Recurrent Neural Networks

By

Dr. T. Hitendra Sarma

Associate Professor

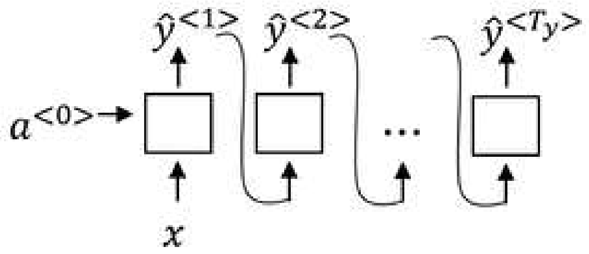
Department of IT

Vasavi College of Engineering, Ibrahim Bagh, Hyderabad

Part-2

Important design patterns for RNNs

- Some examples of important design patterns for recurrent neural networks include the following:
  - Recurrent networks that produce an output at each time step and have recurrent connections between hidden units.
  - Recurrent networks that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step.
  - Recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output.



Notations

- The Training loss of a recurrent network that 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.
- When using softmax outputs, we assume o is the unnormalized log probabilities.
- The loss L internally computes yˆ = softmax(o) and compares this to the target y.
- The RNN has input to hidden connections parametrized by a weight matrix U, hidden-to-hidden recurrent connections parametrized by a weight matrix W, and hidden-to-output connections parametrized by a weight matrix V

Forward propagation equations

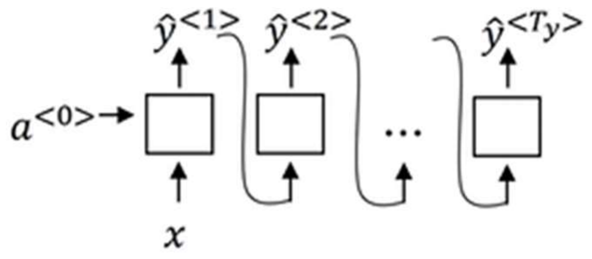
- We now develop the forward propagation equations for the RNN.
- Note that the choice of activation function for the hidden units is not given.
- Here we assume the hyperbolic tangent activation function.
- We assume that the output is discrete, as if the RNN is used to predict words or characters.
- A natural way to represent discrete variables is to regard the output o as giving the unnormalized log probabilities of each possible value of the discrete variable.

Forward propagation equations

- We can then apply the softmax operation as a post-processing step to obtain a vector yˆ of normalized probabilities over the output.
- Forward propagation begins with a specification of the initial state h (0) . Then, for each time step from t = 1 to t = τ, we apply the following update equations:

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

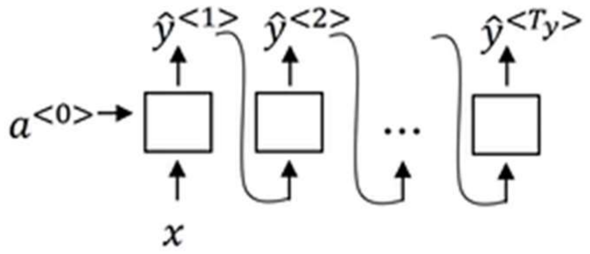
$$\begin{aligned}\boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\ \boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\ \hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)})\end{aligned}$$



Loss Function

- This is an example of a recurrent network that maps an input sequence to an output sequence of the same length.
- The total loss for a given sequence of x values paired with a sequence of y values would then be just the sum of the losses over all the time steps.
- Computing the gradient of this loss function L with respect to the parameters is an expensive operation.
- The gradient computation involves performing a forward propagation pass moving left to right followed by a backward propagation pass moving right to left through the graph.

$$
\begin{aligned}
& L \quad \{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(\tau)}\},\{\boldsymbol{y}^{(1)},\ldots,\boldsymbol{y}^{(\tau)}\} \\
&= \underset{t}{\quad} L^{(t)} \\
&= - \underset{t}{\quad} \log p_{\text{model}} \quad y^{(t)} \mid \{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(t)}\} \quad ,
\end{aligned}
$$

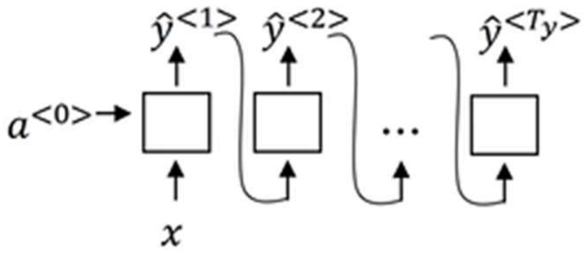


Back-propagation through time

- The runtime is O(τ) and cannot be reduced by parallelization because the forward propagation graph is inherently sequential; each time step may only be computed after the previous one.
- States computed in the forward pass must be stored until they are reused during the backward pass, so the memory cost is also O(τ).
- The back-propagation algorithm applied to the unrolled graph with O(τ) cost is called back-propagation through time or BPTT.

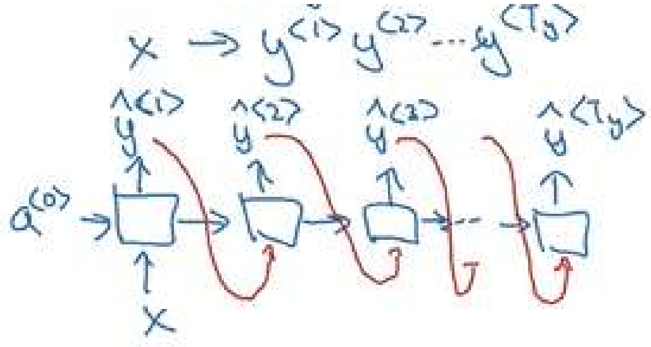
Note

- The network with recurrent connections only from the output at one time step to the hidden units at the next time step is strictly less powerful because it lacks hidden-to-hidden recurrent connections.
- For example, it cannot simulate a universal Turing machine. Because this network lacks hidden-to-hidden recurrence, it requires that the output units capture all of the information about the past that the network will use to predict the future.



Note

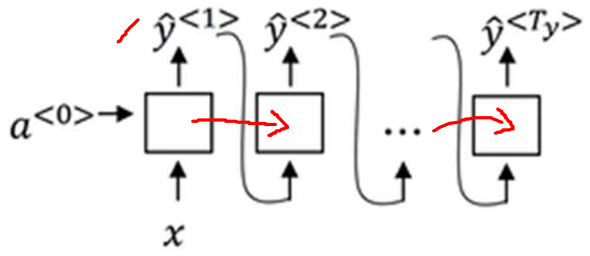
- The advantage of eliminating hidden-to-hidden recurrence is that, for any loss function based on comparing the prediction at time t to the training target at time t, all the time steps are decoupled.
- Training can thus be parallelized, with the gradient for each step t computed in isolation.
- There is no need to compute the output for the previous time step first, because the training set provides the ideal value of that output.



Teacher forcing

- Models that have recurrent connections from their outputs leading back into the model may be trained with *teacher forcing*.
- Teacher forcing is a procedure that emerges from the maximum likelihood criterion, in which during training the model receives the ground truth output y (t) as input at time t + 1.
- We can see this by examining a sequence with two time steps. The conditional maximum likelihood criterion is

$$\log p \;\; \boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}$$



Teacher forcing

- The conditional maximum likelihood criterion is

$$\begin{aligned}&\log p\ \ \boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)} \\ = &\log p\ \ \boldsymbol{y}^{(2)} \mid \boldsymbol{y}^{(1)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\ \ + \log p\ \ \boldsymbol{y}^{(1)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\end{aligned}$$

- In this example, we see that at time t = 2, the model is trained to maximize the conditional probability of y (2) given the x sequence so far and the previous y value from the training set.
- Maximum likelihood thus specifies that during training, rather than feeding the model's own output back into itself, these connections should be fed with the target values specifying what the correct output should be.
- Teacher forcing is allowing us to avoid back-propagation through time in models that lack hidden-to-hidden connections.

Note

- Teacher forcing may still be applied to models that have hidden-to-hidden connections so long as they have connections from the output at one time step to values computed in the next time step.
- However, as soon as the hidden units become a function of earlier time steps, the BPTT algorithm is necessary.
- Some models may thus be trained with both teacher forcing and BPTT

BPTT algorithm

- Consider the nodes of our computational graph include the parameters U, V, W, b and c as well as the sequence of nodes indexed by t for x (t) , h (t) , o (t) and L (t) .
- Here the parameters are the bias vectors b and c along with the weight matrices U, V and W, respectively for input-to-hidden, hidden-to-output and hidden-to- hidden connections.
- For each node N we need to compute the gradient ∇ L recursively, based on the gradient computed at nodes that follow it in the graph.

- We start the recursion with the nodes immediately preceding the final loss.

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

- In this derivation we assume that the outputs o (t) are used as the argument to the softmax function to obtain the vector yˆ of probabilities over the output.
- We also assume that the loss is the negative log-likelihood of the true target y (t) given the input so far.

Computing Gradients

- The gradient ∇ L on the outputs at time step t, for all i,t, is as follows:

$$(\nabla \quad L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \quad {}_{i,y} \quad . \qquad L \xleftarrow{\qquad\qquad} R$$

- We work our way backwards, starting from the end of the sequence.
- At the final time step τ , h (τ) only has o (τ) as a descendent, so its gradient is simple:

$$\nabla \quad L = (\nabla \quad L) \frac{\partial \boldsymbol{o}^{(\tau)}}{\partial \boldsymbol{h}^{(\tau)}} = (\nabla \quad L) \boldsymbol{V} \quad ✓$$

Computing Gradients

- We can then iterate backwards in time to back-propagate gradients through time, from t = τ − 1 down to t= 1, noting that h (t) (for t < τ) has as descendents both o (t) and h (t+1) .
- Its gradient is thus given by

$$\nabla \qquad L = (\nabla \qquad L)\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}} + (\nabla \quad L)\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}$$

$$= (\nabla \qquad L)\,\underline{\mathrm{diag}}\ \ 1 - \ \ \boldsymbol{h}^{(t+1)}\ \ ^{2}\ \ \boldsymbol{W} + (\nabla \quad L)\,\boldsymbol{V}$$

where diag $1 - \ \boldsymbol{h}^{(t+1)}\ ^{2}$ indicates the diagonal matrix containing the elements $1 - (h_i^{(t+1)})^2$. This is the Jacobian of the hyperbolic tangent associated with the hidden unit i at time $t + 1$.

- Once the gradients on the internal nodes of the computational graph are obtained, we can obtain the gradients on the parameter nodes, which have descendents at all the time steps:

$$\nabla\ L = \ {}_{t}(\nabla \quad L)\frac{\partial \boldsymbol{o}^{(t)}}{\partial \underline{\boldsymbol{c}}} = \ {}_{t}\nabla \quad L$$

$$\nabla\ L = \ {}_{t}(\nabla \quad L)\frac{\partial \boldsymbol{h}^{(t)}}{\partial \underline{\boldsymbol{b}}} = \ {}_{t}(\nabla \quad L)\,\mathrm{diag}\ \ 1 - \ \ \boldsymbol{h}^{(t)}\ \ {}^{2}$$

$$\nabla\ \ L = \ {}_{t}(\nabla \quad L)\frac{\partial \boldsymbol{o}^{(t)}}{\partial \underline{\boldsymbol{V}}} = \ {}_{t}(\nabla \quad L)\,\boldsymbol{h}^{(t)}$$

$$\nabla\ \ L = \ {}_{t}(\nabla \quad L)\frac{\partial \boldsymbol{h}^{(t)}}{\partial \underline{\boldsymbol{W}}} = \ {}_{t}(\nabla \quad L)\,\mathrm{diag}\ \ 1 - \ \ \boldsymbol{h}^{(t)}\ \ {}^{2}\ \ \boldsymbol{h}^{(t-1)}$$

$$\nabla\ \ L = \ {}_{t}(\nabla \quad L)\frac{\partial \boldsymbol{h}^{(t)}}{\partial \underline{\boldsymbol{U}}} = \ {}_{t}(\nabla \quad L)\,\mathrm{diag}\ \ 1 - \ \ \boldsymbol{h}^{(t)}\ \ {}^{2}\ \ \boldsymbol{x}^{(t)}$$

- When we use a predictive log-likelihood training objective, we train the RNN to estimate the conditional distribution of the next sequence element y (t) given the past inputs.
- This may mean that we maximize the log-likelihood

$$\log p(\boldsymbol{y}^{(t)} \mid \underline{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}})$$

or, if the model includes connections from the output at one time step to the next time step

$$\log p(\boldsymbol{y}^{(t)} \mid \underline{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}}, \underline{\boldsymbol{y}^{(1)}, \ldots, \boldsymbol{y}^{(t-1)}})$$

Vanishing Gradients

- So, we learned about how RNNs work and how they can be applied to problems like
  - name entity recognition
  - language modeling
- We also saw how backpropagation can be used to train in RNN.
- One of the problems with a basic RNN algorithm is that it runs into vanishing gradient problems.
- Let us discuss some solutions that will help to address this problem

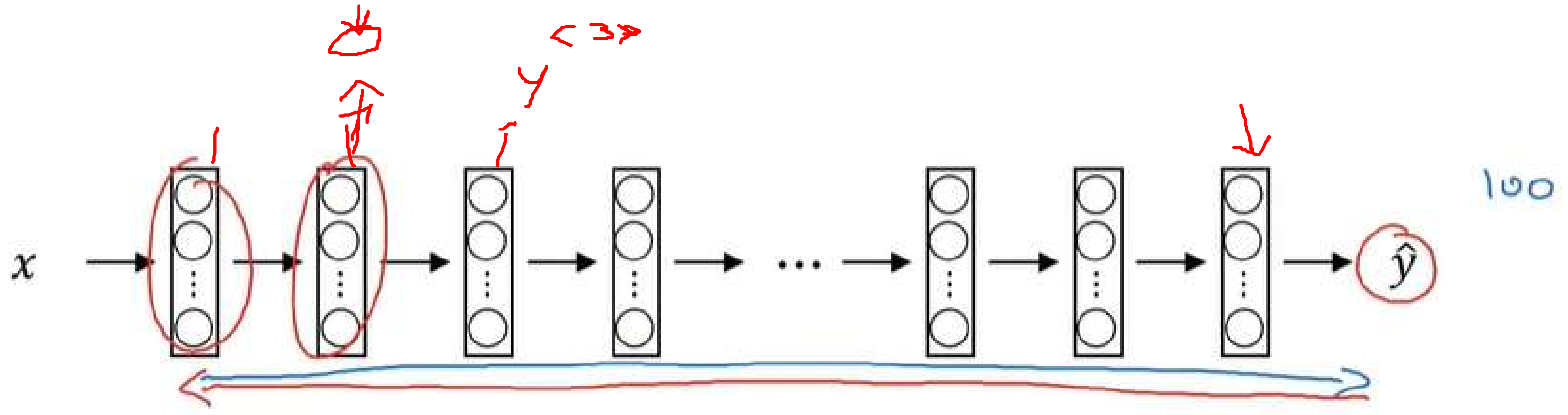
Long Term Dependencies – Language Modeling Example

- Consider the sentences;
  - The cat which already ate and maybe already ate a bunch of food that was delicious dot, dot, dot, dot, was full.
  - The cats which already ate a bunch of food was delicious, and apples, and pears, and so on, were full.
  - So to be consistent, it should be cat-was or cats-were.
  - And this is one example of when language can have very long-term dependencies, where it worked at this much earlier can affect what needs to come much later in the sentence.

Vanishing Gradient in RNNs

- The basic RNN that are explained so far is not very good at capturing very long-term dependencies.
- Why?
  - Training very deep neural networks - the vanishing gradients problem.
  - In a very, very deep neural network say, 100 layers or even much deeper than you would carry out forward prop, from left to right and then back prop.
  - If this is a very deep neural network, then the gradient from just output y, would have a very hard time propagating back to affect the weights of these earlier layers, to affect the computations in the earlier layers.
  - Hence for an RNN with a similar problem, you have forward prop came from left to right, and then back prop, going from right to left. And it can be quite difficult, because of the same vanishing gradients problem, for the outputs of the errors associated with the later time steps to affect the computations that are earlier.

Understanding vanishing gradient

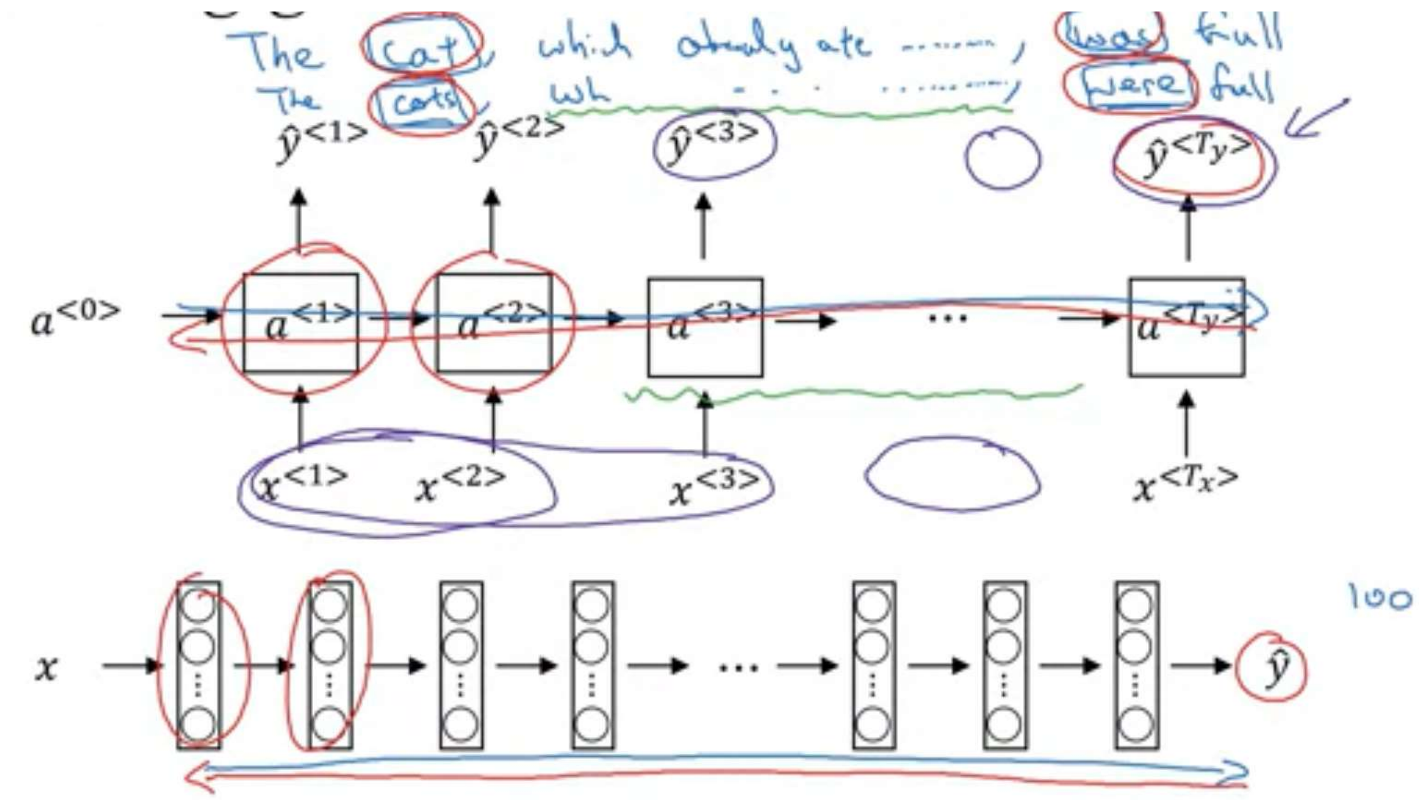


And so in practice, what this means is, it might be difficult to get a neural network to realize that it needs to memorize the just see a singular noun or a plural noun, so that later on in the sequence that can generate either was or were, depending on whether it was singular or plural.

The basic RNN model has many local influences, meaning that the output y^<3> is mainly influenced by values close to y^<3>

Make RNNs tend to be very good at capturing long-range dependencies.

Vanishing Gradients with RNNs



Exploding Gradient – Gradient Clipping

- When we are doing back prop, the gradients should not just decrease exponentially, they may also increase exponentially with the number of layers we go through.
- Vanishing gradients tends to be the bigger problem with training RNNs
- Although when exploding gradients happens, it can be catastrophic because the exponentially large gradients can cause your parameters to become so large that your neural network parameters get really messed up.
- So it turns out that exploding gradients are easier to spot because the parameters just blow up and you might often see NaNs, or not a numbers, meaning results of a numerical overflow in your neural network computation.
- Solution:
  - Gradient clipping: Look at your gradient vectors, and if it is bigger than some threshold, re-scale some of your gradient vector so that is not too big. So there are clips according to some maximum value.

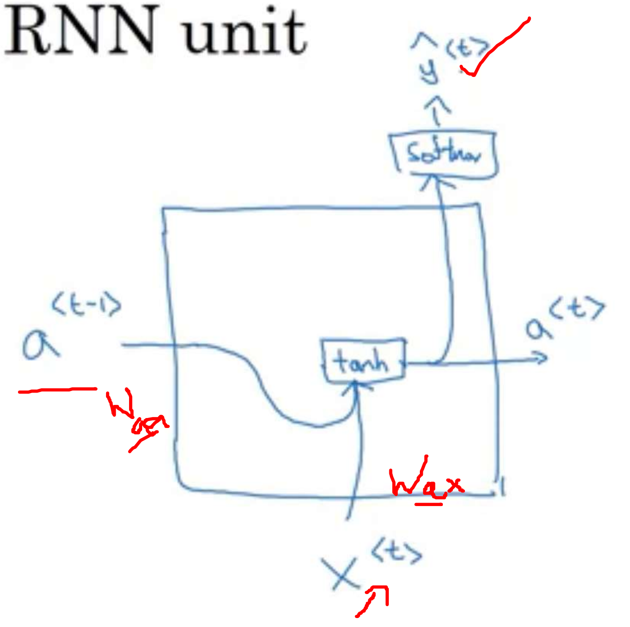
GRU
The Gated Recurrent Units

how GRU allows to learn very long range connections in a sequence?

Gated Recurrent Unit (GRU)

- Gated Recurrent Unit which is a modification to the RNN hidden layer that makes it much better capturing long range connections and helps a lot with the vanishing gradient problems.

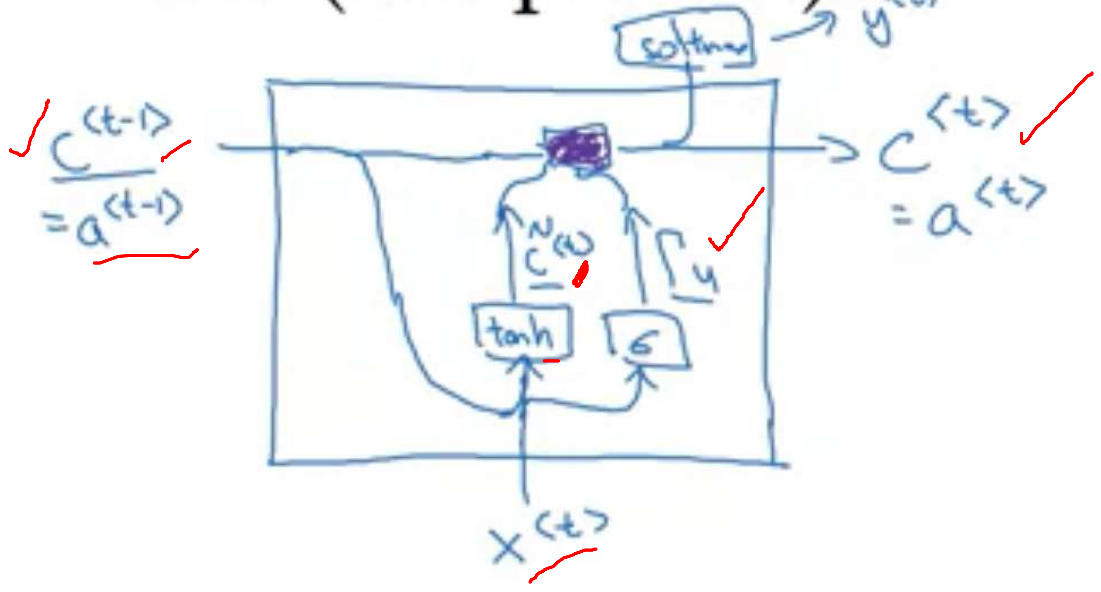
RNN unit



$$a^{<t>} = \overset{\text{tanh}}{g}\underbrace{(W_a[a^{<t-1>}, x^{<t>}] + b_a)}$$

GRU Unit

GRU (simplified)



$c^{\langle t-1\rangle} = a^{\langle t-1\rangle}$
softmax
$\hat{y}^{\langle t\rangle}$
$c^{\langle t\rangle} = a^{\langle t\rangle}$
$\tilde{c}^{\langle t\rangle}$
Γ_u
tanh
σ
$x^{\langle t\rangle}$

$\rightarrow$ c = memory cell

$$\rightarrow c^{\langle t\rangle} = a^{\langle t\rangle}$$

$$\rightarrow \tilde{c}^{\langle t\rangle} = \tanh(W_c[c^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_c)$$

$$\rightarrow \underset{\text{"update"}}{\Gamma_u} = \sigma(W_u[c^{\langle t-1\rangle}, x^{\langle t\rangle}] + b_u)$$

$$c^{\langle t\rangle} = \underset{=1}{\Gamma_u} * \tilde{c}^{\langle t\rangle} + (1-\Gamma_u) * c^{\langle t-1\rangle}$$

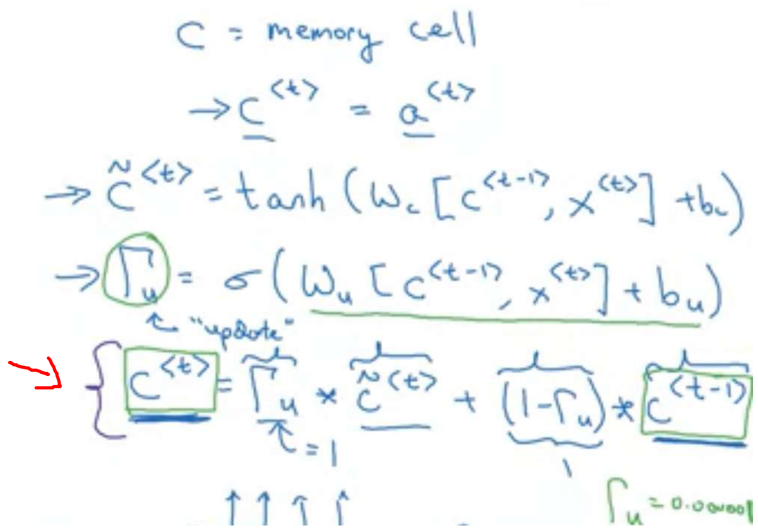
Addressing very long range dependencies

The gate is quite easy to set to zero. So when that is the case, then this updated equation and subsetting c<t> equals c<t-1> . So this is very good at maintaining the value for the cell.

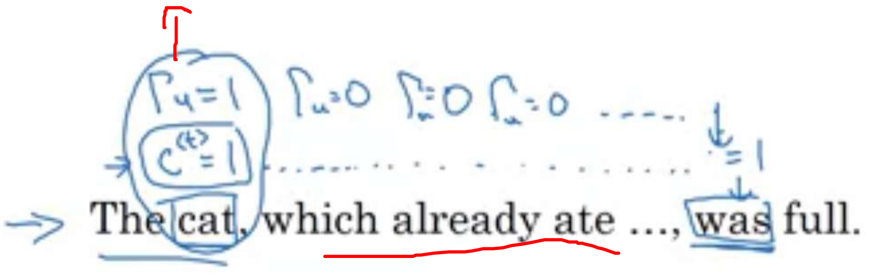
As gamma can be so close to zero, can be 0.000001 or even smaller than that, it doesn't suffer from much of a vanishing gradient problem.

If gamma so close to zero this becomes essentially c<t> equals c<t-1> and the value of c<t> is maintained pretty much exactly even across many time-steps.

So this can help significantly with the vanishing gradient problem and therefore allow a neural network to go on even very long range dependencies, such as a cat and was related even if they're separated by a lot of words in the middle.



C = memory cell
$c^{<t>} = a^{<t>}$
$\tilde{c}^{<t>} = \tanh(W_c[c^{<t-1>}, x^{<t>}] + b_c)$
$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$
"update"
$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1-\Gamma_u) * c^{<t-1>}$
$\Gamma_u = 0.000001$



The Full GRU

$$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$$

$$\Gamma_u = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$

$$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) + c^{<t-1>}$$

Gamma<r> tells you how relevant is c<t-1> to computing the next candidate for c<t>

Long term dependencies & Vanishing Gradients with RNNs

- Long Short-Term Memory (**LSTM**) networks are a type of recurrent neural network capable of learning order dependence in sequence prediction problems.

GRU and LSTM

GRU

$$\underline{\tilde{c}^{<t>}} = \tanh(W_c[\Gamma_r * \underline{c^{<t-1>}}, x^{<t>}] + b_c)$$

$$\underline{\Gamma_u} = \sigma(W_u[c^{<t-1>}, x^{<t>}] + b_u)$$

$$\underline{\Gamma_r} = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$$

$$\underline{c^{<t>}} = \underbrace{\boxed{\Gamma_u}} * \underline{\tilde{c}^{<t>}} + \underbrace{\boxed{(1 - \Gamma_u)}}_{\Gamma_f} * c^{<t-1>}$$

$$\underline{a^{<t>} = c^{<t>}}$$

LSTM

$$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$$

(update) $\Gamma_u = \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u)$

(forget) $\Gamma_f = \sigma(W_f[a^{<t-1>}, x^{<t>}] + b_f)$

(output) $\Gamma_o = \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o)$

$$c^{<t>} = \underline{\Gamma_u} * \underline{\tilde{c}^{<t>}} + \underline{\Gamma_f} * \underline{c^{<t-1>}}$$

$$a^{<t>} = \Gamma_o * c^{<t>}$$

LSTM in Picture

$$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$$

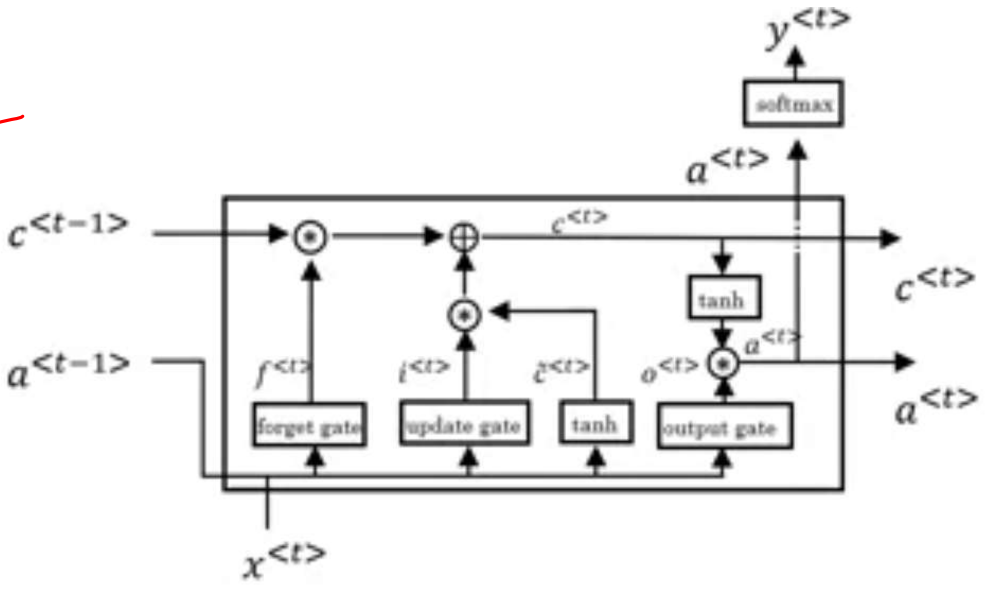
$$\Gamma_u = \sigma(W_u[\, a^{<t-1>}, x^{<t>}] + b_u)$$

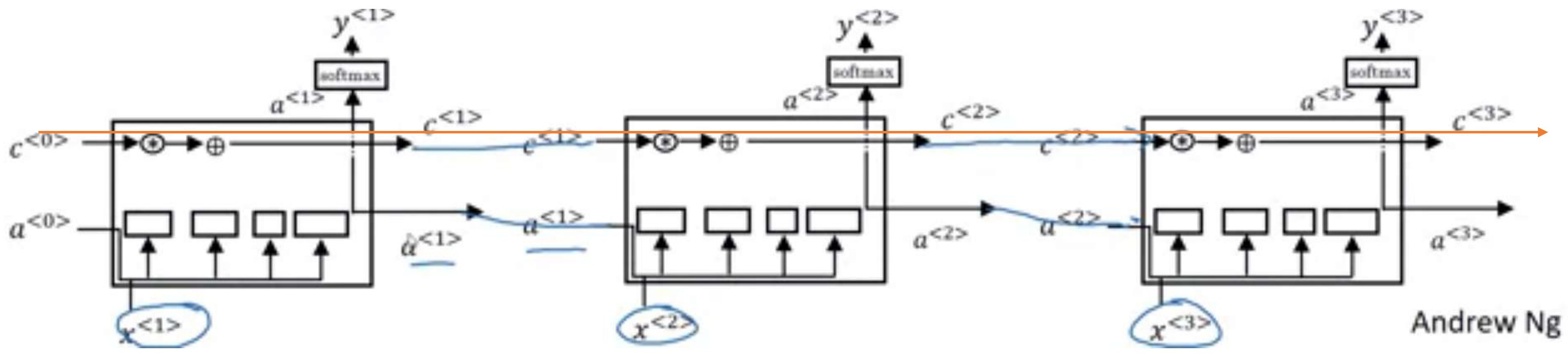
$$\Gamma_f = \sigma(W_f[\, a^{<t-1>}, x^{<t>}] + b_f)$$

$$\Gamma_o = \sigma(W_o[\, a^{<t-1>}, x^{<t>}] + b_o)$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>}$$

$$\underline{a^{<t>}} = \underline{\Gamma_o * \tanh c}^{<t>}$$





Notice is that there is a line at the top that shows how, so long as you set the forget and the update gate appropriately, it is relatively easy for the LSTM to have some value c<0> and have that be passed all the way to the right to have your, maybe, c<3> equals c<0>.

And this is why the LSTM, as well as the GRU, is very good at memorizing certain values even for a long time, for certain real values stored in the memory cell even for many, many timesteps.

Note

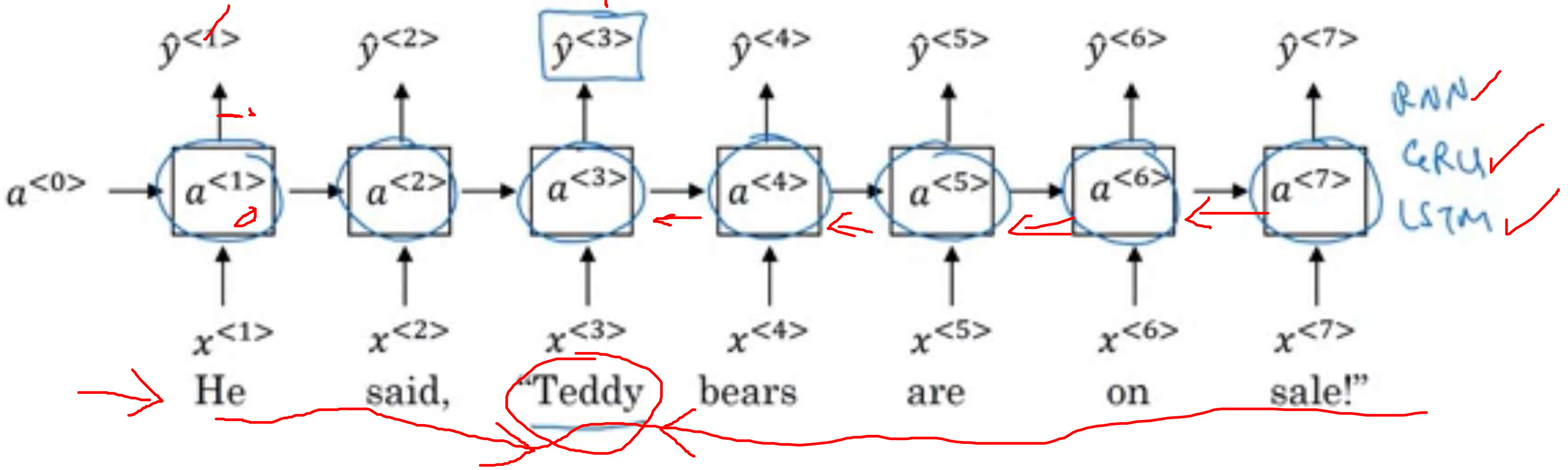
- In the history of deep learning, LSTMs actually came much earlier, and then GRUs were relatively recent invention.
- The advantage of the GRU is that it's a simpler model and so it is actually easier to build a much bigger network, it only has two gates, so computationally, it runs a bit faster. So, it scales the building somewhat bigger models.
- The LSTM is more powerful and more effective since it has three gates instead of two. LSTM has been the historically more proven choice.

Bidirectional RNN (BRNN)

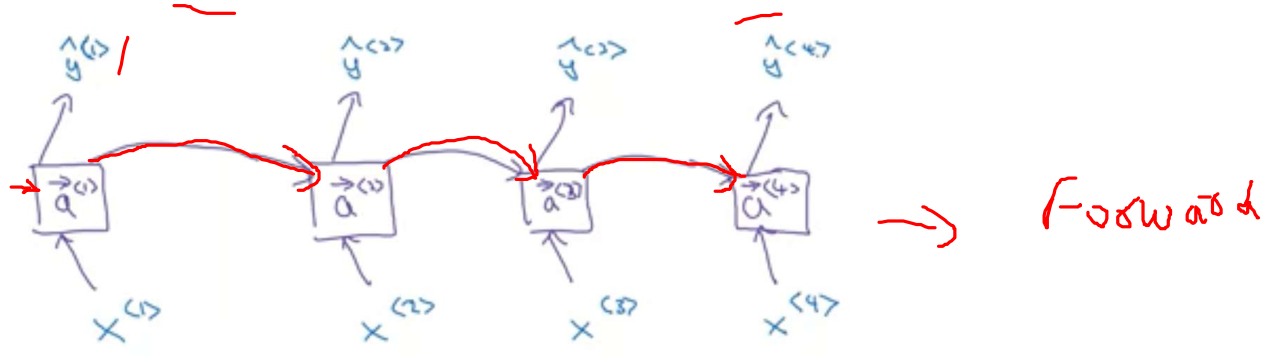
Getting information from the future

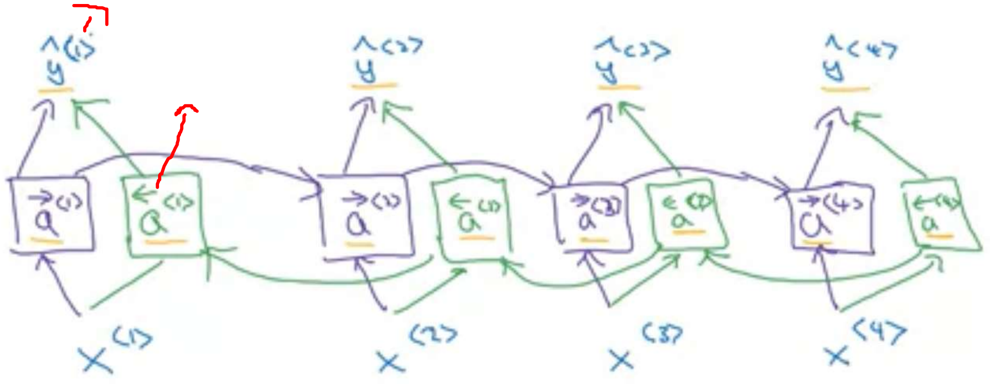
He said, "Teddy bears are on sale!"

He said, "Teddy Roosevelt was a great President!"



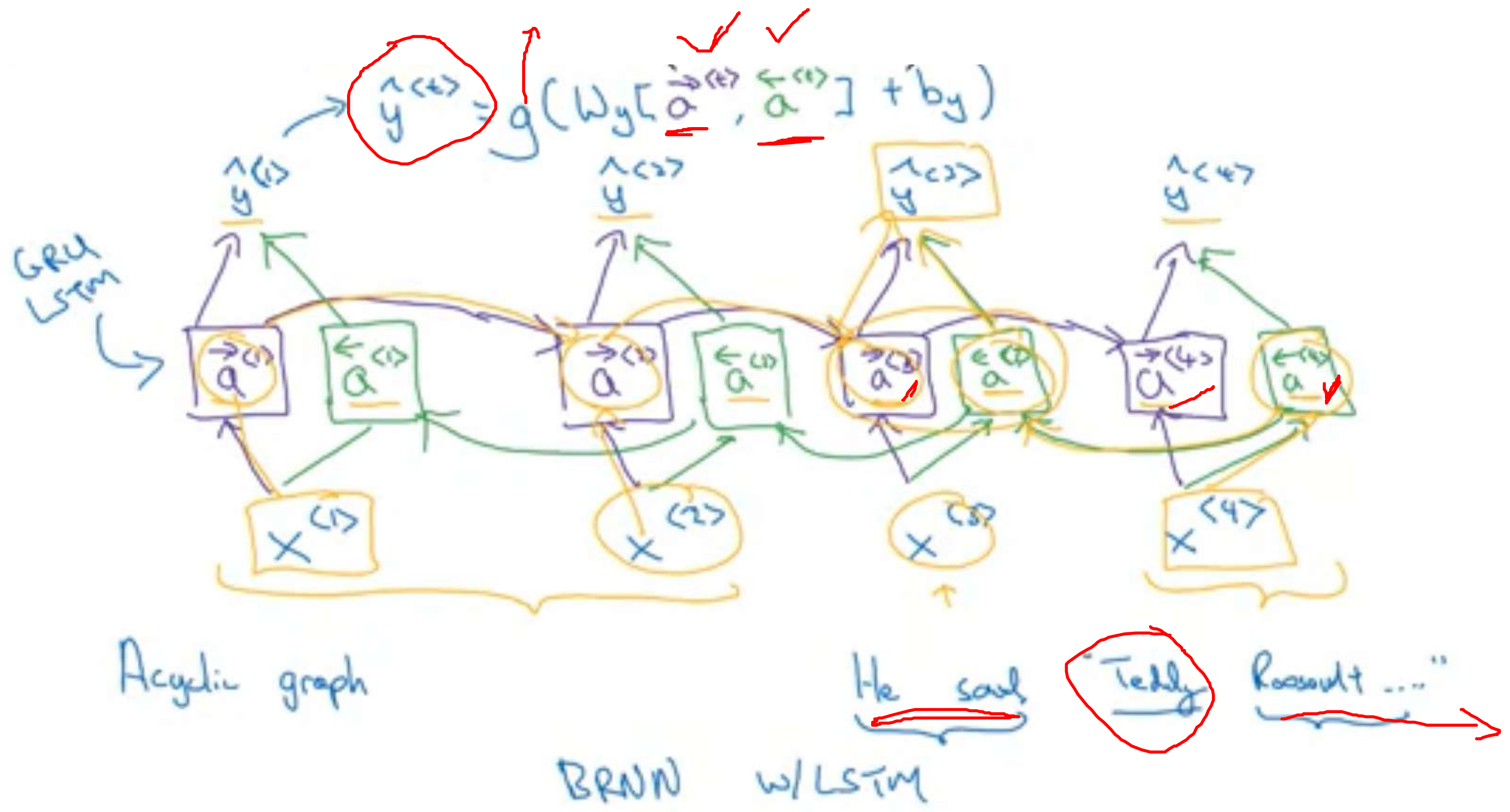
RNNs with Forward and Backward computations





Acyclic graph

BRNN

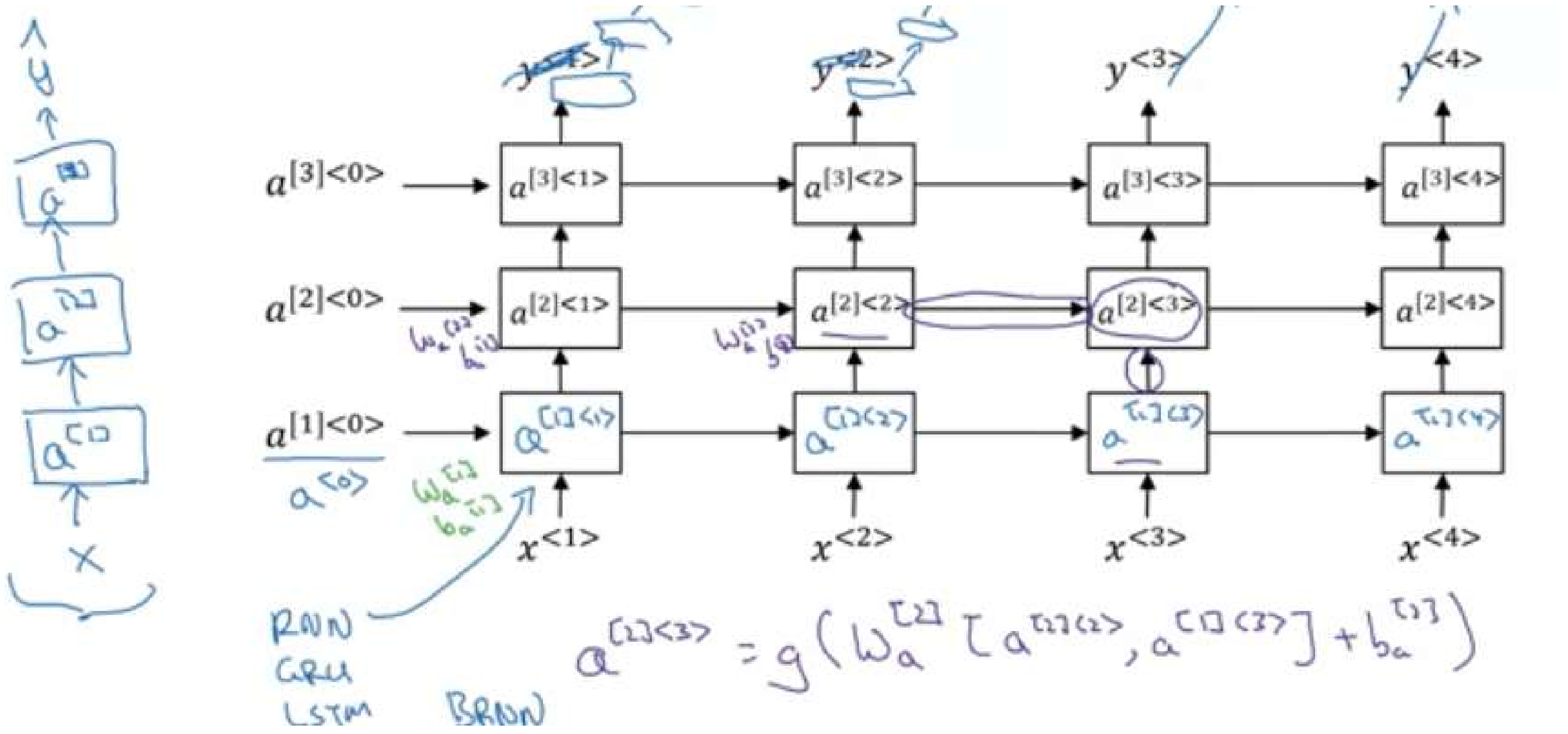


$\hat{y}^{<t>} = g(W_y[\overrightarrow{a}^{<t>}, \overleftarrow{a}^{<t>}] + b_y)$
$\hat{y}^{<1>}$
$\hat{y}^{<2>}$
$\hat{y}^{<3>}$
$\hat{y}^{<4>}$
GRU
LSTM
$\overrightarrow{a}^{<1>}$
$\overleftarrow{a}^{<1>}$
$\overrightarrow{a}^{<2>}$
$\overleftarrow{a}^{<2>}$
$\overrightarrow{a}^{<3>}$
$\overleftarrow{a}^{<3>}$
$\overrightarrow{a}^{<4>}$
$\overleftarrow{a}^{<4>}$
$x^{<1>}$
$x^{<2>}$
$x^{<3>}$
$x^{<4>}$
Acyclic graph
He said "Teddy Roosevelt ...."
BRNN w/ LSTM

Note

- NLP problem - a bidirectional RNN with LSTM blocks both forward and backward
- By making changes to the bidirectional RNN we can have a model that uses RNN and or GRU or LSTM and is able to make predictions anywhere even in the middle of a sequence by taking into account information potentially from the entire sequence.
- The disadvantage of the bidirectional RNN is that you do need the entire sequence of data before you can make predictions anywhere.
    - For example, if you're building a speech recognition system, then the BRNN will let you take into account the entire speech utterance but if you use this straightforward implementation, you need to wait for the person to stop talking to get the entire utterance before you can actually process it and make a speech recognition prediction.
- So for a real type speech recognition applications, they are somewhat more complex modules as well rather than just using the standard bidirectional RNN.
- But for a lot of natural language processing applications where you can get the entire sentence all the same time, the standard BRNN algorithm is actually very effective.

Example Deep RNN



$$a^{[2]<3>} = g(W_a^{[2]}[a^{[2]<2>}, a^{[1]<3>}] + b_a^{[2]})$$

Questions?

You can write your questions

hitendrasarma@ieee.org

hitendrasarma@staff.vce.ac.in