Basic Introduction to Keras

Keras is an API designed to make developing neural network easy for humans. It is also the API that we have used 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 will need to understand the code that we described in the book and point you to resources where you can find more. If you want to learn everything about Keras, the most efficient way is to buy and study the book *Deep Learning with Python* by François Chollet and of course (and even better) to read the official documentation at <https://keras.io>.

In a few words, citing F. Chollet[[1]](#footnote-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 please check Appendix 2 where we will cover those topics, although briefly.

# Some History

Keras was first developed by F. Chollet, and its first version was made available on March 27th, 2015. Up and including version 2.3, Keras needed a backend, in other words a system that performs the operations at low level that are needed by your code. At the very beginning Keras was not working with TensorFlow (the first supported backend was Theano). The tf.keras package was introduced in TensorFlow 1.10.0. Note that the two imports

import keras

And

from tensorflow import keras

Are two very different things. After Keras 2.3.0 F. Chollet has declared that this release will be in sync with tf.keras and that in the future practitioners should use tf.keras and **not** keras anymore. After this release, keras will not support multiple backends anymore.

**Note** in your code you should always use from tensorflow import keras.

# Sequential Model

A sequential model is simply a plain stack of layers, where each has one input and one output tensor. The easiest way of creating a sequential model 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 the code above it is assumed you have imported

from tensorflow.keras import layers

There is an alternative way of creating 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]](#footnote-2):

* Your model has multiple inputs or multiple outputs
* Any of your layers has multiple inputs or multiple outputs
* You need to do layer sharing
* You want non-linear topology (e.g. a residual connection, a multi-branch model)

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

# Keras Layers

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

* Dense – a layer as the ones we have seen in the FFNN discussion
* Conv1D, Conv2D and Conv3D – convolutional layers in multiple dimensions
* MaxPooling1D, MaxPooling2D and MaxPooling3D – Maxpooling layers
* AveragePooling1D, AveragePooling2D and AveragePooling3D – Average Pooling layers
* LSTM layers
* Regularization layers as Dropout

And many more. Remember that any operations that takes a tensor as input and gives a tensor as output is a layer in Keras language. For example flattening a 2D image into a one dimensional 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 operations that takes a tensor as input and gives a tensor as output is a layer in Keras language.

In Appendix 2, we will briefly discuss how to develop your own layers and what are the steps necessary. Note that you can also easily do the following things with layers:

* Retrieve the gradients (see next Appendix)
* Retrieve the weights (see next Appendix)
* Add regularization losses (as we have discussed in the main part of the book)
* You can set the weights to values of your choices (see next Appendix)
* You can use initializers for the weights (for example He, Glorot, etc.)

## Setting the Activation Function

To set the activation function in layers you can set the activation property. For example the code

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

creates a layer with 2 neurons with the ReLU activation function. Note that if you don’t specity any activation none is used (or in other words the identity function is used as activation function). As usual Keras offers many activation functions[[4]](#footnote-4): relu, sigmoid, softmax, softplus, softsign, tanh, selu, elu and exponential function.

# 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 (or both). The idea behind it (if you don’t understand it simply ignore it) is that neural networks are normally a directed acyclic graphs, so with the functional APIs you can build graphs of layers (therefore you can build non-linear architectures). For example the pevious network would be built as

inputs = keras.Input(shape=(...,))

x = layers.Dense(2, activation="relu")(inputs)

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

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 is like drawing a connection between two layers. This model can be graphically plotted with Keras[[5]](#footnote-5) and it looks like Figure 1-1.

Diagram

Description automatically generated

Figure 1-1: 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 1-2.

Diagram

Description automatically generated

Figure 1-2: a graphical representation of a network with multiple inputs built with the Keras Function API.

You can get really creative with the architectures you can create. You will find lots of examples on 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 optimizer and the MSE as loss function you would use

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

# Putting all together and Training

The easiest way of training a model in Keras needs three steps:

1. Create the network by specifying the architecture (number of layers, number of neurons, type of layers, activation functions, etc.). You would do this in our examples with, for example, the keras.Sequential().
2. Compile the model with the compile() method. This step specify 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 number of epochs, batch size and many more parameters.

Here is a minimal example

model = keras.Sequential(

[

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

layers.Dense(3, activation="relu"),

layers.Dense(1),

]

)

This specify the network architecture. After that we compile the model with the compile() method.

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

where we have specified the optimizer Adam and the MSE loss. After that we can train the model

model.fit(x, y, batch\_size=32, epochs=10)

where x would be the inputs, y the labels, the batch\_size has been specified as 32 and we want to train for 10 epochs.

**Note** Creating and training a model in Keras need, in the easiest approach, three steps: 1) create the architecture, 2) compile the model with the compile(), 3) train the model with the fit() method.

The fit() method accepts lots of parameters. You can specify for example

* How much output you want by specifying the verbose parameter (0 for no output, 1 for a progress bar, 2 for one line for each epoch)
* You can take actions at different point during the training by specifying which callbacks functions the fit() method should use. For more information on callbacks check the next sections
* You can specify a validation dataset by simply giving a validation\_split parameter (that is the fraction of the data that you want to use as validation dataset). The fit() method will give you the metrics also for this dataset.

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

# Model 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]](#footnote-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)

# Callback Functions

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

It is instructive to understand a bit better what are Keras callback functions since they are used quite often while developing models. From the official documentation[[7]](#footnote-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 be then called at each stage of the training. Their use is rather easy. For the fit() function you would use them as

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 as

* ModelCheckPoint: save weights and model at specific frequencies
* LearningRateScheduler: change the learning rate according to some schedule
* TerminateOnNaN: stop the training if NaN appears (so you don’t waste time or computing resources)
* And many more.

As usual you can find many information on the official documentation at <https://keras.io/api/callbacks/>. In the next section I will discuss how to develop your own custom callback class, since this is one of the best way to check and control the training at various stages.

# Save and load models

It is often useful to save a model on disk, to be able to continue the training at a later stage, or to reuse a previously trained model. To show how you can do it, let's consider again the MNIST dataset for the sake of giving a concrete example[[8]](#footnote-8). The entire code is available in a dedicated notebook in the book github repository under the folder "chapter 2".

You will need the following imports

import os

import tensorflow as tf

from tensorflow import keras

And again, let's load the MNIST dataset and take the first 5000 observations.

(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

and then let's build a simple Keras model with a Dense layer with 512 neurons, a bit of dropout and the classical 10 neuron output layer for classification (remember the MNIST dataset has 10 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 simply need to do is first to define where we want to save the model on the disk. And we can do it (for example) in this way

checkpoint\_path = "training/cp.ckpt"

checkpoint\_dir = os.path.dirname(checkpoint\_path)

After that we need to define a callback (remember what we did in the last section) that will save the weights

cp\_callback = tf.keras.callbacks.ModelCheckpoint(checkpoint\_path,

save\_weights\_only=True,

verbose=1)

Note that now we don't need to define a Class as we have done in the previous section, since ModelCheckpoint inherit from the class Callback.

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 run a !ls command, you should see at least three files:

* cp.ckpt.data-00000-of-00001 🡪 contains the weights (in case the number of weights is big, you will get many files like this one)
* cp.ckpt.index 🡪 this file contains information on which weights are in which file
* checkpoint 🡪 this text file contains information on the checkpoint itself

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

you 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))

you should get the result

5000/5000 [==============================] - 0s 50us/step

Restored model, accuracy: 92.06%

That makes again sense, since the new model is now using the weights on the old trained model. Keep in mind that to load pre-trained weights in a new model, the latter needs to have the exact same architecture than the one you have used when saving the weights.

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

As we will see again and again, the basic idea is to use callbacks and define a custom one that will save our weights. Of course, we can customize our callback function. For example, if want to save the weights every 100 epochs and each time with a different filename, so that we could decide to restore a specific check point we 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)

and we 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 keys in logs (passed in on\_epoch\_end that we have seen in the previous section)[[9]](#footnote-9). You can check the original code for tf.keras.callbacks.ModelCheckpoint and you will find that the formatting is done with in the method on\_epoch\_end(self, epoch, logs)

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

you can define your filename with information with both the epoch number and 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 1 epoch. So for example your directory content may look like this one

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 just the latest checkpoint, without bothering to search its filename. This can be done easily with the following code

latest = tf.train.latest\_checkpoint('training')  
model.load\_weights(latest)

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

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

## Save your weights manually

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

model.save\_weights('./checkpoints/my\_checkpoint')

this command will generate three files, all starting with the string you have given as a name, in this case my\_checkpoint. Running the code above will generate the three files we have already described above:

checkpoint

my\_checkpoint.data-00000-of-00001

my\_checkpoint.index

Reloading the weights in a new model is as simple as

model.load\_weights('./checkpoints/my\_checkpoint')

keep in mind that to be able to reload saved weights in a new model, the latter must have the same architecture of the new one. It must be exactly the same.

## Saving the entire model

Keras gives you also the possibility of saving the entire model on disk: weights, the architecture and optimizer. In this way you can recreate the same model by simply moving some files. For example, we could use the following code

model.save('my\_model.h5')

this will save in one file, "my\_model.h5", the entire model. You can simply move the file to a different computer and recreate the same trained model with

new\_model = keras.models.load\_model('my\_model.h5')

and note that this model will have the same trained weights of your original model, so is 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.

1. <https://keras.io/getting_started/intro_to_keras_for_researchers/> [↑](#footnote-ref-1)
2. As taken from the official documentation at <https://keras.io/guides/sequential_model/>. [↑](#footnote-ref-2)
3. If you want to see the complete list you can consult the official documentation at <https://keras.io/api/layers/>. [↑](#footnote-ref-3)
4. As of November 2021. [↑](#footnote-ref-4)
5. To plot a model you can use the useful call keras.utils.plot\_model(model, "..."). Swap the three dots with a file name that you want to use. [↑](#footnote-ref-5)
6. This is relevant when, for example, dealing with dropout that has a different behavior during training or during testing. [↑](#footnote-ref-6)
7. https://keras.io/callbacks/ [↑](#footnote-ref-7)
8. The example has been inspired by the official Keras documentation https://www.tensorflow.org/tutorials/keras/save\_and\_restore\_models [↑](#footnote-ref-8)
9. Check the official documentation at https://goo.gl/SnKgyQ [↑](#footnote-ref-9)