

Implementation of RNNs

Deep Learning

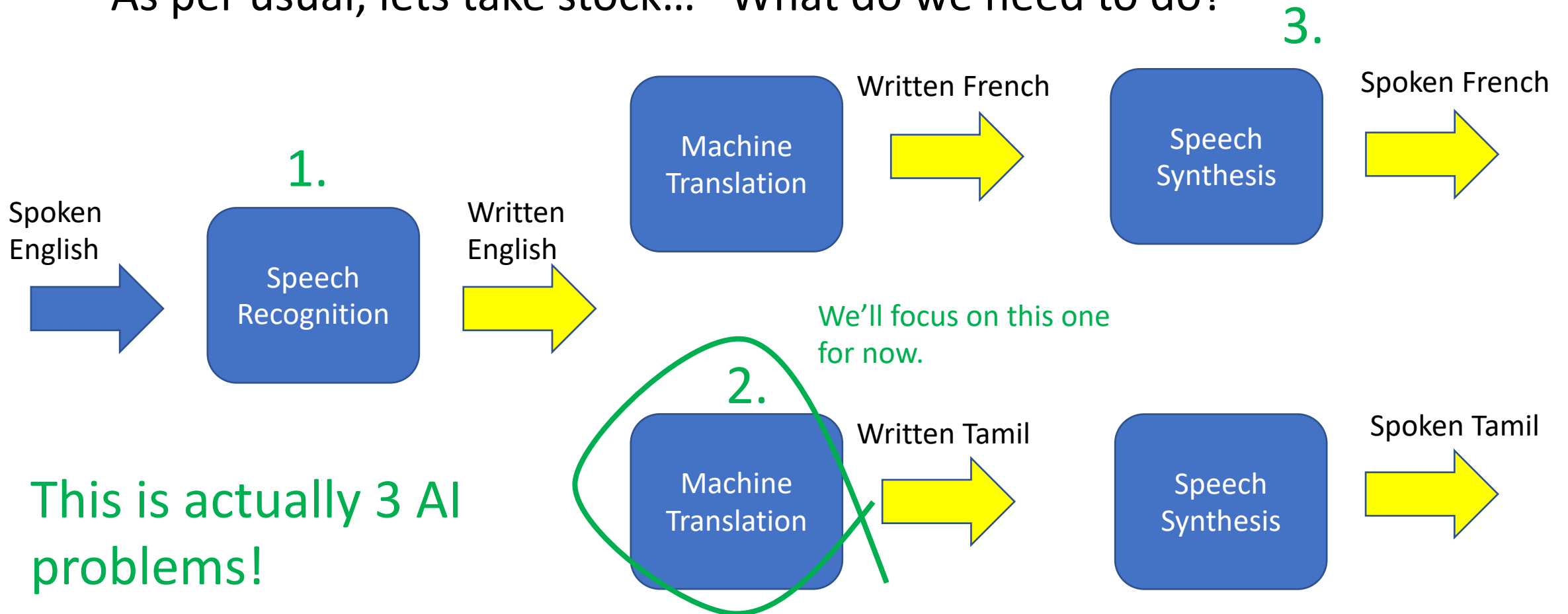
[Brad Quinton](#), [Scott Chin](#)

Case Study: Machine Translation

- Recall, we have been assigned the task of using AI to simultaneously translate all Toronto Raptors basketball games into 24 of the most common languages spoken in Toronto
- Motivated by our new project, we have learned how RNNs give us an elegant and effective way to manage sequential data
- But how do we make implement them? Train them? and apply them to Natural Language Processing (NLP) tasks like machine translation?

Back to our hypothetical case study....

- As per usual, lets take stock... What do we need to do?



RNN Notation

- We have seen some of the notation conventions for RNNs last lecture, but let's formalize that before we dive into implementations
- **Inputs:** $x^{(i)<t>}$ where $i = 1:m$ and $t = 1:T_x^{(i)}$
- **Outputs:** $y^{(i)<t>}$ where $i = 1:m$ and $t = 1:T_y^{(i)}$

RNN Notation

- We have seen some of the notation conventions for RNNs last lecture, but let's formalize that before we dive into implementations

- **Inputs:** $x^{(i)<t>}$ where $i = 1:m$ and $t = 1:T_x^{(i)}$

- **Outputs:** $y^{(i)<t>}$ where $i = 1:m$ and $t = 1:T_y^{(i)}$

Like before, we have
 m training examples.

RNN Notation

- We have seen some of the notation conventions for RNNs last lecture, but let's formalize that before we dive into implementations

- **Inputs:** $x^{(i)<t>}$ where $i = 1:m$ and $t = 1:T_x^{(i)}$

Now, each input and output in the training example has a sequence length T .

- **Outputs:** $y^{(i)<t>}$ where $i = 1:m$ and $t = 1:T_y^{(i)}$

Like before, we have m training examples.

NLP Word Representation

- Let's imagine we have an input sentence:

x : "Cats are nice."

- We know that we will want to make each word an element of our sequence:

$$x^{<1>} = \text{"Cats"}, x^{<2>} = \text{"are"}, x^{<3>} = \text{"nice"}, x^{<4>} = \text{"."}$$

NLP Word Representation

- But, how should we represent each word?

NLP Word Representation

- But, how should we represent each word?
- A standard Artificial Neural Network can only accept numbers as inputs and outputs
- ...otherwise we would have to re-think our parameter representations, and our partial derivatives for gradient descent!
- We we need to find a way to assign each word a number

NLP Word Representation

- We can create an ordered dictionary (often called a Vocabulary) and then assign each word number based on its position in the sequence:


Word	Position
a	1
aaron	2
...	
cats	520
....	
nice	3024
...	
zulu	19,999
<eos> , “.”	20,000

NLP Word Representation

- We can create an ordered dictionary (often called a Vocabulary) and then assign each word number based on its position in the sequence:

Word	Position
a	1
aaron	2
...	
cats	520
....	
nice	3024
...	
zulu	19,999
<eos> , “.”	20,000

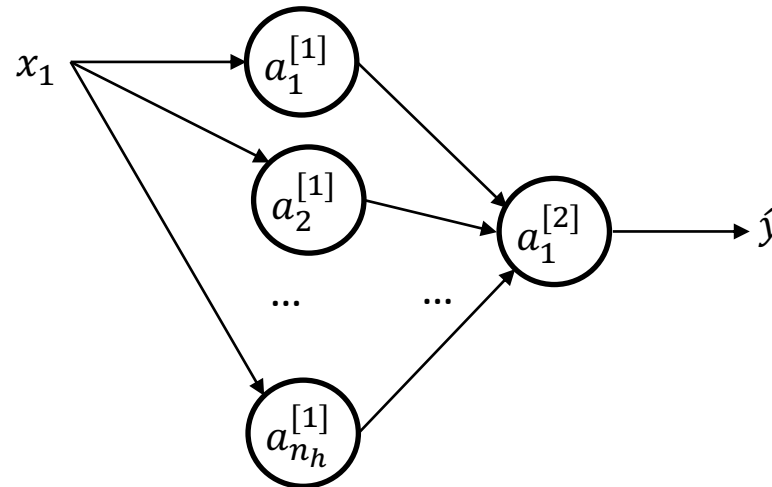
Now, assuming that our vocabulary is sufficient, we will have a numerical representation of each word in our input and output sentences.



NLP Word Representation

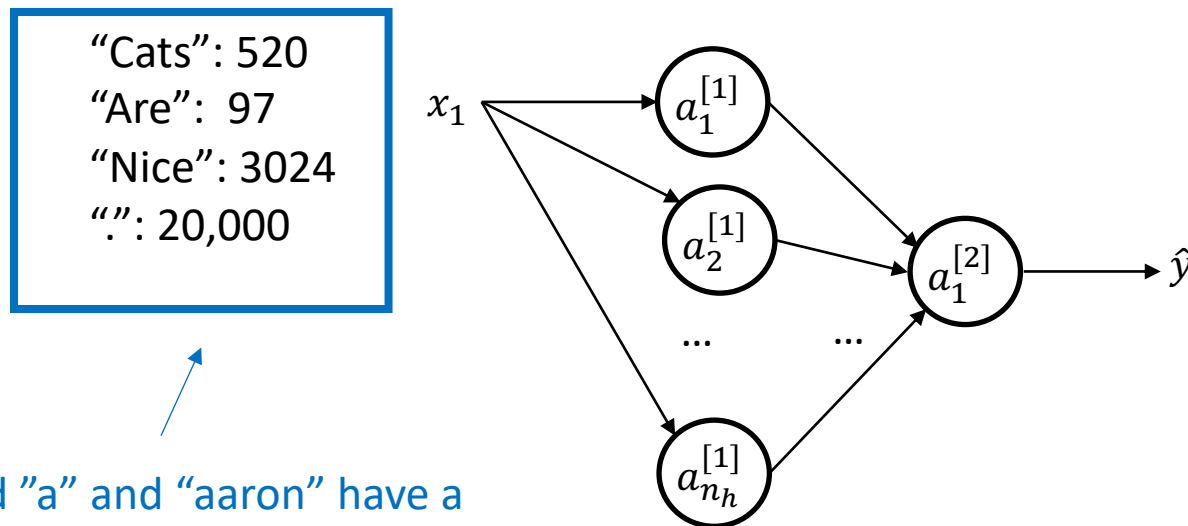
- This could work, but we have made our learning task very hard and added un-intentioned bias, consider:

"Cats": 520
"Are": 97
"Nice": 3024
".": 20,000



NLP Word Representation

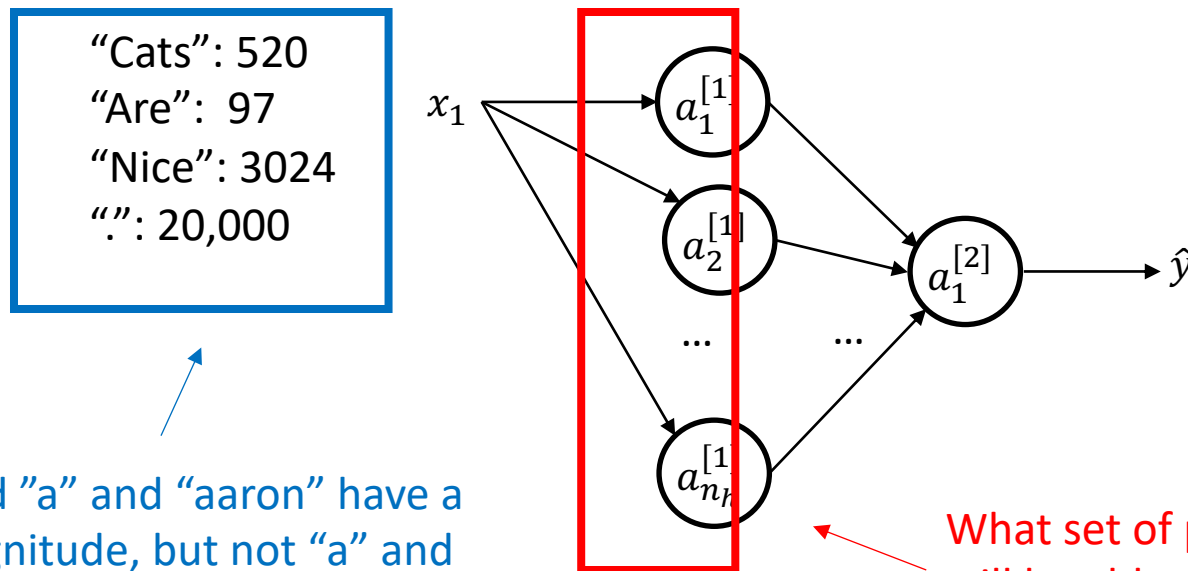
- This could work, but we have made our learning task very hard and added un-intentioned bias, consider:



Why should "a" and "aaron" have a similar magnitude, but not "a" and "erin", or "tom"? Words are biased together based on their position in the alphabet!

NLP Word Representation

- This could work, but we have made our learning task very hard and added un-intentioned bias, consider:



Why should "a" and "aaron" have a similar magnitude, but not "a" and "erin", or "tom"? Words are biased together based on their position in the alphabet!

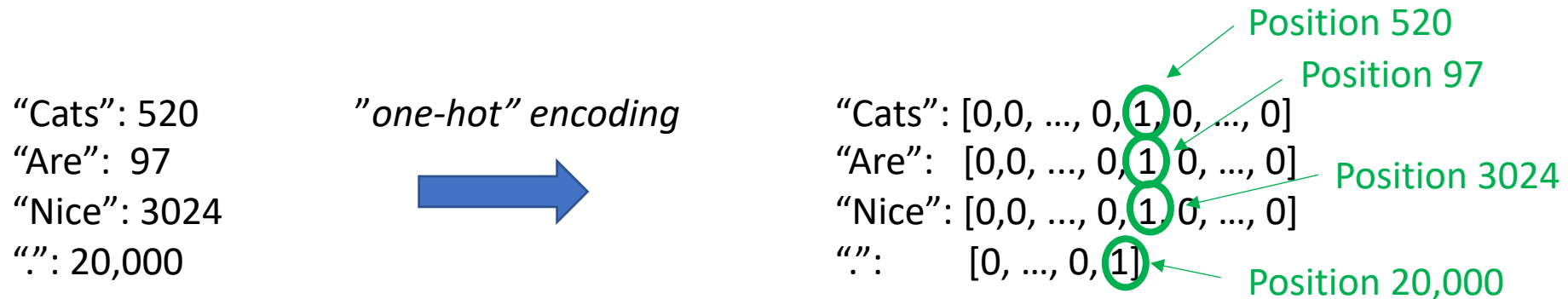
What set of parameters will be able to create a distinct activation for each word?

NLP Word Representation

- We need a more “normalized” representation and less compressed representation.

NLP Word Representation

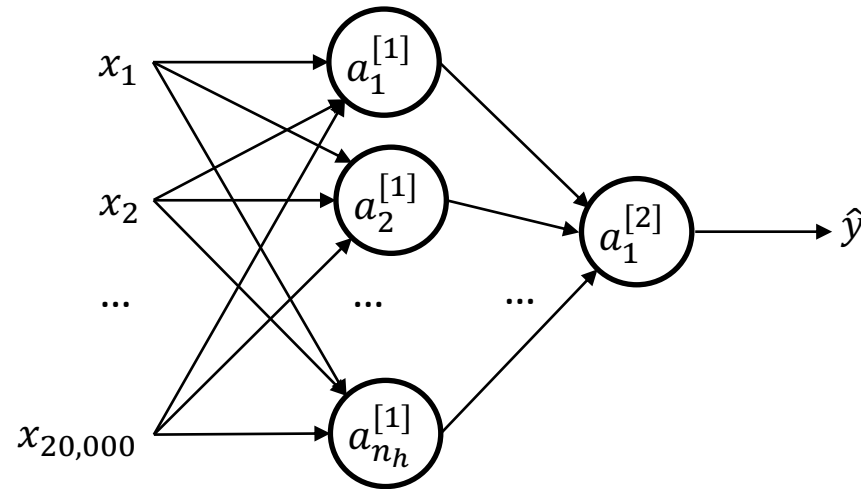
- We need a more “normalized” and less compressed representation.



Each 20,000-element vector simply marks which word it **is** and which word it is **not**.

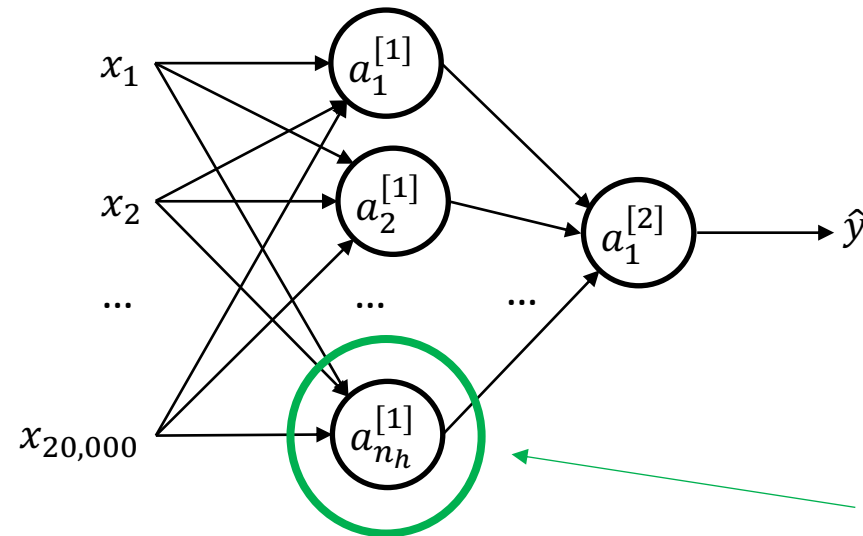
NLP Word Representation

- Now, with one-hot encoding there is no order bias, and it is much easier to imagine and set of parameters that will create distinct activations



NLP Word Representation

- Now, with one-hot encoding there is no order bias, and it is much easier to imagine and set of parameters that will create distinct activations



However, notice that each node in the first hidden layer has 20,000 parameters.

Aside - Input Encodings, “Hand Engineering”

- There is an important take away here for all of ML
- Not everything should be learned! One-hot encoding is an example “hand engineering”, or using our human understanding of the problem to improve the system
- There is a tendency in some ML circles to “look down” at any human intervention in the learning process, however, in my experience there is a huge gain in understanding the problem and not forcing the ML to learn something you already know!

Unknown words

- What happens when you encounter a word you don't know?

Unknown words

- What happens when you encounter a word you don't know?
- There is a simple solution: create one more vector element which we can call Unknown Word ("UKW")
- Now, all unknown words alias to this same "word". You can even allow "UKW" as an output if that makes sense in your system

UKW effectiveness

- But will this work?
- As long as your vocabulary includes all the words that are important for your NLP task there should be no problem mapping some words to UKW
- For example, we can't leave '*basketball*' out of our Raptors translation vocabulary! But we could probably leave out '*pottery*'

But, Can We Make RNNs Learn Effectively?

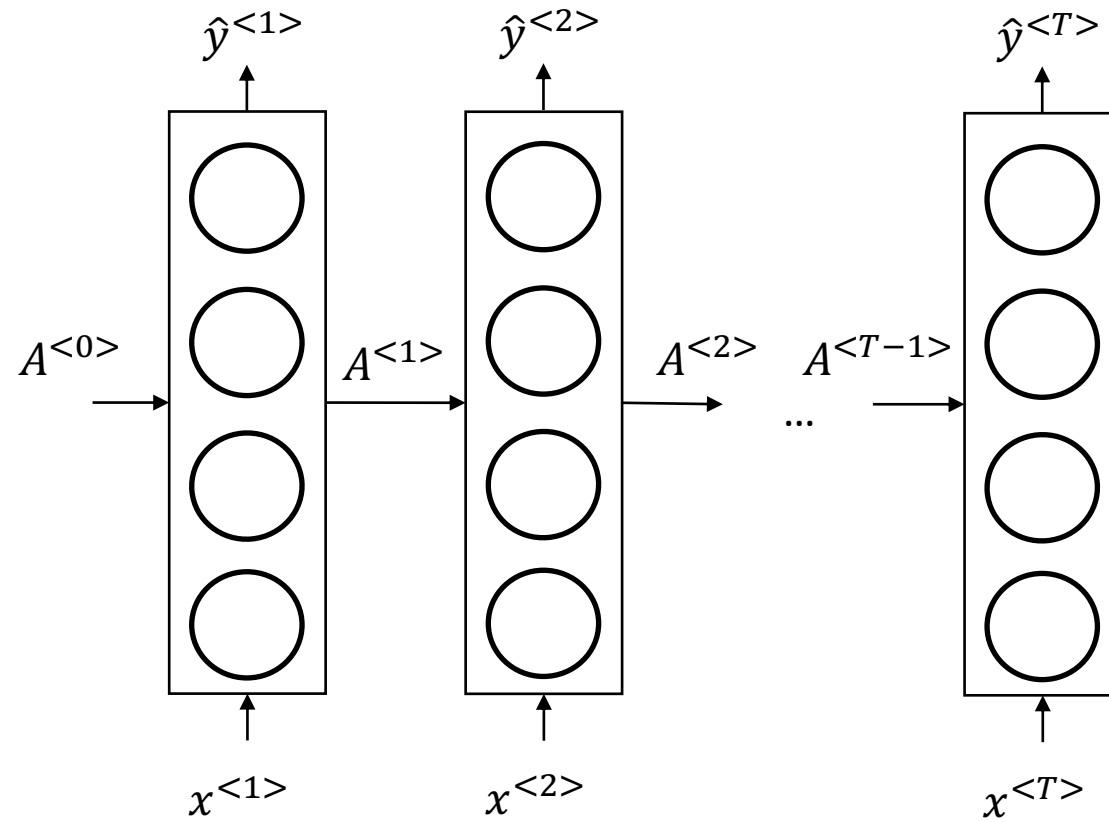
- We now understand the structure, terminology and proper input and output encodings, but to be effective we need to be able to train our RNNs
- The marketing savvy Deep Learning community has labelled this “*Backpropagation Through Time*” ... Which makes it sounds harder than it really is
- The really good news, in fact, is that learning in RNNs, leverages all the understanding and techniques we have already developed!

RNN Loss Function

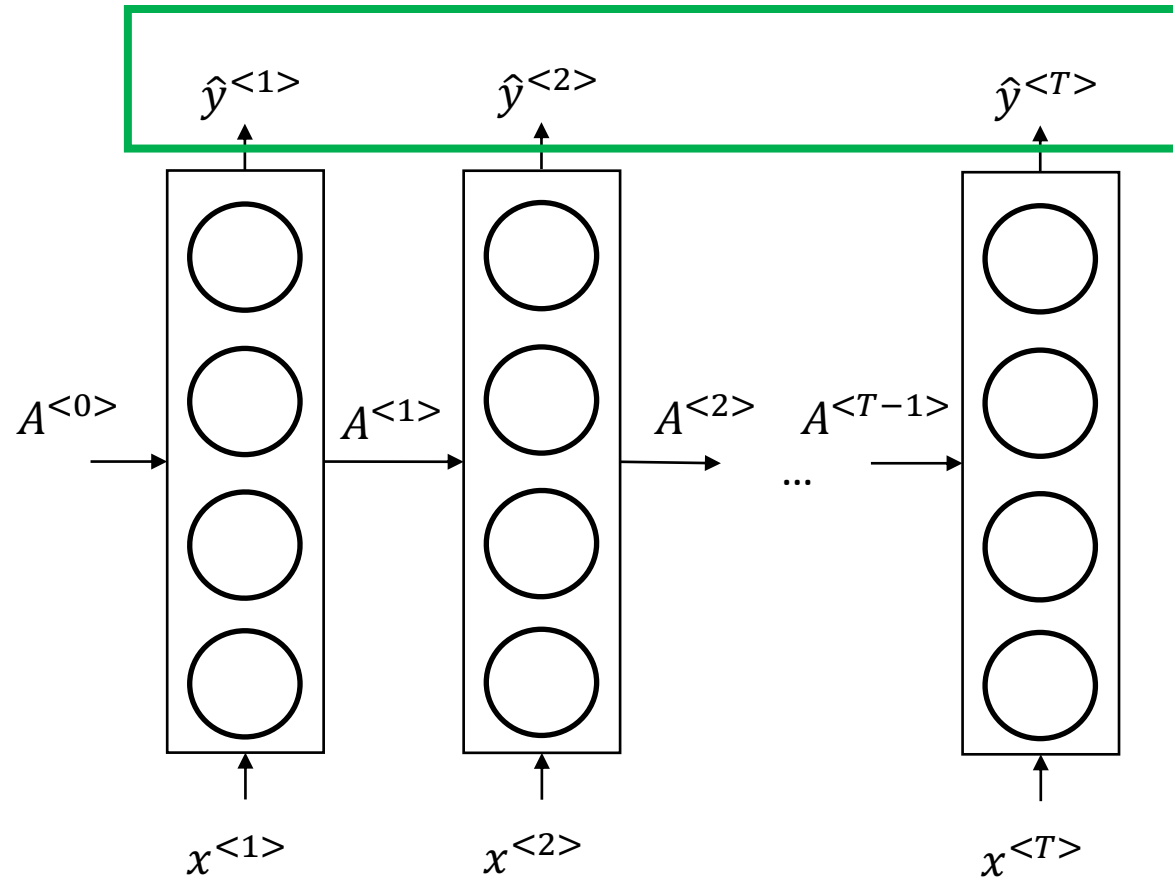
- As we have seen in all of our other ML problems, we need to define a *loss function*, so that we have something to optimize*
- For RNNs we simply need to expand this the idea of a loss function over our entire output sequence
- Let's look at our RNN structure to see how

**In fact whenever you are doing ML you should be asking yourself, what is the loss/cost function, and how does it align to my system goals.*

RNN Loss Function



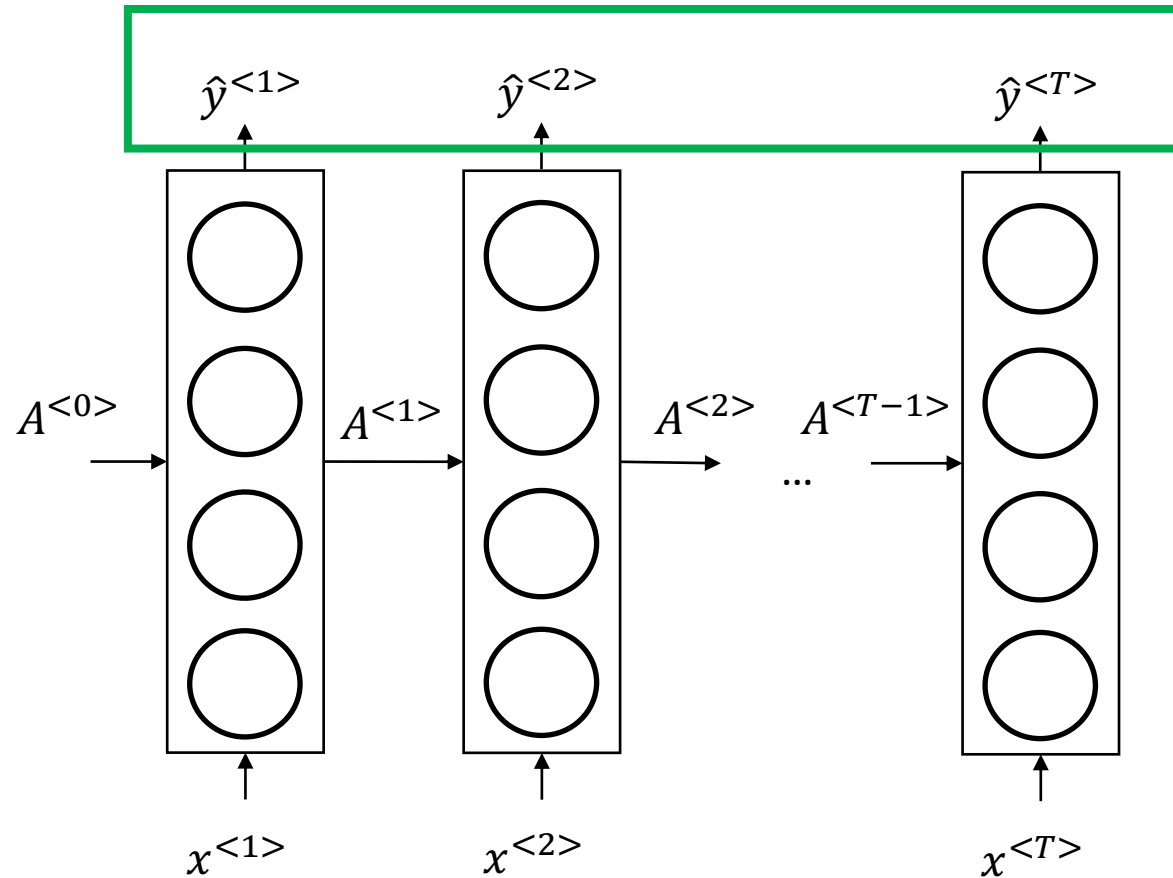
RNN Loss Function



We can define the overall loss to be the sum of the loss of each element of the sequence:

$$L(\hat{y}, y) = \sum_{t=1}^{T_y} L^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

RNN Loss Function



We can define the overall loss to be the sum of the loss of each element of the sequence:

$$L(\hat{y}, y) = \sum_{t=1}^{T_y} L^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

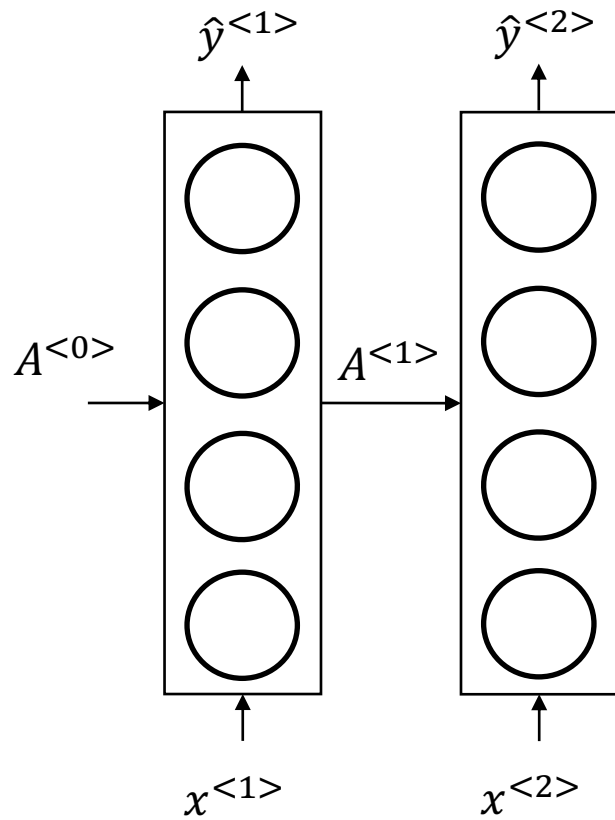
And, given we are using one-hot encodings, we can re-use Cross Entropy Loss for each element:

$$\begin{aligned} L^{<t>}(\hat{y}^{<t>}, y^{<t>}) &= -y^{<t>} \log(\hat{y}^{<t>}) \\ &\quad - (1 - y^{<t>}) \log(1 - \hat{y}^{<t>}) \end{aligned}$$

Implementing Backprop in RNNs

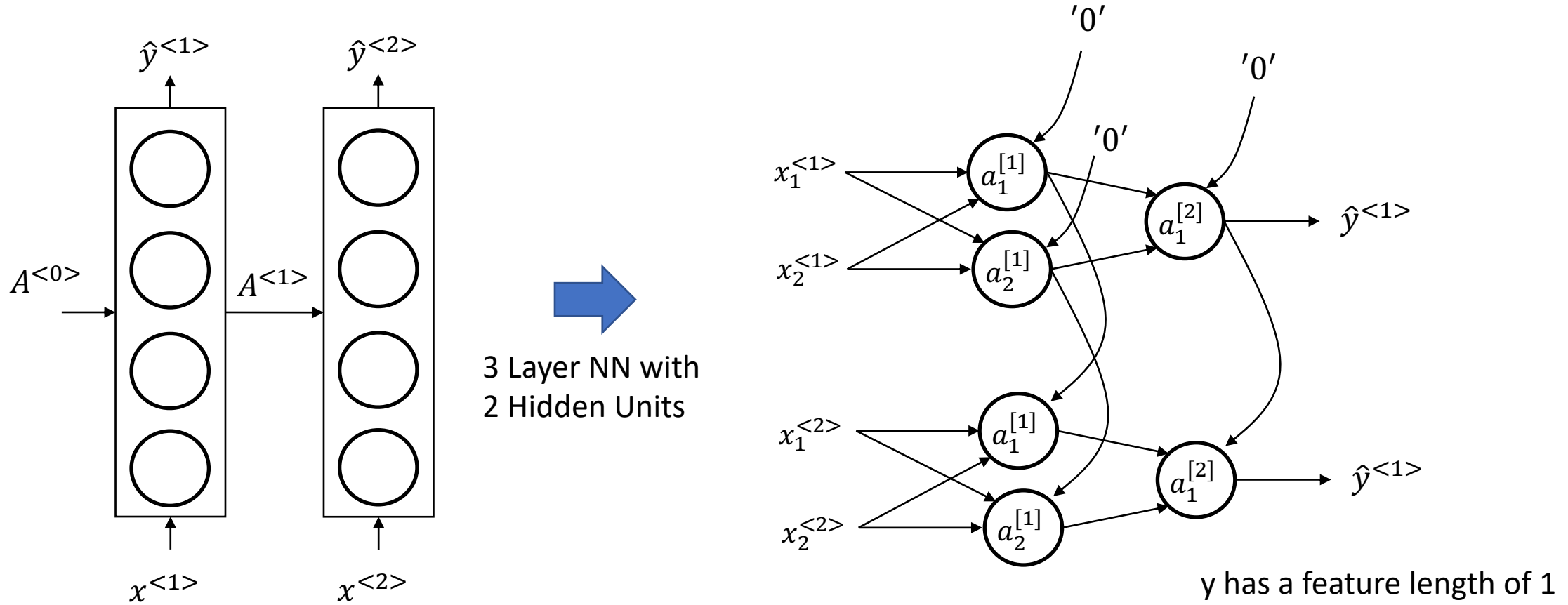
- Here all the hard work you have already put into this course really pays off
- If we can map our RNN structure to a computation graph, then we will be able to apply everything we know to find the gradients and implement backpropagation!
- So, let's take a look...

Computation Graph



Let's imagine we have a $T_y = T_x = 2$

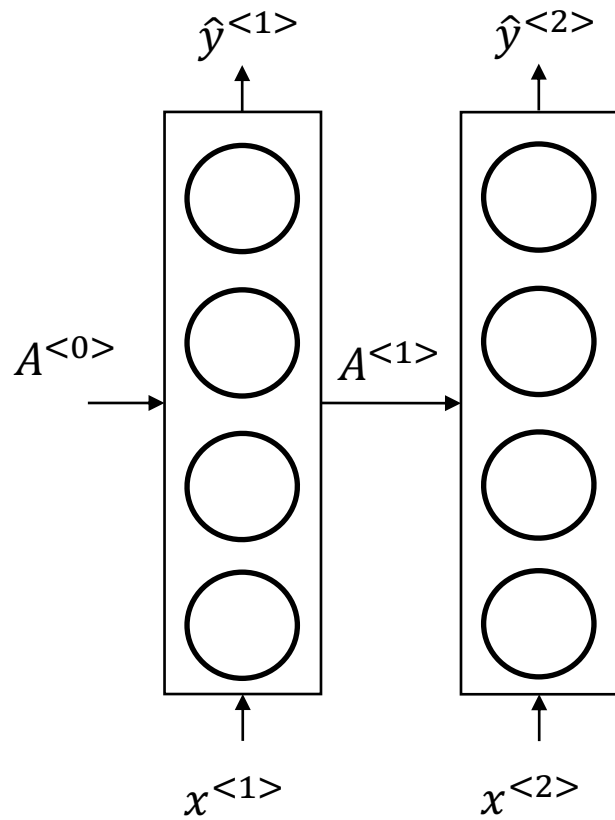
RNN Computation Graph



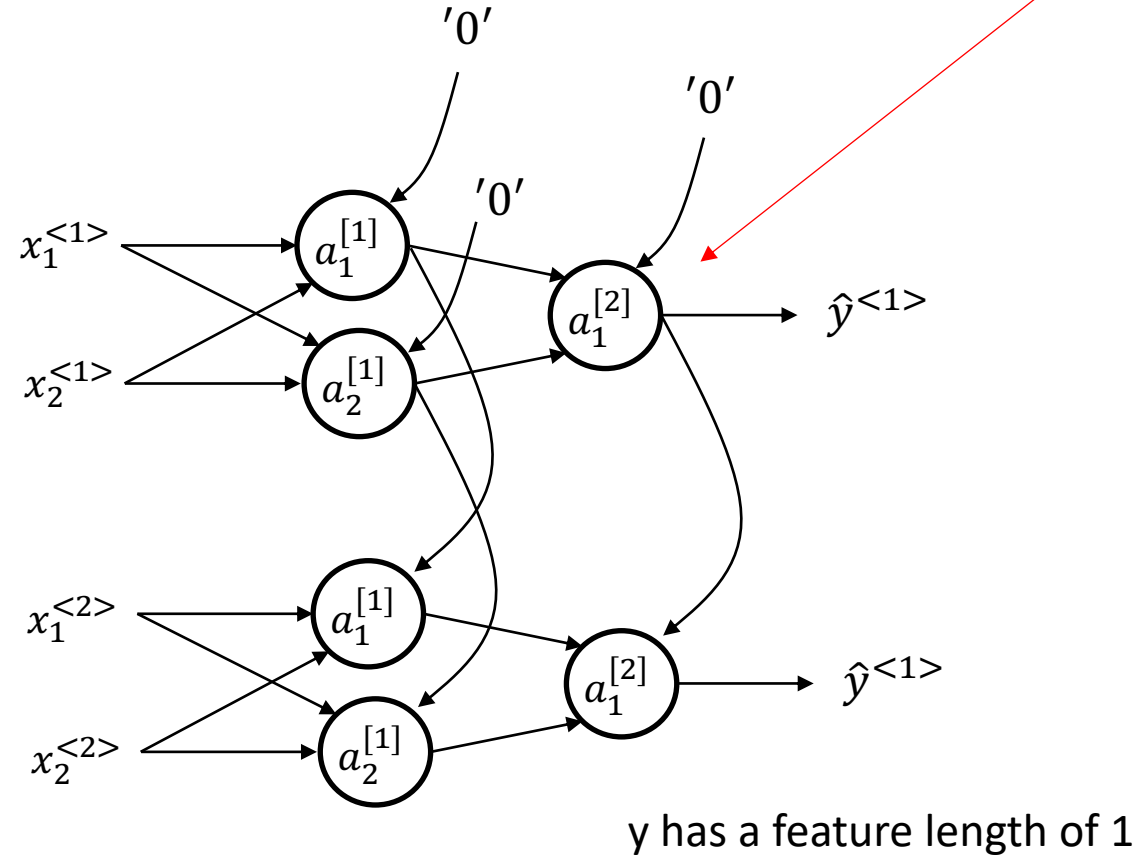
Let's imagine we have a $T_y = T_x = 2$

x has a feature length of 2

RNN Computation Graph



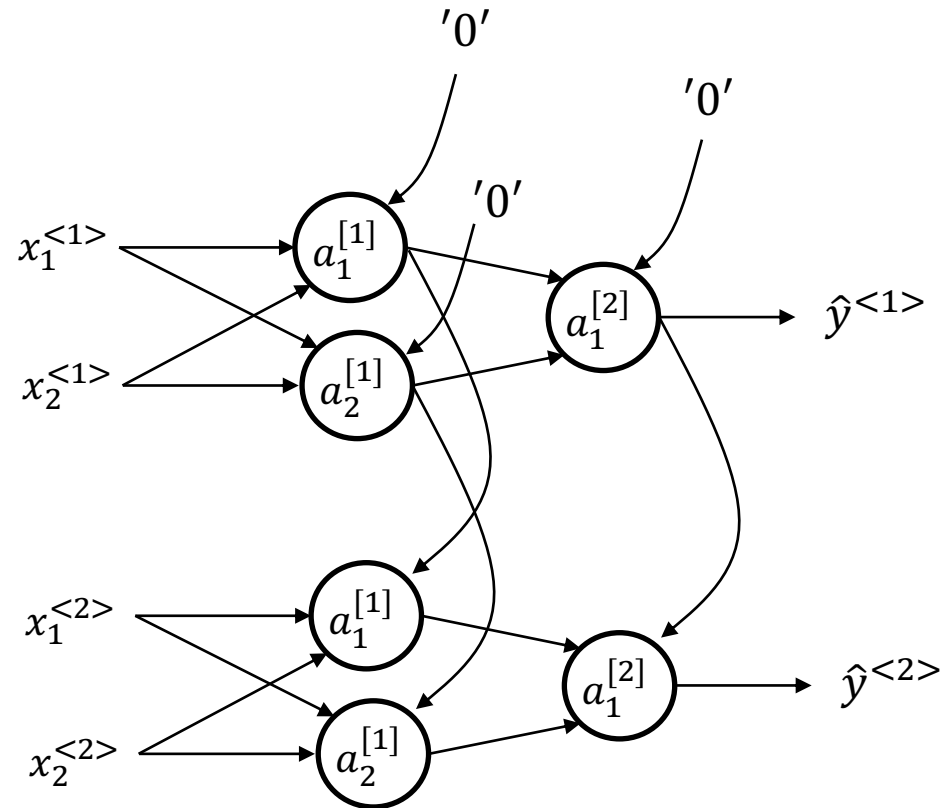
3 Layer NN with
2 Hidden Units



Let's imagine we have a $T_y = T_x = 2$

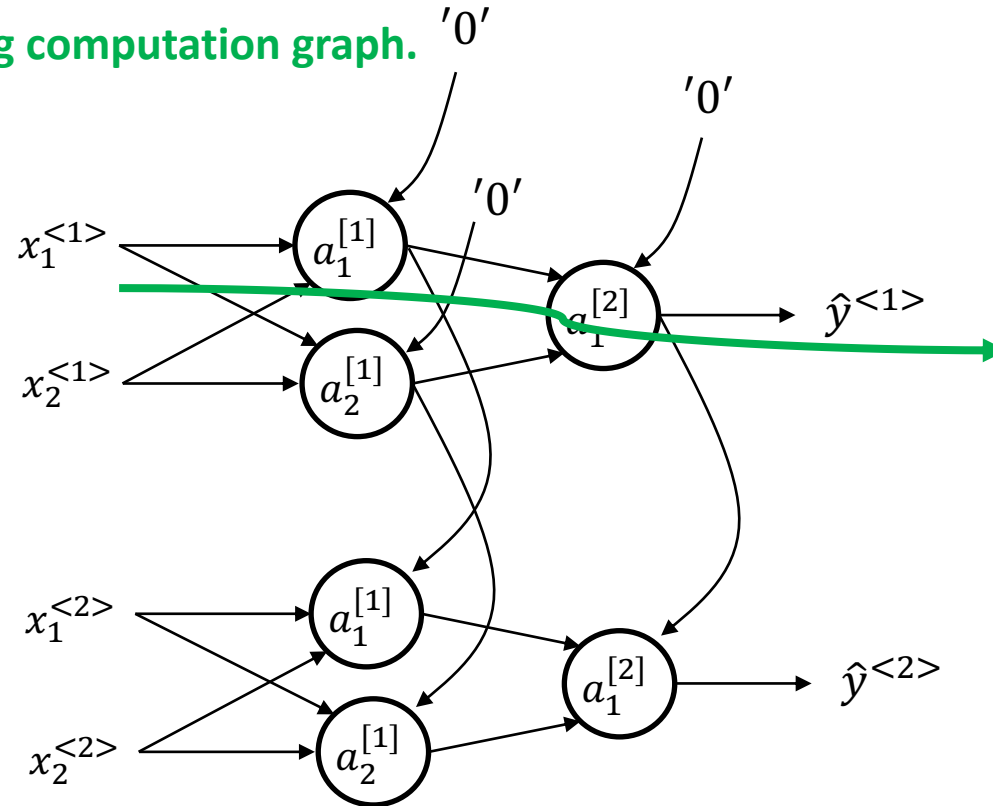
x has a feature length of 2

Backprop on an RNN



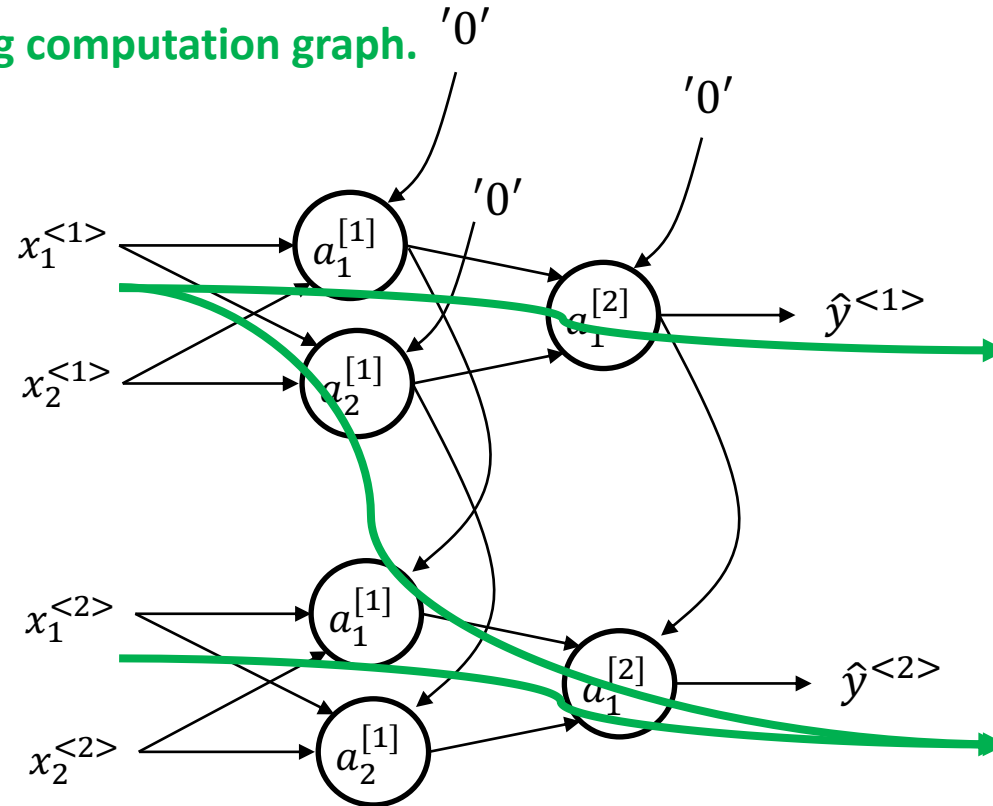
Backprop on an RNN

Step 1: Calculate \hat{y} using computation graph.



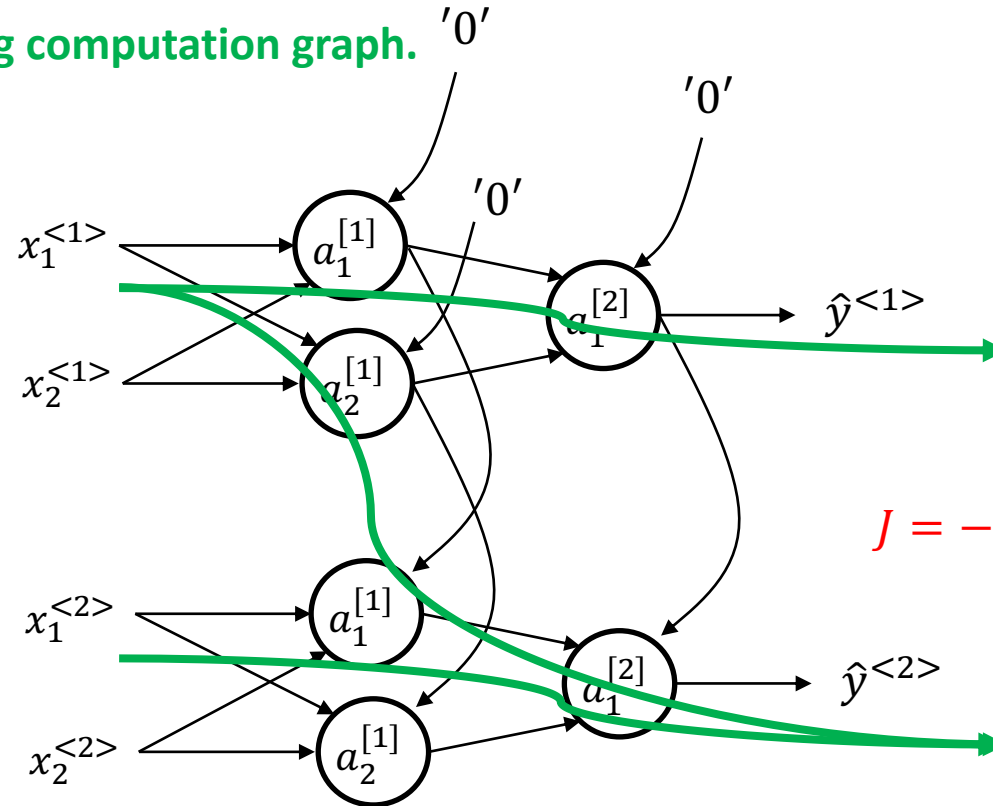
Backprop on an RNN

Step 1: Calculate \hat{y} using computation graph.



Backprop on an RNN

Step 1: Calculate \hat{y} using computation graph.

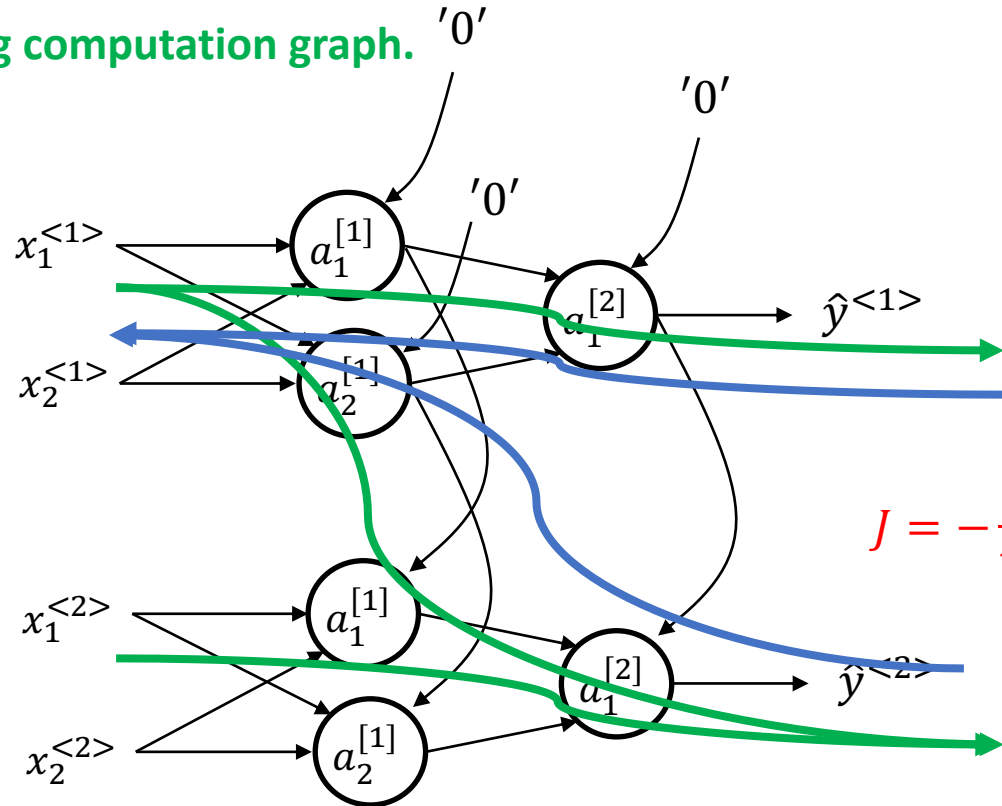


Step 2: Determine the loss.

$$J = -\frac{1}{m} \left(\sum_{i=1}^m y^i \log(\hat{y}^{(i)}) + \sum_{i=1}^m (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right)$$

Backprop on an RNN

Step 1: Calculate \hat{y} using computation graph.



Step 2: Determine the loss.

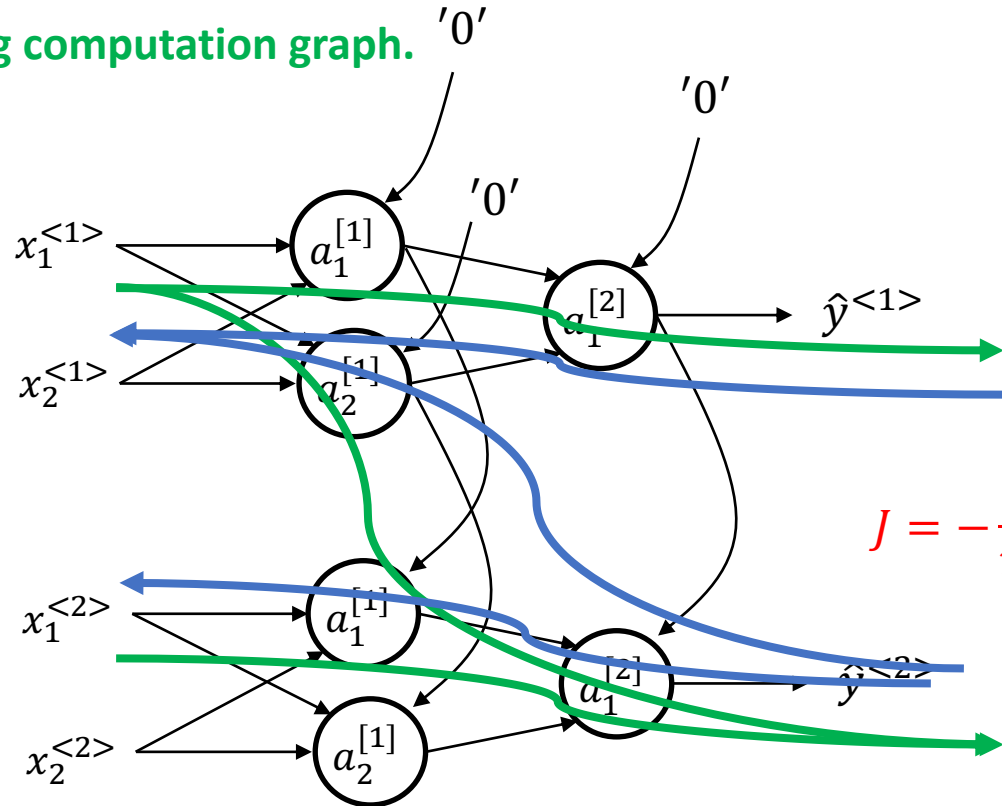
$$J = -\frac{1}{m} \left(\sum_{i=1}^m y^i \log(\hat{y}^{(i)}) + \sum_{i=1}^m (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right)$$

Step 3: Update *each* parameter (Using the partial derivative of cost).

$$w = w - \alpha \frac{\partial J}{\partial w}; b = b - \alpha \frac{\partial J}{\partial b}$$

Backprop on an RNN

Step 1: Calculate \hat{y} using computation graph.



Step 2: Determine the loss.

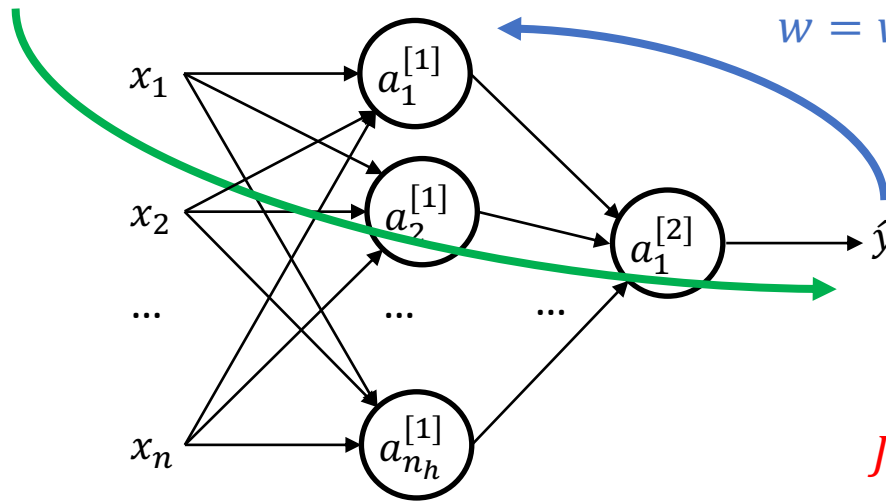
$$J = -\frac{1}{m} \left(\sum_{i=1}^m y^i \log(\hat{y}^{(i)}) + \sum_{i=1}^m (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right)$$

Step 3: Update *each* parameter (Using the partial derivative of cost).

$$w = w - \alpha \frac{\partial J}{\partial w}; b = b - \alpha \frac{\partial J}{\partial b}$$

RECALL: Standard NN Backprop

Step 1: Calculate \hat{y} using computation graph.



Step 3: Update *each* parameter (Using the partial derivative of cost).

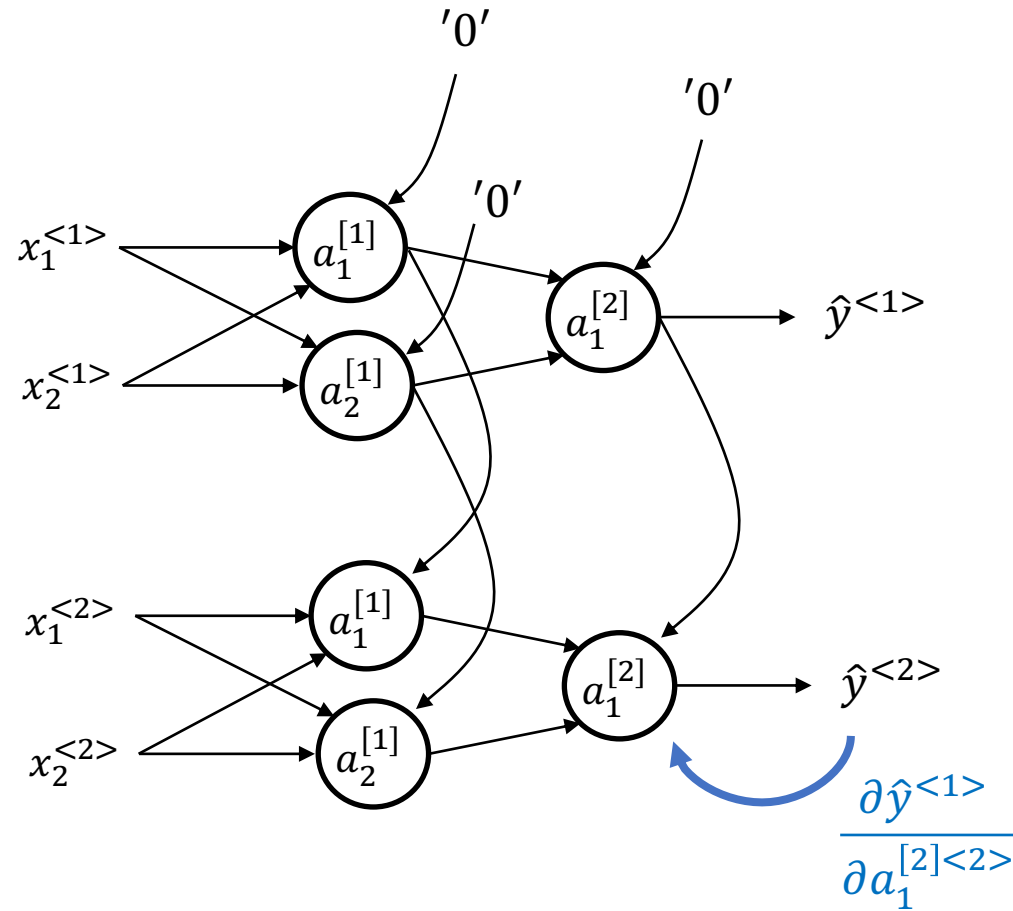
$$w = w - \alpha \frac{\partial J}{\partial w}; b = b - \alpha \frac{\partial J}{\partial b}$$

Step 2: Determine the loss.

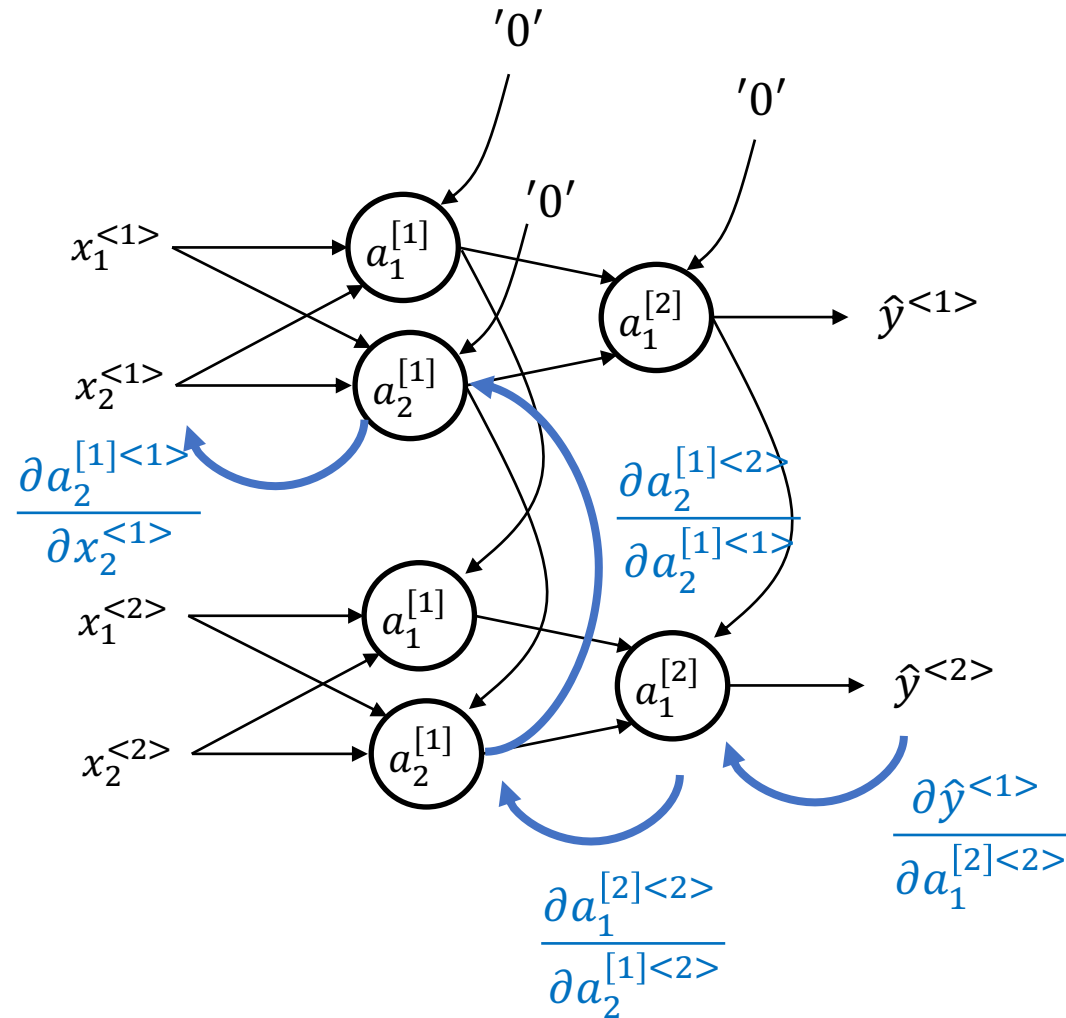
$$J = -\frac{1}{m} \left(\sum_{i=1}^m y^i \log(\hat{y}^{(i)}) + \sum_{i=1}^m (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}) \right)$$

Step 4: Repeat until $J < \text{target}$.

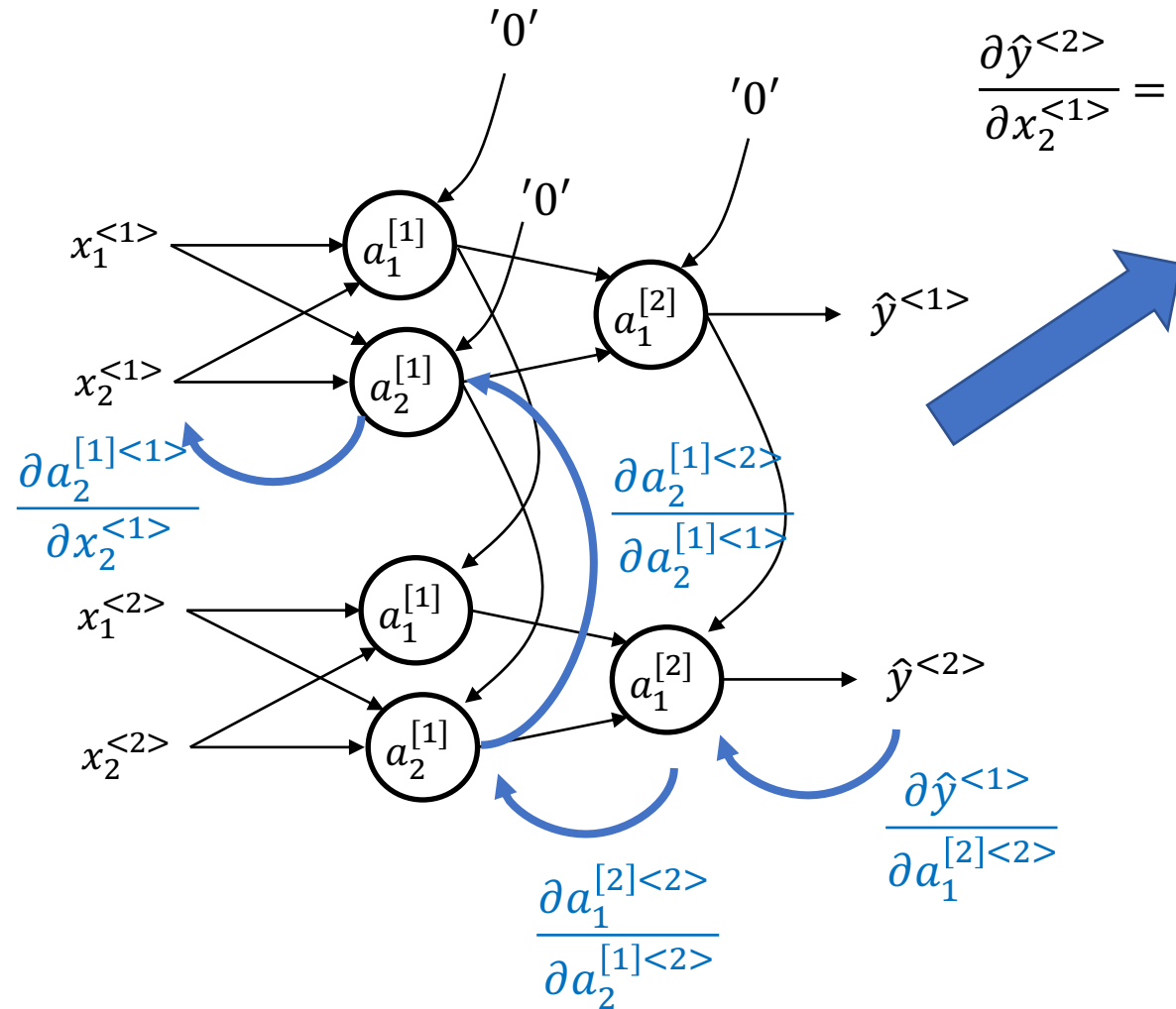
Partial Derivatives on a RNN



Partial Derivatives on an RNN

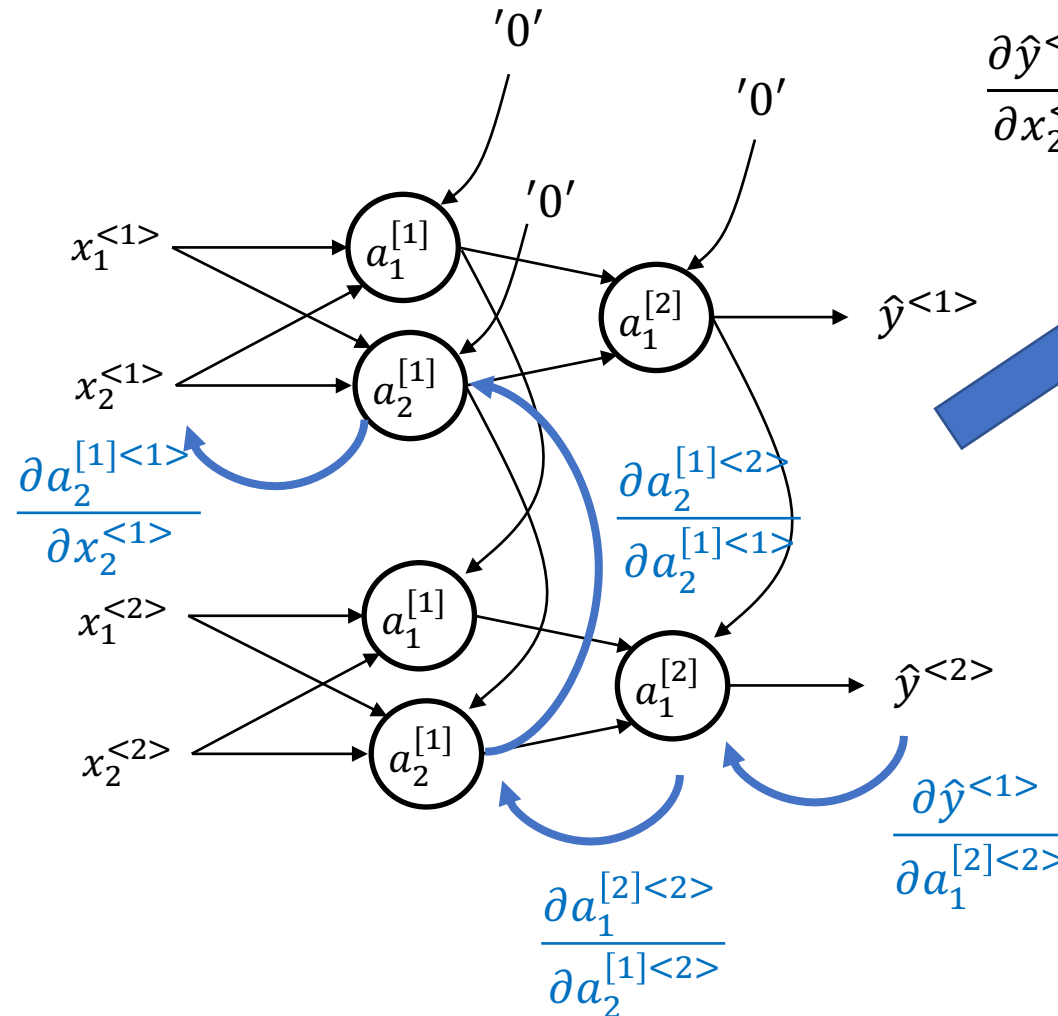


Partial Derivatives on an RNN



$$\frac{\partial \hat{y}^{<2>}}{\partial x_2^{<1>}} = \frac{\partial \hat{y}^{<2>}}{\partial a_1^{[2]} <2>} \cdot \frac{\partial a_1^{[2]} <2>}{\partial a_2^{[1]} <2>} \cdot \frac{\partial a_2^{[1]} <2>}{\partial a_2^{[1]} <1>} \cdot \frac{\partial a_2^{[1]} <1>}{\partial x_2^{<1>}}$$

Partial Derivatives on an RNN



$$\frac{\partial \hat{y}^{<2>}}{\partial x_2^{<1>}} = \frac{\partial \hat{y}^{<2>}}{\partial a_1^{[2]<2>}} \cdot \frac{\partial a_1^{[2]<2>}}{\partial a_2^{[1]<2>}} \cdot \frac{\partial a_2^{[1]<2>}}{\partial a_2^{[1]<1>}} \cdot \frac{\partial a_2^{[1]<1>}}{\partial x_2^{<1>}}$$

This is a really exciting results because we can re-use essentially everything we know about implementing backprop on NNs!

Pulling it Altogether

- We won't derive all the equations here, but it should be apparent that other than having to be very careful about naming and keeping track of our internal variables, this is a straight-forward exercise
- It is important to note that, similar to the case with filter parameters in CNNs, the RNN parameters are being updated with the average gradients on each sample (*i.e.* If sequence length is 3, then a given sample will have 3 updates for each parameter)

Aside – Why did it workout?

- It is worth asking thinking about why backprop can be applied so directly to this new RNN structure

Aside – Why did it workout?

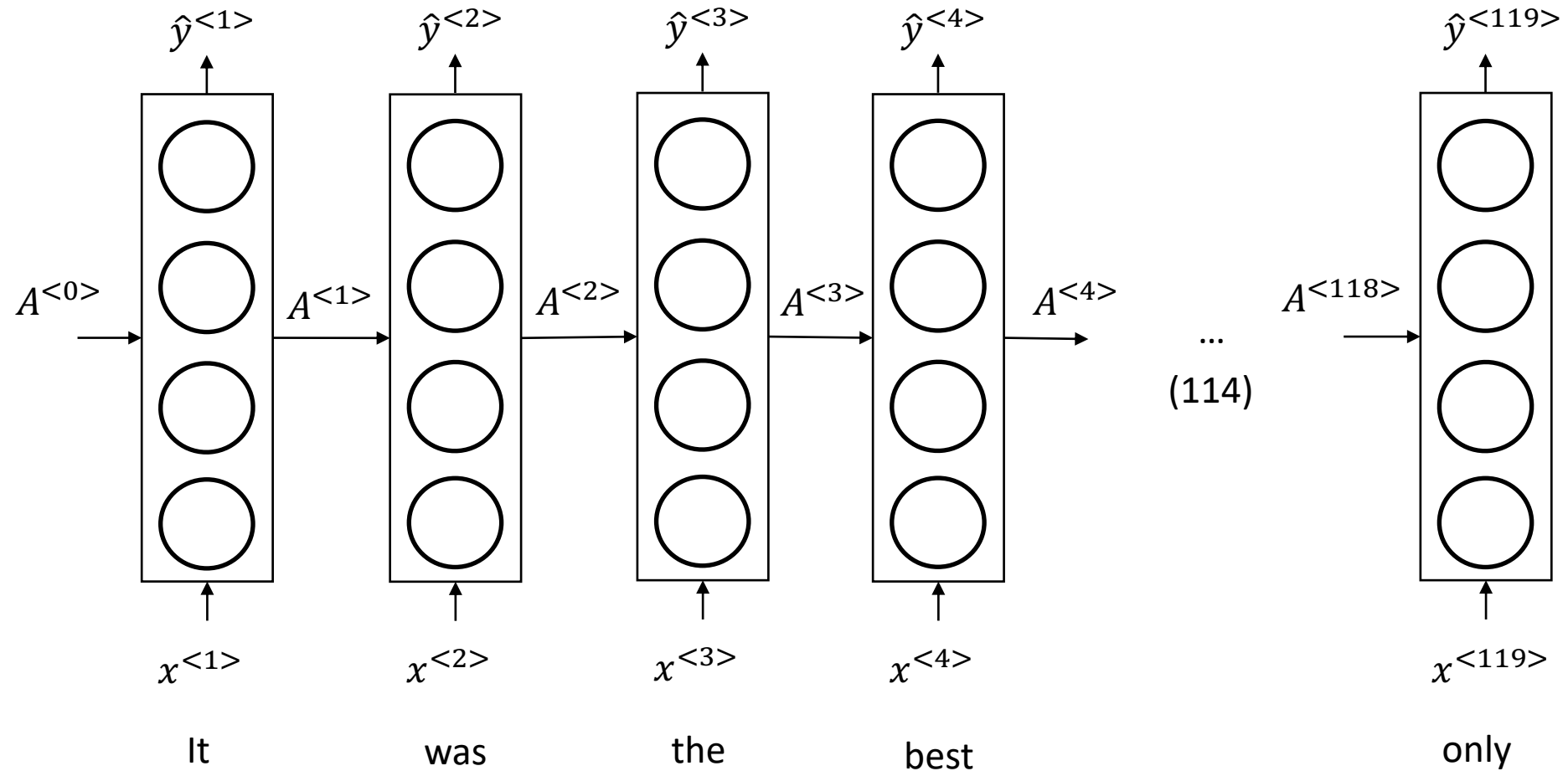
- It is worth asking thinking about why backprop can be applied so directly to this new RNN structure
- The answer is because everything we derived about backprop was focused on a generalized computation graph* (not on a specific architecture like a Neural Network, CNN, etc.)
- This means that we can expect that we can create new ML architectures and still use backprop to search for parameters that minimize a target cost function!

**We have, however, restricted ourselves to acyclic computation graph (i.e. we can't handle feedback loops with what we have done so far)*

Vanishing Gradients In RNNs

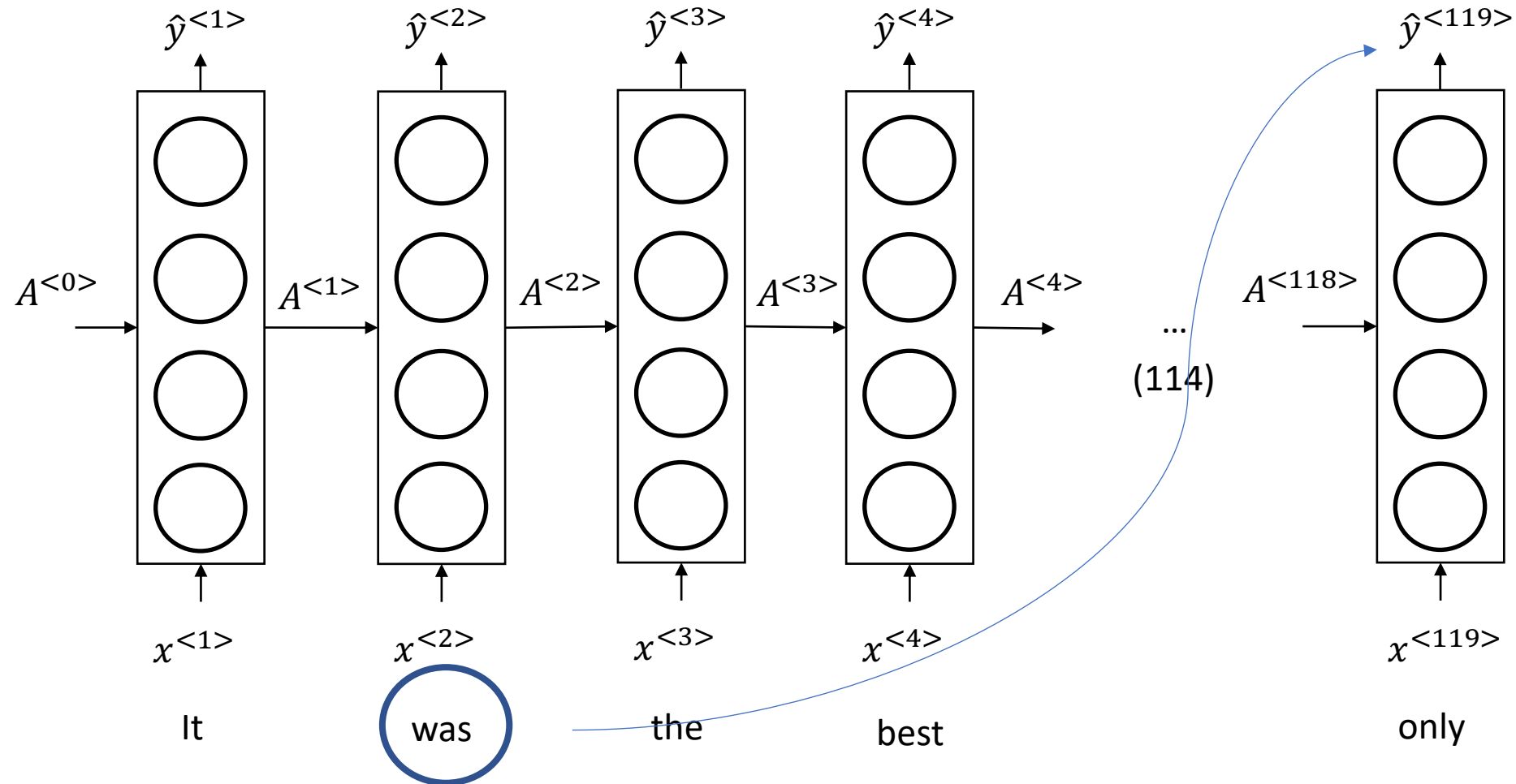
- Similar to deep NNs, RNNs have the potential to experience the problem of vanishing gradients
- As sequences get long it can be difficult to enable earlier elements to correctly influence later outputs.

Vanishing Gradients In RNNs



Vanishing Gradients In RNNs

For example, the tense of the second word in the sentence maybe critical to translate the last word!



Vanishing Gradients In RNNs

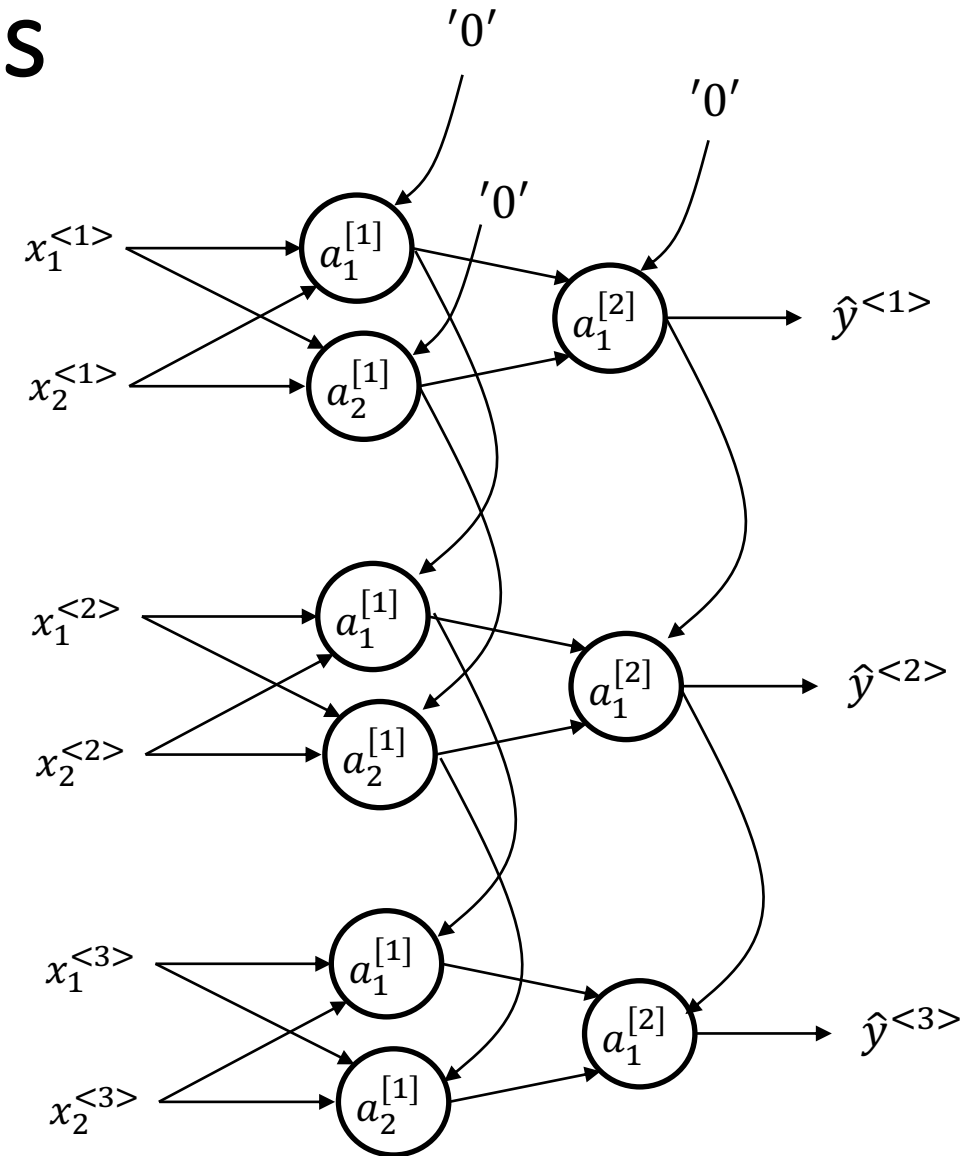
- One way to visualize the vanishing gradient problem in RNNs is to imagine the influence of early elements of the sequences are “washed out” by the repeat multiply accumulates in each iteration
- Recall the activation function for our an RNN unit, for example:

$$a_1^{[1]<t>} = g(w_{ax}X_1 + w_{aa}a_1^{[1]<t-1>} + b)$$

- How can we maintain the influence of a value across many sequences?

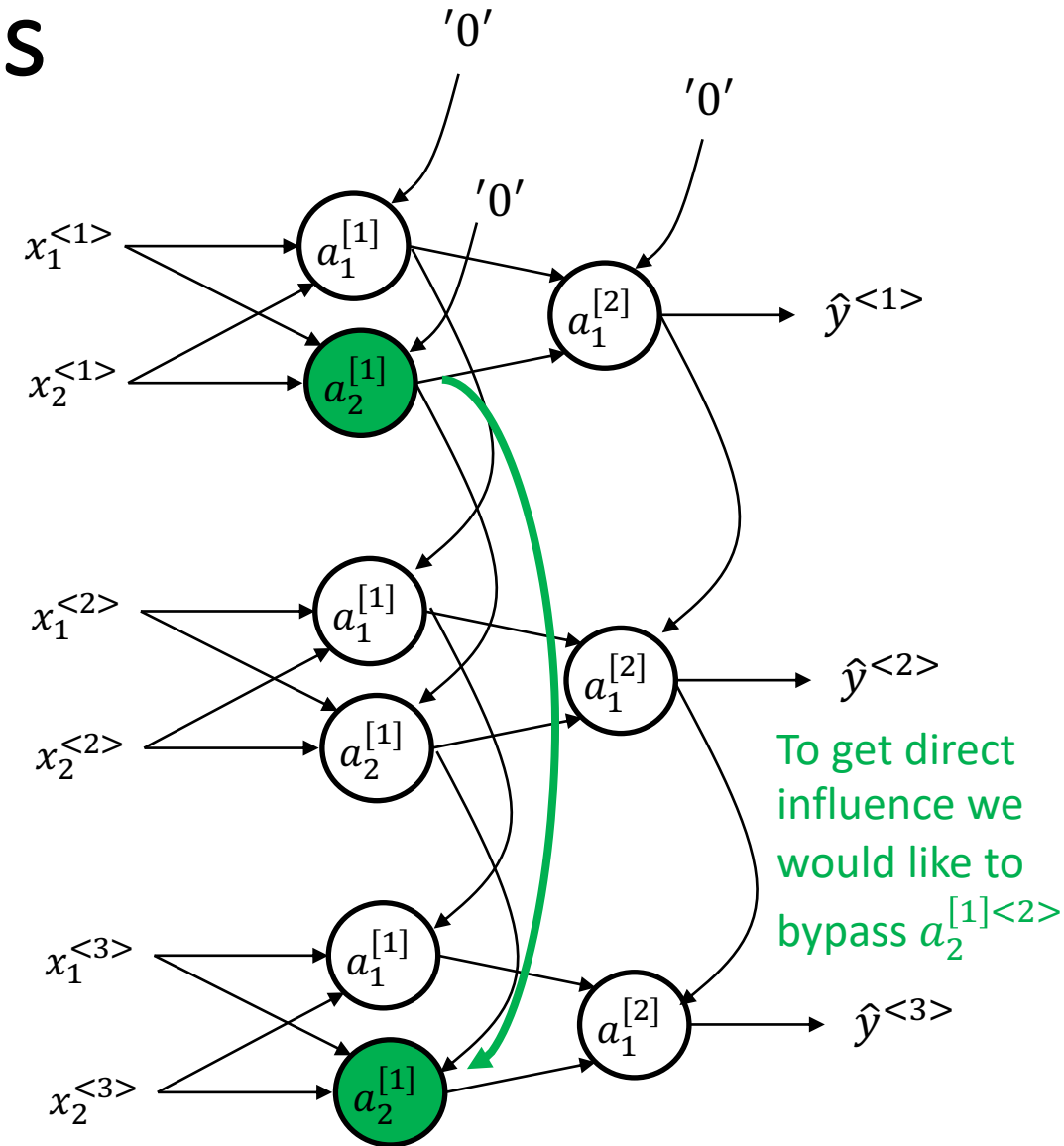
Vanishing Gradients In RNNs

- Let's slightly increase the size of our previous computation graph representation to see if we can get some intuition



Vanishing Gradients In RNNs

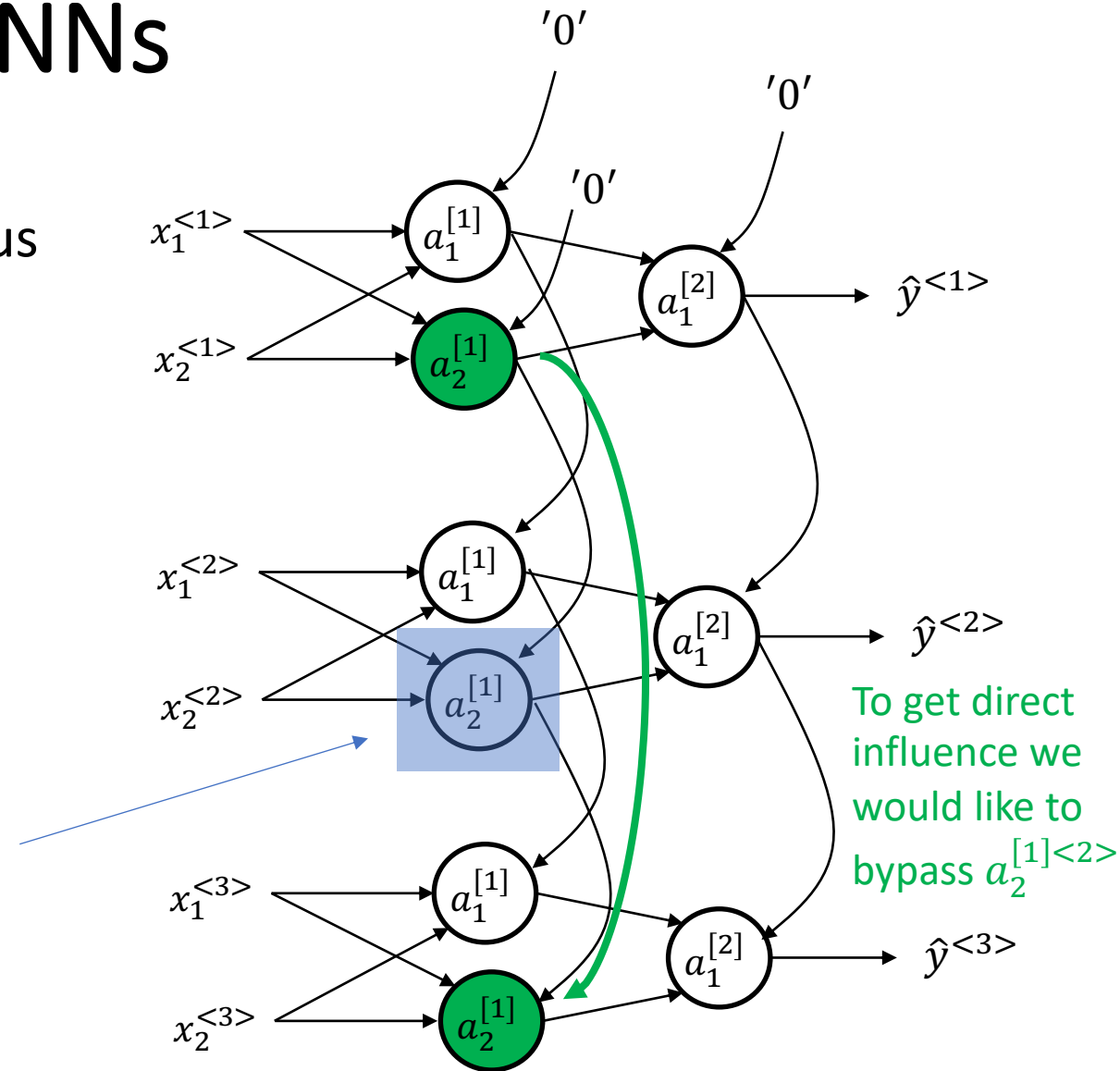
- Let's slightly increase the size of our previous computation graph representation to see if we can get some intuition



Vanishing Gradients In RNNs

- Let's slightly increase the size of our previous computation graph representation to see if we can get some intuition

Recall that we are repeatedly applying the *same* RNN activation units, so to “bypass” the current activation we simply hold the previous value. (i.e. we add a memory to our RNN)



Simplified Gated Recurrent Unit (GRU)

- We can do this by creating a “gating function” for our activation:

$$\Gamma_{\mu} = \sigma(w_{\mu x}X_1 + w_{\mu a}a_1^{[1]<t-1>} + b_{\mu})$$

- With this function we will get a value between 0 and 1 based on *learned* parameters and our standard RNN unit inputs
- We can then use this function to decide if we should keep the previous activation or update it:

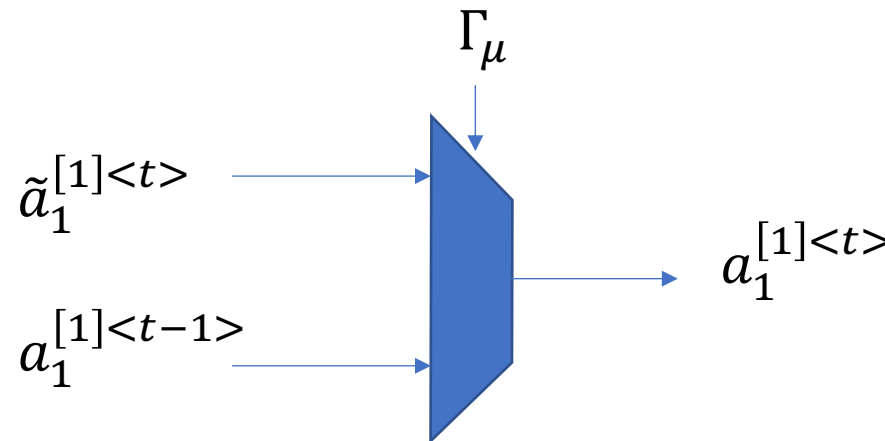
$$a_1^{[1]<t>} = \Gamma_{\mu} * \tilde{a}_1^{[1]<t>} + (1 - \Gamma_{\mu}) * a_1^{[1]<t-1>}$$

Simplified Gated Recurrent Unit (GRU)

- Where our standard activation calculation now becomes a “candidate”

$$\tilde{a}_1^{[1]<t>} = g(w_{ax}X_1 + w_{aa}a_1^{[1]<t-1>} + b)$$

- Note, this is very similar to creating a function that controls a mux:



However this is NOT binary, instead it use the sigmoid to make sure it is a continuous (i.e. differentiable) function.

Long Short Term Memory (LSTM)

- In practice, most RNNs use the somewhat more general LSTM approach to manage the vanishing gradient problem in RNNs
- To use this approach we have to create a new variable, c , which can be viewed as a more explicit expression of the internal memory idea
- LSTMs also split the gating concept into three parts: update, forget and output

Long Short Term Memory (LSTM)

- We can imagine we would create three, independent, learned functions:

$$\Gamma_{\mu} = \sigma(w_{\mu x}X_1 + w_{\mu a}a_1^{[1]<t-1>} + b_{\mu}) \quad \text{Update}$$

$$\Gamma_f = \sigma(w_{fx}X_1 + w_{fa}a_1^{[1]<t-1>} + b_f) \quad \text{Forget}$$

$$\Gamma_o = \sigma(w_{ox}X_1 + w_{oa}a_1^{[1]<t-1>} + b_o) \quad \text{Output}$$

- Now we can use these function to manage our internal memory and output values

Long Short Term Memory (LSTM)

- We can create the candidate memory:

$$\tilde{c}_1^{[1]<t>} = g(w_{ax}X_1 + w_{aa}a_1^{[1]<t-1>} + b)$$

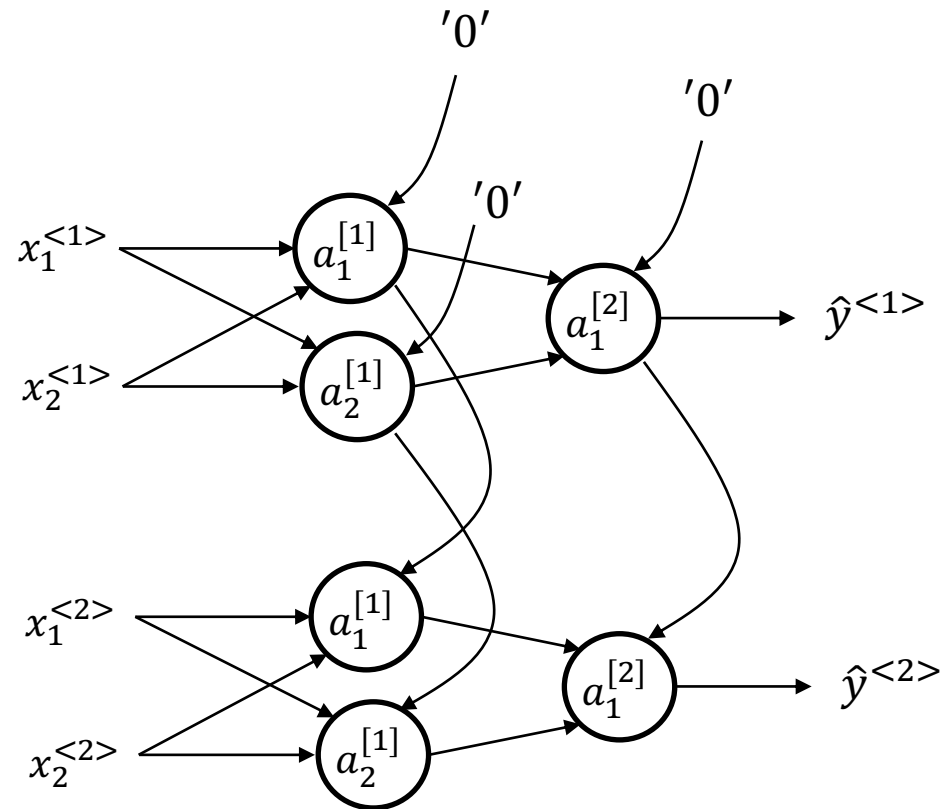
- And then update our internal memory with the idea of both updating and forgetting:

$$c_1^{[1]<t>} = \Gamma_{\mu} * \tilde{c}_1^{[1]<t>} + \Gamma_f * c_1^{[1]<t-1>}$$

- And now we can create our output separate from our internal memory:

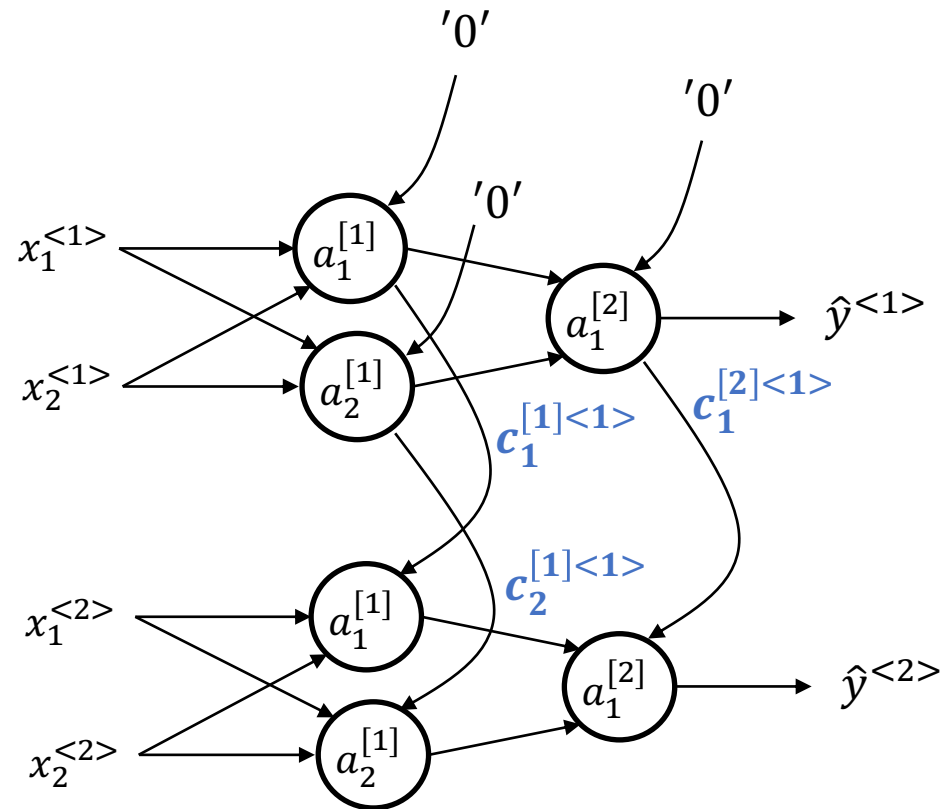
$$a_1^{[1]<t>} = \Gamma_o * \tanh(c_1^{[1]<t>})$$

Long Short Term Memory (LSTM)



- Now we can re-draw our RNN picture, reflecting the explicit separation of the activation, a and the memory, c

Long Short Term Memory (LSTM)



- Now we can re-draw our RNN picture, reflecting the explicit separation of the activation, a and the memory, c

Intuition as to why this is Important

- The GRU and the more general LSTM improvements to the basic RNN architecture have proven to be extremely important to achieving success with RNNs, especially in NLP applications
- The reason has to do with the structure of many sequential data sets, where a key element is critical for a period of time (or across a set of sequences), and then no longer relevant

“The dog, who lived in England and worked hard, was highly prized.”

Pulling Everything Together

- We have now shown that we can use backprop to train RNNs very similarly to how we trained NNs and CNNs
- We have demonstrated how important it is to encode the input data correctly
- We have shown how GRUs and LSTMs control the vanishing gradient problem that can be critical to NLP success
- Next lecture we will focus on specific NLP applications and refinements