Docs » ユーティリティ

CustomObjectScope [source]

```
keras.utils.CustomObjectScope()
```

`_GLOBAL_CUSTOM_OBJECTS`をエスケープできないスコープを提供します.

`with`では，名前によってcustomオブジェクトにアクセス可能です． グローバルなcustomオブジェクトへの変更は`with`で囲まれた中でのみ持続し， `with`から抜けると，グローバルなcustomオブジェクトは`with`の最初の状態に戻ります.

例

`MyObject`というcustomオブジェクトの例です.

```
with CustomObjectScope({'MyObject':MyObject}):
    layer = Dense(..., kernel_regularizer='MyObject')
    # save, load, etc. will recognize custom object by name
```

HDF5Matrix [source]

```
keras.utils.HDF5Matrix(datapath, dataset, start=0, end=None, normalizer=None)
```

Numpy 配列の代わりに使えるHDF5 datasetの表現です.

例

```
x_data = HDF5Matrix('input/file.hdf5', 'data')
model.predict(x_data)
```

`start`と`end`を指定することでdatasetをスライスできます.

normalizer関数（やラムダ式）を渡せます．normalizer関数は取得されたすべてのスライスに適用されます.

引数

- **datapath**: 文字列，HDF5ファイルへのパス
- **dataset**: 文字列，datapathで指定されたファイル中におけるHDF5 datasetの名前
- **start**: 整数，指定されたdatasetのスライスの開始インデックス
- **end**: 整数，指定されたdatasetのスライスの終了インデックス
- **normalizer**: 読み込まれた時にデータに対して適用する関数

戻り値

array-likeなHDF5 dataset.

Sequence

[source]

```
keras.utils.Sequence()
```

datasetのようなデータの系列にfittingのためのベースオブジェクト.

Sequenceは `__getitem__` と `__len__` メソッドを実装しなければなりません. エポックの間にデータセットを変更したい場合には `on_epoch_end` を実装すべきです. `__getitem__` メソッドは完全なバッチを返すべきです.

例

```
from skimage.io import imread
from skimage.transform import resize
import numpy as np
import math

# Here, `x_set` is list of path to the images
# and `y_set` are the associated classes.

class CIFAR10Sequence(Sequence):

    def __init__(self, x_set, y_set, batch_size):
        self.x, self.y = x_set, y_set
        self.batch_size = batch_size

    def __len__(self):
        return math.ceil(len(self.x) / self.batch_size)

    def __getitem__(self, idx):
        batch_x = self.x[idx * self.batch_size:(idx + 1) * self.batch_size]
        batch_y = self.y[idx * self.batch_size:(idx + 1) * self.batch_size]

        return np.array([
            resize(imread(file_name), (200, 200))
               for file_name in batch_x]), np.array(batch_y)
```

---

to_categorical

```
keras.utils.to_categorical(y, num_classes=None)
```

整数のクラスベクトルから2値クラスの行列への変換します.

例えば, `categorical_crossentropy` のために使います.

引数

- y: 行列に変換されるクラスベクトル（0から `num_classes` までの整数）

- **num_classes**: 総クラス数

戻り値

入力のバイナリ行列表現.

---

normalize

```
keras.utils.normalize(x, axis=-1, order=2)
```

Numpy配列の正規化

引数

- **x**: 正規化するNumpy 配列.
- **axis**: 正規化する軸.
- **order**: 正規化するorder（例: L2ノルムでは2）.

戻り値

Numpy配列の正規化されたコピー.

---

get_file

```
keras.utils.get_file(fname, origin, untar=False, md5_hash=None, file_hash=None, cache_subdir
```

キャッシュ済みでなければURLからファイルをダウンロードします.

デフォルトではURL `origin` からのファイルはcache_dir `~/.keras` のcache_subdir `datasets` にダウンロードされます. これは `fname` と名付けられます. よって `example.txt` の最終的な場所は `~/.keras/datasets/example.txt` .となります.

更にファイルがtarやtar.gz，tar.bz，zipであれば展開されます. ダウンロード後にハッシュ値を渡せば検証します. コマンドラインプログラムの `shasum` や `sha256sum` がこのハッシュの計算に使えます.

引数

- **fname**: ファイル名. 絶対パス `/path/to/file.txt` を指定すればその場所に保存されます.
- **origin**: ファイルのオリジナルURL.
- **untar**: 'extract'を推奨しない. 真理値で，ファイルを展開するかどうか.
- **md5_hash**: 'file_hash'を推奨しない. ファイルの検証のためのmd5ハッシュ.

- **file_hash**: ダウンロード後に期待されるハッシュの文字列．sha256とmd5の両方のハッシュアルゴリズムがサポートされている．
- **cache_subdir**: Kerasのキャッシュディレクトリ下のどこにファイルが保存されるか．絶対パス `/path/to/folder` を指定すればその場所に保存されます
- **hash_algorithm**: ファイル検証のハッシュアルゴリズムの選択．オプションは'md5', 'sha256'または'auto'．デフォルトの'auto'は使われているハッシュアルゴリズムを推定します．
- **extract**: tarやzipのようなアーカイブとしてファイルを展開する実際の試行．
- **archive_format**: ファイルの展開に使うアーカイブフォーマット．オプションとしては'auto', 'tar', 'zip'またはNone．'tar'はtarやtar.gz，tar.bzファイルを含みます．デフォルトの'auto'は['tar', 'zip']です．Noneや空のリストでは何も合わなかったと返します．
- **cache_dir**: キャッシュファイルの保存場所で，Noneならばデフォルトで Keras Directory になります．

戻り値

ダウンロードしたファイルへのパス

---

print_summary

```
keras.utils.print_summary(model, line_length=None, positions=None, print_fn=<built-in functi
```

モデルのサマリを表示します．

引数

- **model**: Kerasのモデルインスタンス．
- **line_length**: 表示行数の合計（例えば別のターミナルウィンドウのサイズに合わせる為にセットします）．
- **positions**: 行毎のログの相対または絶対位置．指定しなければ[.33, .55, .67, 1.]の用になります．
- **print_fn**: 使うためのプリント関数．サマリの各行で呼ばれます．サマリの文字列をキャプチャするためにカスタム関数を指定することもできます．

---

plot_model

```
keras.utils.plot_model(model, to_file='model.png', show_shapes=False, show_layer_names=True,
```

Kerasモデルをdotフォーマットに変換しファイルに保存します．

引数

- **model**: Kerasのモデルインスタンス
- **to_file**: 保存するファイル名

- **show_shapes**: shapeの情報を表示するかどうか
- **show_layer_names**: レイヤー名を表示するかどうか
- **rankdir**: PyDotに渡す`rankdir`引数，プロットのフォーマットを指定する文字列：'TB' はvertical plot，'LR'はhorizontal plot.

---

multi_gpu_model

```
keras.utils.multi_gpu_model(model, gpus)
```

異なるGPUでモデルを反復します.

具体的に言えば，この関数はマルチGPUの1台のマシンでデータ並列化を実装しています.
次のような方法で動作しています.

- モデルの入力を複数のサブバッチに分割します.
- サブバッチ毎にモデルのコピーをします．どのモデルのコピーもそれぞれのGPUで実行されます.
- （CPUで）各々の結果を1つの大きなバッチとして連結させます.

例えば`batch_size`が64で`gpus=2`の場合，入力を32個のサンプルの2つのサブバッチに分け，サブバッチ毎に1つのGPUで処理され，64個の処理済みサンプルとしてバッチを返します.

8GPUまでは準線形の高速化を実現しています.

現状ではこの関数はTensorFlowバックエンドでのみ利用可能です.

引数

- model: Kerasのモデルインスタンス．このインスタンスのモデルはOOMエラーを避けるためにCPU上でビルドされるべきです（下記の使用例を参照してください）.
- gpus: 2以上の整数でGPUの個数，またはGPUのIDである整数のリスト．モデルのレプリカ作成に使われます.

返り値

初めに用いられた`model`に似たKerasの`Model`インスタンスですが，複数のGPUにワークロードが分散されたものです.

例

```
import tensorflow as tf
from keras.applications import Xception
from keras.utils import multi_gpu_model
import numpy as np

num_samples = 1000
height = 224
width = 224
num_classes = 1000

# Instantiate the base model (or "template" model).
# We recommend doing this with under a CPU device scope,
# so that the model's weights are hosted on CPU memory.
# Otherwise they may end up hosted on a GPU, which would
# complicate weight sharing.
with tf.device('/cpu:0'):
    model = Xception(weights=None,
                     input_shape=(height, width, 3),
                     classes=num_classes)

# Replicates the model on 8 GPUs.
# This assumes that your machine has 8 available GPUs.
parallel_model = multi_gpu_model(model, gpus=8)
parallel_model.compile(loss='categorical_crossentropy',
                       optimizer='rmsprop')

# Generate dummy data.
x = np.random.random((num_samples, height, width, 3))
y = np.random.random((num_samples, num_classes))

# This `fit` call will be distributed on 8 GPUs.
# Since the batch size is 256, each GPU will process 32 samples.
parallel_model.fit(x, y, epochs=20, batch_size=256)

# Save model via the template model (which shares the same weights):
model.save('my_model.h5')
```

モデルの保存

マルチGPUのモデルを保存するには，`multi_gpu_model`の返り値のモデルではなく，テンプレートになった（`multi_gpu_model`の引数として渡した）モデルで`.save(fname)`か`.save_weights(fname)`を使ってください.