

**Why need?**

You might have experienced short of memory while feeding datasets into the model. Even state of the art configurations might go out of memory sometimes to process the whole data. That is the reason why we need to find other ways to do that task efficiently. In this blog post, we are going to show you how to generate your dataset on multiple cores in real time and feed it right away to your deep learning model.

**Your keras script should be like this!**

import numpy as np  
from keras.models import Sequential  
  
# Load entire dataset  
X, y = np.load('some\_training\_set\_with\_labels.npy')  
  
# Design model  
model = Sequential()  
[...] # Your architecture  
model.compile()  
  
# Train model on your dataset  
model.fit(x=X, y=y)

In order to feed whole data set or maximum (dataset), let’s dive step by step that builds a data generator suited for this situation.

**Important points to know!**

Before diving into this, lets first know some important tips that might be useful.

Let ID be the Python string that identifies a given sample of the dataset. Consider the following framework to keep track of samples and their labels:

1. Create a dictionary called partition where you gather:

* in partition['train'] a list of training IDs
* in partition['validation'] a list of validation IDs

2. Create a dictionary called labels where for each ID of the dataset, the associated label is given by labels[ID]

For example, let’s say that our training set contains id1, id2 and id3 with respective labels 0, 1 and 2, with a validation set containing id4 with label 1. In that case, the Python variables partition and labels look like

>>> partition  
{'train': ['id1', 'id2', 'id3'], 'validation': ['id4']}

and

>>> labels  
{'id1': 0, 'id2': 1, 'id3': 2, 'id4': 1}

For the sake of modularity, we will write Keras code and customized classes in separate files.

Code you seeing here is more or like general so you may also can use it in your projects too!

**Data generator**

def \_\_init\_\_(self, list\_IDs, labels, batch\_size=32, dim=(32,32,32), n\_channels=1,  
 n\_classes=10, shuffle=True):  
 'Initialization'  
 self.dim = dim  
 self.batch\_size = batch\_size  
 self.labels = labels  
 self.list\_IDs = list\_IDs  
 self.n\_channels = n\_channels  
 self.n\_classes = n\_classes  
 self.shuffle = shuffle  
 self.on\_epoch\_end()

Here, the method on\_epoch\_end is triggered once at the very beginning as well as at the end of each epoch. If the shuffleparameter is set to True, we will get a new order of exploration at each pass.

def on\_epoch\_end(self):  
 'Updates indexes after each epoch'  
 self.indexes = np.arange(len(self.list\_IDs))  
 if self.shuffle == True:  
 np.random.shuffle(self.indexes)

Shuffling the order in which examples are fed to the classifier is helpful so that batches between epochs do not look alike. Doing so will eventually make our model more robust.

Another method that is core to the generation process is the one that achieves the most crucial job: producing batches of data. The private method in charge of this task is called **\_\_data\_generation** and takes as argument the list of IDs of the target batch.

def \_\_data\_generation(self, list\_IDs\_temp):  
 'Generates data containing batch\_size samples' # X : (n\_samples, \*dim, n\_channels)  
 # Initialization  
 X = np.empty((self.batch\_size, \*self.dim, self.n\_channels))  
 y = np.empty((self.batch\_size), dtype=int)  
  
 # Generate data  
 for i, ID in enumerate(list\_IDs\_temp):  
 # Store sample  
 X[i,] = np.load('data/' + ID + '.npy')  
  
 # Store class  
 y[i] = self.labels[ID]  
  
 return X, keras.utils.to\_categorical(y, num\_classes=self.n\_classes)

During data generation, this code reads the NumPy array of each example from its corresponding file ID.npy. Since our code is multicore-friendly, note that you can do more complex operations instead (e.g. computations from source files) without worrying that data generation becomes a bottleneck in the training process.

Now comes the part where we build up all these components together. Each call requests a batch index between 0 and the total number of batches, where the latter is specified in the \_\_len\_\_ method.

def \_\_len\_\_(self):  
 'Denotes the number of batches per epoch'  
 return int(np.floor(len(self.list\_IDs) / self.batch\_size))

A common practice is to set this value to

**[# samples/batch size]**

so that the model sees the training samples at most once per epoch.

Now, when the batch corresponding to a given index is called, the generator executes the \_\_getitem\_\_ method to generate it.

def \_\_getitem\_\_(self, index):  
 'Generate one batch of data'  
 # Generate indexes of the batch  
 indexes = self.indexes[index\*self.batch\_size:(index+1)\*self.batch\_size]  
  
 # Find list of IDs  
 list\_IDs\_temp = [self.list\_IDs[k] for k in indexes]  
  
 # Generate data  
 X, y = self.\_\_data\_generation(list\_IDs\_temp)  
  
 return X, y

Now, we have to modify our Keras script accordingly so that it accepts the generator that we just created above.

import numpy as np  
  
from keras.models import Sequential  
from my\_classes import DataGenerator  
  
# Parameters  
params = {'dim': (32,32,32),  
 'batch\_size': 64,  
 'n\_classes': 6,  
 'n\_channels': 1,  
 'shuffle': True}  
  
# Datasets  
partition = # IDs  
labels = # Labels  
  
# Generators  
training\_generator = DataGenerator(partition['train'], labels, \*\*params)  
validation\_generator = DataGenerator(partition['validation'], labels, \*\*params)  
  
# Design model  
model = Sequential()  
[...] # Architecture  
model.compile()  
  
# Train model on dataset  
model.fit\_generator(generator=training\_generator,  
 validation\_data=validation\_generator,  
 use\_multiprocessing=True,  
 workers=6)

As you can see, we called from model the fit\_generator method instead of fit, where we just had to give our training generator as one of the arguments. Keras takes care of the rest!

This is it! You can now run your Keras script. Then you will see that during the training phase, data is generated in parallel by the CPU and then directly fed to the GPU.

You can find whole code on Github **here**.

**Hamza Abdullah**

**https://medium.com/the-21st-century**