TensorFlow: advanced topics

The TensorFlow library has come a long way from its first appearance. Especially in the last year, many more features have become available that can make the life of researchers a lot easier. Things like Eager execution and Keras allows scientists to test and experiment much faster and debug models in ways that were not possible before. It is essential for any researcher to know those methods and know when it makes sense to use them. In this chapter we will look at few of them: eager execution, GPU acceleration, Keras, how to freeze parts of a network and train only specific part (used very often especially in transfer learning and image recognition), and finally how to save and restore models already trained. Those technical skills will be very useful, not only to study this book, but in real life research projects.

The goal of this chapter is not to teach you how to use Keras from the ground up, or to teach you all the intricacies of the methods, but to show you some advanced techniques to solve some specific problems. Consider the different sections as hints. Remember that is always a good idea to study the official documentation, since methods and functions change very often. In this chapter I will avoid copying the official documentation, and instead give you few advanced examples of techniques that are very useful and are used very often. To go deeper (pun intended) you should study the official TensorFlow documentation that can be found here https://www.tensorflow.org/.

To study and understand advanced topics a good basis in Tensorflow and Keras is required. A very good resource to get up to speed with Keras is the book Learn Keras for Deep Neural Networks - A Fast-Track Approach to Modern Deep Learning with Python from Jojo John Moolayil (https://goo.gl/mW4Ubg). If you don't have much experience, I suggest you get this book, and study it before starting this one.

# Tensorflow eager execution

TensorFlow's eager execution is an imperative programming environment[[1]](#footnote-1). That, loosely formulated, means that the commands are evaluated immediately. That also means that no graph is built. Operations return immediately concrete values instead of having first to construct a computational graph, open a session and then run it. This makes very easy to start with TensorFlow, since it resembles classical Python programming. Eager execution provides the following advantages

* Easier debugging: you can debug your models with classical Python debugging tools for immediate checks
* Intuitive interface: you can structure your code naturally, as you would do in a classical Python program
* GPU acceleration is supported

To be able to use this execution mode you will need the latest version of TensorFlow. If you have not yet installed it, check Chapter 1 to see how to do it.

## Enabling eager execution

To enable eager execution, you can use the following code

import tensorflow as tf

tf.enable\_eager\_execution()

Remember that you need to do that right at the beginning after the imports and before any other command. Otherwise you will get an error message. If that is the case, you can simply restart the kernel of the notebook.

For example, you can easily add two tensors

print(tf.add(1, 2))

and get immediately the result

tf.Tensor(3, shape=(), dtype=int32)

If you don't enable eager execution and try again the print command given above, you will get this result

Tensor("Add:0", shape=(), dtype=int32)

since TensorFlow does not evaluate yet the node. You would need the following code to get the result

sess = tf.Session()

print(sess.run(tf.add(1,2)))

sess.close()

the result will be, of course, 3. This second version of the code creates a graph, then open a session and then evaluates it. With eager you get the result immediately. You can easily check if you have enabled eager execution with

tf.executing\_eagerly()

that should return True or False depending if you have enabled it or not.

## Polynomial fitting with eager execution

Let's check how eager execution is working in a practical example[[2]](#footnote-2).

Keep in mind you need the following imports

import tensorflow as tf

import numpy as np

import matplotlib.pyplot as plt

import tensorflow.contrib.eager as tfe

tf.enable\_eager\_execution()

Let's generate some fake data for the function

with the code

x = np.arange(0, 5, 0.1)

y = x\*\*3 - 4\*x\*\*2 - 2\*x + 2

y\_noise = y + np.random.normal(0, 1.5, size=(len(x),))

with the code we have created two NumPy arrays: y that contains the function evaluated over the array x, and y\_noise that contains y with added some noise. You can see how the data looks like in Figure 1.

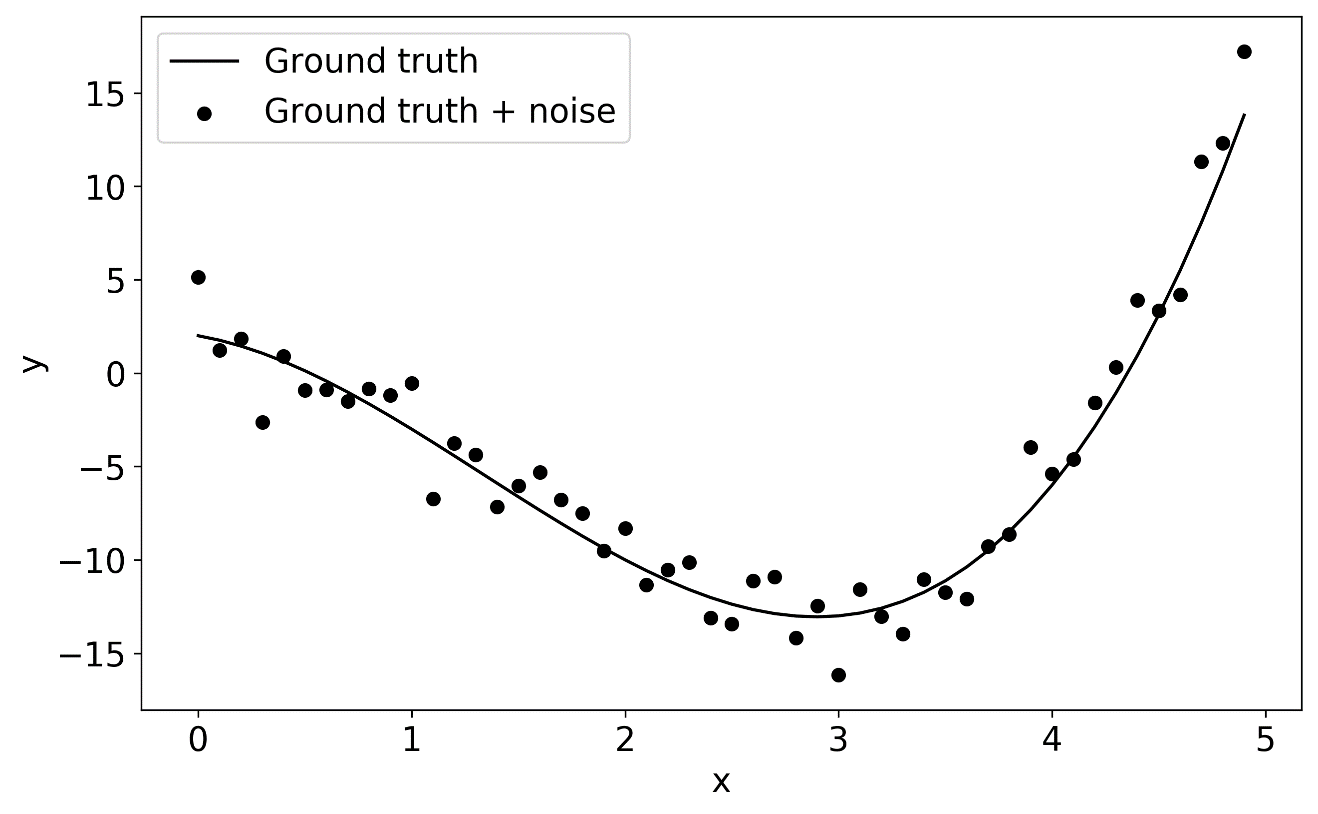


Figure 1: the plot shows the two numpy arrays y (Ground Truth) and y\_noise (Ground truth + noise).

Now we need to define a model that we want to fit and define our loss function (the one we want to minimize with TensorFlow). Remember we are facing a regression problem so we will use the Mean Squared error (MSE) as our loss function. The functions we need are

class Model(object):

def \_\_init\_\_(self):

self.w = tfe.Variable(tf.random\_normal([4])) # The 4 parameters

def f(self, x):

return self.w[0] \* x \*\* 3 + self.w[1] \* x \*\* 2 + self.w[2] \* x + self.w[3]

and

def loss(model, x, y):

err = model.f(x) - y

return tf.reduce\_mean(tf.square(err))

Now is easy to minimize the loss function. First let's define some variables we will need

model = Model()

grad = tfe.implicit\_gradients(loss)

optimizer = tf.train.AdamOptimizer()

and then let's, with a for loop, minimize the loss function

iters = 20000

for i in range(iters):

optimizer.apply\_gradients(grad(model, x, y))

if i % 1000 == 0:

print("Iteration {}, loss: {}".format(i+1, loss(model, x, y).numpy()))

This code will produce some outputs showing you the value for the loss function each 1000 iterations. Note that we are feeding all the data in one batch to the optimizer (since we have only 50 datapoints we don't really need to use mini-batches).

You should see several output lines that are like this one

Iteration 20000, loss: 0.004939439240843058

The loss function plot vs. the number of the iterations can be seen in Fig.2 and is decreasing constantly as expected.

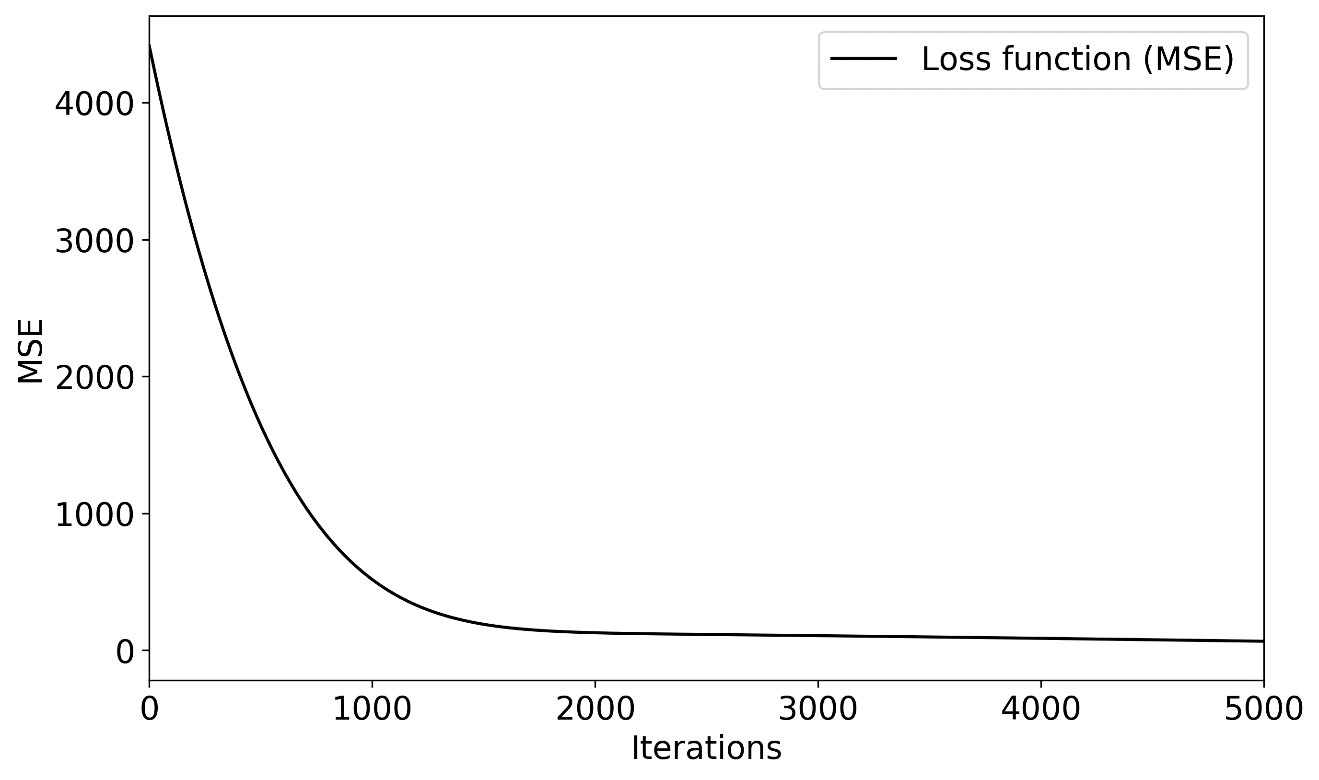


Figure 2: the loss function (MSE) vs. the iteration number is decreasing as expected. That shows clearly that the optimizer is doing a good job in finding the best weights to minimize the loss function.

In Fig. 3 you can see what is the function that the optimizer was able to find, by minimizing the weights.

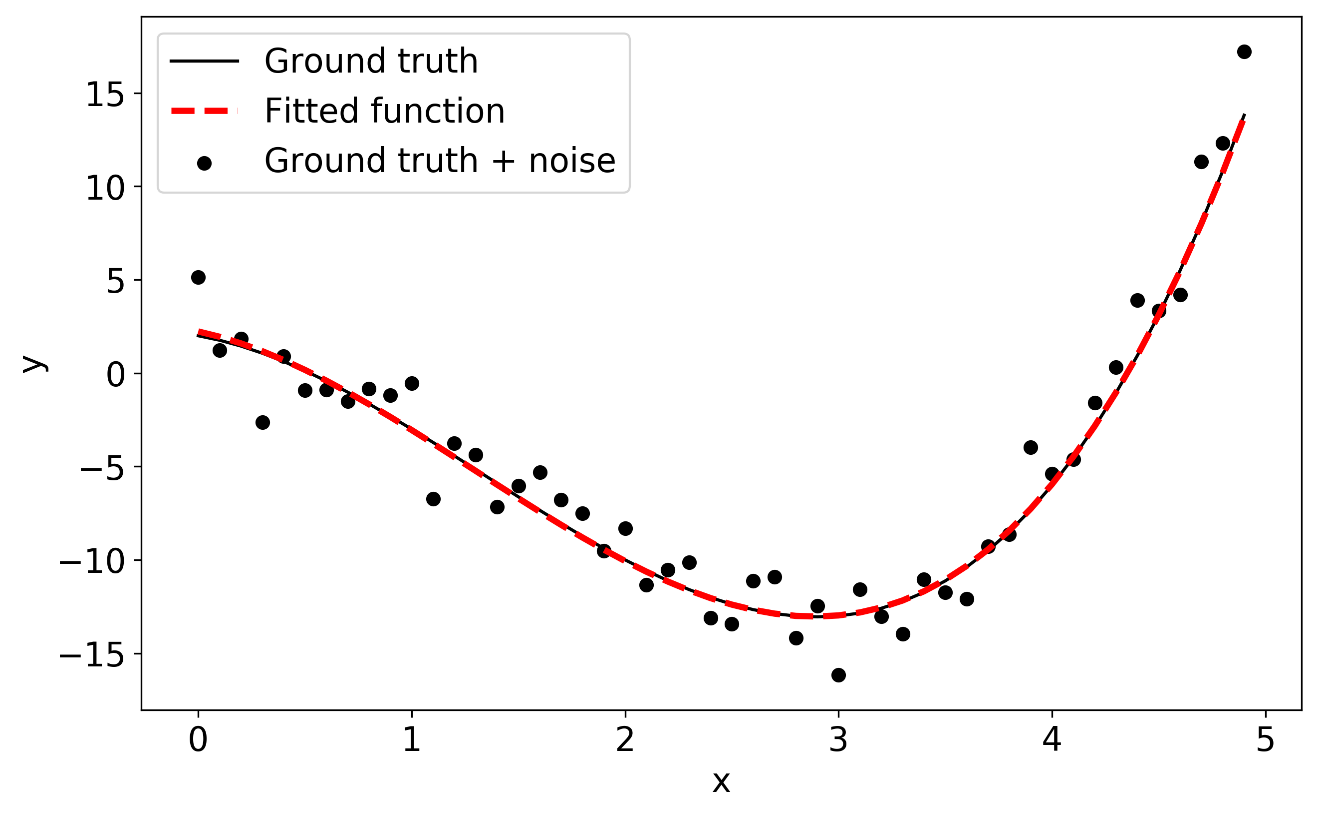


Figure 3: the red dashed line is the function obtained by minimizing the loss function with the Adam optimizer. The method worked perfectly and found the right function efficiently.

What you should note is that we have not created first a computational graph and then evaluated it in a session. We simply used the commands as we would with any Python code. For example, in the code

for i in range(iters):

optimizer.apply\_gradients(grad(model, x, y))

we simply call a tensorflow operation in a loop without the need of a session. With eager execution is easy to start using TensorFlow operations quickly without too much overhead.

## MNIST classification with eager execution

To give you another example on how you could build a model with eager execution, let's build a classifier for the famous MNIST dataset. This is a dataset containing 60000 images of handwritten digits (from 0 to 9), each with a dimension of 28x28 in gray levels (each pixel has a value ranging from 0 to 255). If you have never seen the MNIST dataset, I suggest you check the original website <https://goo.gl/yF0yH> where you will find all information. We will implement the following steps

* Load the dataset
* Normalize the features and one-hot encode the labels
* Convert the data in a tf.data.Dataset object
* Build a Keras model with two layers, each with 1024 neurons
* Define the optimizer and the loss function
* Minimize the loss function using the gradients and the optimizer directly

Let's start.

While following the code, note how we implement each piece as we would do with plain NumPy, meaning without the need of creating a graph or opening a TensorFlow session.

So first let's load the MNIST dataset using the keras.datasets.mnist package, reshape it and one-hot encode the labels.

num\_classes = 10

mnist = tf.keras.datasets.mnist

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

image\_vector\_size = 28\*28

x\_train = x\_train.reshape(x\_train.shape[0], image\_vector\_size)

x\_test = x\_test.reshape(x\_test.shape[0], image\_vector\_size)

y\_train = keras.utils.to\_categorical(y\_train, num\_classes)

y\_test = keras.utils.to\_categorical(y\_test, num\_classes)

Then let's convert the arrays in a tf.data.Dataset object. In case you don't understand what this is, don't worry, we will look at it more later in this chapter, for the moment it suffices to know that is a convenient way of using mini-batches while you train your network.

dataset = tf.data.Dataset.from\_tensor\_slices(

(tf.cast(x\_train/255.0, tf.float32),

tf.cast(y\_train,tf.int64)))

dataset = dataset.shuffle(60000).batch(64)

and then let's build the model with a feed-forward neural network with two layers, each with 1024 neurons:

mnist\_model = tf.keras.Sequential([

tf.keras.layers.Dense(1024, input\_shape=(784,)),

tf.keras.layers.Dense(1024),

tf.keras.layers.Dense(10)

])

Up to now we have not done anything particularly new, so you should be able to follow what we did quite easily. The next step is to define the optimizer (we will use Adam) and the list that will contain the loss function history

optimizer = tf.train.AdamOptimizer()

loss\_history = []

at this point we can start with the actual training. We will have two nested loops, the first is for the epochs, the second for the batches.

for i in range(10): # Epochs

print ("\nEpoch:", i)

for (batch, (images, labels)) in enumerate(dataset.take(60000)):

if batch % 100 == 0:

print('.', end='')

with tf.GradientTape() as tape:

logits = mnist\_model(images, training=True) # Prediction of the model

loss\_value = tf.losses.sparse\_softmax\_cross\_entropy(tf.argmax(labels, axis = 1), logits)

loss\_history.append(loss\_value.numpy())

grads = tape.gradient(loss\_value, mnist\_model.variables) # Evaluation of gradients

optimizer.apply\_gradients(zip(grads, mnist\_model.variables),

global\_step=tf.train.get\_or\_create\_global\_step())

The part of the code that is probably new to you, is the one that contains the two lines

grads = tape.gradient(loss\_value, mnist\_model.variables)

optimizer.apply\_gradients(zip(grads, mnist\_model.variables),

global\_step=tf.train.get\_or\_create\_global\_step())

The first one calculate the gradients of the loss\_value TensorFlow operation with respect to the mnist\_model.variables (the weights basically), and the second uses the gradients to let the optimizer update the weights. To understand how Keras evaluates gradients automatically I suggest you check the official documentation here https://goo.gl/s9Uqjc. Running the code will finally train the network. As the training progress you should see an output like this for each epoch

Epoch: 0

..........

Now to check the accuracy you can simply run the following two lines (that should be self-explanatory)

probs = tf.nn.softmax(mnist\_model(x\_train))

print(tf.reduce\_mean(tf.cast(tf.equal(tf.argmax(probs, axis=1), tf.argmax(y\_train, axis = 1)), tf.float32)))

this will give you as a result a tensor, that will contain the accuracy reached by the model

tf.Tensor(0.8980333, shape=(), dtype=float32)

in this example we reached 89.8% accuracy, a relatively good results for such a simple network. Of course, you could try to train the model for more epochs or try to change the learning rate for example. In case you are wondering where we defined the learning rate, we did not. When we define the optimizer as tf.train.AdamOptimizer, TensorFlow will use, if not specified differently, the standard value of . You can check this looking at the documentation <https://goo.gl/pU7yrB>.

We could check one prediction easily. Let's get one image from our dataset

image = x\_train[4:5,:]

label = y\_train[4]

if we plot the image, we can see is a nine (see Fig. 4).

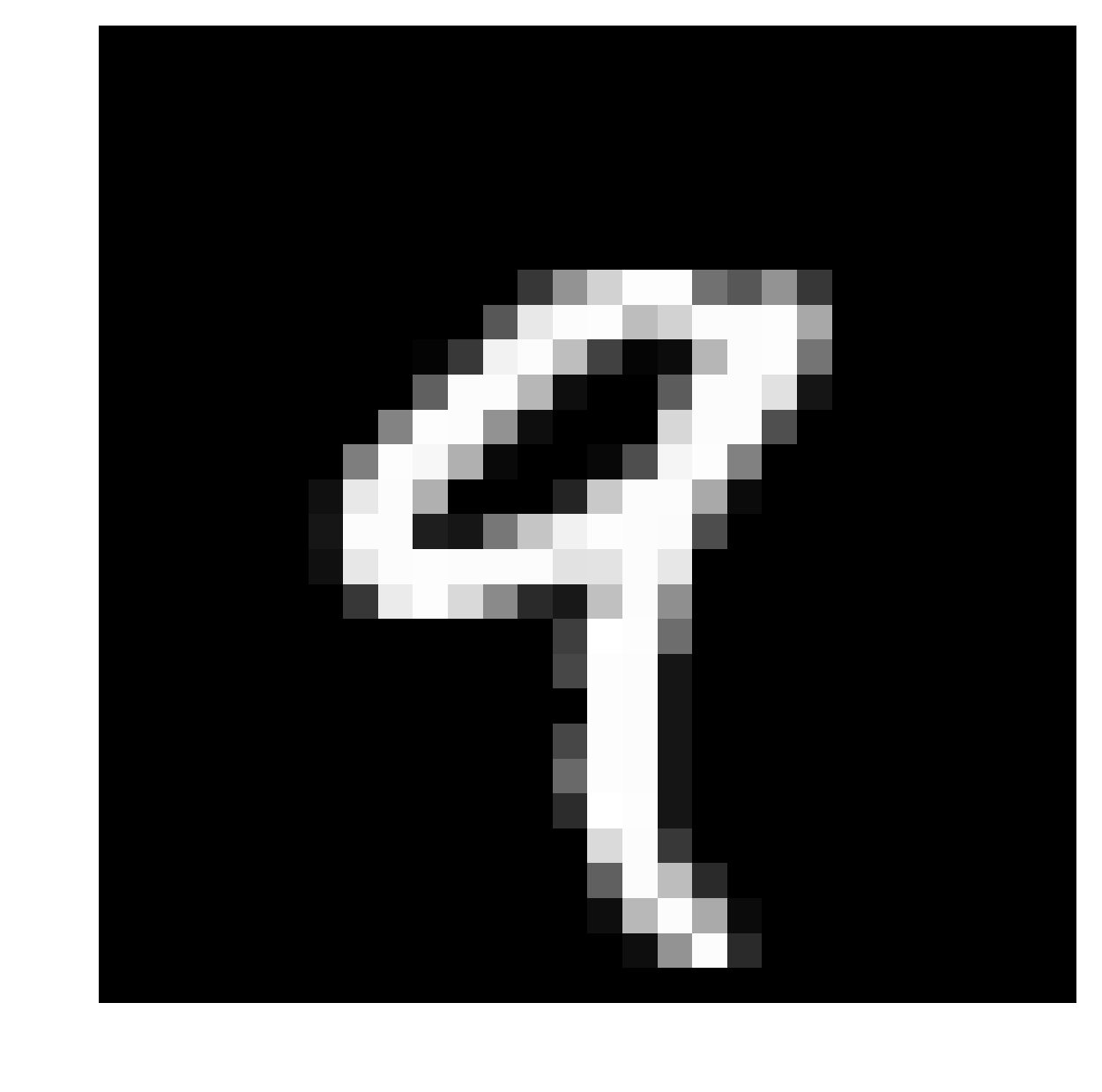


Figure 4: one image from the MNIST dataset. This happens to be a 9.

And we can check easily what the model is predicting

print(tf.argmax(tf.nn.softmax(mnist\_model(image)), axis = 1))

that returns

tf.Tensor([9], shape=(1,), dtype=int64)

as we expected. You should note how we have written the code. We have not created a graph, but we simply used functions and operations as we would have done with numpy. No need to think in graphs and sessions. This is how eager execution is working.

# TensorFlow and Numpy compatibility

TensorFlow make switching to and from NumPy arrays very easy:

* TensorFlow convert NumPy arrays to Tensors
* NumPy convert Tensors to NumPy arrays

Converting a tensor to a numpy array is very easy, and is enough to invoke the .numpy() method. This operation is quite fast and cheap since the numpy array and the tensor should share the memory, so no shifting around in memory is happening. Now this is not possible if you are using GPU hardware acceleration since numpy arrays cannot be stored in GPU memory and tensors can. Converting will involve copying data from the GPU memory to the CPU memory. Simply something to keep in mind.

Typically, TensorFlow Tensors and numpy arrays should share the same memory. Converting one into another is quite a cheap operation. But if you use GPU accelerations, tensors may be held in the GPU memory, and numpy arrays cannot, so copying of data will be required. This may be more expensive in term of running time.

# Hardware acceleration

## Checking availability of GPU

It is worth to show briefly how to use GPUs and what difference it may make, just to give you a feeling for it. If you have never seen it is quite impressive. The easiest way of testing GPU acceleration is to use Google colab. Create a new notebook in Google Colab, activate GPU[[3]](#footnote-3) acceleration and import TensorFlow as usual

import tensorflow as tf

then we need to test if we have a GPU at our disposal. This can be easily done with the code

print(tf.test.is\_gpu\_available())

this will return True or False depending if a GPU is available or not respectively. In a slightly more sophisticated way, it can be done in this way

device\_name = tf.test.gpu\_device\_name()

if device\_name != '/device:GPU:0':

raise SystemError('GPU device not found.')

print('Found GPU at: {}'.format(device\_name))

If you run the code, you may get this error

SystemErrorTraceback (most recent call last)  
<ipython-input-1-d1680108c58e> in <module>()  
 2 device\_name = tf.test.gpu\_device\_name()  
 3 if device\_name != '/device:GPU:0':  
----> [4](https://localhost:8080/) raise SystemError('GPU device not found')  
 5 print('Found GPU at: {}'.format(device\_name))  
SystemError: GPU device not found

The reason is that you may have not yet configured the notebook (if you are in Google Colab) to use a GPU or if you are working on a laptop or desktop you may have not installed the right TensorFlow version or you may not have a compatible GPU available.

To enable the GPU hardware acceleration in Google Colab go into the menu "Edit"-"Notebook settings". You are then presented with a window where you can setup the Hardware Accelerator. By default, it is set to None. If you set it to GPU and run the code above again you should now get this message

Found GPU at: /device:GPU:0

## Device names

Note how the device name, in our case /device:GPU:0 encodes many information. This name ends with GPU:<NUMBER>, where <NUMBER> is an integer that can be as big as the number of GPU you have at your disposal. You can get a list of all the devices you have at your disposal with the code

local\_device\_protos = device\_lib.list\_local\_devices()

print(local\_device\_protos)

and you will get a list of all the devices. Each list entry will be like this one (this example refers to a GPU device)

name: "/device:XLA\_GPU:0"

device\_type: "XLA\_GPU"

memory\_limit: 17179869184

locality {

}

incarnation: 16797530695469281809

physical\_device\_desc: "device: XLA\_GPU device"

with a function like this one

def get\_available\_gpus():

local\_device\_protos = device\_lib.list\_local\_devices()

return [x.name for x in local\_device\_protos if x.device\_type.endswith('GPU')]

you will get a much easier to read result like this one[[4]](#footnote-4)

['/device:XLA\_GPU:0', '/device:GPU:0']

## Explicit device placement

It is very easy to place an operation on a specific device. That can be achieved using the tf.device context. For example, to place an operation on a CPU you can use the following code

with tf.device("/cpu:0"):  
 # SOME OPERATION

or to place an operation on a GPU you can use the code

with tf.device('/gpu:0'):

# SOME OPERATION

Unless explicitly declared, TensorFlow automatically decides on which device each operation must be run. Don't assume that if you don't specify the device explicitly that your code will run on a CPU.

## GPU acceleration demonstration: matrix multiplication

It is interesting to see what effect hardware acceleration may have. To learn more about using GPUs it is instructive to read the official documentation that can be found here https://www.TensorFlow.org/guide/using\_gpu.

Start with the following code[[5]](#footnote-5)

config = tf.ConfigProto()

config.gpu\_options.allow\_growth = True

sess = tf.Session(config=config)

The second line is there since TensorFlow starts to allocate little GPU memory. As the session is started and the processes run, more GPU memory is then allocated as needed. Then a session is created. Let's try now to multiply two matrices of dimensions 10000x10000 filled with random values and let's see if using a GPU makes any difference. The following code will run the multiplication on a GPU

%%time

with tf.device('/gpu:0'):

tensor1 = tf.random\_normal((10000, 10000))

tensor2 = tf.random\_normal((10000, 10000))

prod = tf.linalg.matmul(tensor1, tensor2)

prod\_sum = tf.reduce\_sum(prod)

sess.run(prod\_sum)

And the following on one CPU

%%time

with tf.device('/cpu:0'):

tensor1 = tf.random\_normal((10000, 10000))

tensor2 = tf.random\_normal((10000, 10000))

prod = tf.linalg.matmul(tensor1, tensor2)

prod\_sum = tf.reduce\_sum(prod)

sess.run(prod\_sum)

When I ran the code, I got 1.86 sec total time on a GPU and 1min4sec on a CPU: a factor 32 faster. You can image then when doing such calculations over and over, as is often the case in deep learning, you can get quite a performance boost in your evaluations. Using TPUs is slightly more complicated and goes beyond the scope of this book, so we will skip it.

Note that using a GPU is not always giving you a performance boost. When the tensors involved are small you will not see a huge difference while using a GPU instead of a CPU. The real difference will become evident when the dimensions of the tensors start to grow.

If you try to run the same code above but for smaller tensors, for example 100x100 you will not see any difference at all between using a GPU and a CPU. The tensors are small enough that a CPU will get the result as fast as a GPU. For two 100x100 matrices both GPU and CPU gives a result in roughly 20ms. Typically, practitioners let CPUs do all the preprocessing (for example normalization, loading of data, etc.) and then let GPUs perform all the big tensor operations during training.

Typically, you should evaluate only expensive tensor operations (like matrix multiplications or convolution) on GPUs, and do all preprocessing (like data loading, cleaning, etc.) on a CPU.

We will see later in the book (where applicable) how to do that. But don't be afraid, you will be able to use the code and follow the examples without a GPU at your disposal.

## Effect of GPU acceleration on the MNIST example

It is instructive to see the effect of hardware acceleration on the MNIST example we discussed above. To run the training of the model completely on the CPU we need to force TensorFlow to do it, since otherwise it will try to place expensive operations on a GPU when available. To do that you can use the code

with tf.device('/cpu:0'):

for i in range(10): # Loop for the Epochs

print ("\nEpoch:", i)

for (batch, (images, labels)) in enumerate(dataset.take(60000)): # Loop for the mini-batches

if batch % 100 == 0:

print('.', end='')

with tf.GradientTape() as tape:

logits = mnist\_model(images, training=True)

loss\_value = tf.losses.sparse\_softmax\_cross\_entropy(tf.argmax(labels, axis = 1), logits)

loss\_history.append(loss\_value.numpy())

grads = tape.gradient(loss\_value, mnist\_model.variables)

optimizer.apply\_gradients(zip(grads, mnist\_model.variables),

global\_step=tf.train.get\_or\_create\_global\_step())

this code, on google colab, runs in roughly 8 minutes and 41 seconds. If we put all possible operations on a GPU, using the code

for i in range(10): # Loop for the Epochs

print ("\nEpoch:", i)

for (batch, (images, labels)) in enumerate(dataset.take(60000)): # Loop for the mini-batches

if batch % 100 == 0:

print('.', end='')

labels = tf.cast(labels, dtype = tf.int64)

with tf.GradientTape() as tape:

with tf.device('/gpu:0'):

logits = mnist\_model(images, training=True)

with tf.device('/cpu:0'):

tgmax = tf.argmax(labels, axis = 1, output\_type=tf.int64)

with tf.device('/gpu:0'):

loss\_value = tf.losses.sparse\_softmax\_cross\_entropy(tgmax, logits)

loss\_history.append(loss\_value.numpy())

grads = tape.gradient(loss\_value, mnist\_model.variables)

optimizer.apply\_gradients(zip(grads, mnist\_model.variables),

global\_step=tf.train.get\_or\_create\_global\_step())

will run in 1 minute and 24 seconds. The reason why the tf.argmax() has been placed on a CPU, is that at the time of writing the GPU implementation of tf.argmax has a bug and does not work as intended.

You can clearly see what dramatic effect GPU acceleration has, even on a simple network as the one we have used.

# Training only specific layers

You should now know that Keras works with layers. When you define one, let's say a Dense layer

layer1 = Dense(32)

You can pass a trainable argument (that is boolean) to a layer constructor. This will stop the optimizer to update its weights

layer1 = dense(32, trainable = False)

but this would not be very useful. What is needed is the possibility of changing this property after instantiation. This is easy to do. For example, you can use the following code

layer = Dense(32)

# something useful happens here

layer.trainable = False

For a change of the trainable property to take effect, you will need to call the compile() method on your model. Otherwise the change will not have any effect, while using the fit() method.

## Training only specific layers: an example

To understand better how it works, let's make an example. Let's again consider a feed forward network with two layers

model = Sequential()

model.add(Dense(32, activation='relu', input\_dim=784, name = 'input'))

model.add(Dense(32, activation='relu', name = 'hidden1'))

note how we have created a model with two Dense layers with a name property. One is called input, and the second is called hidden1. Now you can check the network structure with model.summary(). In this simply example you will get the following output

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input (Dense) (None, 32) 25120

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

hidden1 (Dense) (None, 32) 1056

=================================================================

Total params: 26,176

Trainable params: 26,176

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Note first how all parameters are trainable and how you can find the layer name in the first column. Please take note since assigning each layer a name will be useful in the future. To freeze the layer called hidden1, you simply need to find the layer with the name and change its trainable property

model.get\_layer('hidden1').trainable = False

now if you check again the model summary you will see a different number of trainable parameters

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input (Dense) (None, 32) 25120

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

hidden1 (Dense) (None, 32) 1056

=================================================================

Total params: 26,176

Trainable params: 25,120

Non-trainable params: 1,056

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

As you can see now, the 1056 parameters contained in the hidden1 layer, are now not trainable anymore. The layer is now frozen. If you have not assigned names to the layers and you want to find out how the layers are called, you can use the model.summary() function or you can simply loop through the layers in the model with

for layer in model.layers:

print (layer.name)

that will give you the output

input

hidden1

as expected. Note that model.layers is simply a list with layers as elements. As such you can use the classical way of accessing elements from a list. For example, to access the last layer you can use

model.layers[-1]

or to access the first

model.layers[0]

to freeze the last layer for example you can simply use

model.layers[-1].trainable = False

When you change a property of a layer in Keras, like the trainable property, remember to recompile the model with the compile() function, otherwise the change will not take effect during the training.

To summarize consider the following code[[6]](#footnote-6)

x = Input(shape=(4,))

layer = Dense(8)

layer.trainable = False

y = layer(x)

frozen\_model = Model(x, y)

now running the following code

frozen\_model.compile(optimizer='Adam', loss='mse')

frozen\_model.fit(data, labels)

will NOT modify the weights of layer. In fact calling frozen\_model.summary() gives

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_1 (InputLayer) (None, 4) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_6 (Dense) (None, 8) 40

=================================================================

Total params: 40

Trainable params: 0

Non-trainable params: 40

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

And as expected there are no trainable parameters.

Now we can simply modify the property layer.trainable

layer.trainable = True

trainable\_model = Model(x, y)

compile and fit the model

trainable\_model.compile(optimizer='Adam', loss='mse')

trainable\_model.fit(data, labels)

and this time the weights of layer will be updated. We can check that with trainable\_model.summary()

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input\_1 (InputLayer) (None, 4) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_6 (Dense) (None, 8) 40

=================================================================

Total params: 40

Trainable params: 40

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

And now all parameters are trainable, as we wanted.

## Removing layers

Something very useful is to remove one or more of the last layers in a model and add different ones to fine tune it. The idea is used very often in transfer learning, when you train a network and want just to fine tune its behavior training only the last few layers. Let's consider the following model

model = Sequential()

model.add(Dense(32, activation='relu', input\_dim=784, name = 'input'))

model.add(Dense(32, activation='relu', name = 'hidden1'))

model.add(Dense(32, activation='relu', name = 'hidden2'))

the summary() call will give the output

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input (Dense) (None, 32) 25120

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

hidden1 (Dense) (None, 32) 1056

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

hidden2 (Dense) (None, 32) 1056

=================================================================

Total params: 27,232

Trainable params: 27,232

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Now if you want to build a second model, keeping your trained weights in the input and hidden1 layers, but you want to substitute the hidden2 layer with a different one (let's say one with 16 neurons) you can easily do in this way

model2 = Sequential()

for layer in model.layers[:-1]:

model2.add(layer)

and this gives you

Layer (type) Output Shape Param #

=================================================================

input (Dense) (None, 32) 25120

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

hidden1 (Dense) (None, 32) 1056

=================================================================

Total params: 26,176

Trainable params: 26,176

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

At this point you can simply add a new layer with

model2.add(Dense(16, activation='relu', name = 'hidden3'))

that has the structure

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

input (Dense) (None, 32) 25120

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

hidden1 (Dense) (None, 32) 1056

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

hidden3 (Dense) (None, 16) 528

=================================================================

Total params: 26,704

Trainable params: 26,704

Non-trainable params: 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

After that remember to compile your model. For example, for a regression problem your code may look like this

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

# Keras callback 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 stages of the training [https://keras.io/callbacks/, Accessed 01/02/2019]. Keunwoo Choi has written a nice overview on how to write a callback class that you can find here <https://goo.gl/hL37wq>. Let's summarize it here and expand it with some practical examples.

## Custom callback class

The abstract base class Callback can be found at the moment of writing at

[tensorflow/python/keras/callbacks.py](https://www.tensorflow.org/code/stable/tensorflow/python/keras/callbacks.py) (https://goo.gl/uMrMbH).

To start you need to define a custom class. The main methods you want to redefine are typically 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 code

import keras

class My\_Callback(keras.callbacks.Callback):

    def on\_train\_begin(self, logs={}):

        return

    def on\_train\_end(self, logs={}):

        return

    def on\_epoch\_begin(self, epoch, logs={}):

        return

    def on\_epoch\_end(self, epoch, logs={}):

        return

    def on\_batch\_begin(self, batch, logs={}):

        return

    def on\_batch\_end(self, batch, logs={}):

        self.losses.append(logs.get('loss'))

        return

Each of the methods have slightly different inputs that you may use in your class. Let's look at them briefly (you can find them in the original Python code at https://goo.gl/uMrMbH).

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. Same code you have already seen 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

and then 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 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

you 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. You 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 our 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 training 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

will give the 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 now that we can access and use. In the dictionary now we have val\_loss, val\_acc and acc. So let's customize our output a bit. Let's set verbose = 0 in the fit() call to suppress the standard output and let's 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

and 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

and 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 simply 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

this will give you the following output if you train your network for a 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 on how you can perform several things during the training. A typical usage of callbacks that we will look at in the next section, is saving your model every few epochs. But you can for example save accuracy values in lists to be able to plot them later, or simply plot metrics to check how your training is going.

# 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 (obtainable with !ls training) 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.

# Dataset abstraction

## Introduction

The tf.data.Dataset[[10]](#footnote-10) is a new abstraction in TensorFlow that is very useful to build data pipelines. Is also very useful when you are dealing with datasets that does not fit in memory. We will see in the book how to use it in more details. In the next sections I will try to give you some hints on a couple of ways in which you can use it in your projects. To learn how to use it, a good starting point is to study the official documentation at <https://www.tensorflow.org/guide/datasets>. Remember: always start there when you want to learn more about a specific method or feature of TensorFlow.

Basically, a Dataset it is simply a sequence of elements, in which each element contains one or more Tensors. Typically, each element will be one training example or a batch of them. The basic idea is that first you create a Dataset with some data, and then you chain method calls on it, for example applying the Dataset.map() to apply a function to each element. Note that a dataset is made of elements each with the same structure.

As usual, let's consider an example to understand how this work and how to use it. Let's suppose we have as input a matrix of 10 rows and 10 columns defined by

inp = tf.random\_uniform([10, 10])

we can simply create a dataset with

dataset = tf.data.Dataset.from\_tensor\_slices(inp)

using print(dataset) will get you the output

<TensorSliceDataset shapes: (10,), types: tf.float32>

That tells you that each element in the dataset is a tensor with 10 elements (the rows in the inp tensor). A nice possibility is to apply specific functions to each element in a dataset. For example, we could multiply all elements by two

dataset2 = dataset.map(lambda x: x\*2)

to check what happened we could print the first element in each dataset. This can be easily done (more on that later) with

dataset.make\_one\_shot\_iterator().get\_next()

and

dataset2.make\_one\_shot\_iterator().get\_next()

from the first you will get (your number will be different since we are dealing with random numbers here)

<tf.Tensor: id=62, shape=(10,), dtype=float32, numpy=

array([0.2215631 , 0.32099664, 0.04410303, 0.8502971 , 0.2472974 ,

0.25522232, 0.94817066, 0.7719344 , 0.60333145, 0.75336015],

dtype=float32)>

and from the second

<tf.Tensor: id=71, shape=(10,), dtype=float32, numpy=

array([0.4431262 , 0.6419933 , 0.08820605, 1.7005942 , 0.4945948 ,

0.51044464, 1.8963413 , 1.5438688 , 1.2066629 , 1.5067203 ],

dtype=float32)>

and as expected the second contains all numbers of the first multiplied by two.

tf.data.dataset is designed to build data processing pipelines. For example, in image recognition you could do data augmentation, preparation, normalization and so on in this way.

I strongly suggest you check the official documentation to get more information on different ways of applying a function to each element. For example, you may need to apply a transformation to the data and then flatten the result (see flat\_map() for example).

## Iterating over a dataset

Once you have your dataset, you probably want to process the elements one by one, or in batches. To do that you need an iterator. For example to process the elements of what we defined before one by one you can instantiate a so called make\_one\_shot\_iterator() as

iterator = dataset.make\_one\_shot\_iterator()

and then you can iterate over the elements using the get\_next() method

for i in range(10):

value = print(iterator.get\_next())

this will give you all elements in the dataset. They will look like this one (note your number will be different)

tf.Tensor(

[0.2215631 0.32099664 0.04410303 0.8502971 0.2472974 0.25522232

0.94817066 0.7719344 0.60333145 0.75336015], shape=(10,), dtype=float32)

Note that once you reach the end of the dataset, using the method get\_next() will raise a tf.errors.OutOfRangeError.

## Simple batching

The most fundamental way of batching consists in stacking n consecutive elements of a dataset in a single group. This will be very useful when we will train our networks with mini-batches. This can be done using the batch() method. Let's get back to our example. Remember our dataset has 10 elements. Let's suppose we want to create batches, each having 2 elements. This can be done with

batched\_dataset = dataset.batch(2)

now let's define again an iterator with

iterator = batched\_dataset.make\_one\_shot\_iterator()

and now let's check what get\_next() will return with

print(iterator.get\_next())

the output will be

tf.Tensor(

[[0.2215631 0.32099664 0.04410303 0.8502971 0.2472974 0.25522232

0.94817066 0.7719344 0.60333145 0.75336015]

[0.28381765 0.3738917 0.8146689 0.20919728 0.5753969 0.9356725

0.7362906 0.76200795 0.01308048 0.14003313]], shape=(2, 10), dtype=float32)

That is two elements of our dataset.

Batching with batch() method is really useful when we train a neural networks with mini-batches. Since we don't have to bother anymore in creating the batches ourself. tf.data.dataset will do it for us.

## Simple batching with the MNIST dataset

We first load the data (as before)

num\_classes = 10

mnist = tf.keras.datasets.mnist

(x\_train, y\_train), (x\_test, y\_test) = mnist.load\_data()

image\_vector\_size = 28\*28

x\_train = x\_train.reshape(x\_train.shape[0], image\_vector\_size)

x\_test = x\_test.reshape(x\_test.shape[0], image\_vector\_size)

y\_train = keras.utils.to\_categorical(y\_train, num\_classes)

y\_test = keras.utils.to\_categorical(y\_test, num\_classes)

then we create our training Dataset

mnist\_ds\_train = tf.data.Dataset.from\_tensor\_slices((x\_train, y\_train))

Now let's build our Keras model with a simple feed forward network with two layers

img = tf.placeholder(tf.float32, shape=(None, 784))

x = Dense(128, activation='relu')(img) # fully-connected layer with 128 units and ReLU activation

x = Dense(128, activation='relu')(x)

preds = Dense(10, activation='softmax')(x)

labels = tf.placeholder(tf.float32, shape=(None, 10))

loss = tf.reduce\_mean(categorical\_crossentropy(labels, preds))

correct\_prediction = tf.equal(tf.argmax(preds,1), tf.argmax(labels,1))

accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32))

train\_step = tf.train.AdamOptimizer(0.001).minimize(loss)

init\_op = tf.global\_variables\_initializer()

Now we need to define the batch size

train\_batched = mnist\_ds\_train.batch(1000)

and now let's define the iterator

train\_iterator = train\_batched.make\_initializable\_iterator() # So we can restart from the beginning

next\_batch = train\_iterator.get\_next()

it\_init\_op = train\_iterator.initializer

the it\_init\_op operation will be used to reset the iterator and be able to start from the beginning each epoch. Note that the operation next\_batch has the following structure

(<tf.Tensor 'IteratorGetNext\_6:0' shape=(?, 784) dtype=uint8>, <tf.Tensor 'IteratorGetNext\_6:1' shape=(?, 10) dtype=float32>)

Since it contains the images and the labels. So during our training we will need to get the batches in the form

train\_batch\_x, train\_batch\_y = sess.run(next\_batch)

Finally let's train our network

with tf.Session() as sess:

sess.run(init\_op)

for epoch in range(50):

sess.run(it\_init\_op)

try:

while True:

train\_batch\_x, train\_batch\_y = sess.run(next\_batch)

sess.run(train\_step,feed\_dict={img: train\_batch\_x, labels: train\_batch\_y})

except tf.errors.OutOfRangeError:

pass

if (epoch % 10 == 0 ):

print('epoch',epoch)

print(sess.run(accuracy,feed\_dict={img: x\_train,

labels: y\_train}))

now I have used a few tricks here that are good to know. In particular, since you don't know how many batches you have you can use the following construct to avoid getting error messages

try:

while True:

# Do something

except tf.errors.OutOfRangeError:

pass

in this way, when you get an OutOfRangeError since you run out of batches the except will simply go on without interrupting your code. Note how, each epoch, we call

sess.run(it\_init\_op)

to reset the iterator, otherwise we would get an OutOfRangeError immediately. Running this code will get you very fast to roughly 99% accuracy. You should see an output like this one (I report here only the output for epoch 40 for brevity)

epoch 40

0.98903334

This quick overview of the Dataset is not exhaustive by any means but should give you an idea of its power. If you want to learn more the best place, as usual, is the official documentation.

tf.data.Dataset is an extremely convenient way of building pipelines for data, beginning from loading, to manipulating, normalizing, augmenting and so on. Especially in image recognition problems this can be very useful. Remember that using it, means adding nodes to your computational graph. So no data is processed until the session evaluates the graph.

## tf.data.Dataset in eager execution mode

I would like to complete the Chapter with one final hint. If you are working in eager execution mode, your life with Datasets is even easier. For example, to iterate over a batched dataset you can simply do as you would do with classical Python (for x in …). To understand what I mean let's make an easy example. First, you need to enable eager execution

import tensorflow as tf

from tensorflow import keras

import tensorflow.contrib.eager as tfe

tf.enable\_eager\_execution()

then you can simply do

dataset = tf.data.Dataset.from\_tensor\_slices(tf.random\_uniform([4, 2]))

dataset = dataset.batch(2)

for batch in dataset:

print(batch)

this can be very useful sometime when you need to iterate over a dataset batch by batch. The output would be (your numbers will be different, due to the tf.random.uniform() call)

tf.Tensor(

[[0.07181489 0.46992648]

[0.00652897 0.9028846 ]], shape=(2, 2), dtype=float32)

tf.Tensor(

[[0.9167508 0.8379569 ]

[0.33501422 0.3299384 ]], shape=(2, 2), dtype=float32)

# Conclusions

This chapter had the goal of showing you a few techniques that we will use in this book, and that will be very helpful to your projects. The goal was not to explain you those methods in details, as that would require a separate book. But it should point you in the right direction when trying to do specific things, as saving the weights of your model regularly. In the next chapters we will use some of those techniques. In case you want to learn a bit more about them, remember to always check the official documentation.

1. <https://www.tensorflow.org/guide/eager>, (accessed 17th Jan. 2019) [↑](#footnote-ref-1)
2. You can find the notebook with the entire code in the book repository. To find it, go to the APRESS book website and click on the button "Download code". The link points to the github repository. The notebook is in "chapter2" folder. [↑](#footnote-ref-2)
3. You can check this article <https://goo.gl/hXKNnf> to learn how to do it. [↑](#footnote-ref-3)
4. The result was obtained when calling the function in a google colab notebook. [↑](#footnote-ref-4)
5. The code has been inspired by the Google code in the Google Colab documentation. [↑](#footnote-ref-5)
6. Check the official documentation for the example. <https://keras.io/getting-started/faq/#how-can-i-freeze-keras-layers> [↑](#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)
10. https://www.tensorflow.org/guide/datasets [↑](#footnote-ref-10)