Note on Learning Neural Network

Wang Jian

Version 0.3 from 03/11/2015

Revision History

Revision	Date	Author(s)	Description
0.1	12/14/2014	Wang Jian	Basic backpropagation
0.2	01/11/2015	Wang Jian	RTRL for RNN
0.3	03/11/2015	Wang Jian	BPTT for RNN

Contents

1 Preliminary

1.1 Non-linear function

1.1.1 Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{1.1}$$

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x)) \tag{1.2}$$

1.1.2 Hyperbolic tangent

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \tag{1.3}$$

$$\frac{d\tanh(x)}{dx} = 1-\tanh^{2}(x) \tag{1.4}$$

1.1.3 Softmax

$$S_i(\boldsymbol{z}) = \frac{e^{z_i}}{\sum_k e^{z_k}} \tag{1.5}$$

$$\frac{\partial \log(S_i(\boldsymbol{z}))}{\partial z_j} = \delta_{ij} - S_i(\boldsymbol{z}) \tag{1.6}$$

where δ_{ij} is Kronecker delta,

$$\delta_{ij} = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

1.2 Cross entropy

$$J(\boldsymbol{w}) = -\sum_k t_k \log z_k \tag{1.7}$$

1.3 Gradient descent

In order to find a local minimum of a convex function, we can iteratively update the parameters using the following equation,

$$\Delta \boldsymbol{w} = -\alpha \frac{\partial J}{\partial \boldsymbol{w}} \tag{1.8}$$

where, α is the learning rate, controlling step size of every iteration.

1.3.1 Batch Gradient Descent

For data set with many examples, we can apply Equation 1.8 by simply adding up all examples,

$$\Delta \boldsymbol{w} = -\alpha \sum_{r} \frac{\partial J^r}{\partial \boldsymbol{w}} \tag{1.9}$$

where, r runs through all examples in data set. The algorithm for BGD is shown in algorithm 1.

Algorithm 1: Batch Gradient Descent

Input: N: number of iterations, R: number of examples
for $n = 1 : N$ **do**
  $\Delta \boldsymbol{w} = 0$
  for $r = 1 : R$ **do**
    $\Delta \boldsymbol{w}$ += $-\alpha \frac{\partial J^r}{\partial \boldsymbol{w}}$
  end
  $\boldsymbol{w}$ += $\Delta \boldsymbol{w}$
end

1.3.2 Stochastic Gradient Descent

BGD have to run through all the samples to do a single update, for large data set this may cause problems. To make the updating faster, we can do the update

with every one example, which leads to the SGD algorithm, shown in algorithm 2.

Algorithm 2: Stochastic Gradient Descent

Input: N: number of iterations, R: set of examples
for $n = 1 : N$ **do**
  for $r = UniqRandom(R)$ **do**
    $\Delta \boldsymbol{w} = -\alpha \frac{\partial J^r}{\partial \boldsymbol{w}}$
    $\boldsymbol{w}$ += $\Delta \boldsymbol{w}$
  end
end

where the *UniqRandom* function randomly and not repeatedly chooses one example from data set.

SGD often converges much faster compared to BGD but the error function is not as well minimised as in the case of BGD. To compromise, we can use a method called Mini-Batch SGD, which uses a small subset instead the whole training set to do the update, shown in algorithm 3.

Algorithm 3: Mini-Batch Stochastic Gradient Descent

Input: N: number of iterations, B: set of mini-batch, R: set of examples
for $n = 1 : N$ **do**
  for $B = UniqRandomSet(R)$ **do**
    $\Delta \boldsymbol{w} = 0$
    for $r = UniqRandom(B)$ **do**
      $\Delta \boldsymbol{w}$ += $-\alpha \frac{\partial J^r}{\partial \boldsymbol{w}}$
    end
    $\boldsymbol{w}$ += $\Delta \boldsymbol{w}$
  end
end

As we can see, batch gradient descent is deterministic, which means that every time you run BGD for a given training set, you will get the same optimum in the same number of iterations. Stochastic gradient descent is, however, stochastic. Because you are no longer using your entire training set a once, and instead picking one or more examples at a time in some likely random fashion, each time you tun SGD you will obtain a different optimum and a unique cost vs. iteration history.

1.4 The Multivariable Chain Rule

Multivariable Chain Rules allow us to differentiate with respect to any of the variables involved. One can use the *variable-dependence diagram* as a simple way

to apply this Chain Rule. we can compute the derivatives by simply adding up all paths starting at the dependent variables and ending at the independent variables, multiplying derivatives along each path.

For example, suppose that we have $x = f_1(u)$, $x = f_2(v)$,$y = f_3(u)$,$y = f_4(v)$,$z = f_5(x)$ and $z = f_6(y)$, Figure 1 shows the diagram,

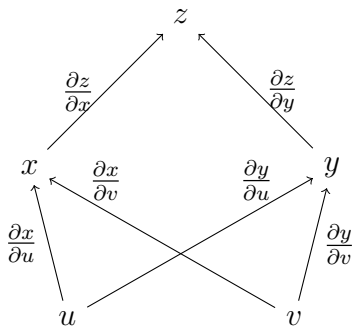

Figure 1: Variable-dependence diagram

Then we can get the derivatives,

$$\frac{\partial z}{\partial u} = \frac{\partial z}{\partial x}\frac{\partial x}{\partial u} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial u}$$

$$\frac{\partial z}{\partial v} = \frac{\partial z}{\partial x}\frac{\partial x}{\partial v} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial v}$$

1.5 Network architecture

For the network discussed in this paper, the *activation function* of input and hidden layer is sigmoid or hyperbolic tangent, the activation function of output layer is softmax, and the *loss function* is cross entropy.

The target vector is 1-of-V coding, i.e.

$$t_k = \begin{cases} 1 & k = k_0 \\ 0 & k \neq k_0 \end{cases}$$

then the loss function (1.7) becomes:

$$J(\boldsymbol{w}) = -\sum_k t_k \log z_k = -\log z_{k_0} \tag{1.10}$$

Throughout this paper, x_i denotes the input, y_j denotes hidden layer activation, z_k denotes output layer activation. Moreover, I is the input layer size ant $\mathcal{I}$ is the set of indices for input layer, H is the hidden layer size and $\mathcal{H}$ is the set of indices for hidden layer, O is the output layer size and $\mathcal{O}$ is the set of indices for output layer. To simplify notations, whenever involving summation, i, j and k always runs through the proper set.

2 Feed-forward Network

A *Feed-forward Neural Network*[1] is an artificial neural network where connections between the units do not form a directed cycle.

2.1 Forward pass

Input layer to hidden layer:

$$net_j^h = \sum_i w_{ji} x_i \tag{2.1}$$

$$y_j = \sigma(net_j^h) \tag{2.2}$$

Hidden layer to output layer:

$$net_k^o = \sum_j w_{kj} y_j \tag{2.3}$$

$$z_k = S_k(\boldsymbol{net^o}) \tag{2.4}$$

2.2 Backpropagation

2.2.1 Weight between hidden layer and output layer

The variable-dependence diagram for differentiating J w.r.t. w_{kj} is shown in Figure 2,

So the derivative of w_{kj} is

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k^o} \frac{\partial net_k^o}{\partial w_{kj}} \tag{2.5}$$

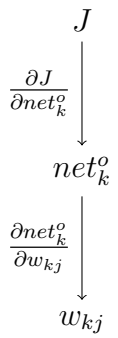

Figure 2: Variable-dependence diagram for w_{kj}

for the first term on the right side:

$$\begin{aligned}
\frac{\partial J}{\partial net_k^o} = \frac{\partial(-\log z_{k_0})}{\partial net_k^o} = \frac{\partial(-\log S_{k_0}(\boldsymbol{net^o}))}{\partial net_k^o} & \quad \text{substituting (1.10) and (2.4)} \\
= \delta_{kk_0} - S_{k_0}(\boldsymbol{net^o}) & \quad \text{from (1.6)} \\
= \delta_{kk_0} - z_{k_0} & \quad \text{from (2.4)} \qquad (2.6)
\end{aligned}$$

and the second term can be deduced by differentiating (2.3) :

$$\frac{\partial net_k^o}{\partial w_{kj}} = y_j \tag{2.7}$$

We can define the *error* of output layer is

$$e_k \equiv \delta_{kk_0} - z_{k_0} \tag{2.8}$$

then, we get

$$\frac{\partial J}{\partial w_{kj}} = e_k y_j \tag{2.9}$$

2.2.2 Weight between input layer and hidden layer

Figure 3 shows the variable-dependence diagram for differentiating J w.r.t. w_{ji}, The derivative of w_{ji} is

$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial w_{ji}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial net_j^h}\frac{\partial net_j^h}{\partial w_{ji}} \tag{2.10}$$

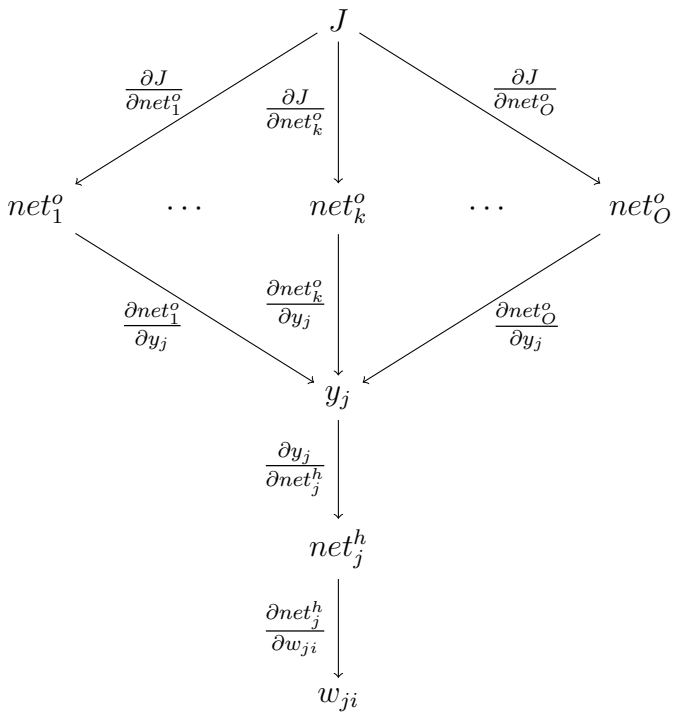

Figure 3: Variable-dependence diagram for w_{ji}

where, $\frac{\partial J}{\partial y_j}$ is

$$\begin{aligned}
\frac{\partial J}{\partial y_j} &= \sum_k \frac{\partial J}{\partial net_k^o}\frac{\partial net_k^o}{\partial y_j} \\
&= \sum_k (\delta_{kk_0} - z_{k_0})\frac{\partial net_k^o}{\partial y_j} && \text{substituting (2.6)} \\
&= \sum_k (\delta_{kk_0} - z_{k_0}) w_{kj} && \text{take partial derivative} \\
&= \sum_k e_k w_{kj} && \text{substituting (2.8)} \qquad (2.11)
\end{aligned}$$

make use of (1.2) and (2.2)

$$\frac{\partial y_j}{\partial net_j^h} = \sigma'(net_j^h) = y_j(1 - y_j) \qquad (2.12)$$

and

$$\frac{\partial net_j^h}{\partial w_{ji}} = x_i \tag{2.13}$$

Similarly, we define the error of hidden layer is

$$e_j \equiv \sigma'(net_j^h) \sum_k e_k w_{kj} \tag{2.14}$$

so that the gradient of w_{ji} takes the same form as w_{kj},

$$\frac{\partial J}{\partial w_{ji}} = e_j x_i \tag{2.15}$$

Thus, for the non-output layer, calculating weight update involves: a) the input x_i; b) the derivative of output $\sigma'(net_j^h)$; and c) the error back propagating from upper layer $\sum_k e_k w_{kj}$.

3 Recurrent Neural Network

Recurrent Neural Network[2] is a neural network where connections between hidden layer units form directed cycle.

3.1 Forward pass

Input layer to hidden layer:

$$net_j^h(t) = \sum_i w_{ji} x_i(t) \tag{3.1}$$

Hidden layer to hidden layer:

$$net_j^r(t) = \sum_{s \in \mathcal{S}} w_{js} y_s(t-1) \tag{3.2}$$

where we denote the new set of indices for hidden-to-hidden weights as $\mathcal{S}$.

Hidden layer activiation:

$$net_j^s(t) = net_j^h(t) + net_j^r(t) \tag{3.3}$$

$$y_j(t) = \sigma(net_j^s(t)) \tag{3.4}$$

Hidden layer to output layer:

$$net_k^o(t) = \sum_j w_{kj} y_j(t) \tag{3.5}$$

$$z_k(t) = S_k(\boldsymbol{net}^o(t)) \tag{3.6}$$

3.2 Backpropagation

Gradient for w_{kj} is essentially the same as Feed-forward Network, i.e.

$$\frac{\partial J}{\partial w_{kj}} = e_k y_j \tag{3.7}$$

For w_{ji} and w_{js}, there are several methods to calculate their gradients.

3.2.1 Real-Time Recurrent Learning

RTRL[4] computes the derivatives in an online way, thus the name "Real-Time". Because of the recurrent term (3.2), variations in w_{ji} give rise to variations in the error function through variations in the all y_js.

The variable-dependence diagram for differentiating J w.r.t. w_{ji} is shown in Figure 4,

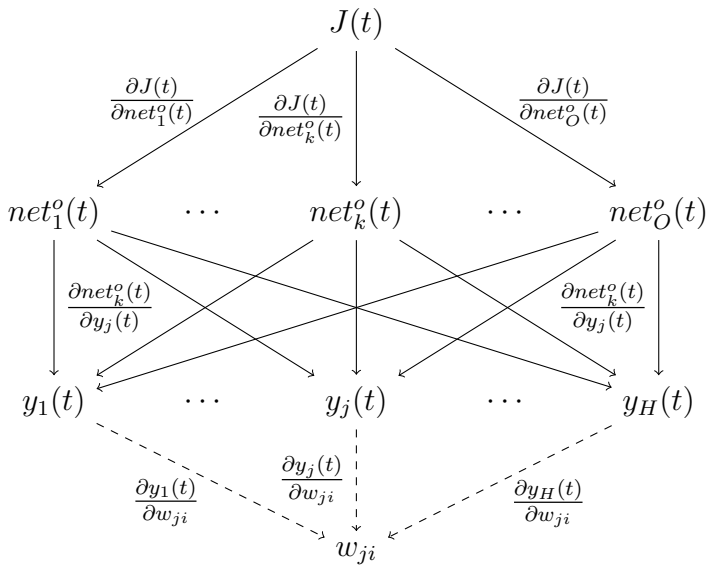

Figure 4: Variable-dependence diagram for w_{ji} of RTRL. Dashed line denotes a path which may through several variables.

We can write the derivative of J w.r.t. w_{ji},

$$\frac{\partial J}{\partial w_{ji}} = \sum_k \frac{\partial J}{\partial net_k^o} \sum_{l \in \mathcal{S}} \frac{\partial net_k^o}{\partial y_l(t)} \frac{\partial y_l(t)}{\partial w_{ji}} \tag{3.8}$$

As for feed-forward network, $\frac{\partial J}{\partial net_k^o} = e_k$ and $\frac{\partial net_k^o}{\partial y_l(t)} = w_{kl}$. For the $\frac{\partial y_l(t)}{\partial w_{ji}}$ part, the variable-dependence diagram is shown in Figure 5,

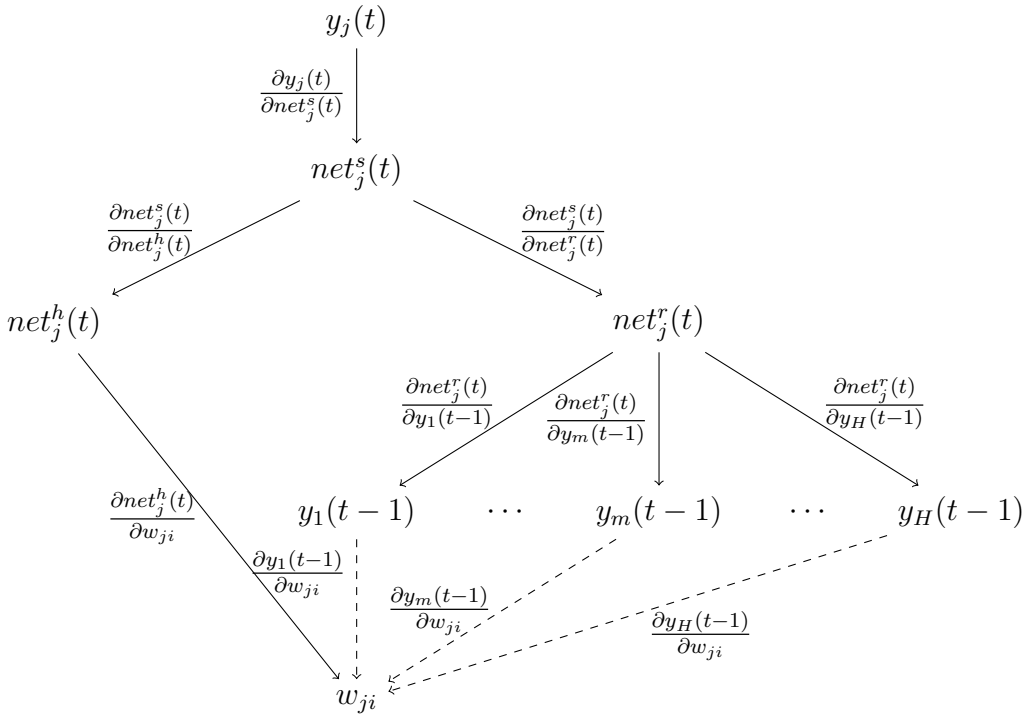

Figure 5: Variable-dependence diagram for $\frac{\partial y_l(t)}{\partial w_{ji}}$ part of RTRL, where $l = j$.

Figure 5 shows the situation where $l = j$. The diagram of $l \neq j$ does not have the left-most path from $net_j^s(t)$ to w_{ji}. Therefore, the derivative of $y_l(t)$ w.r.t. w_{ji} is,

$$\frac{\partial y_l(t)}{\partial w_{ji}} = \delta_{lj} \frac{\partial y_l(t)}{\partial net_l^s(t)} \frac{\partial net_l^s(t)}{\partial net_l^h(t)} \frac{\partial net_l^h(t)}{\partial w_{ji}} + \frac{\partial y_l(t)}{\partial net_l^s(t)} \frac{\partial net_l^s(t)}{\partial net_l^r(t)} \sum_{m \in \mathcal{S}} \frac{\partial net_l^r(t)}{\partial y_m(t-1)} \frac{\partial y_m(t-1)}{\partial w_{ji}} \tag{3.9}$$

Note that, $\frac{\partial net_l^s(t)}{net_l^h(t)} = \frac{\partial net_l^s(t)}{net_l^r(t)} = 1$, so the first term of right side is the same as (2.12) and (2.13),

$$\frac{\partial y_l(t)}{\partial net_l^s(t)} \frac{\partial net_l^s(t)}{\partial net_l^h(t)} \frac{\partial net_l^h(t)}{\partial w_{ji}} = \sigma'(net_l^s(t)) x_i(t) \tag{3.10}$$

However, the second term involves recurrent variable,

$$\frac{\partial y_l(t)}{\partial net_l^s(t)} \frac{\partial net_l^s(t)}{\partial net_l^r(t)} \sum_{m \in \mathcal{S}} \frac{\partial net_l^r(t)}{\partial y_m(t-1)} \frac{\partial y_m(t-1)}{\partial w_{ji}} = \sigma'(net_l^s(t)) \sum_{m \in \mathcal{S}} w_{jm} \frac{\partial y_m(t-1)}{\partial w_{ji}} \tag{3.11}$$

thus, combining (3.10) and (3.11), we get,

$$\frac{\partial y_l(t)}{\partial w_{ji}} = \sigma'(net_l^s(t))(\delta_{lj} x_i(t) + \sum_{m \in \mathcal{S}} w_{jm} \frac{\partial y_m(t-1)}{\partial w_{ji}}) \tag{3.12}$$

Put all together, we get the final result,

$$\frac{\partial J}{\partial w_{ji}} = \sum_k e_k \sum_{l \in \mathcal{S}} w_{kl} \sigma'(net_l^s(t))(\delta_{lj} x_i(t) + \sum_{m \in \mathcal{S}} w_{jm} \frac{\partial y_m(t-1)}{\partial w_{ji}}) \tag{3.13}$$

The hidden to hidden layer weights w_{js} is similar with w_{ji}, but the variable-dependence diagram for $\frac{\partial y_l(t)}{\partial w_{js}}$ also does not have the left-most net_j^h-through path in Figure 5. i.e.,

$$\frac{\partial y_l(t)}{\partial w_{js}} = \frac{\partial y_l(t)}{\partial net_l^s(t)} \frac{\partial net_l^s(t)}{\partial net_l^r(t)} \frac{\partial net_l^r(t)}{\partial w_{js}} \tag{3.14}$$

Next, From (3.2) we can get $\frac{\partial net_l^r(t)}{\partial w_{js}}$,

$$\frac{\partial net_l^r(t)}{\partial w_{js}} = \delta_{lj} y_s(t-1) + \sum_{m \in \mathcal{S}} w_{jm} \frac{\partial y_m(t-1)}{\partial w_{js}} \tag{3.15}$$

Therefore, we get the derivative of w_{js},

$$\frac{\partial J}{\partial w_{js}} = \sum_k e_k \sum_{l \in \mathcal{S}} w_{kl} \sigma'(net_l^s(t))(\delta_{lj} y_s(t-1) + \sum_{m \in \mathcal{S}} w_{jm} \frac{\partial y_m(t-1)}{\partial w_{js}}) \tag{3.16}$$

Following [4], we can combine (3.13) and (3.16), if we define

$$p_n(t) = \begin{cases} x_n(t) & n \in \mathcal{I} \\ y_n(t-1) & n \in \mathcal{S} \end{cases}$$

where $n \in \mathcal{I} \cup \mathcal{S}$, then we can get a unified form of result [1],

$$\frac{\partial J}{\partial w_{jn}} = \sum_k e_k \sum_{l \in \mathcal{S}} w_{kl} \sigma'(net_l^s(t))(\delta_{lj} p_n(t) + \sum_{m \in \mathcal{S}} w_{jm} \frac{\partial y_m(t-1)}{\partial w_{jn}}) \tag{3.17}$$

[1]The result in the original paper [4] seems simpler than ours, this is because the network in their paper does not have a output layer.

3.2.2 Backpropagation Through Time

BPTT[3] can be derived by unfolding the temporal operation of a network into a multilayer feedforward network that grows by one layer on each time step. In detail, in every time step, we performs followings sequentiallyas shown in Figure 6,

1. In the forward pass, we add one hidden layer, connecting from input layer and hidden layer for time $t-1$, then compute the activation of the new layer as activation of time t;

2. In the backpropagation, we compute the derivatives using BP through the whole network at time t, fixing all weights the same, then update the weights simultaneously.

Note that each hidden layer in the unfolded network corresponds a single time step, thus when we propagate errors through layers, we are actually "backpropagating through time".

The unfold network is simply a feed-forward network, thus we can directly use the results obtained in section 2. At time t_0, we first compute the errors for each layer,

$$e_j(t) = \begin{cases} \sum_k e_k(t) w_{kj} & t = t_0 \\ \sum_{l \in \mathcal{S}} e_l(t+1) w_{lj} & t < t_0 \end{cases} \tag{3.18}$$

Gradient of w_{kj} is the same as (2.9). For $w_{ji}(t)$, denoting the weight comes out from $x_i(t)$, we have

$$\frac{\partial J}{\partial w_{ji}(t)} = x_i(t) e_j(t) \tag{3.19}$$

Because that we'd like to update the weights simultaneously in one time step, we add up all t_0-layer gradients to compute the $\Delta \boldsymbol{w}$,

$$\Delta w_{ji}(t_0) = -\alpha \sum_{t'=1}^{t_0} \frac{\partial J}{\partial w_{ji}(t')} = -\alpha \sum_{t'=1}^{t_0} x_i(t') e_j(t') \tag{3.20}$$

Similarly, the updating value of w_{js} is

$$\Delta w_{js}(t_0) = -\alpha \sum_{t'=1}^{t_0} \frac{\partial J}{\partial w_{js}(t')} = -\alpha \sum_{t'=1}^{t_0} y_s(t'-1) e_j(t') \tag{3.21}$$

It is easy to see that back propagate for the whole time history for every time step is too complexity, and the error gradients quickly vanish or explode). So several

steps of unfolding are sufficient, which leads to the *Truncated BPTT*. Suppose we only unfold for T time step, then (3.20) and (3.21) become

$$\Delta w_{ji}(t_0) = -\alpha \sum_{t'=t_0-T}^{t_0} x_i(t') e_j(t') \tag{3.22}$$

and

$$\Delta w_{js}(t_0) = -\alpha \sum_{t'=t_0-T}^{t_0} y_s(t'-1) e_j(t') \tag{3.23}$$

As the on-line update would lead to large computational complexity, we can update the weights in mini-batches. The flow of gradients is illustrated in Figure 7.

References

[1] R.O. Duda, P.E. Hart, and D.G. Stork. *Pattern Classification*. Wiley, 2012.

[2] Tomáš Mikolov. *Statistical language models based on neural networks*. PhD thesis, Ph. D. thesis, Brno University of Technology, 2012.

[3] Ronald J. Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 2:490–501, 1990.

[4] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

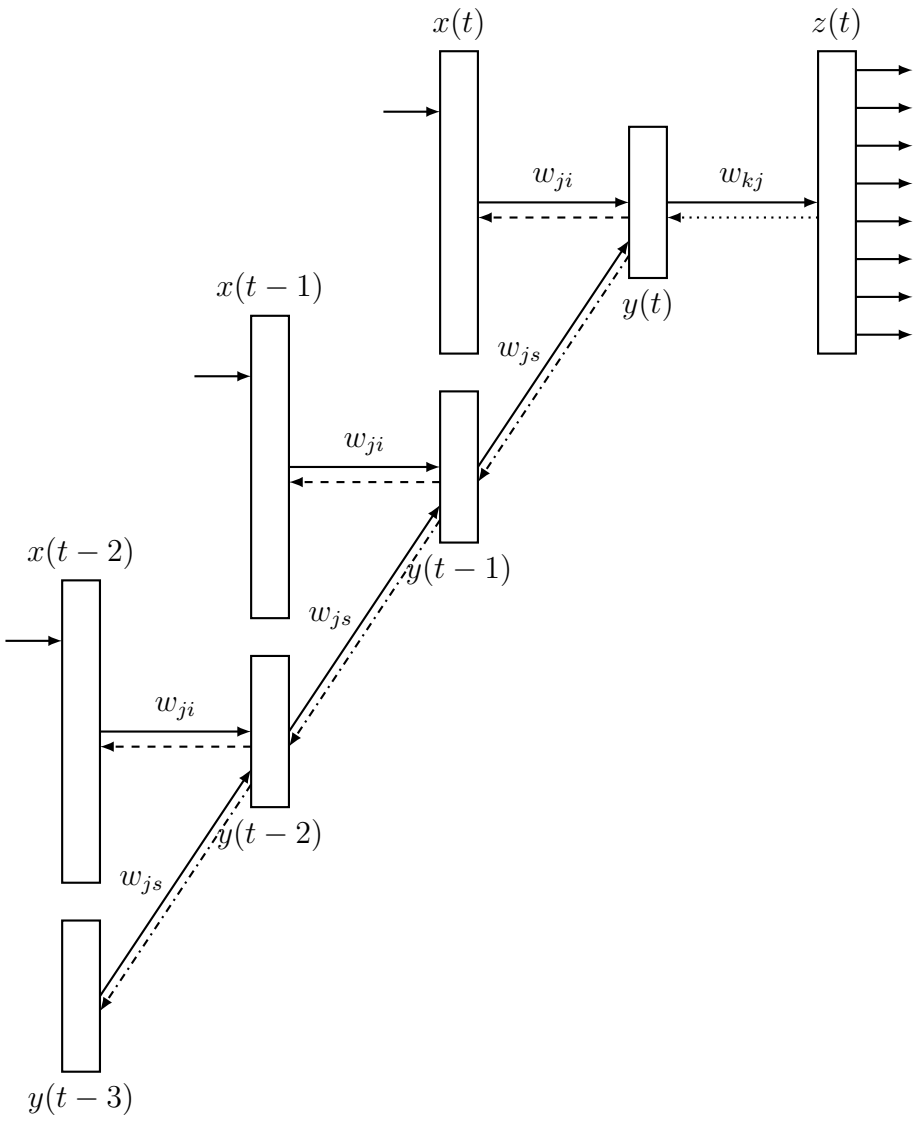

Figure 6: RNN unfolded as a feed-forward NN, here 3 time steps back in time. Dashed and dotted arrows indicate how the gradients are propagated

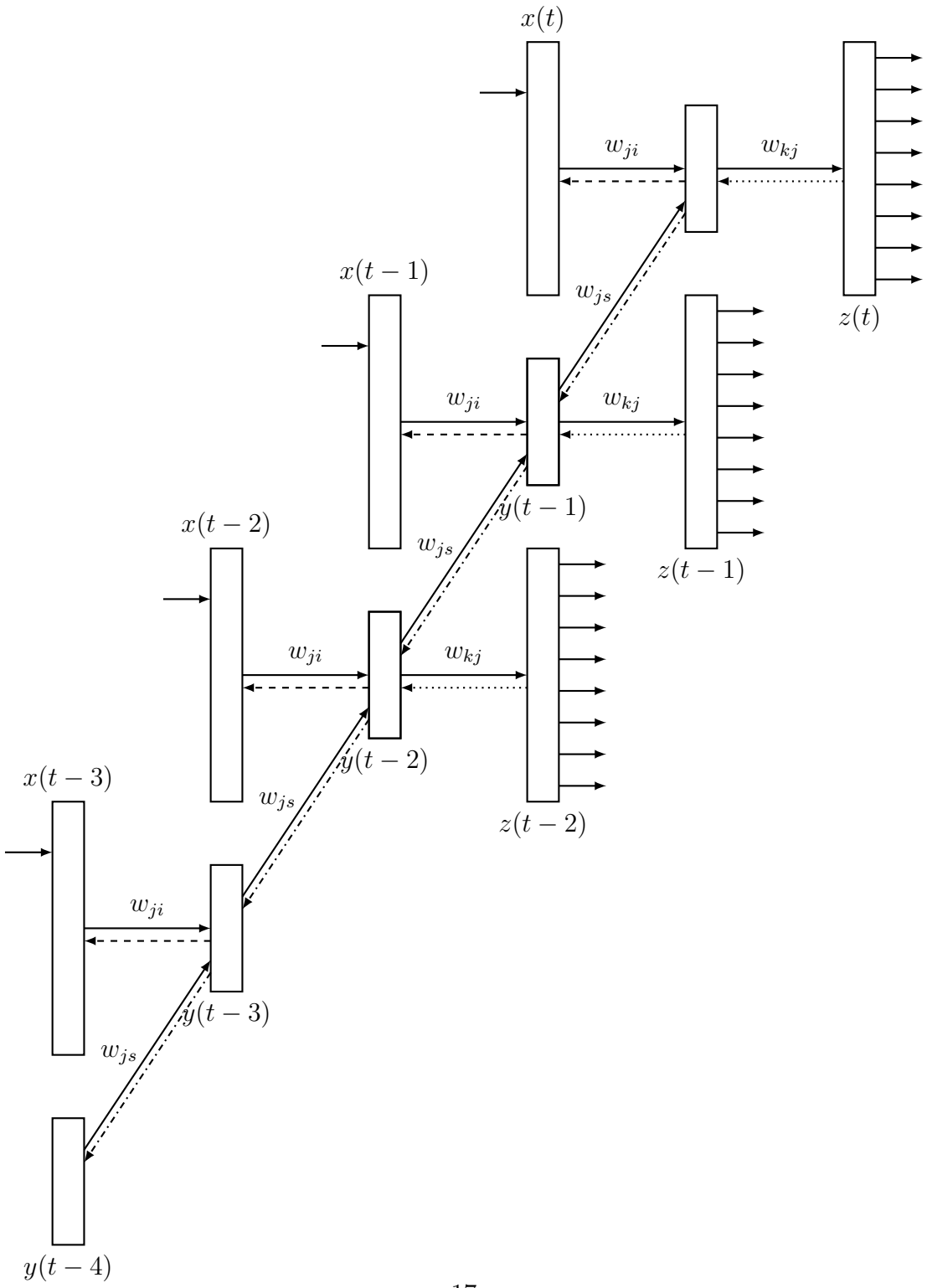

Figure 7: Mini-Batch BPTT training, with mini-batch size 3 and 2 time steps back