Long/Short-Term Memory

ECE 417: Multimedia Signal Processing
Mark Hasegawa-Johnson

University of Illinois

November 21, 2019

Outline

Notation

Today's lecture will try to use notation similar to the Wikipedia page for LSTM (and similar to MP7). This notation is different from what we've used before, so here are some symbols:

- $x[t]$ = input at time t
- $y[t]$ = target/desired output
- $c[t]$ = excitation at time t **OR** LSTM cell
- $h[t]$ = activation at time t **OR** LSTM output
- u = feedback coefficient
- w = feedforward coefficient
- b = bias

Running Example: a Pocket Calculator

The rest of this lecture will refer to toy application called "pocket calculator."

Pocket Calculator

- When $x[t] > 0$, add it to the current tally: $c[t] = c[t-1] + x[t]$.
- When $x[t] = 0$,
  1. Print out the current tally, $h[t] = c[t-1]$, and then
  2. Reset the tally to zero, $c[t] = 0$.

Example Signals

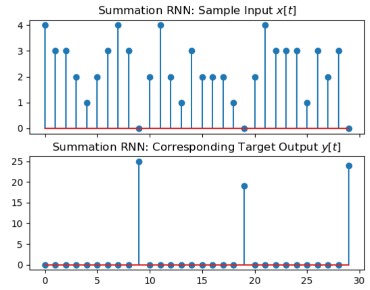
$$\textbf{Input: } x[t] = 1, 2, 1, 0, 1, 1, 1, 0$$

$$\textbf{Target Output: } y[t] = 0, 0, 0, 4, 0, 0, 0, 3$$

Pocket Calculator

- When $x[t] > 0$, add it to the current tally: $c[t] = c[t-1] + x[t]$.
- When $x[t] = 0$,
  1. Print out the current tally, $h[t] = c[t-1]$, and then
  2. Reset the tally to zero, $c[t] = 0$.

Pocket Calculator



Outline

One-Node One-Tap Linear RNN

Suppose that we have a very simple RNN:

$$\text{Excitation: } c[t] = x[t] + uh[t-1]$$
$$\text{Activation: } h[t] = \sigma_h\left(c[t]\right)$$

where $\sigma_h()$ is some feedback nonlinearity. In this simple example, let's just use $\sigma_h(c[t]) = c[t]$, i.e., no nonlinearity.
GOAL: Find u so that $h[t] \approx y[t]$. In order to make the problem easier, we will only score an "error" when $y[t] \neq 0$:

$$E = \frac{1}{2} \sum_{t:y[t]>0} \left(h[t] - y[t]\right)^2$$

RNN: $u = 1$?

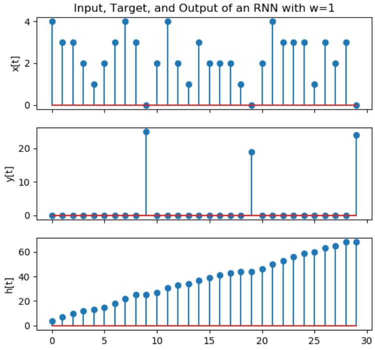
Obviously, if we want to just add numbers, we should just set $u = 1$. Then the RNN is computing

Excitation: $c[t] = x[t] + h[t-1]$

Activation: $h[t] = \sigma_h(c[t])$

That works until the first zero-valued input. But then it just keeps on adding.

RNN with $u = 1$

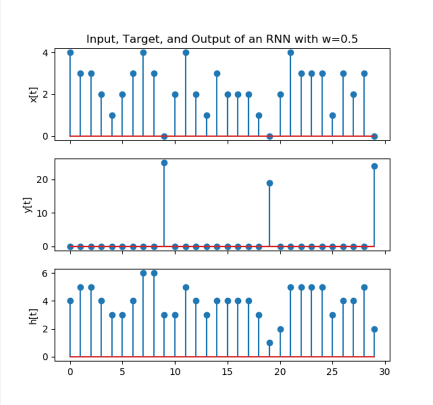


RNN: $u = 0.5$?

Can we get decent results using $u = 0.5$?

- Advantage: by the time we reach $x[t] = 0$, the sum has kind of leaked away from us ($c[t] \approx 0$), so a hard-reset is not necessary.
- Disadvantage: by the time we reach $x[t] = 0$, the sum has kind of leaked away from us ($h[t] \approx 0$).

RNN with $u = 0.5$



Gradient Descent

$$c[t] = x[t] + uh[t-1]$$
$$h[t] = \sigma_h\left(c[t]\right)$$

Let's try initializing $u = 0.5$, and then performing gradient descent to improve it. Gradient descent has five steps:

1. **Forward Propagation:** $c[t] = x[t] + uh[t-1]$, $h[t] = c[t]$.
2. **Synchronous Backprop:** $\epsilon[t] = \partial E/\partial c[t]$.
3. **Back-Prop Through Time:** $\delta[t] = dE/dc[t]$.
4. **Weight Gradient:** $dE/du = \sum_t \delta[t]h[t-1]$
5. **Gradient Descent:** $u \leftarrow u - \eta dE/du$

Gradient Descent

$$\text{Excitation: } c[t] = x[t] + uh[t-1]$$

$$\text{Activation: } h[t] = \sigma_h\left(c[t]\right)$$

$$\text{Error: } E = \frac{1}{2}\sum_{t:y[t]>0}\left(h[t] - y[t]\right)^2$$

So the back-prop stages are:

$$\textbf{Synchronous Backprop: } \epsilon[t] = \frac{\partial E}{\partial c[t]} = \begin{cases} (h[t] - y[t]) & y[t] > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\textbf{BPTT: } \delta[t] = \frac{dE}{dc[t]} = \epsilon[t] + u\delta[t+1]$$

$$\textbf{Wt Grad: } \frac{dE}{du} = \sum_t \delta[t]h[t-1]$$

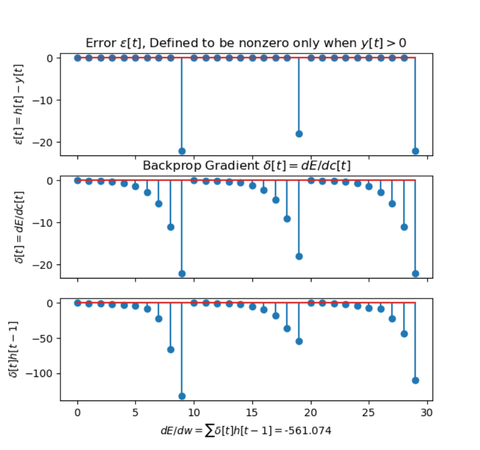
Backprop Stages

$$\epsilon[t] = \begin{cases} (h[t] - y[t]) & y[t] > 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\delta[t] = \epsilon[t] + u\delta[t+1]$$

$$\frac{dE}{du} = \sum_t \delta[t]h[t-1]$$

Backprop Stages, $u = 0.5$

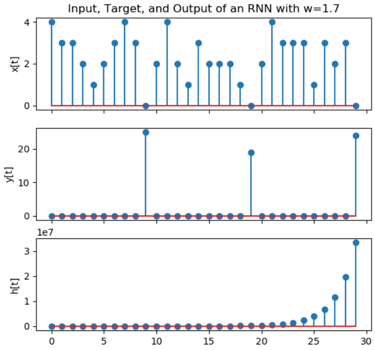


Vanishing Gradient and Exploding Gradient

- Notice that, with $|u| < 1$, $\delta[t]$ tends to vanish exponentially fast as we go backward in time. This is called the **vanishing gradient** problem. It is a big problem for RNNs with long time-dependency, and for deep neural nets with many layers.
- If we set $|u| > 1$, we get an even worse problem, sometimes called the **exploding gradient** problem.

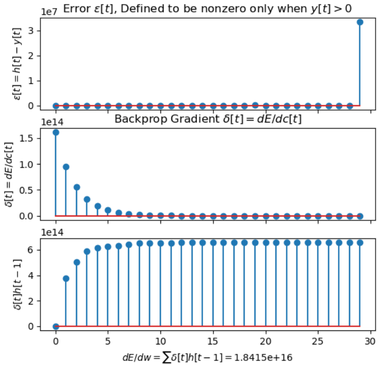
RNN, $u = 1.7$

$$c[t] = x[t] + uh[t-1]$$



RNN, $u = 1.7$

$$\delta[t] = \epsilon[t] + u\delta[t+1]$$



Outline

Hochreiter and Schmidhuber's Solution: The Forget Gate

Instead of multiplying by the same weight, u, at each time step, Hochreiter and Schmidhuber proposed: let's make the feedback coefficient a function of the input!

$$\begin{aligned} \text{Excitation: } & c[t] = x[t] + f[t]h[t-1] \\ \text{Activation: } & h[t] = \sigma_h\left(c[t]\right) \\ \text{Forget Gate: } & f[t] = \sigma_g\left(w_f x[t] + u_f h[t-1] + b_f\right) \end{aligned}$$

Where $\sigma_h()$ and $\sigma_g()$ might be different nonlinearities. In particular, it's OK for $\sigma_h()$ to be linear ($\sigma_h(c) = c$), but $\sigma_g()$ should be clipped so that $0 \le f[t] \le 1$, in order to avoid gradient explosion.

The Forget-Gate Nonlinearity

The forget gate is

$$f[t] = \sigma_g \left(w_f x[t] + u_f h[t-1] + b_f \right)$$

where $\sigma_g()$ is some nonlinearity such that $0 \leq \sigma_g() \leq 1$. Two such nonlinearities are worth knowing about.

Forget-Gate Nonlinearity #1: CReLU

The first useful nonlinearity is the CReLU (clipped rectified linear unit), defined as

$$\sigma_g(w_f x + u_f h + b_f) = \min\left(1, \max\left(0, w_f x + u_f h + b_f\right)\right)$$

- The CReLU is particularly useful for **knowledge-based design**. That's because $\sigma(1) = 1$ and $\sigma(0) = 0$, so it is relatively easy to design the weights w_f, u_f, and b_f to get the results you want.
- The CReLU is not very useful, though, if you want to choose your weights using **gradient descent**. What usually happens is that w_f grows larger and larger for the first 2-3 epochs of training, and then suddenly w_f is so large that $\dot{\sigma}(w_f x + u_f h + b_f) = 0$ for all training tokens. At that point, the gradient is $dE/dw = 0$, so further gradient-descent training is useless.

Forget-Gate Nonlinearity #1: Logistic Sigmoid

The second useful nonlinearity is the logistic sigmoid, defined as:

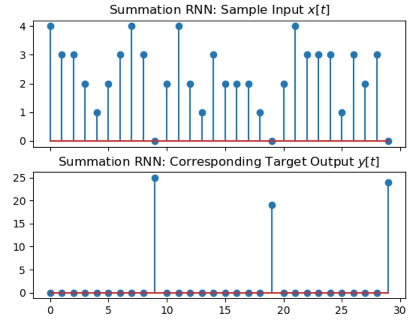
$$\sigma_g(w_f x + u_f h + b_f) = \frac{1}{1 + e^{-(w_f x + u_f h + b_f)}}$$

- The logistic sigmoid is particularly useful for **gradient descent**. That's because its gradient is defined for all values of w_f. In fact, it has a really simple form, that can be written in terms of the output: $\dot{\sigma} = \sigma(1 - \sigma)$.
- The logistic sigmoid is not as useful for **knowledge-based design**. That's because $0 < \sigma < 1$: as $x \rightarrow -\infty$, $\sigma(x) \rightarrow 0$, but it never quite reaches it. Likewise as $x \rightarrow \infty$, $\sigma(x) \rightarrow 1$, but it never quite reaches it.

Pocket Calculator

- When $x[t] > 0$, accumulate the input, and print out nothing.
- When $x[t] = 0$, print out the accumulator, then reset.

…but the "print out nothing" part is not scored, only the accumulation. Furthermore, nonzero input is always $x[t] \geq 1$.

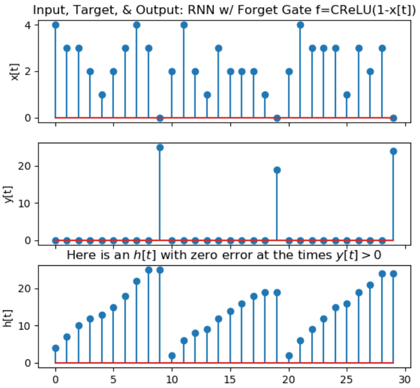


Pocket Calculator

With zero error, we can approximate the pocket calculator as

- When $x[t] \geq 1$, accumulate the input.
- When $x[t] = 0$, print out the accumulator, then reset.

$E = \frac{1}{2}\sum_{t:y[t]>0}\left(h[t] - y[t]\right)^2 = 0$



Forget-Gate Implementation of the Pocket Calculator

It seems like we can approximate the pocket calculator as:

- When $x[t] \geq 1$, accumulate the input: $c[t] = x[t] + h[t-1]$.
- When $x[t] = 0$, print out the accumulator, then reset: $c[t] = x[t]$.

So it seems that we just want the forget gate set to

$$f[t] = \begin{cases} 1 & x[t] \geq 1 \\ 0 & x[t] = 0 \end{cases}$$

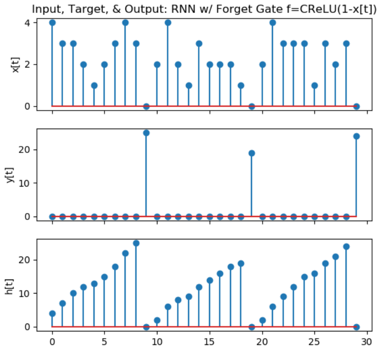
This can be accomplished as

$$f[t] = \text{CReLU}\left(x[t]\right) = \max\left(0, \min\left(1, x[t]\right)\right)$$

Forget Gate Implementation of the Pocket Calculator

$$c[t] = x[t] + f[t]h[t-1]$$
$$h[t] = c[t]$$
$$f[t] = \text{CReLU}\left(x[t]\right)$$

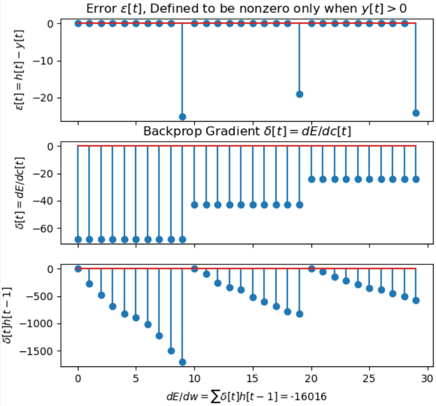
Forward Prop



Forget Gate Implementation of the Pocket Calculator

$$c[t] = x[t] + f[t]h[t-1]$$
$$h[t] = c[t]$$
$$f[t] = \mathrm{CReLU}(x[t])$$

Back Prop



What Went Wrong?

- The forget gate correctly turned itself on (remember the past) when $x[t] > 0$, and turned itself off (forget the past) when $x[t] = 0$.
- Unfortuately, we don't want to forget the past when $x[t] = 0$. We want to forget the past on the **next time step after** $x[t] = 0$.
- Coincidentally, we also don't want any output when $x[t] > 0$. The error criterion doesn't score those samples, but maybe it should.

Outline

Long Short-Term Memory (LSTM)

The LSTM solves those problems by defining two types of memory, and three types of gates. The two types of memory are

1. The "cell," $c[t]$, corresponds to the excitation in an RNN.
2. The "output" or "prediction," $h[t]$, corresponds to the activation in an RNN.

The three gates are:

1. The cell remembers the past only when the forget gate is on, $f[t] = 1$.
2. The cell accepts input only when the input gate is on, $i[t] = 1$.
3. The cell is output only when the output gate is on, $o[t] = 1$.

Long Short-Term Memory (LSTM)

The three gates are:

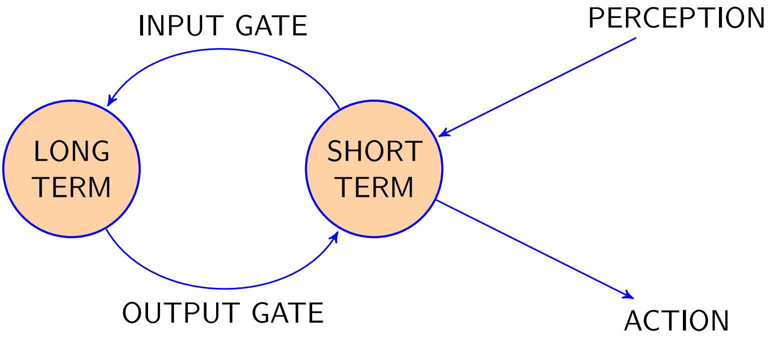
1. The cell remembers the past only when the forget gate is on, $f[t] = 1$.
2. The cell accepts input only when the input gate is on, $i[t] = 1$.

$$c[t] = f[t]c[t-1] + i[t]\sigma_h\left(w_c x[t] + u_c h[t-1] + b_c\right)$$

3. The cell is output only when the output gate is on, $o[t] = 1$.

$$h[t] = o[t]c[t]$$

Characterizing Human Memory



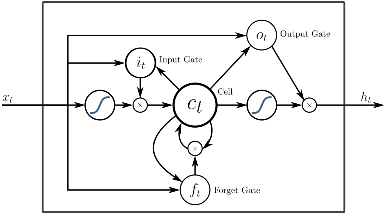
$$\Pr\{\text{remember}\} = p_{LTM} e^{-t/T_{LTM}} + (1 - p_{LTM}) e^{-t/T_{STM}}$$

When Should You Remember?

$$c[t] = f[t]c[t-1] + i[t]\sigma_h\left(w_c x[t] + u_c h[t-1] + b_c\right)$$
$$h[t] = o[t]c[t]$$

1. The forget gate is a function of current input and past output, $f[t] = \sigma_g\left(w_f x[t] + u_f h[t-1] + b_f\right)$
2. The input gate is a function of current input and past output, $i[t] = \sigma_g\left(w_i x[t] + u_i h[t-1] + b_i\right)$
3. The output gate is a function of current input and past output, $o[t] = \sigma_g\left(w_o x[t] + u_o h[t-1] + b_o\right)$

Neural Network Model: LSTM

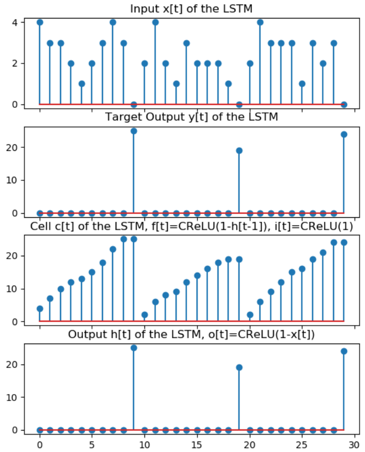


$$i[t] = \text{input gate} = \sigma_g(w_i x[t] + u_i h[t-1] + b_i)$$
$$o[t] = \text{output gate} = \sigma_g(w_o x[t] + u_o h[t-1] + b_o)$$
$$f[t] = \text{forget gate} = \sigma_g(w_f x[t] + u_f h[t-1] + b_f)$$
$$c[t] = \text{memory cell} = f[t]c[t-1] + i[t]\sigma_h(w_c x[t] + u_c h[t-1] + b_c)$$
$$h[t] = \text{output} = o[t]c[t]$$

Example: Pocket Calculator

$$i[t] = \text{CReLU}(1)$$
$$o[t] = \text{CReLU}(1 - x[t])$$
$$f[t] = \text{CReLU}(1 - h[t-1])$$
$$c[t] = f[t]c[t-1] + i[t]x[t]$$
$$h[t] = o[t]c[t]$$

Back Prop



Outline

Backprop for a normal RNN

In a normal RNN, each epoch of gradient descent has five steps:

1. **Forward-prop:** find the node excitation and activation, moving forward through time.
2. **Synchronous backprop:** find the partial derivative of error w.r.t. node excitation at each time, assuming all other time steps are constant.
3. **Back-prop through time:** find the total derivative of error w.r.t. node excitation at each time.
4. **Weight gradient:** find the total derivative of error w.r.t. each weight and each bias.
5. **Gradient descent:** adjust each weight and bias in the direction of the negative gradient

Backprop for an LSTM

An LSTM differs from a normal RNN in that, instead of just one memory unit at each time step, we now have two memory units and three gates. Each of them depends on the previous time-step. Since there are so many variables, let's stop back-propagating to excitations. Instead, we'll just back-prop to compute the derivative of the error w.r.t. each of the variables:

$$\epsilon_h[t] = \frac{\partial E}{\partial h[t]} \quad \epsilon_c[t] = \frac{\partial E}{\partial c[t]}, \quad \epsilon_i[t] = \frac{\partial E}{\partial i[t]}, \quad \epsilon_o[t] = \frac{\partial E}{\partial o[t]}, \quad \epsilon_f[t] = \frac{\partial E}{\partial f[t]}$$

$$\delta_h[t] = \frac{dE}{dh[t]} \quad \delta_c[t] = \frac{dE}{dc[t]}, \quad \delta_i[t] = \frac{dE}{di[t]}, \quad \delta_o[t] = \frac{dE}{do[t]}, \quad \delta_f[t] = \frac{dE}{df[t]}$$

Backprop for an LSTM

In an LSTM, we'll implement each epoch of gradient descent with five steps:

1. **Forward-prop:** find all five of the variables at each time step, moving forward through time.
2. **Synchronous backprop:** find the partial derivative of error w.r.t. $h[t]$.
3. **Back-prop through time:** find the total derivative of error w.r.t. each of the five variables at each time, starting with $h[t]$.
4. **Weight gradient:** find the total derivative of error w.r.t. each weight and each bias.
5. **Gradient descent:** adjust each weight and bias in the direction of the negative gradient

Synchronous Back-Prop: the Output

Suppose the error term is

$$E = \frac{1}{2}\sum_{t=-\infty}^{\infty}(h[t]-y[t])^2$$

Then the first step, in back-propagation, is to calculate the partial derivative w.r.t. the prediction term $h[t]$:

$$\epsilon_h[t] = \frac{\partial E}{\partial h[t]} = h[t]-y[t]$$

Synchronous Back-Prop: the other variables

Remember that the error is defined only in terms of the output, $h[t]$. So, actually, partial derivatives with respect to the other variables are all zero!

$$\epsilon_i[t] = \frac{\partial E}{\partial i[t]} = 0$$

$$\epsilon_o[t] = \frac{\partial E}{\partial o[t]} = 0$$

$$\epsilon_f[t] = \frac{\partial E}{\partial f[t]} = 0$$

$$\epsilon_c[t] = \frac{\partial E}{\partial c[t]} = 0$$

Back-Prop Through Time

Back-prop through time is really tricky in an LSTM, because four of the five variables depend on the previous time step, either on $h[t-1]$ or on $c[t-1]$:

$$
\begin{aligned}
i[t] &= \sigma_g(w_i x[t] + u_i h[t-1] + b_i)\\
o[t] &= \sigma_g(w_o x[t] + u_o h[t-1] + b_o)\\
f[t] &= \sigma_g(w_f x[t] + u_f h[t-1] + b_f)\\
c[t] &= f[t]c[t-1] + i[t]\sigma_h\left(w_c x[t] + u_c h[t-1] + b_c\right)\\
h[t] &= o[t]c[t]
\end{aligned}
$$

Back-Prop Through Time

Taking the partial derivative of each variable at time t w.r.t. the variables at time $t-1$, we get

$$\frac{\partial i[t]}{\partial h[t-1]} = \dot{\sigma}_g(w_i x[t] + u_i h[t-1] + b_i) u_i$$

$$\frac{\partial o[t]}{\partial h[t-1]} = \dot{\sigma}_g(w_o x[t] + u_o h[t-1] + b_o) u_o$$

$$\frac{\partial o[t]}{\partial h[t-1]} = \dot{\sigma}_g(w_f x[t] + u_f h[t-1] + b_f) u_f$$

$$\frac{\partial c[t]}{\partial h[t-1]} = i[t]\dot{\sigma}_h\left(w_c x[t] + u_c h[t-1] + b_c\right) u_c$$

$$\frac{\partial c[t]}{\partial c[t-1]} = f[t]$$

Back-Prop Through Time

Using the standard rule for partial and total derivatives, we get a really complicated rule for $h[t]$:

$$\frac{dE}{dh[t]} = \frac{\partial E}{\partial h[t]} + \frac{dE}{di[t+1]}\frac{\partial i[t+1]}{\partial h[t]} + \frac{dE}{do[t+1]}\frac{\partial o[t+1]}{\partial h[t]} + \frac{dE}{df[t+1]}\frac{\partial f[t+1]}{\partial h[t]} + \frac{dE}{dc[t+1]}\frac{\partial c[t+1]}{\partial h[t]}$$

The rule for $c[t]$ is a bit simpler, because $\partial E/\partial c[t] = 0$, so we don't need to include it:

$$\frac{dE}{dc[t]} = \frac{dE}{dh[t]}\frac{\partial h[t]}{\partial c[t]} + \frac{dE}{dc[t+1]}\frac{\partial c[t+1]}{\partial c[t]}$$

Back-Prop Through Time

If we define $\delta_h[t] = dE/dh[t]$, and so on, then we have

$$\begin{aligned}\delta_h[t] = \epsilon_h[t] &+ \delta_i[t+1]\dot{\sigma}_g(w_i x[t+1] + u_i h[t] + b_i)u_i \\ &+ \delta_o[t+1]\dot{\sigma}_g(w_o x[t+1] + u_o h[t] + b_o)u_o \\ &+ \delta_f[t+1]\dot{\sigma}_g(w_f x[t+1] + u_f h[t] + b_f)u_f \\ &+ i[t+1]\delta_c[t+1]\dot{\sigma}_h\left(w_c x[t+1] + u_c h[t] + b_c\right)u_c\end{aligned}$$

The rule for $c[t]$ is a bit simpler:

$$\delta_c[t] = \delta_h[t]o[t] + \delta_c[t+1]f[t+1]$$

Back-Prop Through Time

BPTT through the gates is simplified in the same way as for the cell, i.e., since there is no direct dependence of the error on any gate, we only have to deal with the indirect dependences:

$$\frac{dE}{do[t]} = \frac{dE}{dh[t]}\frac{\partial h[t]}{\partial o[t]}$$
$$\frac{dE}{di[t]} = \frac{dE}{dc[t]}\frac{\partial c[t]}{\partial i[t]}$$
$$\frac{dE}{df[t]} = \frac{dE}{dc[t]}\frac{\partial c[t]}{\partial f[t]}$$

where the partial derivatives are defined by the forward equations:

$$c[t] = f[t]c[t-1] + i[t]\sigma_h\left(w_c x[t] + u_c h[t-1] + b_c\right)$$
$$h[t] = o[t]c[t]$$

Back-Prop Through Time

The partial derivatives are defined by the forward equations:

$$c[t] = f[t]c[t-1] + i[t]\sigma_h\left(w_c x[t] + u_c h[t-1] + b_c\right)$$
$$h[t] = o[t]c[t]$$

so:

$$\frac{\partial h[t]}{\partial o[t]} = c[t]$$
$$\frac{\partial h[t]}{\partial i[t]} = \sigma_h\left(w_c x[t] + u_c h[t-1] + b_c\right)$$
$$\frac{\partial h[t]}{\partial f[t]} = c[t-1]$$

Back-Prop Through Time

BPTT through the gates is simplified in the same way as for the cell, i.e., since there is no direct dependence of the error on any gate, we only have to deal with the indirect dependences:

$$\delta_o[t] = \delta_h[t]c[t]$$
$$\delta_i[t] = \delta_c[t]\sigma_h\left(w_c x[t] + u_c h[t-1] + b_c\right)$$
$$\delta_f[t] = \delta_c[t]c[t-1]$$

Outline

- RNNs suffer from either exponentially decreasing memory (if $|w| < 1$) or exponentially increasing memory (if $|w| > 1$). This is one version of a more general problem sometimes called the **gradient vanishing** problem.
- The forget gate solves that problem by making the feedback coefficient a function of the input.
- LSTM defines two types of memory (cell=excitation, and output=activation), and three types of gates (input, output, forget).
- Each epoch of LSTM training has the same steps as in a regular RNN:
  1. Forward propagation: find $h[t]$.
  2. Synchronous backprop: find the time-synchronous partial derivatives $\epsilon[t]$.
  3. BPTT: find the total derivatives $\delta[t]$.
  4. Weight gradients
  5. Gradient descent