Customizing Keras

# 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).

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

from tensorflow 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 many times:

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. 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.