4
Data Representation Using Autoencoders

This chapter will introduce unsupervised applications of deep learning using autoencoders. In this chapter, we will cover the following topics:

- Setting up autoencoders
- Data normalization
- Setting up a regularized autoencoder
- Fine-tuning the parameters of the autoencoder
- Setting up stacked autoencoders
- Setting up denoising autoencoders
- Building and comparing stochastic encoders and decoders
- Learning manifolds from autoencoders
- Evaluating the sparse decomposition

Introduction

Neural networks aim to find a non-linear relationship between input X with output y, as $y=f(x)$**.** An autoencoder is a form of unsupervised neural network which tries to find a relationship between features in space such that $h=f(x)$, which helps us learn the relationship between input space and can be used for data compression, dimensionality reduction, and feature learning.

An autoencoder consists of an encoder and decoder. The encoder helps encode the input x in a latent representation y, whereas a decoder converts back the y to x. Both the encoder and decoder possess a similar representation of form.

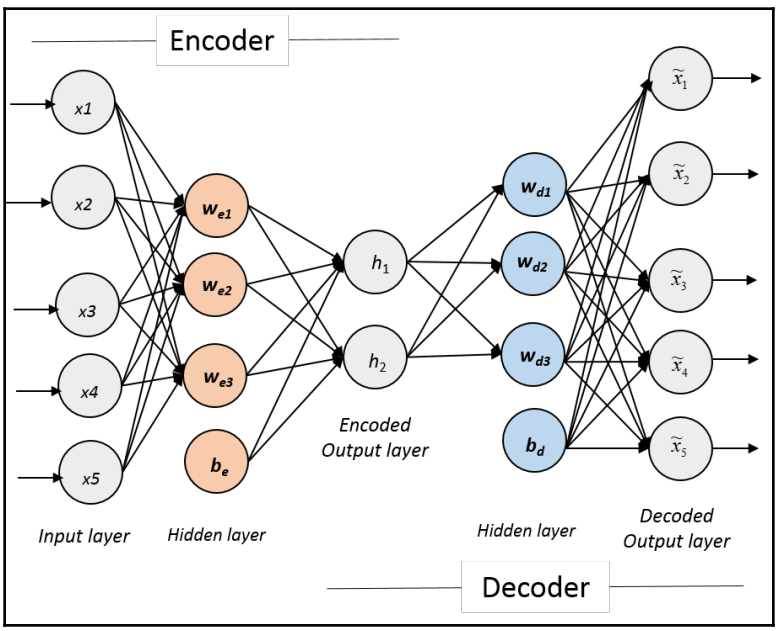
Here is a representation of a one layer autoencoder:

$$\boldsymbol{h = f(x) = \sigma(W_e^T X + b_e)}$$

$$\boldsymbol{\tilde{X} = f(h) = \sigma(W_d^T h + b_d)}$$

The coder encodes input X to h under a hidden layer contain, whereas the decoder helps to attain the original data from encoded output h. The matrices W_e and W_d represent the weights of the encoder and decoder layers, respectively. The function f is the activation function.

An illustration of an autoencoder is shown in the following diagram:



The constraints in the form of nodes allows the autoencoder to discover interesting structures within the data. For example, in the encoder in the preceding diagram, the five-input dataset must pass through three-node compression to get an encoded value *h.* The **Encoded Output layer** of an encoder can have a dimensionality that is the same, lower, or higher than the **input/output Decoded Output layer**. The **Encoded Output layer** with a fewer number of nodes than the input layer is referred to as an under-complete representation, and can be thought of as data compression transforming data into low-dimensional representation.

An **Encoded Output layer** with a larger number of input layers is referred to as an over-complete representation and is used in a **sparse autoencoder** as a regularization strategy. The objective of an autoencoder is to find *y*, capturing the main factors along the variation of data, which is similar to **Principal Component Analysis (PCA)**, and thus can be used for compression as well.

Setting up autoencoders

There exist a lot of different architectures of autoencoders distinguished by cost functions used to capture data representation. The most basic autoencoder is known as a vanilla autoencoder. It's a two-layer neural network with one hidden layer the same number of nodes at the input and output layers, with an objective to minimize the cost function. The typical choices, but not limited to, for a loss function are **mean square error** (**MSE**) for regression and cross entropy for classification. The current approach can be easily extended to multiple layers, also known as multilayer autoencoder.

The number of nodes plays a very critical role in autoencoders. If the number of nodes in the hidden layer is less than the input layer then an autoencoder is known as an **under-complete** autoencoder. A higher number of nodes in the hidden layer represents an **over-complete** autoencoder or sparse autoencoder.

The sparse autoencoder aims to impose sparsity in the hidden layer. This sparsity can be achieved by introducing a higher number of nodes than the input in the hidden layer or by introducing a penalty in the loss function that will move the weights for the hidden layer toward zero. Some autoencoders attain the sparsity by manually zeroing out the weight for nodes; these are referred to as **K-sparse autoencoders**. We will set up an autoencoder on the occupancy dataset discussed in `Chapter 1`, *Getting Started*. The hidden layer for the current example can be tweaked around.

Getting ready

Let's use the `Occupancy` dataset to set up an autoencoder:

- Download the `Occupancy` dataset as described in `Chapter 1`, *Getting Started*
- TensorFlow installation in R and Python

How to do it...

The current occupancy dataset as described in `Chapter 1`, *Getting Started*, is used to demonstrate the autoencoder setup in R using TensorFlow:

1. Set up the R TensorFlow environment.
2. The `load_occupancy_data` function can be used to load the data by setting the correct working directory path using `setwd`:

```
# Function to load Occupancy data
load_occupancy_data<-function(train){
xFeatures = c("Temperature", "Humidity", "Light", "CO2",
  "HumidityRatio")
yFeatures = "Occupancy"
  if(train){
    occupancy_ds <-
as.matrix(read.csv("datatraining.txt",stringsAsFactors = T))
  } else
  {
    occupancy_ds <-
as.matrix(read.csv("datatest.txt",stringsAsFactors = T))
  }
  occupancy_ds<-apply(occupancy_ds[, c(xFeatures, yFeatures)], 2,
FUN=as.numeric)
  return(occupancy_ds)
}
```

3. The train and test occupancy dataset can be loaded to the R environment with the following script:

```
occupancy_train <-load_occupancy_data(train=T)
occupancy_test <- load_occupancy_data(train = F)
```

Data normalization

Data normalization is a critical step in machine learning to bring data to a similar scale. It is also known as feature scaling and is performed as data preprocessing.

The correct normalization is very critical in neural networks, else it will lead to saturation within the hidden layers, which in turn leads to zero gradient and no learning will be possible.

Getting ready

There are multiple ways to perform normalization:

- **Min-max standardization**: The min-max retains the original distribution and scales the feature values between *[0, 1]*, with *0* as the minimum value of the feature and *1* as the maximum value. The standardization is performed as follows:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

  Here, x' is the normalized value of the feature. The method is sensitive to outliers in the dataset.

- **Decimal scaling**: This form of scaling is used where values of different decimal ranges are present. For example, two features with different bounds can be brought to a similar scale using decimal scaling as follows:

$$x'=x/10^n$$

- **Z-score:** This transformation scales the value toward a normal distribution with a zero mean and unit variance. The Z-score is computed as:

$$Z=(x-\mu)/\sigma$$

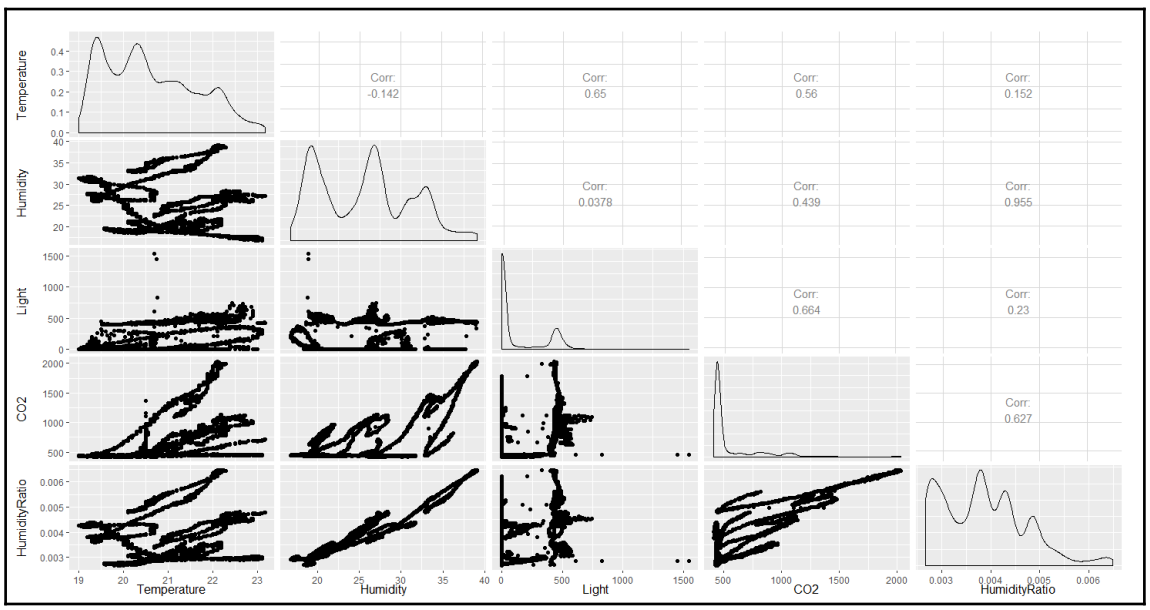
Here, μ is the mean and σ is the standard deviation of the feature. These distributions are very efficient for a dataset with a Gaussian distribution.

All the preceding methods are sensitive to outliers; there are other more robust approaches for normalization that you can explore, such as **Median Absolute Deviation** (**MAD**), tanh-estimator, and double sigmoid.

Visualizing dataset distribution

Let's look at the distribution of features for the occupation data:

```
> ggpairs(occupancy_train$data[, occupancy_train$xFeatures])
```



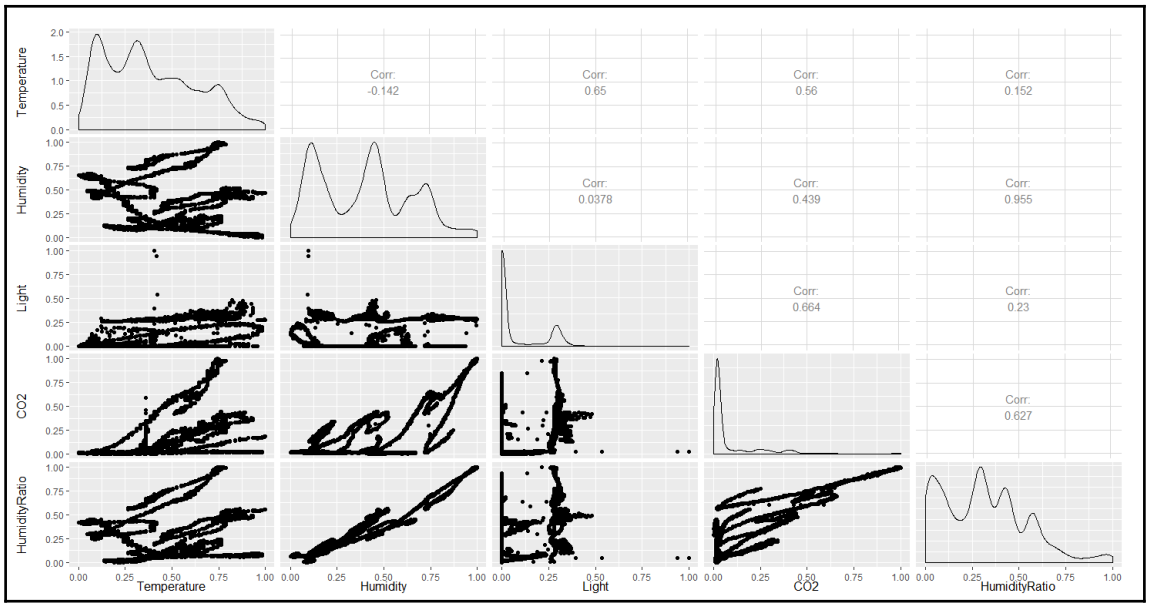
The figure shows that the features have linear correlations and the distributions are non-normal. The non-normality can be further validated using the Shapiro-Wilk test, using the `shapiro.test` function from R. Let's use min-max standardization for the occupation data.

How to do it...

1. Perform the following operation for data normalization:

```
minmax.normalize<-function(ds, scaler=NULL){
  if(is.null(scaler)){
    for(f in ds$xFeatures){
      scaler[[f]]$minval<-min(ds$data[,f])
      scaler[[f]]$maxval<-max(ds$data[,f])
      ds$data[,f]<-(ds$data[,f]-
scaler[[f]]$minval)/(scaler[[f]]$maxval-scaler[[f]]$minval)
    }
    ds$scaler<-scaler
  } else
  {
    for(f in ds$xFeatures){
      ds$data[,f]<-(ds$data[,f]-
scaler[[f]]$minval)/(scaler[[f]]$maxval-scaler[[f]]$minval)
    }
  }
  return(ds)
}
```

2. The `minmax.normalize` function normalizes the data using min-max normalization. When the `scaler` variable is `NULL`, it performs normalization using the dataset provided, or normalizes using `scaler` values. The normalized data pair plot is shown in the following figure:



This figure shows min-max normalization bringing the values within bounds *[0, 1]* and it does not change the distribution and correlations between features.

How to set up an autoencoder model

The next step is to set up the autoencoder model. Let's set up a vanilla autoencoder using TensorFlow:

1. Reset the `graph` and start `InteractiveSession`:

```
# Reset the graph and set-up a interactive session
tf$reset_default_graph()
sess<-tf$InteractiveSession()
```

2. Define the input parameter where `n` and `m` are the number of samples and features, respectively. To build, network `m` is used to set up the input parameter:

```
# Network Parameters
n_hidden_1 = 5 # 1st layer num features
n_input = length(xFeatures) # Number of input features
nRow<-nrow(occupancy_train)
```

   When `n_hidden_1` is low, the autoencoder is compressing the data and is referred to as an under-complete autoencoder; whereas, when `n_hidden_1` is large, then the autoencoder is sparse and is referred to as an over-complete autoencoder.

3. Define graph input parameters that include the input tensor and layer definitions for the encoder and decoder:

```
# Define input feature
x <- tf$constant(unlist(occupancy_train[, xFeatures]),
shape=c(nRow, n_input), dtype=np$float32)

# Define hidden and bias layer for encoder and decoders
hiddenLayerEncoder<-tf$Variable(tf$random_normal(shape(n_input,
n_hidden_1)), dtype=np$float32)
biasEncoder <- tf$Variable(tf$zeros(shape(n_hidden_1)),
dtype=np$float32)
hiddenLayerDecoder<-tf$Variable(tf$random_normal(shape(n_hidden_1,
n_input)))
biasDecoder <- tf$Variable(tf$zeros(shape(n_input)))
```

   The preceding script designs a single-layer encoder and decoder.

4. Define a function to evaluate the response:

```
auto_encoder<-function(x, hiddenLayerEncoder, biasEncoder){
  x_transform <- tf$nn$sigmoid(tf$add(tf$matmul(x,
hiddenLayerEncoder), biasEncoder))
  x_transform
}
```

   The `auto_encoder` function takes the node bias weights and computes the output. The same function can be used for `encoder` and `decoder` by passing respective weights.

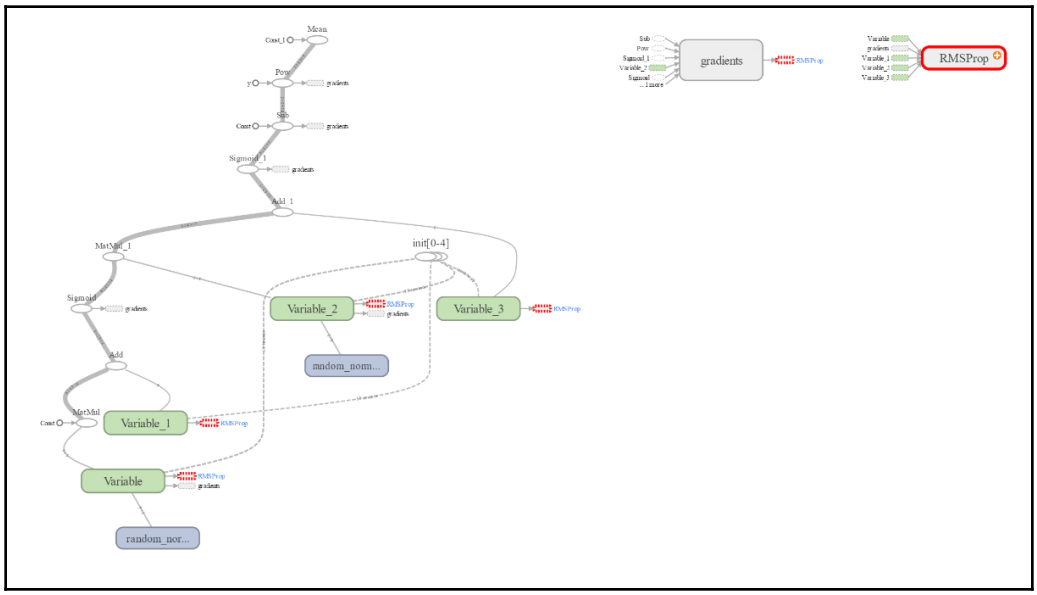
5. Create `encoder` and `decoder` objects by passing symbolic TensorFlow variables:

```
encoder_obj = auto_encoder(x,hiddenLayerEncoder, biasEncoder)
y_pred = auto_encoder(encoder_obj, hiddenLayerDecoder, biasDecoder)
```

6. The `y_pred` is the outcome from `decoder`, which takes the `encoder` object as input with nodes and bias weights:

```
Define loss function and optimizer module.
learning_rate = 0.01
cost = tf$reduce_mean(tf$pow(x - y_pred, 2))
optimizer = tf$train$RMSPropOptimizer(learning_rate)$minimize(cost)
```

The preceding script defines mean square error as the cost function, and uses `RMSPropOptimizer` from TensorFlow with 0.1 learning rate for the optimization of weights. The TensorFlow graph for the preceding model is shown in the following diagram:



Running optimization

The next step is to run optimizer optimization. Executing this process in TensorFlow consists of two steps:

1. The first step is parameter initialization of the variables defined in the graph. The initialization is performed by calling the `global_variables_initializer` function from TensorFlow:

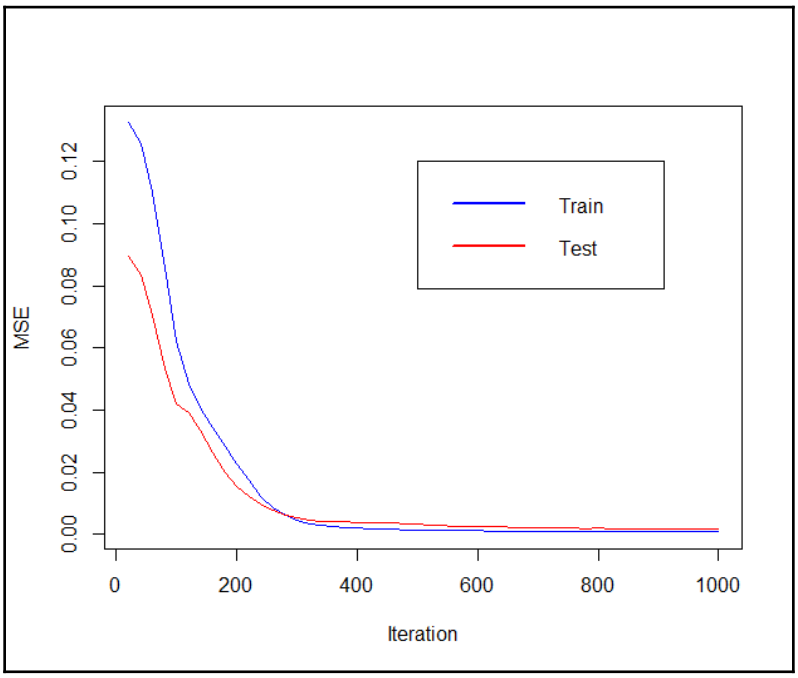
```
# Initializing the variables
init = tf$global_variables_initializer()
sess$run(init)
```

   Optimization is performed based on optimizing and monitoring the train and test performance:

```
costconvergence<-NULL
for (step in 1:1000) {
  sess$run(optimizer)
  if (step %% 20==0){
    costconvergence<-rbind(costconvergence, c(step, sess$run(cost),
sess$run(costt)))
    cat(step, "-", "Traing Cost ==>", sess$run(cost), "\n")
  }
}
```

2. The cost function from train and test can be observed to understand convergence of the model, as shown in the following figure:

```
costconvergence<-data.frame(costconvergence)
colnames(costconvergence)<-c("iter", "train", "test")
plot(costconvergence[, "iter"], costconvergence[, "train"], type =
"l", col="blue", xlab = "Iteration", ylab = "MSE")
lines(costconvergence[, "iter"], costconvergence[, "test"],
col="red")
legend(500,0.25, c("Train","Test"), lty=c(1,1),
lwd=c(2.5,2.5),col=c("blue","red"))
```



This graph shows that the model major `convergence` is at around **400** iterations; however, it is still converging at a very slow rate even after **1,000** iterations. The model is stable in both the train and holdout test datasets.

Setting up a regularized autoencoder

A regularized autoencoder extends the standard autoencoder by adding a regularization parameter to the `cost` function.

Getting ready

The regularized autoencoder is an extension of the standard autoencoder. The set-up will require:

1. TensorFlow installation in R and Python.
2. Implementation of a standard autoencoder.

How to do it...

The code setup for the autoencoder can directly be converted to a regularized autoencoder by replacing the cost definition with the following lines:

```
Lambda=0.01
cost = tf$reduce_mean(tf$pow(x - y_pred, 2))
Regularize_weights = tf$nn$l2_loss(weights)
cost = tf$reduce_mean(cost + lambda * Regularize_weights)
```

How it works...

As mentioned earlier, a regularized autoencoder extends the standard autoencoder by adding a regularization parameter to the cost function, shown as follows:

$$\sum L(x,\tilde{x}) + \lambda \left\| W_{ij}^2 \right\|$$

Here, λ is the regularization parameter and i and j are the node indexes with W representing the hidden layer weights for the autoencoder. The regularization autoencoder aims to ensure more robust encoding and prefers a low weight h function. The concept is further utilized to develop a contractive autoencoder, which utilizes the Frobenius norm of the Jacobian matrix on input, represented as follows:

$$L(\mathbf{X},\tilde{\mathbf{X}}) + \lambda \left\| J(x) \right\|^2$$

where **J(x)** is the Jacobian matrix and is evaluated as follows:

$$\left\| J(x) \right\|_F^2 = \sum_{ij} \frac{\partial h_j(x)}{\partial x_i}$$

For a linear encoder, a contractive encoder and regularized encoder converge to L2 weight decay. The regularization helps in making the autoencoder less sensitive to the input; however, the minimization of the cost function helps the model to capture the variation and remain sensitive to manifolds of high density. These autoencoders are also referred to as **contractive autoencoders**.

Fine-tuning the parameters of the autoencoder

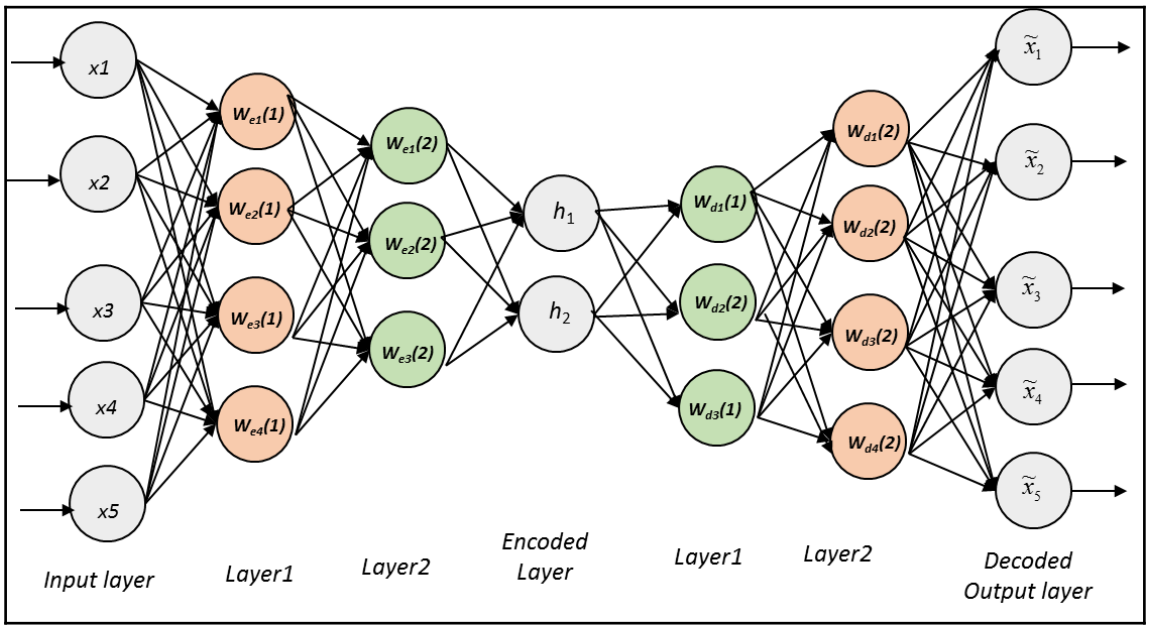
The autoencoder involves a couple of parameters to tune, depending on the type of autoencoder we are working on. The major parameters in an autoencoder include the following:

- Number of nodes in any hidden layer
- Number of hidden layers applicable for deep autoencoders
- Activation unit such as sigmoid, tanh, softmax, and ReLU activation functions
- Regularization parameters or weight decay terms on hidden unit weights
- Fraction of the signal to be corrupted in a denoising autoencoder
- Sparsity parameters in sparse autoencoders that control the expected activation of neurons in hidden layers
- Batch size, if using batch gradient descent learning; learning rate and momentum parameter for stochastic gradient descent
- Maximum iterations to be used for the training
- Weight initialization
- Dropout regularization if dropout is used

These hyperparameters can be trained by setting the problem as a grid search problem. However, each hyperparameter combination requires training the neuron weights for the hidden layer(s), which results in increasing computational complexity with an increase in the number of layers and number of nodes within each layer. To deal with these critical parameters and training issues, stacked autoencoder concepts have been proposed that train each layer separately to get pretrained weights, and then the model is fine-tuned using the obtained weights. This approach tremendously improves the training performance over the conventional mode of training.

Setting up stacked autoencoders

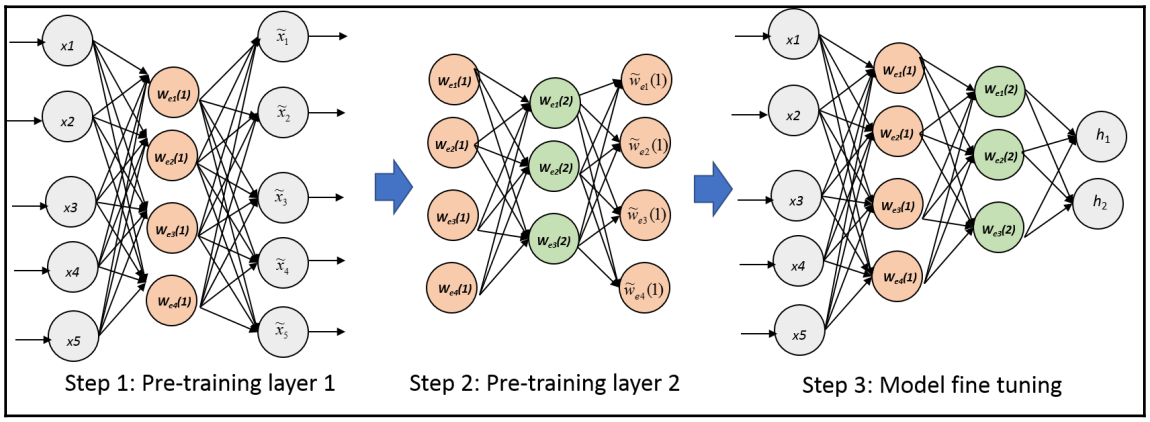
The stacked autoencoder is an approach to train deep networks consisting of multiple layers trained using the greedy approach. An example of a stacked autoencoder is shown in the following diagram:



An example of a stacked autoencoder

Getting ready

The preceding diagram demonstrates a stacked autoencoder with two layers. A stacked autoencoder can have *n* layers, where each layer is trained using one layer at a time. For example, the previous layer will be trained as follows:



Training of a stacked autoencoder

The initial pre-training of layer 1 is obtained by training it over the actual input x_i . The first step is to optimize the *We(1)* layer of the encoder with respect to output X. The second step in the preceding example is to optimize the weights *We(2)* in the second layer, using *We(1)* as input and output. Once all the layers of *We(i)* where *i*=1, 2, ...,*n* is number of layers are pretrained, model fine-tuning is performed by connecting all the layers together, as shown in step 3 of the preceding diagram. The concept can also be applied to denoising to train multilayer networks, which is known as a stacked denoising autoencoder. The code developed in a denoising autoencoder can be easily tweaked to develop a **stacked denoising autoencoder**, which is an extension of the stacked autoencoder.

The requirements for this recipe are:

1. R should be installed.
2. `SAENET` package, The package can be download from Cran using the command `install.packages("SAENET")`.

How to do it...

There are other popular libraries in R to develop stacked autoencoders. Let's utilize the `SAENET` package from R to set up a stacked autoencoder. The `SAENET` is a stacked autoencoder implementation, using a feedforward neural network using the `neuralnet` package from CRAN:

1. Get the `SAENET` package from the CRAN repository, if not installed already:

```
install.packages("SAENET")
```

2. Load all library dependencies:

```
require(SAENET)
```

3. Load the train and test occupancy dataset using `load_occupancy_data`:

```
occupancy_train <-load_occupancy_data(train=T)
occupancy_test <- load_occupancy_data(train = F)
```

4. Normalize the dataset using the `minmax.normalize` function:

```
# Normalize dataset
occupancy_train<-minmax.normalize(occupancy_train, scaler = NULL)
occupancy_test<-minmax.normalize(occupancy_test, scaler =
occupancy_train$scaler)
```

5. The stacked autoencoder model can be built using the `SAENET.train` train function from the `SAENET` package:

```
# Building Stacked Autoencoder
SAE_obj<-SAENET.train(X.train= subset(occupancy_train$data,
select=-c(Occupancy)), n.nodes=c(4, 3, 2), unit.type ="tanh",
lambda = 1e-5, beta = 1e-5, rho = 0.01, epsilon = 0.01,
max.iterations=1000)
```

The output of the last node can be extracted using the `SAE_obj[[n]]$X.output` command.

Setting up denoising autoencoders

Denoising autoencoders are a special kind of autoencoder with a focus on extracting robust features from the input dataset. Denoising autoencoders are similar to the previous model except with a major difference that the data is corrupted before training the network. Different approaches for corruption can be used such as masking, which induces random error into the data.

Getting ready

Let's use the CIFAR-10 image data to set up a denoising dataset:

- Download the CIFAR-10 dataset using the `download_cifar_data` function (covered in `Chapter 3`, *Convolution Neural Network*)
- TensorFlow installation in R and Python

How to do it...

We first need to read the dataset.

Reading the dataset

1. Load the `CIFAR` dataset using the steps explained in `Chapter 3`, *Convolution Neural Network*. The data files `data_batch_1` and `data_batch_2` are used to train. The `data_batch_5` and `test_batch` files are used for validation and testing, respectively. The data can be flattened using the `flat_data` function:

```
train_data <- flat_data(x_listdata = images.rgb.train)
test_data <- flat_data(x_listdata = images.rgb.test)
valid_data <- flat_data(x_listdata = images.rgb.valid)
```

2. The `flat_data` function flattens the dataset as *NCOL = (Height * Width * number of channels)*, thus the dimension of the dataset is (# of images X NCOL). The images in `CIFAR` are 32 x 32 with three RGB channels; thus, we obtain 3,072 columns after data flattening:

```
> dim(train_data$images)
[1] 40000  3072
```

Corrupting data to train

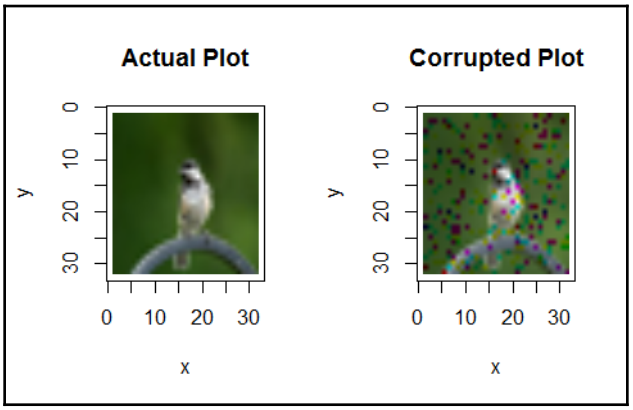
1. The next critical function needed to set up a denoising autoencoder is data corruption:

```
# Add noise using masking or salt & pepper noise method
add_noise<-function(data, frac=0.10, corr_type=c("masking",
"saltPepper", "none")){
  if(length(corr_type)>1) corr_type<-corr_type[1]
  # Assign a copy of data
  data_noise = data
  # Evaluate chaining parameters for autoencoder
  nROW<-nrow(data)
  nCOL<-ncol(data)
  nMask<-floor(frac*nCOL)
  if(corr_type=="masking"){
    for( i in 1:nROW){
      maskCol<-sample(nCOL, nMask)
      data_noise[i,maskCol,,]<-0
    }
  } else if(corr_type=="saltPepper"){
    minval<-min(data[,,1,])
    maxval<-max(data[,,1,])
    for( i in 1:nROW){
      maskCol<-sample(nCOL, nMask)
      randval<-runif(length(maskCol))
      ixmin<-randval<0.5
      ixmax<-randval>=0.5
      if(sum(ixmin)>0) data_noise[i,maskCol[ixmin],,]<-minval
      if(sum(ixmax)>0) data_noise[i,maskCol[ixmax],,]<-maxval
    }
  } else
  {
    data_noise<-data
  }
  return(data_noise)
}
```

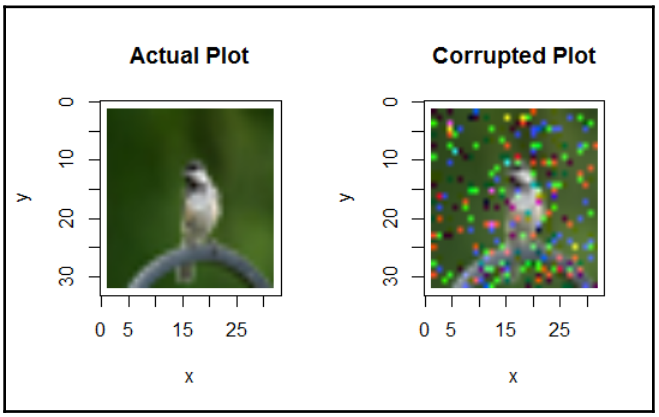
2. The CIFAR-10 data can be corrupted using the following script:

```
# Corrupting input signal
xcorr<-add_noise(train_data$images, frac=0.10, corr_type="masking")
```

3. An example image after corruption is as follows:



4. The preceding figure uses the masking approach to add noise. This method adds zero values at random image locations with a defined fraction. Another approach to add noise is by using salt & pepper noise. This method selects random locations in the image and replaces them, adding min or max values to the image using the flip of coin principle. An example of the salt and pepper approach for data corruption is shown in the following figure:



The data corruption helps the autoencoder to learn more robust representation.

Setting up a denoising autoencoder

The next step is to set up the autoencoder model:

1. First, reset the graph and start an interactive session as follows:

```
# Reset the graph and set-up an interactive session
tf$reset_default_graph()
sess<-tf$InteractiveSession()
```

2. The next step is to define two placeholders for the input signal and corrupt signal:

```
# Define Input as Placeholder variables
x = tf$placeholder(tf$float32, shape=shape(NULL, img_size_flat),
name='x')
x_corrput<-tf$placeholder(tf$float32, shape=shape(NULL,
img_size_flat), name='x_corrput')
```

   The `x_corrupt` will be used as input in the autoencoder, and *x* is the actual image that will be used as output.

3. Set up a denoising autoencoder function as shown in the following code:

```
# Setting-up denoising autoencoder
denoisingAutoencoder<-function(x, x_corrput, img_size_flat=3072,
hidden_layer=c(1024, 512), out_img_size=256){

  # Building Encoder
  encoder = NULL
  n_input<-img_size_flat
  curentInput<-x_corrput
  layer<-c(hidden_layer, out_img_size)
  for(i in 1:length(layer)){
    n_output<-layer[i]
    W = tf$Variable(tf$random_uniform(shape(n_input, n_output),
-1.0 / tf$sqrt(n_input), 1.0 / tf$sqrt(n_input)))
    b = tf$Variable(tf$zeros(shape(n_output)))
    encoder<-c(encoder, W)
    output = tf$nn$tanh(tf$matmul(curentInput, W) + b)
    curentInput = output
    n_input<-n_output
  }
  # latent representation
  z = curentInput
  encoder<-rev(encoder)
  layer_rev<-c(rev(hidden_layer), img_size_flat)
```

```
  # Build the decoder using the same weights
  decoder<-NULL
  for(i in 1:length(layer_rev)){
    n_output<-layer_rev[i]
    W = tf$transpose(encoder[[i]])
    b = tf$Variable(tf$zeros(shape(n_output)))
    output = tf$nn$tanh(tf$matmul(curentInput, W) + b)
    curentInput = output
  }
  # now have the reconstruction through the network
  y = curentInput
  # cost function measures pixel-wise difference
  cost = tf$sqrt(tf$reduce_mean(tf$square(y - x)))
  return(list("x"=x, "z"=z, "y"=y, "x_corrput"=x_corrput,
"cost"=cost))
}
```

4. Create the denoising object:

```
# Create denoising AE object
dae_obj<-denoisingAutoencoder(x, x_corrput, img_size_flat=3072,
hidden_layer=c(1024, 512), out_img_size=256)
```

5. Set the cost function:

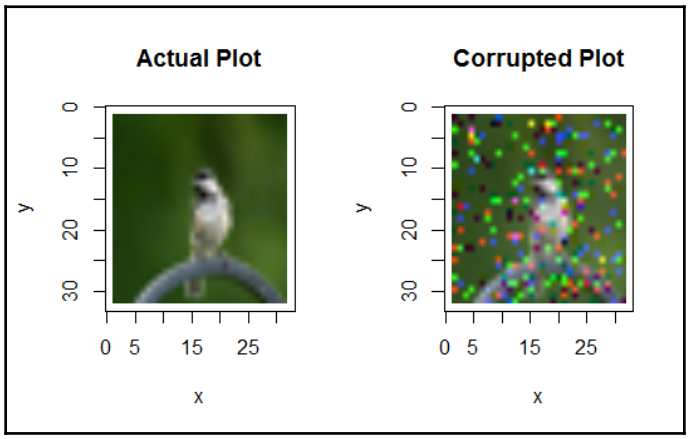
```
# Learning set-up
learning_rate = 0.001
optimizer =
tf$train$AdamOptimizer(learning_rate)$minimize(dae_obj$cost)
```

6. Run optimization:

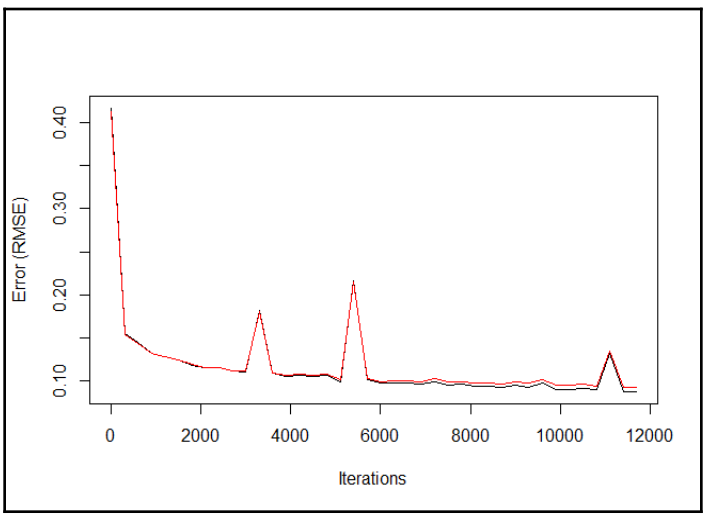
```
# We create a session to use the graph
sess$run(tf$global_variables_initializer())
for(i in 1:500){
  spls <- sample(1:dim(xcorr)[1],1000L)
  if (i %% 1 == 0) {
    x_corrput_ds<-add_noise(train_data$images[spls, ], frac = 0.3,
corr_type = "masking")
    optimizer$run(feed_dict = dict(x=train_data$images[spls, ],
x_corrput=x_corrput_ds))
    trainingCost<-dae_obj$cost$eval((feed_dict =
dict(x=train_data$images[spls, ], x_corrput=x_corrput_ds)))
    cat("Training Cost - ", trainingCost, "\n")
  }
}
```

How it works...

The autoencoder keeps learning about the function form for the feature to capture the relationship between input and output. An example of how the computer is visualizing the image after 1,000 iterations is shown in the following figure:



After 1,000 iterations, the computer can distinguish between a major part of the object and environment. As we run the algorithm further to fine-tune the weights, the computer keeps learning more features about the object itself, as shown in the following figure:



The preceding graph shows that the model is still learning, but the learning rate has become smaller over the iterations as it starts learning fine features about objects, as shown in the following image. There are instances when the model starts ascending instead of descending, due to batch gradient descent:

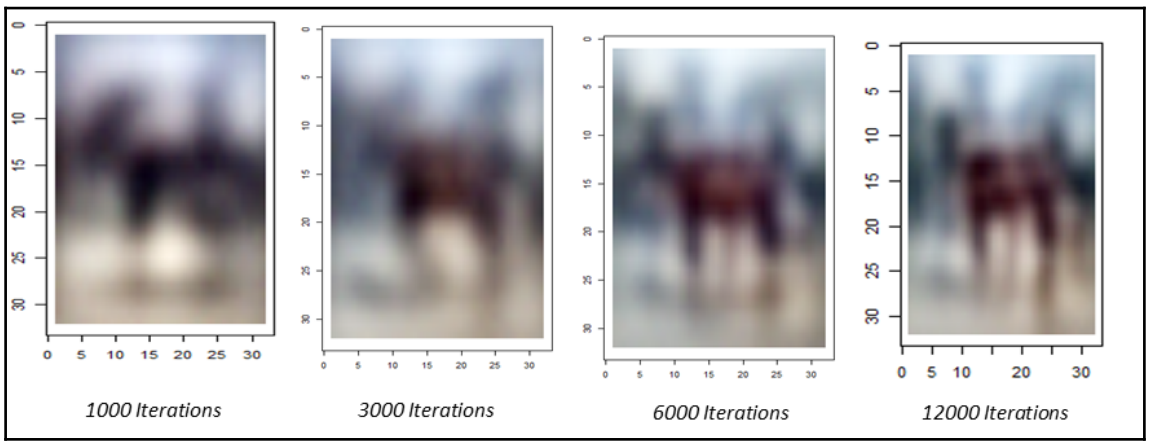
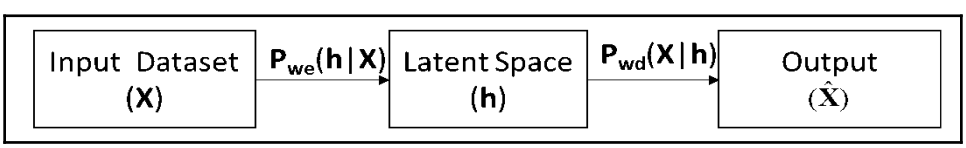


Illustration of learning using denoising autoencoder

Building and comparing stochastic encoders and decoders

Stochastic encoders fall into the domain of generative modeling, where the objective is to learn join probability *P(X)* over given data *X* transformed into another high-dimensional space. For example, we want to learn about images and produce similar, but not exactly the same, images by learning about pixel dependencies and distribution. One of the popular approaches in generative modeling is **Variational autoencoder** (**VAE**), which combines deep learning with statistical inference by making a strong distribution assumption on *h ~ P(h),* such as Gaussian or Bernoulli. For a given weight *W*, the *X* can be sampled from the distribution as *Pw(X|h)*. An example of VAE architecture is shown in the following diagram:



The cost function of VAE is based on log likelihood maximization. The cost function consists of reconstruction and regularization error terms:

Cost = Reconstruction Error + Regularization Error

The **reconstruction error** is how well we could map the outcome with the training data and the **regularization error** puts a penalty on the distribution formed at the encoder and decoder.

Getting ready

TensorFlow needs to be installed and loaded in the environment:

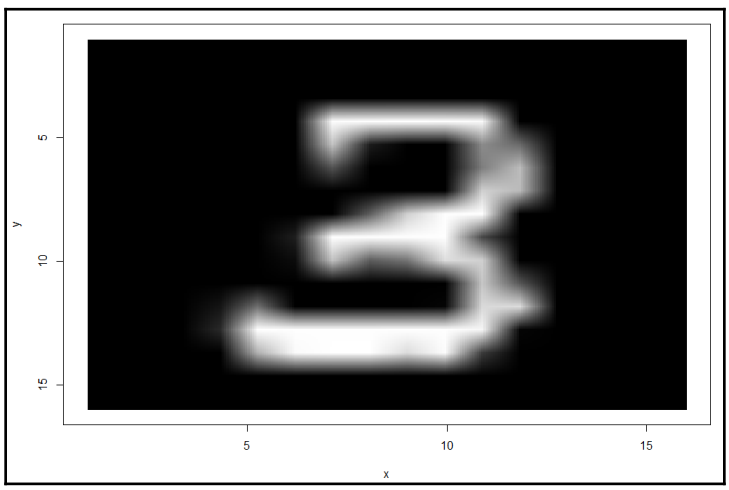
```
require(tensorflow)
```

Dependencies need to be loaded:

```
require(imager)
require(caret)
```

The `MNIST` dataset needs to be loaded. The dataset is normalized using the following script:

```
# Normalize Dataset
normalizeObj<-preProcess(trainData, method="range")
trainData<-predict(normalizeObj, trainData)
validData<-predict(normalizeObj, validData)
```



How to do it...

1. The `MNIST` dataset is used to demonstrate the concept of sparse decomposition. The `MNIST` dataset uses handwritten digits. It is downloaded from the `tensorflow` dataset library. The dataset consists of handwritten images of `28 x 28` pixels. It consists of 55,000 training examples, 10,000 test examples, and 5,000 test examples. The dataset 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)
```

2. For computational simplicity, the `MNIST` image size is reduced from `28 x 28` pixels to `16 x 16` pixels using the following function:

```
# Function to reduce image size
reduceImage<-function(actds, n.pixel.x=16, n.pixel.y=16){
  actImage<-matrix(actds, ncol=28, byrow=FALSE)
  img.col.mat <- imappend(list(as.cimg(actImage)),"c")
  thmb <- resize(img.col.mat, n.pixel.x, n.pixel.y)
  outputImage<-matrix(thmb[,,1,1], nrow = 1, byrow = F)
  return(outputImage)
}
```

3. The following script can be used to prepare the `MNIST` training data with a 16 x 16 pixel image:

```
# Covert train data to 16 x 16  image
trainData<-t(apply(mnist$train$images, 1, FUN=reduceImage))
validData<-t(apply(mnist$test$images, 1, FUN=reduceImage))
```

4. The `plot_mnist` function can be used to visualize the selected `MNIST` image:

```
# Function to plot MNIST dataset
plot_mnist<-function(imageD, pixel.y=16){
  actImage<-matrix(imageD, ncol=pixel.y, byrow=FALSE)
  img.col.mat <- imappend(list(as.cimg(actImage)), "c")
  plot(img.col.mat)
}
```

Setting up a VAE model

1. Start a new TensorFlow environment:

```
tf$reset_default_graph()
sess<-tf$InteractiveSession()
```

2. Define network parameters:

```
n_input=256
n.hidden.enc.1<-64
```

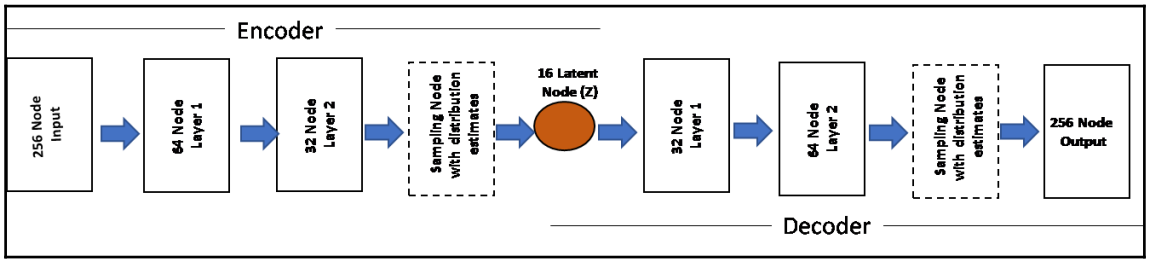
3. Start a new TensorFlow environment:

```
tf$reset_default_graph()
sess<-tf$InteractiveSession()
```

4. Define network parameters:

```
n_input=256
n.hidden.enc.1<-64
```

The preceding parameter will form a VAE network as follows:



5. Define the model initialization function, defining weights and biases at each layer of `encoder` and `decoder`:

```
model_init<-function(n.hidden.enc.1, n.hidden.enc.2,
                                n.hidden.dec.1,  n.hidden.dec.2,
                                n_input, n_h)
{ weights<-NULL
 ############################
 # Set-up Encoder
 ############################
 # Initialize Layer 1 of encoder
 weights[["encoder_w"]][["h1"]]=tf$Variable(xavier_init(n_input,
```

```
 n.hidden.enc.1))
 weights[["encoder_w"]]
[["h2"]]=tf$Variable(xavier_init(n.hidden.enc.1, n.hidden.enc.2))
weights[["encoder_w"]][["out_mean"]]=tf$Variable(xavier_init(n.hidd
en.enc.2, n_h))
weights[["encoder_w"]][["out_log_sigma"]]=tf$Variable(xavier_init(n
.hidden.enc.2, n_h))
weights[["encoder_b"]][["b1"]]=tf$Variable(tf$zeros(shape(n.hidden.
enc.1), dtype=tf$float32))
weights[["encoder_b"]][["b2"]]=tf$Variable(tf$zeros(shape(n.hidden.
enc.2), dtype=tf$float32))
weights[["encoder_b"]][["out_mean"]]=tf$Variable(tf$zeros(shape(n_h
), dtype=tf$float32))
weights[["encoder_b"]][["out_log_sigma"]]=tf$Variable(tf$zeros(shap
e(n_h), dtype=tf$float32))
 ############################
 # Set-up Decoder
 ############################
 weights[['decoder_w']][["h1"]]=tf$Variable(xavier_init(n_h,
n.hidden.dec.1))
weights[['decoder_w']][["h2"]]=tf$Variable(xavier_init(n.hidden.dec
.1, n.hidden.dec.2))
weights[['decoder_w']][["out_mean"]]=tf$Variable(xavier_init(n.hidd
en.dec.2, n_input))
weights[['decoder_w']][["out_log_sigma"]]=tf$Variable(xavier_init(n
.hidden.dec.2, n_input))
weights[['decoder_b']][["b1"]]=tf$Variable(tf$zeros(shape(n.hidden.
dec.1), dtype=tf$float32))
weights[['decoder_b']][["b2"]]=tf$Variable(tf$zeros(shape(n.hidden.
dec.2), dtype=tf$float32))
weights[['decoder_b']][["out_mean"]]=tf$Variable(tf$zeros(shape(n_i
nput), dtype=tf$float32))
weights[['decoder_b']][["out_log_sigma"]]=tf$Variable(tf$zeros(shap
e(n_input), dtype=tf$float32))
 return(weights)
}
```

The `model_init` function returns `weights`, which is a two-dimensional list. The first dimension captures the weight's association and type. For example, it describes if the `weights` variable is assigned to the encoder or decoder and if it stores the weight of the node or bias. The `xavier_init` function in `model_init` is used to assign initial weights for model training:

```
# Xavier Initialization using Uniform distribution
xavier_init<-function(n_inputs, n_outputs, constant=1){
  low = -constant*sqrt(6.0/(n_inputs + n_outputs))
  high = constant*sqrt(6.0/(n_inputs + n_outputs))
```

```
  return(tf$random_uniform(shape(n_inputs, n_outputs), minval=low,
maxval=high, dtype=tf$float32))
}
```

6. Set up the encoder evaluation function:

```
# Encoder update function
vae_encoder<-function(x, weights, biases){
  layer_1 = tf$nn$softplus(tf$add(tf$matmul(x, weights[['h1']]),
biases[['b1']]))
  layer_2 = tf$nn$softplus(tf$add(tf$matmul(layer_1,
weights[['h2']]), biases[['b2']]))
  z_mean = tf$add(tf$matmul(layer_2, weights[['out_mean']]),
biases[['out_mean']])
  z_log_sigma_sq = tf$add(tf$matmul(layer_2,
weights[['out_log_sigma']]), biases[['out_log_sigma']])
  return (list("z_mean"=z_mean, "z_log_sigma_sq"=z_log_sigma_sq))
}
```

The `vae_encoder` computes the mean and variance to sample the layer, using the weights and bias from the hidden layer.

7. Set up the decoder evaluation function:

```
# Decoder update function
vae_decoder<-function(z, weights, biases){
  layer1<-tf$nn$softplus(tf$add(tf$matmul(z, weights[["h1"]]),
biases[["b1"]]))
  layer2<-tf$nn$softplus(tf$add(tf$matmul(layer1, weights[["h2"]]),
biases[["b2"]]))
  x_reconstr_mean<-tf$nn$sigmoid(tf$add(tf$matmul(layer2,
weights[['out_mean']]), biases[['out_mean']]))
  return(x_reconstr_mean)
}
```

The `vae_decoder` function computes the mean and standard deviation associated with the sampling layer at output and average output.

8. Set up the function for reconstruction estimation:

```
# Parameter evaluation
network_ParEval<-function(x, network_weights, n_h){

  distParameter<-vae_encoder(x, network_weights[["encoder_w"]],
network_weights[["encoder_b"]])
  z_mean<-distParameter$z_mean
  z_log_sigma_sq <-distParameter$z_log_sigma_sq
```

```
  # Draw one sample z from Gaussian distribution
  eps = tf$random_normal(shape(BATCH, n_h), 0, 1, dtype=tf$float32)
  # z = mu + sigma*epsilon
  z = tf$add(z_mean, tf$multiply(tf$sqrt(tf$exp(z_log_sigma_sq)),
eps))
  # Use generator to determine mean of
  # Bernoulli distribution of reconstructed input
  x_reconstr_mean <- vae_decoder(z, network_weights[["decoder_w"]],
network_weights[["decoder_b"]])
  return(list("x_reconstr_mean"=x_reconstr_mean,
"z_log_sigma_sq"=z_log_sigma_sq, "z_mean"=z_mean))
}
```

9. Define the cost function for optimization:

```
# VAE cost function
vae_optimizer<-function(x, networkOutput){
  x_reconstr_mean<-networkOutput$x_reconstr_mean
  z_log_sigma_sq<-networkOutput$z_log_sigma_sq
  z_mean<-networkOutput$z_mean
  loss_reconstruction<--1*tf$reduce_sum(x*tf$log(1e-10 +
x_reconstr_mean)+
                                        (1-x)*tf$log(1e-10 + 1 -
x_reconstr_mean), reduction_indices=shape(1))
  loss_latent<--0.5*tf$reduce_sum(1+z_log_sigma_sq-
tf$square(z_mean)-
                                    tf$exp(z_log_sigma_sq),
reduction_indices=shape(1))
  cost = tf$reduce_mean(loss_reconstruction + loss_latent)
  return(cost)
}
```

10. Set up the model to train:

```
# VAE Initialization
x = tf$placeholder(tf$float32, shape=shape(NULL, img_size_flat),
name='x')
network_weights<-model_init(n.hidden.enc.1, n.hidden.enc.2,
                            n.hidden.dec.1,  n.hidden.dec.2,
                            n_input, n_h)
networkOutput<-network_ParEval(x, network_weights, n_h)
cost=vae_optimizer(x, networkOutput)
optimizer = tf$train$AdamOptimizer(lr)$minimize(cost)
```

11. Run optimization:

```
sess$run(tf$global_variables_initializer())
for(i in 1:ITERATION){
```

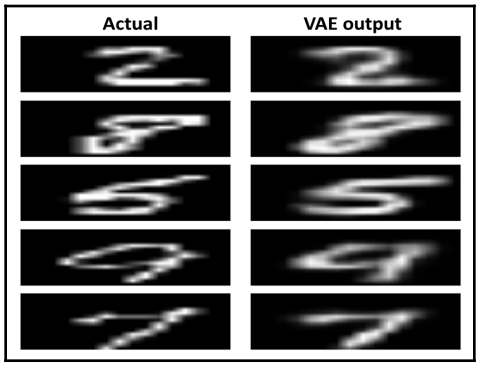
```
  spls <- sample(1:dim(trainData)[1],BATCH)
  out<-optimizer$run(feed_dict = dict(x=trainData[spls,]))
  if (i %% 100 == 0){
  cat("Iteration - ", i, "Training Loss - ",  cost$eval(feed_dict =
dict(x=trainData[spls,])), "\n")
  }
}
```

Output from the VAE autoencoder

1. The outcome can be generated using the following script:

```
spls <- sample(1:dim(trainData)[1],BATCH)
networkOutput_run<-sess$run(networkOutput, feed_dict =
dict(x=trainData[spls,]))

# Plot reconstructured Image
x_sample<-trainData[spls,]
NROW<-nrow(networkOutput_run$x_reconstr_mean)
n.plot<-5
par(mfrow = c(n.plot, 2), mar = c(0.2, 0.2, 0.2, 0.2), oma = c(3,
3, 3, 3))
pltImages<-sample(1:NROW,n.plot)
for(i in pltImages){
  plot_mnist(x_sample[i,])
  plot_mnist(networkOutput_run$x_reconstr_mean[i,])
}
```

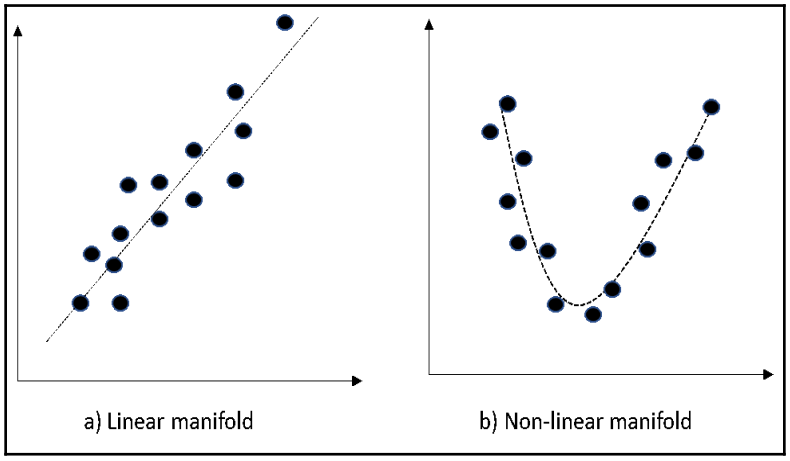
The outcome obtained after `20,000` iterations from the preceding VAE autoencoder is shown in the following figure:



Additionally, as VAE is a generative model, the outcome is not an exact replica of the input and would vary with runs, as a representative sample is extracted from the estimated distribution.

Learning manifolds from autoencoders

Manifold learning is an approach in machine learning that assumes that data lies on a manifold of a much lower dimension. These manifolds can be linear or non-linear. Thus, the area tries to project the data from high-dimension space to a low dimension. For example, **principle component analysis** (**PCA**) is an example of linear manifold learning whereas an autoencoder is a **non-linear dimensionality reduction** (**NDR**) with the ability to learn non-linear manifolds in low dimensions. A comparison of linear and non-linear manifold learning is shown in the following figure:



As you can see from graph **a)**, the data is residing at a linear manifold, whereas in graph graph **b)**, the data is residing on a second-order non-linear manifold.

How to do it...

Let's take an output from the stacked autoencoder section and analyze how manifolds look when transferred into a different dimension.

Setting up principal component analysis

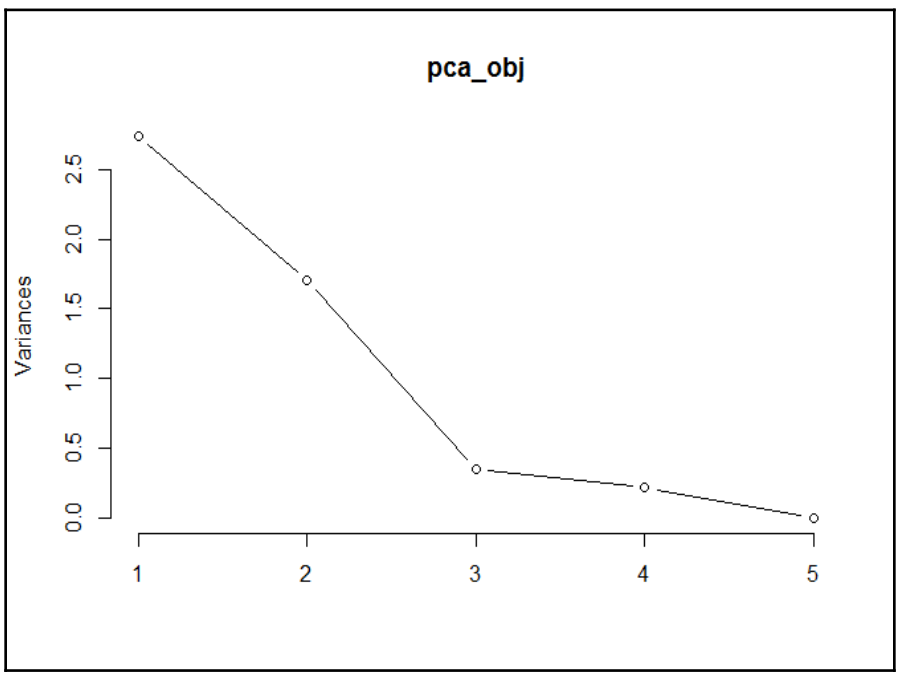
1. Before getting into non-linear manifolds, let's analyze principal component analysis on the occupancy data:

```
# Setting-up principal component analysis
pca_obj <- prcomp(occupancy_train$data,
                  center = TRUE,
                  scale. = TRUE)
                  scale. = TRUE)
```

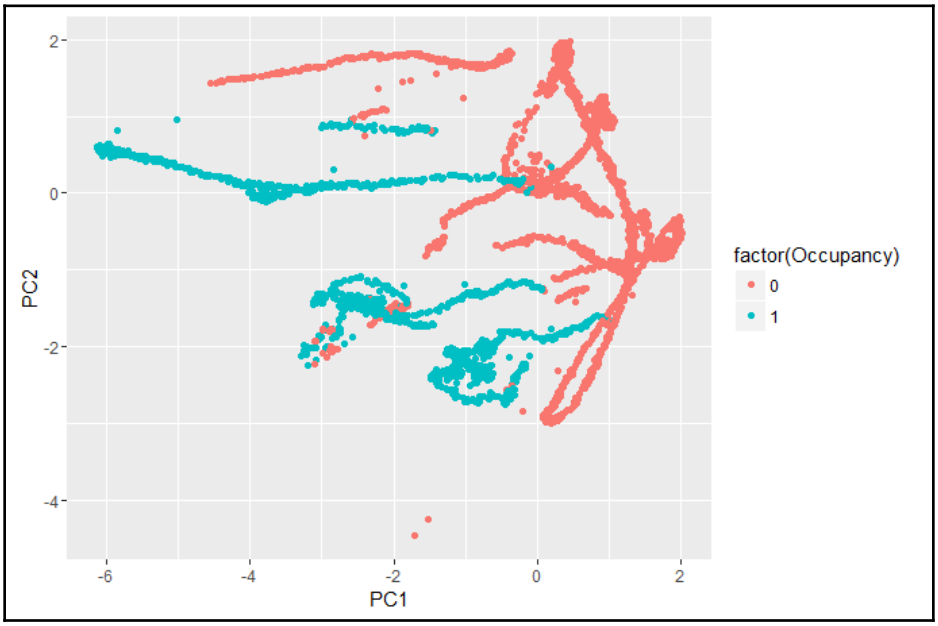
2. The preceding function will transform the data into six orthogonal directions specified as linear combinations of features. The variance explained by each dimension can be viewed using the following script:

```
plot(pca_obj, type = "l")
```

3. The preceding command will plot the variance across principal components, as shown in the following figure:



4. For the occupancy dataset, the first two principal components capture the majority of the variation, and when the principal component is plotted, it shows separation between the positive and negative classes for occupancy, as shown in the following figure:

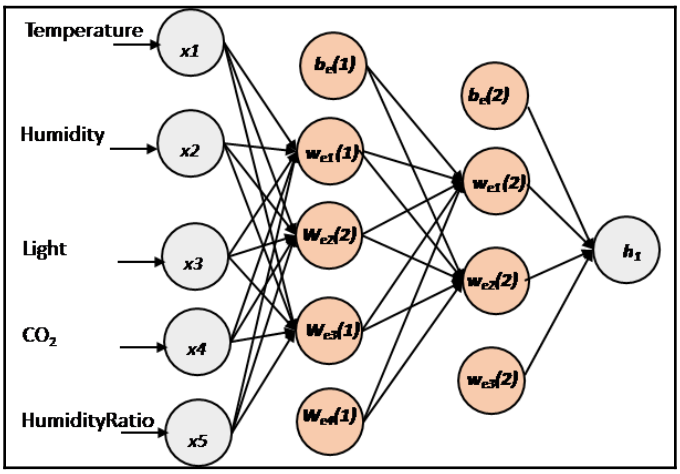


Output from the first two principal components

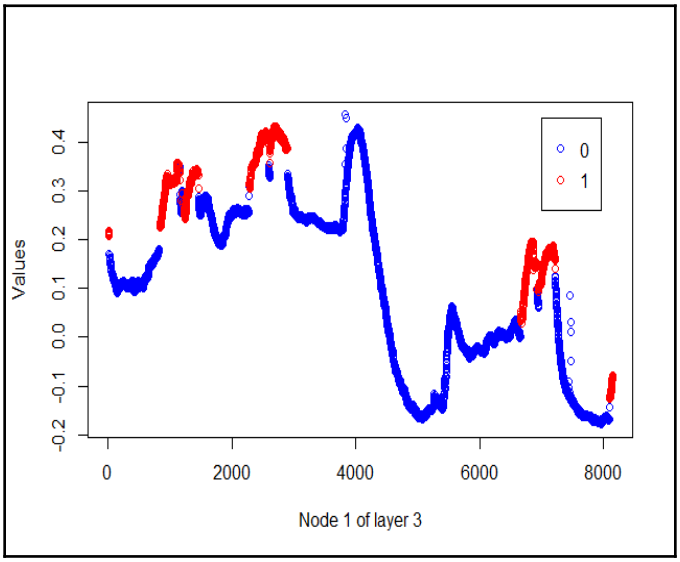
5. Let's visualize the manifold in a low dimension learned by the autoencoder. Let's use only one dimension to visualize the outcome, as follows:

```
SAE_obj<-SAENET.train(X.train= subset(occupancy_train$data,
select=-c(Occupancy)), n.nodes=c(4, 3, 1), unit.type ="tanh",
lambda = 1e-5, beta = 1e-5, rho = 0.01, epsilon = 0.01,
max.iterations=1000)
```

6. The encoder architecture for the preceding script is shown as follows:



The hidden layer outcome with one latent node from the stacked autoencoder is shown as follows:

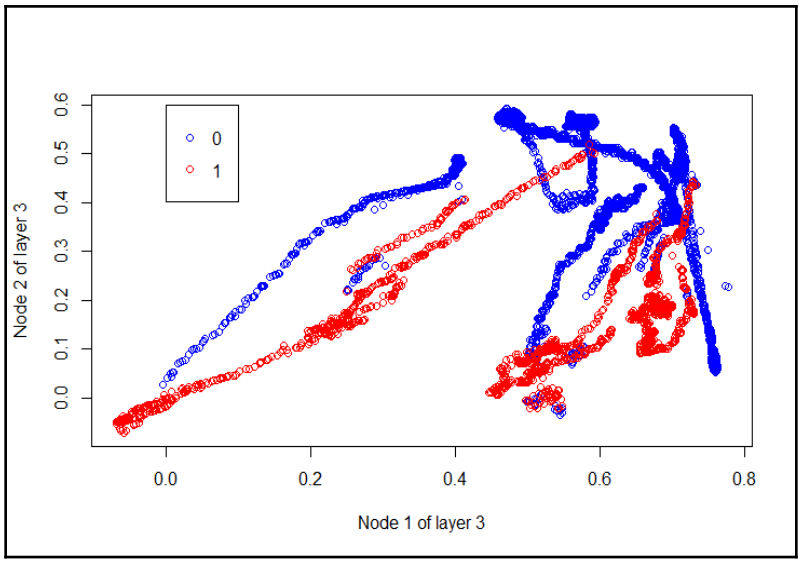


7. The preceding graph shows that occupancy is true at the peaks of the latent variables. However, the peaks are found at different values. Let's increase the latent variables 2, as captured by PCA. The model can be developed and data can be plotted using the following script:

```
SAE_obj<-SAENET.train(X.train= subset(occupancy_train$data,
```

```
select=-c(Occupancy)), n.nodes=c(4, 3, 2), unit.type ="tanh",
lambda = 1e-5, beta = 1e-5, rho = 0.01, epsilon = 0.01,
max.iterations=1000)

# plotting encoder values
plot(SAE_obj[[3]]$X.output[,1], SAE_obj[[3]]$X.output[,2],
col="blue", xlab = "Node 1 of layer 3", ylab = "Node 2 of layer 3")
ix<-occupancy_train$data[,6]==1
points(SAE_obj[[3]]$X.output[ix,1], SAE_obj[[3]]$X.output[ix,2],
col="red")
```

8. The encoded values with two layers are shown in the following diagram:



Evaluating the sparse decomposition

The sparse autoencoder is also known as over-complete representation and has a higher number of nodes in the hidden layer. The sparse autoencoders are usually executed with the sparsity parameter (regularization), which acts as a constraint and restricts the node to being active. The sparsity can also be assumed as nodes dropout caused due to sparsity constraints. The loss function for a sparse autoencoder consists of a reconstruction error, a regularization term to contain the weight decay, and KL divergence for sparsity constraint. The following representation gives a very good illustration of what we are talking about:

$$\|\mathbf{X} - f(\mathbf{h})\|^2 + \frac{\lambda}{2}\|\mathbf{W}\|^2 + \beta J_{KL}(\rho\|\hat{\rho})$$

where, $\mathbf{X}$ and $f(\mathbf{h})$ represent input matrix and output from decoder. The $\mathbf{W}$ represent weight matrix for nodes and λ and β are regularization parameter and penalty to sparsity term. The term $J_{KL}(\rho\|\hat{\rho})$ is KL divergence value captured for sparsity term ρ. KL divergence for given sparsity parameter can be computed as

$$J_{KL}(\rho\|\hat{\rho}) = \sum_{i=1}^{n'} \rho \log(\frac{\rho}{\hat{\rho}_i}) + (1-\rho)\log(\frac{1-\rho}{1-\hat{\rho}_i})$$

where, n' is number of nodes in layer $\mathbf{h}$ and $\hat{\rho}$ is vector of average activities in hidden layer on normalized neurons.

Getting ready

1. The dataset is loaded and set up.
2. Install and load the `autoencoder` package using the following script:

```
install.packages("autoencoder")
require(autoencoder)
```

How to do it...

1. The standard autoencoder code of TensorFlow can easily be extended to the sparse autoencoder module by updating the cost function. This section will introduce the autoencoder package of R, which comes with built-in functionality to run the sparse autoencoder:

```
### Setting-up parameter
nl<-3
N.hidden<-100
unit.type<-"logistic"
lambda<-0.001
rho<-0.01
beta<-6
```

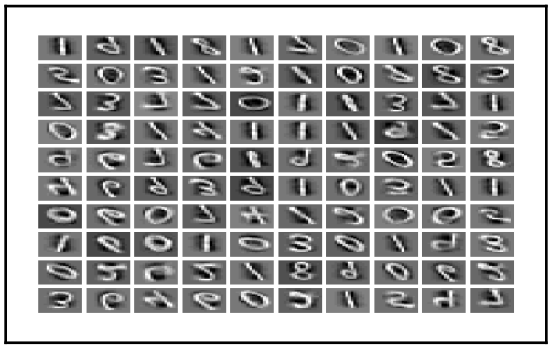
```
max.iterations<-2000
epsilon<-0.001

### Running sparse autoencoder
spe_ae_obj <- autoencode(X.train=trainData,  X.test = validData,
nl=nl, N.hidden=N.hidden,
unit.type=unit.type,lambda=lambda,beta=beta,
epsilon=epsilon,rho=rho,max.iterations=max.iterations, rescale.flag
= T)
```

The major parameters in the `autoencode` functions are as follows:

- `nl`: This is the number of layers including the input and output layer (the default is three).
- `N.hidden`: This is the vector with the number of neurons in each hidden layer.
- `unit.type`: This is the type of activation function to be used.
- `lambda`: This is the regularization parameter.
- `rho`: This is the sparsity parameter.
- `beta`: This is the penalty for the sparsity term.
- `max.iterations`: This is the maximum number of iterations.
- `epsilon`: This is the parameter for weight initialization. The weights are initialized using Gaussian distribution ~N(0, epsilon2).

How it works...

The following figure shows the shapes and orientation of digits from `MNIST` captured by the sparse autoencoder:



Filter generated by the sparse autoencoder to get the digit outcome

The filter learned by the sparse autoencoder can be visualized using the `visualize.hidden.units` function from the autoencoder package. The package plots the weight of the final layer with respect to the output. In the current scenario, 100 is the number of neurons in the hidden layer and 256 is the number of nodes in the output layer.