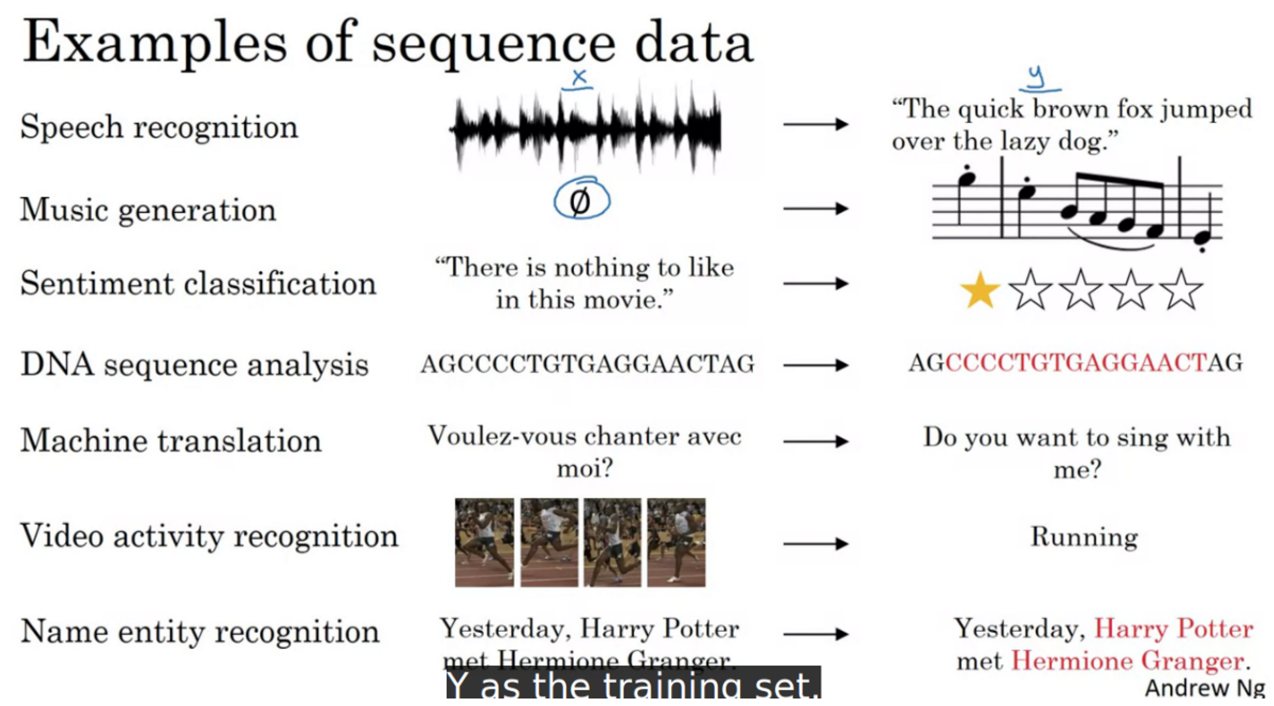
Sequence Model — Week 01

aakashgoel12.medium.com/sequence-model-week-01-ed99d8417ef6

February 21, 2021

Aakash Goel

·Just now



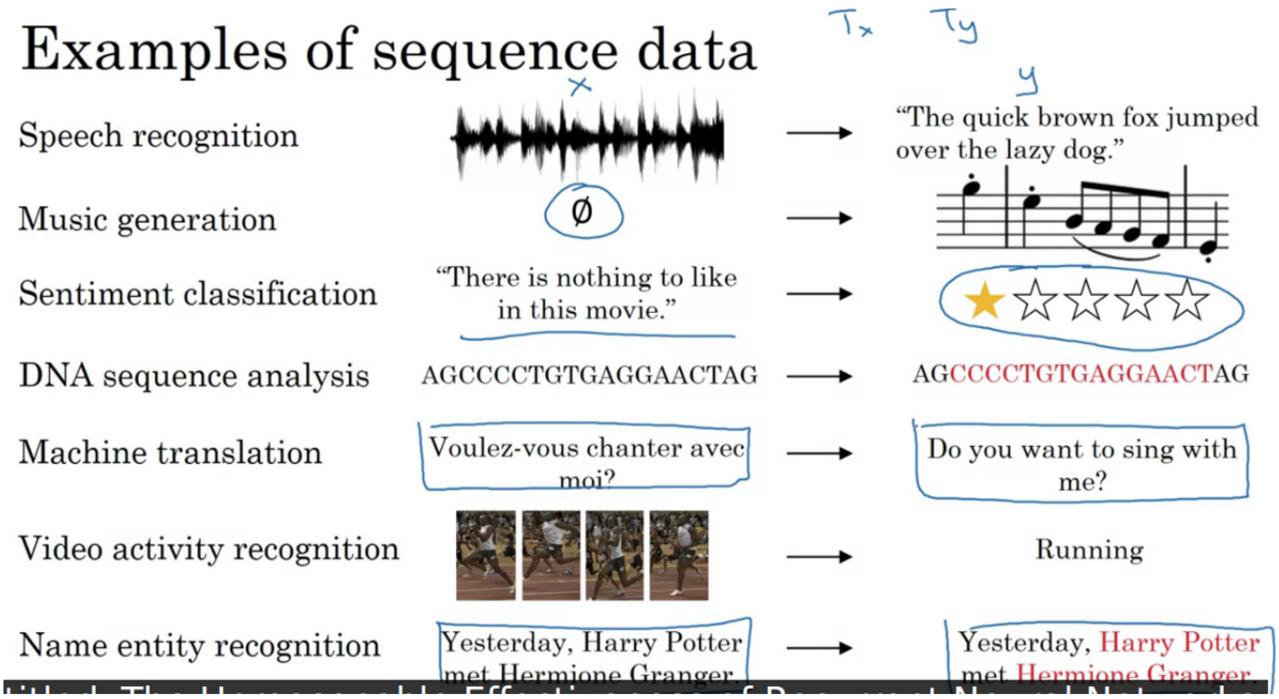
Examples of sequence data

Speech recognition	x (audio) →	y "The quick brown fox jumped over the lazy dog."
Music generation	$\emptyset$ →	(music)
Sentiment classification	"There is nothing to like in this movie." →	★☆☆☆☆
DNA sequence analysis	AGCCCCTGTGAGGAACTAG →	AGCCCCTGTGAGGAACTAG
Machine translation	Voulez-vous chanter avec moi? →	Do you want to sing with me?
Video activity recognition	(video frames) →	Running
Name entity recognition	Yesterday, Harry Potter met Hermione Granger. →	Yesterday, Harry Potter met Hermione Granger.

Y as the training set.

Andrew Ng

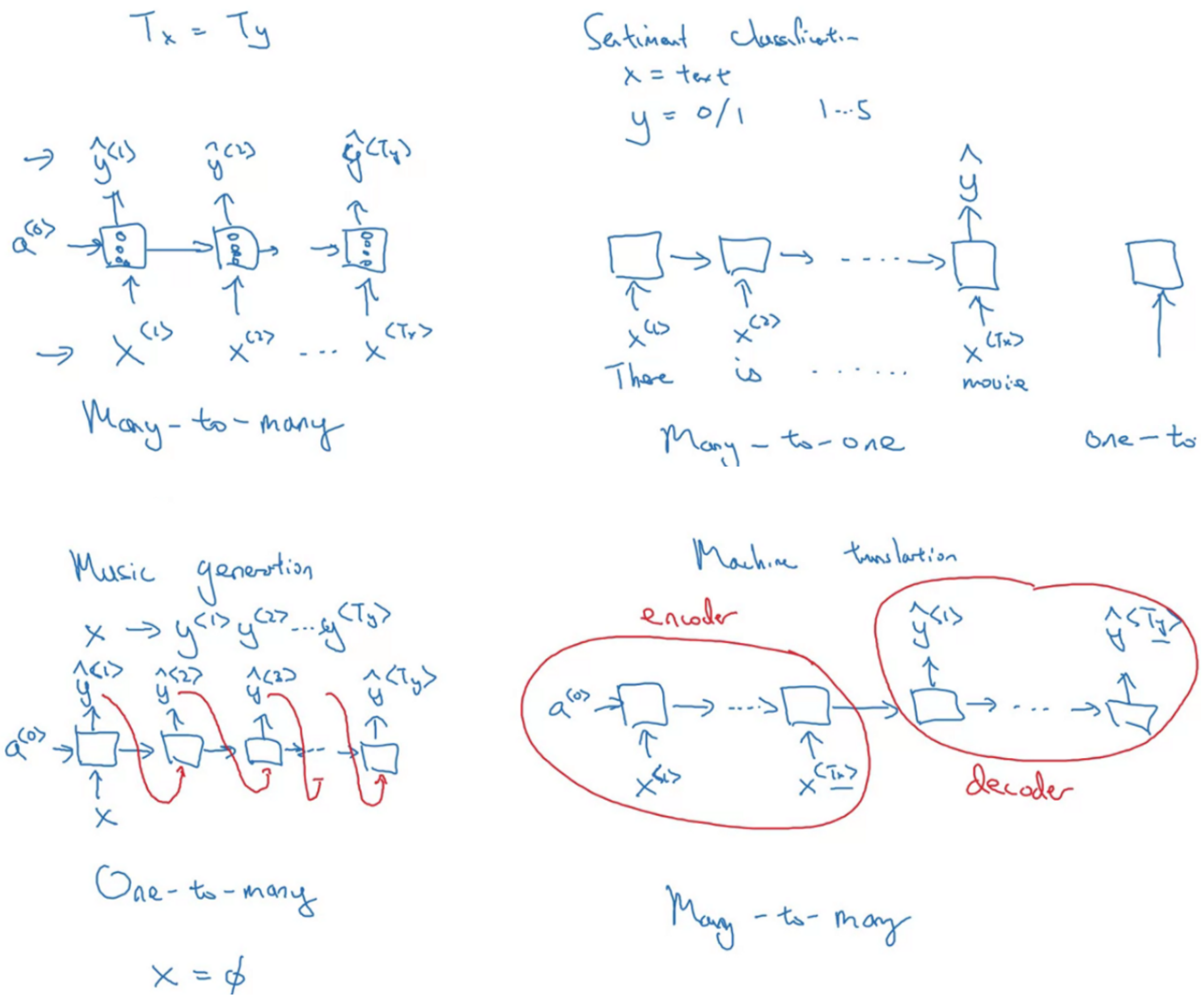
Different Types of RNN



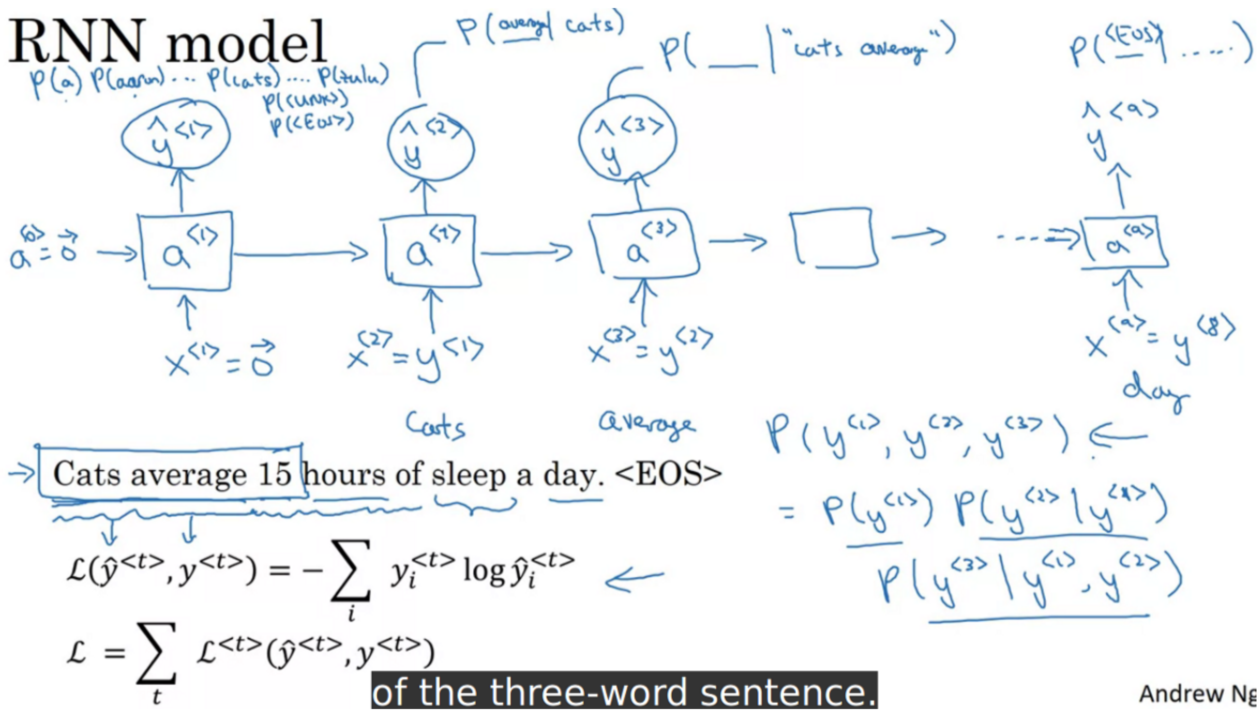
Examples of sequence data

T_x T_y

Speech recognition	x (audio) →	y "The quick brown fox jumped over the lazy dog."
Music generation	$\emptyset$ →	(music)
Sentiment classification	"There is nothing to like in this movie." →	★☆☆☆☆
DNA sequence analysis	AGCCCCTGTGAGGAACTAG →	AGCCCCTGTGAGGAACTAG
Machine translation	Voulez-vous chanter avec moi? →	Do you want to sing with me?
Video activity recognition	(video frames) →	Running
Name entity recognition	Yesterday, Harry Potter met Hermione Granger. →	Yesterday, Harry Potter met Hermione Granger.



Language Modelling → Cost Function



RNN model

Cats average 15 hours of sleep a day. <EOS>

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

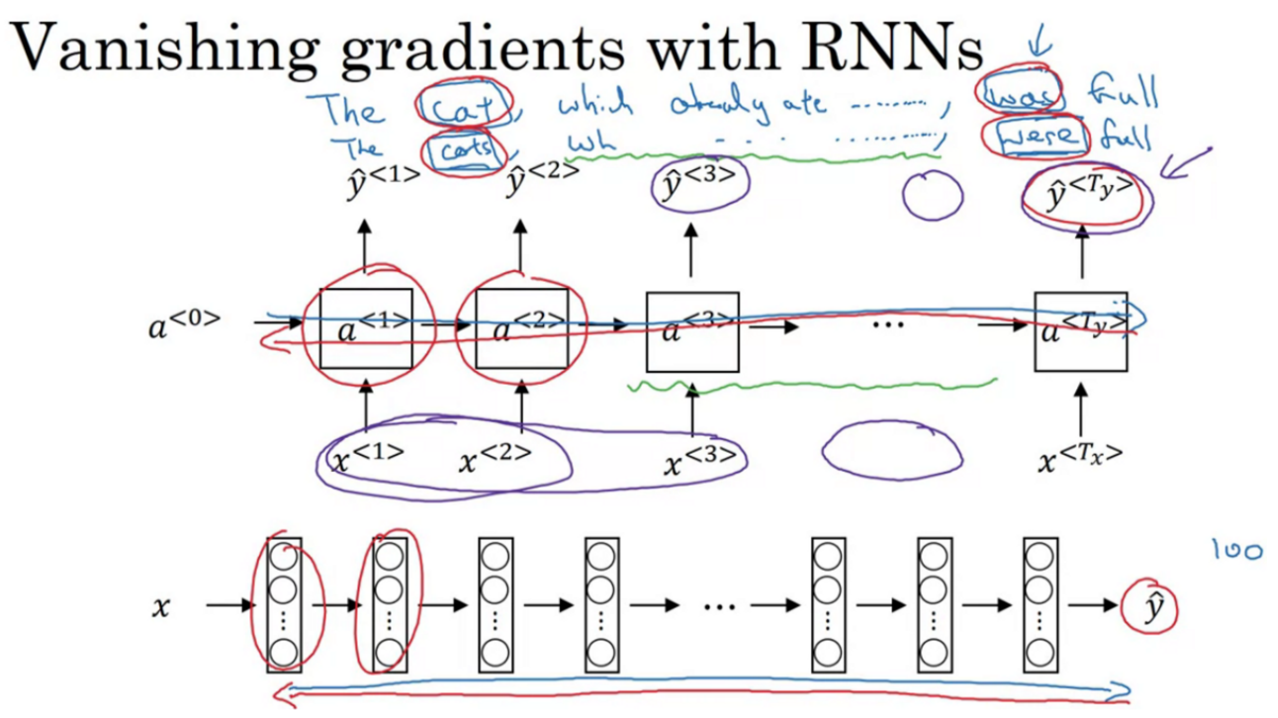
$$\mathcal{L} = \sum_t \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

$$P(y^{<1>}, y^{<2>}, y^{<3>}) = P(y^{<1>})\, P(y^{<2>} \mid y^{<1>})\, P(y^{<3>} \mid y^{<1>}, y^{<2>})$$

of the three-word sentence.

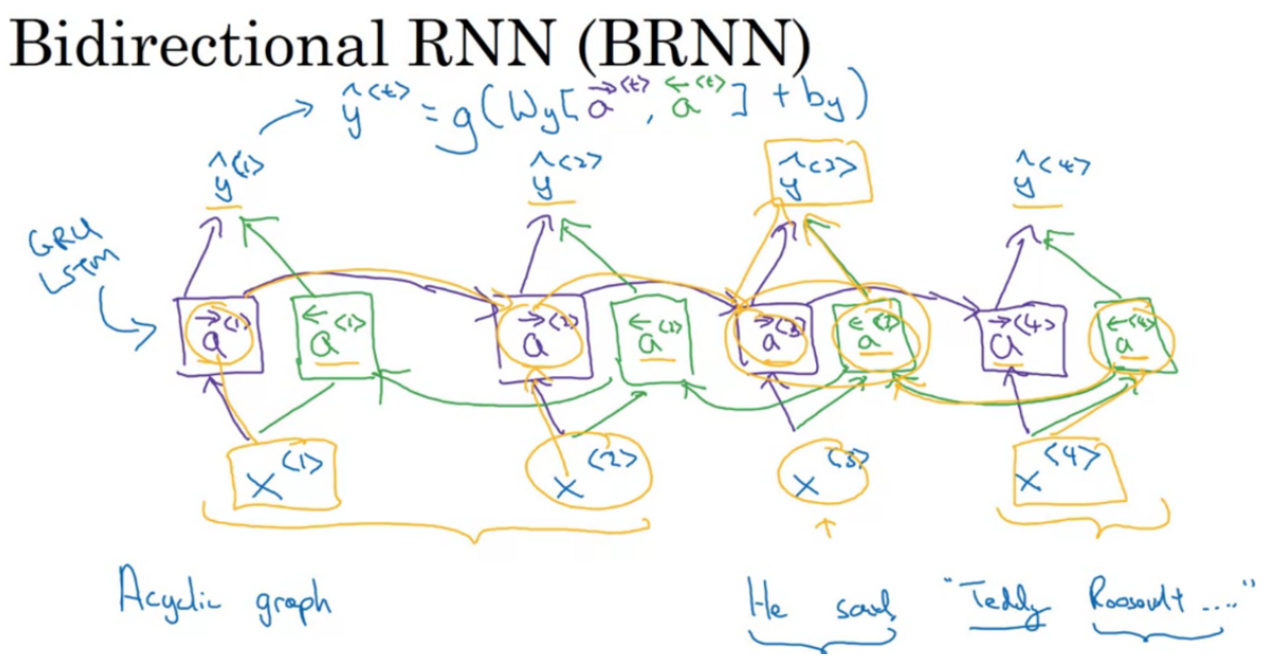
Andrew Ng

RNN Model


Vanishing gradients with RNNs
The cat, which already ate ...., was full
The cats, wh . . . ., were full
$\hat{y}^{<1>}$ $\hat{y}^{<2>}$ $\hat{y}^{<3>}$ $\hat{y}^{<T_y>}$
$a^{<0>}$ $a^{<1>}$ $a^{<2>}$ $a^{<3>}$ $a^{<T_y>}$
$x^{<1>}$ $x^{<2>}$ $x^{<3>}$ $x^{<T_x>}$
x $\hat{y}$
100

Exploding Gradients are easy to capture as parameters just blow up and you might often see NaNs (Not a numbers → results of numerical overflow, in Neural network computation) → Apply Gradient Clipping i.e. Look at Gradient Vectors and if it is bigger than some threshold, re-scale some of your gradient vector so that is not too big.

Bi-directional RNN


Bidirectional RNN (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 ..."

LSTM

Peephole Connections → Gate Values may depend not just on a_t-1 and x_t but also on previous memory cell value

QUIZ

1. 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?

[✓ $x^{(i)<j>}$]

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

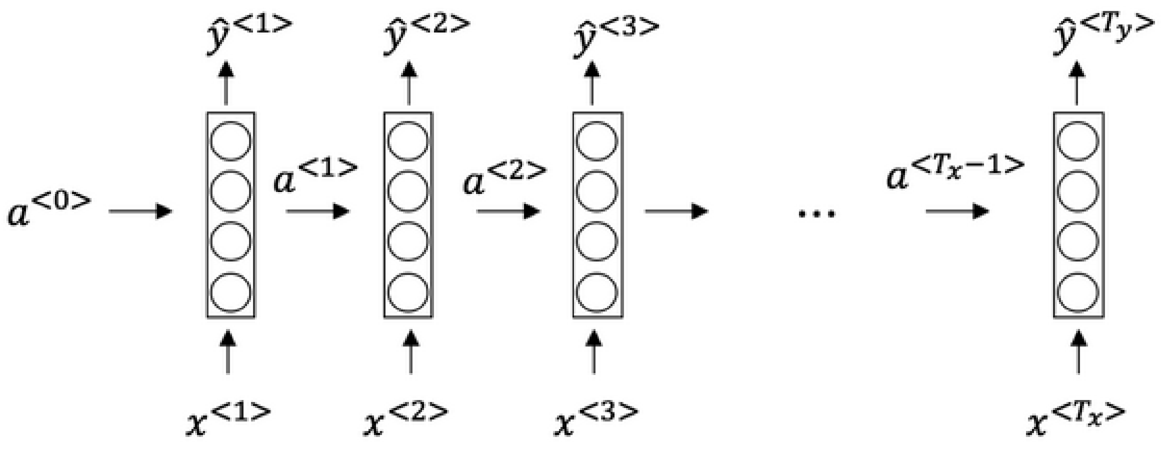
$x^{(j)<i>}$

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

✓ **Correct**

We index into the i^{th} row first to get the i^{th} training example (represented by parentheses), then the j^{th} column to get the j^{th} word (represented by the brackets).

2. Consider this RNN:



This specific type of architecture is appropriate when:

[✓ $T_x = T_y$]

$T_x < T_y$

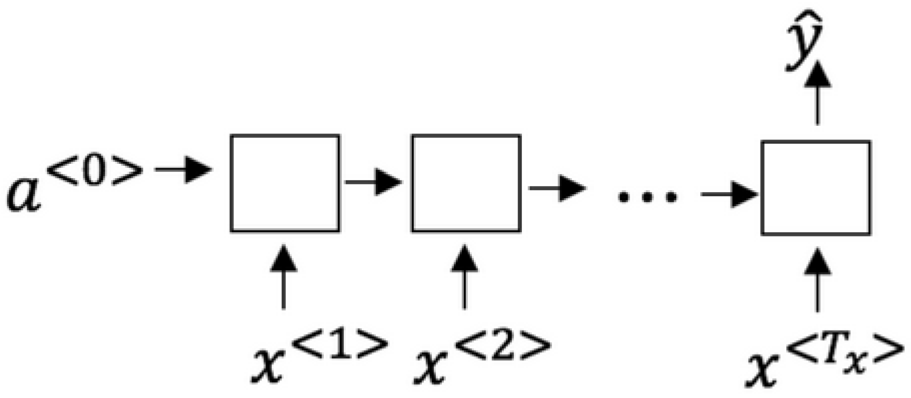
$T_x > T_y$

$T_x = 1$

✓ **Correct**

It is appropriate when every input should be matched to an output.

3. To which of these tasks would you apply a many-to-one RNN architecture? (Check all that apply).



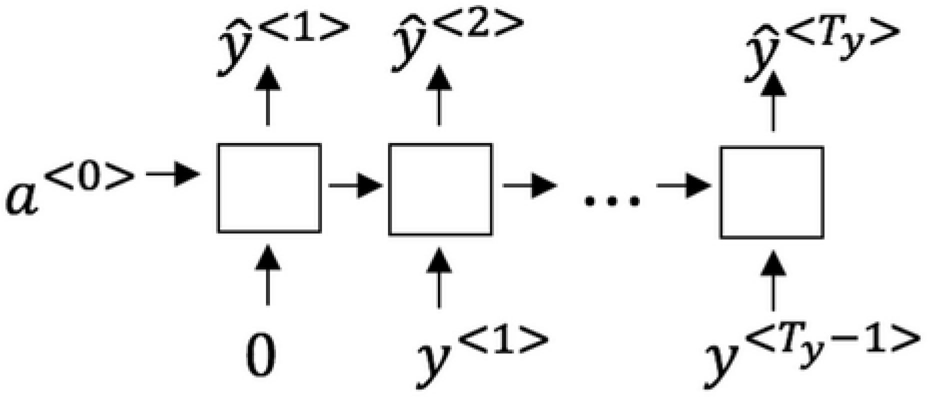
- [ ] Speech recognition (input an audio clip and output a transcript)
- [x] Sentiment classification (input a piece of text and output a 0/1 to denote positive or negative sentiment)

✓ **Correct**
Correct!

- [ ] Image classification (input an image and output a label)
- [x] Gender recognition from speech (input an audio clip and output a label indicating the speaker's gender)

✓ **Correct**
Correct!

4. You are training this RNN language model.

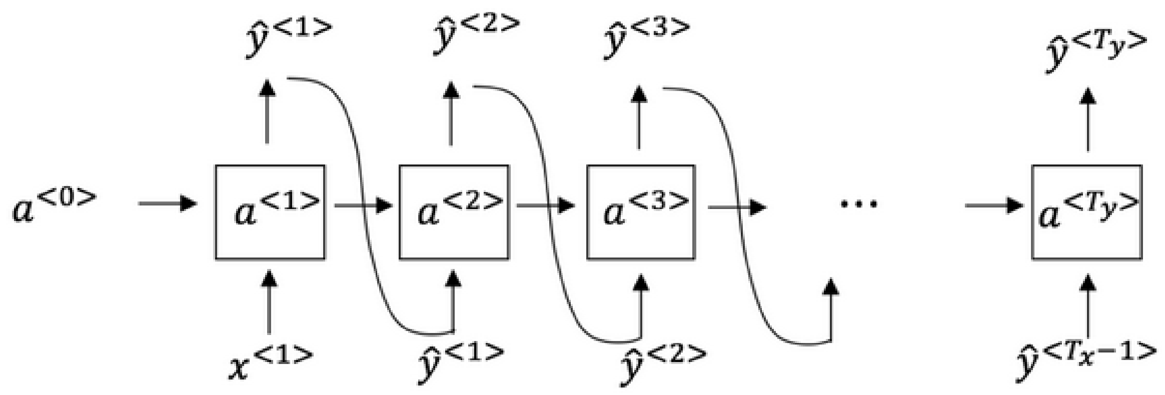


At the t^{th} time step, what is the RNN doing? Choose the best answer.

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

✓ **Correct**
Yes, in a language model we try to predict the next step based on the knowledge of all prior steps.

5. 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 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.]

✓ **Correct**

Yes!

6. 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.

✓ **Correct**

7. 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

✓ **Correct**

Correct, Γ_u is a vector of dimension equal to the number of hidden units in the LSTM.

8. Here're the update equations for the GRU.

GRU

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

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

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

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

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

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_r), because if $\Gamma_u \approx 1$ for a timestep, the gradient can propagate back through that timestep without much decay.

✓ **Correct**
Yes. For the signal to backpropagate without vanishing, we need $c^{<t>}$ to be highly dependant on $c^{<t-1>}$.

For the signal to backpropagate without vanishing, we need $c^{<t>}$ to be highly dependant on $c^{<t-1>}$.

9. Here are the equations for the GRU and the LSTM:

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

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

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

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

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

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

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

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

✓ **Correct**

Yes, correct!

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 $x^{<1>}, \ldots, x^{<365>}$. You've also collected data on your dog's mood, which you represent as $y^{<1>}, \ldots, y^{<365>}$. 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 $x^{<1>}, \ldots, x^{<t>}$, but not on $x^{<t+1>}, \ldots, x^{<365>}$]
- Unidirectional RNN, because the value of $y^{<t>}$ depends only on $x^{<t>}$, and not other days' weather.

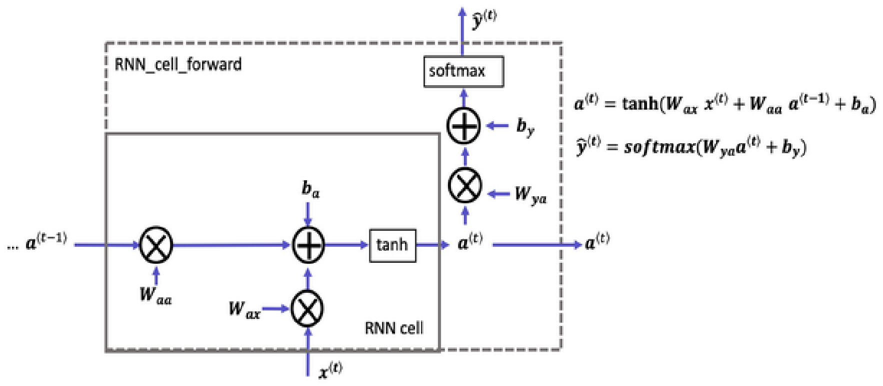
✓ **Correct**

Yes!

Notebook Exercise

1.1 - RNN cell

A recurrent neural network can be seen as the repeated use of a single cell. You are first going to implement the computations for a single time-step. The following figure describes the operations for a single time-step of an RNN cell.



RNN_cell_forward
softmax
$\hat{y}^{\langle t \rangle}$
b_y
W_{ya}
b_a
tanh
$a^{\langle t \rangle}$
... $a^{\langle t-1 \rangle}$
W_{aa}
W_{ax}
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 2: Basic RNN cell. Takes as input $x^{\langle t \rangle}$ (current input) and $a^{\langle t-1 \rangle}$ (previous hidden state containing information from the past), and outputs $a^{\langle t \rangle}$ which is given to the next RNN cell and also used to predict $\hat{y}^{\langle t \rangle}$

rnn cell versus rnn_cell_forward

- Note that an RNN cell outputs the hidden state $a^{\langle t \rangle}$.
  - The rnn cell is shown in the figure as the inner box which has solid lines.
- The function that we will implement, `rnn_cell_forward`, also calculates the prediction $\hat{y}^{\langle t \rangle}$
  - The rnn_cell_forward is shown in the figure as the outer box that has dashed lines.

Exercise: Implement the RNN-cell described in Figure (2).

Instructions:

1. Compute the hidden state with tanh activation: $a^{\langle t \rangle} = \tanh(W_{aa} a^{\langle t-1 \rangle} + W_{ax} x^{\langle t \rangle} + b_a)$.
2. Using your new hidden state $a^{\langle t \rangle}$, compute the prediction $\hat{y}^{\langle t \rangle} = softmax(W_{ya} a^{\langle t \rangle} + b_y)$. We provided the function `softmax`.
3. Store $(a^{\langle t \rangle}, a^{\langle t-1 \rangle}, x^{\langle t \rangle}, parameters)$ in a `cache`.
4. Return $a^{\langle t \rangle}$, $\hat{y}^{\langle t \rangle}$ and `cache`

Additional Hints

- numpy.tanh
- We've created a `softmax` function that you can use. It is located in the file 'rnn_utils.py' and has been imported.
- For matrix multiplication, use numpy.dot

```
def rnn_cell_forward(xt, a_prev, parameters):
    """
    Implements a single forward step of the RNN-cell as described in Figure (2)

    Arguments:
    xt -- your input data at timestep "t", numpy array of shape (n_x, m).
    a_prev -- Hidden state at timestep "t-1", numpy array of shape (n_a, m)
    parameters -- python dictionary containing:
                        Wax -- Weight matrix multiplying the input, numpy array of shape (n_a, n_x)
                        Waa -- Weight matrix multiplying the hidden state, numpy array of shape (n_a, n_a)
                        Wya -- Weight matrix relating the hidden-state to the output, numpy array of shape
                        (n_y, n_a)
                        ba --  Bias, numpy array of shape (n_a, 1)
                        by -- Bias relating the hidden-state to the output, numpy array of shape (n_y, 1)
    Returns:
    a_next -- next hidden state, of shape (n_a, m)
    yt_pred -- prediction at timestep "t", numpy array of shape (n_y, m)
    cache -- tuple of values needed for the backward pass, contains (a_next, a_prev, xt, parameters)
    """
    # Retrieve parameters from "parameters"
    Wax = parameters["Wax"]
    Waa = parameters["Waa"]
    Wya = parameters["Wya"]
    ba = parameters["ba"]
    by = parameters["by"]
    ### START CODE HERE ### (≈2 lines)
    # compute next activation state using the formula given above
    a_next =  np.tanh(np.dot(Waa,a_prev)+np.dot(Wax,xt)+ba)
    # compute output of the current cell using the formula given above
    yt_pred = softmax(np.dot(Wya,a_next)+by)
    ### END CODE HERE ###
    # store values you need for backward propagation in cache
    cache = (a_next, a_prev, xt, parameters)

    return a_next, yt_pred, cache
```

:

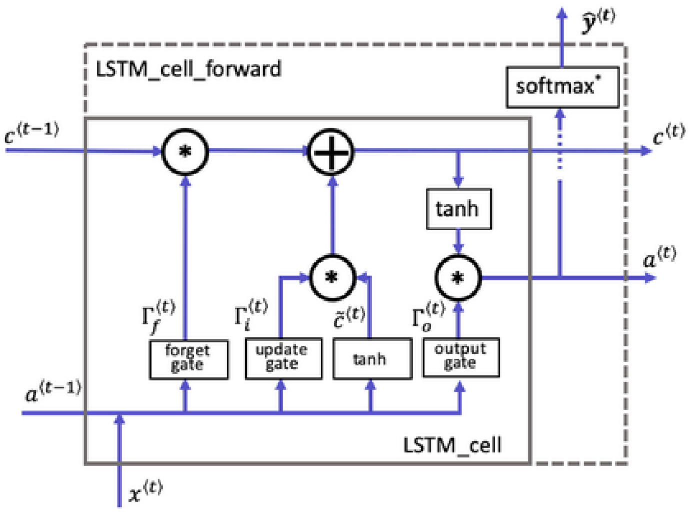
```
np.random.seed(1)
xt_tmp = np.random.randn(3,10)
a_prev_tmp = np.random.randn(5,10)
parameters_tmp = {}
parameters_tmp['Waa'] = np.random.randn(5,5)
parameters_tmp['Wax'] = np.random.randn(5,3)
parameters_tmp['Wya'] = np.random.randn(2,5)
parameters_tmp['ba'] = np.random.randn(5,1)
parameters_tmp['by'] = np.random.randn(2,1)

a_next_tmp, yt_pred_tmp, cache_tmp = rnn_cell_forward(xt_tmp, a_prev_tmp, parameters_tmp)
print("a_next[4] = \n", a_next_tmp[4])
print("a_next.shape = \n", a_next_tmp.shape)
print("yt_pred[1] =\n", yt_pred_tmp[1])
print("yt_pred.shape = \n", yt_pred_tmp.shape)
```

Situations when this RNN will perform better:

- This will work well enough for some applications, but it suffers from the vanishing gradient problems.
- The RNN works best when each output $\hat{y}^{\langle t \rangle}$ can be estimated using "local" context.
- "Local" context refers to information that is close to the prediction's time step t.
- More formally, local context refers to inputs $x^{\langle t' \rangle}$ and predictions $\hat{y}^{\langle t \rangle}$ where t' is close to t.

In the next part, you will build a more complex LSTM model, which is better at addressing vanishing gradients. The LSTM will be better able to remember a piece of information and keep it saved for many timesteps.

The following figure shows the operations of an LSTM-cell.



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

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 $a^{\langle t \rangle}$.
Note, the $softmax^{*}$ includes a dense layer and softmax

Similar to the RNN example above, you will start by implementing the LSTM cell for a single time-step. Then you can iteratively call it from inside a "for-loop" to have it process an input with T_x time-steps.

Overview of gates and states

- Forget gate $\mathbf{\Gamma}_{f}$

- Let's assume we are reading words in a piece of text, and plan to use an LSTM to keep track of grammatical structures, such as whether the subject is singular ("puppy") or plural ("puppies").
- If the subject changes its state (from a singular word to a plural word), the memory of the previous state becomes outdated, so we "forget" that outdated state.
- The "forget gate" is a tensor containing values that are between 0 and 1.
  - If a unit in the forget gate has a value close to 0, the LSTM will "forget" the stored state in the corresponding unit of the previous cell state.
  - If a unit in the forget gate has a value close to 1, the LSTM will mostly remember the corresponding value in the stored state.

Equation

$$\mathbf{\Gamma}_f^{\langle t \rangle} = \sigma(\mathbf{W}_f[\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_f)$$

Explanation of the equation:

- $\mathbf{W_{f}}$ contains weights that govern the forget gate's behavior.
- The previous time step's hidden state $[a^{\langle t-1 \rangle}$ and current time step's input $x^{\langle t \rangle}]$ are concatenated together and multiplied by $\mathbf{W_{f}}$.
- A sigmoid function is used to make each of the gate tensor's values $\mathbf{\Gamma}_f^{\langle t \rangle}$ range from 0 to 1.
- The forget gate $\mathbf{\Gamma}_f^{\langle t \rangle}$ has the same dimensions as the previous cell state $c^{\langle t-1 \rangle}$.
- This means that the two can be multiplied together, element-wise.
- Multiplying the tensors $\mathbf{\Gamma}_f^{\langle t \rangle} * \mathbf{c}^{\langle t-1 \rangle}$ is like applying a mask over the previous cell state.
- If a single value in $\mathbf{\Gamma}_f^{\langle t \rangle}$ is 0 or close to 0, then the product is close to 0.
  - This keeps the information stored in the corresponding unit in $\mathbf{c}^{\langle t-1 \rangle}$ from being remembered for the next time step.
- Similarly, if one value is close to 1, the product is close to the original value in the previous cell state.
  - The LSTM will keep the information from the corresponding unit of $\mathbf{c}^{\langle t-1 \rangle}$, to be used in the next time step.

Variable names in the code

The variable names in the code are similar to the equations, with slight differences.

- `Wf`: forget gate weight $\mathbf{W}_f$
- `bf`: forget gate bias $\mathbf{b}_f$
- `ft`: forget gate $\Gamma_f^{\langle t \rangle}$

Candidate value $\tilde{\mathbf{c}}^{\langle t \rangle}$

- The candidate value is a tensor containing information from the current time step that **may** be stored in the current cell state $\mathbf{c}^{\langle t \rangle}$.
- Which parts of the candidate value get passed on depends on the update gate.
- The candidate value is a tensor containing values that range from -1 to 1.
- The tilde "~" is used to differentiate the candidate $\tilde{\mathbf{c}}^{\langle t \rangle}$ from the cell state $\mathbf{c}^{\langle t \rangle}$.

Equation

$$\tilde{\mathbf{c}}^{\langle t \rangle} = \tanh\left(\mathbf{W}_c [\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_c \right)$$

Explanation of the equation

- The 'tanh' function produces values between -1 and +1.

Variable names in the code

- `cct`: candidate value $\tilde{\mathbf{c}}^{\langle t \rangle}$

- Update gate $\mathbf{\Gamma}_i$

- We use the update gate to decide what aspects of the candidate $\tilde{\mathbf{c}}^{\langle t \rangle}$ to add to the cell state $c^{\langle t \rangle}$.
- The update gate decides what parts of a "candidate" tensor $\tilde{\mathbf{c}}^{\langle t \rangle}$ are passed onto the cell state $\mathbf{c}^{\langle t \rangle}$.
- The update gate is a tensor containing values between 0 and 1.
  - When a unit in the update gate is close to 1, it allows the value of the candidate $\tilde{\mathbf{c}}^{\langle t \rangle}$ to be passed onto the hidden state $\mathbf{c}^{\langle t \rangle}$
  - When a unit in the update gate is close to 0, it prevents the corresponding value in the candidate from being passed onto the hidden state.
- Notice that we use the subscript "i" and not "u", to follow the convention used in the literature.

Equation

$$\mathbf{\Gamma}_i^{\langle t \rangle} = \sigma(\mathbf{W}_i [a^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_i)$$

Explanation of the equation

- Similar to the forget gate, here $\mathbf{\Gamma}_i^{\langle t \rangle}$, the sigmoid produces values between 0 and 1.
- The update gate is multiplied element-wise with the candidate, and this product ($\mathbf{\Gamma}_i^{\langle t \rangle} * \tilde{c}^{\langle t \rangle}$) is used in determining the cell state $\mathbf{c}^{\langle t \rangle}$.

Variable names in code (Please note that they're different than the equations)

In the code, we'll use the variable names found in the academic literature. These variables don't use "u" to denote "update".

- `Wi` is the update gate weight $\mathbf{W}_i$ (not "Wu")
- `bi` is the update gate bias $\mathbf{b}_i$ (not "bu")
- `it` is the forget gate $\mathbf{\Gamma}_i^{\langle t \rangle}$ (not "ut")

- Cell state $\mathbf{c}^{\langle t \rangle}$

- The cell state is the "memory" that gets passed onto future time steps.
- The new cell state $\mathbf{c}^{\langle t \rangle}$ is a combination of the previous cell state and the candidate value.

Equation

$$\mathbf{c}^{\langle t \rangle} = \mathbf{\Gamma}_f^{\langle t \rangle} * \mathbf{c}^{\langle t-1 \rangle} + \mathbf{\Gamma}_i^{\langle t \rangle} * \tilde{\mathbf{c}}^{\langle t \rangle}$$

Explanation of equation

- The previous cell state $\mathbf{c}^{\langle t-1 \rangle}$ is adjusted (weighted) by the forget gate $\mathbf{\Gamma}_f^{\langle t \rangle}$
- and the candidate value $\tilde{\mathbf{c}}^{\langle t \rangle}$, adjusted (weighted) by the update gate $\mathbf{\Gamma}_i^{\langle t \rangle}$

Variable names and shapes in the code

- `c`: cell state, including all time steps, $\mathbf{c}$ shape (n_a, m, T)
- `c_next`: new (next) cell state, $\mathbf{c}^{\langle t \rangle}$ shape (n_a, m)
- `c_prev`: previous cell state, $\mathbf{c}^{\langle t-1 \rangle}$, shape (n_a, m)

- Output gate $\mathbf{\Gamma}_o$

- The output gate decides what gets sent as the prediction (output) of the time step.
- The output gate is like the other gates. It contains values that range from 0 to 1.

Equation

$$\mathbf{\Gamma}_o^{\langle t \rangle} = \sigma(\mathbf{W}_o[\mathbf{a}^{\langle t-1 \rangle}, \mathbf{x}^{\langle t \rangle}] + \mathbf{b}_o)$$

Explanation of the equation

- The output gate is determined by the previous hidden state $\mathbf{a}^{\langle t-1 \rangle}$ and the current input $\mathbf{x}^{\langle t \rangle}$
- The sigmoid makes the gate range from 0 to 1.

Variable names in the code

- `Wo`: output gate weight, $\mathbf{W_o}$
- `bo`: output gate bias, $\mathbf{b_o}$
- `ot`: output gate, $\mathbf{\Gamma}_o^{\langle t \rangle}$

- Hidden state $\mathbf{a}^{\langle t \rangle}$

- The hidden state gets passed to the LSTM cell's next time step.
- It is used to determine the three gates ($\mathbf{\Gamma}_f, \mathbf{\Gamma}_u, \mathbf{\Gamma}_o$) of the next time step.
- The hidden state is also used for the prediction $y^{\langle t \rangle}$.

EXERCISE $\rightarrow$ 02

Equation

$$\mathbf{a}^{\langle t \rangle} = \mathbf{\Gamma}_o^{\langle t \rangle} * \tanh(\mathbf{c}^{\langle t \rangle})$$

Explanation of equation

- The hidden state $\mathbf{a}^{\langle t \rangle}$ is determined by the cell state $\mathbf{c}^{\langle t \rangle}$ in combination with the output gate $\mathbf{\Gamma}_o$.
- The cell state state is passed through the "tanh" function to rescale values between -1 and +1.
- The output gate acts like a "mask" that either preserves the values of $\tanh(\mathbf{c}^{\langle t \rangle})$ or keeps those values from being included in the hidden state $\mathbf{a}^{\langle t \rangle}$

Variable names and shapes in the code

- a: hidden state, including time steps. $\mathbf{a}$ has shape (n_a, m, T_x)
- 'a_prev`: hidden state from previous time step. $\mathbf{a}^{\langle t-1 \rangle}$ has shape (n_a, m)
- `a_next`: hidden state for next time step. $\mathbf{a}^{\langle t \rangle}$ has shape (n_a, m)

- Prediction $\mathbf{y}^{\langle t \rangle}_{pred}$

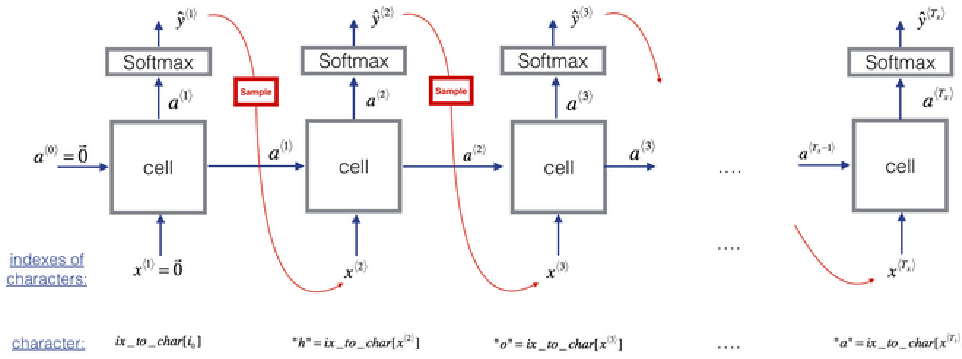
- The prediction in this use case is a classification, so we'll use a softmax.

The equation is:

$$\mathbf{y}^{\langle t \rangle}_{pred} = \textrm{softmax}(\mathbf{W}_{y} \mathbf{a}^{\langle t \rangle} + \mathbf{b}_{y})$$

Variable names and shapes in the code

- `y_pred`: prediction, including all time steps. $\mathbf{y}_{pred}$ has shape (n_{y}, m, T_{x}). Note that $(T_{y} = T_{x})$ for this example.
- `yt_pred`: prediction for the current time step t. $\mathbf{y}^{\langle t \rangle}_{pred}$ has shape (n_{y}, m)



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

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 sample one character at a time.

Exercise: Implement the `sample` function below to sample characters. You need to carry out 4 steps:

- **Step 1**: Input the "dummy" vector of zeros $x^{\langle 1 \rangle} = \vec{0}$.
  - This is the default input before we've generated any characters. We also set $a^{\langle 0 \rangle} = \vec{0}$

- **Step 2**: Run one step of forward propagation to get $a^{\langle 1 \rangle}$ and $\hat{y}^{\langle 1 \rangle}$. Here are the equations:

hidden state:

$$a^{\langle t+1 \rangle} = \tanh(W_{ax} x^{\langle t+1 \rangle} + W_{aa} a^{\langle t \rangle} + b)$$

activation:

$$z^{\langle t+1 \rangle} = W_{ya} a^{\langle t+1 \rangle} + b_y$$

prediction:

$$\hat{y}^{\langle t+1 \rangle} = softmax(z^{\langle t+1 \rangle})$$

- Details about $\hat{y}^{\langle t+1 \rangle}$:
  - Note that $\hat{y}^{\langle t+1 \rangle}$ is a (softmax) probability vector (its entries are between 0 and 1 and sum to 1).
  - $\hat{y}^{\langle t+1 \rangle}_i$ represents the probability that the character indexed by "i" is the next character.
  - We have provided a `softmax()` function that you can use

- Details about $\hat{y}^{\langle t+1 \rangle}$:
  - Note that $\hat{y}^{\langle t+1 \rangle}$ is a (softmax) probability vector (its entries are between 0 and 1 and sum to 1).
  - $\hat{y}^{\langle t+1 \rangle}_i$ represents the probability that the character indexed by "i" is the next character.
  - We have provided a `softmax()` function that you can use.

Additional Hints

- $x^{\langle 1 \rangle}$ is x in the code. When creating the one-hot vector, make a numpy array of zeros, with the number of rows equal to the number of unique characters, and the number of columns equal to one. It's a 2D and not a 1D array.
- $a^{\langle 0 \rangle}$ is a_`prev` in the code. It is a numpy array of zeros, where the number of rows is n_a, and number of columns is 1. It is a 2D array as well. n_a is retrieved by getting the number of columns in W_{aa} (the numbers need to match in order for the matrix multiplication $W_{aa}a^{\langle t \rangle}$ to work.
- numpy.dot
- numpy.tanh

Using 2D arrays instead of 1D arrays

- You may be wondering why we emphasize that $x^{\langle 1 \rangle}$ and $a^{\langle 0 \rangle}$ are 2D arrays and not 1D vectors.
- For matrix multiplication in numpy, if we multiply a 2D matrix with a 1D vector, we end up with with a 1D array.
- This becomes a problem when we add two arrays where we expected them to have the same shape.
- When two arrays with a different number of dimensions are added together, Python "broadcasts" one across the other.
- Here is some sample code that shows the difference between using a 1D and 2D array.

- **Step 3**: Sampling:
  - Now that we have $y^{\langle t+1 \rangle}$, we want to select the next letter in the dinosaur name. If we select the most probable, the model will always generate the same result given a starting letter. To make the results more interesting, we will use np.random.choice to select a next letter that is *likely*, but not always the same.
  - Pick the next character's **index** according to the probability distribution specified by $\hat{y}^{\langle t+1 \rangle}$.
  - This means that if $\hat{y}^{\langle t+1 \rangle}_i = 0.16$, you will pick the index "i" with 16% probability.
  - Use np.random.choice.

    Example of how to use `np.random.choice()`:

    ```
    np.random.seed(0)
    probs = np.array([0.1, 0.0, 0.7, 0.2])
    idx = np.random.choice(range(len((probs)), p = probs)
    ```

  - This means that you will pick the index (`idx`) according to the distribution:

    $P(index = 0) = 0.1, P(index = 1) = 0.0, P(index = 2) = 0.7, P(index = 3) = 0.2.$
  - Note that the value that's set to p should be set to a 1D vector.
  - Also notice that $\hat{y}^{\langle t+1 \rangle}$, which is y in the code, is a 2D array.
  - Also notice, while in your implementation, the first argument to np.random.choice is just an ordered list [0,1,.., vocab_len-1], it is *Not* appropriate to use char_to_ix.values(). The *order* of values returned by a python dictionary .values() call will be the same order as they are added to the dictionary. The grader may have a different order when it runs your routine than when you run it in your notebook.

Additional Hints

- range
- numpy.ravel takes a multi-dimensional array and returns its contents inside of a 1D vector.

```
arr = np.array([[1,2],[3,4]])
print("arr")
print(arr)
print("arr.ravel()")
print(arr.ravel())
```

  Output:

```
arr
[[1 2]
[3 4]]
arr.ravel()
[1 2 3 4]
```

- Note that `append` is an "in-place" operation. In other words, don't do this:

```
fun_hobbies = fun_hobbies.append('learning')  ## Doesn't give you what you want
```

- **Step 4**: Update to $x^{\langle t \rangle}$
  - The last step to implement in `sample()` is to update the variable `x`, which currently stores $x^{\langle t \rangle}$, with the value of $x^{\langle t+1 \rangle}$.
  - You will represent $x^{\langle t+1 \rangle}$ by creating a one-hot vector corresponding to the character that you have chosen as your prediction.
  - You will then forward propagate $x^{\langle t+1 \rangle}$ in Step 1 and keep repeating the process until you get a "\n" character, indicating that you have reached the end of the dinosaur name.

Additional Hints

- In order to reset `x` before setting it to the new one-hot vector, you'll want to set all the values to zero.
  - You can either create a new numpy array: numpy.zeros
  - Or fill all values with a single number: numpy.ndarray.fill