6
Recurrent Neural Networks

The current chapter will introduce Recurrent Neural Networks used for the modeling of sequential datasets. In this chapter, we will cover:

- Setting up a basic Recurrent Neural Network
- Setting up a bidirectional RNN model
- Setting up a deep RNN model
- Setting up a Long short-term memory based sequence model

Setting up a basic Recurrent Neural Network

Recurrent Neural Networks (**RNN**) are used for sequential modeling on datasets where high autocorrelation exists among observations. For example, predicting patient journeys using their historical dataset or predicting the next words in given sentences. The main commonality among these problem statements is that input length is not constant and there is a sequential dependence. Standard neural network and deep learning models are constrained by fixed size input and produce a fixed length output. For example, deep learning neural networks built on occupancy datasets have six input features and a binomial outcome.

Getting ready

Generative models in machine learning domains are referred to as models that have an ability to generate observable data values. For example, training a generative model on an images repository to generate new images like it. All generative models aim to compute the joint distribution over given datasets, either implicitly or explicitly:

1. Install and set up TensorFlow.
2. Load required packages:

```
library(tensorflow)
```

How to do it...

The section will provide steps to set-up an RNN model.

1. Load the `MNIST` dataset:

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

2. Reset the graph and start an interactive session:

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

3. Reduce image size to 16 x 16 pixels using the `reduceImage` function from `Chapter 4`, *Data Representation using Autoencoders*:

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

4. Extract labels for the defined `train` and `valid` datasets:

```
labels <- mnist$train$labels
labels_valid <- mnist$test$labels
```

5. Define model parameters such as size of input pixels (`n_input`), step size (`step_size`), number of hidden layers (`n.hidden`), and number of outcome classes (`n.classes`):

```
# Define Model parameter
n_input<-16
step_size<-16
n.hidden<-64
n.class<-10
```

6. Define training parameters such as learning rate (`lr`), number of inputs per batch run (`batch`), and number of iterations (`iteration`):

```
lr<-0.01
batch<-500
iteration = 100
```

7. Define a function `rnn` that takes in batch input dataset (`x`), weight matrix (`weight`), and bias vector (`bias`); and returns a final outcome predicted vector of a most basic RNN:

```
# Set up a most basic RNN
rnn<-function(x, weight, bias){
  # Unstack input into step_size
  x = tf$unstack(x, step_size, 1)
  # Define a most basic RNN
  rnn_cell = tf$contrib$rnn$BasicRNNCell(n.hidden)
  # create a Recurrent Neural Network
  cell_output = tf$contrib$rnn$static_rnn(rnn_cell, x,
dtype=tf$float32)
  # Linear activation, using rnn inner loop
  last_vec=tail(cell_output[[1]], n=1)[[1]]
  return(tf$matmul(last_vec, weights) + bias)
}
Define a function eval_func to evaluate mean accuracy using actual
(y) and predicted labels (yhat):
# Function to evaluate mean accuracy
eval_acc<-function(yhat, y){
  # Count correct solution
  correct_Count = tf$equal(tf$argmax(yhat,1L), tf$argmax(y,1L))
  # Mean accuracy
  mean_accuracy = tf$reduce_mean(tf$cast(correct_Count,
tf$float32))
  return(mean_accuracy)
}
```

8. Define `placeholder` variables (`x` and `y`) and initialize weight matrix and bias vector:

```
with(tf$name_scope('input'), {
# Define placeholder for input data
x = tf$placeholder(tf$float32, shape=shape(NULL, step_size,
n_input), name='x')
y <- tf$placeholder(tf$float32, shape(NULL, n.class), name='y')

# Define Weights and bias
weights <- tf$Variable(tf$random_normal(shape(n.hidden, n.class)))
bias <- tf$Variable(tf$random_normal(shape(n.class)))
})
```

9. Generate the predicted labels:

```
# Evaluate rnn cell output
yhat = rnn(x, weights, bias)
Define the loss function and optimizer
cost =
tf$reduce_mean(tf$nn$softmax_cross_entropy_with_logits(logits=yhat,
labels=y))
optimizer = tf$train$AdamOptimizer(learning_rate=lr)$minimize(cost)
```

10. Run the optimization post initializing a session using the global variables initializer:

```
sess$run(tf$global_variables_initializer())
for(i in 1:iteration){
  spls <- sample(1:dim(trainData)[1],batch)
  sample_data<-trainData[spls,]
  sample_y<-labels[spls,]
  # Reshape sample into 16 sequence with each of 16 element
  sample_data=tf$reshape(sample_data, shape(batch, step_size,
n_input))
  out<-optimizer$run(feed_dict = dict(x=sample_data$eval(),
y=sample_y))
  if (i %% 1 == 0){
    cat("iteration - ", i, "Training Loss - ",  cost$eval(feed_dict
= dict(x=sample_data$eval(), y=sample_y)), "\n")
  }
}
```

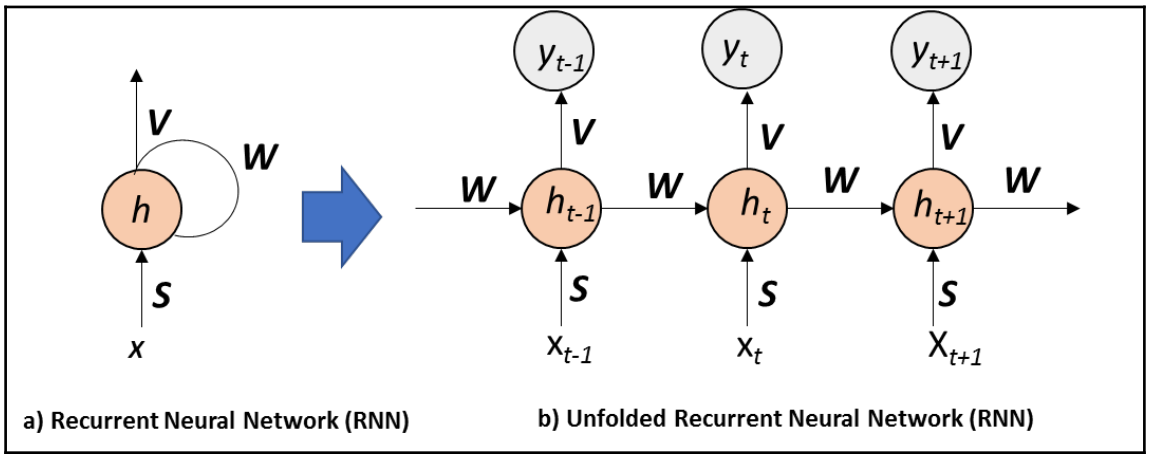
11. Get the mean accuracy on `valid_data`:

```
valid_data=tf$reshape(validData, shape(-1, step_size, n_input))
cost$eval(feed_dict=dict(x=valid_data$eval(), y=labels_valid))
```

How it works...

Any changes to the structure require model retraining. However, these assumptions may not be valid for a lot of sequential datasets, such as text-based classifications that may have varying input and output. RNN architecture helps to address the issue of variable input length.

The standard architecture for RNN with input and output is shown in the following figure:



Recurrent Neural Network architecture

The RNN architecture can be formulated as follows:

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, x_t; \mathbf{S}, \mathbf{W})$$

Where h_t is state at time/index t and x_t is input at time/index t. The matrix W represents weights to connect hidden nodes and S connects input with the hidden layer. The output node at time/index t is related to state h_t as shown as follows:

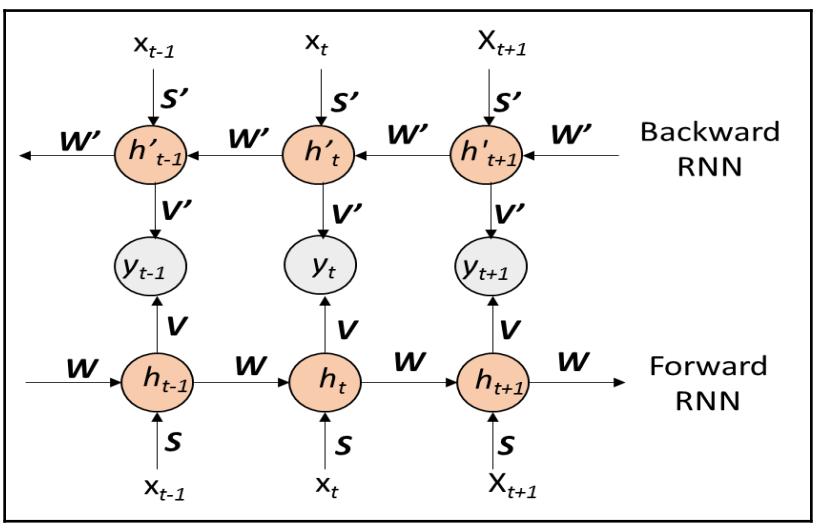
$$\mathbf{y}_t = f(\mathbf{h}_t; \mathbf{V})$$

In the previous Equations layer, weights remain constant across state and time.

Setting up a bidirectional RNN model

Recurrent Neural Networks focus on capturing the sequential information at time *t* by using historical states only. However, bidirectional RNN train the model from both directions using two RNN layers with one moving forwards from start to end and another RNN layer moving backwards from end to start of sequence.

Thus, the model is dependent on historical and future data. The bidirectional RNN models are useful where causal structure exists such as in text and speech. The unfolded structure of bidirectional RNN is shown in the following figure:



Unfolded bidirectional RNN architecture

Getting ready

Install and set up TensorFlow:

1. Load required packages:

```
library(tensorflow)
```

2. Load `MNIST` dataset.
3. The image from `MNIST` dataset is reduced to 16 x 16 pixels and normalized (Details are discussed in the *Setting-up RNN model* section).

How to do it...

This section covers the steps to set-up a bidirectional RNN model.

1. Reset the graph and start an interactive session:

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

2. Reduce image size to 16 x 16 pixels using the `reduceImage` function from `Chapter 4`, *Data Representation using Autoencoders*:

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

3. Extract labels for the defined `train` and `valid` datasets:

```
labels <- mnist$train$labels
labels_valid <- mnist$test$labels
```

4. Define model parameters such as the size of input pixels (`n_input`), step size (`step_size`), number of hidden layers (`n.hidden`), and number of outcome classes (`n.classes`):

```
# Define Model parameter
n_input<-16
step_size<-16
n.hidden<-64
n.class<-10
```

5. Define training parameters such as learning rate (`lr`), number of inputs per batch run (`batch`), and number of iterations (`iteration`):

```
lr<-0.01
batch<-500
iteration = 100
```

6. Define a function to perform `bidirectional` Recurrent Neural Network:

```
bidirectionRNN<-function(x, weights, bias){
  # Unstack input into step_size
  x = tf$unstack(x, step_size, 1)
  # Forward lstm cell
  rnn_cell_forward = tf$contrib$rnn$BasicRNNCell(n.hidden)
```

```
  # Backward lstm cell
  rnn_cell_backward = tf$contrib$rnn$BasicRNNCell(n.hidden)
  # Get lstm cell output
  cell_output =
tf$contrib$rnn$static_bidirectional_rnn(rnn_cell_forward,
rnn_cell_backward, x, dtype=tf$float32)
  # Linear activation, using rnn inner loop last output
  last_vec=tail(cell_output[[1]], n=1)[[1]]
  return(tf$matmul(last_vec, weights) + bias)
}
```

7. Define an `eval_func` function to evaluate mean accuracy using actual (`y`) and predicted labels (`yhat`):

```
# Function to evaluate mean accuracy
eval_acc<-function(yhat, y){
  # Count correct solution
  correct_Count = tf$equal(tf$argmax(yhat,1L), tf$argmax(y,1L))
  # Mean accuracy
  mean_accuracy = tf$reduce_mean(tf$cast(correct_Count,
tf$float32))
  return(mean_accuracy)
}
```

8. Define `placeholder` variables (`x` and `y`) and initialize weight matrix and bias vector:

```
with(tf$name_scope('input'), {
# Define placeholder for input data
x = tf$placeholder(tf$float32, shape=shape(NULL, step_size,
n_input), name='x')
y <- tf$placeholder(tf$float32, shape(NULL, n.class), name='y')

# Define Weights and bias
weights <- tf$Variable(tf$random_normal(shape(n.hidden, n.class)))
bias <- tf$Variable(tf$random_normal(shape(n.class)))
})
```

9. Generate the predicted labels:

```
# Evaluate rnn cell output
yhat = bidirectionRNN(x, weights, bias)
```

10. Define the loss function and optimizer:

```
cost =
tf$reduce_mean(tf$nn$softmax_cross_entropy_with_logits(logits=yhat,
```

```
labels=y))
optimizer = tf$train$AdamOptimizer(learning_rate=lr)$minimize(cost)
```

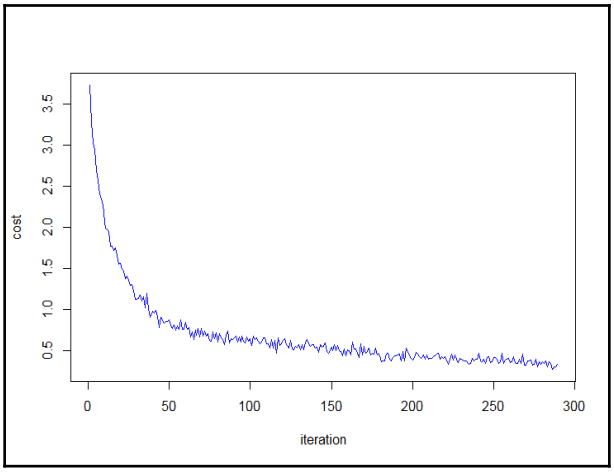
11. Run the optimization post initializing a session using global variables initializer:

```
sess$run(tf$global_variables_initializer())
# Running optimization
for(i in 1:iteration){
  spls <- sample(1:dim(trainData)[1],batch)
  sample_data<-trainData[spls,]
  sample_y<-labels[spls,]
  # Reshape sample into 16 sequence with each of 16 element
  sample_data=tf$reshape(sample_data, shape(batch, step_size,
n_input))
  out<-optimizer$run(feed_dict = dict(x=sample_data$eval(),
y=sample_y))
  if (i %% 1 == 0){
    cat("iteration - ", i, "Training Loss - ",  cost$eval(feed_dict
= dict(x=sample_data$eval(), y=sample_y)), "\n")
  }
}
```

12. Get the mean accuracy on valid data:

```
valid_data=tf$reshape(validData, shape(-1, step_size, n_input))
cost$eval(feed_dict=dict(x=valid_data$eval(), y=labels_valid))
```

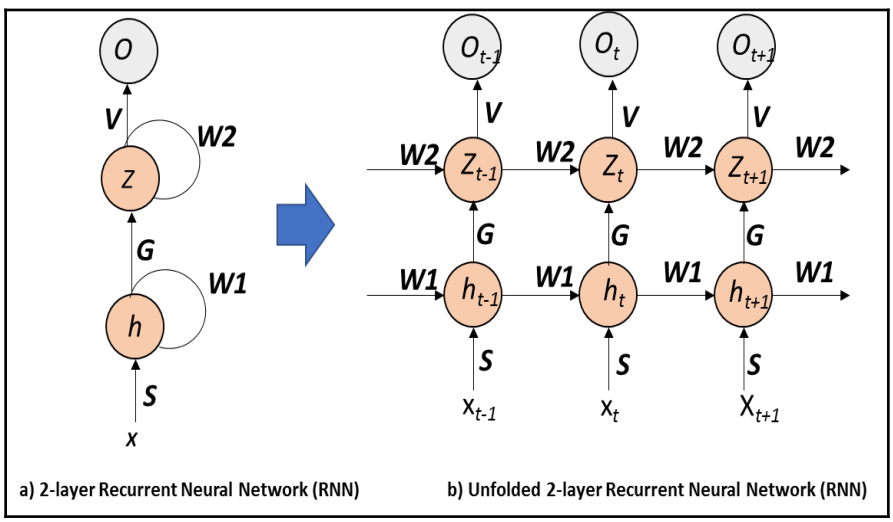
13. The convergence of cost function for RNN is shown in the following figure:



Bidirectional Recurrent Neural Network convergence plot on MNIST dataset

Setting up a deep RNN model

The RNN architecture is composed of input, hidden, and output layers. A RNN network can be made deep by decomposing the hidden layer into multiple groups or by adding computational nodes within RNN architecture such as including model computation such as multilayer perceptron for micro learning. The computational nodes can be added between input-hidden, hidden-hidden, and hidden-output connection. An example of a multilayer deep RNN model is shown in the following figure:



An example of two-layer Deep Recurrent Neural Network architecture

How to do it...

The RNN models in TensorFlow can easily be extended to Deep RNN models by using `MultiRNNCell`. The previous `rnn` function can be replaced with the `stacked_rnn`function to achieve a deep RNN architecture:

1. Define the number of layers in the deep RNN architecture:

```
num_layers <- 3
```

2. Define a `stacked_rnn` function to perform multi-hidden layers deep RNN:

```
stacked_rnn<-function(x, weight, bias){
  # Unstack input into step_size
  x = tf$unstack(x, step_size, 1)
```

```
  # Define a most basic RNN
  network = tf$contrib$rnn$GRUCell(n.hidden)
  # Then, assign stacked RNN cells
  network =
tf$contrib$rnn$MultiRNNCell(lapply(1:num_layers,function(k,network)
{network},network))
  # create a Recurrent Neural Network
  cell_output = tf$contrib$rnn$static_rnn(network, x,
dtype=tf$float32)
  # Linear activation, using rnn inner loop
  last_vec=tail(cell_output[[1]], n=1)[[1]]
  return(tf$matmul(last_vec, weights) + bias)
}
```

Setting up a Long short-term memory based sequence model

In sequence learning the objective is to capture short-term and long-term memory. The short-term memory is captured very well by standard RNN, however, they are not very effective in capturing long-term dependencies as the gradient vanishes (or explodes rarely) within an RNN chain over time.

The gradient vanishes when the weights have small values that on multiplication vanish over time, whereas in contrast, scenarios where weights have large values keep increasing over time and lead to divergence in the learning process. To deal with the issue **Long Short Term Memory** (**LSTM**) is proposed.

How to do it...

The RNN models in TensorFlow can easily be extended to LSTM models by using `BasicLSTMCell`. The previous `rnn` function can be replaced with the `lstm` function to achieve an LSTM architecture:

```
# LSTM implementation
lstm<-function(x, weight, bias){
  # Unstack input into step_size
  x = tf$unstack(x, step_size, 1)
  # Define a lstm cell
  lstm_cell = tf$contrib$rnn$BasicLSTMCell(n.hidden, forget_bias=1.0,
state_is_tuple=TRUE)
  # Get lstm cell output
```

```
    cell_output = tf$contrib$rnn$static_rnn(lstm_cell, x, dtype=tf$float32)
    # Linear activation, using rnn inner loop last output
    last_vec=tail(cell_output[[1]], n=1)[[1]]
    return(tf$matmul(last_vec, weights) + bias)
}
```

For brevity the other parts of the code are not replicated.

How it works...

The LSTM has a similar structure to RNN, however, the basic cell is very different as traditional RNN uses single **multi-layer perceptron** (**MLP**), whereas a single cell of LSTM includes four input layers interacting with each other. These three layers are:

- forget gate
- input gate
- output gate

The *forget gate* in LSTM decides which information to throw away and it depends on the last hidden state output h_{t-1}, X_t, which represents input at time *t*.

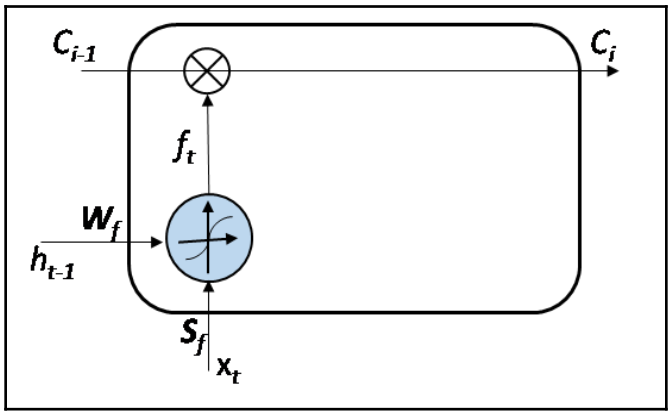


Illustration of forget gate

In the earlier figure, C_t represents cell state at time *t*. The input data is represented by X_t and the hidden state is represented as h_{t-1}. The earlier layer can be formulated as:

$$f_t = \sigma(\mathbf{S}_f \mathbf{x}_t + \mathbf{W}_f \mathbf{h}_{t-1} + \mathbf{b}_f)$$

The *input gate* decides update values and decides the candidate values of the memory cell and updates the cell state, as shown in the following figure:

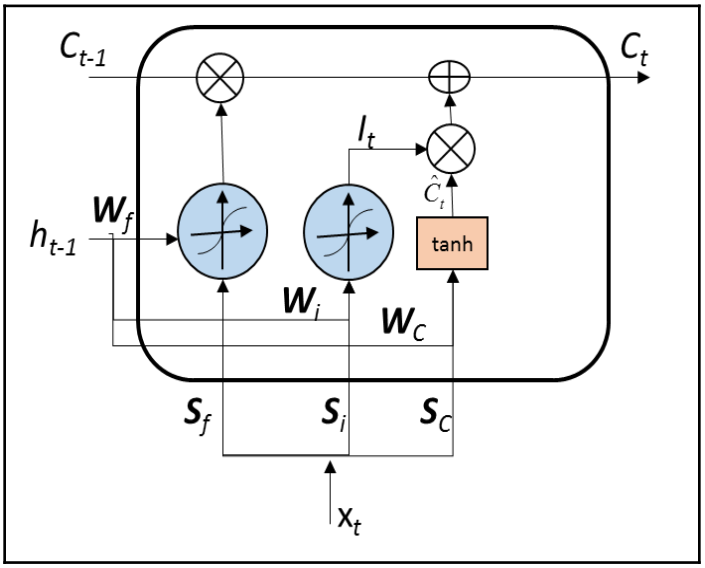


Illustration of input gate

- The input i_t at time t is updated as:

$$i_t = \sigma(\mathbf{S}_i \mathbf{x}_t + \mathbf{W}_i \mathbf{h}_{t-1} + \mathbf{b}_i)$$

$$\hat{C}_t = \tanh(\mathbf{S}_C \mathbf{x}_t + \mathbf{W}_C \mathbf{h}_{t-1} + \mathbf{b}_C)$$

- The expected value of current state $\hat{C}_t$ and the output from input gate i_t is used to update the current state C_t at time t as:

$$C_t = I_t * \hat{C}_t + f_t * C_{t-1}$$

The output gates, as shown in the following figure, compute the output from the LSTM cell based on input X_t , previous layer output h_{t-1}, and current state C_t:

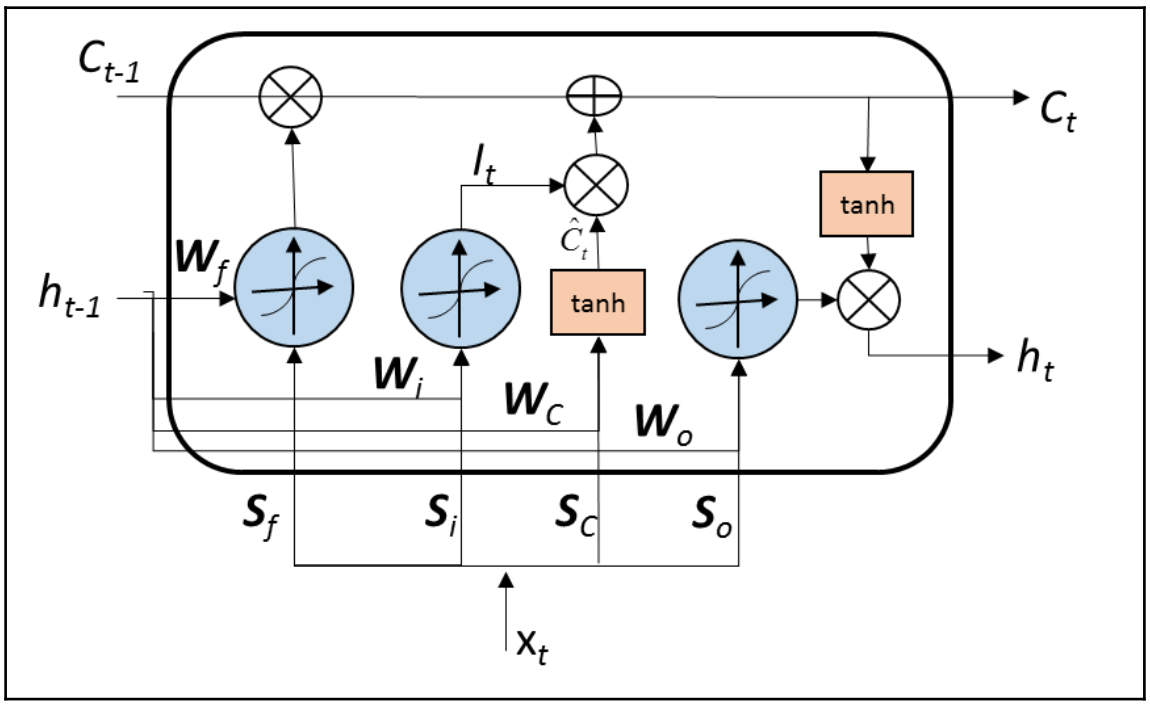


Illustration of output gate

The output based on *output gate* can be computed as follows:

$$O_t = \sigma(\mathbf{S}_o\mathbf{x}_t + \mathbf{W}_o\mathbf{h}_{t-1} + \mathbf{b}_o)$$

$$h_t = O_t * \tanh(C_t)$$