Keras FAQ: Kerasに関するよくある質問

Kerasを引用するには？

Kerasがあなたの仕事の役に立ったなら，ぜひ著書のなかでKerasを引用してください．
BibTexの例は以下の通りです：

```
@misc{chollet2015keras,
  title={Keras},
  author={Chollet, Fran\c{c}ois and others},
  year={2015},
  howpublished={\url{https://keras.io}},
}
```

KerasをGPUで動かすには？

バックエンドで**TensorFlow**か**CNTK**を使っている場合，利用可能なGPUがあれば自動的にGPUが使われます．バックエンドが**Theano**の場合，以下の方法があります:

方法1: Theanoフラグを使う:

```
THEANO_FLAGS=device=gpu,floatX=float32 python my_keras_script.py
```

'gpu'の部分はデバイス識別子に合わせて変更してください（例: `gpu0`, `gpu1`など）.

方法2: `.theanorc`を使う: 使い方

方法3: コードの先頭で, `theano.config.device`, `theano.config.floatX`を手動で設定:

```
import theano
theano.config.device = 'gpu'
theano.config.floatX = 'float32'
```

KerasをマルチGPUで動かすには？

TensorFlowバックエンドの使用を推奨します. 複数のGPUで1つのモデルを実行するには**データ並列化**と**デバイス並列化**の2つの方法があります.

多くの場合，必要となるのはデータ並列化でしょう.

データ並列化

データ並列化は，ターゲットのモデルをデバイス毎に1つずつ複製することと，それぞれのレプリカを入力データ内の異なる部分の処理に用いることから成ります. Kerasには組み込みのユーティリティとして`keras.utils.multi_gpu_model`があり，どんなモデルに対してもデータ並列化バージョンを作成できて，最大8個のGPUで準線形の高速化を達成しています.

より詳細な情報はマルチGPUモデルを参照してください. 簡単な例は次の通りです：

```
from keras.utils import multi_gpu_model

# Replicates `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')

# 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)
```

デバイス並列化

デバイス並列化は同じモデルを異なるデバイスで実行することから成っています. 並列アーキテクチャを持つモデルには最適でしょう. 例としては2つのブランチを持つようなモデルがあります.

これはTensorFlowのデバイススコープを使用することで実現できます．簡単な例は次の通りです：

```
# Model where a shared LSTM is used to encode two different sequences in parallel
input_a = keras.Input(shape=(140, 256))
input_b = keras.Input(shape=(140, 256))

shared_lstm = keras.layers.LSTM(64)

# Process the first sequence on one GPU
with tf.device_scope('/gpu:0'):
    encoded_a = shared_lstm(tweet_a)
# Process the next sequence on another GPU
with tf.device_scope('/gpu:1'):
    encoded_b = shared_lstm(tweet_b)

# Concatenate results on CPU
with tf.device_scope('/cpu:0'):
    merged_vector = keras.layers.concatenate([encoded_a, encoded_b],
                                             axis=-1)
```

"sample"，"batch"，"epoch" の意味は？

Kerasを正しく使うためには，以下の定義を知り，理解しておく必要があります：

- **Sample**: データセットの一つの要素．
- 例: 一つの画像は畳み込みネットワークの一つの**sample**です
- 例: 一つの音声ファイルは音声認識モデルのための一つの**sample**です
- **Batch**: *N*のsampleのまとまり．**batch**内のサンプルは独立して並列に処理されます． 訓練中は，batchの処理結果によりモデルが1回更新されます．
- 一般的に**batch**は，それぞれの入力のみの場合に比べて，入力データのばらつきをよく近似します．batchが大きいほど，その近似は精度が良くなります．しかし，そのようなbatchの処理には時間がかかるにも関わらず更新が一度しかされません．推論（もしくは評価，予測）のためには，メモリ領域を超えなくて済む最大のbatchサイズを選ぶのをおすすめします．(なぜなら，batchが大きければ，通常は高速な評価や予測につながるからです)
- **Epoch**: "データセット全体に対する1回の処理単位"と一般的に定義されている，任意の区切りのこと．訓練のフェーズを明確に区切って，ロギングや周期的な評価するのに利用されます．
- `evaluation_data` もしくは `evaluation_split` がKeras modelの `fit` 関数とともに使われるとき，その評価は，各**epoch**が終わる度に行われます．
- Kerasでは，**epoch**の終わりに実行されるように callbacks を追加することができます．これにより例えば，学習率を変化させることやモデルのチェックポイント（保存）が行えます．

Keras modelを保存するには？

モデル全体の保存/読み込み（アーキテクチャ + 重み + オプティマイザの状態）

*Keras*のモデルを保存するのに，*pickle*や*cPickle*を使うことは推奨されません．

`model.save(filepath)`を使うことで，単一のHDF5ファイルにKerasのモデルを保存できます．このHDF5ファイルは以下を含みます．

- 再構築可能なモデルの構造
- モデルの重み
- 学習時の設定 (loss，optimizer)
- optimizerの状態．これにより，学習を終えた時点から正確に学習を再開できます

`keras.models.load_model(filepath)`によりモデルを再インスタンス化できます．`load_model`は，学習時の設定を利用して，モデルのコンパイルも行います（ただし，最初にモデルを定義した際に，一度もコンパイルされなかった場合を除く）．

例:

```
from keras.models import load_model

model.save('my_model.h5')  # creates a HDF5 file 'my_model.h5'
del model  # deletes the existing model

# returns a compiled model
# identical to the previous one
model = load_model('my_model.h5')
```

モデルのアーキテクチャのみの保存/読み込み

モデルのアーキテクチャ（weightパラメータや学習時の設定は含まない）のみを保存する場合は，以下のように行ってください:

```
# save as JSON
json_string = model.to_json()

# save as YAML
yaml_string = model.to_yaml()
```

生成されたJSON / YAMLファイルは，人が読むことができ，必要に応じて編集可能です．

保存したデータから，以下のように新しいモデルを作成できます:

```
# model reconstruction from JSON:
from keras.models import model_from_json
model = model_from_json(json_string)

# model reconstruction from YAML
from keras.models import model_from_yaml
model = model_from_yaml(yaml_string)
```

モデルの重みのみのセーブ/ロード

モデルの重みを保存する必要がある場合，以下のコードのようにHDF5を利用できます．

```
model.save_weights('my_model_weights.h5')
```

モデルのインスタンス作成後，同じアーキテクチャのモデルへ，予め保存しておいたweightパラメータをロードできます:

```
model.load_weights('my_model_weights.h5')
```

例えば，ファインチューニングや転移学習のために，異なるアーキテクチャのモデル（ただしいくつか共通のレイヤーを保持）へweightsパラメータをロードする必要がある場合，レイヤーの名前を指定することでweightsパラメータをロードできます：

```
model.load_weights('my_model_weights.h5', by_name=True)
```

`h5py` をインストールする方法についてはKerasでモデルを保存するためにHDF5やh5pyをインストールするには？も参照してください.

例:

```
"""
Assuming the original model looks like this:
    model = Sequential()
    model.add(Dense(2, input_dim=3, name='dense_1'))
    model.add(Dense(3, name='dense_2'))
    ...
    model.save_weights(fname)
"""

# new model
model = Sequential()
model.add(Dense(2, input_dim=3, name='dense_1'))  # will be loaded
model.add(Dense(10, name='new_dense'))  # will not be loaded

# load weights from first model; will only affect the first layer, dense_1.
model.load_weights(fname, by_name=True)
```

保存済みモデルでのカスタムレイヤ（またはその他カスタムオブジェクト）の取り扱い

読み込もうとしているモデルにカスタムレイヤーやその他カスタムされたクラスや関数が含まれている場合，`custom_objects` 引数を使ってロード機構にそのカスタムレイヤーなどを渡すことができます.

```
from keras.models import load_model
# Assuming your model includes instance of an "AttentionLayer" class
model = load_model('my_model.h5', custom_objects={'AttentionLayer': AttentionLayer})
```

あるいは custom object scopeを使うことも出来ます：

```
from keras.utils import CustomObjectScope

with CustomObjectScope({'AttentionLayer': AttentionLayer}):
    model = load_model('my_model.h5')
```

カスタムオブジェクトは `load_model`, `model_from_json`, `model_from_yaml` と同じように取り扱えます:

```
from keras.models import model_from_json
model = model_from_json(json_string, custom_objects={'AttentionLayer': AttentionLayer})
```

---

training lossがtesting lossよりもはるかに大きいのはなぜ？

Kerasモデルにはtrainingとtestingという2つのモードがあります．DropoutやL1/L2正則化のような，正則化手法はtestingの際には機能しません．

さらに，training lossは訓練データの各バッチのlossの平均です．モデルは変化していくため，各epochの最初のバッチの誤差は最後のバッチの誤差よりもかなり大きくなります．一方，testing lossは各epochの最後の状態のモデルを使って計算されるため，誤差が小さくなります．

中間レイヤーの出力を得るには？

シンプルな方法は，着目しているレイヤーの出力を行うための新しい `Model` を作成することです：

```
from keras.models import Model

model = ...  # create the original model

layer_name = 'my_layer'
intermediate_layer_model = Model(inputs=model.input,
                                 outputs=model.get_layer(layer_name).output)
intermediate_output = intermediate_layer_model.predict(data)
```

別の方法として，ある入力が与えられたときに，あるレイヤーの出力を返すKeras functionを以下のように記述することでも可能です：

```
from keras import backend as K

# with a Sequential model
get_3rd_layer_output = K.function([model.layers[0].input],
                                  [model.layers[3].output])
layer_output = get_3rd_layer_output([x])[0]
```

同様に，TheanoやTensorFlowのfunctionを直接利用することもできます．

ただし，学習時とテスト時でモデルの振る舞いが異なる場合(例えば `Dropout` や `BatchNormalization` の利用時など)，以下のようにlearning phaseフラグを利用してください:

```
get_3rd_layer_output = K.function([model.layers[0].input, K.learning_phase()],
                                  [model.layers[3].output])

# output in test mode = 0
layer_output = get_3rd_layer_output([x, 0])[0]

# output in train mode = 1
layer_output = get_3rd_layer_output([x, 1])[0]
```

メモリに載らない大きさのデータを扱うには？

`model.train_on_batch(x, y)` と `model.test_on_batch(x, y)` を使うことでバッチ学習ができます．詳細はモデルに関するドキュメントを見てください．

代わりに，訓練データのバッチを生成するジェネレータを記述して，`model.fit_generator(data_generator, samples_per_epoch, epochs)` の関数を使うこともできます．

実際のバッチ学習の方法については，CIFAR10 exampleを見てください．

---

validation lossが減らなくなったときに学習を中断するには？

コールバック関数の `EarlyStopping` を利用してください:

```
from keras.callbacks import EarlyStopping
early_stopping = EarlyStopping(monitor='val_loss', patience=2)
model.fit(x, y, validation_split=0.2, callbacks=[early_stopping])
```

詳細はコールバックに関するドキュメントを見てください．

---

validation splitはどのように実行されますか？

`model.fit` の引数 `validation_split` を例えば0.1に設定すると，データの最後の*10*%が検証のために利用されます．例えば，0.25に設定すると，データの最後の25%が検証に使われます．ここで，validation splitからデータを抽出する際にはデータがシャッフルされないことに注意してください．なので，検証は文字通り入力データの最後のx%のsampleに対して行われます．

（同じ `fit` 関数が呼ばれるならば）全てのepochにおいて，同じ検証用データが使われます．

訓練時にデータはシャッフルされますか？

`model.fit`の引数`shuffle`が`True`であればシャッフルされます（初期値はTrueです）．各epochで訓練データはランダムにシャッフルされます．

検証用データはシャッフルされません．

各epochのtraining/validationのlossやaccuracyを記録するには？

`model.fit`が返す`History`コールバックの`history`見てください．`history`はlossや他の指標のリストを含んでいます．

```
hist = model.fit(x, y, validation_split=0.2)
print(hist.history)
```

レイヤーを"freeze"するには？

レイヤーを"freeze"することは学習からそのレイヤーを除外することを意味します，その場合，そのレイヤーの重みは更新されなくなります．このことはモデルのファインチューニングやテキスト入力のための固定されたembeddingsを使用する際に有用です．

レイヤーのコンストラクタの`trainable`引数に真理値を渡すことで，レイヤーを訓練しないようにできます．

```
frozen_layer = Dense(32, trainable=False)
```

加えて，インスタンス化後にレイヤーの`trainable`プロパティに`True`か`False`を設定することができます．設定の有効化のためには，`trainable`プロパティの変更後のモデルで`compile()`を呼ぶ必要があります．以下にその例を示します:

```
x = Input(shape=(32,))
layer = Dense(32)
layer.trainable = False
y = layer(x)

frozen_model = Model(x, y)
# in the model below, the weights of `layer` will not be updated during training
frozen_model.compile(optimizer='rmsprop', loss='mse')

layer.trainable = True
trainable_model = Model(x, y)
# with this model the weights of the layer will be updated during training
# (which will also affect the above model since it uses the same layer instance)
trainable_model.compile(optimizer='rmsprop', loss='mse')

frozen_model.fit(data, labels)  # this does NOT update the weights of `layer`
trainable_model.fit(data, labels)  # this updates the weights of `layer`
```

---

stateful RNNを利用するには？

RNNをstatefulにするとは，各バッチのサンプルの状態が，次のバッチのサンプルのための初期状態として再利用されるということを意味します．

stateful RNNが使われるときには以下のような状態となっているはずです：

- 全てのバッチのサンプル数が同じである
- `x1` と `x2` が連続するバッチであるとき，つまり各 `i` について `x2[i]` は `x1[i]` のfollow-upシーケンスになっている

実際にstateful RNNを利用するには，以下を行う必要があります:

- `batch_size` 引数をモデルの最初のレイヤーに渡して，バッチサイズを明示的に指定してください． 例えば，サンプル数が32，タイムステップが10，特徴量の次元が16の場合には， `batch_size=32` としてください．
- RNNレイヤーで `stateful=True` を指定してください．
- fit()を呼ぶときには `shuffle=False` を指定してください．

蓄積された状態をリセットするには:

- モデルの全てのレイヤーの状態をリセットするには， `model.reset_states()` を利用してください
- 特定のstateful RNNレイヤーの状態をリセットするには， `layer.reset_states()` を利用してください

例:

```
x  # this is our input data, of shape (32, 21, 16)
# we will feed it to our model in sequences of length 10

model = Sequential()
model.add(LSTM(32, input_shape=(10, 16), batch_size=32, stateful=True))
model.add(Dense(16, activation='softmax'))

model.compile(optimizer='rmsprop', loss='categorical_crossentropy')

# we train the network to predict the 11th timestep given the first 10:
model.train_on_batch(x[:, :10, :], np.reshape(x[:, 10, :], (32, 16)))

# the state of the network has changed. We can feed the follow-up sequences:
model.train_on_batch(x[:, 10:20, :], np.reshape(x[:, 20, :], (32, 16)))

# let's reset the states of the LSTM layer:
model.reset_states()

# another way to do it in this case:
model.layers[0].reset_states()
```

`predict` , `fit` , `train_on_batch` , `predict_classes` などの関数はいずれもstatefulレイヤーの状態を更新することに注意してください．そのため，statefulな訓練だけでなく，statefulな予測も可能となります．

Sequentialモデルからレイヤーを取り除くには？

`.pop()`を使うことで，Sequentialモデルへ最後に追加したレイヤーを削除できます：

```
model = Sequential()
model.add(Dense(32, activation='relu', input_dim=784))
model.add(Dense(32, activation='relu'))

print(len(model.layers))  # "2"

model.pop()
print(len(model.layers))  # "1"
```

Kerasで事前学習したモデルを使うには？

以下の画像分類のためのモデルのコードと事前学習した重みが利用可能です：

- Xception
- VGG16
- VGG19
- ResNet50
- Inception v3
- Inception-ResNet v2
- MobileNet v1

これらのモデルは`keras.applications`からインポートできます：

```
from keras.applications.xception import Xception
from keras.applications.vgg16 import VGG16
from keras.applications.vgg19 import VGG19
from keras.applications.resnet50 import ResNet50
from keras.applications.inception_v3 import InceptionV3
from keras.applications.inception_resnet_v2 import InceptionResNetV2
from keras.applications.mobilenet import MobileNet

model = VGG16(weights='imagenet', include_top=True)
```

シンプルな使用例については， Applications moduleについてのドキュメントを見てください.

特徴量抽出やfine-tuningのために事前学習したモデルの使用例の詳細は，このブログ記事を見てください.

また，VGG16はいくつかのKerasのサンプルスクリプトの基礎になっています.

- Style transfer
- Feature visualization
- Deep dream

KerasでHDF5ファイルを入力に使うには？

`keras.utils.io_utils`から`HDF5Matrix`を使うことができます．詳細はHDF5Matrixに関するドキュメントを確認してください．

また，HDF5のデータセットを直接使うこともできます：

```
import h5py
with h5py.File('input/file.hdf5', 'r') as f:
    x_data = f['x_data']
    model.predict(x_data)
```

`h5py`をインストールする方法についてはKerasでモデルを保存するためにHDF5やh5pyをインストールするには？も参照してください．

Kerasの設定ファイルの保存場所は？

Kerasの全てのデータが格納されているデフォルトのディレクトリは以下の場所です：

```
$HOME/.keras/
```

Windowsユーザは`$HOME`を`%USERPROFILE%`に置換する必要があることに注意してください．(パーミッション等の問題によって，）Kerasが上記のディレクトリを作成できない場合には，`/tmp/.keras/`がバックアップとして使われます．

Kerasの設定ファイルはJSON形式で`$HOME/.keras/keras.json`に格納されます．デフォルトの設定ファイルは以下のようになっています：

```
{
    "image_data_format": "channels_last",
    "epsilon": 1e-07,
    "floatx": "float32",
    "backend": "tensorflow"
}
```

この設定ファイルは次のような項目を含んでいます：

- デフォルトで画像処理のレイヤーやユーティリティで使われる画像データのフォーマット（`channels_last`もしくは`channels_first`）．
- 数値演算におけるゼロ除算を防ぐために使われる，数値の微小量`epsilon`．
- デフォルトの浮動小数点数データの型．
- デフォルトのバックエンド．backendに関するドキュメントを参照してください．

同様に，`get_file()`でダウンロードされた，キャッシュ済のデータセットのファイルは，デフォルトでは`$HOME/.keras/datasets/`に格納されます．

開発中にKerasを用いて再現可能な結果を得るには？

モデルの開発中に，パフォーマンスの変化が実際のモデルやデータの変更によるものなのか，単に新しいランダムサンプルの結果によるものなのかを判断するために，実行毎に再現性のある結果を得られると便利な場合があります．以下のコードスニペットは，再現可能な結果を取得する方法の例を示しています．これは，Python 3環境のTensorFlowバックエンド向けです．

```
import numpy as np
import tensorflow as tf
import random as rn

# The below is necessary in Python 3.2.3 onwards to
# have reproducible behavior for certain hash-based operations.
# See these references for further details:
# https://docs.python.org/3.4/using/cmdline.html#envvar-PYTHONHASHSEED
# https://github.com/keras-team/keras/issues/2280#issuecomment-306959926

import os
os.environ['PYTHONHASHSEED'] = '0'

# The below is necessary for starting Numpy generated random numbers
# in a well-defined initial state.

np.random.seed(42)

# The below is necessary for starting core Python generated random numbers
# in a well-defined state.

rn.seed(12345)

# Force TensorFlow to use single thread.
# Multiple threads are a potential source of
# non-reproducible results.
# For further details, see: https://stackoverflow.com/questions/42022950/which-seeds-have-to

session_conf = tf.ConfigProto(intra_op_parallelism_threads=1, inter_op_parallelism_threads=1

from keras import backend as K

# The below tf.set_random_seed() will make random number generation
# in the TensorFlow backend have a well-defined initial state.
# For further details, see: https://www.tensorflow.org/api_docs/python/tf/set_random_seed

tf.set_random_seed(1234)

sess = tf.Session(graph=tf.get_default_graph(), config=session_conf)
K.set_session(sess)

# Rest of code follows ...
```

KerasでモデルをHDF5やh5pyをインストールするには？

KerasのモデルをHDF5ファイルとして保存する場合（例えば`keras.callbacks.ModelCheckpoint`を用いるような時），KerasではPythonパッケージのh5pyを使います．Kerasはこのパッケージと依存関係があり，デフォルトでインストールされるはずです．Debianベースのディストリビューションでは`libhdf5`のインストールも追加で必要かもしれません：

```
sudo apt-get install libhdf5-serial-dev
```

h5pyがインストールされているかわからない場合はPythonシェルを開いて次のようにもジュルーをロードできます.

```
import h5py
```

エラーなしでインポートできたらh5pyはインストールされています．そうでなければ詳細なインストール方法をこちらでご覧ください：http://docs.h5py.org/en/latest/build.html