Class 3: Training Recurrent Nets

Arvind Ramanathan
Computational Science & Engineering, Oak Ridge National Laboratory, Oak Ridge, TN 37830
ramanathana@ornl.gov

Last class

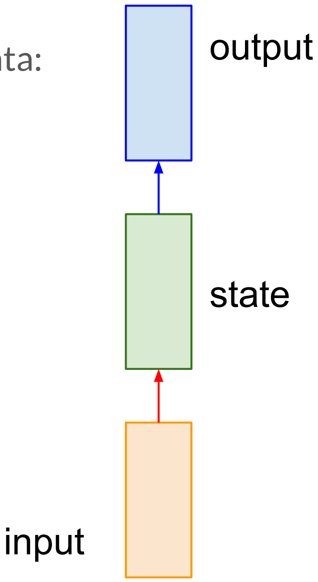
Basics of RNNs

Recurrent network modeling

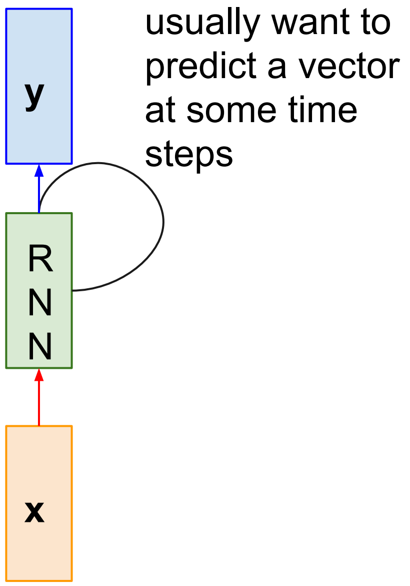
How to build a RNN and its different types

Quick Recap (1): Vanilla (E.g., Convolutional) nets

- Most convolutional nets are limited in their ability to represent data:
  - Take a fixed size input vector and output a fixed size vector
    - E.g., take image and classify
  - Only fixed number of layers/ computational steps
    - E.g., LeNet has five layers
- Efficient to train -- but representation is still limited to neighborhood information
  - Does not capture potentially long range interactions
- Usually applicable in “discriminative” situations…
  - Referred to as “one-to-one” architectures



Quick Recap (2): RNN and its components

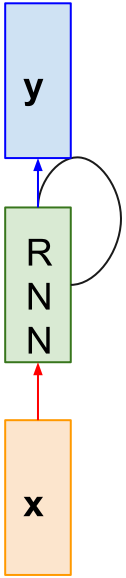


RNNs combine the input vector with their state vector with a fixed (but learned) function to produce a new state vector

Think of running a “fixed” program + some internal variables on every input

RNNs represent programs: RNNs are Turing complete -- meaning they can run any arbitrary program!

Quick Recap (3): RNN + recurrence formula



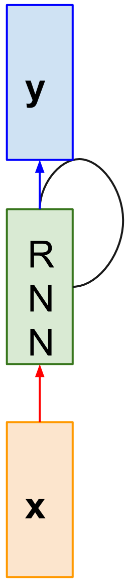
$$h_t = f_W(h_{t-1}, x_t)$$

- h_t: New state
- f_W: Some function with parameters W
- h_{t-1}: Old state
- x_t: Input vector at time t

- We can process a sequence of vectors x by applying a recurrence formula at every time step
- The same function and same set of parameters are used every time step.

A simple RNN

The state consists of a single hidden vector **h**:

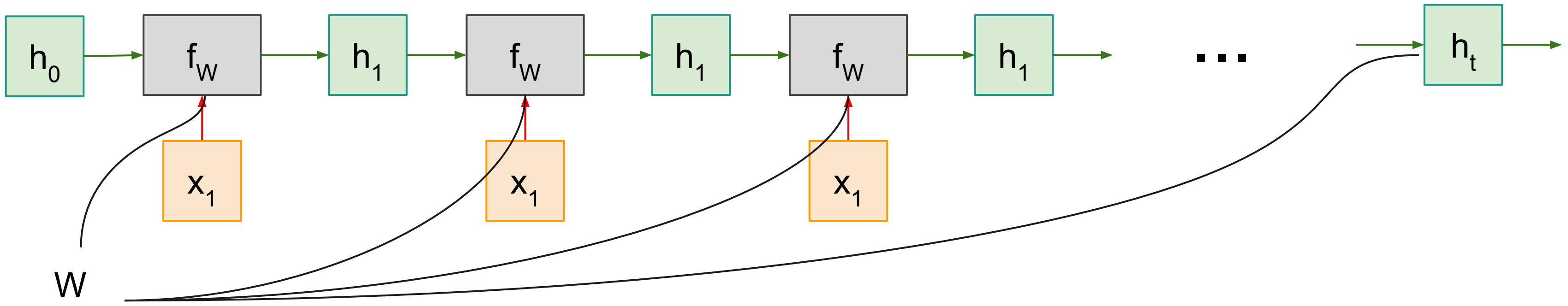


$$h_t = f_W(h_{t-1}, x_t)$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

Advancing / Unrolling the RNN → Computational Graph Representation



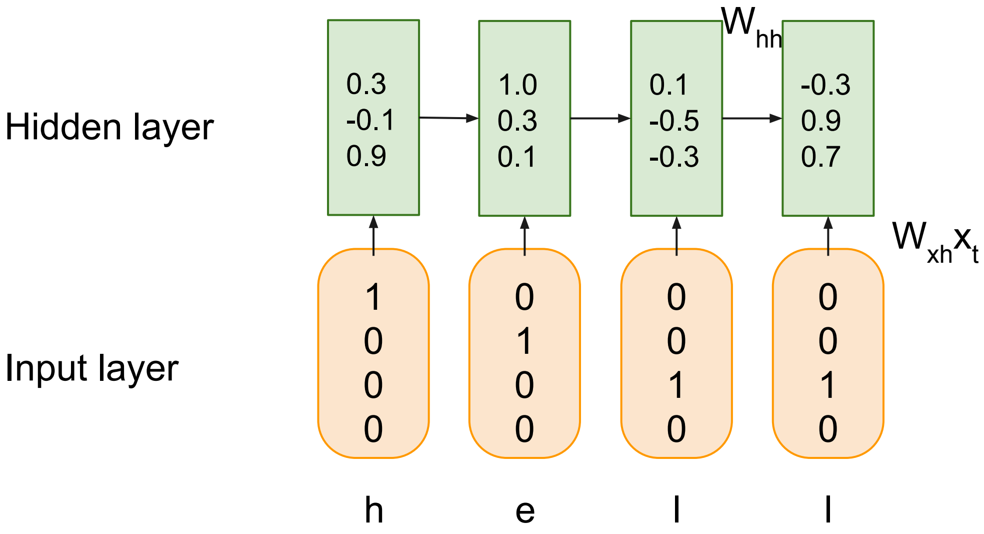
Example: Character level language model

Vocabulary: [h, e, l, o]

Example training sequence:

“hello”

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

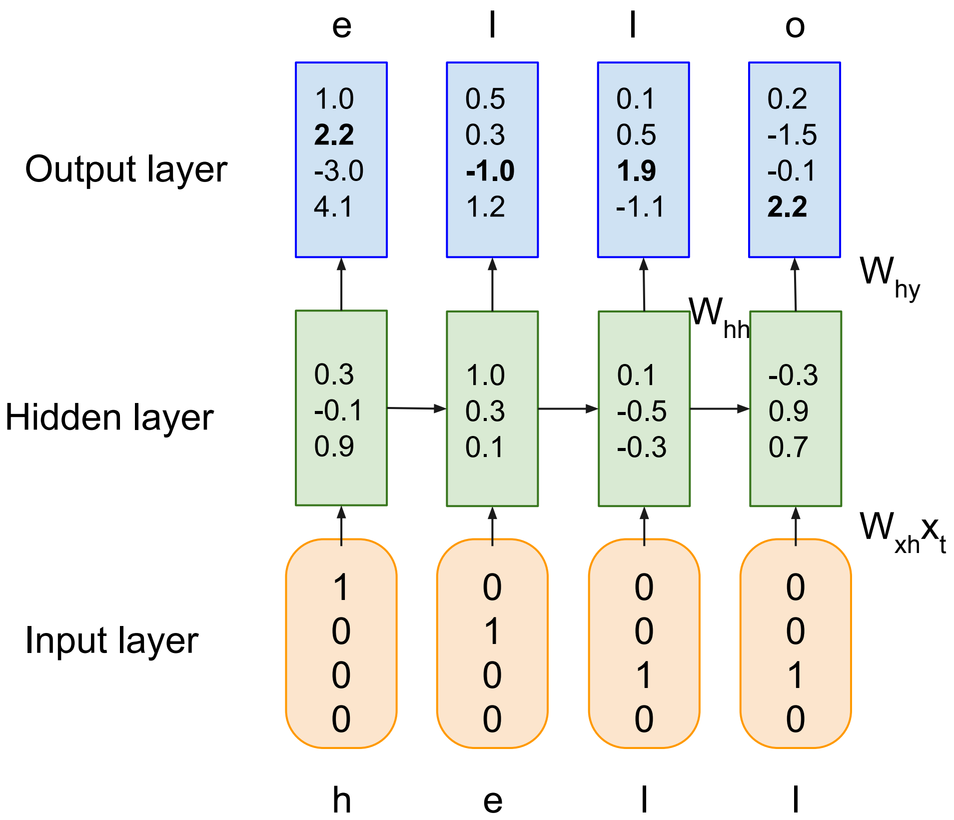


Example: Character level language model

Vocabulary: [h, e, l, o]

Example training sequence:

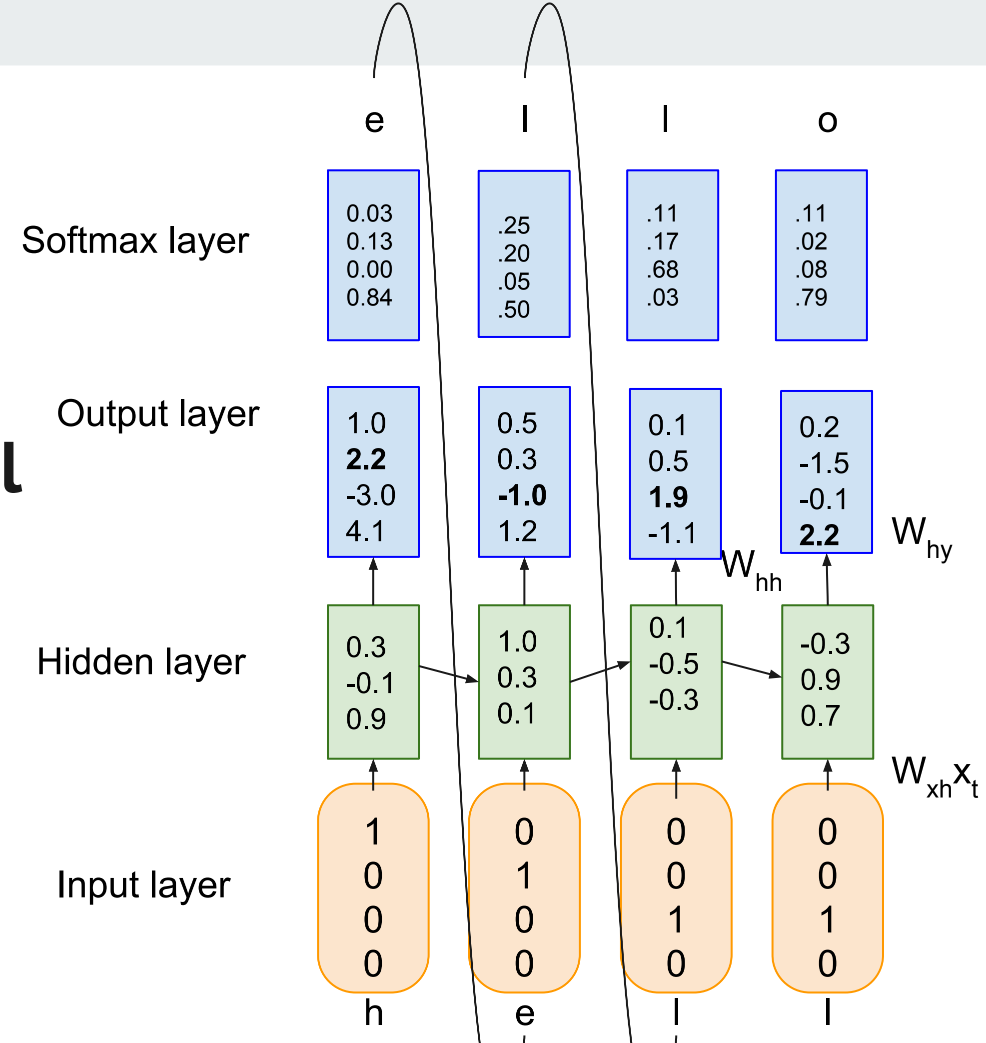
“hello”



Example: Character level language model sampling

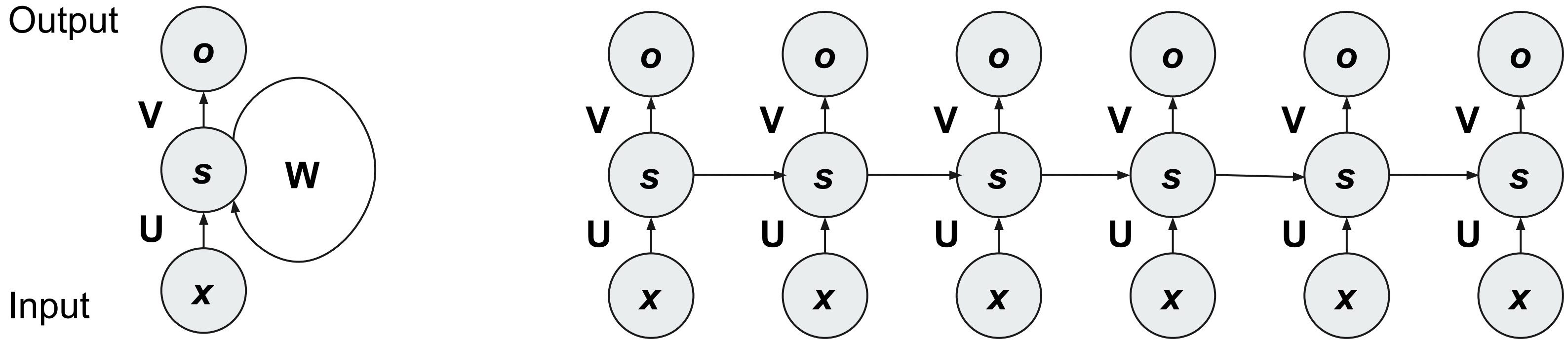
Vocabulary: [h, e, l, o]

At test-time sample characters one at a time, feed back to model



Training your first RNN...

Let’s take a simple example and explore...



$$s_t = \tanh(Ux_t + Ws_{t-1})$$

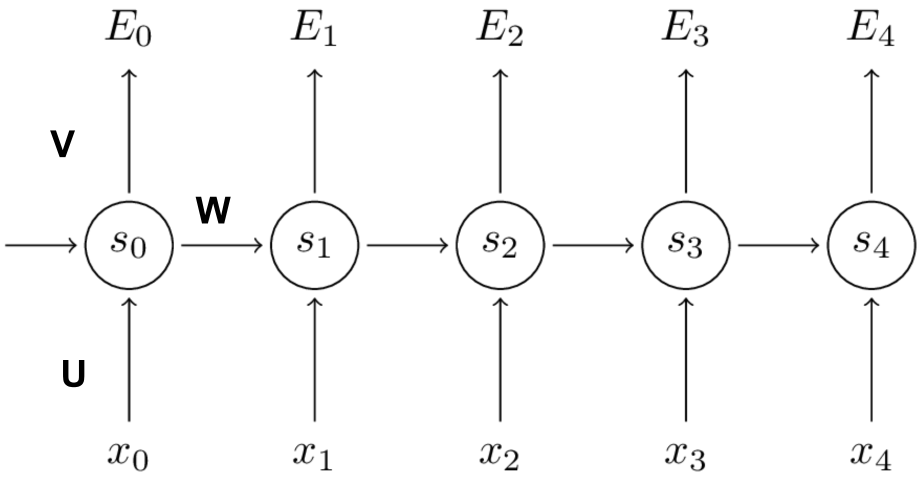
$$\hat{o}_t = \mathrm{softmax}(Vs_t)$$

Expanding log loss of the model...

$$E_t(o_t, \hat{o}_t) = -o_t \log \hat{o}_t$$

$$E(o, \hat{o}) = \sum_t E_t(o_t, \hat{o}_t)$$

$$= -\sum_t o_t \log \hat{o}_t$$



http://www.wildml.com/2015/10/recurrent-neural-networks-tutorial-part-3-backpropagation-through-time-and-vanishing-gradients/

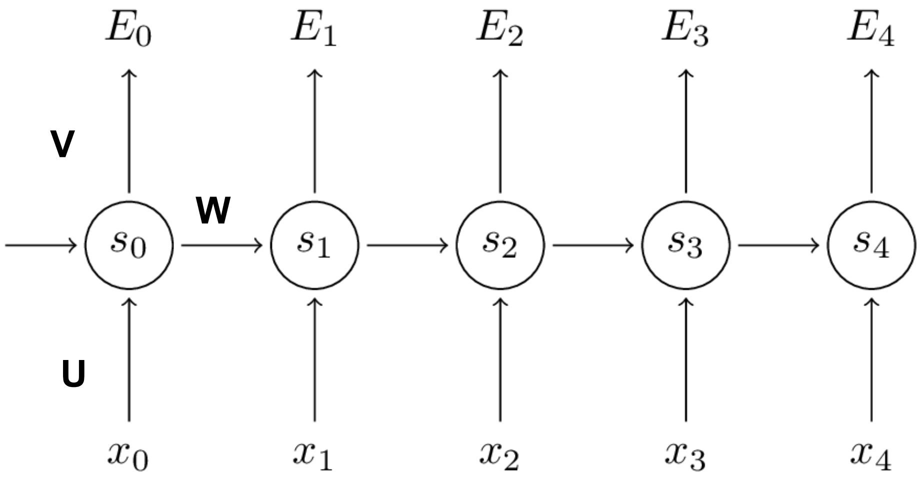
How do we compute the gradients?

We need to compute gradients of the error with respect to our parameters U, V, W

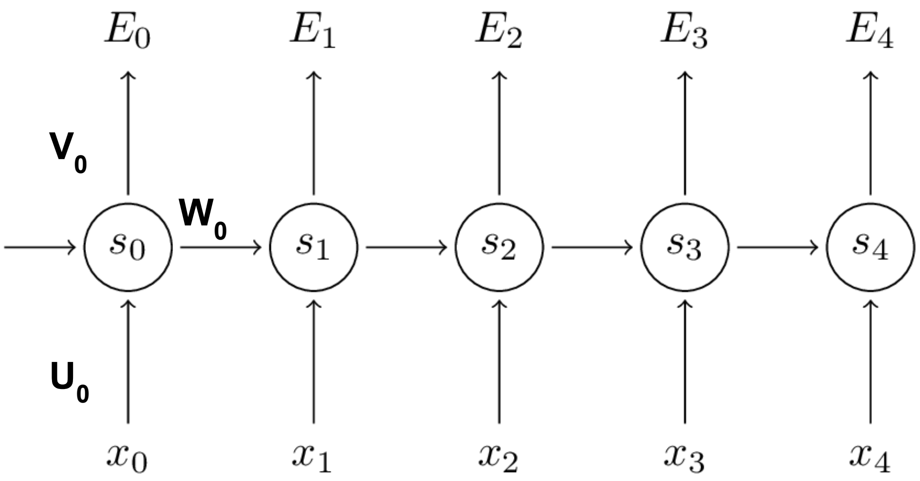
Use Stochastic Gradient Descent

sum up the gradients at each time step for one training example

$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$$



Computing gradients at E3



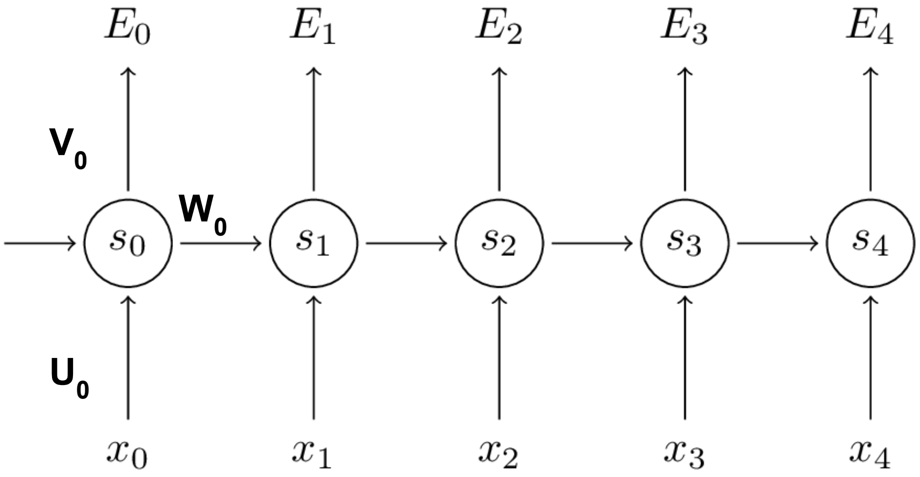
$$
\begin{aligned}
\frac{\partial E_3}{\partial V} &= \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial V} \\
&= \frac{\partial E_3}{\partial \hat{y}_3}\frac{\partial \hat{y}_3}{\partial z_3}\frac{\partial z_3}{\partial V_3} \\
&= (\hat{y}_3 - y_3) \otimes s_3
\end{aligned}
$$

$Z_3 = Vs_3$

Important note: Gradient values at E3 depend only on the current timestep...

Computing gradient wrt V is easy…..

What about computing gradient wrt W?

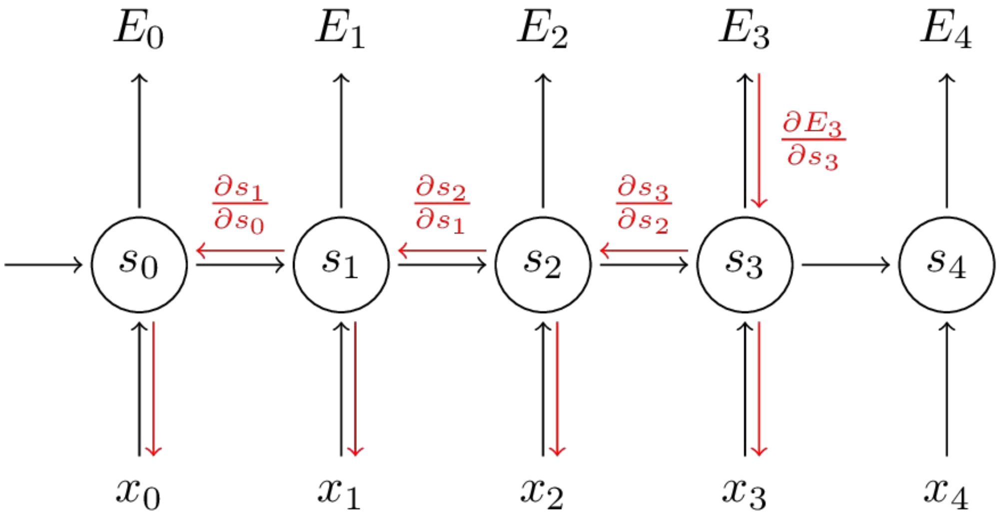


$$\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial W}$$

$$s_3 = \tanh(Ux_t + Ws_2)$$
$$s_2 = \tanh(Ux_t + Ws_1) \ldots$$

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W}$$

Unrolling the gradients through the computational graph



$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W}$$

Exactly the same backpropagation algorithm -- key difference is that for W at each time step we sum up the gradients until that step

How do we write it in Python?

A naive implementation

Includes two for loops

- One for time-range (sequence length)
- One for propagating the gradients

This should give you a sense of why BPTT is expensive computationally

- A serial computation embedded within what could be potentially parallel

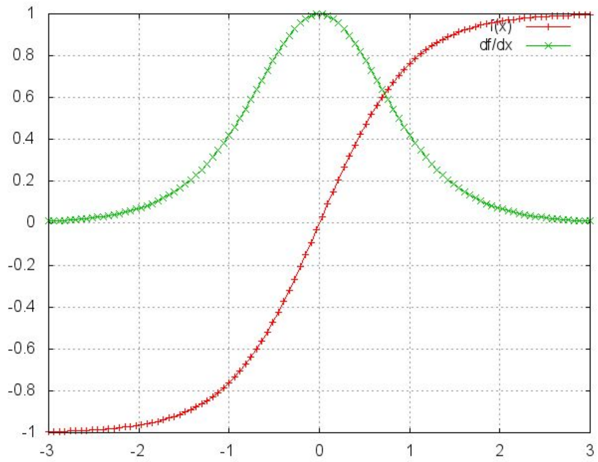
Arbitrary length sequences can make it even more expensive to compute backprop…

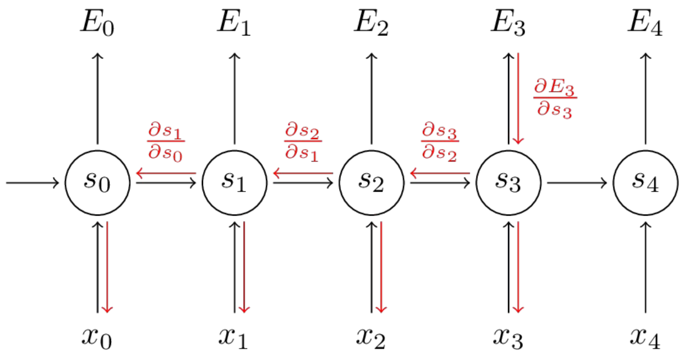
```
def bptt(self, x, y):
    T = len(y)
    # Perform forward propagation
    o, s = self.forward_propagation(x)
    # We accumulate the gradients in these variables
    dLdU = np.zeros(self.U.shape)
    dLdV = np.zeros(self.V.shape)
    dLdW = np.zeros(self.W.shape)
    delta_o = o
    delta_o[np.arange(len(y)), y] -= 1.
    # For each output backwards...
    for t in np.arange(T)[::-1]:
        dLdV += np.outer(delta_o[t], s[t].T)
        # Initial delta calculation: dL/dz
        delta_t = self.V.T.dot(delta_o[t]) * (1 - (s[t] ** 2))
        # Backpropagation through time (for at most self.bptt_truncate steps)
        for bptt_step in np.arange(max(0, t-self.bptt_truncate), t+1)[::-1]:
            # print "Backpropagation step t=%d bptt step=%d " % (t, bptt_step)
            # Add to gradients at each previous step
            dLdW += np.outer(delta_t, s[bptt_step-1])
            dLdU[:,x[bptt_step]] += delta_t
            # Update delta for next step dL/dz at t-1
            delta_t = self.W.T.dot(delta_t) * (1 - s[bptt_step-1] ** 2)
    return [dLdU, dLdV, dLdW]
```

Problems galore with BPTT...

There is a product of gradients that propagates ...

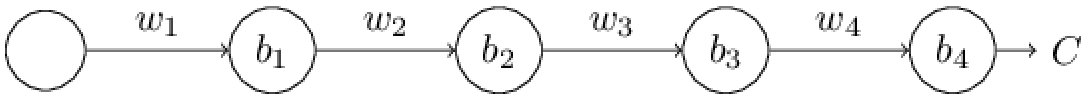
$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \frac{\partial s_3}{\partial s_k} \frac{\partial s_k}{\partial W}$$





$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial s_3} \left(\Pi_{j=k+1}^{3} \frac{\partial s_j}{\partial s_{j-1}} \right) \frac{\partial s_k}{\partial W}$$

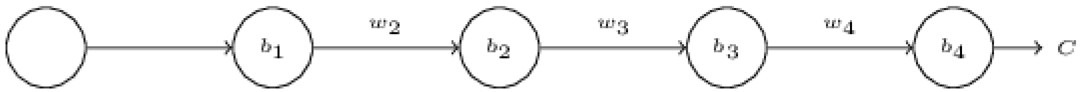
Your first tryst with the Vanishing Gradient...



Output a_j from the j[th] neuron is $\sigma(z_j)$. Input is the weighted neurons

$$z_j = w_j a_{j-1} + b_j$$

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$



Why does vanishing gradient occur

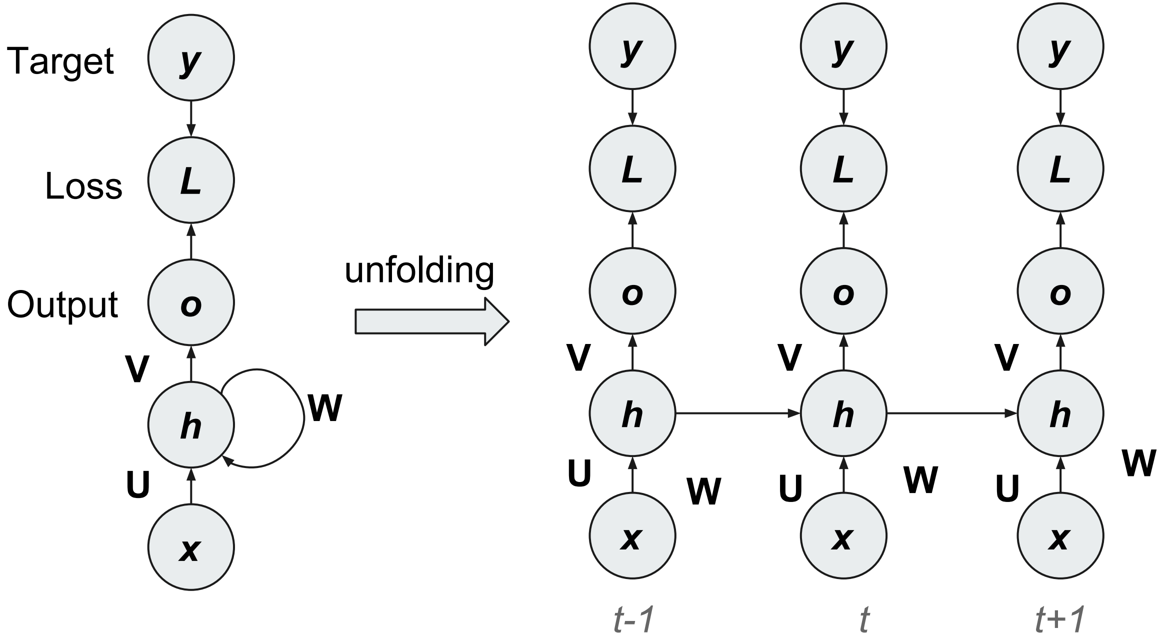
$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \overbrace{w_2\sigma'(z_2)}^{< \frac{1}{4}} \overbrace{w_3\sigma'(z_3)}^{< \frac{1}{4}} \underbrace{w_4\sigma'(z_4)\frac{\partial C}{\partial a_4}}$$

$\updownarrow$ common terms

$$\frac{\partial C}{\partial b_3} = \sigma'(z_3) \overbrace{w_4\sigma'(z_4)\frac{\partial C}{\partial a_4}}$$

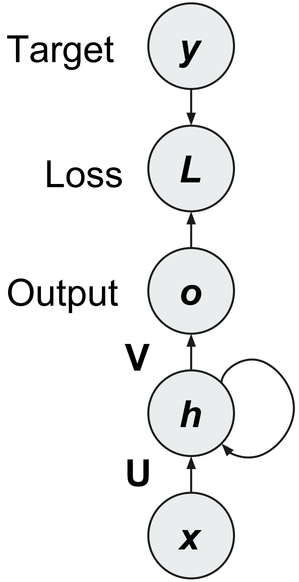
A similar argument holds for “exploding”gradients

Let’s take a relatively complex example...



- maps an input sequence of x values to a corresponding sequence of output o values
- A loss L measures how far each o is from the corresponding training target y
- The loss L internally computes y = softmax(o) and compares this to the target y
- Input to hidden connections parametrized by a weight matrix U,
- Hidden-to-hidden recurrent connections parametrized by a weight matrix W ,
- Hidden-to-output connections parameterize by a weight matrix

Forward Propagation



$$\vec{a}^{(t)} = \vec{b} + \mathbf{W}\tilde{\mathbf{h}}^{(\mathbf{t-1})} + \mathbf{U}\tilde{\mathbf{x}}^{(\mathbf{t})}$$

$$\vec{h}^{(t)} = \tanh(\vec{a}^{(t)})$$

$$\vec{o}^{(t)} = \vec{c} + \mathbf{V}\tilde{\mathbf{h}}^{(\mathbf{t})}$$

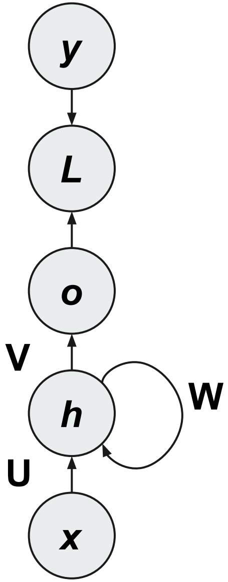
$$\hat{\vec{y}} = softmax(\vec{o}^{(t)})$$

What is the total loss for the output sequence?

$$L(\{\vec{x}^{(1)}, \ldots\ldots, \vec{x}^{(\tau)}\}, \{\vec{y}^{(1)}, \ldots\ldots, \vec{y}^{(\tau)}\}) = \Sigma_t L^{(t)}$$
$$= -\Sigma_t \log p_{model}(y^{(t)} | \{\vec{x}^{(1)}, \ldots\ldots, \vec{x}^{(\tau)}\})$$

- Recall that training requires us to compute the gradients over this log likelihood (loss) function
- Expensive!!
    - Forward propagation from left to right of the unrolled graph
    - Backward propagation from right to left
    - O(\tau) computation is inherently serial; cannot be parallel, needs O(\tau) memory too
- New training algorithm: Backward propagation through time (BPTT)
- Same holds for recurrence between hidden units

Understanding the computational graph...



$$\vec{a}^{(t)} = \vec{b} + \mathbf{W}\tilde{\mathbf{h}}^{(\mathbf{t-1})} + \mathbf{U}\tilde{\mathbf{x}}^{(\mathbf{t})}$$

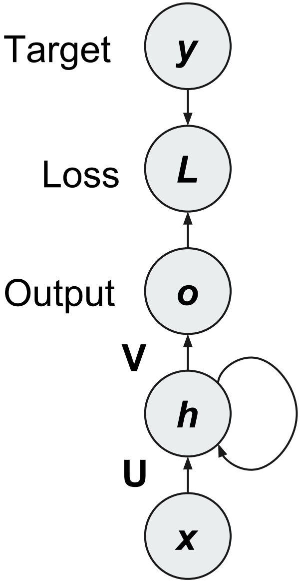
$$\vec{h}^{(t)} = \tanh(\vec{a}^{(t)})$$

$$\vec{o}^{(t)} = \vec{c} + \mathbf{V}\tilde{\mathbf{h}}^{(\mathbf{t})}$$

$$\hat{\vec{y}} = softmax(\vec{o}^{(t)})$$

Parameters

Computing the gradients (1)



For each node N, we need to evaluate gradient…

$$\nabla_N L \qquad \frac{\partial L}{\partial L^{(t)}} = 1$$

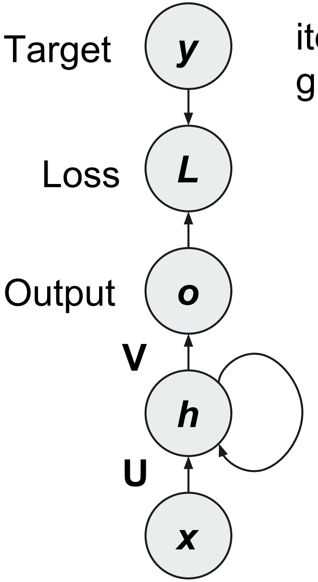
The gradient $\nabla_{\vec{o}^{(t)}} L$ for all (i, t), is as follows

$$(\nabla_{\vec{o}^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y_i}^{(t)} - \vec{1}_{i,y^{(t)}}$$

We start working backward from the end of the sequence. At the final step h only has o as its descendent.

$$\nabla_{\vec{h}^{(\tau)}} = \mathbf{V}^T \nabla_{\vec{o}^{(\tau)}} L$$

Computing the gradients (2)



iterate backward in time to back-propagate gradients through time

$$\begin{aligned}
\nabla_{\boldsymbol{c}} L &= \sum_t \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}}\right)^{\top} \nabla_{\boldsymbol{o}^{(t)}} L = \sum_t \nabla_{\boldsymbol{o}^{(t)}} L, \\
\nabla_{\boldsymbol{b}} L &= \sum_t \left(\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}^{(t)}}\right)^{\top} \nabla_{\boldsymbol{h}^{(t)}} L = \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \nabla_{\boldsymbol{h}^{(t)}} L \\
\nabla_{\boldsymbol{V}} L &= \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}}\right) \nabla_{\boldsymbol{V}} o_i^{(t)} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \boldsymbol{h}^{(t)\top}, \\
\nabla_{\boldsymbol{W}} L &= \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)} \\
&= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{h}^{(t-1)\top}, \\
\nabla_{\boldsymbol{U}} L &= \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{\boldsymbol{U}^{(t)}} h_i^{(t)} \\
&= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{x}^{(t)\top},
\end{aligned}$$

$$\nabla_{\vec{h}^{(t)}} L = \left(\frac{\partial \vec{h}^{(t+1)}}{\partial \vec{h}^{(t)}}\right)^T (\nabla_{\vec{h}^{(t+1)}} L) + \left(\frac{\partial \vec{o}^{(t)}}{\partial \vec{h}^{(t)}}\right)^T (\nabla_{\vec{o}^{(t)}} L)$$

$$\mathbf{W}^T (\nabla_{\vec{h}^{(t+1)}} L) \operatorname{diag}\left(1 - (\vec{h}^{(t+1)})^2\right) + \mathbf{V}^T (\nabla_{\vec{o}^{(t)}} L)$$

diagonal matrix calculating the gradients along the elements of the hidden unit

Computing gradients is hard...

At any given time t, there is a need to look τ steps behind to get the right gradients

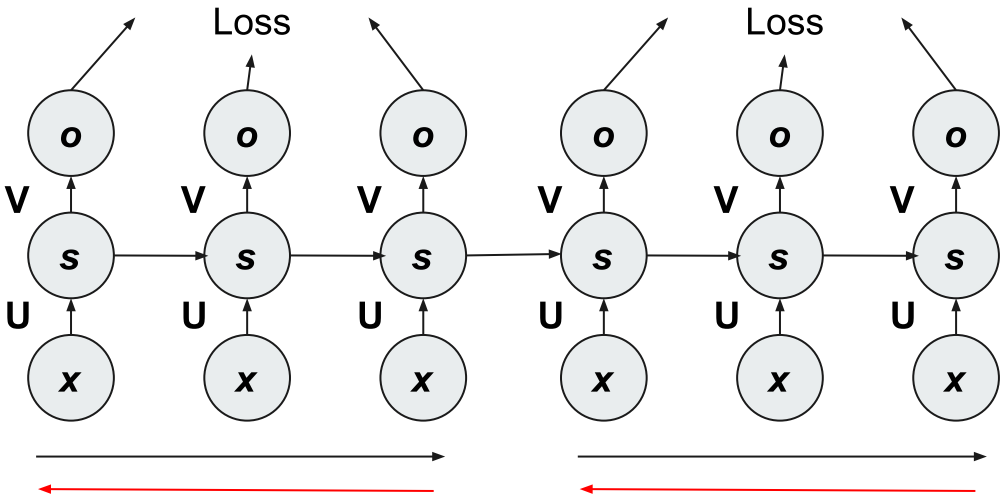
The τ steps to be taken can be arbitrarily large:

- We may want to capture dependencies in the sequence long enough
- How long these dependencies are is unknown a priori

Training a RNN can be hard: need practical solutions to solve this problem

- Try to stop BPTT to some number of steps
- Change the internal network representation to ensure “gated” information flow

Solution 1: Truncate Backprop...



- Run forward and backward through chunks of the sequence instead of whole sequence
- Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps

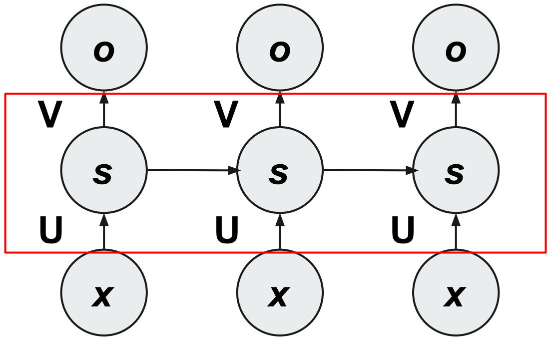
Solution 2: Handling vanishing/exploding gradients by changing recurrent functions

The tanh () function has a gradient behavior that can potentially vanish/explode

Replace the single tanh with additional layers

Long Short Term Memory (LSTM)

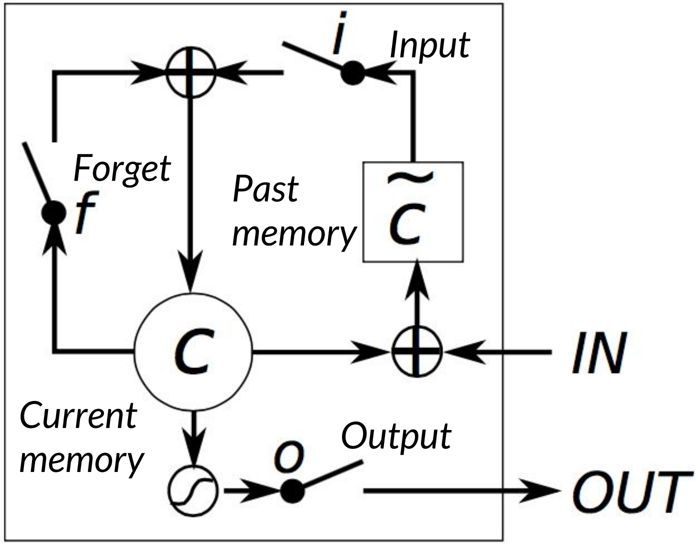
Gated Recurrent Units (GRU)



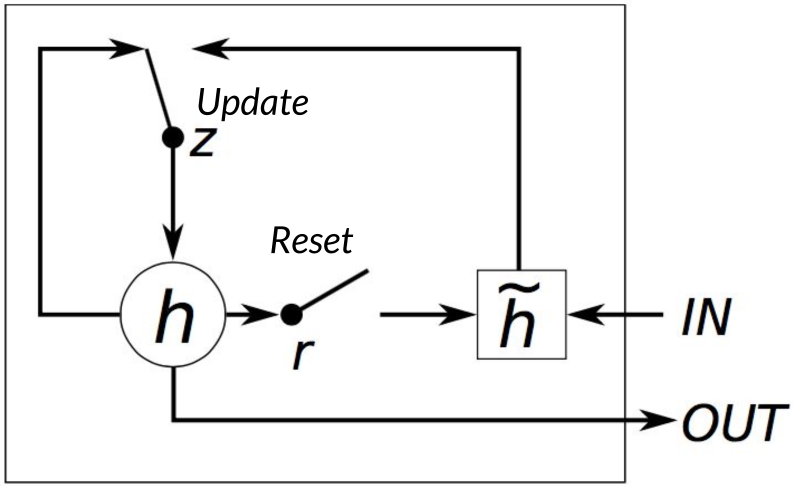
$$s_t = \tanh(Ux_t + Ws_{t-1})$$

$$\hat{o}_t = \mathrm{softmax}(Vs_t)$$

“Gating” Information

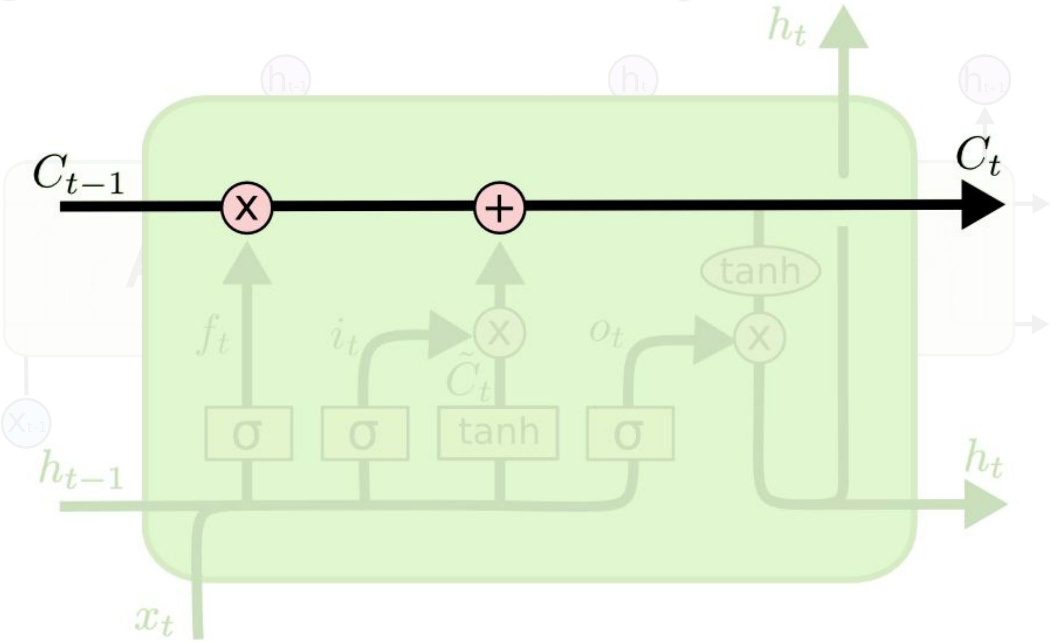


LSTM: Long Short Term Memory



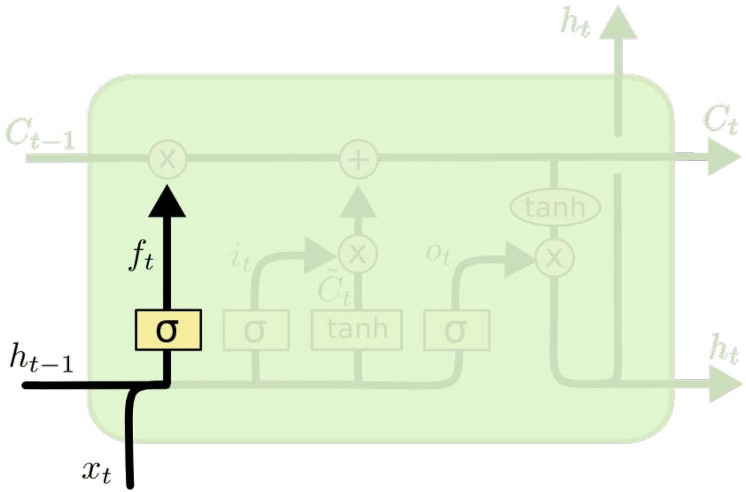
GRU: Gated Recurrent Units

Long Short Term Memory (LSTM)



http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM (1): Controlling information let through

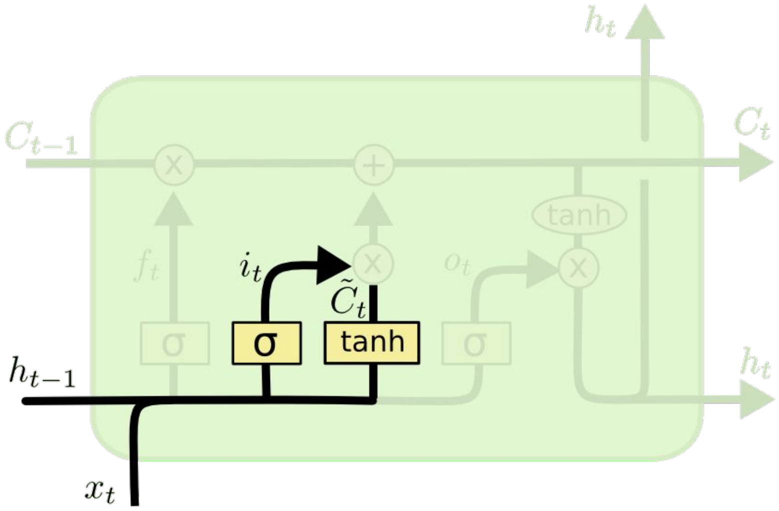


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right)$$

Intuitively, forget gate keeps track of what information to “lose”
Or how to weigh the information such that they can be propagated further

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM (2): Controlling information let through



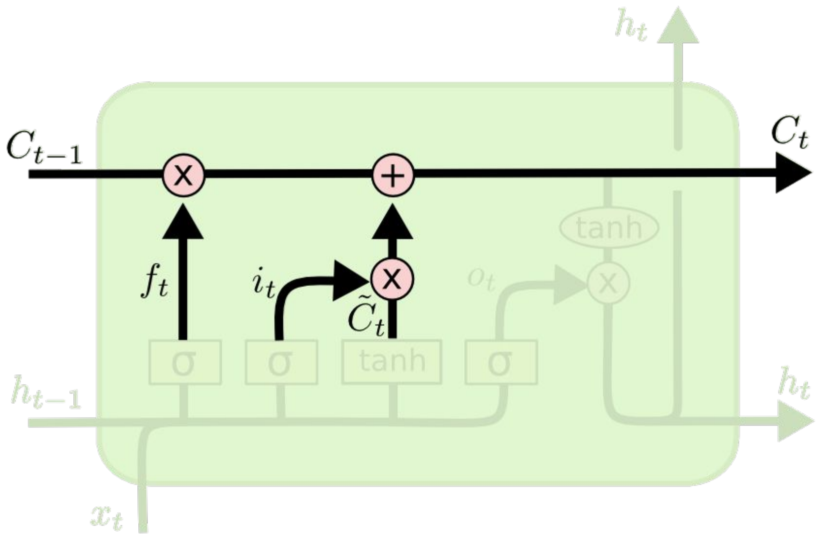
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

Next step is to keep track of what information we are going to store in the cell
Sigmoid layer determines which values to update
Tanh creates a vector of new candidate values

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM (3): Controlling information let through

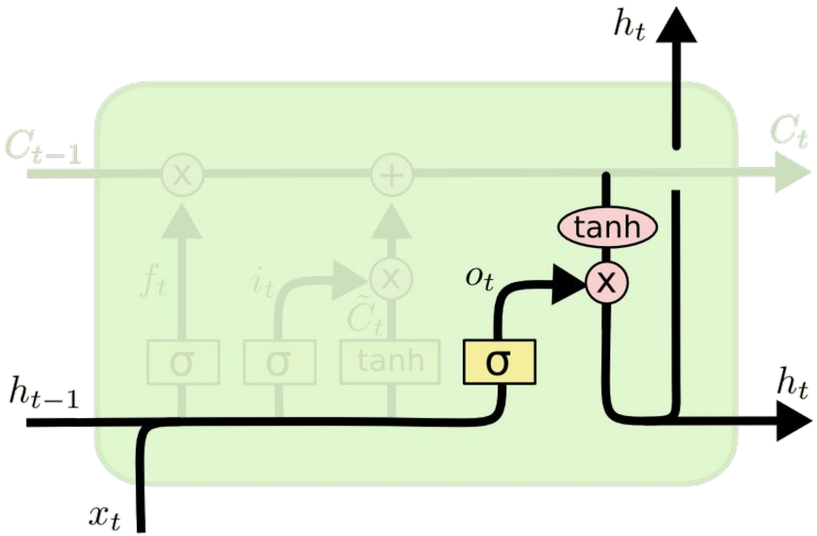


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Next step: update the old cell state with the new cell state
Ct-1 is already available, just a simple vector add is sufficient to get this state

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM (4): Controlling information let through



Decide what we are going to ouput: determined by a filter
sigmoid layer which decides what parts of the cell state we're going to output

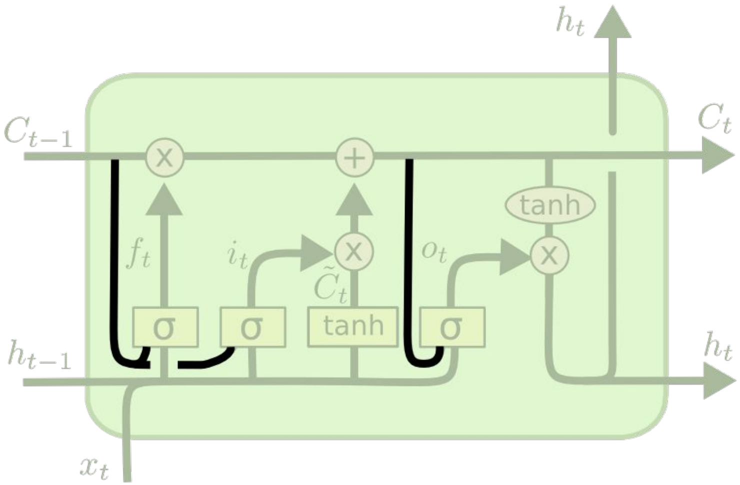
$$o_t = \sigma\left(W_o \left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

Tanh decides what values should be output (by quashing values between -1 and +1

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

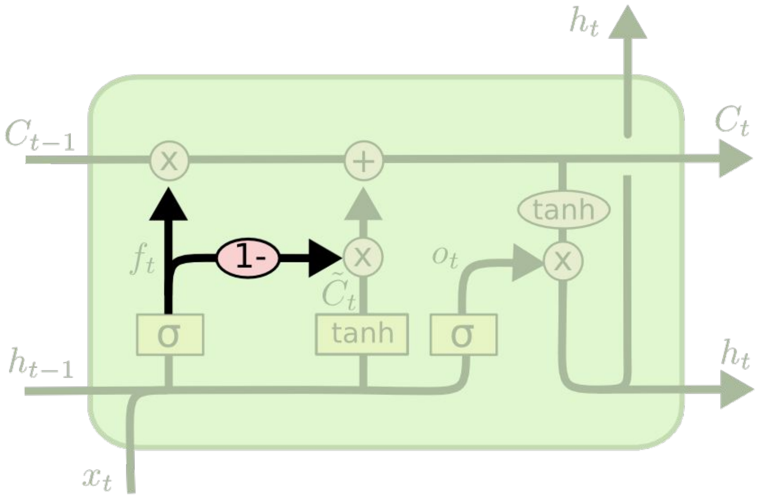
Variants of LSTM



$$f_t = \sigma\left(W_f \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \;+\; b_f\right)$$
$$i_t = \sigma\left(W_i \cdot [\boldsymbol{C_{t-1}}, h_{t-1}, x_t] \;+\; b_i\right)$$
$$o_t = \sigma\left(W_o \cdot [\boldsymbol{C_t}, h_{t-1}, x_t] \;+\; b_o\right)$$

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

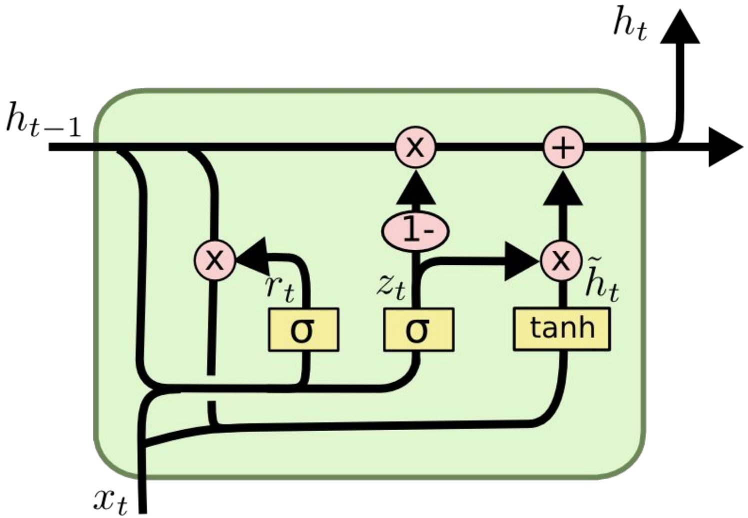
Variants of LSTM (2)



$$C_t = f_t * C_{t-1} + (1 - \boldsymbol{f_t}) * \tilde{C}_t$$

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Gated recurrent unit (GRU)



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

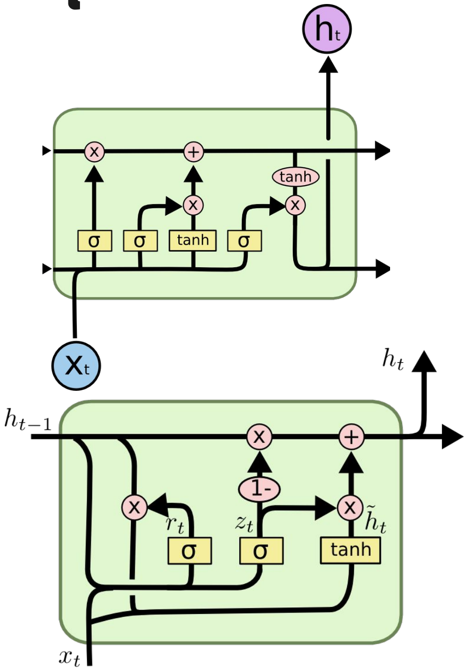
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Equivalence of LSTM and GRU



$$f_t = \sigma(W_f.[h_{t-1}, x_t] + b_f)$$

$$i_t = \sigma(W_i.[h_{t-1}, x_t] + b_i)$$

$$o_t = \sigma(W_o.[h_{t-1}, x_t] + b_o)$$

$$\hat{C}_t = \tanh(W_c.[h_{t-1}, x_t] + b_c)$$

$$C_t = f_t * C_{t-1} + i_t * \hat{C}_t$$

$$h_t = o_t * \tanh(C_t)$$

$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

What you must have learned thus far...

General principles of a recurrent neural network (RNN)

Training an RNN comes with unique challenges:

- Propagating sequences makes it less amenable for parallel implementations
- Vanishing/exploding gradients can be a problem

Variants of a RNN cell using LSTM and GRU

Next class: building a minimal RNN for Language modeling

Thank you!!
ramanathana@ornl.gov