7.1 Going beyond the Sequential model: the Keras functional API

Until now, all neural networks introduced in this book have been implemented using the `Sequential` model. The `Sequential` model makes the assumption that the network has exactly one input and exactly one output, and that it consists of a linear stack of layers (see figure 7.1).

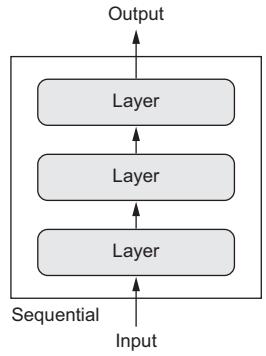


Figure 7.1 A `Sequential` model: a linear stack of layers

This is a commonly verified assumption; the configuration is so common that we've been able to cover many topics and practical applications in these pages so far using only the `Sequential` model class. But this set of assumptions is too inflexible in a number of cases. Some networks require several independent inputs, others require multiple outputs, and some networks have internal branching between layers that makes them look like *graphs* of layers rather than linear stacks of layers.

Some tasks, for instance, require *multimodal* inputs: they merge data coming from different input sources, processing each type of data using different kinds of neural layers. Imagine a deep-learning model trying to predict the most likely market price of a second-hand piece of clothing, using the following inputs: user-provided metadata (such as the item's brand, age, and so on), a user-provided text description, and a picture of the item. If you had only the metadata available, you could one-hot encode it and use a densely connected network to predict the price. If you had only the text description available, you could use an RNN or a 1D convnet. If you had only the picture, you could use a 2D convnet. But how can you use all three at the same time? A naive approach would be to train three separate models and then do a weighted average of their predictions. But this may be suboptimal, because the information extracted by the models may be redundant. A better way is to *jointly* learn a more accurate model of the data by using a model that can see all available input modalities simultaneously: a model with three input branches (see figure 7.2).

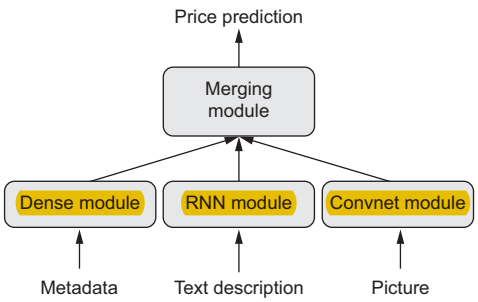


Figure 7.2 A multi-input model

Similarly, some tasks need to predict multiple target attributes of input data. Given the text of a novel or short story, you might want to automatically classify it by genre (such as romance or thriller) but also predict the approximate date it was written. Of course, you could train two separate models: one for the genre and one for the date. But because these attributes aren't statistically independent, you could build a better model by learning to jointly predict both genre and date at the same time. Such a joint model would then have two outputs, or *heads* (see figure 7.3). Due to correlations between genre and date, knowing the date of a novel would help the model learn rich, accurate representations of the space of novel genres, and vice versa.

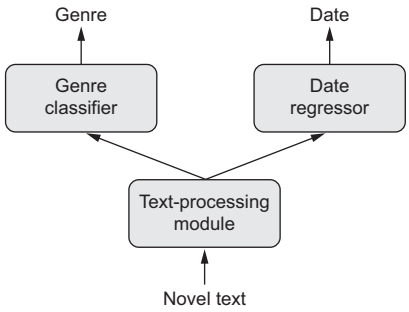


Figure 7.3 A multi-output (or multihead) model

Additionally, many recently developed neural architectures require nonlinear network topology: networks structured as directed acyclic graphs. The Inception family of networks (developed by Szegedy et al. at Google),[1] for instance, relies on *Inception modules*, where the input is processed by several parallel convolutional branches whose outputs are then merged back into a single tensor (see figure 7.4). There's also the recent trend of adding *residual connections* to a model, which started with the ResNet family of networks (developed by He et al. at Microsoft).[2] A residual connection consists of reinjecting previous representations into the downstream flow of data by adding a past output tensor to a later output tensor (see figure 7.5), which helps prevent information loss along the data-processing flow. There are many other examples of such graph-like networks.

[1] Christian Szegedy et al., "Going Deeper with Convolutions," Conference on Computer Vision and Pattern Recognition (2014), https://arxiv.org/abs/1409.4842.

[2] Kaiming He et al., "Deep Residual Learning for Image Recognition," Conference on Computer Vision and Pattern Recognition (2015), https://arxiv.org/abs/1512.03385.

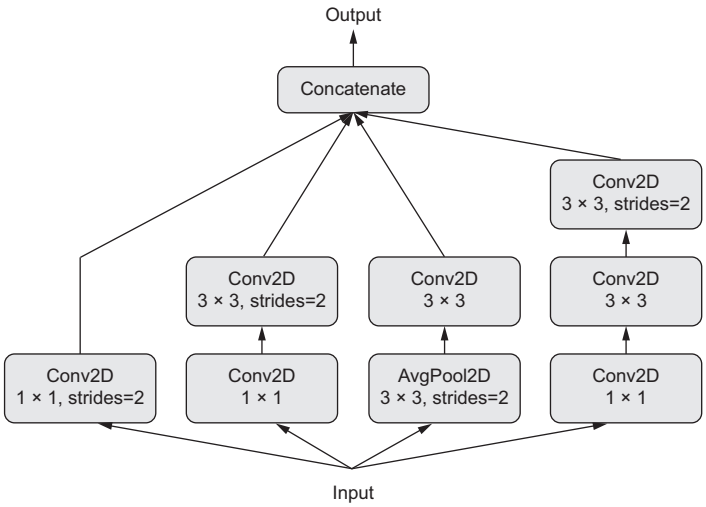


Figure 7.4 An Inception module: a subgraph of layers with several parallel convolutional branches

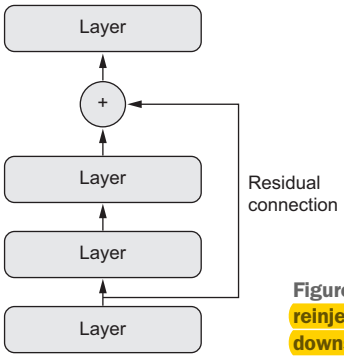


Figure 7.5 A residual connection: reinjection of prior information downstream via feature-map addition

These three important use cases—multi-input models, multi-output models, and graph-like models—aren't possible when using only the `Sequential` model class in Keras. But there's another far more general and flexible way to use Keras: the *functional API*. This section explains in detail what it is, what it can do, and how to use it.

7.1.1 Introduction to the functional API

In the functional API, you directly manipulate tensors, and you use layers as *functions* that take tensors and return tensors (hence, the name *functional API*):

```
from keras import Input, layers
input_tensor = Input(shape=(32,))
```

◁— A tensor

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

A layer is a function.

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

Let's start with a minimal example that shows side by side a simple `Sequential` model and its equivalent in the functional API:

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

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'))

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)
model.summary()
```

Sequential model, which you already know about

Its functional equivalent

The Model class turns an input tensor and output tensor into a model.

Let's look at it!

This is what the call to `model.summary()` displays:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 64)                0
_________________________________________________________________
dense_1 (Dense)              (None, 32)                2080
_________________________________________________________________
dense_2 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_3 (Dense)              (None, 10)                330
=================================================================
Total params: 3,466
Trainable params: 3,466
Non-trainable params: 0
_________________________________________________________________
```

The only part that may seem a bit magical at this point is instantiating a `Model` object using only an input tensor and an output tensor. Behind the scenes, Keras retrieves every layer involved in going from `input_tensor` to `output_tensor`, bringing them together into a graph-like data structure—a `Model`. Of course, the reason it works is that `output_tensor` was obtained by repeatedly transforming `input_tensor`. If you tried to build a model from inputs and outputs that weren't related, you'd get a `RuntimeError`:

```
>>> unrelated_input = Input(shape=(32,))
>>> bad_model = model = Model(unrelated_input, output_tensor)
```

```
RuntimeError: Graph disconnected: cannot
obtain value for tensor
➥Tensor("input_1:0", shape=(?, 64), dtype=float32) at layer "input_1".
```

This error tells you, in essence, that Keras couldn't reach `input_1` from the provided output tensor.

When it comes to compiling, training, or evaluating such an instance of `Model`, the API is the same as that of `Sequential`:

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

Compiles the model

```
import numpy as np
x_train = np.random.random((1000, 64))
y_train = np.random.random((1000, 10))
```

Generates dummy Numpy data to train on

```
model.fit(x_train, y_train, epochs=10, batch_size=128)
```

Trains the model for 10 epochs

```
score = model.evaluate(x_train, y_train)
```

Evaluates the model

7.1.2 Multi-input models

The functional API can be used to build models that have multiple inputs. Typically, such models at some point merge their different input branches using a layer that can combine several tensors: by adding them, concatenating them, and so on. This is usually done via a Keras merge operation such as `keras.layers.add`, `keras.layers.concatenate`, and so on. Let's look at a very simple example of a multi-input model: a question-answering model.

A typical question-answering model has two inputs: a natural-language question and a text snippet (such as a news article) providing information to be used for answering the question. The model must then produce an answer: in the simplest possible setup, this is a one-word answer obtained via a softmax over some predefined vocabulary (see figure 7.6).

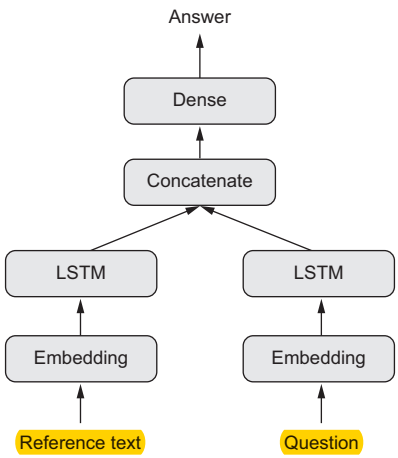


Figure 7.6 A question-answering model

Following is an example of how you can build such a model with the functional API. You set up two independent branches, encoding the text input and the question input as representation vectors; then, concatenate these vectors; and finally, add a softmax classifier on top of the concatenated representations.

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.

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.

Listing 7.2 Feeding data to a multi-input model

```
import numpy as np

num_samples = 1000
max_length = 100

text = np.random.randint(1, text_vocabulary_size,
                         size=(num_samples, max_length))
```

Generates dummy Numpy data

```
question = np.random.randint(1, question_vocabulary_size,
                             size=(num_samples, max_length))
answers = np.random.randint(0, 1,
                            size=(num_samples, answer_vocabulary_size))

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

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

Answers are one-hot encoded, not integers

Fitting using a list of inputs

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

7.1.3 Multi-output models

In the same way, you can use the functional API to build models with multiple outputs (or multiple *heads*). A simple example is a network that attempts to simultaneously predict different properties of the data, such as a network that takes as input a series of social media posts from a single anonymous person and tries to predict attributes of that person, such as age, gender, and income level (see figure 7.7).

Listing 7.3 Functional API implementation of a three-output model

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

vocabulary_size = 50000
num_income_groups = 10

posts_input = Input(shape=(None,), dtype='int32', name='posts')
embedded_posts = layers.Embedding(256, vocabulary_size)(posts_input)
x = layers.Conv1D(128, 5, activation='relu')(embedded_posts)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.MaxPooling1D(5)(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.Conv1D(256, 5, activation='relu')(x)
x = layers.GlobalMaxPooling1D()(x)
x = layers.Dense(128, activation='relu')(x)

age_prediction = layers.Dense(1, name='age')(x)
income_prediction = layers.Dense(num_income_groups,
                                 activation='softmax',
                                 name='income')(x)
gender_prediction = layers.Dense(1, activation='sigmoid', name='gender')(x)

model = Model(posts_input,
              [age_prediction, income_prediction, gender_prediction])
```

Note that the output layers are given names.

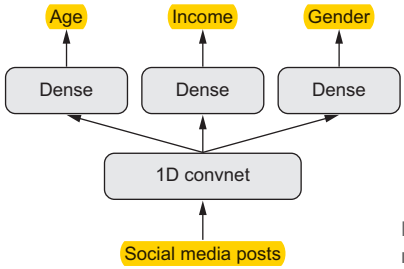


Figure 7.7 A social media model with three heads

Importantly, training such a model requires the ability to specify different loss functions for different heads of the network: for instance, age prediction is a scalar regression task, but gender prediction is a binary classification task, requiring a different training procedure. But because gradient descent requires you to minimize a *scalar*, you must combine these losses into a single value in order to train the model. The simplest way to combine different losses is to sum them all. In Keras, you can use either a list or a dictionary of losses in `compile` to specify different objects for different outputs; the resulting loss values are summed into a global loss, which is minimized during training.

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)

Note that very imbalanced loss contributions will cause the model representations to be optimized preferentially for the task with the largest individual loss, at the expense of the other tasks. To remedy this, you can assign different levels of importance to the loss values in their contribution to the final loss. This is useful in particular if the losses' values use different scales. For instance, the mean squared error (MSE) loss used for the age-regression task typically takes a value around 3–5, whereas the cross-entropy loss used for the gender-classification task can be as low as 0.1. In such a situation, to balance the contribution of the different losses, you can assign a weight of 10 to the crossentropy loss and a weight of 0.25 to the MSE loss.

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)

Much as in the case of multi-input models, you can pass Numpy data to the model for training either via a list of arrays or via a dictionary of arrays.

Listing 7.6 Feeding data to a multi-output model

```
model.fit(posts, [age_targets, income_targets, gender_targets],
          epochs=10, batch_size=64)
```

```
model.fit(posts, {'age': age_targets,
                  'income': income_targets,
                  'gender': gender_targets},
          epochs=10, batch_size=64)
```

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

age_targets, income_targets, and gender_targets are assumed to be Numpy arrays.

7.1.4 Directed acyclic graphs of layers

With the functional API, not only can you build models with multiple inputs and multiple outputs, but you can also implement networks with a complex internal topology. Neural networks in Keras are allowed to be arbitrary *directed acyclic graphs* of layers. The qualifier *acyclic* is important: these graphs can't have cycles. It's impossible for a tensor `x` to become the input of one of the layers that generated `x`. The only processing *loops* that are allowed (that is, recurrent connections) are those internal to recurrent layers.

Several common neural-network components are implemented as graphs. Two notable ones are Inception modules and residual connections. To better understand how the functional API can be used to build graphs of layers, let's take a look at how you can implement both of them in Keras.

INCEPTION MODULES

Inception[3] is a popular type of network architecture for convolutional neural networks; it was developed by Christian Szegedy and his colleagues at Google in 2013–2014, inspired by the earlier *network-in-network* architecture.[4] It consists of a stack of modules that themselves look like small independent networks, split into several parallel branches. The most basic form of an Inception module has three to four branches starting with a 1 × 1 convolution, followed by a 3 × 3 convolution, and ending with the concatenation of the resulting features. This setup helps the network separately learn

[3] https://arxiv.org/abs/1409.4842.

[4] Min Lin, Qiang Chen, and Shuicheng Yan, "Network in Network," International Conference on Learning Representations (2013), https://arxiv.org/abs/1312.4400.

spatial features and channel-wise features, which is more efficient than learning them jointly. More-complex versions of an Inception module are also possible, typically involving pooling operations, different spatial convolution sizes (for example, 5 × 5 instead of 3 × 3 on some branches), and branches without a spatial convolution (only a 1 × 1 convolution). An example of such a module, taken from Inception V3, is shown in figure 7.8.

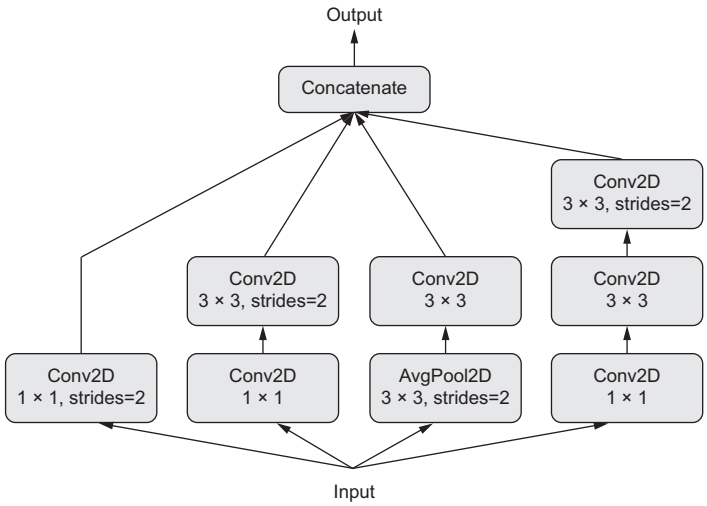
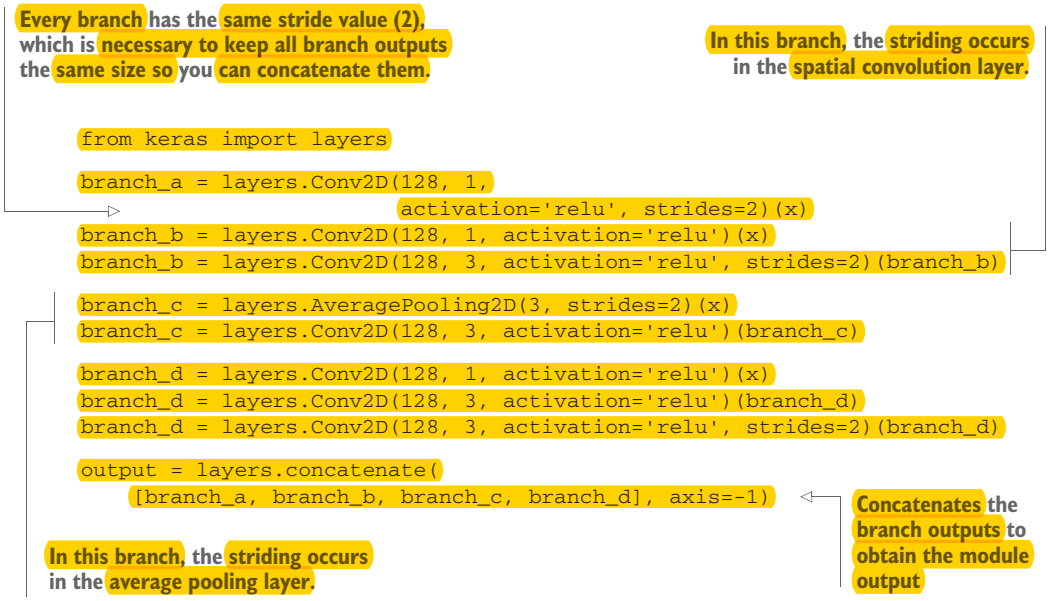


Figure 7.8 An Inception module

The purpose of 1 × 1 convolutions

You already know that convolutions extract spatial patches around every tile in an input tensor and apply the same transformation to each patch. An edge case is when the patches extracted consist of a single tile. The convolution operation then becomes equivalent to running each tile vector through a `Dense` layer: it will compute features that mix together information from the channels of the input tensor, but it won't mix information across space (because it's looking at one tile at a time). Such 1 × 1 convolutions (also called *pointwise convolutions*) are featured in Inception modules, where they contribute to factoring out channel-wise feature learning and space-wise feature learning—a reasonable thing to do if you assume that each channel is highly autocorrelated across space, but different channels may not be highly correlated with each other.

Here's how you'd implement the module featured in figure 7.8 using the functional API. This example assumes the existence of a 4D input tensor `x`:



Note that the full Inception V3 architecture is available in Keras as `keras.applications.inception_v3.InceptionV3`, including weights pretrained on the ImageNet dataset. Another closely related model available as part of the Keras applications module is *Xception*.[5] Xception, which stands for *extreme inception*, is a convnet architecture loosely inspired by Inception. It takes the idea of separating the learning of channel-wise and space-wise features to its logical extreme, and replaces Inception modules with depthwise separable convolutions consisting of a depthwise convolution (a spatial convolution where every input channel is handled separately) followed by a pointwise convolution (a 1 × 1 convolution)—effectively, an extreme form of an Inception module, where spatial features and channel-wise features are fully separated. Xception has roughly the same number of parameters as Inception V3, but it shows better runtime performance and higher accuracy on ImageNet as well as other large-scale datasets, due to a more efficient use of model parameters.

RESIDUAL CONNECTIONS

Residual connections are a common graph-like network component found in many post-2015 network architectures, including Xception. They were introduced by He et al. from Microsoft in their winning entry in the ILSVRC ImageNet challenge in late 2015.[6] They tackle two common problems that plague any large-scale deep-learning model: vanishing gradients and representational bottlenecks. In general, adding residual connections to any model that has more than 10 layers is likely to be beneficial.

[5] François Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," Conference on Computer Vision and Pattern Recognition (2017), https://arxiv.org/abs/1610.02357.

[6] He et al., "Deep Residual Learning for Image Recognition," https://arxiv.org/abs/1512.03385.

A residual connection consists of making the output of an earlier layer available as input to a later layer, effectively creating a shortcut in a sequential network. Rather than being concatenated to the later activation, the earlier output is summed with the later activation, which assumes that both activations are the same size. If they're different sizes, you can use a linear transformation to reshape the earlier activation into the target shape (for example, a `Dense` layer without an activation or, for convolutional feature maps, a 1 × 1 convolution without an activation).

Here's how to implement a residual connection in Keras when the feature-map sizes are the same, using identity residual connections. This example assumes the existence of a 4D input tensor `x`:

```
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

And the following implements a residual connection when the feature-map sizes differ, using a linear residual connection (again, assuming the existence of a 4D input tensor `x`):

```
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

Representational bottlenecks in deep learning

In a `Sequential` model, each successive representation layer is built on top of the previous one, which means it only has access to information contained in the activation of the previous layer. If one layer is too small (for example, it has features that are too low-dimensional), then the model will be constrained by how much information can be crammed into the activations of this layer.

(continued)

You can grasp this concept with a signal-processing analogy: if you have an audio-processing pipeline that consists of a series of operations, each of which takes as input the output of the previous operation, then if one operation crops your signal to a low-frequency range (for example, 0–15 kHz), the operations downstream will never be able to recover the dropped frequencies. Any loss of information is permanent. Residual connections, by reinjecting earlier information downstream, partially solve this issue for deep-learning models.

Vanishing gradients in deep learning

Backpropagation, the master algorithm used to train deep neural networks, works by propagating a feedback signal from the output loss down to earlier layers. If this feedback signal has to be propagated through a deep stack of layers, the signal may become tenuous or even be lost entirely, rendering the network untrainable. This issue is known as *vanishing gradients*.

This problem occurs both with deep networks and with recurrent networks over very long sequences—in both cases, a feedback signal must be propagated through a long series of operations. You're already familiar with the solution that the `LSTM` layer uses to address this problem in recurrent networks: it introduces a *carry track* that propagates information parallel to the main processing track. Residual connections work in a similar way in feedforward deep networks, but they're even simpler: they introduce a purely linear information carry track parallel to the main layer stack, thus helping to propagate gradients through arbitrarily deep stacks of layers.

7.1.5 Layer weight sharing

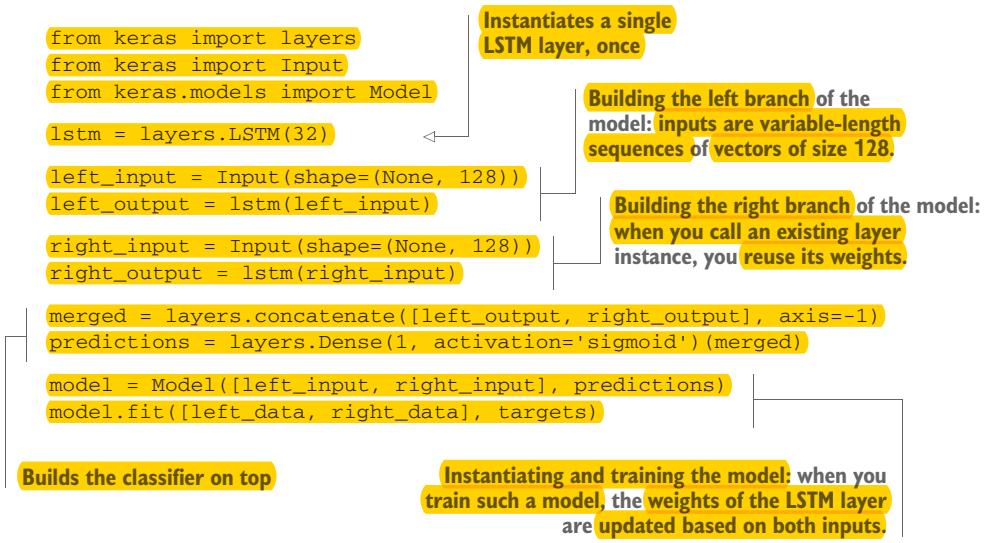
One more important feature of the functional API is the ability to reuse a layer instance several times. When you call a layer instance twice, instead of instantiating a new layer for each call, you reuse the same weights with every call. This allows you to build models that have shared branches—several branches that all share the same knowledge and perform the same operations. That is, they share the same representations and learn these representations simultaneously for different sets of inputs.

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.

In this setup, the two input sentences are interchangeable, because semantic similarity is a symmetrical relationship: the similarity of A to B is identical to the similarity of B to A. For this reason, it wouldn't make sense to learn two independent models for

processing each input sentence. Rather, you want to process both with a single `LSTM` layer. The representations of this `LSTM` layer (its weights) are learned based on both inputs simultaneously. This is what we call a *Siamese LSTM* model or a *shared LSTM*.

Here's how to implement such a model using layer sharing (layer reuse) in the Keras functional API:




```
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)
```

Naturally, a layer instance may be used more than once—it can be called arbitrarily many times, reusing the same set of weights every time.

7.1.6 Models as layers

Importantly, in the functional API, models can be used as you'd use layers—effectively, you can think of a model as a "bigger layer." This is true of both the `Sequential` and `Model` classes. This means you can call a model on an input tensor and retrieve an output tensor:

```
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])
```

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.

One simple practical example of what you can build by reusing a model instance is a vision model that uses a dual camera as its input: two parallel cameras, a few centimeters (one inch) apart. Such a model can perceive depth, which can be useful in many applications. You shouldn't need two independent models to extract visual

features from the left camera and the right camera before merging the two feeds. Such low-level processing can be shared across the two inputs: that is, done via layers that use the same weights and thus share the same representations. Here's how you'd implement a Siamese vision model (shared convolutional base) in Keras:

```
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_input = 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.

7.1.7 Wrapping up

This concludes our introduction to the Keras functional API—an essential tool for building advanced deep neural network architectures. Now you know the following:

- To step out of the `Sequential` API whenever you need anything more than a linear stack of layers
- How to build Keras models with several inputs, several outputs, and complex internal network topology, using the Keras functional API
- How to reuse the weights of a layer or model across different processing branches, by calling the same layer or model instance several times

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

In this section, we'll review ways to gain greater access to and control over what goes on inside your model during training. Launching a training run on a large dataset for tens of epochs using `model.fit()` or `model.fit_generator()` can be a bit like launching a paper airplane: past the initial impulse, you don't have any control over its trajectory or its landing spot. If you want to avoid bad outcomes (and thus wasted paper airplanes), it's smarter to use not a paper plane, but a drone that can sense its environment, send data back to its operator, and automatically make steering decisions based on its current state. The techniques we present here will transform the call to `model.fit()` from a paper airplane into a smart, autonomous drone that can self-introspect and dynamically take action.

7.2.1 Using callbacks to act on a model during training

When you're training a model, there are many things you can't predict from the start. In particular, you can't tell how many epochs will be needed to get to an optimal validation loss. The examples so far have adopted the strategy of training for enough epochs that you begin overfitting, using the first run to figure out the proper number of epochs to train for, and then finally launching a new training run from scratch using this optimal number. Of course, this approach is wasteful.

A much better way to handle this is to stop training when you measure that the validation loss in no longer improving. This can be achieved using a Keras callback. A *callback* is an object (a class instance implementing specific methods) that is passed to the model in the call to `fit` and that is called by the model at various points during training. It has access to all the available data about the state of the model and its performance, and it can take action: interrupt training, save a model, load a different weight set, or otherwise alter the state of the model.

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!

The `keras.callbacks` module includes a number of built-in callbacks (this is not an exhaustive list):

```
keras.callbacks.ModelCheckpoint
keras.callbacks.EarlyStopping
```

```
keras.callbacks.LearningRateScheduler
keras.callbacks.ReduceLROnPlateau
keras.callbacks.CSVLogger
```

Let's review a few of them to give you an idea of how to use them: `ModelCheckpoint`, `EarlyStopping`, and `ReduceLROnPlateau`.

The ModelCheckpoint and EarlyStopping callbacks

You can use the `EarlyStopping` callback to interrupt training once a target metric being monitored has stopped improving for a fixed number of epochs. For instance, this callback allows you to interrupt training as soon as you start overfitting, thus avoiding having to retrain your model for a smaller number of epochs. This callback is typically used in combination with `ModelCheckpoint`, which lets you continually save the model during training (and, optionally, save only the current best model so far: the version of the model that achieved the best performance at the end of an epoch):

Callbacks are passed to the model via the callbacks argument in fit, which takes a list of callbacks. You can pass any number of callbacks.

Interrupts training when improvement stops

Monitors the model's validation accuracy

Interrupts training when accuracy has stopped improving for more than one epoch (that is, two epochs)

Saves the current weights after every epoch

Path to the destination model file

These two arguments mean you won't overwrite the model file unless val_loss has improved, which allows you to keep the best model seen during training.

You monitor accuracy, so it should be part of the model's metrics.

Note that because the callback will monitor validation loss and validation accuracy, you need to pass validation_data to the call to fit.

```
import keras

callbacks_list = [
    keras.callbacks.EarlyStopping(
        monitor='acc',
        patience=1,
    ),
    keras.callbacks.ModelCheckpoint(
        filepath='my_model.h5',
        monitor='val_loss',
        save_best_only=True,
    )
]

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

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

The ReduceLROnPlateau callback

You can use this callback to reduce the learning rate when the validation loss has stopped improving. Reducing or increasing the learning rate in case of a *loss plateau* is is an effective strategy to get out of local minima during training. The following example uses the `ReduceLROnPlateau` callback:

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

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.

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

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

WRITING YOUR OWN CALLBACK

If you need to take a specific action during training that isn't covered by one of the built-in callbacks, you can write your own callback. Callbacks are implemented by subclassing the class `keras.callbacks.Callback`. You can then implement any number of the following transparently named methods, which are called at various points during training:

```
on_epoch_begin      <-- Called at the start of every epoch
on_epoch_end        <-- Called at the end of every epoch

on_batch_begin      <-- Called right before processing each batch
on_batch_end        <-- Called right after processing each batch

on_train_begin      <-- Called at the start of training
on_train_end        <-- Called at the end of training
```

These methods all are called with a `logs` argument, which is a dictionary containing information about the previous batch, epoch, or training run: training and validation metrics, and so on. Additionally, the callback has access to the following attributes:

- `self.model`—The model instance from which the callback is being called
- `self.validation_data`—The value of what was passed to `fit` as validation data

Here's a simple example of a custom callback that saves to disk (as Numpy arrays) the activations of every layer of the model at the end of every epoch, computed on the first sample of the validation set:

```
import keras
import numpy as np

class ActivationLogger(keras.callbacks.Callback):

    def set_model(self, model):
        self.model = model
        layer_outputs = [layer.output for layer in model.layers]
        self.activations_model = keras.models.Model(model.input,
                                                    layer_outputs)

    def on_epoch_end(self, epoch, logs=None):
        if self.validation_data is None:
            raise RuntimeError('Requires validation_data.')
```

Called by the parent model before training, to inform the callback of what model will be calling it

Model instance that returns the activations of every layer

```
validation_sample = self.validation_data[0][0:1]
activations = self.activations_model.predict(validation_sample)
f = open('activations_at_epoch_' + str(epoch) + '.npz', 'w')
np.savez(f, activations)
f.close()
```

Obtains the first input sample of the validation data

Saves arrays to disk

This is all you need to know about callbacks—the rest is technical details, which you can easily look up. Now you're equipped to perform any sort of logging or preprogrammed intervention on a Keras model during training.

7.2.2 Introduction to TensorBoard: the TensorFlow visualization framework

To do good research or develop good models, you need rich, frequent feedback about what's going on inside your models during your experiments. That's the point of running experiments: to get information about how well a model performs—as much information as possible. Making progress is an iterative process, or loop: you start with an idea and express it as an experiment, attempting to validate or invalidate your idea. You run this experiment and process the information it generates. This inspires your next idea. The more iterations of this loop you're able to run, the more refined and powerful your ideas become. Keras helps you go from idea to experiment in the least possible time, and fast GPUs can help you get from experiment to result as quickly as possible. But what about processing the experiment results? That's where TensorBoard comes in.

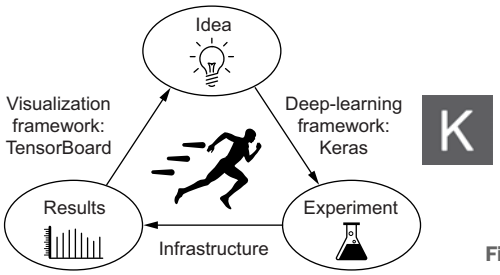


Figure 7.9 The loop of progress

This section introduces TensorBoard, a browser-based visualization tool that comes packaged with TensorFlow. Note that it's only available for Keras models when you're using Keras with the TensorFlow backend.

The key purpose of TensorBoard is to help you visually monitor everything that goes on inside your model during training. If you're monitoring more information than just the model's final loss, you can develop a clearer vision of what the model does and doesn't do, and you can make progress more quickly. TensorBoard gives you access to several neat features, all in your browser:

7.3 Getting the most out of your models

Trying out architectures blindly works well enough if you just need something that works okay. In this section, we'll go beyond "works okay" to "works great and wins machine-learning competitions" by offering you a quick guide to a set of must-know techniques for building state-of-the-art deep-learning models.

7.3.1 Advanced architecture patterns

We covered one important design pattern in detail in the previous section: residual connections. There are two more design patterns you should know about: normalization and depthwise separable convolution. These patterns are especially relevant when you're building high-performing deep convnets, but they're commonly found in many other types of architectures as well.

BATCH NORMALIZATION

Normalization is a broad category of methods that seek to make different samples seen by a machine-learning model more similar to each other, which helps the model learn and generalize well to new data. The most common form of data normalization is one you've seen several times in this book already: centering the data on 0 by subtracting the mean from the data, and giving the data a unit standard deviation by dividing the data by its standard deviation. In effect, this makes the assumption that the data follows a normal (or Gaussian) distribution and makes sure this distribution is centered and scaled to unit variance:

```
normalized_data = (data - np.mean(data, axis=...)) / np.std(data, axis=...)
```

Previous examples normalized data before feeding it into models. But data normalization should be a concern after every transformation operated by the network: even if the data entering a `Dense` or `Conv2D` network has a 0 mean and unit variance, there's no reason to expect a priori that this will be the case for the data coming out.

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.

[7] Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *Proceedings of the 32nd International Conference on Machine Learning* (2015), https://arxiv.org/abs/1502.03167.

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

```
conv_model.add(layers.Conv2D(32, 3, activation='relu'))
conv_model.add(layers.BatchNormalization())

dense_model.add(layers.Dense(32, activation='relu'))
dense_model.add(layers.BatchNormalization())
```

After a Conv layer

After a Dense layer

The `BatchNormalization` layer takes an `axis` argument, which specifies the feature axis that should be normalized. This argument defaults to -1, the last axis in the input tensor. This is the correct value when using `Dense` layers, `Conv1D` layers, `RNN` layers, and `Conv2D` layers with `data_format` set to `"channels_last"`. But in the niche use case of `Conv2D` layers with `data_format` set to `"channels_first"`, the features axis is axis 1; the `axis` argument in `BatchNormalization` should accordingly be set to 1.

Batch renormalization

A recent improvement over regular batch normalization is *batch renormalization*, introduced by Ioffe in 2017.[a] It offers clears benefits over batch normalization, at no apparent cost. At the time of writing, it's too early to tell whether it will supplant batch normalization—but I think it's likely. Even more recently, Klambauer et al. introduced *self-normalizing neural networks*,[b] which manage to keep data normalized after going through any `Dense` layer by using a specific activation function (`selu`) and a specific initializer (`lecun_normal`). This scheme, although highly interesting, is limited to densely connected networks for now, and its usefulness hasn't yet been broadly replicated.

[a] Sergey Ioffe, "Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models" (2017), https://arxiv.org/abs/1702.03275.

[b] Günter Klambauer et al., "Self-Normalizing Neural Networks," Conference on Neural Information Processing Systems (2017), https://arxiv.org/abs/1706.02515.

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. This is equivalent to separating the learning of spatial features and the learning of channel-wise features, which makes a lot of sense if you assume that spatial locations in the input are highly correlated, but different channels are fairly independent. It requires significantly fewer parameters and involves fewer computations, thus resulting in smaller, speedier models. And because it's a more representationally efficient way to perform convolution, it tends to learn better representations using less data, resulting in better-performing models.

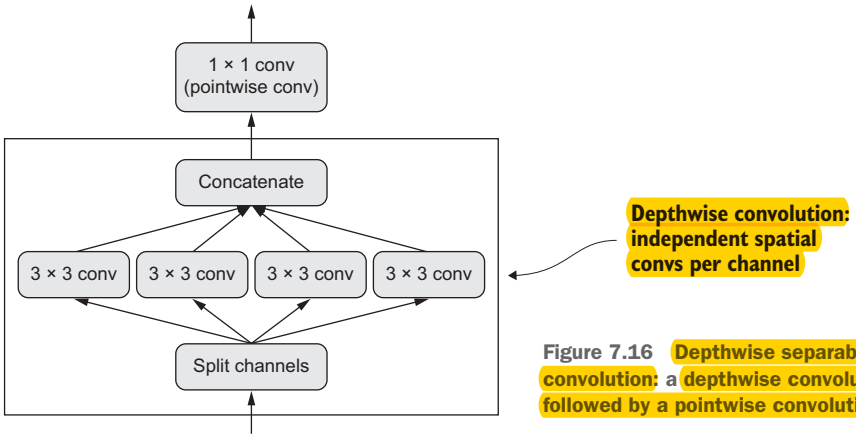


Figure 7.16 Depthwise separable convolution: a depthwise convolution followed by a pointwise convolution

These advantages become especially important when you're training small models from scratch on limited data. For instance, here's how you can build a lightweight, depthwise separable convnet for an image-classification task (softmax categorical classification) on a small dataset:

```
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')
```

When it comes to larger-scale models, depthwise separable convolutions are the basis of the Xception architecture, a high-performing convnet that comes packaged with Keras. You can read more about the theoretical grounding for depthwise separable

convolutions and Xception in my paper "Xception: Deep Learning with Depthwise Separable Convolutions."[8]

7.3.2 Hyperparameter optimization

When building a deep-learning model, you have to make many seemingly arbitrary decisions: How many layers should you stack? How many units or filters should go in each layer? Should you use `relu` as activation, or a different function? Should you use `BatchNormalization` after a given layer? How much dropout should you use? And so on. These architecture-level parameters are called *hyperparameters* to distinguish them from the parameters of a model, which are trained via backpropagation.

In practice, experienced machine-learning engineers and researchers build intuition over time as to what works and what doesn't when it comes to these choices—they develop hyperparameter-tuning skills. But there are no formal rules. If you want to get to the very limit of what can be achieved on a given task, you can't be content with arbitrary choices made by a fallible human. Your initial decisions are almost always suboptimal, even if you have good intuition. You can refine your choices by tweaking them by hand and retraining the model repeatedly—that's what machine-learning engineers and researchers spend most of their time doing. But it shouldn't be your job as a human to fiddle with hyperparameters all day—that is better left to a machine.

Thus you need to explore the space of possible decisions automatically, systematically, in a principled way. You need to search the architecture space and find the best-performing ones empirically. That's what the field of automatic hyperparameter optimization is about: it's an entire field of research, and an important one.

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.

The key to this process is the algorithm that uses this history of validation performance, given various sets of hyperparameters, to choose the next set of hyperparameters to evaluate. Many different techniques are possible: Bayesian optimization, genetic algorithms, simple random search, and so on.

Training the weights of a model is relatively easy: you compute a loss function on a mini-batch of data and then use the Backpropagation algorithm to move the weights

[8] See note 5 above.

in the right direction. Updating hyperparameters, on the other hand, is extremely challenging. Consider the following:

- Computing the feedback signal (does this set of hyperparameters lead to a high-performing model on this task?) can be extremely expensive: it requires creating and training a new model from scratch on your dataset.
- The hyperparameter space is typically made of discrete decisions and thus isn't continuous or differentiable. Hence, you typically can't do gradient descent in hyperparameter space. Instead, you must rely on gradient-free optimization techniques, which naturally are far less efficient than gradient descent.

Because these challenges are difficult and the field is still young, we currently only have access to very limited tools to optimize models. Often, it turns out that random search (choosing hyperparameters to evaluate at random, repeatedly) is the best solution, despite being the most naive one. But one tool I have found reliably better than random search is Hyperopt (https://github.com/hyperopt/hyperopt), a Python library for hyperparameter optimization that internally uses trees of Parzen estimators to predict sets of hyperparameters that are likely to work well. Another library called Hyperas (https://github.com/maxpumperla/hyperas) integrates Hyperopt for use with Keras models. Do check it out.

> **NOTE** One important issue to keep in mind when doing automatic hyperparameter optimization at scale is validation-set overfitting. Because you're updating hyperparameters based on a signal that is computed using your validation data, you're effectively training them on the validation data, and thus they will quickly overfit to the validation data. Always keep this in mind.

Overall, hyperparameter optimization is a powerful technique that is an absolute requirement to get to state-of-the-art models on any task or to win machine-learning competitions. Think about it: once upon a time, people handcrafted the features that went into shallow machine-learning models. That was very much suboptimal. Now, deep learning automates the task of hierarchical feature engineering—features are learned using a feedback signal, not hand-tuned, and that's the way it should be. In the same way, you shouldn't handcraft your model architectures; you should optimize them in a principled way. At the time of writing, the field of automatic hyperparameter optimization is very young and immature, as deep learning was some years ago, but I expect it to boom in the next few years.

7.3.3 Model ensembling

Another powerful technique for obtaining the best possible results on a task is *model ensembling*. Ensembling consists of pooling together the predictions of a set of different models, to produce better predictions. If you look at machine-learning competitions, in particular on Kaggle, you'll see that the winners use very large ensembles of models that inevitably beat any single model, no matter how good.

8.4 Generating images with variational autoencoders

Sampling from a latent space of images to create entirely new images or edit existing ones is currently the most popular and successful application of creative AI. In this section and the next, we'll review some high-level concepts pertaining to image generation, alongside implementations details relative to the two main techniques in this domain: *variational autoencoders* (VAEs) and *generative adversarial networks* (GANs). The techniques we present here aren't specific to images—you could develop latent spaces of sound, music, or even text, using GANs and VAEs—but in practice, the most interesting results have been obtained with pictures, and that's what we focus on here.

8.4.1 Sampling from latent spaces of images

The key idea of image generation is to develop a low-dimensional *latent space* of representations (which naturally is a vector space) where any point can be mapped to a realistic-looking image. The module capable of realizing this mapping, taking as input a latent point and outputting an image (a grid of pixels), is called a *generator* (in the case of GANs) or a *decoder* (in the case of VAEs). Once such a latent space has been developed, you can sample points from it, either deliberately or at random, and, by mapping them to image space, generate images that have never been seen before (see figure 8.9).

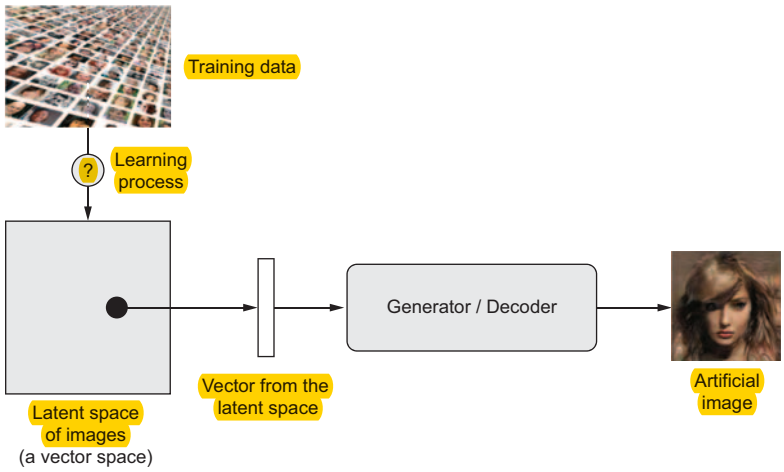


Figure 8.9 Learning a latent vector space of images, and using it to sample new images

GANs and VAEs are two different strategies for learning such latent spaces of image representations, each with its own characteristics. VAEs are great for learning latent spaces that are well structured, where specific directions encode a meaningful axis of variation in the data. GANs generate images that can potentially be highly realistic, but the latent space they come from may not have as much structure and continuity.

Figure 8.10 A continuous space of faces generated by Tom White using VAEs

8.4.2 Concept vectors for image editing

We already hinted at the idea of a *concept vector* when we covered word embeddings in chapter 6. The idea is still the same: given a latent space of representations, or an embedding space, certain directions in the space may encode interesting axes of variation in the original data. In a latent space of images of faces, for instance, there may be a *smile vector*, such that if latent point `z` is the embedded representation of a certain face, then latent point `z + s` is the embedded representation of the same face, smiling. Once you've identified such a vector, it then becomes possible to edit images by projecting them into the latent space, moving their representation in a meaningful way, and then decoding them back to image space. There are concept vectors for essentially any independent dimension of variation in image space—in the case of faces, you may discover vectors for adding sunglasses to a face, removing glasses, turning a male face into as female face, and so on. Figure 8.11 is an example of a smile vector, a concept vector discovered by Tom White from the Victoria University School of Design in New Zealand, using VAEs trained on a dataset of faces of celebrities (the CelebA dataset).

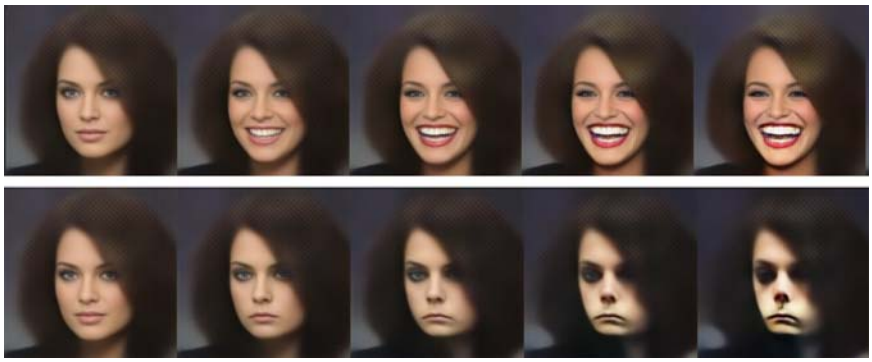


Figure 8.11 The smile vector

8.4.3 *Variational autoencoders*

Variational autoencoders, simultaneously discovered by Kingma and Welling in December 2013[6] and Rezende, Mohamed, and Wierstra in January 2014,[7] are a kind of generative model that's especially appropriate for the task of image editing via concept vectors. They're a modern take on autoencoders—a type of network that aims to encode an input to a low-dimensional latent space and then decode it back—that mixes ideas from deep learning with Bayesian inference.

A classical image autoencoder takes an image, maps it to a latent vector space via an encoder module, and then decodes it back to an output with the same dimensions as the original image, via a decoder module (see figure 8.12). It's then trained by using as target data the *same images* as the input images, meaning the autoencoder learns to reconstruct the original inputs. By imposing various constraints on the code (the output of the encoder), you can get the autoencoder to learn more-or-less interesting latent representations of the data. Most commonly, you'll constrain the code to be low-dimensional and sparse (mostly zeros), in which case the encoder acts as a way to compress the input data into fewer bits of information.

[6] Diederik P. Kingma and Max Welling, "Auto-Encoding Variational Bayes, arXiv (2013), https://arxiv.org/abs/1312.6114.

[7] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra, "Stochastic Backpropagation and Approximate Inference in Deep Generative Models," arXiv (2014), https://arxiv.org/abs/1401.4082.

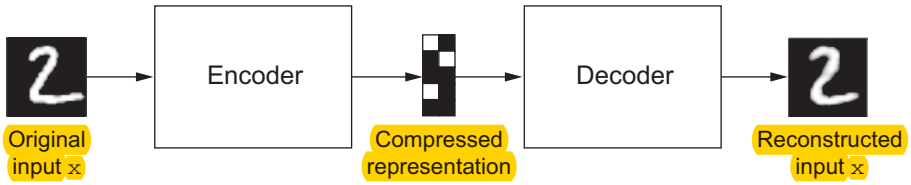


Figure 8.12 An autoencoder: mapping an input x to a compressed representation and then decoding it back as x'

In practice, such classical autoencoders don't lead to particularly useful or nicely structured latent spaces. They're not much good at compression, either. For these reasons, they have largely fallen out of fashion. VAEs, however, augment autoencoders with a little bit of statistical magic that forces them to learn continuous, highly structured latent spaces. They have turned out to be a powerful tool for image generation.

A VAE, instead of compressing its input image into a fixed code in the latent space, turns the image into the parameters of a statistical distribution: a mean and a variance. Essentially, this means you're assuming the input image has been generated by a statistical process, and that the randomness of this process should be taken into accounting during encoding and decoding. The VAE then uses the mean and variance parameters to randomly sample one element of the distribution, and decodes that element back to the original input (see figure 8.13). The stochasticity of this process improves robustness and forces the latent space to encode meaningful representations everywhere: every point sampled in the latent space is decoded to a valid output.

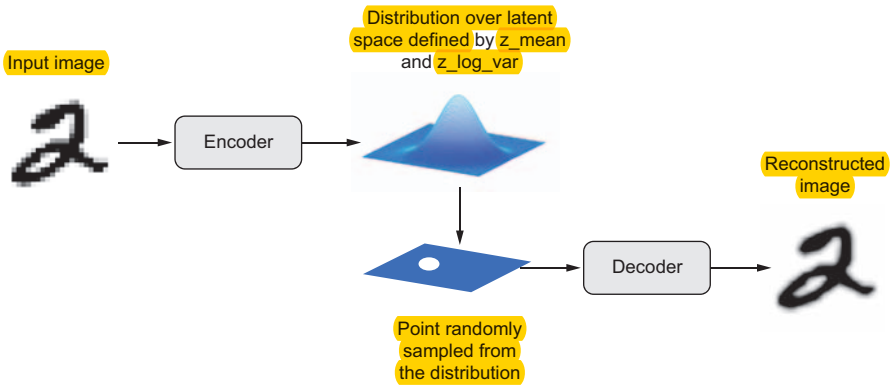


Figure 8.13 A VAE maps an image to two vectors, z_mean and z_log_sigma, which define a probability distribution over the latent space, used to sample a latent point to decode.

In technical terms, here's how a VAE works:

1. An encoder module turns the input samples `input_img` into two parameters in a latent space of representations, `z_mean` and `z_log_variance`.
2. You randomly sample a point `z` from the latent normal distribution that's assumed to generate the input image, via `z = z_mean + exp(z_log_variance) * epsilon`, where `epsilon` is a random tensor of small values.
3. A decoder module maps this point in the latent space back to the original input image.

Because `epsilon` is random, the process ensures that every point that's close to the latent location where you encoded `input_img` (`z-mean`) can be decoded to something similar to `input_img`, thus forcing the latent space to be continuously meaningful. Any two close points in the latent space will decode to highly similar images. Continuity, combined with the low dimensionality of the latent space, forces every direction in the latent space to encode a meaningful axis of variation of the data, making the latent space very structured and thus highly suitable to manipulation via concept vectors.

The parameters of a VAE are trained via two loss functions: a *reconstruction loss* that forces the decoded samples to match the initial inputs, and a *regularization loss* that helps learn well-formed latent spaces and reduce overfitting to the training data. Let's quickly go over a Keras implementation of a VAE. Schematically, it looks like this:

```
z_mean, z_log_variance = encoder(input_img)
z = z_mean + exp(z_log_variance) * epsilon
reconstructed_img = decoder(z)
model = Model(input_img, reconstructed_img)
```

Encodes the input into a mean and variance parameter

Draws a latent point using a small random epsilon

Decodes z back to an image

Instantiates the autoencoder model, which maps an input image to its reconstruction

You can then train the model using the reconstruction loss and the regularization loss.

The following listing shows the encoder network you'll use, mapping images to the parameters of a probability distribution over the latent space. It's a simple convnet that maps the input image `x` to two vectors, `z_mean` and `z_log_var`.

Listing 8.23 VAE encoder network

```
import keras
from keras import layers
from keras import backend as K
from keras.models import Model
import numpy as np

img_shape = (28, 28, 1)
batch_size = 16
latent_dim = 2

input_img = keras.Input(shape=img_shape)
```

Dimensionality of the latent space: a 2D plane

```
x = layers.Conv2D(32, 3,
                  padding='same', activation='relu')(input_img)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu',
                  strides=(2, 2))(x)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu')(x)
x = layers.Conv2D(64, 3,
                  padding='same', activation='relu')(x)
shape_before_flattening = K.int_shape(x)

x = layers.Flatten()(x)
x = layers.Dense(32, activation='relu')(x)

z_mean = layers.Dense(latent_dim)(x)
z_log_var = layers.Dense(latent_dim)(x)
```

The input image ends up being encoded into these two parameters.

Next is the code for using `z_mean` and `z_log_var`, the parameters of the statistical distribution assumed to have produced `input_img`, to generate a latent space point `z`. Here, you wrap some arbitrary code (built on top of Keras backend primitives) into a `Lambda` layer. In Keras, everything needs to be a layer, so code that isn't part of a built-in layer should be wrapped in a `Lambda` (or in a custom layer).

Listing 8.24 Latent-space-sampling function

```
def sampling(args):
    z_mean, z_log_var = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim),
                              mean=0., stddev=1.)
    return z_mean + K.exp(z_log_var) * epsilon

z = layers.Lambda(sampling)([z_mean, z_log_var])
```

The following listing shows the decoder implementation. You reshape the vector `z` to the dimensions of an image and then use a few convolution layers to obtain a final image output that has the same dimensions as the original `input_img`.

Listing 8.25 VAE decoder network, mapping latent space points to images

```
decoder_input = layers.Input(K.int_shape(z)[1:])

x = layers.Dense(np.prod(shape_before_flattening[1:]),
                 activation='relu')(decoder_input)

x = layers.Reshape(shape_before_flattening[1:])(x)

x = layers.Conv2DTranspose(32, 3,
                           padding='same',
                           activation='relu',
                           strides=(2, 2))(x)
x = layers.Conv2D(1, 3,
                  padding='same',
                  activation='sigmoid')(x)
```

Input where you'll feed z

Upsamples the input

Uses a Conv2DTranspose layer and Conv2D layer to decode z into a feature map the same size as the original image input

Reshapes z into a feature map of the same shape as the feature map just before the last Flatten layer in the encoder model

```
decoder = Model(decoder_input, x)

z_decoded = decoder(z)
```

Instantiates the decoder model, which turns "decoder_input" into the decoded image

Applies it to z to recover the decoded z

The dual loss of a VAE doesn't fit the traditional expectation of a sample-wise function of the form `loss(input, target)`. Thus, you'll set up the loss by writing a custom layer that internally uses the built-in `add_loss` layer method to create an arbitrary loss.

Listing 8.26 Custom layer used to compute the VAE loss

```
class CustomVariationalLayer(keras.layers.Layer):

    def vae_loss(self, x, z_decoded):
        x = K.flatten(x)
        z_decoded = K.flatten(z_decoded)
        xent_loss = keras.metrics.binary_crossentropy(x, z_decoded)
        kl_loss = -5e-4 * K.mean(
            1 + z_log_var - K.square(z_mean) - K.exp(z_log_var), axis=-1)
        return K.mean(xent_loss + kl_loss)

    def call(self, inputs):
        x = inputs[0]
        z_decoded = inputs[1]
        loss = self.vae_loss(x, z_decoded)
        self.add_loss(loss, inputs=inputs)
        return x

y = CustomVariationalLayer()([input_img, z_decoded])
```

You implement custom layers by writing a call method.

You don't use this output, but the layer must return something.

Calls the custom layer on the input and the decoded output to obtain the final model output

Finally, you're ready to instantiate and train the model. Because the loss is taken care of in the custom layer, you don't specify an external loss at compile time (`loss=None`), which in turn means you won't pass target data during training (as you can see, you only pass `x_train` to the model in `fit`).

Listing 8.27 Training the VAE

```
from keras.datasets import mnist

vae = Model(input_img, y)
vae.compile(optimizer='rmsprop', loss=None)
vae.summary()

(x_train, _), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_train = x_train.reshape(x_train.shape + (1,))
x_test = x_test.astype('float32') / 255.
x_test = x_test.reshape(x_test.shape + (1,))

vae.fit(x=x_train, y=None,
        shuffle=True,
        epochs=10,
        batch_size=batch_size,
        validation_data=(x_test, None))
```

Once such a model is trained—on MNIST, in this case—you can use the `decoder` network to turn arbitrary latent space vectors into images.

Listing 8.28 Sampling a grid of points from the 2D latent space and decoding them to images

```
import matplotlib.pyplot as plt
from scipy.stats import norm

n = 15
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
grid_x = norm.ppf(np.linspace(0.05, 0.95, n))
grid_y = norm.ppf(np.linspace(0.05, 0.95, n))

for i, yi in enumerate(grid_x):
    for j, xi in enumerate(grid_y):
        z_sample = np.array([[xi, yi]])
        z_sample = np.tile(z_sample, batch_size).reshape(batch_size, 2)
        x_decoded = decoder.predict(z_sample, batch_size=batch_size)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size,
               j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(10, 10))
plt.imshow(figure, cmap='Greys_r')
plt.show()
```

You'll display a grid of 15 × 15 digits (255 digits total).

Transforms linearly spaced coordinates using the SciPy ppf function to produce values of the latent variable z (because the prior of the latent space is Gaussian)

Repeats z multiple times to form a complete batch

Reshapes the first digit in the batch from 28 × 28 × 1 to 28 × 28

Decodes the batch into digit images

The grid of sampled digits (see figure 8.14) shows a completely continuous distribution of the different digit classes, with one digit morphing into another as you follow a path through latent space. Specific directions in this space have a meaning; for example, there's a direction for "four-ness," "one-ness," and so on.

In the next section, we'll cover in detail the other major tool for generating artificial images: generative adversarial networks (GANs).

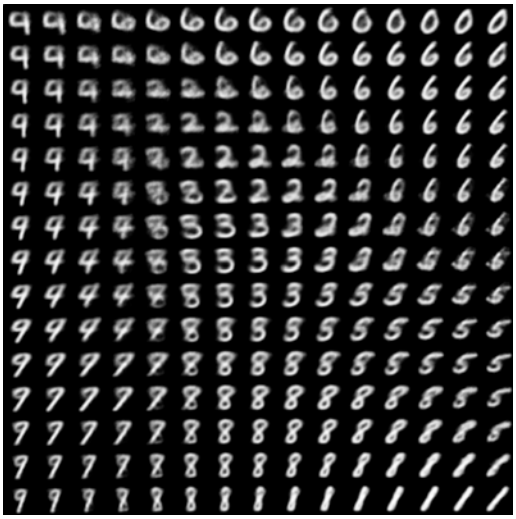


Figure 8.14 Grid of digits decoded from the latent space

8.4.4 Wrapping up

- Image generation with deep learning is done by learning latent spaces that capture statistical information about a dataset of images. By sampling and decoding points from the latent space, you can generate never-before-seen images. There are two major tools to do this: VAEs and GANs.
- VAEs result in highly structured, continuous latent representations. For this reason, they work well for doing all sorts of image editing in latent space: face swapping, turning a frowning face into a smiling face, and so on. They also work nicely for doing latent-space-based animations, such as animating a walk along a cross section of the latent space, showing a starting image slowly morphing into different images in a continuous way.
- GANs enable the generation of realistic single-frame images but may not induce latent spaces with solid structure and high continuity.

Most successful practical applications I have seen with images rely on VAEs, but GANs are extremely popular in the world of academic research—at least, circa 2016–2017. You'll find out how they work and how to implement one in the next section.

TIP To play further with image generation, I suggest working with the Large-scale Celeb Faces Attributes (CelebA) dataset. It's a free-to-download image dataset containing more than 200,000 celebrity portraits. It's great for experimenting with concept vectors in particular—it definitely beats MNIST.

8.5 Introduction to generative adversarial networks

Generative adversarial networks (GANs), introduced in 2014 by Goodfellow et al.,[8] are an alternative to VAEs for learning latent spaces of images. They enable the generation of fairly realistic synthetic images by forcing the generated images to be statistically almost indistinguishable from real ones.

An intuitive way to understand GANs is to imagine a forger trying to create a fake Picasso painting. At first, the forger is pretty bad at the task. He mixes some of his fakes with authentic Picassos and shows them all to an art dealer. The art dealer makes an authenticity assessment for each painting and gives the forger feedback about what makes a Picasso look like a Picasso. The forger goes back to his studio to prepare some new fakes. As times goes on, the forger becomes increasingly competent at imitating the style of Picasso, and the art dealer becomes increasingly expert at spotting fakes. In the end, they have on their hands some excellent fake Picassos.

That's what a GAN is: a forger network and an expert network, each being trained to best the other. As such, a GAN is made of two parts:

- *Generator network*—Takes as input a random vector (a random point in the latent space), and decodes it into a synthetic image
- *Discriminator network (or adversary)*—Takes as input an image (real or synthetic), and predicts whether the image came from the training set or was created by the generator network.

The generator network is trained to be able to fool the discriminator network, and thus it evolves toward generating increasingly realistic images as training goes on: artificial images that look indistinguishable from real ones, to the extent that it's impossible for the discriminator network to tell the two apart (see figure 8.15). Meanwhile, the discriminator is constantly adapting to the gradually improving capabilities of the generator, setting a high bar of realism for the generated images. Once training is over, the generator is capable of turning any point in its input space into a believable image. Unlike VAEs, this latent space has fewer explicit guarantees of meaningful structure; in particular, it isn't continuous.

[8] Ian Goodfellow et al., "Generative Adversarial Networks," arXiv (2014), https://arxiv.org/abs/1406.2661.

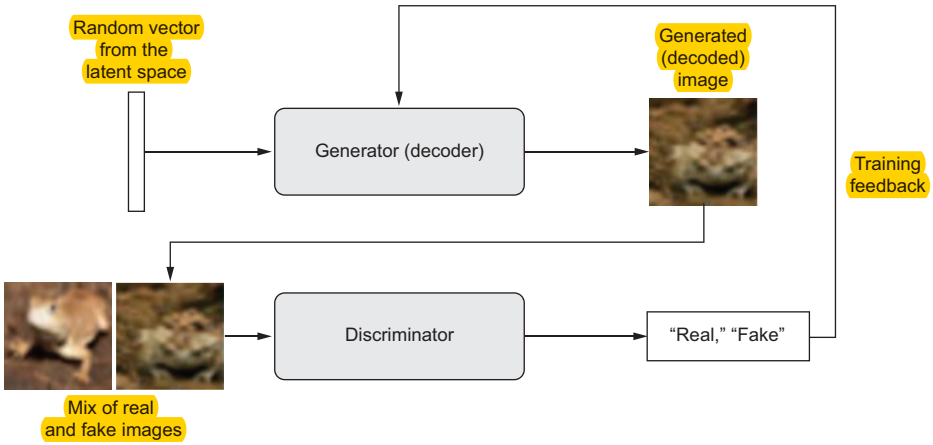


Figure 8.15 A generator transforms random latent vectors into images, and a discriminator seeks to tell real images from generated ones. The generator is trained to fool the discriminator.

Remarkably, a GAN is a system where the optimization minimum isn't fixed, unlike in any other training setup you've encountered in this book. Normally, gradient descent consists of rolling down hills in a static loss landscape. But with a GAN, every step taken down the hill changes the entire landscape a little. It's a dynamic system where the optimization process is seeking not a minimum, but an equilibrium between two forces. For this reason, GANs are notoriously difficult to train—getting a GAN to work requires lots of careful tuning of the model architecture and training parameters.

Figure 8.16 Latent space dwellers. Images generated by Mike Tyka using a multistaged GAN trained on a dataset of faces (www.miketyka.com).

8.5.1 A schematic GAN implementation

In this section, we'll explain how to implement a GAN in Keras, in its barest form—because GANs are advanced, diving deeply into the technical details would be out of scope for this book. The specific implementation is a *deep convolutional GAN* (DCGAN): a GAN where the generator and discriminator are deep convnets. In particular, it uses a `Conv2DTranspose` layer for image upsampling in the generator.

You'll train the GAN on images from CIFAR10, a dataset of 50,000 32 × 32 RGB images belonging to 10 classes (5,000 images per class). To make things easier, you'll only use images belonging to the class "frog."

Schematically, the GAN looks like this:

1. A `generator` network maps vectors of shape `(latent_dim,)` to images of shape `(32, 32, 3)`.
2. A `discriminator` network maps images of shape `(32, 32, 3)` to a binary score estimating the probability that the image is real.
3. A `gan` network chains the generator and the discriminator together: `gan(x) = discriminator(generator(x))`. Thus this `gan` network maps latent space vectors to the discriminator's assessment of the realism of these latent vectors as decoded by the generator.
4. You train the discriminator using examples of real and fake images along with "real"/"fake" labels, just as you train any regular image-classification model.
5. To train the generator, you use the gradients of the generator's weights with regard to the loss of the `gan` model. This means, at every step, you move the weights of the generator in a direction that makes the discriminator more likely to classify as "real" the images decoded by the generator. In other words, you train the generator to fool the discriminator.

8.5.2 A bag of tricks

The process of training GANs and tuning GAN implementations is notoriously difficult. There are a number of known tricks you should keep in mind. Like most things in deep learning, it's more alchemy than science: these tricks are heuristics, not theory-backed guidelines. They're supported by a level of intuitive understanding of the phenomenon at hand, and they're known to work well empirically, although not necessarily in every context.

Here are a few of the tricks used in the implementation of the GAN generator and discriminator in this section. It isn't an exhaustive list of GAN-related tips; you'll find many more across the GAN literature:

- We use `tanh` as the last activation in the generator, instead of `sigmoid`, which is more commonly found in other types of models.
- We sample points from the latent space using a *normal distribution* (Gaussian distribution), not a uniform distribution.

- Stochasticity is good to induce robustness. Because GAN training results in a dynamic equilibrium, GANs are likely to get stuck in all sorts of ways. Introducing randomness during training helps prevent this. We introduce randomness in two ways: by using dropout in the discriminator and by adding random noise to the labels for the discriminator.
- Sparse gradients can hinder GAN training. In deep learning, sparsity is often a desirable property, but not in GANs. Two things can induce gradient sparsity: max pooling operations and `ReLU` activations. Instead of max pooling, we recommend using strided convolutions for downsampling, and we recommend using a `LeakyReLU` layer instead of a `ReLU` activation. It's similar to `ReLU`, but it relaxes sparsity constraints by allowing small negative activation values.
- In generated images, it's common to see checkerboard artifacts caused by unequal coverage of the pixel space in the generator (see figure 8.17). To fix this, we use a kernel size that's divisible by the stride size whenever we use a strided `Conv2DTranpose` or `Conv2D` in both the generator and the discriminator.

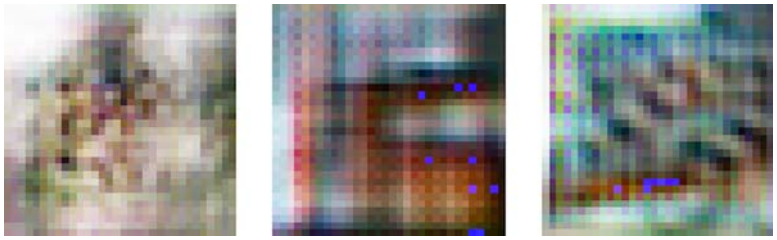


Figure 8.17 Checkerboard artifacts caused by mismatching strides and kernel sizes, resulting in unequal pixel-space coverage: one of the many gotchas of GANs

8.5.3 The generator

First, let's develop a `generator` model that turns a vector (from the latent space—during training it will be sampled at random) into a candidate image. One of the many issues that commonly arise with GANs is that the generator gets stuck with generated images that look like noise. A possible solution is to use dropout on both the discriminator and the generator.

Listing 8.29 GAN generator network

```
import keras
from keras import layers
import numpy as np

latent_dim = 32
height = 32
width = 32
channels = 3
```

```
generator_input = keras.Input(shape=(latent_dim,))

x = layers.Dense(128 * 16 * 16)(generator_input)
x = layers.LeakyReLU()(x)
x = layers.Reshape((16, 16, 128))(x)

x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)

x = layers.Conv2DTranspose(256, 4, strides=2, padding='same')(x)
x = layers.LeakyReLU()(x)

x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)

x = layers.Conv2D(channels, 7, activation='tanh', padding='same')(x)
generator = keras.models.Model(generator_input, x)
generator.summary()
```

Transforms the input into a 16 × 16 128-channel feature map

Upsamples to 32 × 32

Produces a 32 × 32 1-channel feature map (shape of a CIFAR10 image)

Instantiates the generator model, which maps the input of shape (latent_dim,) into an image of shape (32, 32, 3)

8.5.4 The discriminator

Next, you'll develop a `discriminator` model that takes as input a candidate image (real or synthetic) and classifies it into one of two classes: "generated image" or "real image that comes from the training set."

Listing 8.30 The GAN discriminator network

```
discriminator_input = layers.Input(shape=(height, width, channels))
x = layers.Conv2D(128, 3)(discriminator_input)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Flatten()(x)

x = layers.Dropout(0.4)(x)

x = layers.Dense(1, activation='sigmoid')(x)

discriminator = keras.models.Model(discriminator_input, x)
discriminator.summary()

discriminator_optimizer = keras.optimizers.RMSprop(
    lr=0.0008,
    clipvalue=1.0,
    decay=1e-8)

discriminator.compile(optimizer=discriminator_optimizer,
                      loss='binary_crossentropy')
```

One dropout layer: an important trick!

Classification layer

Instantiates the discriminator model, which turns a (32, 32, 3) input into a binary classification decision (fake/real)

Uses gradient clipping (by value) in the optimizer

To stabilize training, uses learning-rate decay

8.5.5 The adversarial network

Finally, you'll set up the GAN, which chains the generator and the discriminator. When trained, this model will move the generator in a direction that improves its ability to fool the discriminator. This model turns latent-space points into a classification decision—"fake" or "real"—and it's meant to be trained with labels that are always "these are real images." So, training `gan` will update the weights of `generator` in a way that makes `discriminator` more likely to predict "real" when looking at fake images. It's very important to note that you set the discriminator to be frozen during training (non-trainable): its weights won't be updated when training `gan`. If the discriminator weights could be updated during this process, then you'd be training the discriminator to always predict "real," which isn't what you want!

Listing 8.31 Adversarial network

```
discriminator.trainable = False

gan_input = keras.Input(shape=(latent_dim,))
gan_output = discriminator(generator(gan_input))
gan = keras.models.Model(gan_input, gan_output)

gan_optimizer = keras.optimizers.RMSprop(lr=0.0004, clipvalue=1.0, decay=1e-8)
gan.compile(optimizer=gan_optimizer, loss='binary_crossentropy')
```

Sets discriminator weights to non-trainable (this will only apply to the gan model)

8.5.6 How to train your DCGAN

Now you can begin training. To recapitulate, this is what the training loop looks like schematically. For each epoch, you do the following:

1. Draw random points in the latent space (random noise).
2. Generate images with `generator` using this random noise.
3. Mix the generated images with real ones.
4. Train `discriminator` using these mixed images, with corresponding targets: either "real" (for the real images) or "fake" (for the generated images).
5. Draw new random points in the latent space.
6. Train `gan` using these random vectors, with targets that all say "these are real images." This updates the weights of the generator (only, because the discriminator is frozen inside `gan`) to move them toward getting the discriminator to predict "these are real images" for generated images: this trains the generator to fool the discriminator.

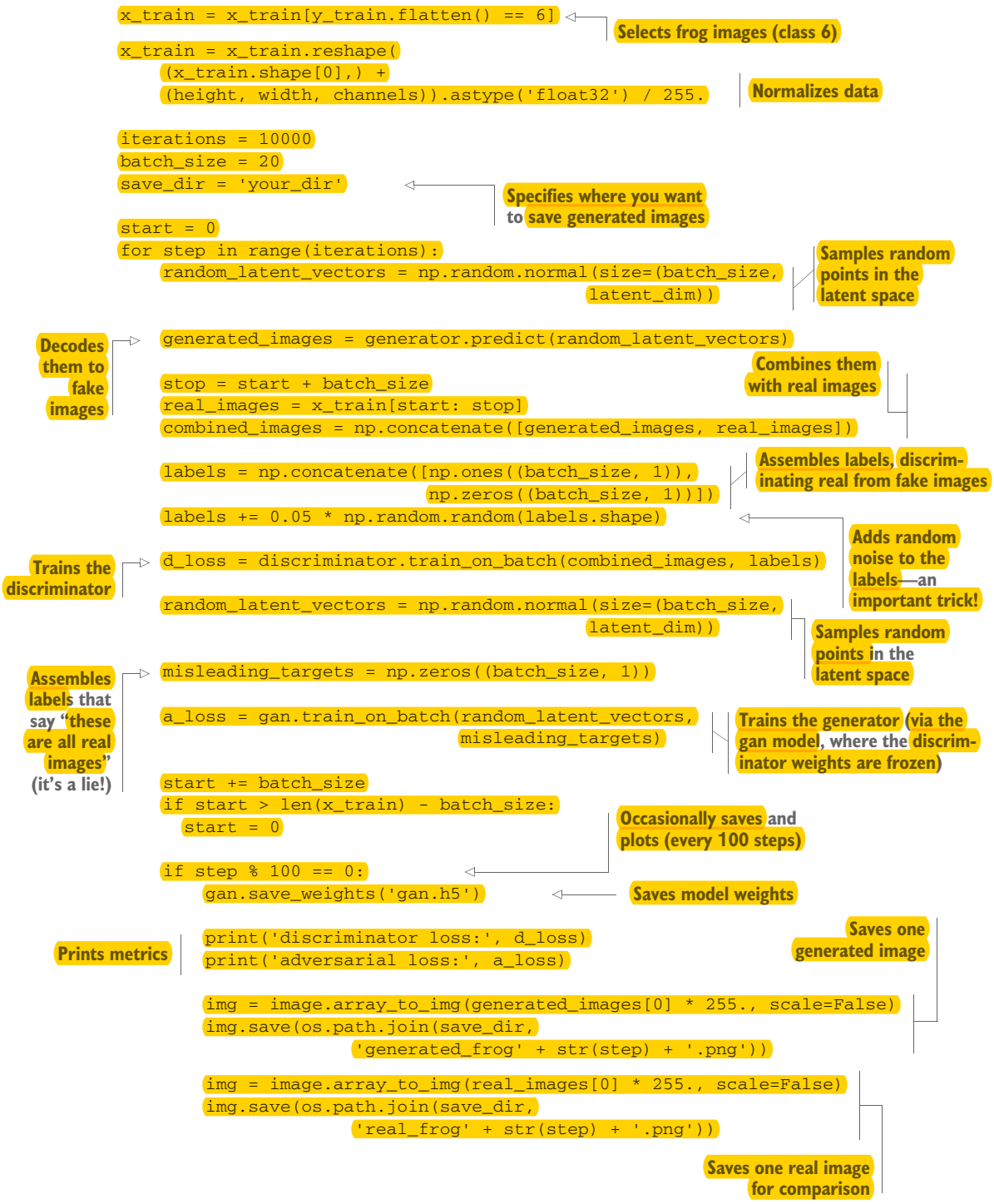
Let's implement it.

Listing 8.32 Implementing GAN training

```
import os
from keras.preprocessing import image

(x_train, y_train), (_, _) = keras.datasets.cifar10.load_data()
```

Loads CIFAR10 data



When training, you may see the adversarial loss begin to increase considerably, while the discriminative loss tends to zero—the discriminator may end up dominating the generator. If that's the case, try reducing the discriminator learning rate, and increase the dropout rate of the discriminator.

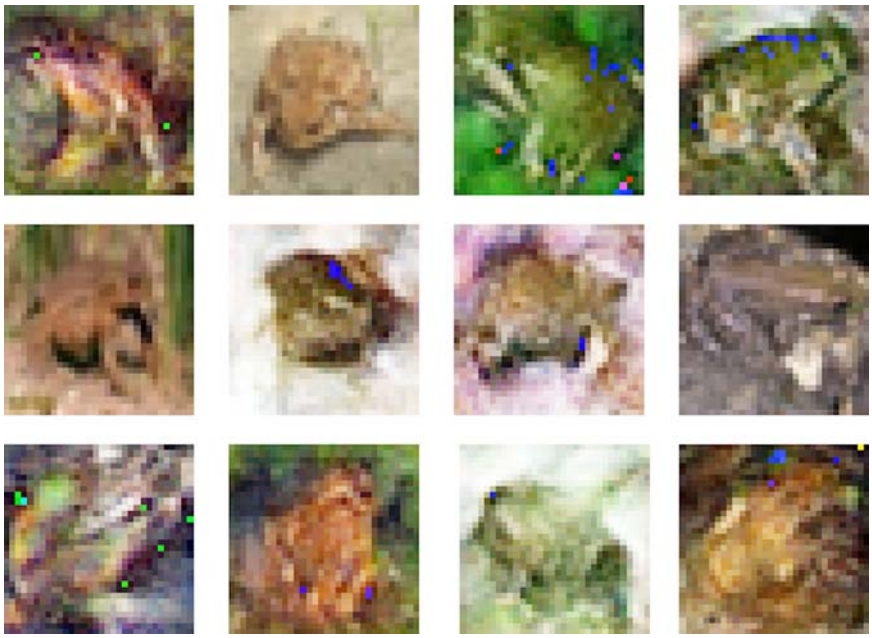


Figure 8.18 Play the discriminator: in each row, two images were dreamed up by the GAN, and one image comes from the training set. Can you tell them apart? (Answers: the real images in each column are middle, top, bottom, middle.)

8.5.7 Wrapping up

- A GAN consists of a generator network coupled with a discriminator network. The discriminator is trained to differenciate between the output of the generator and real images from a training dataset, and the generator is trained to fool the discriminator. Remarkably, the generator nevers sees images from the training set directly; the information it has about the data comes from the discriminator.
- GANs are difficult to train, because training a GAN is a dynamic process rather than a simple gradient descent process with a fixed loss landscape. Getting a GAN to train correctly requires using a number of heuristic tricks, as well as extensive tuning.
- GANs can potentially produce highly realistic images. But unlike VAEs, the latent space they learn doesn't have a neat continuous structure and thus may not be suited for certain practical applications, such as image editing via latent-space concept vectors.