Shervine Amidi
- About
- Projects
- Teaching
- Blog
- About

Afshine Amidi

A detailed example of how to use data generators with Keras

`python` `keras 2` `fit_generator` `large dataset` `multiprocessing`

Note: you can fork the minimal example presented in this tutorial on GitHub.

Motivation

Have you ever had to load a dataset that was so memory consuming that you wished a magic trick could seamlessly take care of that? Large datasets are increasingly becoming part of our lives, as we are able to harness an ever-growing quantity of data.

We have to keep in mind that in some cases, even the most state-of-the-art configuration won't have enough memory space to process the data the way we used to do it. 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**.

The framework used in this tutorial is the one provided by Python's high-level package *Keras*, which can be used on top of a GPU installation of either *TensorFlow* or *Theano*.

Tutorial

Previous situation

Before reading this article, your Keras script probably looked 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)
```

This article is all about changing the line loading the entire dataset at once. Indeed, this task may cause issues as all of the training samples may not be able to fit in memory at the same time.

In order to do so, let's dive into a step by step recipe that builds a data generator suited for this situation. By the way, the following code is a good skeleton to use for your own project; you can copy/paste the following pieces of code and fill the blanks accordingly.

Notations

Before getting started, let's go through a few organizational tips that are particularly useful when dealing with large datasets.

Let `ID` be the Python string that identifies a given sample of the dataset. A good way to keep track of samples and their labels is to adopt the following framework:

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 `id-1`, `id-2` and `id-3` with respective labels `0`, `1` and `2`, with a validation set containing `id-4` with label `1`. In that case, the Python variables `partition` and `labels` look like

```
>>> partition
{'train': ['id-1', 'id-2', 'id-3'], 'validation': ['id-4']}
```

and

```
>>> labels
{'id-1': 0, 'id-2': 1, 'id-3': 2, 'id-4': 1}
```

Also, for the sake of **modularity**, we will write Keras code and customized classes in separate files, so that your folder looks like

```
folder/
├── my_classes.py
```

```
├── keras_script.py
└── data/
```

where `data/` is assumed to be the folder containing your dataset.

Finally, it is good to note that the code in this tutorial is aimed at being **general** and **minimal**, so that you can easily adapt it for your own dataset.

Data generator

Now, let's go through the details of how to set the Python class `DataGenerator`, which will be used for real-time data feeding to your Keras model.

First, let's write the initialization function of the class. We make the latter inherit the properties of `keras.utils.Sequence` so that we can leverage nice functionalities such as *multiprocessing*.

```
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()
```

We put as arguments relevant information about the data, such as dimension sizes (e.g. a volume of length 32 will have `dim=(32,32,32)`), number of channels, number of classes, batch size, or decide whether we want to shuffle our data at generation. We also store important information such as labels and the list of IDs that we wish to generate at each pass.

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 (or just keep a linear exploration scheme otherwise).

```
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.

Also, please note that we used Keras' `keras.utils.to_categorical` function to convert our numerical labels stored in `y` to a binary form (e.g. in a 6-class problem, the third label corresponds to `[0 0 1 0 0 0]`) suited for classification.

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

$$\left\lfloor \frac{\#\text{ samples}}{\text{batch size}} \right\rfloor$$

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
```

The complete code corresponding to the steps that we described in this section is shown below.

```
import numpy as np
import keras

class DataGenerator(keras.utils.Sequence):
```

```
        'Generates data for Keras'
        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()

        def __len__(self):
            'Denotes the number of batches per epoch'
            return int(np.floor(len(self.list_IDs) / self.batch_size))

        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

        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)

        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)
```

Keras script

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

```
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!

Note that our implementation enables the use of the `multiprocessing` argument of `fit_generator`, where the number of threads specified in `n_workers` are those that generate batches in parallel. A high enough number of workers assures that CPU computations are efficiently managed, *i.e.* that the bottleneck is indeed the neural network's forward and backward operations on the GPU (and not data generation).

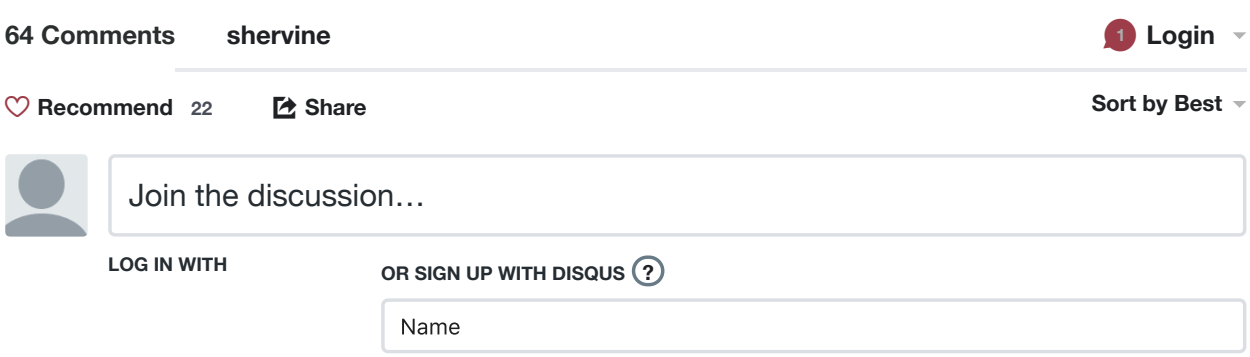
Conclusion

This is it! You can now run your Keras script with the command

```
python3 keras_script.py
```

and 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 a complete example of this strategy on applied on a specific example on GitHub where codes of data generation as well as the Keras script are available.



Deep learning enthusiast • 10 months ago
Nice work!!
2 • Reply • Share ›

Shervine Author → Deep learning enthusiast • 10 months ago
Thank you!
• Reply • Share ›

deep_enthusiast → Shervine • 2 months ago
How does one guarantee (with your solution) that each example is only fed once?
• Reply • Share ›

Shervine Author → deep_enthusiast • a month ago
Thanks for your interest in this!
This is guaranteed by the fact that we specified an appropriate number of batches per epoch in the `__len__` method. Keras takes care of the rest!
• Reply • Share ›

Karl • a month ago
What a great tutorial.

Could you elaborate a bit more on the choice of dim=(32,32,32)?
1 • Reply • Share ›

Shervine Author → Karl • a month ago
Thank you very much for your message Karl!
Sure! `dim = (32,32,32)` was for volumes of size 32x32x32. But let's say that you wish to use this code to generate RGB images of size 224x224, then you would need to set `dim = (224,224)` and `n_channels = 3`.
• Reply • Share ›

ThomasTiddlyWinkChauncyPepperw • 6 months ago
2 quick questions:

(1) I don't understand what is happening here:

```
imax = int(len(indexes)/self.batch_size)
for i in range(imax):
# Find list of IDs
list_IDs_temp = [list_IDs[k] for k in indexes[i*self.batch_size:(i+1)*self.batch_size]]
```

Specifically in regards to iterating through range(imax): I thought it was basically doing steps_per_epoch, but then I saw you specify steps_per_epoch in model.fit_generator.

(2) Can I pass this class to ImageDataGenerator and and use it for data augmentation (e.g. flips, shifts, etc.?)

Thank you, great post!
1 • Reply • Share ›

Shervine Author → ThomasTiddlyWinkChauncyPepperw • 6 months ago
Hi Thomas, thanks for your message! To answer your questions:
(1) The `steps_per_epoch` argument of the `fit_generator` method indicates how many batches Keras takes from the generator before it moves to the next epoch.
The loop you highlighted in my code is coordinated with that fact, and is written in a way that keeps track of the kind of data that is generated at each epoch.

(2) I just checked the docs and it looks that Keras' `ImageDataGenerator` class is an end-to-end black box. No worries though as there is still the possibility to add these functionalities manually to `DataGenerator`.

Reply • Share ›

Meme • a month ago

Hi, thanks for this interesting tutorial .. I am wondering if I have .png images instead of .npy, for example I have the train folder which has three sub-folders each indicates to a specific class and contains png images data. should I load it and convert it to .npy and then process it with your code, or is there a way to process with your code directly. Thanks a lot

Reply • Share ›

Shervine Author → Meme • a month ago

Hi, thank you for your message!
Yes of course, you can adapt the code to load images directly from their `.png` extension. This tutorial was just using `.npy` files as proof of concept. The loading process is not file-specific!

Reply • Share ›

Mohamed Alabasiri • a month ago

Hi there **@Shervine** , truly amazing post that guides through the process of creating a keras `keras.utils.Sequence` generator; I appreciate that you kept everything neat and modular it brings me peace and serenity.
I just started with keras and python generators and have a question, what if I add augmentation to the `samples` that I feed the generator which will in turn increase the number of batches `len`, do I have to specify it "manually" for lack of a better work?

Reply • Share ›

Shervine Author → Mohamed Alabasiri • a month ago

Thank you so much for your kind message Mohamed!
My opinion is that you would not need to change anything manually, simply because you would perform data augmentation on the fly, as samples are being generated (i.e. we do not change the size of the dataset). To do so, please feel free to add any augmentation operations that you wish in the `for` loop of the `__data_generation` method.

Reply • Share ›

Phuc Cuong Ngo • a month ago

Thank you for the great post!
How to load images into partition dictionary?
Thank you very much

Reply • Share ›

Shervine Author → Phuc Cuong Ngo • a month ago

Thank you very much for your kind message!
The trick here is to restrain from loading any images in the `partition` dictionary. That would be memory consuming and not tractable for large datasets. Instead, we specify there only the `ids` that we wish to generate.
To have a minimal working example, please feel free to follow the steps mentioned at the beginning of the tutorial regarding how to organize files in directories.

Reply • Share ›

Toby • a month ago

thanks for this great tutorial. I followed this for my own data, however the validation loss is not being computed (it is not shown after epoch with the training loss). Does anyone have any ideas about why this can happen? much appreciated.

Reply • Share ›

Shervine Author → Toby • a month ago

Hi Toby, thank you very much for your message!
Have you defined, as in the tutorial, a `validation_generator` object of your validation data that you passed in the `fit_generator` method?

Reply • Share ›

Maya Rani • a month ago

Thanks for your guide.

I have a question: In my case I split the train/validation and test data in different .txt files, where each line contains the "directory/name.npy label" (i.e. the training.txt has ".....\standing\person19_handwaving_d1_s1.npy 0",".....\walking\person23_walking_d4_s3.npy 1" etc.) ..how can I load the every item? ...and also they have different size, so probably need to do padding based on the mximum length of an item in the file? Thank you in advance!

Reply • Share ›

Shervine Author → Maya Rani • a month ago

Hi Maya, thanks for your message!
1. One quick solution is to specify the complete path of each file in its associated `id` in the dictionaries that are called `partition` and `labels` in the tutorial.
2. Yes, padding is a possible option. You can also think of resizing or cropping the images.

Reply • Share ›

Shen Linyao • 2 months ago

Great work! Thank you for that! And I have a question about how to do data augmentation in data generator, do you have some ideas?

Reply • Share ›

Shervine Author → Shen Linyao • a month ago

Hi Shen, thank you very much for your kind words!
You can do so by adding any operations that you wish in the `__data_generation` method. That way, augmentation will be done on the fly and will immediately follow data loading.

Reply • Share ›

林敬翔 • 2 months ago

Hi, thank you for your nice work.
It's work really fine in my project, but there are several questions I faced for advanced application
(1) I want to augment my dataset by shift and rotation, I have already defined the function like this:

```
def augmentation(self, image, imageB, org_width=512,org_height=512, width=536, height=536):
max_angle=20
image=cv2.resize(image,(height,width))
imageB=cv2.resize(imageB,(height,width))
print(np.max(image))
# image rotate
angle=np.random.randint(max_angle)
if np.random.randint(2):
angle=-angle
image = rotate(image,angle,resize=True)
imageB = rotate(imageB,angle,resize=True)

image=cv2.resize(image,(org_height,org_width))
imageB=cv2.resize(imageB,(org_height,org_width))
return image,imageB
```

But it just can't use directory in DataGenerator

(2) For my understanding, this work still limited for batch size. Large batch size may be a problem. So I want to know is there any way to train an arbitrary batch size

So I want to know is there any way to train on arbitrary batch size.

Please forgive me for any misunderstanding, and appreciate for any reply.

Reply • Share ›

Shervine Author → 林敬翔 • a month ago

Thank you for your kind message! To answer your two questions:
1. Great! Now you can directly call your function in the `__data_generation` method. More generally, you can append any preprocessing steps that you wish in the `for` loop.
2. The only thing that limits batch size now is your RAM/GPU memory, and there isn't really anything you can do about it (besides adding more memory).

Reply • Share ›

327beckham • 2 months ago

Thank you very much! This tutorial saves me a lot of time.
This works nice until this issue https://github.com/keras-te....
When set the TensorBoard with histogram_freq=1, the error will raise. Maybe the community will fix this soon.

Reply • Share ›

Shervine Author → 327beckham • 2 months ago

Thank you for your kind message!
In the meantime regarding the Tensorboard issue, a practical workaround is to load the validation set in memory beforehand. This is usually fine if you constrain the number of validation samples accordingly.

Reply • Share ›

Luc Soret • 2 months ago

Hi, it works fine, thank you for that !
Just, do you have any idea of how adapt the model.predict_generator to do prediction based on this ?

Reply • Share ›

Shervine Author → Luc Soret • 2 months ago

Hi Luc, thanks for your comment!
The structure of the generator is exactly the same as above, where `list_IDs` now contains IDs of the samples you wish to predict. The corresponding line of code in your Keras script will be then written

```
model.predict_generator(generator=testing_generator, use_multiprocessing=True, worl
```

Reply • Share ›

Luc Soret → Shervine • 2 months ago

Perfect, thank you !

Reply • Share ›

James Johnson • 3 months ago

Hi, thanks for the amazing tutorial on data generator.I request your help on creating ''

some_training_set_with_labels.npy". I want to create a dataset(data.npy) that has the same format as cifar10. The sample omniglot dataset contains

data =
[0001_01.png,0001_02.png,0001_03.png,0001_04.png,0002_01.png,0002_02.png,0002_03.png]

class_name,filename = data[0].split('_')

Each file is append with class.There are 1600 classes and each class has 20 samples. The expected dataset(data.npy) shape is (1600,20,784). But the shape i got is (1,20,784).

Below given is the snippet

```
classes = []
examples = []
prev = files[0].split('_')[0]
for f in files:
cur_id = f.split('_')[0]
```

see more

Reply • Share ›

Shervine Author → James Johnson • 2 months ago

Hi James, thank you for your kind message.
Regarding your current issue, an output shape of `(1,20,784)` seems to correspond to an array containing all 20 samples of a single class. From there, you can save each element separately with the appropriate naming in your `data/` folder.

Reply • Share ›

Rishab • 3 months ago

Hey, I wanted to predict on my test data and I can't figure out how to load test images using flow_from_directory() method.
All the test images are in '/test/'. There are no sub-folders.
Please, suggest a way to do it

Reply • Share ›

Shervine Author → Rishab • 2 months ago

Hi Rishab, you can use this tutorial where your `test/` folder can act as the `data/` folder presented above. Then, you can customize the operations you wish to make on the fly by modifying the `__data_generation` method of the `DataGenerator` class accordingly.

Reply • Share ›

Luke Tonin • 3 months ago

Hi, thanks for the the great article.

I have 2 questions:
1. Is there a negative impact of having one training example per .npy file? (e.g data takes up more memory on disk? slower performance due to loading an .npy as many times as there are training examples.)
2. Did you consider creating your DataGenerator class as a subclass of the Keras Sequence class which appears to be the recommended method for creating data generators for Keras? If so why didn't you use it?

Have a nice day.

Reply • Share ›

Shervine Author → Luke Tonin • 2 months ago

Hi Luke, thank you very much for your kind words.

About your two questions:
1. You are raising good points here. Overall, it is of course less efficient than just loading everything at once (if you can do it), as you have to repeat the same operations all over again. However, the fact that this process is cheap computationally and that it can be done in parallel makes data generation optimal enough so it does not become the bottleneck of the training process.
2. Thank you for this great suggestion. I was not aware of this functionality when I wrote the first version of this tutorial. Now I updated everything accordingly so that `DataGenerator` inherits from the nice functionalities of `keras.utils.Sequence`.

All the best!

All the best!

Reply • Share ›

deepLearningBeginner • 4 months ago

Hi! Thank you for the tutorial! I'm a bit confused regarding the --generate(self, labels, list_IDs): imax = int(len(indexes)/self.batch_size) --> does this this line determine the number of batch within an epoch? Is the number of batch not the number of the training samples divided by the batch size?--- But in this case, len(indexes) corresponds to the number of classes, since list_IDs is the list of classes (id-1, id-2,...)? Am I missing something? Thanks in advance!

Reply • Share ›

Shervine Author ➚ deepLearningBeginner • 2 months ago

Thank you for your message!
Exactly -- the number of batches is the number of training samples divided by the the batch size. Here, `list_IDs` is the list of samples to generate and `labels` denote their associated labels.

Reply • Share ›

Vova Kuzmenkov • 4 months ago

Hi! Cool article.
Would you advise on the shuffle parameter please?
fit_generator has it's own `shuffle=True` parameter, and also your generator has one, so... how does fit_generator's shuffle=True affect batches, if at all (I mean would it even affect anything in this case)?

Reply • Share ›

Shervine Author ➚ Vova Kuzmenkov • 4 months ago

Hi Vova, thank you for your message!
According to Keras docs, the `shuffle` option of `fit_generator` **only** applies to `keras.utils.Sequence` instances.
Therefore, this argument will have no impact on the `shuffle` parameter of the data generator described in this post.

Edit: since the current version of the data generator presented in this tutorial now inherits from `keras.utils.Sequence`, please note that enabling `shuffle=True` in `fit_generator` will have the effect of making `__getitem__` calls with random indexes.

1 Reply • Share ›

wesjlizo • 5 months ago

Great! :)

Reply • Share ›

Shervine Author ➚ wesjlizo • 5 months ago

Thank you!

Reply • Share ›

Laureline • 6 months ago

Hi!
Thanks for the tuto, it's excellent.
I remain with one question though: how can I normalize my training data set according to one dimension, and save the corresponding means and standard deviations? Shall I run through the whole dataset prior to use the data generator (which is quite costly)? Or would a batch normalization be enough (but then I don't have access to the overall means and stds)?

Cheers,
Lauréline

Reply • Share ›

Shervine Author → Laureline • 6 months ago

Hi Lauréline, thank you for your kind words!

Personally, I would go for the first option. Indeed if your dataset is fixed or if you suppose that your sample is a good representation of the true data distrubution, then you can compute means and standard deviations only once and consider that this is a correct approximation. But even if this doesn't hold, you can have in mind that a single pass through the dataset is relatively cheap compared to the whole training process of your model (~100 epochs).

I have written a method you could add to the `DataGenerator` class to do this for you:

```
def compute_stats(self, list_IDs):
    'Computes mean and standard deviation of a given dataset'
    # Parameters
    n_samples = len(list_IDs)
    n_dimensions = 3
    average_on_axes = (0,1)

    # Initialization
    means, stds = np.zeros(n_dimensions), np.zeros(n_dimensions)

    # Loop for means
    for ID in list_IDs:
        # Generate sample
```

see more

Reply • Share ›

Laureline → Shervine • 5 months ago

Thanks for the answer, and for having taken the time to add the code! I will try both batch normalization and global normalization to compare both.
Best wishes for the year to come!

Reply • Share ›

Kong sea • 6 months ago

The best introduction to Keras data generator I have ever read.
Thank you!

Reply • Share ›

Shervine Author → Kong sea • 6 months ago

Great to hear, thank you so much!

Reply • Share ›

enty • 6 months ago

Thanks for sharing your code. Sparsify should be called densify because it goes form a sparse form (y=5) to a dense form using one-hot encoding (y=[0, 0, 0, 0, 0, 1, 0]).

Reply • Share ›

Shervine Author → enty • 6 months ago

Hi enty, thank you for your message. In fact, the term `sparsify` here indicates the sparse nature of the output matrix which has lots of zeros.

Reply • Share ›

Paul Roggeveen • 6 months ago

Hi Shervine,

Great piece of code! I have slightly modified it because 1.) I'd like to use pictures which I import in the function instead of reading from .npy and 2.) I'd like to use two inputs (pictures (X1) and features (X2) describing the pictures). I have made extra dictionaries, one with paths to each picture and one with the additional features (50 features) describing the pictures. The ID is the id of the picture. This seems to work perfectly when I use the __next__() on the generator. Dimensions