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 `shuffle`parameter 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