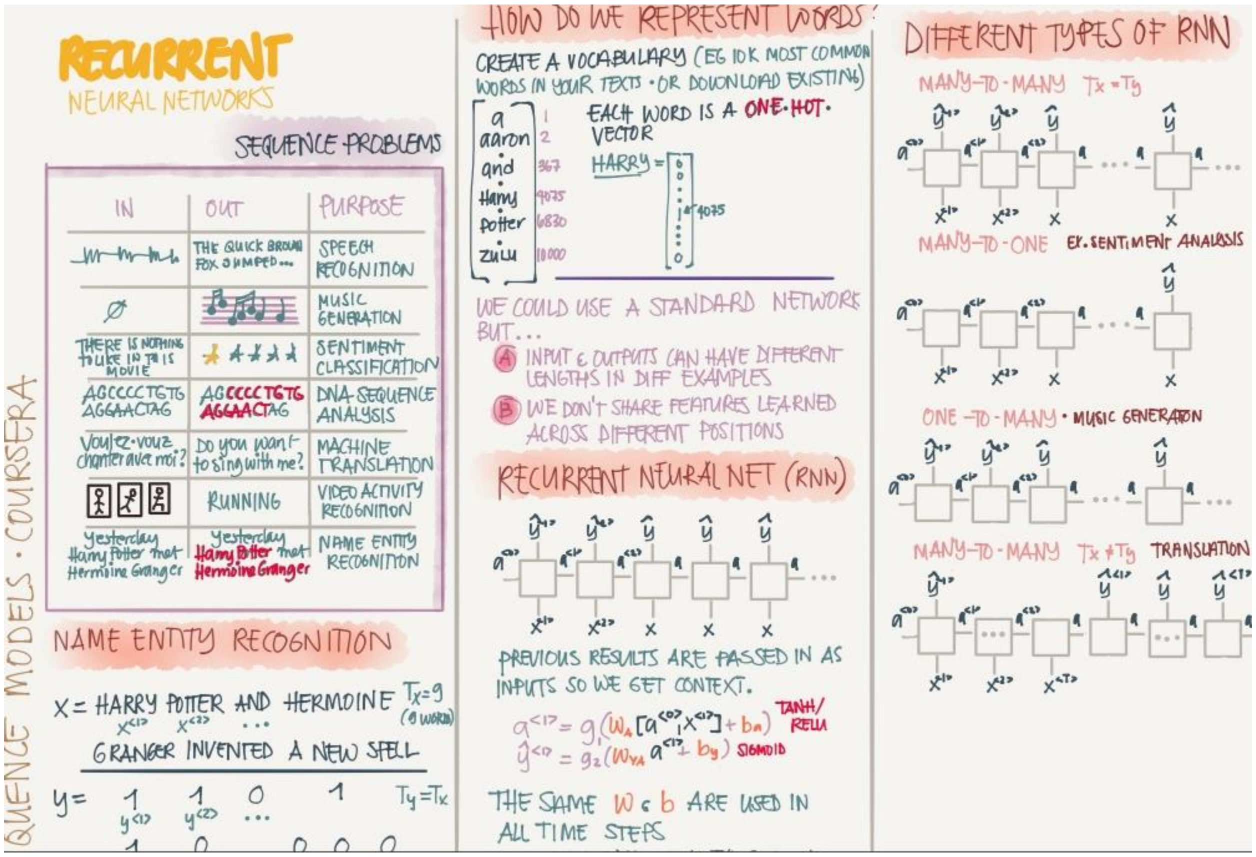
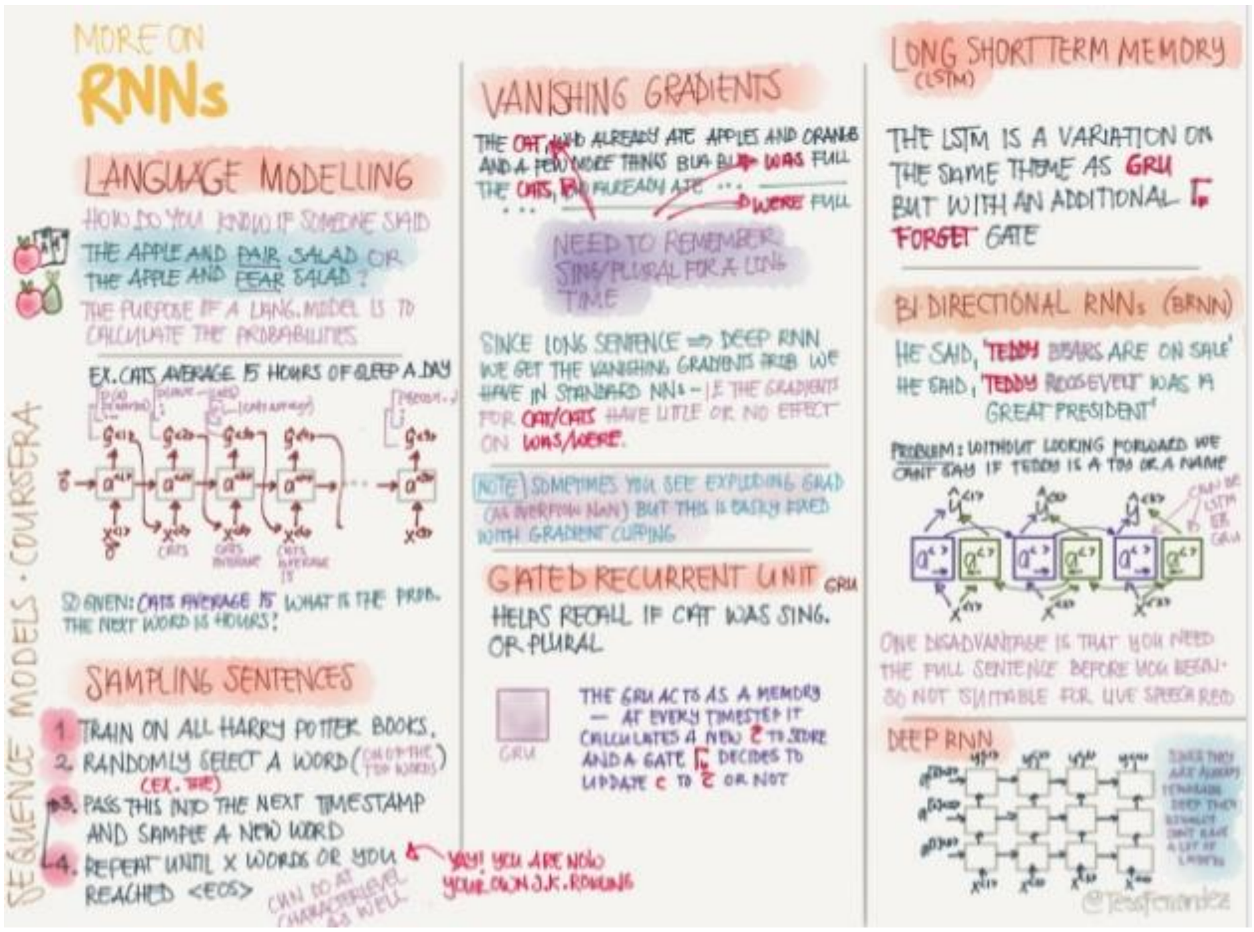
Recurrent Neural Networks



RECURRENT
NEURAL NETWORKS
SEQUENCE MODELS · COURSERA
SEQUENCE PROBLEMS
IN | OUT | PURPOSE
THE QUICK BROWN FOX JUMPED... | SPEECH RECOGNITION
∅ | MUSIC GENERATION
THERE IS NOTHING TO LIKE IN THIS MOVIE | SENTIMENT CLASSIFICATION
AGCCCCTGTG AGGAACTAG | AGCCCCTGTG AGGAACTAG | DNA SEQUENCE ANALYSIS
Voulez-vous chanter avec moi? | Do you want to sing with me? | MACHINE TRANSLATION
RUNNING | VIDEO ACTIVITY RECOGNITION
Yesterday Harry Potter met Hermione Granger | Yesterday Harry Potter met Hermione Granger | NAME ENTITY RECOGNITION
NAME ENTITY RECOGNITION
X = HARRY POTTER AND HERMOINE GRANGER INVENTED A NEW SPELL
$x^{<1>}$ $x^{<2>}$ ...
$T_x=9$ (9 words)
y = 1 1 0 ... 1
$y^{<1>}$ $y^{<2>}$
1 0 0 0 0
$T_y=T_x$
HOW DO WE REPRESENT WORDS?
CREATE A VOCABULARY (EG 10K MOST COMMON WORDS IN YOUR TEXTS · OR DOWNLOAD EXISTING)
a 1
aaron 2
and 367
Harry 4075
potter 6830
zulu 10000
EACH WORD IS A ONE-HOT-VECTOR
HARRY = [0 0 ... 1 ... 0] 4075
WE COULD USE A STANDARD NETWORK BUT...
A INPUT & OUTPUTS CAN HAVE DIFFERENT LENGTHS IN DIFF EXAMPLES
B WE DON'T SHARE FEATURES LEARNED ACROSS DIFFERENT POSITIONS
RECURRENT NEURAL NET (RNN)
PREVIOUS RESULTS ARE PASSED IN AS INPUTS SO WE GET CONTEXT.
$a^{<1>} = g_1(W_a[a^{<0>}, x^{<1>}] + b_a)$ TANH/RELU
$\hat{y}^{<1>} = g_2(W_{ya} a^{<1>} + b_y)$ SIGMOID
THE SAME W & b ARE USED IN ALL TIME STEPS
DIFFERENT TYPES OF RNN
MANY-TO-MANY $T_x=T_y$
MANY-TO-ONE EX. SENTIMENT ANALYSIS
ONE-TO-MANY · MUSIC GENERATION
MANY-TO-MANY $T_x \neq T_y$ TRANSLATION



MORE ON

RNNs

SEQUENCE MODELS · COURSERA

LANGUAGE MODELLING

HOW DO YOU KNOW IF SOMEONE SAID THE APPLE AND PAIR SALAD OR THE APPLE AND PEAR SALAD?

THE PURPOSE OF A LANG. MODEL IS TO CALCULATE THE PROBABILITIES

EX. CATS AVERAGE 15 HOURS OF SLEEP A DAY

SO GIVEN: CATS AVERAGE 15 WHAT IS THE PROB. THE NEXT WORD IS HOURS?

SAMPLING SENTENCES

1. TRAIN ON ALL HARRY POTTER BOOKS.
2. RANDOMLY SELECT A WORD (ONE OF THE TOP WORDS) (EX. THE)
3. PASS THIS INTO THE NEXT TIMESTAMP AND SAMPLE A NEW WORD
4. REPEAT UNTIL X WORDS OR YOU REACHED <EOS>

CAN DO AT CHARACTERLEVEL AS WELL

YAY! YOU ARE NOW YOUR OWN J.K. ROWLING

VANISHING GRADIENTS

THE CAT, WHO ALREADY ATE APPLES AND ORANGES AND A FEW MORE THINGS BLA BLA WAS FULL

THE CATS, WHO ALREADY ATE ... WERE FULL

NEED TO REMEMBER SING/PLURAL FOR A LONG TIME

SINCE LONG SENTENCE ⇒ DEEP RNN WE GET THE VANISHING GRADIENTS PROB WE HAVE IN STANDARD NNs – I.E THE GRADIENTS FOR CAT/CATS HAVE LITTLE OR NO EFFECT ON WAS/WERE.

NOTE SOMETIMES YOU SEE EXPLODING GRAD (AS OVERFLOW NAN) BUT THIS IS EASILY FIXED WITH GRADIENT CLIPPING

GATED RECURRENT UNIT GRU

HELPS RECALL IF CAT WAS SING. OR PLURAL

GRU

THE GRU ACTS AS A MEMORY — AT EVERY TIMESTEP IT CALCULATES A NEW $\tilde{c}$ TO STORE AND A GATE Γ_u DECIDES TO UPDATE c TO $\tilde{c}$ OR NOT

LONG SHORT TERM MEMORY (LSTM)

THE LSTM IS A VARIATION ON THE SAME THEME AS GRU BUT WITH AN ADDITIONAL Γ_f FORGET GATE

BI DIRECTIONAL RNNs (BRNN)

HE SAID, 'TEDDY BEARS ARE ON SALE'

HE SAID, 'TEDDY ROOSEVELT WAS A GREAT PRESIDENT'

PROBLEM: WITHOUT LOOKING FORWARD WE CAN'T SAY IF TEDDY IS A TOY OR A NAME

CAN BE LSTM OR GRU

ONE DISADVANTAGE IS THAT YOU NEED THE FULL SENTENCE BEFORE YOU BEGIN · SO NOT SUITABLE FOR LIVE SPEECH REC

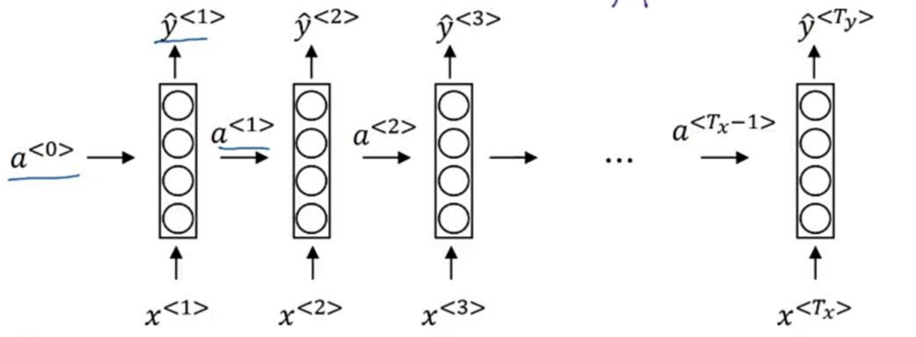
DEEP RNN

SINCE THEY ARE ALREADY TEMPORAL DEEP THEY USUALLY DON'T HAVE A LOT OF LAYERS

@TessFernandez

Forward Propagation

$a \leftarrow W_{ax} x^{<1>}$



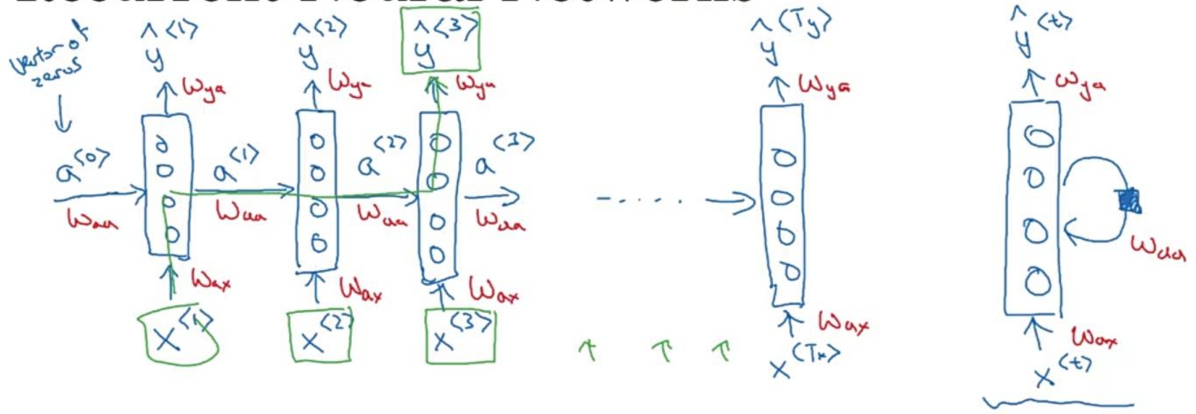
$a^{<0>} = \vec{0}.$

$a^{<1>} = g(W_{aa} a^{<0>} + W_{ax} x^{<1>} + b_a) \leftarrow$ tanh / ReLU

$\hat{y}^{<1>} = g(W_{ya} a^{<1>} + b_y) \leftarrow$ sigmoid

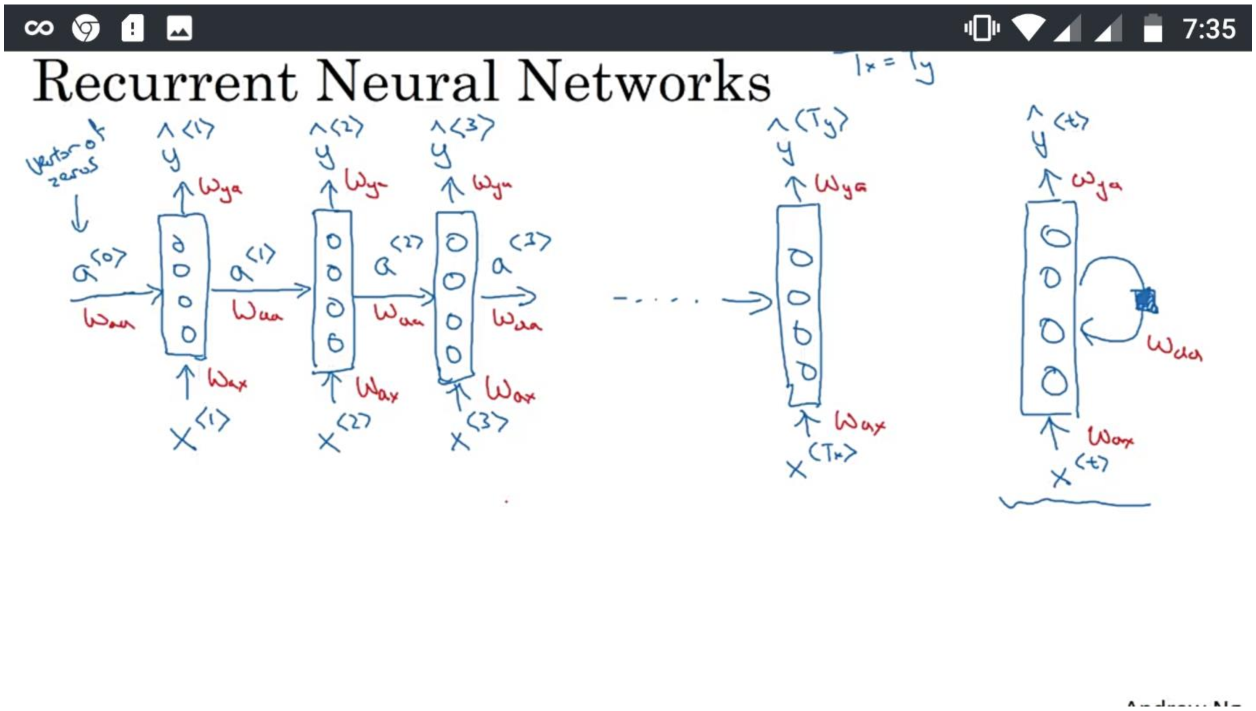
Recurrent Neural Networks

$T_x = T_y$

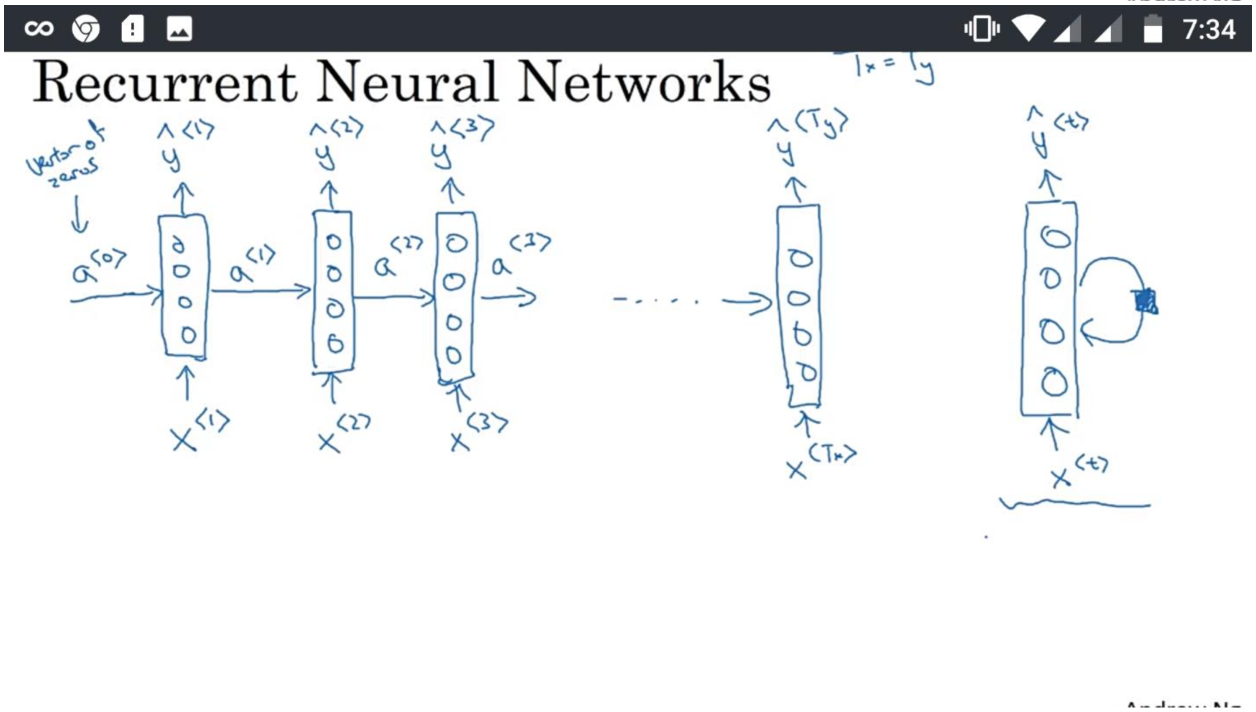


He said, "Teddy Roosevelt was a great President."

He said, "Teddy bears are on sale!"

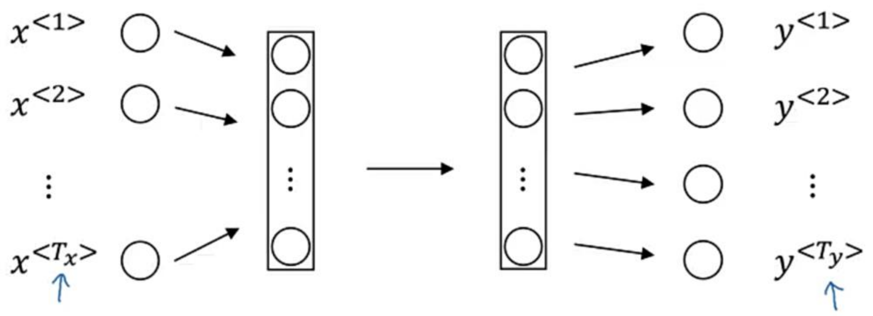

7:35

Recurrent Neural Networks


7:34

Recurrent Neural Networks

Why not a standard network?



Problems:

- Inputs, outputs can be different lengths in different examples.
- Doesn't share features learned across different positions of text.

Simplified RNN notation

$$a^{<t>} = g(\underset{(100,100)}{\underset{\uparrow}{\boxed{W_{aa}}}}\underset{100}{a^{<t-1>}} + \underset{(100,10{,}000)}{\underset{\uparrow}{\boxed{W_{ax}}}}\underset{10{,}000}{x^{<t>}} + b_a)$$

$$\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y)$$

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

$$100\updownarrow\left[\underbrace{W_{aa}}_{100} \vdots \underbrace{W_{ax}}_{10000}\right] = \underset{(100, 10100)}{W_a}$$

Recurrent Neural Networks

Learn about recurrent neural networks. This type of model has been proven to perform extremely well on temporal data. It has several variants including LSTMs, GRUs and Bidirectional RNNs, which you are going to learn about in this section.

Why sequence models

- Sequence Models like RNN and LSTMs have greatly transformed learning on sequences in the past few years.
- Examples of sequence data in applications:
  - Speech recognition (**sequence to sequence**):
    - X: wave sequence
    - Y: text sequence
  - Music generation (**one to sequence**):
    - X: nothing or an integer
    - Y: wave sequence
  - Sentiment classification (**sequence to one**):
    - X: text sequence
    - Y: integer rating from one to five
  - DNA sequence analysis (**sequence to sequence**):
    - X: DNA sequence
    - Y: DNA Labels
  - Machine translation (**sequence to sequence**):
    - X: text sequence (in one language)
    - Y: text sequence (in other language)
  - Video activity recognition (**sequence to one**):
    - X: video frames
    - Y: label (activity)
  - Name entity recognition (**sequence to sequence**):
    - X: text sequence
    - Y: label sequence
    - Can be used by seach engines to index different type of words inside a text.
- All of these problems with different input and output (sequence or not) can be addressed as supervised learning with label data X, Y as the training set.

Notation

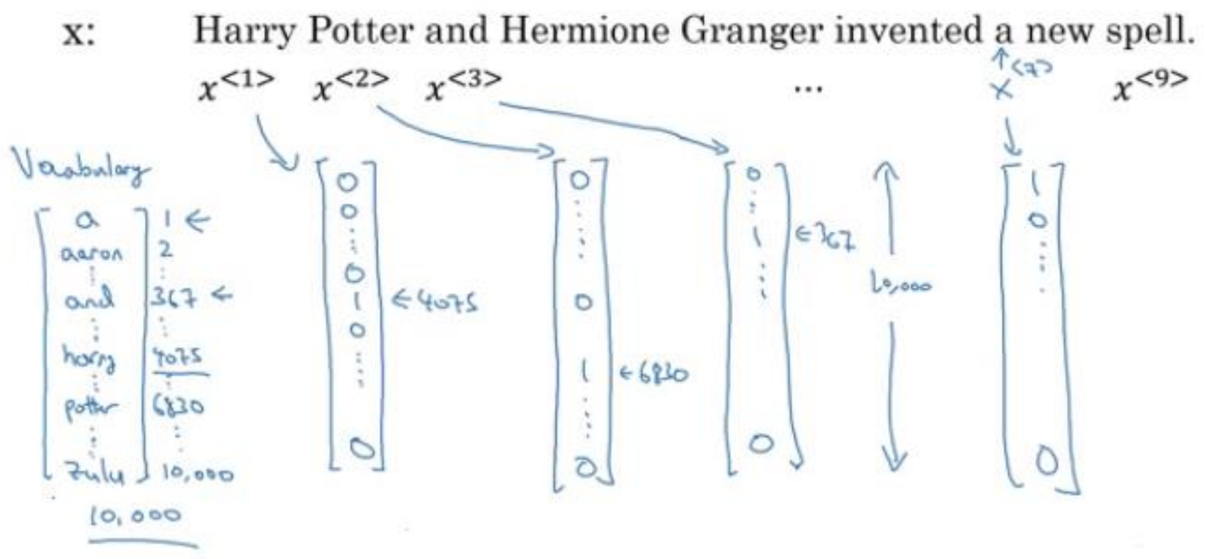
- In this section we will discuss the notations that we will use through the course.
- **Motivating example**:
    - Named entity recognition example:
        - X: "Harry Potter and Hermoine Granger invented a new spell."
        - Y: 1 1 0 1 1 0 0 0 0
        - Both elements has a shape of 9. 1 means its a name, while 0 means its not a name.
- We will index the first element of x by $x^{<1>}$, the second $x^{<2>}$ and so on.
    - $x^{<1>}$ = Harry
    - $x^{<2>}$ = Potter
- Similarly, we will index the first element of y by $y^{<1>}$, the second $y^{<2>}$ and so on.
    - $y^{<1>} = 1$
    - $y^{<2>} = 1$
- T_x is the size of the input sequence and T_y is the size of the output sequence.
    - $T_x = T_y = 9$ in the last example although they can be different in other problems.
- $x^{(i)<t>}$ is the element t of the sequence of input vector i. Similarly $y^{(i)<t>}$ means the t-th element in the output sequence of the i training example.
- $T_x^{(i)}$ the input sequence length for training example i. It can be different across the examples. Similarly for $T_y^{(i)}$ will be the length of the output sequence in the i-th training example.
- **Representing words**:
    - We will now work in this course with **NLP** which stands for natural language processing. One of the challenges of NLP is how can we represent a word?

    ii. We need a **vocabulary** list that contains all the words in our target sets.
        - Example:
            - [a ... And ... Harry ... Potter ... Zulu]

        - Each word will have a unique index that it can be represented with.
        - The sorting here is in alphabetical order.
    - Vocabulary sizes in modern applications are from 30,000 to 50,000. 100,000 is not uncommon. Some of the bigger companies use even a million.
    - To build vocabulary list, you can read all the texts you have and get m words with the most occurrence, or search online for m most occurrent words.

iii. Create a **one-hot encoding** sequence for each word in your dataset given the vocabulary you have created.
    - While converting, what if we meet a word thats not in your dictionary?
    - We can add a token in the vocabulary with name `<UNK>` which stands for unknown text and use its index for your one-hot vector.

  - Full example:

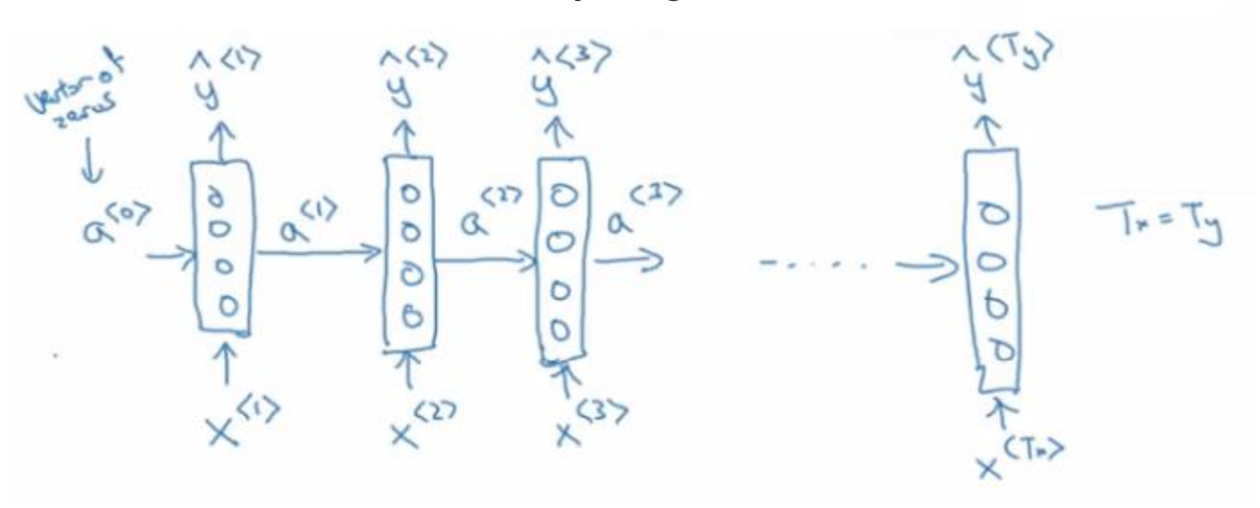

x: Harry Potter and Hermione Granger invented a new spell.
$x^{<1>}$ $x^{<2>}$ $x^{<3>}$ ... $x^{<7>}$ $x^{<9>}$
Vocabulary
a 1
aaron 2
and 367
harry 4075
potter 6830
zulu 10,000
10,000
←4075
←6830
←367
10,000

- The goal is given this representation for x to learn a mapping using a sequence model to then target output y as a supervised learning problem.

Recurrent Neural Network Model

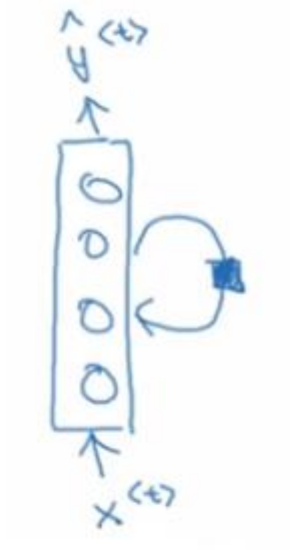
- Why not to use a standard network for sequence tasks? There are two problems:
  - Inputs, outputs can be different lengths in different examples.

    - This can be solved for normal NNs by paddings with the maximum lengths but it's not a good solution.
  - Doesn't share features learned across different positions of text/sequence.
    - Using a feature sharing like in CNNs can significantly reduce the number of parameters in your model. That's what we will do in RNNs.
- Recurrent neural network doesn't have either of the two mentioned problems.
- Lets build a RNN that solves **name entity recognition** task:



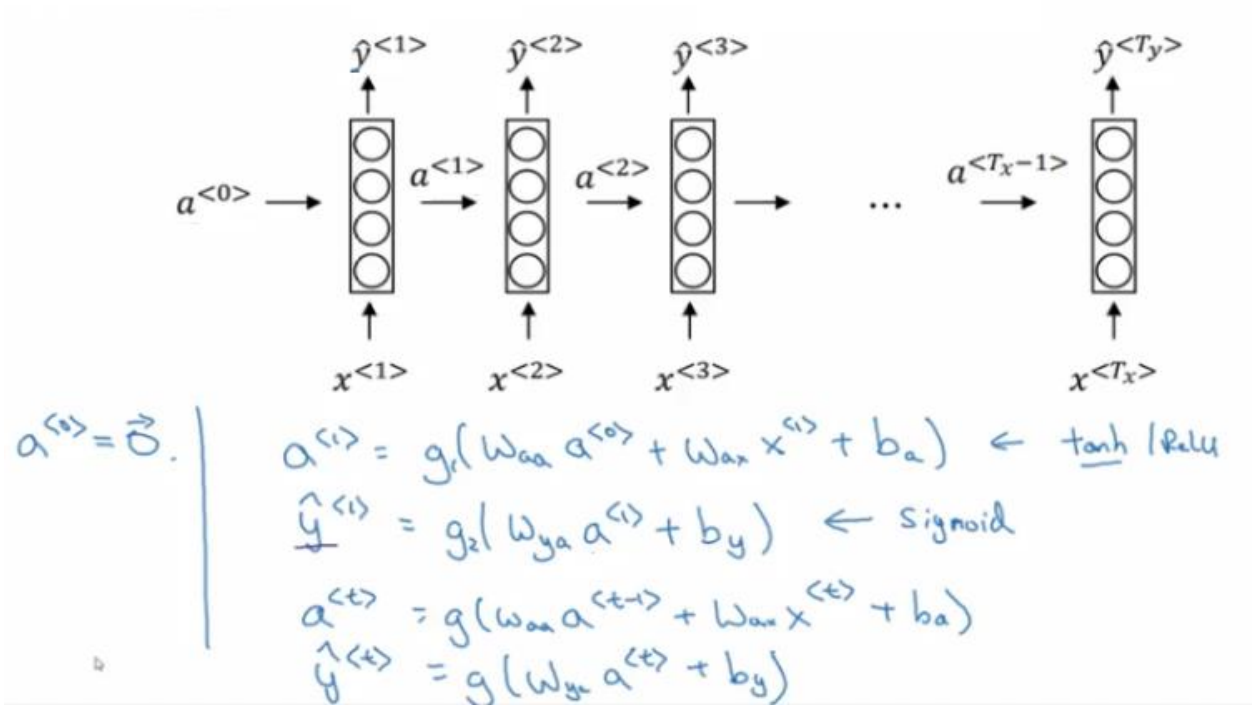
  - In this problem $T_x = T_y$. In other problems where they aren't equal, the RNN architecture may be different.
  - $a^{<0>}$ is usually initialized with zeros, but some others may initialize it randomly in some cases.
  - There are three weight matrices here: W_{ax}, W_{aa}, and W_{ya} with shapes:
    - W_{ax}: (NoOfHiddenNeurons, n_x)
    - W_{aa}: (NoOfHiddenNeurons, NoOfHiddenNeurons)
    - W_{ya}: (n_y, NoOfHiddenNeurons)
- The weight matrix W_{aa} is the memory the RNN is trying to maintain from the previous layers.

- A lot of papers and books write the same architecture this way:



  - It's harder to interpreter. It's easier to roll this drawings to the unrolled version.
- In the discussed RNN architecture, the current output $\hat{y}^{<t>}$ depends on the previous inputs and activations.
- Let's have this example 'He Said, "Teddy Roosevelt was a great president"'. In this example Teddy is a person name but we know that from the word **president** that came after Teddy not from **He** and **said** that were before it.
- So limitation of the discussed architecture is that it can not learn from elements later in the sequence. To address this problem we will later discuss **Bidirectional RNN** (BRNN).

- Now let's discuss the forward propagation equations on the discussed architecture:



$a^{<0>} = \vec{0}$.

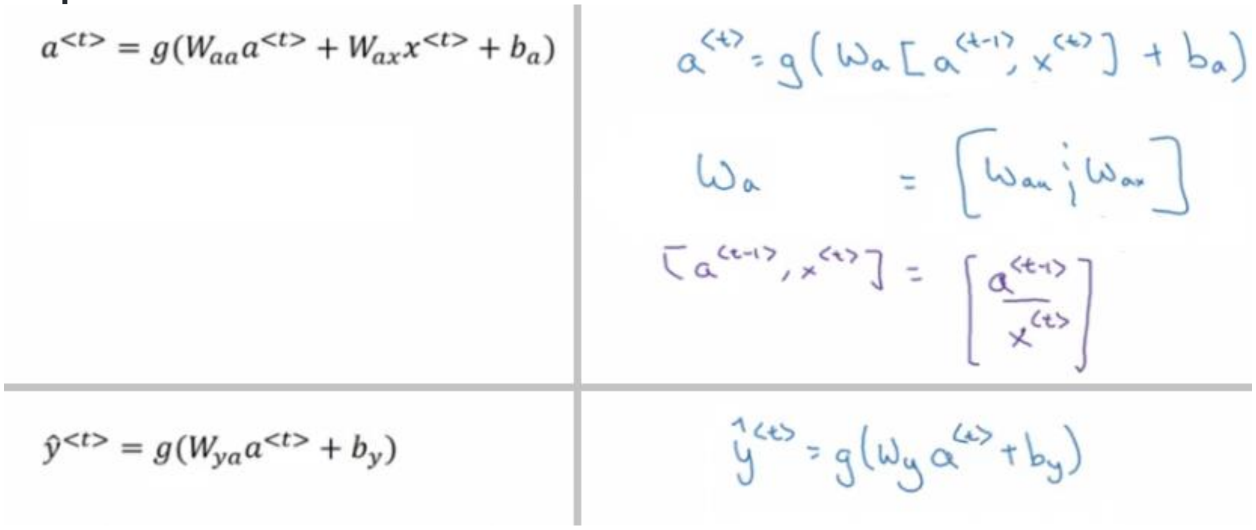
$a^{<1>} = g_1(W_{aa} a^{<0>} + W_{ax} x^{<1>} + b_a) \leftarrow \underline{\text{tanh}} / \text{ReLU}$

$\underline{\hat{y}}^{<1>} = g_2(W_{ya} a^{<1>} + b_y) \leftarrow \text{Sigmoid}$

$a^{<t>} = g(W_{aa} a^{<t-1>} + W_{ax} x^{<t>} + b_a)$

$\hat{y}^{<t>} = g(W_{ya} a^{<t>} + b_y)$

  - The activation function of a is usually tanh or ReLU and for y depends on your task choosing some activation functions like sigmoid and softmax. In name entity recognition task we will use sigmoid because we only have two classes.
- In order to help us develop complex RNN architectures, the last equations needs to be simplified a bit.
- **Simplified RNN notation**:



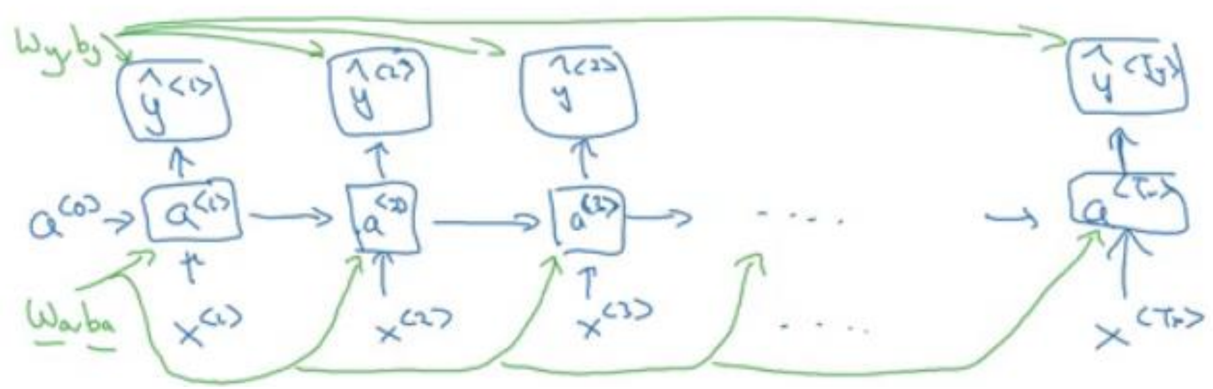
$a^{<t>} = g(W_{aa}a^{<t>} + W_{ax}x^{<t>} + b_a)$	$a^{<t>} = g(W_a[a^{<t-1>}, x^{<t>}] + b_a)$ $W_a = [W_{aa} \vdots W_{ax}]$ $[a^{<t-1>}, x^{<t>}] = \begin{bmatrix} a^{<t-1>} \\ \hline x^{<t>} \end{bmatrix}$
$\hat{y}^{<t>} = g(W_{ya}a^{<t>} + b_y)$	$\hat{y}^{<t>} = g(W_y a^{<t>} + b_y)$

  - w_a is w_{aa} and w_{ax} stacked horizontally.
  - $[a^{<t-1>}, x^{<t>}]$ is $a^{<t-1>}$ and $x^{<t>}$ stacked vertically.

- w_a shape: (NoOfHiddenNeurons, NoOfHiddenNeurons + n_x)
- $[a^{<t-1>}, x^{<t>}]$ shape: (NoOfHiddenNeurons + n_x, 1)

Backpropagation through time

- Let's see how backpropagation works with the RNN architecture.
- Usually deep learning frameworks do backpropagation automatically for you. But it's useful to know how it works in RNNs.
- Here is the graph:

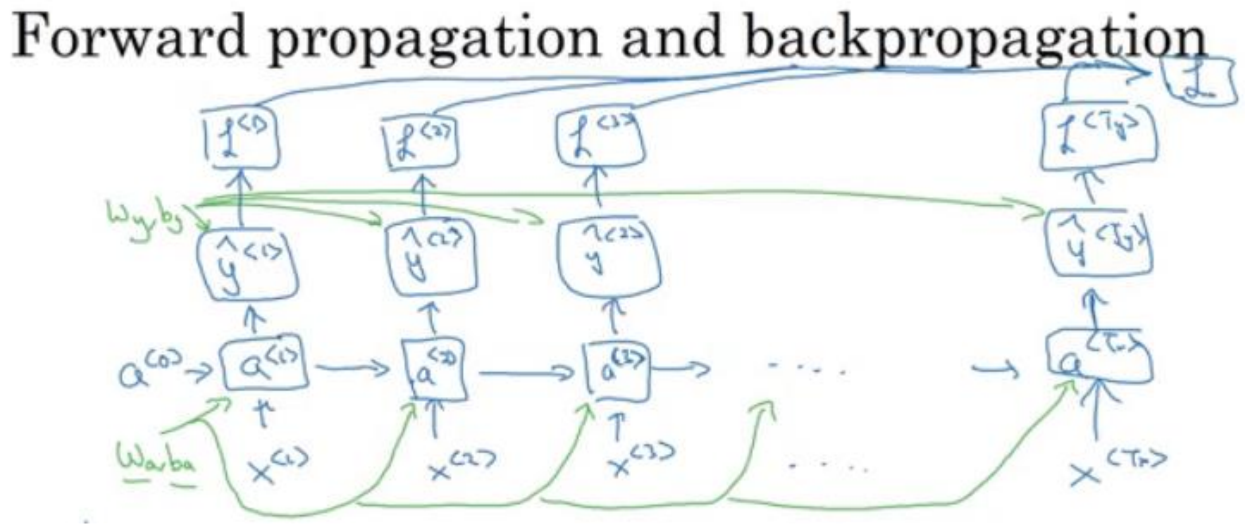


  - Where w_a, b_a, w_y, and b_y are shared across each element in a sequence.
- We will use the cross-entropy loss function:

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

  - Where the first equation is the loss for one example and the loss for the whole sequence is given by the summation over all the calculated single example losses.

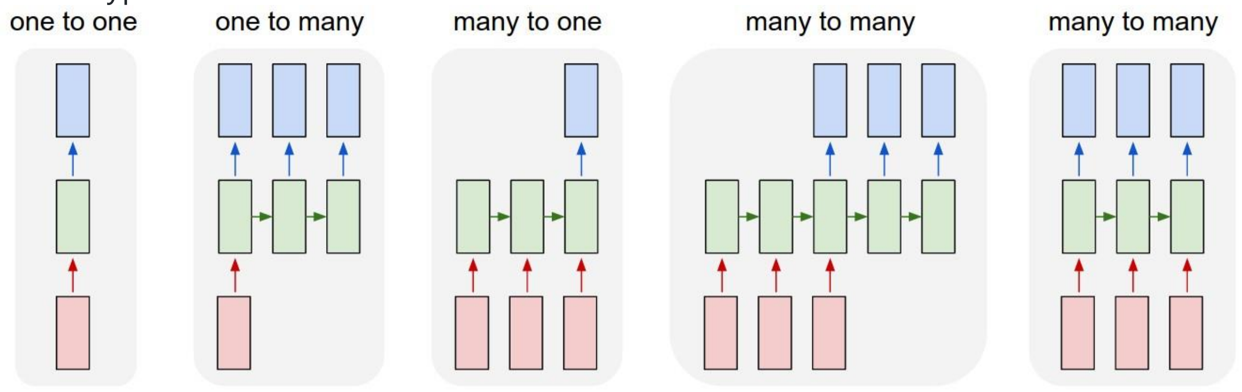
- Graph with losses:



- The backpropagation here is called **backpropagation through time** because we pass activation a from one sequence element to another like backwards in time.

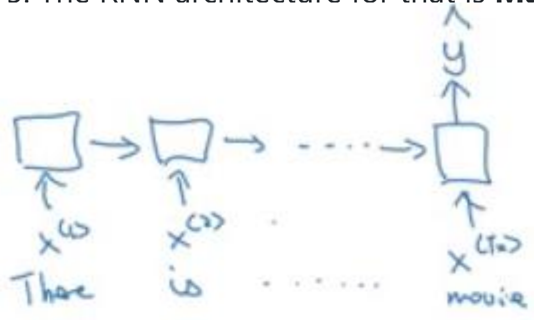
Different types of RNNs

- So far we have seen only one RNN architecture in which T_x equals T_Y. In some other problems, they may not equal so we need different architectures.
- The ideas in this section was inspired by Andrej Karpathy blog. Mainly this image has all types:

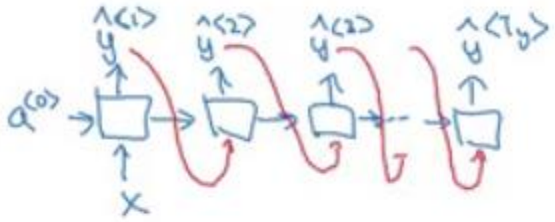


- The architecture we have descried before is called **Many to Many**.

- In sentiment analysis problem, X is a text while Y is an integer that rangers from 1 to 5. The RNN architecture for that is **Many to One** as in Andrej Karpathy image.

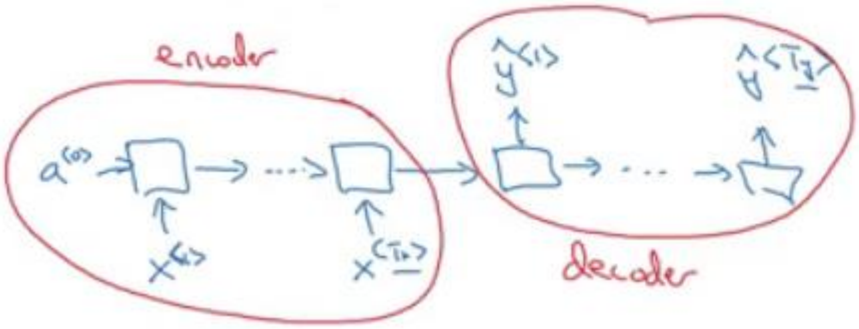


- A **One to Many** architecture application would be music generation.



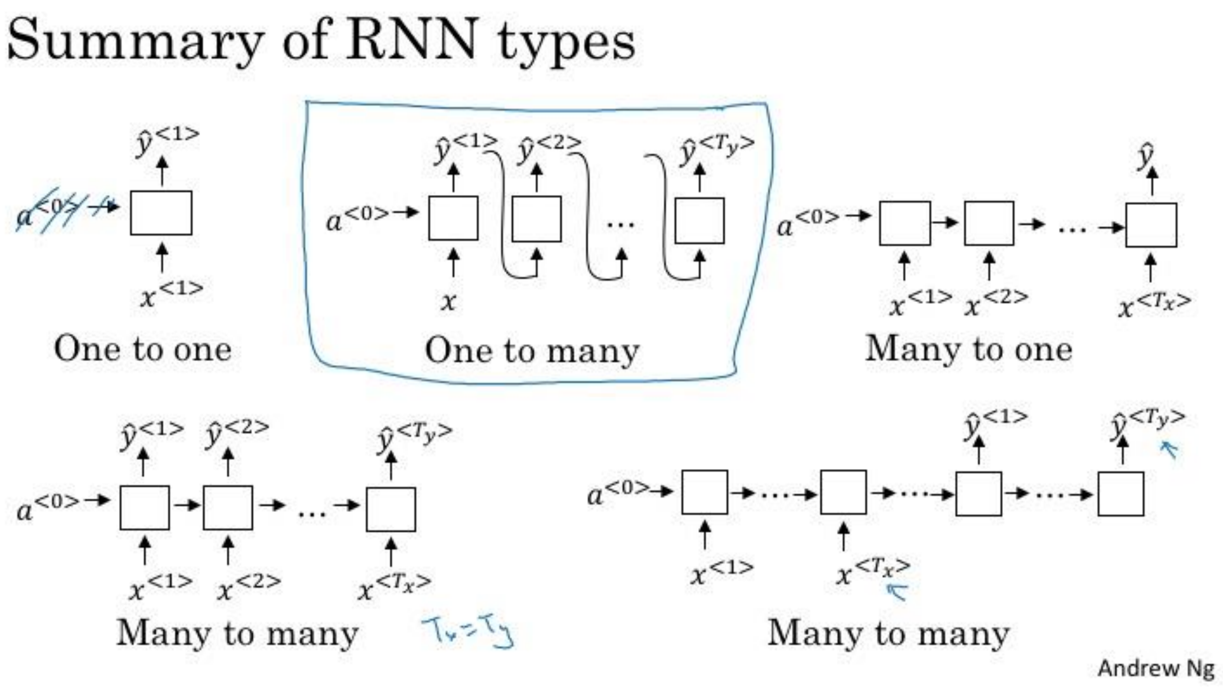
    - Note that starting the second layer we are feeding the generated output back to the network.

- There are another interesting architecture in **Many To Many**. Applications like machine translation inputs and outputs sequences have different lengths in most of the cases. So an alternative *Many To Many* architecture that fits the translation would be as follows:



    - There are an encoder and a decoder parts in this architecture. The encoder encodes the input sequence into one matrix and feed it to the decoder to generate the outputs. Encoder and decoder have different weight matrices.

- Summary of RNN types:

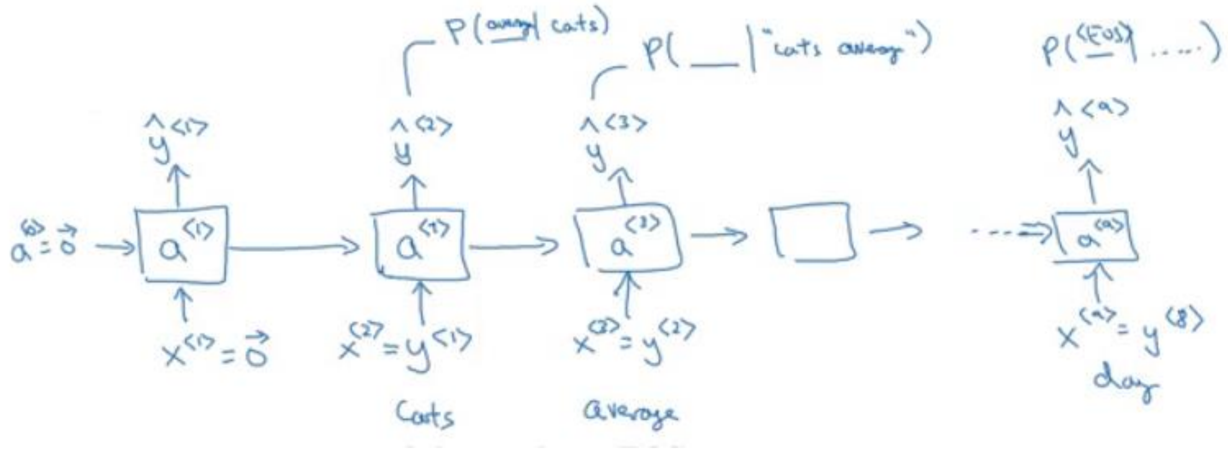


- There is another architecture which is the **attention** architecture which we will talk about in chapter 3.

Language model and sequence generation

- RNNs do very well in language model problems. In this section, we will build a language model using RNNs.
- **What is a language model**
    - Let's say we are solving a speech recognition problem and someone says a sentence that can be interpreted into to two sentences:
        - The apple and **pair** salad
        - The apple and **pear** salad
    - **Pair** and **pear** sounds exactly the same, so how would a speech recognition application choose from the two.
    - That's where the language model comes in. It gives a probability for the two sentences and the application decides the best based on this probability.
- The job of a language model is to give a probability of any given sequence of words.
- **How to build language models with RNNs?**

    - The first thing is to get a **training set**: a large corpus of target language text.
    - Then tokenize this training set by getting the vocabulary and then one-hot each word.
    - Put an end of sentence token `<EOS>` with the vocabulary and include it with each converted sentence. Also, use the token `<UNK>` for the unknown words.
- Given the sentence "Cats average 15 hours of sleep a day. `<EOS>`"
    - In training time we will use this:



    - The loss function is defined by cross-entropy loss:

$$\mathcal{L}(\hat{y}^{<t>}, y^{<t>}) = -\sum_i y_i^{<t>} \log \hat{y}_i^{<t>}$$

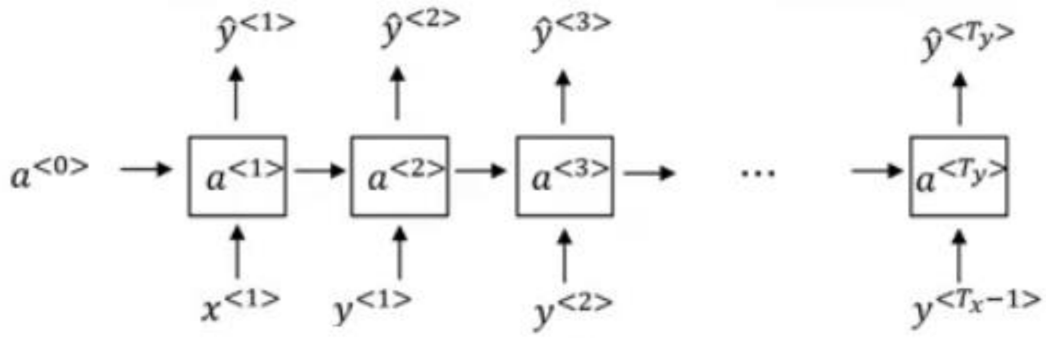
$$\mathcal{L} = \sum \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

        - `i` is for all elements in the corpus, `t` - for all timesteps.
- To use this model:
    i. For predicting the chance of **next word**, we feed the sentence to the RNN and then get the final $y^{\wedge<t>}$ hot vector and sort it by maximum probability.
    ii. For taking the **probability of a sentence**, we compute this:
        - $p(y^{<1>}, y^{<2>}, y^{<3>}) = p(y^{<1>}) * p(y^{<2>} \mid y^{<1>}) * p(y^{<3>} \mid y^{<1>}, y^{<2>})$
        - This is simply feeding the sentence into the RNN and multiplying the probabilities (outputs).

Sampling novel sequences

- After a sequence model is trained on a language model, to check what the model has learned you can apply it to sample novel sequence.

- Lets see the steps of how we can sample a novel sequence from a trained sequence language model:
    i. Given this model:

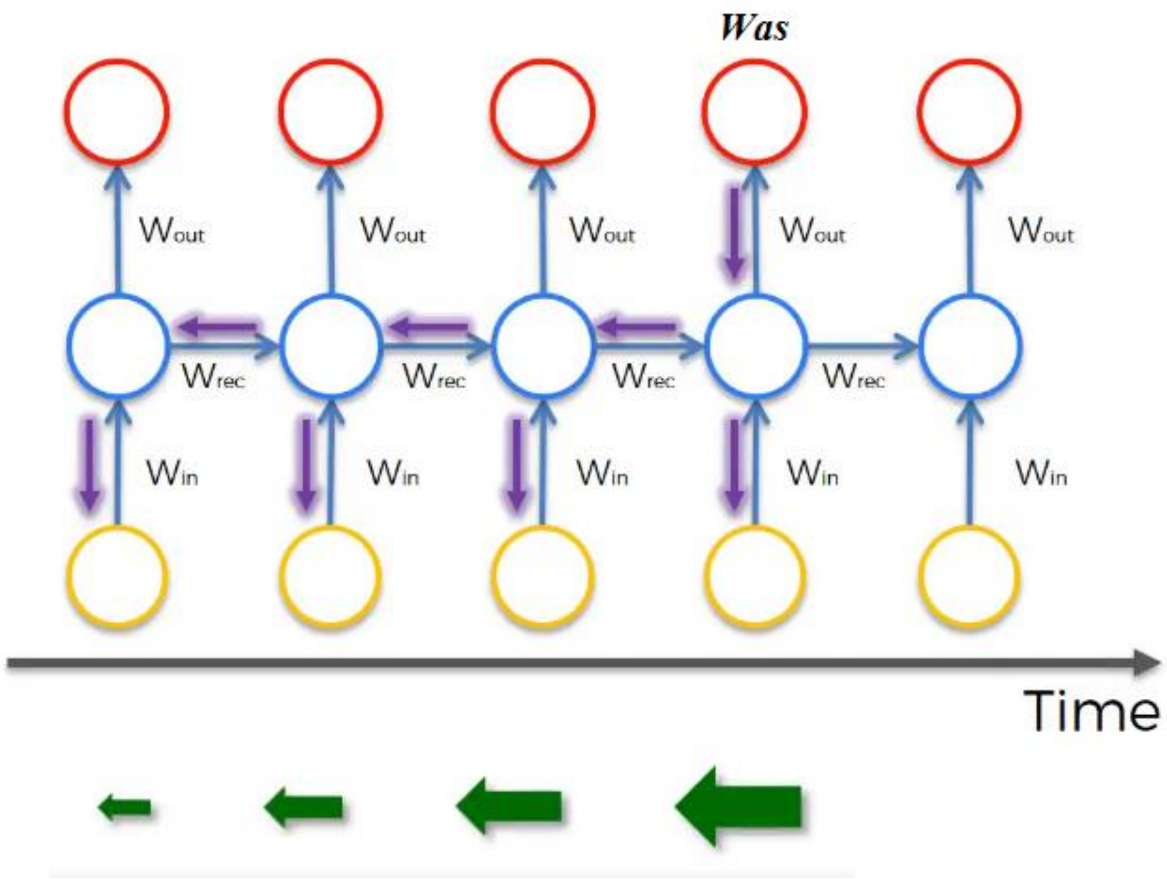


    ii. We first pass $a^{<0>}$ = zeros vector, and $x^{<1>}$ = zeros vector.
    iii. Then we choose a prediction randomly from distribution obtained by $\hat{y}^{<1>}$. For example it could be "The".
        - In numpy this can be implemented using: `numpy.random.choice(...)`
        - This is the line where you get a random beginning of the sentence each time you sample run a novel sequence.
    iv. We pass the last predicted word with the calculated $a^{<1>}$
    v. We keep doing 3 & 4 steps for a fixed length or until we get the `<EOS>` token.
    vi. You can reject any `<UNK>` token if you mind finding it in your output.
- So far we have to build a word-level language model. It's also possible to implement a **character-level** language model.
- In the character-level language model, the vocabulary will contain `[a-zA-Z0-9]`, punctuation, special characters and possibly token.
- Character-level language model has some pros and cons compared to the word-level language model
    - Pros:
        a. There will be no `<UNK>` token - it can create any word.
    - Cons:
        . The main disadvantage is that you end up with much longer sequences.
        a. Character-level language models are not as good as word-level language models at capturing long range dependencies between how the the earlier parts of the sentence also affect the later part of the sentence.
        b. Also more computationally expensive and harder to train.

- The trend Andrew has seen in NLP is that for the most part, a word-level language model is still used, but as computers get faster there are more and more applications where people are, at least in some special cases, starting to look at more character-level models. Also, they are used in specialized applications where you might need to deal with unknown words or other vocabulary words a lot. Or they are also used in more specialized applications where you have a more specialized vocabulary.

Vanishing gradients with RNNs

- One of the problems with naive RNNs that they run into **vanishing gradient** problem.
- An RNN that process a sequence data with the size of 10,000 time steps, has 10,000 deep layers which is very hard to optimize.
- Let's take an example. Suppose we are working with language modeling problem and there are two sequences that model tries to learn:
  - "The **cat**, which already ate ..., **was** full"
  - "The **cats**, which already ate ..., **were** full"
  - Dots represent many words in between.
- What we need to learn here that "was" came with "cat" and that "were" came with "cats". The naive RNN is not very good at capturing very long-term dependencies like this.
- As we have discussed in Deep neural networks, deeper networks are getting into the vanishing gradient problem. That also happens with RNNs with a long sequence size.

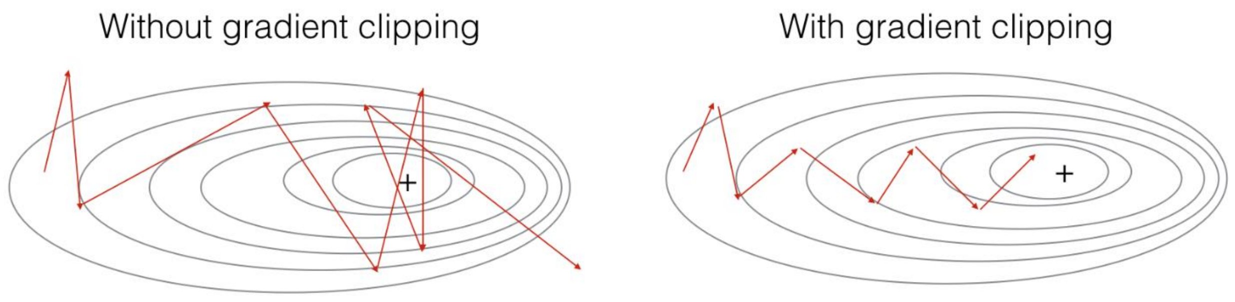


- For computing the word "was", we need to compute the gradient for everything behind. Multiplying fractions tends to vanish the gradient, while multiplication of large number tends to explode it.

  - Therefore some of your weights may not be updated properly.

- In the problem we descried it means that its hard for the network to memorize "was" word all over back to "cat". So in this case, the network won't identify the singular/plural words so that it gives it the right grammar form of verb was/were.
- The conclusion is that RNNs aren't good in **long-term dependencies**.
- In theory, RNNs are absolutely capable of handling such "long-term dependencies." A human could carefully pick parameters for them to solve toy problems of this form. Sadly, in practice, RNNs don't seem to be able to learn them. http://colah.github.io/posts/2015-08-Understanding-LSTMs/
- *Vanishing gradients* problem tends to be the bigger problem with RNNs than the *exploding gradients* problem. We will discuss how to solve it in next sections.

- Exploding gradients can be easily seen when your weight values become `NaN`. So one of the ways solve exploding gradient is to apply **gradient clipping** means if your gradient is more than some threshold - re-scale some of your gradient vector so that is not too big. So there are cliped according to some maximum value.

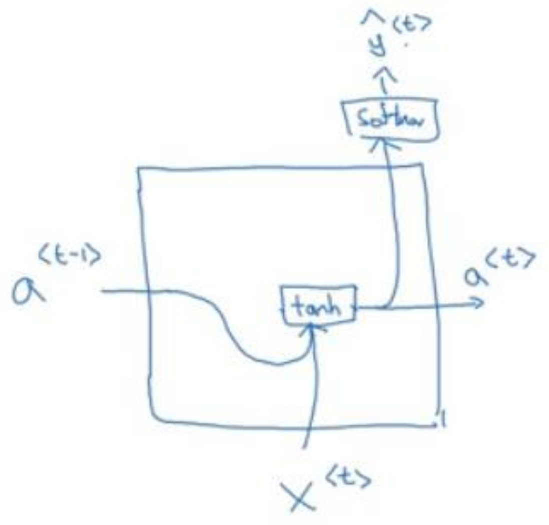


- **Extra**:
  - Solutions for the Exploding gradient problem:
    - Truncated backpropagation.
      - Not to update all the weights in the way back.
      - Not optimal. You won't update all the weights.
    - Gradient clipping.
  - Solution for the Vanishing gradient problem:
    - Weight initialization.
      - Like He initialization.
    - Echo state networks.
    - Use LSTM/GRU networks.
      - Most popular.
      - We will discuss it next.

Gated Recurrent Unit (GRU)

- GRU is an RNN type that can help solve the vanishing gradient problem and can remember the long-term dependencies.

- The basic RNN unit can be visualized to be like this:



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

- We will represent the GRU with a similar drawings.
- Each layer in **GRUs** has a new variable c which is the memory cell. It can tell to whether memorize something or not.
- In GRUs, C$^{<t>}$ = a$^{<t>}$
- Equations of the GRUs:

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

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

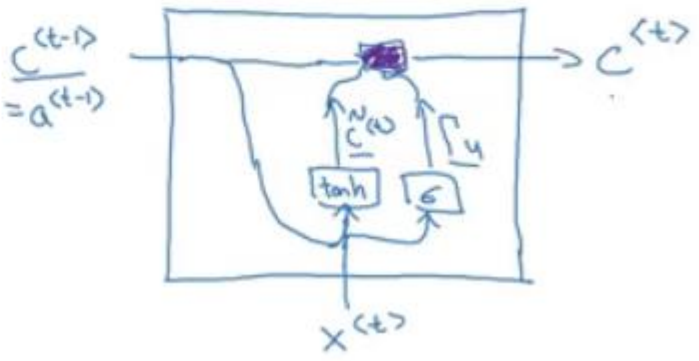
$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1-\Gamma_u) * c^{<t-1>}$$

  - The update gate is between 0 and 1
    - To understand GRUs imagine that the update gate is either 0 or 1 most of the time.
  - So we update the memory cell based on the update cell and the previous cell.
- Lets take the cat sentence example and apply it to understand this equations:
  - Sentence: "The **cat**, which already ate ...., **was** full"

- We will suppose that U is 0 or 1 and is a bit that tells us if a singular word needs to be memorized.
- Splitting the words and get values of C and U at each place:
  -

Word	Update gate(U)	
The	0	val
cat	1	new_va
which	0	new_va
already	0	new_va
...	0	new_va
was	1 (I don't need it anymore)	newer_
full	..	..

- Drawing for the GRUs



  - Drawings like in http://colah.github.io/posts/2015-08-Understanding-LSTMs/ is so popular and makes it easier to understand GRUs and LSTMs. But Andrew Ng finds it's better to look at the equations.

- Because the update gate U is usually a small number like 0.00001, GRUs doesn't suffer the vanishing gradient problem.
  - In the equation this makes $C^{<t>} = C^{<t-1>}$ in a lot of cases.
- Shapes:
  - $a^{<t>}$ shape is (NoOfHiddenNeurons, 1)
  - $c^{<t>}$ is the same as $a^{<t>}$
  - $c^{\sim<t>}$ is the same as $a^{<t>}$
  - $u^{<t>}$ is also the same dimensions of $a^{<t>}$
- The multiplication in the equations are element wise multiplication.
- What has been descried so far is the Simplified GRU unit. Let's now describe the full one:
  - The full GRU contains a new gate that is used with to calculate the candidate C. The gate tells you how relevant is $C^{<t-1>}$ to $C^{<t>}$
  - Equations:

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

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

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

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

  - Shapes are the same
- So why we use these architectures, why don't we change them, how we know they will work, why not add another gate, why not use the simpler GRU instead of the full GRU; well researchers has experimented over years all the various types of these architectures with many many different versions and also addressing the vanishing gradient problem. They have found that full GRUs are one of the best RNN architectures to be used for many different problems. You can make your design but put in mind that GRUs and LSTMs are standards.

Long Short Term Memory (LSTM)

- LSTM - the other type of RNN that can enable you to account for long-term dependencies. It's more powerful and general than GRU.
- In LSTM , $C^{<t>}$!= $a^{<t>}$
- Here are the equations of an LSTM unit:

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

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

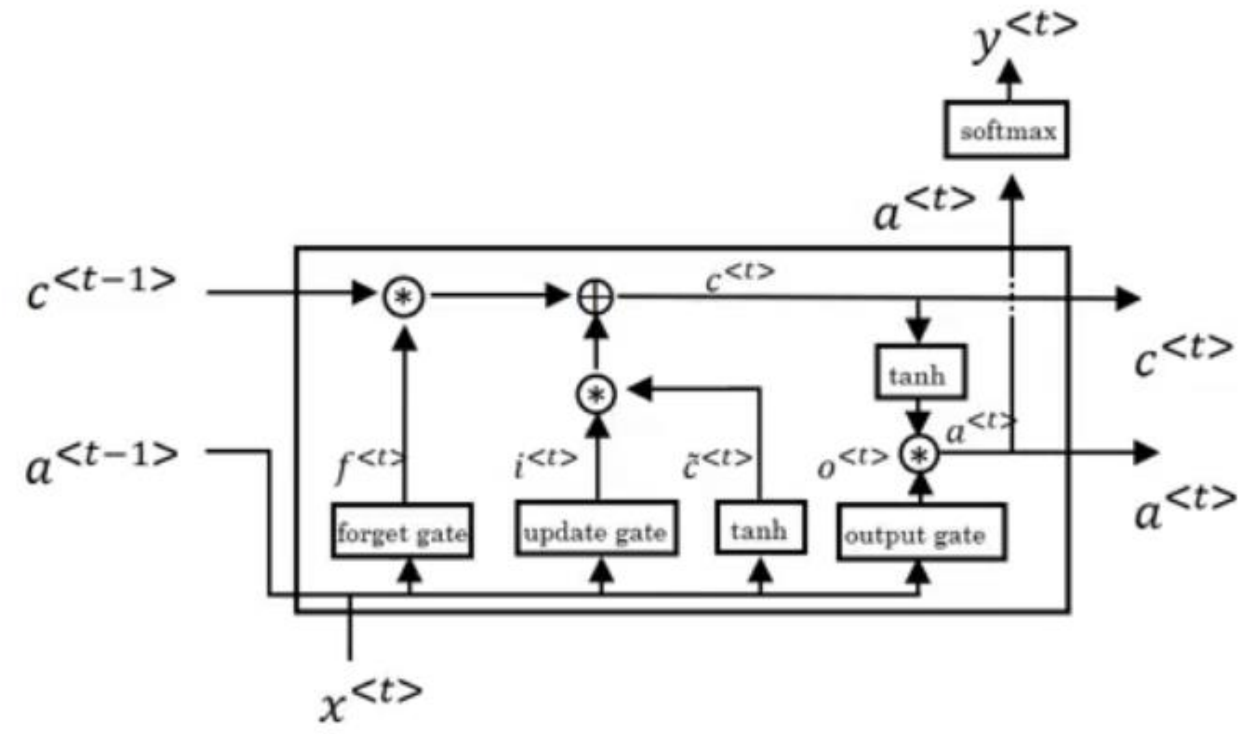
$$\text{(forget)} - \Gamma_f = \sigma(W_f[\, a^{<t-1>}, x^{<t>}] + b_f)$$

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

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

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

- In GRU we have an update gate U, a relevance gate r, and a candidate cell variables $C^{\sim<t>}$ while in LSTM we have an update gate U (sometimes it's called input gate I), a forget gate F, an output gate O, and a candidate cell variables $C^{\sim<t>}$
- Drawings (inspired by http://colah.github.io/posts/2015-08-Understanding-LSTMs/):

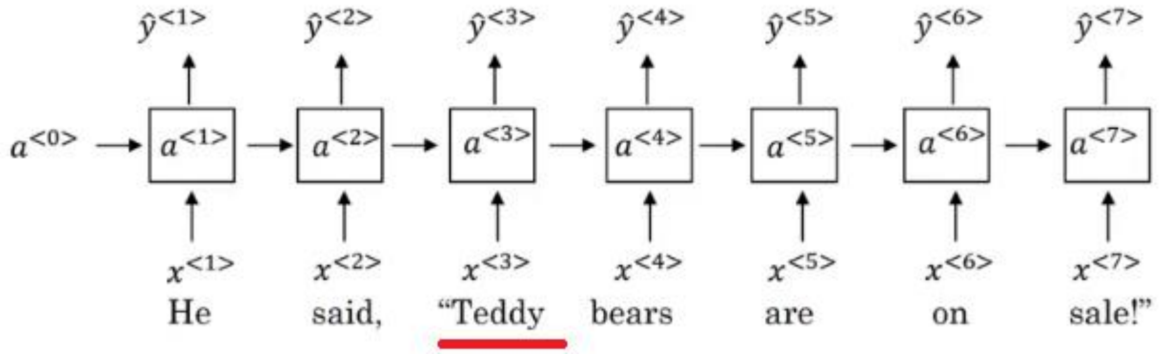


- Some variants on LSTM includes:

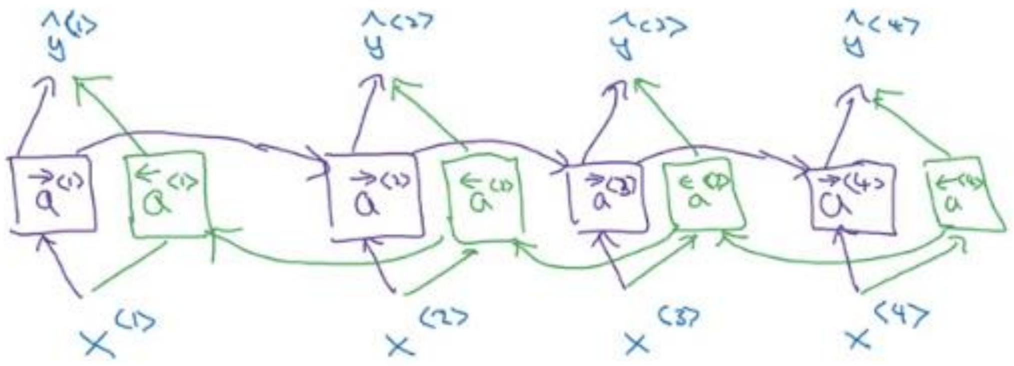
    - LSTM with **peephole connections**.
        - The normal LSTM with $C^{<t-1>}$ included with every gate.
- There isn't a universal superior between LSTM and it's variants. One of the advantages of GRU is that it's simpler and can be used to build much bigger network but the LSTM is more powerful and general.

Bidirectional RNN

- There are still some ideas to let you build much more powerful sequence models. One of them is bidirectional RNNs and another is Deep RNNs.
- As we saw before, here is an example of the Name entity recognition task:



- The name **Teddy** cannot be learned from **He** and **said**, but can be learned from **bears**.
- BiRNNs fixes this issue.
- Here is BRNNs architecture:

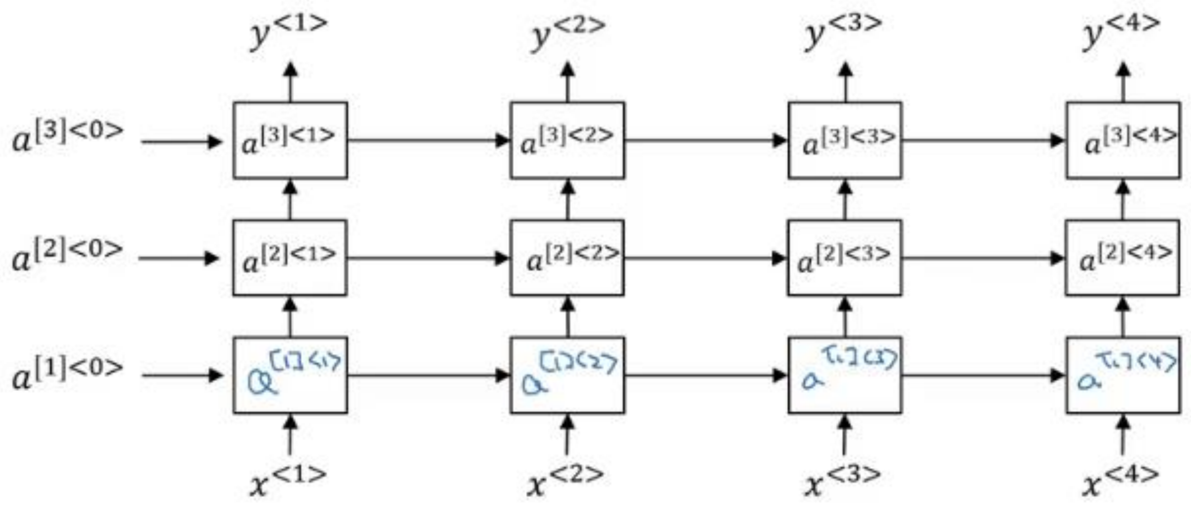

$\hat{y}^{<1>}$
$\hat{y}^{<2>}$
$\hat{y}^{<3>}$
$\hat{y}^{<4>}$
$\overrightarrow{a}^{<1>}$
$\overleftarrow{a}^{<1>}$
$\overrightarrow{a}^{<2>}$
$\overleftarrow{a}^{<2>}$
$\overrightarrow{a}^{<3>}$
$\overleftarrow{a}^{<3>}$
$\overrightarrow{a}^{<4>}$
$\overleftarrow{a}^{<4>}$
$x^{<1>}$
$x^{<2>}$
$x^{<3>}$
$x^{<4>}$

- Note, that BiRNN is an **acyclic graph**.
- Part of the forward propagation goes from left to right, and part - from right to left. It learns from both sides.

- To make predictions we use ŷ$^{<t>}$ by using the two activations that come from left and right.
- The blocks here can be any RNN block including the basic RNNs, LSTMs, or GRUs.
- For a lot of NLP or text processing problems, a BiRNN with LSTM appears to be commonly used.
- The disadvantage of BiRNNs that you need the entire sequence before you can process it. For example, in live speech recognition if you use BiRNNs you will need to wait for the person who speaks to stop to take the entire sequence and then make your predictions.

Deep RNNs

- In a lot of cases the standard one layer RNNs will solve your problem. But in some problems its useful to stack some RNN layers to make a deeper network.
- For example, a deep RNN with 3 layers would look like this:

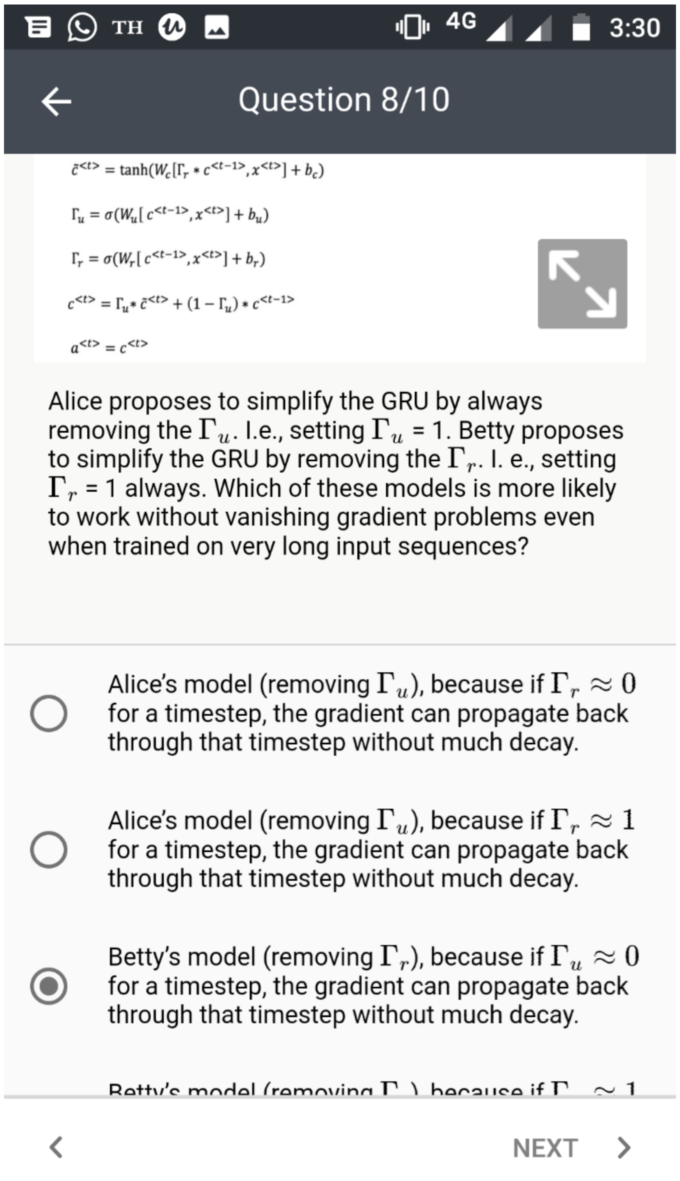


- In feed-forward deep nets, there could be 100 or even 200 layers. In deep RNNs stacking 3 layers is already considered deep and expensive to train.
- In some cases you might see some feed-forward network layers connected after recurrent cell.

Back propagation with RNNs

- In modern deep learning frameworks, you only have to implement the forward pass, and the framework takes care of the backward pass, so most deep learning engineers do not need to bother with the details of the backward pass. If however you are an expert in calculus and want to see the details of backprop in RNNs, you can work through this optional portion of the notebook.

Quiz Notes:



TH 4G 3:30

Question 8/10

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

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

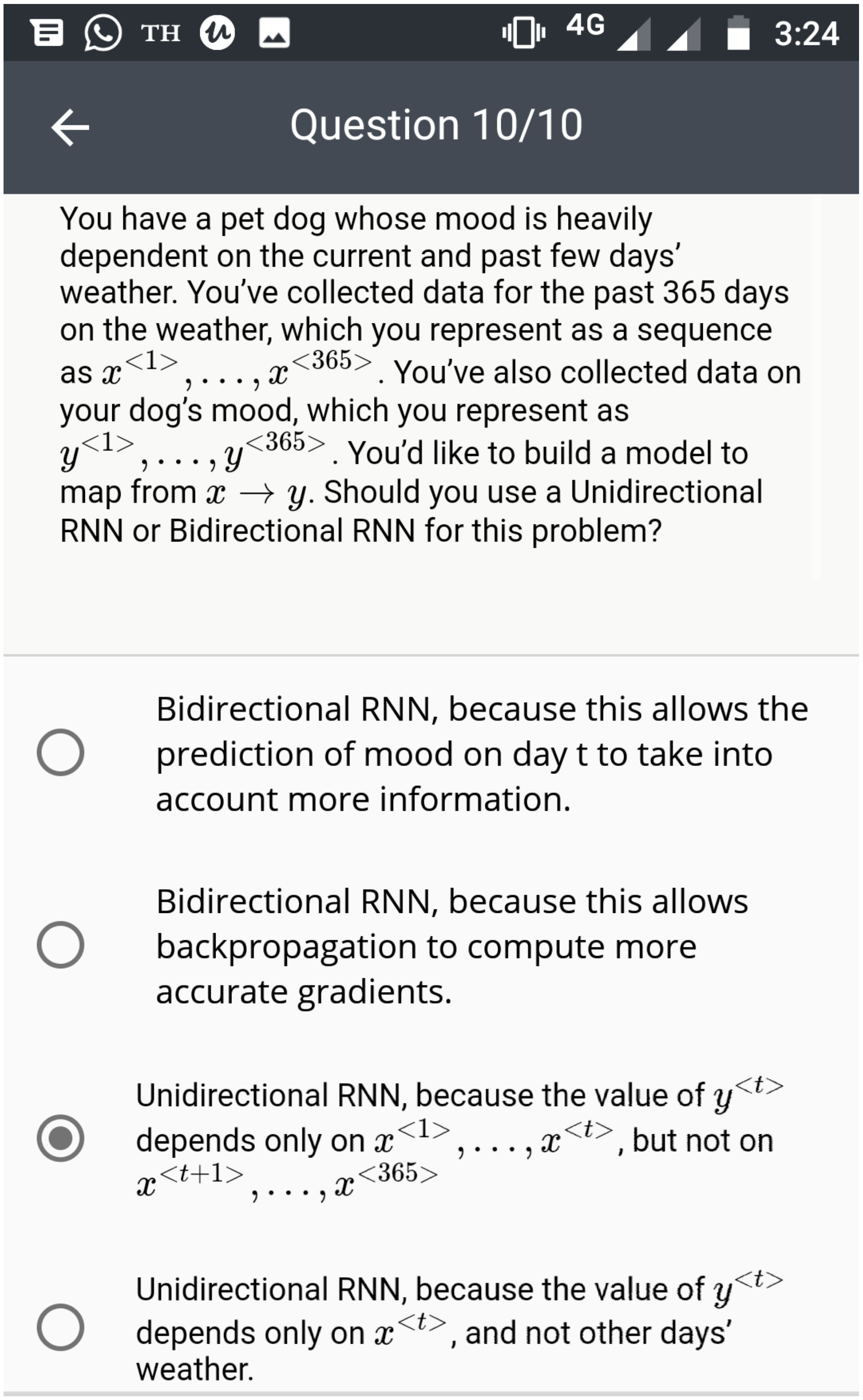
$\Gamma_r = \sigma(W_r[c^{<t-1>}, x^{<t>}] + b_r)$

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

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

Alice proposes to simplify the GRU by always removing the Γ_u. I.e., setting Γ_u = 1. Betty proposes to simplify the GRU by removing the Γ_r. I. e., setting Γ_r = 1 always. Which of these models is more likely to work without vanishing gradient problems even when trained on very long input sequences?

- [ ] Alice's model (removing Γ_u), because if $\Gamma_r \approx 0$ for a timestep, the gradient can propagate back through that timestep without much decay.
- [ ] Alice's model (removing Γ_u), because if $\Gamma_r \approx 1$ for a timestep, the gradient can propagate back through that timestep without much decay.
- [✓] Betty's model (removing Γ_r), because if $\Gamma_u \approx 0$ for a timestep, the gradient can propagate back through that timestep without much decay.
- [ ] Betty's model (removing Γ) because if Γ ≈ 1



Question 10/10

You have a pet dog whose mood is heavily dependent on the current and past few days' weather. You've collected data for the past 365 days on the weather, which you represent as a sequence as $x^{<1>}, \dots, x^{<365>}$. You've also collected data on your dog's mood, which you represent as $y^{<1>}, \dots, y^{<365>}$. You'd like to build a model to map from $x \to y$. Should you use a Unidirectional RNN or Bidirectional RNN for this problem?

- ○ Bidirectional RNN, because this allows the prediction of mood on day t to take into account more information.
- ○ Bidirectional RNN, because this allows backpropagation to compute more accurate gradients.
- [✓ Unidirectional RNN, because the value of $y^{<t>}$ depends only on $x^{<1>}, \dots, x^{<t>}$, but not on $x^{<t+1>}, \dots, x^{<365>}$]
- ○ Unidirectional RNN, because the value of $y^{<t>}$ depends only on $x^{<t>}$, and not other days' weather.

GRU

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

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

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

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

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

LSTM

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

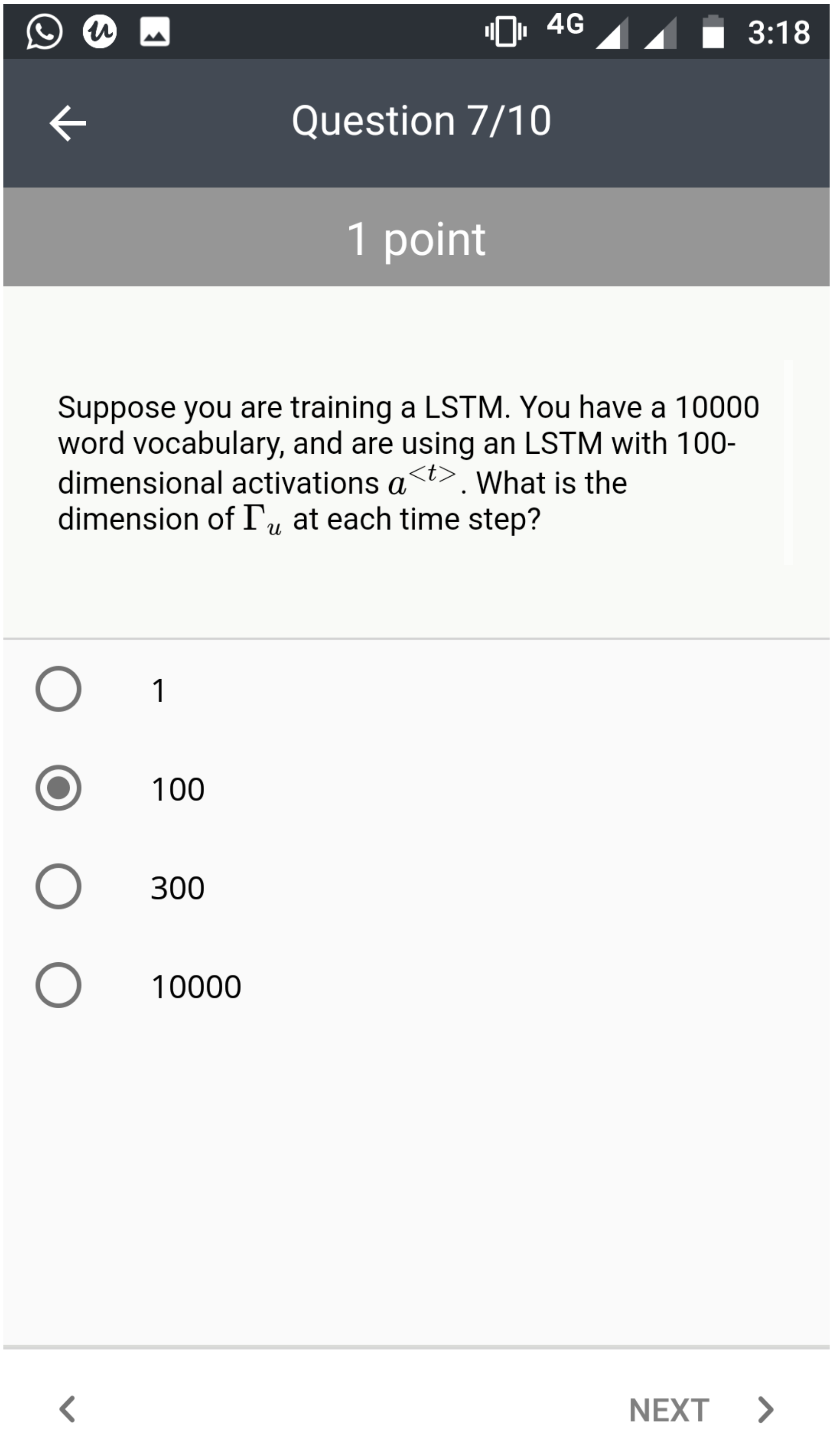
$$\Gamma_u = \sigma(W_u[\, a^{<t-1>}, x^{<t>}] + b_u)$$

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

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

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

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



Question 7/10

1 point

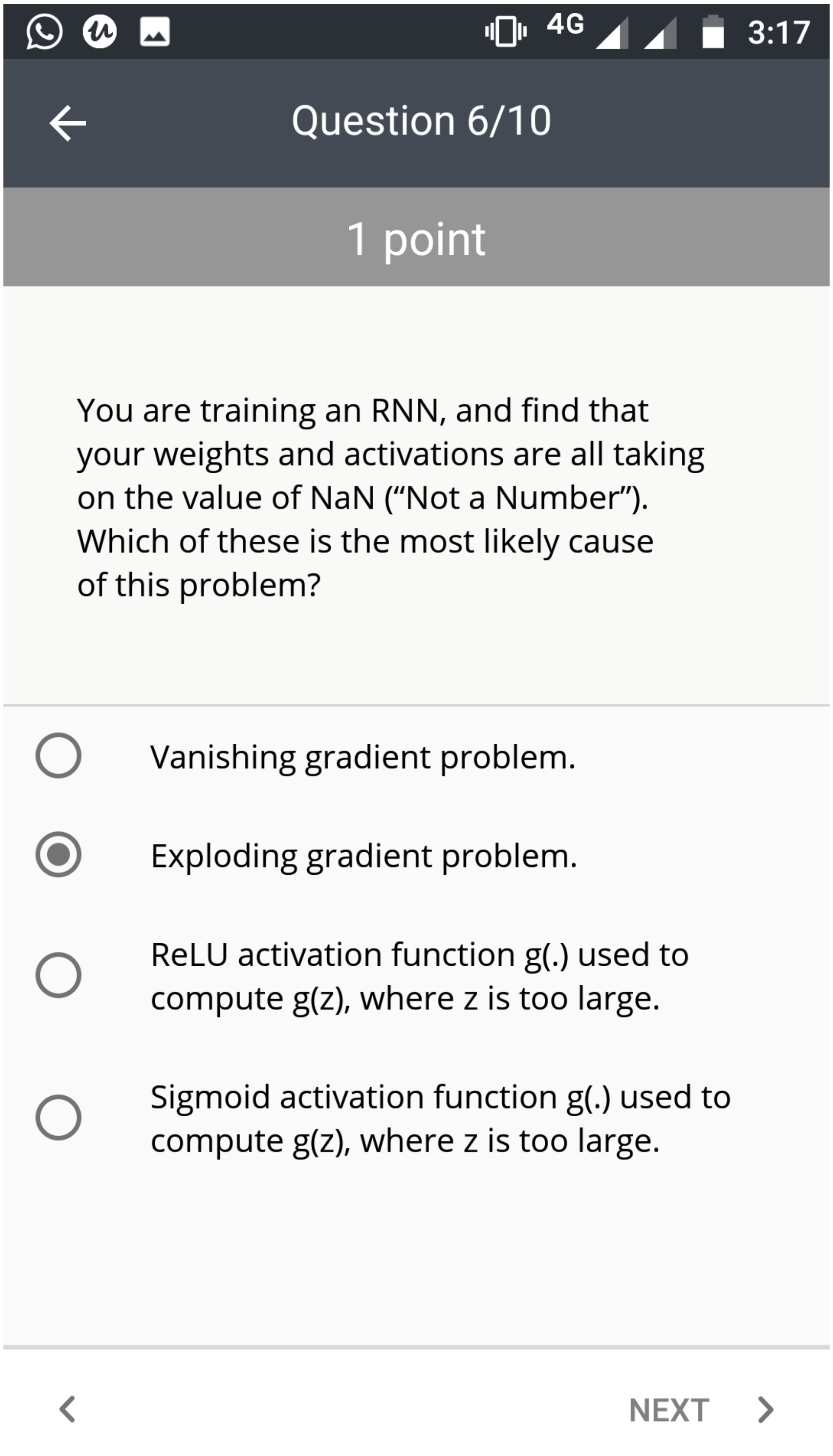
Suppose you are training a LSTM. You have a 10000 word vocabulary, and are using an LSTM with 100-dimensional activations $a^{<t>}$. What is the dimension of Γ_u at each time step?

○ 1

[✓ 100]

○ 300

○ 10000

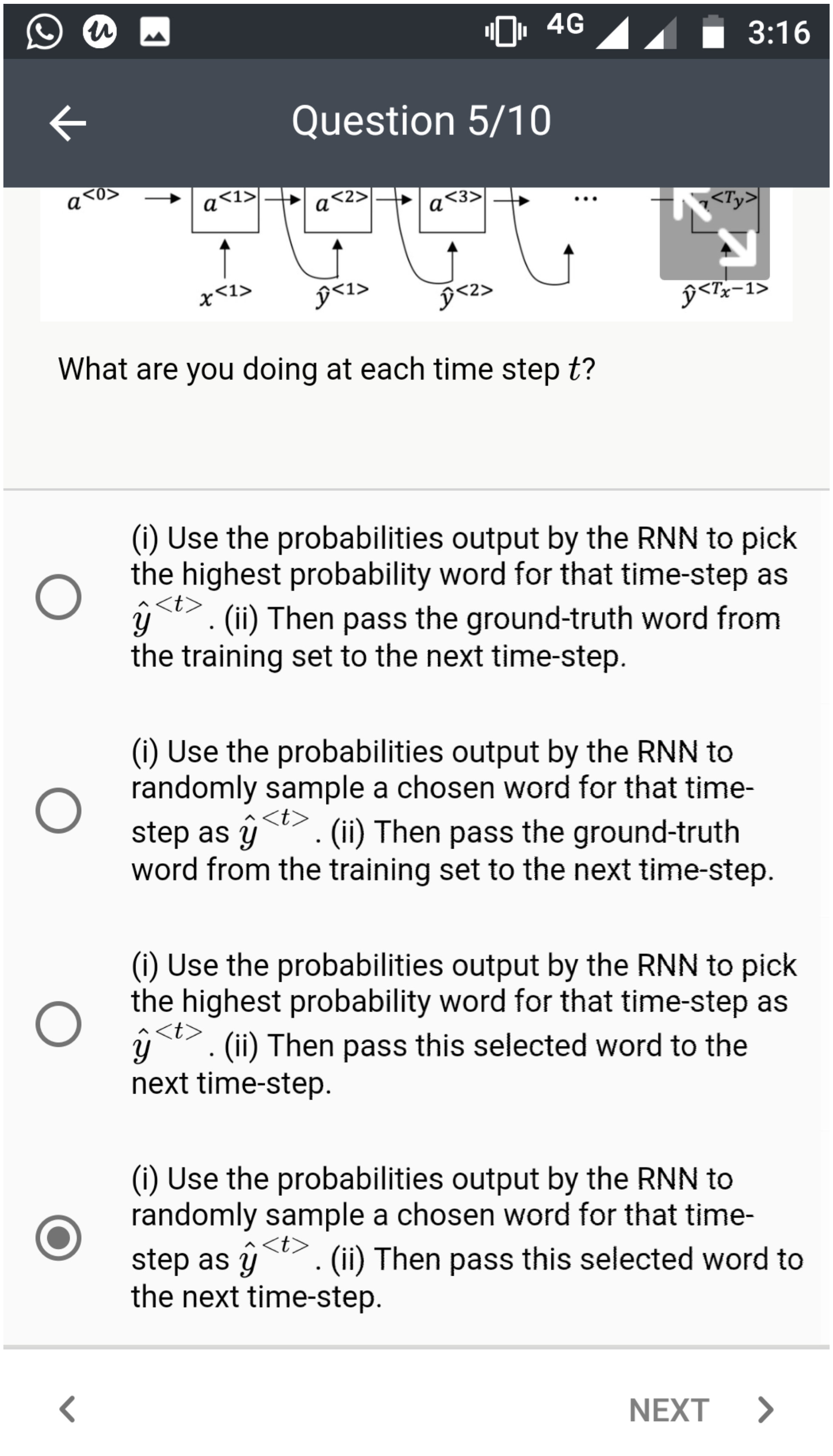


Question 6/10

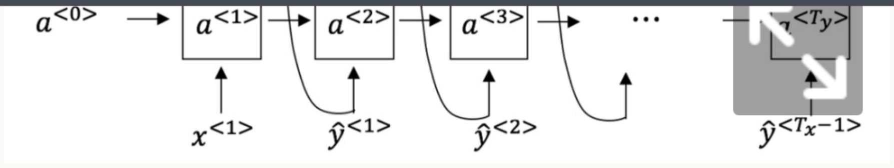
1 point

You are training an RNN, and find that your weights and activations are all taking on the value of NaN ("Not a Number"). Which of these is the most likely cause of this problem?

- ○ Vanishing gradient problem.
- [✓ Exploding gradient problem.]
- ○ ReLU activation function g(.) used to compute g(z), where z is too large.
- ○ Sigmoid activation function g(.) used to compute g(z), where z is too large.



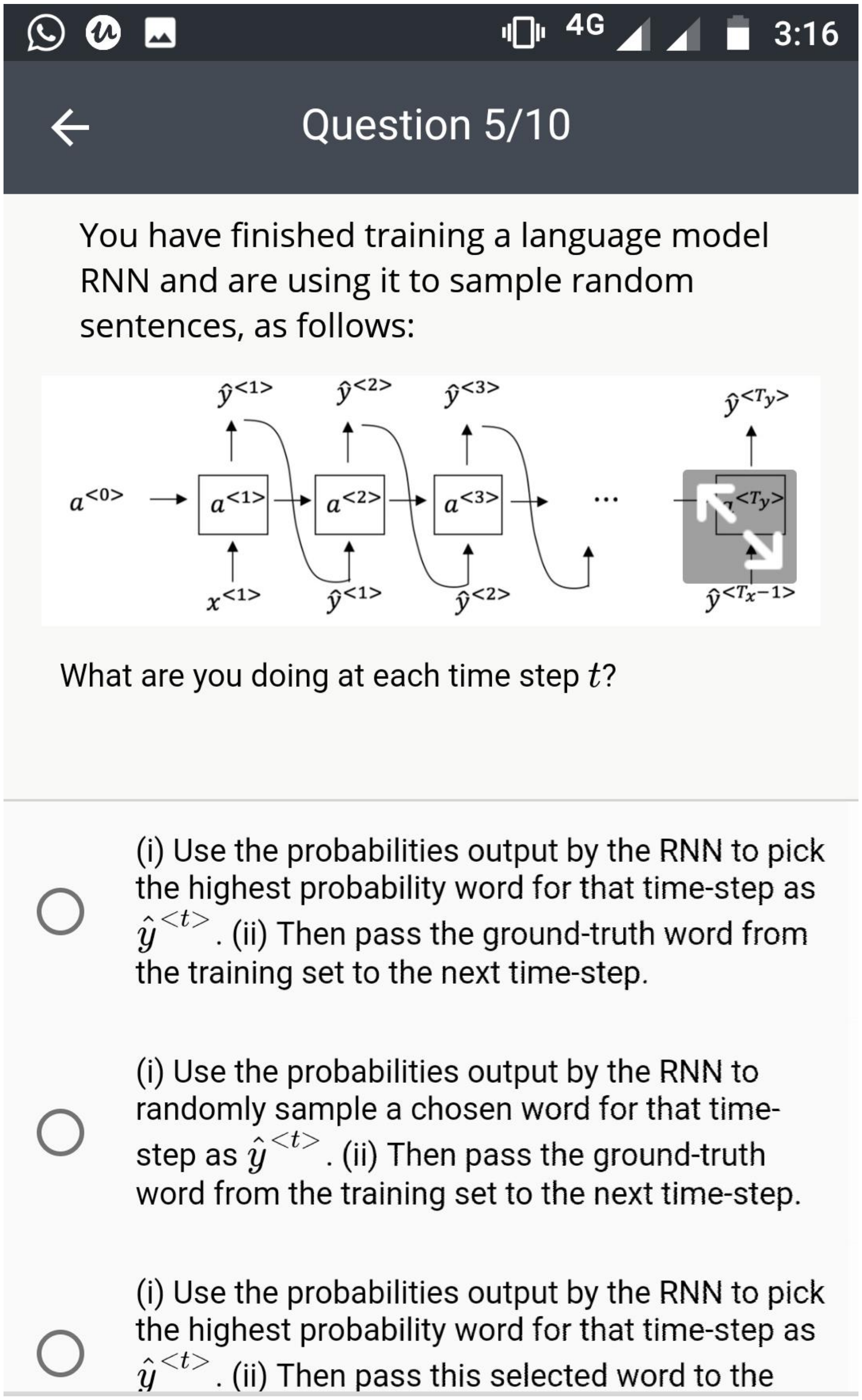
Question 5/10



What are you doing at each time step t?

- ○ (i) Use the probabilities output by the RNN to pick the highest probability word for that time-step as $\hat{y}^{<t>}$. (ii) Then pass the ground-truth word from the training set to the next time-step.
- ○ (i) Use the probabilities output by the RNN to randomly sample a chosen word for that time-step as $\hat{y}^{<t>}$. (ii) Then pass the ground-truth word from the training set to the next time-step.
- ○ (i) Use the probabilities output by the RNN to pick the highest probability word for that time-step as $\hat{y}^{<t>}$. (ii) Then pass this selected word to the next time-step.
- [✓ (i) Use the probabilities output by the RNN to randomly sample a chosen word for that time-step as $\hat{y}^{<t>}$. (ii) Then pass this selected word to the next time-step.]

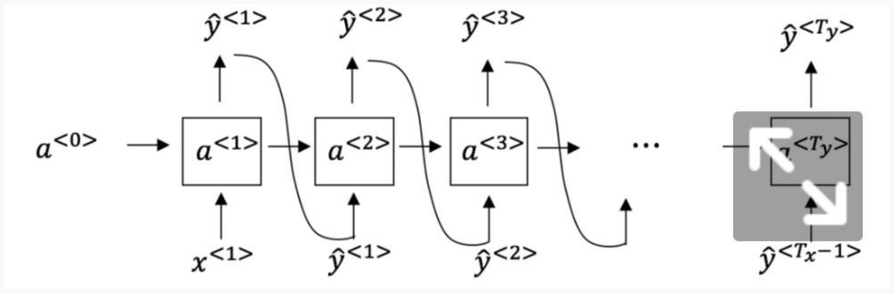
NEXT



3:16

Question 5/10

You have finished training a language model RNN and are using it to sample random sentences, as follows:



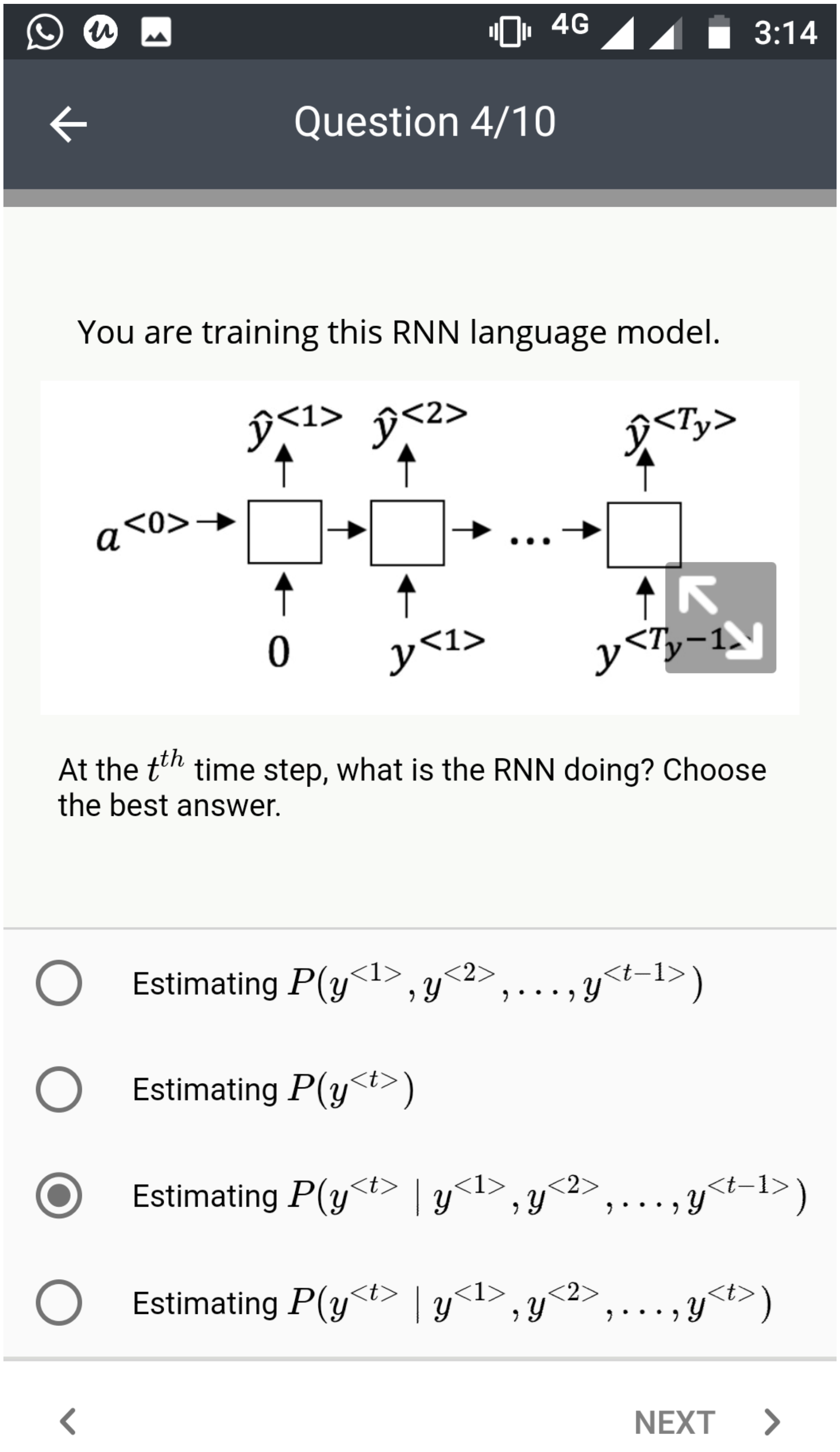
What are you doing at each time step t?

(i) Use the probabilities output by the RNN to pick the highest probability word for that time-step as $\hat{y}^{<t>}$. (ii) Then pass the ground-truth word from the training set to the next time-step.

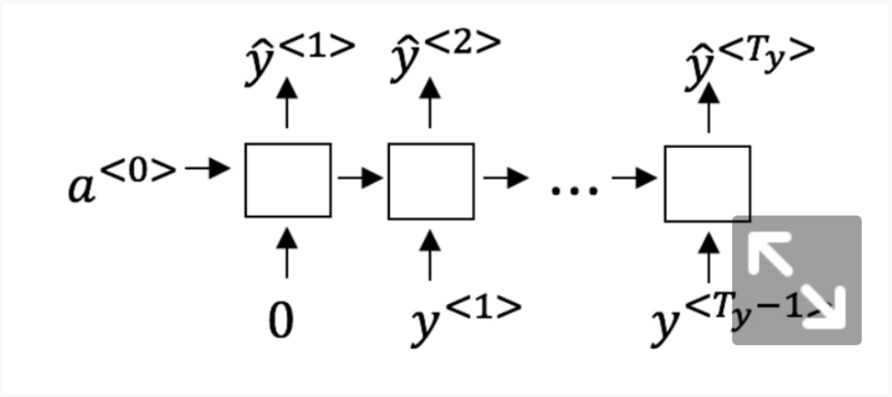
(i) Use the probabilities output by the RNN to randomly sample a chosen word for that time-step as $\hat{y}^{<t>}$. (ii) Then pass the ground-truth word from the training set to the next time-step.

(i) Use the probabilities output by the RNN to pick the highest probability word for that time-step as $\hat{y}^{<t>}$. (ii) Then pass this selected word to the

NEXT

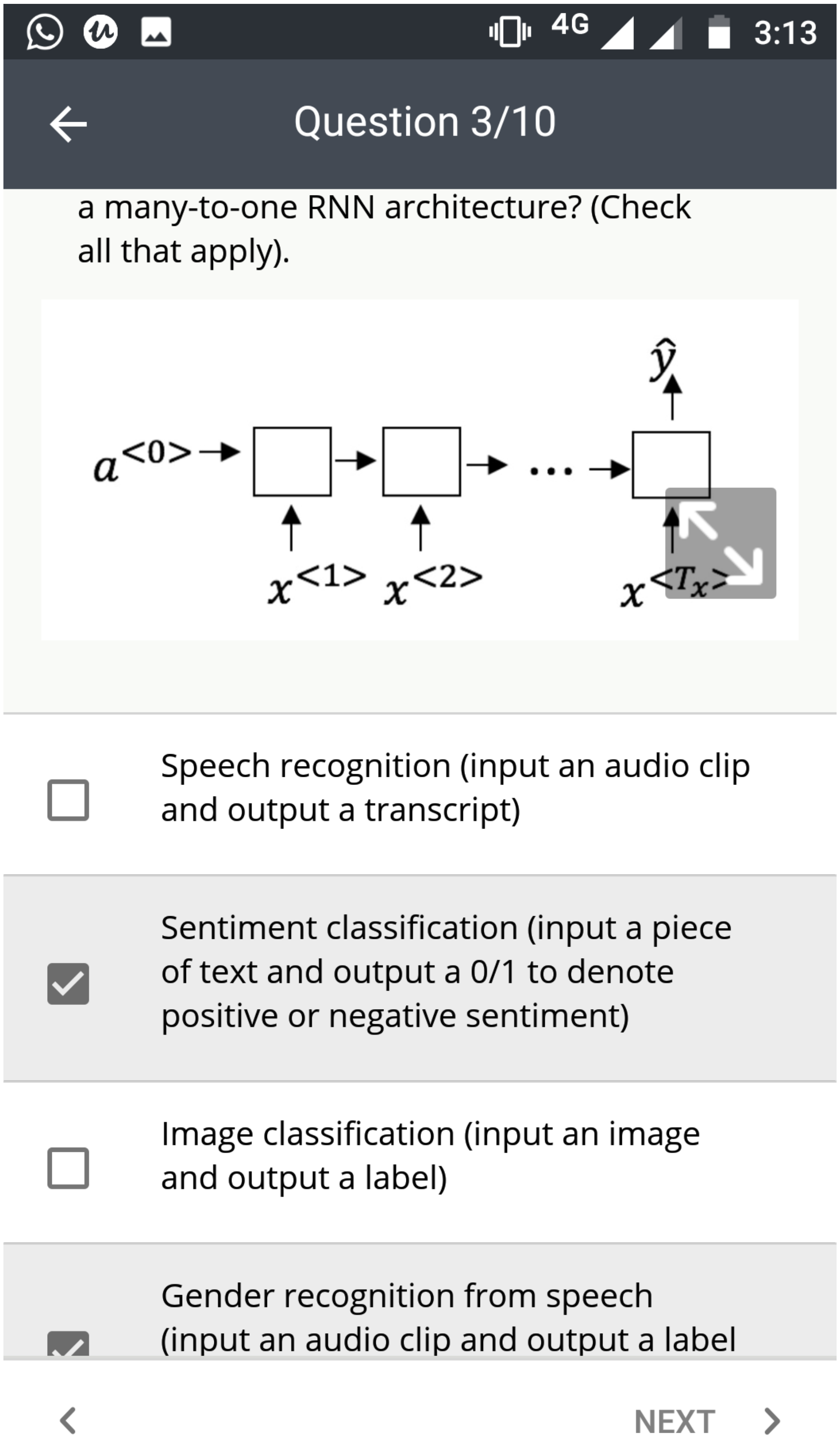


You are training this RNN language model.



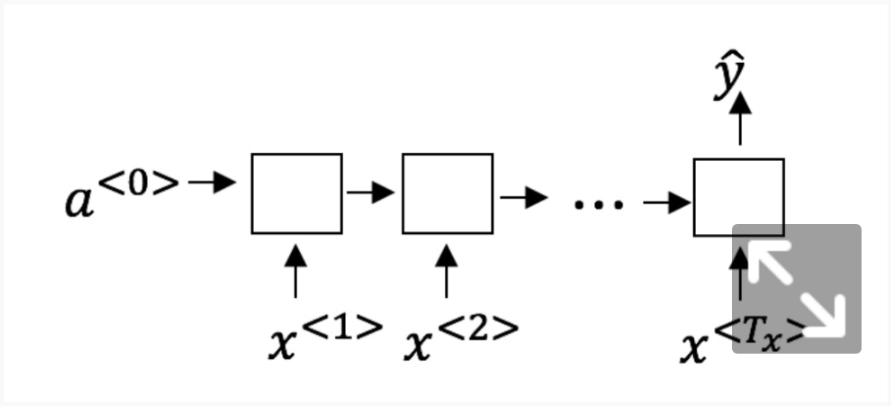
At the t^{th} time step, what is the RNN doing? Choose the best answer.

- Estimating $P(y^{<1>}, y^{<2>}, \dots, y^{<t-1>})$
- Estimating $P(y^{<t>})$
- [✓ Estimating $P(y^{<t>} \mid y^{<1>}, y^{<2>}, \dots, y^{<t-1>})$]
- Estimating $P(y^{<t>} \mid y^{<1>}, y^{<2>}, \dots, y^{<t>})$

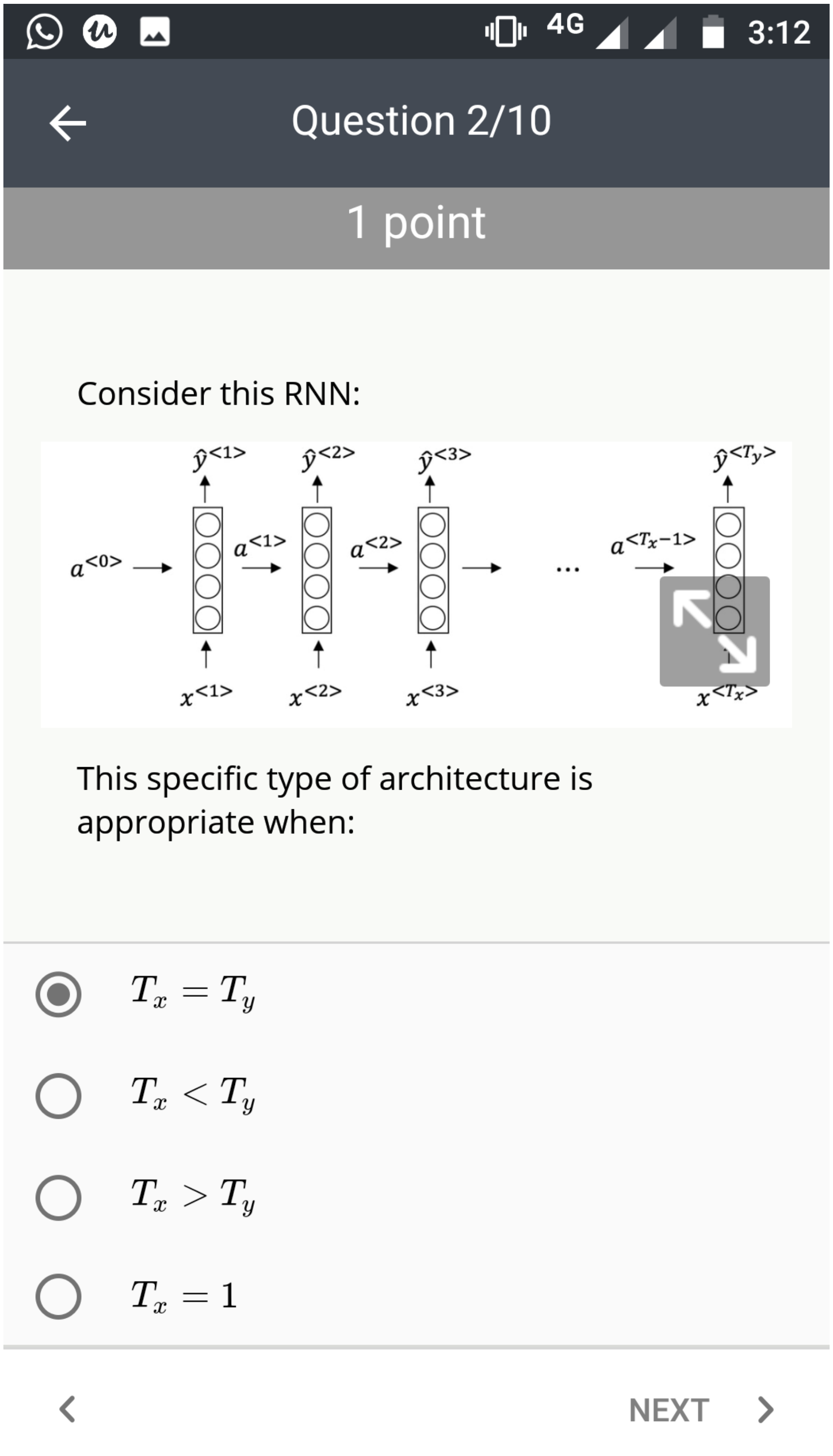


Question 3/10

a many-to-one RNN architecture? (Check all that apply).



- [ ] Speech recognition (input an audio clip and output a transcript)
- [✓] Sentiment classification (input a piece of text and output a 0/1 to denote positive or negative sentiment)
- [ ] Image classification (input an image and output a label)
- [✓] Gender recognition from speech (input an audio clip and output a label

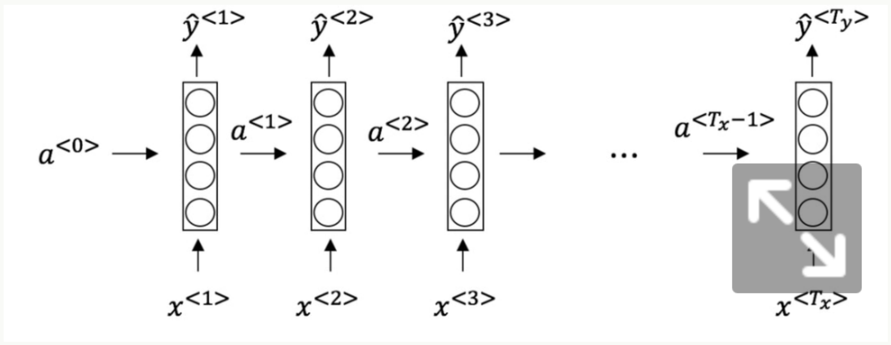


3:12

Question 2/10

1 point

Consider this RNN:



This specific type of architecture is appropriate when:

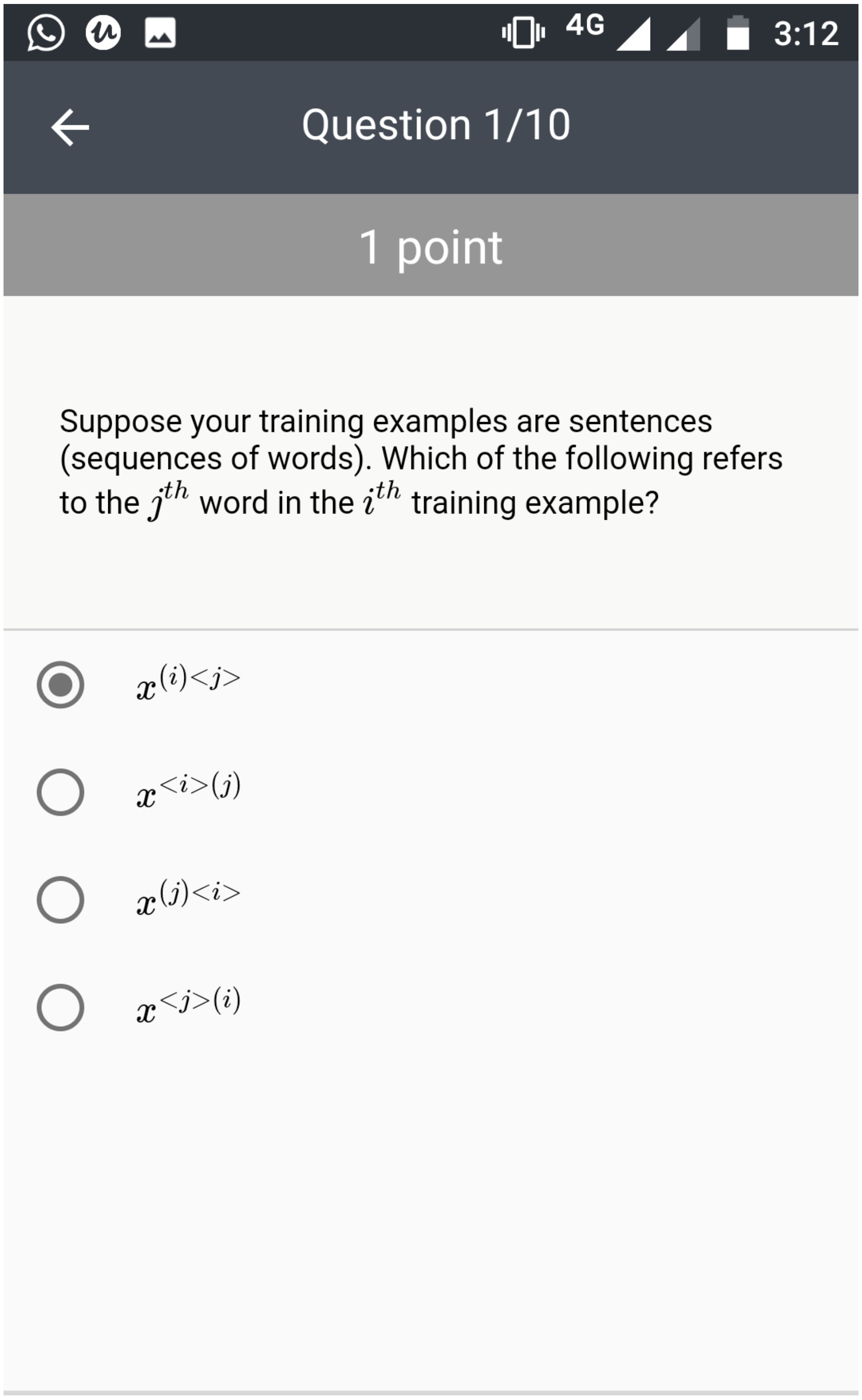
[✓ $T_x = T_y$]

$T_x < T_y$

$T_x > T_y$

$T_x = 1$

NEXT



Question 1/10

1 point

Suppose your training examples are sentences (sequences of words). Which of the following refers to the j^{th} word in the i^{th} training example?

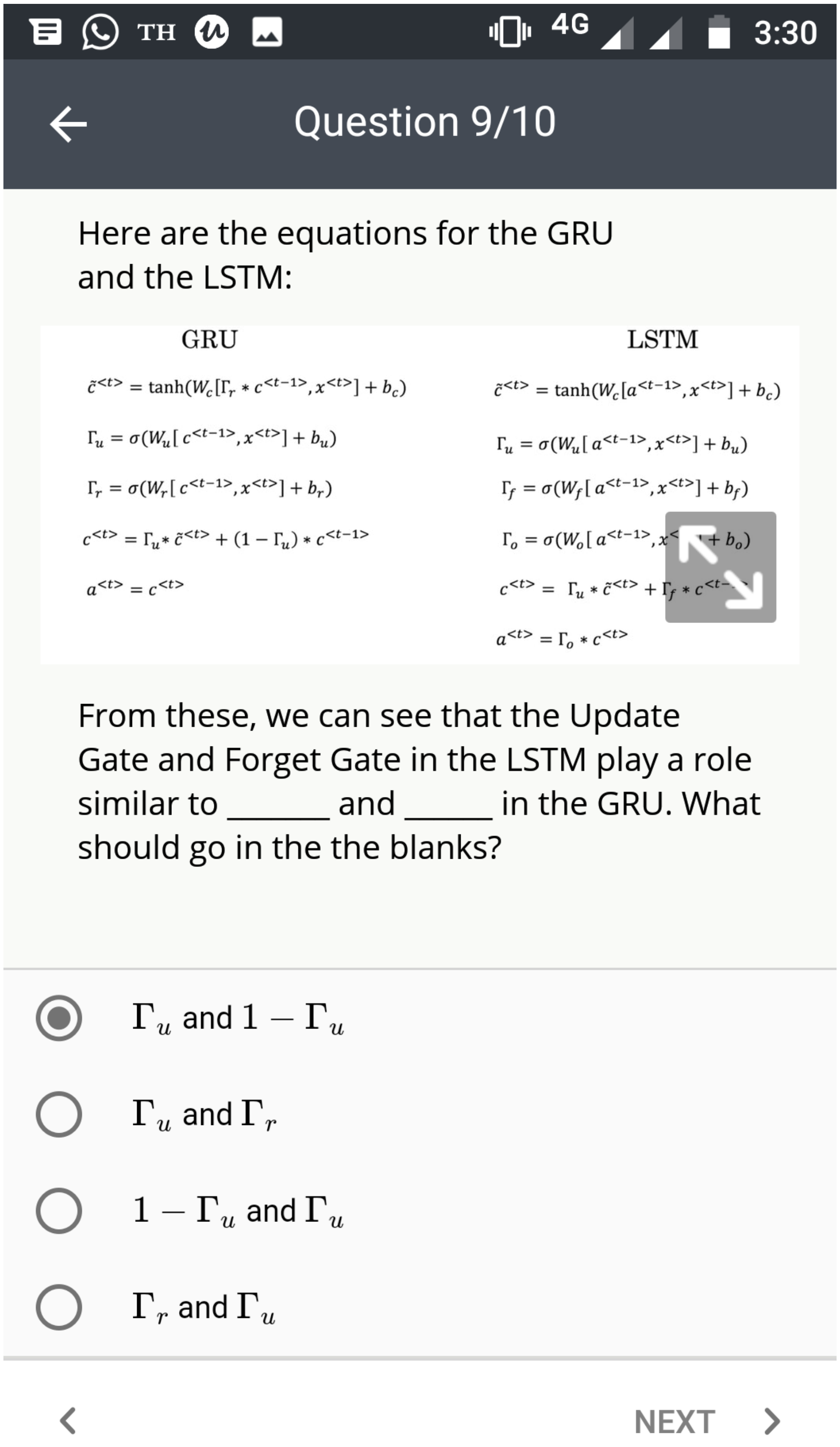
[✓ $x^{(i)<j>}$]

$x^{<i>(j)}$

$x^{(j)<i>}$

$x^{<j>(i)}$

NEXT



Question 9/10

Here are the equations for the GRU and the LSTM:

GRU	LSTM
$\tilde{c}^{<t>} = \tanh(W_c[\Gamma_r * c^{<t-1>}, x^{<t>}] + b_c)$	$\tilde{c}^{<t>} = \tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c)$
$\Gamma_u = \sigma(W_u[\,c^{<t-1>}, x^{<t>}] + b_u)$	$\Gamma_u = \sigma(W_u[\,a^{<t-1>}, x^{<t>}] + b_u)$
$\Gamma_r = \sigma(W_r[\,c^{<t-1>}, x^{<t>}] + b_r)$	$\Gamma_f = \sigma(W_f[\,a^{<t-1>}, x^{<t>}] + b_f)$
$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$	$\Gamma_o = \sigma(W_o[\,a^{<t-1>}, x^{<}$ [illegible] $+ b_o)$
$a^{<t>} = c^{<t>}$	$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-}$ [illegible]
	$a^{<t>} = \Gamma_o * c^{<t>}$

From these, we can see that the Update Gate and Forget Gate in the LSTM play a role similar to _______ and ______ in the GRU. What should go in the the blanks?

[✓ Γ_u and $1 - \Gamma_u$]

Γ_u and Γ_r

$1 - \Gamma_u$ and Γ_u

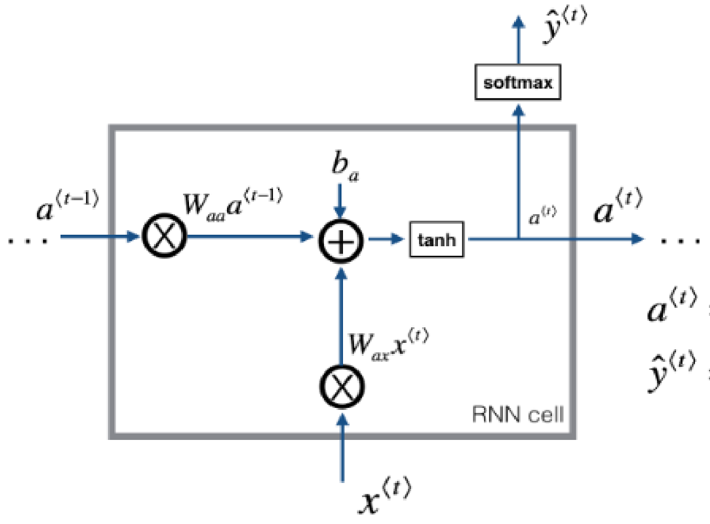
Γ_r and Γ_u

NEXT

Learnings from Assignments:

Assignment 1:

Simple RNN Graph and Equations


$\hat{y}^{\langle t \rangle}$
softmax
b_a
$a^{\langle t-1 \rangle}$
$W_{aa}a^{\langle t-1 \rangle}$
tanh
$a^{\langle t \rangle}$
$a^{\langle t \rangle}$
$W_{ax}x^{\langle t \rangle}$
RNN cell
$x^{\langle t \rangle}$

$$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a)$$
$$\hat{y}^{\langle t \rangle} = softmax(W_{ya}a^{\langle t \rangle} + b_y)$$

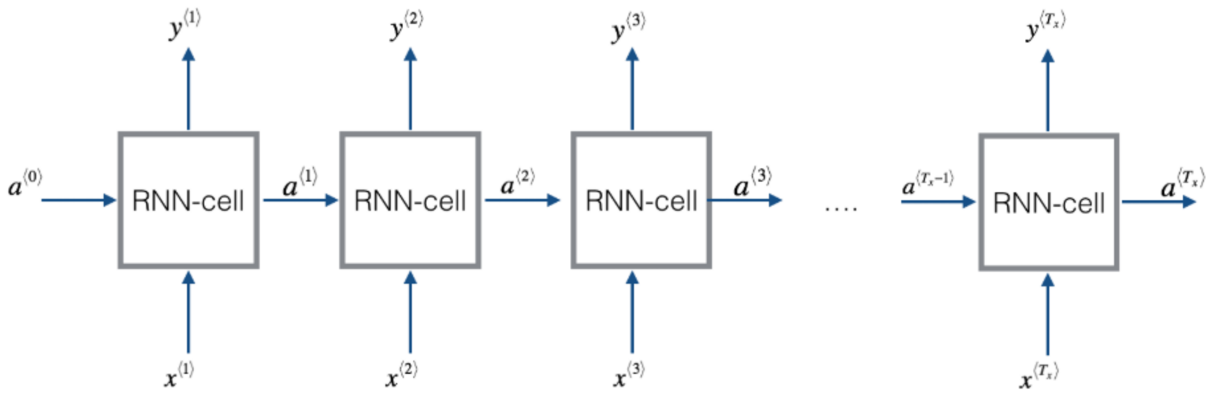
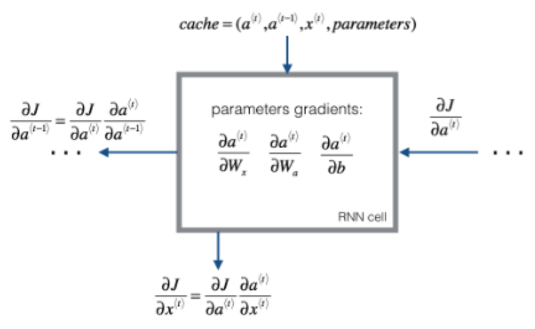


Figure 1: Basic RNN model

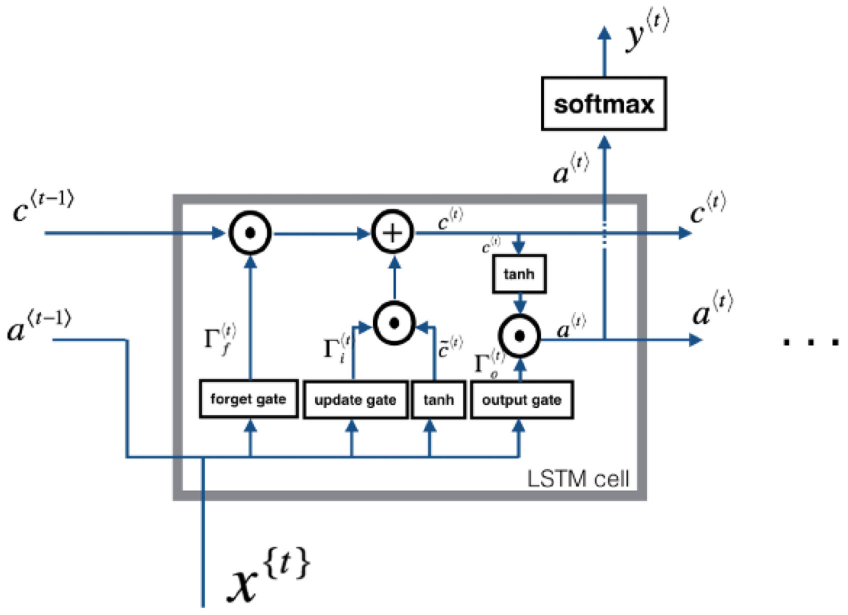
We will start by computing the backward pass for the basic RNN-cell.


$cache = (a^{\langle t \rangle}, a^{\langle t-1 \rangle}, x^{\langle t \rangle}, parameters)$
parameters gradients:
$\frac{\partial a^{\langle t \rangle}}{\partial W_x}$ $\frac{\partial a^{\langle t \rangle}}{\partial W_a}$ $\frac{\partial a^{\langle t \rangle}}{\partial b}$
RNN cell
$\frac{\partial J}{\partial a^{\langle t-1 \rangle}} = \frac{\partial J}{\partial a^{\langle t \rangle}} \frac{\partial a^{\langle t \rangle}}{\partial a^{\langle t-1 \rangle}}$
$\frac{\partial J}{\partial a^{\langle t \rangle}}$
$\frac{\partial J}{\partial x^{\langle t \rangle}} = \frac{\partial J}{\partial a^{\langle t \rangle}} \frac{\partial a^{\langle t \rangle}}{\partial x^{\langle t \rangle}}$

$$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b)$$
$$\frac{\partial \tanh(x)}{\partial x} = 1 - \tanh(x)^2$$
$$\frac{\partial a^{\langle t \rangle}}{\partial W_{ax}} = (1 - \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b)^2)x^{\langle t \rangle T}$$
$$\frac{\partial a^{\langle t \rangle}}{\partial W_{aa}} = (1 - \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b)^2)a^{\langle t-1 \rangle T}$$
$$\frac{\partial a^{\langle t \rangle}}{\partial b} = \sum_{batch} (1 - \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b)^2)$$
$$\frac{\partial a^{\langle t \rangle}}{\partial x^{\langle t \rangle}} = W_{ax}{}^T.(1 - \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b)^2)$$
$$\frac{\partial a^{\langle t \rangle}}{\partial a^{\langle t-1 \rangle}} = W_{aa}{}^T.(1 - \tanh(W_{ax}x^{\langle t-1 \rangle} + W_{aa}a^{\langle t-1 \rangle} + b)^2)$$

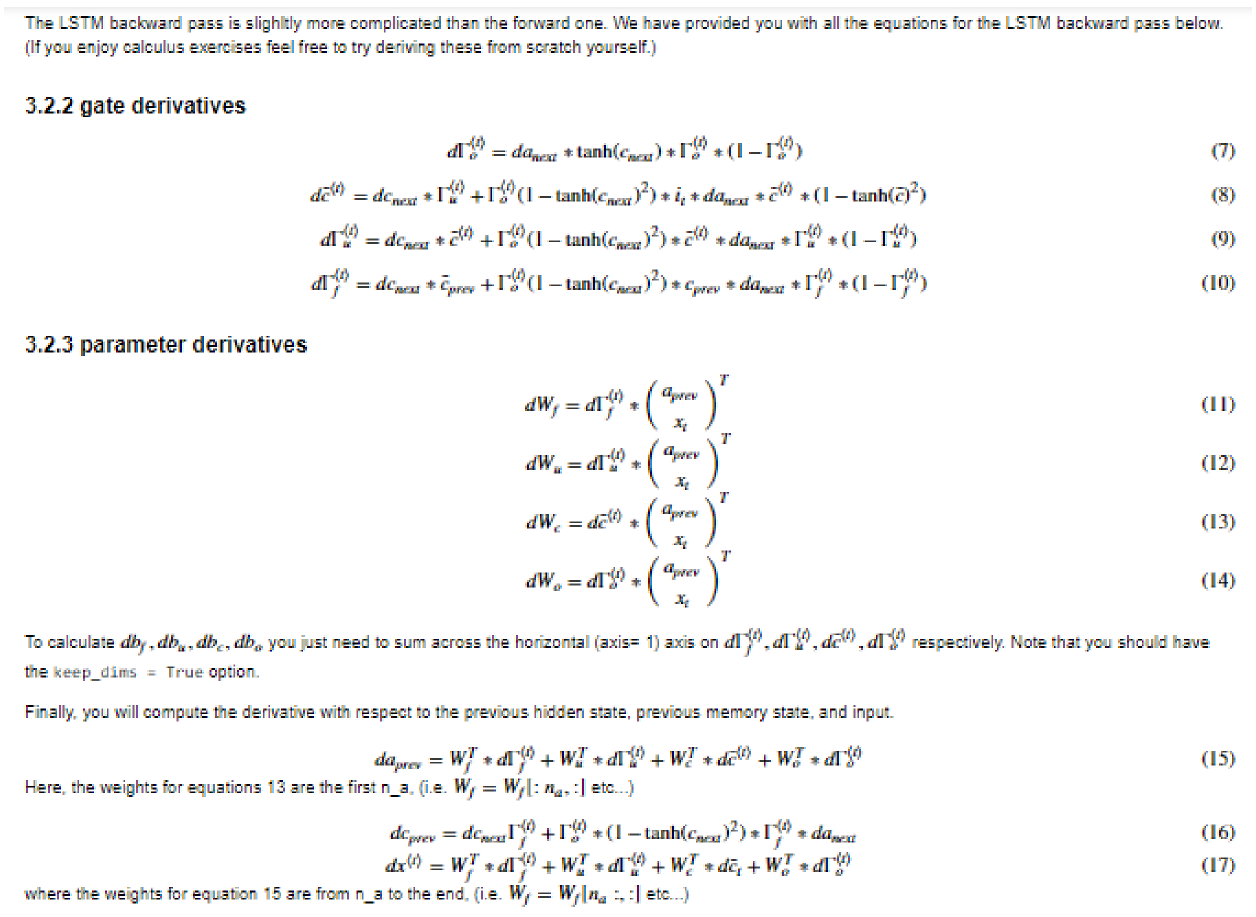
Figure 5: RNN-cell's backward pass. Just like in a fully-connected neural network, the derivative of the cost function J backpropagates through the RNN by following the chain-rule from calculas. The chain-rule is also used to calculate $(\frac{\partial J}{\partial W_{ax}}, \frac{\partial J}{\partial W_{aa}}, \frac{\partial J}{\partial b})$ to update the parameters (W_{ax}, W_{aa}, b_a).

LSTM Cell Equations


$y^{\langle t \rangle}$
softmax
$a^{\langle t \rangle}$
$c^{\langle t-1 \rangle}$
$c^{\langle t \rangle}$
$a^{\langle t-1 \rangle}$
$a^{\langle t \rangle}$
tanh
$\Gamma_f^{\langle t \rangle}$
$\Gamma_i^{\langle t \rangle}$
$\tilde{c}^{\langle t \rangle}$
$\Gamma_o^{\langle t \rangle}$
forget gate
update gate
tanh
output gate
LSTM cell
$x^{\{t\}}$

$$\Gamma_f^{\langle t \rangle} = \sigma(W_f[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_f)$$
$$\Gamma_u^{\langle t \rangle} = \sigma(W_u[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_u)$$
$$\tilde{c}^{\langle t \rangle} = \tanh(W_C[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_C$$
$$c^{\langle t \rangle} = \Gamma_f^{\langle t \rangle} \circ c^{\langle t-1 \rangle} + \Gamma_u^{\langle t \rangle} \circ \tilde{c}^{\langle t \rangle}$$
$$\Gamma_o^{\langle t \rangle} = \sigma(W_o[a^{\langle t-1 \rangle}, x^{\langle t \rangle}] + b_o)$$
$$a^{\langle t \rangle} = \Gamma_o^{\langle t \rangle} \circ \tanh(c^{\langle t \rangle})$$

Figure 4: LSTM-cell. This tracks and updates a "cell state" or memory variable $c^{\langle t \rangle}$ at every time-step, which can be different from $a^{\langle t \rangle}$.



The LSTM backward pass is slighltly more complicated than the forward one. We have provided you with all the equations for the LSTM backward pass below. (If you enjoy calculus exercises feel free to try deriving these from scratch yourself.)

3.2.2 gate derivatives

$$d\Gamma_o^{\langle t\rangle} = da_{next} * \tanh(c_{next}) * \Gamma_o^{\langle t\rangle} * (1 - \Gamma_o^{\langle t\rangle}) \tag{7}$$

$$d\tilde{c}^{\langle t\rangle} = dc_{next} * \Gamma_u^{\langle t\rangle} + \Gamma_o^{\langle t\rangle}(1 - \tanh(c_{next})^2) * i_t * da_{next} * \tilde{c}^{\langle t\rangle} * (1 - \tanh(\tilde{c})^2) \tag{8}$$

$$d\Gamma_u^{\langle t\rangle} = dc_{next} * \tilde{c}^{\langle t\rangle} + \Gamma_o^{\langle t\rangle}(1 - \tanh(c_{next})^2) * \tilde{c}^{\langle t\rangle} * da_{next} * \Gamma_u^{\langle t\rangle} * (1 - \Gamma_u^{\langle t\rangle}) \tag{9}$$

$$d\Gamma_f^{\langle t\rangle} = dc_{next} * \tilde{c}_{prev} + \Gamma_o^{\langle t\rangle}(1 - \tanh(c_{next})^2) * c_{prev} * da_{next} * \Gamma_f^{\langle t\rangle} * (1 - \Gamma_f^{\langle t\rangle}) \tag{10}$$

3.2.3 parameter derivatives

$$dW_f = d\Gamma_f^{\langle t\rangle} * \begin{pmatrix} a_{prev} \\ x_t \end{pmatrix}^T \tag{11}$$

$$dW_u = d\Gamma_u^{\langle t\rangle} * \begin{pmatrix} a_{prev} \\ x_t \end{pmatrix}^T \tag{12}$$

$$dW_c = d\tilde{c}^{\langle t\rangle} * \begin{pmatrix} a_{prev} \\ x_t \end{pmatrix}^T \tag{13}$$

$$dW_o = d\Gamma_o^{\langle t\rangle} * \begin{pmatrix} a_{prev} \\ x_t \end{pmatrix}^T \tag{14}$$

To calculate db_f, db_u, db_c, db_o you just need to sum across the horizontal (axis= 1) axis on $d\Gamma_f^{\langle t\rangle}, d\Gamma_u^{\langle t\rangle}, d\tilde{c}^{\langle t\rangle}, d\Gamma_o^{\langle t\rangle}$ respectively. Note that you should have the `keep_dims = True` option.

Finally, you will compute the derivative with respect to the previous hidden state, previous memory state, and input.

$$da_{prev} = W_f^T * d\Gamma_f^{\langle t\rangle} + W_u^T * d\Gamma_u^{\langle t\rangle} + W_c^T * d\tilde{c}^{\langle t\rangle} + W_o^T * d\Gamma_o^{\langle t\rangle} \tag{15}$$

Here, the weights for equations 13 are the first n_a, (i.e. $W_f = W_f[:n_a,:]$ etc...)

$$dc_{prev} = dc_{next}\Gamma_f^{\langle t\rangle} + \Gamma_o^{\langle t\rangle} * (1 - \tanh(c_{next})^2) * \Gamma_f^{\langle t\rangle} * da_{next} \tag{16}$$

$$dx^{\langle t\rangle} = W_f^T * d\Gamma_f^{\langle t\rangle} + W_u^T * d\Gamma_u^{\langle t\rangle} + W_c^T * d\tilde{c}_t + W_o^T * d\Gamma_o^{\langle t\rangle} \tag{17}$$

where the weights for equation 15 are from n_a to the end, (i.e. $W_f = W_f[n_a:,:]$ etc...)

Assignment 2:

How to use RNN for building Character Level Language (Generation) Models

Simple RNN Model Steps:

1.2 - Overview of the model

Your model will have the following structure:

- Initialize parameters
- Run the optimization loop
  - Forward propagation to compute the loss function
  - Backward propagation to compute the gradients with respect to the loss function
  - Clip the gradients to avoid exploding gradients
  - Using the gradients, update your parameter with the gradient descent update rule.
- Return the learned parameters

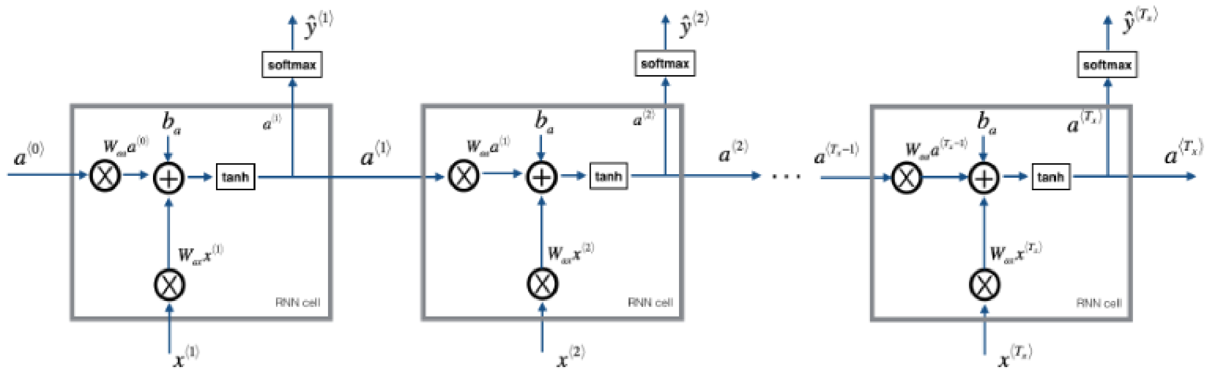

$\hat{y}^{\langle 1 \rangle}$
$\hat{y}^{\langle 2 \rangle}$
$\hat{y}^{\langle T_x \rangle}$
softmax
softmax
softmax
$a^{\langle 0 \rangle}$
$W_{aa}a^{\langle 0 \rangle}$
b_a
$a^{\langle 1 \rangle}$
tanh
$W_{ax}x^{\langle 1 \rangle}$
RNN cell
$x^{\langle 1 \rangle}$
$a^{\langle 1 \rangle}$
$W_{aa}a^{\langle 1 \rangle}$
b_a
$a^{\langle 2 \rangle}$
tanh
$W_{ax}x^{\langle 2 \rangle}$
RNN cell
$x^{\langle 2 \rangle}$
$a^{\langle 2 \rangle}$
$a^{\langle T_x - 1 \rangle}$
$W_{aa}a^{\langle T_x - 1 \rangle}$
b_a
$a^{\langle T_x \rangle}$
tanh
$W_{ax}x^{\langle T_x \rangle}$
RNN cell
$x^{\langle T_x \rangle}$
$a^{\langle T_x \rangle}$

Figure 1: Recurrent Neural Network, similar to what you had built in the previous notebook "Building a RNN - Step by Step".

At each time-step, the RNN tries to predict what is the next character given the previous characters. The dataset $X = (x^{\langle 1 \rangle}, x^{\langle 2 \rangle}, \ldots, x^{\langle T_x \rangle})$ is a list of characters in the training set, while $Y = (y^{\langle 1 \rangle}, y^{\langle 2 \rangle}, \ldots, y^{\langle T_x \rangle})$ is such that at every time-step t, we have $y^{\langle t \rangle} = x^{\langle t+1 \rangle}$.

Two Key steps:

1. Gradient Clipping

2.1 - Clipping the gradients in the optimization loop

In this section you will implement the `clip` function that you will call inside of your optimization loop. Recall that your overall loop structure usually consists of a forward pass, a cost computation, a backward pass, and a parameter update. Before updating the parameters, you will perform gradient clipping when needed to make sure that your gradients are not "exploding," meaning taking on overly large values.

In the exercise below, you will implement a function `clip` that takes in a dictionary of gradients and returns a clipped version of gradients if needed. There are different ways to clip gradients; we will use a simple element-wise clipping procedure, in which every element of the gradient vector is clipped to lie between some range [-N, N]. More generally, you will provide a `maxValue` (say 10). In this example, if any component of the gradient vector is greater than 10, it would be set to 10; and if any component of the gradient vector is less than -10, it would be set to -10. If it is between -10 and 10, it is left alone.

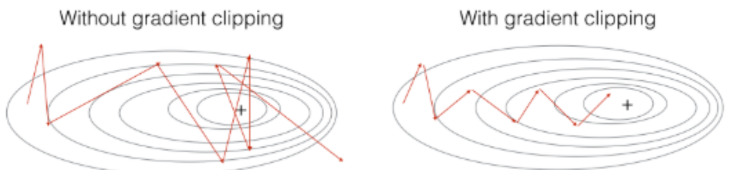
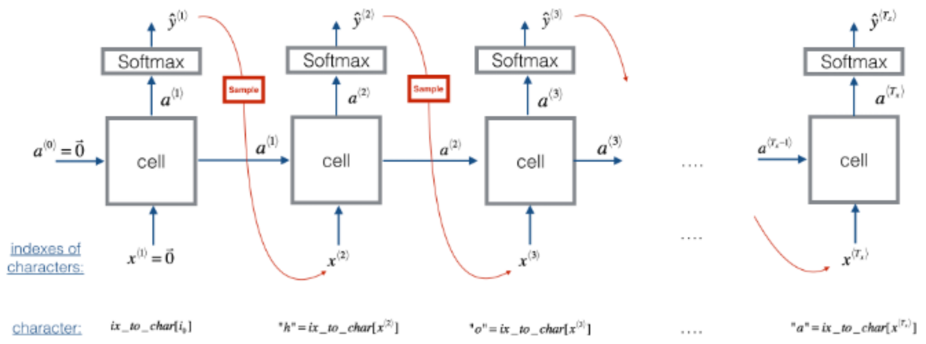


Figure 2: Visualization of gradient descent with and without gradient clipping, in a case where the network is running into slight "exploding gradient" problems.

2. Sampling (after we assume the model is built)

Now assume that your model is trained. You would like to generate new text (characters). The process of generation is explained in the picture below:



Softmax
Sample
cell
indexes of characters:
character:
$x^{\langle 1 \rangle} = \vec{0}$
$a^{\langle 0 \rangle} = \vec{0}$
$ix\_to\_char[i_0]$
"h" = $ix\_to\_char[x^{\langle 2 \rangle}]$
"o" = $ix\_to\_char[x^{\langle 3 \rangle}]$
"a" = $ix\_to\_char[x^{\langle T_x \rangle}]$

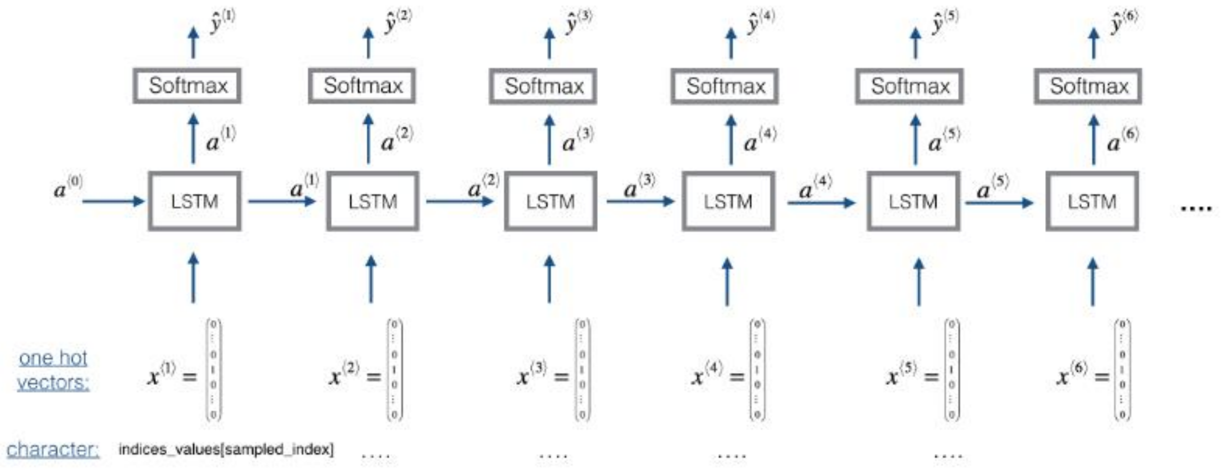
Figure 3: In this picture, we assume the model is already trained. We pass in $x^{\langle 1 \rangle} = \vec{0}$ at the first time step, and have the network then sample one character at a time.

Assignment 3:

Music Generation using LSTM

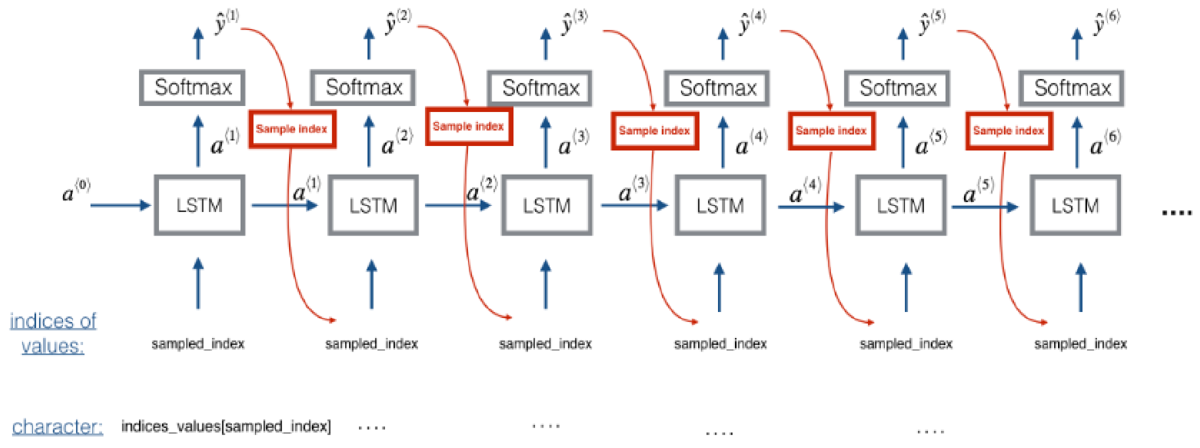
Training the Model:

$$x^{\langle i+1 \rangle} = y^{\langle i \rangle}$$



How to generate music with a trained model:

3.1 - Predicting & Sampling



$\hat{y}^{\langle 1 \rangle}$
$\hat{y}^{\langle 2 \rangle}$
$\hat{y}^{\langle 3 \rangle}$
$\hat{y}^{\langle 4 \rangle}$
$\hat{y}^{\langle 5 \rangle}$
$\hat{y}^{\langle 6 \rangle}$
Softmax
Sample index
$a^{\langle 0 \rangle}$
$a^{\langle 1 \rangle}$
$a^{\langle 2 \rangle}$
$a^{\langle 3 \rangle}$
$a^{\langle 4 \rangle}$
$a^{\langle 5 \rangle}$
$a^{\langle 6 \rangle}$
LSTM
....
indices of values:
sampled_index
character:
indices_values[sampled_index]
....