5
Generative Models in Deep Learning

In this chapter, we will cover the following topics:

- Comparing principal component analysis with the Restricted Boltzmann machine
- Setting up a Restricted Boltzmann machine for Bernoulli distribution input
- Training a Restricted Boltzmann machine
- Backward or reconstruction phase of RBM
- Understanding the contrastive divergence of the reconstruction
- Initializing and starting a new TensorFlow session
- Evaluating the output from an RBM
- Setting up a Restricted Boltzmann machine for Collaborative Filtering
- Performing a full run of training an RBM
- Setting up a Deep Belief Network
- Implementing a feed-forward backpropagation Neural Network
- Setting up a Deep Restricted Boltzmann Machine

Comparing principal component analysis with the Restricted Boltzmann machine

In this section, you will learn about two widely recommended dimensionality reduction techniques--**Principal component analysis** (**PCA**) and the **Restricted Boltzmann machine** (**RBM**). Consider a vector v in n-dimensional space. The dimensionality reduction technique essentially transforms the vector v into a relatively smaller (or sometimes equal) vector v' with m-dimensions ($m<n$). The transformation can be either linear or nonlinear.

PCA performs a linear transformation on features such that orthogonally adjusted components are generated that are later ordered based on their relative importance of variance capture. These m components can be considered as new input features, and can be defined as follows:

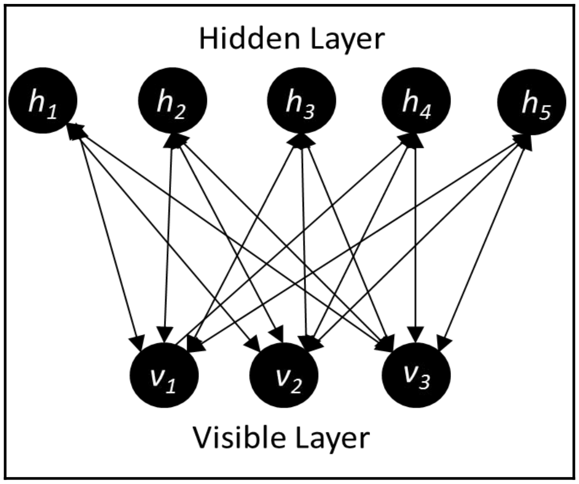
Vector $v' = \sum_{i=1}^{m} w_i c_i$

Here, w and c correspond to weights (loading) and transformed components, respectively.

Unlike PCA, RBMs (or DBNs/autoencoders) perform non-linear transformations using connections between visible and hidden units, as described in `Chapter 4`, *Data Representation Using Autoencoders*. The nonlinearity helps in better understanding the relationship with latent variables. Along with information capture, they also tend to remove noise. RBMs are generally based on stochastic distribution (either Bernoulli or Gaussian).

A large amount of Gibbs sampling is performed to learn and optimize the connection weights between visible and hidden layers. The optimization happens in two passes: a forward pass where hidden layers are sampled using given visible layers and a backward pass where visible layers are resampled using given hidden layers. The optimization is performed to minimize the reconstruction error.

The following image represents a restricted Boltzmann machine:



Getting ready

For this recipe, you will require R (the `rbm` and `ggplot2` packages) and the MNIST dataset. The MNIST dataset can be downloaded from the TensorFlow dataset library. The dataset consists of handwritten images of 28 x 28 pixels. It has 55,000 training examples and 10,000 test examples. It can be downloaded from the `tensorflow` library using the following script:

```
library(tensorflow)
datasets <- tf$contrib$learn$datasets
mnist <- datasets$mnist$read_data_sets("MNIST-data", one_hot = TRUE)
```

How to do it...

1. Extract the train dataset (`trainX` with all 784 independent variables and `trainY` with the respective 10 binary outputs):

```
trainX <- mnist$train$images
trainY <- mnist$train$labels
```

2. Run a PCA on the `trainX` data:

```
PCA_model <- prcomp(trainX, retx=TRUE)
```

3. Run an RBM on the `trainX` data:

```
RBM_model <- rbm(trainX, retx=TRUE, max_epoch=500,num_hidden =900)
```

4. Predict on the train data using the generated models. In the case of the RBM model, generate probabilities:

```
PCA_pred_train <- predict(PCA_model)
RBM_pred_train <- predict(RBM_model,type='probs')
```

5. Convert the outcomes into data frames:

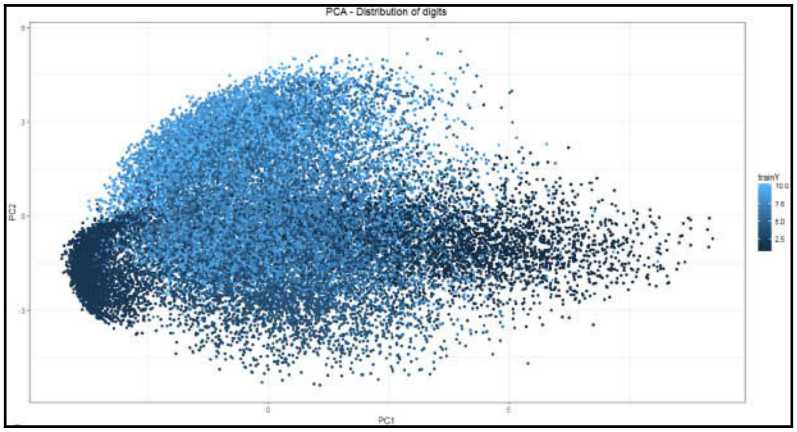
```
PCA_pred_train <- as.data.frame(PCA_pred_train)
 class="MsoSubtleEmphasis">RBM_pred_train <-
as.data.frame(as.matrix(RBM_pred_train))
```

6. Convert the 10-class binary `trainY` data frame into a numeric vector:

```
    trainY_num<-
as.numeric(stringi::stri_sub(colnames(as.data.frame(trainY))[max.co
l(as.data.frame(trainY),ties.method="first")],2))
```

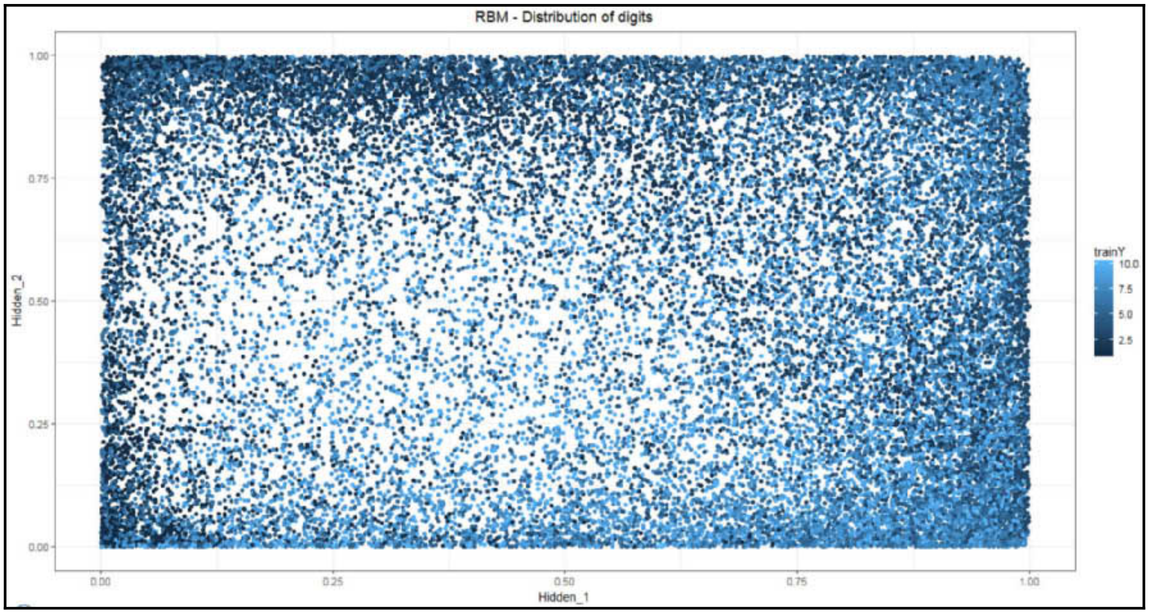
7. Plot the components generated using PCA. Here, the *x*-axis represents component 1 and the *y*-axis represents component 2. The following image shows the outcome of the PCA model:

```
ggplot(PCA_pred_train, aes(PC1, PC2))+
  geom_point(aes(colour = trainY))+
  theme_bw()+labs()+
  theme(plot.title = element_text(hjust = 0.5))
```



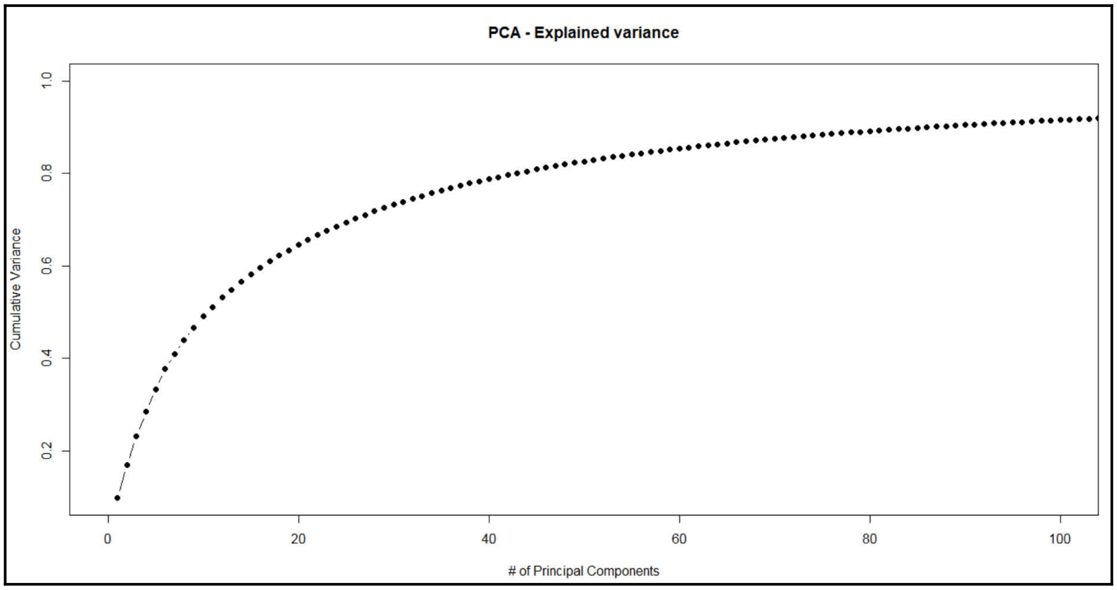
8. Plot the hidden layers generated using PCA. Here, the *x*-axis represents hidden 1 and *y*-axis represents hidden 2. The following image shows the outcome of the RBM model:

```
ggplot(RBM_pred_train, aes(Hidden_2, Hidden_3))+
  geom_point(aes(colour = trainY))+
  theme_bw()+labs()+
  theme(plot.title = element_text(hjust = 0.5))
```



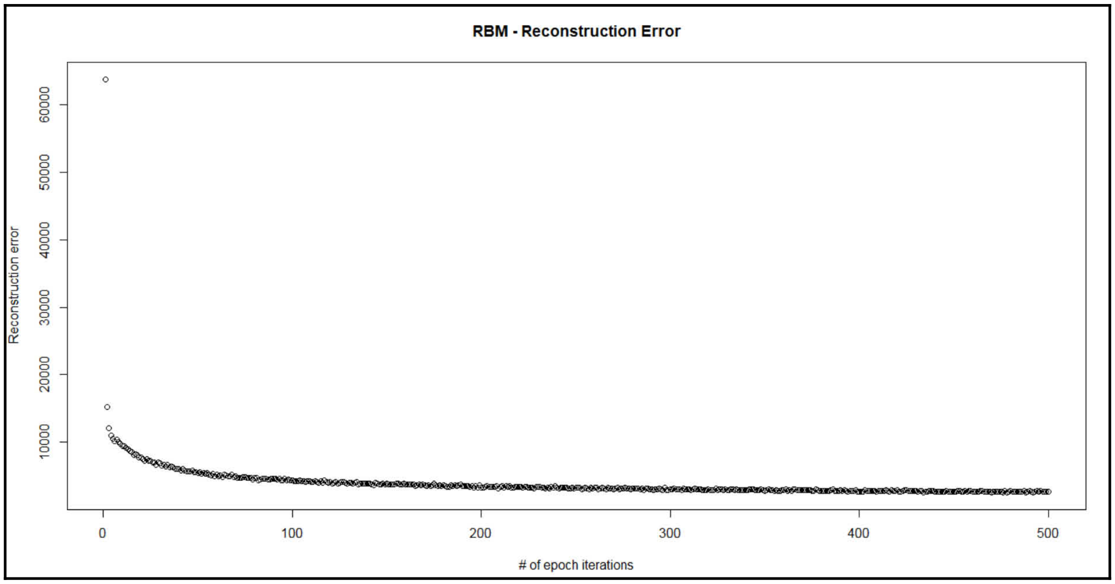
The following code and image shows the cumulative variance explained by the principal components:

```
var_explain <- as.data.frame(PCA_model$sdev^2/sum(PCA_model$sdev^2))
var_explain <- cbind(c(1:784),var_explain,cumsum(var_explain[,1]))
colnames(var_explain) <- c("PcompNo.","Ind_Variance","Cum_Variance")
plot(var_explain$PcompNo.,var_explain$Cum_Variance, xlim =
c(0,100),type='b',pch=16,xlab = "# of Principal Components",ylab =
"Cumulative Variance",main = 'PCA - Explained variance')
```



The following code and image shows the decrease in the reconstruction training error while generating an RBM using multiple epochs:

```
plot(RBM_model,xlab = "# of epoch iterations",ylab = "Reconstruction
error",main = 'RBM - Reconstruction Error')
```



Setting up a Restricted Boltzmann machine for Bernoulli distribution input

In this section, let's set up a restricted Boltzmann machine for Bernoulli distributed input data, where each attribute has values ranging from 0 to 1 (equivalent to a probability distribution). The dataset (MNIST) used in this recipe has input data satisfying a Bernoulli distribution.

An RBM comprises of two layers: a visible layer and a hidden layer. The visible layer is an input layer of nodes equal to the number of input attributes. In our case, each image in the MNIST dataset is defined using 784 pixels (28 x 28 size). Hence, our visible layer will have 784 nodes.

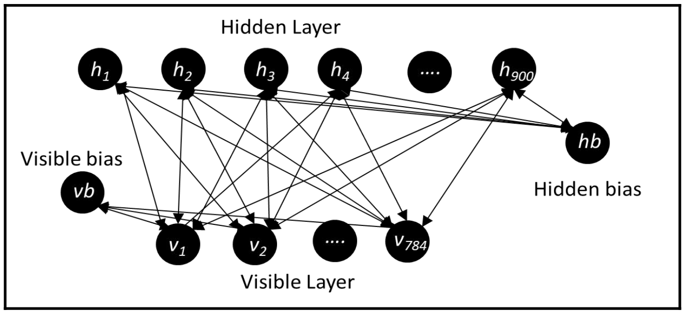
On the other hand, the hidden layer is generally user-defined. The hidden layer has a set of binary activated nodes, with each node having a probability of linkage with all other visible nodes. In our case, the hidden layer will have 900 nodes. As an initial step, all the nodes in the visible layer are connected with all the nodes in the hidden layer bidirectionally.

Each connection is defined using a weight, and hence a weight matrix is defined where the rows represent the number of input nodes and the columns represent the number of hidden nodes. In our case, the weight matrix (*w*) will be a tensor of dimensions 784 x 900.

In addition to weights, all the nodes in each layer are assisted by a bias node. The bias node of the visible layer will have connections with all the visible nodes (that is, the 784 nodes) and is represented with **vb**, whereas the bias node of the hidden layer will have connections with all the hidden nodes (that is, the 900 nodes) and is represented as **vh**.

A point to remember with RBMs is that there will be no connections among nodes within each layer. In other words, the connections will be interlayer, but not intralayer.

The following image represents an RBM with the visible layer, hidden layer, and interconnections:



Getting ready

This section provides the requirements for setting up an RBM.

- TensorFlow in R is installed and set up
- The `mnist` data is downloaded and loaded for setting up RBM

How to do it...

This section provides the steps to set up the visible and hidden layers of an RBM using TensorFlow:

1. Start a new interactive TensorFlow session:

```
# Reset the graph
tf$reset_default_graph()
# Starting session as interactive session
sess <- tf$InteractiveSession()
```

2. Define the model parameters. The `num_input` parameter defines the number of nodes in the visible layer and `num_hidden` defines the number of nodes in the hidden layer:

```
num_input<-784L
num_hidden<-900L
```

3. Create a placeholder variable for the weight matrix:

```
W <- tf$placeholder(tf$float32, shape = shape(num_input,
num_hidden))
```

4. Create placeholder variables of the visible and hidden biases:

```
vb <- tf$placeholder(tf$float32, shape = shape(num_input))
hb <- tf$placeholder(tf$float32, shape = shape(num_hidden))
```

Training a Restricted Boltzmann machine

Every training step of an RBM goes through two phases: the forward phase and the backward phase (or reconstruction phase). The reconstruction of visible units is fine tuned by making several iterations of the forward and backward phases.

Training a forward phase: In the forward phase, the input data is passed from the visible layer to the hidden layer and all the computation occurs within the nodes of the hidden layer. The computation is essentially to take a stochastic decision of each connection from the visible to the hidden layer. In the hidden layer, the input data (`X`) is multiplied by the weight matrix (`W`) and added to a hidden bias vector (`hb`).

The resultant vector of a size equal to the number of hidden nodes is then passed through a sigmoid function to determine each hidden node's output (or activation state). In our case, each input digit will produce a tensor vector of 900 probabilities, and as we have 55,000 input digits, we will have an activation matrix of the size 55,000 x 900. Using the hidden layer's probability distribution matrix, we can generate samples of activation vectors that can be used later to estimate negative phase gradients.

Getting ready

This section provides the requirements for setting up an RBM.

- TensorFlow in R is installed and set up
- The `mnist` data is downloaded and loaded for setting up the RBM
- The RBM model is set up as described in the recipe *Setting up a Restricted Boltzmann machine for Bernoulli distribution input*

Example of a sampling

Consider a constant vector `s1` equivalent to a tensor vector of probabilities. Then, create a new random uniformly distributed sample `s2` using the distribution of the constant vector `s1`. Then calculate the difference and apply a rectified linear activation function.

How to do it...

This section provides the steps to set up the script for running the RBM model using TensorFlow:

```
X = tf$placeholder(tf$float32, shape=shape(NULL, num_input))
prob_h0= tf$nn$sigmoid(tf$matmul(X, W) + hb)
h0 = tf$nn$relu(tf$sign(prob_h0 - tf$random_uniform(tf$shape(prob_h0))))
```

Use the following code to execute the graph created in TensorFlow:

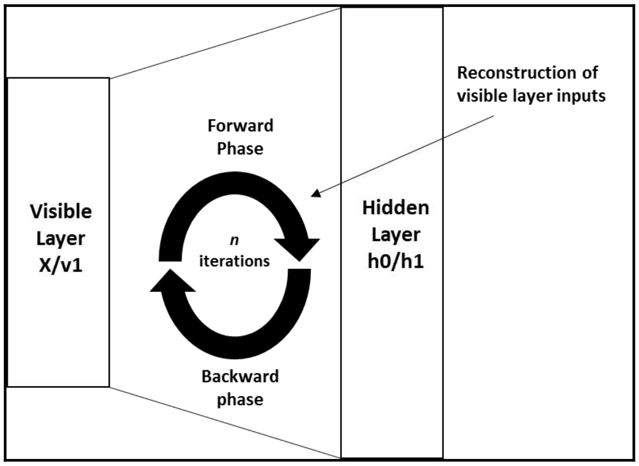
```
sess$run(tf$global_variables_initializer())
s1 <- tf$constant(value = c(0.1,0.4,0.7,0.9))
cat(sess$run(s1))
s2=sess$run(tf$random_uniform(tf$shape(s1)))
cat(s2)
cat(sess$run(s1-s2))
cat(sess$run(tf$sign(s1 - s2)))
cat(sess$run(tf$nn$relu(tf$sign(s1 - s2))))
```

Backward or reconstruction phase of RBM

In the reconstruction phase, the data from the hidden layer is passed back to the visible layer. The hidden layer vector of probabilities `h0` is multiplied by the transpose of the weight matrix `W` and added to a visible layer bias `vb`, which is then passed through a sigmoid function to generate a reconstructed input vector `prob_v1`.

A sample input vector is created using the reconstructed input vector, which is then multiplied by the weight matrix `W` and added to the hidden bias vector `hb` to generate an updated hidden vector of probabilities `h1`.

This is also called Gibbs sampling. In some scenarios, the sample input vector is not generated and the reconstructed input vector `prob_v1` is directly used to update the hi



Getting ready

This section provides the requirements for image reconstruction using the input probability vector.

- `mnist` data is loaded in the environment
- The RBM model is trained using the recipe *Training a Restricted Boltzmann machine*

How to do it...

This section covers the steps to perform backward reconstruction and evaluation:

1. The backward image reconstruction can be performed using the input probability vector with the following script:

```
prob_v1 = tf$nn$sigmoid(tf$matmul(h0, tf$transpose(W)) + vb)
v1 = tf$nn$relu(tf$sign(prob_v1 -
tf$random_uniform(tf$shape(prob_v1))))
h1 = tf$nn$sigmoid(tf$matmul(v1, W) + hb)
```

2. The evaluation can be performed using a defined metric, such as **mean squared error** (**MSE**), which is computed between the actual input data (`X`) and the reconstructed input data (`v1`). The MSE is computed after each epoch and the key objective is to minimize the MSE:

```
err = tf$reduce_mean(tf$square(X - v1))
```

Understanding the contrastive divergence of the reconstruction

As an initial start, the objective function can be defined as the minimization of the average negative log-likelihood of reconstructing the visible vector *v* where *P(v)* denotes the vector of generated probabilities:

$$\text{arg min (w) -E} \left[\sum_{\vartheta \in V} \log P(\vartheta)\right]$$

Getting ready

This section provides the requirements for image reconstruction using the input probability vector.

- `mnist` data is loaded in the environment
- The images are reconstructed using the recipe *Backward or reconstruction phase*

How to do it...

This current recipe present the steps for, a **contrastive divergence** (**CD**) technique used to speed up the sampling process:

1. Compute a positive weight gradient by multiplying (outer product) the input vector `X` with a sample of the hidden vector `h0` from the given probability distribution `prob_h0`:

```
w_pos_grad = tf$matmul(tf$transpose(X), h0)
```

2. Compute a negative weight gradient by multiplying (outer product) the sample of the reconstructed input data `v1` with the updated hidden activation vector `h1`:

```
w_neg_grad = tf$matmul(tf$transpose(v1), h1)
```

3. Then, compute the `CD` matrix by subtracting the negative gradient from the positive gradient and dividing by the size of the input data:

```
CD = (w_pos_grad - w_neg_grad) / tf$to_float(tf$shape(X)[0])
```

4. Then, update the weight matrix `W` to `update_W` using a learning rate (*alpha*) and the CD matrix:

```
update_w = W + alpha * CD
```

5. Additionally, update the visible and hidden bias vectors:

```
update_vb = vb + alpha * tf$reduce_mean(X - v1)
update_hb = hb + alpha * tf$reduce_mean(h0 - h1)
```

How it works...

The objective function can be minimized using stochastic gradient descent by indirectly modifying (and optimizing) the weight matrix. The entire gradient can be further divided into two forms based on the probability density: positive gradient and negative gradient. The positive gradient primarily depends on the input data and the negative gradient depends only on the generated model.

In the positive gradient, the probability toward the reconstructing training data increases, and in the negative gradient, the probability of randomly generated uniform samples by the model decreases.

The CD technique is used to optimize the negative phase. In the CD technique, the weight matrix is adjusted in each iteration of reconstruction. The new weight matrix is generated using the following formula. The learning rate is defined as *alpha*, in our case:

$$W' = W + learning\ rate * CD$$

Initializing and starting a new TensorFlow session

A big part of calculating the error metric such as mean square error (MSE) is initialization and starting a new TensorFlow session. Here is how we proceed with it.

Getting ready

This section provides the requirements for starting a new TensorFlow session used to compute the error metric.

- `mnist` data is loaded in the environment
- The TensorFlow graph for the RBM is loaded

How to do it...

This section provides the steps for optimizing the error using reconstruction from an RBM:

1. Initialize the current and previous vector of biases and matrices of weights:

```
cur_w = tf$Variable(tf$zeros(shape = shape(num_input, num_hidden),
dtype=tf$float32))
cur_vb = tf$Variable(tf$zeros(shape = shape(num_input),
dtype=tf$float32))
cur_hb = tf$Variable(tf$zeros(shape = shape(num_hidden),
dtype=tf$float32))
prv_w = tf$Variable(tf$random_normal(shape=shape(num_input,
num_hidden), stddev=0.01, dtype=tf$float32))
prv_vb = tf$Variable(tf$zeros(shape = shape(num_input),
dtype=tf$float32))
prv_hb = tf$Variable(tf$zeros(shape = shape(num_hidden),
dtype=tf$float32))
```

2. Start a new TensorFlow session:

```
sess$run(tf$global_variables_initializer())
```

3. Perform a first run with the full input data (**trainX**) and obtain the first set of weight matrix and bias vectors:

```
output <- sess$run(list(update_w, update_vb, update_hb), feed_dict
= dict(X=trainX,
W = prv_w$eval(),
vb = prv_vb$eval(),
hb = prv_hb$eval()))
prv_w <- output[[1]]
prv_vb <-output[[2]]
prv_hb <-output[[3]]
```

4. Let's look at the error of the first run:

```
sess$run(err, feed_dict=dict(X= trainX, W= prv_w, vb= prv_vb, hb=
prv_hb))
```

5. The full model for the RBM can be trained using the following script:

```
epochs=15
errors <- list()
weights <- list()
u=1
for(ep in 1:epochs){
  for(i in seq(0,(dim(trainX)[1]-100),100)){
    batchX <- trainX[(i+1):(i+100),]
    output <- sess$run(list(update_w, update_vb, update_hb),
feed_dict = dict(X=batchX,
W = prv_w,
vb = prv_vb,
hb = prv_hb))
    prv_w <- output[[1]]
    prv_vb <- output[[2]]
    prv_hb <-  output[[3]]
    if(i%%10000 == 0){
      errors[[u]] <- sess$run(err, feed_dict=dict(X= trainX, W=
prv_w, vb= prv_vb, hb= prv_hb))
      weights[[u]] <- output[[1]]
      u <- u+1
     cat(i , " : ")
    }
  }
  cat("epoch :", ep, " : reconstruction error : ",
errors[length(errors)][[1]],"\n")
}
```

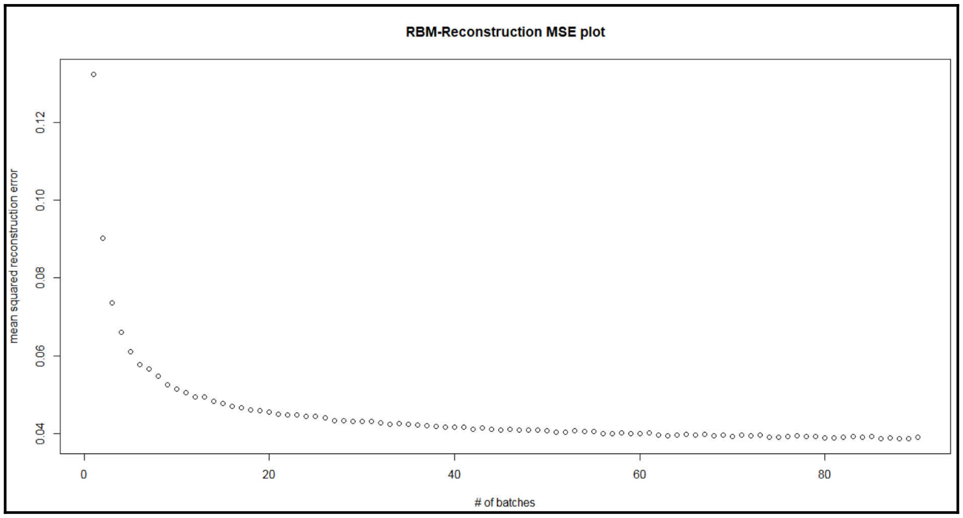
6. Plot reconstruction using mean squared errors:

```
error_vec <- unlist(errors)
plot(error_vec,xlab="# of batches",ylab="mean squared
reconstruction error",main="RBM-Reconstruction MSE plot")
```

How it works...

Here, we will run 15 epochs (or iterations) where, in each epoch, a batchwise (size = 100) optimization is performed. In each batch, the CD is computed and, accordingly, weights and biases are updated. To keep track of the optimization, MSE is calculated after every batch of 10,000 rows.

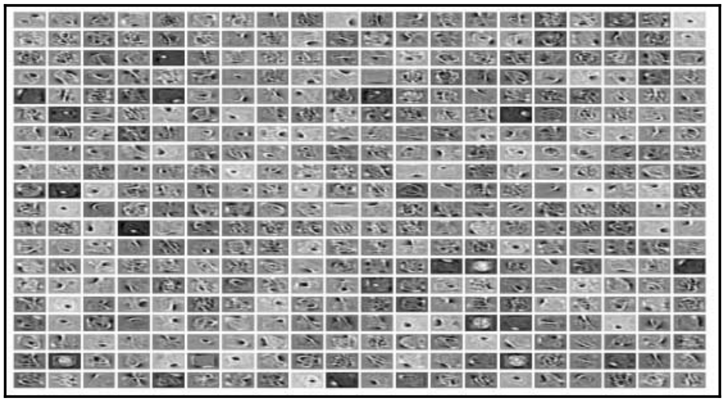
The following image shows the declining trend of mean squared reconstruction errors computed for 90 batches:



Evaluating the output from an RBM

Here, let's plot the weights of the final layer with respect to the output (reconstruction input data). In the current scenario, 900 is the number of nodes in the hidden layer and 784 is the number of nodes in the output (reconstructed) layer.

In the following image, the first 400 nodes in the hidden layer are seen:



Here, each tile represents a vector of connections formed between a hidden node and all the visible layer nodes. In each tile, the black region represents negative weights (weight < 0), the white region represents positive weights (weight > 1), and the grey region represents no connection (weight = 0). The higher the positive value, the greater the chance of activation in hidden nodes, and vice versa. These activations help determine which part of the input image is being determined by a given hidden node.

Getting ready

This section provides the requirements for running the evaluation recipe:

- `mnist` data is loaded in the environment
- The RBM model is executed using TensorFlow and the optimal weights are obtained

How to do it...

This recipe covers the steps for the evaluation of weights obtained from an RBM:

1. Run the following code to generate the image of 400 hidden nodes:

```
uw = t(weights[[length(weights)]])   # Extract the most recent
weight matrix
numXpatches = 20    # Number of images in X-axis (user input)
numYpatches=20      # Number of images in Y-axis (user input)
pixels <- list()
op <- par(no.readonly = TRUE)
par(mfrow = c(numXpatches,numYpatches), mar = c(0.2, 0.2, 0.2,
0.2), oma = c(3, 3, 3, 3))
for (i in 1:(numXpatches*numYpatches)) {
  denom <- sqrt(sum(uw[i, ]^2))
  pixels[[i]] <- matrix(uw[i, ]/denom, nrow = numYpatches, ncol =
numXpatches)
  image(pixels[[i]], axes = F, col = gray((0:32)/32))
}
par(op)
```

2. Select a sample of four actual input digits from the training data:

```
sample_image <- trainX[1:4,]
```

3. Then, visualize these sample digits using the following code:

```
mw=melt(sample_image)
mw$X3=floor((mw$X2-1)/28)+1
mw$X2=(mw$X2-1)%%28 + 1;
mw$X3=29-mw$X3
ggplot(data=mw)+geom_tile(aes(X2,X3,fill=value))+facet_wrap(~X1,nro
w=2)+
scale_fill_continuous(low='black',high='white')+coord_fixed(ratio=1
)+
  labs(x=NULL,y=NULL,)+
  theme(legend.position="none")+
  theme(plot.title = element_text(hjust = 0.5))
```

4. Now, reconstruct these four sample images using the final weights and biases obtained:

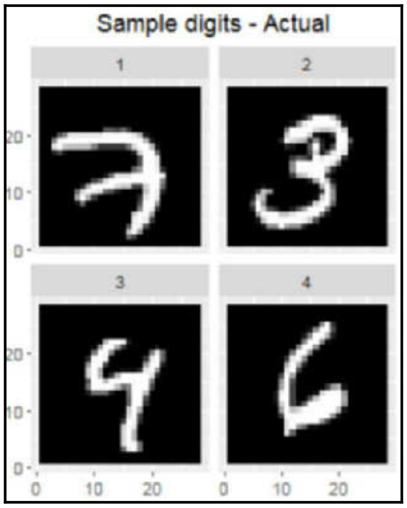
```
hh0 = tf$nn$sigmoid(tf$matmul(X, W) + hb)
vv1 = tf$nn$sigmoid(tf$matmul(hh0, tf$transpose(W)) + vb)
feed = sess$run(hh0, feed_dict=dict( X= sample_image, W= prv_w, hb=
prv_hb))
rec = sess$run(vv1, feed_dict=dict( hh0= feed, W= prv_w, vb=
prv_vb))
```

5. Then, visualize the reconstructed sample digits using the following code:

```
mw=melt(rec)
mw$X3=floor((mw$X2-1)/28)+1
mw$X2=(mw$X2-1)%%28 + 1
mw$X3=29-mw$X3
ggplot(data=mw)+geom_tile(aes(X2,X3,fill=value))+facet_wrap(~X1,nro
w=2)+
scale_fill_continuous(low='black',high='white')+coord_fixed(ratio=1
)+
  labs(x=NULL,y=NULL,)+
  theme(legend.position="none")+
  theme(plot.title = element_text(hjust = 0.5))
```

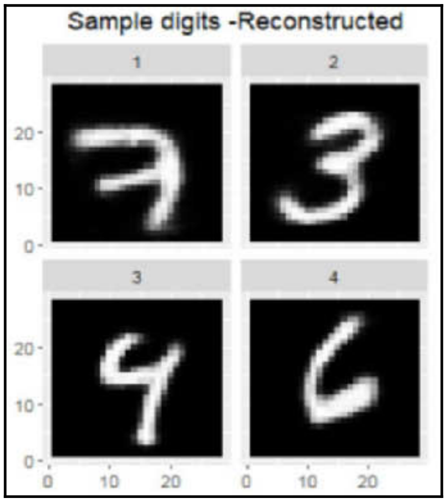
How it works...

The following image illustrates a raw image of the four sample digits:



The reconstructed images seemed to have had their noise removed, especially in the case of digits **3** and **6**.

The following image illustrates a reconstructed image of the same four digits:



Setting up a Restricted Boltzmann machine for Collaborative Filtering

In this recipe, you will learn how to build a collaborative-filtering-based recommendation system using an RBM. Here, for every user, the RBM tries to identify similar users based on their past behavior of rating various items, and then tries to recommend the next best item.

Getting ready

In this recipe, we will use the movielens dataset from the Grouplens research organization. The datasets (`movies.dat` and `ratings.dat`) can be downloaded from the following link. `Movies.dat` contains information of 3,883 movies and `Ratings.dat` contains information of 1,000,209 user ratings for these movies. The ratings range from 1 to 5, with 5 being the highest.

`http://files.grouplens.org/datasets/movielens/ml-1m.zip`

How to do it...

This recipe covers the steps for setting up collaborative filtering using an RBM.

1. Read the `movies.dat` datasets in R:

```
txt <- readLines("movies.dat", encoding = "latin1")
txt_split <- lapply(strsplit(txt, "::"), function(x)
as.data.frame(t(x), stringsAsFactors=FALSE))
movies_df <- do.call(rbind, txt_split)
names(movies_df) <- c("MovieID", "Title", "Genres")
movies_df$MovieID <- as.numeric(movies_df$MovieID)
```

2. Add a new column (`id_order`) to the movies dataset, as the current ID column (`UserID`) cannot be used to index movies because they range from 1 to 3,952:

```
movies_df$id_order <- 1:nrow(movies_df)
```

3. Read the `ratings.dat` dataset in R:

```
ratings_df <- read.table("ratings.dat",
sep=":",header=FALSE,stringsAsFactors = F)
ratings_df <- ratings_df[,c(1,3,5,7)]
colnames(ratings_df) <- c("UserID","MovieID","Rating","Timestamp")
```

4. Merge the movies and ratings datasets with `all=FALSE`:

```
merged_df <- merge(movies_df, ratings_df, by="MovieID",all=FALSE)
```

5. Remove the non-required columns:

```
merged_df[,c("Timestamp","Title","Genres")] <- NULL
```

6. Convert the ratings to percentages:

```
merged_df$rating_per <- merged_df$Rating/5
```

7. Generate a matrix of ratings across all the movies for a sample of 1,000 users:

```
num_of_users <- 1000
num_of_movies <- length(unique(movies_df$MovieID))
trX <- matrix(0,nrow=num_of_users,ncol=num_of_movies)
for(i in 1:num_of_users){
  merged_df_user <- merged_df[merged_df$UserID %in% i,]
  trX[i,merged_df_user$id_order] <- merged_df_user$rating_per
}
```

8. Look at the distribution of the `trX` training dataset. It seems to follow a Bernoulli distribution (values in the range of 0 to 1):

```
summary(trX[1,]); summary(trX[2,]); summary(trX[3,])
```

9. Define the input model parameters:

```
num_hidden = 20
num_input = nrow(movies_df)
```

10. Start a new TensorFlow session:

```
sess$run(tf$global_variables_initializer())
output <- sess$run(list(update_w, update_vb, update_hb), feed_dict
= dict(v0=trX,
W = prv_w$eval(),
vb = prv_vb$eval(),
hb = prv_hb$eval()))
prv_w <- output[[1]]
prv_vb <- output[[2]]
prv_hb <-  output[[3]]
sess$run(err_sum, feed_dict=dict(v0=trX, W= prv_w, vb= prv_vb, hb=
prv_hb))
```

11. Train the RBM using 500 epoch iterations and a batch size of 100:

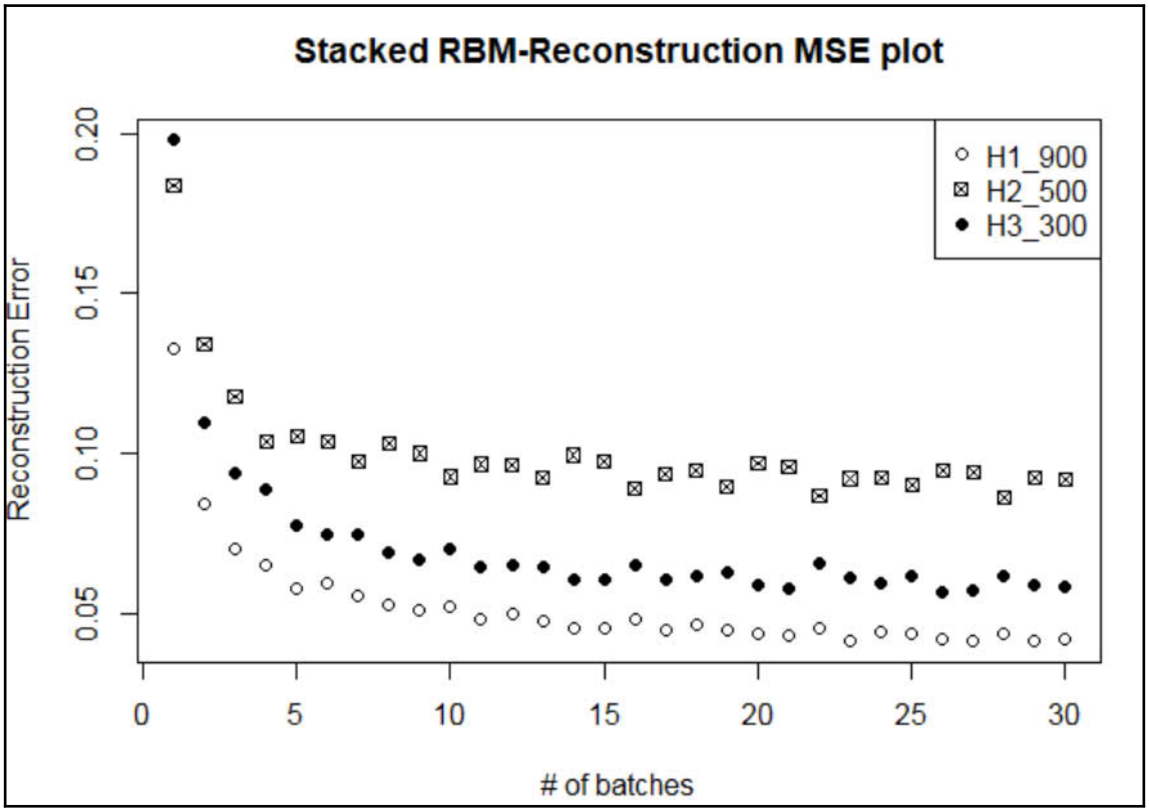
```
epochs= 500
errors <- list()
weights <- list()

for(ep in 1:epochs){
  for(i in seq(0,(dim(trX)[1]-100),100)){
    batchX <- trX[(i+1):(i+100),]
    output <- sess$run(list(update_w, update_vb, update_hb),
feed_dict = dict(v0=batchX,
W = prv_w,
vb = prv_vb,
hb = prv_hb))
    prv_w <- output[[1]]
    prv_vb <- output[[2]]
    prv_hb <-  output[[3]]
    if(i%%1000 == 0){
      errors <- c(errors,sess$run(err_sum,
feed_dict=dict(v0=batchX, W= prv_w, vb= prv_vb, hb= prv_hb)))
      weights <- c(weights,output[[1]])
      cat(i , " : ")
    }
  }
  cat("epoch :", ep, " : reconstruction error : ",
errors[length(errors)][[1]],"\n")
}
```

12. Plot reconstruction mean squared errors:

```
error_vec <- unlist(errors)
plot(error_vec,xlab="# of batches",ylab="mean squared
reconstruction error",main="RBM-Reconstruction MSE plot")
```

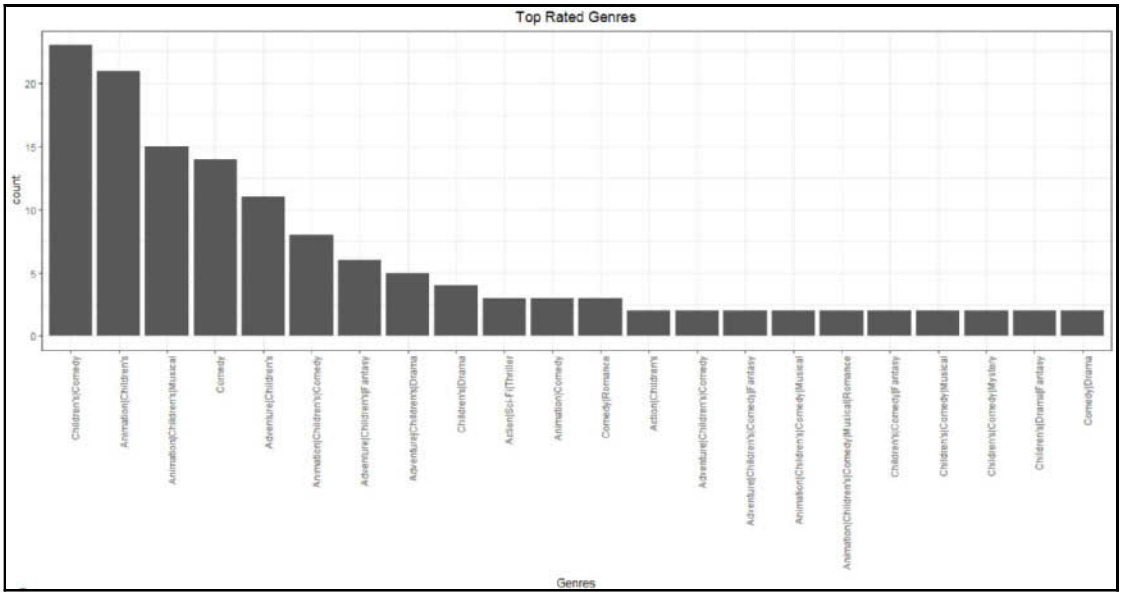
Performing a full run of training an RBM

Using the same RBM setup mentioned in the preceding recipe, train the RBM on the user ratings dataset (`trX`) using 20 hidden nodes. To keep a track of the optimization, the MSE is calculated after every batch of 1,000 rows. The following image shows the declining trend of mean squared reconstruction errors computed for 500 batches (equal to epochs):

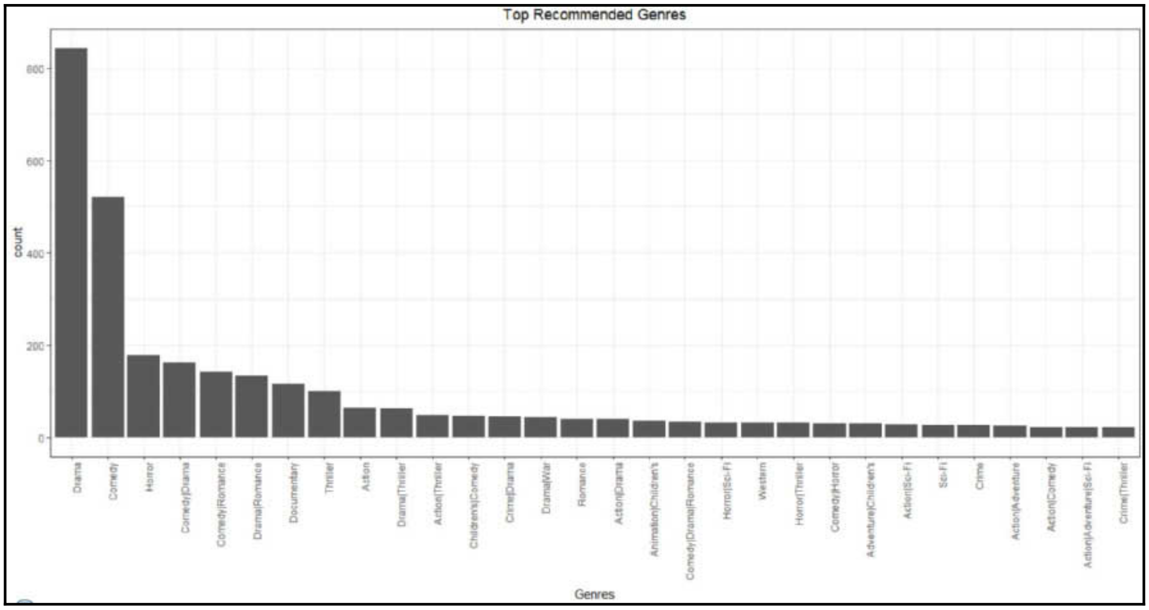


Looking into RBM recommendations: Let's now look into the recommendations generated by RBM-based collaborative filtering for a given user ID. Here, we will look into the top-rated genres and top-recommended genres of this user ID, along with the top 10 movie recommendations.

The following image illustrates a list of top-rated genres:



The following image illustrates a list of top-recommended genres:



Getting ready

This section provides the requirements for collaborative filtering the output evaluation:

- TensorFlow in R is installed and set up
- The `movies.dat` and `ratings.dat` datasets are loaded in environment
- The recipe *Setting up a Restricted Boltzmann machine for Collaborative Filtering* has been executed

How to do it...

This recipe covers the steps for evaluating the output from RBM-based collaborative filtering:

1. Select the ratings of a user:

```
inputUser = as.matrix(t(trX[75,]))
names(inputUser) <- movies_df$id_order
```

2. Remove the movies that were not rated by the user (assuming that they have yet to be seen):

```
inputUser <- inputUser[inputUser>0]
```

3. Plot the top genres seen by the user:

```
top_rated_movies <-
movies_df[as.numeric(names(inputUser)[order(inputUser,decreasing =
TRUE)]),]$Title
top_rated_genres <-
movies_df[as.numeric(names(inputUser)[order(inputUser,decreasing =
TRUE)]),]$Genres
top_rated_genres <-
as.data.frame(top_rated_genres,stringsAsFactors=F)
top_rated_genres$count <- 1
top_rated_genres <-
aggregate(count~top_rated_genres,FUN=sum,data=top_rated_genres)
top_rated_genres <- top_rated_genres[with(top_rated_genres, order(-
count)), ]
top_rated_genres$top_rated_genres <-
factor(top_rated_genres$top_rated_genres, levels =
top_rated_genres$top_rated_genres)
ggplot(top_rated_genres[top_rated_genres$count>1,],aes(x=top_rated_
genres,y=count))+
geom_bar(stat="identity")+
theme_bw()+
theme(axis.text.x = element_text(angle = 90, hjust = 1))+
labs(x="Genres",y="count",)+
theme(plot.title = element_text(hjust = 0.5))
```

4. Reconstruct the input vector to obtain the recommendation percentages for all the genres/movies:

```
hh0 = tf$nn$sigmoid(tf$matmul(v0, W) + hb)
vv1 = tf$nn$sigmoid(tf$matmul(hh0, tf$transpose(W)) + vb)
feed = sess$run(hh0, feed_dict=dict( v0= inputUser, W= prv_w, hb=
prv_hb))
rec = sess$run(vv1, feed_dict=dict( hh0= feed, W= prv_w, vb=
prv_vb))
names(rec) <- movies_df$id_order
```

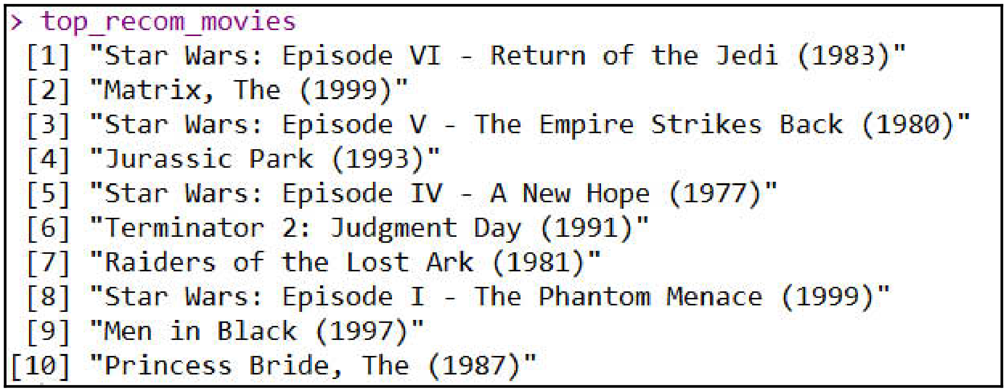
5. Plot the top-recommended genres:

```
top_recom_genres <-
movies_df[as.numeric(names(rec)[order(rec,decreasing =
TRUE)]),]$Genres
top_recom_genres <-
as.data.frame(top_recom_genres,stringsAsFactors=F)
top_recom_genres$count <- 1
top_recom_genres <-
aggregate(count~top_recom_genres,FUN=sum,data=top_recom_genres)
top_recom_genres <- top_recom_genres[with(top_recom_genres, order(-
count)), ]
top_recom_genres$top_recom_genres <-
factor(top_recom_genres$top_recom_genres, levels =
top_recom_genres$top_recom_genres)
ggplot(top_recom_genres[top_recom_genres$count>20,],aes(x=top_recom
_genres,y=count))+
geom_bar(stat="identity")+
theme_bw()+
theme(axis.text.x = element_text(angle = 90, hjust = 1))+
labs(x="Genres",y="count",)+
theme(plot.title = element_text(hjust = 0.5))
```

6. Find the top 10 recommended movies:

```
top_recom_movies <-
movies_df[as.numeric(names(rec)[order(rec,decreasing =
TRUE)]),]$Title[1:10]
```

The following image shows the top 10 recommended movies:



```
> top_recom_movies
 [1] "Star Wars: Episode VI - Return of the Jedi (1983)"
 [2] "Matrix, The (1999)"
 [3] "Star Wars: Episode V - The Empire Strikes Back (1980)"
 [4] "Jurassic Park (1993)"
 [5] "Star Wars: Episode IV - A New Hope (1977)"
 [6] "Terminator 2: Judgment Day (1991)"
 [7] "Raiders of the Lost Ark (1981)"
 [8] "Star Wars: Episode I - The Phantom Menace (1999)"
 [9] "Men in Black (1997)"
[10] "Princess Bride, The (1987)"
```

Setting up a Deep Belief Network

Deep belief networks are a type of **Deep Neural Network** (**DNN**), and are composed of multiple hidden layers (or latent variables). Here, the connections are present only between the layers and not within the nodes of each layer. The DBN can be trained both as an unsupervised and supervised model.

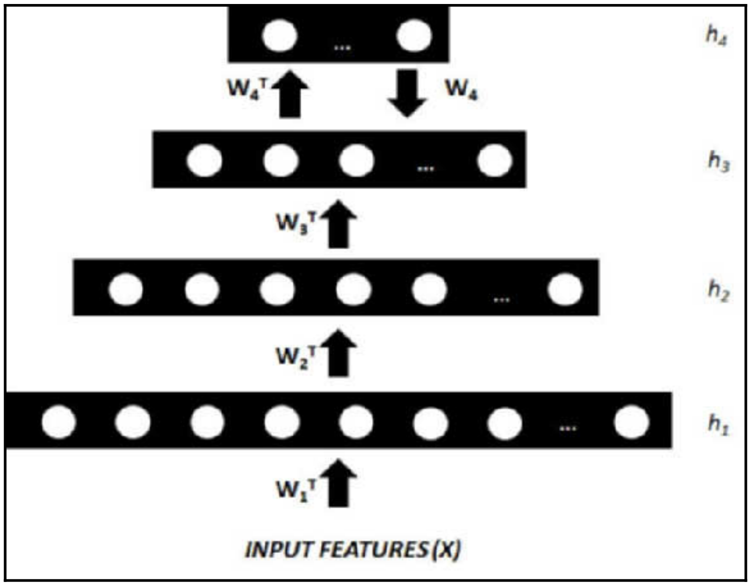
The unsupervised model is used to reconstruct the input with noise removal and the supervised model (after pretraining) is used to perform classification. As there are no connections within the nodes in each layer, the DBNs can be considered as a set of unsupervised RBMs or autoencoders, where each hidden layer serves as a visible layer to its subsequent connected hidden layer.

This kind of stacked RBM enhances the performance of input reconstruction where CD is applied across all layers, starting from the actual input training layer and finishing at the last hidden (or latent) layer.

DBNs are a type of graphical model that train the stacked RBMs in a greedy manner. Their networks tend to learn the deep hierarchical representation using joint distributions between the input feature vector *i* and hidden layers $h_{1,2....m}$:

$$P(i, h_1, h_2 \dots h_m) = \left(\prod_{k=0}^{m-2} P(h_k | h_{k+1})\right) * P(h_{m-1}, h_m)$$

Here, $i = h_0$; $P(h_{k-1} | h_k)$ is a conditional distribution of reconstructed visible units on the hidden layers of the RBM at level *k*; $P(h_{m-1}, h_m)$ is the joint distribution of hidden and visible units (reconstructed) at the final RBM layer of the DBN. The following image illustrates a DBN of four hidden layers, where **W** represents the weight matrix:



DBNs can also be used to enhance the robustness of DNNs. DNNs face an issue of local optimization while implementing backpropagation. This is possible in scenarios where an error surface features numerous troughs, and the gradient descent, due to backpropagation occurs inside a local deep trough (not a global deep trough). DBNs, on the other hand, perform pretraining of the input features, which helps the optimization direct toward the global deepest trough, and then use backpropagation, to perform a gradient descent to gradually minimize the error rate.

Training a stack of three RBMs: In this recipe, we will train a DBN using three stacked RBMs, where the first hidden layer will have 900 nodes, the second hidden layer will have 500 nodes, and the third hidden layer will have 300 nodes.

Getting ready

This section provides the requirements for TensorFlow.

- The dataset is loaded and set up
- Load the `TensorFlow` package using the following script:

```
require(tensorflow)
```

How to do it...

This recipe covers the steps for setting up **Deep belief network** (**DBM**):

1. Define the number of nodes in each hidden layer as a vector:

```
RBM_hidden_sizes = c(900, 500 , 300 )
```

2. Generate an RBM function leveraging the codes illustrated in the *Setting up a Restricted Boltzmann Machine for Bernoulli distribution input* recipe with the following input and output parameters mentioned:

Type of Parameter	Parameter name	Parameter description
Input (Pre RBM)	*input_data*	Matrix of train MNIST data
Input (Pre RBM)	*num_input*	Number of independent variables
Input (Pre RBM)	*num_output*	Number of nodes in the corresponding hidden layer
Input (Pre RBM)	*epochs*	Number of iterations
Input (Pre RBM)	*alpha*	Learning rate for updating weight matrix
Input (Pre RBM)	*batchsize*	Number of observations per batch run
Output (Post RBM)	*output_data*	Matrix of reconstructed output
Output (Post RBM)	*error_list*	List of reconstruction errors for every run of 10 batch
Output (Post RBM)	*weight_list*	List of weight matrices for every run of 10 batch
Output (Post RBM)	*weight_final*	Matrix of final weights obtained after all epochs
Output (Post RBM)	*bias_final*	Vector of final biases obtained after all epochs

Here is the function for setting up up the RBM:

```
RBM <- function(input_data, num_input, num_output, epochs = 5,
alpha = 0.1, batchsize=100){
# Placeholder variables
```

```
vb <- tf$placeholder(tf$float32, shape = shape(num_input))
hb <- tf$placeholder(tf$float32, shape = shape(num_output))
W <- tf$placeholder(tf$float32, shape = shape(num_input,
num_output))
# Phase 1 : Forward Phase
X = tf$placeholder(tf$float32, shape=shape(NULL, num_input))
prob_h0= tf$nn$sigmoid(tf$matmul(X, W) + hb) #probabilities of the
hidden units
h0 = tf$nn$relu(tf$sign(prob_h0 -
tf$random_uniform(tf$shape(prob_h0)))) #sample_h_given_X
# Phase 2 : Backward Phase
prob_v1 = tf$nn$sigmoid(tf$matmul(h0, tf$transpose(W)) + vb)
v1 = tf$nn$relu(tf$sign(prob_v1 -
tf$random_uniform(tf$shape(prob_v1))))
h1 = tf$nn$sigmoid(tf$matmul(v1, W) + hb)
# calculate gradients
w_pos_grad = tf$matmul(tf$transpose(X), h0)
w_neg_grad = tf$matmul(tf$transpose(v1), h1)
CD = (w_pos_grad - w_neg_grad) / tf$to_float(tf$shape(X)[0])
update_w = W + alpha * CD
update_vb = vb + alpha * tf$reduce_mean(X - v1)
update_hb = hb + alpha * tf$reduce_mean(h0 - h1)
# Objective function
err = tf$reduce_mean(tf$square(X - v1))
# Initialize variables
cur_w = tf$Variable(tf$zeros(shape = shape(num_input, num_output),
dtype=tf$float32))
cur_vb = tf$Variable(tf$zeros(shape = shape(num_input),
dtype=tf$float32))
cur_hb = tf$Variable(tf$zeros(shape = shape(num_output),
dtype=tf$float32))
prv_w = tf$Variable(tf$random_normal(shape=shape(num_input,
num_output), stddev=0.01, dtype=tf$float32))
prv_vb = tf$Variable(tf$zeros(shape = shape(num_input),
dtype=tf$float32))
prv_hb = tf$Variable(tf$zeros(shape = shape(num_output),
dtype=tf$float32))
# Start tensorflow session
sess$run(tf$global_variables_initializer())
output <- sess$run(list(update_w, update_vb, update_hb), feed_dict
= dict(X=input_data,
W = prv_w$eval(),
vb = prv_vb$eval(),
hb = prv_hb$eval()))
prv_w <- output[[1]]
prv_vb <- output[[2]]
prv_hb <- output[[3]]
sess$run(err, feed_dict=dict(X= input_data, W= prv_w, vb= prv_vb,
hb= prv_hb))
errors <- weights <- list()
u=1
for(ep in 1:epochs){
for(i in seq(0,(dim(input_data)[1]-batchsize),batchsize)){
batchX <- input_data[(i+1):(i+batchsize),]
output <- sess$run(list(update_w, update_vb, update_hb), feed_dict
= dict(X=batchX,
W = prv_w,
vb = prv_vb,
hb = prv_hb))
prv_w <- output[[1]]
prv_vb <- output[[2]]
prv_hb <- output[[3]]
if(i%%10000 == 0){
errors[[u]] <- sess$run(err, feed_dict=dict(X= batchX, W= prv_w,
vb= prv_vb, hb= prv_hb))
weights[[u]] <- output[[1]]
u=u+1
cat(i , " : ")
}
}
cat("epoch :", ep, " : reconstruction error : ",
errors[length(errors)][[1]],"\n")
}
w <- prv_w
vb <- prv_vb
hb <- prv_hb
# Get the output
input_X = tf$constant(input_data)
ph_w = tf$constant(w)
ph_hb = tf$constant(hb)
out = tf$nn$sigmoid(tf$matmul(input_X, ph_w) + ph_hb)
sess$run(tf$global_variables_initializer())
return(list(output_data = sess$run(out),
error_list=errors,
weight_list=weights,
weight_final=w,
bias_final=hb))
}
```

3. Train the RBM for all three different types of hidden nodes in a sequence. In other words, first train RBM1 with 900 hidden nodes, then use the output of RBM1 as an input for RBM2 with 500 hidden nodes and train RBM2, then use the output of RBM2 as an input for RBM3 with 300 hidden nodes and train RBM3. Store the outputs of all three RBMs as a list, `RBM_output`:

```
inpX = trainX
RBM_output <- list()
for(i in 1:length(RBM_hidden_sizes)){
size <- RBM_hidden_sizes[i]
# Train the RBM
RBM_output[[i]] <- RBM(input_data= inpX,
num_input= ncol(trainX),
num_output=size,
epochs = 5,
alpha = 0.1,
batchsize=100)
# Update the input data
inpX <- RBM_output[[i]]$output_data
# Update the input_size
num_input = size
cat("completed size :", size,"\n")
}
```

4. Create a data frame of batch errors across three hidden layers:

```
error_df <-
data.frame("error"=c(unlist(RBM_output[[1]]$error_list),unlist(RBM_
output[[2]]$error_list),unlist(RBM_output[[3]]$error_list)),
"batches"=c(rep(seq(1:length(unlist(RBM_output[[1]]$error_list))),t
imes=3)),
"hidden_layer"=c(rep(c(1,2,3),each=length(unlist(RBM_output[[1]]$er
ror_list)))),
stringsAsFactors = FALSE)
```

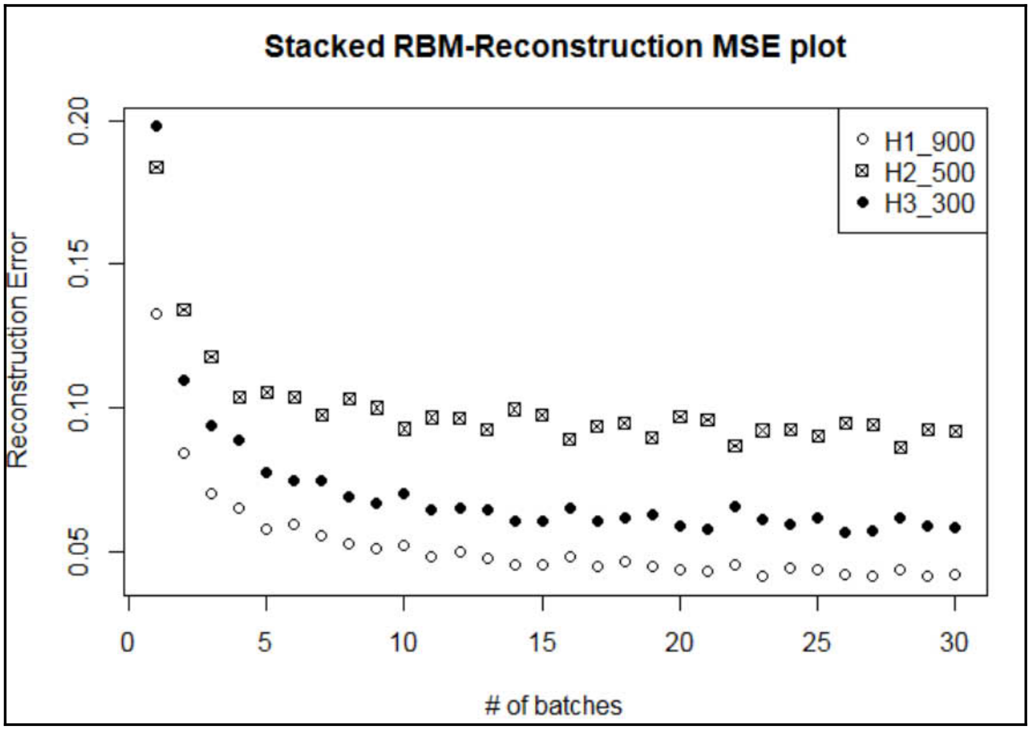
5. Plot reconstruction mean squared errors:

```
plot(error ~ batches,
xlab = "# of batches",
ylab = "Reconstruction Error",
pch = c(1, 7, 16)[hidden_layer],
main = "Stacked RBM-Reconstruction MSE plot",
data = error_df)
legend('topright',
c("H1_900","H2_500","H3_300"),
pch = c(1, 7, 16))
```

How it works...

Assessing the performance of training three stacked RBMs: Here, we will run five epochs (or iterations) for each RBM. Each epoch will perform batchwise (size = 100) optimization. In each batch, CD is computed and, accordingly, weights and biases are updated.

To keep a track of optimization, the MSE is calculated after every batch of 10,000 rows. The following image shows the declining trend of mean squared reconstruction errors computed for 30 batches for three RBMs separately:



Implementing a feed-forward backpropagation Neural Network

In this recipe, we will implement a willow neural network with backpropagation. The input of the neural network is the outcome of the third (or last) RBM. In other words, the reconstructed raw data (`trainX`) is actually used to train the neural network as a supervised classifier of (10) digits. The backpropagation technique is used to further fine-tune the performance of classification.

Getting ready

This section provides the requirements for TensorFlow.

- The dataset is loaded and set up
- The `TensorFlow` package is set up and loaded

How to do it...

This section covers the steps for setting up a feed-forward backpropagation Neural Network:

1. Let's define the input parameters of the neural network as function parameters. The following table describes each parameter:

Parameter name	Parameter description
Xdata	Matrix of train input MNIST data
Ydata	Matrix of train output MNIST data
Xtestdata	Matrix of test input MNIST data
Ytestdata	Matrix of test output MNIST data
input_size	Number of attributes (or pixels) in train data
learning_rate	Learning rate for updating weight matrix
momentum	Increment in size of steps to jump out of local minima
epochs	Number of iterations
batchsize	Number of observations per batch run
rbm_list	List of outcomes of stacked RBM
dbn_sizes	Vector of sizes of hidden layers in stacked RBM

The neural network function will have a structure as shown in the following script:

```
NN_train <- function(Xdata,Ydata,Xtestdata,Ytestdata,input_size,
learning_rate=0.1,momentum = 0.1,epochs=10,
batchsize=100,rbm_list,dbn_sizes){
library(stringi)
## insert all the codes mentioned in next 11 points
}
```

2. Initialize a weight and bias list of length 4, with the first being a tensor of random normal distribution (with a standard deviation of 0.01) of dimensions 784 x 900, the second being 900 x 500, the third being 500 x 300, and the fourth being 300 x 10:

```
weight_list <- list()
bias_list <- list()
# Initialize variables
for(size in c(dbn_sizes,ncol(Ydata))){
#Initialize weights through a random uniform distribution
weight_list <-
c(weight_list,tf$random_normal(shape=shape(input_size, size),
stddev=0.01, dtype=tf$float32))
#Initialize bias as zeroes
bias_list <- c(bias_list, tf$zeros(shape = shape(size),
dtype=tf$float32))
input_size = size
}
```

3. Check whether the outcome of the stacked RBM conforms to the sizes of the hidden layers mentioned in the `dbn_sizes` parameter:

```
#Check if expected dbn_sizes are correct
if(length(dbn_sizes)!=length(rbm_list)){
stop("number of hidden dbn_sizes not equal to number of rbm outputs
generated")
# check if expected sized are correct
for(i in 1:length(dbn_sizes)){
if(dbn_sizes[i] != dbn_sizes[i])
stop("Number of hidden dbn_sizes do not match")
}
}
```

4. Now, place the weights and biases in suitable positions within `weight_list` and `bias_list`:

```
for(i in 1:length(dbn_sizes)){
weight_list[[i]] <- rbm_list[[i]]$weight_final
bias_list[[i]] <- rbm_list[[i]]$bias_final
}
```

5. Create placeholders for the input and output data:

```
input <- tf$placeholder(tf$float32, shape =
shape(NULL,ncol(Xdata)))
output <- tf$placeholder(tf$float32, shape =
shape(NULL,ncol(Ydata)))
```

6. Now, use the weights and biases obtained from the stacked RBM to reconstruct the input data and store each RBM's reconstructed data in the list `input_sub`:

```
input_sub <- list()
weight <- list()
bias <- list()
for(i in 1:(length(dbn_sizes)+1)){
weight[[i]] <- tf$cast(tf$Variable(weight_list[[i]]),tf$float32)
bias[[i]] <- tf$cast(tf$Variable(bias_list[[i]]),tf$float32)
}
input_sub[[1]] <- tf$nn$sigmoid(tf$matmul(input, weight[[1]]) +
bias[[1]])
for(i in 2:(length(dbn_sizes)+1)){
input_sub[[i]] <- tf$nn$sigmoid(tf$matmul(input_sub[[i-1]],
weight[[i]]) + bias[[i]])
}
```

7. Define the cost function--that is, the mean squared error of difference between prediction and actual digits:

```
cost = tf$reduce_mean(tf$square(input_sub[[length(input_sub)]] -
output))
```

8. Implement backpropagation for the purpose of minimizing the cost:

```
train_op <- tf$train$MomentumOptimizer(learning_rate,
momentum)$minimize(cost)
```

9. Generate the prediction results:

```
predict_op =
tf$argmax(input_sub[[length(input_sub)]],axis=tf$cast(1.0,tf$int32)
)
```

10. Perform iterations of training:

```
train_accuracy <- c()
test_accuracy <- c()
for(ep in 1:epochs){
for(i in seq(0,(dim(Xdata)[1]-batchsize),batchsize)){
```

```
batchX <- Xdata[(i+1):(i+batchsize),]
batchY <- Ydata[(i+1):(i+batchsize),]
#Run the training operation on the input data
sess$run(train_op,feed_dict=dict(input = batchX,
output = batchY))
}
for(j in 1:(length(dbn_sizes)+1)){
# Retrieve weights and biases
weight_list[[j]] <- sess$run(weight[[j]])
bias_list[[j]] <- sess$ run(bias[[j]])
}
train_result <- sess$run(predict_op, feed_dict = dict(input=Xdata,
output=Ydata))+1
train_actual <-
as.numeric(stringi::stri_sub(colnames(as.data.frame(Ydata))[max.col
(as.data.frame(Ydata),ties.method="first")],2))
test_result <- sess$run(predict_op, feed_dict =
dict(input=Xtestdata, output=Ytestdata))+1
test_actual <-
as.numeric(stringi::stri_sub(colnames(as.data.frame(Ytestdata))[max
.col(as.data.frame(Ytestdata),ties.method="first")],2))
train_accuracy <-
c(train_accuracy,mean(train_actual==train_result))
test_accuracy <- c(test_accuracy,mean(test_actual==test_result))
cat("epoch:", ep, " Train Accuracy: ",train_accuracy[ep]," Test
Accuracy : ",test_accuracy[ep],"\n")
}
```

11. Finally, return a list of four outcomes, which are train accuracy (`train_accuracy`), test accuracy (`test_accuracy`), a list of weight matrices generated in each iteration (`weight_list`), and a list of bias vectors generated in each iteration (`bias_list`):

```
return(list(train_accuracy=train_accuracy,
test_accuracy=test_accuracy,
weight_list=weight_list,
bias_list=bias_list))
```

12. Run the iterations for the defined neural network for training:

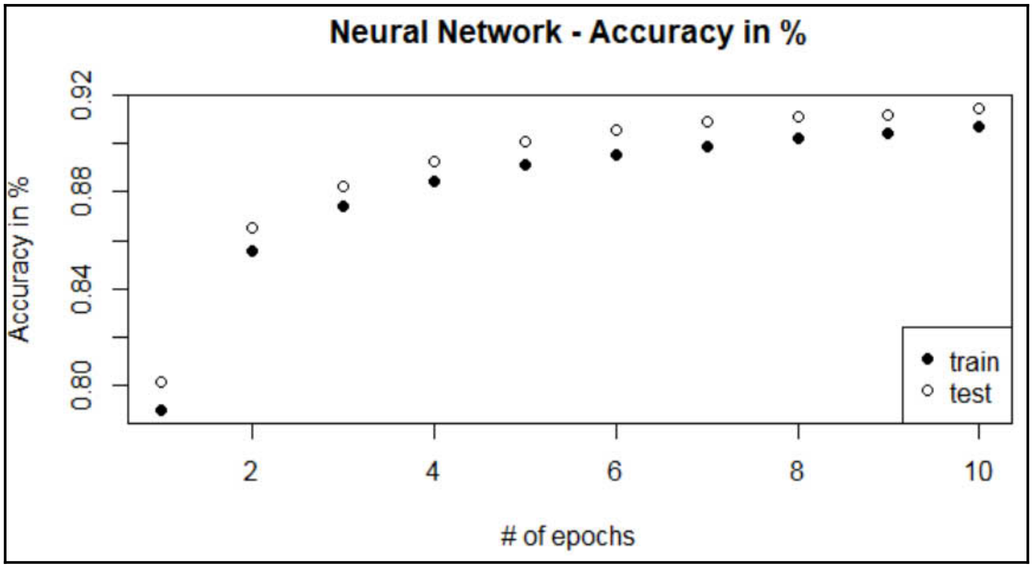
```
NN_results <- NN_train(Xdata=trainX,
Ydata=trainY,
Xtestdata=testX,
Ytestdata=testY,
input_size=ncol(trainX),
rbm_list=RBM_output,
dbn_sizes = RBM_hidden_sizes)
```

13. The following code is used to plot the train and test accuracy:

```
accuracy_df <-
data.frame("accuracy"=c(NN_results$train_accuracy,NN_results$test_a
ccuracy),
"epochs"=c(rep(1:10,times=2)),
"datatype"=c(rep(c(1,2),each=10)),
stringsAsFactors = FALSE)
plot(accuracy ~ epochs,
xlab = "# of epochs",
ylab = "Accuracy in %",
pch = c(16, 1)[datatype],
main = "Neural Network - Accuracy in %",
data = accuracy_df)
legend('bottomright',
c("train","test"),
pch = c( 16, 1))
```

How it works...

Assessing the train and test performance of the neural network: The following image shows the increasing trend of train and test accuracy observed while training the neural network:



Setting up a Deep Restricted Boltzmann Machine

Unlike DBNs, **Deep Restricted Boltzmann Machines** (**DRBM**) are undirected networks of interconnected hidden layers with the capability to learn joint probabilities over these connections. In the current setup, centering is performed where visible and hidden variables are subtracted from offset bias vectors after every iteration. Research has shown that centering optimizes the performance of DRBMs and can reach higher log-likelihood values in comparison with traditional RBMs.

Getting ready

This section provides the requirements for setting up a DRBM:

- The `MNIST` dataset is loaded and set up
- The `tensorflow` package is set up and loaded

How to do it...

This section covers detailed the steps for setting up the DRBM model using TensorFlow in R:

1. Define the parameters for the DRBM:

```
learning_rate = 0.005
momentum = 0.005
minbatch_size = 25
hidden_layers = c(400,100)
biases = list(-1,-1)
```

2. Define a sigmoid function using a hyperbolic arc tangent *[(log(1+x) -log(1-x))/2]*:

```
arcsigm <- function(x){
return(atanh((2*x)-1)*2)
}
```

3. Define a sigmoid function using only a hyperbolic tangent *[$(e^x-e^{-x})/(e^x+e^{-x})$]*:

```
sigm <- function(x){
return(tanh((x/2)+1)/2)
}
```

4. Define a `binarize` function to return a matrix of binary values (0,1):

```
binarize <- function(x){
# truncated rnorm
trnrom <- function(n, mean, sd, minval = -Inf, maxval = Inf){
qnorm(runif(n, pnorm(minval, mean, sd), pnorm(maxval, mean, sd)),
mean, sd)
}
return((x > matrix(
trnrom(n=nrow(x)*ncol(x),mean=0,sd=1,minval=0,maxval=1), nrow(x),
ncol(x)))*1)
}
```

5. Define a `re_construct` function to return a matrix of pixels:

```
re_construct <- function(x){
x = x - min(x) + 1e-9
x = x / (max(x) + 1e-9)
return(x*255)
}
```

6. Define a function to perform `gibbs` activation for a given layer:

```
gibbs <- function(X,l,initials){
if(l>1){
bu <- (X[l-1][[1]] -
matrix(rep(initials$param_O[[l-1]],minbatch_size),minbatch_size,byr
ow=TRUE)) %*%
initials$param_W[l-1][[1]]
} else {
bu <- 0
}
if((l+1) < length(X)){
td <- (X[l+1][[1]] -
matrix(rep(initials$param_O[[l+1]],minbatch_size),minbatch_size,byr
ow=TRUE))%*%
t(initials$param_W[l][[1]])
} else {
td <- 0
}
X[[l]] <-
binarize(sigm(bu+td+matrix(rep(initials$param_B[[l]],minbatch_size)
,minbatch_size,byrow=TRUE)))
return(X[[l]])
}
```

7. Define a function to perform the reparameterization of bias vectors:

```
reparamBias <- function(X,l,initials){
if(l>1){
bu <- colMeans((X[[l-1]] -
matrix(rep(initials$param_O[[l-1]],minbatch_size),minbatch_size,byr
ow=TRUE))%*%
initials$param_W[[l-1]])
} else {
bu <- 0
}
if((l+1) < length(X)){
td <- colMeans((X[[l+1]] -
matrix(rep(initials$param_O[[l+1]],minbatch_size),minbatch_size,byr
ow=TRUE))%*%
t(initials$param_W[[l]]))
} else {
td <- 0
}
initials$param_B[[l]] <- (1-momentum)*initials$param_B[[l]] +
momentum*(initials$param_B[[l]] + bu + td)
return(initials$param_B[[l]])
}
```

8. Define a function to perform the reparameterization of offset bias vectors:

```
reparamO <- function(X,l,initials){
initials$param_O[[l]] <- colMeans((1-
momentum)*matrix(rep(initials$param_O[[l]],minbatch_size),minbatch_
size,byrow=TRUE) + momentum*(X[[l]]))
return(initials$param_O[[l]])
}
```

9. Define a function to initialize weights, biases, offset biases, and input matrices:

```
DRBM_initialize <- function(layers,bias_list){
# Initialize model parameters and particles
param_W <- list()
for(i in 1:(length(layers)-1)){
param_W[[i]] <- matrix(0L, nrow=layers[i], ncol=layers[i+1])
}
param_B <- list()
for(i in 1:length(layers)){
param_B[[i]] <- matrix(0L, nrow=layers[i], ncol=1) + bias_list[[i]]
}
param_O <- list()
for(i in 1:length(param_B)){
param_O[[i]] <- sigm(param_B[[i]])
}
param_X <- list()
for(i in 1:length(layers)){
param_X[[i]] <- matrix(0L, nrow=minbatch_size, ncol=layers[i]) +
matrix(rep(param_O[[i]],minbatch_size),minbatch_size,byrow=TRUE)
}
return(list(param_W=param_W,param_B=param_B,param_O=param_O,param_X
=param_X))
}
```

10. Use the MNIST train data (`trainX`) introduced in the previous recipes. Standardize the `trainX` data by dividing it by 255:

```
X <- trainX/255
```

11. Generate the initial weight matrices, bias vectors, offset bias vectors, and input matrices:

```
layers <- c(784,hidden_layers)
bias_list <-
list(arcsigm(pmax(colMeans(X),0.001)),biases[[1]],biases[[2]])
initials <-DRBM_initialize(layers,bias_list)
```

12. Subset a sample (`minbatch_size`) of the input data `X`:

```
batchX <- X[sample(nrow(X))[1:minbatch_size],]
```

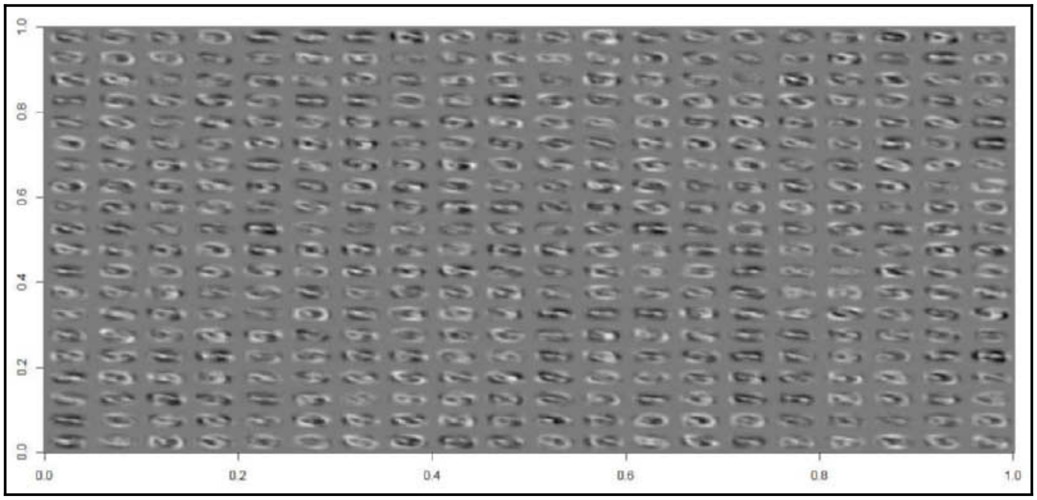
13. Perform a set of 1,000 iterations. Within each iteration, update the initial weights and biases 100 times and plot the images of the weight matrices:

```
for(iter in 1:1000){

# Perform some learnings
for(j in 1:100){
# Initialize a data particle
dat <- list()
dat[[1]] <- binarize(batchX)
for(l in 2:length(initials$param_X)){
dat[[l]] <- initials$param_X[l][[1]]*0 +
matrix(rep(initials$param_O[l][[1]],minbatch_size),minbatch_size,by
row=TRUE)
}

# Alternate gibbs sampler on data and free particles
for(l in rep(c(seq(2,length(initials$param_X),2),
seq(3,length(initials$param_X),2)),5)){
dat[[l]] <- gibbs(dat,l,initials)
}
for(l in rep(c(seq(2,length(initials$param_X),2),
seq(1,length(initials$param_X),2)),1)){
initials$param_X[[l]] <- gibbs(initials$param_X,l,initials)
}

# Parameter update
for(i in 1:length(initials$param_W)){
initials$param_W[[i]] <- initials$param_W[[i]] +
(learning_rate*((t(dat[[i]] -
matrix(rep(initials$param_O[i][[1]],minbatch_size),minbatch_size,by
row=TRUE)) %*%
(dat[[i+1]] -
matrix(rep(initials$param_O[i+1][[1]],minbatch_size),minbatch_size,
byrow=TRUE))) -
(t(initials$param_X[[i]] -
matrix(rep(initials$param_O[i][[1]],minbatch_size),minbatch_size,by
row=TRUE)) %*%
(initials$param_X[[i+1]] -
matrix(rep(initials$param_O[i+1][[1]],minbatch_size),minbatch_size,
byrow=TRUE))))/nrow(batchX))
}

for(i in 1:length(initials$param_B)){
initials$param_B[[i]] <-
colMeans(matrix(rep(initials$param_B[[i]],minbatch_size),minbatch_s
ize,byrow=TRUE) + (learning_rate*(dat[[i]] -
initials$param_X[[i]])))
}

# Reparameterization
for(l in 1:length(initials$param_B)){
initials$param_B[[l]] <- reparamBias(dat,l,initials)
}
for(l in 1:length(initials$param_O)){
initials$param_O[[l]] <- reparamO(dat,l,initials)
}
}

# Generate necessary outputs
cat("Iteration:",iter," ","Mean of W of VL-
HL1:",mean(initials$param_W[[1]])," ","Mean of W of HL1-
HL2:",mean(initials$param_W[[2]]) ,"\n")
cat("Iteration:",iter," ","SDev of W of VL-
HL1:",sd(initials$param_W[[1]])," ","SDev of W of HL1-
HL2:",sd(initials$param_W[[2]]) ,"\n")

# Plot weight matrices
```

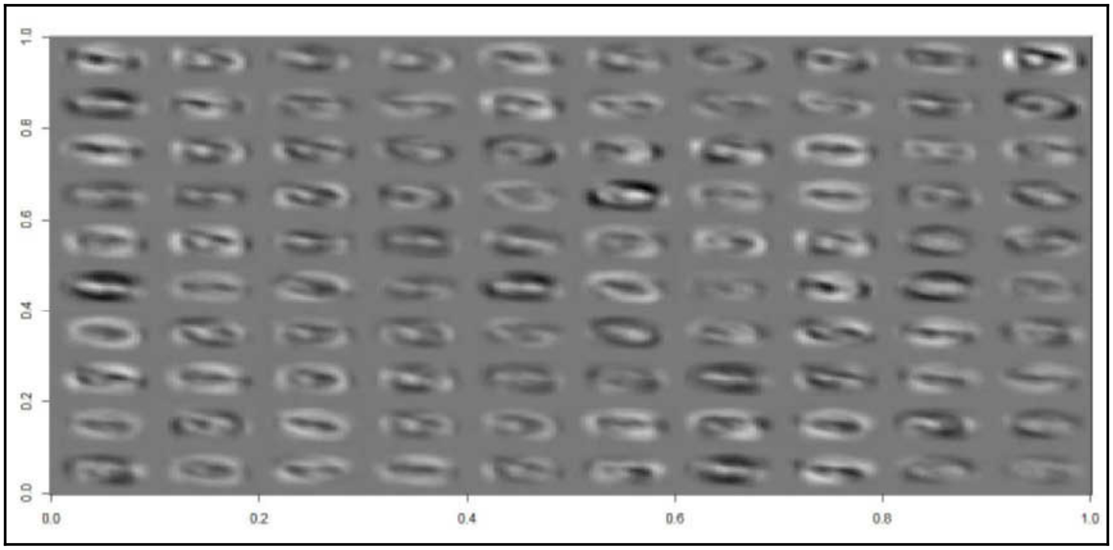
```
W=diag(nrow(initials$param_W[[1]]))
for(l in 1:length(initials$param_W)){
W = W %*% initials$param_W[[l]]
m = dim(W)[2] * 0.05
w1_arr <- matrix(0,28*m,28*m)
i=1
for(k in 1:m){
for(j in 1:28){
vec <- c(W[(28*j-28+1):(28*j),(k*m-m+1):(k*m)])
w1_arr[i,] <- vec
i=i+1
}
}
w1_arr = re_construct(w1_arr)
w1_arr <- floor(w1_arr)
image(w1_arr,axes = TRUE, col = grey(seq(0, 1, length = 256)))
}
}
```

How it works...

As the preceding DRBM is trained using two hidden layers, we generate two weight matrices. The first weight matrix defines the connection between the visible layer and the first hidden layer. The second weight matrix defines the connection between the first and second hidden layer. The following image shows pixel images of the first weight matrices:



The following image shows the second pixel images of the second weight matrix:


0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0