SequentialモデルでKerasを始めてみよう

`Sequential`（系列）モデルは層を積み重ねたものです．

`Sequential`モデルはコンストラクタにレイヤーのインスタンスのリストを与えることで作れます:

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

`.add()`メソッドで簡単にレイヤーを追加できます．

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

入力のshapeを指定する

モデルはどのような入力のshapeを想定しているのかを知る必要があります． このため，`Sequential`モデルの最初のレイヤーに入力のshapeについての情報を与える必要があります（最初のレイヤー以外は入力のshapeを推定できるため，指定する必要はありません）． 入力のshapeを指定する方法はいくつかあります:

- 最初のレイヤーの`input_shape`引数を指定する．この引数にはshapeを示すタプルを与えます（このタプルの要素は整数か`None`を取ります． `None`は任意の正の整数を期待することを意味します）．
- `Dense`のような2次元の層では`input_dim`引数を指定することで入力のshapeを指定できます．同様に，3次元のレイヤーでは`input_dim`引数と`input_length`引数を指定することで入力のshapeを指定できます．
- （stateful reccurent networkなどで）バッチサイズを指定したい場合，`batch_size`引数を指定することができます．もし，`batch_size=32`と`input_shape=(6, 8)`を同時に指定した場合，想定されるバッチごとの入力のshapeは`(32, 6, 8)`となります．

このため，次のコードは等価となります．

```
model = Sequential()
model.add(Dense(32, input_shape=(784,)))
```

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

コンパイル

モデルの学習を始める前に，`compile`メソッドを用いどのような学習処理を行なうかを設定する必要があります．`compile`メソッドは3つの引数を取ります:

- 最適化アルゴリズム: 引数として，定義されている最適化手法の識別子を文字列として与える（`rmsprop`や`adagrad`など），もしくは`Optimizer`クラスのインスタンスを与えることができます． 参考: 最適化
- 損失関数: モデルが最小化しようとする目的関数です．引数として，定義されている損失関数の識別子を文字列として与える（`categorical_crossentropy`や`mse`など），もしくは目的関数を関数として与えることができます．参考：損失関数
- 評価関数のリスト: 分類問題では精度として`metrics=['accuracy']`を指定したくなるでしょう．引数として，定義されている評価関数の識別子を文字列として与える，もしくは自分で定義した関数を関数として与えることができます．

```
# マルチクラス分類問題の場合
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# 2値分類問題の場合
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# 平均二乗誤差を最小化する回帰問題の場合
model.compile(optimizer='rmsprop',
              loss='mse')

# 独自定義の評価関数を定義
import keras.backend as K

def mean_pred(y_true, y_pred):
    return K.mean(y_pred)

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy', mean_pred])
```

訓練

KerasのモデルはNumpy 配列として入力データとラベルデータから訓練します．モデルを訓練するときは，一般に`fit`関数を使います． ドキュメントはこちら.

```
# 1つの入力から2クラス分類をするモデルにおいては

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# ダミーデータの作成
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(2, size=(1000, 1))

# 各イテレーションのバッチサイズを32で学習を行なう
model.fit(data, labels, epochs=10, batch_size=32)
```

```
# 1つの入力から10クラスの分類を行なう場合について（カテゴリ分類）

model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# ダミーデータ作成
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(10, size=(1000, 1))

# ラベルデータをカテゴリの1-hotベクトルにエンコードする
one_hot_labels = keras.utils.to_categorical(labels, num_classes=10)

# 各イテレーションのバッチサイズを32で学習を行なう
model.fit(data, one_hot_labels, epochs=10, batch_size=32)
```

例

いますぐKerasを始められるようにいくつか例を用意しました！

examples folderフォルダにはリアルデータセットを利用したモデルがあります.

- CIFAR10 小規模な画像分類：リアルタイムなdata augmentationを用いたConvolutional Neural Network (CNN)
- IMDB 映画レビューのセンチメント分類：単語単位のLSTM
- Reuters 記事のトピック分類：多層パーセプトロン (MLP)
- MNIST 手書き文字認識：MLPとCNN
- LSTMを用いた文字単位の文章生成

...など

多層パーセプトロン (MLP) を用いた多値分類:

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

# ダミーデータ生成
import numpy as np
x_train = np.random.random((1000, 20))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(1000, 1)), num_classes=10)
x_test = np.random.random((100, 20))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)), num_classes=10)

model = Sequential()
# Dense(64) は，64個のhidden unitを持つ全結合層です．
# 最初のlayerでは，想定する入力データshapeを指定する必要があり，ここでは20次元としてます．
model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

MLPを用いた二値分類:

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout

# 疑似データの生成
x_train = np.random.random((1000, 20))
y_train = np.random.randint(2, size=(1000, 1))
x_test = np.random.random((100, 20))
y_test = np.random.randint(2, size=(100, 1))

model = Sequential()
model.add(Dense(64, input_dim=20, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

VGG-likeなconvnet

```
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import SGD

# 疑似データ生成
x_train = np.random.random((100, 100, 100, 3))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)), num_classes=10)
x_test = np.random.random((20, 100, 100, 3))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(20, 1)), num_classes=10)

model = Sequential()
# 入力: サイズが100x100で3チャンネルをもつ画像 -> (100, 100, 3) のテンソル
# それぞれのlayerで3x3の畳み込み処理を適用している
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(100, 100, 3)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd)

model.fit(x_train, y_train, batch_size=32, epochs=10)
score = model.evaluate(x_test, y_test, batch_size=32)
```

LSTMを用いた系列データ分類:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Embedding
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, output_dim=256))
model.add(LSTM(128))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

model.fit(x_train, y_train, batch_size=16, epochs=10)
score = model.evaluate(x_test, y_test, batch_size=16)
```

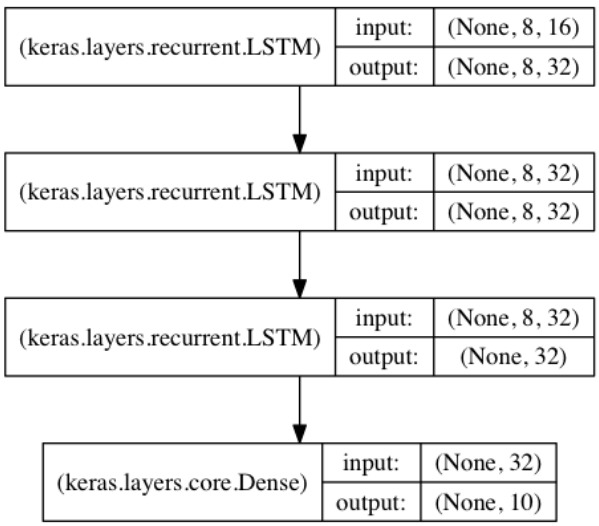
1D Convolutionを用いた系列データ分類:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Embedding
from keras.layers import Conv1D, GlobalAveragePooling1D, MaxPooling1D

model = Sequential()
model.add(Conv1D(64, 3, activation='relu', input_shape=(seq_length, 100)))
model.add(Conv1D(64, 3, activation='relu'))
model.add(MaxPooling1D(3))
model.add(Conv1D(128, 3, activation='relu'))
model.add(Conv1D(128, 3, activation='relu'))
model.add(GlobalAveragePooling1D())
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

model.fit(x_train, y_train, batch_size=16, epochs=10)
score = model.evaluate(x_test, y_test, batch_size=16)
```

Stacked LSTMを用いた系列データ分類

このモデルは，3つのLSTM layerを繋げ，高等表現を獲得できるような設計となっています．

最初の2つのLSTMは出力系列をすべて出力しています． しかし，最後のLSTMは最後のステップの状態のみを出力しており，データの次元が落ちています(入力系列を一つのベクトルにしているようなものです)．



```
from keras.models import Sequential
from keras.layers import LSTM, Dense
import numpy as np

data_dim = 16
timesteps = 8
num_classes = 10

# 想定する入力データshape: (batch_size, timesteps, data_dim)
model = Sequential()
model.add(LSTM(32, return_sequences=True,
               input_shape=(timesteps, data_dim)))  # 32次元のベクトルのsequenceを出力する
model.add(LSTM(32, return_sequences=True)) # 32次元のベクトルのsequenceを出力する
model.add(LSTM(32))  # 32次元のベクトルを一つ出力する
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

# 疑似訓練データを生成する
x_train = np.random.random((1000, timesteps, data_dim))
y_train = np.random.random((1000, num_classes))

# 疑似検証データを生成する
x_val = np.random.random((100, timesteps, data_dim))
y_val = np.random.random((100, num_classes))

model.fit(x_train, y_train,
          batch_size=64, epochs=5,
          validation_data=(x_val, y_val))
```

同じようなStacked LSTMを"stateful"にする

Stateful recurent modelは，バッチを処理し得られた内部状態を次のバッチの内部状態の初期値として再利用するモデルの一つです． このため，計算複雑度を調整できるようにしたまま，長い系列を処理することができるようになりました．

FAQにもstateful RNNsについての情報があります

```
from keras.models import Sequential
from keras.layers import LSTM, Dense
import numpy as np

data_dim = 16
timesteps = 8
num_classes = 10
batch_size = 32

# 想定している入力バッチshape: (batch_size, timesteps, data_dim)
# 注意: ネットワークがstatefulであるため，batch_input_shapewをすべてうめて与えなければなりません
# バッチkのi番目のサンプルは，バッチk-1のi番目のサンプルの次の時系列となります.
model = Sequential()
model.add(LSTM(32, return_sequences=True, stateful=True,
               batch_input_shape=(batch_size, timesteps, data_dim)))
model.add(LSTM(32, return_sequences=True, stateful=True))
model.add(LSTM(32, stateful=True))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

# 疑似訓練データを生成
x_train = np.random.random((batch_size * 10, timesteps, data_dim))
y_train = np.random.random((batch_size * 10, num_classes))

# 疑似検証データを生成
x_val = np.random.random((batch_size * 3, timesteps, data_dim))
y_val = np.random.random((batch_size * 3, num_classes))

model.fit(x_train, y_train,
          batch_size=batch_size, epochs=5, shuffle=False,
          validation_data=(x_val, y_val))
```