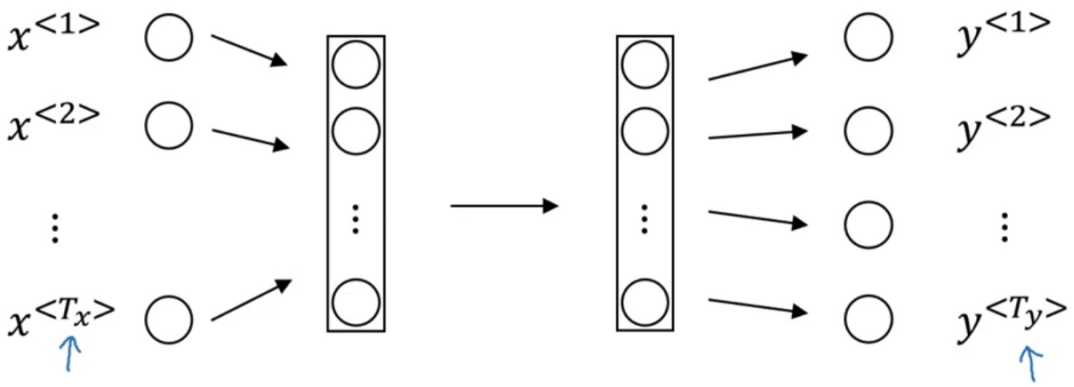
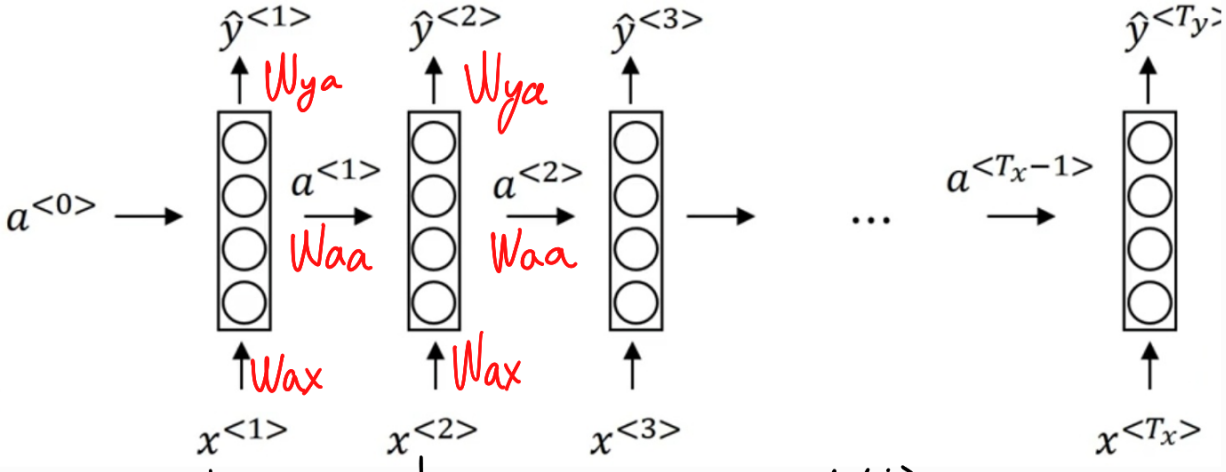
Why not a standard network?



Problems:

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

Forward Propagation



A RNN, computes each $a^{<i>}$ and $\hat{y}^{<i>}$, using $x^{<i-1>}$ and $a^{<i-1>}$, which defines a recursive function with base case $a^{<0>}$ and $x^{<1>}$.

However, eventhough it is great that it uses past information, it does not use future one. Therefore, RNN would not be able to distinguish who is a person here:

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

He said, "Teddy bears are on sale!"

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

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

Let's simplify:

$W_a = [W_{aa} \vdots W_{ax}]$ (they both have same number of rows)

$\hookrightarrow \in \mathbb{R}^{n\_rows\_both \times (n\_cols\_W_{aa} + n\_cols\_W_{ax})}$

$$[a^{<t-1>}, x^{<t>}] = \begin{bmatrix} a^{<t-1>} \\ \cdots\cdots \\ x^{<t>} \end{bmatrix} \in \mathbb{R}^{(n\_rows\_a^{<t-1>} + n\_rows\_x^{<t>}) \times 1}$$

So, we rewrite $a^{<t>}$ as:

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

→ usually tanh, less usually ReLU

And $\hat{y}^{<t>}$ as:

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

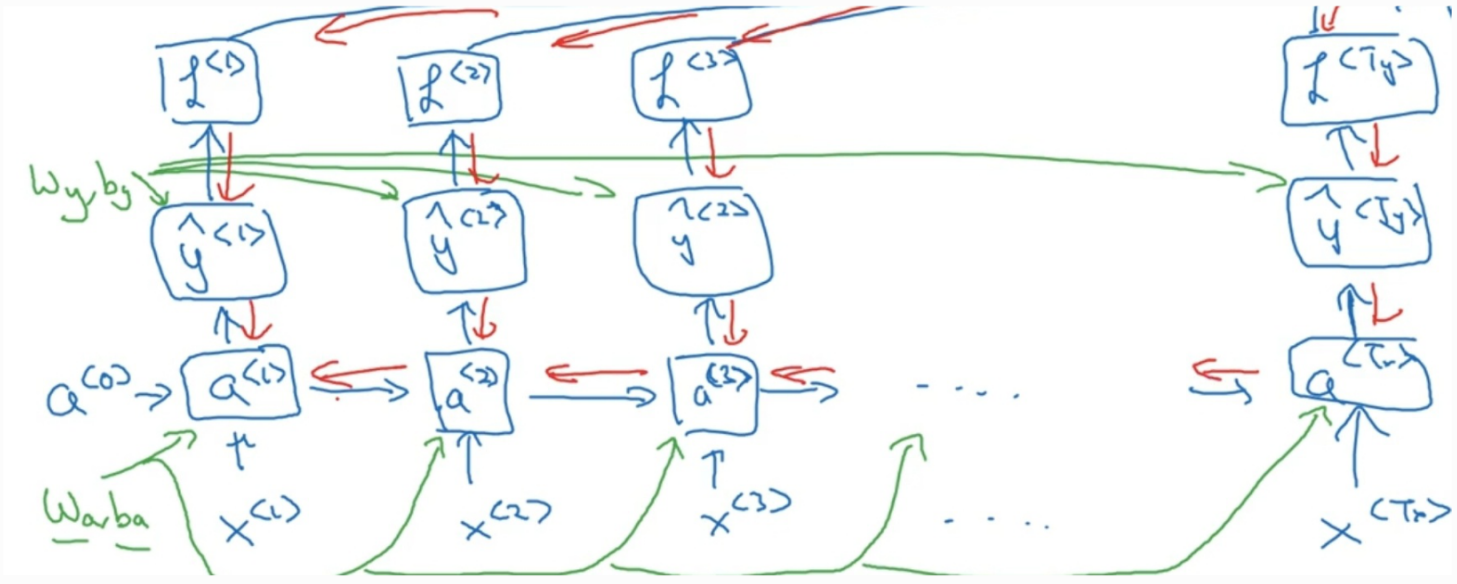
→ softmax, sigmoid

Loss function:

$\mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})$ = cross-entropy loss, from logistic regression =

$$-y^{<t>} \log \hat{y}^{<t>} - (1 - y^{<t>}) \log (1 - \hat{y}^{<t>})$$

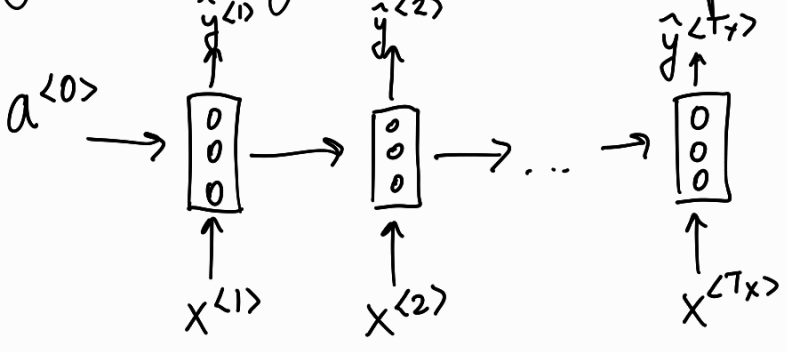
$$\left[\mathcal{L}(\hat{y}, y) = \sum_{t=1}^{T_y} \mathcal{L}^{<t>}(\hat{y}^{<t>}, y^{<t>})\right]$$



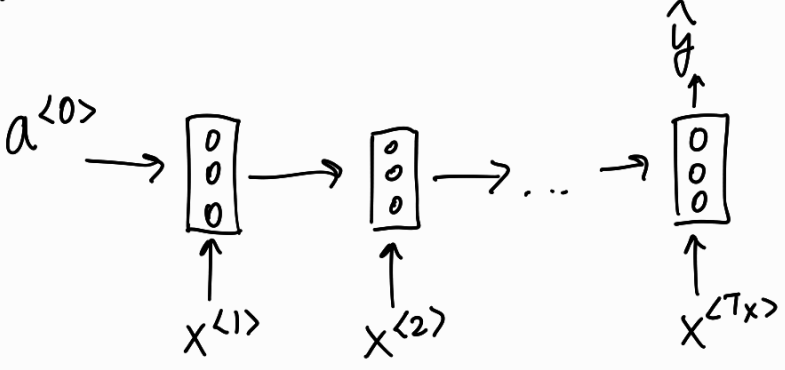
Backpropagation through time.

RNNs architectures:

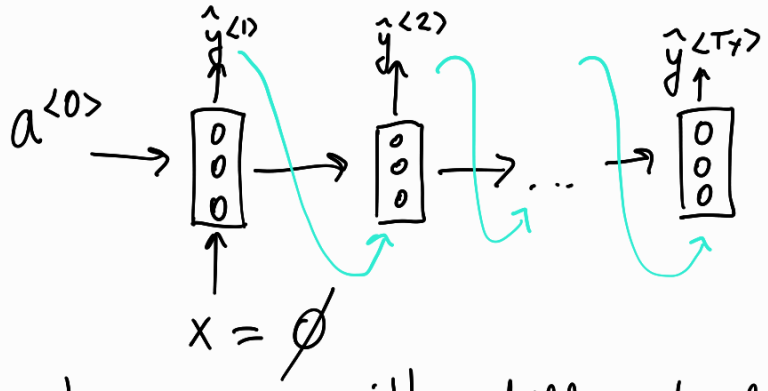
- Many-to-many (the one exposed above):



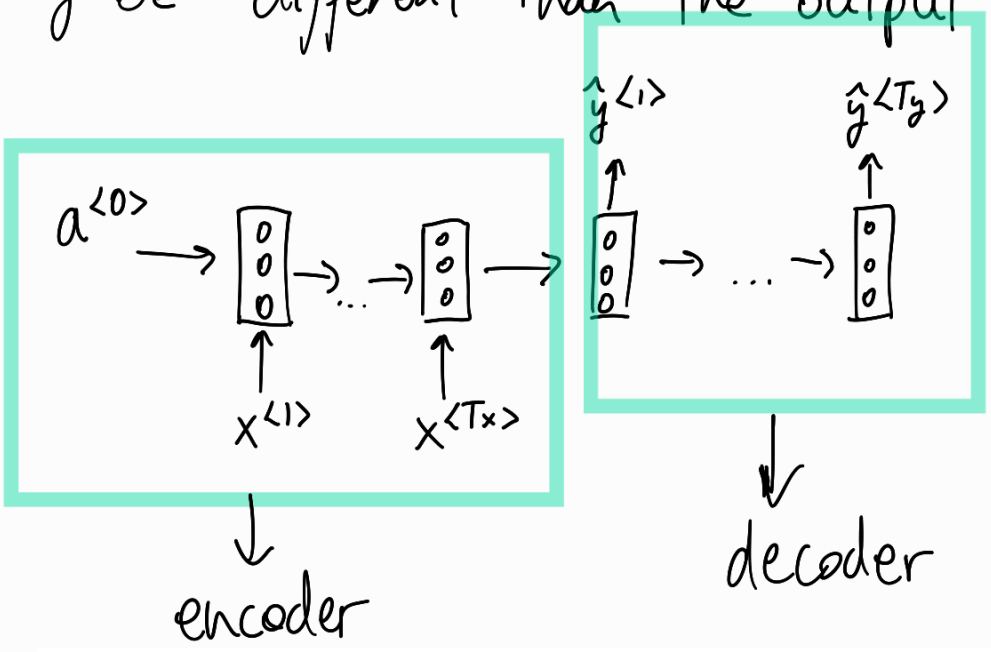
- Many-to-one: for sentiment classification for example



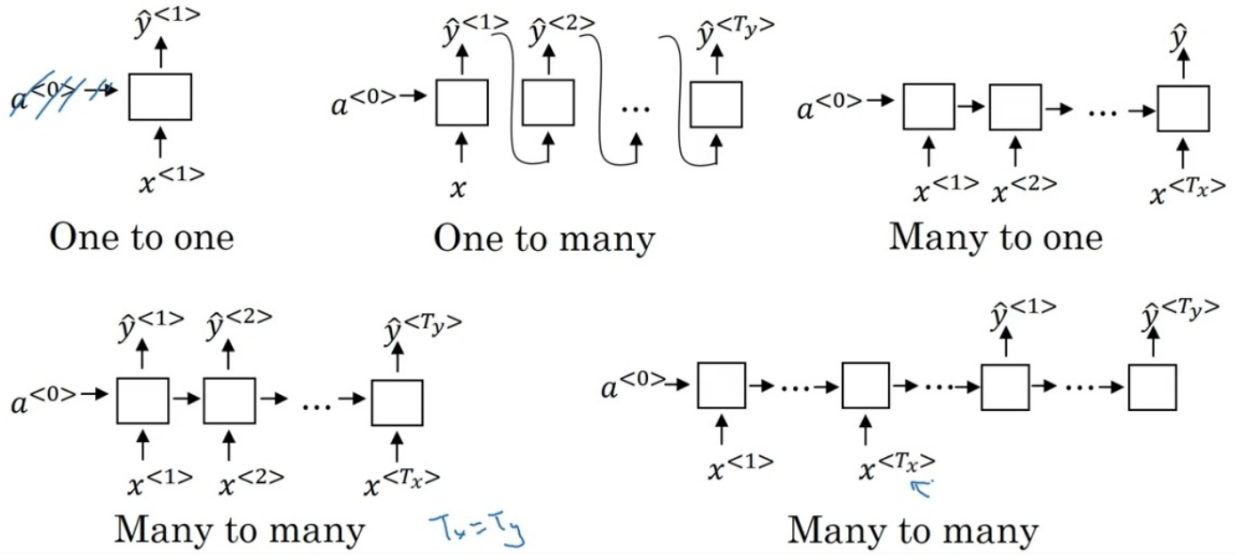
- One-to-many: for example, music generation.



- Many-to-many, with different lengths for input and output. For example, in machine translation, the number of words in the input sentence (in french) may be different than the output one (in english).



Summary of RNN types



Vanishing gradients and forgetting information:

Each $y^{<i>}$, is mainly influenced by close $x^{<j>}$ to it, $j<i$.

That means, the RNN "forgets" as it advances in sequences. So for instance, when generating sentences like:

"the cat, ..., was full"

"the cats, ..., were full"

The RNN, may forget about if there was 1 or more cats.

That is because of backpropagation.

- Solution: GRUs

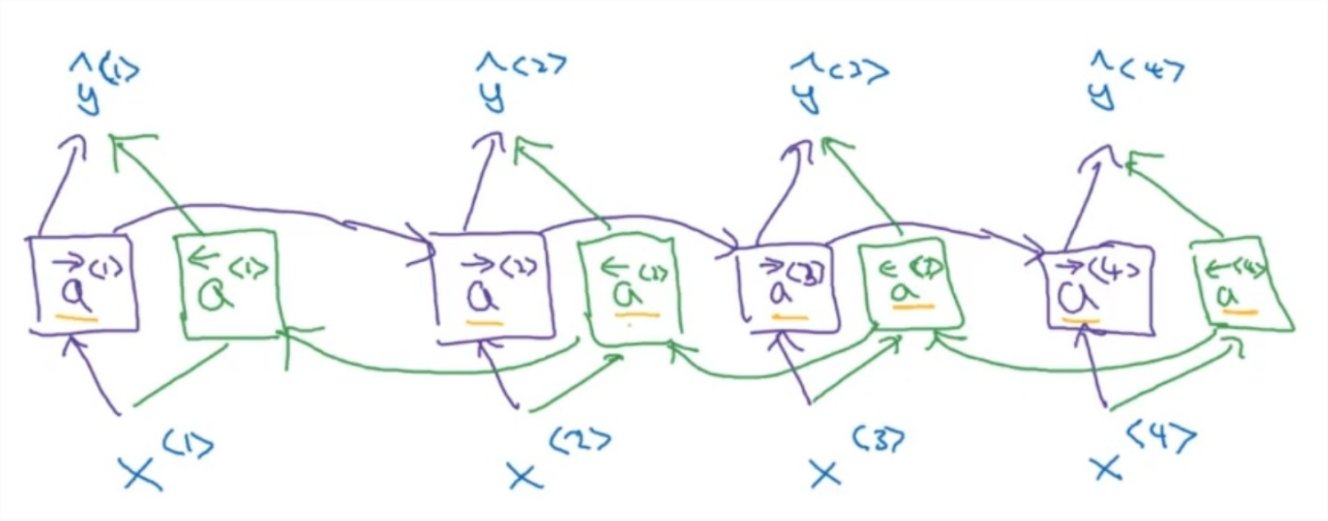
Exploding gradients: Use gradient clipping.

Bidirectional RNNs:

Getting information from the future

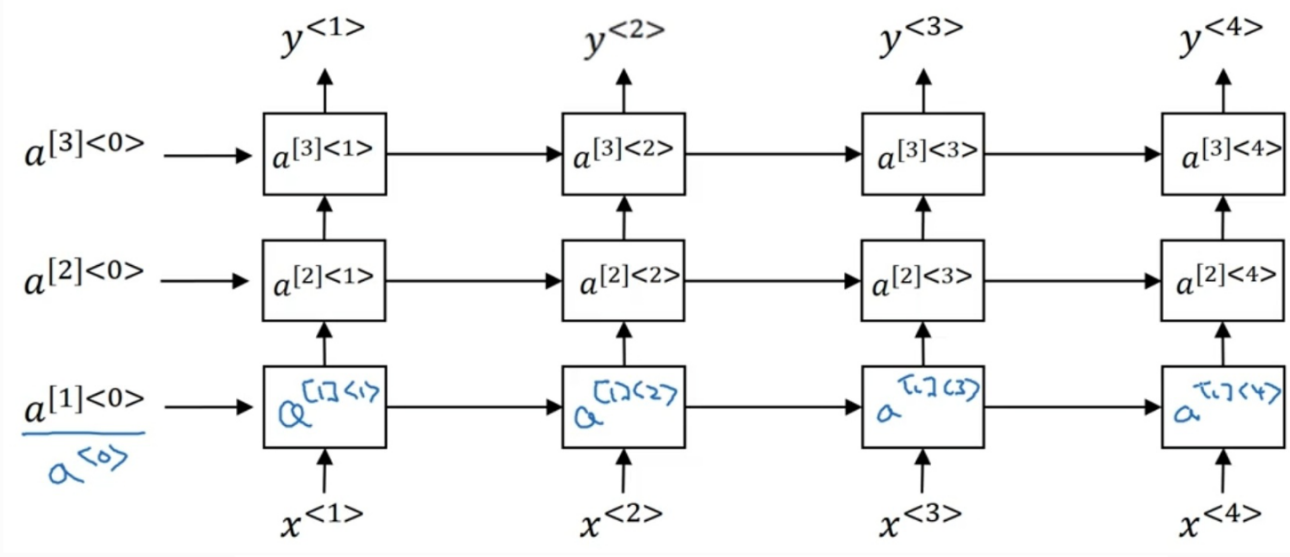
He said, "Teddy bears are on sale!"

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



Disadvantage: we need the entire sequence before we can predict.

Deep RNNs:



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

What you should remember:

- The recurrent neural network, or RNN, is essentially the repeated use of a single cell.
- A basic RNN reads inputs one at a time, and remembers information through the hidden layer activations (hidden states) that are passed from one time step to the next.
  - The time step dimension determines how many times to re-use the RNN cell
- Each cell takes two inputs at each time step:
  - The hidden state from the previous cell
  - The current time step's input data
- Each cell has two outputs at each time step:
  - A hidden state
  - A prediction

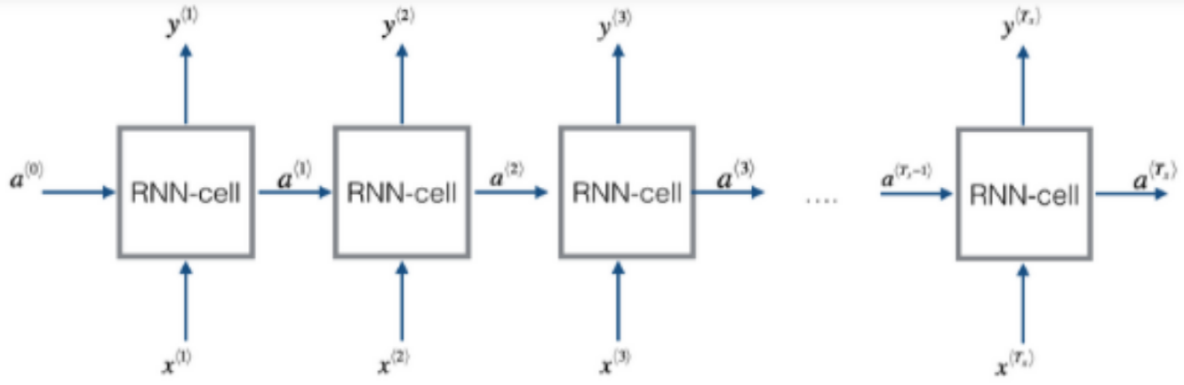


Figure 1: Basic RNN model

Dimensions of input x

Input with n_x number of units

- For a single time step of a single input example, $x^{(i)\langle t \rangle}$ is a one-dimensional input vector
- Using language as an example, a language with a 5000-word vocabulary could be one-hot encoded into a vector that has 5000 units. So $x^{(i)\langle t \rangle}$ would have the shape (5000,)
- The notation n_x is used here to denote the number of units in a single time step of a single training example

Time steps of size T_x

- A recurrent neural network has multiple time steps, which you'll index with t.
- In the lessons, you saw a single training example $x^{(i)}$ consisting of multiple time steps T_x. In this notebook, T_x will denote the number of timesteps in the longest sequence.

Batches of size m

- Let's say we have mini-batches, each with 20 training examples
- To benefit from vectorization, you'll stack 20 columns of $x^{(i)}$ examples
- For example, this tensor has the shape (5000,20,10)
- You'll use m to denote the number of training examples
- So, the shape of a mini-batch is (n_x, m, T_x)

3D Tensor of shape (n_x, m, T_x)

- The 3-dimensional tensor x of shape (n_x, m, T_x) represents the input x that is fed into the RNN

Taking a 2D slice for each time step: $x^{\langle t \rangle}$

- At each time step, you'll use a mini-batch of training examples (not just a single example)
- So, for each time step t, you'll use a 2D slice of shape (n_x, m)
- This 2D slice is referred to as $x^{\langle t \rangle}$. The variable name in the code is `xt`.

Definition of hidden state a

- The activation $a^{\langle t \rangle}$ that is passed to the RNN from one time step to another is called a "hidden state."

Dimensions of hidden state a

- Similar to the input tensor x, the hidden state for a single training example is a vector of length n_a
- If you include a mini-batch of m training examples, the shape of a mini-batch is (n_a, m)
- When you include the time step dimension, the shape of the hidden state is (n_a, m, T_x)
- You'll loop through the time steps with index t, and work with a 2D slice of the 3D tensor
- This 2D slice is referred to as $a^{\langle t \rangle}$
- In the code, the variable names used are either `a_prev` or `a_next`, depending on the function being implemented
- The shape of this 2D slice is (n_a, m)

Dimensions of prediction $\hat{y}$

- Similar to the inputs and hidden states, $\hat{y}$ is a 3D tensor of shape (n_y, m, T_y)
  - n_y: number of units in the vector representing the prediction
  - m: number of examples in a mini-batch
  - T_y: number of time steps in the prediction
- For a single time step t, a 2D slice $\hat{y}^{\langle t \rangle}$ has shape (n_y, m)
- In the code, the variable names are:
  - `y_pred` : $\hat{y}$
  - `yt_pred` : $\hat{y}^{\langle t \rangle}$

Here's how you can implement an RNN:

Steps:

1. Implement the calculations needed for one time step of the RNN.
2. Implement a loop over T_x time steps in order to process all the inputs, one at a time.

1.1 - RNN Cell

You can think of the recurrent neural network as the repeated use of a single cell. First, you'll implement the computations for a single time step. The following figure describes the operations for a single time step of an RNN cell:

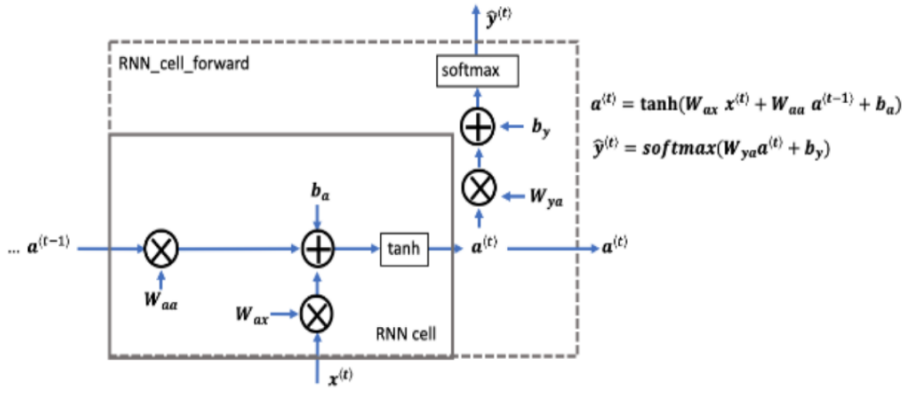


Figure 2: Basic RNN cell. Takes as input $x^{\langle t \rangle}$ (current input) and $a^{\langle t-1 \rangle}$ (previous hidden state containing information from the past), and outputs $a^{\langle t \rangle}$ which is given to the next RNN cell and also used to predict $\hat{y}^{\langle t \rangle}$

1.2 - RNN Forward Pass

- A recurrent neural network (RNN) is a repetition of the RNN cell that you've just built.
  - If your input sequence of data is 10 time steps long, then you will re-use the RNN cell 10 times
- Each cell takes two inputs at each time step:
  - $a^{\langle t-1 \rangle}$: The hidden state from the previous cell
  - $x^{\langle t \rangle}$: The current time step's input data
- It has two outputs at each time step:
  - A hidden state ($a^{\langle t \rangle}$)
  - A prediction ($y^{\langle t \rangle}$)
- The weights and biases (W_{aa}, b_a, W_{ax}, b_x) are re-used each time step
  - They are maintained between calls to `rnn_cell_forward` in the 'parameters' dictionary

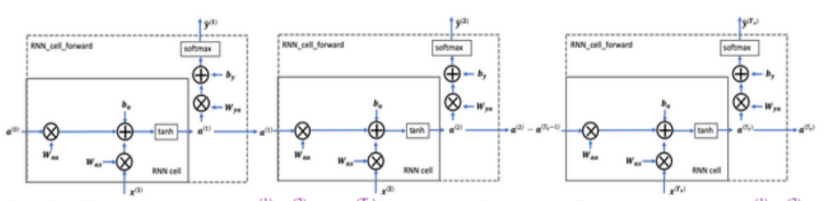


Figure 3: Basic RNN. The input sequence $x = (x^{\langle 1 \rangle}, x^{\langle 2 \rangle}, \ldots, x^{\langle T_x \rangle})$ is carried over T_x time steps. The network outputs $y = (y^{\langle 1 \rangle}, y^{\langle 2 \rangle}, \ldots, y^{\langle T_x \rangle})$.

Exercise 2 - rnn_forward

Implement the forward propagation of the RNN described in Figure 3.

Instructions:

- Create a 3D array of zeros, a of shape (n_a, m, T_x) that will store all the hidden states computed by the RNN
- Create a 3D array of zeros, $\hat{y}$, of shape (n_y, m, T_x) that will store the predictions
  - Note that in this case, $T_y = T_x$ (the prediction and input have the same number of time steps)
- Initialize the 2D hidden state `a_next` by setting it equal to the initial hidden state, a_0
- At each time step t:
  - Get $x^{\langle t \rangle}$, which is a 2D slice of x for a single time step t
    - $x^{\langle t \rangle}$ has shape (n_x, m)
    - x has shape (n_x, m, T_x)
  - Update the 2D hidden state $a^{\langle t \rangle}$ (variable name `a_next`), the prediction $\hat{y}^{\langle t \rangle}$ and the cache by running `rnn_cell_forward`
    - $a^{\langle t \rangle}$ has shape (n_a, m)
  - Store the 2D hidden state in the 3D tensor a, at the t^{th} position
    - a has shape (n_a, m, T_x)
  - Store the 2D $\hat{y}^{\langle t \rangle}$ prediction (variable name `yt_pred`) in the 3D tensor $\hat{y}_{pred}$ at the t^{th} position
    - $\hat{y}^{\langle t \rangle}$ has shape (n_y, m)
    - $\hat{y}$ has shape (n_y, m, T_x)
  - Append the cache to the list of caches
- Return the 3D tensor a and $\hat{y}$, as well as the list of caches