Write your own Custom Data Generator for TensorFlow Keras

This tutorial is at an intermediate level and expects the reader to be aware of basic concepts of Python, TensorFlow, and Keras.

So you want to use a custom data generator to feed in values to a model. Interesting. Why do you need data generators? Why generators at all? What is this 'generator'? What is the relevance of a data generator for machine learning?

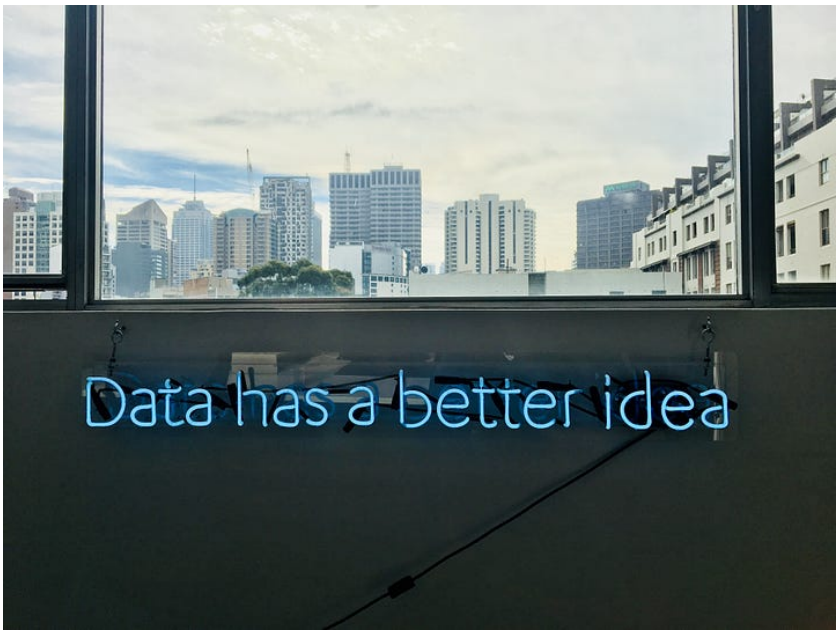

Photo by Franki Chamaki on Unsplash

What are generators?

As mentioned in the PEP document:

> *When a producer function has a hard enough job that it requires maintaining state between values produced, most programming languages offer no pleasant and efficient solution beyond adding a callback function to the producer's argument list, to be called with each value produced.*

So basically what happens is that instead of doing the hard (computation-intensive or memory-intensive) job as a whole, it breaks it down into batches and work on it as a batch. This way the parent function which invokes the child function does not have to wait until the parent function is done processing but can work on the go. A generator function we use in our day-to-day code would be `range` function. When you write a `for` loop with `range(start, end, step)` , it does not create a list with all the elements from `start` to `end` , but instead, it created a generator that can generate values from `start` to `end` and then it will create values on the go.

What are Data Generators?

Have you ever encountered a problem where the dataset you have is too big to be loaded into memory at once that you run out of RAM? For example, while training an image classifier, we won't be able to load all those images into memory before training. Even if it is possible for a small dataset, it won't be feasible for a large dataset. We should be able to use the full potential of available data.

So, if we create a data generator, we can read images on the go when they will be used for training. Since we are reading the images on the go, we are saving memory and even a system with 8GB RAM can be used for a 100GB dataset.

Standard Keras Data Generator

Keras provides a data generator for image datasets. This is available in `tf.keras.preprocessing.image` as ImageDataGenerator class. The advantage of using `ImageDataGenerator` is that it will generate batches of data with augmentation. `ImageDataGenerator` is used as follows

The `train_generator` will be a generator object which can be used in `model.fit`. The `train_datagen` object has 3 ways to feed data: `flow`, `flow from dataframe` and `flow from directory`. In this example, `flow_from_directory` is used, which means that is the data is loaded according to the directory structure. If you have a dataframe with image paths and labels, it can be used with the `flow_from_dataframe` method. Refer to this gist and ImageDataGenerator documentation to know more.

Why do I need a custom Data Generator?

The Standard Keras Generator has limited functionalities. For example, if your network has multiple output nodes, you won't be able to use the standard data generator. It also does not support tabular data. So if you have a large tabular dataset, you will need to write a custom generator. Above all, it is very easy to implement a data generator for Keras and it is extremely powerful and flexible. We can implement complex functions and preprocessing on the data before it is given as input to the model.

How to write a Custom Data Generator

To understand the custom data generators, you should be familiar with the basic way of model development and how to use `ImageDataGenerator` in `tf.keras`. If you do, it's very easy.

One method to write a custom data generator is to write a simple generator itself. Such a generator is given in this post. This also works for `model.fit` but it is recommended to use `tf.keras.utils.Sequence` to create data generators for Tensorflow Keras. It is explained in the documentation as:

> `Sequence` *are a safer way to do multiprocessing. This structure guarantees that the network will only train once on each sample per epoch which is not the case with generators*

Data Generator using Sequence

To create a custom data generator a class inherited from `tf.keras.utils.Sequence` needs to be created. As mentioned in the documentation:

> *Every* `Sequence` *must implement the* `__getitem__` *and the* `__len__` *methods. If you want to modify your dataset between epochs you may implement* `on_epoch_end`*. The method* `__getitem__` *should return a complete batch.*

It's that easy. We can implement the `__getitem__` and `__len__` methods in whichever way we want. The only requirement is that `__getitem__` method should return `(X, y)` value pair where `X` represents the input and `y` represents the output. The input and output can take any shape with `[batch_size, ...]` inside.

Let's get into the CODES

As mentioned in the documentation, we need to implement 2 methods. But I will also be implementing some helper functions as we use in real scenarios.

My objective is to read a csv file containing some metadata and train an image classification model. The data I have is object detection data. Therefore, one single image will contain multiple objects and I need to crop out those from the image to use for image classification training. Also, my network has two output nodes, one for classifying `name` and the other for `type`. Using the standard Keras `ImageDataGenerator` class, would not be impossible.

The Data

The data will be a CSV file and it will contain the columns

1. `filename` / `filepath` — where the image is located in the drive
2. `name` — will contain classes for first label 'name'
3. `type` — will contain classes for second label 'type'
4. `region_shape_attribues` — area of interest in the image

My data would look something like this:

filename	region_shape_attributes	name	type
[www.google.com][58]MCIM02058016_Bosch-microwa...	{'name': 'rect', 'x': 411, 'y': 92, 'width': 1...	microwave	over_the_range
459816586.jpg	{'name': 'rect', 'x': 218, 'y': 102, 'width': ...	sink	single_bowl
441746422.png	{'name': 'rect', 'x': 200, 'y': 202, 'width': ...	bathtub	alcove
257721378.png	{'name': 'rect', 'x': 434, 'y': 54, 'width': 6...	shower	standalone_shower
kitchen927.jpeg	{'name': 'rect', 'x': 64, 'y': 222, 'width': 1...	refrigerator	side_by_side

— The input to the model will be images with shape `(None, input_height, input_width, input_channel)` where `None` represents the `batch_size`.

— The output from the model will be a tuple containing two arrays `([None, n_name], [None, n_type])`, where `None` represents the `batch_size`, `n_name` is the number of classes of label 'name' and `n_type` is the number of classes of the label 'type'.

Getting started

The input to the data generator will be the dataframe, and which columns to use. Also, declare three methods as mentioned in the documentation `__getitem__`, `__len__` and `on_epoch_end`. Let's leave `__getitem__` and `on_epoch_end` empty for now. `__len__` will return the number of batches the generator can produce and it will be `floor(number_of_samples // batch_size)`. We will also calculate the number of unique values in 'name' and 'type'.

`__getitem__` method

The role of `__getitem__` method is to generate one batch of data. In this case, one batch of data will be `(X, y)` value pair where `X` represents the input and `y` represents the output.

- `X` will be a NumPy array of shape `[batch_size, input_height, input_width, input_channel]`. The image should be read from the disk, and the area of interest will be cropped out the image and preprocessing, if anything, has to be done according to the dataframe.
- `y` will be a tuple with two NumPy arrays of shape `[batch_size, n_name]` and `[batch_size, n_type])`. These will be one hot encoded value of the labels.

We will define two helper functions, one for input and the other for output.

Helper function for output

This function takes the path for one image, the bounding box coordinated as a dict, and the target size for the image. The function will return a NumPy array of shape `[target_height, target_width, 3]`.

Helper function for output

This function takes the value and number of classes and returns a one-hot encoded NumPy array of shape `[num_classes,]`

Generates data containing batch_size samples

This function will take a batch of data, the X_col as a string and y_col as a dict. It will iterate over the batch and call helper function, aggregate them, and returns as a tuple `(X, y)` accordingly.

The `__getitem__` function taking a batch of data from the dataframe using indexing and passing it into the `__get_data` function to get `X` and `y` to be used for training. The `index` passed into the function will be done by the `fit` function while training. Select the correct batch of data by using this index.

__getitem__ method calling __getdata method with one batch data

on_epoch_end

This function will be called at the end of every epoch by the `fit` method. Thus any modification to the dataset between epochs may be implemented in this function. We will shuffle the dataset in this function to make some randomness.

Shuffling the table after each epoch

CustomDataGen

Coming together, the generator will look like this

CustomDataGen class as a whole

Now we just need to create an object of this class and pass it o the fit function like this

Creating Instances and fitting to the model

Conclusion

Now you can change according to any input and output for your Keras model by changing the `__get_input`, `__get_output`, and `__get_data` functions. Since our CustomDataGen is inheriting from the `Sequence` module, the whole operation will be threaded and data is generated in parallel by the CPU and then directly fed to the GPU.

Some common errors and how to avoid them

- The most common error which will be made while making a custom data generator will be that lists will be used instead of NumPy arrays. Tensorflow will throw an error that will have no relation to this and you probably will search for hours before realizing that this was the issue. So always check whether your data and labels are all NumPy arrays
- Another common error is when the shape of the input/output is wrong when multiple input/output nodes are there. Always remember to have a batch item tuple with `([batch_size, ...], [batch_size, ...], ...)` instead of `([...], [...], ... [(batch_size – 1)nth item])`. Each tuple item must be of size `[batch_size, ...])`.
- Make sure that the output of `__len__` is `int` type. The training will throw an error if it is of any other type, even `np.int`.
- Try to get an item from the generator: `X, y= traingen[0]`, this will help in making sure that your outputs from the generators are what you expect them to be
- Make sure to encode all your inputs in the dataframe before since `to_categorical` will require integer inputs. If not, implement a system to take care of it inside the generator class.