Lab 9

Hüseyin Anil Gündüz

2021-01-14

In the first part of the exercise, we will derive the backpropagation equations for a simple RNN from forward propagation equations theoretically. Then, in the second part, we will implement forward and backward propagation functions for a simple RNN-model, and we optimize the parameters of a model using these functions. This model will predict the temperature values in the future, from real past weather metrics.

Exercise 1 - Derivation of backpropagation steps of vanilla RNN

In this part, we derive the backpropagation equations for a simple RNN from forward propagation equations. For further information or if you get stuck while solving the first exercise, please visit https://www.deeplearningbook.org/contents/rnn.html, which we also refer as a source for this part of the exercise. We will apply our derivations in the second part of this exercise.

The forward-propagation equations for a simple RNN with hyperbolic tangent activation are given below.

$$\mathbf{h}^{[t]} = \tanh\left(\mathbf{W}\mathbf{h}^{[t-1]} + \mathbf{U}\mathbf{x}^{[t]} + \mathbf{b}\right) \tag{1}$$

$$\mathbf{y}^{[t]} = \mathbf{V}\mathbf{h}^{[t]} + \mathbf{c} \tag{2}$$

where the parameters are the bias vectors $\mathbf{b}$ and $\mathbf{c}$ along with the weight matrices $\mathbf{U}$,$\mathbf{V}$ and $\mathbf{W}$, respectively, for input-to-hidden, hidden-to-output and hidden-to-hidden connections. As we will use RNN for a regression problem in the of the exercise, we do not use an activation function in order to compute the output $\mathbf{y}^{[t]}$ (at time t). If that would be a classification problem, we would be better off with a $softmax$ activation in order to calculate the normalized probabilities over the output.

Given the above equations, we will derive the backpropagation equations for a simple RNN. More specifically, we will derive $\nabla_{\mathbf{c}} L$, $\nabla_{\mathbf{b}} L$, $\nabla_{\mathbf{V}} L$, $\nabla_{\mathbf{W}} L$ and $\nabla_{\mathbf{U}} L$ respectively in terms of hidden vectors $\mathbf{h}^{[t]}$, input vectors $\mathbf{x}^{[t]}$, weight matrices $\mathbf{W}$ and $\mathbf{V}$, and the gradient of the loss with respect to the output $\nabla_{\mathbf{y}^{[t]}} L$. Please note that, it is not necessary to compute the gradient with respect to input vectors $\mathbf{x}^{[t]}$ for training, since we can calculate the gradients with respect to all trainable parameters without calculating that.

For convenience, we will divide the mentioned derivations into two parts.

In the first part,we compute $\nabla_{\mathbf{h}^{[t]}} L$ for $t \in 0, ..., \tau$ in terms of hidden vectors $\mathbf{h}^{[t]}$, weight matrices $\mathbf{W}$ and $\mathbf{V}$, and the gradients of the loss with respect to the output $\nabla_{\mathbf{y}^{[t]}} L$.

Hint: This is backpropagation and so we start from the end of the sequence. At the time step $t \in 0, ..., \tau - 1$, the gradient $\nabla_{\mathbf{h}^{[t]}} L$ depends on $\nabla_{\mathbf{y}^{[t]}} L$ as well as $\nabla_{\mathbf{h}^{[t+1]}} L$, whereas the gradient $\nabla_{\mathbf{h}^{[\tau]}} L$ does not depend on $\nabla_{\mathbf{h}^{[\tau+1]}} L$.

#TODO

$$\nabla_{\mathbf{h}^{[\tau]}} L = \left(\frac{\partial \mathbf{y}^{[\tau]}}{\partial \mathbf{h}^{[\tau]}}\right)^T (\nabla_{\mathbf{y}^{[\tau]}} L) = \mathbf{V}^T (\nabla_{\mathbf{y}^{[\tau]}} L) \tag{3}$$

$$\nabla_{\mathbf{h}^{[t]}} L = \left(\frac{\partial \mathbf{h}^{[t+1]}}{\partial \mathbf{h}^{[t]}}\right)^T (\nabla_{\mathbf{h}^{[t+1]}} L) + \left(\frac{\partial \mathbf{y}^{[t]}}{\partial \mathbf{h}^{[t]}}\right)^T (\nabla_{\mathbf{y}^{[t]}} L) = \mathbf{W}^T diag\left(1 - \left(\mathbf{h}^{[t+1]}\right)^2\right)(\nabla_{\mathbf{h}^{[t+1]}} L) + \mathbf{V}^T (\nabla_{\mathbf{y}^{[t]}} L) \tag{4}$$

#TODO ends

In the second part of the derivation of backpropagation equations, we derive $\nabla_{\mathbf{c}} L$, $\nabla_{\mathbf{b}} L$, $\nabla_{\mathbf{V}} L$, $\nabla_{\mathbf{W}} L$ and $\nabla_{\mathbf{U}} L$ respectively, in terms of hidden vectors $\mathbf{h}^{[t]}$, input vectors $\mathbf{x}^{[t]}$ and the gradients of the loss with respect to the output $\nabla_{\mathbf{y}^{[t]}} L$ and to the hidden vectors $\nabla_{\mathbf{h}^{[t]}} L$.

#TODO

$$\nabla_{\mathbf{c}} L = \sum_t \left(\frac{\partial \mathbf{y}^{[t]}}{\partial \mathbf{c}}\right)^T \nabla_{\mathbf{y}^{[t]}} L = \sum_t \nabla_{\mathbf{y}^{[t]}} L \tag{5}$$

$$\nabla_{\mathbf{b}} L = \sum_t \left(\frac{\partial \mathbf{h}^{[t]}}{\partial \mathbf{b}}\right)^T \nabla_{\mathbf{h}^{[t]}} L = \sum_t diag\left(1 - \left(\mathbf{h}^{[t]}\right)^2\right) \nabla_{\mathbf{h}^{[t]}} L \tag{6}$$

$$\nabla_{\mathbf{V}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial y_i^{[t]}}\right) \nabla_{\mathbf{V}^{[t]}} y_i^{[t]} = \sum_t (\nabla_{\mathbf{y}^{[t]}} L) \mathbf{h}^{[t]^T} \tag{7}$$

$$\nabla_{\mathbf{W}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{[t]}}\right) \nabla_{\mathbf{W}} h_i^{[t]} = \sum_t diag\left(1 - \left(\mathbf{h}^{[t]}\right)^2\right) (\nabla_{\mathbf{h}^{[t]}} L) \mathbf{h}^{[t-1]^T} \tag{8}$$

$$\nabla_{\mathbf{U}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{[t]}}\right) \nabla_{\mathbf{U}} h_i^{[t]} = \sum_t diag\left(1 - \left(\mathbf{h}^{[t]}\right)^2\right) (\nabla_{\mathbf{h}^{[t]}} L) \mathbf{x}^{[t]^T} \tag{9}$$

#TODO ends

In the second exercise, we are going to be estimating only the temperature value of the next hour from the given past 24 hours of weather-related information. Thus we will not be computing any intermediate output of RNN and only one scalar value at the final step. Additionally, we will use mean square error as a loss function, thus the loss function will be $L = \overline{(\hat{\mathbf{y}} - \mathbf{y})^2}$, where $\hat{\mathbf{y}}$ denotes the true labels. Given this information, simplify $\nabla_{\mathbf{h}^{[t]}} L$ for $t \in 0, ..., \tau$, $\nabla_{\mathbf{c}} L$ and $\nabla_{\mathbf{V}} L$ as further as possible.

#TODO

$$\nabla_{\mathbf{h}^{[\tau]}} L = \mathbf{V}^T (\nabla_{\mathbf{y}^{[\tau]}} L) = \mathbf{V}^T \frac{\partial L}{\partial y} = 2(\hat{y} - y)\mathbf{V}^T \tag{10}$$

$$\nabla_{\mathbf{h}^{[t]}} L = \left(\frac{\partial \mathbf{h}^{[t+1]}}{\partial \mathbf{h}^{[t]}}\right)^T (\nabla_{\mathbf{h}^{[t+1]}} L) = \mathbf{W}^T diag\left(1 - \left(\mathbf{h}^{[t+1]}\right)^2\right)(\nabla_{\mathbf{h}^{[t+1]}} L) \tag{11}$$

$$(\nabla_c L) = \sum_t \nabla_{\mathbf{y}^{[t]}} L = \nabla_{\mathbf{y}^{[\tau]}} L = \frac{\partial L}{\partial y} = 2(\hat{y} - y) \tag{12}$$

$$(\nabla_{\mathbf{V}} L) = \sum_t (\nabla_{\mathbf{y}^{[t]}} L) \mathbf{h}^{[t]^T} = 2(\hat{y} - y)\mathbf{h}^{[\tau]^T} \tag{13}$$

#TODO ends

Column number	Column name	Column description
1	No	Row number
2	year	Year
3	month	Month
4	day	Day
5	hour	Hour
6	pm2.5	Pollution in PM2.5 concentration
7	DEWP	Dew Point
8	TEMP	Temperature
9	PRES	Pressure
10	cbwd	Combined wind direction
11	Iws	Cumulated wind speed
12	Is	Cumulated hours of snow
13	Ir	Cumulated hours of rain

Exercise 2 - Manual Implementation of vanilla RNN

In this exercise, we will implement forward and backward propagation steps of the simple RNN and train it on a real data. We will stick to the notation the we used in the first part of the exercise.

Prepare the data

In this exercise we will develop a model that estimates the temperature of the following hour from different weather parameters in the last 24 hours.

The goal of the exercise is to reinforce our knowledge on RNNs by implementing the full learning cycle of a simple RNN with forward and backward propagation. Please have a look at the script at the end of this exercise if you want to understand the main frame of the exercise or if you get stuck during the exercise and want to get a hint regarding how we will be using the functions to be implemented in the exercise.

We will be using the dataset provided in https://raw.githubusercontent.com/jbrownlee/Datasets/master/pollution.csv. This is a dataset that reports on the weather and the level of pollution each hour for five years in Beijing, China. The file is already downloaded and can be found in the exercise folder with name "rnn_dataset". The dataset includes the features described in the table.

We read this file and print out the first rows and the dimensions of file. We will use DEWP, TEMP, PRES, cbwd, Iws, Is, Ir features as input and not the pollution, since pm2.5 contains some NA values we do not want to deal with. Save the corresponding columns of the file for these features and they will be used in the rest of the assignment.

```
csv_file <- read.csv(file = 'rnn_dataset.csv',nrows=43000, stringsAsFactors = FALSE)
head(csv_file)
```

```
##   No year month day hour pm2.5 DEWP TEMP PRES cbwd   Iws Is Ir
## 1  1 2010     1   1    0    NA  -21  -11 1021   NW  1.79  0  0
## 2  2 2010     1   1    1    NA  -21  -12 1020   NW  4.92  0  0
## 3  3 2010     1   1    2    NA  -21  -11 1019   NW  6.71  0  0
## 4  4 2010     1   1    3    NA  -21  -14 1019   NW  9.84  0  0
## 5  5 2010     1   1    4    NA  -20  -12 1018   NW 12.97  0  0
## 6  6 2010     1   1    5    NA  -19  -10 1017   NW 16.10  0  0
```

```
dim(csv_file)
```

```
## [1] 43000    13
```

```
# write code here
csv_file <- (csv_file[,c(7,8,9,11,12,13)])
# end of code

data <- array(0,dim=dim(csv_file))
for (i in 1:dim(data)[2]){
```

```
  data[,i] <- array(csv_file[[i]],dim=c(1,length(csv_file[[i]])))
}
```

Now we have the data. The next step is to divide the data into smaller sequences of 25, so that we can use the first 24 values of different features to estimate the following temperature value. While we are at dividing data into smaller sequences, we will also divide the data into mini-batches, so that we can feed different smaller batches to the network during training instead of feeding the whole dataset each time (remember that this would take too much time). We set the batch size to 40. Be aware that the value in the first dimension of the data correspond to batch number, the second dimension to the data sample number (in a given batch), the third value corresponds to the data point number in mini-sequences (max 25) and the last one to the feature number.

```
data <- array(data,c(25,43,40,(dim(data)[2])))
data <- aperm(data,c(2,3,1,4))
```

We split the data into training and test sets. The training set contains 80% of the whole data.

```
index<-sample(1:(dim(data)[2]),(dim(data)[2])*0.8,replace=FALSE)
data_train <- data[,index,,]
data_test <- data[,-index,,]
```

Now we will scale the data using min-max scaling, where the minimum value for each column will be 0 and the maximum value will be 1. Note that when we determine the minimum and maximum value of each feature, we should only consider the minimum and maximum values of features in the training subset. Because, the model should not leverage any information from the validation subset during the learning process.

```
# create the arrays that will save the min and max values in the training set for each features
min_array <- c()
max_array <- c()

# save the min and max values in the training set for each features
for (i in 1:dim(data_train)[4]){
  min_array <- c(min_array, min(data_train[,,,i]))
  max_array <- c(max_array, max(data_train[,,,i]))
}

# create the 4-D arrays that will save the scaled training and test sets
data_train_scaled <- array(0, dim=dim(data_train))
data_test_scaled <- array(0, dim=dim(data_test))

#save the scaled training and test sets
for (i in 1:length(min_array)){
  data_train_scaled[,,,i] <- (data_train[,,,i]-min_array[i])/(max_array[i]-min_array[i])
  data_test_scaled[,,,i] <- (data_test[,,,i]-min_array[i])/(max_array[i]-min_array[i])
}
```

Here, we create our train and test subsets with X being the feature matrix and Y being the target value. The first 24 datapoints (the weather features for 24 hours) for each data samples and features from the corresponding (training or validation) from scaled subsets will be assigned to X and the temperature value (one feature) of the last datapoint (the data for the following hour after 24 hours) will be assigned to Y.

```
estimate_variable <- 2

X_train_all <- data_train_scaled[,,1:(dim(data_train_scaled)[3]-1),]
Y_train_all <- data_train_scaled[,,dim(data_train_scaled)[3],estimate_variable]
X_test_all <- data_test_scaled[,,1:(dim(data_test_scaled)[3]-1),]
Y_test_all <- data_test_scaled[,,dim(data_test_scaled)[3],estimate_variable]
```

Implement RNN

Let's intialize U, W, V, b and c weights randomly and set values for hidden_state and the learning rate lr.

```
hidden_state = 10
U <- array(rnorm(dim(data)[4]*hidden_state, mean=0, sd=0.001), dim=c(hidden_state,dim(data)[4]))
W <- array(rnorm(hidden_state*hidden_state, mean=0, sd=0.001), dim=c(hidden_state,hidden_state))
b <- array(rnorm(hidden_state, mean=0, sd=0), dim=c(hidden_state,1))
V <- array(rnorm(hidden_state, mean=0, sd=0.001), dim=c(1,hidden_state))
c <- rnorm(1, mean=0, sd=0)
lr <- 0.02
```

Now we will define functions for forward propagation. First we will define a function which calculates the next hidden vector *h_next* from the previous hidden vector *h_prev* and input x using equation 1.

```
update_state <- function(x, h_prev, U, W, b){
  # Compute the following state from the previous hidden h_prev and current input (x),
  # using the input weights (U), recursive weights (W) and biases (b).

  # write code here
  h_next <- tanh(U %*% x + W %*% h_prev + b)
  # end of code
  return(h_next)
}
```

This function should compute all state vectors and record them into a matrix H. Return the hidden state activations in a matrix. H[x,y,:] contains the hidden vector corresponding to x^{th} data sample and is calculated after $(y-1)^{th}$ input vector is fed into the model. H[:,1,:] is set to the zero vector as an initial hidden state (the one before no input vector is processed). Hint: Use update_state function.

```
forward_states <- function(X, U, W, b){
  # Compute all state vectors and record them into a matrix H given the input matrix X,
  # input weights U, and recursive weights W and biases b.
  # Initialize the matrix H
  H <- array(0, dim=c(dim(X)[1],dim(X)[2]+1, dim(U)[1]))

  for (i in 1:(dim(X)[1])){
    for (k in 1:(dim(X)[2])){
      #write code here
      H[i,k+1,] <- update_state(X[i,k,], H[i,k,], U, W, b)
      #end of code
    }
  }
  return(H)
}
```

This function computes the estimated output value of RNN from the the final hidden state vectors using equation 2.

```
predict_output <- function(H_last, V, c){
  # Compute the output using the final hidden states in H_last, output matrix
  # weights (V) and biases (c).
  # H_last should have 2 dimensions, where the first dimension corresponds to the
  # data samples in the mini-batch, e.g. H_last[3,] is the last hidden vector of
  # 3rd data sample in the mini-batch.

  # create output vector, where each value in the vector is the calculated
  # prediction for one data sample input.
  y <- array(0, dim=dim(H_last)[1])
  for (i in 1:dim(H_last)[1]){
    # write code here
```

```
    y[i] <- c + V %*% H_last[i,]
    # end of code
  }
  return(y)
}
```

Calculate the mean-squared-error loss from the predicted output vector Y_est and the true output vector Y. The reason that they are vectors comes from that we compute by mini-batches, and thus each element of the vectors represents a prediction for one data sample or true label.

```
loss <- function(Y_est, Y){
  # write code here
  mse_loss <- mean((Y_est-Y)**2)
  # end of code
  return (mse_loss)
}
```

The output_gradient function calculates $\nabla_{\mathbf{h}^{[\tau]}} L$ the partial derivative of the loss function with respect to the final hidden vector. The first dimension denotes different data samples in grad_out matrix and each derivative vector for corresponding data sample is saved in the second dimension.

```
output_gradient <- function(y, t, V){
  # Gradient of the MSE loss function with respect to the output y.

  grad_out <- array(0, dim=c(length(t),length(V)))
  for (i in 1:length(t)){
  # write code here
    grad_out[i,] <- 2 * (y[i] - t[i]) * t(V)
  # end of code
  }
  return(grad_out)
}
```

With this function, we compute and save $\nabla_{\mathbf{h}^{[t]}} L$ for $t \in 0, ..., \tau$ to a 3-D array. As similarly as the function above, the first dimension of the array denotes the data sample number in the mini-batch. The second dimension denotes temporal indicator t, and each derivative vector for corresponding data sample is saved in the third dimension.

```
backward_gradient <- function(H, grad_out){
  # initialize 3-D array grad_over_time
  grad_over_time <- array(0, dim=c(dim(H)[1], dim(H)[2], dim(H)[3]))
  # assign the vectors in grad_out calculated in output_gradient function as the
  # last vectors in grad_over_time
  grad_over_time[,dim(grad_over_time)[2],] <- grad_out

  for (i in 1:dim(H)[1]){
    for (k in (dim(H)[2]):1){
      # Compute the gradients with respect to the previous hidden size vector
      # using gradients wrt next hidden vector
      # write code here
      grad_over_time[i,k-1,] <- t(W) %*% diag(1-(H[i,k,])**2) %*% grad_over_time[i,k,]
     # end of code
    }
  }
  return (grad_over_time)
}
```

"*backward_c*" function calculates the derivative of the loss function $\nabla_c L$ with respect to output bias of RNN c.

```
backward_c <- function(Y_est, Y){
    # Set the gradient accumulations to 0
    c_grad <- 0
    # compute the parameters for each data sample
    for (i in 1:(dim(Y_est)[1])){
        # Compute the parameter gradients and accumulate the results.
        # write code here
        c_grad <- c_grad + 2 * (Y_est[i] - Y[i])
        # end of code
    }
    return (c_grad)
}
```

"*backward_b*" function calculates the derivative of the loss function $\nabla_{\mathbf{b}} L$ with respect to hidden state bias of RNN $\mathbf{b}$.

```
backward_b <- function(H, grad_over_time){
   # Set the gradient accumulations to 0
   b_grad <- array(0,dim=c(dim(grad_over_time)[3],1))
    # compute the parameters for each data sample
   for (i in 1:(dim(grad_over_time)[1])){
     for (k in 2:(dim(grad_over_time)[2])){
       # Compute the parameter gradients and accumulate the results.
       # write code here
       b_grad <- b_grad + diag(1-(H[i,k,])**2) %*% grad_over_time[i,k,]
       # end of code
     }
   }
   return (b_grad)
}
```

"*backward_V*" function calculates the derivative of the loss function $\nabla_{\mathbf{V}} L$ with respect to $\mathbf{V}$.

```
backward_V <- function(Y_est, Y, last_H){

    # Set the gradient accumulations to 0
    V_grad <- array(0, dim = c(1,dim(last_H)[2]))

    for (i in 1:(dim(H)[1])){
        # Compute the parameter gradients and accumulate the results.
        # write code here
        V_grad <- 2 * (Y_est[i] - Y[i]) * array(last_H[i,],dim = c(1,dim(last_H)[2]))
        # end of code
    }
    return (V_grad)
}
```

"*backward_W*" function calculates the derivative of the loss function $\nabla_{\mathbf{W}} L$ with respect to $\mathbf{W}$.

```
backward_W <- function(H, grad_over_time){
    # Set the gradient accumulations to 0
    W_grad <- array(0,dim=c(dim(grad_over_time)[3],dim(grad_over_time)[3]))
    for (i in 1:(dim(H)[1])){
      for (k in (2:dim(H)[2])){
        # Compute the parameter gradients and accumulate the results.
        # write code here
        W_grad <- W_grad + diag(1-(H[i,k,])**2) %*%
        array(grad_over_time[i,k,], dim = c(dim(H)[3],1)) %*%
        array(H[i,k-1,],dim = c(1,dim(H)[3]))
        # end of code
```

```
      }
    }
    return (W_grad)
}
```

"*backward_U*" function calculates the derivative of the loss function $\nabla_{\mathbf{U}} L$ with respect to $\mathbf{U}$.

```
backward_U <- function(X, H, grad_over_time){

    # Set the gradient accumulations to 0
    U_grad <- array(0,dim=c(dim(grad_over_time)[3],dim(X)[3]))

    for (i in 1:(dim(X)[1])){
      for (k in (1:dim(X)[2])){
        # Compute the parameter gradients and accumulate the results.
        # write code here
        U_grad <- U_grad + diag(1-(H[i,k+1,])**2) %*%
        array(grad_over_time[i,k+1,], dim = c(dim(grad_over_time)[3],1)) %*%
        array(X[i,k,],dim = c(1,dim(X)[3]))
        # end of code
      }
    }
    return (U_grad)
}
```

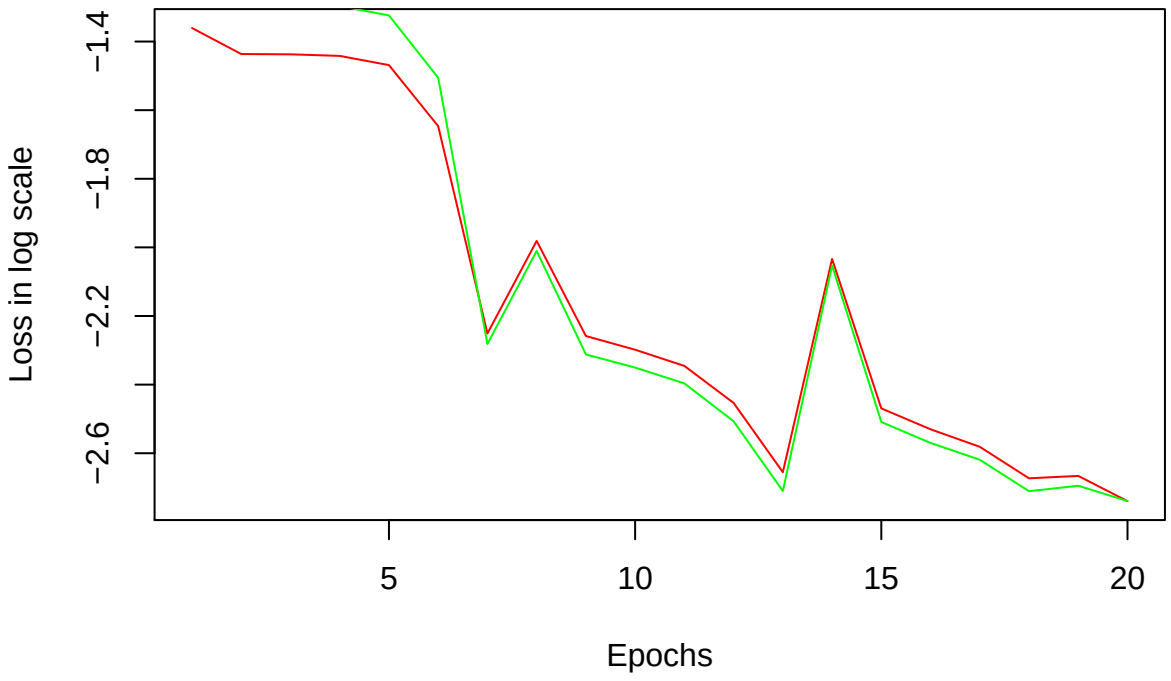
The function below is used to update any weight matrices or vectors $\mathbf{w}$ using gradient descent. The update is towards opposite direction of the loss gradient $\nabla_w L$ with respect to $\mathbf{w}$ and it weighted by the learning rate lr.

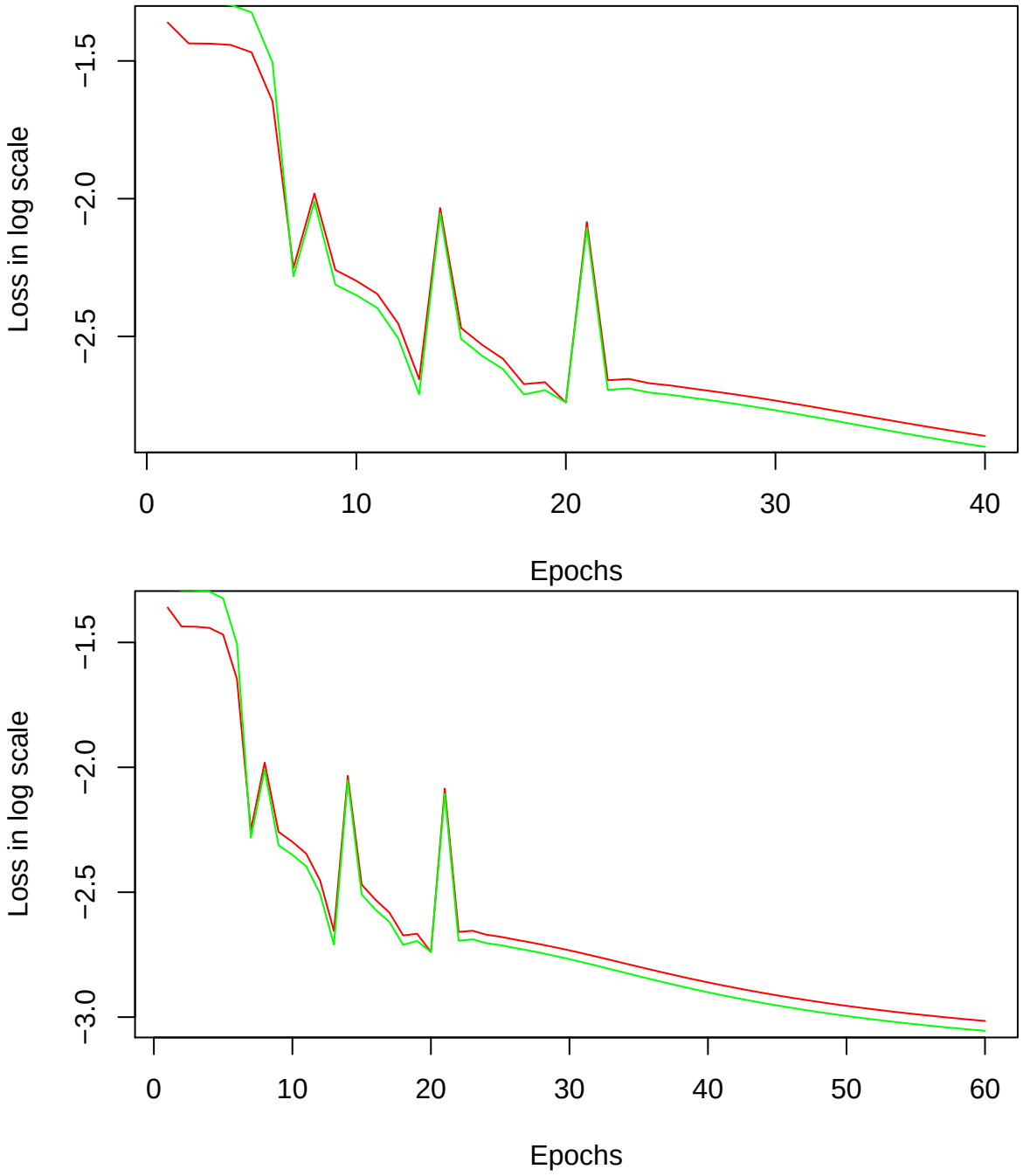
```
update_weights <- function(w, lr, w_grad){
  # write code here
  w <- w - lr * w_grad
  # end of code

  return(w)
}
```

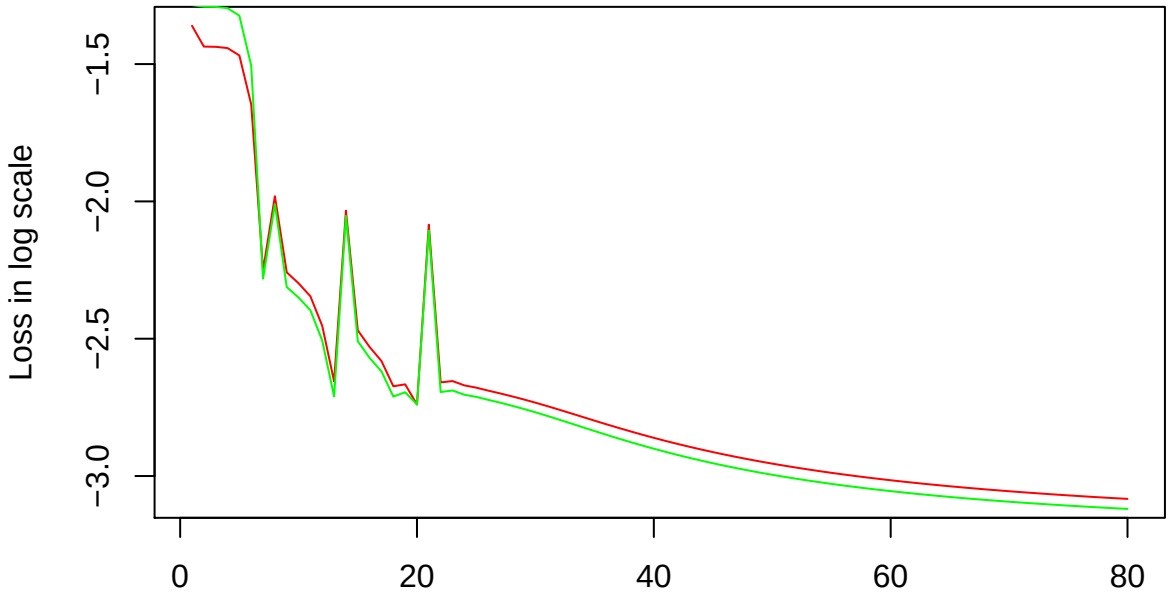
This is the main script of the exercise. We will make use the functions we have defined so far in this scipt. This is already written for you.

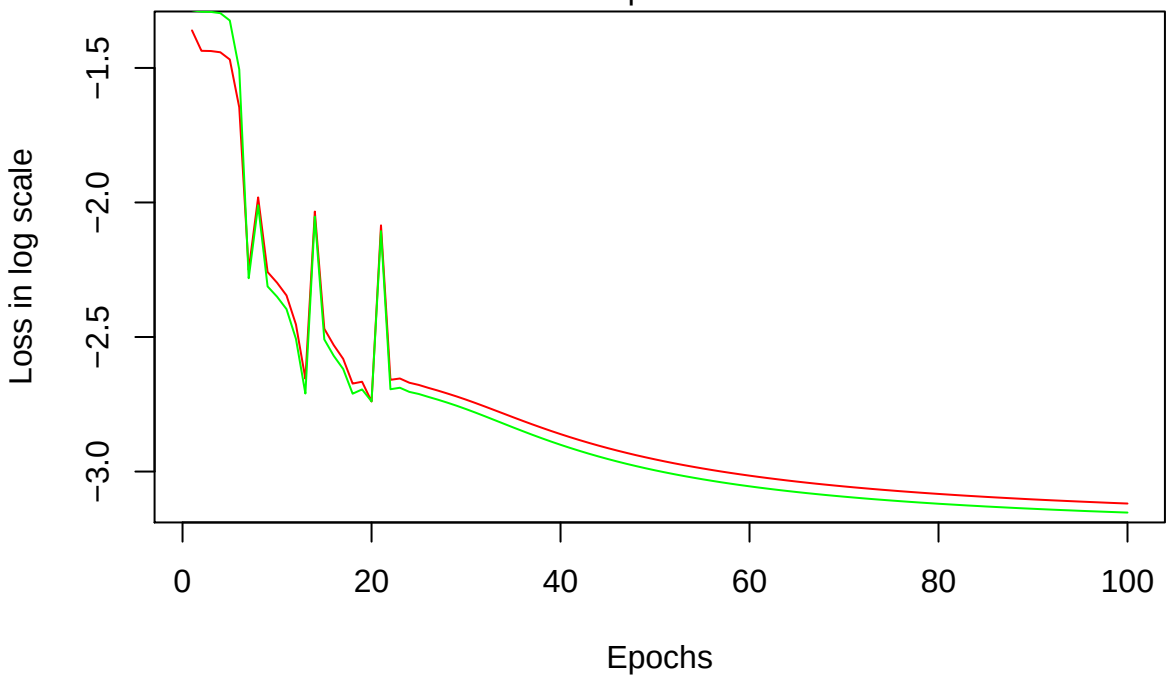
```
# create loss arrays, so that we can append the train and test losses to the array
loss_train_array <- c()
loss_test_array <- c()
for (t in 1:100){
  # iterations over every single mini-batch in dataset
  for (i in 1:(dim(data)[1])){
    # choose the corresponding mini-batches in whole sets for training and
    # validation sets besides for input values and target values.
    X_train <- X_train_all[i,,,]
    Y_train <- Y_train_all[i,]
    X_test <- X_test_all[i,,,]
    Y_test <- Y_test_all[i,]
    # calculate the hidden states for the mini-batch
    H <- forward_states(X_train, U, W, b)
    # calculate the estimated output values for input values in the mini-batch
    Y_est <- predict_output(H[,dim(H)[2],], V, c)
    # calculate gradients with respect to output and hidden states
    grad_out <- output_gradient(Y_est, Y_train, V)
    grad_over_time <- backward_gradient(H, grad_out)
    # calculate the loss gradients with respect to each trainable weights
```

```
    b_grad <- array(backward_b(H, grad_over_time), dim=c(length(b),1))
    W_grad <- backward_W(H, grad_over_time)
    U_grad <- backward_U(X_train, H, grad_over_time)
    c_grad <- backward_c(Y_est, Y_train)
    V_grad <- backward_V(Y_est, Y_train, H[,dim(H)[2],])
    # update the weights
    U <- update_weights(U, lr, U_grad)
    b <- update_weights(b, lr, b_grad)
    c <- update_weights(c, lr, c_grad)
    V <- update_weights(V, lr, V_grad)
    W <- update_weights(W, lr, W_grad)
    # calculate the train loss and save it
    loss_train <- loss(Y_est, Y_train)
    loss_train_array <- c(loss_train_array, loss_train)
    # forward propagation for test set and estimations for the test set inputs
    H <- forward_states(X_test, U, W, b)
    Y_est <- predict_output(H[,dim(H)[2],], V, c)
    # calculate the test loss and save it
    loss_test <- loss(Y_est, Y_test)
    loss_test_array <- c(loss_test_array,loss_test)
  }

  if (t%%20==0){
    # plot training and test loss in logarithmic scale
    # The mean value is calculated for each epoch (note that losses are so far
    # saved after each mini-batch)
    loss_train_array_averaged <- colMeans(matrix(loss_train_array, (dim(X_train_all)[1])))
    loss_test_array_averaged <- colMeans(matrix(loss_test_array, (dim(X_test_all)[1])))

    x=1:((length(loss_train_array_averaged)))
    plot(x,log10(loss_train_array_averaged),type="l",xlab="Epochs",
         ylab="Loss in log scale",col="red")
    lines(log10(loss_test_array_averaged),col="green")
  }
}
```




Loss in log scale
−1.5
−2.0
−2.5
0
10
20
30
40
Epochs
Loss in log scale
−1.5
−2.0
−2.5
−3.0
0
10
20
30
40
50
60
Epochs






```
#calculate the unscaled predicted output for test input and the target values and
#compare the first results between the predicted output and target values

Y_est_unscaled <- min_array[estimate_variable] +
Y_est*(max_array[estimate_variable]-min_array[estimate_variable])
Y_test_unscaled <- min_array[estimate_variable] +
Y_test*(max_array[estimate_variable]-min_array[estimate_variable])
head(Y_est_unscaled)
```

```
## [1] 26.4994073 10.7665142  2.3571391 32.6786862 -0.7295833  9.0695589
```

```
head(Y_test_unscaled)
```

```
## [1] 27 10  3 34 -1 11
```