CSCE 636 Neural Networks (Deep Learning)

Lecture 10: Advanced deep-learning best practices

Anxiao (Andrew) Jiang

Keras Functional API

Sequential Model

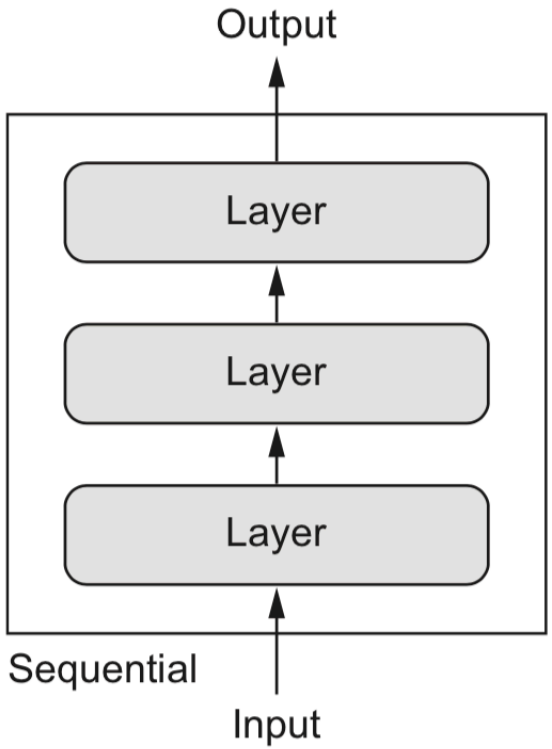


Figure 7.1 A `Sequential` model: a linear stack of layers

But in general, the neural network model can be any directed acyclic graph (DAC)

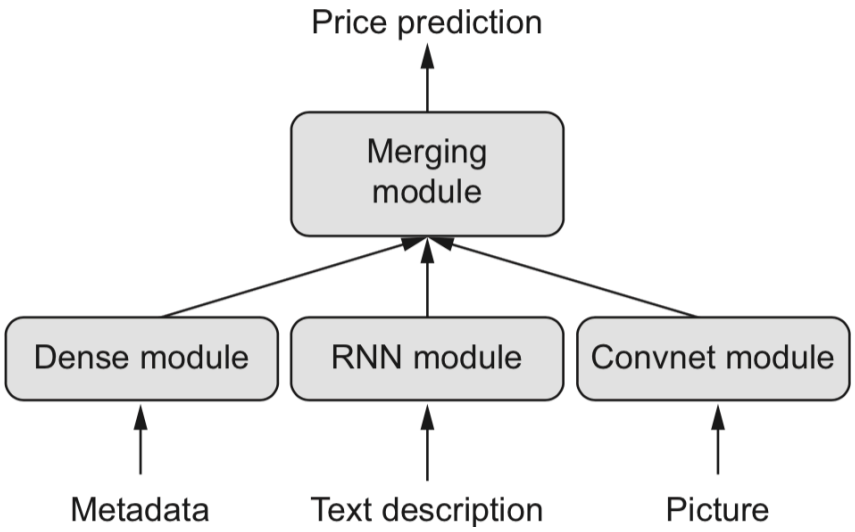


Figure 7.2 A multi-input model

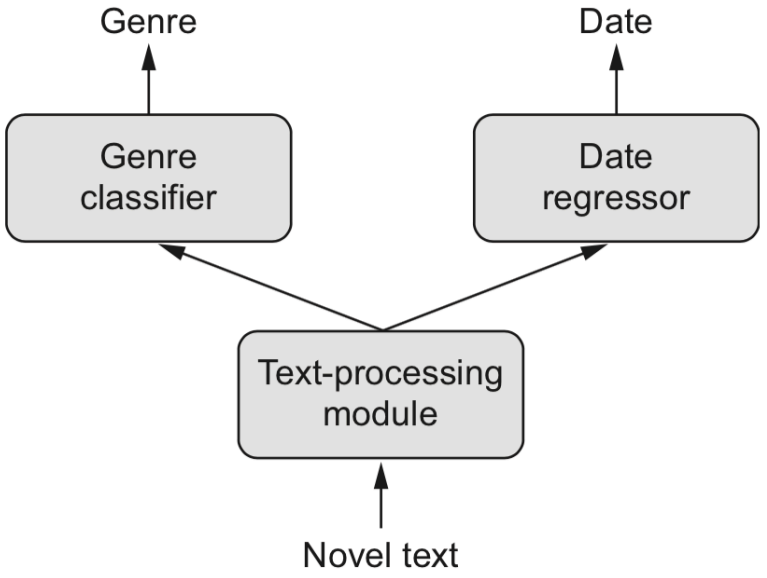


Figure 7.3 A multi-output (or multihead) model

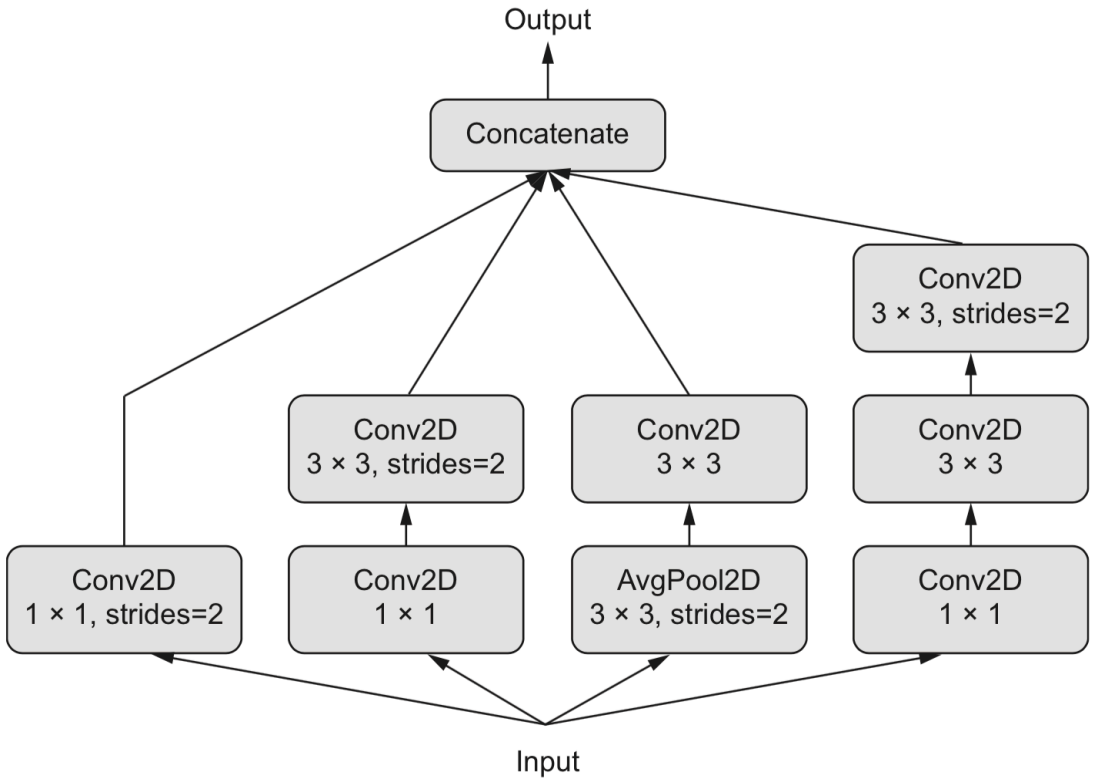
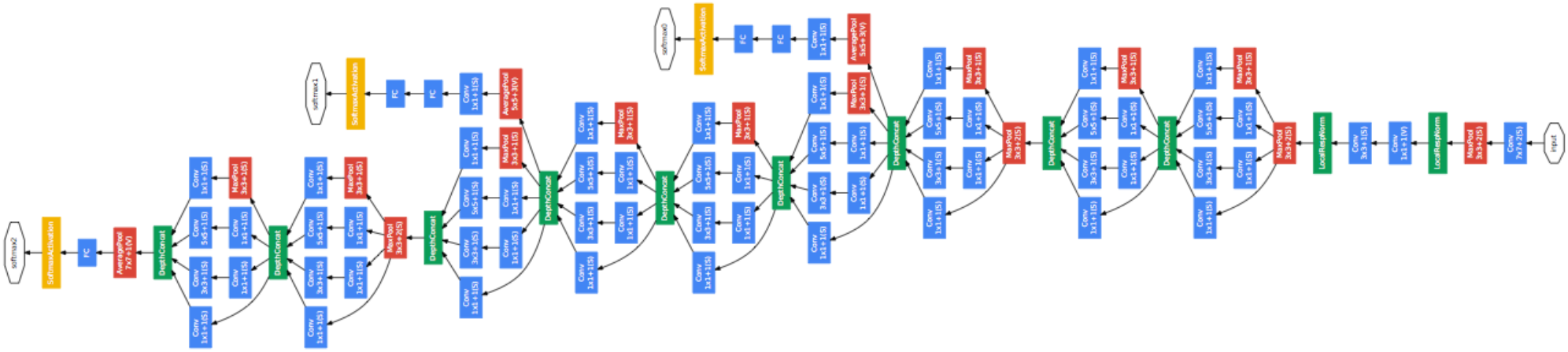


Figure 7.4 An Inception module: a subgraph of layers with several parallel convolutional branches

Google “Inception” deep neural network architecture



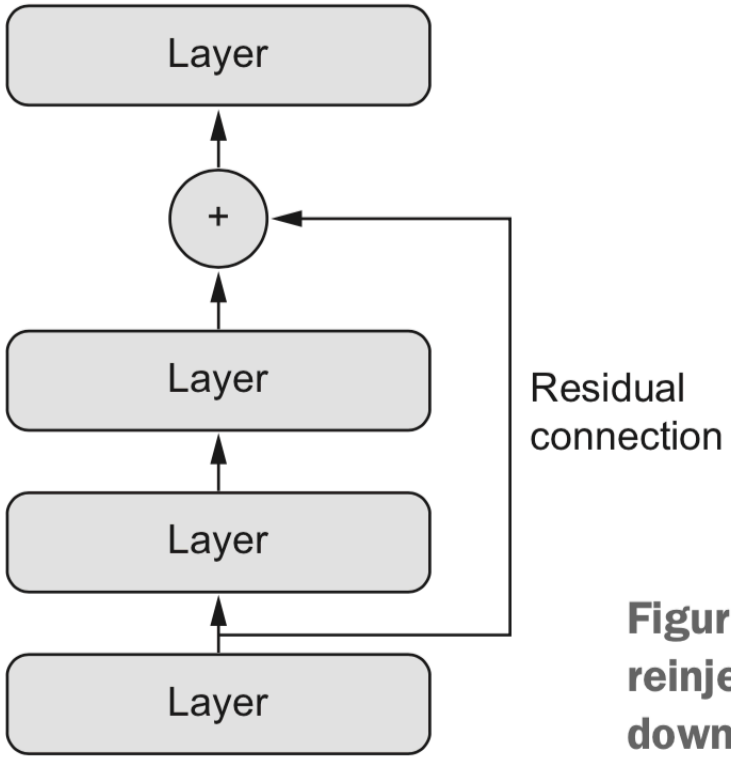
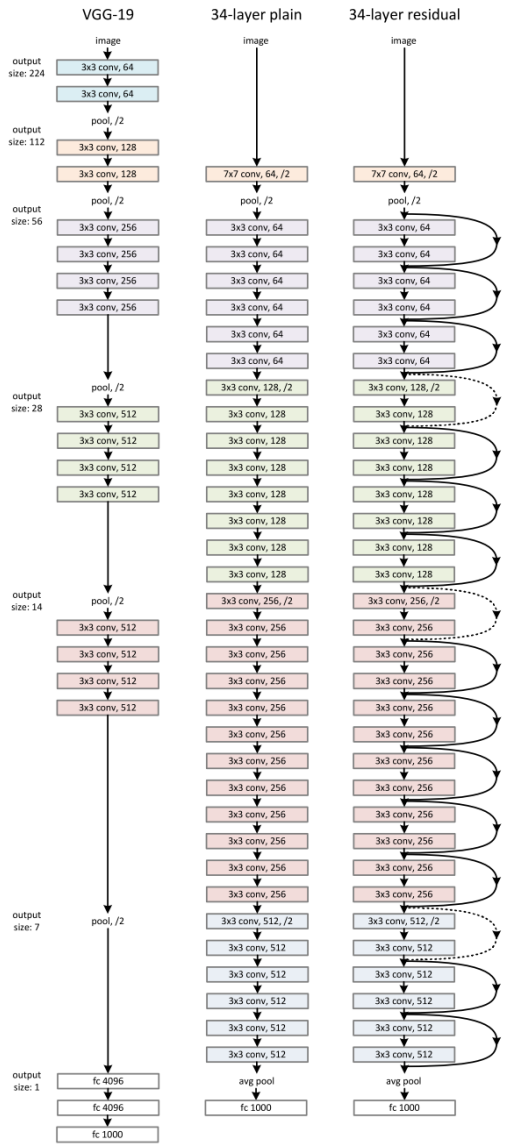


Figure 7.5 A residual connection: reinjection of prior information downstream via feature-map addition


VGG-19
34-layer plain
34-layer residual
image
output size: 224
3x3 conv, 64
3x3 conv, 64
pool, /2
output size: 112
3x3 conv, 128
3x3 conv, 128
pool, /2
output size: 56
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
pool, /2
output size: 28
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 14
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 7
fc 4096
fc 4096
fc 1000
output size: 1
7x7 conv, 64, /2
pool, /2
3x3 conv, 64
3x3 conv, 128, /2
3x3 conv, 128
3x3 conv, 256, /2
3x3 conv, 256
3x3 conv, 512, /2
3x3 conv, 512
avg pool
fc 1000

ResNet

Examples of functional API

- See every layer as a function y = f(x)
- Every input and output has a name, which makes it feasible to define the model graph conveniently.

```
from keras import Input, layers

input_tensor = Input(shape=(32,))

dense = layers.Dense(32, activation='relu')

output_tensor = dense(input_tensor)
```

A tensor

A layer is a function.

A layer may be called on a tensor, and it returns a tensor.

```
from keras import Input, layers

input_tensor = Input(shape=(32,))

dense = layers.Dense(32, activation='relu')

output_tensor = dense(input_tensor)
```

A tensor

A layer is a function.

A layer may be called on a tensor, and it returns a tensor.

input_tensor

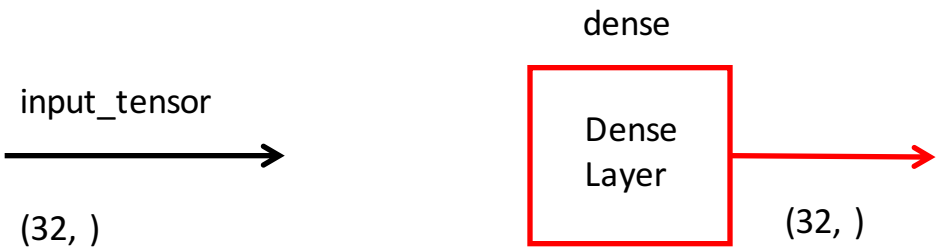
(32,)

```
from keras import Input, layers

input_tensor = Input(shape=(32,))
dense = layers.Dense(32, activation='relu')
output_tensor = dense(input_tensor)
```

A tensor

A layer is a function.

A layer may be called on a tensor, and it returns a tensor.

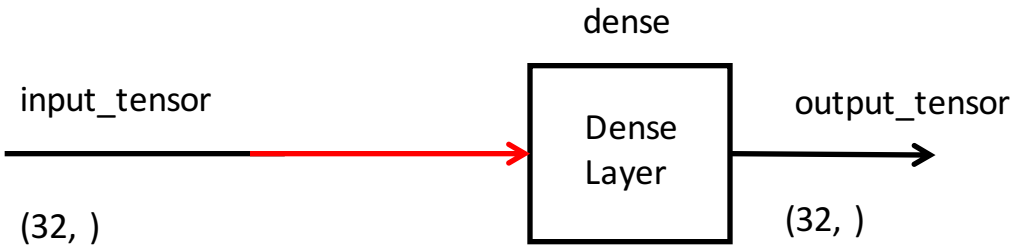


```
from keras import Input, layers

input_tensor = Input(shape=(32,))

dense = layers.Dense(32, activation='relu')

output_tensor = dense(input_tensor)
```

A tensor

A layer is a function.

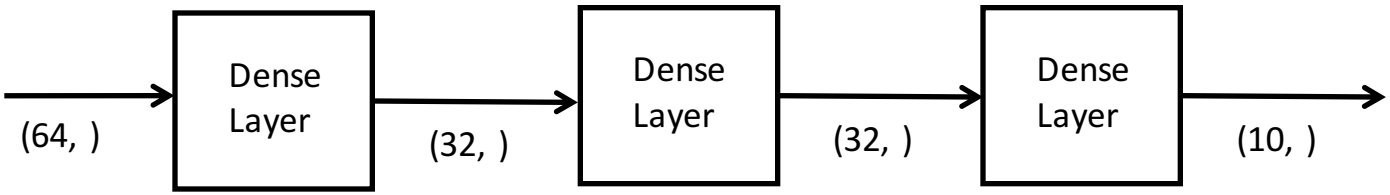
A layer may be called on a tensor, and it returns a tensor.



Another example

Sequential model

```
seq_model = Sequential()
seq_model.add(layers.Dense(32, activation='relu', input_shape=(64,)))
seq_model.add(layers.Dense(32, activation='relu'))
seq_model.add(layers.Dense(10, activation='softmax'))
```



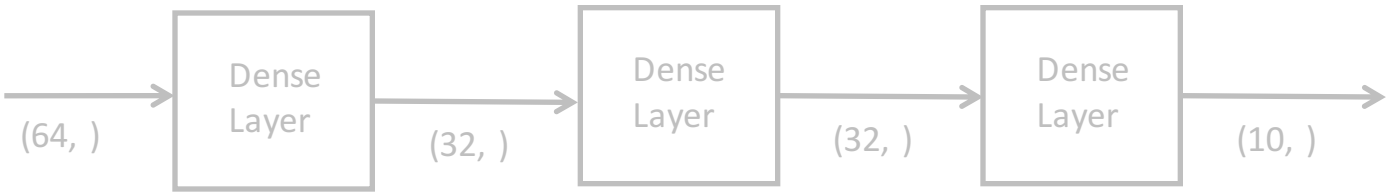
Name of model:
seq_model

Sequential model

```
seq_model = Sequential()
seq_model.add(layers.Dense(32, activation='relu', input_shape=(64,)))
seq_model.add(layers.Dense(32, activation='relu'))
seq_model.add(layers.Dense(10, activation='softmax'))
```

Equivalent functional API

```
input_tensor = Input(shape=(64,))
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x)
model = Model(input_tensor, output_tensor)
```

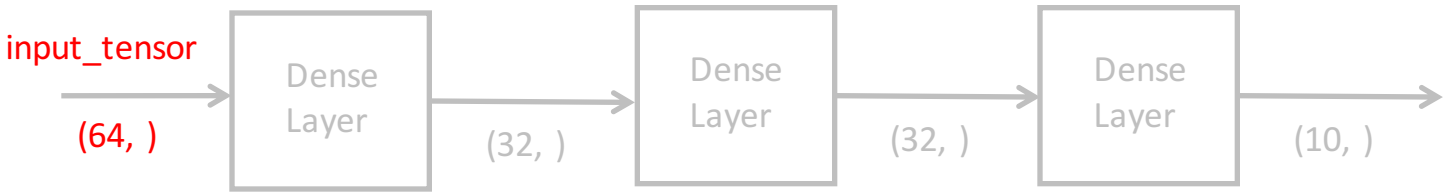


Sequential model

```
seq_model = Sequential()
seq_model.add(layers.Dense(32, activation='relu', input_shape=(64,)))
seq_model.add(layers.Dense(32, activation='relu'))
seq_model.add(layers.Dense(10, activation='softmax'))
```

Equivalent functional API

```
input_tensor = Input(shape=(64,))
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x)
model = Model(input_tensor, output_tensor)
```

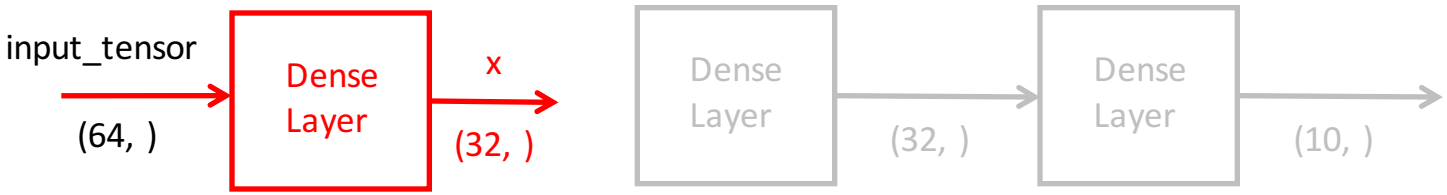


Sequential model

```
seq_model = Sequential()
seq_model.add(layers.Dense(32, activation='relu', input_shape=(64,)))
seq_model.add(layers.Dense(32, activation='relu'))
seq_model.add(layers.Dense(10, activation='softmax'))
```

Equivalent functional API

```
input_tensor = Input(shape=(64,))
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x)
model = Model(input_tensor, output_tensor)
```

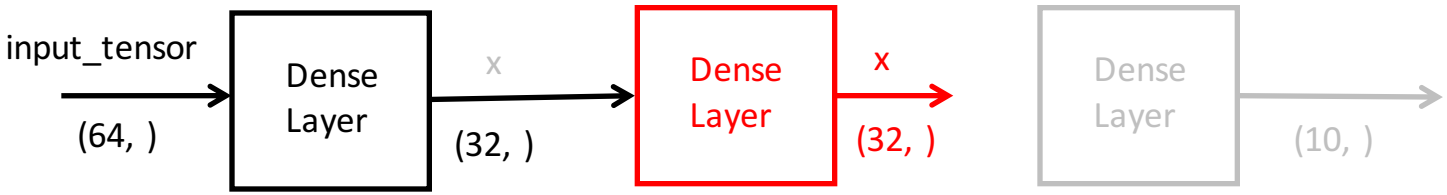


Sequential model

```
seq_model = Sequential()
seq_model.add(layers.Dense(32, activation='relu', input_shape=(64,)))
seq_model.add(layers.Dense(32, activation='relu'))
seq_model.add(layers.Dense(10, activation='softmax'))
```

Equivalent functional API

```
input_tensor = Input(shape=(64,))
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x)
model = Model(input_tensor, output_tensor)
```

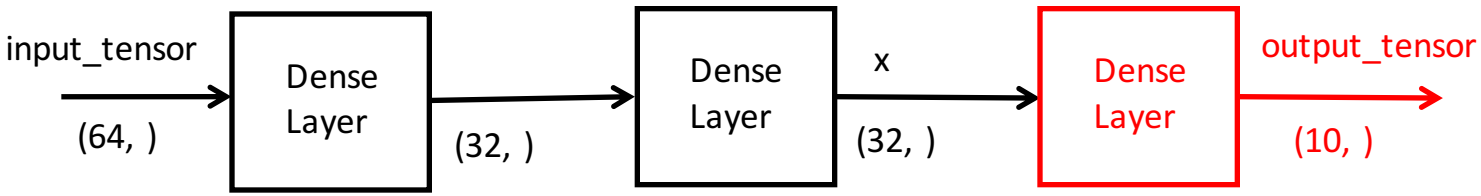


Sequential model

```
seq_model = Sequential()
seq_model.add(layers.Dense(32, activation='relu', input_shape=(64,)))
seq_model.add(layers.Dense(32, activation='relu'))
seq_model.add(layers.Dense(10, activation='softmax'))
```

Equivalent functional API

```
input_tensor = Input(shape=(64,))
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x)
model = Model(input_tensor, output_tensor)
```

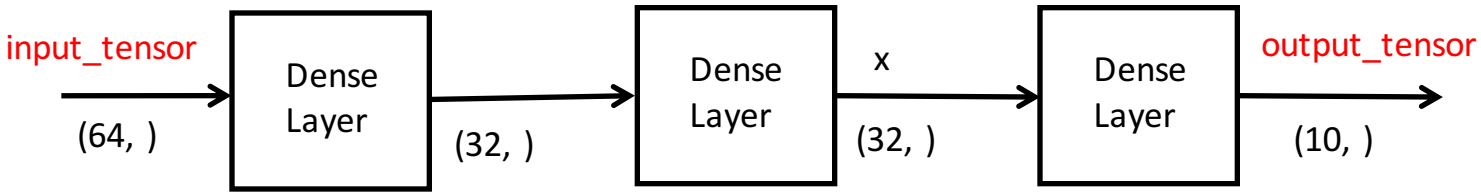


Sequential model

```
seq_model = Sequential()
seq_model.add(layers.Dense(32, activation='relu', input_shape=(64,)))
seq_model.add(layers.Dense(32, activation='relu'))
seq_model.add(layers.Dense(10, activation='softmax'))
```

Equivalent functional API

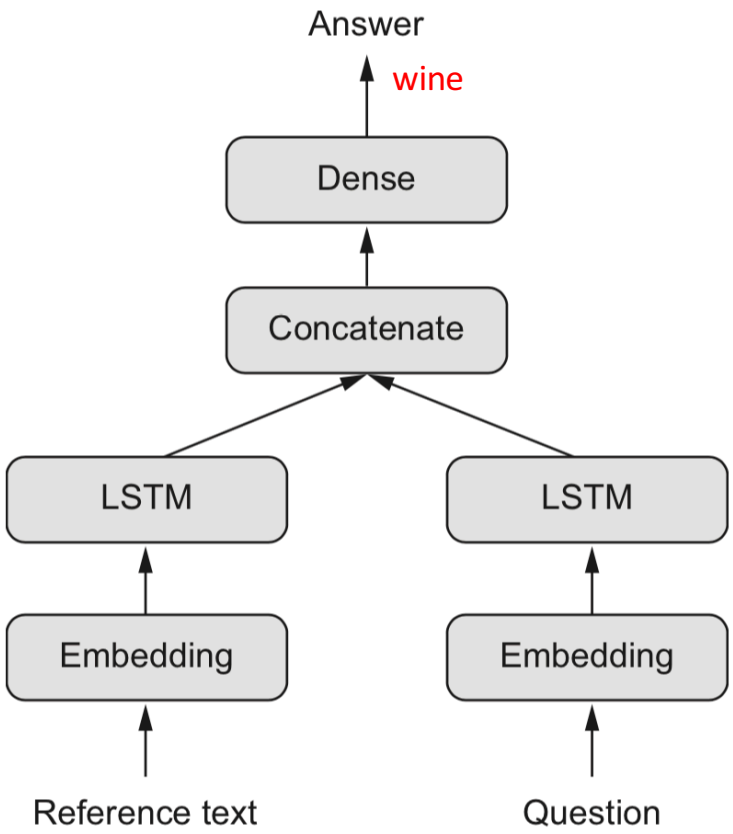
```
input_tensor = Input(shape=(64,))
x = layers.Dense(32, activation='relu')(input_tensor)
x = layers.Dense(32, activation='relu')(x)
output_tensor = layers.Dense(10, activation='softmax')(x)
model = Model(input_tensor, output_tensor)
```



Name of model:
model

Multi-input models

Example application: question-answering



Tom went out to buy wine and bread.
He came back with only bread.

What did Tom not buy?

Listing 7.1 Functional API implementation of a two-input question-answering model

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

text_vocabulary_size = 10000
question_vocabulary_size = 10000
answer_vocabulary_size = 500

text_input = Input(shape=(None,), dtype='int32', name='text')

embedded_text = layers.Embedding(
    64, text_vocabulary_size)(text_input)

encoded_text = layers.LSTM(32)(embedded_text)

question_input = Input(shape=(None,),
                       dtype='int32',
                       name='question')

embedded_question = layers.Embedding(
    32, question_vocabulary_size)(question_input)
encoded_question = layers.LSTM(16)(embedded_question)

concatenated = layers.concatenate([encoded_text, encoded_question],
                                  axis=-1)

answer = layers.Dense(answer_vocabulary_size,
                      activation='softmax')(concatenated)

model = Model([text_input, question_input], answer)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['acc'])
```

The text input is a variable-length sequence of integers. Note that you can optionally name the inputs.

Embeds the inputs into a sequence of vectors of size 64

Encodes the vectors in a single vector via an LSTM

Same process (with different layer instances) for the question

Concatenates the encoded question and encoded text

Adds a softmax classifier on top

At model instantiation, you specify the two inputs and the output.

Listing 7.1 Functional API implementation of a two-input question-answering model

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

text_vocabulary_size = 10000
question_vocabulary_size = 10000
answer_vocabulary_size = 500

text_input = Input(shape=(None,), dtype='int32', name='text')

embedded_text = layers.Embedding(
    64, text_vocabulary_size)(text_input)

encoded_text = layers.LSTM(32)(embedded_text)

question_input = Input(shape=(None,),
                       dtype='int32',
                       name='question')

embedded_question = layers.Embedding(
    32, question_vocabulary_size)(question_input)
encoded_question = layers.LSTM(16)(embedded_question)

concatenated = layers.concatenate([encoded_text, encoded_question],
                                  axis=-1)

answer = layers.Dense(answer_vocabulary_size,
                      activation='softmax')(concatenated)

model = Model([text_input, question_input], answer)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['acc'])
```

The text input is a variable-length sequence of integers. Note that you can optionally name the inputs.

Embeds the inputs into a sequence of vectors of size 64

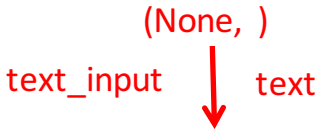
Encodes the vectors in a single vector via an LSTM

Same process (with different layer instances) for the question

Concatenates the encoded question and encoded text

Adds a softmax classifier on top

At model instantiation, you specify the two inputs and the output.



Listing 7.1 Functional API implementation of a two-input question-answering model

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

text_vocabulary_size = 10000
question_vocabulary_size = 10000
answer_vocabulary_size = 500

text_input = Input(shape=(None,), dtype='int32', name='text')

embedded_text = layers.Embedding(
    64, text_vocabulary_size)(text_input)

encoded_text = layers.LSTM(32)(embedded_text)

question_input = Input(shape=(None,),
                       dtype='int32',
                       name='question')

embedded_question = layers.Embedding(
    32, question_vocabulary_size)(question_input)
encoded_question = layers.LSTM(16)(embedded_question)

concatenated = layers.concatenate([encoded_text, encoded_question],
                                  axis=-1)

answer = layers.Dense(answer_vocabulary_size,
                      activation='softmax')(concatenated)

model = Model([text_input, question_input], answer)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['acc'])
```

The text input is a variable-length sequence of integers. Note that you can optionally name the inputs.

Embeds the inputs into a sequence of vectors of size 64

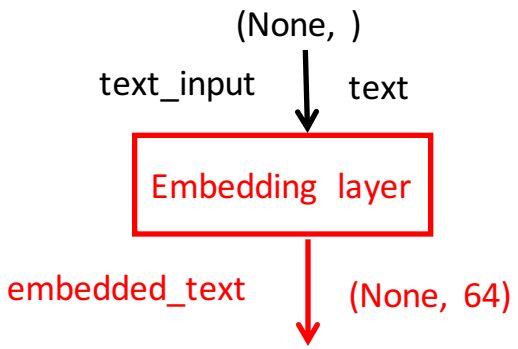
Encodes the vectors in a single vector via an LSTM

Same process (with different layer instances) for the question

Concatenates the encoded question and encoded text

Adds a softmax classifier on top

At model instantiation, you specify the two inputs and the output.



Listing 7.1 Functional API implementation of a two-input question-answering model

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

text_vocabulary_size = 10000
question_vocabulary_size = 10000
answer_vocabulary_size = 500

text_input = Input(shape=(None,), dtype='int32', name='text')

embedded_text = layers.Embedding(
    64, text_vocabulary_size)(text_input)

encoded_text = layers.LSTM(32)(embedded_text)

question_input = Input(shape=(None,),
                       dtype='int32',
                       name='question')

embedded_question = layers.Embedding(
    32, question_vocabulary_size)(question_input)
encoded_question = layers.LSTM(16)(embedded_question)

concatenated = layers.concatenate([encoded_text, encoded_question],
                                  axis=-1)

answer = layers.Dense(answer_vocabulary_size,
                      activation='softmax')(concatenated)

model = Model([text_input, question_input], answer)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['acc'])
```

The text input is a variable-length sequence of integers. Note that you can optionally name the inputs.

Embeds the inputs into a sequence of vectors of size 64

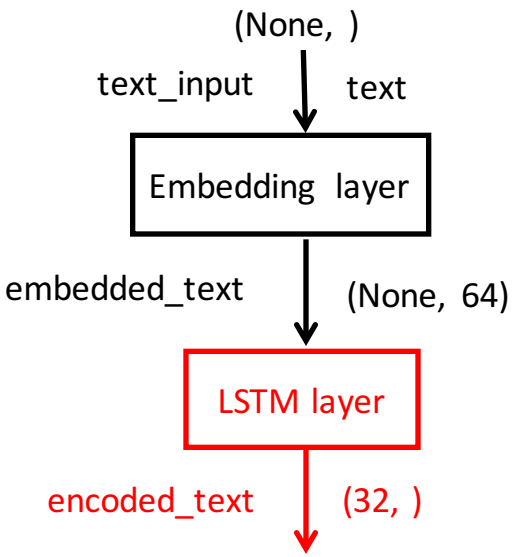
Encodes the vectors in a single vector via an LSTM

Same process (with different layer instances) for the question

Concatenates the encoded question and encoded text

Adds a softmax classifier on top

At model instantiation, you specify the two inputs and the output.



Listing 7.1 Functional API implementation of a tw

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

text_vocabulary_size = 10000
question_vocabulary_size = 10000
answer_vocabulary_size = 500

text_input = Input(shape=(None,), dtype='int3

embedded_text = layers.Embedding(
    64, text_vocabulary_size)(text_input)

encoded_text = layers.LSTM(32)(embedded_text)

question_input = Input(shape=(None,),
                       dtype='int32',
                       name='question')

embedded_question = layers.Embedding(
    32, question_vocabulary_size)(question_input)
encoded_question = layers.LSTM(16)(embedded_question)

concatenated = layers.concatenate([encoded_text, encoded_question],
                                  axis=-1)

answer = layers.Dense(answer_vocabulary_size,
                      activation='softmax')(concatenated)

model = Model([text_input, question_input], answer)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['acc'])
```

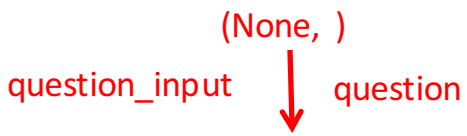
Encodes the vectors in a single vector via an LSTM

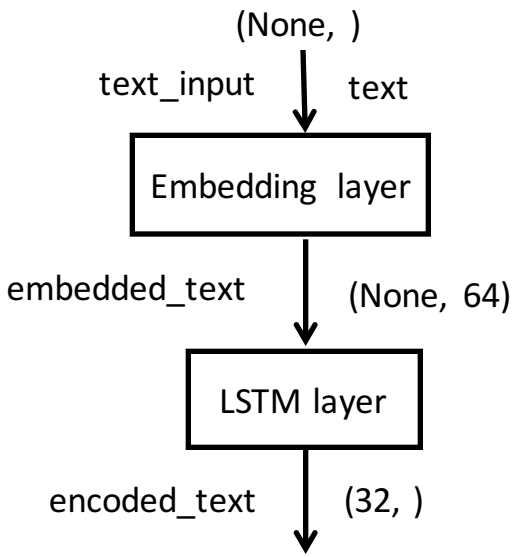
Same process (with different layer instances) for the question

Concatenates the encoded question and encoded text

Adds a softmax classifier on top

At model instantiation, you specify the two inputs and the output.





Listing 7.1 Functional API implementation of a tw

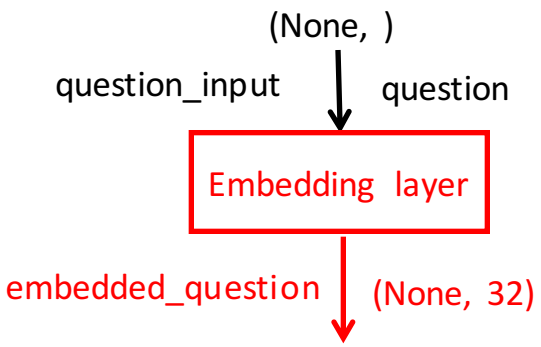
```
from keras.models import Model
from keras import layers
from keras import Input

text_vocabulary_size = 10000
question_vocabulary_size = 10000
answer_vocabulary_size = 500

text_input = Input(shape=(None,), dtype='int3

embedded_text = layers.Embedding(
    64, text_vocabulary_size)(text_input)

encoded_text = layers.LSTM(32)(embedded_text)

question_input = Input(shape=(None,),
                       dtype='int32',
                       name='question')

embedded_question = layers.Embedding(
    32, question_vocabulary_size)(question_input)
encoded_question = layers.LSTM(16)(embedded_question)

concatenated = layers.concatenate([encoded_text, encoded_question],
                                  axis=-1)

answer = layers.Dense(answer_vocabulary_size,
                      activation='softmax')(concatenated)

model = Model([text_input, question_input], answer)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['acc'])
```

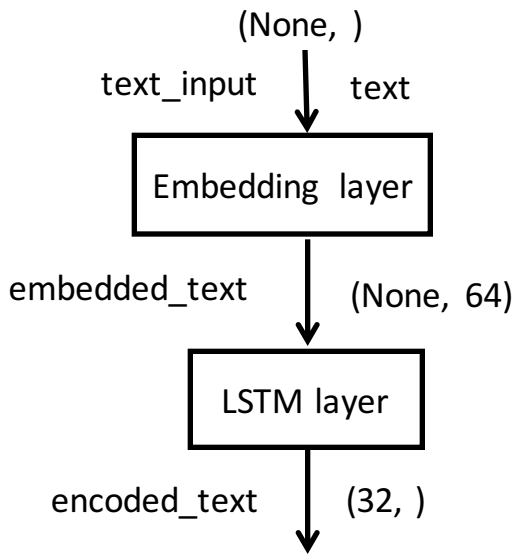
Same process (with different layer instances) for the question

Concatenates the encoded question and encoded text

Adds a softmax classifier on top

At model instantiation, you specify the two inputs and the output.





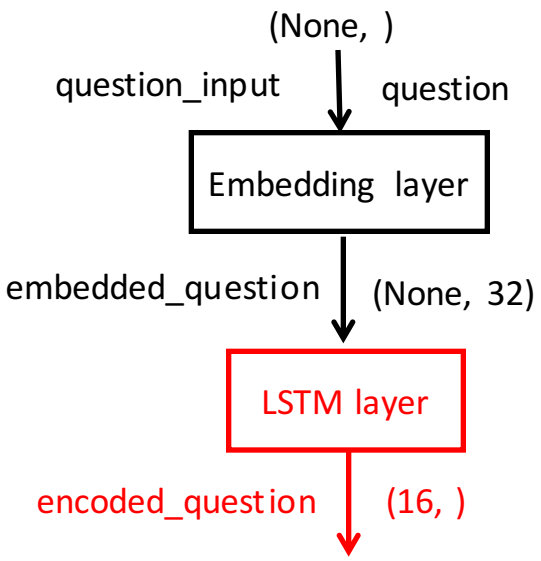
Listing 7.1 Functional API implementation of a tw

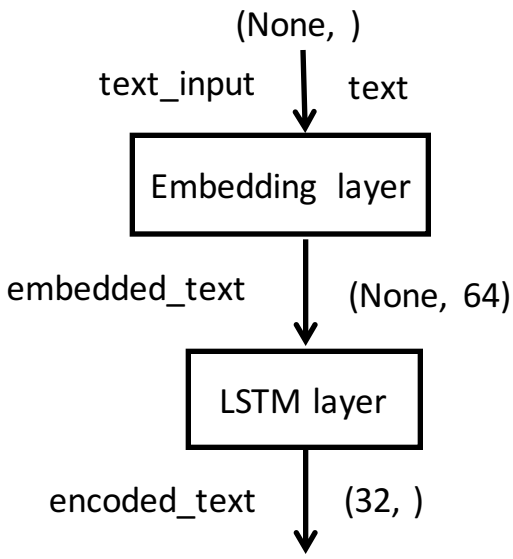
```
from keras.models import Model
from keras import layers
from keras import Input

text_vocabulary_size = 10000
question_vocabulary_size = 10000
answer_vocabulary_size = 500

text_input = Input(shape=(None,), dtype='int3

embedded_text = layers.Embedding(
    64, text_vocabulary_size)(text_input)

encoded_text = layers.LSTM(32)(embedded_text)

question_input = Input(shape=(None,),
                       dtype='int32',
                       name='question')

embedded_question = layers.Embedding(
    32, question_vocabulary_size)(question_input)
encoded_question = layers.LSTM(16)(embedded_question)

concatenated = layers.concatenate([encoded_text, encoded_question],
                                  axis=-1)

answer = layers.Dense(answer_vocabulary_size,
                      activation='softmax')(concatenated)

model = Model([text_input, question_input], answer)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['acc'])
```

Concatenates the encoded question and encoded text

Adds a softmax classifier on top

At model instantiation, you specify the two inputs and the output.





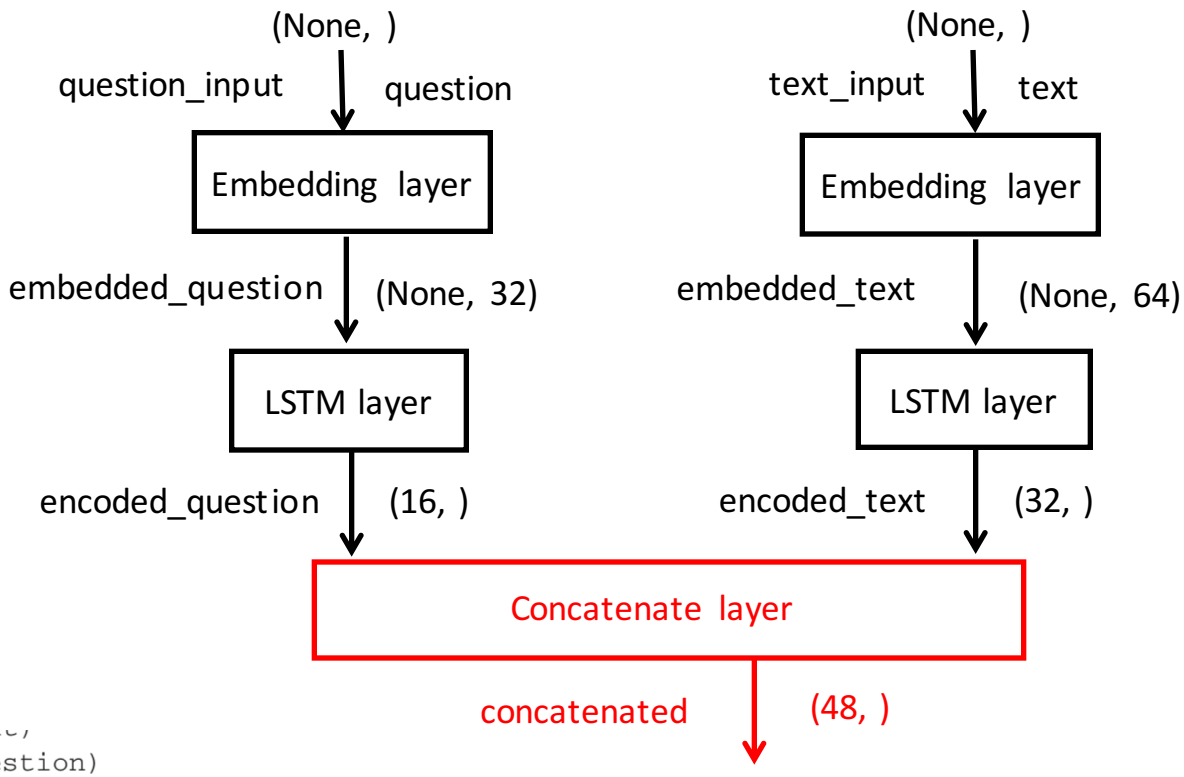
Listing 7.1 Functional API implementation of a tw

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

text_vocabulary_size = 10000
question_vocabulary_size = 10000
answer_vocabulary_size = 500

text_input = Input(shape=(None,), dtype='int3

embedded_text = layers.Embedding(
    64, text_vocabulary_size)(text_input)

encoded_text = layers.LSTM(32)(embedded_text)

question_input = Input(shape=(None,),
                       dtype='int32',
                       name='question')

embedded_question = layers.Embedding(
    32, question_vocabulary_size)(question_in
encoded_question = layers.LSTM(16)(embedded_question)

concatenated = layers.concatenate([encoded_text, encoded_question],
                                  axis=-1)

answer = layers.Dense(answer_vocabulary_size,
                      activation='softmax')(concatenated)

model = Model([text_input, question_input], answer)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['acc'])
```

Concatenates the encoded question and encoded text

Adds a softmax classifier on top

At model instantiation, you specify the two inputs and the output.



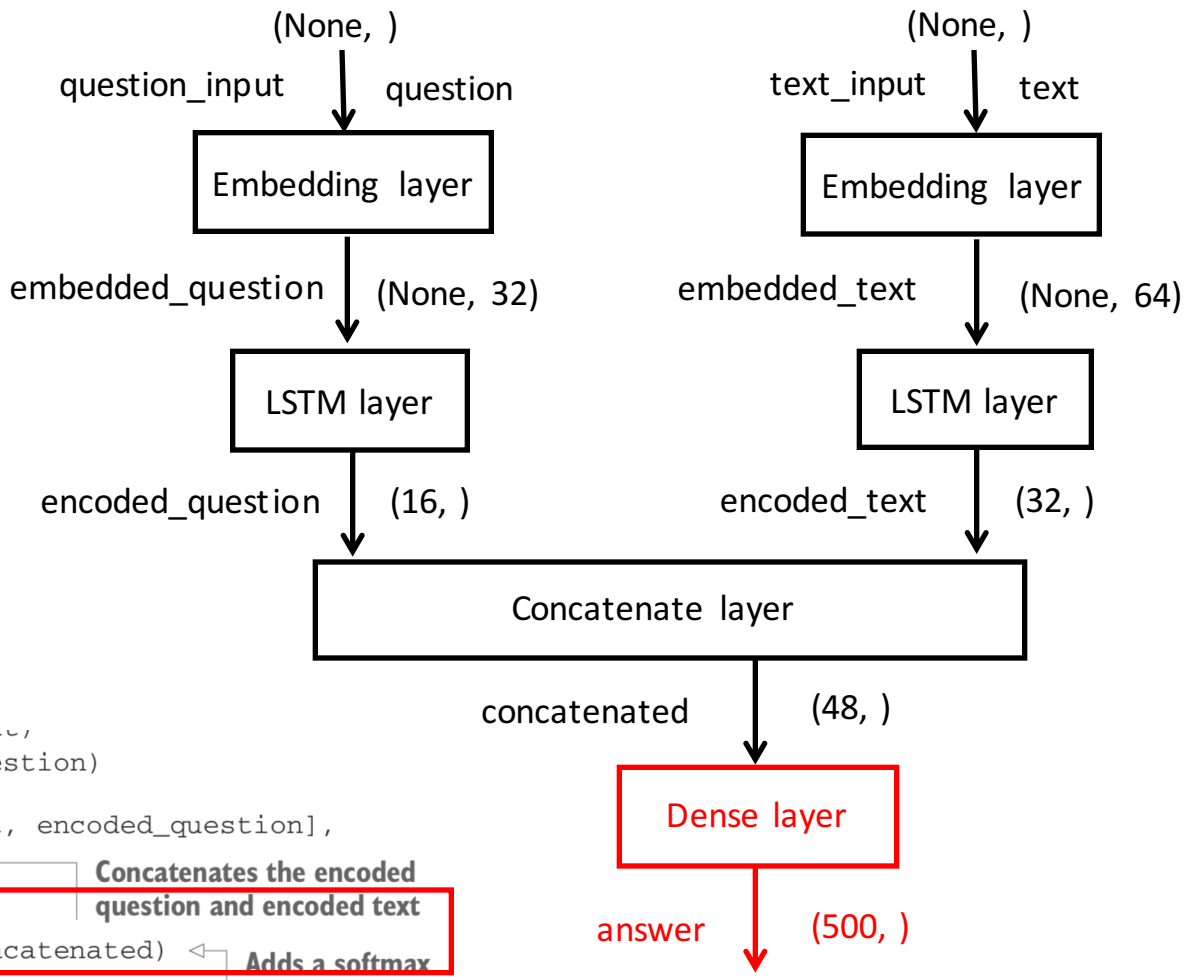
Listing 7.1 Functional API implementation of a tw

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

text_vocabulary_size = 10000
question_vocabulary_size = 10000
answer_vocabulary_size = 500

text_input = Input(shape=(None,), dtype='int3

embedded_text = layers.Embedding(
    64, text_vocabulary_size)(text_input)

encoded_text = layers.LSTM(32)(embedded_text)

question_input = Input(shape=(None,),
                       dtype='int32',
                       name='question')

embedded_question = layers.Embedding(
    32, question_vocabulary_size)(question_in
encoded_question = layers.LSTM(16)(embedded_question)

concatenated = layers.concatenate([encoded_text, encoded_question],
                                  axis=-1)

answer = layers.Dense(answer_vocabulary_size,
                      activation='softmax')(concatenated)

model = Model([text_input, question_input], answer)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['acc'])
```

Concatenates the encoded question and encoded text

Adds a softmax classifier on top

At model instantiation, you specify the two inputs and the output.



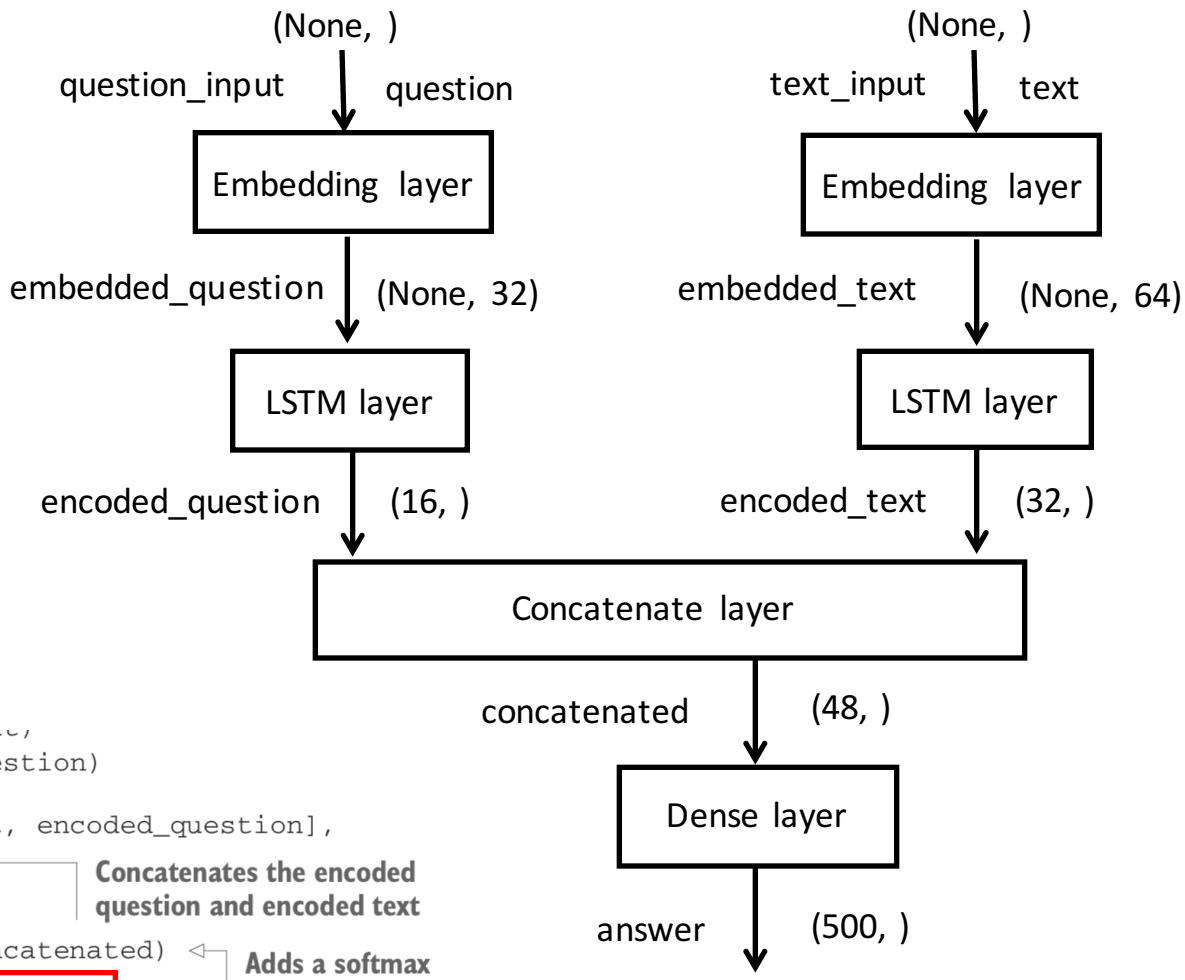
Listing 7.1 Functional API implementation of a tw

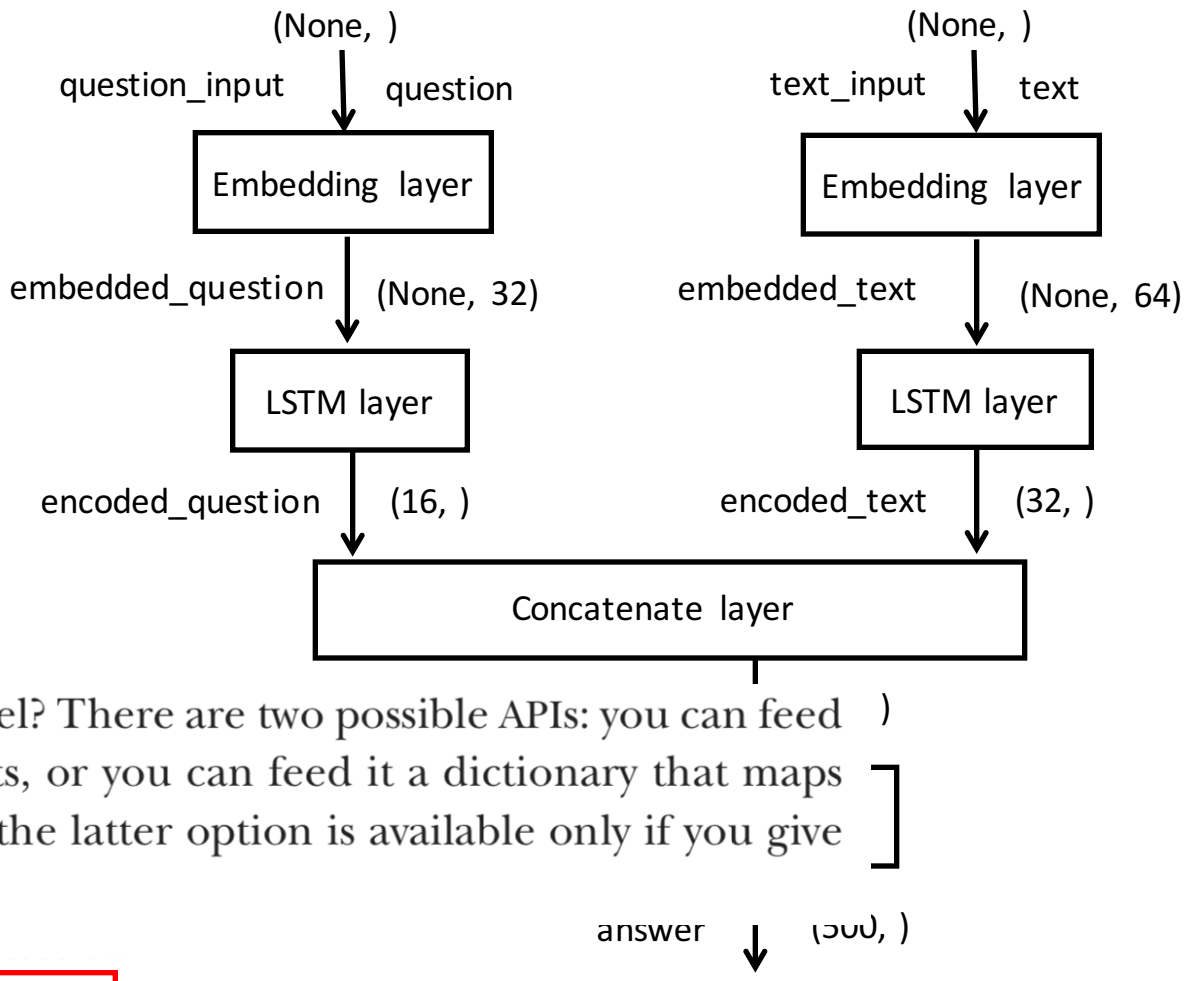
```
from keras.models import Model
from keras import layers
from keras import Input

text_vocabulary_size = 10000
question_vocabulary_size = 10000
answer_vocabulary_size = 500

text_input = Input(shape=(None,), dtype='int3

embedded_text = layers.Embedding(
    64, text_vocabulary_size)(text_input)

encoded_text = layers.LSTM(32)(embedded_text)

question_input = Input(shape=(None,),
                       dtype='int32',
                       name='question')

embedded_question = layers.Embedding(
    32, question_vocabulary_size)(question_input)
encoded_question = layers.LSTM(16)(embedded_question)

concatenated = layers.concatenate([encoded_text, encoded_question],
                                  axis=-1)

answer = layers.Dense(answer_vocabulary_size,
                      activation='softmax')(concatenated)

model = Model([text_input, question_input], answer)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['acc'])
```

Concatenates the encoded question and encoded text

Adds a softmax classifier on top

At model instantiation, you specify the two inputs and the output.

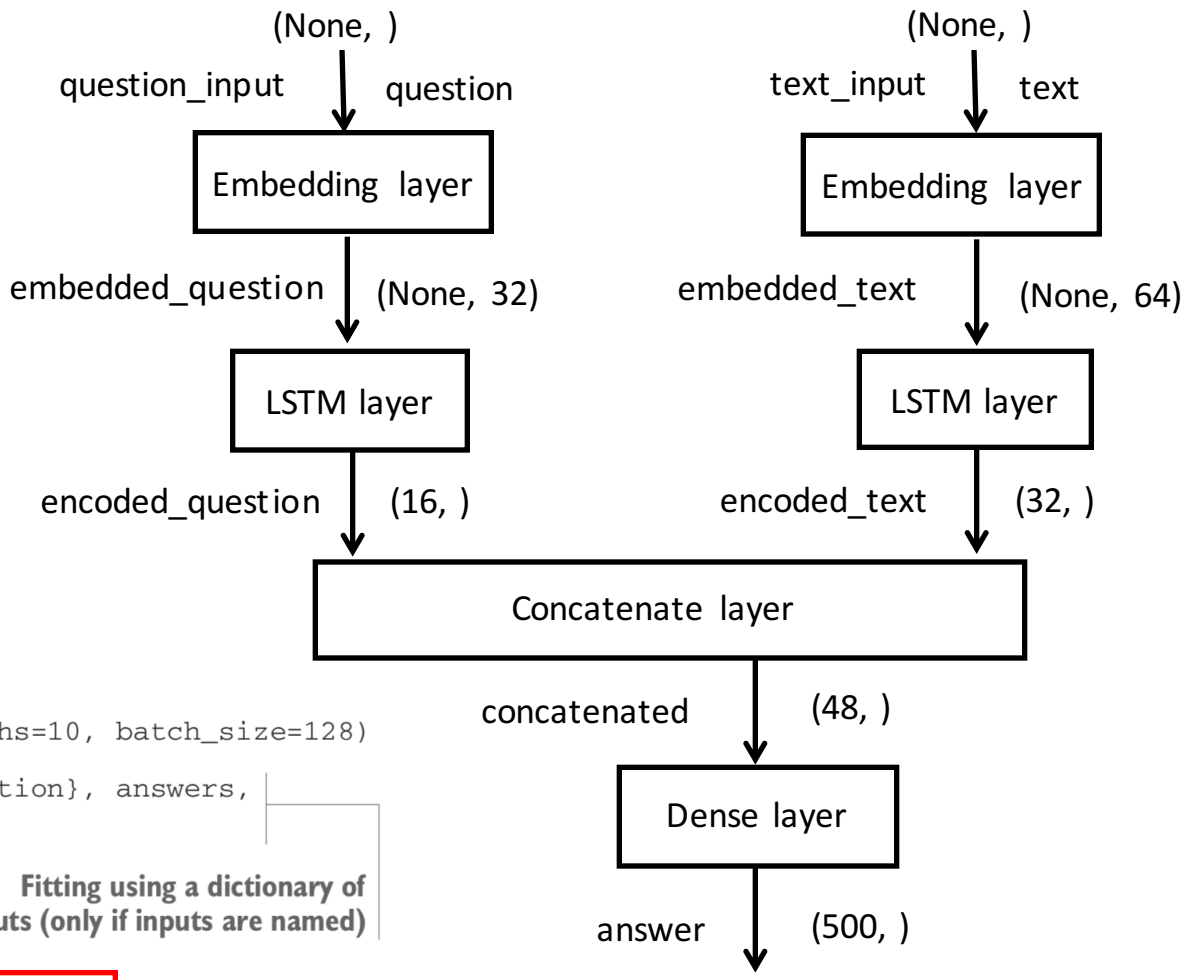




Now, how do you train this two-input model? There are two possible APIs: you can feed the model a list of Numpy arrays as inputs, or you can feed it a dictionary that maps input names to Numpy arrays. Naturally, the latter option is available only if you give names to your inputs.

answer (500,)

```
model = Model([text_input, question_input], answer)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['acc'])
```




```
model.fit([text, question], answers, epochs=10, batch_size=128)
```

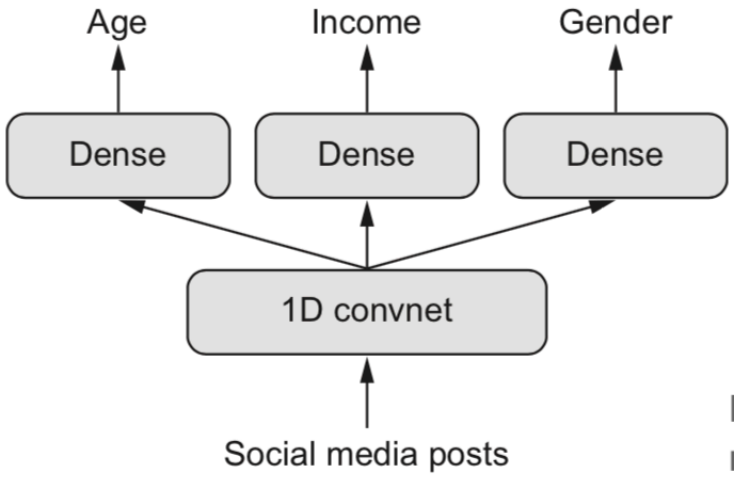
Fitting using a list of inputs

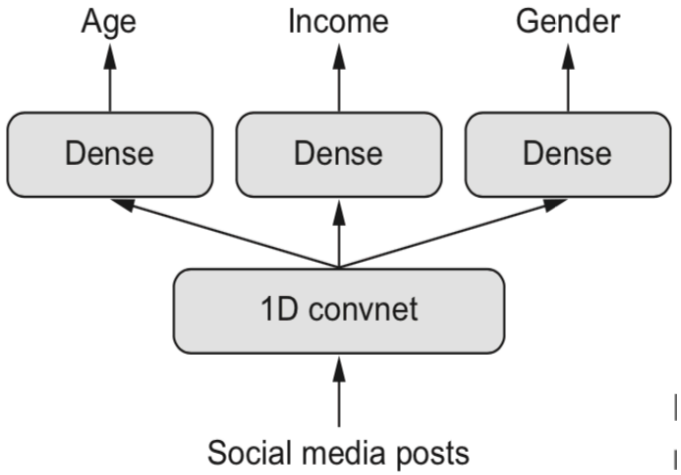
```
model.fit({'text': text, 'question': question}, answers,
          epochs=10, batch_size=128)
```

Fitting using a dictionary of inputs (only if inputs are named)

```
model = Model([text_input, question_input], answer)
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['acc'])
```

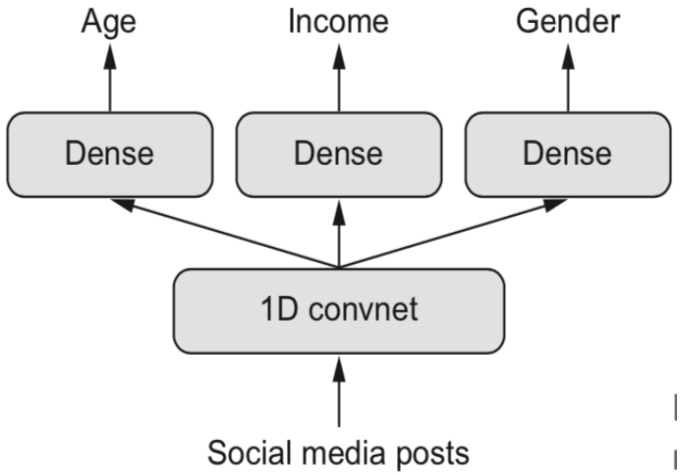
Multi-output models (see textbook for details)





How to train the multi-output model:

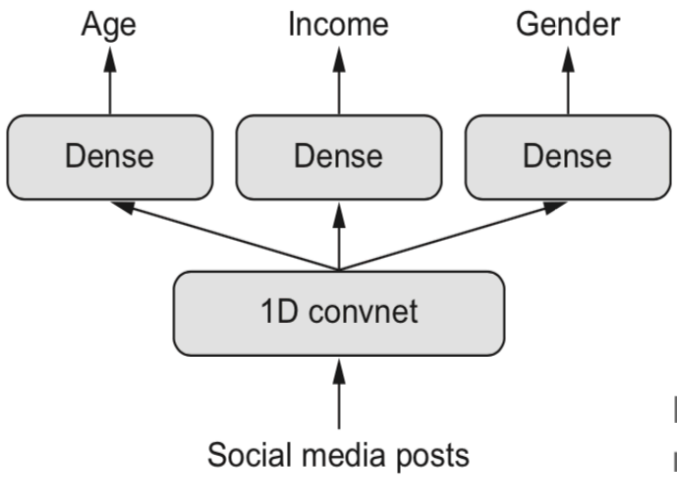
1. Choose a loss function for each output.
   (And we can choose a weight for each loss function.)
2. Keras will take their (weighted) sum as the overall loss function.



Listing 7.4 Compilation options of a multi-output model: multiple losses

```
model.compile(optimizer='rmsprop',
              loss=['mse', 'categorical_crossentropy', 'binary_crossentropy'])

model.compile(optimizer='rmsprop',
              loss={'age': 'mse',
                    'income': 'categorical_crossentropy',
                    'gender': 'binary_crossentropy'})
```

Equivalent (possible only if you give names to the output layers)



Listing 7.5 Compilation options of a multi-output model: loss weighting

```
model.compile(optimizer='rmsprop',
              loss=['mse', 'categorical_crossentropy', 'binary_crossentropy'],
              loss_weights=[0.25, 1., 10.])
```

```
model.compile(optimizer='rmsprop',
              loss={'age': 'mse',
                    'income': 'categorical_crossentropy',
                    'gender': 'binary_crossentropy'},
              loss_weights={'age': 0.25,
                            'income': 1.,
                            'gender': 10.})
```

Equivalent (possible only if you give names to the output layers)

Directed acyclic graphs of layers

A neural network model can be any directed acyclic graph.

Inception modules

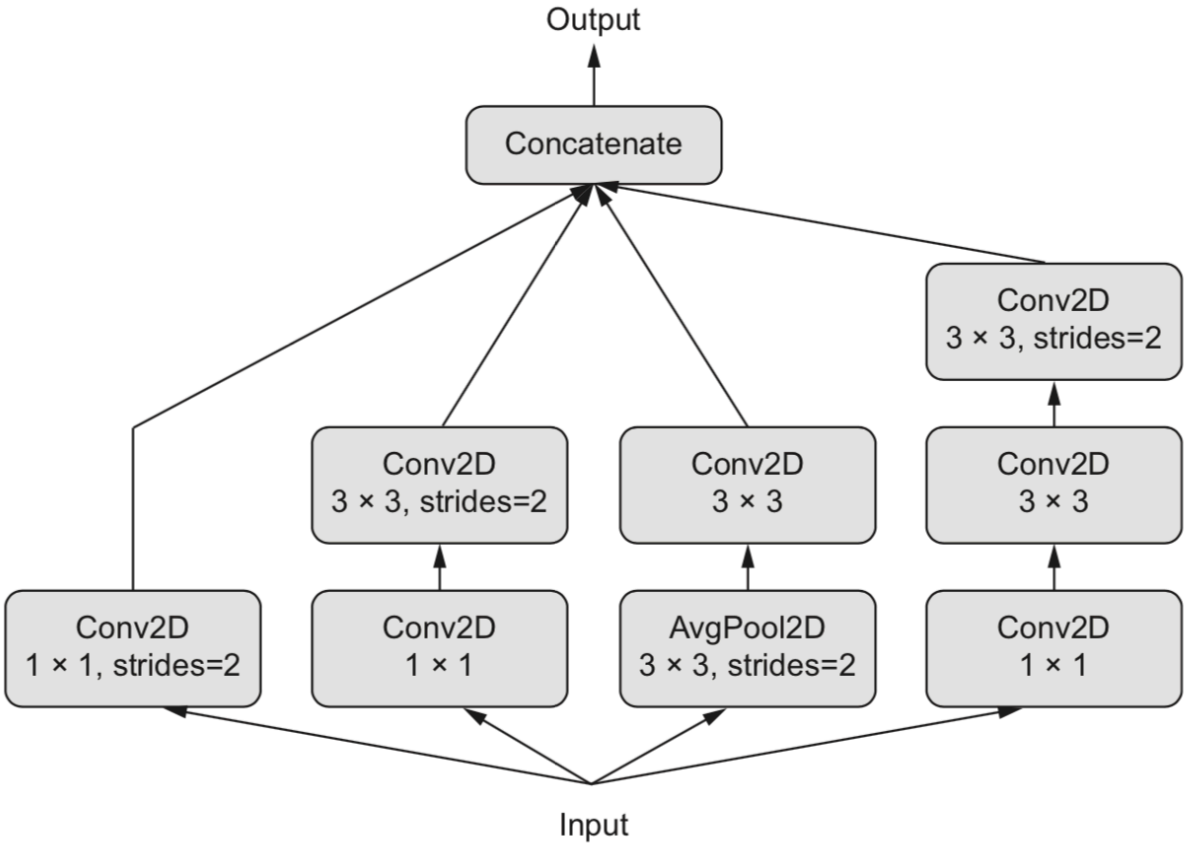
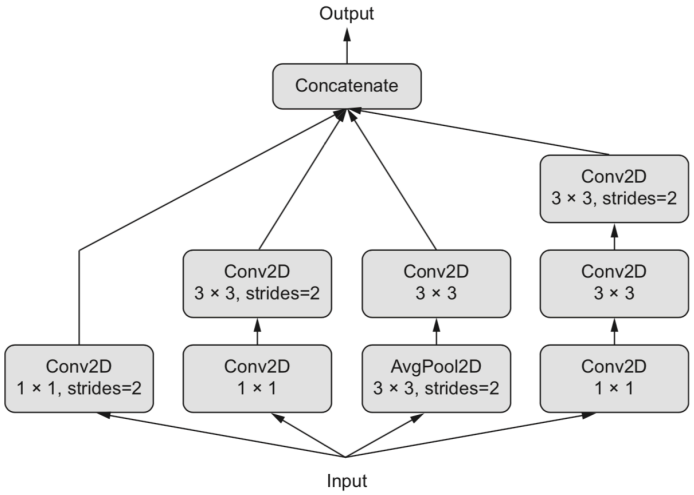


Figure 7.8 An Inception module

Inception modules



Every branch has the same stride value (2), which is necessary to keep all branch outputs the same size so you can concatenate them.

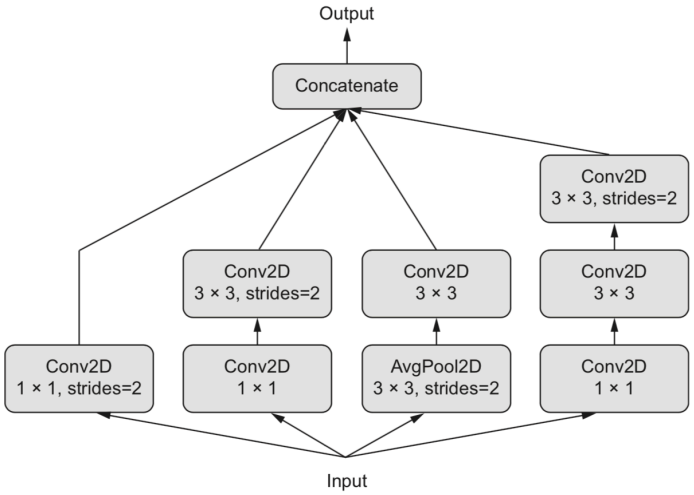
In this branch, the striding occurs in the spatial convolution layer.

```
from keras import layers

branch_a = layers.Conv2D(128, 1,
                         activation='relu', strides=2)(x)
branch_b = layers.Conv2D(128, 1, activation='relu')(x)
branch_b = layers.Conv2D(128, 3, activation='relu', strides=2)(branch_b)

branch_c = layers.AveragePooling2D(3, strides=2)(x)
branch_c = layers.Conv2D(128, 3, activation='relu')(branch_c)

branch_d = layers.Conv2D(128, 1, activation='relu')(x)
branch_d = layers.Conv2D(128, 3, activation='relu')(branch_d)
branch_d = layers.Conv2D(128, 3, activation='relu', strides=2)(branch_d)

output = layers.concatenate(
    [branch_a, branch_b, branch_c, branch_d], axis=-1)
```

In this branch, the striding occurs in the average pooling layer.

Concatenates the branch outputs to obtain the module output

Inception modules



Every branch has the same stride value (2), which is necessary to keep all branch outputs the same size so you can concatenate them.

In this branch, the striding occurs in the spatial convolution layer.

```
from keras import layers

branch_a = layers.Conv2D(128, 1,
                         activation='relu', strides=2)(x)
branch_b = layers.Conv2D(128, 1, activation='relu')(x)
branch_b = layers.Conv2D(128, 3, activation='relu', strides=2)(branch_b)

branch_c = layers.AveragePooling2D(3, strides=2)(x)
branch_c = layers.Conv2D(128, 3, activation='relu')(branch_c)

branch_d = layers.Conv2D(128, 1, activation='relu')(x)
branch_d = layers.Conv2D(128, 3, activation='relu')(branch_d)
branch_d = layers.Conv2D(128, 3, activation='relu', strides=2)(branch_d)

output = layers.concatenate(
    [branch_a, branch_b, branch_c, branch_d], axis=-1)
```

In this branch, the striding occurs in the average pooling layer.

Concatenates the branch outputs to obtain the module output

Xception (extreme inception) module: separating the learning of channel-wise and space-wise features to its logical extreme.

Residual connections

- Tackles two common problems: vanishing gradients and representation bottlenecks.
- Winning the ILSVRC ImageNet Challenge in 2015.
- In general, adding residual connections to any model that has more than 10 layers is likely to be beneficial.

Example:

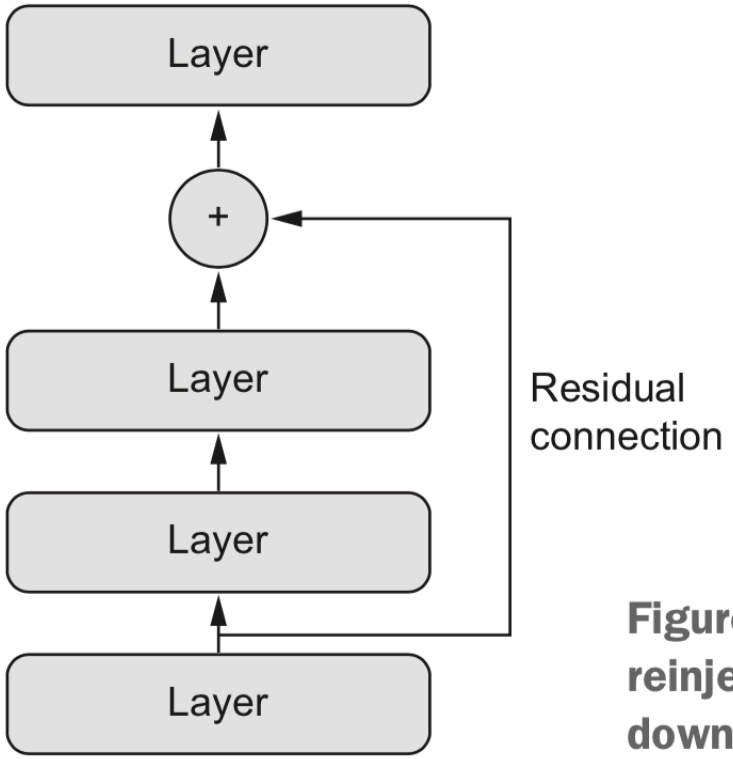
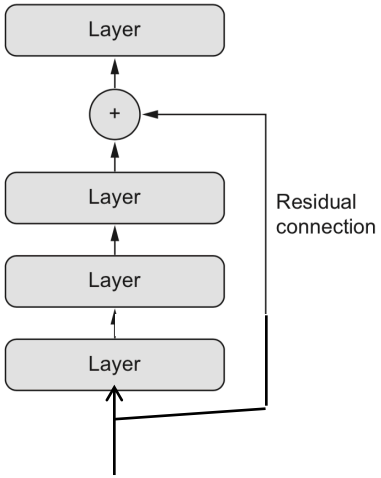


Figure 7.5 A residual connection: reinjection of prior information downstream via feature-map addition

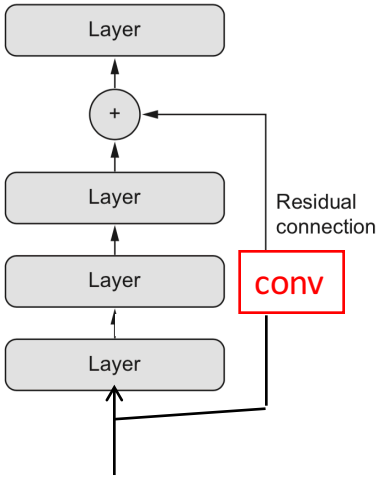



```
from keras import layers

x = ...
y = layers.Conv2D(128, 3, activation='relu', padding='same')(x)
y = layers.Conv2D(128, 3, activation='relu', padding='same')(y)
y = layers.Conv2D(128, 3, activation='relu', padding='same')(y)

y = layers.add([y, x])
```

Applies a transformation to x

Adds the original x back to the output features

Point-wise add. Here X and Y need to have the same shape.




```
from keras import layers

x = ...
y = layers.Conv2D(128, 3, activation='relu', padding='same')(x)
y = layers.Conv2D(128, 3, activation='relu', padding='same')(y)
y = layers.MaxPooling2D(2, strides=2)(y)

residual = layers.Conv2D(128, 1, strides=2, padding='same')(x)

y = layers.add([y, residual])
```

Uses a 1 × 1 convolution to linearly downsample the original x tensor to the same shape as y

Adds the residual tensor back to the output features

Point-wise add. Here X and Y need to have the same shape.

Layer weight sharing

- It is useful to share weights. (Think of convolutional filters.)
- We can reuse a layer instance several times. The share the same weights (because they are the same layer instance).

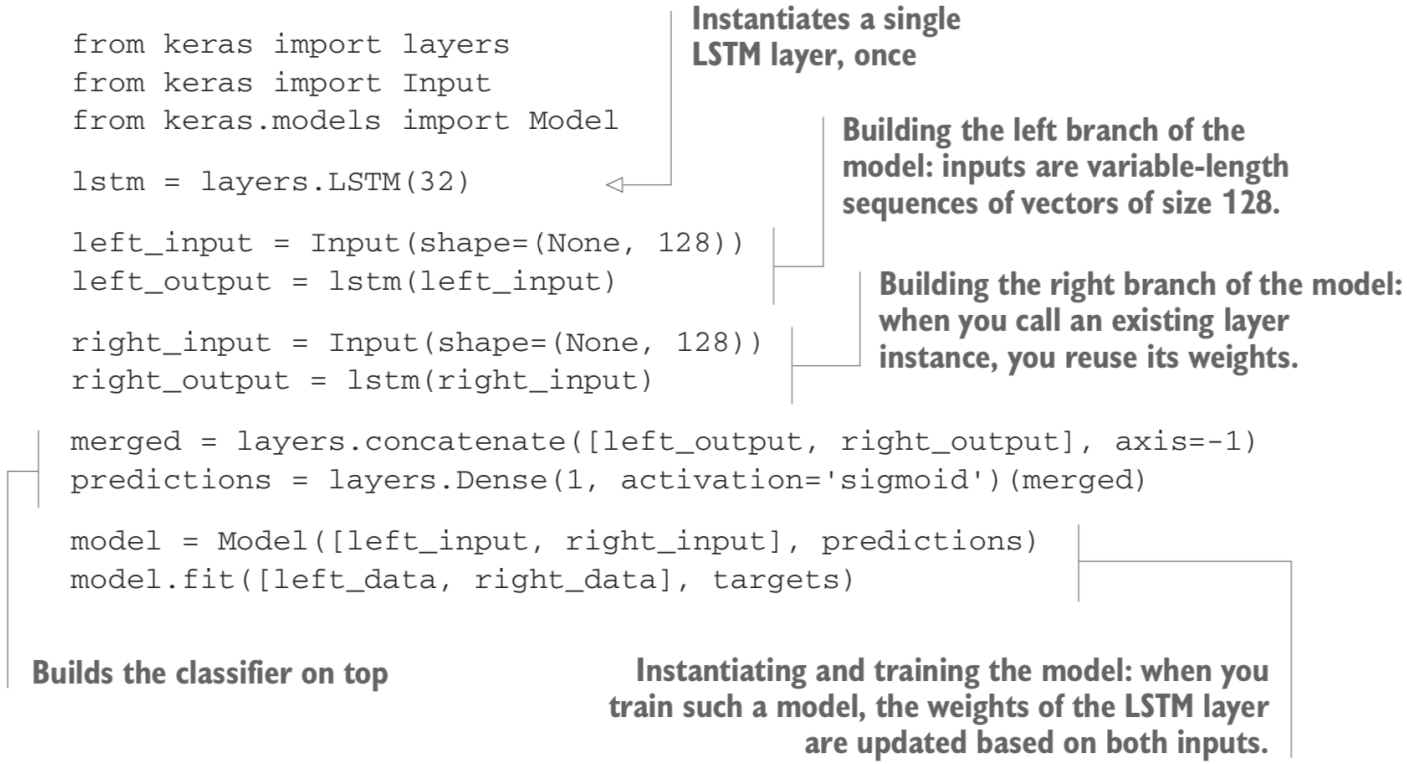
For example, consider a model that attempts to assess the semantic similarity between two sentences. The model has two inputs (the two sentences to compare) and outputs a score between 0 and 1, where 0 means unrelated sentences and 1 means sentences that are either identical or reformulations of each other. Such a model could be useful in many applications, including deduplicating natural-language queries in a dialog system.

For example, consider a model that attempts to assess the semantic similarity between two sentences. The model has two inputs (the two sentences to compare) and outputs a score between 0 and 1, where 0 means unrelated sentences and 1 means sentences that are either identical or reformulations of each other. Such a model could be useful in many applications, including deduplicating natural-language queries in a dialog system.

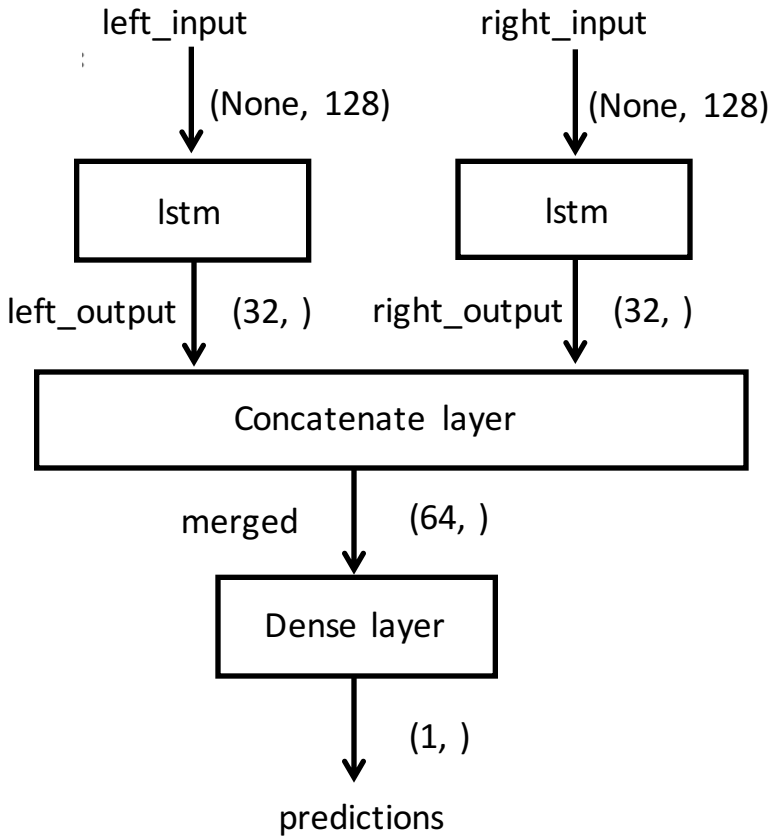
We use the same LSTM layer to turn each sentence into a set of features.

We then compare the similarity of the two sets of features, to decide if the two sentences are similar or not.

The representations of this LSTM layer (its weights) are learned based on both inputs simultaneously.
This is called a Siamese LSTM model or a shared LSTM.


from keras import layers
from keras import Input
from keras.models import Model
lstm = layers.LSTM(32)
left_input = Input(shape=(None, 128))
left_output = lstm(left_input)
right_input = Input(shape=(None, 128))
right_output = lstm(right_input)
merged = layers.concatenate([left_output, right_output], axis=-1)
predictions = layers.Dense(1, activation='sigmoid')(merged)
model = Model([left_input, right_input], predictions)
model.fit([left_data, right_data], targets)
Instantiates a single LSTM layer, once
Building the left branch of the model: inputs are variable-length sequences of vectors of size 128.
Building the right branch of the model: when you call an existing layer instance, you reuse its weights.
Builds the classifier on top
Instantiating and training the model: when you train such a model, the weights of the LSTM layer are updated based on both inputs.

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

lstm = layers.LSTM(32)

left_input = Input(shape=(None, 128))
left_output = lstm(left_input)

right_input = Input(shape=(None, 128))
right_output = lstm(right_input)

merged = layers.concatenate([left_output, right_output], axis=-1)
predictions = layers.Dense(1, activation='sigmoid')(merged)

model = Model([left_input, right_input], predictions)
model.fit([left_data, right_data], targets)
```



Models as layers

- A model can be used like a layer. (We can think of model as a “bigger layer”.)

```
y = model(x)
```

If the model has multiple input tensors and multiple output tensors, it should be called with a list of tensors:

```
y1, y2 = model([x1, x2])
```

Reuse weights of a model:

When you call a model instance, you're reusing the weights of the model—exactly like what happens when you call a layer instance. Calling an instance, whether it's a layer instance or a model instance, will always reuse the existing learned representations of the instance—which is intuitive.

Example: Dual camera that can perceive depth

We use the convolutional base of the Xception network (that is, we remove it top dense layer, which is used for classification) to extract features of the input image.

The same model can be used for both cameras.
(It is a Siamese vision model, or shared convolutional base.)

```
from keras import layers
from keras import applications
from keras import Input

xception_base = applications.Xception(weights=None,
                                      include_top=False)

left_input = Input(shape=(250, 250, 3))
right_input = Input(shape=(250, 250, 3))

left_features = xception_base(left_input)
right_features = xception_base(right_input)

merged_features = layers.concatenate(
    [left_features, right_input], axis=-1)
```

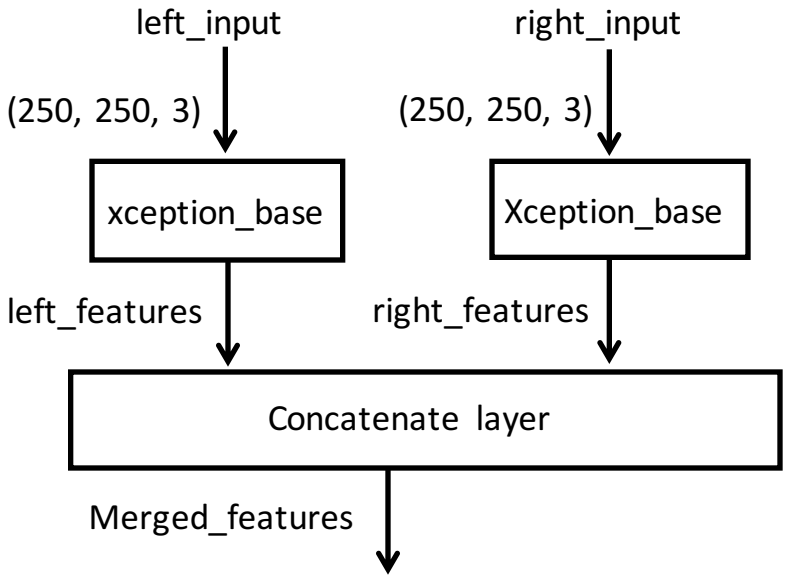
The base image-processing model is the Xception network (convolutional base only).

The inputs are 250 × 250 RGB images.

Calls the same vision model twice

The merged features contain information from the right visual feed and the left visual feed.

```
from keras import layers
from keras import applications
from keras import Input

xception_base = applications.Xception(weights=None,
                                      include_top=False)

left_input = Input(shape=(250, 250, 3))
right_input = Input(shape=(250, 250, 3))

left_features = xception_base(left_input)
right_features = xception_base(right_input)

merged_features = layers.concatenate(
    [left_features, right_input], axis=-1)
```



Inspecting and monitoring deep-learning models using Keras callbacks and TensorBoard

Using callbacks to act on a model during training

Here are some examples of ways you can use callbacks:

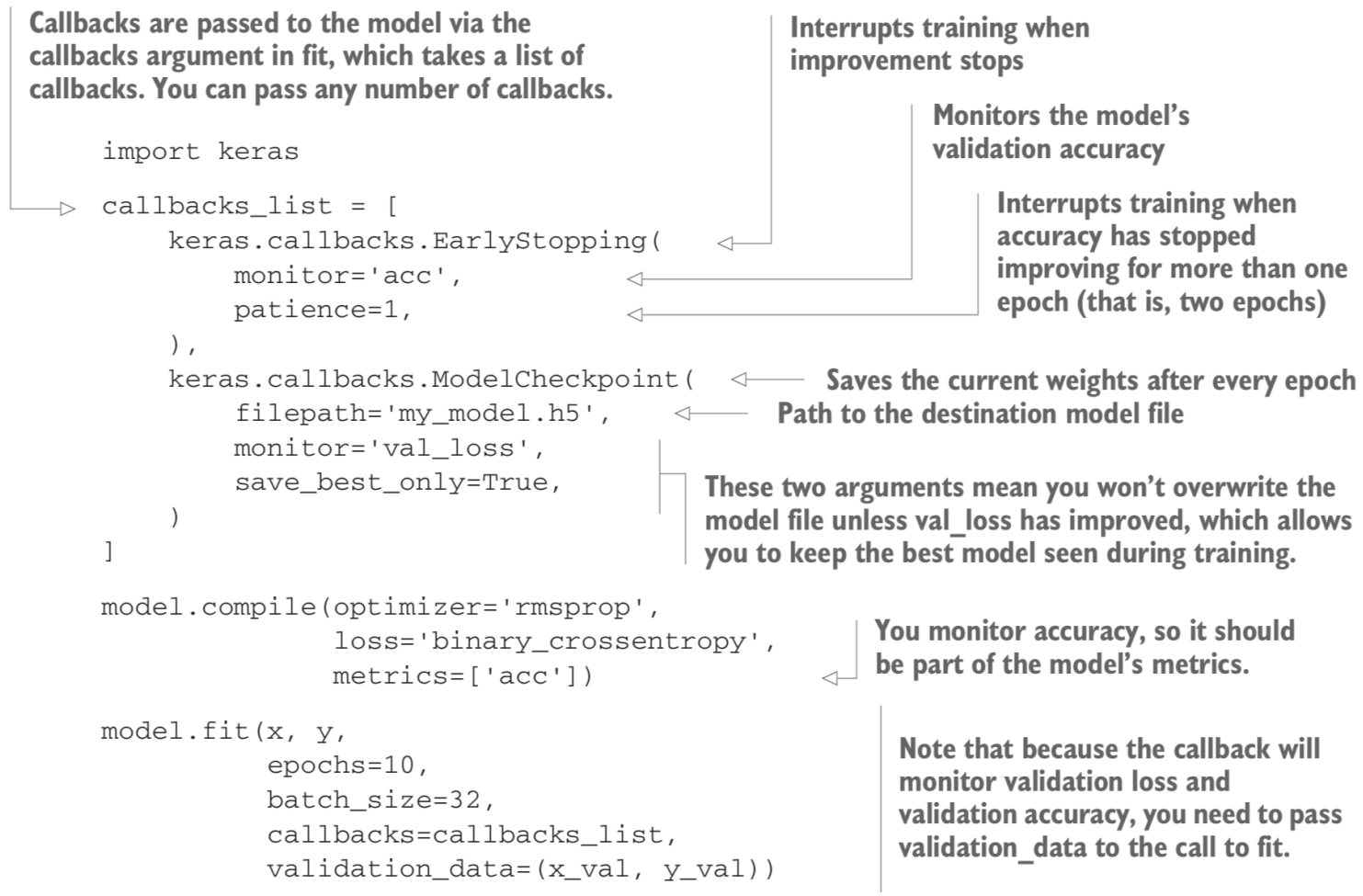
- *Model checkpointing*—Saving the current weights of the model at different points during training.
- *Early stopping*—Interrupting training when the validation loss is no longer improving (and of course, saving the best model obtained during training).
- *Dynamically adjusting the value of certain parameters during training*—Such as the learning rate of the optimizer.
- *Logging training and validation metrics during training, or visualizing the representations learned by the model as they're updated*—The Keras progress bar that you're familiar with is a callback!

Example: the ModelCheckpoint and EarlyStopping callbacks

Save the model
(its weights)
in a file

Stop training when a monitored metric
(such as accuracy or loss) stops
improving

Example: the ModelCheckpoint and EarlyStopping callbacks



Example: the ReduceLROnPlateau callback

- Reduce (or increase) the Learning Rate when the monitored metric (such as validation loss) has stopped improving.

- It is an effective way to get out of local minimal during training.

Example: the ReduceLROnPlateau callback

```
callbacks_list = [
    keras.callbacks.ReduceLROnPlateau(
        monitor='val_loss'
        factor=0.1,
        patience=10,
    )
]

model.fit(x, y,
          epochs=10,
          batch_size=32,
          callbacks=callbacks_list,
          validation_data=(x_val, y_val))
```

Monitors the model's validation loss

Divides the learning rate by 10 when triggered

The callback is triggered after the validation loss has stopped improving for 10 epochs.

Because the callback will monitor the validation loss, you need to pass validation_data to the call to fit.

Example: the ReduceLROnPlateau callback

```
callbacks_list = [
    keras.callbacks.ReduceLROnPlateau(
        monitor='val_loss'
        factor=0.1,
        patience=10,
    )
]

model.fit(x, y,
          epochs=10,
          batch_size=32,
          callbacks=callbacks_list,
          validation_data=(x_val, y_val))
```

Monitors the model's validation loss

Divides the learning rate by 10 when triggered

The callback is triggered after the validation loss has stopped improving for 10 epochs.

Because the callback will monitor the validation loss, you need to pass validation_data to the call to fit.

Your can write your own callbacks. (For details, see textbook.)

TensorBoard: the TensorFlow visualization framework

- TensorFlow is a library (platform) for deep learning.
- Keras is built on top of TensorFlow (and other platforms).
- TensorBoard is a nice visualization tool for deep learning.

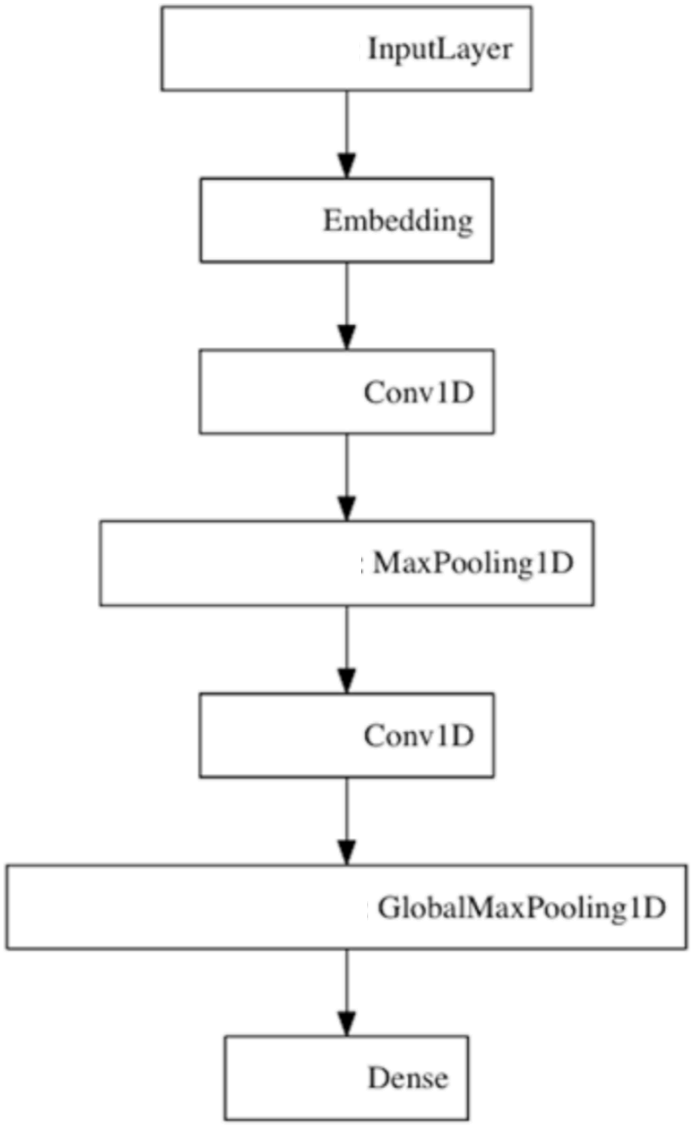
Example: Visualize DNN training for IMDB dataset

```
import keras
from keras import layers
from keras.datasets import imdb
from keras.preprocessing import sequence

max_features = 2000
max_len = 500

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=max_features)
x_train = sequence.pad_sequences(x_train, maxlen=max_len)
x_test = sequence.pad_sequences(x_test, maxlen=max_len)

model = keras.models.Sequential()
model.add(layers.Embedding(max_features, 128,
                           input_length=max_len,
                           name='embed'))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))
model.summary()
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
```

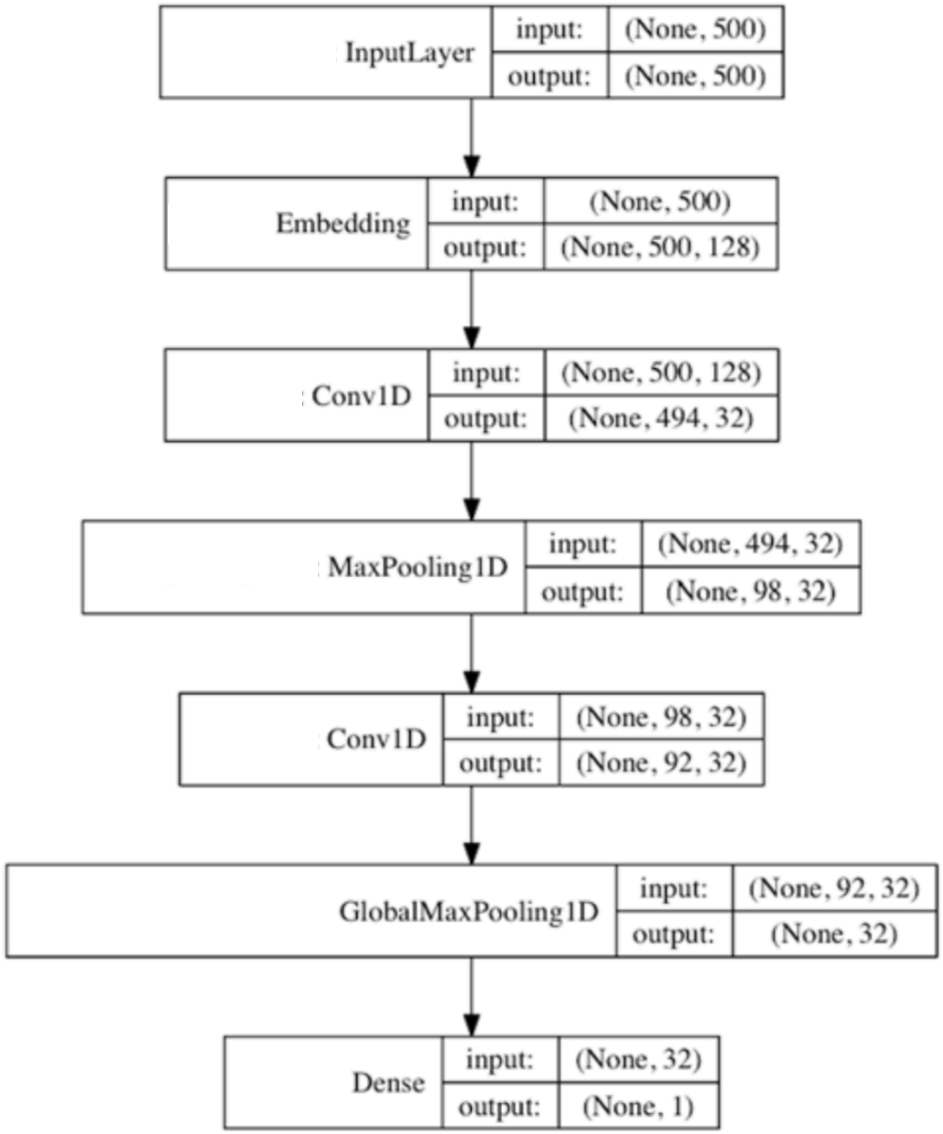
Number of words to consider as features

Cuts off texts after this number of words (among max_features most common words)

```
model = keras.models.Sequential()
model.add(layers.Embedding(max_features, 128,
                           input_length=max_len,
                           name='embed'))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))
```



```
model = keras.models.Sequential()
model.add(layers.Embedding(max_features, 128,
                           input_length=max_len,
                           name='embed'))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))
```



Listing 7.9 Training the model with a `TensorBoard` callback

```
callbacks = [
    keras.callbacks.TensorBoard(
        log_dir='my_log_dir',
        histogram_freq=1,
        embeddings_freq=1,
    )
]
history = model.fit(x_train, y_train,
                    epochs=20,
                    batch_size=128,
                    validation_split=0.2,
                    callbacks=callbacks)
```

Log files will be written at this location.

Records activation histograms every 1 epoch

Records embedding data every 1 epoch

You can then browse to http://localhost:6006 and look at your model training

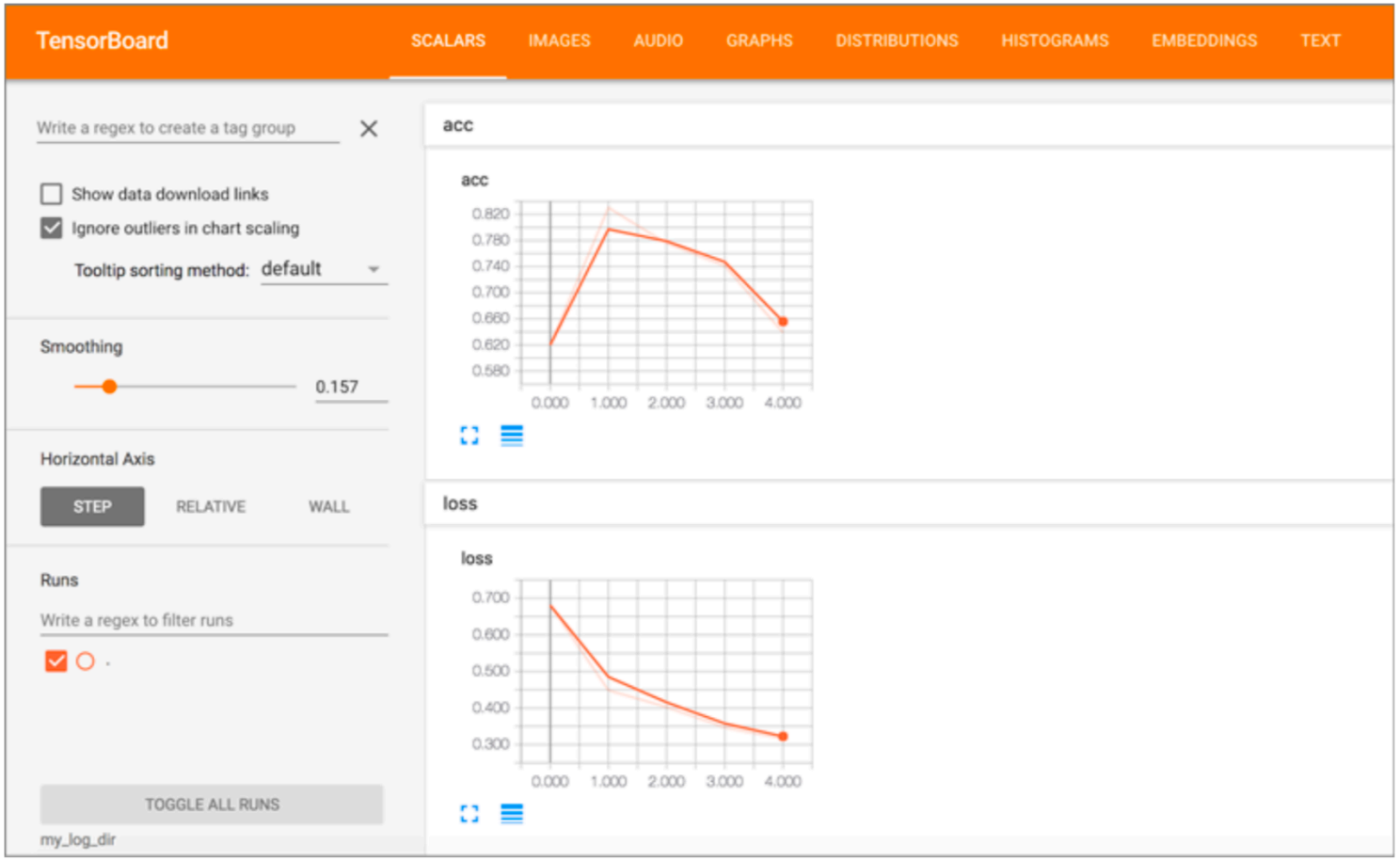
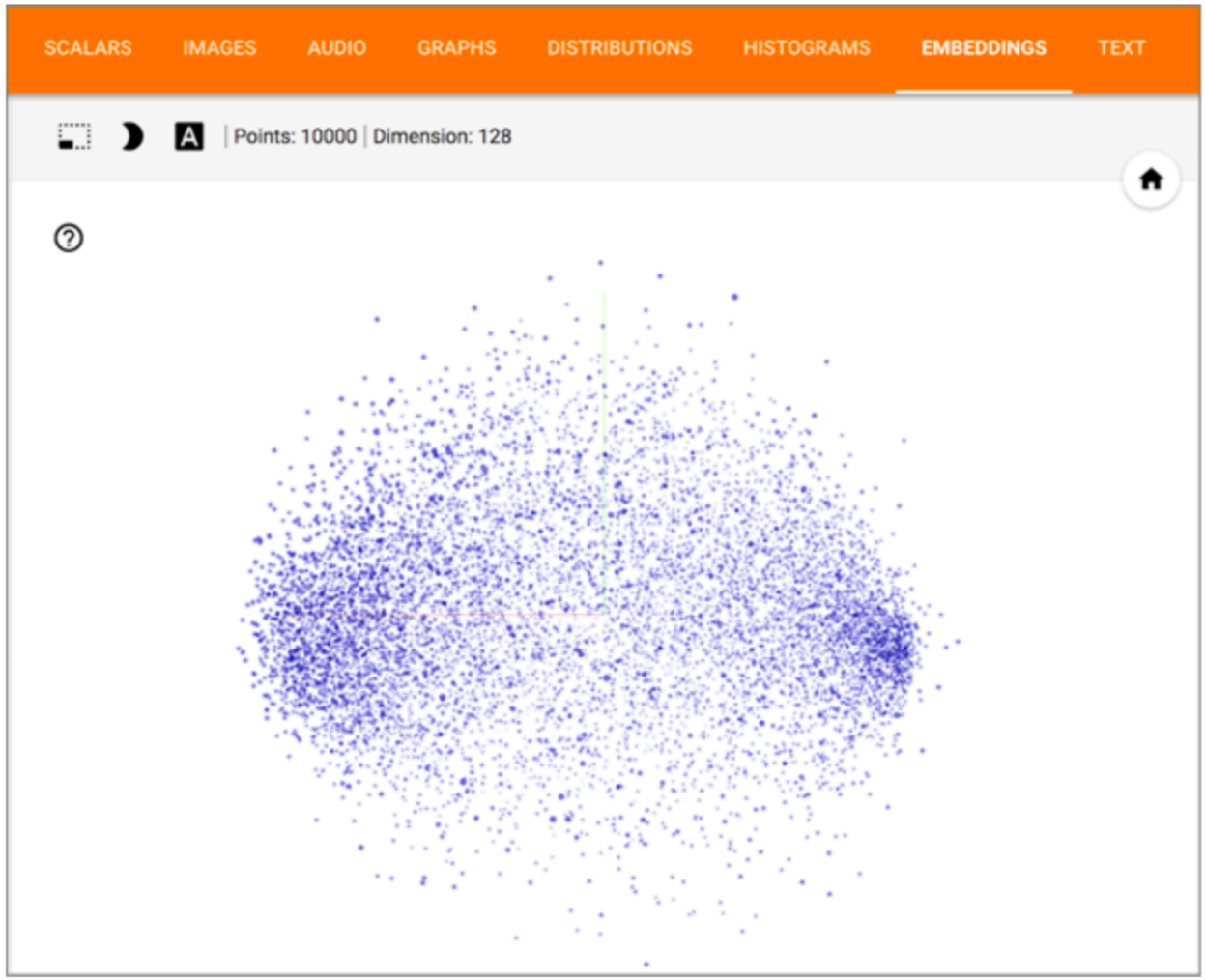


Figure 7.10 TensorBoard: metrics monitoring

Monitor word embedding (visualized in a 2-D space)



Words are forming two clusters:
One positive cluster,
One negative cluster.

```
from keras.utils import plot_model

plot_model(model, to_file='model.png')
```

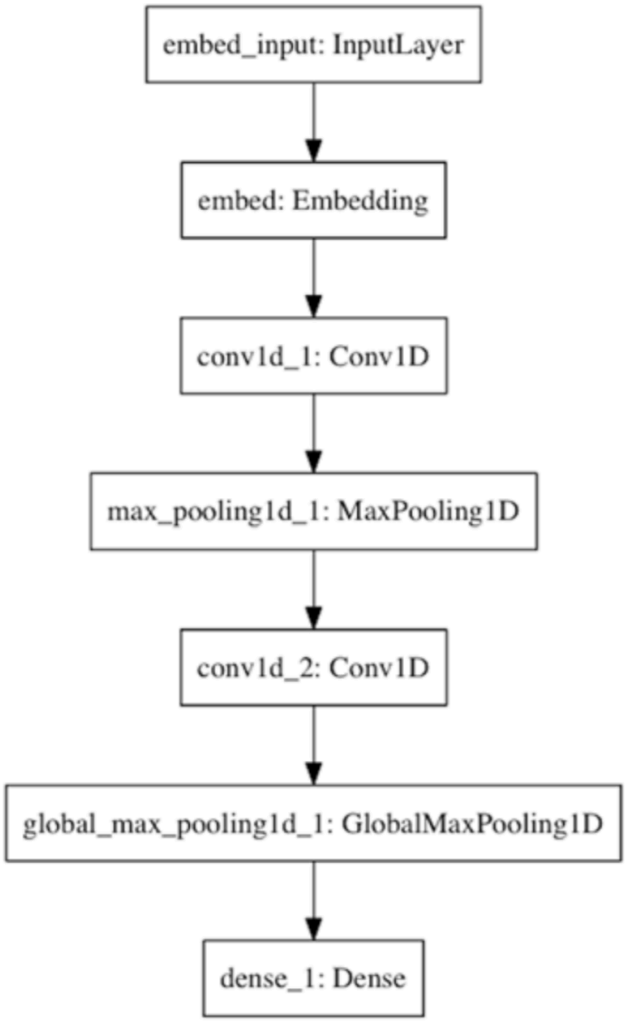


Figure 7.14 A model plot as a graph of layers, generated with `plot_model`

```
from keras.utils import plot_model

plot_model(model, show_shapes=True, to_file='model.png')
```

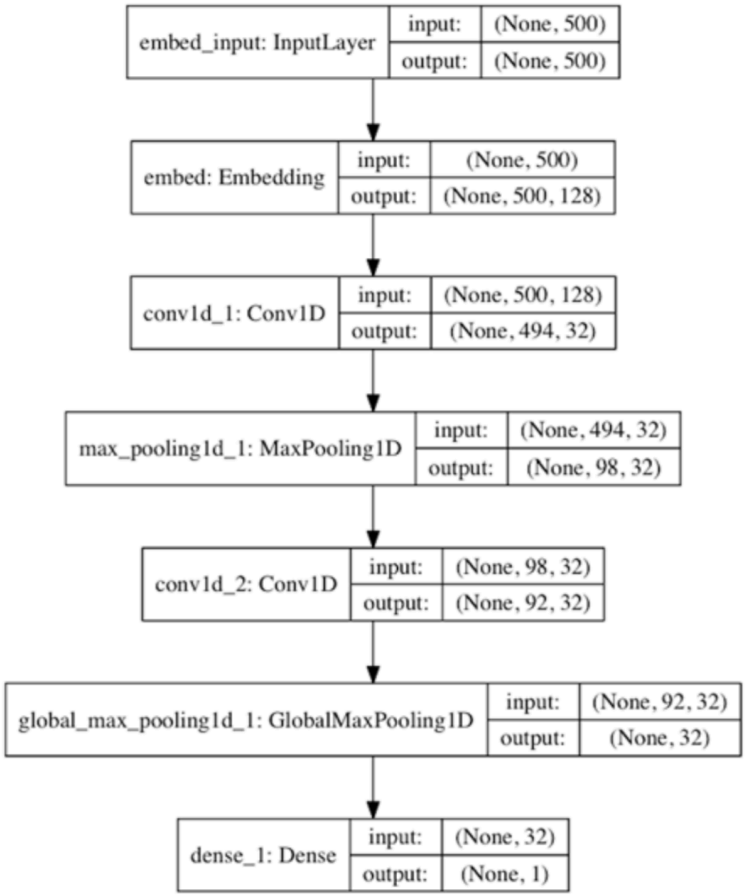


Figure 7.15 A model plot with shape information

Getting the most out of your models

Techniques for improving DNN performance

Batch normalization

How to normalize the output of a layer (not just the input data to the whole model)?

Batch normalization is a type of layer (`BatchNormalization` in Keras) introduced in 2015 by Ioffe and Szegedy;[7] it can adaptively normalize data even as the mean and variance change over time during training. It works by internally maintaining an exponential moving average of the batch-wise mean and variance of the data seen during training. The main effect of batch normalization is that it helps with gradient propagation—much like residual connections—and thus allows for deeper networks. Some very deep networks can only be trained if they include multiple `BatchNormalization` layers. For instance, `BatchNormalization` is used liberally in many of the advanced convnet architectures that come packaged with Keras, such as ResNet50, Inception V3, and Xception.

The `BatchNormalization` layer is typically used after a convolutional or densely connected layer:

```
conv_model.add(layers.Conv2D(32, 3, activation='relu'))  <-- After a Conv layer
conv_model.add(layers.BatchNormalization())

dense_model.add(layers.Dense(32, activation='relu'))  <-- After a Dense layer
dense_model.add(layers.BatchNormalization())
```

Depthwise separable convolution

What if I told you that there's a layer you can use as a drop-in replacement for `Conv2D` that will make your model lighter (fewer trainable weight parameters) and faster (fewer floating-point operations) and cause it to perform a few percentage points better on its task? That is precisely what the *depthwise separable convolution* layer does (`SeparableConv2D`). This layer performs a spatial convolution on each channel of its input, independently, before mixing output channels via a pointwise convolution (a 1×1 convolution), as shown in figure 7.16.

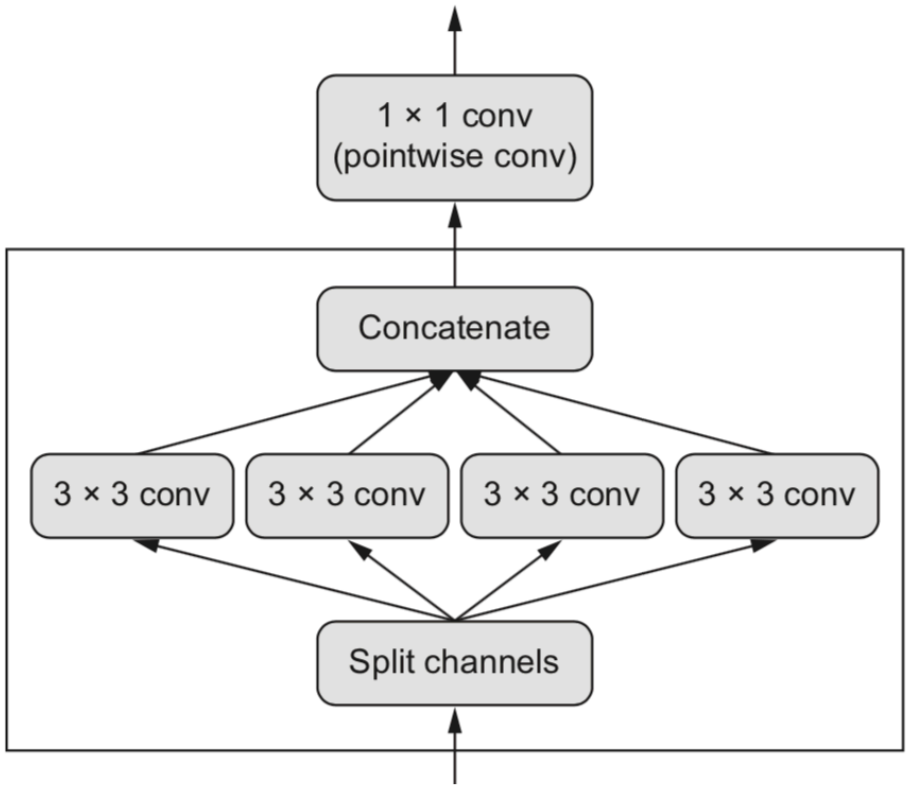


Figure 7.16 Depthwise separable convolution: a depthwise convolution followed by a pointwise convolution

```
from keras.models import Sequential, Model
from keras import layers

height = 64
width = 64
channels = 3
num_classes = 10

model = Sequential()
model.add(layers.SeparableConv2D(32, 3,
                                 activation='relu',
                                 input_shape=(height, width, channels,)))
model.add(layers.SeparableConv2D(64, 3, activation='relu'))
model.add(layers.MaxPooling2D(2))

model.add(layers.SeparableConv2D(64, 3, activation='relu'))
model.add(layers.SeparableConv2D(128, 3, activation='relu'))
model.add(layers.MaxPooling2D(2))

model.add(layers.SeparableConv2D(64, 3, activation='relu'))
model.add(layers.SeparableConv2D(128, 3, activation='relu'))
model.add(layers.GlobalAveragePooling2D())

model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(num_classes, activation='softmax'))

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

Hyper-parameter optimization

- Hyper-parameter: parameters that define the architecture of the model and the training process.
- Examples: How many layers? How many filters? Size of each filter? Which activation function to use? Should we use batch normalization? How much dropout to use? What is the learning rate? Etc.

The process of optimizing hyperparameters typically looks like this:

1. Choose a set of hyperparameters (automatically).
2. Build the corresponding model.
3. Fit it to your training data, and measure the final performance on the validation data.
4. Choose the next set of hyperparameters to try (automatically).
5. Repeat.
6. Eventually, measure performance on your test data.

Challenges:

1. Long time: every time the hyper-parameters are tuned, a new model need be trained. Then we get to know if the new hyper-parameters work well or not.

2. Discrete choices: the hyper-parameters are often not continuous. So gradient descent usually cannot be used for optimization.

AutoML: automatically choose and optimize the network architecture and hyper-parameters

- Techniques: Bayesian optimization, genetic algorithms, random search, etc.

- Some easy-to-use tools: Hyperopt, Hyperas

Model ensembling

- Train a large set of very different models
- Combine their results to get the final result

```
preds_a = model_a.predict(x_val)
preds_b = model_b.predict(x_val)
preds_c = model_c.predict(x_val)
preds_d = model_d.predict(x_val)

final_preds = 0.5 * preds_a + 0.25 * preds_b + 0.1 * preds_c + 0.15 * preds_d
```

These weights (0.5, 0.25, 0.1, 0.15) are assumed to be learned empirically.