CHAPTER 2
Deep Learning

Let's start with a basic definition of deep learning:

> Deep learning is a class of machine learning algorithm that uses multiple stacked layers of processing units to learn high-level representations from unstructured data.

To understand deep learning fully, and particularly why it is so useful within generative modeling, we need to delve into this definition a bit further. First, what do we mean by "unstructured data" and its counterpart, "structured data"?

Structured and Unstructured Data

Many types of machine learning algorithm require *structured*, tabular data as input, arranged into columns of features that describe each observation. For example, a person's age, income, and number of website visits in the last month are all features that could help to predict if the person will subscribe to a particular online service in the coming month. We could use a structured table of these features to train a logistic regression, random forest, or XGBoost model to predict the binary response variable —did the person subscribe (1) or not (0)? Here, each individual feature contains a nugget of information about the observation, and the model would learn how these features interact to influence the response.

Unstructured data refers to any data that is not naturally arranged into columns of features, such as images, audio, and text. There is of course spatial structure to an image, temporal structure to a recording, and both spatial and temporal structure to video data, but since the data does not arrive in columns of features, it is considered unstructured, as shown in Figure 2-1.

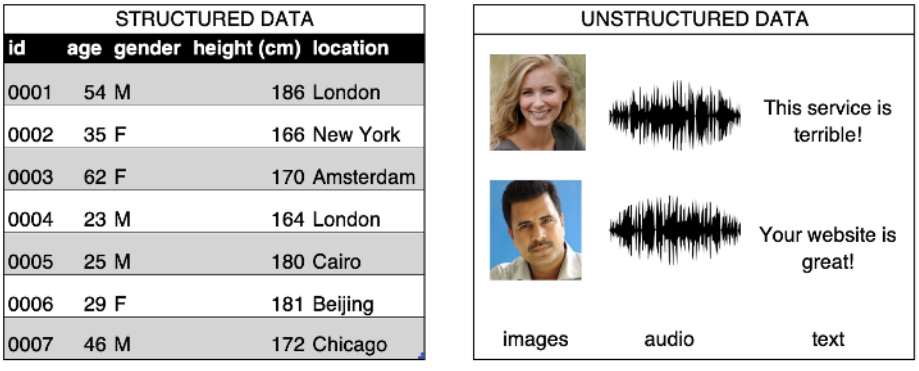


Figure 2-1. The difference between structured and unstructured data

When our data is unstructured, individual pixels, frequencies, or characters are almost entirely uninformative. For example, knowing that pixel 234 of an image is a muddy shade of brown doesn't really help identify if the image is of a house or a dog, and knowing that character 24 of a sentence is an *e* doesn't help predict if the text is about football or politics.

Pixels or characters are really just the dimples of the canvas into which higher-level informative features, such as an image of a chimney or the word *striker*, are embedded. If the chimney in the image were placed on the other side of the house, the image would still contain a chimney, but this information would now be carried by completely different pixels. If the word *striker* appeared slightly earlier or later in the text, the text would still be about football, but different character positions would provide this information. The granularity of the data combined with the high degree of spatial dependence destroys the concept of the pixel or character as an informative feature in its own right.

For this reason, if we train logistic regression, random forest, or XGBoost algorithms on raw pixel values, the trained model will often perform poorly for all but the simplest of classification tasks. These models rely on the input features to be informative and not spatially dependent. A deep learning model, on the other hand, can learn how to build high-level informative features by itself, directly from the unstructured data.

Deep learning can be applied to structured data, but its real power, especially with regard to generative modeling, comes from its ability to work with unstructured data. Most often, we want to generate unstructured data such as new images or original strings of text, which is why deep learning has had such a profound impact on the field of generative modeling.

Deep Neural Networks

The majority of deep learning systems are *artificial neural networks* (ANNs, or just *neural networks* for short) with multiple stacked hidden layers. For this reason, *deep learning* has now almost become synonymous with *deep neural networks*. However, it is important to point out that any system that employs many layers to learn high level representations of the input data is also a form of deep learning (e.g., deep belief networks and deep Boltzmann machines).

Let's start by taking a high-level look at how a deep neural network can make a prediction about a given input.

A deep neural network consists of a series of stacked *layers*. Each layer contains *units*, that are connected to the previous layer's units through a set of *weights*. As we shall see, there are many different types of layer, but one of the most common is the *dense* layer that connects all units in the layer directly to every unit in the previous layer. By stacking layers, the units in each subsequent layer can represent increasingly sophisticated aspects of the original input.

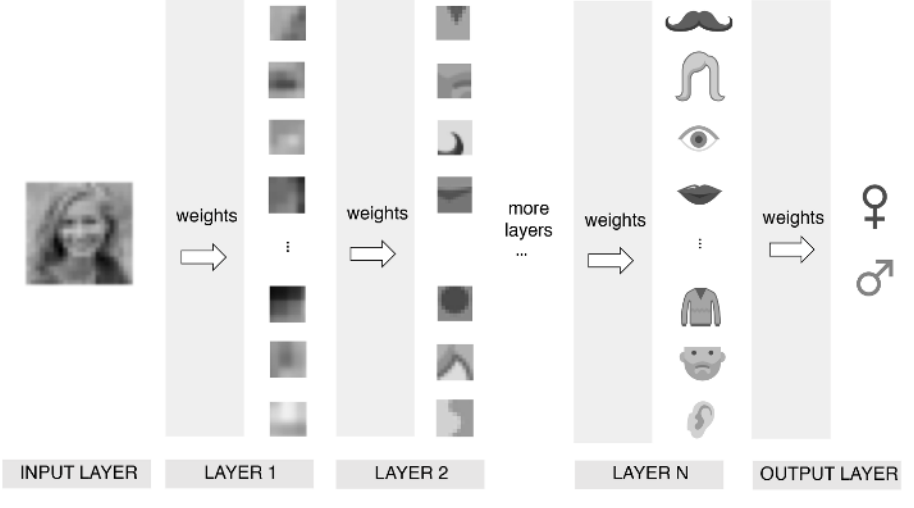


Figure 2-2. Deep learning conceptual diagram

For example, in Figure 2-2, layer 1 consists of units that activate more strongly when they detect particular basic properties of the input image, such as edges. The output from these units is then passed to the units of layer 2, which are able to use this information to detect slightly more complex features—and so on, through the network. The final output layer is the culmination of this process, where the network outputs a set of numbers that can be converted into probabilities, to represent the chance that the original input belongs to one of *n* categories.

The magic of deep neural networks lies in finding the set of weights for each layer that results in the most accurate predictions. The process of finding these weights is what we mean by *training* the network.

During the training process, batches of images are passed through the network and the output is compared to the ground truth. The error in the prediction is then propagated backward through the network, adjusting each set of weights a small amount in the direction that improves the prediction most significantly. This process is appropriately called *backpropagation*. Gradually, each unit becomes skilled at identifying a particular feature that ultimately helps the network to make better predictions.

Deep neural networks can have any number of middle or *hidden* layers. For example, ResNet,[1] designed for image recognition, contains 152 layers. We shall see in Chapter 3 that we can use deep neural networks to influence high-level features of an image, such as hair color or expression of a face, by manually tweaking the values of these hidden layers. This is only possible because the deeper layers of the network are capturing high-level features that we can work with directly.

Next, we'll dive straight into the practical side of deep learning and get set up with Keras and TensorFlow, the two libraries that will enable you to start building your own generative deep neural networks.

Keras and TensorFlow

Keras is a high-level Python library for building neural networks and is the core library that we shall be using in this book. It is extremely flexible and has a very user-friendly API, making it an ideal choice for getting started with deep learning. Moreover, Keras provides numerous useful building blocks that can be plugged together to create highly complex deep learning architectures through its functional API.

Keras is not the library that performs the low-level array operations required to train neural networks. Instead Keras utilizes one of three backend libraries for this purpose: TensorFlow, CNTK, or Theano. You are free to choose whichever you are most comfortable with, or whichever library works fastest for a particular network architecture. For most purposes, it doesn't matter which you choose as you usually won't be coding directly using the underlying backend framework. In this book we use TensorFlow as it is the most widely adopted and best documented of the three.

TensorFlow is an open-source Python library for machine learning, developed by Google. It is now one of the most utilized frameworks for building machine learning solutions, with particular emphasis on the manipulation of tensors (hence the name).

1 Kaiming He et al., "Deep Residual Learning for Image Recognition," 10 December 2015, *https://arxiv.org/abs/1512.03385*.

Within the context of deep learning, tensors are simply multidimensional arrays that store the data as it flows through the network. As we shall see, understanding how each layer of a neural network changes the shape of the data as it flows through the network is a key part of truly understanding the mechanics of deep learning.

If you are just getting started with deep learning, I highly recommend that you choose Keras with a TensorFlow backend as your toolkit. These two libraries are a powerful combination that will allow you to build any network that you can think of in a production environment, while also giving you the easy-to-learn API that is so important for rapid development of new ideas and concepts.

REVIEWED
By Ivan at 2:23 pm, Jun 07, 2020

Your First Deep Neural Network

Let's start by seeing how easy it is to build a deep neural network in Keras.

We will be working through the Jupyter notebook in the book repository called *02_01_deep_learning_deep_neural_network.ipynb*.

colab - https://drive.google.com/file/d/1fAwklUFShfgJJd4A5QgosT6E0PWOGNPS/view?usp=sharing

Loading the Data

For this example we will be using the CIFAR-10 dataset, a collection of 60,000 32 × 32–pixel color images that comes bundled with Keras out of the box. Each image is classified into exactly one of 10 classes, as shown in Figure 2-3.

The following code loads and scales the data:

```
import numpy as np
from keras.utils import to_categorical
from keras.datasets import cifar10

(x_train, y_train), (x_test, y_test) = cifar10.load_data() ❶

NUM_CLASSES = 10

x_train = x_train.astype('float32') / 255.0 ❷
x_test = x_test.astype('float32') / 255.0

y_train = to_categorical(y_train, NUM_CLASSES) ❸
y_test = to_categorical(y_test, NUM_CLASSES)
```

❶ Loads the CIFAR-10 dataset. `x_train` and `x_test` are `numpy` arrays of shape `[50000, 32, 32, 3]` and `[10000, 32, 32, 3]`, respectively. `y_train` and `y_test` are `numpy` arrays with shape `[50000, 1]` and `[10000, 1]`, respectively, containing the integer labels in the range 0 to 9 for the class of each image.

❷ By default the image data consists of integers between 0 and 255 for each pixel channel. Neural networks work best when each input is inside the range –1 to 1, so we need to divide by 255.

❸ We also need to change the integer labeling of the images to one-hot-encoded vectors. If the class integer label of an image is *i*, then its one-hot encoding is a vector of length 10 (the number of classes) that has 0s in all but the *i*th element, which is 1. The new shapes of `y_train` and `y_test` are therefore `[50000, 10]` and `[10000, 10]` respectively.

Figure 2-3. Example images from the CIFAR-10 dataset[2]

2 Source: Alex Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," 2009, *https://www.cs.toronto.edu/~kriz/cifar.html*.

It's worth noting the shape of the image data in `x_train`: `[50000, 32, 32, 3]`. The first dimension of this array references the index of the image in the dataset, the second and third relate to the size of the image, and the last is the channel (i.e., red, green, or blue, since these are RGB images). There are no *columns* or *rows* in this dataset; instead, this is a *tensor* with four dimensions. For example, the following entry refers to the green channel (1) value of the pixel in the (12,13) position of image 54:

REVIEWED
By Ivan at 2:59 pm, Jun 07, 2020

```
x_train[54, 12, 13, 1]
# 0.36862746
```

Building the Model

In Keras there are two ways to define the structure of your neural network: as a Sequential model or using the Functional API.

A Sequential model is useful for quickly defining a linear stack of layers (i.e., where one layer follows on directly from the previous layer without any branching). However, many of the models in this book require that the output from a layer is passed to multiple separate layers beneath it, or conversely, that a layer receives input from multiple layers above it.

To be able to build networks with branches, we need to use the Functional API, which is a lot more flexible. I recommend that even if you are just starting out building linear models with Keras, you still use the Functional API rather than Sequential models, since it will serve you better in the long run as your neural networks become more architecturally complex. The Functional API will give you complete freedom over the design of your deep neural network.

To demonstrate the difference between the two methods, Examples 2-1 and 2-2 show the same network coded using a Sequential model and the Functional API. Feel free to try both and observe that they give the same result.

Example 2-1. The architecture using a Sequential model

```
from keras.models import Sequential
from keras.layers import Flatten, Dense

model = Sequential([
    Dense(200, activation = 'relu', input_shape=(32, 32, 3)),
    Flatten(),
    Dense(150, activation = 'relu'),
    Dense(10, activation = 'softmax'),
])
```

Example 2-2. The architecture using the Functional API

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

input_layer = Input(shape=(32,32, 3))

x = Flatten()(input_layer)

x = Dense(units=200, activation = 'relu')(x)
x = Dense(units=150, activation = 'relu')(x)

output_layer = Dense(units=10, activation = 'softmax')(x)

model = Model(input_layer, output_layer)>
```

Here, we are using three different types of layer: `Input`, `Flatten`, and `Dense`.

The `Input` layer is an entry point into the network. We tell the network the shape of each data element to expect as a tuple. Notice that we do not specify the batch size; this isn't necessary as we can pass any number of images into the `Input` layer simultaneously. We do not need to explicitly state the batch size in the `Input` layer definition.

Next we flatten this input into a vector, using a `Flatten` layer. This results in a vector of length 3,072 (= 32 × 32 × 3). The reason we do this is because the subsequent `Dense` layer requires that its input is flat, rather than a multidimensional array. As we shall see later, other layer types require multidimensional arrays as input, so you need to be aware of the required input and output shape of each layer type to understand when it is necessary to use `Flatten`.

The `Dense` layer is perhaps the most fundamental layer type in any neural network. It contains a given number of units that are densely connected to the previous layer—that is, every unit in the layer is connected to every unit in the previous layer, through a single connection that carries a weight (which can be positive or negative). The output from a given unit is the weighted sum of the input it receives from the previous layer, which is then passed through a nonlinear *activation function* before being sent to the following layer. The activation function is critical to ensure the neural network is able to learn complex functions and doesn't just output a linear combination of its input.

There are many kinds of activation function, but the three most important are ReLU, sigmoid, and softmax.

The *ReLU* (rectified linear unit) activation function is defined to be zero if the input is negative and is otherwise equal to the input. The *LeakyReLU* activation function is very similar to ReLU, with one key difference: whereas the ReLU activation function returns zero for input values less than zero, the LeakyReLU function returns a small

negative number proportional to the input. ReLU units can sometimes *die* if they always output zero, because of a large bias toward negative values preactivation. In this case, the gradient is zero and therefore no error is propagated back through this unit. LeakyReLU activations fix the issue by always ensuring the gradient is nonzero. ReLU-based functions are now established to be the most reliable activations to use between the layers of a deep network to encourage stable training.

已審核
Ivan Lin (Test) 20/06/07, 18:18

The *sigmoid* activation is useful if you wish the output from the layer to be scaled between 0 and 1—for example, for binary classification problems with one output unit or multilabel classification problems, where each observation can belong to more than one class. Figure 2-4 shows ReLU, LeakyReLU, and sigmoid activation functions side by side for comparison.

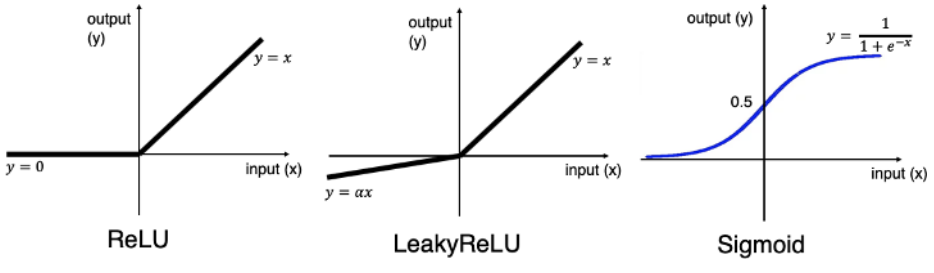


Figure 2-4. The ReLU, LeakyReLU, and sigmoid activation functions

The *softmax* activation is useful if you want the total sum of the output from the layer to equal 1, for example, for multiclass classification problems where each observation only belongs to exactly one class. It is defined as:

$$y_i = \frac{e^{x_i}}{\Sigma_{j=1}^{J} e^{x_j}}$$

Here, J is the total number of units in the layer. In our neural network, we use a softmax activation in the final layer to ensure that the output is a set of 10 probabilities that sum to 1, which can be interpreted as the chance that the image belongs to each class.

In Keras, activation functions can also be defined in a separate layer as follows:

```
x = Dense(units=200)(x)
x = Activation('relu')(x)
```

This is equivalent to:

```
x = Dense(units=200, activation = 'relu')(x)
```

In our example, we pass the input through two dense hidden layers, the first with 200 units and the second with 150, both with ReLU activation functions. A diagram of the total network is shown in Figure 2-5.

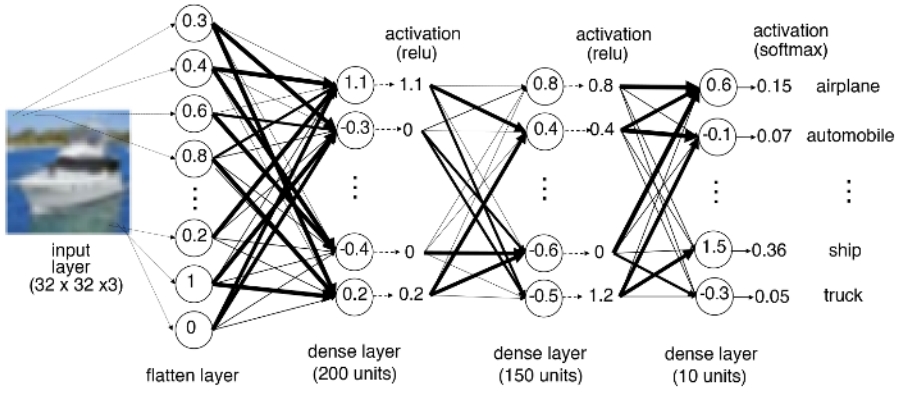


Figure 2-5. A diagram of the neural network trained on CIFAR-10 data

The final step is to define the model itself, using the `Model` class. In Keras a model is defined by the input and output layers. In our case, we have one input layer that we defined earlier, and the output layer is the final `Dense` layer of 10 units. It is also possible to define models with multiple input and output layers; we shall see this in action later in the book.

In our example, as required, the shape of our `Input` layer matches the shape of `x_train` and the shape of our `Dense` output layer matches the shape of `y_train`. To illustrate this, we can use the `model.summary()` method to see the shape of the network at each layer as shown in Figure 2-6.

```
input_layer = Input(shape=(32,32, 3))

x = Flatten()(input_layer)

x = Dense(units=200, activation = 'relu')(x)
x = Dense(units=150, activation = 'relu')(x)

output_layer = Dense(units=10, activation = 'softmax')(x)

model = Model(input_layer, output_layer)>
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 32, 32, 3)         0
_________________________________________________________________
flatten_1 (Flatten)          (None, 3072)              0
_________________________________________________________________
dense_1 (Dense)              (None, 200)               614600
_________________________________________________________________
dense_2 (Dense)              (None, 150)               30150
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1510
=================================================================
Total params: 646,260
Trainable params: 646,260
Non-trainable params: 0
_________________________________________________________________
```

3072 x 200 + 200

200x150 + 150

150x100+10

Figure 2-6. The summary of the model

Notice how Keras uses `None` as a marker to show that it doesn't yet know the number of observations that will be passed into the network. In fact, it doesn't need to; we could just as easily pass one observation through the network at a time as 1,000. That's because tensor operations are conducted across all observations simultaneously using linear algebra—this is the part handled by TensorFlow. It is also the reason why you get a performance increase when training deep neural networks on GPUs instead of CPUs: GPUs are optimized for large tensor multiplications since these calculations are also necessary for complex graphics manipulation.

The `summary` method also gives the number of parameters (weights) that will be trained at each layer. If ever you find that your model is training too slowly, check the summary to see if there are any layers that contain a huge number of weights. If so, you should consider whether the number of units in the layer could be reduced to speed up training.

Compiling the Model

In this step, we compile the model with an optimizer and a loss function:

```
from keras.optimizers import Adam

opt = Adam(lr=0.0005)
model.compile(loss='categorical_crossentropy', optimizer=opt,
              metrics=['accuracy'])
```

The loss function is used by the neural network to compare its predicted output to the ground truth. It returns a single number for each observation; the greater this number, the worse the network has performed for this observation.

Keras provides many built-in loss functions to choose from, or you can create your own. Three of the most commonly used are mean squared error, categorical cross-entropy, and binary cross-entropy. It is important to understand when it is appropriate to use each.

If your neural network is designed to solve a regression problem (i.e., the output is continuous), then you can use the *mean squared error* loss. This is the mean of the squared difference between the ground truth y_i and predicted value p_i of each output unit, where the mean is taken over all n output units:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - p_i)^2$$

If you are working on a classification problem where each observation only belongs to one class, then *categorical cross-entropy* is the correct loss function. This is defined as follows:

$$-\sum_{i=1}^{n} y_i \log(p_i)$$

Finally, if you are working on a binary classification problem with one output unit, or a multilabel problem where each observation can belong to multiple classes simultaneously, you should use *binary cross-entropy*:

$$-\frac{1}{n}\sum_{i=1}^{n}\left(y_i \log(p_i) + (1 - y_i)\log(1 - p_i)\right)$$

The optimizer is the algorithm that will be used to update the weights in the neural network based on the gradient of the loss function. One of the most commonly used and stable optimizers is *Adam*.[3] In most cases, you shouldn't need to tweak the default parameters of the Adam optimizer, except for the learning rate. The greater the learning rate, the larger the change in weights at each training step. While training is initially faster with a large learning rate, the downside is that it may result in less stable training and may not find the minima of the loss function. This is a parameter that you may want to tune or adjust during training.

3 Diederik Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization," 22 December 2014, *https://arxiv.org/abs/1412.6980v8*.

Another common optimizer that you may come across is *RMSProp*. Again, you shouldn't need to adjust the parameters of this optimizer too much, but it is worth reading the Keras documentation to understand the role of each parameter.

We pass both the loss function and the optimizer into the `compile` method of the model, as well as a `metrics` parameter where we can specify any additional metrics that we would like reporting on during training, such as accuracy.

Training the Model

Thus far, we haven't shown the model any data and have just set up the architecture and compiled the model with a loss function and optimizer.

To train the model, simply call the `fit` method, as shown here:

```
model.fit(x_train ❶
          , y_train ❷
          , batch_size = 32 ❸
          , epochs = 10 ❹
          , shuffle = True ❺
          )
```

❶ The raw image data.

❷ The one-hot-encoded class labels.

❸ The `batch_size` determines how many observations will be passed to the network at each training step.

❹ The `epochs` determine how many times the network will be shown the full training data.

❺ If `shuffle = True`, the batches will be drawn randomly without replacement from the training data at each training step.

This will start training a deep neural network to predict the category of an image from the CIFAR-10 dataset.

The training process works as follows. First, the weights of the network are initialized to small random values. Then the network performs a series of training steps.

At each training step, one batch of images is passed through the network and the errors are backpropagated to update the weights. The `batch_size` determines how many images are in each training step batch. The larger the batch size, the more stable the gradient calculation, but the slower each training step. It would be far too time-consuming and computationally intensive to use the entire dataset to calculate the gradient at each training step, so generally a batch size between 32 and 256 is

used. It is also now recommended practice to increase the batch size as training progresses.[4]

This continues until all observations in the dataset have been seen once. This completes the first *epoch*. The data is then passed through the network again in batches as part of the second epoch. This process repeats until the specified number of epochs have elapsed.

During training, Keras outputs the progress of the procedure, as shown in Figure 2-7. We can see that the training dataset of 50,000 observations has been shown to the network 10 times (i.e., over 10 epochs), at a rate of approximately 160 microseconds per observation. The categorical cross-entropy loss has fallen from 1.842 to 1.357, resulting in an accuracy increase from 33.5% after the first epoch to 51.9% after the tenth epoch.

```
model.fit(x_train
          , y_train
          , batch_size=BATCH_SIZE
          , epochs=EPOCHS
          , shuffle=True)
```

```
Epoch 1/10
50000/50000 [==============================] - 8s 164us/step - loss: 1.8424 - acc: 0.3354
Epoch 2/10
50000/50000 [==============================] - 8s 154us/step - loss: 1.6592 - acc: 0.4048
Epoch 3/10
50000/50000 [==============================] - 8s 153us/step - loss: 1.5733 - acc: 0.4381
Epoch 4/10
50000/50000 [==============================] - 8s 154us/step - loss: 1.5232 - acc: 0.4579
Epoch 5/10
50000/50000 [==============================] - 8s 155us/step - loss: 1.4874 - acc: 0.4698
Epoch 6/10
50000/50000 [==============================] - 8s 165us/step - loss: 1.4569 - acc: 0.4799
Epoch 7/10
50000/50000 [==============================] - 10s 208us/step - loss: 1.4281 - acc: 0.4887
Epoch 8/10
50000/50000 [==============================] - 8s 165us/step - loss: 1.4038 - acc: 0.4984
Epoch 9/10
50000/50000 [==============================] - 8s 153us/step - loss: 1.3797 - acc: 0.5084
Epoch 10/10
50000/50000 [==============================] - 8s 155us/step - loss: 1.3571 - acc: 0.5187
```

Figure 2-7. The output from the fit method

Evaluating the Model

We know the model achieves an accuracy of 51.9% on the training set, but how does it perform on data it has never seen?

To answer this question we can use the `evaluate` method provided by Keras:

```
model.evaluate(x_test, y_test)
```

4 Samuel L. Smith et al., "Don't Decay the Learning Rate, Increase the Batch Size," 1 November 2017, *https://arxiv.org/abs/1711.00489*.

Figure 2-8 shows the output from this method.

```
10000/10000 [==============================] - 1s 55us/step

[1.4358007415771485, 0.4896]
```

Figure 2-8. The output from the evaluate method

The output from this method is a list of the metrics we are monitoring: categorical cross-entropy and accuracy. We can see that model accuracy is still 49.0% even on images that it has never seen before. Note that if the model was guessing randomly, it would achieve approximately 10% accuracy (because there are 10 classes), so 50% is a good result given that we have used a very basic neural network.

We can view some of the predictions on the test set using the `predict` method:

```
CLASSES = np.array(['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog'
                    , 'frog', 'horse', 'ship', 'truck'])

preds = model.predict(x_test) ❶
preds_single = CLASSES[np.argmax(preds, axis = -1)] ❷
actual_single = CLASSES[np.argmax(y_test, axis = -1)]
```

❶ `preds` is an array of shape `[10000, 10]`—i.e., a vector of 10 class probabilities for each observation.

❷ We convert this array of probabilities back into a single prediction using `numpy`'s `argmax` function. Here, `axis = -1` tells the function to collapse the array over the last dimension (the classes dimension), so that the shape of `preds_single` is then `[10000, 1]`.

We can view some of the images alongside their labels and predictions with the following code. As expected, around half are correct:

```
import matplotlib.pyplot as plt

n_to_show = 10
indices = np.random.choice(range(len(x_test)), n_to_show)

fig = plt.figure(figsize=(15, 3))
fig.subplots_adjust(hspace=0.4, wspace=0.4)

for i, idx in enumerate(indices):
    img = x_test[idx]
    ax = fig.add_subplot(1, n_to_show, i+1)
    ax.axis('off')
    ax.text(0.5, -0.35, 'pred = ' + str(preds_single[idx]), fontsize=10
       , ha='center', transform=ax.transAxes)
```

```
ax.text(0.5, -0.7, 'act = ' + str(actual_single[idx]), fontsize=10
    , ha='center', transform=ax.transAxes)
ax.imshow(img)
```

Figure 2-9 shows a randomly chosen selection of predictions made by the model, alongside the true labels.

Figure 2-9. Some predictions made by the model, alongside the actual labels

Congratulations! You've just built your first deep neural network using Keras and used it to make predictions on new data. Even though this is a supervised learning problem, when we come to building generative models in future chapters many of the core ideas from this network (such as loss functions, activation functions, and understanding layer shapes) will still be extremely important. Next we'll look at ways of improving this model, by introducing a few new layer types.

REVIEWED
By Ivan at 11:55 am, Jun 09, 2020

Improving the Model

One of the reasons our network isn't yet performing as well as it might is because there isn't anything in the network that takes into account the spatial structure of the input images. In fact, our first step is to flatten the image into a single vector, so that we can pass it to the first `Dense` layer!

To achieve this we need to use a *convolutional* layer.

Convolutional Layers

First, we need to understand what is meant by a *convolution* in the context of deep learning.

Figure 2-10 shows a $3 \times 3 \times 1$ portion of a grayscale image being convoluted with a $3 \times 3 \times 1$ *filter* (or *kernel*).

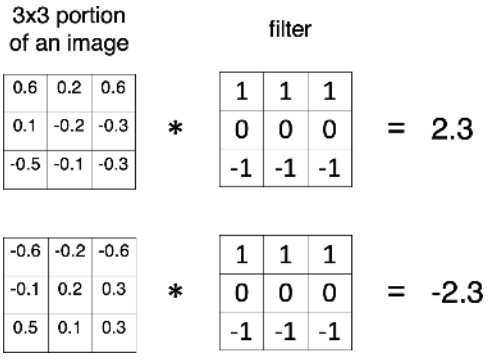


Figure 2-10. The convolution operation

The convolution is performed by multiplying the filter pixelwise with the portion of the image, and summming the result. The output is more positive when the portion of the image closely matches the filter and more negative when the portion of the image is the inverse of the filter.

If we move the filter across the entire image, from left to right and top to bottom, recording the convolutional output as we go, we obtain a new array that picks out a particular feature of the input, depending on the values in the filter.

This is exactly what a convolutional layer is designed to do, but with multiple filters rather than just one. For example, Figure 2-11 shows two filters that highlight horizontal and vertical edges. You can see this convolutional process worked through manually in the notebook *02_02_deep_learning_convolutions.ipynb* in the book repository.

colab: https://drive.google.com/file/d/1UkrQdp3Rs_aS8K0qB6GMs6ZIKAKTuYH4/view?usp=sharing

REVIEWED
By Ivan at 12:10 pm, Jun 09, 2020

If we are working with color images, then each filter would have three channels rather than one (i.e. each having shape $3 \times 3 \times 3$) to match the three channels (red, green, blue) of the image.

In Keras, the `Conv2D` layer applies convolutions to an input tensor with two spatial dimensions (such as an image). For example, the Keras code corresponding to the diagram in Figure 2-11 is:

```
input_layer = Input(shape=(64,64,1))

conv_layer_1 = Conv2D(
    filters = 2
    , kernel_size = (3,3)
    , strides = 1
    , padding = "same"
    )(input_layer)
```

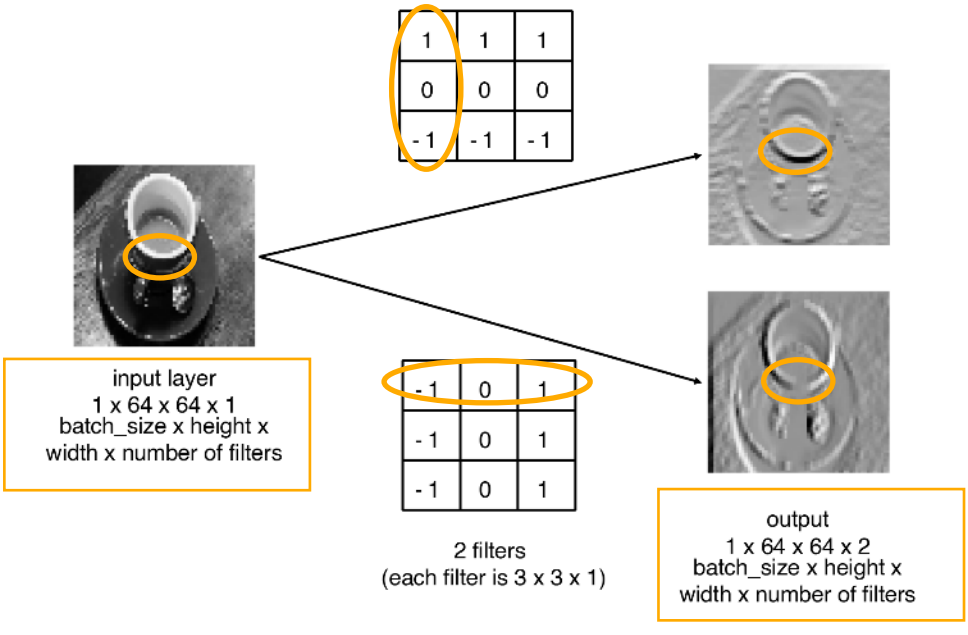


Figure 2-11. Two convolutional filters applied to a grayscale image

Strides

The `strides` parameter is the step size used by the layer to move the filters across the input. Increasing the stride therefore reduces the size of the output tensor. For example, when `strides = 2`, the height and width of the output tensor will be half the size of the input tensor. This is useful for reducing the spatial size of the tensor as it passes through the network, while increasing the number of channels.

Padding

The `padding = "same"` input parameter pads the input data with zeros so that the output size from the layer is exactly the same as the input size when `strides = 1`.

Figure 2-12 shows a 3 × 3 kernel being passed over a 5 × 5 input image, with `padding = "same"` and `strides = 1`. The output size from this convolutional layer would also be 5 × 5, as the padding allows the kernel to extend over the edge of the image, so that it fits five times in both directions. Without padding, the kernel could only fit three times along each direction, giving an output size of 3 × 3.

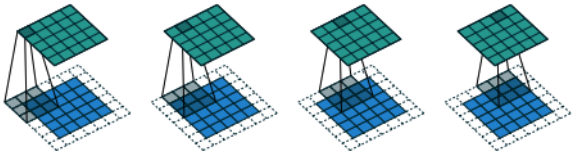


Figure 2-12. A 3 × 3 × 1 kernel (gray) being passed over a 5 × 5 × 1 input image (blue), with padding="same" and strides = 1, to generate the 5 × 5 × 1 output (green)[5]

Setting `padding = "same"` is a good way to ensure that you are able to easily keep track of the size of the tensor as it passes through many convolutional layers.

The values stored in the filters are the weights that are learned by the neural network through training. Initially these are random, but gradually the filters adapt their weights to start picking out interesting features such as edges or particular color combinations.

The output of a `Conv2D` layer is another four-dimensional tensor, now of shape `(batch_size, height, width, filters)`, so we can stack `Conv2D` layers on top of each other to grow the depth of our neural network. It's really important to understand how the shape of the tensor changes as data flows through from one convolutional layer to the next. To demonstrate this, let's imagine we are applying `Conv2D` layers to the CIFAR-10 dataset. This time, instead of one input channel (grayscale) we have three (red, green, and blue).

Figure 2-13 represents the following network in Keras:

```
input_layer = Input(shape=(32,32,3))

conv_layer_1 = Conv2D(
    filters = 10
    , kernel_size = (4,4)
    , strides = 2
    , padding = 'same'
    )(input_layer)

conv_layer_2 = Conv2D(
    filters = 20
    , kernel_size = (3,3)
    , strides = 2
    , padding = 'same'
    )(conv_layer_1)
```

5 Source: Vincent Dumoulin and Francesco Visin, "A Guide to Convolution Arithmetic for Deep Learning," 12 January 2018, *https://arxiv.org/pdf/1603.07285.pdf*.

```
flatten_layer = Flatten()(conv_layer_2)

output_layer = Dense(units=10, activation = 'softmax')(flatten_layer)

model = Model(input_layer, output_layer)
```

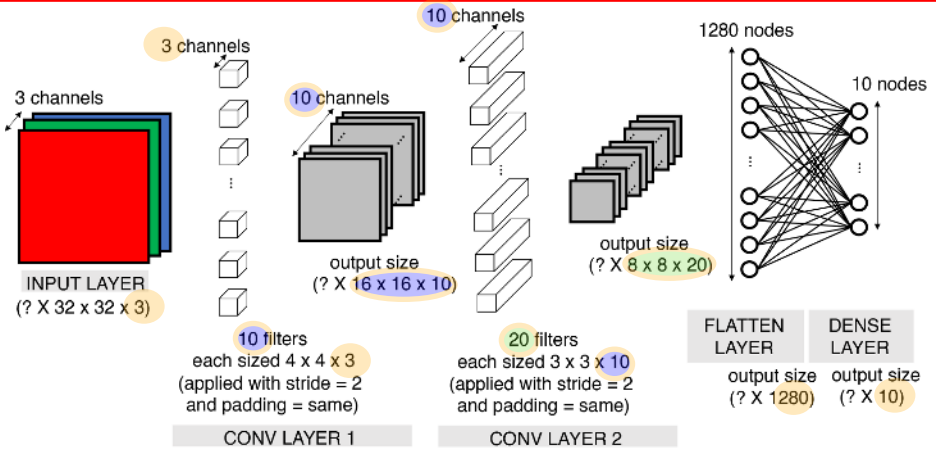


Figure 2-13. A diagram of a convolutional neural network

REVIEWED
By Ivan at 6:46 pm, Jun 12, 2020

We can use the `model.summary()` method to see the shape of the tensor as it passes through the network (Figure 2-14).

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 32, 32, 3)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 10)        490
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 20)          1820
_________________________________________________________________
flatten_1 (Flatten)          (None, 1280)              0
_________________________________________________________________
dense_1 (Dense)              (None, 10)                12810
=================================================================
Total params: 15,120
Trainable params: 15,120
Non-trainable params: 0
_________________________________________________________________
```

4x4x3 * 10 + 10 = 490

3x3x10 *20 + 20 = 1820

8x8x20 * 10 + 10 = 12810

12810 + 1820 + 490 = 15120

Figure 2-14. A convolutional neural network summary

Let's analyze this network from input through to output. The input shape is `(None, 32, 32, 3)`—Keras uses None to represent the fact that we can pass any number of images through the network simultaneously. Since the network is just performing

tensor algebra, we don't need to pass images through the network individually, but instead can pass them through together as a *batch*.

The shape of the filters in the first convolutional layer is $4 \times 4 \times 3$. This is because we have chosen the filter to have height and width of 4 (`kernel_size = (4,4)`) and there are three channels in the preceding layer (red, green, and blue). Therefore, the number of parameters (or weights) in the layer is $(4 \times 4 \times 3 + 1) \times 10 = 490$, where the + 1 is due to the inclusion of a bias term attached to each of the filters. It's worth remembering that the depth of the filters in a layer is *always* the same as the number of channels in the preceding layer.

As before, the output from each filter when applied to each $4 \times 4 \times 3$ section of the input image will be the pixelwise multiplication of the filter weights and the area of the image it is covering. As `strides = 2` and `padding = "same"`, the width and height of the output are both halved to 16, and since there are 10 filters the output of the first layer is a batch of tensors each having shape `[16, 16, 10]`.

In general, the shape of the output from a convolutional layer with `padding="same"` is:

$$\left(None, \frac{input\ height}{stride}, \frac{input\ width}{stride}, filters\right)$$

In the second convolutional layer, we choose the filters to be 3×3 and they now have depth 10, to match the number of channels in the previous layer. Since there are 20 filters in this layer, this gives a total number of parameters (weights) of $(3 \times 3 \times 10 + 1) \times 20 = 1{,}820$. Again, we use `strides = 2` and `padding = "same"`, so the width and height both halve. This gives us an overall output shape of `(None, 8, 8, 20)`.

After applying a series of `Conv2D` layers, we need to flatten the tensor using the Keras `Flatten` layer. This results in a set of $8 \times 8 \times 20 = 1{,}280$ units that we can connect to a final 10-unit `Dense` layer with softmax activation, which represents the probability of each category in a 10-category classification task.

This example demonstrates how we can chain convolutional layers together to create a convolutional neural network. Before we see how this compares in accuracy to our densely connected neural network, I'm going to introduce two more layer types that can also improve performance: `BatchNormalization` and `Dropout`.

REVIEWED
By Ivan at 11:00 am, Jun 13, 2020

Batch Normalization

One common problem when training a deep neural network is ensuring that the weights of the network remain within a reasonable range of values—if they start to become too large, this is a sign that your network is suffering from what is known as the *exploding gradient* problem. As errors are propagated backward through the

network, the calculation of the gradient in the earlier layers can sometimes grow exponentially large, causing wild fluctuations in the weight values. If your loss function starts to return `NaN`, chances are that your weights have grown large enough to cause an overflow error.

This doesn't necessarily happen immediately as you start training the network. Sometimes your network can be happily training for hours when suddenly the loss function returns `NaN` and your network has exploded. This can be incredibly annoying, especially when the network has seemingly been training well for a long time. To prevent this from happening, you need to understand the root cause of the exploding gradient problem.

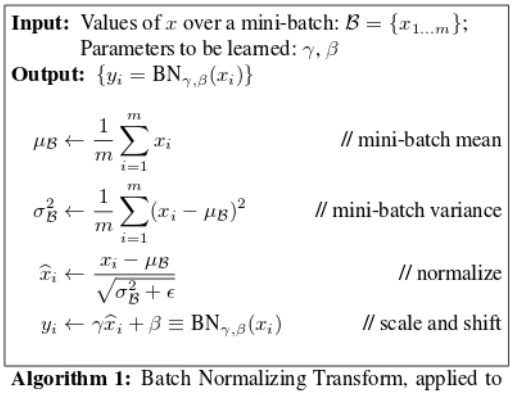
One of the reasons for scaling input data into a neural network is to ensure a stable start to training over the first few iterations. Since the weights of the network are initially randomized, unscaled input could potentially create huge activation values that immediately lead to exploding gradients. For example, instead of passing pixel values from 0–255 into the input layer, we usually scale these values to between –1 and 1.

Because the input is scaled, it's natural to expect the activations from all future layers to be relatively well scaled as well. Initially, this may be true, but as the network trains and the weights move further away from their random initial values, this assumption can start to break down. This phenomenon is known as *covariate shift*.

Imagine you're carrying a tall pile of books, and you get hit by a gust of wind. You move the books in a direction opposite to the wind to compensate, but in doing so, some of the books shift so that the tower is slightly more unstable than before. Initially, this is OK, but with every gust the pile becomes more and more unstable, until eventually the books have shifted so much that the pile collapses. This is covariate shift.

Relating this to neural networks, each layer is like a book in the pile. To remain stable, when the network updates the weights, each layer implicitly assumes that the distribution of its input from the layer beneath is approximately consistent across iterations. However, since there is nothing to stop any of the activation distributions shifting significantly in a certain direction, this can sometimes lead to runaway weight values and an overall collapse of the network.

Batch normalization is a solution that drastically reduces this problem. The solution is surprisingly simple. A batch normalization layer calculates the mean and standard deviation of each of its input channels across the batch and normalizes by subtracting the mean and dividing by the standard deviation. There are then two learned parameters for each channel, the scale (gamma) and shift (beta). The output is simply the normalized input, scaled by gamma and shifted by beta. Figure 2-15 shows the whole process.



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Figure 2-15. The batch normalization process[6]

REVIEWED
By Ivan at 11:13 am, Jun 13, 2020

We can place batch normalization layers after dense or convolutional layers to normalize the output from those layers. It's a bit like connecting the layers of books with small sets of adjustable springs that ensure there aren't any overall huge shifts in their positions over time.

You might be wondering how this layer works at test time. When it comes to prediction, we may only want to predict a single observation, so there is no *batch* over which to take averages. To get around this problem, during training a batch normalization layer also calculates the moving average of the mean and standard deviation of each channel and stores this value as part of the layer to use at test time.

NOTE

How many parameters are contained within a batch normalization layer? For every channel in the preceding layer, two weights need to be learned: the scale (gamma) and shift (beta). These are the *trainable* parameters. The moving average and standard deviation also need to be calculated for each channel but since they are derived from the data passing through the layer rather than trained through backpropagation, they are called *nontrainable* parameters. In total, this gives four parameters for each channel in the preceding layer, where two are trainable and two are nontrainable.

NOTE

In Keras, the `BatchNormalization` layer implements the batch normalization functionality:

```
BatchNormalization(momentum = 0.9)
```

The `momentum` parameter is the weight given to the previous value when calculating the moving average and moving standard deviation.

6 Source: Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," 11 February 2015, *https://arxiv.org/abs/1502.03167*.

Dropout Layers

When studying for an exam, it is common practice for students to use past papers and sample questions to improve their knowledge of the subject material. Some students try to memorize the answers to these questions, but then come unstuck in the exam because they haven't truly understood the subject matter. The best students use the practice material to further their general understanding, so that they are still able to answer correctly when faced with new questions that they haven't seen before.

The same principle holds for machine learning. Any successful machine learning algorithm must ensure that it generalizes to unseen data, rather than simply *remembering* the training dataset. If an algorithm performs well on the training dataset, but not the test dataset, we say that it is suffering from *overfitting*. To counteract this problem, we use *regularization* techniques, which ensure that the model is penalized if it starts to overfit.

There are many ways to regularize a machine learning algorithm, but for deep learning, one of the most common is by using *dropout* layers. This idea was introduced by Geoffrey Hinton in 2012 and presented in a 2014 paper by Srivastava et al.[7]

Dropout layers are very simple. During training, each dropout layer chooses a random set of units from the preceding layer and sets their output to zero, as shown in Figure 2-16.

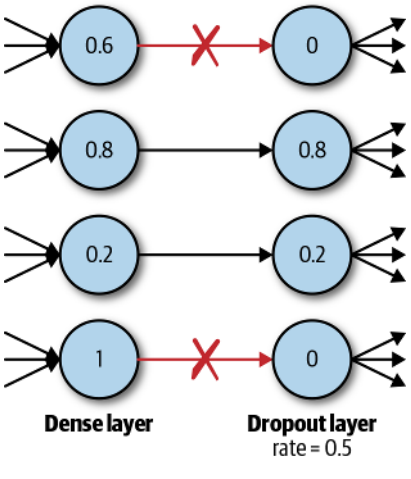


Figure 2-16. A dropout layer

7 Nitish Srivastava et al., "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research* 15 (2014): 1929–1958, *http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf*.

Incredibly, this simple addition drastically reduces overfitting, by ensuring that the network doesn't become overdependent on certain units or groups of units that, in effect, just remember observations from the training set. If we use dropout layers, the network cannot rely too much on any one unit and therefore knowledge is more evenly spread across the whole network. This makes the model much better at generalizing to unseen data, because the network has been trained to produce accurate predictions even under unfamiliar conditions, such as those caused by dropping random units. There are no weights to learn within a dropout layer, as the units to drop are decided stochastically. At test time, the dropout layer doesn't drop any units, so that the full network is used to make predictions.

Returning to our analogy, it's a bit like a math student practicing past papers with a random selection of key formulae missing from their formula book. This way, they learn how to answer questions through an understanding of the core principles, rather than always looking up the formulae in the same places in the book. When it comes to test time, they will find it much easier to answer questions that they have never seen before, due to their ability to generalize beyond the training material.

The `Dropout` layer in Keras implements this functionality, with the `rate` parameter specifying the proportion of units to drop from the preceding layer:

```
Dropout(rate = 0.25)
```

`Dropout` layers are used most commonly after `Dense` layers since these are most prone to overfitting due to the higher number of weights, though you can also use them after convolutional layers.

Batch normalization also has been shown to reduce overfitting, and therefore many modern deep learning architectures don't use dropout at all, and rely solely on batch normalization for regularization. As with most deep learning principles, there is no golden rule that applies in every situation—the only way to know for sure what's best is to test different architectures and see which performs best on a holdout set of data.

Putting It All Together

You've now seen three new Keras layer types: `Conv2D`, `BatchNormalization`, and `Dropout`. Let's put these pieces together into a new deep learning architecture and see how it performs on the CIFAR-10 dataset.

You can run the following example in the Jupyter notebook in the book repository called *02_03_deep_learning_conv_neural_network.ipynb*.

The model architecture we shall test is shown here:

colab - https://drive.google.com/file/d/1gfPb1zWuvSYnnnoXGlgmRqRAgbPTlA2v/view?usp=sharing

```
input_layer = Input((32,32,3))

x = Conv2D(filters = 32, kernel_size = 3
        , strides = 1, padding = 'same')(input_layer)
x = BatchNormalization()(x)
x = LeakyReLU()(x)

x = Conv2D(filters = 32, kernel_size = 3, strides = 2, padding = 'same')(x)
x = BatchNormalization()(x)
x = LeakyReLU()(x)

x = Conv2D(filters = 64, kernel_size = 3, strides = 1, padding = 'same')(x)
x = BatchNormalization()(x)
x = LeakyReLU()(x)

x = Conv2D(filters = 64, kernel_size = 3, strides = 2, padding = 'same')(x)
x = BatchNormalization()(x)
x = LeakyReLU()(x)

x = Flatten()(x)

x = Dense(128)(x)
x = BatchNormalization()(x)
x = LeakyReLU()(x)
x = Dropout(rate = 0.5)(x)

x = Dense(NUM_CLASSES)(x)
output_layer = Activation('softmax')(x)

model = Model(input_layer, output_layer)
```

3x3x3 *32 + 32 = 896

32 x 4 = 128

3x3x32 *32 + 32 = 9248

32 x 4 = 128

3x3x32 *64 + 64 = 18496

64 x 4 = 256

3x3x64 *64 + 64 = 36928

64 x 4 = 256

8 x 8 x 64 * 128 + 128=524416

128 x 4 = 512

128 x 10 + 10 = 1290

We use four stacked `Conv2D` layers, each followed by a `BatchNormalization` and a LeakyReLU layer. After flattening the resulting tensor, we pass the data through a `Dense` layer of size 128, again followed by a `BatchNormalization` and a `LeakyReLU` layer. This is immediately followed by a `Dropout` layer for regularization, and the network is concluded with an output `Dense` layer of size 10.

The order in which to use the `BatchNormalization` and `Activation` layers is a matter of preference. I like to place the `BatchNormalization` before the `Activation`, but some successful architectures use these layers the other way around. If you do choose to use `BatchNormalization` before `Activation` then you can remember the order using the acronym *BAD* (`BatchNormalization`, `Activation` then `Dropout`)!

The model summary is shown in Figure 2-17.

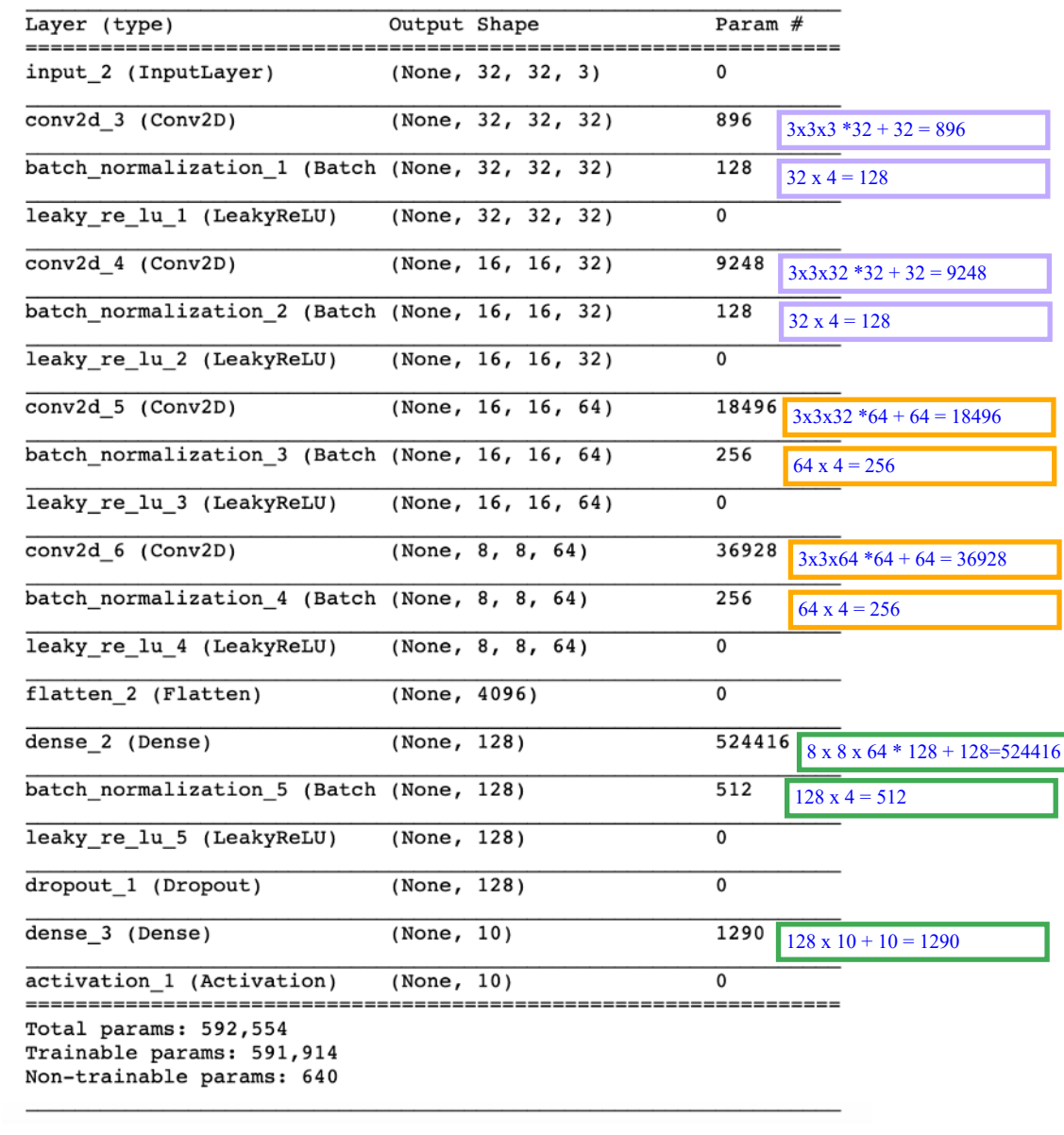


Figure 2-17. Convolutional neural network (CNN) for CIFAR-10

Before moving on, make sure you are able to calculate the output shape and number of parameters for each layer by hand. It's a good exercise to prove to yourself that you have fully understood how each layer is constructed and how it is connected to the preceding layer! Don't forget to include the bias weights that are included as part of the `Conv2D` and `Dense` layers.

We compile and train the model in exactly the same way as before and call the `evaluate` method to determine its accuracy on the holdout set (Figure 2-18).

```
model.evaluate(x_test, y_test, batch_size=1000)

 10000/10000 [==============================] - 15s 1ms/step

[0.8423407137393951, 0.7155999958515167]
```

Figure 2-18. CNN performance

As you can see, this model is now achieving 71.5% accuracy, up from 49.0% previously. Much better! Figure 2-19 shows some predictions from our new convolutional model.

Figure 2-19. CNN predictions

This improvement has been achieved simply by changing the architecture of the model to include convolutional, batch normalization, and dropout layers. Notice that the number of parameters is actually fewer in our new model than the previous model, even though the number of layers is far greater. This demonstrates the importance of being experimental with your model design and being comfortable with how the different layer types can be used to your advantage. When building generative models, it becomes even more important to understand the inner workings of your model since it is the middle layers of your network that capture the high-level features that you are most interested in.

Summary

This chapter introduced the core deep learning concepts that you will need to start building your first deep generative models.

A really important point to take away from this chapter is that deep neural networks are completely flexible by design, and there really are no fixed rules when it comes to model architecture. There are guidelines and best practices but you should feel free to experiment with layers and the order in which they appear. You will need to bear in mind that, like a set of building blocks, some layers will not fit together, simply because the input shape of one does not conform to the output shape of the other. This knowledge will come with experience and a solid understanding of how each layer changes the tensor shape as data flows through the network.

Another point to remember is that it is the *layers* in a deep neural network that are convolutional, rather than the network itself. When people talk about "convolutional neural networks," they really mean "neural networks that contain convolutional layers." It is important to make this distinction, because you shouldn't feel constrained to only use the architectures that you have read about in this book or elsewhere; instead, you should see them as examples of how you can piece together the different layer types. Like a child with a set of building blocks, the design of your neural network is only limited by your own imagination—and, crucially, your understanding of how the various layers fit together.

In the next chapter, we shall see how we can use these building blocks to design a network that can generate images.