モデルについて

Kerasには2つの利用可能なモデルがあります: 1つはSequentialモデル，そしてもう1つは functional APIとともに用いるモデルクラス.

これらのモデルには，共通のメソッドと属性が多数あります.

- `model.layers` はモデルに含れるレイヤーを平滑化したリストです.
- `model.inputs` はモデルの入力テンソルのリストです.
- `model.outputs` はモデルの出力テンソルのリストです.
- `model.summary()` : モデルの要約を出力します．utils.print_summaryへのショートカットです.
- `model.get_config()` : モデルの設定を持つ辞書を返します．下記のように，モデルはそれ自身の設定から再インスタンス化できます.

```
config = model.get_config()
model = Model.from_config(config)
# or, for Sequential:
model = Sequential.from_config(config)
```

- `model.get_weights()` : モデルの全ての重みテンソルをNumpy 配列を要素にもつリスト返します.
- `model.set_weights(weights)` : Numpy 配列のリストからモデルの重みの値をセットします．リスト中のNumpy 配列のshapeは `get_weights()` で得られるリスト中のNumpy配列のshapeと同じ必要があります.
- `model.to_json()` : モデルの表現をJSON文字列として返します．このモデルの表現は，重みを含まないアーキテクチャのみであることに注意してください．下記の様に，JSON文字列から同じアーキテクチャのモデル（重みについては初期化されます）を再インスタンス化できます.

```
from keras.models import model_from_json

json_string = model.to_json()
model = model_from_json(json_string)
```

- `model.to_yaml()` : モデルの表現をYAML文字列として返します．このモデルの表現は，重みを含まないアーキテクチャのみであることに注意してください．下記の様に，YAML文字列から同じアーキテクチャのモデル（重みについては初期化されます）を再インスタンス化できます.

```
from keras.models import model_from_yaml

yaml_string = model.to_yaml()
model = model_from_yaml(yaml_string)
```

- `model.save_weights(filepath)` : モデルの重みをHDF5形式のファイルに保存します.

- `model.load_weights(filepath, by_name=False)`：(`save_weights`によって作られた) モデルの重みをHDF5形式のファイルから読み込みます．デフォルトでは，アーキテクチャは不変であることが望まれます．（いくつかのレイヤーが共通した）異なるアーキテクチャに重みを読み込む場合，`by_name=True`を使うことで，同名のレイヤーにのみ読み込み可能です．

注意：`h5py`のインストール方法についてはFAQの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．

モデルの派生

これらの2種類のモデルに加えて，`Model`クラスを継承して`call`メソッドで順伝播を 実装することにより独自にカスタムしたモデルを作成することができます（`Model`クラスの継承APIはKeras 2.2.0で導入されました）．

次は`Model`を継承して単純なマルチレイヤーパーセプトロンモデルを作成した例です：

```
import keras

class SimpleMLP(keras.Model):

    def __init__(self, use_bn=False, use_dp=False, num_classes=10):
        super(SimpleMLP, self).__init__(name='mlp')
        self.use_bn = use_bn
        self.use_dp = use_dp
        self.num_classes = num_classes

        self.dense1 = keras.layers.Dense(32, activation='relu')
        self.dense2 = keras.layers.Dense(num_classes, activation='softmax')
        if self.use_dp:
            self.dp = keras.layers.Dropout(0.5)
        if self.use_bn:
            self.bn = keras.layers.BatchNormalization(axis=-1)

    def call(self, inputs):
        x = self.dense1(inputs)
        if self.use_dp:
            x = self.dp(x)
        if self.use_bn:
            x = self.bn(x)
        return self.dense2(x)

model = SimpleMLP()
model.compile(...)
model.fit(...)
```

レイヤーは`__init__(self, ...)`で定義されており，順伝播は`call(self, inputs)`で記述しています．`call`では`self.add_loss(loss_tensor)`を呼ぶことで（カスタムレイヤーのように）カスタムした損失も指定できます．

静的なレイヤーのグラフと違って，派生したモデルにおいてモデルのトポロジーはPythonのコードで定義されます． これはモデルのトポロジーは検査もシリアライズもできないということです．結果として，次のメソッドや属性は**派生したモデルでは使えません**：

- `model.inputs`や`model.outputs`．

- `model.to_yaml()` や `model.to_json()`
- `model.get_config()` や `model.save()`.

キーポイント：仕事に対して適切なAPIを使ってください．`Model` の継承APIを使えば複雑なモデルの実装で優れた柔軟性を得られますが，（次のような機能も不足している上に）大きなコストも発生してしまいます：より冗長になり，より複雑になり，エラーの機会もより増えてしまうのです．可能であればfunctional APIを使った方がユーザフレンドリです．