Python Image Generators

Have you tried to load the entire image dataset into numpy arrays? If you tried, you might have realized that it takes over **60GB of RAM**. In this tutorial we'll go over an easy way in Python to get around this problem and load images on the fly from disk (the file itself).

Prereqs

To get the most out of this tutorial, you should be familiar with the following concepts:

- Basic Python data structures
  - What a list is
  - What a dictionary is
- Functions
  - What a function is
  - How to create and use functions
- List Comprehensions
  - What a list comprehension is
  - How to create a simple list comprehension

In [1]:
```
# List
a = [1, 2, 3, 6, 'abc']
print('List:', a)
# Dictionary
b = {1: 32, 'abc': [1, 2, 3]}
print('Dictionary:', b)
```

```
# Function
def hello():
    return 'Hello World'
print('Function "hello" returns:', hello())

# List Comprehensions
c = [item + item for item in a]
print('List Comprehension:', c)
```

```
List: [1, 2, 3, 6, 'abc']
Dictionary: {1: 32, 'abc': [1, 2, 3]}
Function "hello" returns: Hello World
List Comprehension: [2, 4, 6, 12, 'abcabc']
```

Some Basic Terms

Iteration and iterables

Iteration is the repetition of some kind of process over and over again. Python's for loop gives us an easy way to iterate over various objects. Often, you'll iterate over a list, but we can also iterate over other Python objects such as strings and dictionaries.

In [2]:

```
# Iterating over a list
ez_list = [1, 2, 3]
for i in ez_list:
    print(i)
```

```
1
2
3
```

In [3]:

```
# Iterating over a string
ez_string = 'Generators'
for s in ez_string:
    print(s)
```

```
G
e
n
e
r
a
t
o
r
s
```

In [4]:

```
# Iterating over a dictionary
ez_dict = {1 : 'First', 2 : 'Second'}
for key, value in ez_dict.items():
    print(key, value)
```

```
1 First
2 Second
```

In each of the above examples, the `for` loop iterates over the sequence we give it. The code above used a list, string, and dictionary, but you can iterate over tuples and sets as well. In each loop above, we `print` each of the items in the sequence in the order they appear. For example, you can confirm that the order of the `ez_list` is replicated in the order that its items are printed out.

We refer to any object that can support iteration as an **iterable**.

What defines an iterable?

Iterables support something called the **Iterator Protocol**. The technical definition for the Iterator Protocol is out of the scope of this article, but it can be thought of as a set of *requirements* to be used for a `for` loop. That is to say: lists, strings and dictionaries all follow the Iterator Protocol, therefore we can use them in `for` loops. Conversely, objects that do not follow the protocol cannot be used in a `for` loop. One example of an object that does not follow the protocol is an integer.

If we try to give an integer to a `for` loop, Python will throw an error.

In [5]:
```
number = 12345
for n in number:
    print(n)
```

```
------------------------------------------------------------
------------
TypeError                                 Traceback (most recen
t call last)
<ipython-input-5-b97de08daa0b> in <module>()
```

```
      1 number = 12345
----> 2 for n in number:
      3     print(n)

TypeError: 'int' object is not iterable
```

An integer is just a singular number, not a sequence. You may argue that the "first" number in `number` is 1, but it is not the same as the first item in a sequence. It doesn't make sense to ask "What's after 1?" from `number` since Python only understands integers as a single entities.

Therefore, one of the requirements to be an iterable is to be able to describe to the `for` loop what the next item to perform the operation on is. For example, lists tell the `for` loop that the next item to iterate on is in the index+1 from the current one (1 comes after 0).

Consequently, an iterable must also signal to a `for` loop when to *stop* iterating. This signal usually comes when we arrive at the end of a sequence (i.e. the end of a list or string). We will explore the specific functions that make something iterable later in this article, the important thing to know is that iterables describe *how* a `for` loop should traverse its contents.

Generators are iterables themselves. As you'll see later, `for` loops are one of the main ways we use a generator, so they must be able to support iteration. We'll delve into how we can create our own generators in the next secton.

Key takeaways: basic terms to know

- Iteration is the idea of repeating some process over a sequence of items. In Python, iteration is usually related to the `for` loop.
- An iterable is an object that supports iteration.
- To be an iterable, it must describe to a for loop two things:

    - What item comes next in the iteration.
    - When should the loop stop iteration.
- Generators are iterables.

Generators and you

If you've never encountered a generator before, the most common real-life example of a generator is a backup generator, which creates — *generates* — electricity for your house or office.

Conceptually, Python generators generate values *one at a time* from a given sequence, instead of giving the entirety of the sequence at once. This one-at-a-time fashion of generators is what makes them so compatible with `for` loops. If this sounds confusing, don't worry too much. As we explain how to create generators, it will become more clear.

There are two ways to create a generator. They differ in their syntax, but the end result is still a generator. We'll teach these concepts by covering their syntax and comparing them to a similar, but non-generator equivalent.

- A generator *function* versus a regular function
- A generator *expression* versus a list comprehension

The generator function

A generator function is just like a regular function but with a key difference: the `yield` keyword replaces `return`.

In [6]:
```
# Regular function
def function_a():
```

```
    return "a"

# Generator function
def generator_a():
    yield "a"
```

The two functions above perform exactly same action (returning/yielding the same string). However, if you try to inspect the generator function, it won't match what the regular function shows.

In [7]:
```
function_a()
```

Out[7]:
```
'a'
```

In [8]:
```
generator_a()
```

Out[8]:
```
<generator object generator_a at 0x7f79695d6e60>
```

Calling a regular function tells Python to go back to where the function is located in our code, perform the code within the block, and return the result. In order to get the generator function to yield its values, you need to pass it into the `next()` function.

`next()` is a special function that asks, "What's the next item in the iteration?" In fact, `next()` is the precise function that is called when you run a for loop! Lists, dictionaries, strings, and the like all implement `next()`, so this is why you can incorporate them into loops in the first place.

In [9]:
```
# Asking the generator what the next item is
next(generator_a())
```

Out[9]:
```
'a'
```

In [10]:
```
# Do not do this
next(generator_a)
```

Python Image Generator Tutorial

Python notebook using data from COMP 540 Spring 2019 · 204 views

4 Fork 1

```
t call last)
<ipython-input-10-1eb7ad9bfd24> in <module>()
      1 # Do not do this
----> 2 next(generator_a)

TypeError: 'function' object is not an iterator
```

Version 5
5 commits

Notebook

Data

Log

Comments

Notice that we have to pass in generator function with the parentheses since the function itself is the generator. Providing only the function name will throw an error since you're trying to give `next()` a function name. As expected, the generator function will `yield` 'a' once we invoke the `next()` function.

This example is not fully representative of what a generator is useful for. Remember that generators produce a stream of values, so `yield`ing a single value doesn't really qualify as a stream. To do

this, we can actually put in multiple `yield` statements into a generator function. These `yield` statements form the sequence that the generator will output.

We'll create a generator and bind it to a varible `mg`. Then, if we keep passing `mg` into `next()`, we'll get to the next yield. If we keep going past, we'll be given a `StopIteration` error to tell us that the generator has no more values to give. The `StopIteration` error is actually how a `for` loop knows when to stop iterating.

In [11]:
```
def multi_generate():
    yield "a"
    yield "b"
    yield "c"
```


In [12]:
```
print(next(mg))
print(next(mg))
print(next(mg))
print(next(mg))
```

```
a
b
c


---------------------------------------------------------------
------------
StopIteration                             Traceback (most recen
t call last)
```

```
<ipython-input-12-4ccf14a1b0d8> in <module>()
      2 print(next(mg))
      3 print(next(mg))
----> 4 print(next(mg))

StopIteration:
```

It's easy to think of generators as a machine that waits for one command and one command only: `next()`. Once you call `next()` on the generator, it will dispense the next value in the sequence it is holding. Otherwise, you can't do much else with a generator. The image below represents our generator as a simple machine.

We've noted that as we keep passing in `mg` into next, we get the other `yield` results. This is possible only if the generator somehow remembers what it last did. This memory is what distinguishes generator functions from regular functions! Once you use a function, it's a one-and-done deal. Once you `return` the value from the function. A generator will keep `yield`ing values until its out.

This brings us to another important property of generators. Once we've finished iterating through them, we can't use them anymore. Once we got through all three `yield` values in `mg`, it can't provide anything to us anymore. We'd have to store another instance of the `multi_generate` generator to begin asking `next()` statements of it again.

Using Generators for our Image Data

If you're following along with the data on your own computer, you'll need to replace `path_to_train` with the path on your computer to where the train images are located. This will enable Python to find it and all the train images

In [13]:

```
from glob import glob
import os

path_to_train = '../input/train/train'
glob_train_imgs = os.path.join(path_to_train, '*_sat.jpg')
glob_train_masks = os.path.join(path_to_train, '*_msk.png')

train_img_paths = glob(glob_train_imgs)
train_mask_paths = glob(glob_train_masks)
print(train_img_paths[:10])
print(train_mask_paths[:10])
```

```
['../input/train/train/10417_sat.jpg', '../input/train/train/53
454_sat.jpg', '../input/train/train/53312_sat.jpg', '../input/t
rain/train/4890_sat.jpg', '../input/train/train/30497_sat.jpg',
'../input/train/train/33634_sat.jpg', '../input/train/train/370
29_sat.jpg', '../input/train/train/50343_sat.jpg', '../input/tr
ain/train/28951_sat.jpg', '../input/train/train/40778_sat.jpg']
['../input/train/train/4698_msk.png', '../input/train/train/401
24_msk.png', '../input/train/train/16663_msk.png', '../input/tr
ain/train/13405_msk.png', '../input/train/train/52877_msk.png',
'../input/train/train/37349_msk.png', '../input/train/train/420
64_msk.png', '../input/train/train/21591_msk.png', '../input/tr
ain/train/9771_msk.png', '../input/train/train/24537_msk.png']
```

Our generator will work in the following way:

- We iterate over the filenames for the images we want to load
- Open the image file using a library like PIL or scikit-image
- Open the corresponding mask
- Yield the image and mask pair

In [14]:

```
from skimage.io import imread
from skimage.transform import resize

# This will be useful so we can construct the corresponding mask
def get_img_id(img_path):
    img_basename = os.path.basename(img_path)
    img_id = os.path.splitext(img_basename)[0][:-len('_sat')]
    return img_id

# Write it like a normal function
def image_gen(img_paths, img_size=(128, 128)):
    # Iterate over all the image paths
    for img_path in img_paths:

        # Construct the corresponding mask path
        img_id = get_img_id(img_path)
        mask_path = os.path.join(path_to_train, img_id + '_msk.p
ng')

        # Load the image and mask, and normalize it to 0-1 range
        img = imread(img_path) / 255.
        mask = imread(mask_path, as_gray=True)

        # Resize the images
```

```
        img = resize(img, img_size, preserve_range=True)
        mask = resize(mask, img_size, mode='constant', preserve_
range=True)
        # Turn the mask back into a 0-1 mask
        mask = (mask >= 0.5).astype(float)

        # Yield the image mask pair
        yield img, mask
```

Let's test it out! We can use `matplotlib`'s `imshow` to visualize the images.

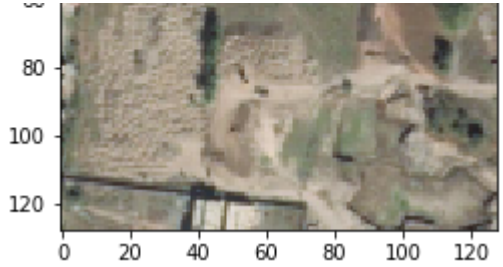
In [15]:

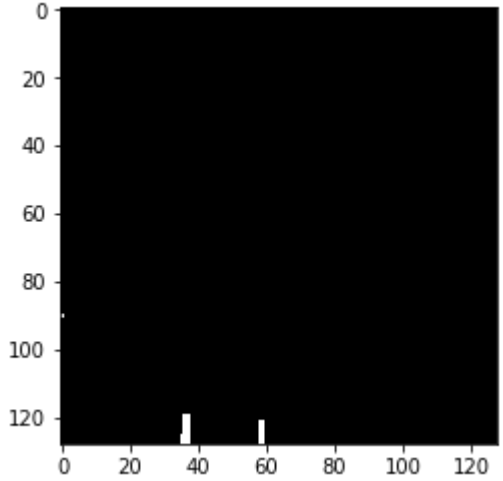
```
import matplotlib.pyplot as plt


ig = image_gen(train_img_paths)

first_img, first_mask = next(ig)

plt.imshow(first_img)
plt.show()
plt.imshow(first_mask, cmap='gray')
plt.show()
```





Using our Generator with Keras + Tensorflow

So we've just made a python generator to efficiently read our images from our disk rather than loading them all into memory. What do we do with it now? Use it to train a model! To do this we'll be using a deep learning library called Keras, built on top of Tensorflow. If you're not familiar with Tensorflow and Keras, take a look at some of these resources:

- Keras Tutorials (https://github.com/fchollet/keras-resources)
- Tensorflow Guide to Keras (https://www.tensorflow.org/guide/keras)
- Building a U-Net in Keras ()

If you want to learn more about deep learning and convolutional neural networks, take a look at these resources:

- Stanford CS231n Course Notes (http://cs231n.github.io/)
- Stanford CS231n ConvNet Notes (http://cs231n.github.io/convolutional-networks/)
- An Intuitive Explanation of ConvNets (https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/)

In [16]:
```
# Create simple model
from keras.layers import Conv2D, Reshape
from keras.models import Sequential

model = Sequential()
model.add(Conv2D(64, 5, activation='relu', padding='same', input
_shape=(128, 128, 3)))
model.add(Conv2D(128, 5, activation='relu', padding='same'))
model.add(Conv2D(1, 5, activation='sigmoid', padding='same'))
model.add(Reshape((128, 128)))
```

```
Using TensorFlow backend.
```

In [17]:
```
import keras.backend as K
from keras.optimizers import Adam
from keras.losses import binary_crossentropy

smooth = 1e-9
```

```
# This is the competition metric implemented using Keras
def dice_coef(y_true, y_pred):
    y_true_f = K.flatten(y_true)
    y_pred = K.cast(y_pred, 'float32')
    y_pred_f = K.cast(K.greater(K.flatten(y_pred), 0.5), 'float3
2')
    intersection = y_true_f * y_pred_f
    score = 2. * (K.sum(intersection) + smooth) / (K.sum(y_true_
f) + K.sum(y_pred_f) + smooth)
    return score

# We'll construct a Keras Loss that incorporates the DICE score
def dice_loss(y_true, y_pred):
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)
    return 1. - (2. * intersection + 1.) / (K.sum(y_true_f) + K.
sum(y_pred_f) + 1.)


def bce_dice_loss(y_true, y_pred):
    return 0.5 * binary_crossentropy(y_true, y_pred) + dice_coef
(y_true, y_pred)

model.compile(Adam(lr=0.01), loss=bce_dice_loss, metrics=[dice_c
oef])
```

Keras `fit_generator`

Keras follows a similar API to the famous python machine learning library, Scikit-Learn. A Keras model comes with a `fit` method that trains the model using some training data. Since training

model comes with a `fit` method that trains the model using some training data. Since training neural networks on image data often involves data that can be quite memory-intensive, Keras also includes a `fit_generator` method that takes as input a python generator of all the image data. Good thing we know how to make one!

See the documentation for `fit_generator` (https://keras.io/models/sequential/#fit_generator)

In [18]:

```
import numpy as np

# Keras takes its input in batches
# (i.e. a batch size of 32 would correspond to 32 images and 32 m
asks from the generator)
# The generator should run forever
def image_batch_generator(img_paths, batchsize=32):
    while True:
        ig = image_gen(img_paths)
        batch_img, batch_mask = [], []

        for img, mask in ig:
            # Add the image and mask to the batch
            batch_img.append(img)
            batch_mask.append(mask)
            # If we've reached our batchsize, yield the batch and
reset
            if len(batch_img) == batchsize:
                yield np.stack(batch_img, axis=0), np.stack(batc
h_mask, axis=0)
                batch_img, batch_mask = [], []

        # If we have an nonempty batch left, yield it out and res
```

```
et
        if len(batch_img) != 0:
            yield np.stack(batch_img, axis=0), np.stack(batch_ma
sk, axis=0)
            batch_img, batch_mask = [], []
```

In [19]:

```
from sklearn.model_selection import train_test_split

BATCHSIZE = 32

# Split the data into a train and validation set
train_img_paths, val_img_paths = train_test_split(train_img_path
s, test_size=0.15)

# Create the train and validation generators
traingen = image_batch_generator(train_img_paths, batchsize=BATC
HSIZE)
valgen = image_batch_generator(val_img_paths, batchsize=BATCHSIZ
E)

def calc_steps(data_len, batchsize):
    return (data_len + batchsize - 1) // batchsize

# Calculate the steps per epoch
train_steps = calc_steps(len(train_img_paths), BATCHSIZE)
val_steps = calc_steps(len(val_img_paths), BATCHSIZE)

# Train the model
history = model.fit_generator(
    traingen,
```

```
    steps_per_epoch=train_steps,
    epochs=2, # Change this to a larger number to train for longe
r
    validation_data=valgen,
    validation_steps=val_steps,
    verbose=1,
    max_queue_size=5  # Change this number based on memory restri
ctions
)
```

```
Epoch 1/2
290/290 [==============================] - 641s 2s/step - loss:
0.3076 - dice_coef: 1.9485e-04 - val_loss: 0.3007 - val_dice_co
ef: 1.0751e-13
Epoch 2/2
290/290 [==============================] - 572s 2s/step - loss:
0.3073 - dice_coef: 3.9618e-05 - val_loss: 0.3007 - val_dice_co
ef: 1.0751e-13
```

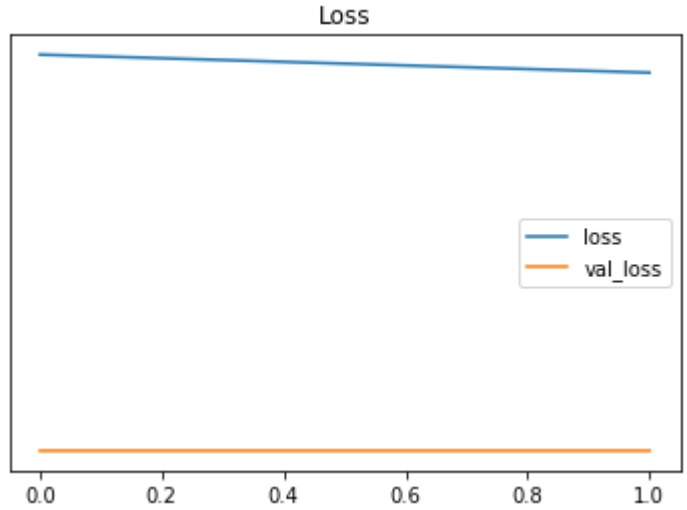
In [20]:

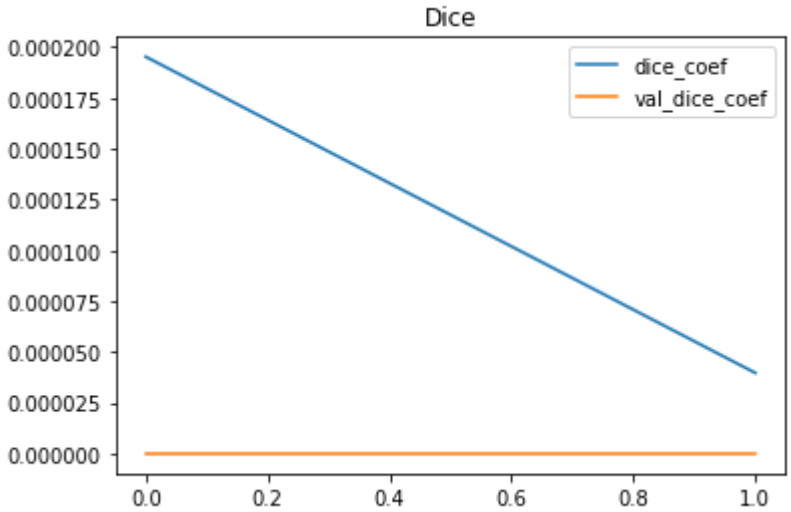
```
import pandas as pd

# Plot the training curve
pd.DataFrame(history.history)[['loss', 'val_loss']].plot(title=
"Loss", logy=True)
pd.DataFrame(history.history)[['dice_coef', 'val_dice_coef']].pl
ot(title="Dice")
```

Out[20]:

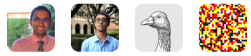
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f794da68780>
```





Did you find this Kernel useful?
Show your appreciation with an upvote

4



Data

Data Sources

- COMP 540 Spring 2...
  - samp... 2169 x 2
  - train.... 10.9k x 4
  - train.zip
    - t... 21794 files
  - val.zip
    - val 2169 files

COMP 540 Spring 2019

Detect roads in satellite images

Last Updated: a month ago

About this Competition

This dataset contains thousands of satellite images that cover a wide range of landscapes (forest, farms, country side, cities, arid landscapes, etc...). Each image covers roughly 16 acres of land in a 256 m by 256 m cube. Each image is of size 512 by 512 resulting in a pixel resolution of 50cm.

This dataset also contains a boolean mask for each satellite image that, when applied to the respective satellite image, reveals only "road" pixels. However, each mask has been segmented by hand which has many limitations. Therefore, some masks may label "road" pixels that are actually not "road" pixels or vice versa. Moreover, small farm roads and trails are not labeled as "road" pixels.

File descriptions

- **train.zip** - A zipped folder that contains the entire training set. The training set includes **i_sat.jpg** as the satellite image **i** and **i_msk.png** the respective boolean mask. The set includes 10,897 jpg satellite images and 10,897 mask images
- **val.zip** - A zipped folder that contains the entire validation set (2,170 jpg satellite images) in the same format as the training set used for submitting predictions to the competition. There should be no overlap between the two sets
- **train.csv** - The masks of the training set in Run Length Encoded (RLE) form. This data is entirely redundant to the png files found in train.zip but is offered as an example for how predicted masks will be submitted to Kaggle

Run Info

Succeeded	True	Run Time	1227.9 seconds
Exit Code	0	Queue Time	0 seconds
Docker Image Name	/python(Dockerfile)	Output Size	0
Timeout Exceeded	False	Used All Space	False
Failure Message			

Log

Download Log

```
Time  Line #  Log Message
2.8s       1  [NbConvertApp] Converting notebook __notebook__.ipynb to notebook
2.9s       2  [NbConvertApp] Executing notebook with kernel: python3
13.3s      3  2019-02-13 20:25:36.298500: I
              tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:964]
              successful NUMA node read from SysFS had negative value (-1), but
              there must be at least one NUMA node, so returning NUMA node zero
13.3s      4  2019-02-13 20:25:36.299098: I
              tensorflow/core/common_runtime/gpu/gpu_device.cc:1432] Found
              device 0 with properties:
              name: Tesla K80 major: 3 minor: 7 memoryClockRate(GHz): 0.8235
              pciBusID: 0000:00:04.0
              totalMemory: 11.17GiB freeMemory: 11.10GiB
              2019-02-13 20:25:36.299129: I
              tensorflow/core/common_runtime/gpu/gpu_device.cc:1511] Adding
              visible gpu devices: 0
14.3s      5  2019-02-13 20:25:37.229945: I
              tensorflow/core/common_runtime/gpu/gpu_device.cc:982] Device
              interconnect StreamExecutor with strength 1 edge matrix:
              2019-02-13 20:25:37.229997: I
              tensorflow/core/common_runtime/gpu/gpu_device.cc:988]      0
              2019-02-13 20:25:37.230010: I
              tensorflow/core/common_runtime/gpu/gpu_device.cc:1001] 0:   N
14.3s      6  2019-02-13 20:25:37.230401: I
              tensorflow/core/common_runtime/gpu/gpu_device.cc:1115] Created
              TensorFlow device (/job:localhost/replica:0/task:0/device:GPU:0
              with 10758 MB memory) -> physical GPU (device: 0, name: Tesla
              K80, pci bus id: 0000:00:04.0, compute capability: 3.7)
1225.2s    7  [NbConvertApp] Writing 128684 bytes to __notebook__.ipynb
1227.1s    8  [NbConvertApp] Converting notebook __notebook__.ipynb to html
1227.5s    9  [NbConvertApp] Support files will be in __results___files/
1227.5s   10  [NbConvertApp] Making directory __results___files
              [NbConvertApp] Making directory __results___files
              [NbConvertApp] Making directory __results___files
              [NbConvertApp] Making directory __results___files
              [NbConvertApp] Writing 310496 bytes to __results__.html
1227.5s   11
1227.5s   13  Complete. Exited with code 0.
```

Comments (6)

Sort by

All Comments | Hotness

Click here to enter a comment...

AlanYu • Posted on Latest Version • 2 days ago • Options • Reply 1

I ran into the same issue here.

Zhenwei F... • Posted on Latest Version • a day ago • Options • Reply 0

Have you solved this problem?
I add one line of code `mask = rgb2gray(mask)` after `mask = resize(mask, img_size, mode='constant', preserve_range=True)` to resize mask image from (128, 128, 3) to (128, 128), then model can train successfully.
I am not sure if this is the right way to do it

AbhijeetM... Kernel Author • Posted on Latest Version • 6 hours ago • Options • Reply 0

The issue has to do with a change in implementation of loading a gray image in scikit-image from when I wrote this kernel, and what Kaggle uses now. @Zhenwei Feng, I think your fix is the best, but put it before the resize. If the Kernel ever seems broken, try more updated code from the 540 Recitations GitHub:

https://github.com/abhmul/540-term-project-recitations

Zhenwei F... • Posted on Latest Version • 5 hours ago • Options • Reply 0

Thanks a lot

Zhenwei Feng • Posted on Latest Version • 3 days ago • Options • Reply

2

I was following the sample code on my machine and I ran into a problem when training the model(`history=model.fit_generator` raised an error). I didn't modify anything in the code. Keras version is 2.2.4 and tensorflow version is 1.12.0. What could be the problem? Thanks

```
---------------------------------------------------------------------------
InvalidArgumentError                      Traceback (most recent call last)
<ipython-input-11-9ae42154d3fc> in <module>()
     25     validation_steps=val_steps,
     26     verbose=1,
---> 27     max_queue_size=5  # Change this number based on memory restrictions
     28 )

/usr/local/lib/python3.6/dist-packages/keras/legacy/interfaces.py in wrapper(*args, **kwargs)
     89                 warnings.warn('Update your `' + object_name + '` call to the ' +
     90                               'Keras 2 API: ' + signature, stacklevel=2)
---> 91             return func(*args, **kwargs)
     92         wrapper._original_function = func
     93         return wrapper

/usr/local/lib/python3.6/dist-packages/keras/engine/training.py in fit_generator(self, generator, steps_per_epoch, epochs, verbose, callbacks, validation_data, validation_steps, class_weight, max_q
   1416             use_multiprocessing=use_multiprocessing,
   1417             shuffle=shuffle,
-> 1418             initial_epoch=initial_epoch)
   1419
   1420     @interfaces.legacy_generator_methods_support

/usr/local/lib/python3.6/dist-packages/keras/engine/training_generator.py in fit_generator(model, generator, steps_per_epoch, epochs, verbose, callbacks, validation_data, validation_steps, class_we
    215                 outs = model.train_on_batch(x, y,
    216                                             sample_weight=sample_weight,
--> 217                                             class_weight=class_weight)
    218
    219                 outs = to_list(outs)

/usr/local/lib/python3.6/dist-packages/keras/engine/training.py in train_on_batch(self, x, y, sample_weight, class_weight)
   1215             ins = x + y + sample_weights
   1216         self._make_train_function()
-> 1217         outputs = self.train_function(ins)
   1218         return unpack_singleton(outputs)
   1219

/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py in __call__(self, inputs)
   2713                 return self._legacy_call(inputs)
   2714
-> 2715             return self._call(inputs)
   2716         else:
   2717             if py_any(is_tensor(x) for x in inputs):

/usr/local/lib/python3.6/dist-packages/keras/backend/tensorflow_backend.py in _call(self, inputs)
   2673             fetched = self._callable_fn(*array_vals, run_metadata=self.run_metadata)
   2674         else:
-> 2675             fetched = self._callable_fn(*array_vals)
   2676         return fetched[:len(self.outputs)]
   2677



/usr/local/lib/python3.6/dist-packages/tensorflow/python/client/session.py in __call__(self, *args, **kwargs)
   1437           ret = tf_session.TF_SessionRunCallable(
   1438               self._session._session, self._handle, args, status,
-> 1439               run_metadata_ptr)
   1440         if run_metadata:
   1441           proto_data = tf_session.TF_GetBuffer(run_metadata_ptr)

/usr/local/lib/python3.6/dist-packages/tensorflow/python/framework/errors_impl.py in __exit__(self, type_arg, value_arg, traceback_arg)
    526             None, None,
    527             compat.as_text(c_api.TF_Message(self.status.status)),
--> 528             c_api.TF_GetCode(self.status.status))
    529     # Delete the underlying status object from memory otherwise it stays alive
    530     # as there is a reference to status from this from the traceback due to

InvalidArgumentError: Incompatible shapes: [49152] vs. [16384]
         [[{{node loss_1/reshape_2_loss/mul_1}}]]
         [[{{node metrics_1/dice_coef/Mean}}]]
```

3 days ago

This Comment was deleted.