2
Introduction to Convolutional Neural Networks

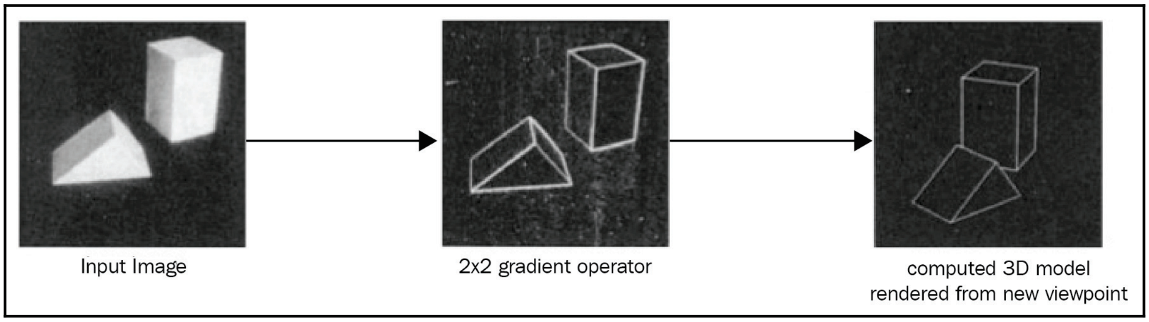
Convolutional Neural Networks (**CNNs**) are everywhere. In the last five years, we have seen a dramatic rise in the performance of visual recognition systems due to the introduction of deep architectures for feature learning and classification. CNNs have achieved good performance in a variety of areas, such as automatic speech understanding, computer vision, language translation, self-driving cars, and games such as Alpha Go. Thus, the applications of CNNs are almost limitless. DeepMind (from Google) recently published WaveNet, which uses a CNN to generate speech that mimics any human voice (`https://deepmind.com/blog/wavenet-generative-model-raw-audio/`).

In this chapter, we will cover the following topics:

- History of CNNs
- Overview of a CNN
- Image augmentation

History of CNNs

There have been numerous attempts to recognize pictures by machines for decades. It is a challenge to mimic the visual recognition system of the human brain in a computer. Human vision is the hardest to mimic and most complex sensory cognitive system of the brain. We will not discuss biological neurons here, that is, the primary visual cortex, but rather focus on artificial neurons. Objects in the physical world are three dimensional, whereas pictures of those objects are two dimensional. In this book, we will introduce neural networks without appealing to brain analogies. In 1963, computer scientist Larry Roberts, who is also known as the **father of computer vision**, described the possibility of extracting 3D geometrical information from 2D perspective views of blocks in his research dissertation titled **BLOCK WORLD**. This was the first breakthrough in the world of computer vision. Many researchers worldwide in machine learning and artificial intelligence followed this work and studied computer vision in the context of BLOCK WORLD. Human beings can recognize blocks regardless of any orientation or lighting changes that may happen. In this dissertation, he said that it is important to understand simple edge-like shapes in images. He extracted these edge-like shapes from blocks in order to make the computer understand that these two blocks are the same irrespective of orientation:

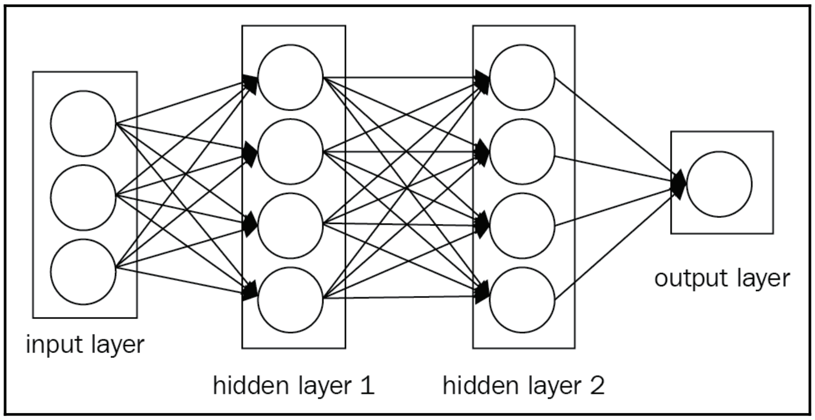


The vision starts with a simple structure. This is the beginning of computer vision as an engineering model. David Mark, an MIT computer vision scientist, gave us the next important concept, that vision is hierarchical. He wrote a very influential book named *VISION*. This is a simple book. He said that an image consists of several layers. These two principles form the basis of deep learning architecture, although they do not tell us what kind of mathematical model to use.

In the 1970s, the first visual recognition algorithm, known as the **generalized cylinder model**, came from the AI lab at Stanford University. The idea here is that the world is composed of simple shapes and any real-world object is a combination of these simple shapes. At the same time, another model, known as the **pictorial structure model**, was published from SRI Inc. The concept is still the same as the generalized cylinder model, but the parts are connected by springs; thus, it introduced a concept of variability. The first visual recognition algorithm was used in a digital camera by Fujifilm in 2006.

Convolutional neural networks

CNNs, or ConvNets, are quite similar to regular neural networks. They are still made up of neurons with weights that can be learned from data. Each neuron receives some inputs and performs a dot product. They still have a loss function on the last fully connected layer. They can still use a nonlinearity function. All of the tips and techniques that we learned from the last chapter are still valid for CNN. As we saw in the previous chapter, a regular neural network receives input data as a single vector and passes through a series of hidden layers. Every hidden layer consists of a set of neurons, wherein every neuron is fully connected to all the other neurons in the previous layer. Within a single layer, each neuron is completely independent and they do not share any connections. The last fully connected layer, also called the **output layer**, contains class scores in the case of an image classification problem. Generally, there are three main layers in a simple ConvNet. They are the **convolution layer**, the **pooling layer**, and the **fully connected layer**. We can see a simple neural network in the following image:



A regular three-layer neural network

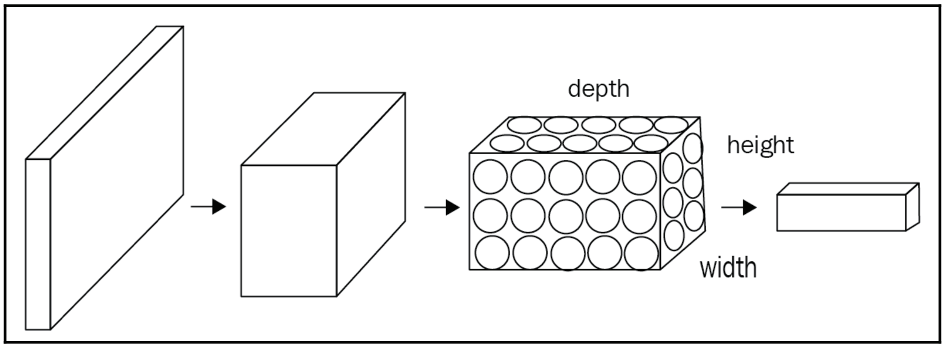
So, what changes? Since a CNN mostly takes images as input, this allows us to encode a few properties into the network, thus reducing the number of parameters.

In the case of real-world image data, CNNs perform better than **Multi-Layer Perceptrons** (**MLPs**). There are two reasons for this:

- In the last chapter, we saw that in order to feed an image to an MLP, we convert the input matrix into a simple numeric vector with no spatial structure. It has no knowledge that these numbers are spatially arranged. So, CNNs are built for this very reason; that is, to elucidate the patterns in multidimensional data. Unlike MLPs, CNNs understand the fact that image pixels that are closer in proximity to each other are more heavily related than pixels that are further apart:

$$CNN = Input\ layer + hidden\ layer + fully\ connected\ layer$$

- CNNs differ from MLPs in the types of hidden layers that can be included in the model. A ConvNet arranges its neurons in three dimensions: **width**, **height**, and **depth**. Each layer transforms its 3D input volume into a 3D output volume of neurons using activation functions. For example, in the following figure, the red input layer holds the image. Thus its width and height are the dimensions of the image, and the depth is three since there are Red, Green, and Blue channels:



ConvNets are deep neural networks that share their parameters across space.

How do computers interpret images?

Essentially, every image can be represented as a matrix of pixel values. In other words, images can be thought of as a function (f) that maps from R^2 to R.

$f(x, y)$ gives the intensity value at the position (x, y). In practice, the value of the function ranges only from *0* to *255*. Similarly, a color image can be represented as a stack of three functions. We can write this as a vector of:

$$f(x, y) = [\ r(x,y)\ g(x,y)\ b(x,y)]$$

Or we can write this as a mapping:

$$f\!: R \ x \ R \text{ --> } R3$$

So, a color image is also a function, but in this case, a value at each (x,y) position is not a single number. Instead it is a vector that has three different light intensities corresponding to three color channels. The following is the code for seeing the details of an image as input to a computer.

Code for visualizing an image

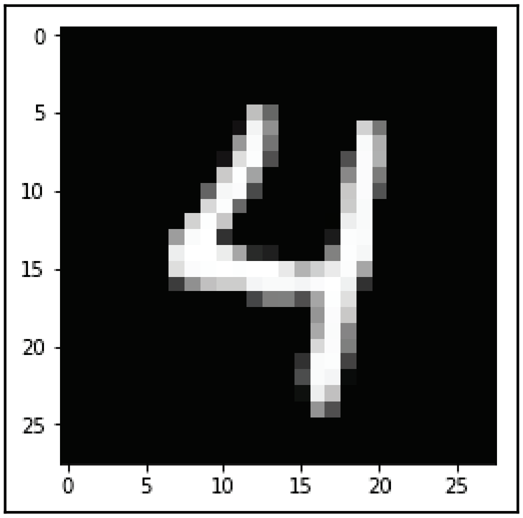
Let's take a look at how an image can be visualized with the following code:

```
#import all required lib
import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
from skimage.io import imread
from skimage.transform import resize

# Load a color image in grayscale
image = imread('sample_digit.png',as_grey=True)
image = resize(image,(28,28),mode='reflect')
print('This image is: ',type(image),
         'with dimensions:', image.shape)

plt.imshow(image,cmap='gray')
```

We obtain the following image as a result:

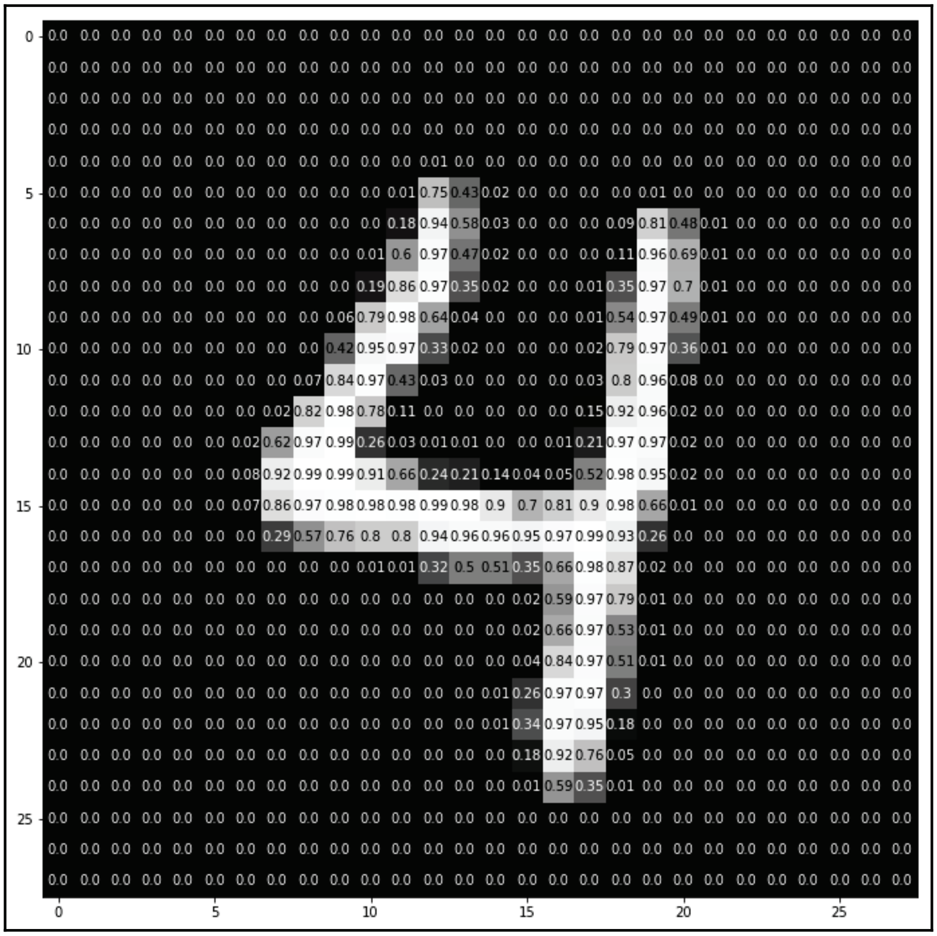


```
def visualize_input(img, ax):

    ax.imshow(img, cmap='gray')
    width, height = img.shape
    thresh = img.max()/2.5
    for x in range(width):
        for y in range(height):
            ax.annotate(str(round(img[x][y],2)), xy=(y,x),
                        horizontalalignment='center',
                        verticalalignment='center',
                        color='white' if img[x][y]<thresh else 'black')

fig = plt.figure(figsize = (12,12))
ax = fig.add_subplot(111)
visualize_input(image, ax)
```

The following result is obtained:



In the previous chapter, we used an MLP-based approach to recognize images. There are two issues with that approach:

- It increases the number of parameters
- It only accepts vectors as input, that is, flattening a matrix to a vector

This means we must find a new way to process images, in which 2D information is not completely lost. CNNs address this issue. Furthermore, CNNs accept matrices as input. Convolutional layers preserve spatial structures. First, we define a convolution window, also called a **filter**, or **kernel**; then slide this over the image.

Dropout

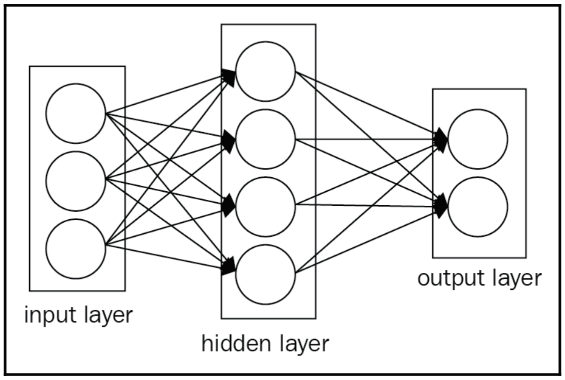
A neural network can be thought of as a search problem. Each node in the neural network is searching for correlation between the input data and the correct output data.

Dropout randomly turns nodes off while forward-propagating and thus helps ward off weights from converging to identical positions. After this is done, it turns on all the nodes and back-propagates. Similarly, we can set some of the layer's values to zero at random during forward propagation in order to perform dropout on a layer.

Use dropout only during training. Do not use it at runtime or on your testing dataset.

Input layer

The **input layer** holds the image data. In the following figure, the input layer consists of three inputs. In a **fully connected layer**, the neurons between two adjacent layers are fully connected pairwise but do not share any connection within a layer. In other words, the neurons in this layer have full connections to all activations in the previous layer. Therefore, their activations can be computed with a simple matrix multiplication, optionally adding a bias term. The difference between a fully connected and convolutional layer is that neurons in a convolutional layer are connected to a local region in the input, and that they also share parameters:



Convolutional layer

The main objective of convolution in relation to ConvNet is to extract features from the input image. This layer does most of the computation in a ConvNet. We will not go into the mathematical details of convolution here but will get an understanding of how it works over images.

The ReLU activation function is extremely useful in CNNs.

Convolutional layers in Keras

To create a convolutional layer in Keras, you must first import the required modules as follows:

```
from keras.layers import Conv2D
```

Then, you can create a convolutional layer by using the following format:

```
Conv2D(filters, kernel_size, strides, padding, activation='relu',
input_shape)
```

You must pass the following arguments:

- `filters`: The number of filters.
- `kernel_size`: A number specifying both the height and width of the (square) convolution window. There are also some additional optional arguments that you might like to tune.
- `strides`: The stride of the convolution. If you don't specify anything, this is set to one.
- `padding`: This is either `valid` or `same`. If you don't specify anything, the padding is set to `valid`.
- `activation`: This is typically `relu`. If you don't specify anything, no activation is applied. You are strongly encouraged to add a ReLU activation function to every convolutional layer in your networks.

It is possible to represent both `kernel_size` and `strides` as either a number or a tuple.

When using your convolutional layer as the first layer (appearing after the input layer) in a model, you must provide an additional `input_shape` argument—`input_shape`. It is a tuple specifying the height, width, and depth (in that order) of the input.

Please make sure that the `input_shape` argument is not included if the convolutional layer is not the first layer in your network.

There are many other tunable arguments that you can set to change the behavior of your convolutional layers:

- **Example 1**: In order to build a CNN with an input layer that accepts images of 200 x 200 pixels in grayscale. In such cases, the next layer would be a convolutional layer of 16 filters with width and height as 2. As we go ahead with the convolution we can set the filter to jump 2 pixels together. Therefore, we can build a convolutional, layer with a filter that doesn't pad the images with zeroes with the following code:

```
Conv2D(filters=16, kernel_size=2, strides=2, activation='relu',
input_shape=(200, 200, 1))
```

- **Example 2**: After we build our CNN model, we can have the next layer in it to be a convolutional layer. This layer will have 32 filters with width and height as 3, which would take the layer that was constructed in the previous example as its input. Here, as we proceed with the convolution, we will set the filter to jump one pixel at a time, such that the convolutional layer will be able to see all the regions of the previous layer too. Such a convolutional layer can be constructed with the help of the following code:

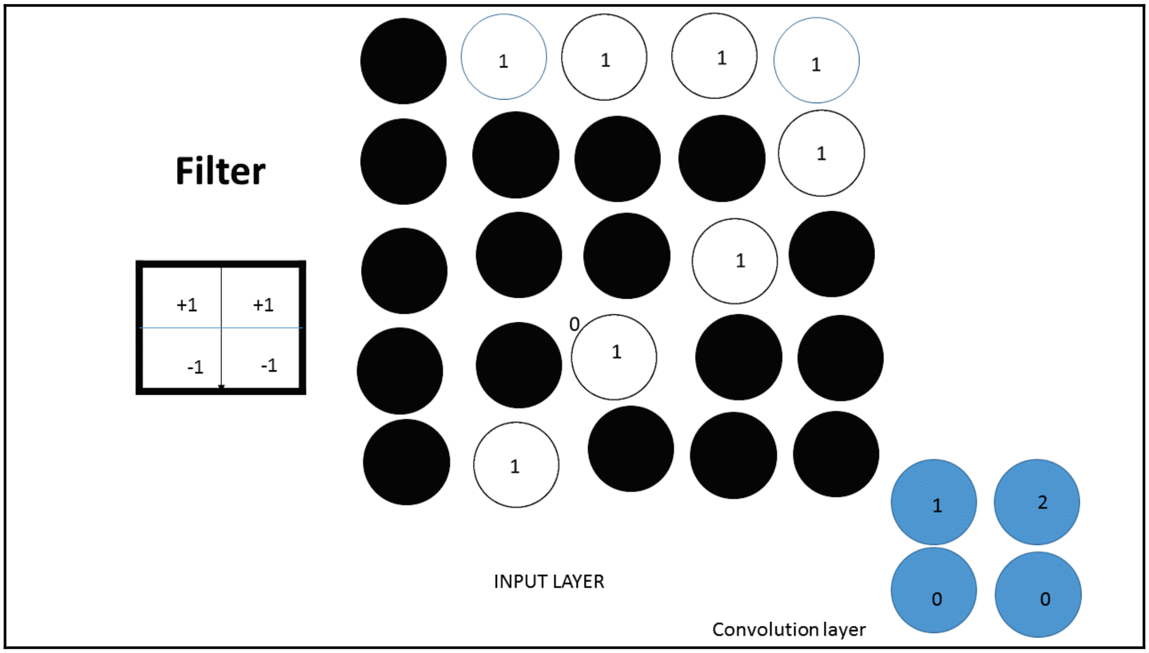
```
Conv2D(filters=32, kernel_size=3, padding='same',
activation='relu')
```

- **Example 3**: You can also construct convolutional layers in Keras of size 2 x 2, with 64 filters and a ReLU activation function. Here, the convolution utilizes a stride of 1 with padding set to `valid` and all other arguments set to their default values. Such a convolutional layer can be built using the following code:

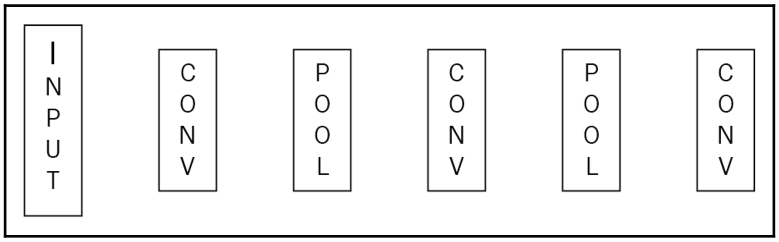
```
Conv2D(64, (2,2), activation='relu')
```

Pooling layer

As we have seen, a convolutional layer is a stack of feature maps, with one feature map for each filter. More filters increase the dimensionality of convolution. Higher dimensionality indicates more parameters. So, the pooling layer controls overfitting by progressively reducing the spatial size of the representation to reduce the number of parameters and computation. The pooling layer often takes the convolutional layer as input. The most commonly used pooling approach is **max pooling**. In addition to max pooling, pooling units can also perform other functions such as **average pooling**. In a CNN, we can control the behavior of the convolutional layer by specifying the size of each filter and the number of filters. To increase the number of nodes in a convolutional layer, we can increase the number of filters, and to increase the size of the pattern, we can increase the size of the filter. There are also a few other hyperparameters that can be tuned. One of them is the stride of the convolution. Stride is the amount by which the filter slides over the image. A stride of 1 moves the filter by 1 pixel horizontally and vertically. Here, the convolution becomes the same as the width and depth of the input image. A stride of 2 makes a convolutional layer of half of the width and height of the image. If the filter extends outside of the image, then we can either ignore these unknown values or replace them with zeros. This is known as **padding**. In Keras, we can set `padding = 'valid'` if it is acceptable to lose a few values. Otherwise, set `padding = 'same'`:



A very simple ConvNet looks like this:



Practical example – image classification

The convolutional layer helps to detect regional patterns in an image. The max pooling layer, present after the convolutional layer, helps reduce dimensionality. Here is an example of image classification using all the principles we studied in the previous sections. One important notion is to first make all the images into a standard size before doing anything else. The first convolution layer requires an additional `input.shape()` parameter. In this section, we will train a CNN to classify images from the CIFAR-10 database. CIFAR-10 is a dataset of 60,000 color images of 32 x 32 size. These images are labeled into 10 categories with 6,000 images each. These categories are airplane, automobile, bird, cat, dog, deer, frog, horse, ship, and truck. Let's see how to do this with the following code:

```
import keras
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

fig = plt.figure(figsize=(20,5))
for i in range(36):
    ax = fig.add_subplot(3, 12, i + 1, xticks=[], yticks=[])
    ax.imshow(np.squeeze(x_train[i]))from keras.datasets import cifar10

# rescale [0,255] --> [0,1]
x_train = x_train.astype('float32')/255
from keras.utils import np_utils

# one-hot encode the labels
num_classes = len(np.unique(y_train))
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

```
# break training set into training and validation sets
(x_train, x_valid) = x_train[5000:], x_train[:5000]
(y_train, y_valid) = y_train[5000:], y_train[:5000]

# print shape of training set
print('x_train shape:', x_train.shape)

# printing number of training, validation, and test images
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
print(x_valid.shape[0], 'validation samples')x_test =
x_test.astype('float32')/255


from keras.models import Sequential
from keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout

model = Sequential()
model.add(Conv2D(filters=16, kernel_size=2, padding='same',
activation='relu',
                        input_shape=(32, 32, 3)))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32, kernel_size=2, padding='same',
activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=64, kernel_size=2, padding='same',
activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Conv2D(filters=32, kernel_size=2, padding='same',
activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.3))
model.add(Flatten())
model.add(Dense(500, activation='relu'))
model.add(Dropout(0.4))
model.add(Dense(10, activation='softmax'))

model.summary()

# compile the model
model.compile(loss='categorical_crossentropy', optimizer='rmsprop',
                  metrics=['accuracy'])
from keras.callbacks import ModelCheckpoint

# train the model
checkpointer = ModelCheckpoint(filepath='model.weights.best.hdf5',
verbose=1,
                               save_best_only=True)
```

```
hist = model.fit(x_train, y_train, batch_size=32, epochs=100,
          validation_data=(x_valid, y_valid), callbacks=[checkpointer],
          verbose=2, shuffle=True)
```

Image augmentation

While training a CNN model, we do not want the model to change any prediction based on the size, angle, and position of the image. The image is represented as a matrix of pixel values, so the size, angle, and position have a huge effect on the pixel values. To make the model more size-invariant, we can add different sizes of the image to the training set. Similarly, in order to make the model more rotation-invariant, we can add images with different angles. This process is known as **image data augmentation**. This also helps to avoid overfitting. Overfitting happens when a model is exposed to very few samples. Image data augmentation is one way to reduce overfitting, but it may not be enough because augmented images are still correlated. Keras provides an image augmentation class called `ImageDataGenerator` that defines the configuration for image data augmentation. This also provides other features such as:

- Sample-wise and feature-wise standardization
- Random rotation, shifts, shear, and zoom of the image
- Horizontal and vertical flip
- ZCA whitening
- Dimension reordering
- Saving the changes to disk

An augmented image generator object can be created as follows:

```
imagedatagen = ImageDataGenerator()
```

This API generates batches of tensor image data in real-time data augmentation, instead of processing an entire image dataset in memory. This API is designed to create augmented image data during the model fitting process. Thus, it reduces the memory overhead but adds some time cost for model training.

After it is created and configured, you must fit your data. This computes any statistics required to perform the transformations to image data. This is done by calling the `fit()` function on the data generator and passing it to the training dataset, as follows:

```
imagedatagen.fit(train_data)
```

The batch size can be configured, the data generator can be prepared, and batches of images can be received by calling the `flow()` function:

```
imagedatagen.flow(x_train, y_train, batch_size=32)
```

Finally, call the `fit_generator()` function instead of calling the `fit()` function on the model:

```
fit_generator(imagedatagen, samples_per_epoch=len(X_train), epochs=200)
```

Let's look at some examples to understand how the image augmentation API in Keras works. We will use the MNIST handwritten digit recognition task in these examples.

Let's begin by taking a look at the first nine images in the training dataset:

```
#Plot images
from keras.datasets import mnist
from matplotlib import pyplot
#loading data
(X_train, y_train), (X_test, y_test) = mnist.load_data()
#creating a grid of 3x3 images
for i in range(0, 9):
  pyplot.subplot(330 + 1 + i)
  pyplot.imshow(X_train[i], cmap=pyplot.get_cmap('gray'))
#Displaying the plot
pyplot.show()
```

The following code snippet creates augmented images from the CIFAR-10 dataset. We will add these images to the training set of the last example and see how the classification accuracy increases:

```
from keras.preprocessing.image import ImageDataGenerator
# creating and configuring augmented image generator
datagen_train = ImageDataGenerator(
 width_shift_range=0.1, # shifting randomly images horizontally (10% of
total width)
 height_shift_range=0.1, # shifting randomly images vertically (10% of
total height)
 horizontal_flip=True) # flipping randomly images horizontally
# creating and configuring augmented image generator
```

```
datagen_valid = ImageDataGenerator(
 width_shift_range=0.1, # shifting randomly images horizontally (10% of
total width)
 height_shift_range=0.1, # shifting randomly images vertically (10% of
total height)
 horizontal_flip=True) # flipping randomly images horizontally
# fitting augmented image generator on data
datagen_train.fit(x_train)
datagen_valid.fit(x_valid)
```

Summary

We began this chapter by briefly looking into the history of CNNs. We introduced you to the implementation of visualizing images.

We studied image classification with the help of a practical example, using all the principles we learned about in the chapter. Finally, we learned how image augmentation helps us avoid overfitting and studied the various other features provided by image augmentation.

In the next chapter, we will learn how to build a simple image classifier CNN model from scratch.