BackPropagation Through Time

Jiang Guo

2013.7.20

Abstract

This report provides detailed description and necessary derivations for the BackPropagation Through Time (BPTT) algorithm. BPTT is often used to learn recurrent neural networks (RNN). Contrary to feed-forward neural networks, the RNN is characterized by the ability of encoding longer past information, thus very suitable for sequential models. The BPTT extends the ordinary BP algorithm to suit the recurrent neural architecture.

1 Basic Definitions

For a two-layer feed-forward neural network, we notate the input layer as $\mathbf{x}$ indexed by variable i, the hidden layer as $\mathbf{s}$ indexed by variable j, and the output layer as $\mathbf{y}$ indexed by variable k. The weight matrix that map the input vector to the hidden layer is $\mathbf{V}$, while the hidden layer is propagated through the weight matrix $\mathbf{W}$, to the output layer. In a simple recurrent neural network, we attach every neural layer a time subscript t. The input layer consists of two components, $\mathbf{x(t)}$ and the privious activation of the hidden layer $\mathbf{s}(t-1)$ indexed by variable h. The corresponding weight matrix is $\mathbf{U}$.

Table 1 lists all the notations used in this report:

Neural layer	Description	Index variable
$x(t)$	input layer	i
$s(t-1)$	previous hidden (state) layer	h
$s(t)$	hidden (state) layer	j
$y(t)$	output layer	k
Weight matrix	**Description**	**Index variables**
V	Input layer $\rightarrow$ Hidden layer	i, j
U	Previous hidden layer $\rightarrow$ Hidden layer	h, j
W	Hidden layer $\rightarrow$ Output layer	j, k

Table 1: Notations in the recurrent neural network.

Then, the recurrent neural network can be processed as the following:

- Input layer $\rightarrow$ Hidden layer

$$s_j(t) = f(net_j(t)) \tag{1}$$

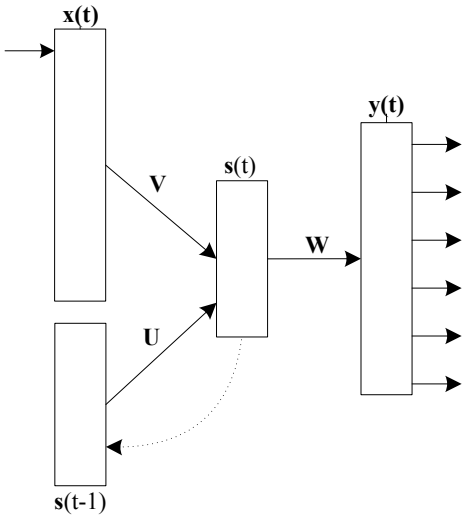

Figure 1: A simple recurrent neural network.

$$net_j(t) = \sum_i^l x_i(t)v_{ji} + \sum_h^m s_h(t-1)u_{jh} + b_j \tag{2}$$

- Hidden layer → Output layer

$$y_k(t) = g(net_k(t)) \tag{3}$$

$$net_k(t) = \sum_j^m s_j(t)w_{kj} + b_k \tag{4}$$

Where f and g are the activate functions in the hidden layer and output layer respectively. The activate function holds the non-linearities of the entire neural network, which greatly improves its expression power. b_j and b_k are the biases.

The architecture of a recurrent neural network is shown in Figure 1. What we need to learn is the three weight matrices U, V, W, as well as the two biases b_j and b_k.

2 BackPropagation

2.1 Error Propagation

Any feed-forward neural networks can be trained with backpropagation (BP) algorithm, as long as the cost function are defferentiable. The most frequently used cost function is the summed squared error (SSE), defined as:

$$C = \frac{1}{2}\sum_p^n \sum_k^o (d_{pk} - y_{pk})^2 \tag{5}$$

where d is the desired output, n is the total number of training samples and o is the number of output units.

BP is actually a propagated gradient descent algorithm, where gradients are propagated backward, leading to very efficient computing of the higher layer weight change. According to the gradient descent, each weight change in the network should be proportional to the negative gradient of the cost function, with respect to the specific weight:

$$\Delta(w) = -\eta \frac{\partial(C)}{\partial(w)} \tag{6}$$

Form of the cost function can be very complicated due to the hierarchical structure of the neural network. Hence the partial gradient for higher layer weights is intuitively not easy to calculate. Here we will show how to efficiently obtain the gradients for weights in every layer, by using the chain rule.

Note that the complexity mainly comes from the non-linearity of the activate function, so we would like to firstly seperate the linear and the non-linear parts in the gradient associated with each neural unit.

$$\Delta(w) = -\eta \frac{\partial(C)}{\partial(net)} \frac{\partial(net)}{\partial(w)} \tag{7}$$

where *net* represents the linear combination of the inputs, and thus $\frac{\partial(net)}{\partial(w)}$ is easy to compute[1]. What we mainly need to care about is $\frac{\partial(C)}{\partial(net)}$. We notate $\delta = -\frac{\partial(C)}{\partial(net)}$, as the **error** (vector) for each node.

For output nodes:

$$\delta_{pk} = -\frac{\partial(C)}{\partial(y_{pk})} \frac{\partial(y_{pk})}{\partial(net_{pk})} = (d_{pk} - y_{pk}) g^{'}(net_{pk}) \tag{8}$$

For hidden nodes:

$$\delta_{pj} = -(\sum_{k}^{o} \frac{\partial(C)}{\partial(y_{pk})} \frac{\partial(y_{pk})}{\partial(net_{pk})} \frac{\partial(net_{pk})}{\partial(s_{pj})}) \frac{\partial(s_{pj})}{\partial(net_{pj})} = \sum_{k}^{o} \delta_{pk} w_{kj} f^{'}(net_{pj}) \tag{9}$$

We could see that all output units contribute to the **error** of each hidden unit. Therefore it is simply a weighted sum operation to propagate the errors backward.

The weight change can then be simply computed by:

$$\Delta w_{kj} = \eta \sum_{p}^{n} \delta_{pk} s_{pj} \tag{10}$$

for hidden $\rightarrow$ output weights W, and

$$\Delta v_{ji} = \eta \sum_{p}^{n} \delta_{pj} x_{pi} \tag{11}$$

for input $\rightarrow$ hidden weights V. The recurrent weights[2] change is:

[1] exactly the input (vector) of that neuron, in other words, it is the activations of the previous layer.

[2] we notate the weights from the previous hidden layer U to hidden layer as recurrent weights.

$$\Delta u_{ji} = \eta \sum_{p}^{n} \delta_{pj} s_{ph}(t-1) \tag{12}$$

2.2 Activate function and cost function

The sigmoid function (also known as logistic function) is often used as the activate funciton, due to its elegant form of derivative.

$$y = g(net) = \frac{1}{1+e^{-net}} \tag{13}$$

$$g^{'}(net) = y(1-y) \tag{14}$$

In fact, for any probability function from the exponential family of probability distributions (such as softmax):

$$g(net_k) = \frac{e^{net_k}}{\sum_q e^{net_q}} \tag{15}$$

Equation 14 holds.

Of course this is not the exclusive reason that makes sigmoid function so attactive. It for sure has some other good properties. But to my knowledge, this is the most facinating one.

The cost function can be any differentiable function that is able to measure the loss of the predicted values from the gold answers. The SSE loss mentioned before is frequently-used, and works well in the training of conventional feed-forward neural networks.

For recurrent neural works, another appropriate cost function is the so-called cross-entropy:

$$C = -\sum_{p}^{n}\sum_{k}^{o} d_{pk} \ln y_{pk} + (1-d_{pk}) \ln(1-y_{pk}) \tag{16}$$

The cross-entropy loss is used in Recurrent Neural Network Language Models (RNNLM) and performs well [2]. With sigmoid or softmax as the activate funciton, the cross-entropy loss holds good properties:

$$\delta_{pk} = -\frac{\partial(C)}{\partial(y_{pk})}\frac{\partial(y_{pk})}{\partial(net_{pk})} = d_{pk} - y_{pk} \tag{17}$$

Consequently, the weight change of W can be written as:

$$\Delta w_{kj} = \eta \sum_{p}^{n} \delta_{pk} s_{pj} = \eta \sum_{p}^{n} (d_{pk} - y_{pk}) s_{pj} \tag{18}$$

The **error** (vector) is then propagated backward recursively, as illustrated in Equation 9. Input weights V and recurrent weights U can then be updated accordingly.

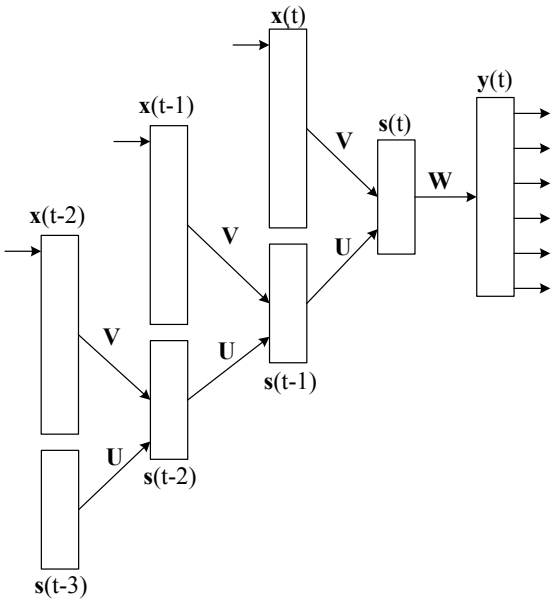

Figure 2: An unfolded recurrent neural network.

3 BackPropagation Through Time

In a recurrent neural network, errors can be propagated further, i.e. more than 2 layers, in order to capture longer history information. This process is usually called *unfolding*. An unfolded RNN is shown in Figure 2.

In an unfolded RNN, the recurrent weight is duplicated spatially for an arbitrary number of time steps, here refered to as τ. In accordance with Equation 9, errors are thus propagated backward as:

$$\delta_{pj}(t-1) = \sum_{h}^{m} \delta_{ph}(t) u_{hj} f^{'}(s_{pj}(t-1)) \tag{19}$$

where h is the index for the hidden node at time step t, and j for the hidden node at time step $t-1$. The error deltas of higher layer weights can then be calculated recursively.

After all error deltas have been obtained, weights are folded back adding up to one big change for each unfolded weights.

3.1 Matrix-vector notations

For concision, here we do not take the index of training samples p into our notations. Then the error deltas of the output layer are represented by:

$$\mathbf{e}_o(t) = \mathbf{d}(t) - \mathbf{y}(t) \tag{20}$$

The output weight matrix W are updated as:

$$\mathbf{W}(t+1) = \mathbf{W}(t) + \eta \mathbf{s}(t) \mathbf{e}_o(t)^T \tag{21}$$

Next, gradients of the errors are propagated from the output layer to the hidden layer:

$$\mathbf{e}_h(t) = d_h(\mathbf{e}_o(t)^T\mathbf{V}, t) \tag{22}$$

where the error vector is obtained using function $d_h(\cdot)$ that is applied element-wise:

$$d_{hj}(x,t) = xf^{'}(net_j) \tag{23}$$

Weights V between the input layer and hidden layer are then updated as

$$\mathbf{V}(t+1) = \mathbf{V}(t) + \eta\mathbf{x}(t)\mathbf{e}_h(t)^T \tag{24}$$

The recurrent weight W are updated as

$$\mathbf{U}(t+1) = \mathbf{U}(t) + \eta\mathbf{s}(t-1)\mathbf{e}_h(t)^T \tag{25}$$

Above demonstrates the matrix-vector notations of conventional BP algorithm for recurrent neural networks. For BPTT training, error propagation is done recursively:

$$\mathbf{e}_h(t-\tau-1) = d_h(\mathbf{e}_h(t-\tau)\mathbf{U}, t-\tau-1) \tag{26}$$

then the weight matrices V and U are updated as:

$$\mathbf{V}(t+1) = \mathbf{V}(t) + \eta\sum_{z=0}^{T}\mathbf{x}(t-z)\mathbf{e}_h(t-z)^T \tag{27}$$

$$\mathbf{U}(t+1) = \mathbf{U}(t) + \eta\sum_{z=0}^{T}\mathbf{s}(t-z-1)\mathbf{e}_h(t-z)^T \tag{28}$$

4 Discussion

The unfolded recurrent neural network can be seen as a deep neural network, except that the recurrent weights are tied. Consequently, in BPTT training, the weight changes at each recurrent layer should be added up to one big change, in order to keep the recurrent weights consistent. A similar algorithm is the so-called BackPropagation Through Time (BPTS) algorithm, which is used for training recursive neural networks [1]. Actually, in specific tasks, the recurrent/recursive weights can also be untied. In that case, much more parameters are to be learned.

References

[1] C. Goller and A. Kuchler. Learning task-dependent distributed representations by backpropagation through structure. In *Neural Networks, 1996., IEEE International Conference on*, volume 1, pages 347–352. IEEE, 1996.

[2] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, pages 1045–1048, 2010.