Getting started with the Keras functional API

The Keras functional API is the way to go for defining complex models, such as multi-output models, directed acyclic graphs, or models with shared layers.

This guide assumes that you are already familiar with the `Sequential` model.

Let's start with something simple.

First example: fully connected network

The `Sequential` model is probably a better choice to implement such a network, but it helps to start with something really simple.

- A layer instance is callable (on a tensor), and it returns a tensor
- Input tensor(s) and output tensor(s) can then be used to define a `Model`
- Such a model can be trained just like Keras `Sequential` models.

```
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(input=inputs, output=predictions)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(data, labels)  # starts training
```

All models are callable, just like layers

With the functional API, it is easy to re-use trained models: you can treat any model as if it were a layer, by calling it on a tensor. Note that by calling a model you aren't just re-using the *architecture* of the model, you are also re-using its weights.

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

This can allow, for instance, to quickly create models that can process *sequences* of inputs. You could turn an image classification model into a video classification model, in just one line.

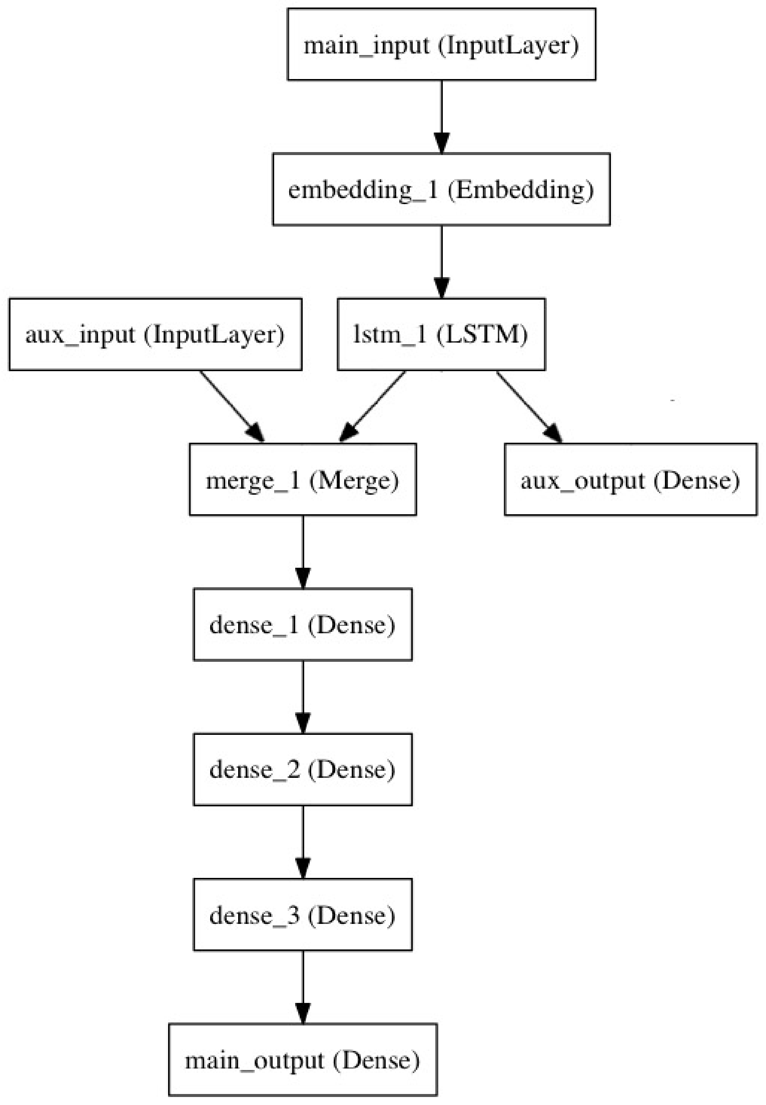
```
from keras.layers import TimeDistributed

# 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)
```

Multi-input and multi-output models

Here's a good use case for the functional API: models with multiple inputs and outputs. The functional API makes it easy to manipulate a large number of intertwined datastreams.

Let's consider the following model. We seek to predict how many retweets and likes a news headline will receive on Twitter. The main input to the model will be the headline itself, as a sequence of words, but to spice things up, our model will also have an auxiliary input, receiving extra data such as the time of day when the headline was posted, etc. The model will also be supervised via two loss functions. Using the main loss function earlier in a model is a good regularization mechanism for deep models.

Here's what our model looks like:



Let's implement it with the functional API.

The main input will receive the headline, as a sequence of integers (each integer encodes a word). The integers will be between 1 and 10,000 (a vocabulary of 10,000 words) and the sequences will be 100 words long.

```
from keras.layers import Input, Embedding, LSTM, Dense, merge
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)
```

Here we insert the auxiliary loss, allowing the LSTM and Embedding layer to be trained smoothly even though the main loss will be much higher in the model.

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

At this point, we feed into the model our auxiliary input data by concatenating it with the LSTM output:

```
auxiliary_input = Input(shape=(5,), name='aux_input')
x = merge([lstm_out, auxiliary_input], mode='concat')

# we stack a deep fully-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_loss = Dense(1, activation='sigmoid', name='main_output')(x)
```

This defines a model with two inputs and two outputs:

```
model = Model(input=[main_input, auxiliary_input], output=[main_loss, auxiliary_loss])
```

We compile the model and assign a weight of 0.2 to the auxiliary loss. To specify different `loss_weights` or `loss` for each different output, you can use a list or a dictionary. Here we pass a single loss as the `loss` argument, so the same loss will be used on all outputs.

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

We can train the model by passing it lists of input arrays and target arrays:

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

Since our inputs and outputs are named (we passed them a "name" argument), We could also have compiled the model via:

```
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},
          nb_epoch=50, batch_size=32)
```

Shared layers

Another good use for the functional API are models that use shared layers. Let's take a look at shared layers.

Let's consider a dataset of tweets. We want to build a model that can tell whether two tweets are from the same person or not (this can allow us to compare users by the similarity of their tweets, for instance).

One way to achieve this is to build a model that encodes two tweets into two vectors, concatenates the vectors and adds a logistic regression of top, outputting a probability that the two tweets share the same author. The model would then be trained on positive tweet pairs and negative tweet pairs.

Because the problem is symmetric, the mechanism that encodes the first tweet should be reused (weights and all) to encode the second tweet. Here we use a shared LSTM layer to encode the tweets.

Let's build this with the functional API. We will take as input for a tweet a binary matrix of shape `(140, 256)`, i.e. a sequence of 140 vectors of size 256, where each dimension in the 256-dimensional vector encodes the presence/absence of a character (out of an alphabet of 256 frequent characters).

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

tweet_a = Input(shape=(140, 256))
tweet_b = Input(shape=(140, 256))
```

To share a layer across different inputs, simply instantiate the layer once, then call it on as many inputs as you want:

```
# 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 = merge([encoded_a, encoded_b], mode='concat', concat_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(input=[tweet_a, tweet_b], output=predictions)

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

Let's pause to take a look at how to read the shared layer's output or output shape.

The concept of layer "node"

Whenever you are calling a layer on some input, you are creating a new tensor (the output of the layer), and you are adding a "node" to the layer, linking the input tensor to the output tensor. When you are calling the same layer multiple times, that layer owns multiple nodes indexed as 0, 1, 2...

In previous versions of Keras, you could obtain the output tensor of a layer instance via `layer.get_output()`, or its output shape via `layer.output_shape`. You still can (except `get_output()` has been replaced by the property `output`). But what if a layer is connected to multiple inputs?

As long as a layer is only connected to one input, there is no confusion, and `.output` will return the one output of the layer:

```
a = Input(shape=(140, 256))

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

assert lstm.output == encoded_a
```

Not so if the layer has multiple inputs:

```
a = Input(shape=(140, 256))
b = Input(shape=(140, 256))

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

lstm.output
```

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

Okay then. The following works:

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

Simple enough, right?

The same is true for the properties `input_shape` and `output_shape`: as long as the layer has only one node, or as long as all nodes have the same input/output shape, then the notion of "layer output/input shape" is well defined, and that one shape will be returned by `layer.output_shape`/`layer.input_shape`. But if, for instance, you apply a same `Convolution2D` layer to an input of shape `(3, 32, 32)`, and then to an input of shape `(3, 64, 64)`, the layer will have multiple input/output shapes, and you will have to fetch them by specifying the index of the node they belong to:

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

conv = Convolution2D(16, 3, 3, border_mode='same')
conved_a = conv(a)

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

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

More examples

Code examples are still the best way to get started, so here are a few more.

Inception module

For more information about the Inception architecture, see Going Deeper with Convolutions.

```
from keras.layers import merge, Convolution2D, MaxPooling2D, Input

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

tower_1 = Convolution2D(64, 1, 1, border_mode='same', activation='relu')(input_img)
tower_1 = Convolution2D(64, 3, 3, border_mode='same', activation='relu')(tower_1)

tower_2 = Convolution2D(64, 1, 1, border_mode='same', activation='relu')(input_img)
tower_2 = Convolution2D(64, 5, 5, border_mode='same', activation='relu')(tower_2)

tower_3 = MaxPooling2D((3, 3), strides=(1, 1), border_mode='same')(input_img)
tower_3 = Convolution2D(64, 1, 1, border_mode='same', activation='relu')(tower_3)

output = merge([tower_1, tower_2, tower_3], mode='concat', concat_axis=1)
```

Residual connection on a convolution layer

For more information about residual networks, see Deep Residual Learning for Image Recognition.

```
from keras.layers import merge, Convolution2D, Input

# input tensor for a 3-channel 256x256 image
x = Input(shape=(3, 256, 256))
# 3x3 conv with 3 output channels (same as input channels)
y = Convolution2D(3, 3, 3, border_mode='same')(x)
# this returns x + y.
z = merge([x, y], mode='sum')
```

Shared vision model

This model re-uses the same image-processing module on two inputs, to classify whether two MNIST digits are the same digit or different digits.

```
from keras.layers import merge, Convolution2D, MaxPooling2D, Input, Dense, Flatten
from keras.models import Model

# first, define the vision modules
digit_input = Input(shape=(1, 27, 27))
x = Convolution2D(64, 3, 3)(digit_input)
x = Convolution2D(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=(1, 27, 27))
digit_b = Input(shape=(1, 27, 27))

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

concatenated = merge([out_a, out_b], mode='concat')
out = Dense(1, activation='sigmoid')(concatenated)

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

Visual question answering model

This model can select the correct one-word answer when asked a natural-language question about a picture.

It works by encoding the question into a vector, encoding the image into a vector, concatenating the two, and training on top a logistic regression over some vocabulary of potential answers.

```
from keras.layers import Convolution2D, MaxPooling2D, Flatten
from keras.layers import Input, LSTM, Embedding, Dense, merge
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(Convolution2D(64, 3, 3, activation='relu', border_mode='same', input_shape=(3,
vision_model.add(Convolution2D(64, 3, 3, activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Convolution2D(128, 3, 3, activation='relu', border_mode='same'))
vision_model.add(Convolution2D(128, 3, 3, activation='relu'))
vision_model.add(MaxPooling2D((2, 2)))
vision_model.add(Convolution2D(256, 3, 3, activation='relu', border_mode='same'))
vision_model.add(Convolution2D(256, 3, 3, activation='relu'))
vision_model.add(Convolution2D(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=(3, 224, 224))
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_input
encoded_question = LSTM(256)(embedded_question)

# let's concatenate the question vector and the image vector:
merged = merge([encoded_question, encoded_image], mode='concat')

# 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(input=[image_input, question_input], output=output)

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

Video question answering model

Now that we have trained our image QA model, we can quickly turn it into a video QA model. With appropriate training, you will be able to show it a short video (e.g. 100-frame human action) and ask a natural language question about the video (e.g. "what sport is the boy playing?" -> "football").

```
from keras.layers import TimeDistributed

video_input = Input(shape=(100, 3, 224, 224))
# 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 seq
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 bef
question_encoder = Model(input=question_input, output=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 = merge([encoded_video, encoded_video_question], mode='concat')
output = Dense(1000, activation='softmax')(merged)
video_qa_model = Model(input=[video_input, video_question_input], output=output)
```