9
Application of Deep Learning to Signal processing

The current chapter will present a case study of creating new music notes using generative modeling techniques such as RBM. In this chapter, we will cover the following topics:

- Introducing and preprocessing music MIDI files
- Building an RBM model
- Generating new music notes

Introducing and preprocessing music MIDI files

In this recipe, we will read a repository of **Musical Instrument Digital Interface** (**MIDI**) files and preprocess them into a suitable format for an RBM. MIDI is one of the formats of storing musical notes, which can be converted to other formats such as `.wav`, `.mp3`, `.mp4`, and so on. MIDI file formats store various kinds of events such as Note-on, Note-off, Tempo, Time Signature, End of track, and so on. However, we will primarily be focusing on the type of note--when it was turned **on**, and when it was turned **off**.

Each song is encoded into a binary matrix, where rows represent time, and columns represent both turned on and turned off notes. At each time, a note is turned on and the same note is turned off. Suppose that, out of n notes, note i is turned on and turned off at time j, then positions $Mji = 1$ and $Mj(n+i) = 1$, and the rest $Mj = 0$.

All the rows together form a song. Currently, in this chapter, we will be leveraging Python codes to encode MIDI songs into binary matrices, which can later be used in a Restricted Boltzmann Machine (RBM).

Getting ready

Let's look at the prerequisites to preprocess MIDI files:

1. Download the MIDI song repository:

```
https://github.com/dshieble/Music_RBM/tree/master/Pop_Music_Midi
```

2. Download the Python codes to manipulate MIDI songs:

```
https://github.com/dshieble/Music_RBM/blob/master/midi_manipulation.py
```

3. Install the `"reticulate"` package, which provides the R interface to Python:

```
Install.packages("reticulate")
```

4. Import Python libraries:

```
use_condaenv("python27")
midi <-
import_from_path("midi",path="C:/ProgramData/Anaconda2/Lib/site-
packages")
np <- import("numpy")
msgpack <-
import_from_path("msgpack",path="C:/ProgramData/Anaconda2/Lib/site-
packages")
psys <- import("sys")
tqdm <-
import_from_path("tqdm",path="C:/ProgramData/Anaconda2/Lib/site-
packages")
midi_manipulation_updated <-
import_from_path("midi_manipulation_updated",path="C:/Music_RBM")
glob <- import("glob")
```

How to do it...

Now that we have set up all the essentials, let's look at the function to define MIDI files:

1. Define the function to read the MIDI files and encode them into a binary matrix:

```
get_input_songs <- function(path){
  files = glob$glob(paste0(path,"/*mid*"))
  songs <- list()
  count <- 1
  for(f in files){
    songs[[count]] <-
np$array(midi_manipulation_updated$midiToNoteStateMatrix(f))
    count <- count+1
  }
  return(songs)
}
path <- 'Pop_Music_Midi'
input_songs <- get_input_songs(path)
```

Building an RBM model

In this recipe, we will build an RBM model as discussed (in detail) in `Chapter 5`, *Generative Models in Deep Learning*.

Getting ready

Let's set up our system for the model:

1. In Piano, the lowest note is 24 and the highest is 102; hence, the range of notes is 78. Thus, the number of columns in the encoded matrix is 156 (that is, 78 for note-on and 78 for note-off):

```
lowest_note = 24L
highest_note = 102L
note_range = highest_note-lowest_note
```

2. We will create notes for 15 number of steps at a time with 2,340 nodes in the input layer and 50 nodes in the hidden layer:

```
num_timesteps  = 15L
num_input      = 2L*note_range*num_timesteps
num_hidden       = 50L
```

3. The learning rate (alpha) is 0.1:

```
alpha<-0.1
```

How to do it...

Looking into the steps of building an RBM model:

1. Define the `placeholder` variables:

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

2. Define a forward pass:

```
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))))
```

3. Then, define a backward pass:

```
prob_v1 = tf$matmul(h0, tf$transpose(W)) + vb
v1 = prob_v1 + tf$random_normal(tf$shape(prob_v1), mean=0.0,
stddev=1.0, dtype=tf$float32)
h1 = tf$nn$sigmoid(tf$matmul(v1, W) + hb)
```

4. Calculate positive and negative gradients accordingly:

```
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)
```

5. Define the objective function:

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

6. Initialize the current and previous variables:

```
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))
```

7. Start a TensorFlow session:

```
sess$run(tf$global_variables_initializer())
song = np$array(trainX)
song =
song[1:(np$floor(dim(song)[1]/num_timesteps)*num_timesteps),]
song = np$reshape(song, newshape=shape(dim(song)[1]/num_timesteps,
dim(song)[2]*num_timesteps))
output <- sess$run(list(update_w, update_vb, update_hb), feed_dict
= dict(X=song,
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= song, W= prv_w, vb= prv_vb, hb=
prv_hb))
```

8. Run `200` training epochs:

```
epochs=200
errors <- list()
weights <- list()
u=1
for(ep in 1:epochs){
  for(i in seq(0,(dim(song)[1]-100),100)){
    batchX <- song[(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%%500 == 0){
```

```
      errors[[u]] <- sess$run(err, feed_dict=dict(X= song, 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")
}
```

Generating new music notes

In this recipe, we will generate new sample music notes. New musical notes can be generated by altering parameter `num_timesteps`. However, one should keep in mind to increase the timesteps, as it can become computationally inefficient to handle increased dimensionality of vectors in the current setup of RBM. These RBMs can be made efficient in learning by creating their stacks (namely **Deep Belief Networks**). Readers can leverage the DBN codes of `Chapter 5`, *Generative Models in Deep Learning,* to generate new musical notes.

How to do it...

1. Create new sample music:

```
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$run(vv1, feed_dict=dict( hh0= feed, W= prv_w, vb=
prv_vb))
S = np$reshape(rec[1,],newshape=shape(num_timesteps,2*note_range))
```

2. Regenerate the MIDI file:

```
midi_manipulation$noteStateMatrixToMidi(S,
name=paste0("generated_chord_1"))
generated_chord_1
```