functional APIでKerasを始めてみよう

functional APIは，複数の出力があるモデルや有向非巡回グラフ，共有レイヤーを持ったモデルなどの複雑なモデルを定義するためのインターフェースです．

ここでは`Sequential`モデルについて既に知識があることを前提として説明します．

シンプルな例から見てきましょう．

例1: 全結合ネットワーク

下記のネットワークは`Sequential`モデルによっても定義可能ですが， functional APIを使ったシンプルな例を見てきましょう．

- レイヤーのインスタンスは関数呼び出し可能で，戻り値としてテンソルを返します
- `Model`を定義することで入力と出力のテンソルは接続されます
- 上記で定義したモデルは`Sequential`と同様に利用可能です

```
from keras.layers import Input, Dense
from keras.models import Model

# This returns a tensor
inputs = Input(shape=(784,))

# a layer instance is callable on a tensor, and returns a tensor
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
predictions = Dense(10, activation='softmax')(x)

# This creates a model that includes
# the Input layer and three Dense layers
model = Model(inputs=inputs, outputs=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)  # starts training
```

全てのモデルはレイヤーと同じように関数呼び出し可能です

functional APIを利用することで，訓練済みモデルの再利用が簡単になります：全てのモデルを，テンソルを引数としたlayerのように扱うことができます．これにより，モデル構造だけでなく，モデルの重みも再利用できます．

```
x = Input(shape=(784,))
# This works, and returns the 10-way softmax we defined above.
y = model(x)
```

一連のシーケンスを処理するモデルを簡単に設計できます． 例えば画像識別モデルをたった1行で動画識別モデルに応用できます．

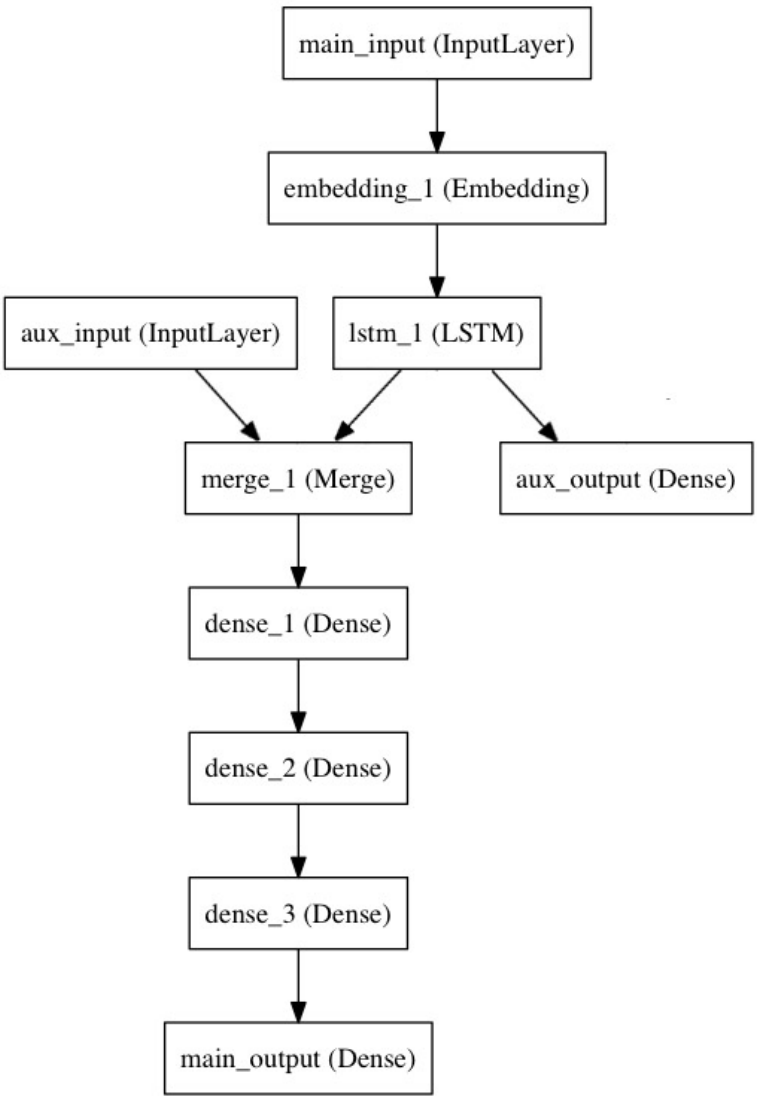
```
from keras.layers import TimeDistributed

# Input tensor for sequences of 20 timesteps,
# each containing a 784-dimensional vector
input_sequences = Input(shape=(20, 784))

# This applies our previous model to every timestep in the input sequences.
# the output of the previous model was a 10-way softmax,
# so the output of the layer below will be a sequence of 20 vectors of size 10.
processed_sequences = TimeDistributed(model)(input_sequences)
```

多入力多出力モデル

functional APIは複数の入出力を持ったモデルに最適です． 複数の複雑なデータストリームを簡単に扱うことができます．

Twitterの新しいニュースヘッドラインを受信した際，そのツイートのリツイートやライクの回数を予測する例を考えます．主な入力はヘッドラインの単語のシーケンスですが，スパイスとして，ヘッドラインの投稿時間などのデータを入力として追加します． このモデルは2つの損失関数によって訓練されます．モデルにおける初期の主損失関数を使うことは，深い層を持つモデルにとっては良い正則化の構造です．

以下がモデルの図になります．



functional APIを利用してこのネットワークを実装してみましょう.

main inputはヘッドラインを整数のシーケンス（それぞれの整数は単語をエンコードしたしたもの）として受け取ります． 整数の範囲は1から10000となり（単語数は10000語），各シーケンスは長さ100単語で構成されます.

```
from keras.layers import Input, Embedding, LSTM, Dense
from keras.models import Model

# Headline input: meant to receive sequences of 100 integers, between 1 and 10000.
# Note that we can name any layer by passing it a "name" argument.
main_input = Input(shape=(100,), dtype='int32', name='main_input')

# This embedding layer will encode the input sequence
# into a sequence of dense 512-dimensional vectors.
x = Embedding(output_dim=512, input_dim=10000, input_length=100)(main_input)

# A LSTM will transform the vector sequence into a single vector,
# containing information about the entire sequence
lstm_out = LSTM(32)(x)
```

ここでは補助損失を追加し，LSTMとEmbeddedingレイヤーをスムーズに訓練できるようにしますが，モデルでは主損失がはるかに高くなります.

```
auxiliary_output = Dense(1, activation='sigmoid', name='aux_output')(lstm_out)
```

この時点で，auxiliary_inputをLSTM出力と連結してモデルに入力します．

```
auxiliary_input = Input(shape=(5,), name='aux_input')
x = keras.layers.concatenate([lstm_out, auxiliary_input])

# We stack a deep densely-connected network on top
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)
x = Dense(64, activation='relu')(x)

# And finally we add the main logistic regression layer
main_output = Dense(1, activation='sigmoid', name='main_output')(x)
```

2つの入力と2つの出力を持ったモデルを定義します．

```
model = Model(inputs=[main_input, auxiliary_input], outputs=[main_output, auxiliary_output])
```

モデルをコンパイルし，補助損失に0.2の重み付けを行います． 様々な`loss_weights`や`loss`を対応付けるためにリストもしくは辞書を利用します． `loss`に1つの損失関数を与えた場合，全ての出力に対して同一の損失関数が適用されます．

```
model.compile(optimizer='rmsprop', loss='binary_crossentropy',
              loss_weights=[1., 0.2])
```

モデルに入力と教師データをリストで渡すことで訓練できます．

```
model.fit([headline_data, additional_data], [labels, labels],
          epochs=50, batch_size=32)
```

入力と出力に名前付けを行っていれば（"name"引数を利用），下記のような方法でモデルをコンパイルできます．

```
model.compile(optimizer='rmsprop',
              loss={'main_output': 'binary_crossentropy', 'aux_output': 'binary_crossentropy
              loss_weights={'main_output': 1., 'aux_output': 0.2})

# And trained it via:
model.fit({'main_input': headline_data, 'aux_input': additional_data},
          {'main_output': labels, 'aux_output': labels},
          epochs=50, batch_size=32)
```

共有レイヤー

その他のfunctional APIの利用例として，共有レイヤーがあります． 共有レイヤーについて考えてみましょう．

ツイートのデータセットの例を考えてみましょう．2つのツイートが同じ人物からつぶやかれたかどうかを判定するモデルを作りたいとします． （例えばこれによりユーザーの類似度を比較できます）

これを実現する一つの方法として，2つのツイートを2つのベクトルにエンコードし，それらをマージした後，ロジスティクス回帰を行うことで，その2つのツイートが同じ人物から投稿されたかどうかの確率を出力できます． このモデルはポジティブなツイートのペアとネガティブなツイートのペアを用いて訓練できます．

問題はシンメトリックであるため，1つめのツイートのエンコードメカニズムは2つめのツイートのエンコード時に再利用出来ます． ここではLSTMの共有レイヤーによりツイートをエンコードします．

functional APIでこのモデルを作成してみましょう． 入力として`(280, 256)`のバイナリー行列をとります． サイズが256の280個のシーケンスで，256次元のベクトルの各次元は文字（アルファベット以外も含めた256文字の出現頻度の高いもの）の有無を表します．

```
import keras
from keras.layers import Input, LSTM, Dense
from keras.models import Model

tweet_a = Input(shape=(280, 256))
tweet_b = Input(shape=(280, 256))
```

それぞれの入力間でレイヤーを共有するために，1つのレイヤーを生成し，そのレイヤーを用いて複数の入力を処理します．

```
# This layer can take as input a matrix
# and will return a vector of size 64
shared_lstm = LSTM(64)

# When we reuse the same layer instance
# multiple times, the weights of the layer
# are also being reused
# (it is effectively *the same* layer)
encoded_a = shared_lstm(tweet_a)
encoded_b = shared_lstm(tweet_b)

# We can then concatenate the two vectors:
merged_vector = keras.layers.concatenate([encoded_a, encoded_b], axis=-1)

# And add a logistic regression on top
predictions = Dense(1, activation='sigmoid')(merged_vector)

# We define a trainable model linking the
# tweet inputs to the predictions
model = Model(inputs=[tweet_a, tweet_b], outputs=predictions)

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
model.fit([data_a, data_b], labels, epochs=10)
```

共有レイヤーの出力や出力のshapeを見てみましょう．

"ノード"の概念

ある入力を用いてレイヤーを関数呼び出しするときは常に新しいテンソル（レイヤーの出力）を生成しており，レイヤーにノードを追加すると入力のテンソルと出力のテンソルはリンクされます． 同じレイヤーを複数回呼び出す際，そのレイヤーは0, 1, 2...とインデックスされた複数のノードを所有することになります．

以前のバージョンのKerasでは，`layer.get_output()`によって出力のテンソルを取得でき，`layer.output_shape`によって形を取得できました． もちろん現在のバージョンでもこれらは利用可能です(`get_output()`は`output`というプロパティーに変更されました）． しかし複数の入力が接続されているレイヤーはどうしたらよいでしょうか？

1つのレイヤーに1つの入力しかない場合は問題はなく`.output`がレイヤーが単一の出力を返すでしょう．

```
a = Input(shape=(280, 256))

lstm = LSTM(32)
encoded_a = lstm(a)

assert lstm.output == encoded_a
```

複数の入力がある場合はそうはなりません．

```
a = Input(shape=(280, 256))
b = Input(shape=(280, 256))

lstm = LSTM(32)
encoded_a = lstm(a)
encoded_b = lstm(b)

lstm.output
```

```
>> AttributeError: Layer lstm_1 has multiple inbound nodes,
hence the notion of "layer output" is ill-defined.
Use `get_output_at(node_index)` instead.
```

下記は正常に動作します．

```
assert lstm.get_output_at(0) == encoded_a
assert lstm.get_output_at(1) == encoded_b
```

シンプルですね．

`input_shape`と`output_shape`についても同じことが言えます． レイヤーが1つのノードしか持っていない，もしくは全てのノードが同じ入出力のshapeであれば，レイヤーの入出力のshapeが一意に定まり，`layer.output_shape`/`layer.input_shape`によって1つのshapeを返します．しかしながら，1つの`Conv2D`レイヤーに`(32, 32, 3)`の入力と`(64, 64, 32)`の入力を行った場合，そのレイヤーは複数のinput/output shapeを持つことになるため，それぞれのshapeはノードのインデックスを指定することで取得できます．

```
a = Input(shape=(32, 32, 3))
b = Input(shape=(64, 64, 3))

conv = Conv2D(16, (3, 3), padding='same')
conved_a = conv(a)

# Only one input so far, the following will work:
assert conv.input_shape == (None, 32, 32, 3)

conved_b = conv(b)
# now the `.input_shape` property wouldn't work, but this does:
assert conv.get_input_shape_at(0) == (None, 32, 32, 3)
assert conv.get_input_shape_at(1) == (None, 64, 64, 3)
```

その他の例

コード例から学び始めることは最良の手法です. その他の例も見てみましょう.

Inception module

Inceptionモデルについての詳細はGoing Deeper with Convolutionsを参照.

```
from keras.layers import Conv2D, MaxPooling2D, Input

input_img = Input(shape=(256, 256, 3))

tower_1 = Conv2D(64, (1, 1), padding='same', activation='relu')(input_img)
tower_1 = Conv2D(64, (3, 3), padding='same', activation='relu')(tower_1)

tower_2 = Conv2D(64, (1, 1), padding='same', activation='relu')(input_img)
tower_2 = Conv2D(64, (5, 5), padding='same', activation='relu')(tower_2)

tower_3 = MaxPooling2D((3, 3), strides=(1, 1), padding='same')(input_img)
tower_3 = Conv2D(64, (1, 1), padding='same', activation='relu')(tower_3)

output = keras.layers.concatenate([tower_1, tower_2, tower_3], axis=1)
```

Residual connection on a convolution layer

Residual networksモデルについての詳細はDeep Residual Learning for Image Recognitionを参照してください.

```
from keras.layers import Conv2D, Input

# input tensor for a 3-channel 256x256 image
x = Input(shape=(256, 256, 3))
# 3x3 conv with 3 output channels (same as input channels)
y = Conv2D(3, (3, 3), padding='same')(x)
# this returns x + y.
z = keras.layers.add([x, y])
```

Shared vision model

このモデルでは，2つのMNISTの数字が同じものかどうかを識別するために，同じ画像処理のモジュールを2つの入力で再利用しています.

```
from keras.layers import Conv2D, MaxPooling2D, Input, Dense, Flatten
from keras.models import Model

# First, define the vision modules
from keras.layers import Conv2D, MaxPooling2D, Input, Dense, Flatten
from keras.models import Model

# First, define the vision modules
digit_input = Input(shape=(27, 27, 1))
x = Conv2D(64, (3, 3))(digit_input)
x = Conv2D(64, (3, 3))(x)
x = MaxPooling2D((2, 2))(x)
out = Flatten()(x)

vision_model = Model(digit_input, out)

# Then define the tell-digits-apart model
digit_a = Input(shape=(27, 27, 1))
digit_b = Input(shape=(27, 27, 1))

# The vision model will be shared, weights and all
out_a = vision_model(digit_a)
out_b = vision_model(digit_b)

concatenated = keras.layers.concatenate([out_a, out_b])
out = Dense(1, activation='sigmoid')(concatenated)

classification_model = Model([digit_a, digit_b], out)
```

Visual question answering model

このモデルは写真に対する自然言語の質問に対して1単語の解答を選択できます.

質問と画像をそれぞれベクトルにエンコードし，それらを1つに結合して，解答となる語彙を正解データとしたロジスティック回帰を訓練させることで実現できます.

```
from keras.layers import Conv2D, MaxPooling2D, Flatten
from keras.layers import Input, LSTM, Embedding, Dense
from keras.models import Model, Sequential

# First, let's define a vision model using a Sequential model.
# This model will encode an image into a vector.
vision_model = Sequential()
vision_model.add(Conv2D(64, (3, 3), activation='relu', padding='same', input_shape=(224, 224
vision_model.add(Conv2D(64, (3, 3), activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))
vision_model.add(Conv2D(128, (3, 3), activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Conv2D(256, (3, 3), activation='relu', padding='same'))
vision_model.add(Conv2D(256, (3, 3), activation='relu'))
vision_model.add(Conv2D(256, (3, 3), activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Flatten())

# Now let's get a tensor with the output of our vision model:
image_input = Input(shape=(224, 224, 3))
encoded_image = vision_model(image_input)

# Next, let's define a language model to encode the question into a vector.
# Each question will be at most 100 word long,
# and we will index words as integers from 1 to 9999.
question_input = Input(shape=(100,), dtype='int32')
embedded_question = Embedding(input_dim=10000, output_dim=256, input_length=100)(question_in
encoded_question = LSTM(256)(embedded_question)

# Let's concatenate the question vector and the image vector:
merged = keras.layers.concatenate([encoded_question, encoded_image])

# And let's train a logistic regression over 1000 words on top:
output = Dense(1000, activation='softmax')(merged)

# This is our final model:
vqa_model = Model(inputs=[image_input, question_input], outputs=output)

# The next stage would be training this model on actual data.
```

Video question answering model

画像のQAモデルを訓練したので，そのモデルを応用して動画のQA modelを作成してみましょう． 適切な訓練を行うことで，短い動画や（例えば，100フレームの人物行動）や動画を用いた自然言語のQAへ応用することができます（例えば，「その少年は何のスポーツをしていますか？」「サッカーです」）．

```
from keras.layers import TimeDistributed

video_input = Input(shape=(100, 224, 224, 3))
# This is our video encoded via the previously trained vision_model (weights are reused)
encoded_frame_sequence = TimeDistributed(vision_model)(video_input)  # the output will be a
encoded_video = LSTM(256)(encoded_frame_sequence)  # the output will be a vector

# This is a model-level representation of the question encoder, reusing the same weights as
question_encoder = Model(inputs=question_input, outputs=encoded_question)

# Let's use it to encode the question:
video_question_input = Input(shape=(100,), dtype='int32')
encoded_video_question = question_encoder(video_question_input)

# And this is our video question answering model:
merged = keras.layers.concatenate([encoded_video, encoded_video_question])
output = Dense(1000, activation='softmax')(merged)
video_qa_model = Model(inputs=[video_input, video_question_input], outputs=output)
```