Chapter 9

Deep Learning

- Convolutional Neural Networks & Dropout
- Tools for Deep Learning
- Contrastive Divergence
- Gibb's Sampling

Deep learning is a relatively new advancement in neural network programming and represents a way to train deep neural networks. Essentially, any neural network with more than two layers is deep. The ability to create deep neural networks has existed since Pitts (1943) introduced the multilayer perceptron. However, we haven't been able to effectively train neural networks until Hinton (1984) became the first researcher to successfully train these complex neural networks.

9.1 Deep Learning Components

Deep learning is comprised of a number of different technologies, and this chapter is an overview of these technologies. Subsequent chapters will contain more information on these technologies. Deep learning typically includes the following features:

- Partially Labeled Data
- Rectified Linear Units (ReLU)
- Convolutional Neural Networks
- Dropout

The succeeding sections provide an overview of these technologies.

9.2 Partially Labeled Data

Most learning algorithms are either supervised or unsupervised. Supervised training data sets provide an expected outcome for each data item. Unsupervised training data sets do not provide an expected outcome, which is called a label. The problem is that most data sets are a mixture of labeled and unlabeled data items.

To understand the difference between labeled and unlabeled data, consider the following real-life example. When you were a child, you probably saw many vehicles as you grew up. Early in your life, you did not know if you were seeing a car, truck, or van. You simply knew that you were seeing some sort of vehicle. You can consider this exposure as the unsupervised part of your vehicle-learning journey. At that point, you learned commonalities of features among these vehicles.

Later in your learning journey, you were given labels. As you encountered different vehicles, an adult told you that you were looking at a car, truck, or van. The unsupervised training created your foundation, and you built upon that knowledge. As you can see, supervised and unsupervised learning

are very common in real life. In its own way, deep learning does well with a combination of unsupervised and supervised learning data.

Some deep learning architectures handle partially labeled data and initialize the weights by using the entire training set without the outcomes. You can independently train the individual layers without the labels. Because you can train the layers in parallel, this process is scalable. Once the unsupervised phase has initialized these weights, the supervised phase can tweak them.

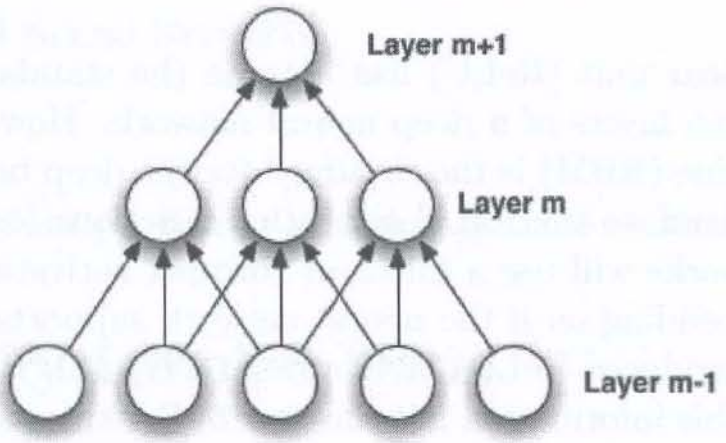
9.3 Rectified Linear Units

The Rectified linear unit (ReLU) has become the standard activation function for the hidden layers of a deep neural network. However, the restricted Boltzmann machine (RBM) is the standard for the deep belief neural network (DBNN). In addition to the ReLU activation functions for the hidden layers, deep neural networks will use a linear or softmax activation function for the output layer, depending on if the neural network supports regression or classification. We introduced ReLUs in Chapter 1, "Neural Network Basics," and expanded upon this information in "Chapter 6, Backpropagation Training."

9.4 Convolutional Neural Networks

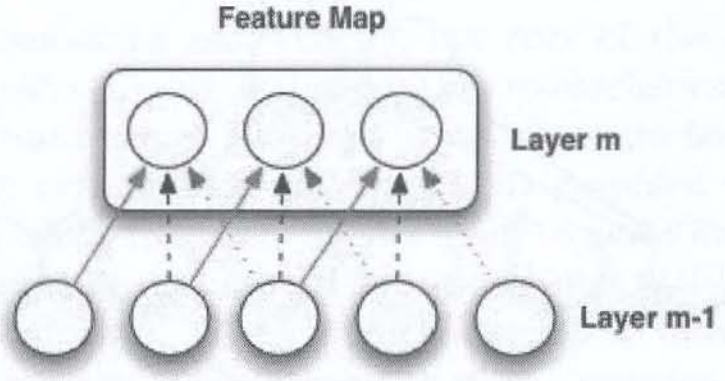
Convolution is an important technology that is often combined with deep learning. Hinton (2014) introduced convolution to allow image-recognition networks to function similarly to biological systems and achieve more accurate results. One approach is sparse connectivity in which we do not create every possible weight. Figure 9.1 shows sparse connectivity:

Figure 9.1: Sparse Connectivity



A regular feedforward neural network usually creates every possible weight connection between two layers. In deep learning terminology, we refer to these layers as dense layers. In addition to not representing every weight possible, convolutional neural networks will also share weights, as seen in Figure 9.2:

Figure 9.2: Shared Weights



As you can see in the above figure, the neurons share only three individual weights. The red (solid), black (dashed), and blue (dotted) lines indicate the individual weights. Sharing weights allows the program to store complex structures while maintaining memory and computation efficiency.

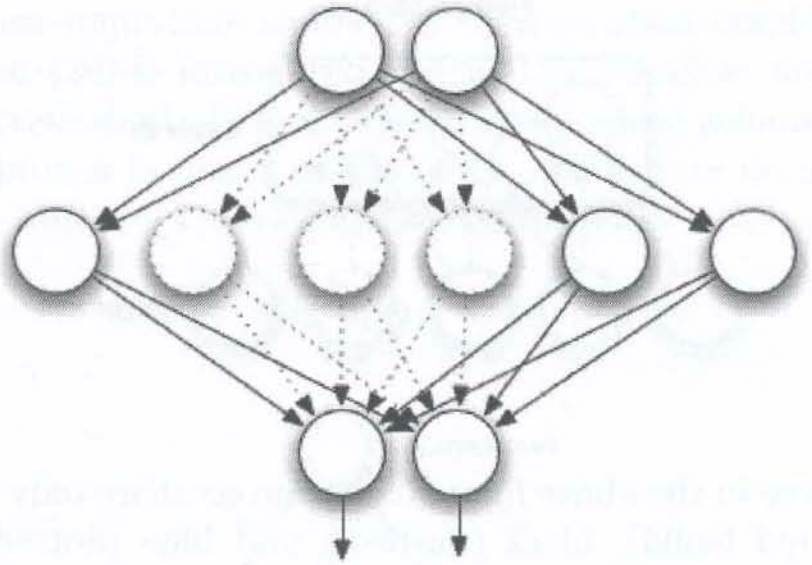
This section presented an overview of convolutional neural networks. Chapter 10, "Convolutional Neural Networks," is devoted entirely to this network type.

9.5 Neuron Dropout

Dropout is a regularization technique that holds many benefits for deep learning. Like most regularization techniques, dropout can prevent overfitting. You can also apply dropout to a neural network in a layer-by-layer fashion as you do in convolution. You must designate a single layer as a dropout layer. In fact, you can mix these dropout layers with regular layers and convolutional layers in the neural network. Never mix the dropout and convolutional layers within a single layer.

Hinton (2012) introduced dropout as a simple and effective regularization algorithm to reduce overfitting. Dropout works by removing certain neurons in the dropout layer. The act of dropping these neurons prevents other neurons from becoming overly dependent on the dropped neurons. The program removes these chosen neurons, along with all of their connections. Figure 9.3 illustrates this process:

Figure 9.3: Dropout Layer



From left to right, the above neural network contains an input layer, a dropout layer, and an output layer. The dropout layer has removed several of the neurons. The circles, made of dotted lines, indicate the neurons that the dropout algorithm removed. The dashed connector lines indicate the weights that the dropout algorithm removed when it eliminated the neurons.

Both dropout and other forms of regularization are extensive topics in the field of neural networks. Chapter 12, "Dropout and Regularization," covers regularization with particular focus on dropout. That chapter also contains an explanation on the L1 and L2 regularization algorithms. L1 and L2 discourage neural networks from the excessive use of large weights and the inclusion of certain irrelevant inputs. Essentially, a single neural network commonly uses dropout as well as other regularization algorithms.

9.6 GPU Training

Hinton (1987) introduced a very novel way to train the deep belief neural network (DBNN) efficiently. We examine this algorithm and DBNNs later in this chapter. As mentioned previously, deep neural networks have existed almost as long as the neural network. However, until Hinton's algorithm, no effective

way to train deep neural networks existed. The backpropagation algorithms are very slow, and the vanishing gradient problem hinders the training.

The graphics processing unit (GPU), the part of the computer that is responsible for graphics display, is the way that researchers solved the training problem of feedforward neural networks. Most of us are familiar with GPUs because of modern video games that utilize 3D graphics. Rendering these graphical images is mathematically intense, and, to perform these operations, early computers relied on the central processing unit (CPU). However, this approach was not effective. The graphics systems in modern video games require dedicated circuitry, which became the GPU, or video card. Essentially, modern GPUs are computers that function within your computer.

As researchers discovered, the processing power contained in a GPU can be harnessed for mathematically intense tasks, such as neural network training. We refer to this utilization of the GPU for general computing tasks, aside from computer graphics, as general-purpose use of the GPU (GPGPU). When applied to deep learning, the GPU performs extraordinarily well. Combining it with ReLU activation functions, regularization, and regular backpropagation can produce amazing results.

However, GPGPU can be difficult to use. Programs written for the GPU must employ a very low-level programming language called C99. This language is very similar to the regular C programming language. However, in many ways, the C99 required by the GPU is much more difficult than the regular C programming language. Furthermore, GPUs are good only at certain tasks–even those conducive to the GPU because optimizing the C99 code is challenging. GPUs must balance several classes of memory, registers, and synchronization of hundreds of processor cores. Additionally, GPU processing has two competing standards–CUDA and OpenCL. Two standards create more processes for the programmer to learn.

Fortunately, you do not need to learn GPU programming to exploit its processing power. Unless you are willing to devote a considerable amount of effort to learn the nuances of a complex and evolving field, we do not recommend that you learn to program the GPU because it is quite different from CPU programming. Good techniques that produce efficient, CPU-based programs will often produce horribly inefficient GPU programs. The reverse is also true. If you would like to use GPU, you should work with an off-the-shelf

package that supports it. If your needs do not fit into a deep learning package, you might consider using a linear algebra package, such as CUBLAS, which contains many highly optimized algorithms for the sorts of linear algebra that machine learning commonly requires.

The processing power of a highly optimized framework for deep learning and a fast GPU can be amazing. GPUs can achieve outstanding results based on sheer processing power. In 2010, the Swiss AI Lab IDSIA showed that, despite the vanishing gradient problem, the superior processing power of GPUs made backpropagation feasible for deep feedforward neural networks (Ciresan et al., 2010). The method outperformed all other machine learning techniques on the famous MNIST handwritten digit problem.

9.7 Tools for Deep Learning

One of the primary challenges of deep learning is the processing time to train a network. We often run training algorithms for many hours, or even days, seeking neural networks that fit well to the data sets. We use several frameworks for our research and predictive modeling. The examples in this book also utilize these frameworks, and we will present all of these algorithms in sufficient detail for you to create your own implementation. However, unless your goal is to conduct research to enhance deep learning itself, you are best served by working with an established framework. Most of these frameworks are tuned to train very fast.

We can divide the examples from this book into two groups. The first group shows you how to implement a neural network or to train an algorithm. However, most of the examples in this book are based on algorithms, and we examine the algorithm at its lowest level.

Application examples are the second type of example contained in this book. These higher-level examples show how to use neural network and deep learning algorithms. These examples will usually utilize one of the frameworks discussed in this section. In this way, the book strikes a balance between theory and real-world application.

9.7.1 H2O

H2O is a machine learning framework that supports a wide variety of programming languages. Though H2O is implemented in Java, it is designed as a web service. H2O can be used with R, Python, Scala, Java, and any language that can communicate with H2O's REST API.

Additionally, H2O can be used with Apache Spark for big data and big compute operations. The Sparking Water package allows H2O to run large models in memory across a grid of computers. For more information about H2O, refer to the following URL:

http://0xdata.com/product/deep-learning/

In addition to deep learning, H2O supports a variety of other machine learning models, such as logistic regression, decision trees, and gradient boosting.

9.7.2 Theano

Theano is a mathematical package for Python, similar to the widely used Python package, Numpy (J. Bergstra, O. Breuleux, F. Bastien, et al., J. Bergstra, O. Breuleux, F. Bastien, 2012). Like Numpy, Theano primarily targets mathematics. Though Theano does not directly implement deep neural networks, it provides all of the mathematical tools necessary for the programmer to create deep neural network applications. Theano also directly supports GPGPU. You can find the Theano package at the following URL:

http://deeplearning.net/software/theano/

The creators of Theano also wrote an extensive tutorial for deep learning, using Theano that can be found at the following URL:

http://deeplearning.net/

9.7.3 Lasagne and NoLearn

Because Theano does not directly support deep learning, several packages have been built upon Theano to make it easy for the programmer to implement deep learning. One pair of packages, often used together, is Lasagne and Nolearn. Nolearn is a package for Python that provides abstractions around several machine learning algorithms. In this way, Nolearn is similar to the popular framework, Scikit-Learn. While Scikit-Learn focuses widely on machine learning, Nolearn specializes on neural networks. One of the neural network packages supported by Nolearn is Lasagne, which provides deep learning and can be found at the following URL:

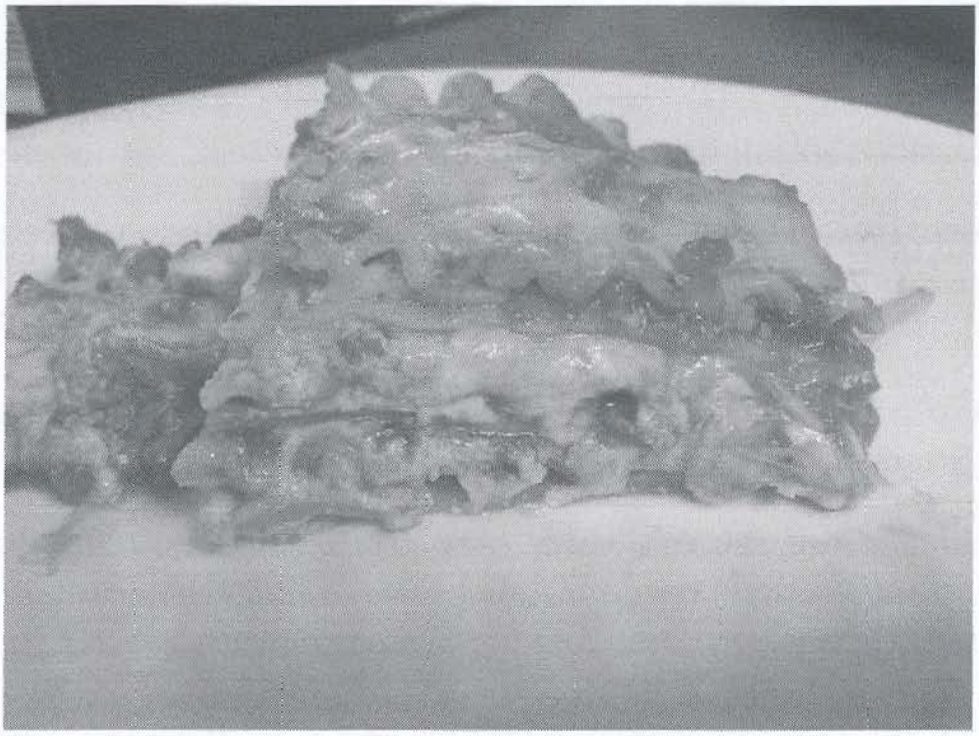
https://pypi.python.org/pypi/Lasagne/0.1dev

You can access the Nolearn package at the following URL:

https://github.com/dnouri/nolearn

The deep learning framework Lasange takes its name from the Italian food lasagna. The spellings "lasange" and "lasagna" are both considered valid spellings of the Italian food. In the Italian language, "lasange" is singular, and "lasagna" is the plural form. Regardless of the spelling used, lasagna is a good name for a deep learning framework. Figure 9.4 shows that, like a deep neural network, lasagna is made up of many layers:

Figure 9.4: Lasagna Layers



9.7.4 ConvNetJS

Deep learning support has also been created for Javascript. The ConvNetJS package implements many deep learning algorithms, particularly in the area of convolutional neural networks. ConvNetJS primarily targets the creation of deep learning examples on websites. We used ConvNetJS to provide many of the deep learning JavaScript examples on this book's website:

http://cs.stanford.edu/people/karpathy/convnetjs/

9.8 Deep Belief Neural Networks

The deep belief neural network (DBNN) was one of the first applications of deep learning. A DBNN is simply a regular belief network with many layers. Belief networks, introduced by Neil in 1992 are different from regular feed-forward neural networks. Hinton (2007) describes DBNNs as "probabilistic generative models that are composed of multiple layers of stochastic, latent

variables." Because this technical description is complicated, we will define some terms.

- **Probabilistic** - DBNNs are used to classify, and their output is the probability that an input belongs to each class.
- **Generative** - DBNNs can produce plausible, randomly created values for the input values. Some DBNN literatures refer to this trait as dreaming.
- **Multiple layers** - Like a neural network, DBNNs can be made of multiple layers.
- **Stochastic, latent variables** - DBNNs are made up of Boltzmann machines that produce random (stochastic) values that cannot be directly observed (latent).

The primary differences between a DBNN and a feedforward neural network (FFNN) are summarized as follows:

- Input to a DBNN must be binary; input to a FFNN is a decimal number.
- The output from a DBNN is the class to which the input belongs; the output from a FFNN can be a class (classification) or a numeric prediction (regression).
- DBNNs can generate plausible input based on a given outcome. FFNNs cannot perform like the DBNNs.

These are important differences. The first bullet item is one of the most limiting factors of DBNNs. The fact that a DBNN can accept only binary input often severely limits the type of problem that it can tackle. You also need to note that a DBNN can be used only for classification and not for regression. In other words, a DBNN could classify stocks into categories such as buy, hold, or sell; however, it could not provide a numeric prediction about the stock, such as the amount that may be attained over the next 30 days. If you need any of these features, you should consider a regular deep feedforward network.

Compared to feedforward neural networks, DBNNs may initially seem somewhat restrictive. However, they do have the ability to generate plausible input cases based on a given output. One of the earliest DBNN experiments was to have a DBNN classify ten digits, using handwritten samples. These digits were from the classic MNIST handwritten digits data set that was included in this book's introduction. Once the DBNN is trained on the MNIST digits, it can produce new representations of each digit, as seen in Figure 9.5:

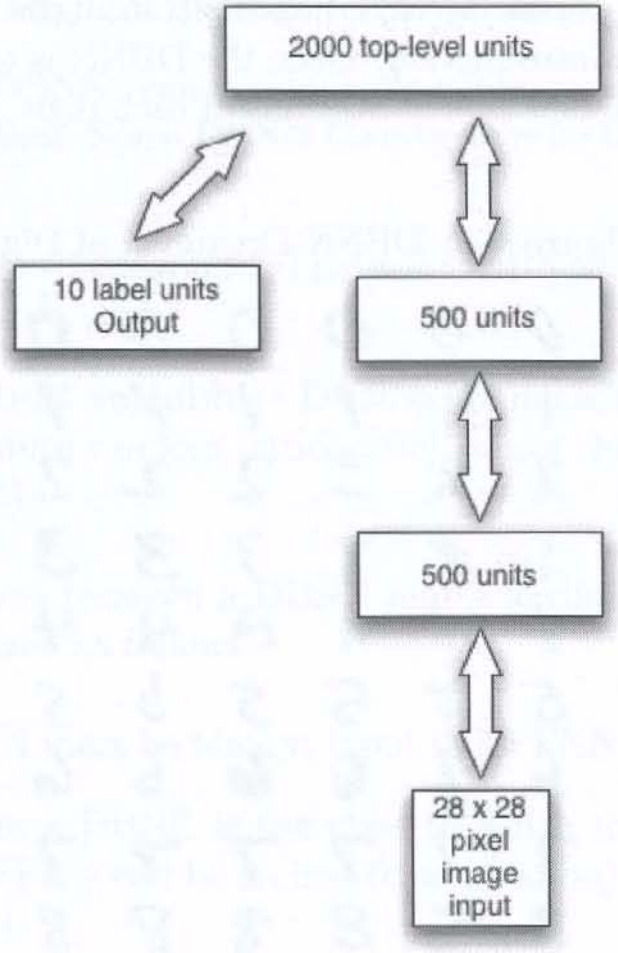
Figure 9.5: DBNN Dreaming of Digits

The above digits were taken from Hinton's (2006) deep learning paper. The first row shows a variety of different zeros that the DBNN generated from its training data.

The restricted Boltzmann machine (RBM) is the center of the DBNN. Input provided to the DBNN passes through a series of stacked RBMs that make up the layers of the network. Creating additional RBM layers causes deeper DBNNs. Though RBMs are unsupervised, the desire is for the resulting DBNN to be supervised. To accomplish the supervision, a final logistic regression layer is added to distinguish one class from another. In the case of Hinton's

experiment, shown in Figure 9.6, the classes are the ten digits:

Figure 9.6: Deep Belief Neural Network (DBNN)



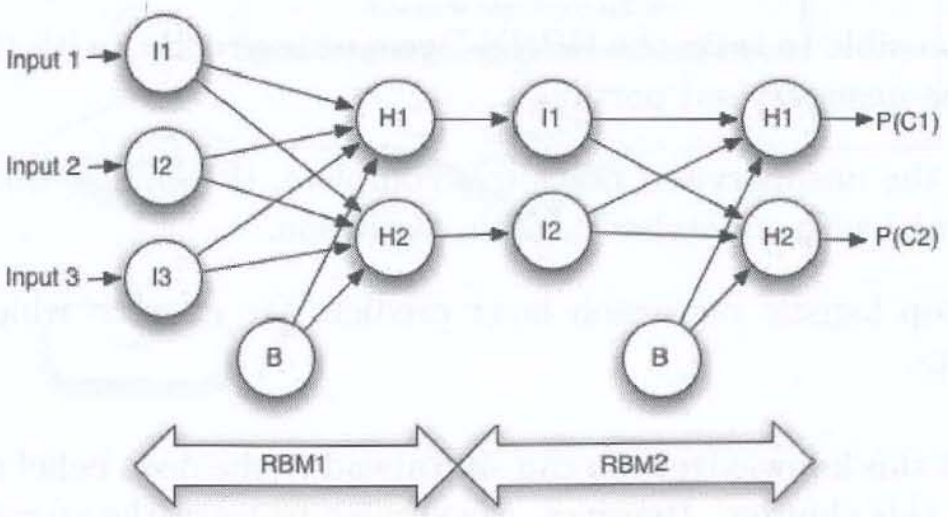
The above diagram shows a DBNN that uses the same hyper-parameters as Hinton's experiment. Hyper-parameters specify the architecture of a neural network, such as the number of layers, hidden neuron counts, and other settings. Each of the digit images presented to the DBNN is 28x28 pixels, or vectors of 784 pixels. The digits are monochrome (black & white) so these 784 pixels are single bits and are thus compatible with the DBNN's requirement that all input be binary. The above network has three layers of stacked RBMs, containing 500 neurons, a second 500-neuron layer, and 2,000 neurons, respectively.

The following sections discuss a number of algorithms used to implement DBNNs.

9.8.1 Restricted Boltzmann Machines

Because Chapter 3, "Hopfield & Boltzmann Machines," includes a discussion of Boltzmann machines, we will not repeat this material here. This chapter deals with the restricted version of the Boltzmann machine and stacking these RBMs to achieve depth. Figure 2.10, from Chapter 3, shows an RBM. The primary difference with an RBM is that the visible (input) neurons and the hidden (output) neurons have the only connections. In the case of a stacked RBM, the hidden units become the output to the next layer. Figure 9.7 shows how two Boltzmann machines are stacked:

Figure 9.7: Stacked RBMs



We can calculate the output from an RBM exactly as shown in Chapter 3, "Hopfield & Boltzmann Machines," in Equation 3.6. The only difference is now we have two Boltzmann machines stacked. The first Boltzmann machine receives three inputs passed to its visible units. The hidden units pass their output directly to the two inputs (visible units) of the second RBM. Notice that there are no weights between the two RBMs, and the output from the H1 and H2 units in RBM1 pass directly to I1 and I2 from RBM2.

9.8.2 Training a DBNN

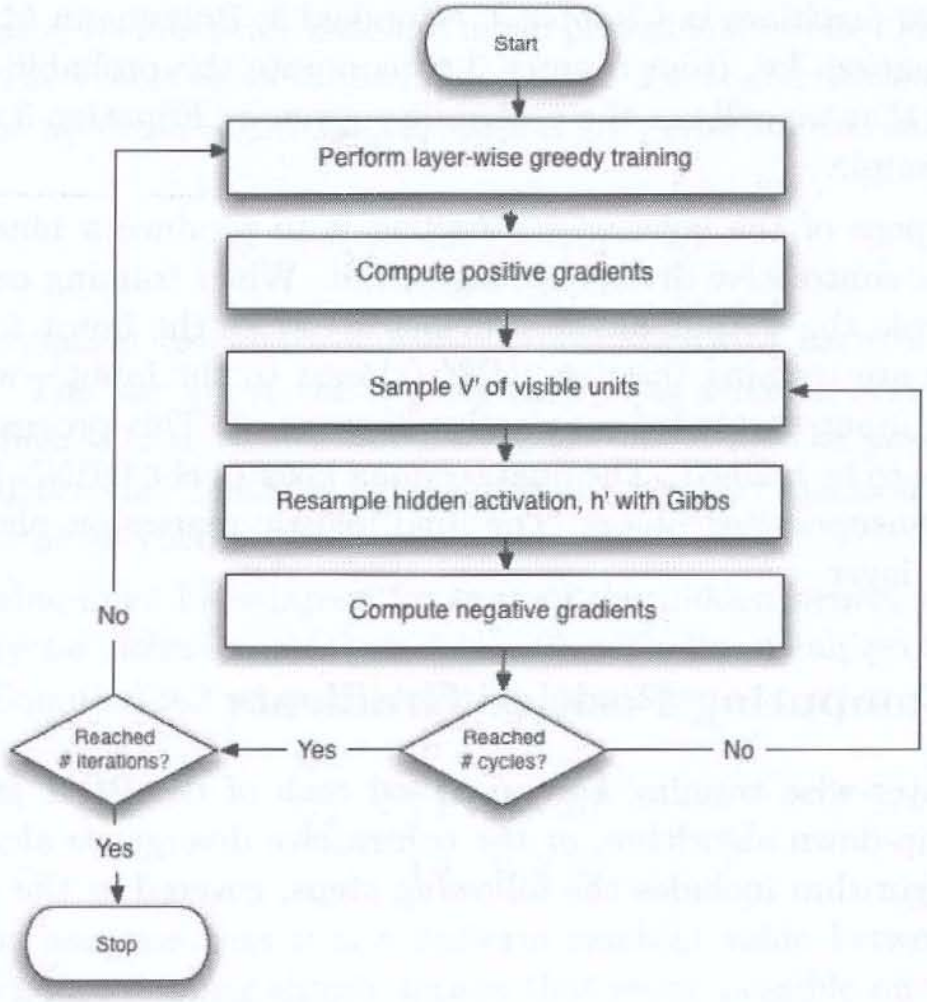
The process of training a DBNN requires a number of steps. Although the mathematics behind this process can become somewhat complex, you don't need to understand every detail for training DBNNs in order to use them. You just need to know the following key points:

- DBNNs undergo supervised and unsupervised training.
- During the unsupervised portion, the DBNN uses training data without their labels, which allows DBNNs to have a mix of supervised and unsupervised data.
- During the supervised portion, only training data with labels are used.
- Each DBNN layer is trained independently during the unsupervised portion.
- It is possible to train the DBNN layers concurrently (with threads) during the unsupervised portion.
- After the unsupervised portion is complete, the output from the layers is refined with supervised logistic regression.
- The top logistic regression layer predicts the class to which the input belongs.

Armed with this knowledge, you can skip ahead to the deep belief classification example in this chapter. However, if you wish to learn the specific details of DBNN training, read on.

Figure 9.8 provides a summary of the steps of DBNN training:

Figure 9.8: DBNN Training



9.8.3 Layer-Wise Sampling

The first step when performing unsupervised training on an individual layer is to calculate all values of the DBNN up to that layer. You will do this calculation for every training set, and the DBNN will provide you with sampled values at the layer that you are currently training. Sampled refers to the fact that the neural network randomly chooses a true/false value based on a probability.

You need to understand that sampling uses random numbers to provide

you with your results. Because of this randomness, you will not always get the same result. If the DBNN determines that a hidden neuron's probability of true is 0.75, then you will get a value of true 75% of the time. Layer-wise sampling is very similar to the method that we used to calculate the output of Boltzmann machines in Chapter 3, "Hopfield & Boltzmann Machines." We will use Equation 3.6, from chapter 3 to compute the probability. The only difference is that we will use the probability given by Equation 3.6 to generate a random sample.

The purpose of the layer-wise sampling is to produce a binary vector to feed into the contrastive divergence algorithm. When training each RBM, we always provide the output of the previous RBM as the input to the current RBM. If we are training the first RBM (closest to the input), we simply use the training input vector for contrastive divergence. This process allows each of the RBMs to be trained. The final softmax layer of the DBNN is not trained during the unsupervised phase. The final logistic regression phase will train the softmax layer.

9.8.4 Computing Positive Gradients

Once the layer-wise training has processed each of the RBM layers, we can utilize the up-down algorithm, or the contrastive divergence algorithm. This complete algorithm includes the following steps, covered in the next sections of this book:

- Computing Positive Gradients
- Gibbs Sampling
- Update Weights and Biases
- Supervised Backpropagation

Like many of the gradient-descent-based algorithms presented in Chapter 6, "Backpropagation Training," the contrastive divergence algorithm is also based on gradient descent. It uses the derivative of a function to find the inputs to the function that produces the lowest output for that function. Several

different gradients are estimated during contrastive divergence. We can use these estimates instead of actual calculations because the real gradients are too complex to calculate. For machine learning, an estimate is often good enough.

Additionally, we must calculate the mean probability of the hidden units by propagating the visible units to the hidden ones. This computation is the "up" portion of the up-down algorithm. Equation 9.1 performs this calculation:

$$\bar{h}_i^+ = sigmoid(\sum_j w_j v_j + b_i) \tag{9.1}$$

The above equation calculates the mean probability of each of the hidden neurons (h). The bar above the h designates it as a mean, and the positive subscript indicates that we are calculating the mean for the positive (or up) part of the algorithm. The bias is added to the sigmoid function value of the weighted sum of all visible units.

Next a value must be sampled for each of the hidden neurons. This value will randomly be either true (1) or false (0) with the mean probability just calculated. Equation 9.2 accomplishes this sampling:

$$h_i^+ = \begin{cases} 1 & r < \bar{h}_i^+ \\ 0 & r \geq \bar{h}_i^+ \end{cases} \tag{9.2}$$

This equation assumes that r is a uniform random value between 0 and 1. A uniform random number simply means that every possible number in that range has an equal probability of being chosen.

9.8.5 Gibbs Sampling

The calculation of the negative gradients is the "down" phase of the up-down algorithm. To accomplish this calculation, the algorithm uses Gibbs sapling to estimate the mean of the negative gradients. Geman and Geman (1984) introduced Gibbs sampling and named it after the physicist Josiah Willard Gibbs. The technique is accomplished by looping through k iterations that improve the quality of the estimate. Each iteration performs two steps:

- Sample visible neurons give hidden neurons.
- Sample hidden neurons give visible neurons.

For the first iteration of Gibbs sampling, we start with the positive hidden neuron samples obtained from the last section. We will sample visible neuron average probabilities from these (first bullet above). Next, we will use these visible hidden neurons to sample hidden neurons (second bullet above). These new hidden probabilities are the negative gradients. For the next cycle, we will use the negative gradients in place of the positive ones. This continues for each iteration and produces better negative gradients. Equation 9.3 accomplishes sampling of the visible neurons (first bullet):

$$\bar{v}_i^- = sigmoid(\sum_j w_j h_j + b_i) \tag{9.3}$$

This equation is essentially the reverse of Equation 9.1. Here, we determine the average visible mean using the hidden values. Again, just like we did for the positive gradients, we sample a negative probability using Equation 9.4:

$$v_i^- = \begin{cases} 1 & r < \bar{v}_i^- \\ 0 & r \geq \bar{v}_i^- \end{cases} \tag{9.4}$$

The above equation assumes that r is a uniform random number between 0 and 1.

The above two equations are only half of the Gibbs sampling step. These equations accomplished the first bullet point above because they sample visible neurons, given hidden neurons. Next, we must accomplish the second bullet

point. We must sample hidden neurons, given visible neurons. This process is very similar to the above section, "Computing Positive Gradients." This time, however, we are calculating the negative gradients.

The visible unit samples just calculated can obtain hidden means, as shown in Equation 9.5:

$$\bar{h}_i^- = sigmoid(\sum_j w_j v_j + b_i) \tag{9.5}$$

Just as before, mean probability can sample an actual value. In this case, we use the hidden mean to sample a hidden value, as demonstrated by Equation 9.6:

$$h_i^- = \begin{cases} 1 & r < \bar{h}_i^- \\ 0 & r \geq \bar{h}_i^- \end{cases} \tag{9.6}$$

The Gibbs sampling process continues. The negative hidden samples can process each iteration. Once this calculation is complete, you have the following six vectors:

- Positive mean probabilities of the hidden neurons
- Positive sampled values of the hidden neurons
- Negative mean probabilities of visible neurons
- Negative sampled values of visible neurons
- Negative mean probabilities of hidden neurons
- Negative sampled values of hidden neurons

These values will update the neural network's weights and biases.

9.8.6 Update Weights & Biases

The purpose of any neural network training is to update the weights and biases. This adjustment is what allows the neural network to learn to perform

the intended task. This is the final step for the unsupervised portion of the DBNN training process. In this step, the weights and biases of a single layer (Boltzmann machine) will be updated. As previously mentioned, the Boltzmann layers are trained independently.

The weights and biases are updated independently. Equation 9.7 shows how to update a weight:

$$\Delta_{ij} = \frac{\epsilon(\bar{h}_i^+ x_j - \bar{h}_i^- v_j^-)}{|x|} \tag{9.7}$$

The learning rate (ϵ, epsilon) specifies how much of a calculated change should be applied. High learning rates will learn quicker, but they might skip over an optimal set of weights. Lower learning rates learn more slowly, but they might have a higher quality result. The value x represents the current training set element. Because x is a vector (array), the x enclosed in two bars represents the length of x. The above equation also uses the positive mean hidden probabilities, the negative mean hidden probabilities, and the negative sampled values.

Equation 9.8 calculates the biases in a similar fashion:

$$\Delta b_i = \frac{\epsilon(h_i^+ - \bar{h}_i^-)}{||x||} \tag{9.8}$$

The above equation uses the sampled hidden value from the positive phase and the mean hidden value from the negative phase, as well as the input vector. Once the weights and biases have been updated, the unsupervised portion of the training is done.

9.8.7 DBNN Backpropagation

Up to this point, the DBNN training has focused on unsupervised training. The DBNN used only the training set inputs (x values). Even if the data set provided an expected output (y values), the unsupervised training didn't use it. Now the DBNN is trained with the expected outputs. We use only data set items that contain an expected output during this last phase. This step allows the program to use DBNN networks with data sets where each item does not

necessarily have an expected output. We refer to the data as partially labeled data sets.

The final layer of the DBNN is simply a neuron for each class. These neurons have weights to the output of the previous RBM layer. These output neurons all use sigmoid activation functions and a softmax layer. The softmax layer ensures that the output for each of the classes sum to 1.

Regular backpropagation trains this final layer. The final layer is essentially the output layer of a feedforward neural network that receives its input from the top RBM. Because Chapter 6, "Backpropagation Training," contains a discussion of backpropagation, we will not repeat the information here. The main idea of a DBNN is that the hierarchy allows each layer to interpret the data for the next layer. This hierarchy allows the learning to spread across the layers. The higher layers learn more abstract notions while the lower layers form from the input data. In practice, DBNNs can process much more complex of patterns than a regular backpropagation-trained feedforward neural network.

9.8.8 Deep Belief Application

This chapter presents a simple example of the DBNN. This example simply accepts a series of input patterns as well as the classes to which these input patterns belong. The input patterns are shown below:

```
[[1, 1, 1, 1, 0, 0, 0, 0],
 [1, 1, 0, 1, 0, 0, 0, 0],
 [1, 1, 1, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 1, 1, 1],
 [0, 0, 0, 0, 1, 1, 0, 1],
 [0, 0, 0, 0, 1, 1, 1, 0]]
```

We provide the expected output from each of these training set elements. This information specifies the class to which each of the above elements belongs and is shown below:

```
[[1, 0],
 [1, 0],
 [1, 0],
 [0, 1],
```

```
 [0, 1],
 [0, 1]]
```

The program provided in the book's example creates a DBNN with the following configuration:

- Input Layer Size: 8
- Hidden Layer #1: 2
- Hidden Layer #2: 3
- Output Layer Size: 2

First, we train each of the hidden layers. Finally, we perform logistic regression on the output layer. The output from this program is shown here:

```
Training Hidden Layer #0
Training Hidden Layer #1
Iteration: 1, Supervised training: error = 0.2478464544753616
Iteration: 2, Supervised training: error = 0.23501688281192523
Iteration: 3, Supervised training: error = 0.2228704042138232
...
Iteration: 287, Supervised training: error = 0.0010805100324100002
Iteration: 288, Supervised training: error = 7.821742124428358E-4
[0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0] -> [0.9649828726012807,
   0.035017127398719 41]
[1.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0] -> [0.9649830045627616,
   0.035016995437238435]
[0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0] -> [0.03413161595489315,
   0.9658683840451069]
[0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 1.0] -> [0.03413137188719462,
   0.9658686281128055]
```

As you can see, the program first trained the hidden layers and then went through 288 iterations of regression. The error level dropped considerably during these iterations. Finally, the sample data quizzed the network. The network responded with the probability of the input sample being in each of the two classes that we specified above.

For example, the network reported the following element:

```
[0.0, 1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0]
```

This element had a 96% probability of being in class 1, but it had only a 4% probability of being in class 2. The two probabilities reported for each item must sum to 100%.

9.9 Chapter Summary

This chapter provided a high-level overview of many of the components of deep learning. A deep neural network is any network that contains more than two hidden layers. Although deep networks have existed for as long as multilayer neural networks, they have lacked good training methods until recently. New training techniques, activation functions, and regularization are making deep neural networks feasible.

Overfitting is a common problem for many areas of machine learning; neural networks are no exception. Regularization can prevent overfitting. Most forms of regularization involve modifying the weights of a neural network as the training occurs. Dropout is a very common regularization technique for deep neural networks that removes neurons as training progresses. This technique prevents the network from becoming overly dependent on any one neuron.

We ended the chapter with the deep belief neural network (DBNN), which classifies data that might be partially labeled. First, both labeled and unlabeled data can initialize the weights of the neural network with unsupervised training. Using these weights, a logistic regression layer can fine-tune the network to the labeled data.

We also discussed the convolutional neural networks (CNN) in this chapter. This type of neural network causes the weights to be shared between the various neurons in the network. This neural network allows the CNN to deal with the types of overlapping features that are very common in computer vision. We provided only a general overview of CNN because we will examine the CNNs in greater detail in the next chapter.