Recurrent Neural Network

10/31/2025

The feed forward neural network works well for functions with static input variables. However, for sequential variables, especially for time series and natural languages (either text or speech), the inputs must be broken apart into segments and the each segment is then used as static input to a feed forward neural network. Recurrent neural network, on the other hand, allows an internal state to memorize the context of previous inputs while the network continuously takes in new external input one at a time. This greatly simplifies the network construction but may cause new problems in network training.

Network Architecture

A recurrent neural network (RNN) is a network with one or more connection loops inside the network, i.e., the content in a neuron (node) is dependent not only on the current input values, but the network previous output values as well. This dependence on network's previous states gives recurrent neural network memories that a traditional feed forward neural network lacks.

A simple recurrent neural network is similar to a feed forward neural network with a single hidden layer except that the output of the hidden layer is stored and then sent to the hidden layer along with the subsequent input to the network. Therefore, the combined output at time t of the hidden layer (h_t) becomes:

$$h_t = \sigma\left(\sum_i w_i^{(i)} \times X_{t,i} + \sum_j w_j^{(h)} \times h_{t-1,j}\right)$$

where σ is the activation function, X_t network input at time t, $w^{(i)}$ the weight for input signals, and $w^{(h)}$ the weight for the previous hidden layer output. The output of the neural network (y_t) is identical to that of a feed forward neural network:

$$y_t = \sigma\left(\sum_i w_i^{(o)} \times h_{t,i}\right)$$

where $w^{(o)}$ is the weight for network output.

Training Recurrent Neural Network

The most frequently used method for training a recurrent neural network is back propagation through time, or BPTT. This method recognizes that for a finite time period, there is an equivalent feed forward neural network for every recurrent neural network if one "unfolds" the recurrent network over time. In other words, the hidden layer loop input to a recurrent network at time t can be viewed as an identical network at time ($t-1$) and the loop input to that network comes from another identical network at time ($t-2$), so on and so forth.

The unfolded network can then be trained using the same back propagation algorithm as a feed forward network but over a time period from beginning to t. The updates of the weights are:

© All rights reserved, Orange County Science Institute, 2025

$$\Delta w^{(o)} = -\eta \times h_t \times \frac{\partial \mathcal{L}}{\partial y_t}$$

$$\Delta w^{(h)} = -\eta \times h_{(t-1)} \times \left(\frac{\partial \mathcal{L}}{\partial w^{(i)}}_{t-1} \times w^{(h)} + \frac{\partial \mathcal{L}}{\partial y}_{t} \times w^{(o)} \right) \times \sigma^{'}(h_t)$$

$$\Delta w^{(i)} = -\eta \times X_t \times \left(\frac{\partial \mathcal{L}}{\partial w^{(i)}}_{t-1} \times w^{(h)} + \frac{\partial \mathcal{L}}{\partial y}_{t} \times w^{(o)} \right) \times \sigma^{'}(h_t)$$

where

$$\frac{\partial \mathcal{L}}{\partial y_t} = (y_{truth} - y_t) \times \sigma^{'}(y_t)$$

$$\frac{\partial \mathcal{L}}{\partial w^{(i)}}_{t-1} = \left(\frac{\partial \mathcal{L}}{\partial w^{(i)}}_{t-2} \times w^{(h)} + \frac{\partial \mathcal{L}}{\partial y}_{t-1} \times w^{(o)} \right) \times \sigma^{'}(h_{t-1})$$

Training Dataset

The training dataset in the following example is designed for binary addition using an RNN. Two random integers are selected as number a_int and b_int and the decimal numbers are converted to 8-bit binary numbers as a and b. The sum of these two integers is also converted into an 8-bit binary number as c (see Figure 1). The binary numbers are grouped one bit at a time as the input variable X and the truth value y where p is used in a for loop to pick the bit from the binary numbers.

```
 bin_dim=8;  max_no=2^bin_dim
a_int = sample(1:(max_no/2), 1)
a = int2bin(a_int)
b_int = sample(1:(max_no/2), 1)
b = int2bin(b_int)
c = int2bin(a_int + b_int)

X = cbind(a[bin_dim - p],b[bin_dim - p])
y = c[bin_dim - p]
```

Figure 1, Training Dataset for Binary Addition using RNN.

Forward Propagation

To predict the binary sum from the input variables a and b, the bits of the two binary numbers are multiplied by the input weight, w_0, plus the matrix product of previous hidden layer vector and the hidden layer weight, w_h. After activation, this vector becomes the hidden layer output vector. This vector is then multiplied by output weight, w_1. Therefore, lyr-2 is the RNN predicted output y.

```
lyr_1 = sigma((X%*%w_0)
                + (lyr_1_val[dim(lyr_1_val)[1],] %*% w_h))
 lyr_2 = sigma(lyr_1 %*% w_1)
```

Figure 2, Forward Propagation of RNN and lyr_2 as the Prediction.

© All rights reserved, Orange County Science Institute, 2025

Back Propagation

The back propagation follows the mathematics as shown on the previous page. The R code is shown in Figure 3.

```
lyr_2_error = y - lyr_2
lyr_2_dels = rbind(lyr_2_dels, lyr_2_error * sigma_d(lyr_2))
lyr_2_delta = lyr_2_dels[dim(lyr_2_dels)[1]-p,]
lyr_1_delta = (future_lyr_1_delta %*% t(w_h)
               + lyr_2_delta %*% t(w_1)) * sigma_d(lyr_1)

w_1_new = w_1_new + matrix(lyr_1) %*% lyr_2_delta
w_h_new = w_h_new + matrix(prev_lyr_1) %*% lyr_1_delta
w_0_new = w_0_new + t(X) %*% lyr_1_delta
```

Figure 3, Backward Propagation of RNN and Weight Updates after All Bits are Accounted for.

Experiments

Figure 4 shows the prediction error as progression in the training session. The number of hidden nodes is 8 and the learning rate is 0.8. The error at 5000th training epoch is 0.127.   With 16 hidden layer nodes, the error at 5000th training epoch increased to 0.173. This error becomes 0.152 with 32 hidden layer nodes. At 64 hidden layer nodes, this error reduced to 0.126. Therefore, the number of hidden layer nodes does not have a significant effect on the prediction accuracy.

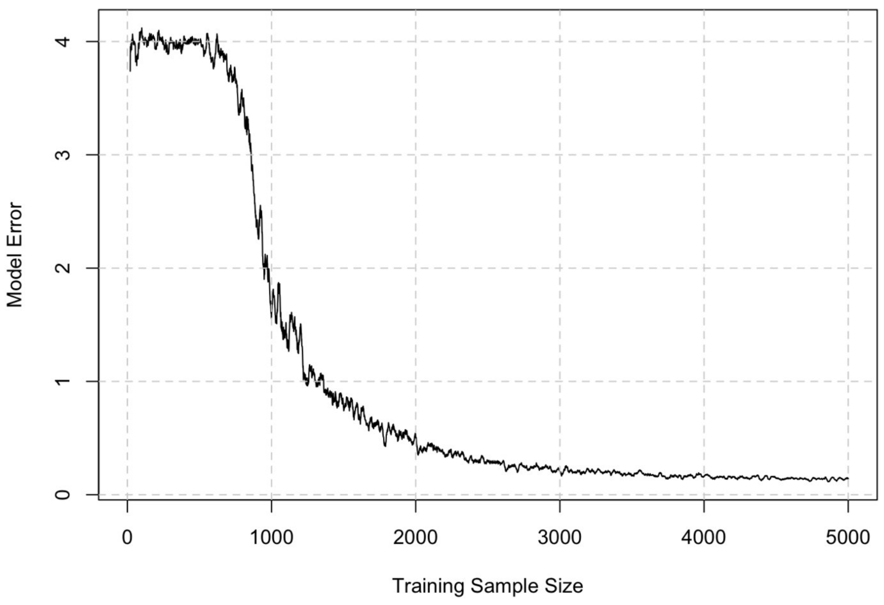

Figure 4, Training Progression for an RNN with 8 hidden nodes and at a learning rate of 0.8.

At learning rate 1.0 the error at 5000th training epoch reduced to 0.121. Further increases in learning rate to 2.0 (error 0.077), 3.0 (error 0.076), and 4.0 (error 0.044) reduces the error even more. At 5.0, the error went up again. Therefore, an optimal learning rate must be selected carefully.

© All rights reserved, Orange County Science Institute, 2025

Appendix R Script

```r
set.seed(100); n_train=5000; del=0; bin_dim=8;  max_no=2^bin_dim
int2bin = function(x){tail(rev(as.integer(intToBits(x))), bin_dim)}
sigma = function(x){1 / (1+exp(-x))}
sigma_d = function(x){x*(1-x)}

# initialization
  l_rate=0.8;in_dim=2;hd_dim=64;out_dim=1
  w_0 = matrix(runif(n = in_dim*hd_dim, min=-1, max=1), nrow=in_dim)
  w_1 = matrix(runif(n = hd_dim*out_dim, min=-1, max=1), nrow=hd_dim)
  w_h = matrix(runif(n = hd_dim*hd_dim, min=-1, max=1), nrow=hd_dim)
  w_0_new = matrix(0, nrow = in_dim, ncol = hd_dim)
  w_1_new = matrix(0, nrow = hd_dim, ncol = out_dim)
  w_h_new = matrix(0, nrow = hd_dim, ncol = hd_dim)

# training
for (j in 1:n_train) {
  a_int = sample(1:(max_no/2), 1)
  a = int2bin(a_int)
  b_int = sample(1:(max_no/2), 1)
  b = int2bin(b_int)
  c = int2bin(a_int - b_int)
  d = matrix(0, nrow = 1, ncol = bin_dim)
  total_error = 0
  lyr_2_dels = matrix(0)
  lyr_1_val = matrix(0, nrow=1, ncol = hd_dim)

  for (p in 0:(bin_dim-1)) {
    X = cbind(a[bin_dim - p],b[bin_dim - p])
    y = c[bin_dim - p]
    lyr_1 = sigma((X%*%w_0)
                + (lyr_1_val[dim(lyr_1_val)[1],] %*% w_h))
    lyr_2 = sigma(lyr_1 %*% w_1)
    lyr_2_error = y - lyr_2
    lyr_2_dels = rbind(lyr_2_dels, lyr_2_error * sigma_d(lyr_2))
    total_error = total_error + abs(lyr_2_error)
    d[bin_dim - p] = round(lyr_2)
    lyr_1_val = rbind(lyr_1_val, lyr_1)
  }
  future_lyr_1_delta = matrix(0, nrow = 1, ncol = hd_dim)

  for (p in 0:(bin_dim-1)) {
    X = cbind(a[p+1], b[p+1])
    lyr_1 = lyr_1_val[dim(lyr_1_val)[1]-p,]
    prev_lyr_1 = lyr_1_val[dim(lyr_1_val)[1]-(p+1),]
    lyr_2_delta = lyr_2_dels[dim(lyr_2_dels)[1]-p,]
    lyr_1_delta = (future_lyr_1_delta %*% t(w_h)
               + lyr_2_delta %*% t(w_1)) * sigma_d(lyr_1)
    w_1_new = w_1_new + matrix(lyr_1) %*% lyr_2_delta
    w_h_new = w_h_new + matrix(prev_lyr_1) %*% lyr_1_delta
    w_0_new = w_0_new + t(X) %*% lyr_1_delta
    future_lyr_1_delta = lyr_1_delta
  }
  w_0 = w_0 + ( w_0_new * l_rate )
  w_1 = w_1 + ( w_1_new * l_rate )
  w_h = w_h + ( w_h_new * l_rate )
  w_0_new = w_0_new * 0
  w_1_new = w_1_new * 0
  w_h_new = w_h_new * 0
  del=c(del,total_error)
}
## plot the training progress
plot(filter(del, filter=rep(1/20, 20), method="convolution", sides=1),
     type="l", ylab="Model Error", xlab="Training Sample Size"); grid(lty=2)
cat(del[5000])
```

© All rights reserved, Orange County Science Institute, 2025