don't waste time or computing resources)
- And many more

As usual, you can find more information on the official documentation at https://keras.io/api/callbacks/. In Appendix B, I discuss how to develop your own custom callback class, since this is one of the best ways to check and control the training process at various stages.

Saving and Loading Models

It is often useful to save a model on disk, so you can continue the training at a later stage or reuse a previously trained model. To learn how to do this, let's consider the MNIST dataset for the sake of giving a concrete example.[8]

You need the following `imports`

```
import os
import tensorflow as tf
from tensorflow import keras
```

Load the MNIST dataset again and take the first 5,000 observations.

[8] The example was inspired by the official Keras documentation at https://www.tensorflow.org/tutorials/keras/save_and_restore_models.

```
(train_images, train_labels), (test_images, test_labels) = tf.keras.
datasets.mnist.load_data()
train_labels = train_labels[:5000]
test_labels = test_labels[:5000]
train_images = train_images[:5000].reshape(-1, 28 * 28) / 255.0
test_images = test_images[:5000].reshape(-1, 28 * 28) / 255.0
```

Let's now build a simple Keras model with a `Dense` layer with 512 neurons, a bit of dropout, and the classical ten-neuron output layer for classification (remember the MNIST dataset has ten classes).

```
model = tf.keras.models.Sequential([
    keras.layers.Dense(512, activation=tf.keras.activations.relu, input_
    shape=(784,)),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(10, activation=tf.keras.activations.softmax)
  ])

model.compile(optimizer='adam',
              loss=tf.keras.losses.sparse_categorical_crossentropy,
              metrics=['accuracy'])
```

We have added a bit of dropout, since this model has 407,050 trainable parameters. You can check this number simply by using `model.summary()`.

What we need to do is first define where we want to save the model on the disk. And we can do that (for example) in this way

```
checkpoint_path = "training/cp.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)
```

After that, we need to use a callback (remember what we did in the last section) that will save the weights[9]

```
cp_callback = tf.keras.callbacks.ModelCheckpoint(checkpoint_path,
                                                 save_weights_only=True,
                                                 verbose=1)
```

[9] The `ModelCheckpoint` callback is a standard Keras callback that you can use. You don't need to develop one yourself.

Note that we don't need to define a class as we did in the previous section, since `ModelCheckpoint` inherits from the `Callback` class.

Then we can simply train the model, specifying the correct callback function

```
model.fit(train_images, train_labels,  epochs = 10,
          validation_data = (test_images,test_labels),
          callbacks = [cp_callback])
```

If you check the contents of the folder where your code is running, you should see at least three files:

- `cp.ckpt.data-00000-of-00001`: Contains the weights (if the number of weights is big, you will see many files like this one)
- `cp.ckpt.index`: Contains information about which weights are in which file
- `checkpoint`: Contains information about the checkpoint itself

We can now test our method. This code will give you a model that will reach an accuracy on the validation dataset of roughly 92%.

If we define a second model

```
model2 = tf.keras.models.Sequential([
    keras.layers.Dense(512, activation=tf.keras.activations.relu, input_
    shape=(784,)),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(10, activation=tf.keras.activations.softmax)
  ])

model2.compile(optimizer='adam',
               loss=tf.keras.losses.sparse_categorical_crossentropy,
               metrics=['accuracy'])
```

and we check its accuracy on the validation dataset with

```
loss, acc = model2.evaluate(test_images, test_labels)
print("Untrained model, accuracy: {:5.2f}%".format(100*acc))
```

we will get an accuracy of roughly 8.6%, That was expected, since this model has not been trained yet. But now we can load the saved weights in this model and try again.

```
model2.load_weights(checkpoint_path)
loss,acc = model2.evaluate(test_images, test_labels)
print("Second model, accuracy: {:5.2f}%".format(100*acc))
```

We should get this result

```
5000/5000 [==============================] - 0s 50us/step
Restored model, accuracy: 92.06%
```

That makes again sense, since the new model is now using the weights of the old, trained model. Keep in mind that, to load pretrained weights in a new model, the model needs to have the exact same architecture as the one used to save the weights.

Note To use saved weights with a new model, the model must have the exact same architecture as the one used to save the weights. Using pretrained weights can save you quite a lot of time, since you don't need to waste time in training the network again.

As you will see again and again, the basic idea is to use a callback that will save your weights. Of course, you can customize the callback function. For example, if you want to save the weights every 100 epochs with a different filename each time, so that you could decide to restore a specific check point, you need first to define the filename in a dynamic way as

```
checkpoint_path = "training/cp-{epoch:04d}.ckpt"
checkpoint_dir = os.path.dirname(checkpoint_path)
```

You should use the following callback

```
cp_callback = tf.keras.callbacks.ModelCheckpoint(
    checkpoint_path, verbose=1, save_weights_only=True,
    period=1)
```

Note that `checkpoint_path` can contain named-formatting options (in the name we have `{epoch:04d}`), which will be filled by the values of `epoch` and `logs` (passed in `on_epoch_end`, which you saw in the previous section).[10] You can check the original code

[10] Check out the official documentation at https://goo.gl/SnKgyQ.

for `tf.keras.callbacks.ModelCheckpoint` and you will find that the formatting is done in the `on_epoch_end(self, epoch, logs)` method.

```
filepath = self.filepath.format(epoch=epoch + 1, **logs)
```

You can define your filename using the epoch number and the values contained in the `logs` dictionary.

Let's get back to our example. Let's start by saving a first version of the model

```
model.save_weights(checkpoint_path.format(epoch=0))
```

and then we can fit the model as usual

```
model.fit(train_images, train_labels,
          epochs = 10, callbacks = [cp_callback],
          validation_data = (test_images,test_labels),
          verbose=0)
```

Be careful since this will save lots of files. In our example, one every epoch. So for example, the directory content may look like this:

```
checkpoint                   cp-0006.ckpt.data-00000-of-00001
cp-0000.ckpt.data-00000-of-00001  cp-0006.ckpt.index
cp-0000.ckpt.index           cp-0007.ckpt.data-00000-of-00001
cp-0001.ckpt.data-00000-of-00001  cp-0007.ckpt.index
cp-0001.ckpt.index           cp-0008.ckpt.data-00000-of-00001
cp-0002.ckpt.data-00000-of-00001  cp-0008.ckpt.index
cp-0002.ckpt.index           cp-0009.ckpt.data-00000-of-00001
cp-0003.ckpt.data-00000-of-00001  cp-0009.ckpt.index
cp-0003.ckpt.index           cp-0010.ckpt.data-00000-of-00001
cp-0004.ckpt.data-00000-of-00001  cp-0010.ckpt.index
cp-0004.ckpt.index           cp.ckpt.data-00000-of-00001
cp-0005.ckpt.data-00000-of-00001  cp.ckpt.index
cp-0005.ckpt.index
```

A last tip before moving on is how to get the latest checkpoint, without bothering to search for the filename. This can be done easily with the following code

```
latest = tf.train.latest_checkpoint('training')
model.load_weights(latest)
```

This will automatically load the weights saved in the latest checkpoint. The variable `latest` is simply a string and contains the last checkpoint filename saved. In this example, that is `training/cp-0010.ckpt`.

Note The checkpoint files are binary files that contain the weights of your model. You will not be able to read them directly, and you should not need to.

Saving Your Weights Manually

Of course, you can simply save your model weights manually when you are done training, without defining a callback function

```
model.save_weights('./checkpoints/my_checkpoint')
```

This command will generate three files, all starting with the string you gave as a name—in this case, `my_checkpoint`. Running this code will generate the three files we described previously:

```
checkpoint
my_checkpoint.data-00000-of-00001
my_checkpoint.index
```

Reloading the weights in a new model is as simple as this:

```
model.load_weights('./checkpoints/my_checkpoint')
```

Keep in mind, that to be able to reload saved weights in a new model, the old model must have the same architecture as the new one. It must be exactly the same.

Saving the Entire Model

Keras also gives us a way to save the entire model on disk: weights, the architecture, and the optimizer. We can re-create the same model by simply moving some files. For example, we could use the following code

```
model.save('my_model.h5')
```

This will save the entire model in one file, called `my_model.h5`. We can simply move the file to a different computer and re-create the same trained model with

```
new_model = keras.models.load_model('my_model.h5')
```

Note that this model will have the same trained weights as your original model, so it's ready to use. This may be helpful if you want to stop training your model and continue the training on a different machine, for example. Or maybe you must stop the training for a while and continue at a later time.

Conclusion

This appendix presented a very quick and superficial overview of Keras with the goal of giving you enough information to start programming basic neural networks with Keras and to understand the code discussed in this book. I hope this short appendix provided a good overview of the fundamentals concepts and methods of Keras.

APPENDIX B

Customizing Keras

This appendix looks in more detail at the code used to build the GAN. If you studied Chapter 11, you will have realized that we did not use the `compile()/fit()` approach, but instead built a custom training loop. It is important that you understand the fundamental concepts of how this works with Keras. This appendix is here exactly for that reason.

This appendix does not cover custom loss functions, custom layers, or custom activation functions. If you are interested in these topics, you will find plenty of examples in the official documentation.

This appendix is intended as a very short reference that I hope will help you quickly understand how to customize Keras and understand the code in Chapter 11 on GANs. I also added for reference a short section on how to customize callback classes. I hope it is useful.

A complete overview on how to customize Keras would require a book of its own[1] and is not the goal of this book.

Customizing Callback Classes

In Appendix A, you learned what callback functions are. In this section, you will see how you can customize them for your purposes, since this is a really useful thing even when you are using the `compile()/fit()` approach. To do this, you need to understand how the abstract base class `keras.callbacks.Callback` works.

The abstract base class `Callback` can be found (at the moment of this writing) at `tensorflow/python/keras/callbacks.py`.

[1] In case you are looking for such a book, a very good introduction is the book by Jojo Moolayil, entitled *Learn Keras for Deep Neural Networks: A Fast-Track Approach to Modern Deep Learning with Python,* and published by Apress.

U. Michelucci, *Applied Deep Learning with TensorFlow 2*, https://doi.org/10.1007/978-1-4842-8020-1

To start customizing, you need simply to define a custom class that inherits from `keras.callbacks.Callback`. The main methods you want to redefine are the following:

- `on_train_begin`: Called at the beginning of training
- `on_train_end`: Called at the end of training
- `on_epoch_begin`: Called at the start of an epoch
- `on_epoch_end`: Called at the end of an epoch
- `on_batch_begin`: Called right before processing a batch
- `on_batch_end`: Called at the end of a batch

This can be done with the following code

```
from tensorflow import keras
class My_Callback(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        # Your code here
       return

    def on_train_end(self, logs={}):
        # Your code here
        return

    def on_epoch_begin(self, epoch, logs={}):
             # Your code here
             return

    def on_epoch_end(self, epoch, logs={}):
             # Your code here
             return

    def on_batch_begin(self, batch, logs={}):
             # Your code here
             return

    def on_batch_end(self, batch, logs={}):
             # Your code here
             self.losses.append(logs.get('loss'))
             return
```

Each of these methods has slightly different inputs that you may use in your class. Let's look at them briefly:

on_epoch_begin, on_epoch_end

```
Arguments:

        epoch: integer, index of epoch.

        logs: dictionary of logs.
```

on_train_begin, on_train_end

```
Arguments:

        logs: dictionary of logs.
```

on_batch_begin, on_batch_end

```
Arguments:

        batch: integer, index of batch within the current epoch.

        logs: dictionary of logs.
```

Let's see with an example how we can use this class.

Example of a Custom Callback Class

Let's again consider the MNIST example. This is the same code you have seen many times by now:

```
import tensorflow as tf
from tensorflow import keras
(train_images, train_labels), (test_images, test_labels) = tf.keras.
datasets.mnist.load_data()

train_labels = train_labels[:5000]
test_labels = test_labels[:5000]

train_images = train_images[:5000].reshape(-1, 28 * 28) / 255.0
test_images = test_images[:5000].reshape(-1, 28 * 28) / 255.0
```

Let's define a `Sequential` model for our example

```
model = tf.keras.models.Sequential([
    keras.layers.Dense(512, activation=tf.keras.activations.relu,
    input_shape=(784,)),
    keras.layers.Dropout(0.2),
    keras.layers.Dense(10, activation=tf.keras.activations.softmax)
  ])

model.compile(optimizer='adam',
               loss=tf.keras.losses.sparse_categorical_crossentropy,
               metrics=['accuracy'])
```

Now let's write a custom callback class, redefining only one of the methods to see what the inputs are. For example, let's see what the variable `logs` contains at the beginning of the training

```
class CustomCallback1(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        print (logs)
        return
```

We can then use it with

```
CC1 = CustomCallback1()
model.fit(train_images, train_labels,  epochs = 2,
          validation_data = (test_images,test_labels),
          callbacks = [CC1])  # pass callback to training
```

Remember to always instantiate the class and pass the `CC1` variable, and not the class itself. We will get

```
Train on 5000 samples, validate on 5000 samples
{}
Epoch 1/2
5000/5000 [==============================] - 1s 274us/step - loss: 0.0976 -
acc: 0.9746 - val_loss: 0.2690 - val_acc: 0.9172
Epoch 2/2
5000/5000 [==============================] - 1s 275us/step - loss: 0.0650 -
acc: 0.9852 - val_loss: 0.2925 - val_acc: 0.9114
{}
```

```
<tensorflow.python.keras.callbacks.History at 0x7f795d750208>
```

The logs dictionary is empty, as you can see from the {}. Let's expand the class a bit

```
class CustomCallback2(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
        print (logs)
        return

    def on_epoch_end(self, epoch, logs={}):
        print ("Just finished epoch", epoch)
        print (logs)
        return
```

Now we train the network with

```
CC2 = CustomCallback2()
model.fit(train_images, train_labels,  epochs = 2,
          validation_data = (test_images,test_labels),
          callbacks = [CC2])  # pass callback to training
```

This will give the following output (reported here for just one epoch for brevity)

```
Train on 5000 samples, validate on 5000 samples
{}
Epoch 1/2
4864/5000 [============================>.] - ETA: 0s - loss: 0.0511 -
acc: 0.9879
Just finished epoch 0
{'val_loss': 0.2545496598124504, 'val_acc': 0.9244, 'loss':
0.05098680723309517, 'acc': 0.9878}
```

Now things are starting to get interesting. The logs dictionary contains a lot more information that we can access and use. In the dictionary, we have val_loss, val_acc, and acc. Let's customize the output a bit. Let's set verbose = 0 in the fit() call to suppress the standard output and generate our own.

Our new class will be

```
class CustomCallback3(keras.callbacks.Callback):
    def on_train_begin(self, logs={}):
```

```
        print (logs)
        return

    def on_epoch_end(self, epoch, logs={}):
        print ("Just finished epoch", epoch)
        print ('Loss evaluated on the validation dataset =',
        logs.get('val_loss'))
        print ('Accuracy reached is', logs.get('acc'))
        return
```

We can train our network with

```
CC3 = CustomCallback3()
model.fit(train_images, train_labels,  epochs = 2,
          validation_data = (test_images,test_labels),
          callbacks = [CC3], verbose = 0)  # pass callback to training
```

Doing this, we will get

```
{}
Just finished epoch 0
Loss evaluated on the validation dataset = 0.2546206972360611
```

The empty {} is simply the empty `logs` dictionary that `on_train_begin` received. Of course, you can simply print information every few epochs. For example, by modifying the `on_epoch_end()` function as

```
def on_epoch_end(self, epoch, logs={}):
        if (epoch % 10 == 0):
          print ("Just finished epoch", epoch)
          print ('Loss evaluated on the validation dataset =',
          logs.get('val_loss'))
          print ('Accuracy reached is', logs.get('acc'))
        return
```

We get the following output if we train the network for 30 epochs

```
{}
Just finished epoch 0
Loss evaluated on the validation dataset = 0.3692033936366439
```

```
Accuracy reached is 0.9932
Just finished epoch 10
Loss evaluated on the validation dataset = 0.3073081444747746
Accuracy reached is 1.0
Just finished epoch 20
Loss evaluated on the validation dataset = 0.31566708440929653
Accuracy reached is 0.9992
<tensorflow.python.keras.callbacks.History at 0x7f796083c4e0>
```

Now you should start to get an idea as to how you can perform several things during training. You can, for example, save accuracy values in lists to plot them later, or simply plot metrics to see how your training is going. The possibilities are almost endless. Callbacks are a great way to customize what happens during training.

Custom Training Loops

The easiest way to train a network with Keras is to use the `compile()/fit()` approach. It makes building and training the network very easy. But the downside of this approach is that you don't have much flexibility as to how the training is implemented. For example, suppose you want to train two networks in alternate fashion (as you learned in Chapter 11 about GANs). In this case, the standard `fit()` call is not enough anymore; you need to implement a custom training loop. Let's see how to do that.

Calculating Gradients

As you know, the `fit()` function will evaluate the gradients of the loss function and, by using the appropriate optimizer, use them to update the weights. The first step in implementing a custom training loop is to understand how to evaluate the gradients of a given function manually. Let's consider the function $y = x^2$. How can we calculate the gradient of it at $x_0 = 2$ with TensorFlow? By manually taking the derivative, we can immediately see that

$$\frac{d}{dx}x^2 = 2x$$

And therefore

$$\frac{d}{dx}x^2\Big|_{x=2} = 4$$

With Keras, we can do the same calculation this way

```
x = tf.constant(2.0)
with tf.GradientTape() as g:
  g.watch(x)
  y = x * x
dy_dx = g.gradient(y, x)
```

The `dy_dx` variable will be a `Tensor` that will have a single value of `4`

```
tf.Tensor(4.0, shape=(), dtype=float32)
```

TensorFlow operations are "recorded" in sequence, like on a tape (hence the name `GradientTape`), when they are executed within this context manager. TensorFlow checks all the called operations and saves all the gradients of the operations that are evaluated in that context. Note that everything that happens outside that context is ignored. In TensorFlow language, every operation and variable that is recorded is being "watched." Fortunately, when dealing with neural networks, all trainable variables (the weights and bias, typically) are automatically watched. But you can manually ask TensorFlow to watch other tensors by using the `watch()` call, as we have done in our example with the `g.watch(x)` code.

To understand what is going on in the background, you need to understand auto-differentiation. But intuitively you can think of the process in this way: when TensorFlow evaluates an operation, it will also save its gradient in memory. By using the `gradientTape`, you are simply asking TensorFlow to keep the evaluated gradients of specific operations in memory so that they can be used and combined properly to get the right result in the end.

Note that as soon as you call the `gradient()` function, all the resources held by the `GradientTape()` are released. If you wanted to calculate the second derivative, for example, you must do it differently than this example. You would need to use the `persistent=True` parameter in the creation of `GradientTape()`. For example, suppose you wanted the second derivative of the following function at $x_0 = 2$

$$y = x^3$$

The result is 12 since

$$\frac{d^2}{dx^2} x^3 = 6x.$$

With Keras, the code would look like this

```
x = tf.constant(2.0)
with tf.GradientTape(persistent=True) as g:
  g.watch(x)
  y = x * x * x
dy_dx = g.gradient(y, x)
dy_dx = g.gradient(y, x)
```

That would give the expected result

```
tf.Tensor(12.0, shape=(), dtype=float32)
```

Running the same code without the `persistent=True` parameter will produce an error message when calling the `gradient()` function a second time. There is another important point to note. Consider the following code, where I removed the `g.watch(x)` call.

```
x = tf.constant(2.0)
with tf.GradientTape() as g:
    y = x * x * x
dy_dx = g.gradient(y, x)
print(dy_dx)
```

The result of this code is `None`. No gradient can be evaluated since the variable `x` is not being "watched" by the `GradientTape()`. Now let's see how to implement a custom training loop with a neural network.

Custom Training Loop for a Neural Network

Now consider a very small FFNN with two layers, each having 64 neurons.

```
inputs = keras.Input(shape=(784,), name="digits")
x1 = layers.Dense(64, activation="relu")(inputs)
x2 = layers.Dense(64, activation="relu")(x1)
outputs = layers.Dense(10, name="predictions")(x2)
model = keras.Model(inputs=inputs, outputs=outputs)
```

As a second step, we need to specify an optimizer and a loss function (remember that we will not use the `compile()` function, so we need to use the Keras functions explicitly):

```
optimizer = keras.optimizers.Adam(learning_rate=1e-2)
loss_fn = keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

Another thing that we need to specify (by using the Keras functions) are the metrics we want to track, because we cannot specify the metrics in the `fit()` call.

```
train_acc_metric = keras.metrics.SparseCategoricalAccuracy()
val_acc_metric = keras.metrics.SparseCategoricalAccuracy()
```

At this point, we have all the ingredients we need. The loop can now be implemented easily

```
epochs = 200
for epoch in range(epochs):
    with tf.GradientTape() as tape:

        # Run the forward pass of the layer.

        logits = model(x_train, training=True)  # Logits for this minibatch

        # Compute the loss funtion
        loss_value = loss_fn(y_train, logits)

    grads = tape.gradient(loss_value, model.trainable_weights)
    optimizer.apply_gradients(zip(grads, model.trainable_weights))

    # Update training metric.
    train_acc_metric.update_state(y_train, logits)

    # Display metrics at the end of each epoch.
    train_acc = train_acc_metric.result()
```

```
if epoch % 20 == 0:
    print(
        "Training loss (for one batch) at step %d: %.4f"
        % (epoch, float(loss_value))
    )
    print("Training acc over epoch: %.4f" % (float(train_acc),))
```

In the `GradientTape()` context, we need the following steps:

- The forward pass: Easily done with `model(x_train, training=True)`
- The loss function: `loss_value = loss_fn(y_train, logits)`
- Calculate the gradients and apply them to update the weights: `grads = tape.gradient(loss_value, model.trainable_weights)`
- `optimizer.apply_gradients(zip(grads, model.trainable_weights))`

The `tape.gradient()` calculates the gradient of the loss function (that is being watched in the `GradientTape` context), and `apply_gradients()` applies the gradients with the optimizers to update the network weights.

After that, we need to keep track of the metrics. The `train_acc_metric.update_state(y_train, logits)` call updates the metrics we defined, and then the `train_acc = train_acc_metric.result()` call saves its value in a variable (`train_acc`) that we can display during training.

This very basic loop shows how you can build your own custom training loops. Of course, there is much more you can do and, as usual, the best place to get all the information is the official Keras documentation.[2] Note that there at least two important things that we have not discussed:

- How to train with mini-batches (the training loop we discussed uses all the data). To do that, we could use

  ```
  train_dataset = tf.data.Dataset.from_tensor_slices
  ((x_train, y_train)
  train_dataset = train_dataset.shuffle(buffer_size=1024).
  batch(batch_size)
  ```

  and then add a loop over the batches with this

  ```
  for step, (x_batch_train, y_batch_train) in enumerate
  (train_dataset):
  ```

- How to speed up the training by using `@tf.function`. This is a more advanced topic that would require a long discussion and goes beyond the scope of this book.

Remember that the goal of this book is to teach you how neural networks work and how easy it is to implement them, not to make you a Keras expert. The goal of this appendix is to give just enough information that you can follow the book. To better understand Keras customization, your best bet is to use the official documentation and work through the examples.

[2] A good place to start is https://www.tensorflow.org/guide/keras/writing_a_training_loop_from_scratch.

Index

A

B

U. Michelucci, *Applied Deep Learning with TensorFlow 2*, https://doi.org/10.1007/978-1-4842-8020-1

C

D

E

F

G

H

I, J

K

L

M

N

O

P, Q

R

S

T

U, V, W, X, Y

Z