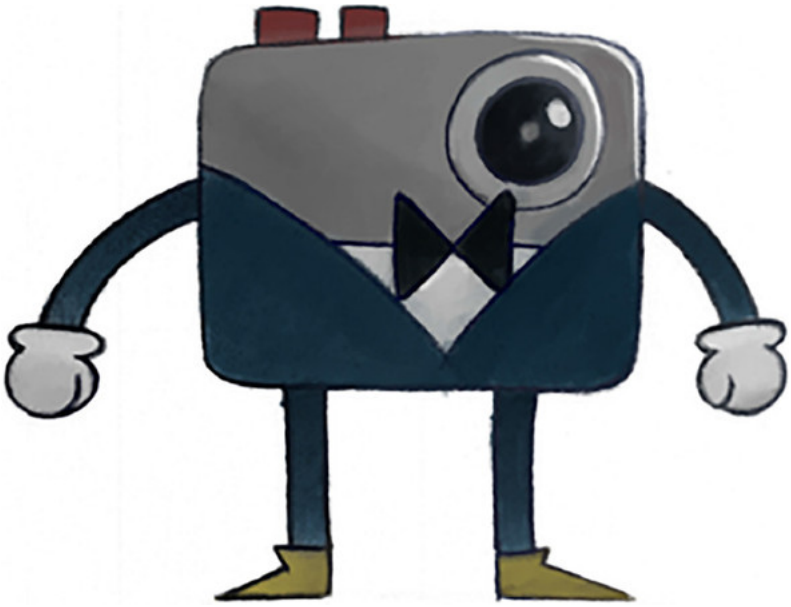
PREV
Chapter 8. Reinforcement learning

NEXT
Chapter 10. Recurrent neural networks

Chapter 9. Convolutional neural networks



This chapter covers

- Examining the components of a convolutional neural network
- Classifying natural images using deep learning
- Improving neural network performance—tips and tricks

Grocery shopping after an exhausting day is a taxing experience. My eyes get bombarded with too much information. Sales, coupons, colors, toddlers, flashing lights, and crowded aisles are just a few examples of all the signals forwarded to my visual cortex, whether or not I actively try to pay attention. The visual system absorbs an abundance of information.

Ever heard the phrase “a picture is worth a thousand words”? That might be true for you or me, but can a machine find meaning within images as well? The photoreceptor cells in our retinas pick up wavelengths of light, but that information doesn’t seem to propagate up to our consciousness. After all, I can’t put into words exactly what wavelengths of lights I’m picking up. Similarly, a camera picks up pixels, yet we want to squeeze out some form of higher-level knowledge instead, such as names or locations of objects. How do we get from pixels to human-level perception?

To achieve intelligent meaning from raw sensory input with machine learning, you'll design a neural network model. In the previous chapters, you've seen a few types of neural network models such as fully connected ones (chapter 8) and autoencoders (chapter 7). In this chapter, you'll meet another type of model called a *convolutional neural network* (CNN), which performs exceptionally well on images and other sensory data such as audio. For example, a CNN model can reliably classify what object is being displayed in an image.

The CNN model that you'll implement in this chapter will learn how to classify images to 1 of 10 candidate categories. In effect, "a picture is worth only *one* word" out of just 10 possibilities. It's a tiny step toward human-level perception, but you have to start somewhere, right?

9.1. DRAWBACK OF NEURAL NETWORKS

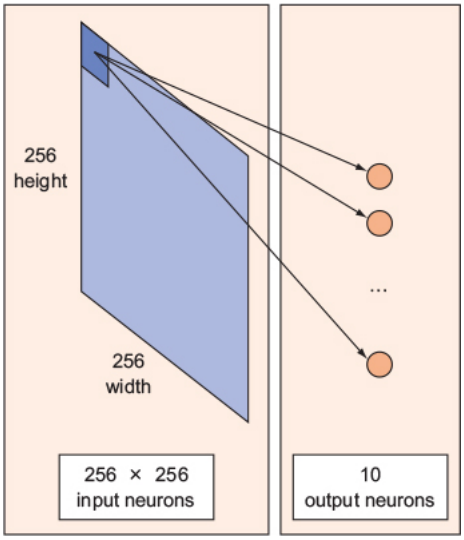
Machine learning constitutes an eternal struggle of designing a model that's expressive enough to represent the data, yet not so flexible that it overfits and memorizes the patterns. Neural networks are proposed as a way to improve that expressive power; yet, as you may guess, they often suffer from the pitfalls of overfitting.

Note

Overfitting occurs when your learned model performs exceptionally well on the training dataset, yet tends to perform poorly on the test dataset. The model is likely too flexible for what little data is available, and it ends up more or less memorizing the training data.

A quick and dirty heuristic you can use to compare the flexibility of two machine-learning models is to count the number of parameters to be learned. As shown in figure 9.1, a fully connected neural network that takes in a 256 × 256 image and maps it to a layer of 10 neurons will have 256 × 256 × 10 = 655,360 parameters! Compare that to a model with perhaps only 5 parameters. It's likely that the fully connected neural network can represent more-complex data than the model with 5 parameters.

Figure 9.1. In a fully connected network, each pixel of an image is treated as an input. For a grayscale image of size 256 × 256, that's 256 × 256 neurons! Connecting each neuron to 10 outputs yields 256 × 256 × 10 = 655,360 weights.



The next section introduces convolutional neural networks, which are a clever way to reduce the number of parameters. Instead of dealing with a fully connected network, the CNN approach reuses the same parameter multiple times.

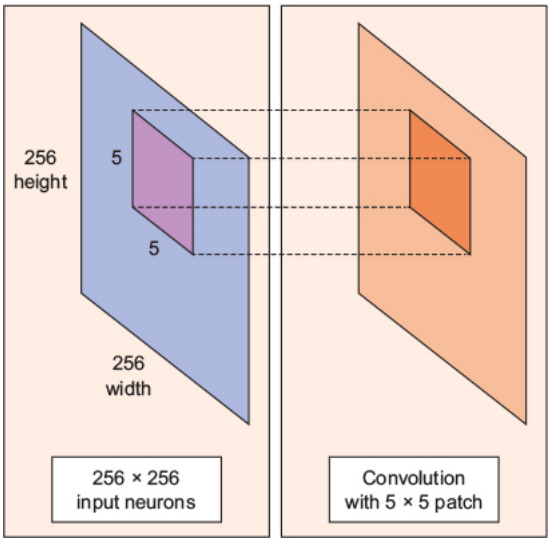
9.2. CONVOLUTIONAL NEURAL NETWORKS

The big idea behind convolutional neural networks is that a local understanding of an image is good enough. The practical benefit is that having fewer parameters greatly improves the time it takes to learn as well as reduces the amount of data required to train the model.

Instead of a fully connected network of weights from each pixel, a CNN has just enough weights to look at a small patch of the image. It's like reading a book by using a magnifying glass; eventually, you read the whole page, but you look at only a small patch of the page at any given time.

Consider a 256 × 256 image. Instead of your TensorFlow code processing the whole image at once, it can efficiently scan it chunk by chunk—say, a 5 × 5 window. The 5 × 5 window slides along the image (usually left to right, and top to bottom), as shown in figure 9.2. How "quickly" it slides is called its *stride length*. For example, a stride length of 2 means the 5 × 5 sliding window moves by 2 pixels at a time until it spans the entire image. In TensorFlow, you can easily adjust the stride length and window size by using the built-in library functions, as you'll soon see.

Figure 9.2. Convolving a 5 × 5 patch over an image, as shown on the left, produces another image, as shown on the right. In this case, the produced image is the same size as the original. Converting an original image to a convolved image requires only 5 × 5 = 25 parameters!



This 5 × 5 window has an associated 5 × 5 matrix of weights.

Definition

A *convolution* is a weighted sum of the pixel values of the image, as the window slides across the whole image. Turns out, this convolution process throughout an image with a weight matrix produces another image (of the same size, depending on the convention). *Convolving* is the process of applying a convolution.

The sliding-window shenanigans happen in the *convolution layer* of the neural network. A typical CNN has multiple convolution layers. Each convolutional layer typically generates many alternate convolutions, so the weight matrix is a tensor of 5 × 5 × n, where n is the number of convolutions.

As an example, let's say an image goes through a convolution layer on a weight matrix of 5 × 5 × 64. It generates 64 convolutions by sliding a 5 × 5 window. Therefore, this model has 5 × 5 × 64 (= 1,600) parameters, which is remarkably fewer parameters than a fully connected network, 256 × 256 (= 65,536).

The beauty of the CNN is that the number of parameters is independent of the size of the original image. You can run the same CNN on a 300 × 300 image, and the number of parameters won't change in the convolution layer!

9.3. PREPARING THE IMAGE

To start implementing CNNs in TensorFlow, let's first obtain some images to work with. The code listings in this section will help you set up a training dataset for the remainder of the chapter.

First, download the CIFAR-10 dataset from www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz (http://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz). This dataset contains 60,000 images, evenly split into 10 categories, which makes it a great resource for classification tasks. Go ahead and extract that file to your working directory. Figure 9.3 shows examples of images from the dataset.

Figure 9.3. Images from the CIFAR-10 dataset. Because they're only 32 × 32 in size, they're a bit difficult to see, but you can generally recognize some of the objects.

You used the CIFAR-10 dataset in the previous chapter about autoencoders, so let's pull up that code again. The following listing comes straight from the CIFAR-10 documentation located at www.cs.toronto.edu/~kriz/cifar.html (http://www.cs.toronto.edu/~kriz/cifar.html). Place the code in a file called cifar_tools.py.

Listing 9.1. Loading images from a CIFAR-10 file in Python

```
import pickle

def unpickle(file):
    fo = open(file, 'rb')
    dict = pickle.load(fo, encoding='latin1')
    fo.close()
    return dict
```

Neural networks are already prone to overfitting, so it's essential that you do as much as you can to minimize that error. For that reason, always remember to clean the data before processing it.

Cleaning data is a core process in the machine-learning pipeline. Listing 9.2 implements the following three steps for cleaning a dataset of images:

1. If you have an image in color, try converting it to grayscale to lower the dimensionality of the input data, and consequently lower the number of parameters.
2. Consider center-cropping the image, because the edges of an image might provide no useful information.
3. Normalize your input by subtracting the mean and dividing by the standard deviation of each data sample so that the gradients during back-propagation don't change too dramatically.

The following listing shows how to clean a dataset of images by using these techniques.

Listing 9.2. Cleaning data

```
import numpy as np

def clean(data):
    imgs = data.reshape(data.shape[0], 3, 32, 32)             1
    grayscale_imgs = imgs.mean(1)                             2
    cropped_imgs = grayscale_imgs[:, 4:28, 4:28]              3
    img_data = cropped_imgs.reshape(data.shape[0], -1)
    img_size = np.shape(img_data)[1]
    means = np.mean(img_data, axis=1)
    meansT = means.reshape(len(means), 1)
    stds = np.std(img_data, axis=1)
    stdsT = stds.reshape(len(stds), 1)
    adj_stds = np.maximum(stdsT, 1.0 / np.sqrt(img_size))
    normalized = (img_data - meansT) / adj_stds               4
    return normalized
```

- ***1* Reorganizes the data so it's a 32 × 32 matrix with three channels**
- ***2* Grayscales the image by averaging the color intensities**
- ***3* Crops the 32 × 32 image to a 24 × 24 image**
- ***4* Normalizes the pixels' values by subtracting the mean and dividing by standard deviation**

Collect all the images from CIFAR-10 into memory, and run your cleaning function on them. The following listing sets up a convenient method to read, clean, and structure your data for use in TensorFlow. Include this in cifar_tools.py, as well.

Listing 9.3. Preprocessing all CIFAR-10 files

```
def read_data(directory):
    names = unpickle('{}/batches.meta'.format(directory))['label_names']
    print('names', names)

    data, labels = [], []
    for i in range(1, 6):
        filename = '{}/data_batch_{}'.format(directory, i)
        batch_data = unpickle(filename)
        if len(data) > 0:
            data = np.vstack((data, batch_data['data']))
            labels = np.hstack((labels, batch_data['labels']))
        else:
            data = batch_data['data']
            labels = batch_data['labels']

    print(np.shape(data), np.shape(labels))

    data = clean(data)
    data = data.astype(np.float32)
    return names, data, labels
```

In another file called using_cifar.py, you can now use the method by importing cifar_tools. Listings 9.4 and 9.5 show how to sample a few images from the dataset and visualize them.

Listing 9.4. Using the `cifar_tools` helper function

```
import cifar_tools
```

```
names, data, labels = \
    cifar_tools.read_data('your/location/to/cifar-10-batches-py')
```

You can randomly select a few images and draw them along their corresponding label. The following listing does exactly that, so you can get a better understanding of the type of data you'll be dealing with.

Listing 9.5. Visualizing images from the dataset

```
import numpy as np
import matplotlib.pyplot as plt
import random
def show_some_examples(names, data, labels):
    plt.figure()
    rows, cols = 4, 4                                          1
    random_idxs = random.sample(range(len(data)), rows * cols)    2
    for i in range(rows * cols):
        plt.subplot(rows, cols, i + 1)
        j = random_idxs[i]
        plt.title(names[labels[j]])
        img = np.reshape(data[j, :], (24, 24))
        plt.imshow(img, cmap='Greys_r')
        plt.axis('off')
    plt.tight_layout()
    plt.savefig('cifar_examples.png')

show_some_examples(names, data, labels)
```

- ***1*** **Change this to as many rows and columns as you desire.**
- ***2*** **Randomly pick images from the dataset to show**

By running this code, you'll generate a file called cifar_examples.png that will look similar to figure 9.3.

9.3.1. Generating filters

In this section, you'll convolve an image with a couple of random 5 × 5 patches, also called *filters*. This is an important step in a convolutional neural network, so you'll carefully examine how the data transforms. To understand a CNN model for image processing, it's wise to observe the way an image filter transforms an image. Filters are a way to extract useful image features such as edges and shapes. You can train a machine-learning model on these features.

Remember: a feature vector indicates how you represent a data point. When you apply a filter to an image, the corresponding point in the transformed image is a feature—a feature that says, "When you apply this filter to this point, it now has this new value." The more filters you use on an image, the greater the dimensionality of the feature vector.

Open a new file called conv_visuals.py. Let's randomly initialize 32 filters. You'll do so by defining a variable called `W` of size 5 × 5 × 1 × 32. The first two dimensions correspond to the filter size. The last dimension corresponds to the 32 convolutions. The 1 in the variable's size corresponds to the input dimension, because the `conv2d` function is capable of convolving images of multiple channels. (In our example, you

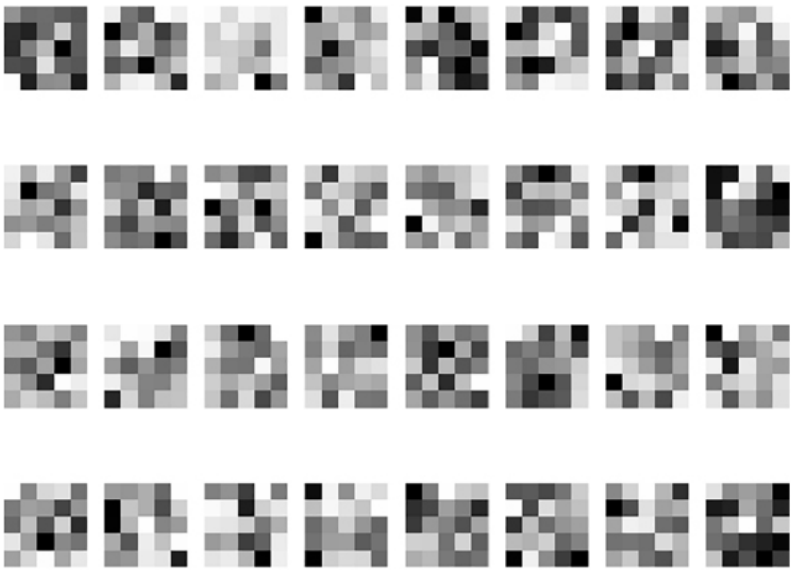
care about only grayscale images, so the number of input channels is 1.) The following listing provides the code to generate filters, which are shown in figure 9.4.

Listing 9.6. Generating and visualizing random filters

```
W = tf.Variable(tf.random_normal([5, 5, 1, 32]))              1

def show_weights(W, filename=None):
    plt.figure()
    rows, cols = 4, 8                                         2
    for i in range(np.shape(W)[3]):                           3
        img = W[:, :, 0, i]
        plt.subplot(rows, cols, i + 1)
        plt.imshow(img, cmap='Greys_r', interpolation='none')
        plt.axis('off')
    if filename:
        plt.savefig(filename)
    else:
        plt.show()
```

- ***1* Defines the tensor representing the random filters**
- ***2* Defines just enough rows and columns to show the 32 figures in figure 9.4**
- ***3* Visualizes each filter matrix**

Figure 9.4. These are 32 randomly initialized matrices, each of size 5 × 5. They represent the filters you'll use to convolve an input image.



Exercise 9.1

Change listing 9.6 to generate 64 filters of size 3 × 3.

Answer

```
W = tf.Variable(tf.random_normal([3, 3, 1, 64]))
```

Use a session, as shown in the following listing, and initialize some weights by using the `global_variables_initializer` op. Call the `show_weights` function to visualize random filters, as shown in figure 9.4.

Listing 9.7. Using a session to initialize weights

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    W_val = sess.run(W)
    show_weights(W_val, 'step0_weights.png')
```

9.3.2. Convolving using filters

The previous section prepared filters to use. In this section, you'll use TensorFlow's convolve function on your randomly generated filters. The following listing sets up code to visualize the convolution outputs. You'll use it later, just as you used `show_weights`.

Listing 9.8. Showing convolution results

```
def show_conv_results(data, filename=None):
    plt.figure()
    rows, cols = 4, 8
    for i in range(np.shape(data)[3]):
        img = data[0, :, :, i]                                  1
        plt.subplot(rows, cols, i + 1)
        plt.imshow(img, cmap='Greys_r', interpolation='none')
        plt.axis('off')
    if filename:
        plt.savefig(filename)
    else:
        plt.show()
```

- ***1*** **Unlike in listing 9.6, this time the shape of the tensor is different.**

Let's say you have an example input image, such as the one shown in figure 9.5. You can convolve the 24 × 24 image by using 5 × 5 filters to produce many convolved images. All these convolutions are unique perspectives of looking at the same image. These different perspectives work together to comprehend the object that exists in the image. The following listing shows how to do this, step by step.

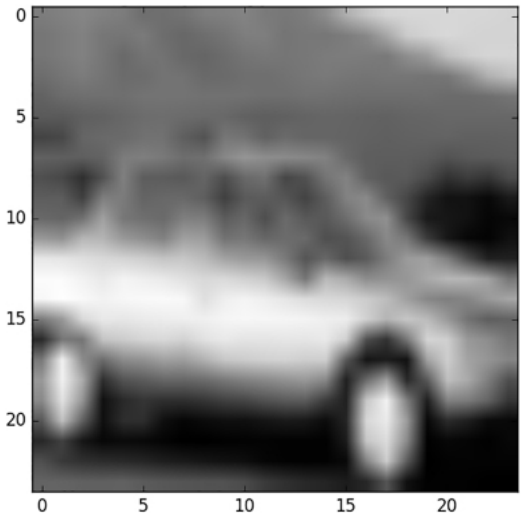
Listing 9.9. Visualizing convolutions

```
raw_data = data[4, :]                                           1
raw_img = np.reshape(raw_data, (24, 24))                        1
plt.figure()                                                    1
plt.imshow(raw_img, cmap='Greys_r')                             1
plt.savefig('input_image.png')                                  1

x = tf.reshape(raw_data, shape=[-1, 24, 24, 1])                 2

b = tf.Variable(tf.random_normal([32]))                         3
conv = tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME') 3
conv_with_b = tf.nn.bias_add(conv, b)                           3
conv_out = tf.nn.relu(conv_with_b)                              3

with tf.Session() as sess:                                      4
    sess.run(tf.global_variables_initializer())                 4
```

```
                                                                    4
    conv_val = sess.run(conv)                                       4
    show_conv_results(conv_val, 'step1_convs.png')                  4
    print(np.shape(conv_val))                                       4
                                                                    4
conv_out_val = sess.run(conv_out)                                   4
    show_conv_results(conv_out_val, 'step2_conv_outs.png')          4
    print(np.shape(conv_out_val))                                   4
```

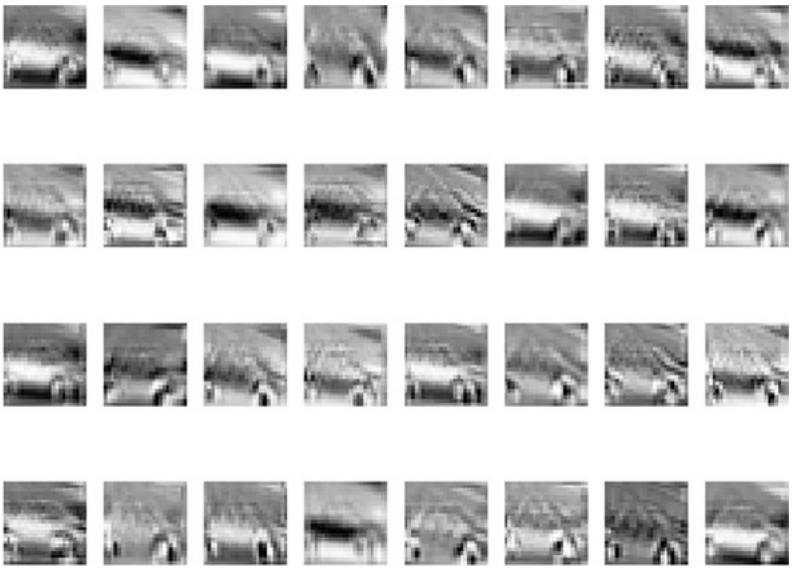
- ***1*** **Gets an image from the CIFAR dataset, and visualizes it**
- ***2*** **Defines the input tensor for the 24 × 24 image**
- ***3*** **Defines the filters and corresponding parameters**
- ***4*** **Runs the convolution on the selected image**

Figure 9.5. An example 24 × 24 image from the CIFAR-10 dataset



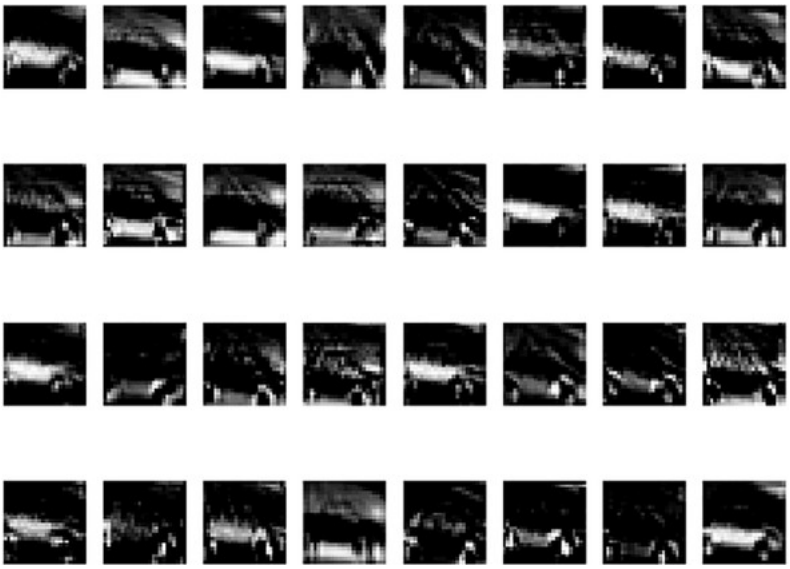
Finally, by running the `conv2d` function in TensorFlow, you get the 32 images in figure 9.6. The idea of convolving images is that each of the 32 convolutions captures different features about the image.

Figure 9.6. Resulting images from convolving the random filters on an image of a car



With the addition of a bias term and an activation function such as `relu` (see listing 9.12 for an example), the convolution layer of the network behaves nonlinearly, which improves its expressiveness. Figure 9.7 shows what each of the 32 convolution outputs becomes.

Figure 9.7. After you add a bias term and an activation function, the resulting convolutions can capture more-powerful patterns within images.



9.3.3. Max pooling

After a convolution layer extracts useful features, it's usually a good idea to reduce the size of the convolved outputs. Rescaling or subsampling a convolved output helps reduce the number of parameters, which in turn can help to not overfit the data.

This is the main idea behind a technique called *max pooling*, which sweeps a window across an image and picks the pixel with the maximum value. Depending on the stride length, the resulting image is a fraction of the size of the original. This is useful because it lessens the dimensionality of the data, consequently reducing the number of parameters in future steps.

Exercise 9.2

Let's say you want to max pool over a 32 × 32 image. If the window size is 2 × 2 and the stride length is 2, how big will the resulting max-pooled image be?

ANSWER

The 2 × 2 window would need to move 16 times in each direction to span the 32 × 32 image, so the image would shrink by half: 16 × 16. Because it shrank by half in both dimensions, the image is one-fourth the size of the original image (½ × ½).

Place the following listing within the `Session` context.

Listing 9.10. Running the `maxpool` function to subsample convolved images

```
    k = 2
    maxpool = tf.nn.max_pool(conv_out,
                             ksize=[1, k, k, 1],
                             strides=[1, k, k, 1],
                             padding='SAME')

with tf.Session() as sess:
    maxpool_val = sess.run(maxpool)
    show_conv_results(maxpool_val, 'step3_maxpool.png')
    print(np.shape(maxpool_val))
```

As a result of running this code, the max-pooling function halves the image size and produces lower-resolution convolved outputs, as shown in figure 9.8.

Figure 9.8. After running `maxpool`, the convolved outputs are halved in size, making the algorithm computationally faster without losing too much information.

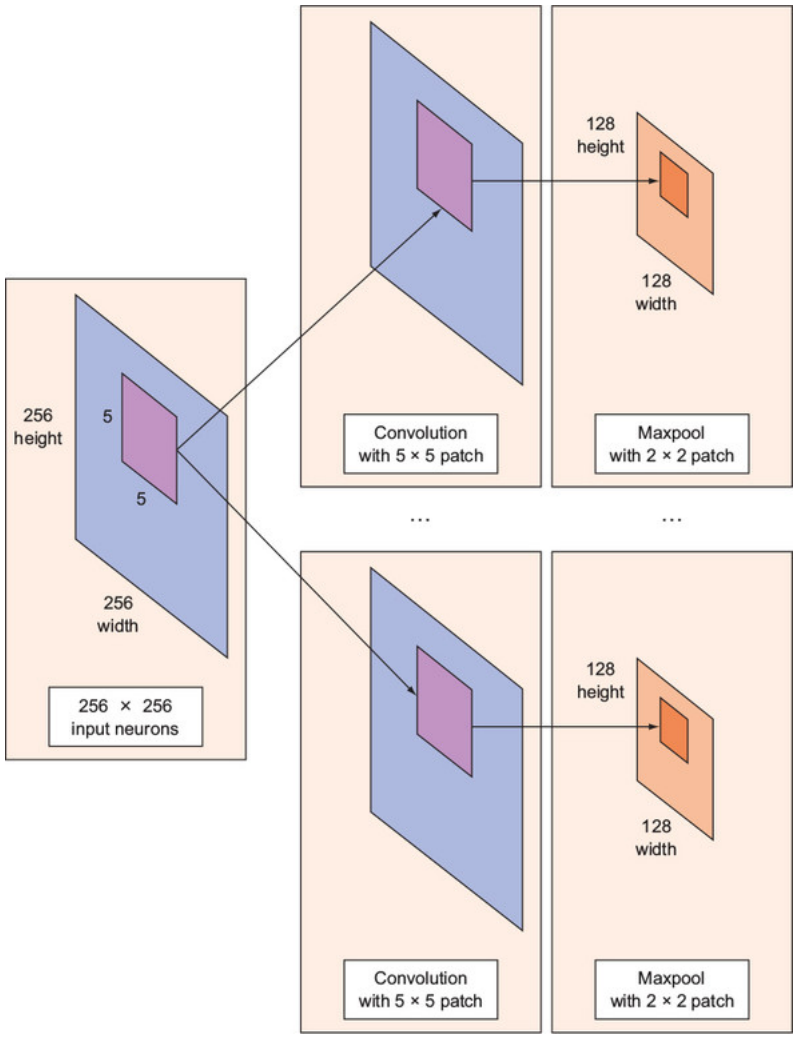
You have the tools necessary to implement the full convolutional neural network. In the next section, you'll finally train the image classifier.

9.4. IMPLEMENTING A CONVOLUTIONAL NEURAL NETWORK IN TENSORFLOW

A convolutional neural network has multiple layers of convolutions and max pooling. The convolution layer offers different perspectives on the image, while the max-pooling layer simplifies the computations by reducing the dimensionality without losing too much information.

Consider a full-size 256 × 256 image convolved by a 5 × 5 filter into 64 convolutions. As shown in figure 9.9, each convolution is subsampled by using max pooling to produce 64 smaller convolved images of size 128 × 128.

Figure 9.9. An input image is convolved by multiple 5 × 5 filters. The convolution layer includes an added bias term with an activation function, resulting in 5 × 5 + 5 = 30 parameters. Next, a max-pooling layer reduces the dimensionality of the data (which requires no extra parameters).



Now that you know how to make filters and use the convolution op, let's create a new source file. You'll start by defining all your variables. In listing 9.11, import all libraries, load the dataset, and, finally, define all variables.

Listing 9.11. Setting up CNN weights

```
import numpy as np
import matplotlib.pyplot as plt
import cifar_tools
import tensorflow as tf
names, data, labels = \
    cifar_tools.read_data('/home/binroot/res/cifar-10-batches-py')     1

x = tf.placeholder(tf.float32, [None, 24 * 24])                        2
y = tf.placeholder(tf.float32, [None, len(names)])                     2

W1 = tf.Variable(tf.random_normal([5, 5, 1, 64]))                      3
b1 = tf.Variable(tf.random_normal([64]))                               3

W2 = tf.Variable(tf.random_normal([5, 5, 64, 64]))                     4
b2 = tf.Variable(tf.random_normal([64]))                               4

W3 = tf.Variable(tf.random_normal([6*6*64, 1024]))                     5
b3 = tf.Variable(tf.random_normal([1024]))                             5

W_out = tf.Variable(tf.random_normal([1024, len(names)]))              6
b_out = tf.Variable(tf.random_normal([len(names)]))                    6
```

- ***1*** **Loads the dataset**

- ***2*** **Defines the input and output placeholders**
- ***3*** **Applies 64 convolutions of window size 5 × 5**
- ***4*** **Applies 64 more convolutions of window size 5 × 5**
- ***5*** **Introduces a fully connected layer**
- ***6*** **Defines the variables for a fully connected linear layer**

In the next listing, you define a helper function to perform a convolution, add a bias term, and then add an activation function. Together, these three steps form a convolution layer of the network.

Listing 9.12. Creating a convolution layer

```
def conv_layer(x, W, b):
    conv = tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
    conv_with_b = tf.nn.bias_add(conv, b)
    conv_out = tf.nn.relu(conv_with_b)
    return conv_out
```

The next listing shows how to define the max-pool layer by specifying the kernel and stride size.

Listing 9.13. Creating a max-pool layer

```
def maxpool_layer(conv, k=2):
    return tf.nn.max_pool(conv, ksize=[1, k, k, 1], strides=[1, k, k, 1],
     padding='SAME')
```

You can stack together the convolution and max-pool layers to define the convolutional neural network architecture. The following listing defines a possible CNN model. The last layer is typically just a fully connected network connected to each of the 10 output neurons.

Listing 9.14. The full CNN model

```
def model():
    x_reshaped = tf.reshape(x, shape=[-1, 24, 24, 1])

    conv_out1 = conv_layer(x_reshaped, W1, b1)
    maxpool_out1 = maxpool_layer(conv_out1)
    norm1 = tf.nn.lrn(maxpool_out1, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75

    conv_out2 = conv_layer(norm1, W2, b2)
    norm2 = tf.nn.lrn(conv_out2, 4, bias=1.0, alpha=0.001 / 9.0, beta=0.75)
    maxpool_out2 = maxpool_layer(norm2)

    maxpool_reshaped = tf.reshape(maxpool_out2, [-1,
     W3.get_shape().as_list()[0]])
    local = tf.add(tf.matmul(maxpool_reshaped, W3), b3)
    local_out = tf.nn.relu(local)

    out = tf.add(tf.matmul(local_out, W_out), b_out)
    return out
```

- ***1*** **Constructs the first layer of convolution and max pooling**

- **2 Constructs the second layer**
- **3 Constructs the concluding fully connected layers**

9.4.1. Measuring performance

With a neural network architecture designed, the next step is to define a cost function that you want to minimize. You'll use TensorFlow's `softmax_cross_entropy_with_ logits` function, which is best described by the official documentation (http://mng.bz/8mEk (http://mng.bz/8mEk)):

> *[The function `softmax_cross_entropy_with_logits`] measures the probability error in discrete classification tasks in which the classes are mutually exclusive (each entry is in exactly one class). For example, each CIFAR-10 image is labeled with one and only one label: an image can be a dog or a truck, but not both.*

Because an image can belong to 1 of 10 possible labels, you'll represent that choice as a 10-dimensional vector. All elements of this vector have a value of 0, except the element corresponding to the label will have a value of 1. This representation, as you've seen in earlier chapters, is called *one-hot encoding*.

As shown in listing 9.15, you'll calculate the cost via the cross-entropy loss function we mentioned in chapter 4. This returns the probability error for your classification. Note that this works only for simple classifications—those in which your classes are mutually exclusive (for example, a truck can't also be a dog). You can employ many types of optimizers, but in this example, let's stick with the AdamOptimizer, which is a simple and fast optimizer (described in detail at http://mng.bz/zW98 (http://mng.bz/zW98)). It may be worth playing around with the arguments to this in real-world applications, but it works well off the shelf.

Listing 9.15. Defining ops to measure the cost and accuracy

```
model_op = model()

cost = tf.reduce_mean(                                                    1
    tf.nn.softmax_cross_entropy_with_logits(logits=model_op, labels=y)
)

train_op = tf.train.AdamOptimizer(learning_rate=0.001).minimize(cost)     2

correct_pred = tf.equal(tf.argmax(model_op, 1), tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

- **1 Defines the classification loss function**
- **2 Defines the training op to minimize the loss function**

Finally, in the next section, you'll run the training op to minimize the cost of the neural network. Doing so multiple times throughout the dataset will learn the optimal weights (or parameters).

9.4.2. Training the classifier

In the following listing, you'll loop through the dataset of images in small batches to train the neural network. Over time, the weights will slowly converge to a local optimum to accurately predict the training images.

Listing 9.16. Training the neural network by using the CIFAR-10 dataset

```
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    onehot_labels = tf.one_hot(labels, len(names), on_value=1., off_value=0.,
     axis=-1)
    onehot_vals = sess.run(onehot_labels)
    batch_size = len(data) // 200
    print('batch size', batch_size)
    for j in range(0, 1000):                                        1
        print('EPOCH', j)
        for i in range(0, len(data), batch_size):                   2
            batch_data = data[i:i+batch_size, :]
            batch_onehot_vals = onehot_vals[i:i+batch_size, :]
            _, accuracy_val = sess.run([train_op, accuracy], feed_dict={x:
     batch_data, y: batch_onehot_vals})
            if i % 1000 == 0:
                print(i, accuracy_val)
        print('DONE WITH EPOCH')
```

- ***1*** **Loops through 1,000 epochs**
- ***2*** **Trains the network in batches**

That's it! You've successfully designed a convolutional neural network to classify images. Beware: it might take more than 10 minutes. If you're running this code on CPU, it might even take hours! Can you imagine discovering a bug in your code after a day of waiting? That's why deep-learning researchers use powerful computers and GPUs to speed up computations.

9.5. TIPS AND TRICKS TO IMPROVE PERFORMANCE

The CNN you developed in this chapter is a simple approach to solve the problem of image classification, but many techniques exist to improve performance after you finish your first working prototype:

- *Augmenting data*—From a single image, you can easily generate new training images. As a start, flip an image horizontally or vertically, and you can quadruple your dataset size. You may also adjust the brightness of the image or the hue to ensure that the neural network generalizes to other fluctuations. You may even want to add random noise to the image to make the classifier robust to small occlusions. Scaling the image up or down can also be helpful; having exactly the same-size items in your training images will almost guarantee overfitting!
- *Early stopping*—Keep track of the training and testing error while you train the neural network. At first, both errors should slowly dwindle, because the network is learning. But sometimes, the test error goes back up. This is a signal that the neural network has started overfitting on the training data and is unable to generalize to previously unseen input. You should stop the training the moment you witness this phenomenon.

- *Regularizing weights*—Another way to combat overfitting is by adding a regularization term to the cost function. You've already seen regularization in previous chapters, and the same concepts apply here.
- *Dropout*—TensorFlow comes with a handy `tf.nn.dropout` function, which can be applied to any layer of the network to reduce overfitting. It turns off a randomly selected number of neurons in that layer during training so that the network must be redundant and robust to inferring output.
- *Deeper architecture*—A deeper architecture results from adding more hidden layers to the neural network. If you have enough training data, it's been shown that adding more hidden layers improves performance.

Exercise 9.3

After the first iteration of this CNN architecture, try applying a couple of tips and tricks mentioned in this chapter.

ANSWER

Fine-tuning is, unfortunately, part of the process. You should begin by adjusting the hyperparameters and retraining the algorithm until you find a setting that works best.

9.6. APPLICATION OF CONVOLUTIONAL NEURAL NETWORKS

Convolutional neural networks blossom when the input contains sensor data from audio or images. Images, in particular, are of major interest in industry. For example, when you sign up for a social network, you usually upload a profile photo, not an audio recording of yourself saying "hello." It seems that humans are naturally more entertained by photos, so let's see how CNNs can be used to detect faces in images.

The overall CNN architecture can be as simple or as complicated as you desire. You should start simple, and gradually tune your model until satisfied. There's no absolutely correct path, because facial recognition isn't completely solved. Researchers are still publishing papers that one-up previous state-of-the-art solutions.

You should first obtain a dataset of images. One of the largest datasets of arbitrary images is ImageNet (http://image-net.org/ (http://image-net.org/)). Here, you can find negative examples for your binary classifier. To obtain positive examples of faces, you can find numerous datasets at the following sites that specialize in human faces:

- VGG Face Dataset: www.robots.ox.ac.uk/~vgg/data/vgg_face/ (http://www.robots.ox.ac.uk/~vgg/data/vgg_face/)
- FDDB: Face Detection Data Set and Benchmark: http://vis-www.cs.umass.edu/fddb/ (http://vis-www.cs.umass.edu/fddb/)
- Databases for Face Detection and Pose Estimation: http://mng.bz/25N6 (http://mng.bz/25N6)
- YouTube Faces Database: www.cs.tau.ac.il/~wolf/ytfaces/ (http://www.cs.tau.ac.il/~wolf/ytfaces/)

9.7. SUMMARY

- Convolutional neural networks make assumptions that capturing the local patterns of a signal are sufficient to characterize it, and thus reduce the number of parameters of a neural network.
- Cleaning data is vital to the performance of most machine-learning models. The hour you spend to write code that cleans data is nothing compared to the amount of time it can take for a neural network to learn that cleaning function by itself.

PREV
Chapter 8. Reinforcement learning

NEXT
Chapter 10. Recurrent neural networks