

# Recurrent Neural Network

**MET CS Generative AI**

Reza Rawassizadeh

# Outline

- Recurrent Neural Network Concepts
- Long Term Short Term Memory (LSTM)
- Gated Recurrent Unit (GRU)
- Bidirectional RNN
- Deep RNN

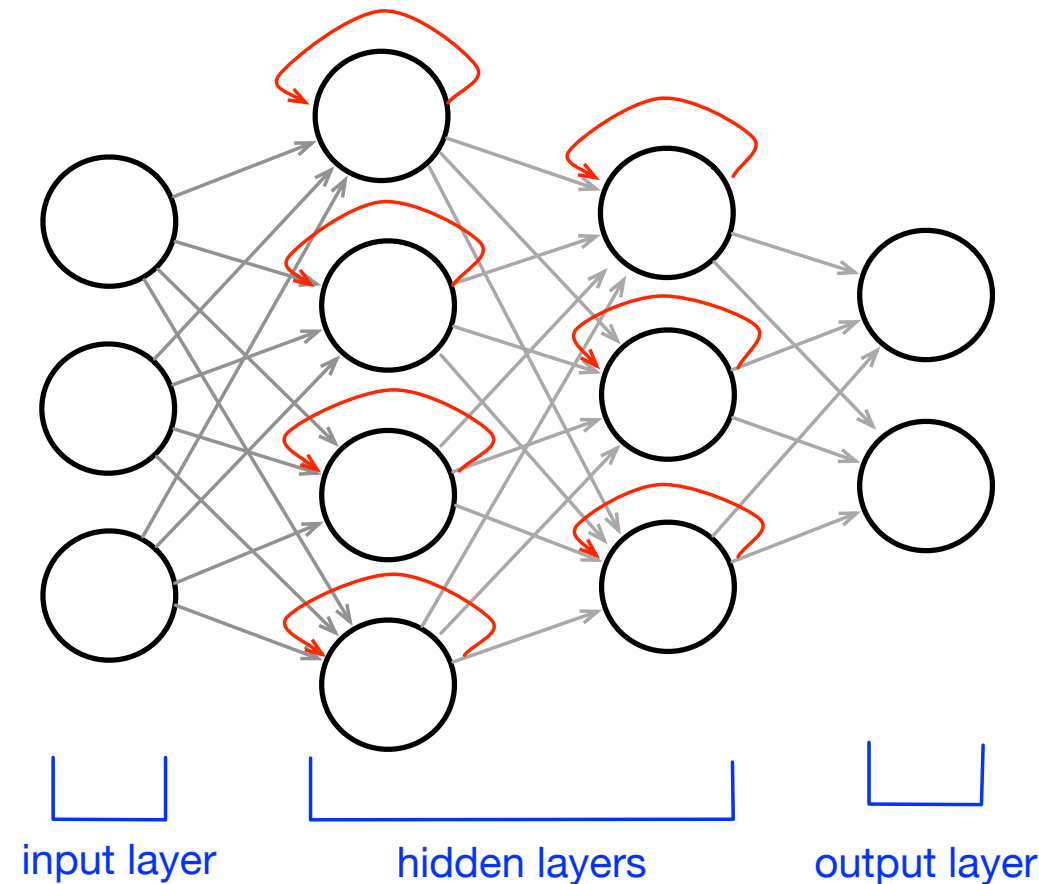
# Outline

- **Recurrent Neural Network Concepts**
- Long Term Short Term Memory (LSTM)
- Gated Recurrent Unit (GRU)
- Bidirectional RNN
- Deep RNN

# Why RNN?

- RNN is used for sequential data, such as sensor data, time-stamped data, medical devices data, audio, and even text (word appearances in a sentence have an order).
- In particular, any data that has either a timestamp or implicit notion of time, or sequence, i.e., temporal data, could be modeled by RNN.
- Even we could use RNN to reconstruct sequential data similar to the data we fed into it. For example, we could feed a book to RNN, and it can generate sentences similar to that book, or we could use RNN to construct music or poem, but there are more accurate methods that leverage RNN and not using RNN alone.

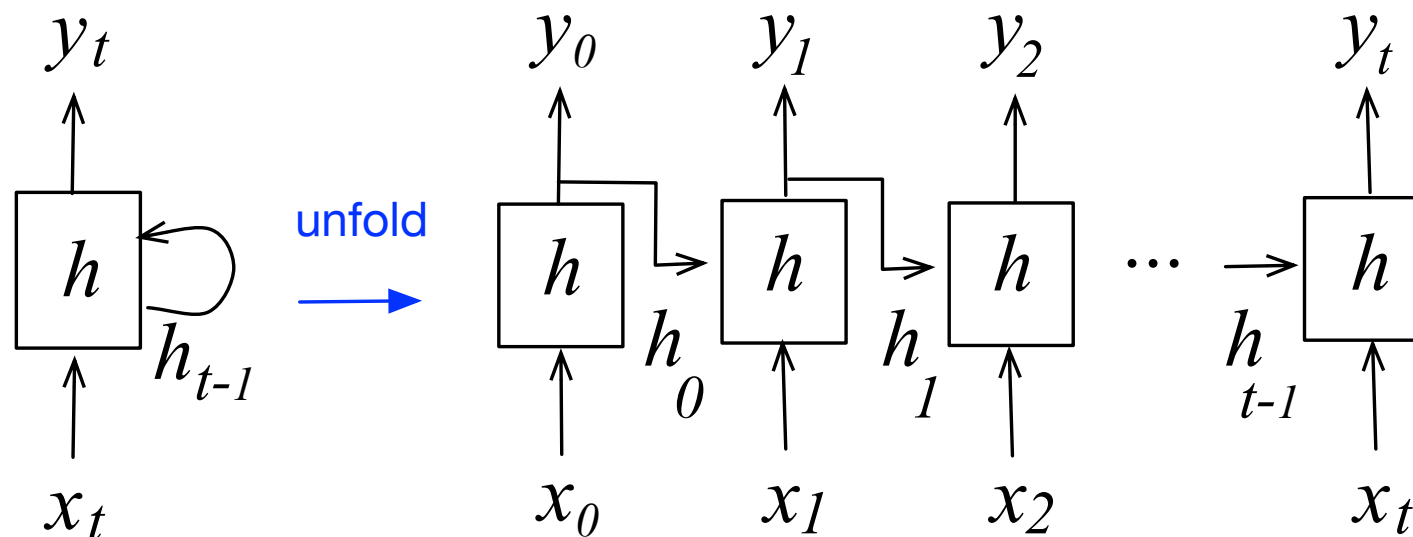
# How RNN works?



- hidden layers of RNN, do give output to the next layer, but, they also feed back their outputs to themselves along with the input (in the next epoch). In other words, they support a temporal loop.

# RNN Hidden Layers

It is one RNN cell and not a single neuron or a layer, but we look at this cell at different time



- On the right side of this figure, we can see that the hidden layer at time  $t$  receives the output of the previous layers ( $t-1$ ) as well, or the layer at a time ( $t-1$ ) receives the output of time  $t-2$  as well.
- Therefore, we could say neurons have a **short-term memory** and remember what has happened in the **previous epoch** (only a single previous epoch not all previous epochs).
- Hidden layers ( $h$ ) in RNN receive input  $x$  and produce output  $y$ . However, the hidden layers have a temporal loop, which means that in addition to giving the output, it feeds the output to itself, as input, for the next epoch as well.

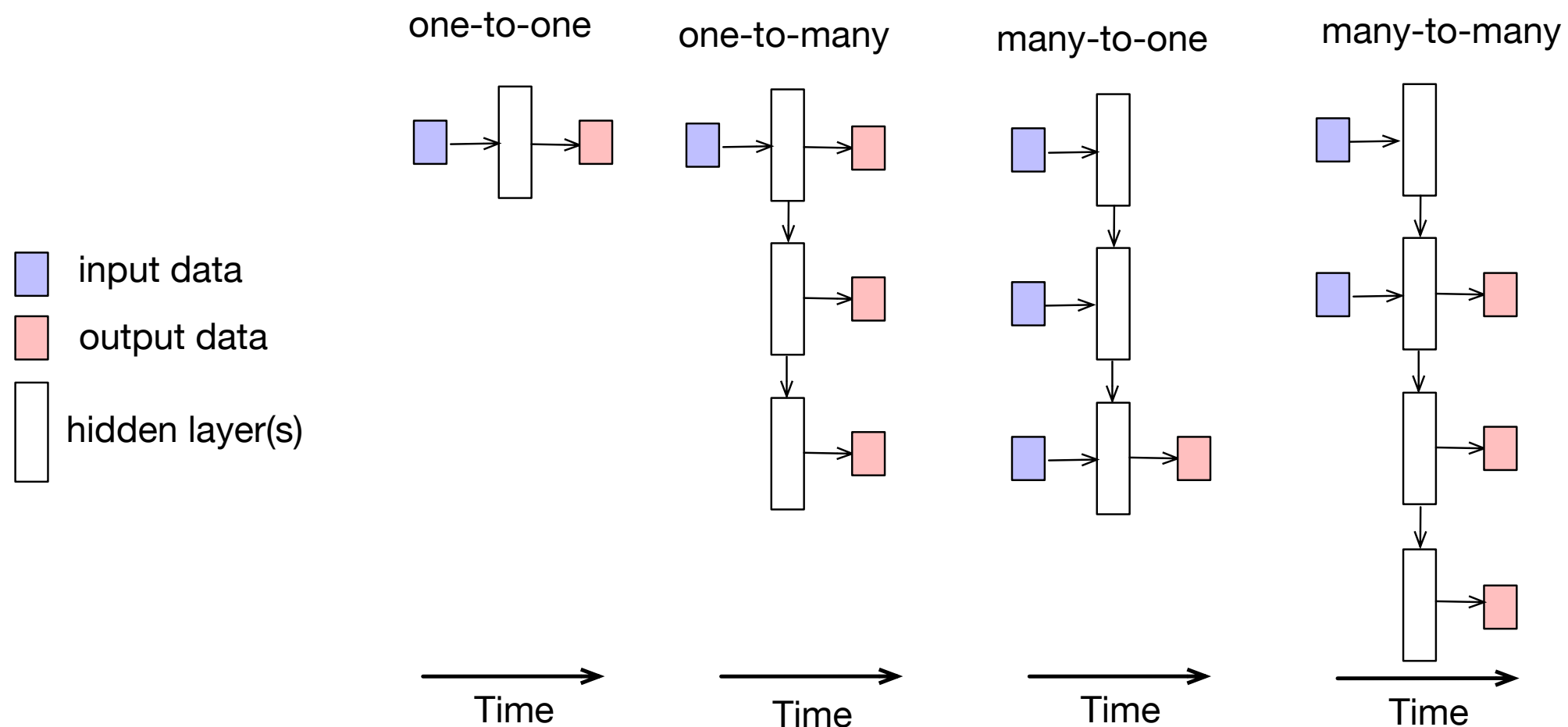
# RNN Hidden Layers

- To formalize this, assuming  $\vec{y}$  is the output vector of hidden layer variables and  $\vec{x}$  is a vector of the input variable, then we have:

$$\vec{y}_t = f(\vec{y}_{t-1}, \vec{x}_t)$$

# Types of RNN based on input/output

- Based on the number of input and output, there are different types of RNN, which are presented as one-to-many, many-to-one, many-to-many and one-to-one.





# Types of RNN based on input/output

- For a one-to-one example, assume a single image given into a neural network and the network finds its label.
- For one-to-many example, assumes we give a single image into the RNN and it generates the caption for the image. A caption is a sentence which is composed of many words.
- For a many-to-one example, assume we give a sentence, which includes several words (many) to a sentiment analyzer and the RNN labels the sentence tone as positive or negative (one).
- For many-to-many examples, we can think of a sentence, which includes many words in Mandarin and the algorithm translates it to a Hindi sentence, which also includes many words as well.

# RNN has Memory

- We have explained that a neuron receives an input, and puts it in the linear function, next it applies an activation function on the result of linear regression, and then produces the output, i.e.,  
$$x(input) \rightarrow z = wx + b \rightarrow \sigma(z) \rightarrow y(output).$$
- While using RNN, we have  $\bar{y}_t = f(\bar{y}_{t-1}, \bar{x}_t)$ , which means that the output depends on the previous outputs as well. Also, its activation function is a hyperbolic tangent activation function, i.e. ***tanh***.

# RNN Memory issue

- RNN is fantastic because of having memory, but it only remembers the previous output, i.e., short-term memory. To mitigate this we need an RNN that keeps track more previous output, i.e., long-term memory.

# Outline

- Recurrent Neural Network Concepts
- **Long Term Short Term Memory (LSTM)**
- Gated Recurrent Unit (GRU)
- Bidirectional RNN
- Deep RNN

# Long-Term Short-Term Memory (LSTM)

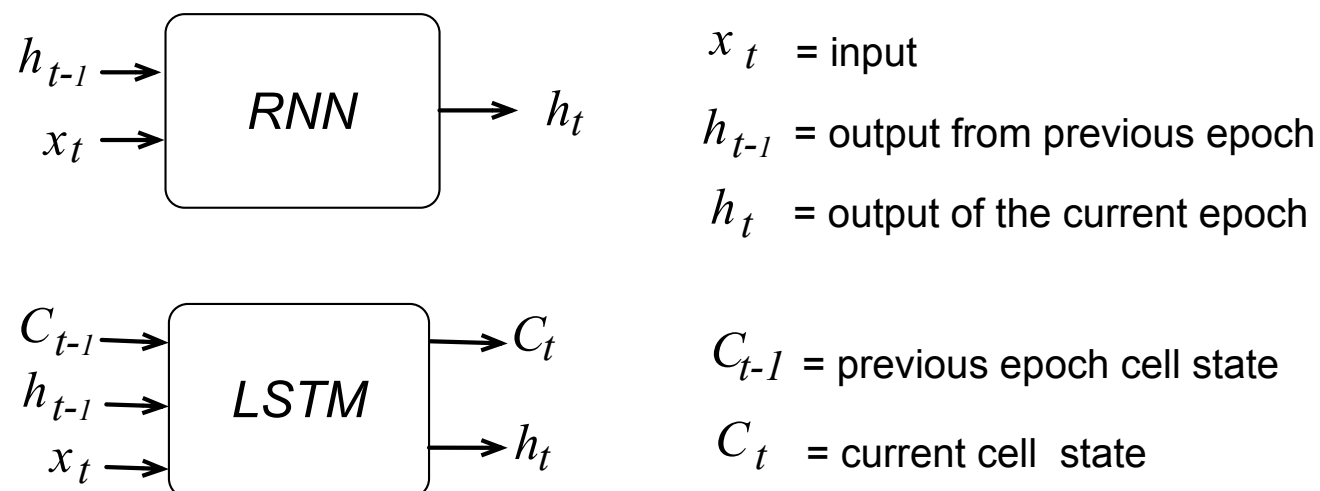
- Vanishing gradient is a severe issue in sequential data, and previous approaches we have explained to mitigate its risk are not very practical for sequential data, especially for sequential data that are long.
- On the other hand, we have explained that the RNN is a gold-fish memory; it has only one short-term memory and can't remember more than one single previous state.
- For example, look at this sentence: *“I love money, and as a consultation fee, I get a fat check.”* The word “*fat*” in this sentence refers to money, not overweight people or nutritional fat. Therefore, if we give this sentence to an RNN, which can only remember the previous word, it cannot recognize the context of the word fat

# LSTM

- A well-known approach to handling time series issues and temporal data is the LSTM algorithm [Hochreiter '97], which is a type of RNN algorithm.
- Instead of one memory, each LSTM layer has four memory components, **long-term memory**, **short-term memory**, **new long-term memory**, and **new short-term memory**.
- It uses the element-wise operator (Hadamard product), i.e.,  $\odot$ , sigmoid and *tanh* activation functions, to decide about the output value.

- An RNN receives  $x_t$  and  $h_{t-1}$  and uses a  $\tanh$  to calculate  $h_t$  as output, which can be written as:  

$$h_t = \tanh(W[h_{t-1}, x_t] + b).$$
- LSTM neurons are referred to as cells.
- An LSTM cell receives information called **cell state** ( $C_{t-1}$  or previous long-term memory), previous output ( $h_{t-1}$  or previous short-term memory) input vector ( $x_t$ ). The output of LSTM is a new cell state ( $C_t$  or new long-term memory), and the output vector ( $h_t$  or new short-term memory).



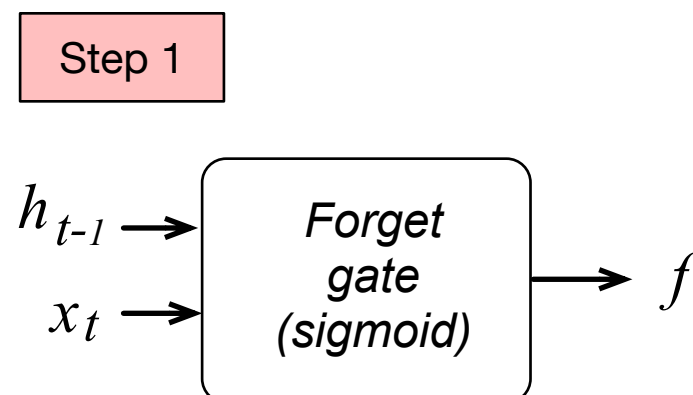
# LSTM Cell Content

- The content of each LSTM cell has input, forget, output, **sigmoid** layers, **hyperbolic tangent** layers (tanh) **gates**, and **point wise operators** (Hadamard product).
- Gates are responsible for deciding *about the information passes the gate or not*. The Sigmoid layer outputs 0 (not passing the gate) or 1 (pass the gate), and *tanh* brings the data into the range -1 to 1.



# LSTM Operation (Step 1)

1. The first step of the LSTM algorithm decides what information is not worth keeping and could be thrown away; this will be done through the **Forget gate**, which is a layer with a Sigmoid function. In particular, it receives the  $h_{t-1}$  (output of the previous time) and  $x_t$  (input), then calculates  $f$  by using the Forget gate (sigmoid function) and outputs  $f = \sigma(W_f[h_{t-1}, x_t] + b_f)$ . At this step, the state is  $C_{t-1}$ , which comes from the previous cell.



# LSTM Operation (Step 2)

2. The second step decides what new information is going to be stored in the cell state.

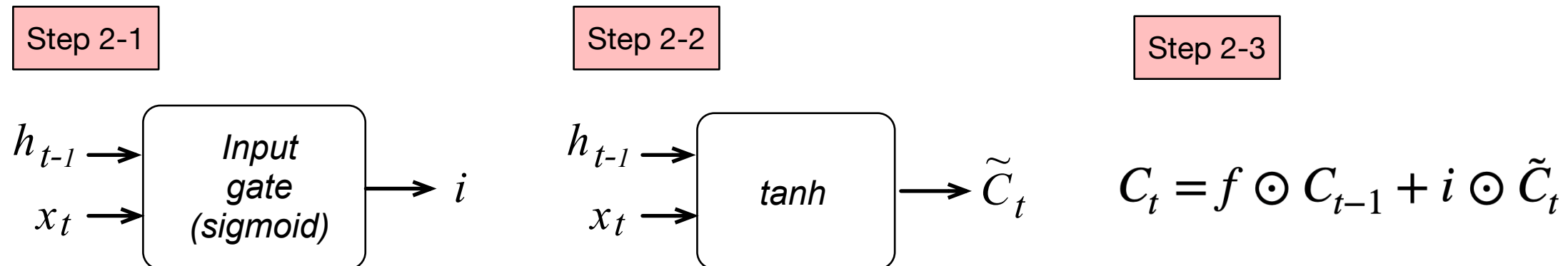
1. First, it uses the **Input gate** to decide which values will be updated. Its equation is written as:

$$i = \sigma(W_i[h_{t-1}, x_t] + b_i).$$

2. Second, a **hyperbolic tangent** function creates a vector of new candidate values, i.e.,  $\tilde{C}_t$ , that its value will be between -1 and 1. These candidate values can be added to a candidate's new cell state (not the final new cell state), and its equation is written as:  $\tilde{C}_t = \tanh(W_C[h_{t-1}, x_t] + b_C)$ .

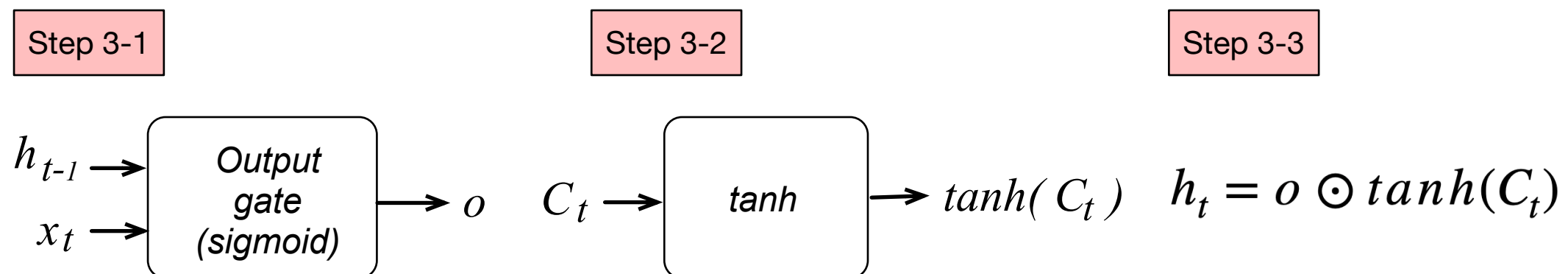
3. Third, after these two sub-steps, their results should be combined together to update the old cell state,  $C_{t-1}$ , into a new cell state:  $C_t$ . To construct the new cell state, the algorithm multiplies the old state by  $f_t$ , to forget the information that is forgettable, and then decides how much a cell state is getting updated by using  $i \odot \tilde{C}_t$ . The third sub-step, which constructs the new cell state ( $C_t$ ) is written as:

$$C_t = f \odot C_{t-1} + i \odot \tilde{C}_t.$$

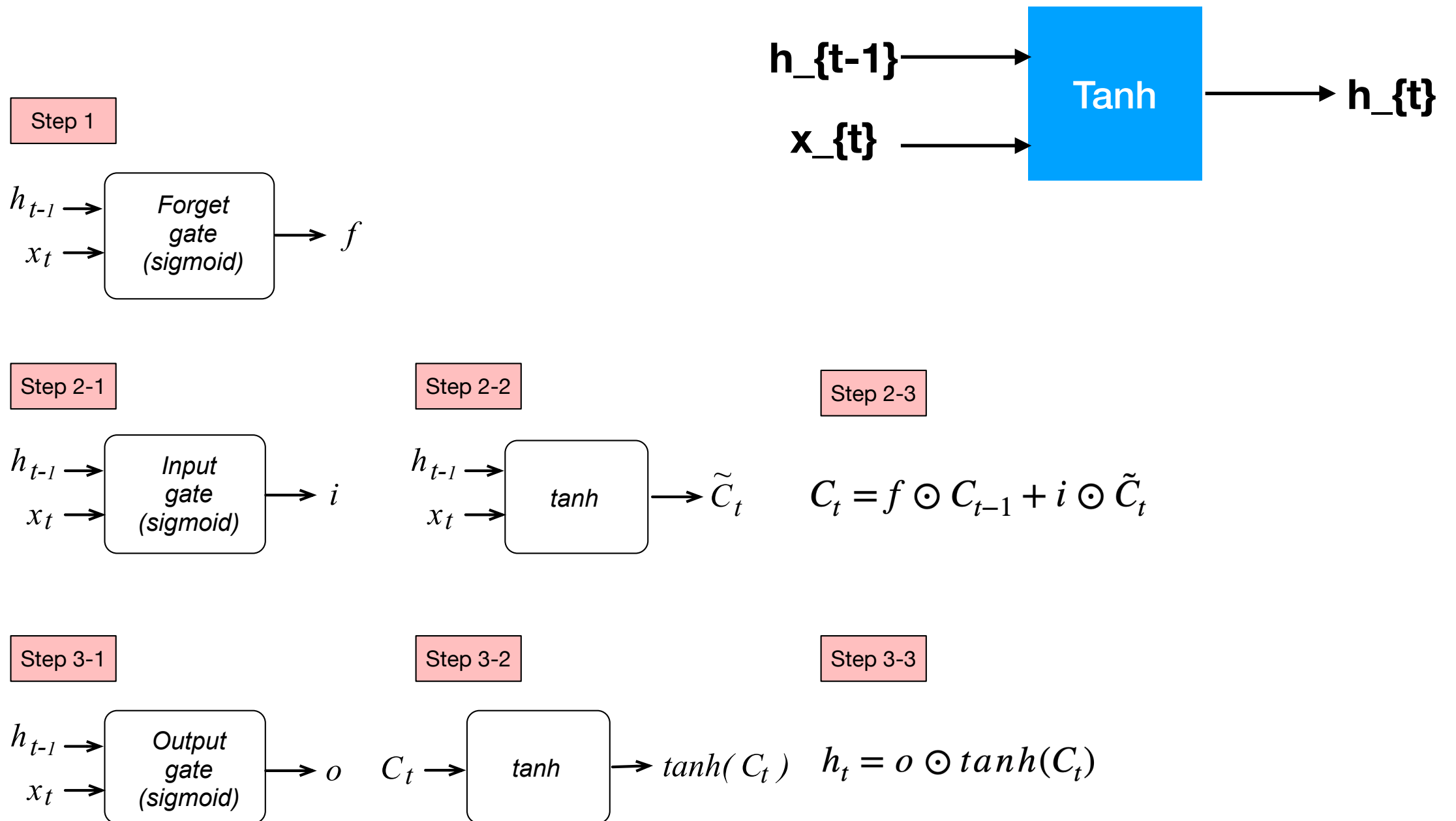


# LSTM Operation (Step 3)

3. In the final step, the algorithm decides what is going to be the output vector, which is based on the filtered version of the  $C_t$  cell state. Here the filter is referred to as passing the  $C_t$  to the  $\tanh$  gate.
  1. In the first sub-step,  $h_{t-1}$  and  $x_t$  sends to **Output gate** (Sigmoid gate), which decides what part of the cell state will be given as output. The equation of this sub-step is written as:  $o = \sigma(W_o[h_{t-1}, x_t] + b_o)$ , see **Figure** Step 3-1.
  2. The cell state will be passed through a  $\tanh$  gate (to transform its values between -1 and 1), (**Figure** Step 3-2).
  3. The third sub-step, constructs the final output,  $h_t$  as  $h_t = o \odot \tanh(C_t)$ , (**Figure** Step 3-3).



# Summary of LSTM



# LSTM Equations

$$i_t = \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi})$$

$$f_t = \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf})$$

$$\tilde{C}_t = \tanh(W_{i\tilde{c}}x_t + b_{i\tilde{c}} + W_{h\tilde{c}}h_{t-1} + b_{h\tilde{c}})$$

$$o_t = \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho})$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$

$$h_t = o_t \odot \tanh(c_t)$$

# LSTM Example

[https://colab.research.google.com/drive/1PLD8HW5akfugMZ81q6GIYFdei2W\\_al8F?usp=sharing](https://colab.research.google.com/drive/1PLD8HW5akfugMZ81q6GIYFdei2W_al8F?usp=sharing)

# Outline

- Recurrent Neural Network Concepts
- Long Term Short Term Memory (LSTM)
- **Gated Recurrent Unit (GRU)**
- Bidirectional RNN
- Deep RNN

# Gated Recurrent Unit (GRU)

- Gated recurrent unit [Chung '14] is the more recent version of mitigating the vanishing gradient problem of RNN.
- GRU is less complex than LSTM and does not have a cell state. Since there is no cell state, its input  $(x_t, h_{t-1})$  and output  $(h_t)$  parameters are the same as simple RNN.
- GRU has two gates that use the Sigmoid function: **reset gate** (r) and **update gate** (z).
- The reset gate decides how much of the previous state to remember. It is responsible for **short-term** dependencies of a given sequence.
- The update gate is similar to the forget gate and input gate of LSTM, and it decides how much information to throw away. It is responsible for **long-term** dependencies of the given sequence.



# GRU Operation (Step 1)

1. The input  $x_t$  and output of the previous time  $h_{t-1}$  will be used to calculate the value for “update” and “reset” gates. They both use the *Sigmoid* function and their equations will be written as follows:

$$z = \sigma(x_t W_{xz} + h_{t-1} W_{hz} + b_z)$$

$$r = \sigma(x_t W_{xr} + h_{t-1} W_{hr} + b_r)$$

if  $z$  is close to 1, it means that the new hidden value  $h_t$  will be very similar to the old hidden value  $h_{t-1}$ . This reveals the fact that lots of the new information are a copy of old information. In contrast,  $z$  close to 0, indicates the new hidden value will be close to the candidate hidden state  $\tilde{h}_t$ .

# GRU Operation (Step 2)

2. Now having the result of both gates we can calculate the candidate hidden state ( $\tilde{h}_t$ ), but it is not the final hidden state.

The candidate's hidden state should stay between -1 and 1 interval. Thus a *tanh* gate is required. Besides, it incorporates the reset gate information, which will be used as an element-wise product of the previous hidden state ( $h_t$ ) and  $r$ .

The candidate's hidden state equation is written as follows:

$$\tilde{h}_t = \tanh(x_t W_{xh} + (r \odot h_{t-1}) W_{hh} + b_h)$$

# GRU Operation (Step 3)

3. After the candidate hidden value ( $\tilde{h}_t$ ) and update gate ( $z$ ) are both identified, the algorithm calculates the hidden state as  $h_t = z \odot h_{t-1} + (1 - z) \odot \tilde{h}_t$ .

The output of GRU is only  $h_t$  and unlike LSTM it does not have a cell state

# GRU Equations

$$r_t = \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{t-1} + b_{hr})$$

$$z_t = \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{t-1} + b_{hz})$$

$$\tilde{h}_t = \tanh(W_{in}x_t + b_{in} + r_t \odot (W_{hn}h_{t-1} + b_{hn}))$$

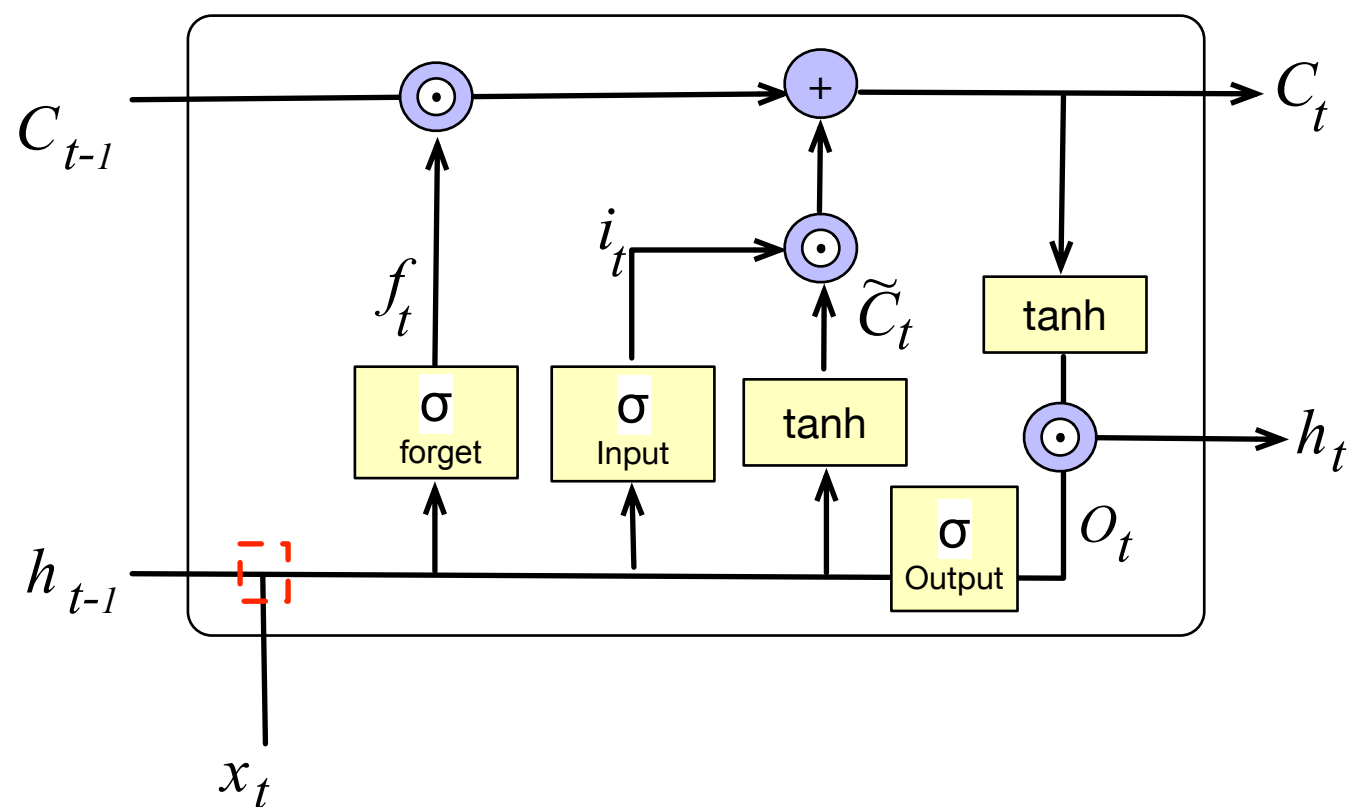
$$h_t = (1 - z_t) \times \tilde{h}_t + z_t \odot h_{t-1}$$

# GRU Example

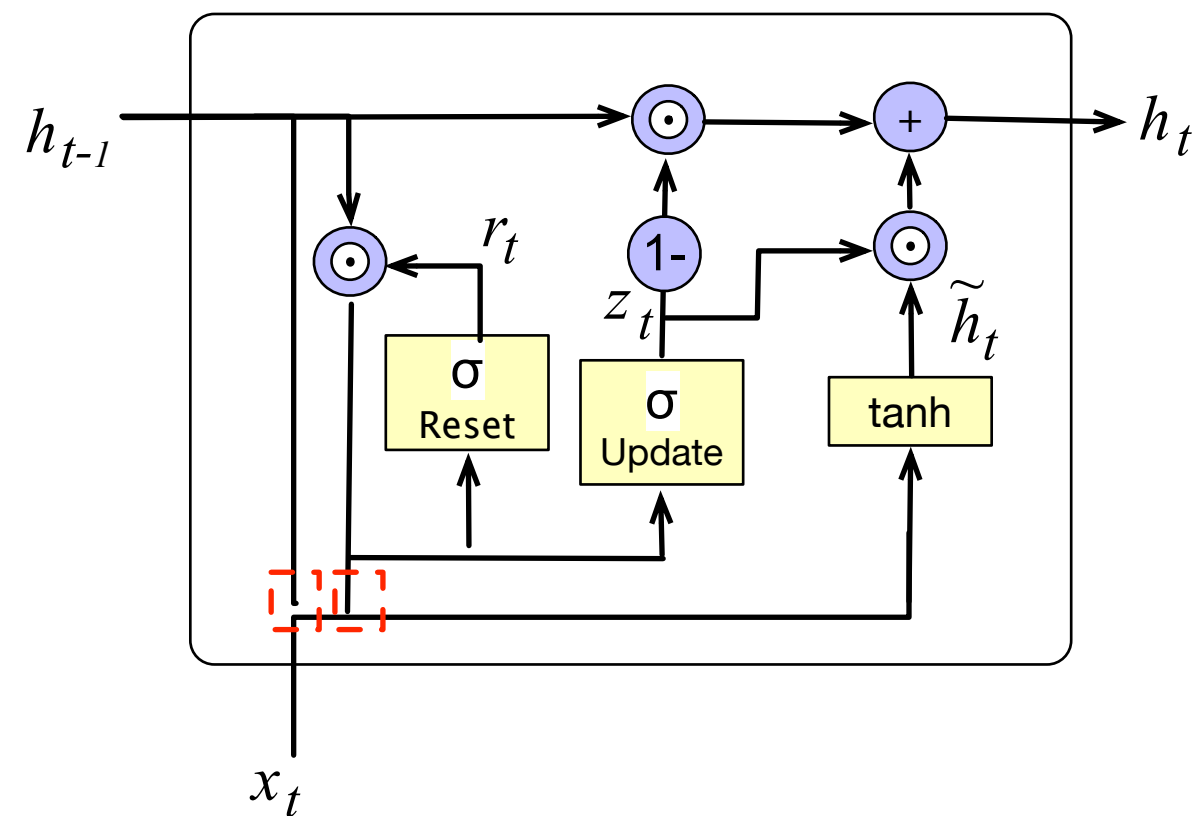
<https://colab.research.google.com/drive/1uSfQzaihF6zfOAV9Nwg--0-VxkOYG2s-?usp=sharing>

[https://colab.research.google.com/drive/1Wjc76Ub\\_u9tmMjY0bKOftYC2KOqKSRB3?usp=sharing](https://colab.research.google.com/drive/1Wjc76Ub_u9tmMjY0bKOftYC2KOqKSRB3?usp=sharing)


## LSTM



## GRU



 Elementwise operator

 Activation function

 Concatenation

# Outline

- Recurrent Neural Network Concepts
- Long Term Short Term Memory (LSTM)
- Gated Recurrent Unit (GRU)
- **Bidirectional RNN**
- Deep RNN

# Bidirectional RNN

- One of the popular problems in NLP applications is referred to as name-entity recognition.
- It refers to a problem in that we need to identify a word that is referring to what entity (person, object, organization, etc.) in a sentence.
- RNN is a useful approach for named entity recognition.

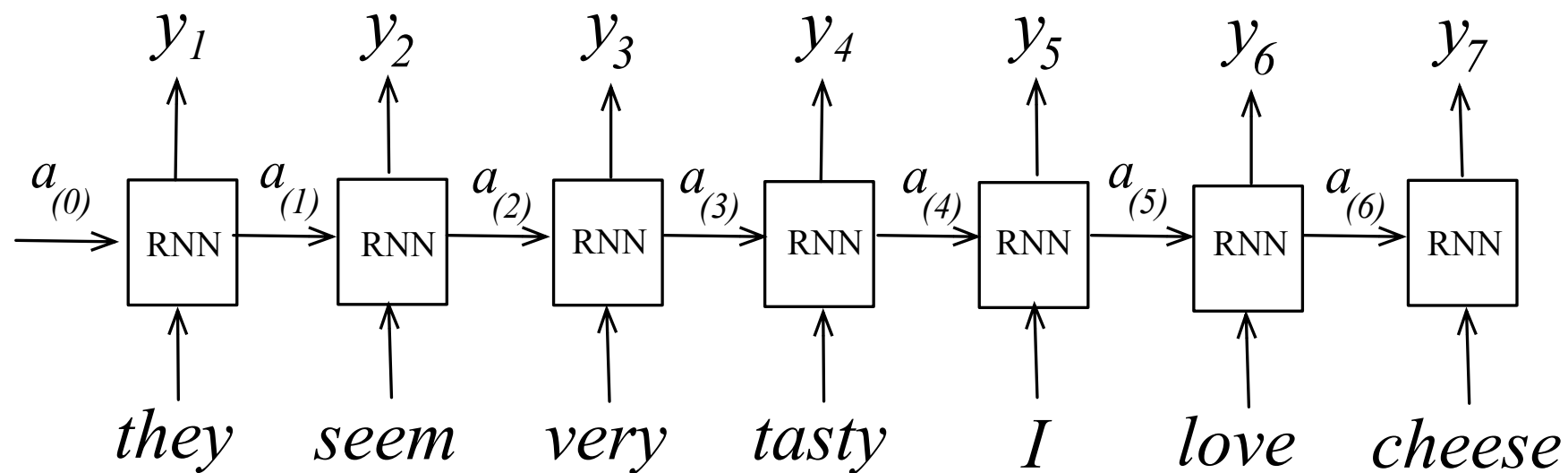


# Problem with RNN on Name Entity Recognition

- “They seem very tasty.” We do not know what “They” mean.
- It refers to a problem that we need to identify a word that is referring to what entity (person, object, organization, etc.) in a sentence.
- RNN is a useful approach for named entity recognition.

# Problem with RNN on Name Entity Recognition

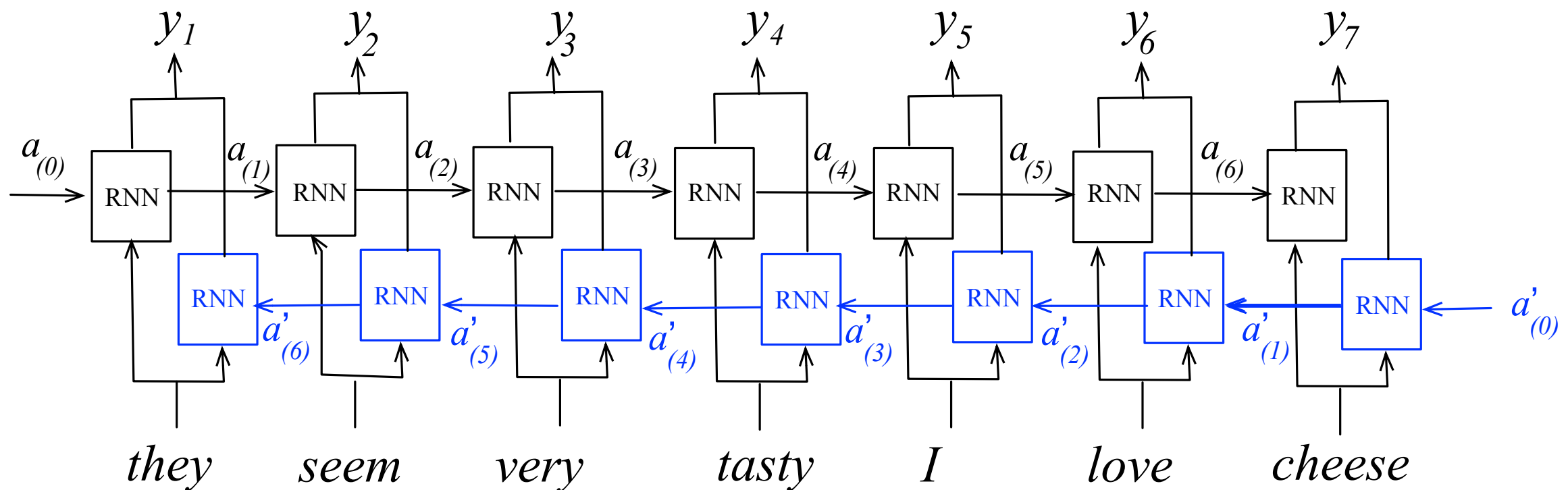
- “They seem very tasty, I love cheese.”
- Now, we can understand that “They” refers to food with cheese or cheese itself.
- A simple RNN cell can not incorporate prior knowledge, before encountering the word “cheese”, because every new state got the information from its previous state, and in this particular sentence, we realize they are referring to food with cheese later in the sentence.



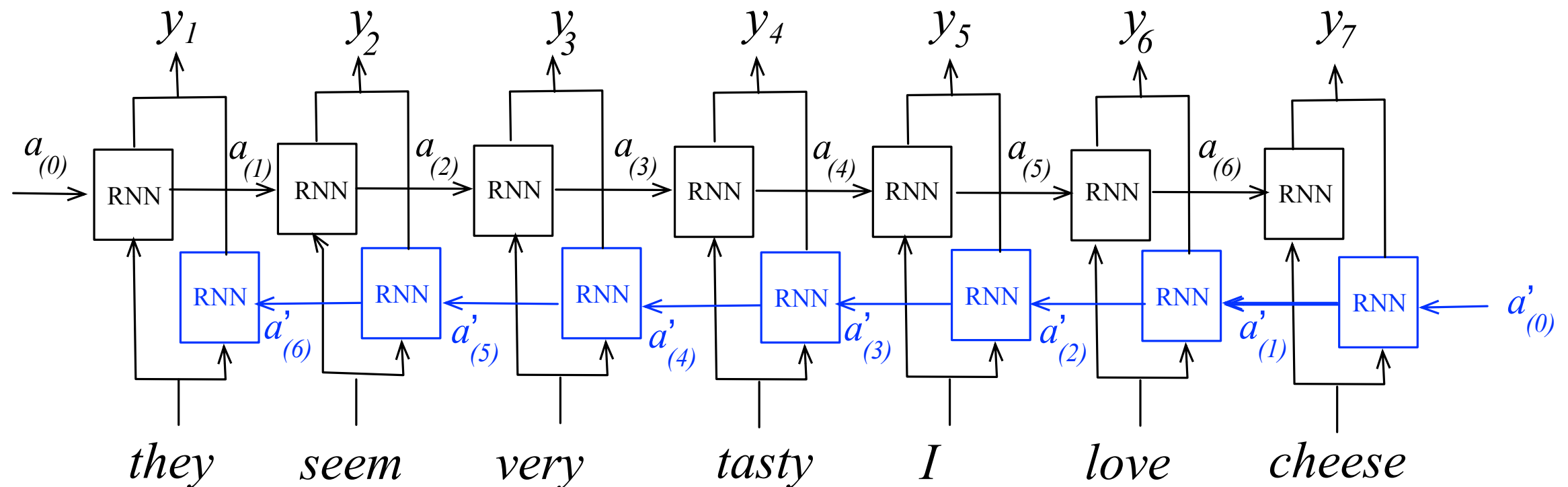
Since the word, “cheese” happened at the  $t_7$  and “They” at  $t_1$ , the RNN can not understand what “They” refers to at any time earlier than  $t_7$ .

# Bi-directional RNN

- A solution to handle this is to incorporate future information in the sequence for predicting data in a sequence. Bidirectional RNN is designed to *incorporate future data into the sequence* as well. It introduces another layer, with exactly the same size, but the direction of activations is in opposite direction (from the end of the sequence to the beginning of the sequence).



# Bi-directional RNN



- This network starts from  $a_{(0)}$  and goes forward until it computes  $a_{(6)}$ , simultaneously it starts from the  $a'_{(1)}$  (blue RNN) and moves backward by computing activations until it reaches  $a'_{(6)}$ . Therefore, assuming our sequence has a size of  $T$ , every  $y_t$  output includes both forward (from 0 to  $t$ ) and backward activation (from  $T$  to  $t$ ) data (simultaneously).

# Bidirectional RNN Example

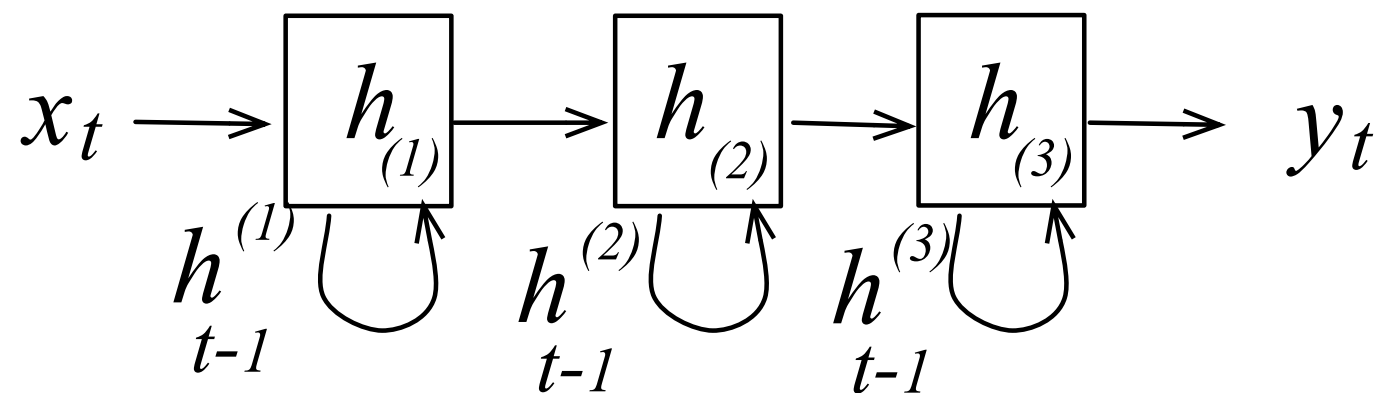
[https://colab.research.google.com/drive/  
1MJxXNV\\_mrtYknPpCq5jZEcPT30hRzAgm?usp=sharing](https://colab.research.google.com/drive/1MJxXNV_mrtYknPpCq5jZEcPT30hRzAgm?usp=sharing)

# Outline

- Recurrent Neural Network Concepts
- Long Term Short Term Memory (LSTM)
- Gated Recurrent Unit (GRU)
- Bidirectional RNN
- **Deep RNN**

# Deep RNN

- If we stack multiple layers of RNN on top of each other, we will refer to this network as Deep RNN.



Note, in this figure, unlike the figure at the beginning, has three hidden layers instead of one hidden layer that is unfolded.