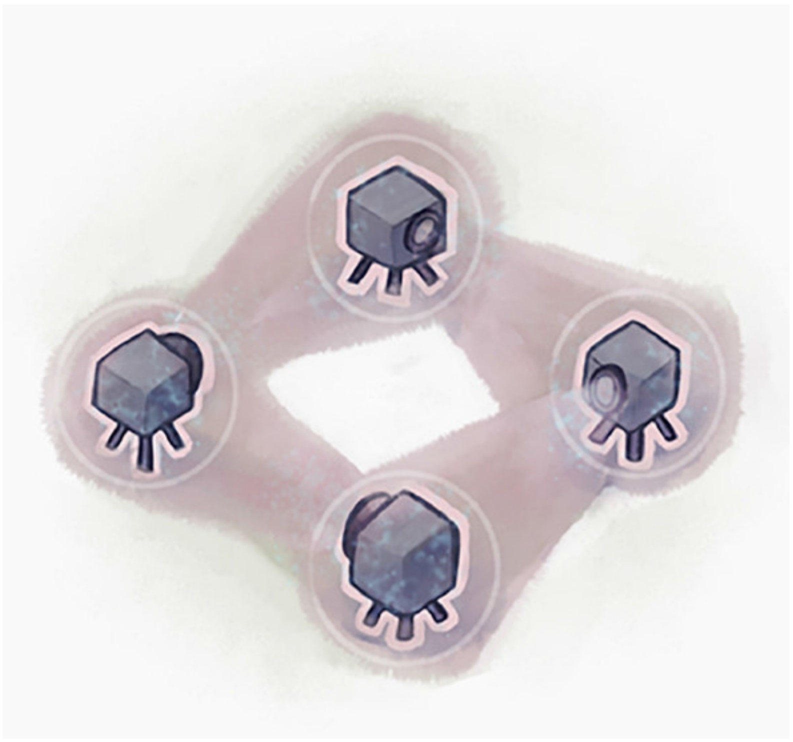
PREV
Part 3. The neural network paradigm

NEXT
Chapter 8. Reinforcement learning

Chapter 7. A peek into autoencoders



This chapter covers

- Getting to know neural networks
- Designing autoencoders
- Representing images by using an autoencoder

Have you ever heard a person humming a melody, and identified the song? It might be easy for you, but I'm comically tone-deaf when it comes to music. Humming, of itself, is an approximation of a song. An even better approximation could be singing. Include some instrumentals, and sometimes a cover of a song sounds indistinguishable from the original.

Instead of songs, in this chapter, you'll approximate functions. Functions are a general notion of relations between inputs and outputs. In machine learning, you typically want to find the function that relates inputs to outputs. Finding the best possible function fit is difficult, but approximating the function is much easier.

Conveniently, artificial neural networks are a model in machine learning that can approximate any function. As you've learned, your model is a function that gives the output you're looking for, given the inputs you have. In ML terms, given training data, you want to build a neural network model that best approximates the implicit function that might have generated the data—one that might not give you the exact answer but that's good enough to be useful.

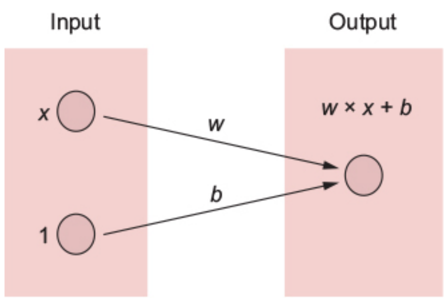
So far, you've generated models by explicitly designing a function, whether it be linear, polynomial, or something more complicated. Neural networks enable a little bit of leeway when it comes to picking out the right function, and consequently the right model. In theory, a neural network can model general-purpose types of transformation —where you don't need to know much at all about the function being modeled!

After section 7.1 introduces neural networks, you'll learn how to use autoencoders, which encode data into smaller, faster representations, in section 7.2.

7.1. NEURAL NETWORKS

If you've heard about neural networks, you've probably seen diagrams of nodes and edges connected in a complicated mesh. That visualization is mostly inspired by biology—specifically, neurons in the brain. As it turns out, it's also a convenient way to visualize functions, such as $f(x) = w \times x + b$, shown in figure 7.1.

Figure 7.1. A graphical representation of the linear equation $f(x) = w \times x + b$. The nodes are represented as circles, and edges are represented as arrows. The values on the edges are often called weights, and they act as a multiplication on the input. When two arrows lead to the same node, they act as a summation of the inputs.



As a reminder, a *linear model* is set of linear functions; for example, $f(x) = w \times x + b$, where (w, b) is the vector of parameters. The learning algorithm drifts around the values of w and b until it finds a combination that best matches the data. After the algorithm successfully converges, it'll find the best possible linear function to describe the data.

Linear is a good place to start, but the real world isn't always that pretty. And thus, we dive into the type of machine learning responsible for TensorFlow's inception; this chapter is your introduction to a type of model called an *artificial neural network*, which can approximate arbitrary functions (not just linear ones).

Exercise 7.1

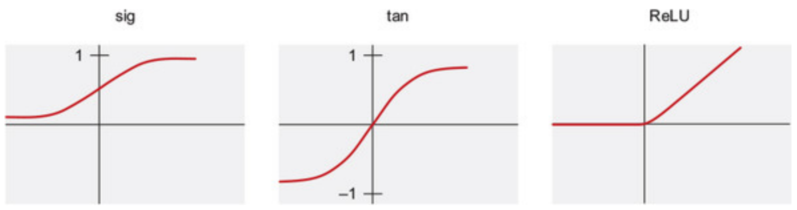
Is $f(x) = |x|$ a linear function?

Answer

No. It's two linear functions stitched together at zero, and that's not a single straight line.

To incorporate the concept of nonlinearity, it's effective to apply a nonlinear function, called the *activation function*, to each neuron's output. Three of the most commonly used activation functions are *sigmoid* (sig), *hyperbolic tangent* (tan), and a type of *ramp* function called a *Rectifying Linear Unit* (ReLU), plotted in figure 7.2.

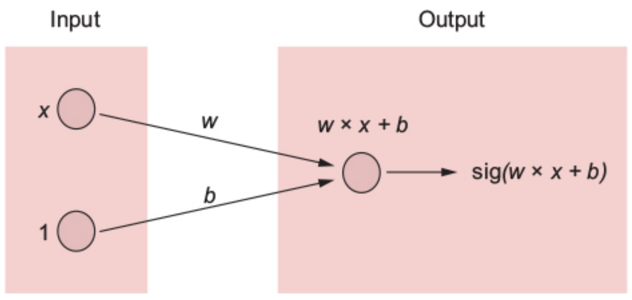
Figure 7.2. Use nonlinear functions such as sig, tan, and ReLU to introduce nonlinearity to your models.



You don't have to worry too much about which activation function is better under what circumstances. That's still an active research topic. Feel free to experiment with the three shown in figure 7.2. Usually, the best one is chosen by using cross-validation to determine which one gives the best model, given the dataset you're working with. Remember our confusion matrix in chapter 4? You test which model gives the fewest false-positives or false-negatives, or whatever other criteria best suits your needs.

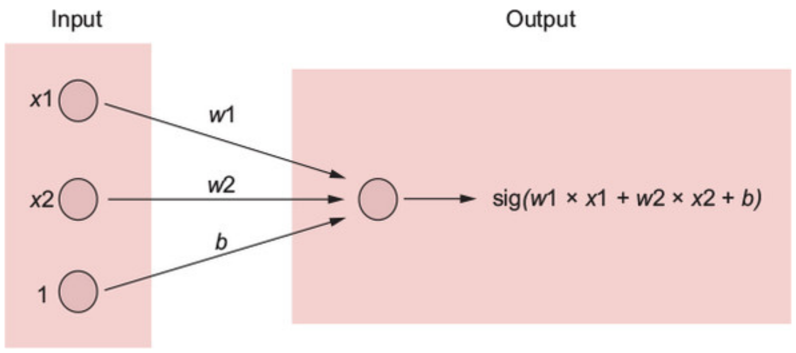
The sigmoid function isn't new to you. As you may recall, the logistic regression classifier in chapter 4 applied this sigmoid function to the linear function $w \times x + b$. The neural network model in figure 7.3 represents the function $f(x) = \text{sig}(w \times x + b)$. It's a one-input, one-output network, where w and b are the parameters of this model.

Figure 7.3. A nonlinear function, such as sigmoid, is applied to the output of a node.



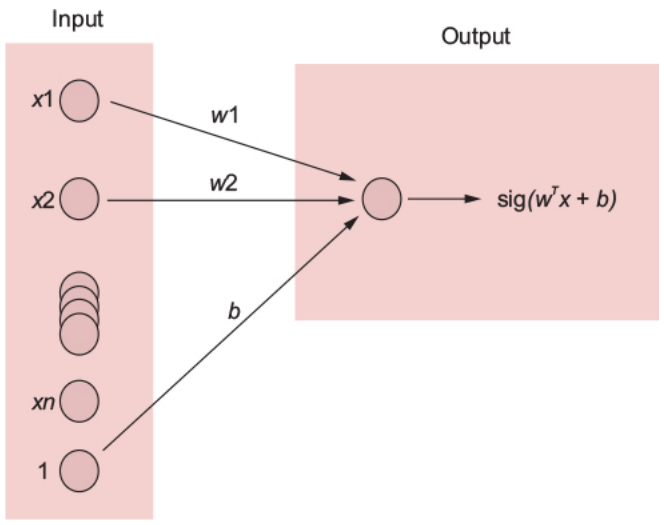
If you have two inputs ($x1$ and $x2$), you can modify your neural network to look like the one in figure 7.4. Given training data and a cost function, the parameters to be learned are $w1$, $w2$, and b. When trying to model data, having multiple inputs to a function is common. For example, image classification takes the entire image (pixel by pixel) as the input.

Figure 7.4. A two-input network will have three parameters (*w*1, *w*2, and *b*). Remember, multiple lines leading to the same node indicate summation.



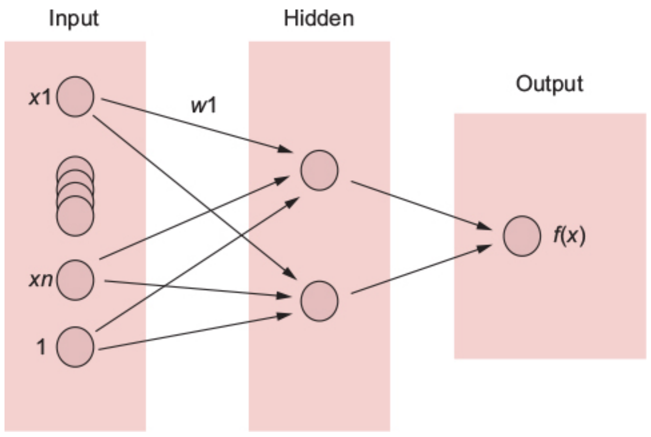
Naturally, you can generalize to an arbitrary number of inputs ($x1$, $x2$, ..., xn). The corresponding neural network represents the function $f(x1, ..., xn) = \text{sig}(wn \times xn + ... + w1 \times x1 + b)$, as shown in figure 7.5.

Figure 7.5. The input dimension can be arbitrarily long. For example, each pixel in a grayscale image can have a corresponding input *xi*. This neural network uses all inputs to generate a single output number, which you might use for regression or classification. The notation w^T means you're transposing *w*, which is an *n* × 1 vector, into a 1 × *n* vector. That way, you can properly multiply it with *x* (which has the dimensions *n* × 1). Such a matrix multiplication is also called a dot product, and it yields a scalar (one-dimensional) value.



So far, you've dealt with only an input layer and an output layer. Nothing's stopping you from arbitrarily adding neurons in between. Neurons that are used as neither input nor output are called *hidden neurons*. They're hidden from the input and output interfaces of the neural network, so no one can directly influence their values. A *hidden layer* is any collection of hidden neurons that don't connect to each other, as shown in figure 7.6. Adding more hidden layers greatly improves the expressive power of the network.

Figure 7.6. Nodes that don't interface to both the input and the output are called *hidden neurons*. A hidden layer is a collection of hidden units that aren't connected to each other.



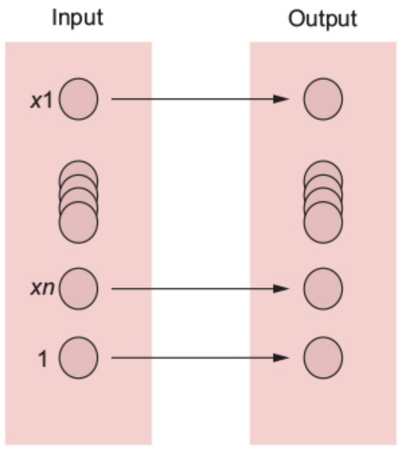
As long as the activation function is something nonlinear, a neural network with at least one hidden layer can approximate arbitrary functions. In linear models, no matter what parameters are learned, the function remains linear. The nonlinear neural network model with a hidden layer, on the other hand, is flexible enough to approximately represent any function! What a time to be alive!

TensorFlow comes with many helper functions to help you obtain the parameters of a neural network in an efficient way. You'll see how to invoke those tools in this chapter when you start using your first neural network architecture: an autoencoder.

7.2. AUTOENCODERS

An *autoencoder* is a type of neural network that tries to learn parameters that make the output as close to the input as possible. An obvious way to do so is to return the input directly, as shown in figure 7.7.

Figure 7.7. If you want to create a network where the input equals the output, you can connect the corresponding nodes and set each parameter's weight to 1.



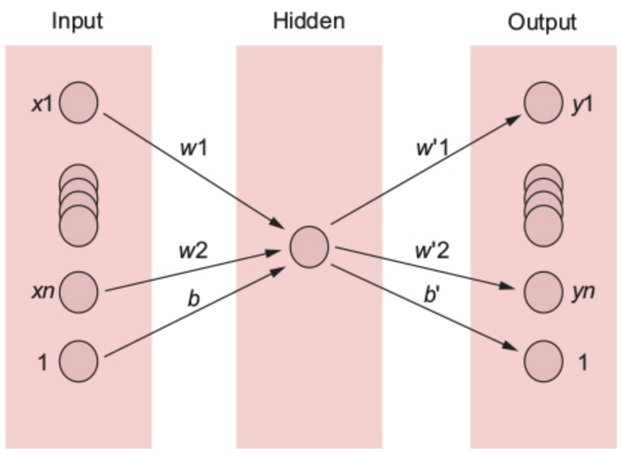
But an autoencoder is more interesting than that. It contains a small hidden layer! If that hidden layer has a smaller dimension than the input, the hidden layer is a compression of your data, called *encoding*.

Encoding data in the real world

A couple of audio formats are out there, but the most popular may be MP3 because of its relatively small file size. You may have already guessed that such efficient storage comes with a trade-off. The algorithm to generate an MP3 file takes original uncompressed audio and shrinks it into a much smaller file that sounds approximately the same to your ears. But it's lossy, meaning that you won't be able to completely recover the original uncompressed audio from the encoded version. Similarly, in this chapter, we want to reduce the dimensionality of the data to make it more workable, but not necessarily create a perfect reproduction.

The process of reconstructing the input from the hidden layer is called *decoding*. Figure 7.8 shows an exaggerated example of an autoencoder.

Figure 7.8. Here, you introduce a restriction to a network that tries to reconstruct its input. Data will pass through a narrow channel, as illustrated by the hidden layer. In this example, there's only one node in the hidden layer. This network is trying to encode (and decode) an *n*-dimensional input signal into just one dimension, which will likely be difficult in practice.



Encoding is a great way to reduce the dimensions of the input. For example, if you can represent a 256 × 256 image in just 100 hidden nodes, you've reduced each data item by a factor of thousands!

Exercise 7.2

Let *x* denote the input vector (*x*1, *x*2, ..., *xn*), and let *y* denote the output vector (*y*1, *y*2, ..., *yn*). Lastly, let *w* and *w*' denote the encoder and decoder weights, respectively. What's a possible cost function to train this neural network?

Answer

See the loss function in listing 7.3.

It makes sense to use an object-oriented programming style to implement an autoencoder. That way, you can later reuse the class in other applications without worrying about tightly coupled code. Creating your code as outlined in listing 7.1 helps build deeper architectures, such as a *stacked autoencoder*, which has been known to perform better empirically.

Tip

Generally, with neural networks, adding more hidden layers seems to improve performance if you have enough data to not overfit the model.

Listing 7.1. Python class schema

```
class Autoencoder:
    def __init__(self, input_dim, hidden_dim):    1

    def train(self, data):                        2

    def test(self, data):                         3
```

- ***1*** **Initializes variables**
- ***2*** **Trains on a dataset**
- ***3*** **Tests on some new data**

Open a new Python source file, and call it autoencoder.py. This file will define the autoencoder class that you'll use from a separate piece of code.

The constructor will set up all the TensorFlow variables, placeholders, optimizers, and operators. Anything that doesn't immediately need a session can go in the constructor. Because you're dealing with two sets of weights and biases (one for the encoding step and the other for the decoding step), you can use TensorFlow's name scopes to disambiguate a variable's name.

For instance, the following listing shows an example of defining a variable within a named scope. Now you can seamlessly save and restore this variable without worrying about name collisions.

Listing 7.2. Using name scopes

```
with tf.name_scope('encode'):
    weights = tf.Variable(tf.random_normal([input_dim, hidden_dim],
     dtype=tf.float32), name='weights')
    biases = tf.Variable(tf.zeros([hidden_dim]), name='biases')
```

Moving on, let's implement the constructor, as shown in the following listing.

Listing 7.3. Autoencoder class

```
import tensorflow as tf
import numpy as np

class Autoencoder:
    def __init__(self, input_dim, hidden_dim, epoch=250,
     learning_rate=0.001):
        self.epoch = epoch                                            1
        self.learning_rate = learning_rate                            2

        x = tf.placeholder(dtype=tf.float32, shape=[None, input_dim])       3

        with tf.name_scope('encode'):                                 4
            weights = tf.Variable(tf.random_normal([input_dim, hidden_dim],
     dtype=tf.float32), name='weights')
            biases = tf.Variable(tf.zeros([hidden_dim]), name='biases')
            encoded = tf.nn.tanh(tf.matmul(x, weights) + biases)
        with tf.name_scope('decode'):                                      5
            weights = tf.Variable(tf.random_normal([hidden_dim, input_dim],
dtype=tf.float32), name='weights')
            biases = tf.Variable(tf.zeros([input_dim]), name='biases')
            decoded = tf.matmul(encoded, weights) + biases

        self.x = x                                                         6
        self.encoded = encoded                                             6
        self.decoded = decoded                                             6

        self.loss = tf.sqrt(tf.reduce_mean(tf.square(tf.subtract(self.x,
self.decoded))))                                                           7
        self.train_op =
tf.train.RMSPropOptimizer(self.learning_rate).minimize(self.loss)         8
        self.saver = tf.train.Saver()                                      9
```

- *1* **Number of learning cycles**
- *2* **Hyperparameter of the optimizer**
- *3* **Defines the input layer dataset**
- *4* **Defines the weights and biases under a name scope so you can tell them apart from the decoder's weights and biases**
- *5* **The decoder's weights and biases are defined under this name scope.**
- *6* **These will be method variables.**
- *7* **Defines the reconstruction cost**
- *8* **Chooses the optimizer**
- *9* **Sets up a saver to save model parameters as they're being learned**

Now, in the next listing, you'll define a class method called `train` that will receive a dataset and learn parameters to minimize its loss.

Listing 7.4. Training the autoencoder

```
    def train(self, data):
        num_samples = len(data)
        with tf.Session() as sess:                                 1
            sess.run(tf.global_variables_initializer())            1
            for i in range(self.epoch):                            2
                for j in range(num_samples):                       3
                    l, _ = sess.run([self.loss, self.train_op],    3
                    feed_dict={self.x: [data[j]]})                 3
                if i % 10 == 0:                                    4
                    print('epoch {0}: loss = {1}'.format(i, l))    4
                    self.saver.save(sess, './model.ckpt')          5
            self.saver.save(sess, './model.ckpt')                  5
```

- *1* **Starts a TensorFlow session, and initializes all variables**
- *2* **Iterates through the number of cycles defined in the constructor**
- *3* **One sample at a time, trains the neural network on a data item**
- *4* **Prints the reconstruction error once every 10 cycles**
- *5* **Saves the learned parameters to file**

You now have enough code to design an algorithm that learns an autoencoder from arbitrary data. Before you start using this class, let's create one more method. As shown in the next listing, the `test` method will let you evaluate the autoencoder on new data.

Listing 7.5. Testing the model on data

```
    def test(self, data):
        with tf.Session() as sess:
            self.saver.restore(sess, './model.ckpt')                   1
            hidden, reconstructed = sess.run([self.encoded, self.decoded],
     feed_dict={self.x: data})                                         2
        print('input', data)
        print('compressed', hidden)
        print('reconstructed', reconstructed)
        return reconstructed
```

- ***1*** **Loads the learned parameters**
- ***2*** **Reconstructs the input**

Finally, create a new Python source file called main.py, and use your `Autoencoder` class, as shown in the following listing.

Listing 7.6. Using your `Autoencoder` class

```
from autoencoder import Autoencoder
from sklearn import datasets

hidden_dim = 1
data = datasets.load_iris().data
input_dim = len(data[0])
ae = Autoencoder(input_dim, hidden_dim)
ae.train(data)
ae.test([[8, 4, 6, 2]])
```

Running the `train` function will output debug info about how the loss decreases over the epochs. The `test` function shows info about the encoding and decoding process:

```
('input', [[8, 4, 6, 2]])
('compressed', array([[ 0.78238308]], dtype=float32))
('reconstructed', array([[ 6.87756062,  2.79838109,  6.25144577,
     2.23120356]], dtype=float32))
```

Notice that you're able to compress a four-dimensional vector into just one dimension and then decode it back into a four-dimensional vector with some loss in data.

7.3. BATCH TRAINING

Training a network one sample at a time is the safest bet if you're not pressured by time. But if your network training is taking longer than desired, one solution is to train it with multiple data inputs at a time, called *batch training*.

Typically, as the batch size increases, the algorithm speeds up but has a lower likelihood of successfully converging. It's a double-edged sword. Go wield it in the following listing. You’ll use that helper function later.

Listing 7.7. Batch helper function

```
def get_batch(X, size):
   a = np.random.choice(len(X), size, replace=False)
   return X[a]
```

To use batch learning, you’ll need to modify the `train` method from listing 7.4. The batch version is shown in the following listing. It inserts an additional inner loop for each batch of data. Typically, the number of batch iterations should be enough so that all data is covered in the same epoch.

Listing 7.8. Batch learning

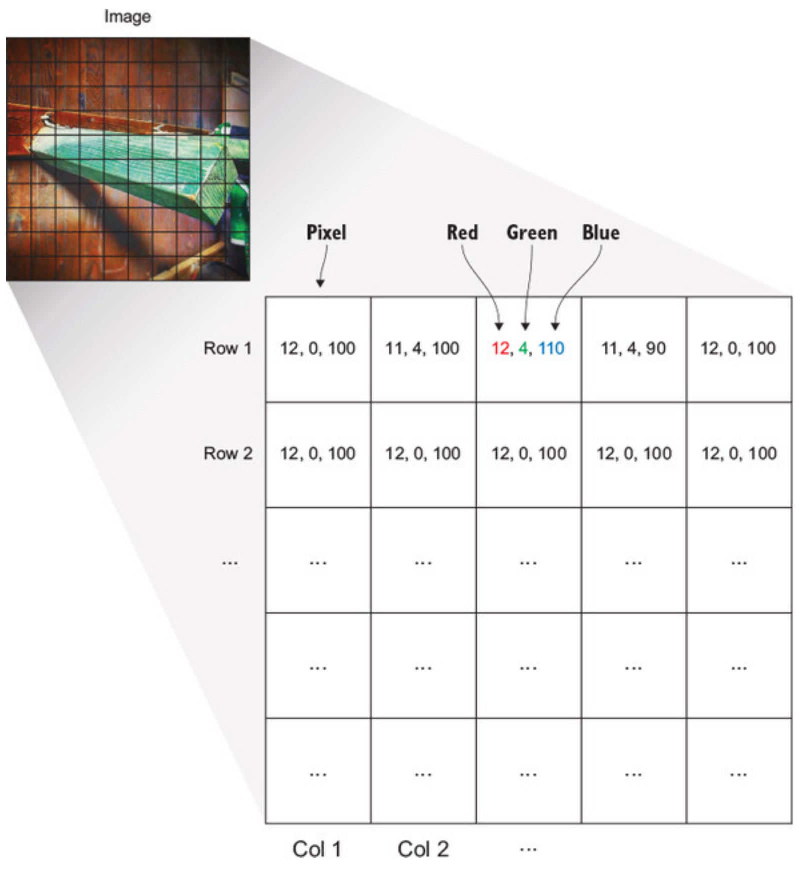
```
def train(self, data, batch_size=10):
        with tf.Session() as sess:
            sess.run(tf.global_variables_initializer())
            for i in range(self.epoch):
                for j in range(500):                                 1
                    batch_data = get_batch(data, self.batch_size)     2
                    l, _ = sess.run([self.loss, self.train_op],
     feed_dict={self.x: batch_data})
                if i % 10 == 0:
                    print('epoch {0}: loss = {1}'.format(i, l))
                    self.saver.save(sess, './model.ckpt')
            self.saver.save(sess, './model.ckpt')
```

- ***1*** **Loops through various batch selections**
- ***2*** **Runs the optimizer on a randomly selected batch**

7.4. WORKING WITH IMAGES

Most neural networks, like your autoencoder, accept only one-dimensional input. Pixels of an image, on the other hand, are indexed by both rows and columns. Moreover, if a pixel is in color, it has a value for its red, green, and blue concentration, as shown in figure 7.9.

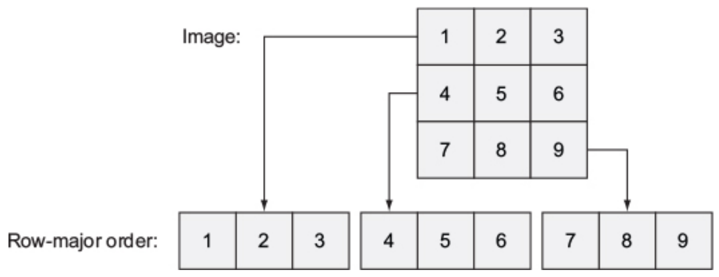
Figure 7.9. A colored image is composed of pixels, and each pixel contains values for red, green, and blue.



A convenient way to manage the higher dimensions of an image involves two steps:

1. Convert the image to grayscale: merge the values of red, green, and blue into the *pixel intensity*, which is a weighted average of the color values.
2. Rearrange the image into row-major order. *Row-major order* stores an array as a longer, single-dimension set; you put all the dimensions of an array on the end of the first dimension. This allows you to index the image by one number instead of two. If an image is 3 × 3 pixels in size, you rearrange it into the structure shown in figure 7.10.

Figure 7.10. An image can be represented in row-major order. That way, you can represent a two-dimensional structure as a onedimensional structure.



You can use images in TensorFlow in many ways. If you have pictures lying around on your hard drive, you can load them using SciPy, which comes with TensorFlow. The following listing shows you how to load an image in grayscale, resize it, and represent it in row-major order.

Listing 7.9. Loading images

```
from scipy.misc import imread, imresize

gray_image = imread(filepath, True)                 1
```

```
    small_gray_image = imresize(gray_image, 1. / 8.)    2
    x = small_gray_image.flatten()                      3
```

- ***1* Loads an image as grayscale**
- ***2* Resizes it to something smaller**
- ***3* Converts it to a one-dimensional structure**

Image processing is a lively field of research, so datasets are readily available for you to use, instead of using your own limited images. For instance, a dataset called CIFAR-10 contains 60,000 labeled images, each 32 × 32 in size.

Exercise 7.3

Can you name other online image datasets? Search online and look around for more!

Answer

Perhaps the most used in the deep-learning community is ImageNet (www.image-net.org (http://www.image-net.org)). A great list can also be found online at http://deeplearning.net/datasets (http://deeplearning.net/datasets).

Download the Python dataset from www.cs.toronto.edu/~kriz/cifar.html (http://www.cs.toronto.edu/~kriz/cifar.html). Place the extracted cifar-10-batches-py folder in your working directory. The following listing is provided from the CIFAR-10 web page; add the code to a new file called main_ imgs.py.

Listing 7.10. Reading from the extracted CIFAR-10 dataset

```
import pickle

def unpickle(file):                             1
    fo = open(file, 'rb')
    dict = pickle.load(fo, encoding='latin1')
    fo.close()
    return dict
```

- ***1* Reads the CIFAR-10 file, returning the loaded dictionary**

Let's read each of the dataset files by using the `unpickle` function you just created. The CIFA-10 dataset contains six files, each prefixed with data_batch_ and followed by a number. Each file contains information about the image data and corresponding label. The following listing shows how to loop through all the files and append the datasets to memory.

Listing 7.11. Reading all CIFAR-10 files to memory

```
import numpy as np

names = unpickle('./cifar-10-batches-py/batches.meta')['label_names']
data, labels = [], []
for i in range(1, 6):                                          1
    filename = './cifar-10-batches-py/data_batch_' + str(i)
```

```
        batch_data = unpickle(filename)                                 2
        if len(data) > 0:
            data = np.vstack((data, batch_data['data']))                3
            labels = np.hstack((labels, batch_data['labels']))          4
        else:
            data = batch_data['data']
            labels = batch_data['labels']
```

- *1* **Loops through the six files**
- *2* **Loads the file to obtain a Python dictionary**
- *3* **The rows of a data sample represent each sample, so you stack it vertically.**
- *4* **Labels are one-dimensional, so you stack them horizontally.**

Each image is represented as a series of red pixels, followed by green pixels, and then blue pixels. Listing 7.12 creates a helper function to convert the image into grayscale by averaging the red, green, and blue values.

Note

You can achieve more-realistic grayscale in other ways, but this approach of averaging the three values gets the job done. Human perception is more sensitive to green light, so in some other versions of grayscaling, green values might have a higher weight in the averaging.

Listing 7.12. Converting CIFAR-10 image to grayscale

```
def grayscale(a):
    return a.reshape(a.shape[0], 3, 32, 32).mean(1).reshape(a.shape[0], -1)

data = grayscale(data)
```

Lastly, let's collect all images of a certain class, such as `horse`. You'll run your autoencoder on all pictures of horses, as shown in the following listing.

Listing 7.13. Setting up the autoencoder

```
from autoencoder import Autoencoder

x = np.matrix(data)
y = np.array(labels)

horse_indices = np.where(y == 7)[0]

horse_x = x[horse_indices]

print(np.shape(horse_x))  # (5000, 3072)

input_dim = np.shape(horse_x)[1]
hidden_dim = 100
ae = Autoencoder(input_dim, hidden_dim)
ae.train(horse_x)
```

You can now encode images similar to your training dataset into just 100 numbers. This autoencoder model is one of the simplest, so clearly it'll be a lossy encoding. Beware: running this code may take up to 10 minutes. The output will trace loss values of every 10 epochs:

```
epoch 0: loss = 99.8635025024
epoch 10: loss = 35.3869667053
epoch 20: loss = 15.9411172867
epoch 30: loss = 7.66391372681
epoch 40: loss = 1.39575612545
epoch 50: loss = 0.00389165547676
epoch 60: loss = 0.00203850422986
epoch 70: loss = 0.00186171964742
epoch 80: loss = 0.00231492402963
epoch 90: loss = 0.00166488380637
epoch 100: loss = 0.00172081717756
epoch 110: loss = 0.0018497039564
epoch 120: loss = 0.00220602494664
epoch 130: loss = 0.00179589167237
epoch 140: loss = 0.00122790911701
epoch 150: loss = 0.0027100709267
epoch 160: loss = 0.00213225837797
epoch 170: loss = 0.00215123943053
epoch 180: loss = 0.00148373935372
epoch 190: loss = 0.00171591725666
```

See the book's website or GitHub repo for a full example of the output: https://www.manning.com/books/machine-learning-with-tensorflow or http://mng.bz/D0Na (http://mng.bz/D0Na).

7.5. APPLICATION OF AUTOENCODERS

This chapter introduced the most straightforward type of autoencoder, but other variants have been studied, each with their benefits and applications. Let's take a look at a few:

- A *stacked autoencoder* starts the same way a normal autoencoder does. It learns the encoding for an input into a smaller hidden layer by minimizing the reconstruction error. The hidden layer is now treated as the input to a new autoencoder that tries to encode the first layer of hidden neurons to an even smaller layer (the second layer of hidden neurons). This continues as desired. Often, the learned encoding weights are used as initial values for solving regression or classification problems in a deep neural network architecture.
- A *denoising autoencoder* receives a noised-up input instead of the original input, and it tries to "denoise" it. The cost function is no longer used to minimize the reconstruction error. Now, you're trying to minimize the error between the denoised image and the original image. The intuition is that our human minds can still comprehend a photograph even after scratches or markings on it. If a machine can also see through the noised input to recover the original data, maybe it has a better understanding of the data. Denoising models have been shown to better capture salient features of an image.
- A *variational autoencoder* can generate new natural images, given the hidden variables directly. Let's say you encode a picture of a man as a 100-dimensional

vector, and then a picture of a woman as another 100-dimensional vector. You can take the average of the two vectors, run it through the decoder, and produce a reasonable image that represents visually a person who's between a man and a woman. This generative power of the variational autoencoder is derived from a type of probabilistic models called *Bayesian networks*.

7.6. SUMMARY

- A neural network is useful when a linear model is ineffective for describing the dataset.
- Autoencoders are unsupervised learning algorithms that try to reproduce their inputs, and in doing so reveal interesting structure about the data.
- Images can easily be fed as input to a neural network by flattening and grayscaling.

PREV
Part 3. The neural network paradigm

NEXT
Chapter 8. Reinforcement learning