CSCI 1470

Eric Ewing

Friday, 3/7/25

Deep Learning

Day 18: RNNs and LSTMs

Some Game Theory

Other Updates

- Will try to handle point deductions over the weekend
- For those who didn’t come forward:
  - I will (eventually) look at ***all*** submissions and look for known AI patterns and send cases I’m certain of.
  - This won’t be quick… I’ll prioritize seniors and masters students first

Recurrent Neural Network (RNN)

Recurrent Neural Networks are networks in the form of a directed ***cyclic*** graph.

They pass previous *state* information from previous computations to the next.

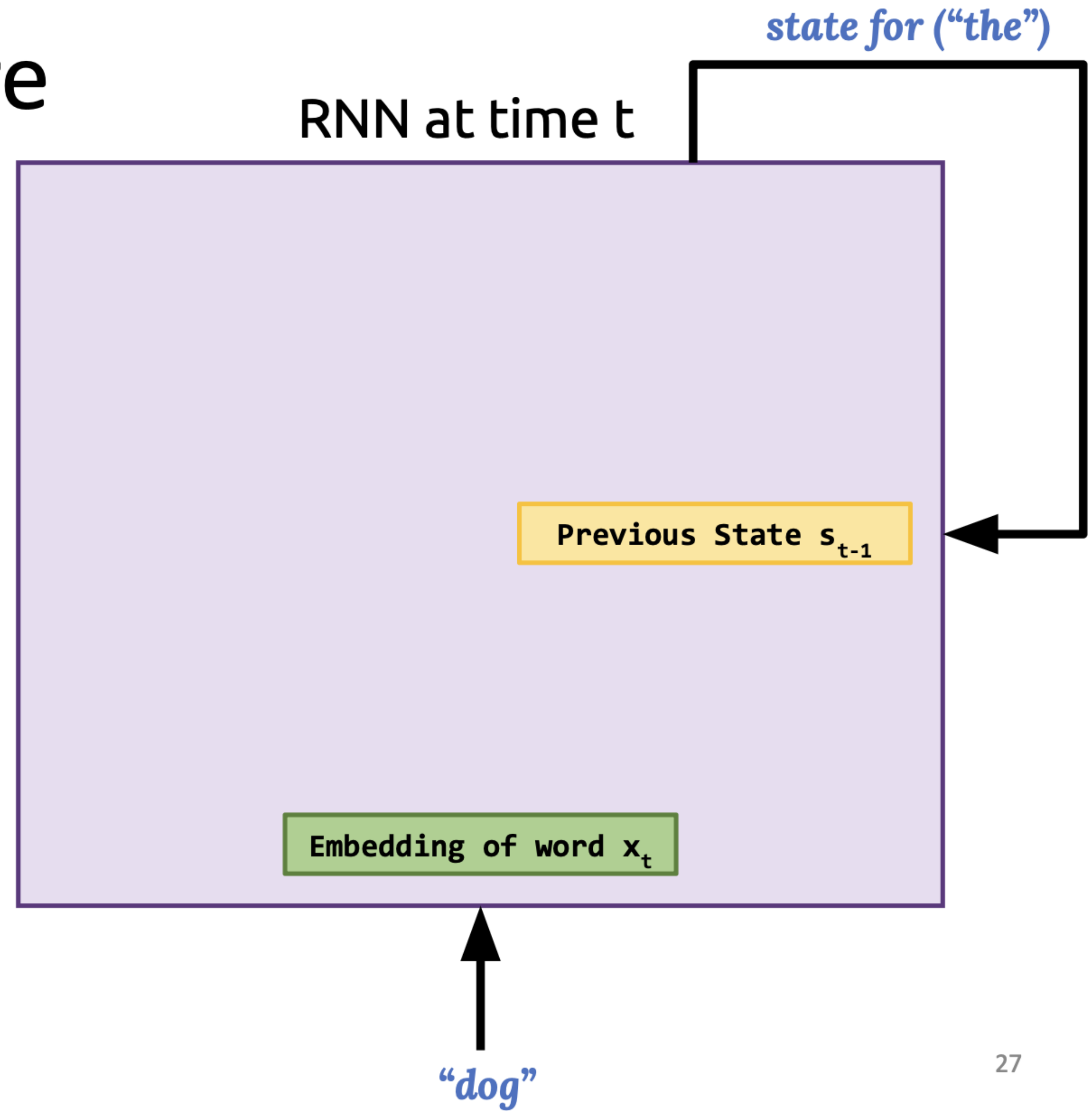
They can be used to process sequence data with relatively low model complexity when compared to feed forward models.

The block of computation that feeds its own output into its input is called the *RNN cell.*

Let's see how we can build one!

RNN Cell Architecture

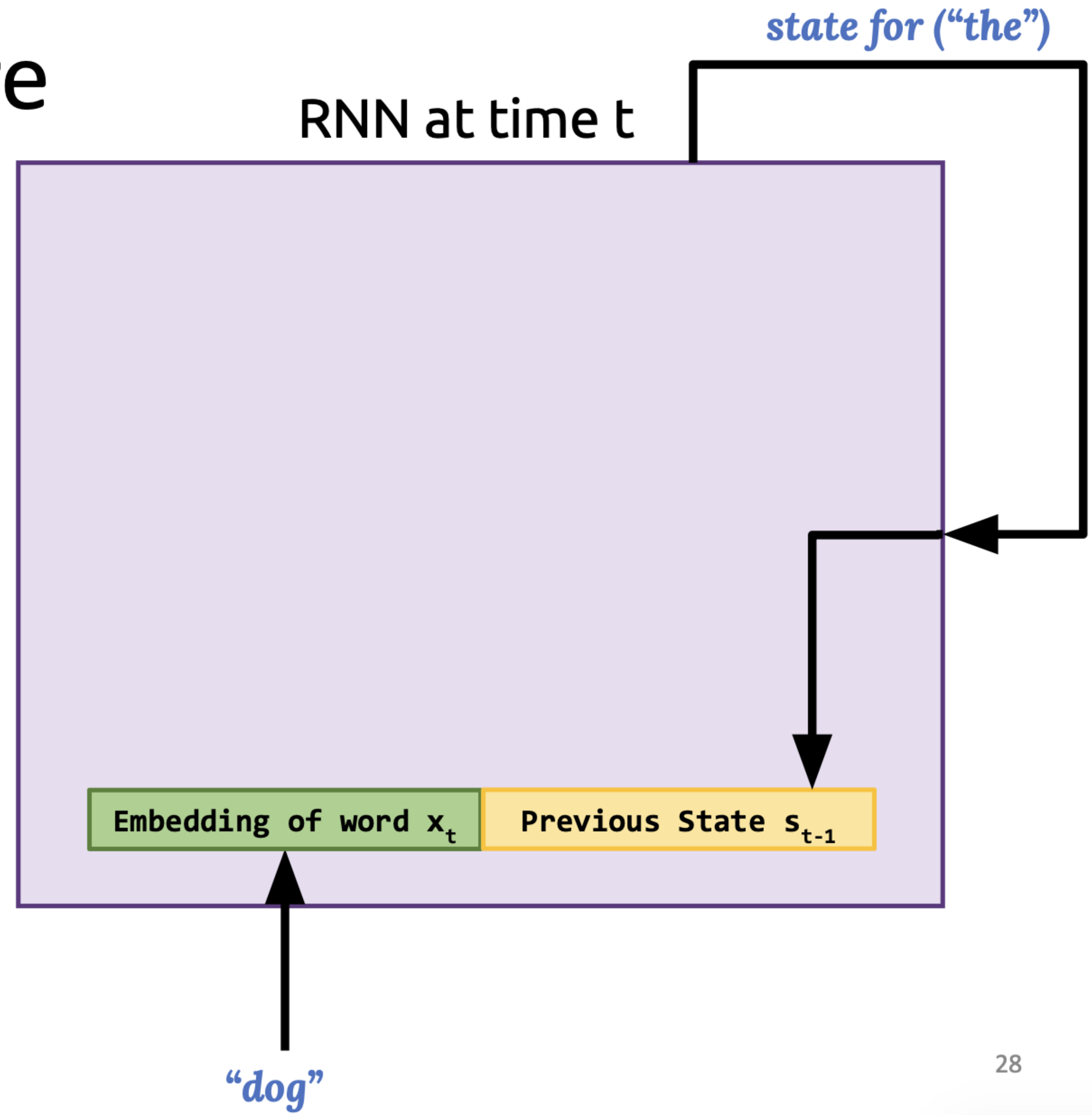
At each step of our RNN, we will get an input word, and a state vector from the previous cell.



RNN Cell Architecture

At each step of our RNN, we will get an input word, and a state vector from the previous cell.

We then concatenate the embedding and state vectors.

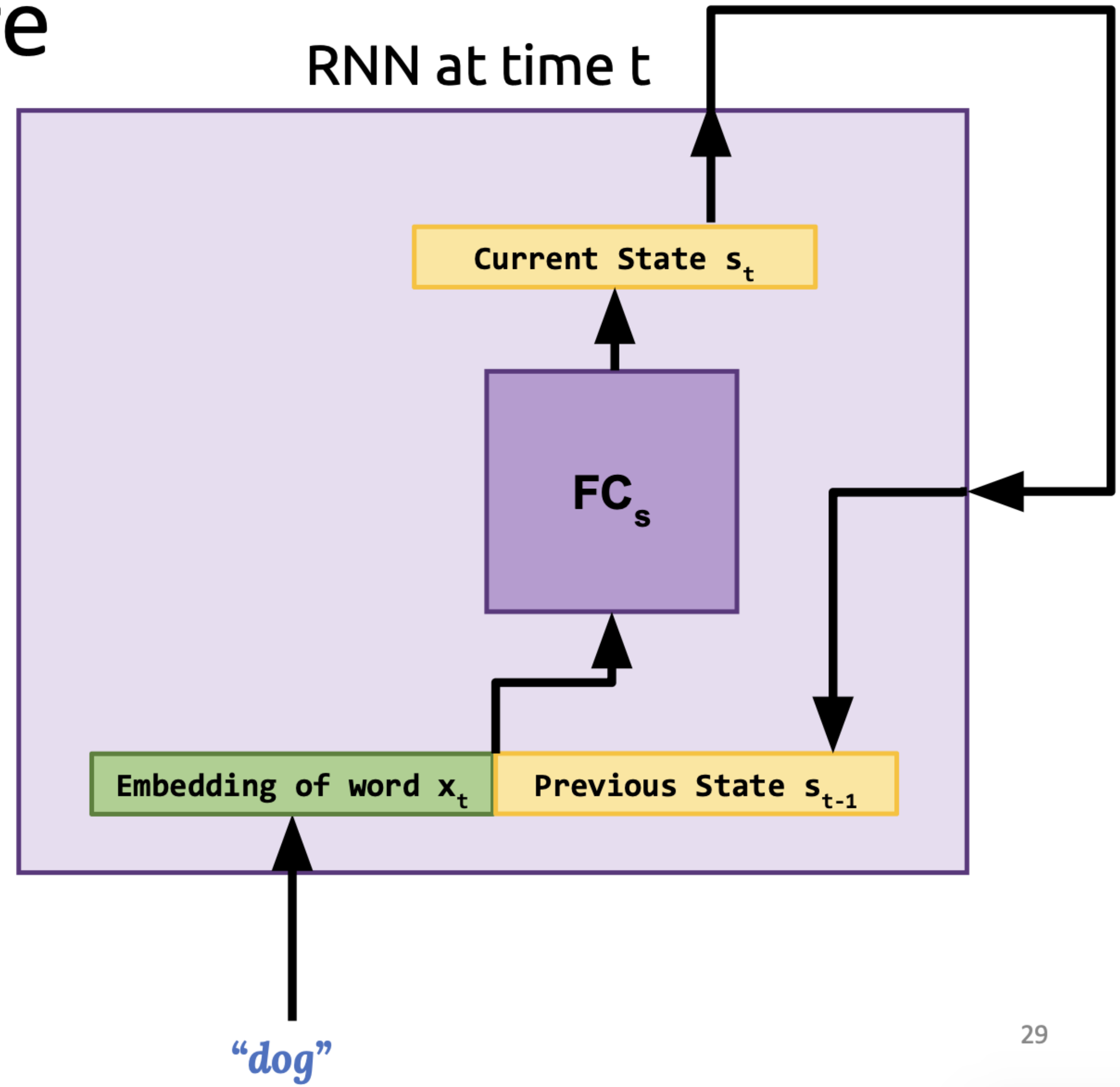


RNN Cell Architecture

At each step of our RNN, we will get an input word, and a state vector from the previous cell.

We then concatenate the embedding and state vectors.

We use a fully connected layer to compute the next state



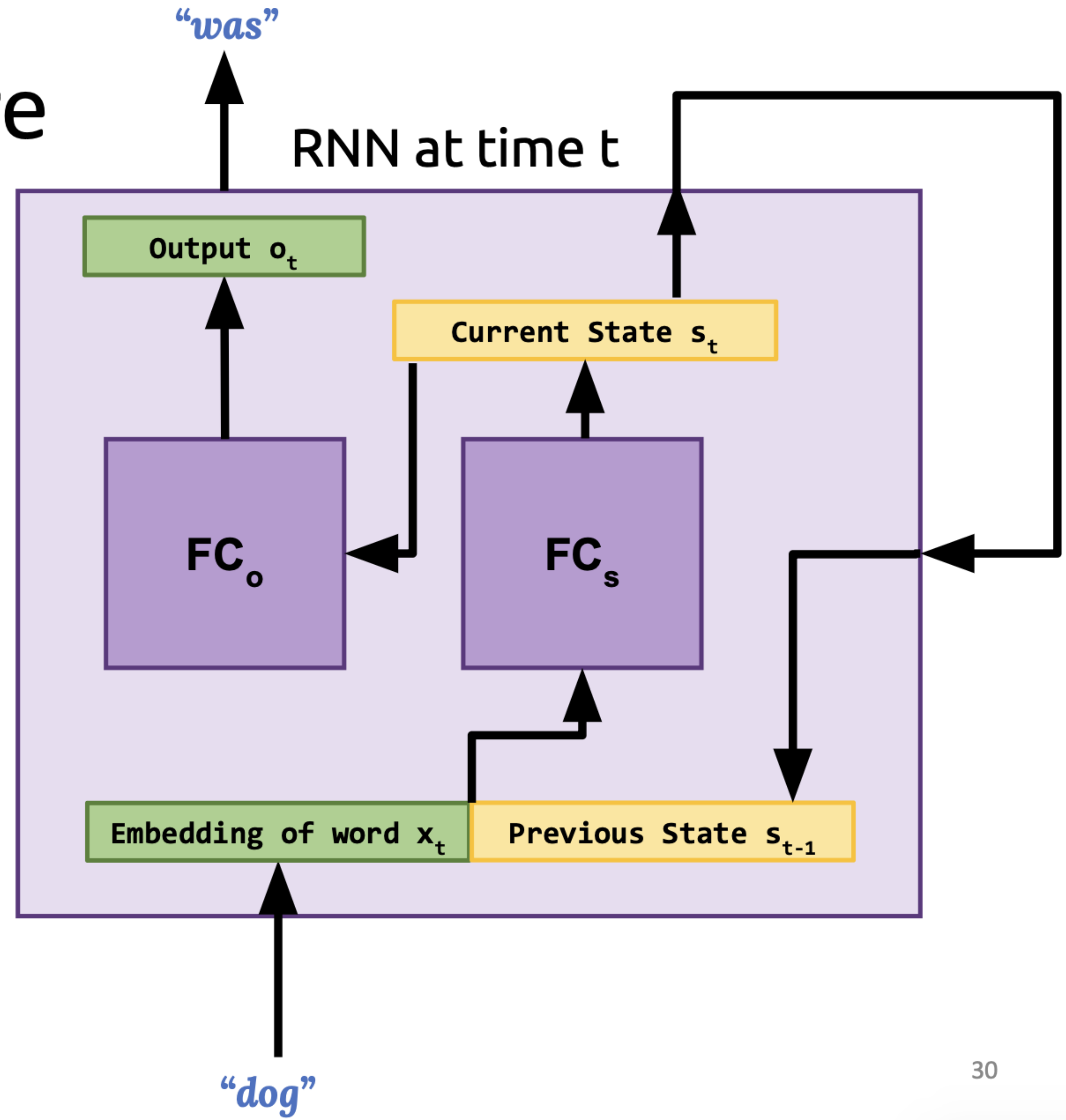
RNN Cell Architecture

At each step of our RNN, we will get an input word, and a state vector from the previous cell.

We then concatenate the embedding and state vectors.

We use a fully connected layer to compute the next state

We use another connected layer to get the output.

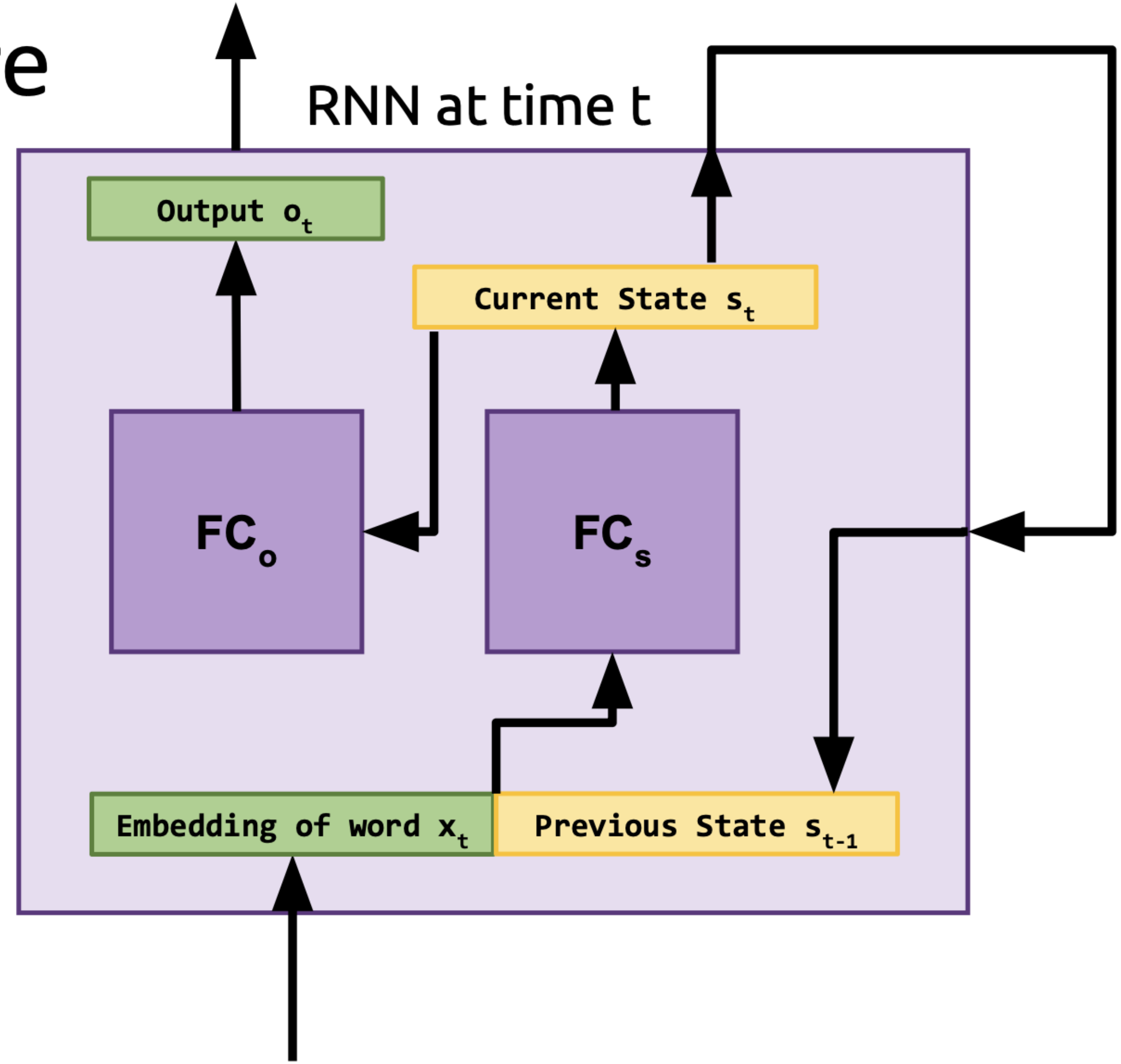


RNN Cell Architecture

We can represent the RNN in with the following equations:

$$s_t = \rho\big((e_t, s_{t-1})W_r + b_r\big)$$

$$o_t = \sigma(s_t W_o + b_o)$$



RNN Cell Architecture

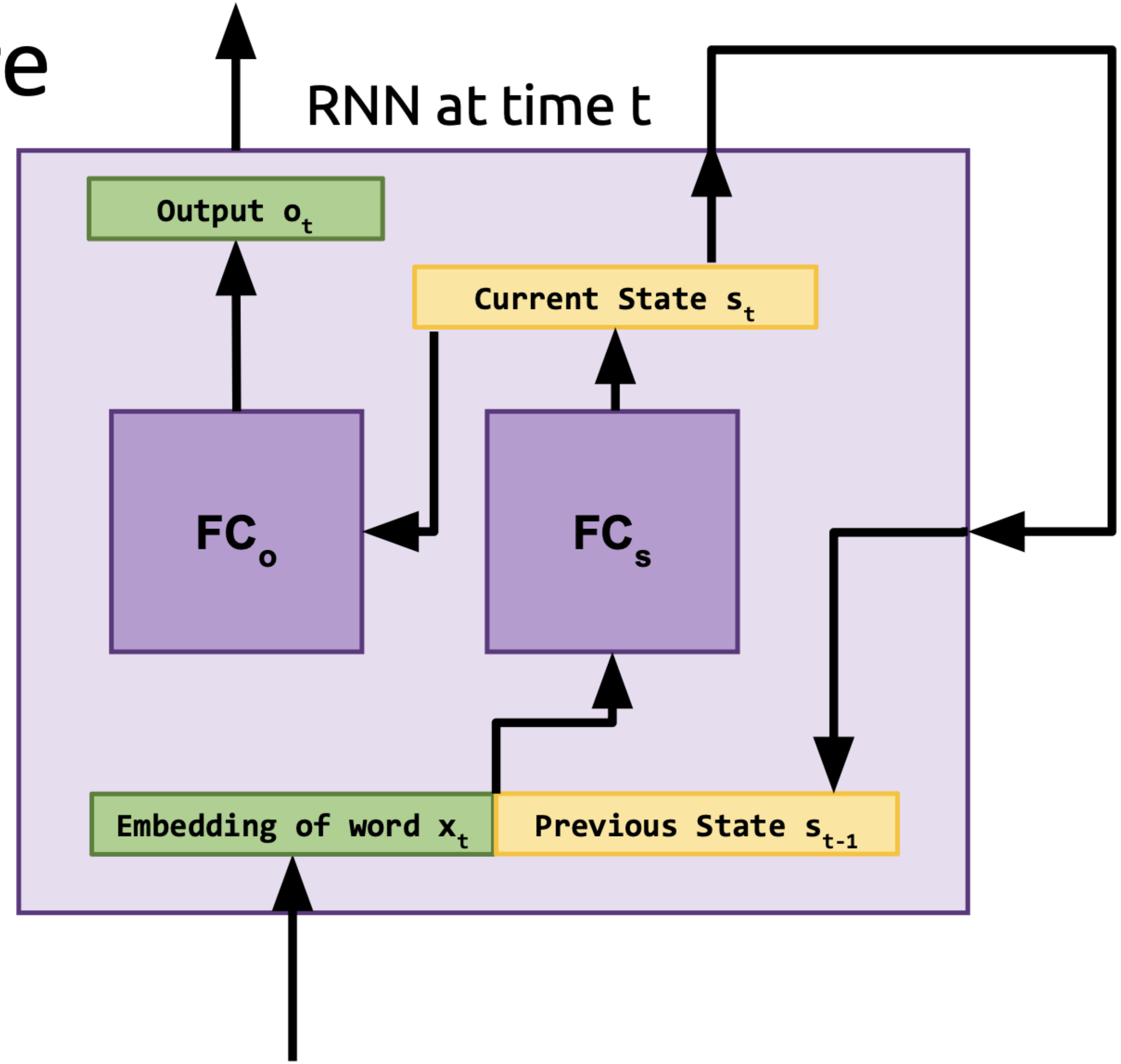
We can represent the RNN in with the following equations:

$$s_t = \boxed{\rho}\big((e_t, s_{t-1})W_r + b_r\big)$$

$$o_t = \boxed{\sigma}(s_t W_o + b_o)$$

Nonlinear activations (e.g. sigmoid, tanh)

Any questions?



RNN Cell Architecture

We can represent the RNN in with the following equations:

$$s_t = \rho\big((e_t, s_{t-1})W_r + b_r\big)$$

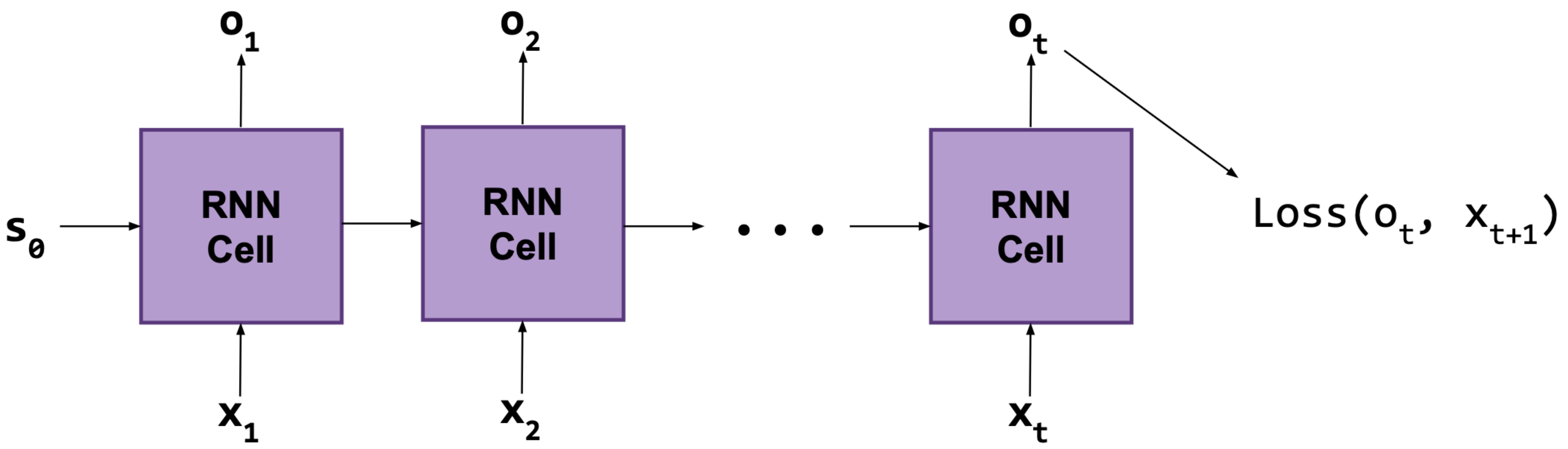
$$o_t = \sigma(s_t W_o + b_o)$$

This brings up an immediate question: **what is $\boldsymbol{s_0}$?**

Typically, we initialize s_0 to be a vector of zeros (i.e. “initially, there is no memory of any previous words”)

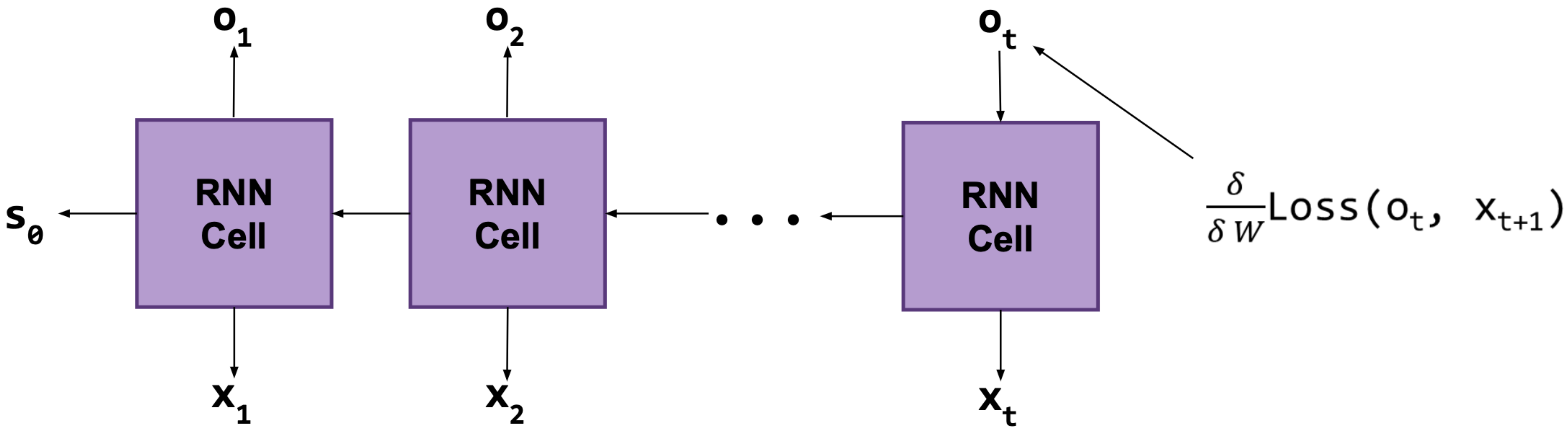
Training RNNs

We can calculate the cross entropy loss just as before since for any sequence of input words (x_1, x_2,..., x_t), we know the true next word x_{t+1}



Training RNNs

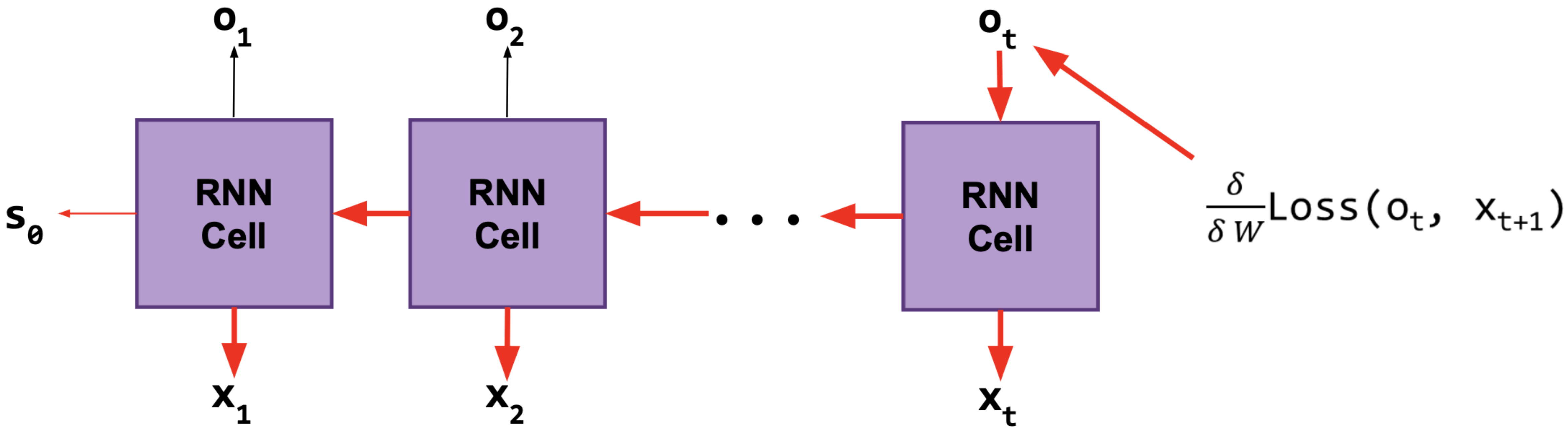
But what happens when we differentiate the loss and backpropagate?



Training RNNs

Not only do our gradients for o_t depend on x_t, but also on all of the previous inputs.

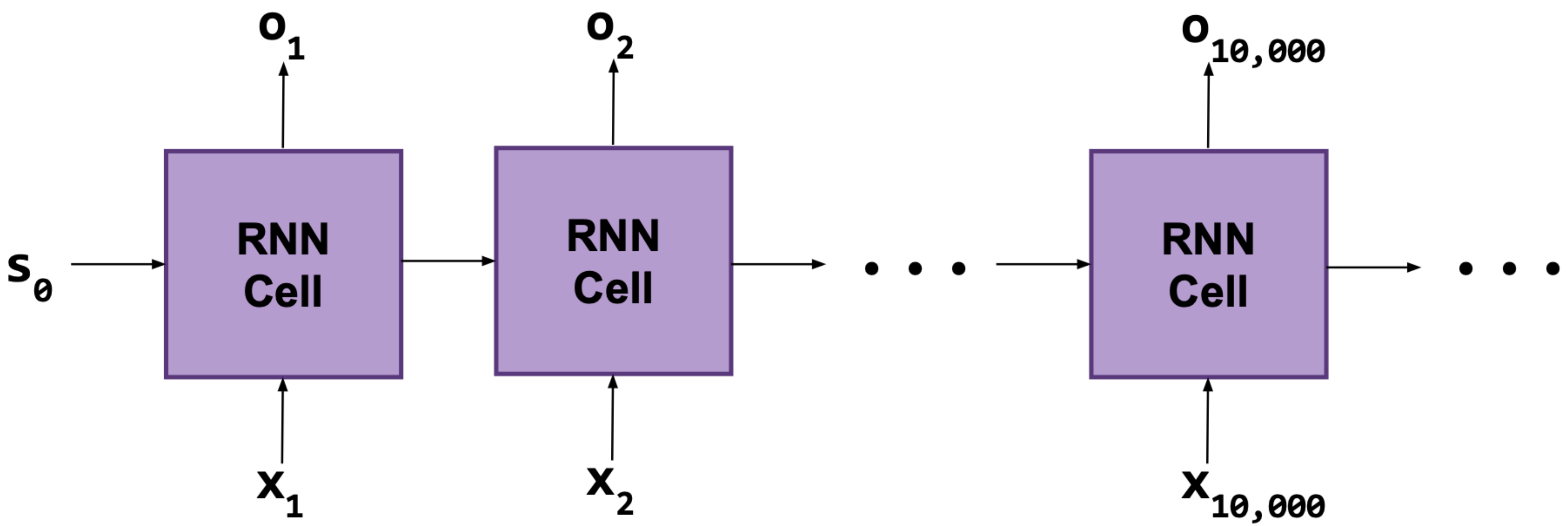
We call this *backpropagation through time.*



Training RNNs

But at what point do we stop and calculate the loss/update?

With this architecture, we can run the RNN cell for as many steps as we want, constantly accumulating memory in the state vector.



Training RNNs

Solution: We define a new hyperparameter called `window_sz`.

We now chop our corpus into sequences of words of size `window_sz`

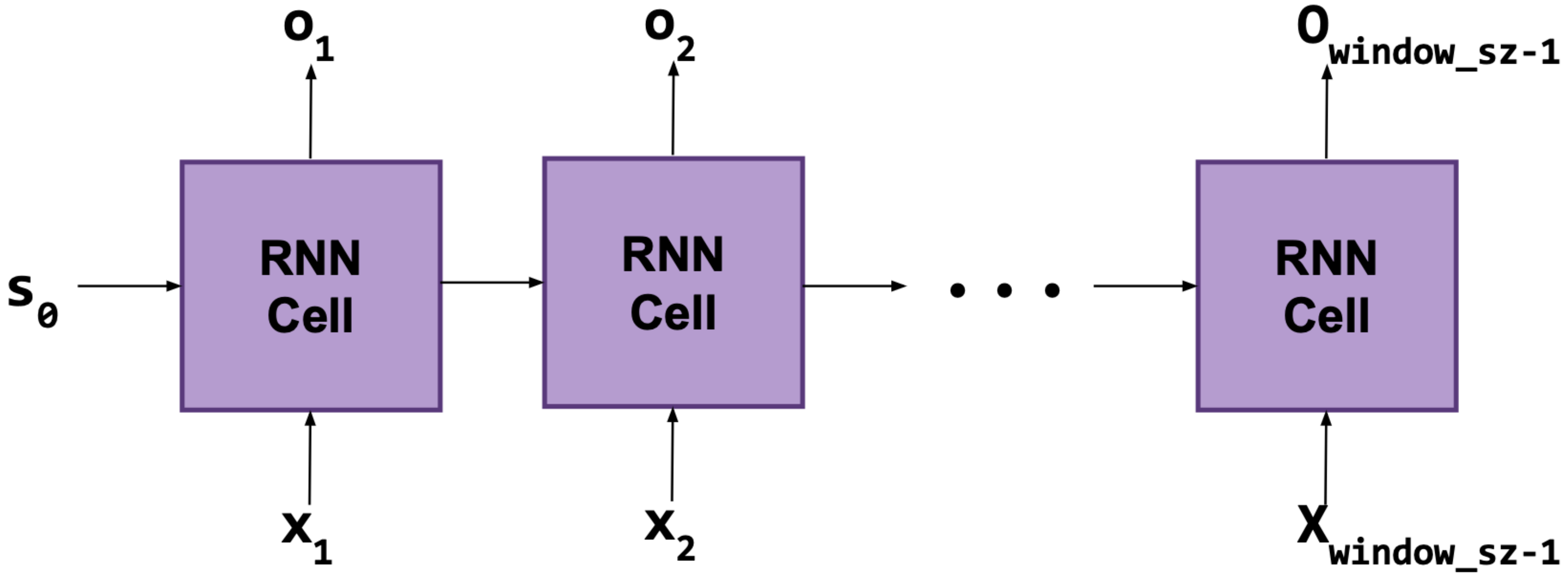
The new shape of our data should be:

```
(batch_sz, window_sz, embedding_sz)
```

Each example in our batch is a “window” of `window_sz` many words. Since each word is represented as an `embedding_sz`, that is the last dimension of the data.

Training RNNs

Now that every example is a window or words, we can run the RNN till the end of that window, and compute the loss for that specific window and update our weights

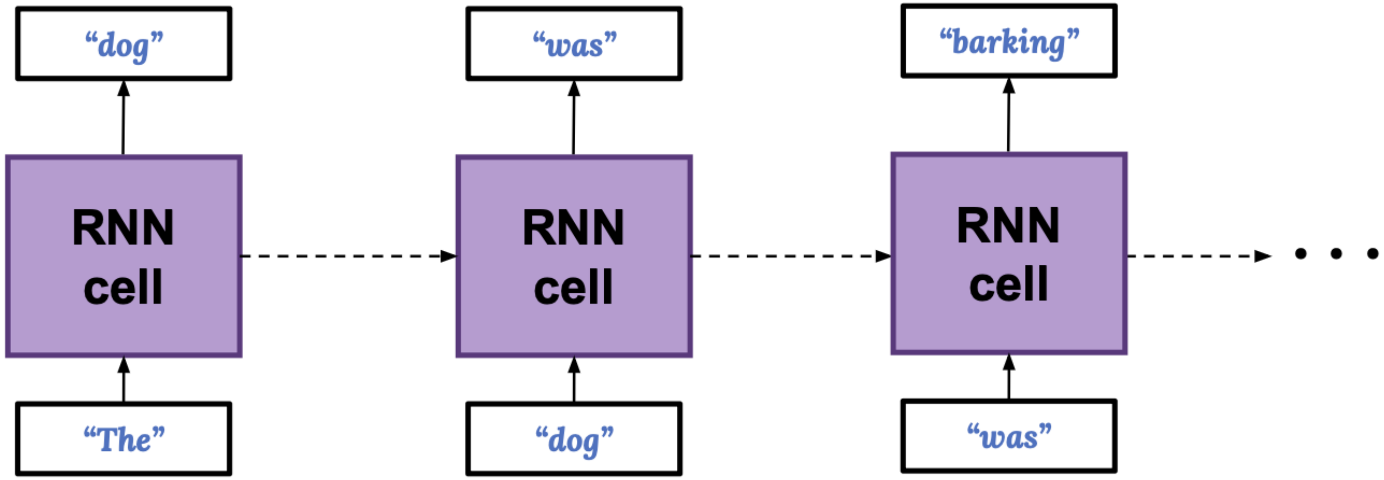


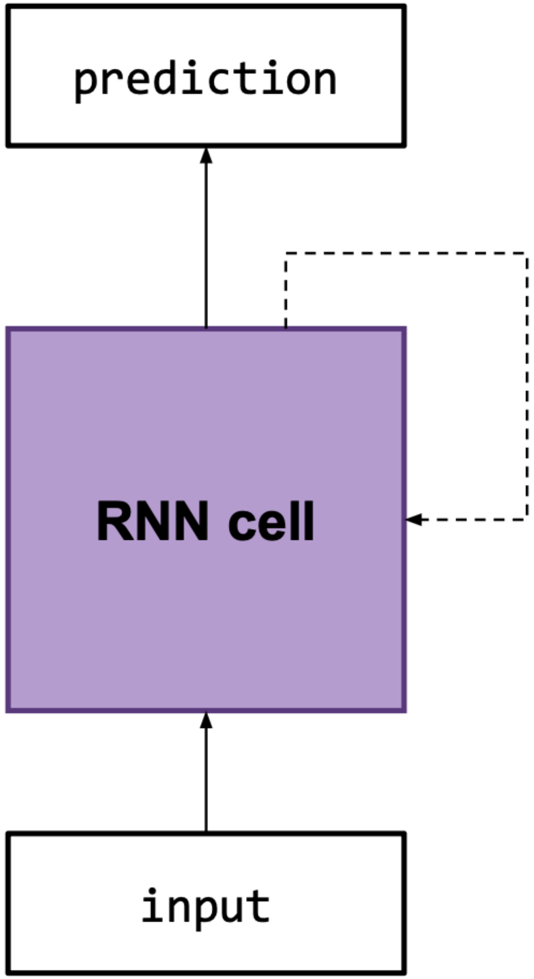
Any questions?

Does RNN fix the limitations of the N-gram model?

1. Number of of weights not dependent on N
2. State gives flexibility to choose context from near or far

"The dog was barking at one of the __cats__."





RNNs in Tensorflow

RNNs can be built from scratch using Python `for` loops:

```
prev_state = Zero vector
for i from 0 to window_sz:
   state_and_input = concat(inputs[i], prev_state)
   current_state = fc_state(state_and_input)
   outputs[i] = fc_output(current_state)
   prev_state = current_state
return outputs
```

RNNs in Tensorflow

RNNs can be built from scratch using Python `for loops`.

There's also a handy built-in Keras recurrent layer:

```
tf.keras.layers.SimpleRNN(units, activation, return_sequences)
```

RNNs in Tensorflow

RNNs can be built from scratch using Python `for loops.`

There's also a handy built-in Keras recurrent layer:

```
tf.keras.layers.SimpleRNN(units, activation, return_sequences)
```

The size of our output vectors

RNNs in Tensorflow

RNNs can be built from scratch using Python `for loops`.

There's also a handy built-in Keras recurrent layer:

```
tf.keras.layers.SimpleRNN(units, activation, return_sequences)
```

The activation function to be used in the FC layers inside of the RNN Cell

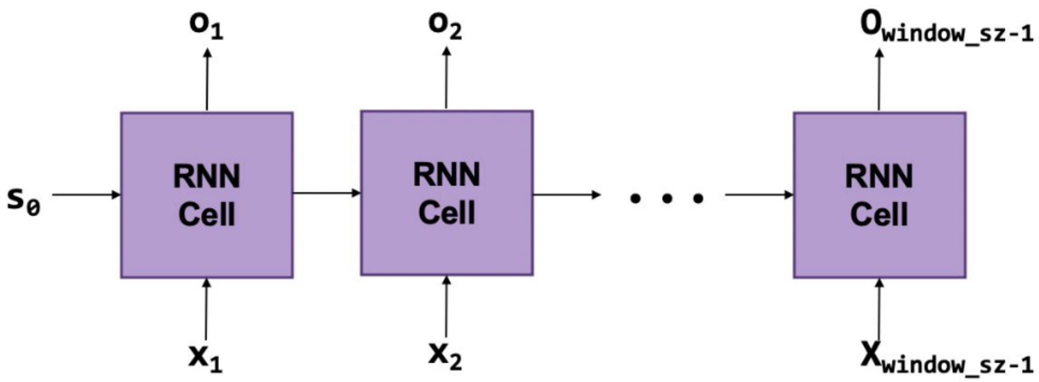
RNNs in Tensorflow

Any intuition why we would want return_sequences to be TRUE?

RNNs can be built from scratch using Python `for loops.`

There's also a handy built-in Keras recurrent layer:

```
tf.keras.layers.SimpleRNN(units, activation, return_sequences)
```



- If **True**: calling the RNN on an input sequence returns the whole sequence of outputs + final state output
- If **False**: calling the RNN on an input sequence returns just the final state output (Default)

RNNs in Tensorflow

RNNs can be built from scratch using Python `for loops`.

There's also a handy built-in Keras recurrent layer:

```
tf.keras.layers.SimpleRNN(units, activation, return_sequences)
```

Usage:

```
RNN = SimpleRNN(10) # RNN with 10-dimensional output vectors
Final_output = RNN(inputs) # inputs: a [batch_sz, seq_length, embedding_sz] tensor
```

RNN

"The dog that my family had when I was a child had a fluffy _____."

Want: ***"tail"***

RNN Weaknesses

But….RNNs are not very good at remembering things ***far*** in the past.

RNN Weaknesses

“The dog that my family had when I was a child had a fluffy _____.”

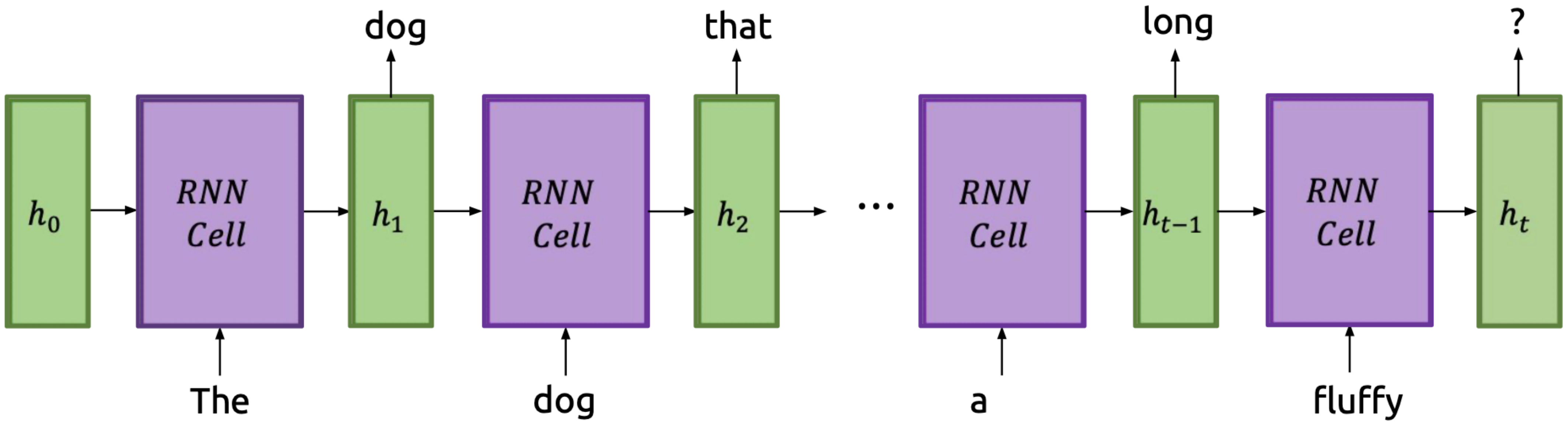
- To predict “tail” RNN needs to remember the subject of the sentence - “dog”

RNN Weaknesses

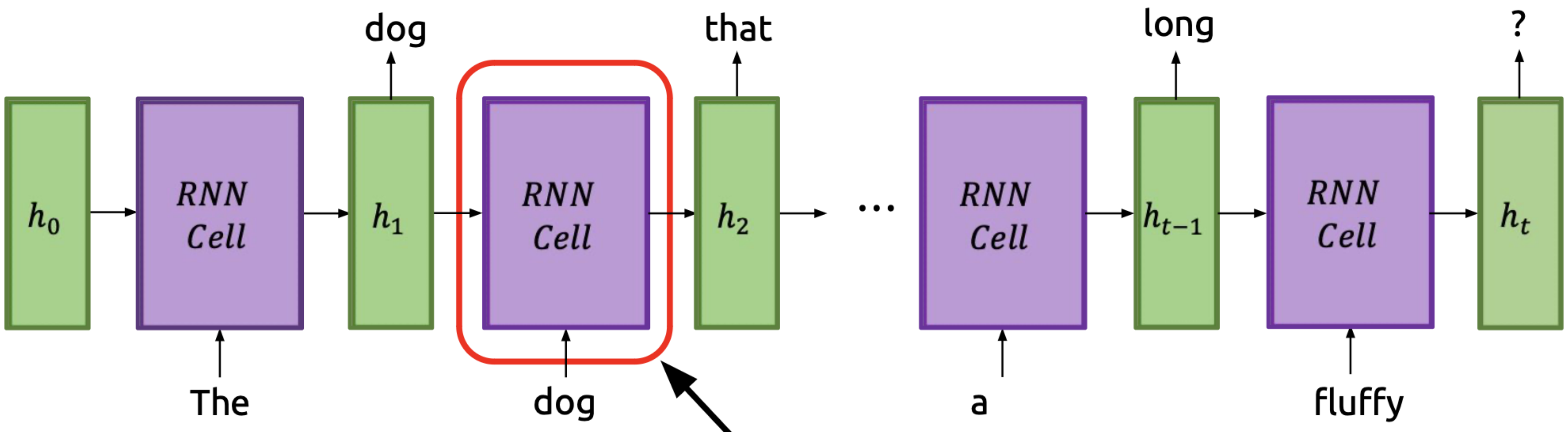
"The dog that my family had when I was a child had a fluffy ______."

- To predict "tail" RNN needs to remember the subject of the sentence - "dog"
- "dog" and predicted word are separated by **12** words
  - On the outer limit of what a vanilla RNN would be able to remember.

An Illustrative Example:

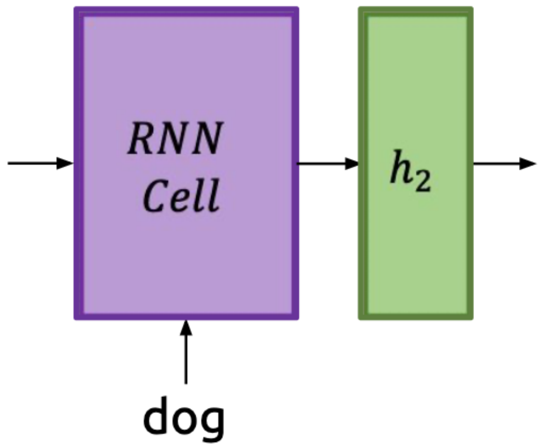


An Illustrative Example:



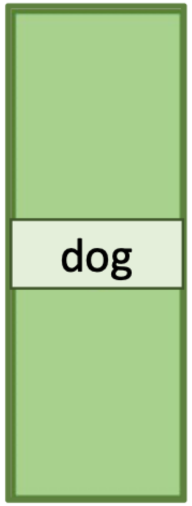
What happens to the information about “dog” as we continue through the network?

An Illustrative Example:



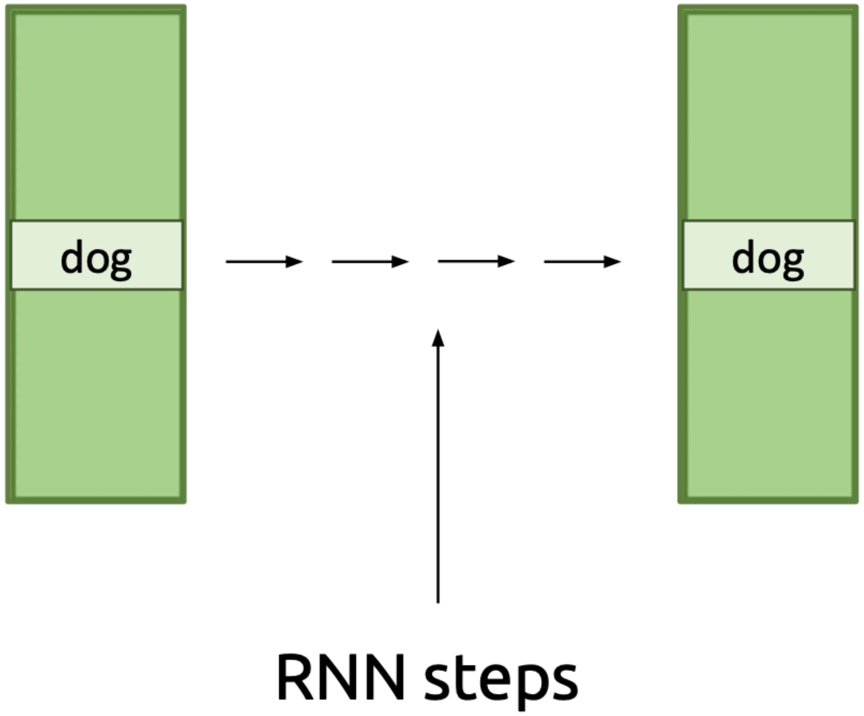
An Illustrative Example:

Can imagine that the information about *"dog"* is stored in some part of the RNN's hidden state vector



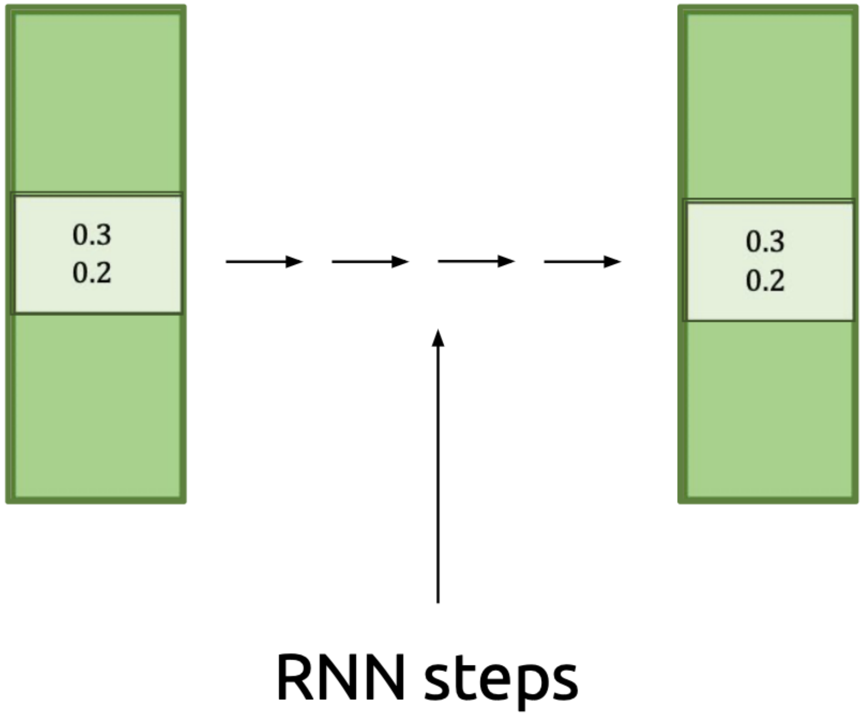
An Illustrative Example:

Through all subsequent RNN steps, we want *"dog"* to stay the same



An Illustrative Example:

If we think of *“dog”* as just a few entries in the vector…



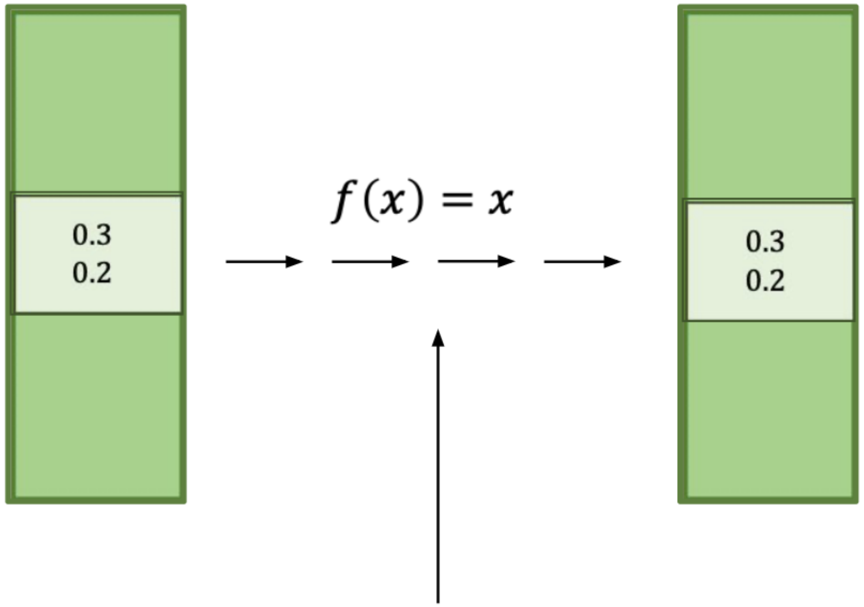
An Illustrative Example:

...to preserve *“dog”* , we need to compute the ***identity function*** over the part of the vector that stores it

But will that happen?

No

Why?



RNN steps

How does this affect the hidden state?

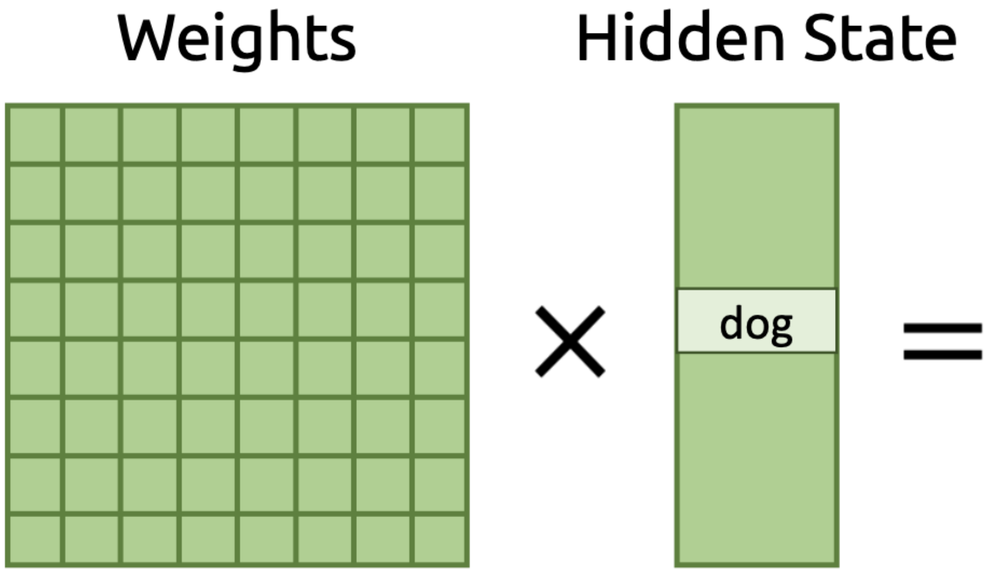
RNN update

$$h_t = \rho((e_t, h_{t-1})W_r + b_r)$$

The hidden state goes through a fully connected layer!

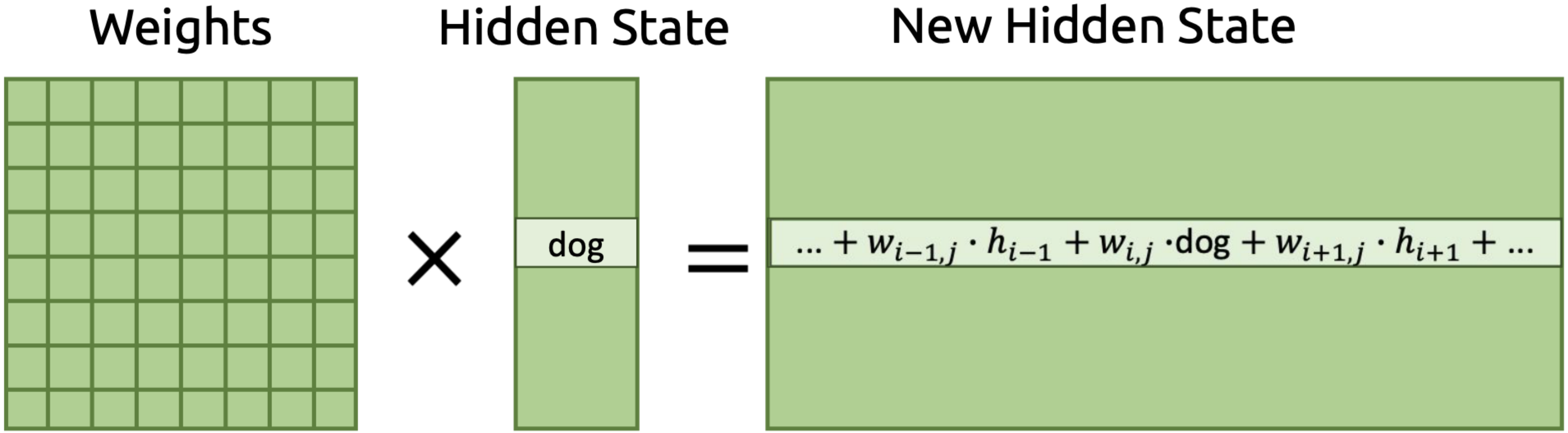
How does this affect the hidden state?

- What will happen to our dog after we multiply our weights by our hidden state?



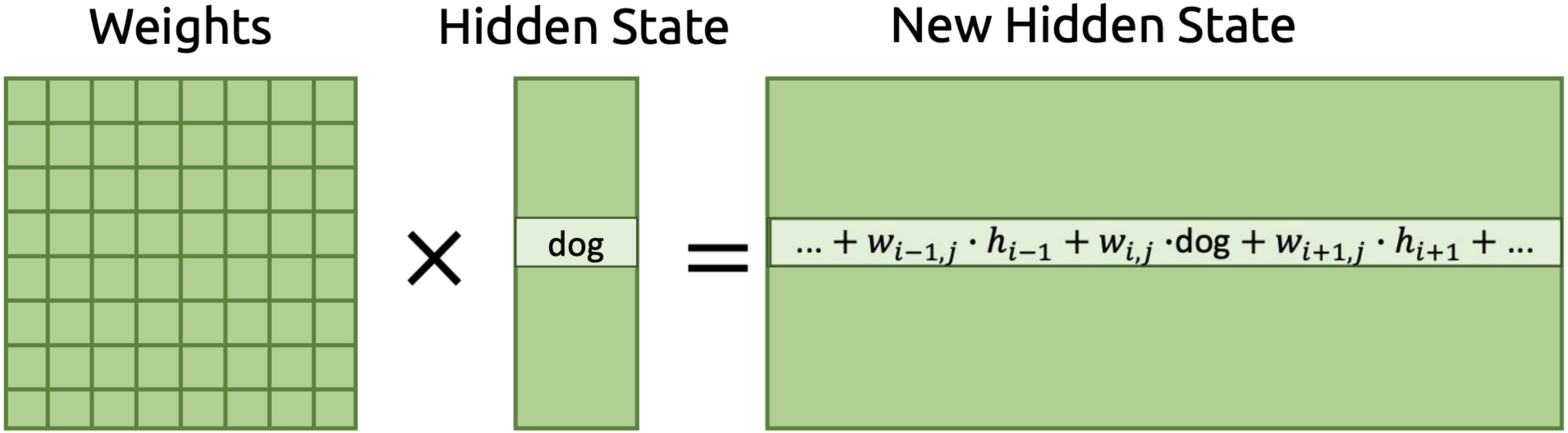
How does this affect the hidden state?

- What will happen to our dog after we multiply our weights by our hidden state?



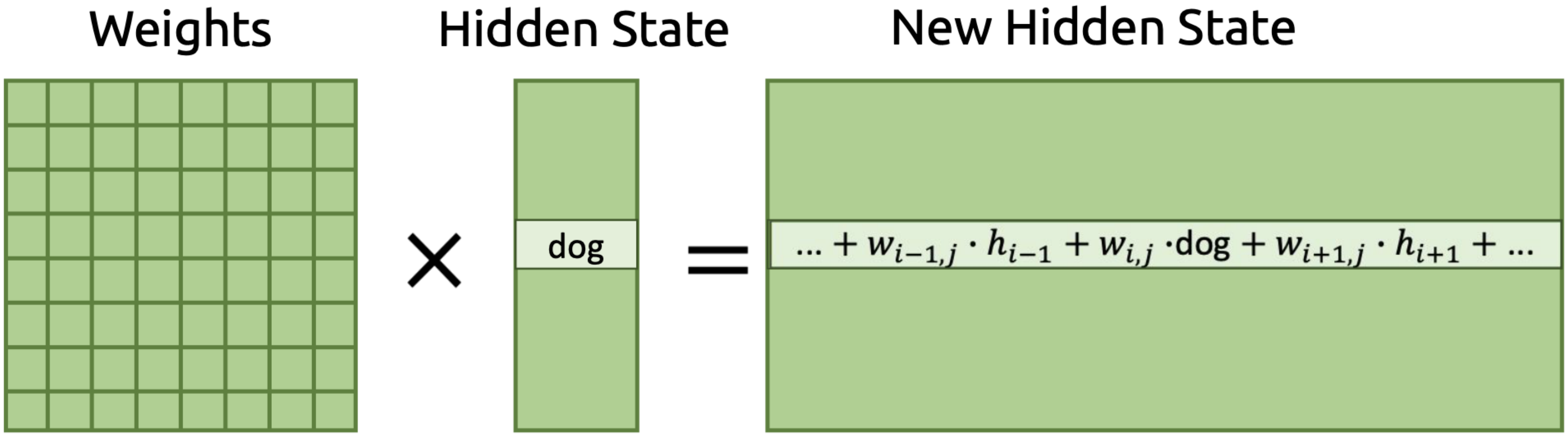
How does this affect the hidden state?

Dog gets lost in all the other information!



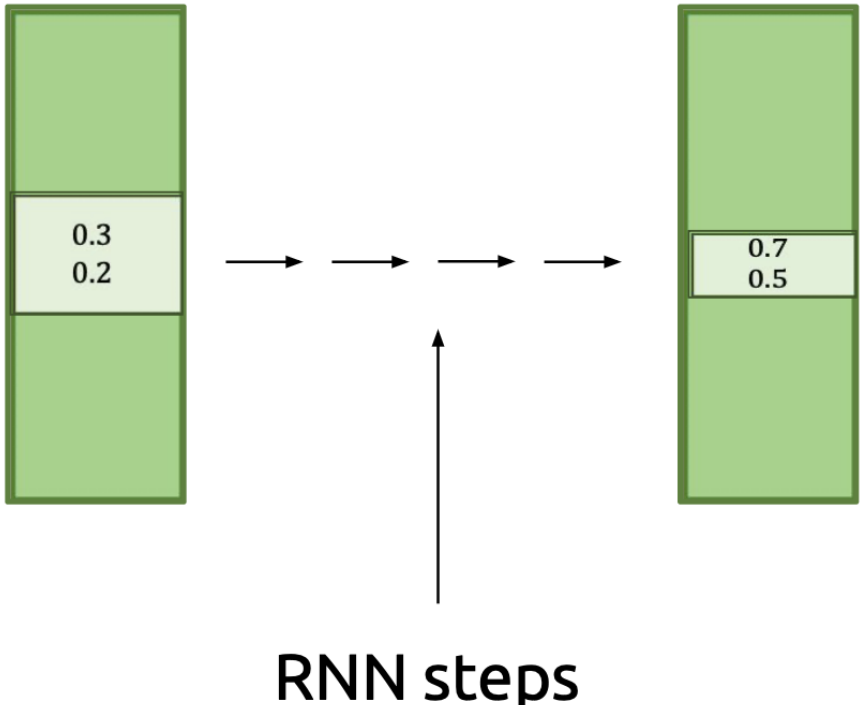
How does this affect the hidden state?

Dog gets lost in all the other information!



How does this affect the hidden state?

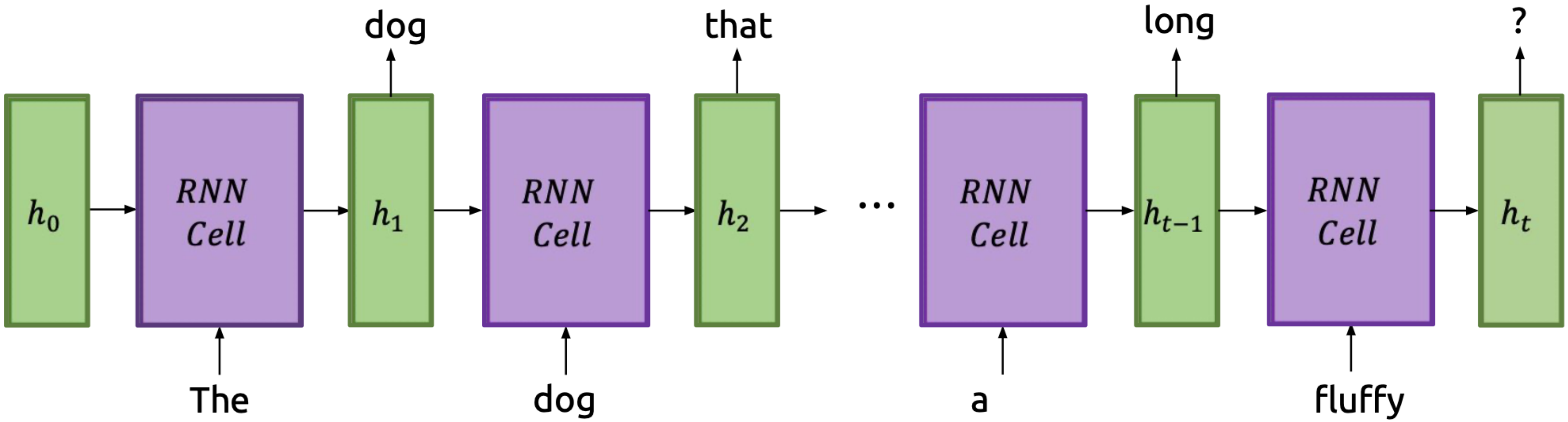
- “dog” in hidden state gets combined and mixed with rest of hidden state



RNN forgets about the dog after a certain time ☹

RNNs cannot learn "long term" dependency

Any questions?

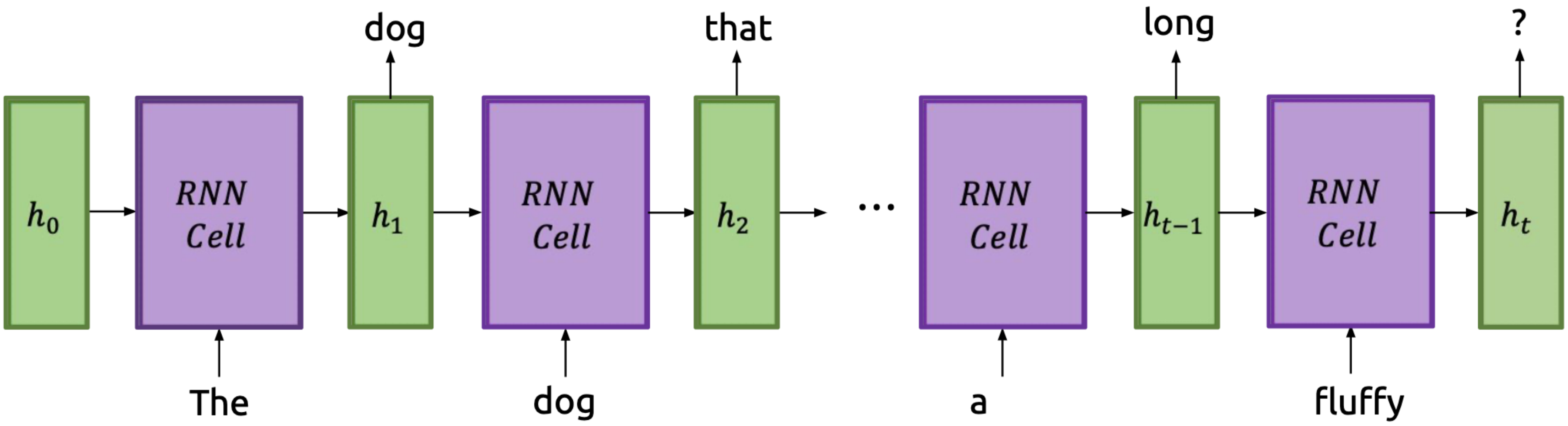


We need new way to update hidden state!

How?

An analogy to human (or computer) memory:

- RNN hidden state → "short term memory/RAM"
  - Like how you lose contents of RAM if you shut down a computer...
  - ...or how human short-term memory fades after time



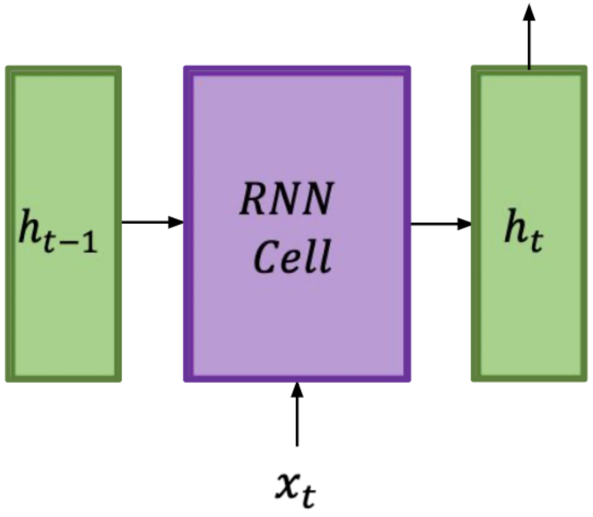
An analogy to human (or computer) memory:

- RNN hidden state → “short term memory/RAM”
  - Like how you lose contents of RAM if you shut down a computer...
  - ...or how human short-term memory fades after time
- What we want → “long term memory/disk”
  - Some state representing knowledge that persists
  - Like how contents of disk persist across shut-downs...
  - ...or how sleep consolidates human memory into long-term memory

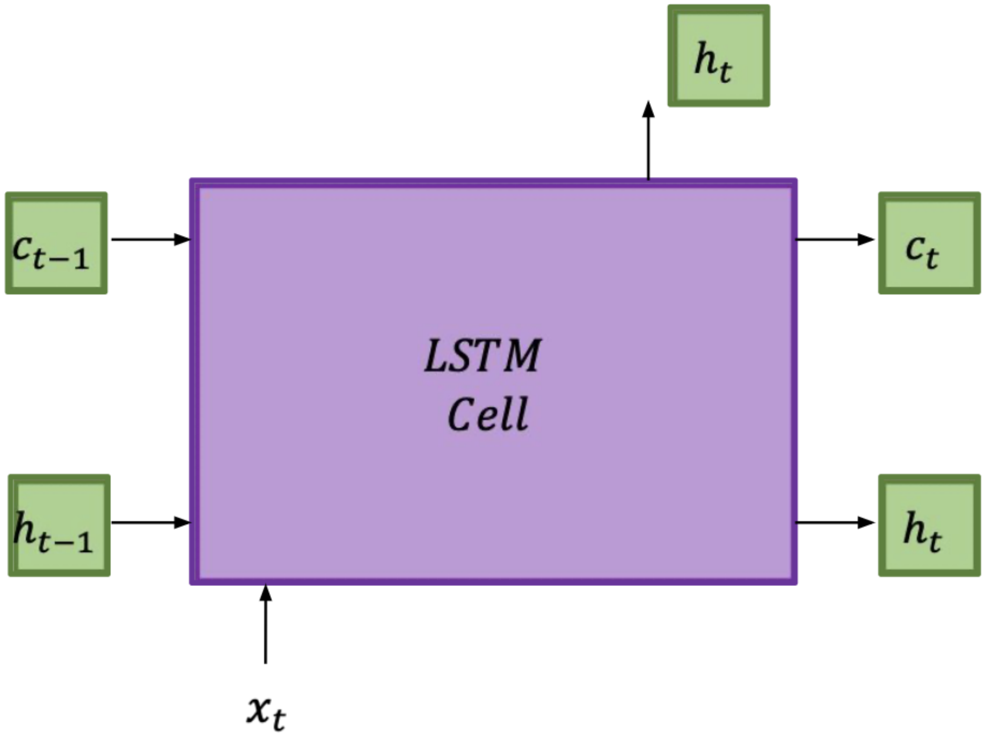
- ***Long*** Short Term Memory (LSTM)
  - “Short-term memory that persists over time”
  - i.e. “hidden states that remember information for longer”

What is different?

Vanilla RNN

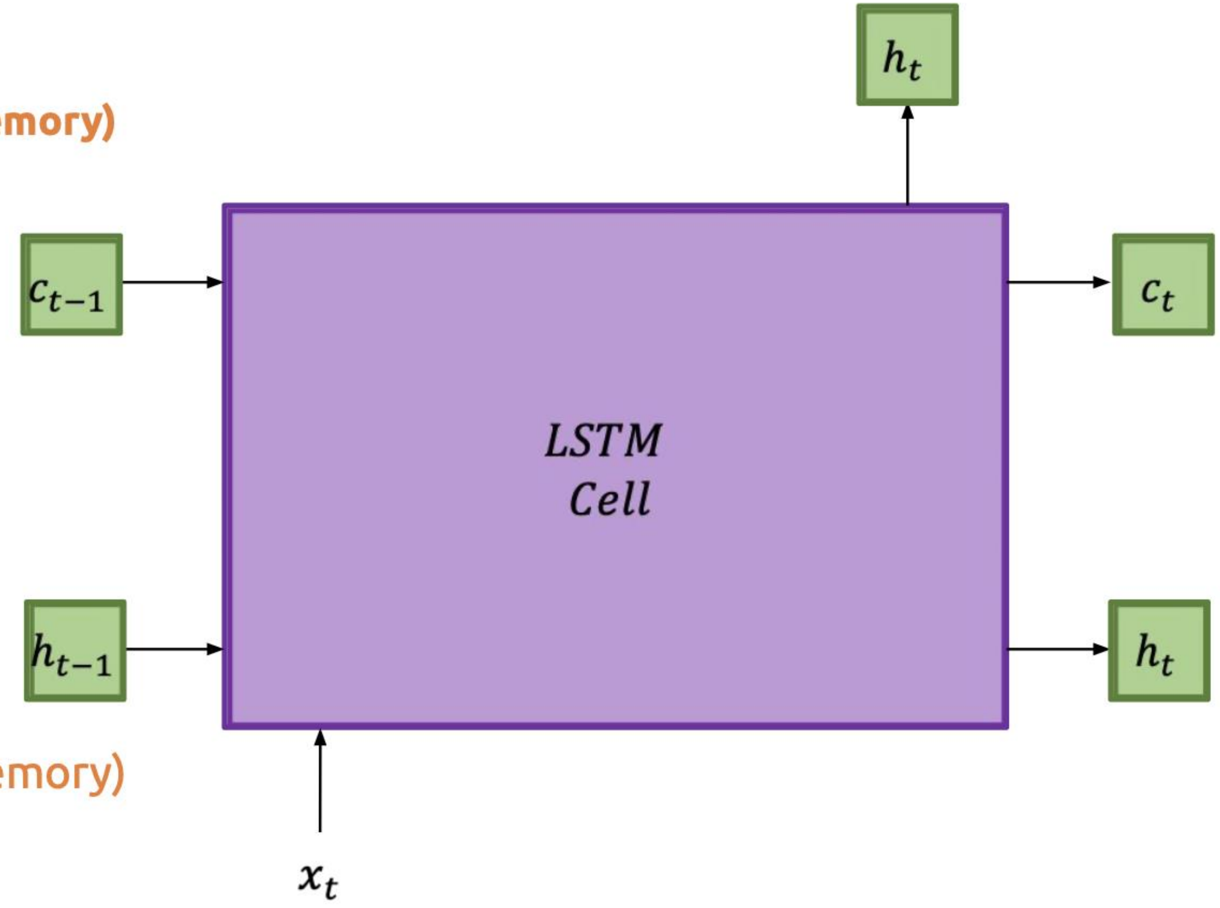


LSTM



LSTM

Cell State (long short-term memory)



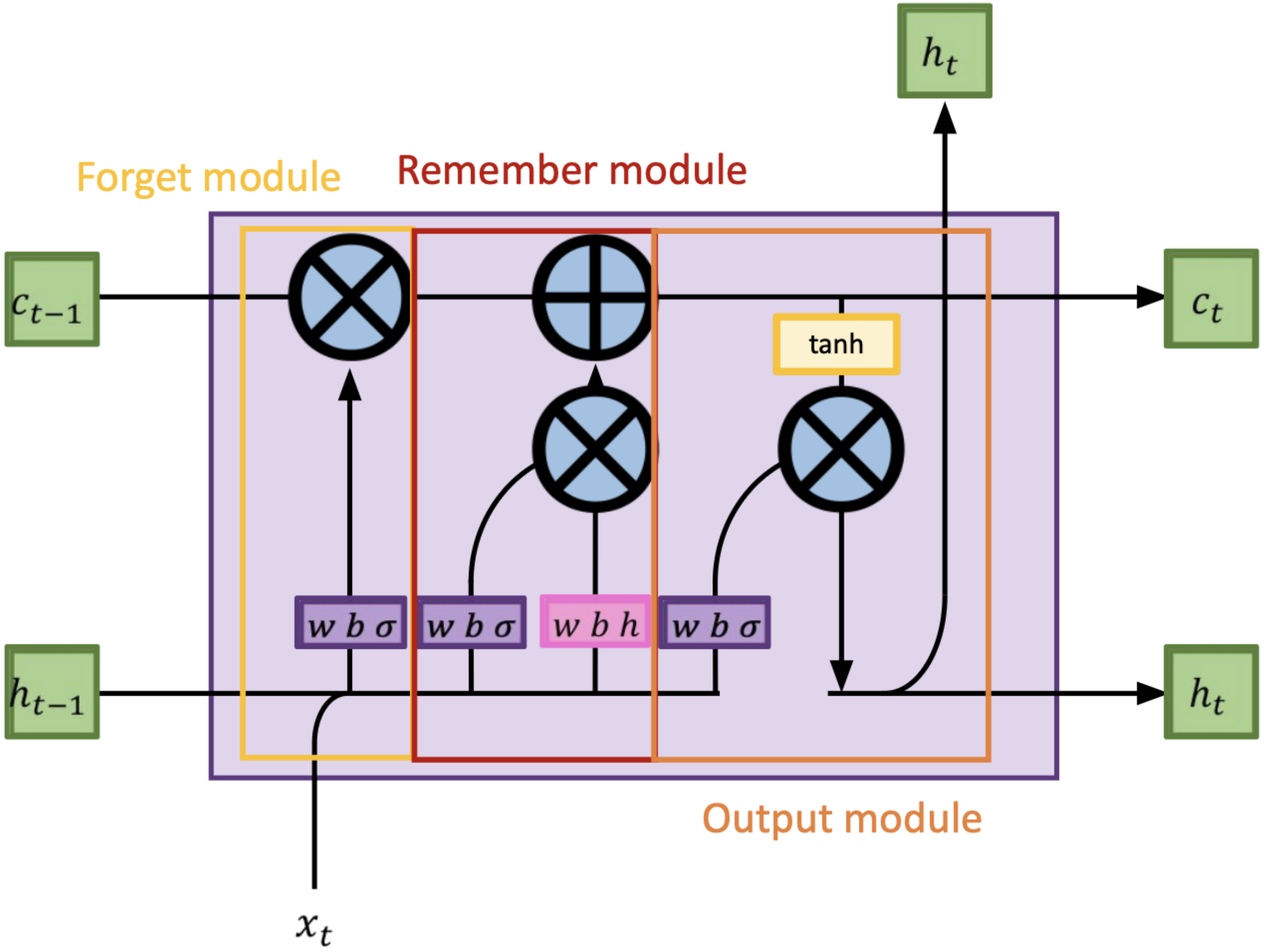
Hidden State (short –term memory)

word embedding

How an LSTM works

- An LSTM consists of 3 major modules:
  - Forget module
  - Remember module
  - Output module

The Complete LSTM



Forget Module

Say we just predicted *“tail”* in *“My dog has a fluffy _____.”*

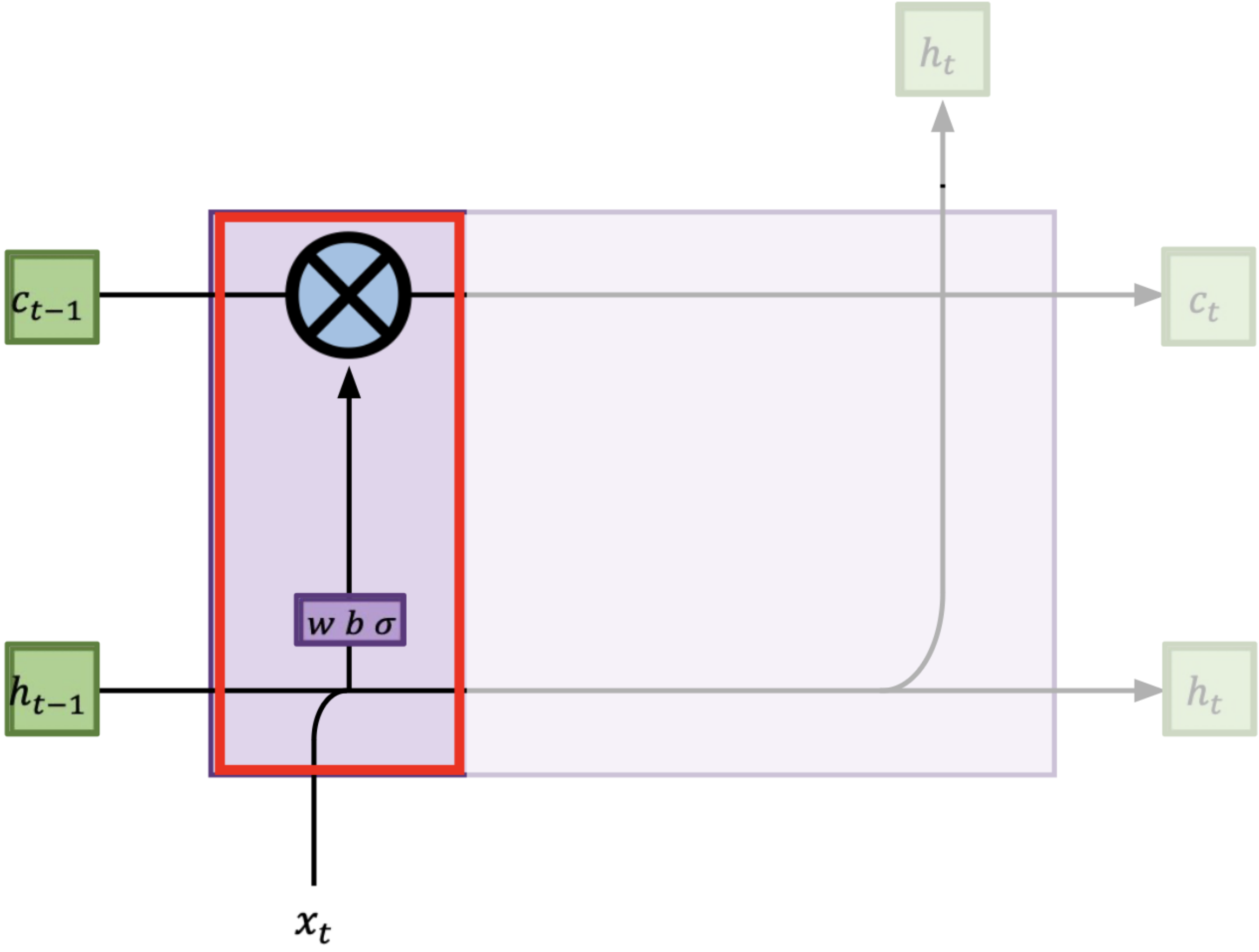
Next set of words: *“I love my dog”*

Forget Module

- Model no longer needs to know about ***"dog"***
- Ready to **delete** information about subject

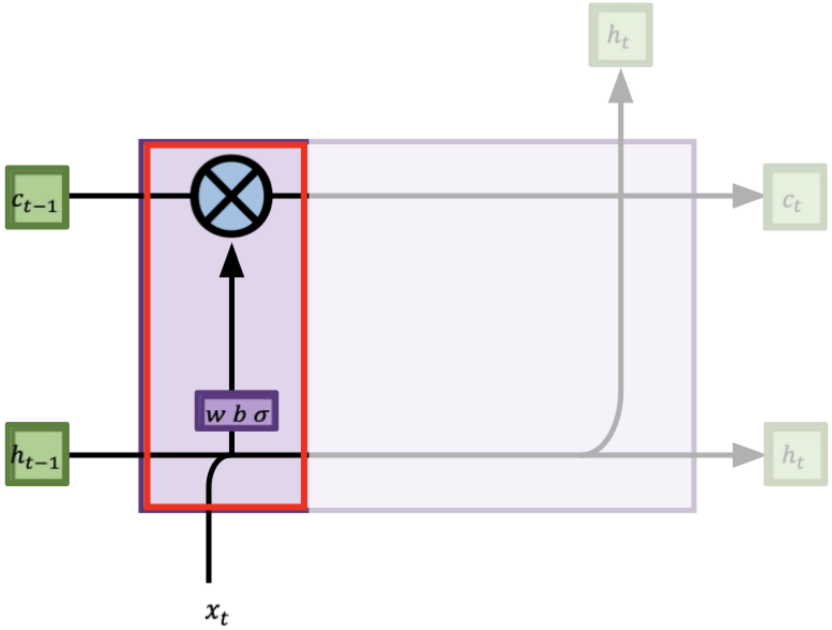
Forget Module

$w\ b\ \sigma$ = fully connected layer with sigmoid
$\otimes$ = pointwise multiplication



Forget Module

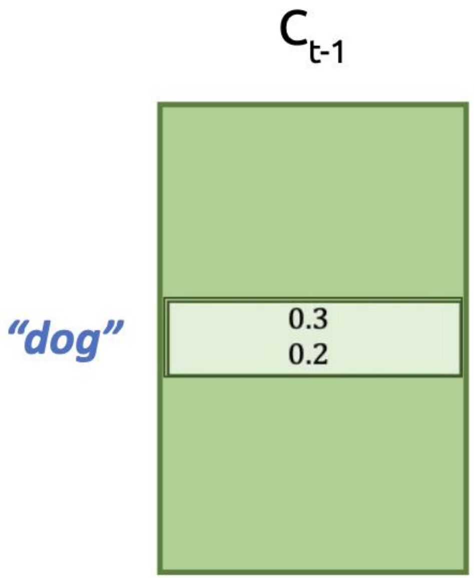
- Filters out what gets allowed into the LSTM cell from the last state
  - Example: If it’s remembering gender pronouns, and a new subject is seen, it will forget the old gender pronouns
- Either lets parts of C_{t-1} pass through or not



My dog has a fluffy tail. ***I love my dog***

Forgetting information

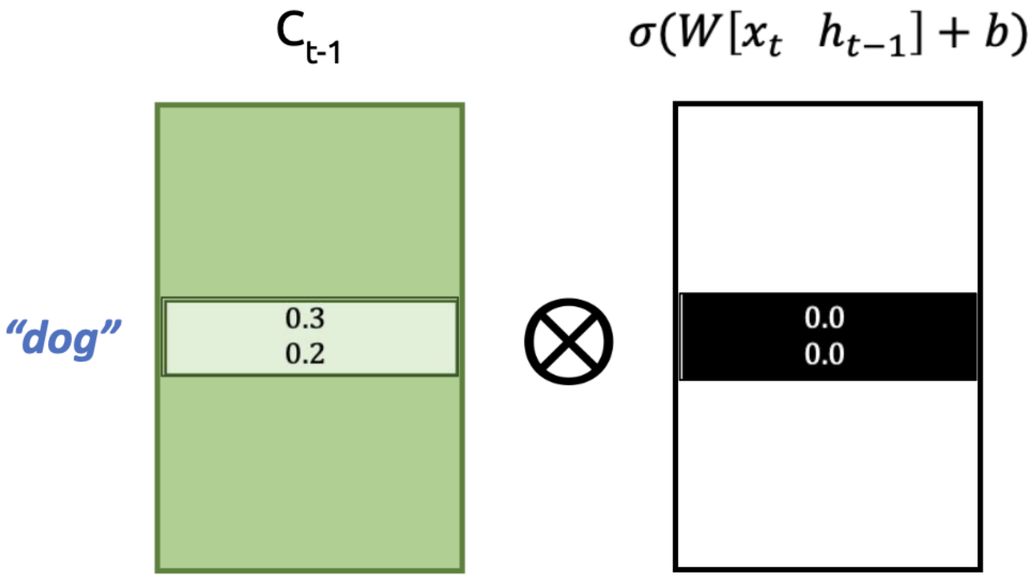
- Use pointwise multiplication by a **mask vector** to forget information
  - What do we want to forget from last cell state?



My dog has a fluffy tail. ***I love my dog***

Forgetting information

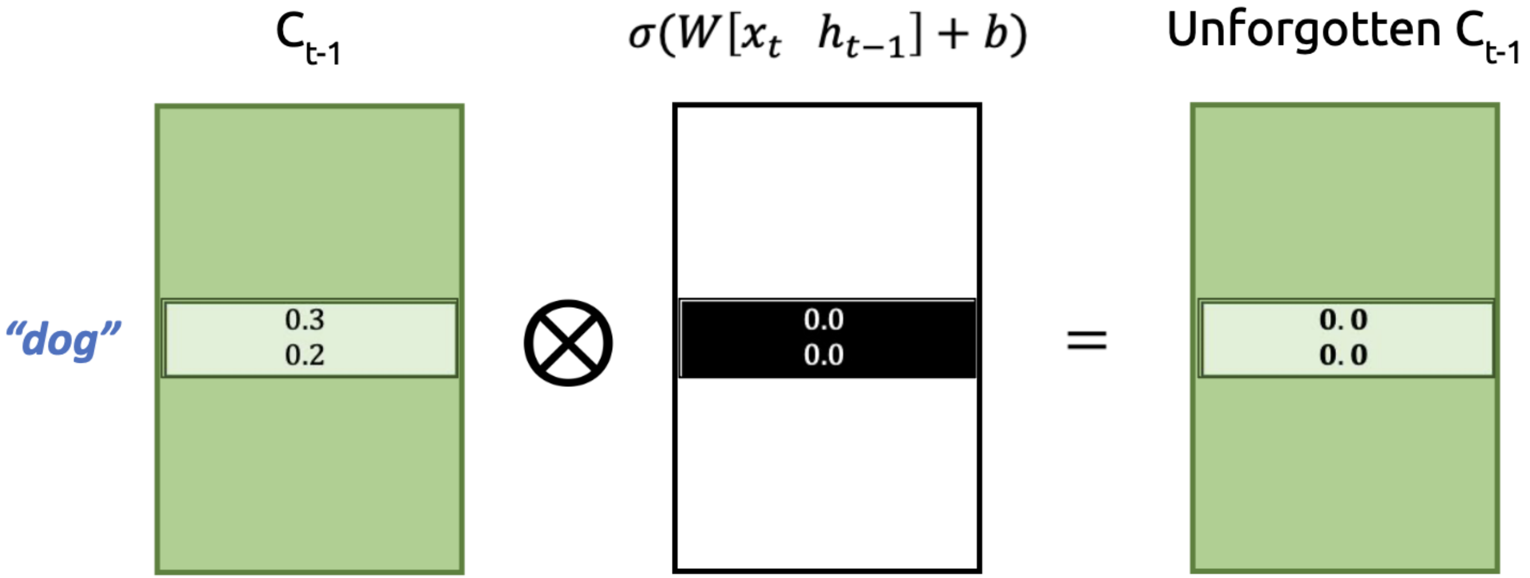
- Use pointwise multiplication by a **mask vector** to forget information
  - What do we want to forget from last cell state?
  - Output of fully connected + sigmoid is what we want to forget



My dog has a fluffy tail. ***I love my dog***

Forgetting information

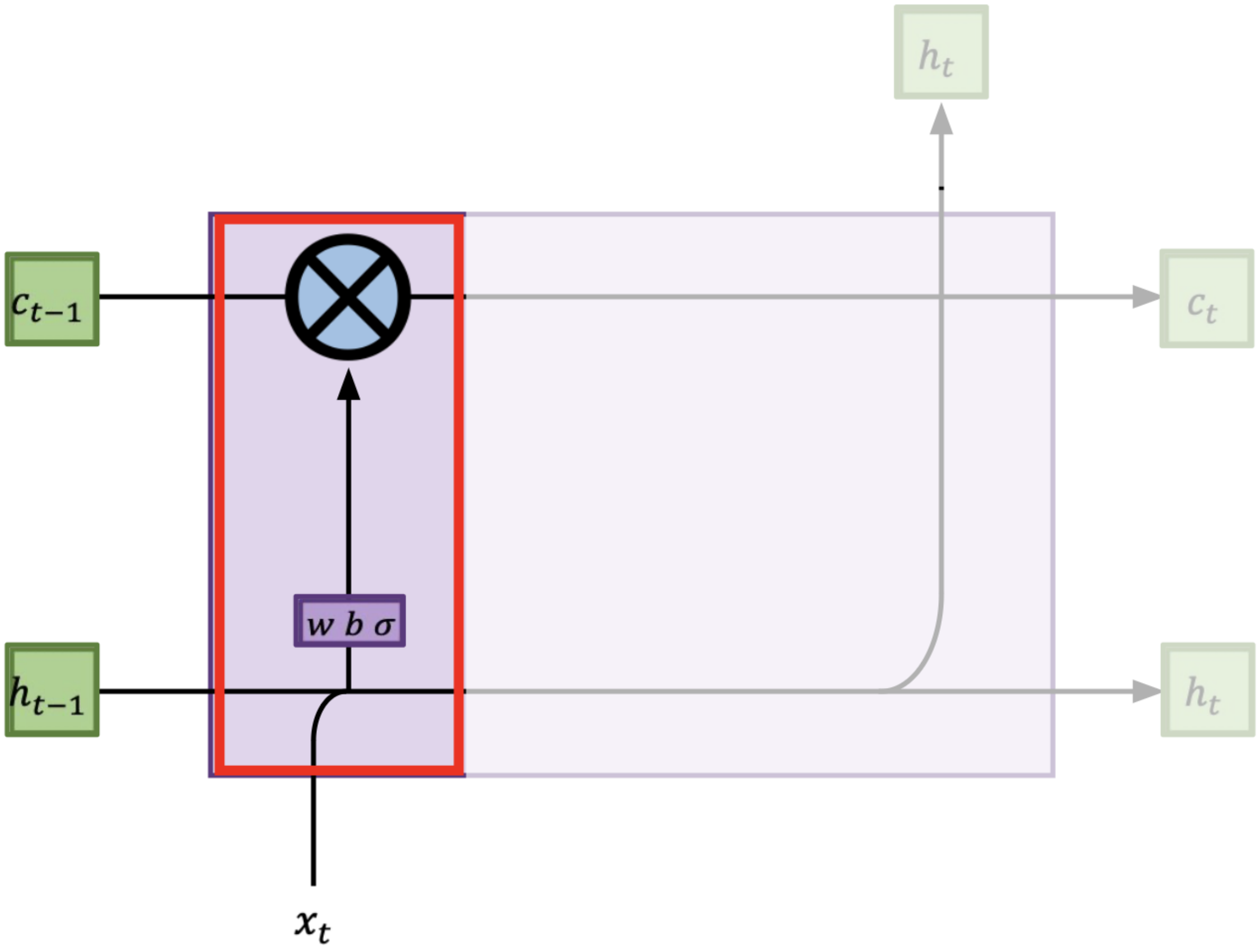
- Use pointwise multiplication by a **mask vector** to forget information
  - What do we want to forget from last cell state?
  - Output of fully connected + sigmoid is what we want to forget
  - “Zeros out” a part of the cell state
  - Pointwise multiplication by a learned mask vector is known as ***gating***



Forget Module

$w\ b\ \sigma$ = fully connected layer with sigmoid

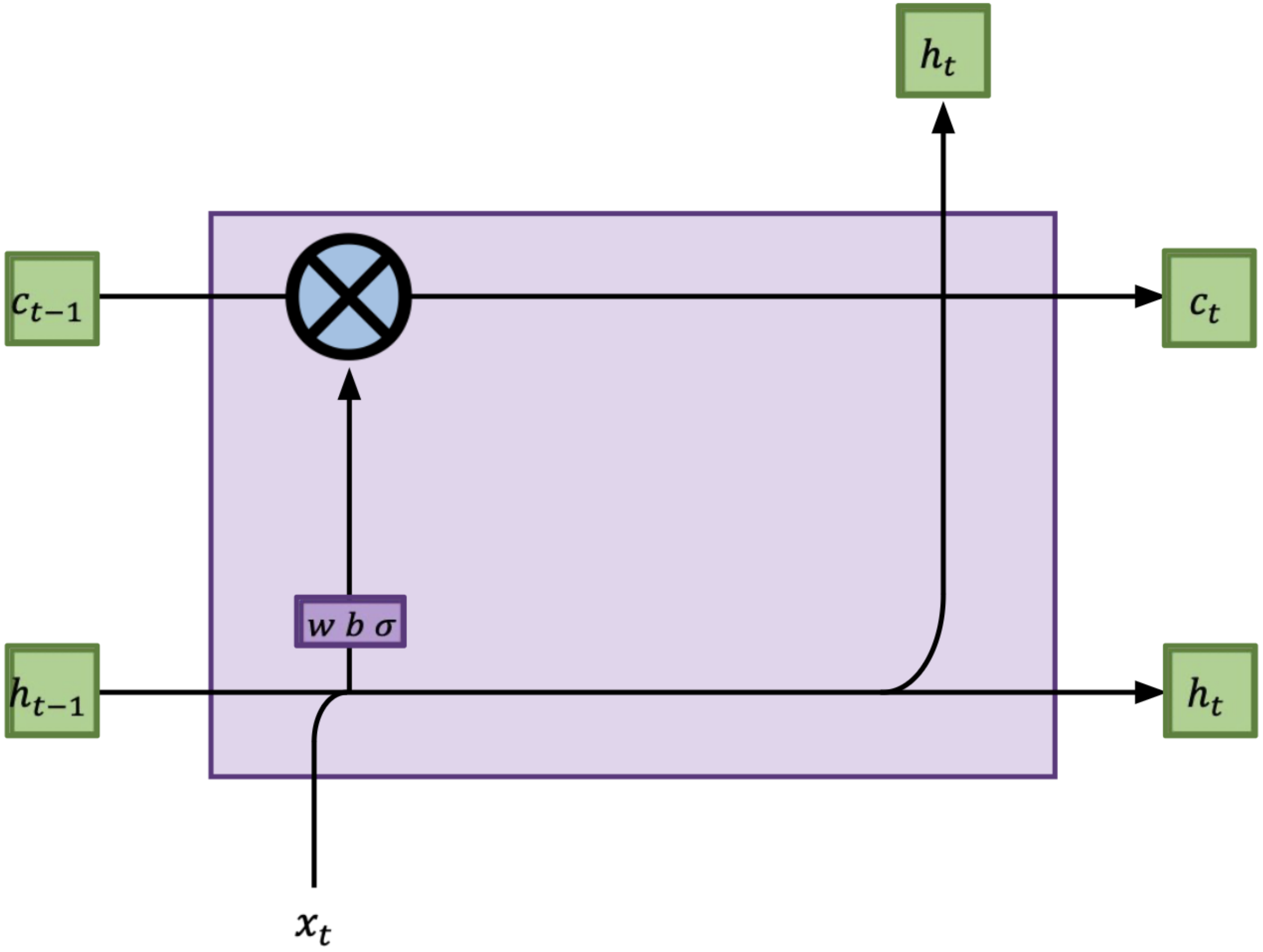
$\otimes$ = pointwise multiplication



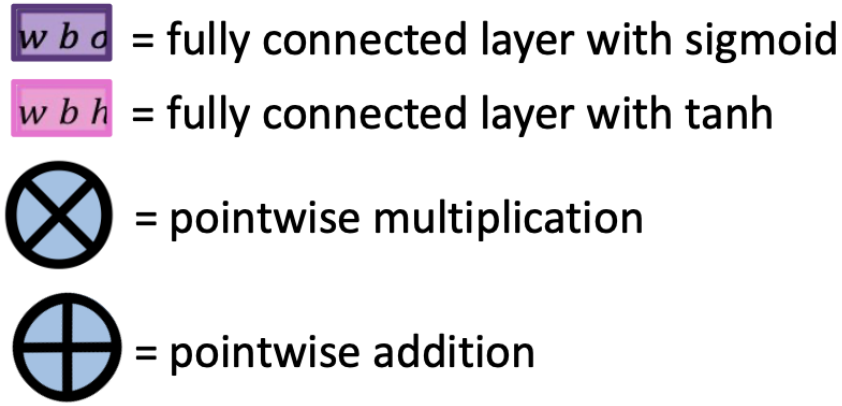
What's next?

$w\ b\ \sigma$ = fully connected layer with sigmoid

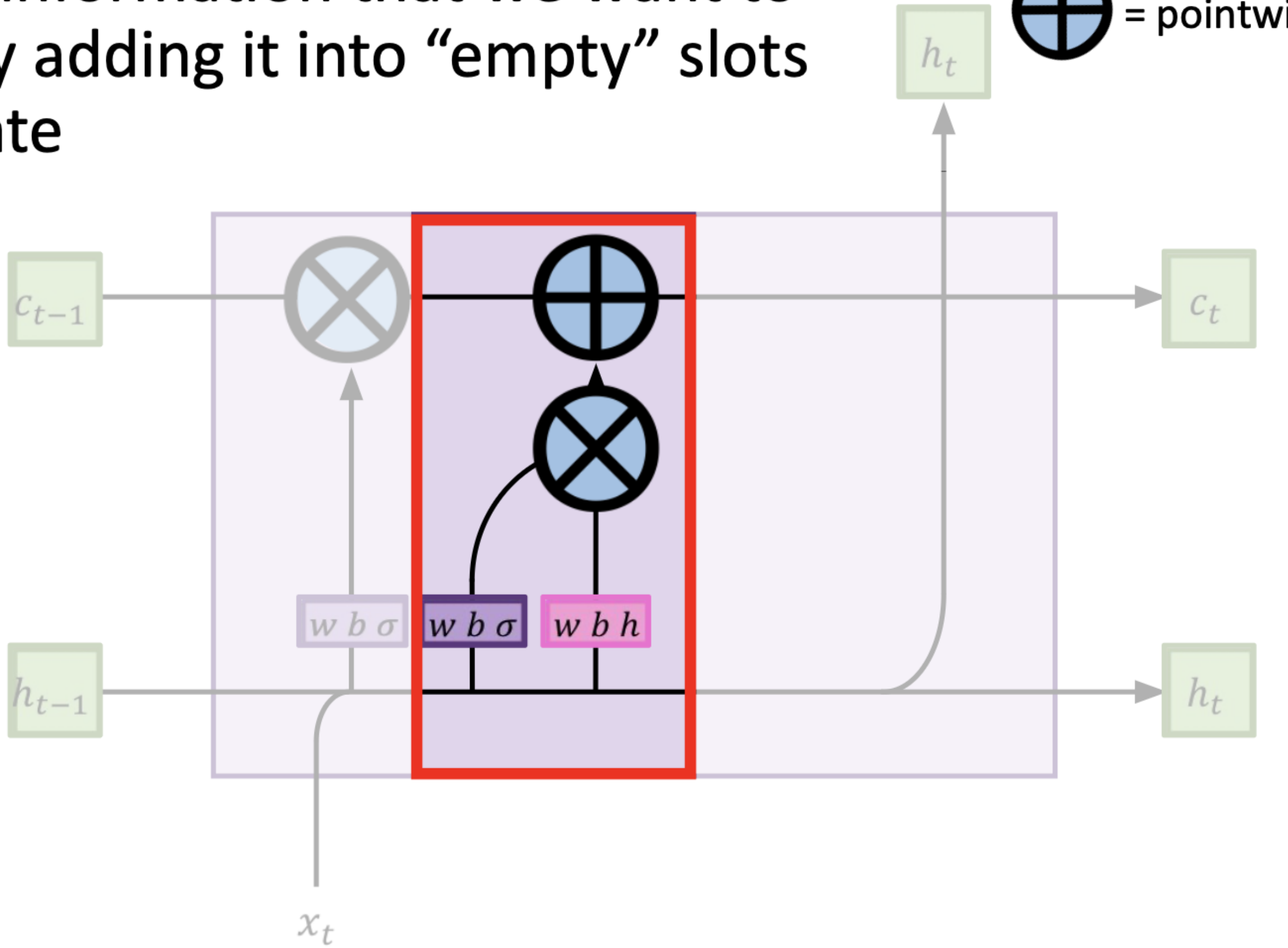
= pointwise multiplication



Remember Module



- We can save information that we want to remember by adding it into “empty” slots in the cell state



Remember Module

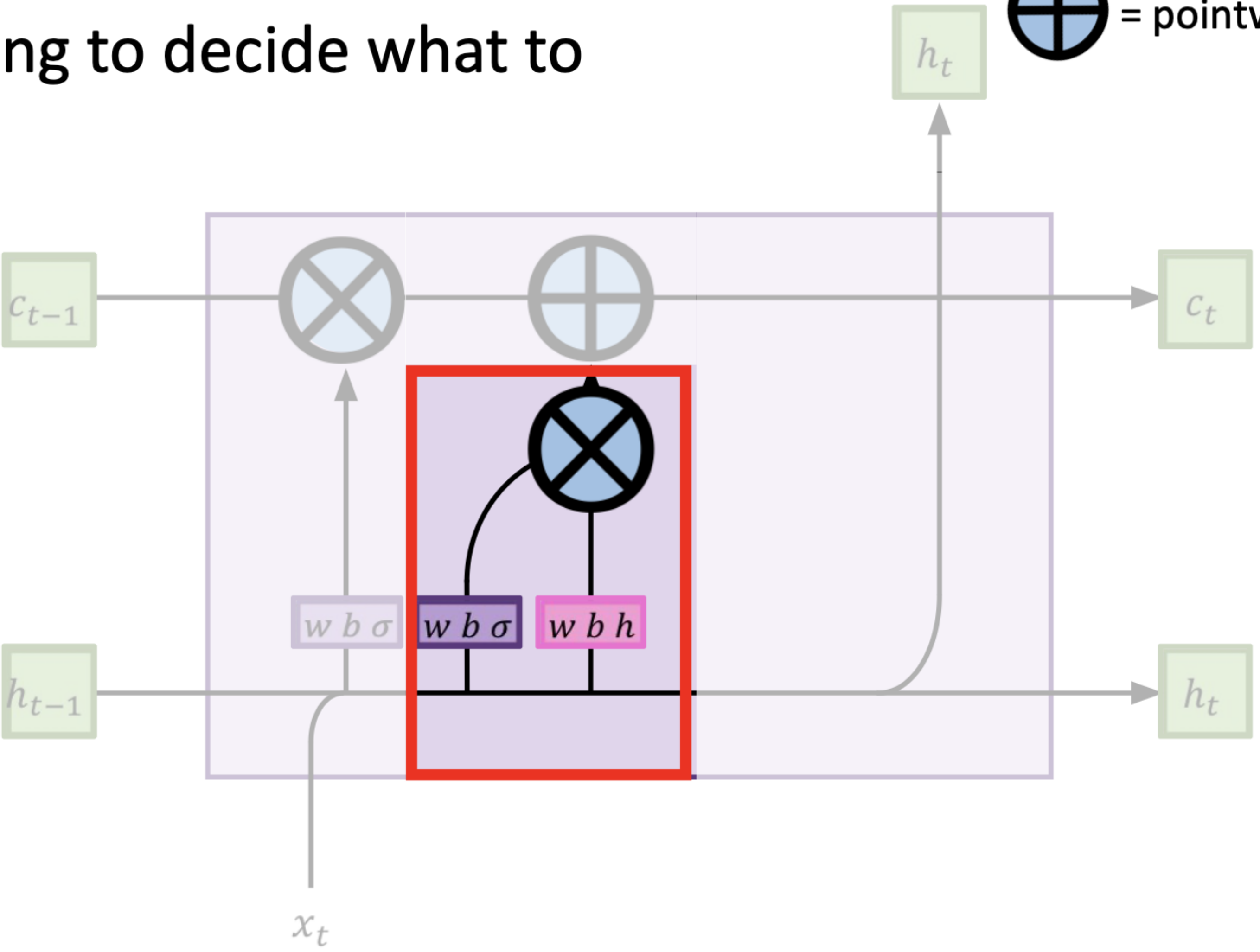
$w\ b\ \sigma$ = fully connected layer with sigmoid

$w\ b\ h$ = fully connected layer with tanh

= pointwise multiplication

= pointwise addition

- First: use gating to decide what to remember

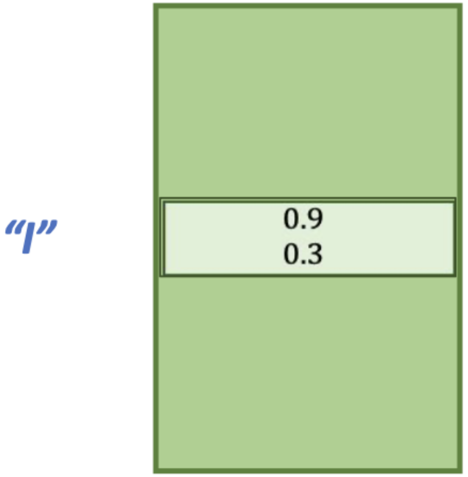


My dog has a fluffy tail. ***I love my dog***

Gating for 'selective memory'

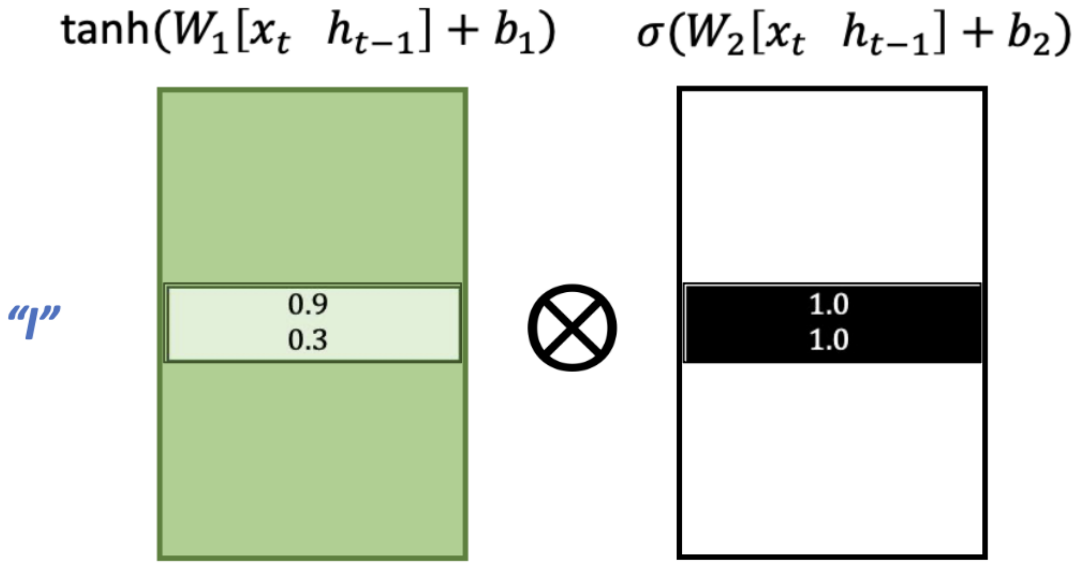
- A fully-connected + tanh on [input, memory] computes some new memory

$\text{tanh}(W_1[x_t \;\; h_{t-1}] + b_1)$



Gating for 'selective memory'

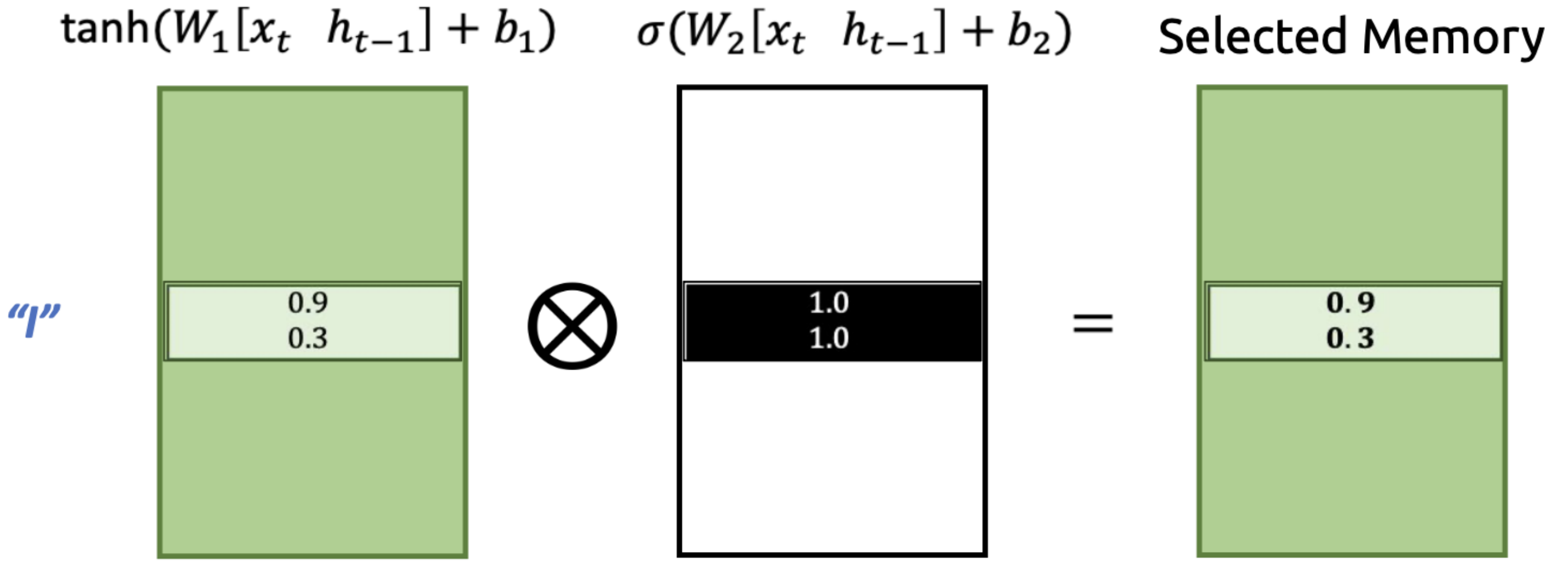
- A fully-connected + tanh on [input, memory] computes some new memory
- We gate this memory to decide what bits of it we want to remember long-term in the cell state



My dog has a fluffy tail. ***I love my dog***

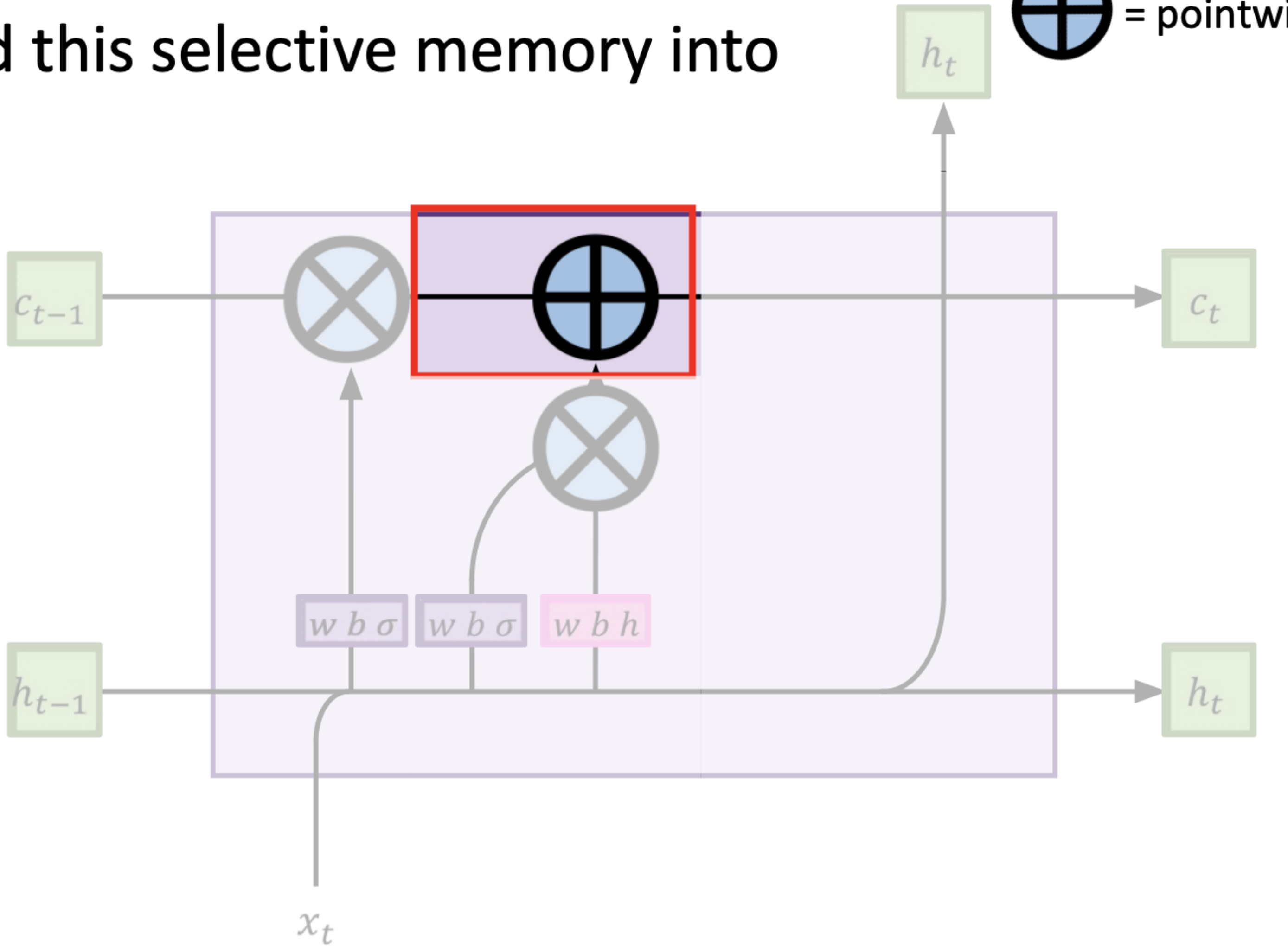
Gating for ‘selective memory’

- A fully-connected + tanh on [input, memory] computes some new memory
- We gate this memory to decide what bits of it we want to remember long-term in the cell state



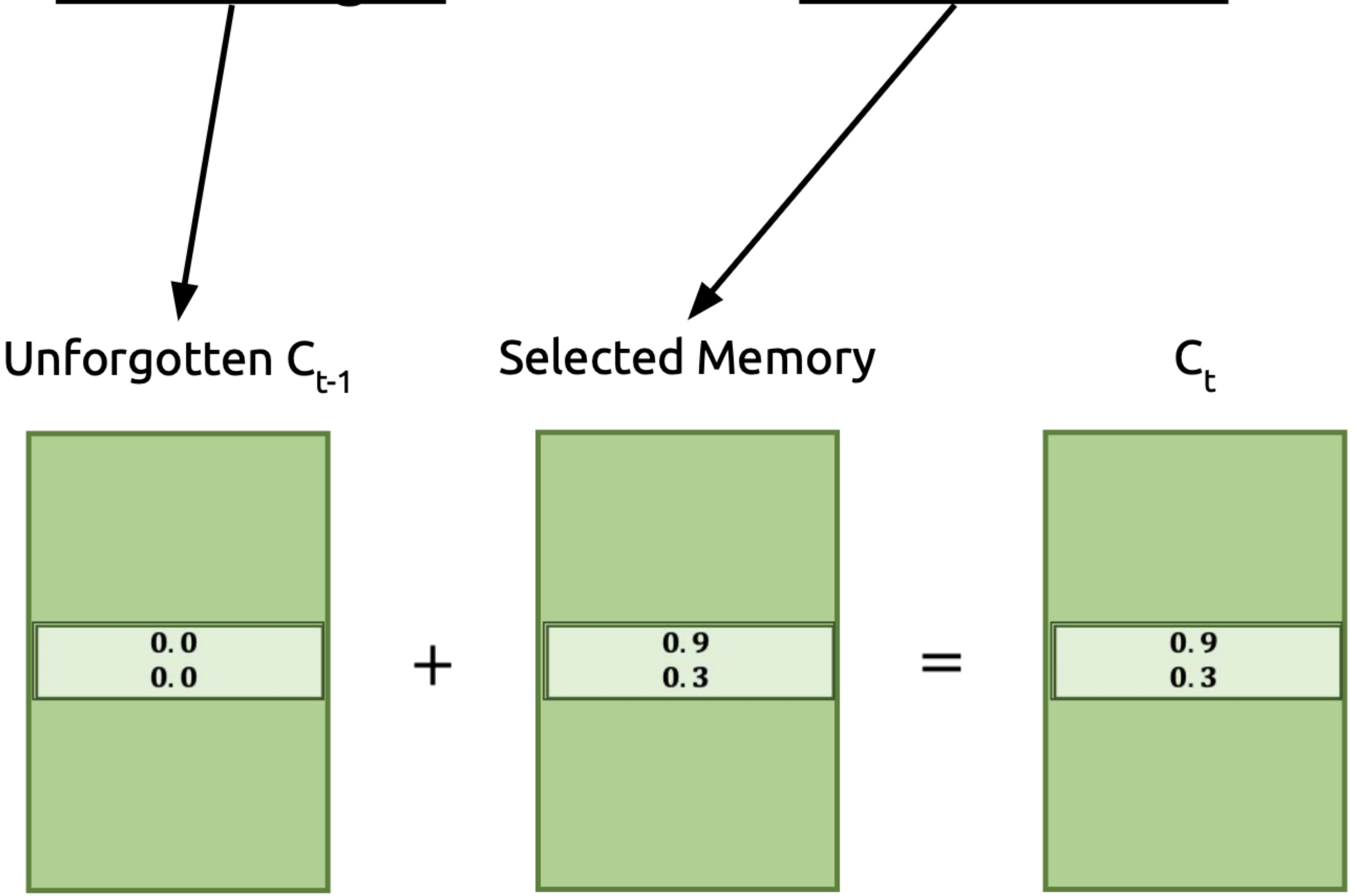
Remember Module

- Then: we add this selective memory into the cell state



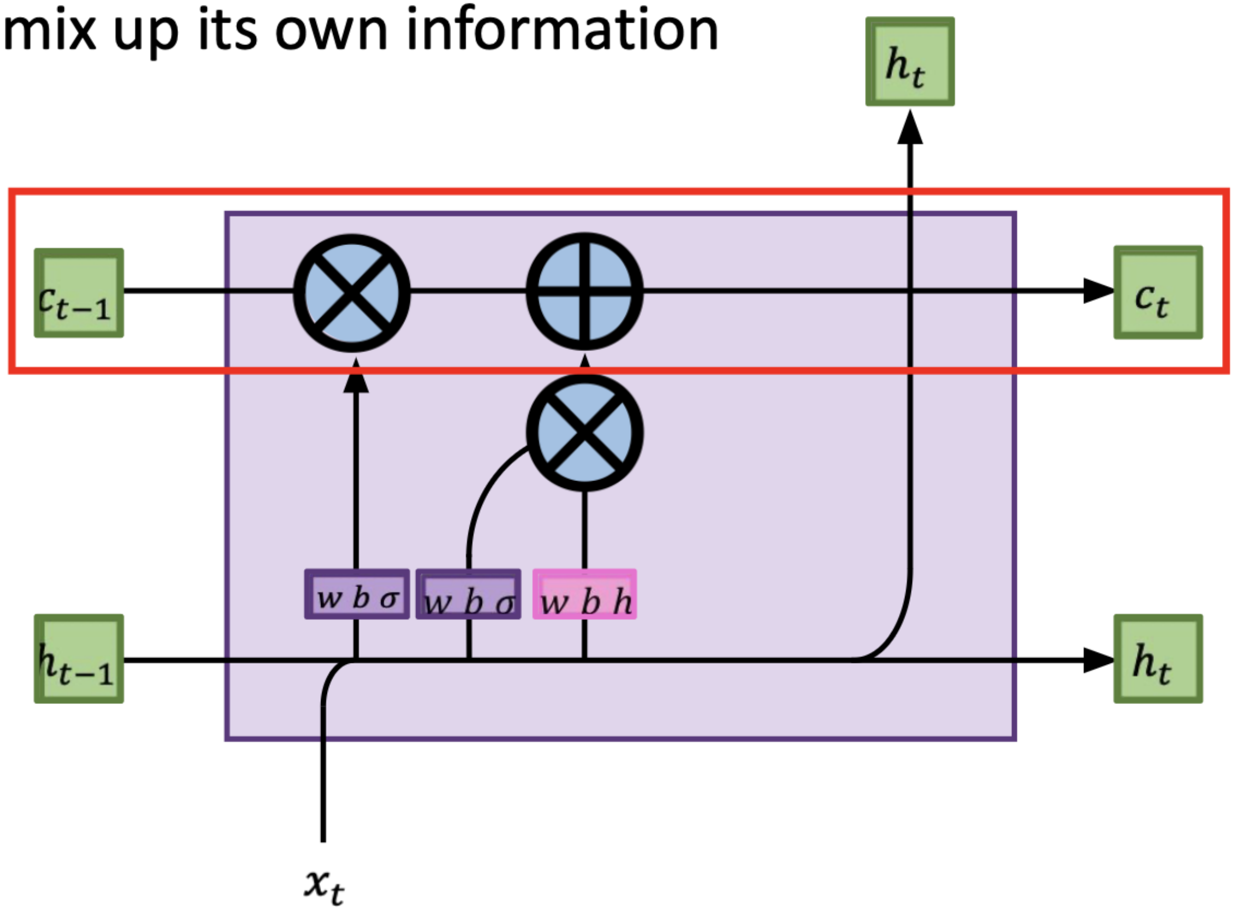
Remembering information

- Add what we <u>didn't forget</u> to what we <u>did remember</u>



Why does this solve our problem?

- Cell state never goes through a fully connected layer!
  - Never has to mix up its own information



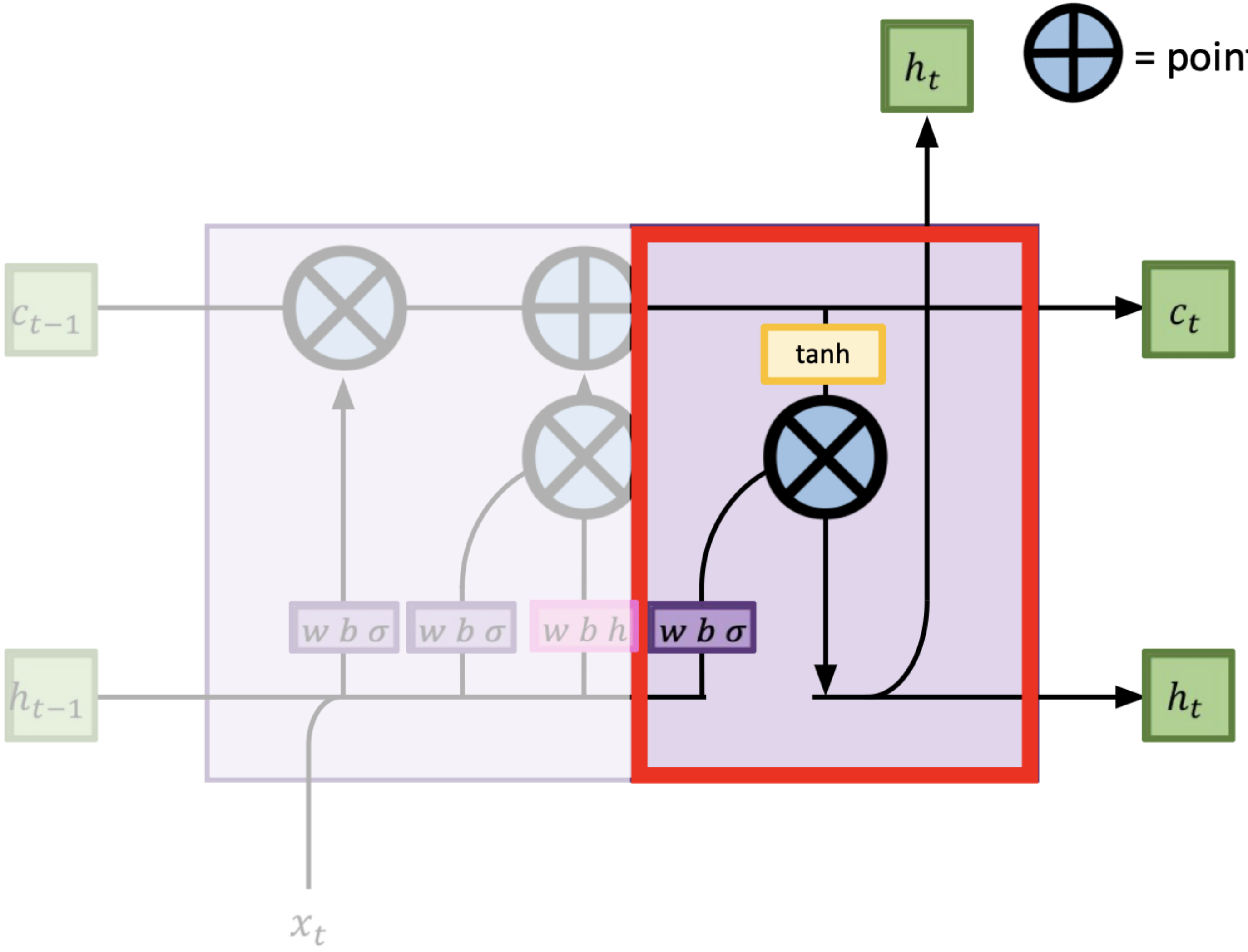
Output Module

$w\ b\ \sigma$ = fully connected layer with sigmoid

$w\ b\ h$ = fully connected layer with tanh

⊗ = pointwise multiplication

⊕ = pointwise addition



Output Module

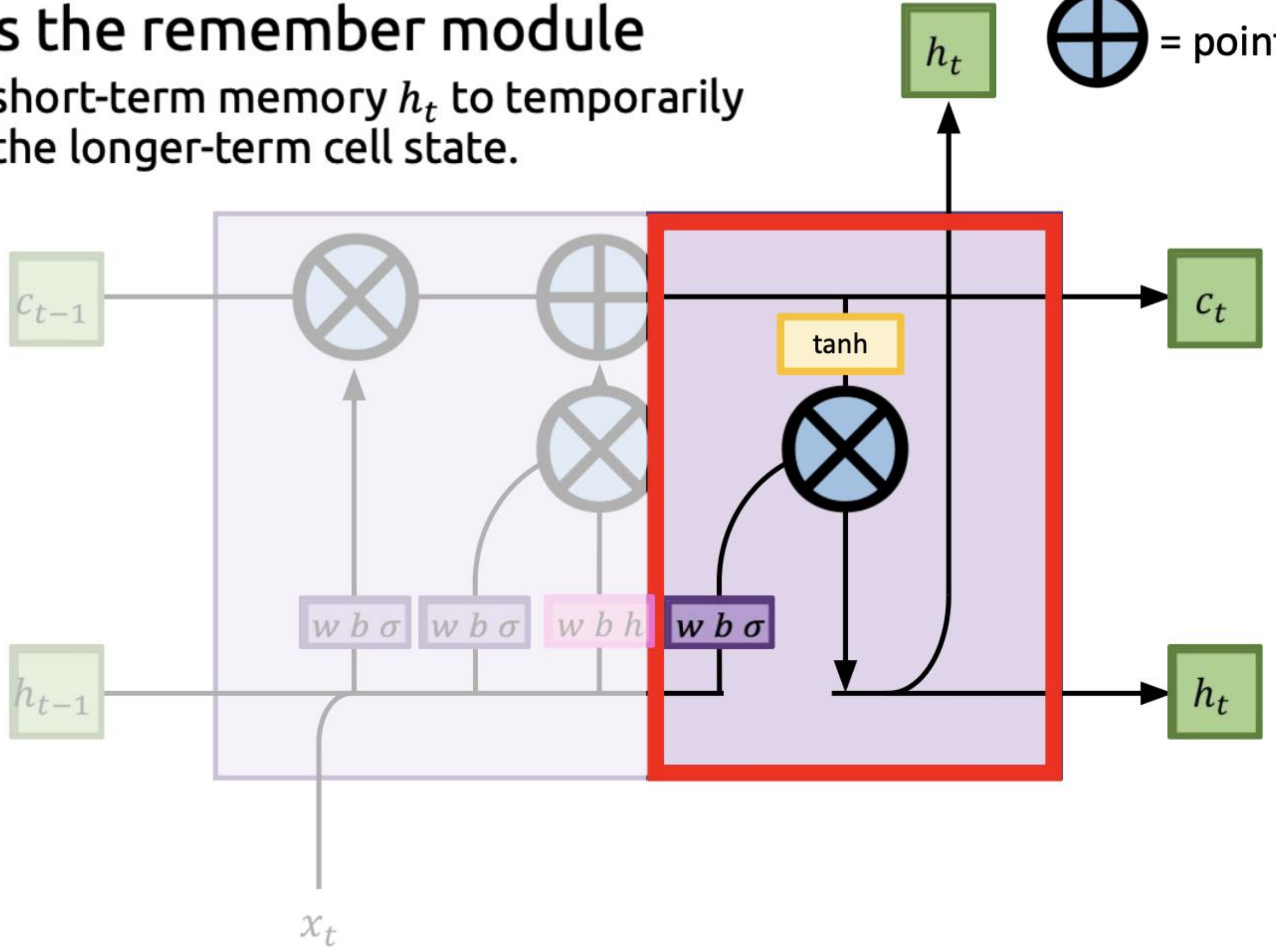
$w\ b\ \sigma$ = fully connected layer with sigmoid

$w\ b\ h$ = fully connected layer with tanh

$\otimes$ = pointwise multiplication

$\oplus$ = pointwise addition

- Same structure as the remember module
  - Provides path for short-term memory h_t to temporarily acquire info from the longer-term cell state.



The Complete LSTM

$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$$
$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f)$$
$$o_t = \sigma(W_o h_{t-1} + U_o x_t + b_o)$$
$$\widetilde{c}_t = tanh(W h_{t-1} + U x_t + b)$$
$$c_t = f_t \circ c_{t-1} + i_t \circ \widetilde{c}_t$$
$$h_t = o_t \circ tanh(c_t)$$
$$y_t = h_t$$