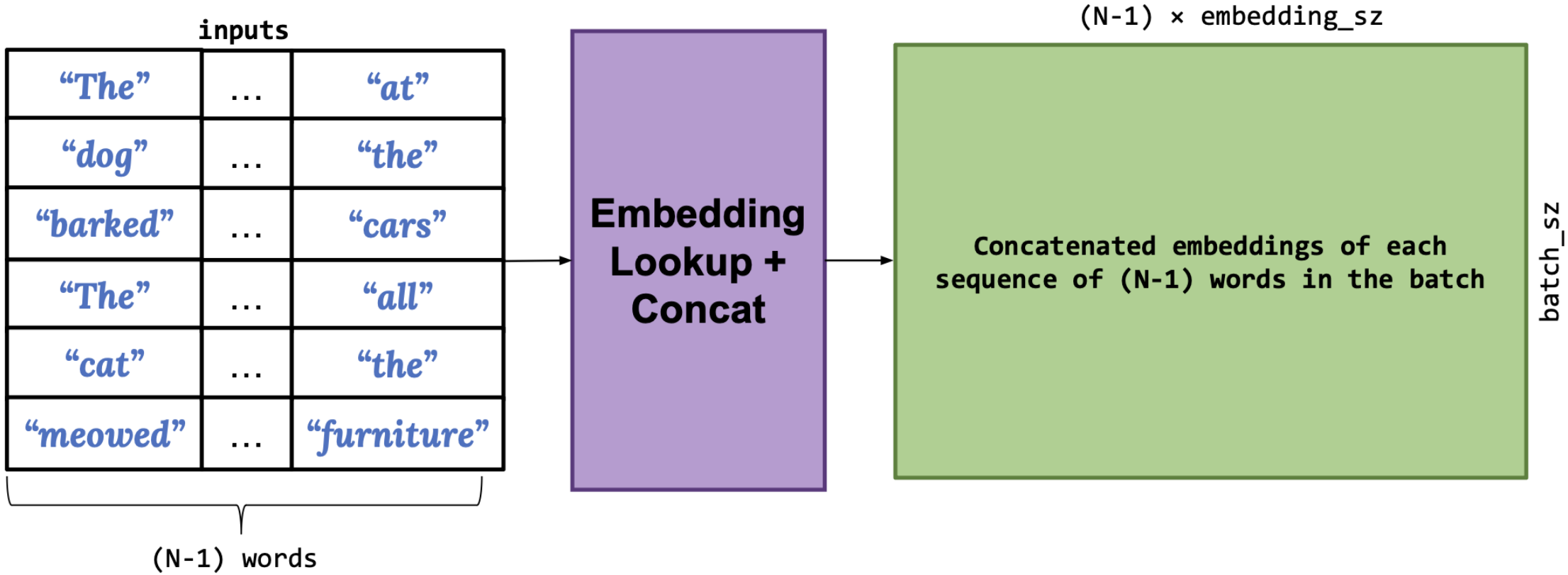

Deep Learning
Day 11: RNNs
CSCI 1470
Eric Ewing
Thursday, 10/9/25

Size of Feed Forward N-gram Model

Embedding lookup + Concatenation still requires only one embedding matrix of size: `(vocab_sz, embedding_sz)`



Lack of Flexibility with N-grams

We would like for our language model to be more aware of context when deciding on how many words in the past to consider as “relevant”.

But when we look at other portions, common phrases and sequences of words may make it impossible to have any idea what should come next.

“The dog barked at one of the cats.”

We want our model to recognize these patterns and dynamically adapt how it makes a prediction based on context.

Any questions?

Limitations of the N-gram model

What problems do we run into using Feed Forward N-gram models?

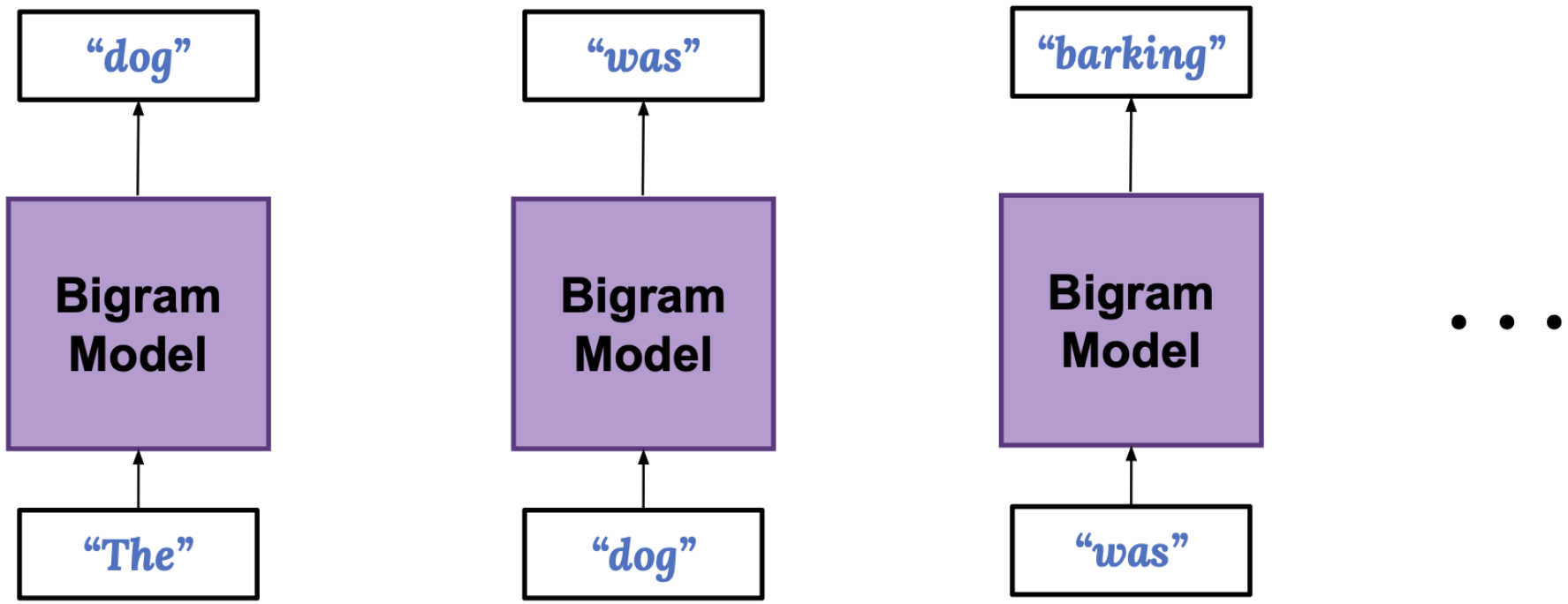
1. As the size of **N** increases, the number of weights needed for the linear layer becomes far too large.

2. Using a fixed **N** creates problems with the flexibility of our model.

We need a solution that is both computationally cheap and more dynamic in terms of its memory of previously seen words.

New Approach

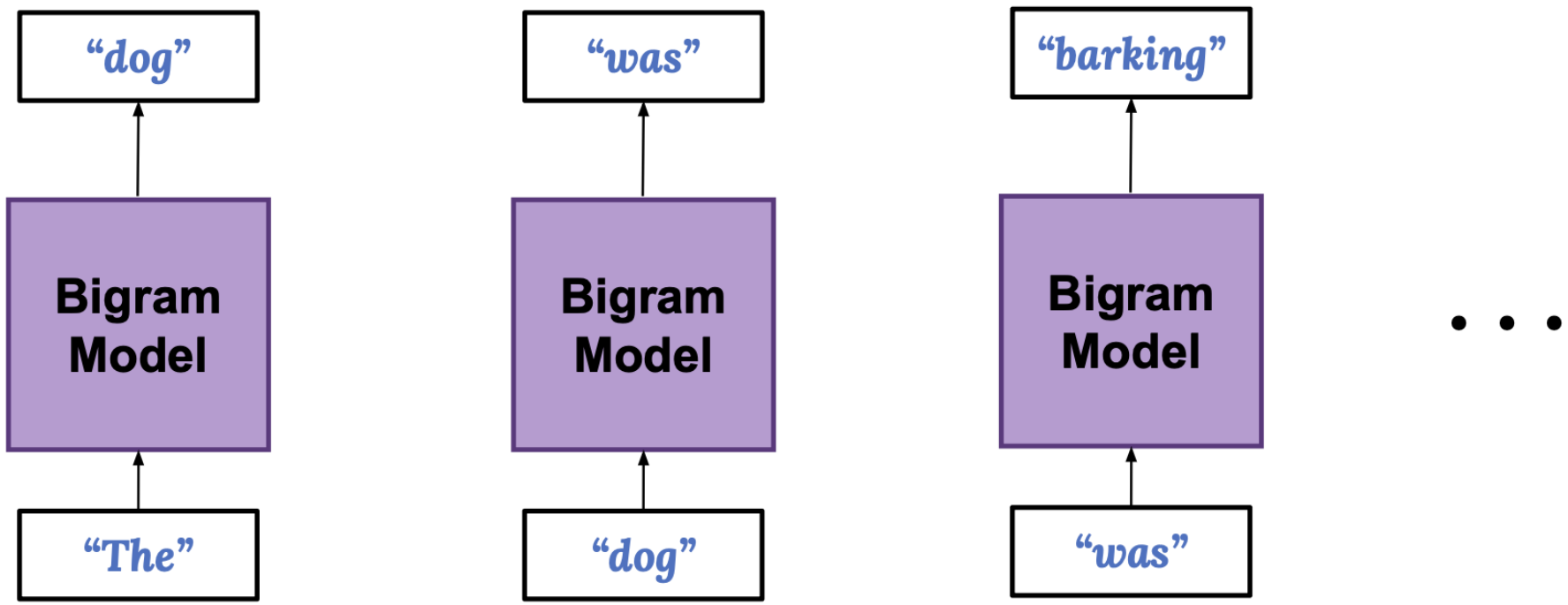
Let's revisit the bigram model and see several iterations of prediction using a bigram model:



New Approach

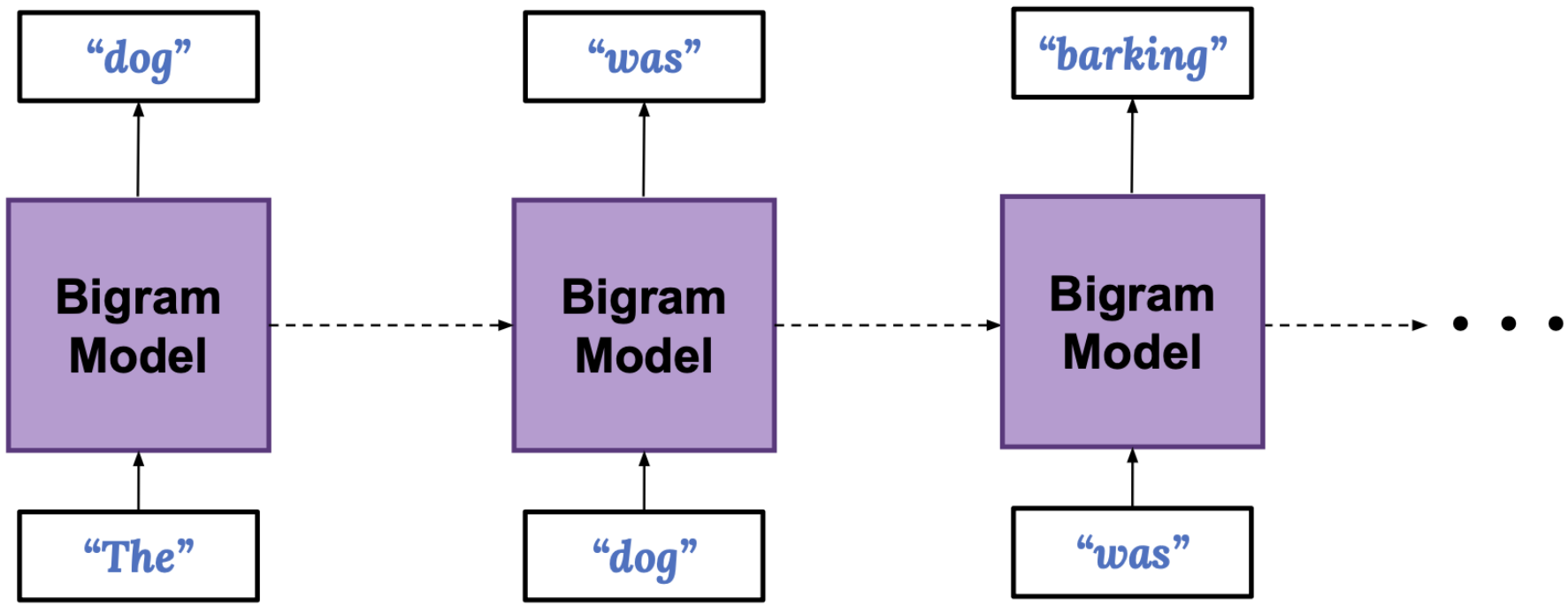
Any ideas?

Ideally, we would like to be able to keep “memory” of what words occurred in the past.



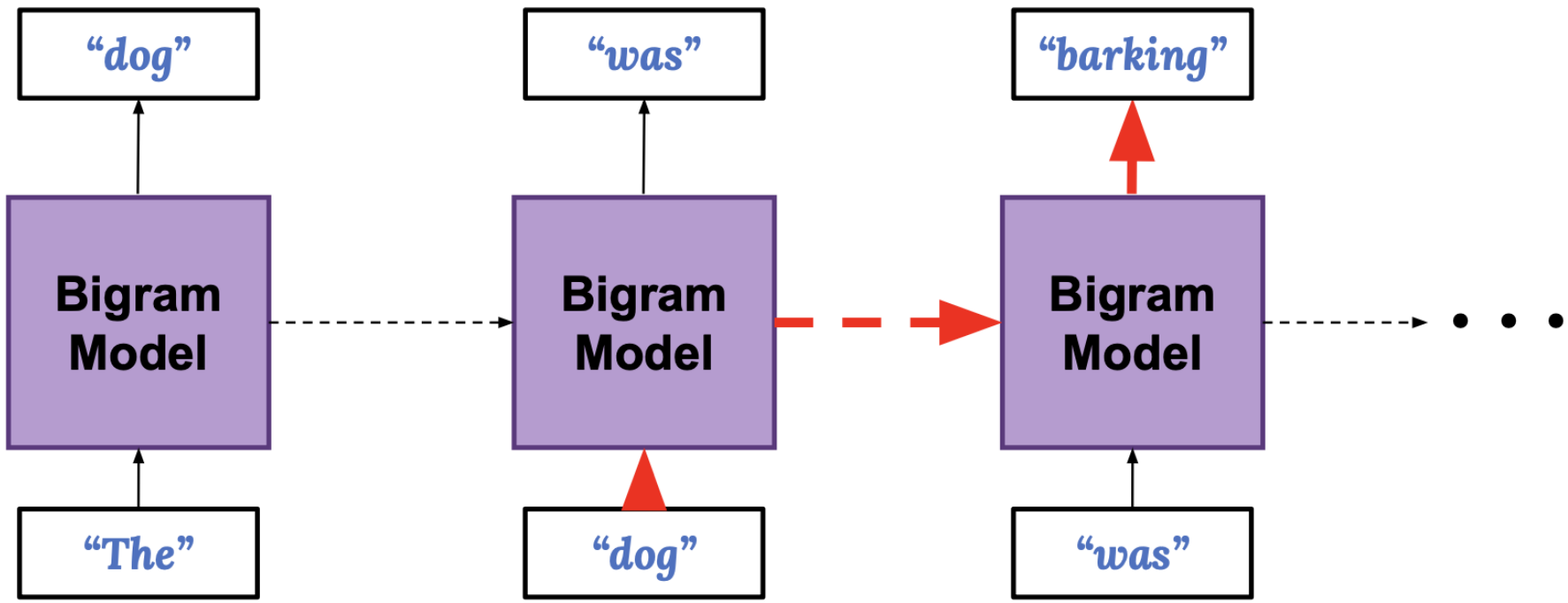
New Approach

What if we sequentially passed information from our previous bigram block into our next block?



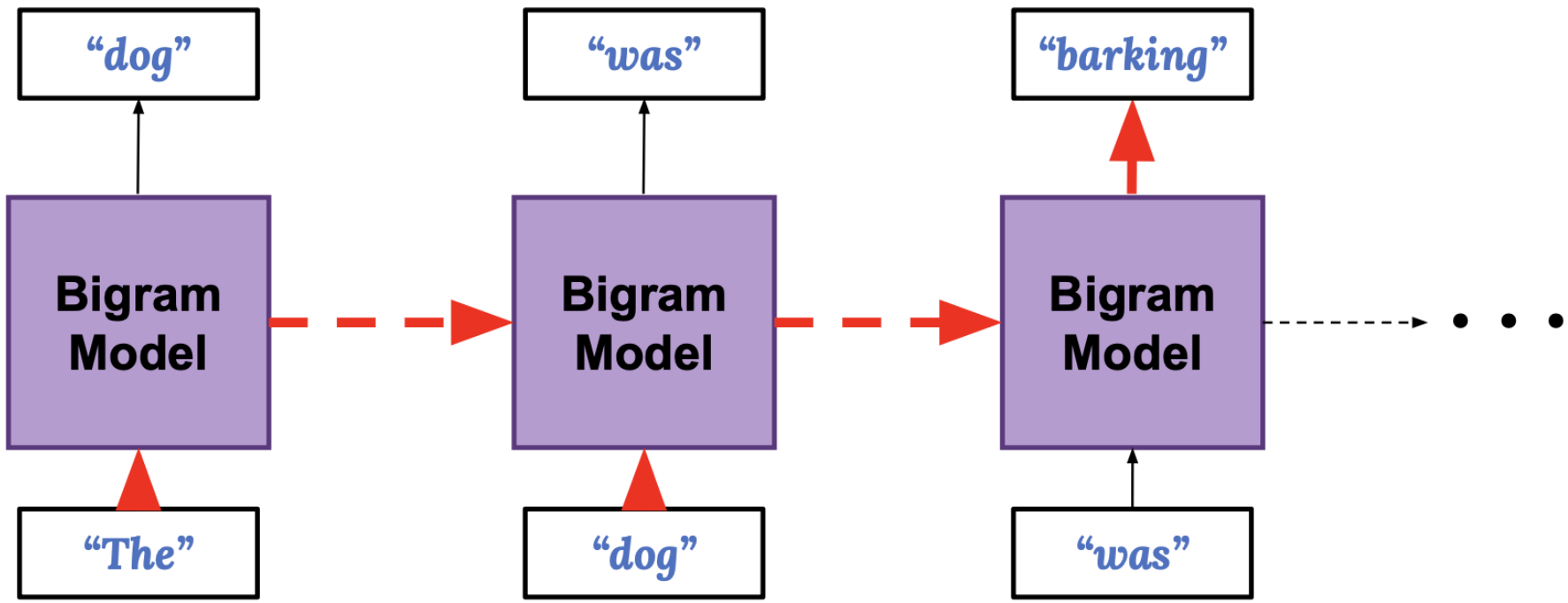
New Approach

If we follow the information flow, we see that when predicting “barking”, we have some way of knowing that “dog” was previously observed:



New Approach

In fact, we even have a way of knowing that “The” was observed!

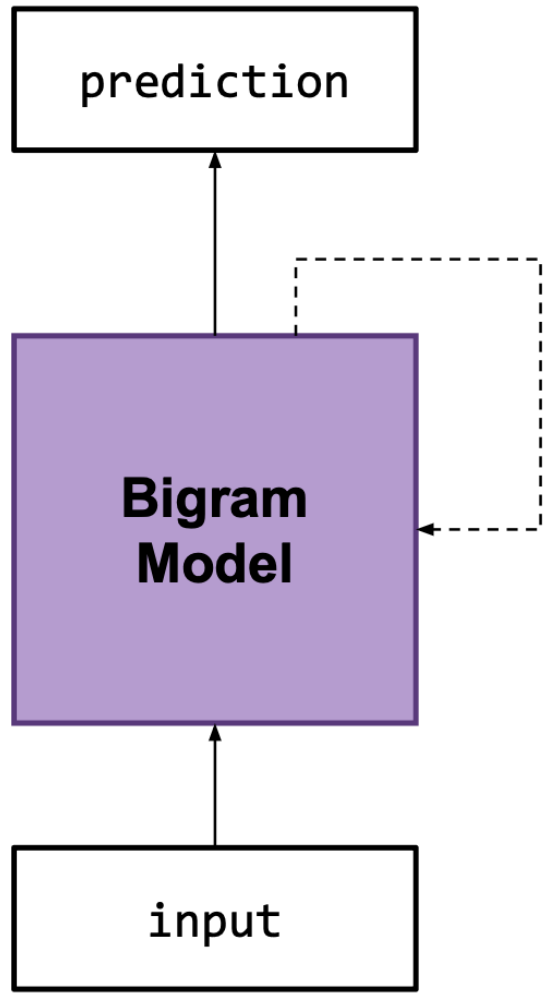


New Approach

We can represent this relationship using only one bigram block and connection that feeds from the output of the model back into the input.

We call this connection a *recurrent* connection.

We call the previous representation the “unrolled” representation.



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.

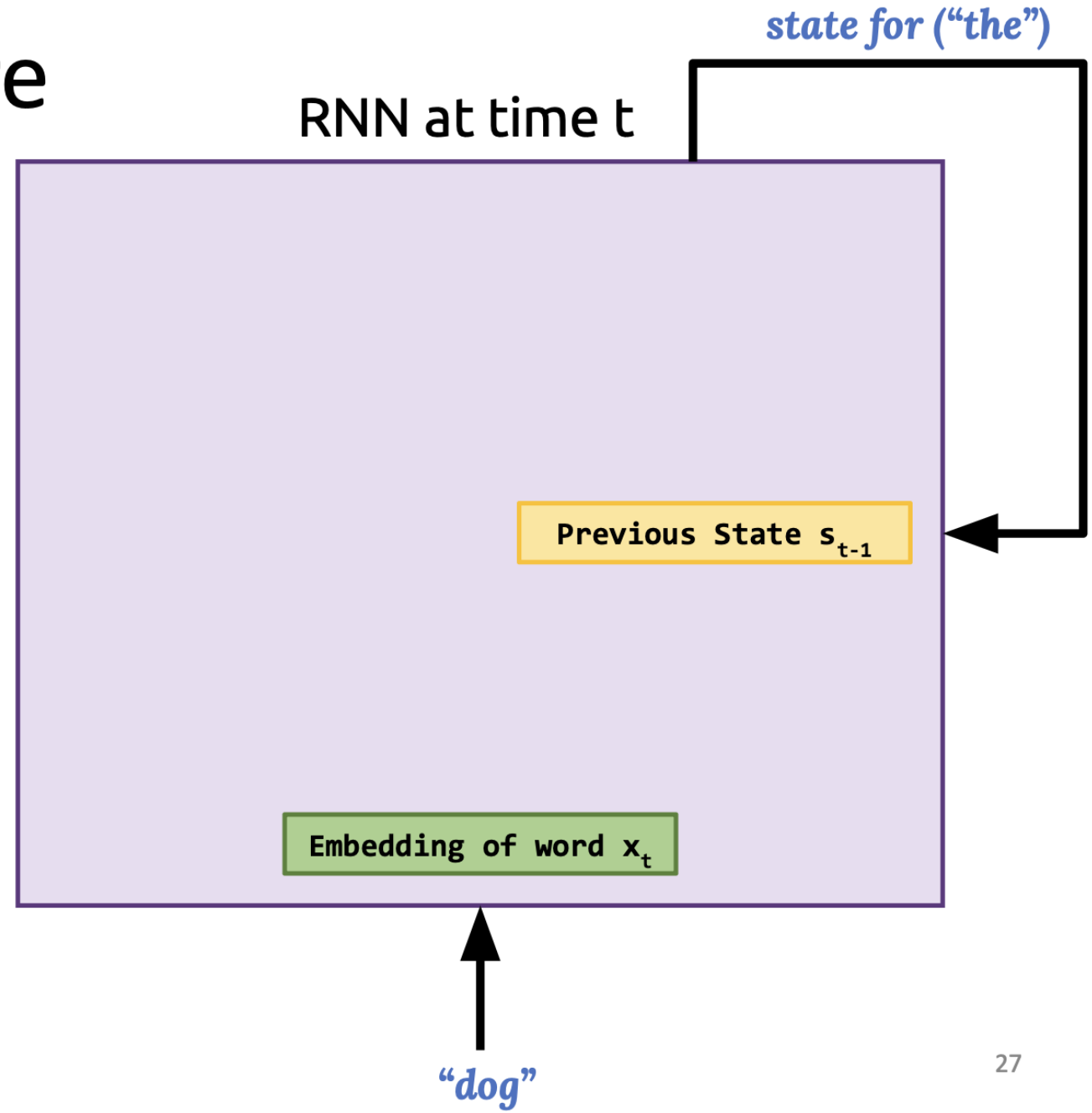
They can be used to process sequence data with relatively low model complexity when compared to feed forward models.

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

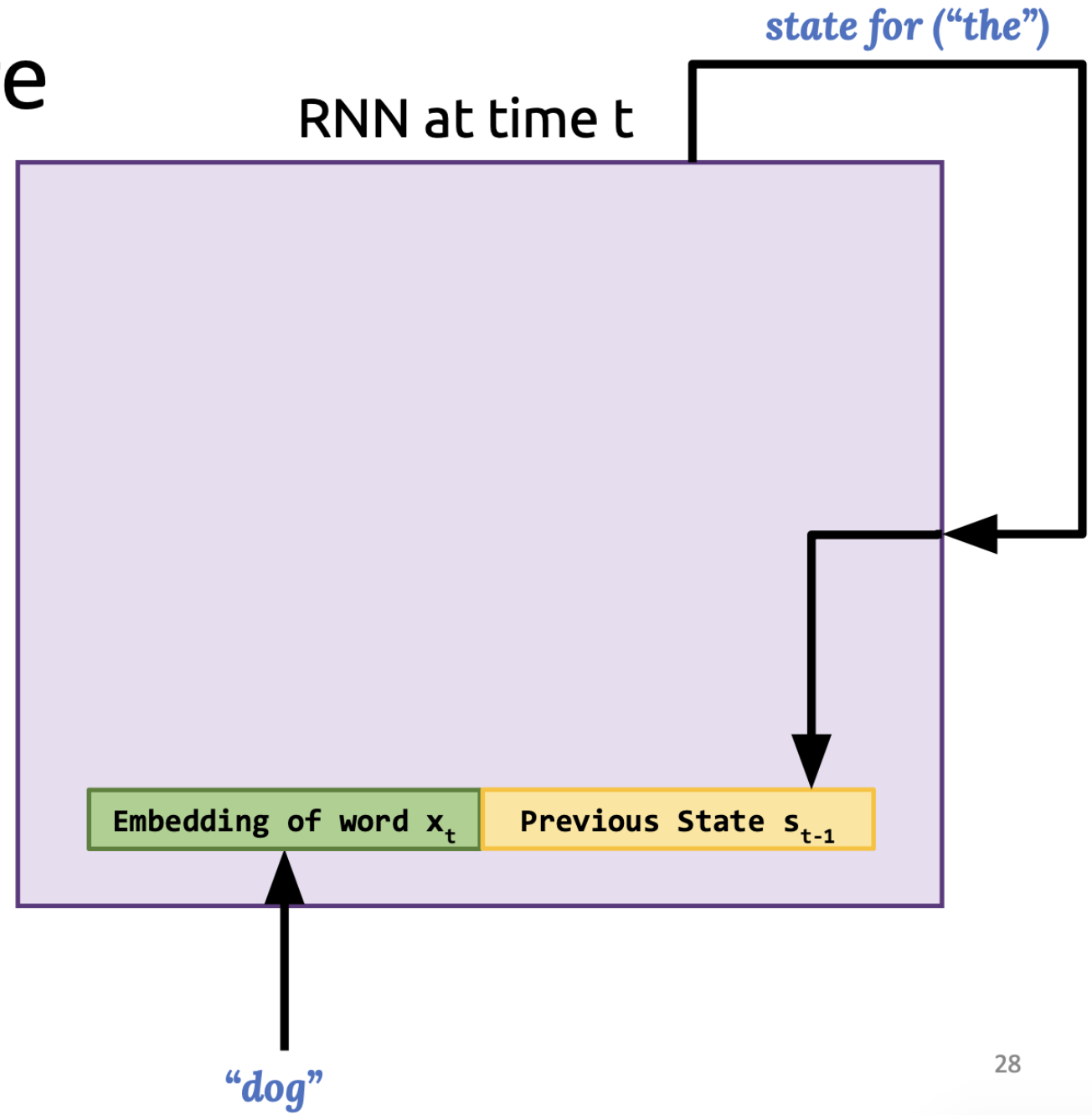
At each step of our RNN, we will get an input word, and a state vector from the previous cell.



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.

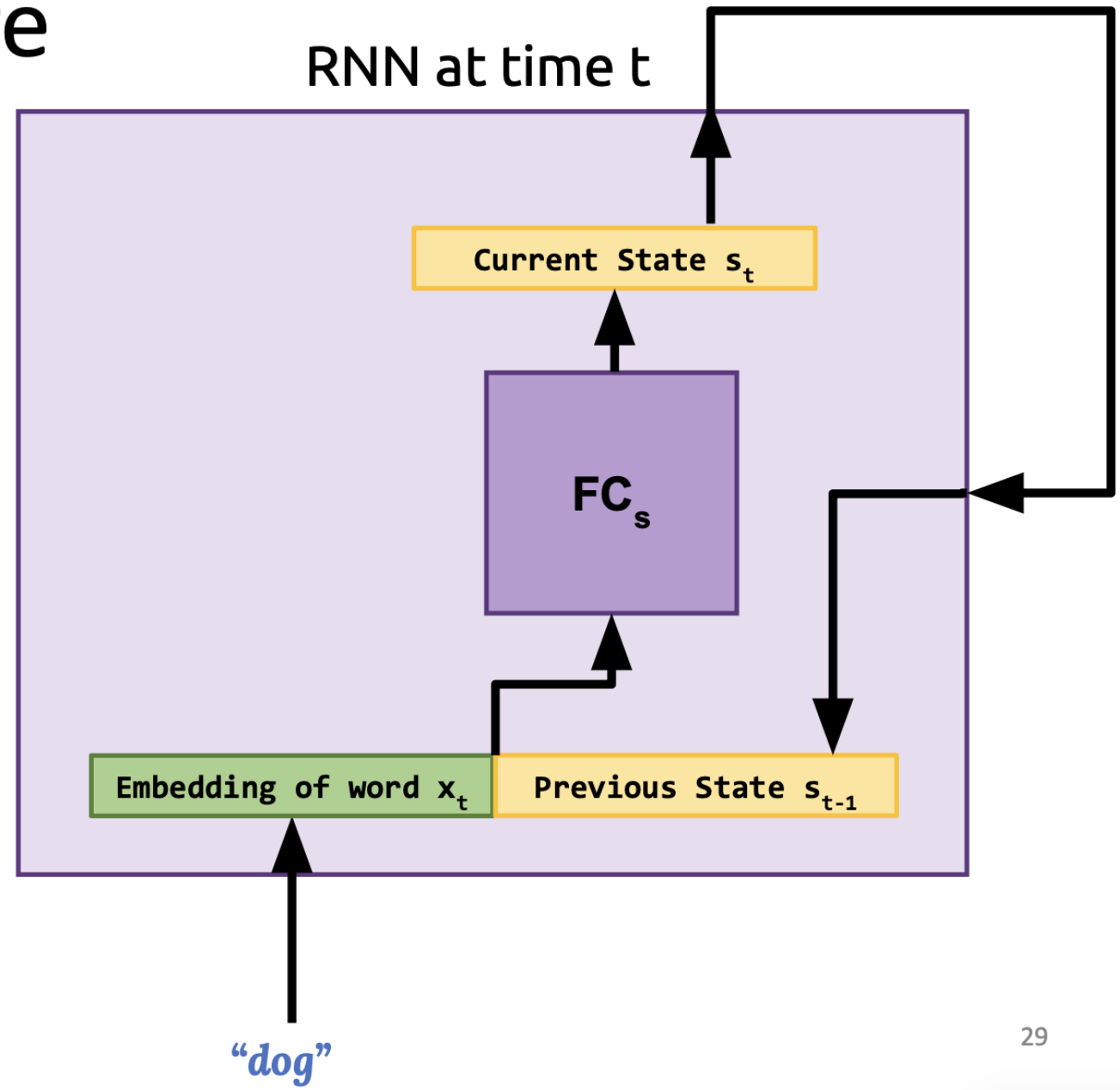


RNN Cell Architecture

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



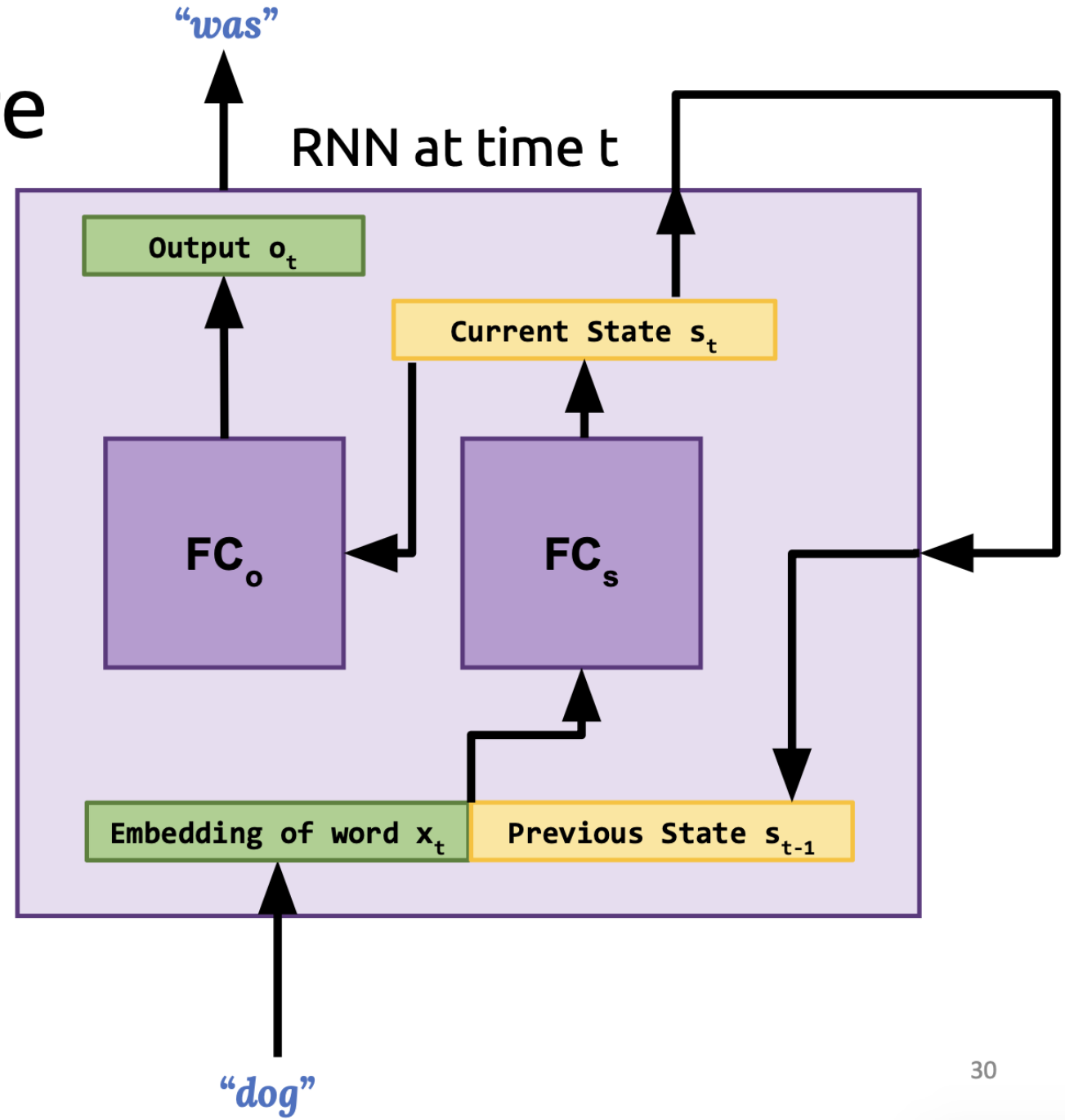
RNN Cell Architecture

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.

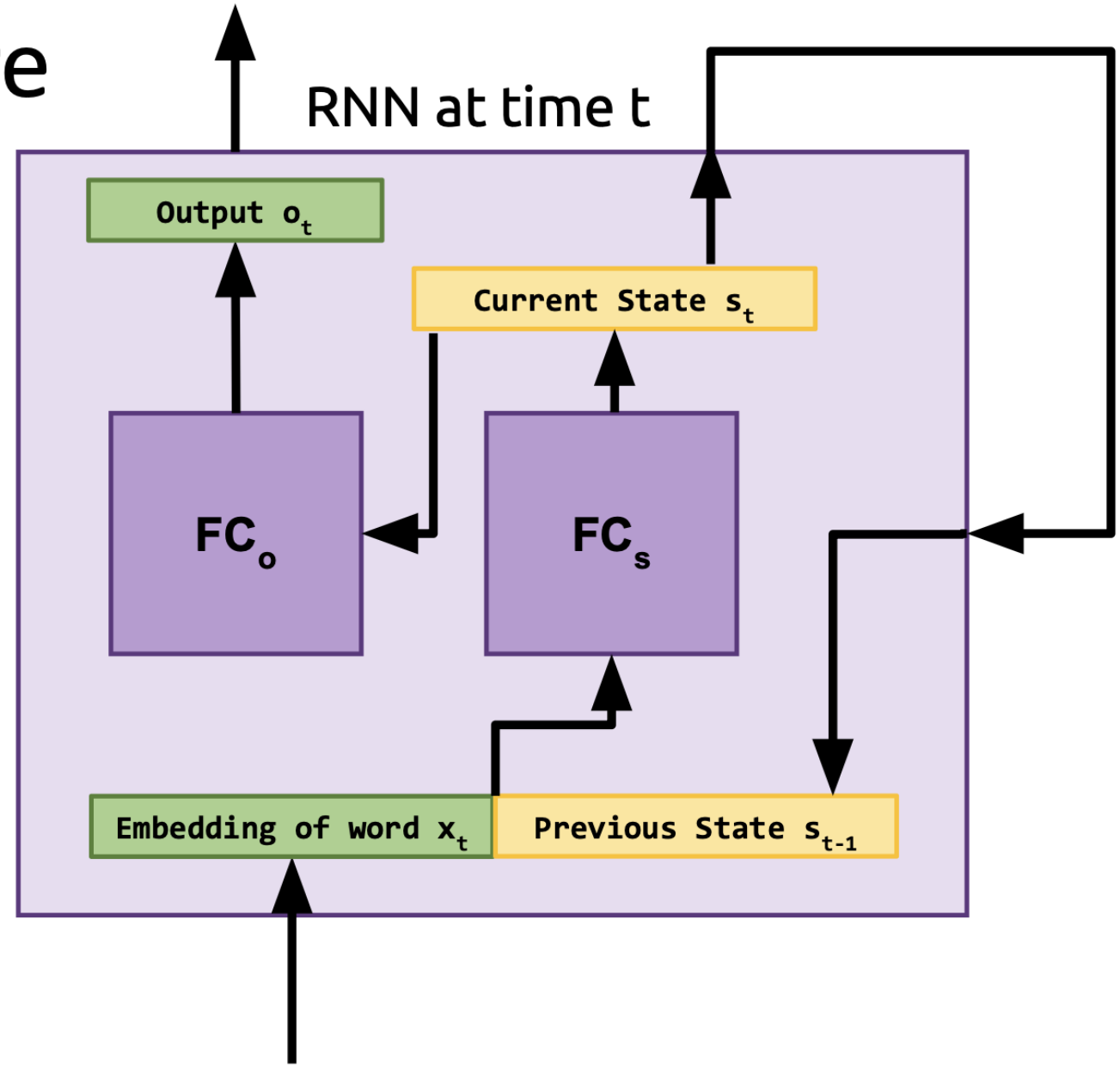


RNN Cell Architecture

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

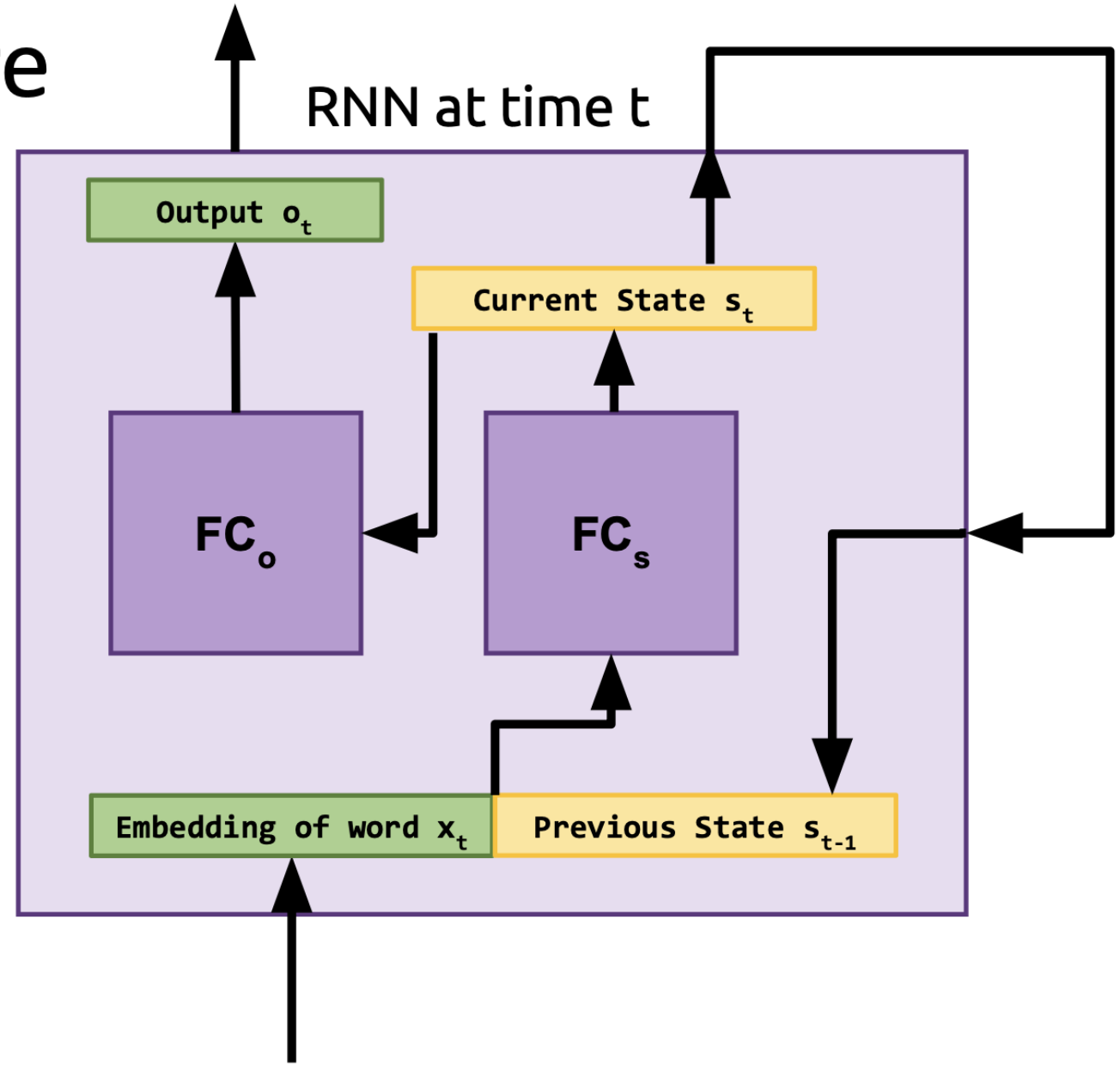
We can represent the RNN in with the following equations:

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

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

Nonlinear activations (e.g. sigmoid, tanh)

Any questions?



RNN Cell Architecture

We can represent the RNN in with the following equations:

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

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

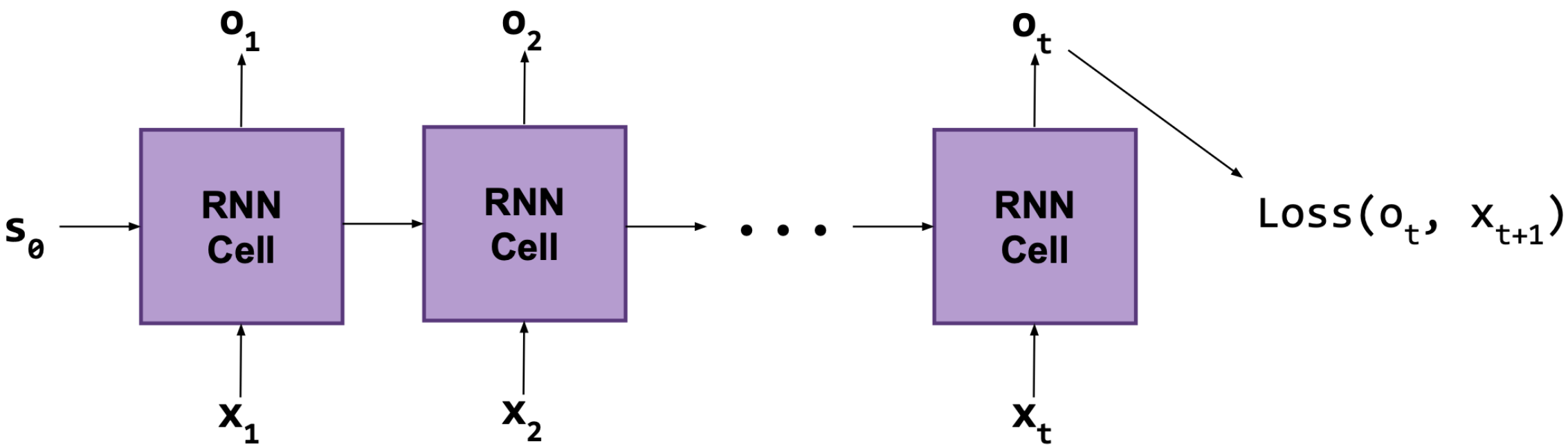
This brings up an immediate question: **what is s_0?**

Typically, we initialize s_0 to be a vector of zeros

(i.e. "initially, there is no memory of any previous words")

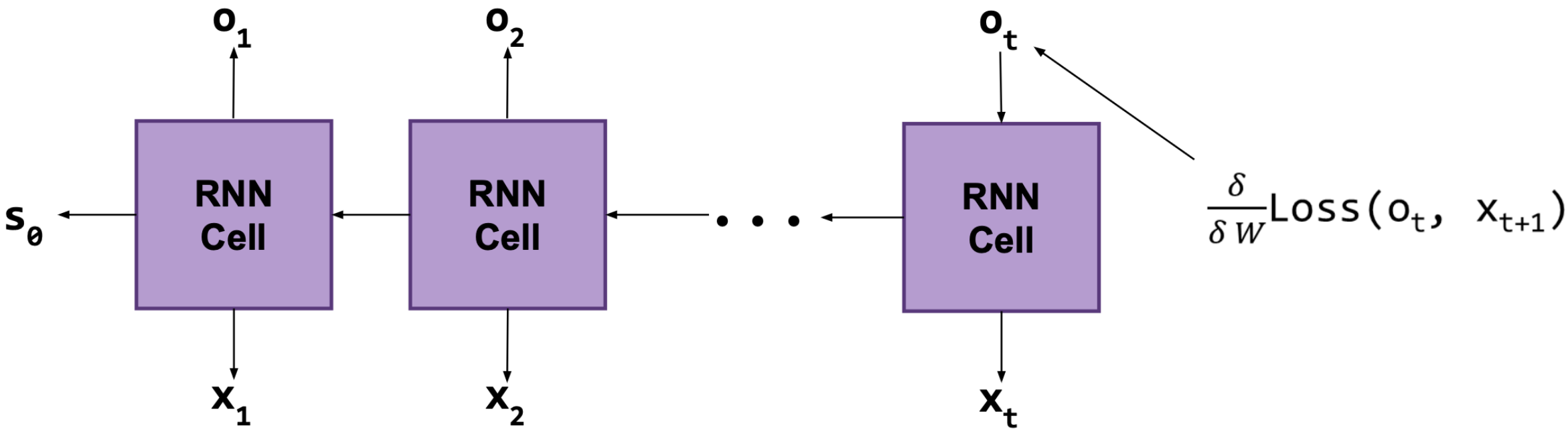
Training RNNs

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

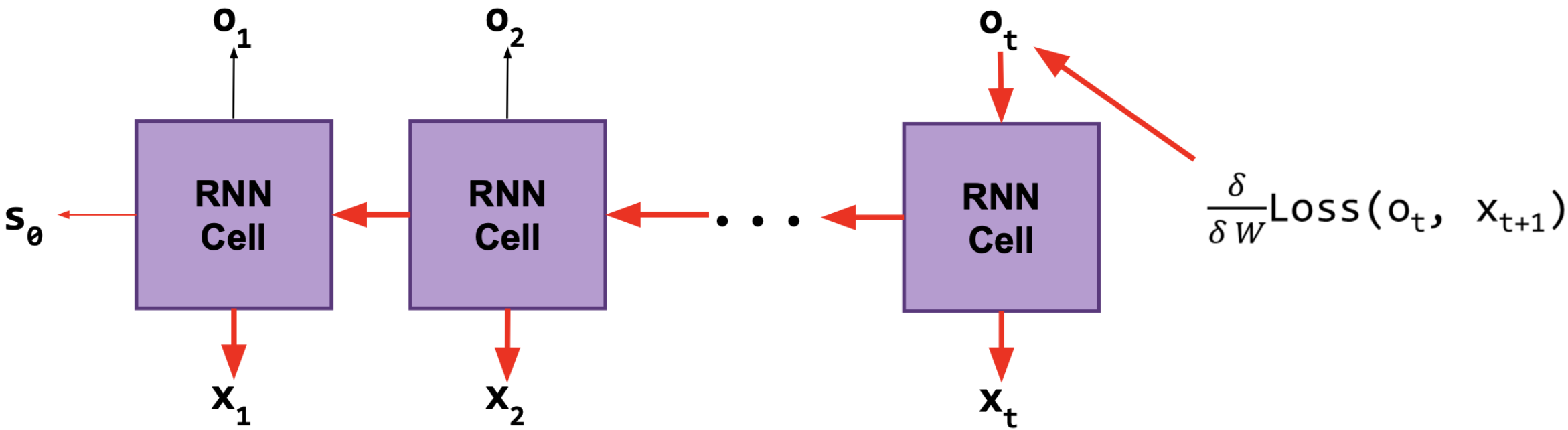
But what happens when we differentiate the loss and backpropagate?



Training RNNs

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

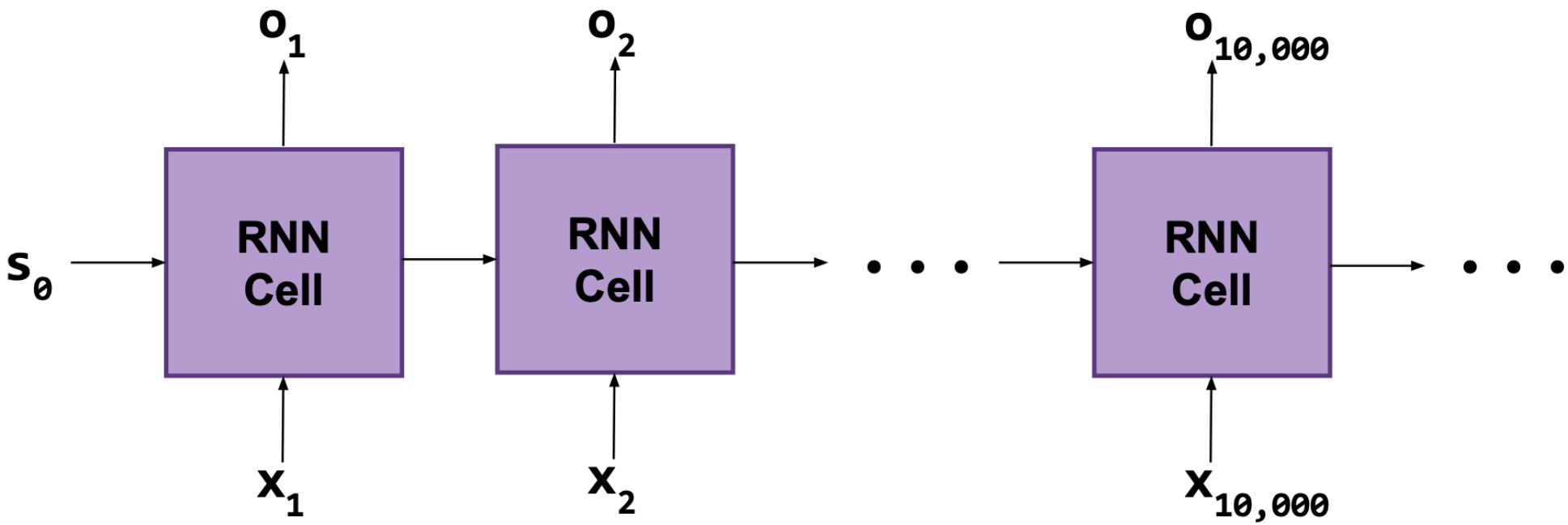
We call this *backpropagation through time.*



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`

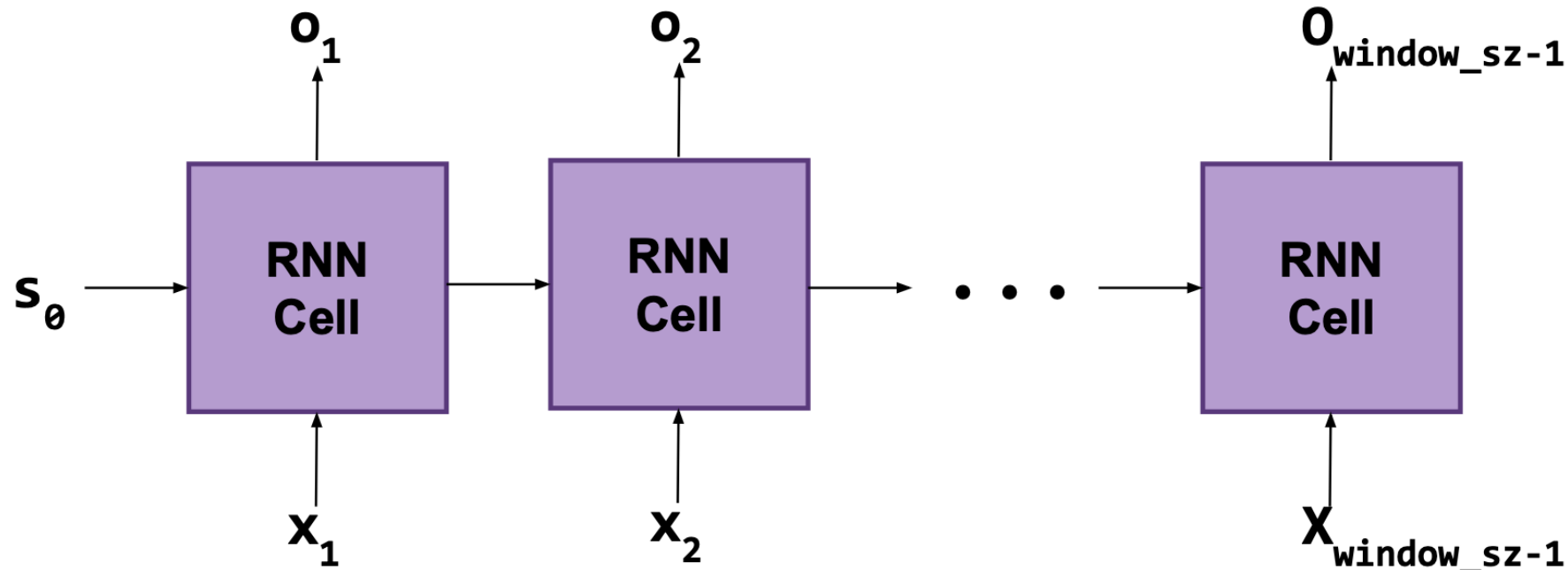
The new shape of our data should be:

`(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

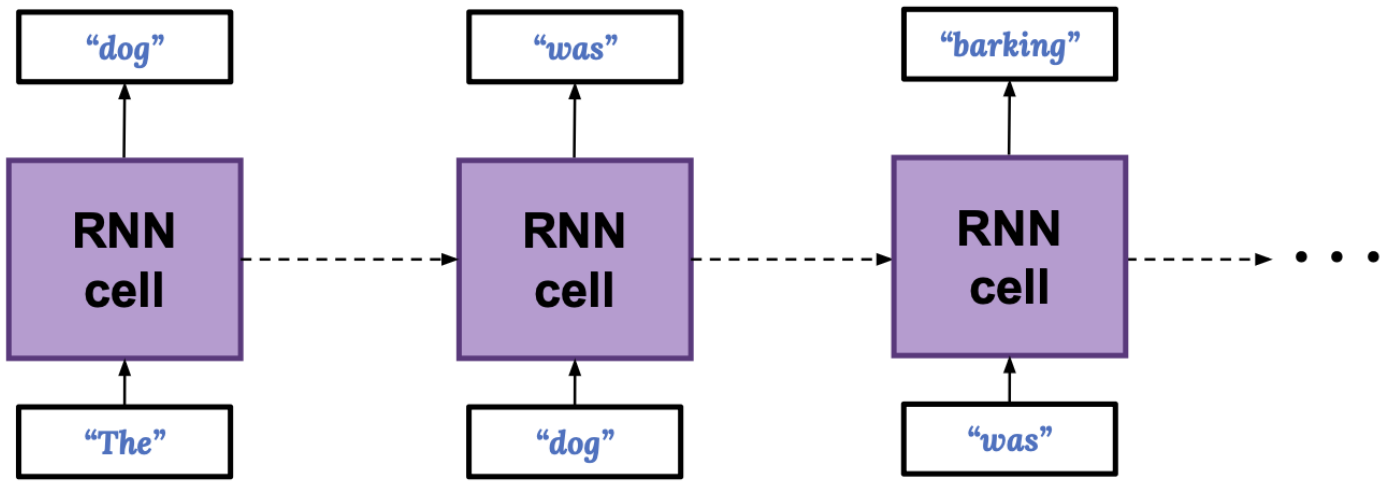


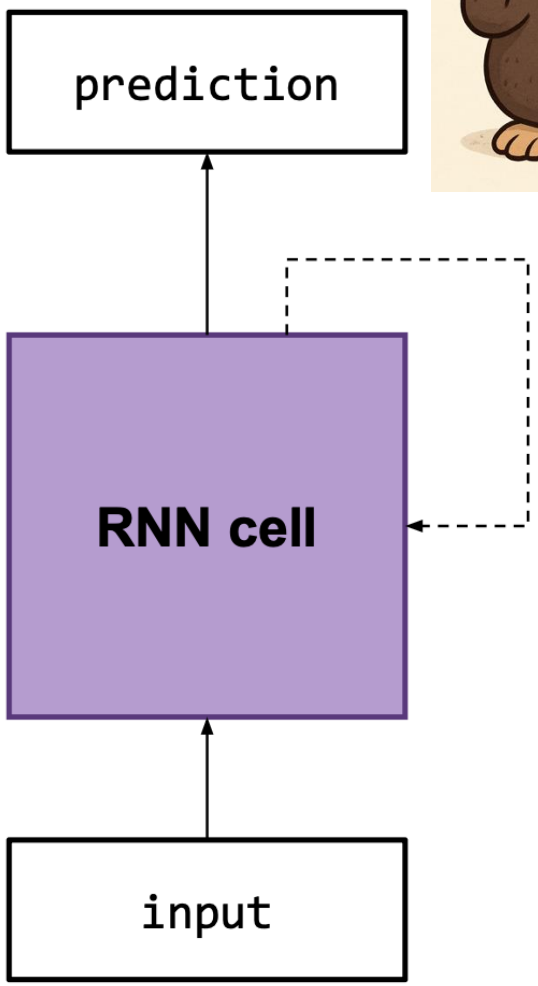
Any questions?

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

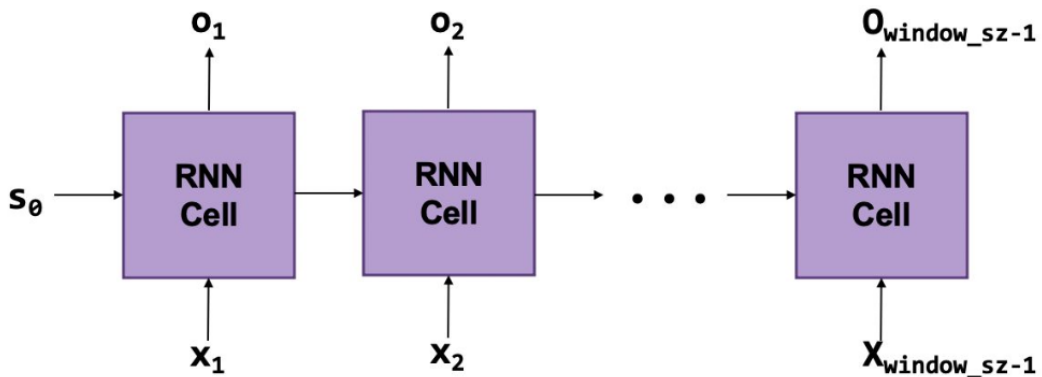
RNNs in Tensorflow

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.

Issue #1: Vanishing Gradients
Issue #2: What should we remember?

RNN Weaknesses

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

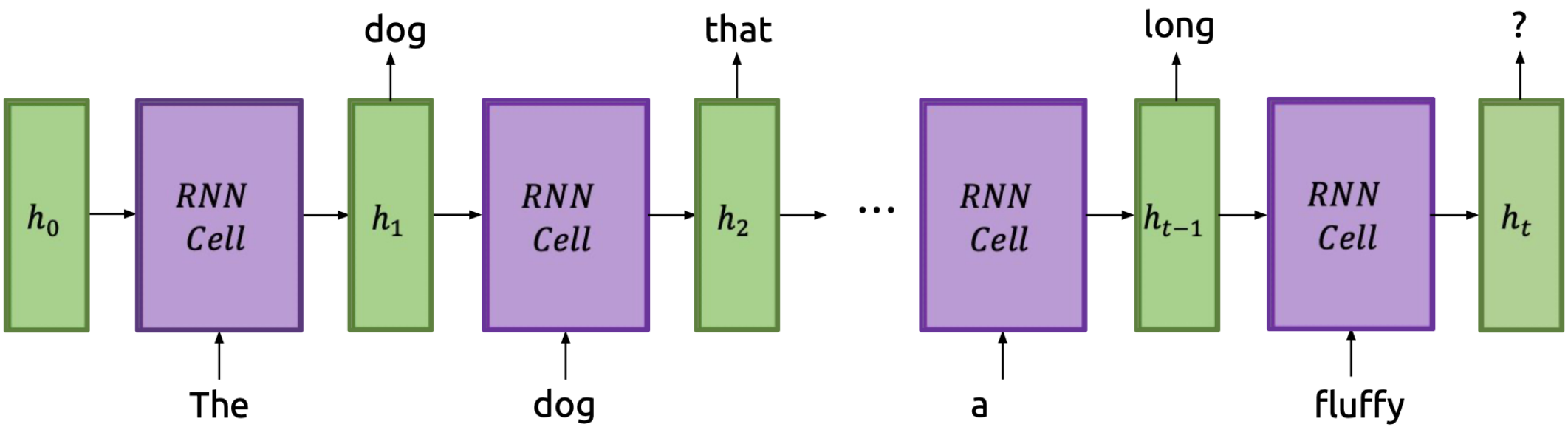
- To predict "tail" RNN needs to remember the subject of the sentence - "dog"

RNN Weaknesses

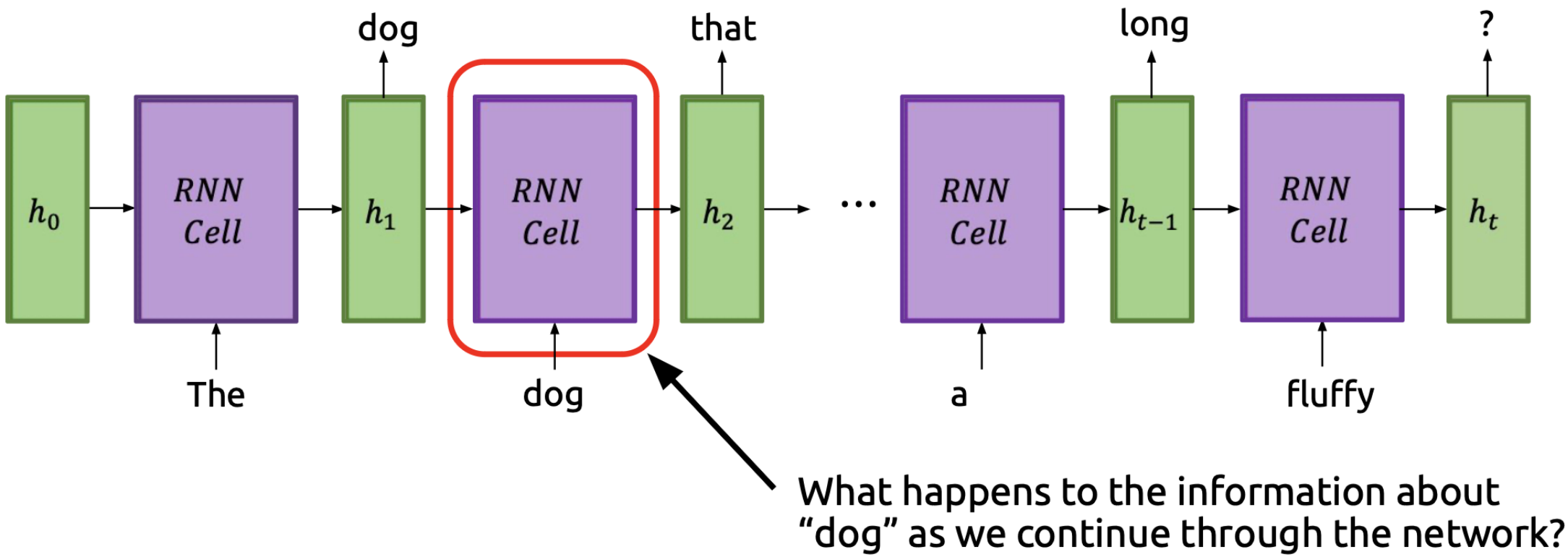
"The dog that my family had when I was a child had a fluffy _____."

- 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.

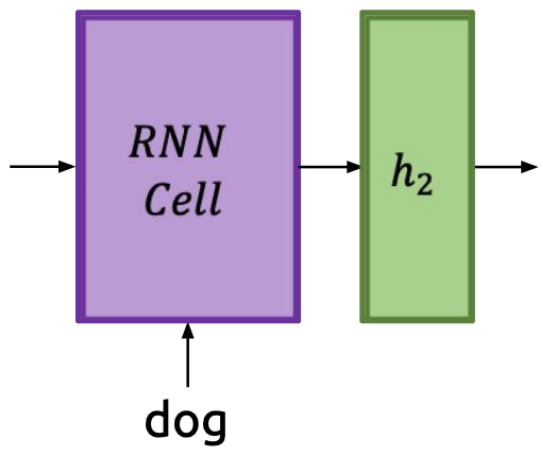
An Illustrative Example:



An Illustrative Example:

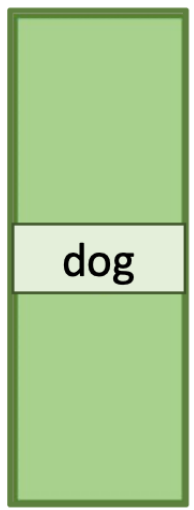


An Illustrative Example:



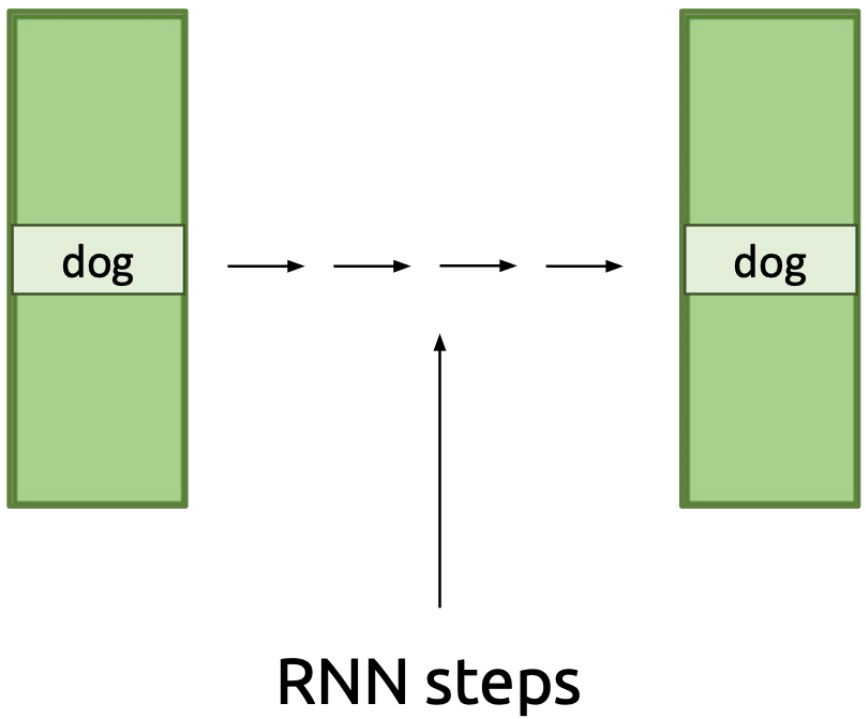
An Illustrative Example:

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



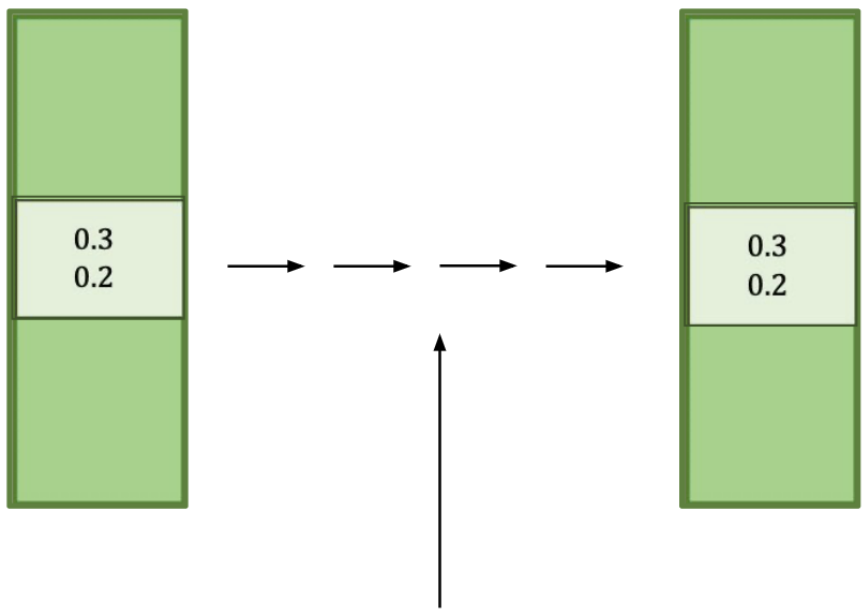
An Illustrative Example:

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



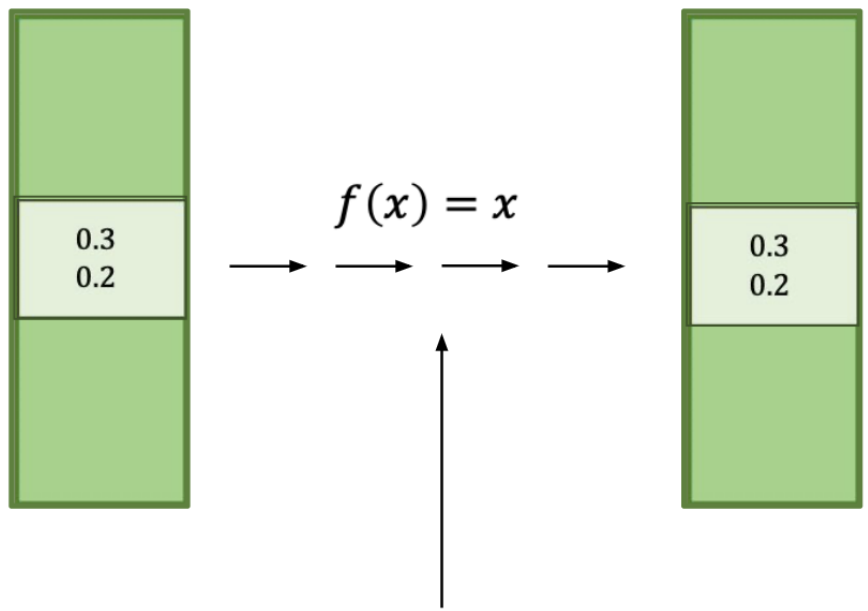
An Illustrative Example:

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?

How does this affect the hidden state?

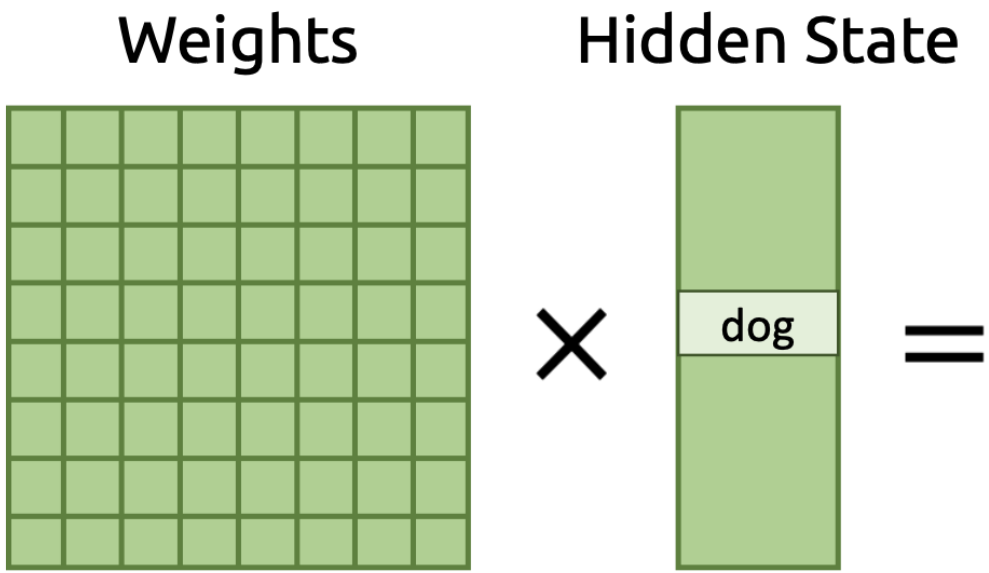
RNN update

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

The hidden state goes through a fully connected layer!

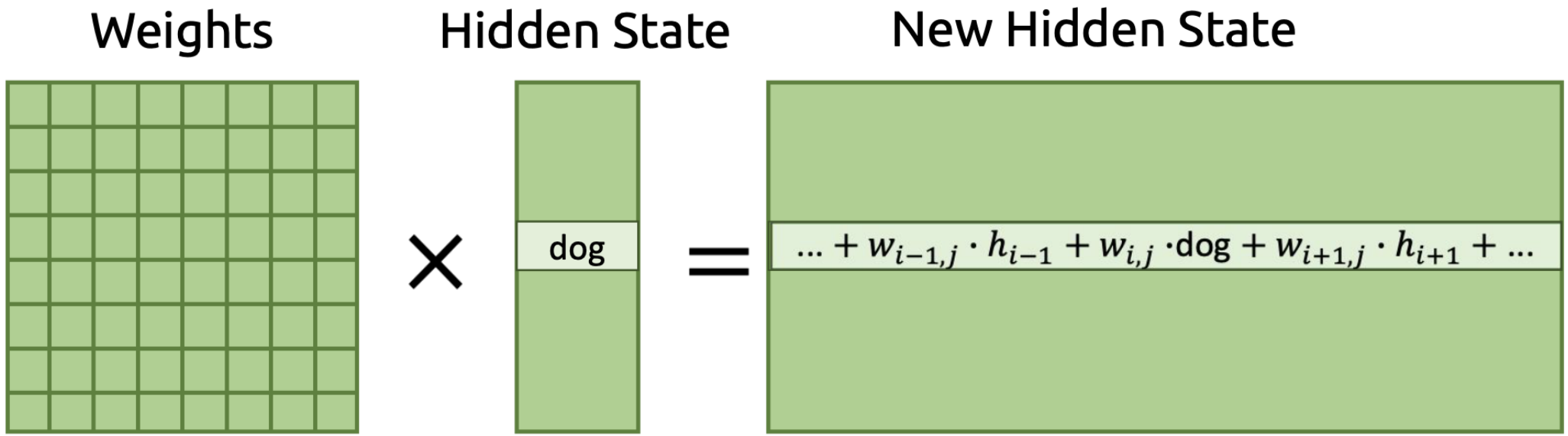
How does this affect the hidden state?

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



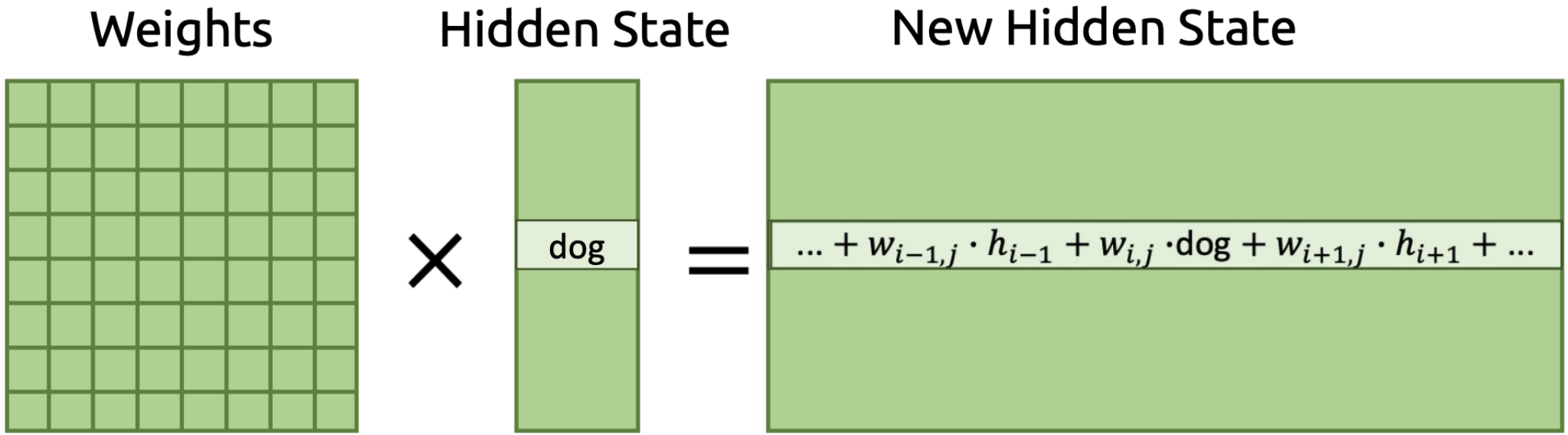
How does this affect the hidden state?

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



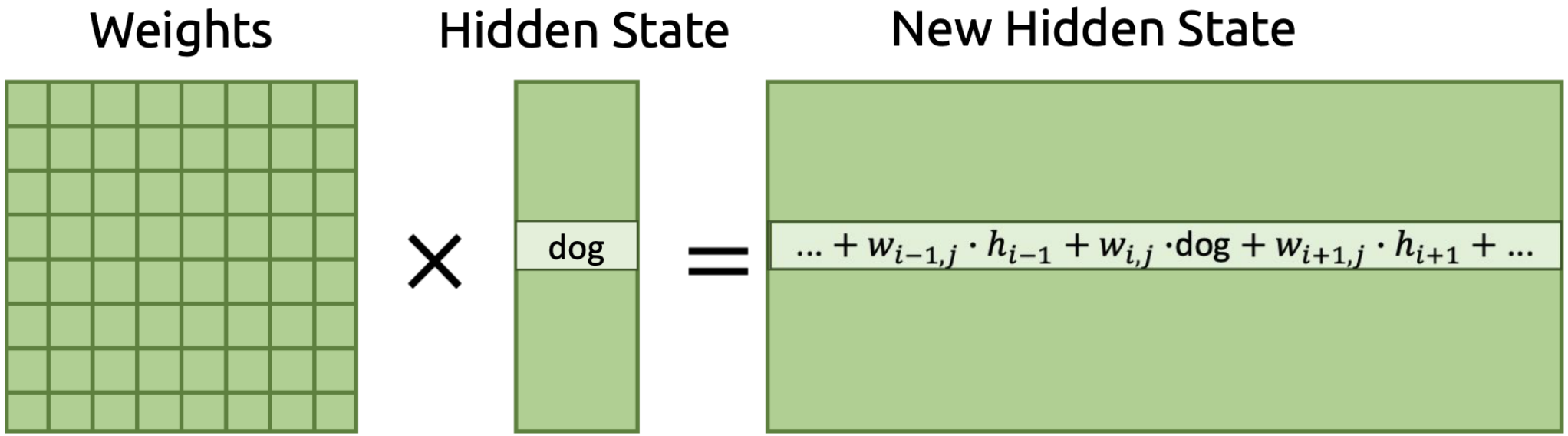
How does this affect the hidden state?

Dog gets lost in all the other information!



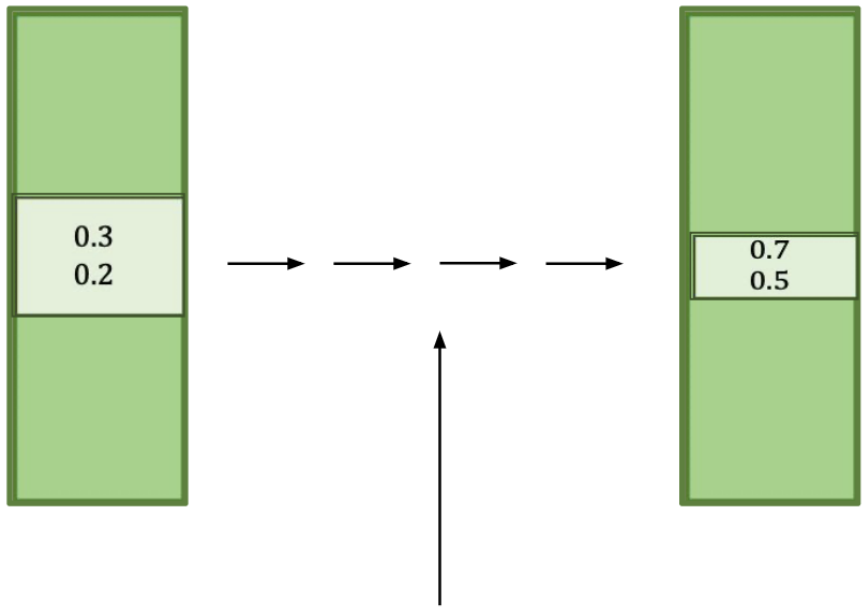
How does this affect the hidden state?

Dog gets lost in all the other information!



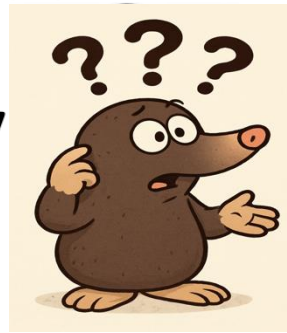
How does this affect the hidden state?

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

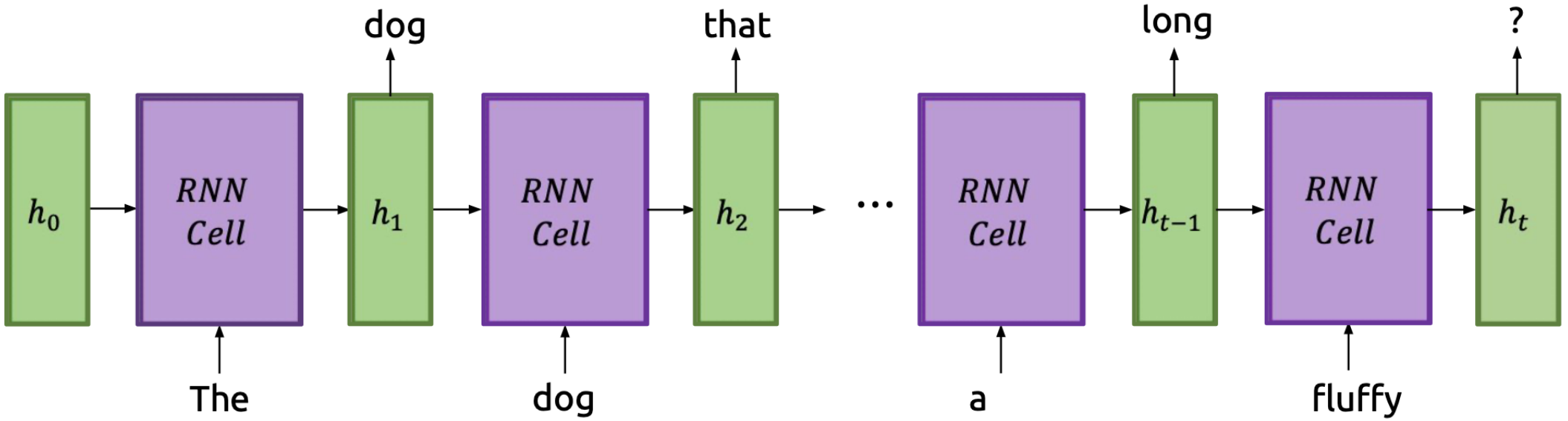


RNN forgets about the dog after a certain time ☹

Any questions?



RNNs cannot learn “long term” dependency

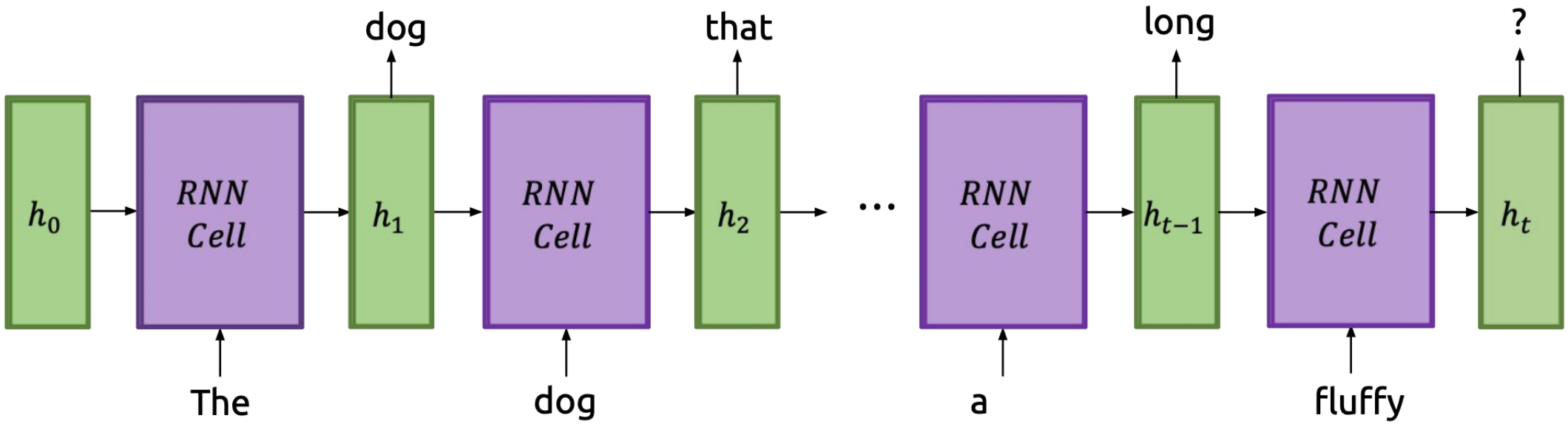


We need new way to update hidden state!

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



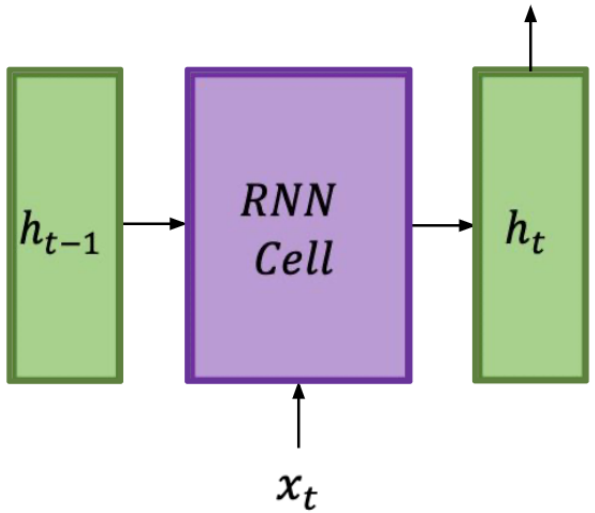
An analogy to human (or computer) memory:

- 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

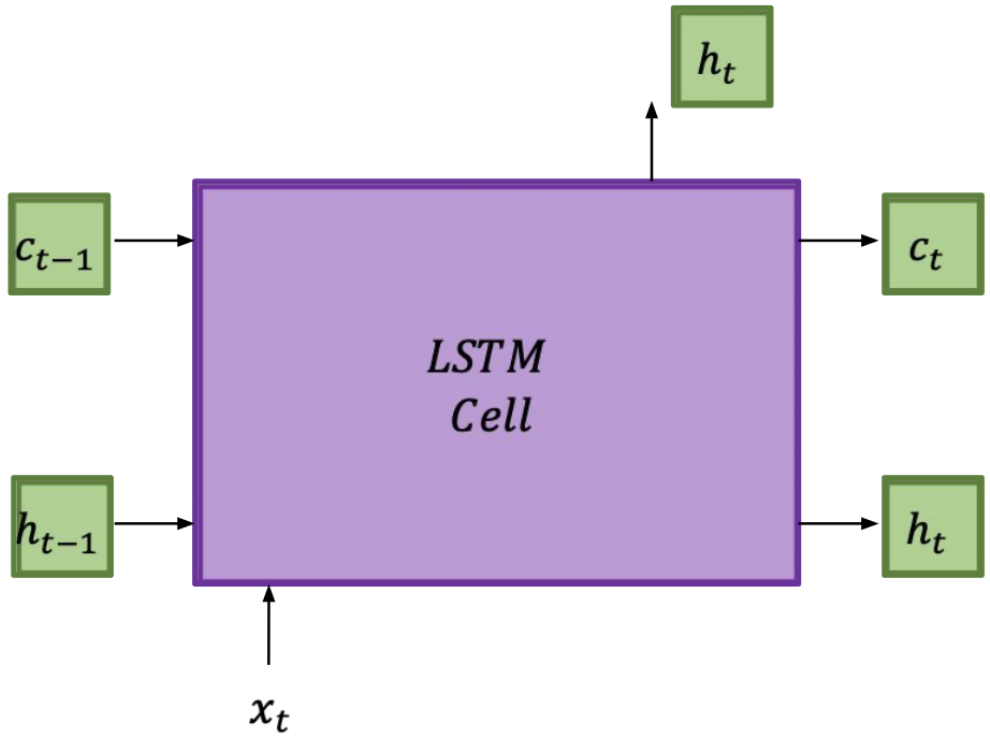
- ***Long*** Short Term Memory (LSTM)
  - "Short-term memory that persists over time"
  - i.e. "hidden states that remember information for longer"

What is different?

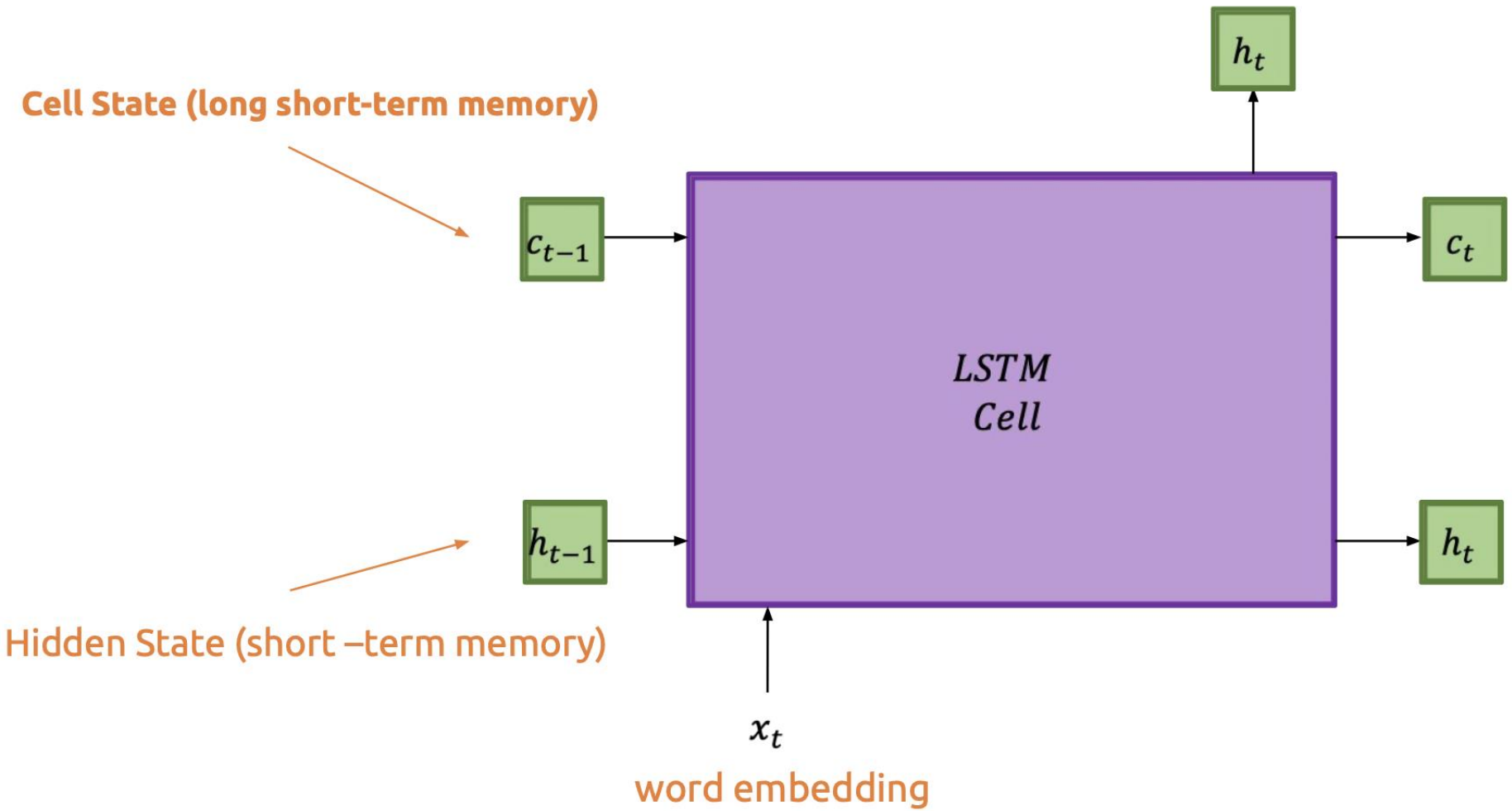
Vanilla RNN



LSTM



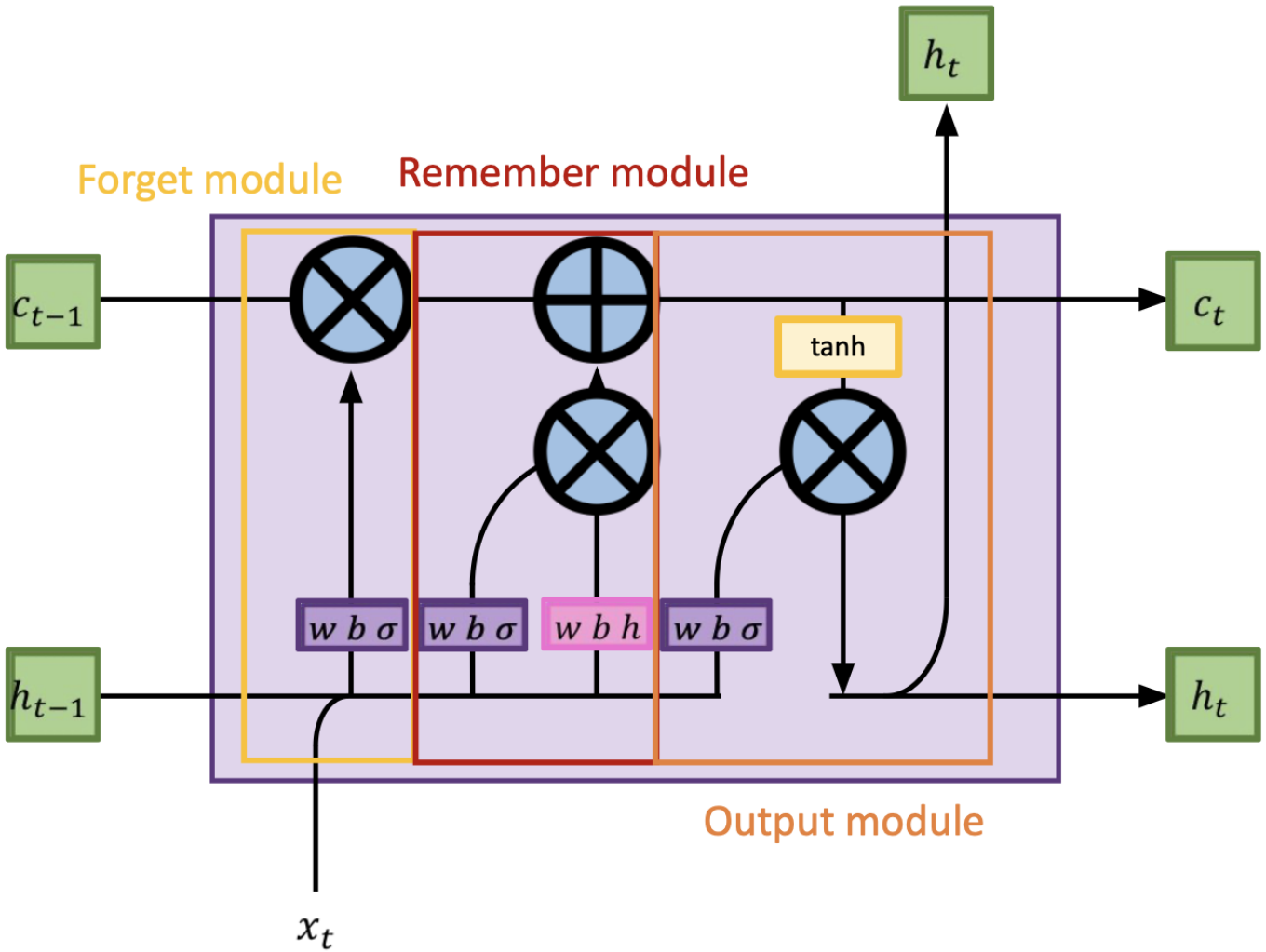
LSTM


Cell State (long short-term memory)
h_t
c_{t-1}
c_t
LSTM
Cell
h_{t-1}
h_t
Hidden State (short –term memory)
x_t
word embedding

How an LSTM works

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

The Complete LSTM



Forget Module

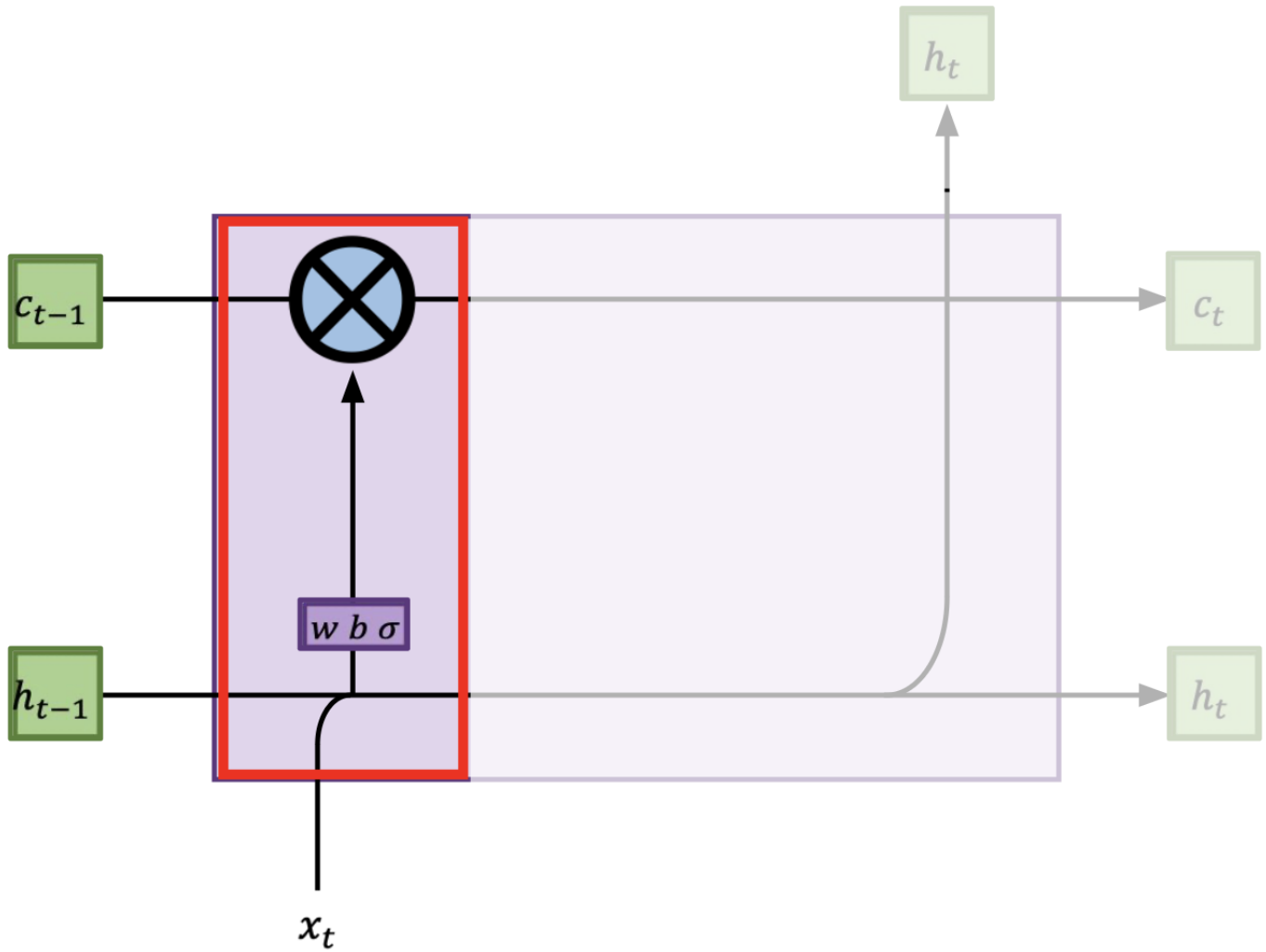
Say we just predicted *“tail”* in *“My dog has a fluffy _____.”*

Next set of words: *“I love my dog”*

Forget Module

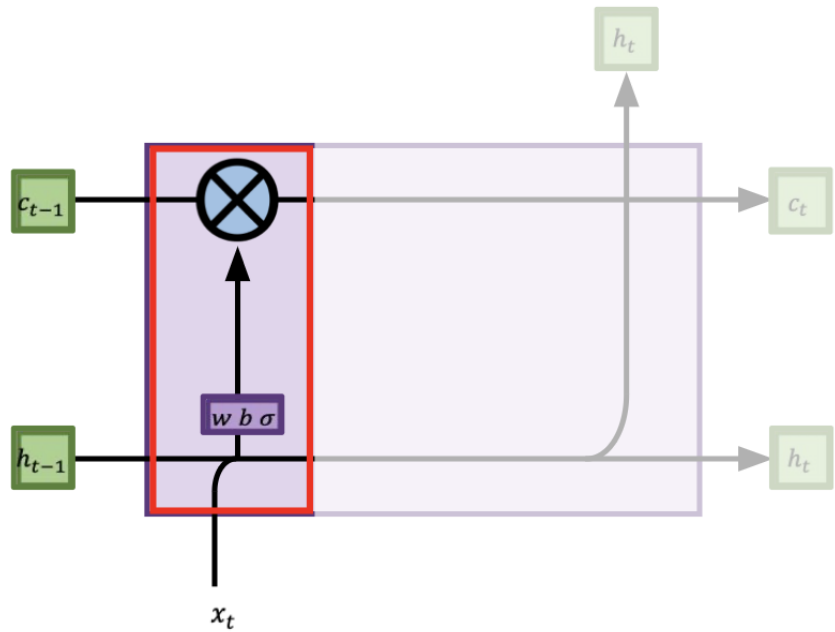
- Model no longer needs to know about *“dog”*
- Ready to **delete** information about subject

Forget Module



Forget Module

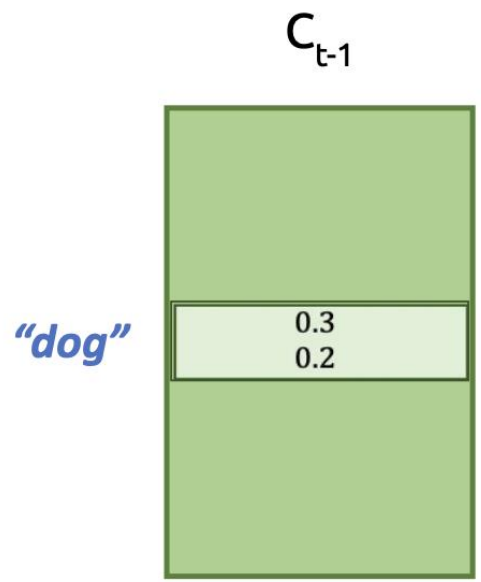
- Filters out what gets allowed into the LSTM cell from the last state
  - Example: If it's remembering gender pronouns, and a new subject is seen, it will forget the old gender pronouns
- Either lets parts of C_{t-1} pass through or not



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

Forgetting information

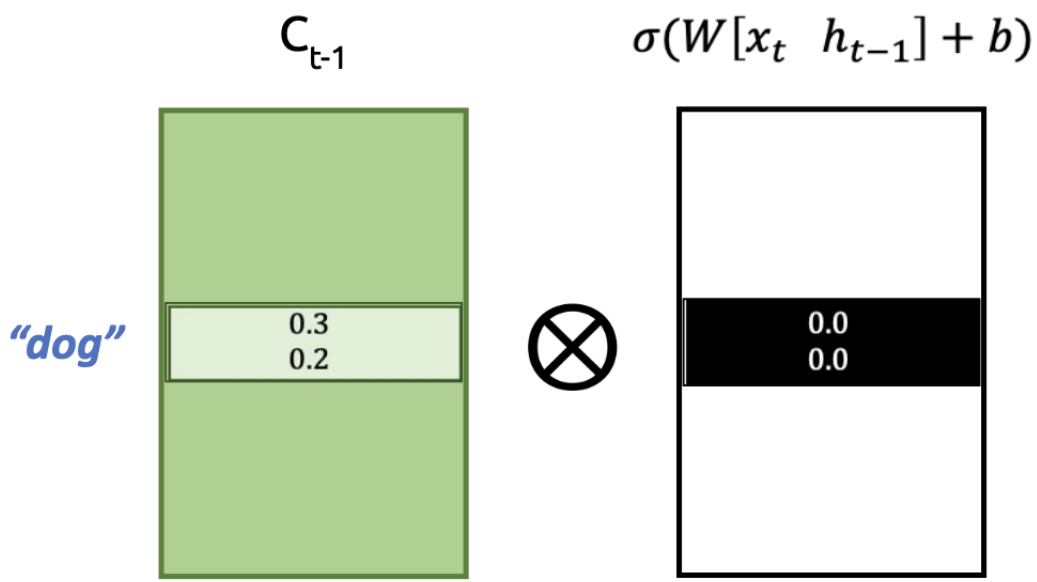
- Use pointwise multiplication by a **mask vector** to forget information
  - What do we want to forget from last cell state?



My dog has a fluffy tail. I love my dog

Forgetting information

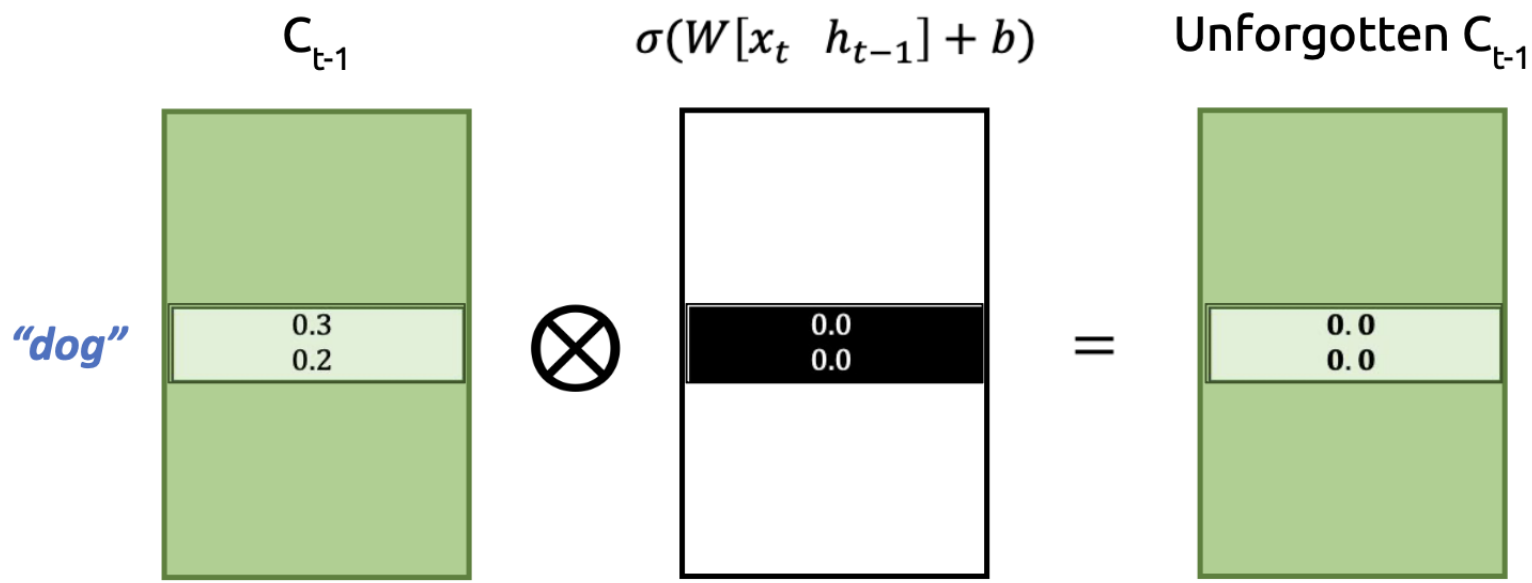
- Use pointwise multiplication by a **mask vector** to forget information
  - What do we want to forget from last cell state?
  - Output of fully connected + sigmoid is what we want to forget



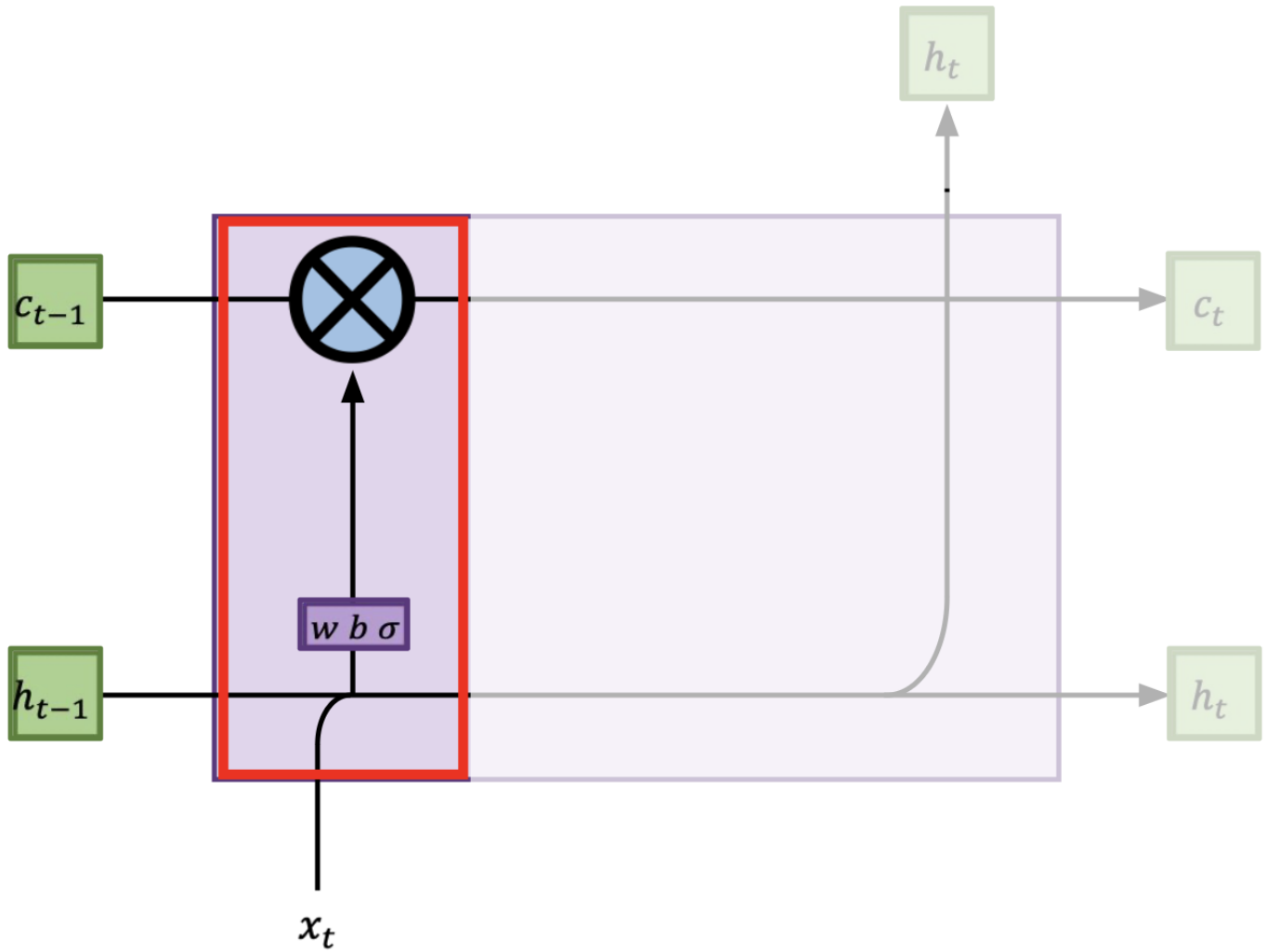
*My dog has a fluffy tail. **I love my dog***

Forgetting information

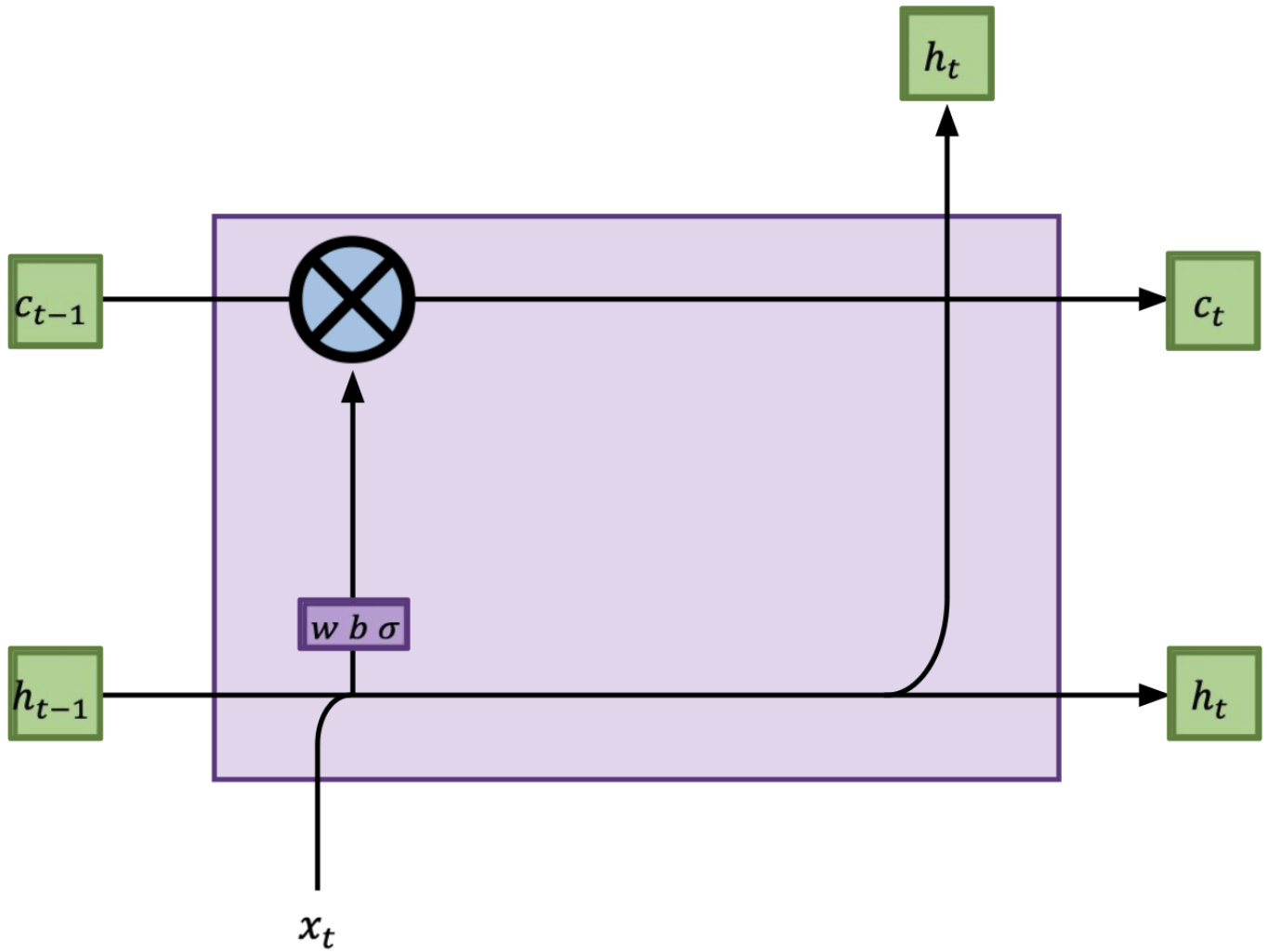
- Use pointwise multiplication by a **mask vector** to forget information
  - What do we want to forget from last cell state?
  - Output of fully connected + sigmoid is what we want to forget
  - "Zeros out" a part of the cell state
  - Pointwise multiplication by a learned mask vector is known as ***gating***



Forget Module

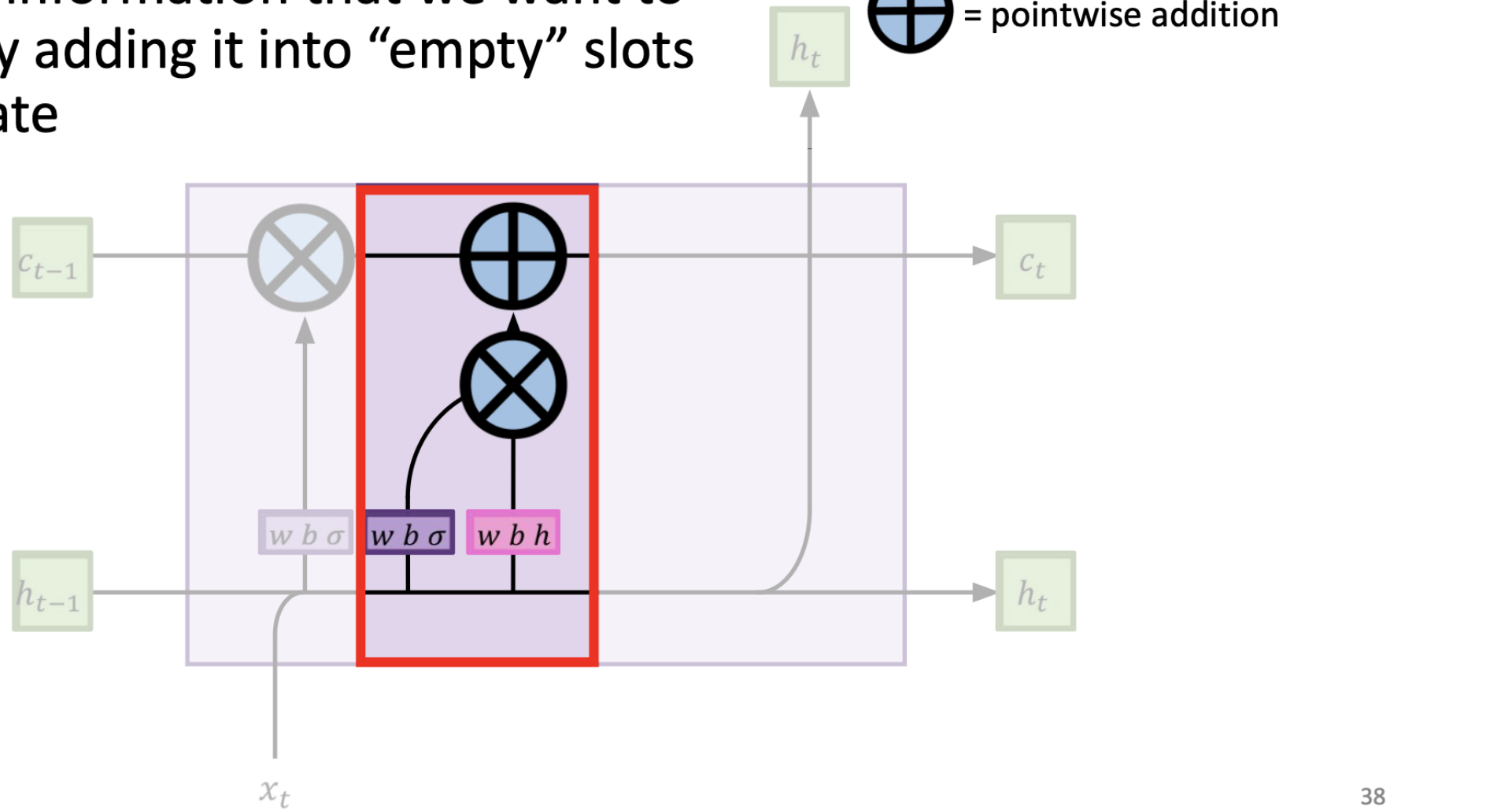


What's next?


$w\ b\ \sigma$ = fully connected layer with sigmoid
= pointwise multiplication
h_t
c_{t-1}
c_t
$w\ b\ \sigma$
h_{t-1}
h_t
x_t

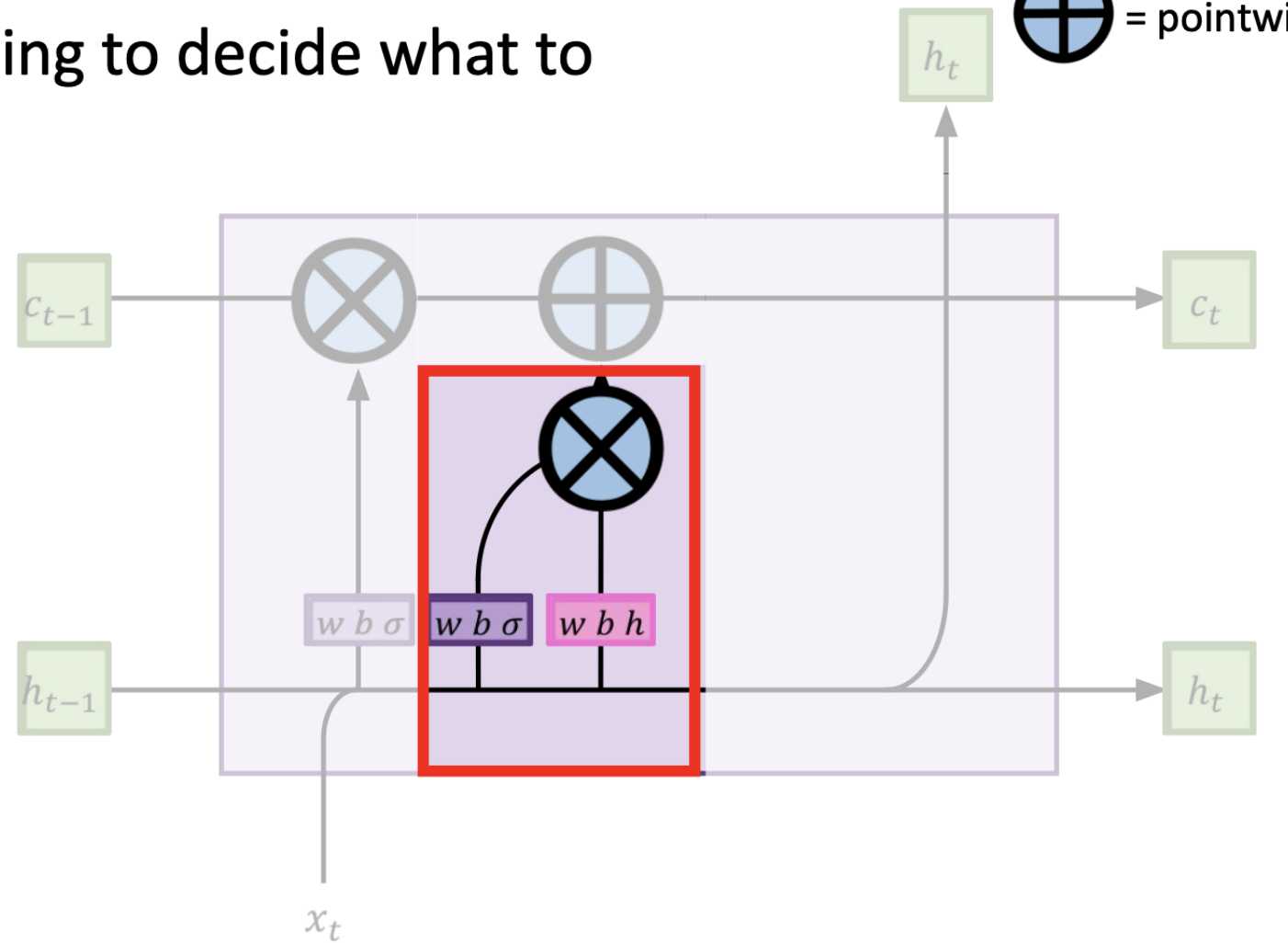
Remember Module

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



Remember Module

- First: use gating to decide what to remember

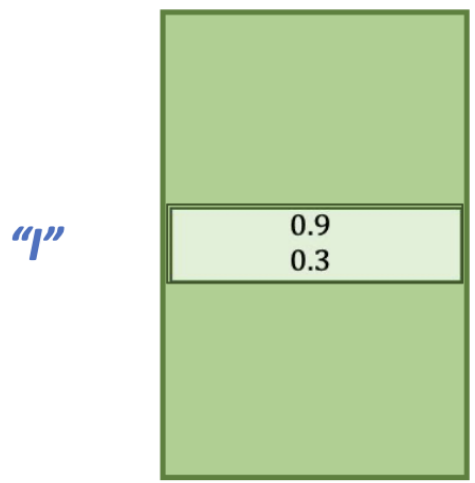


My dog has a fluffy tail. ***I love my dog***

Gating for ‘selective memory’

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

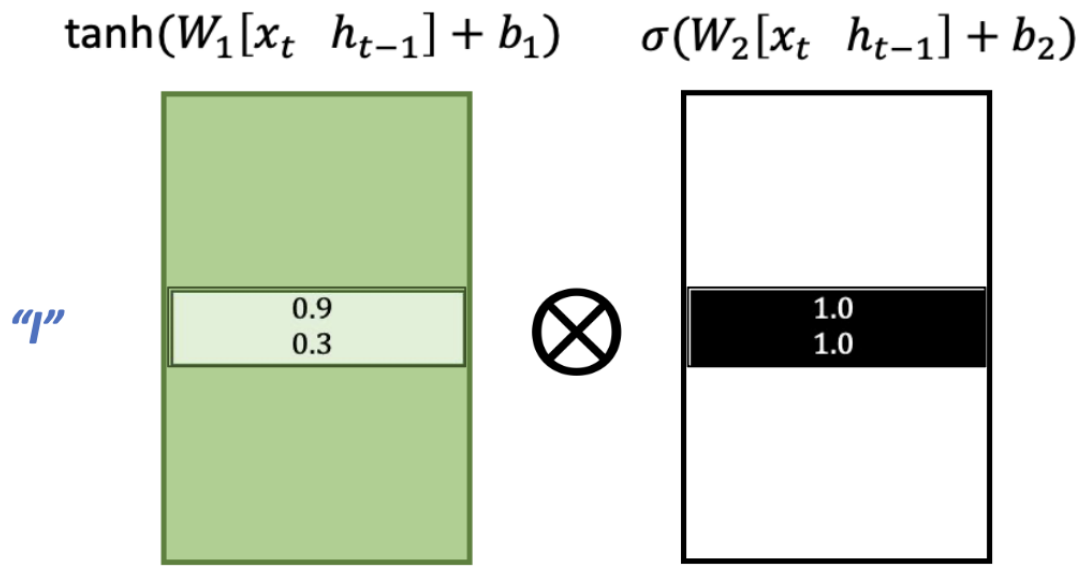
$$\tanh(W_1[x_t \quad h_{t-1}] + b_1)$$



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

Gating for ‘selective memory’

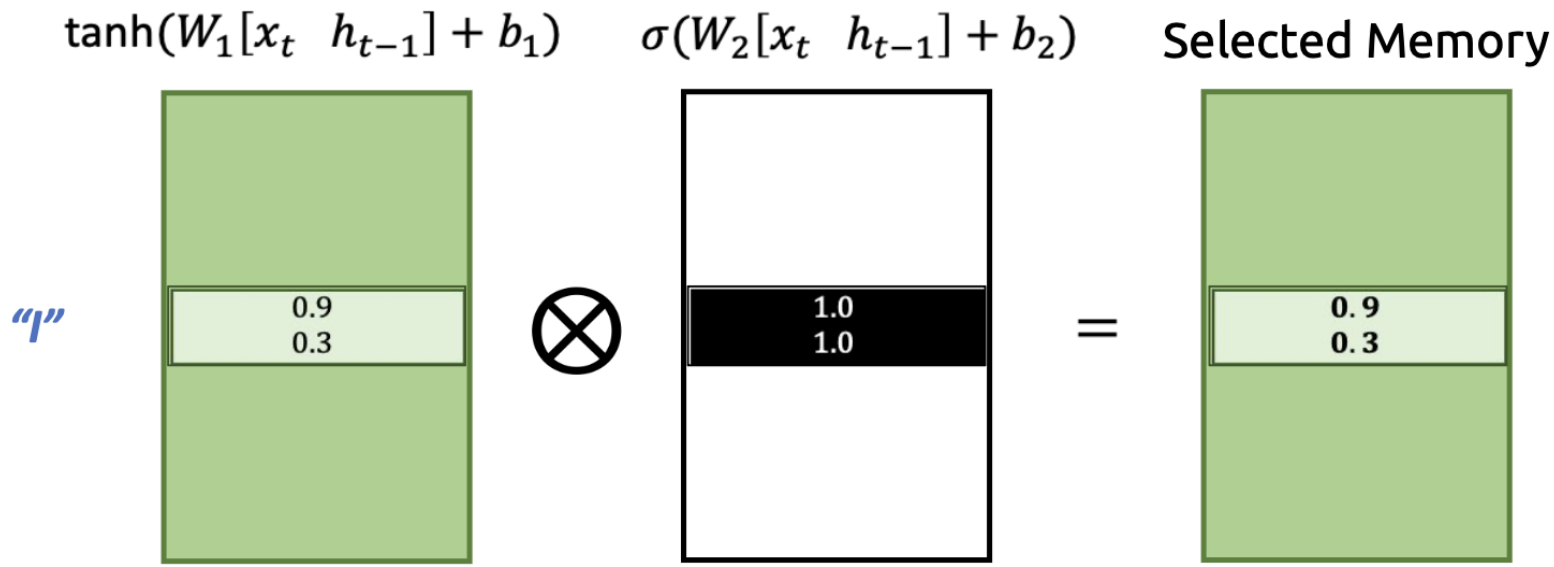
- A fully-connected + tanh on [input, memory] computes some new memory
- We gate this memory to decide what bits of it we want to remember long-term in the cell state



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

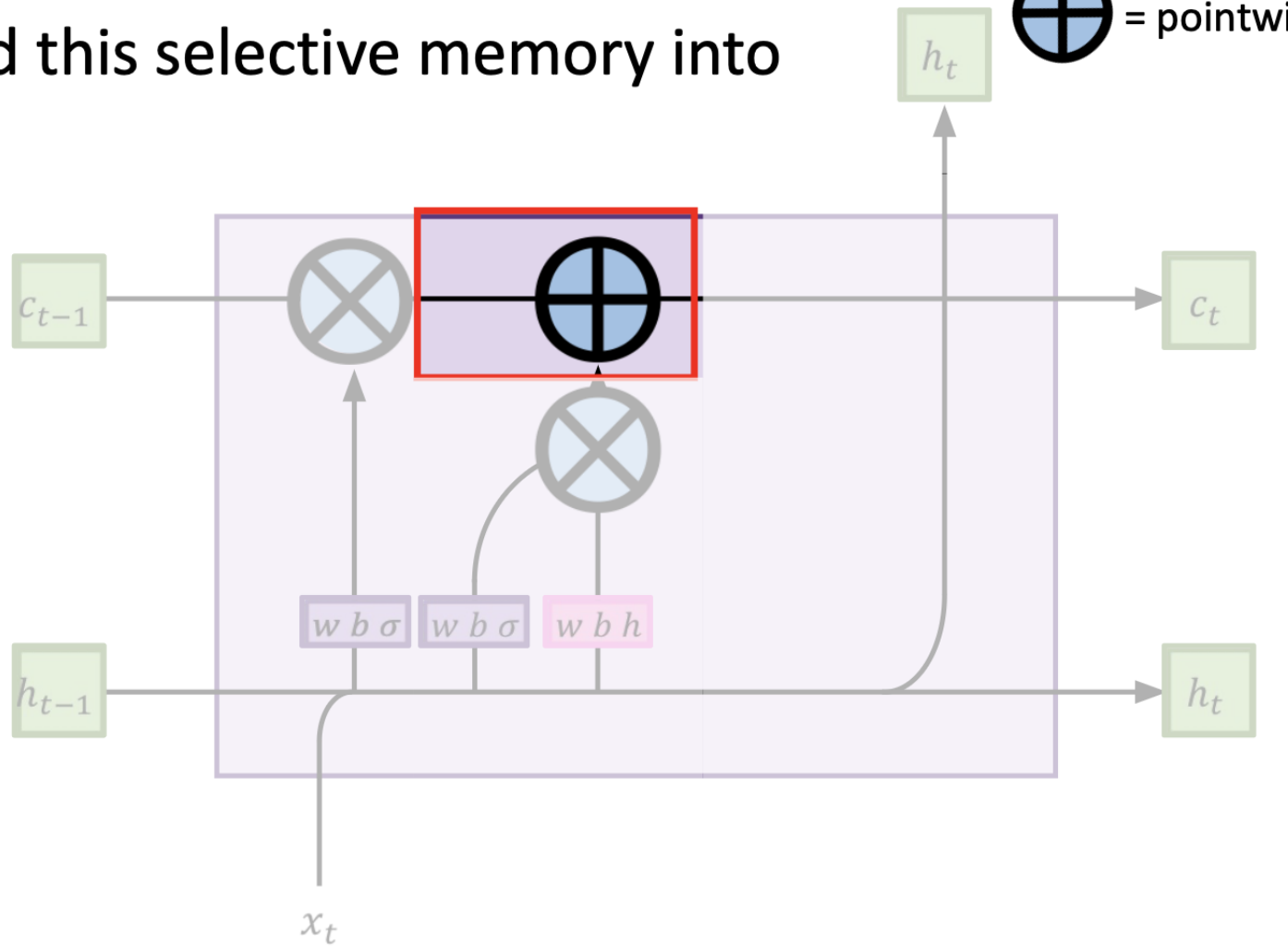
Gating for ‘selective memory’

- 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

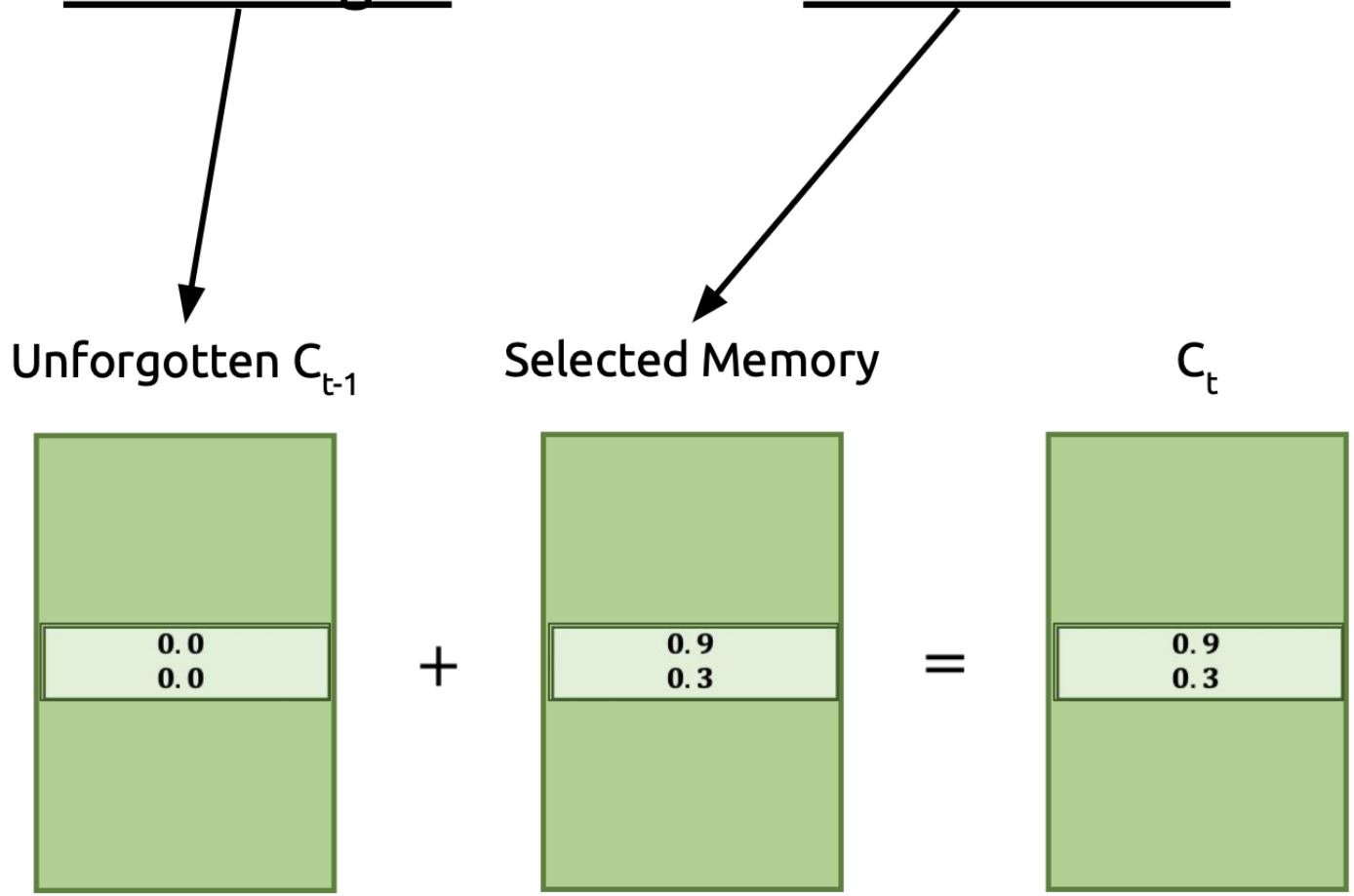
- Then: we add this selective memory into the cell state



My dog has a fluffy tail. I love my dog

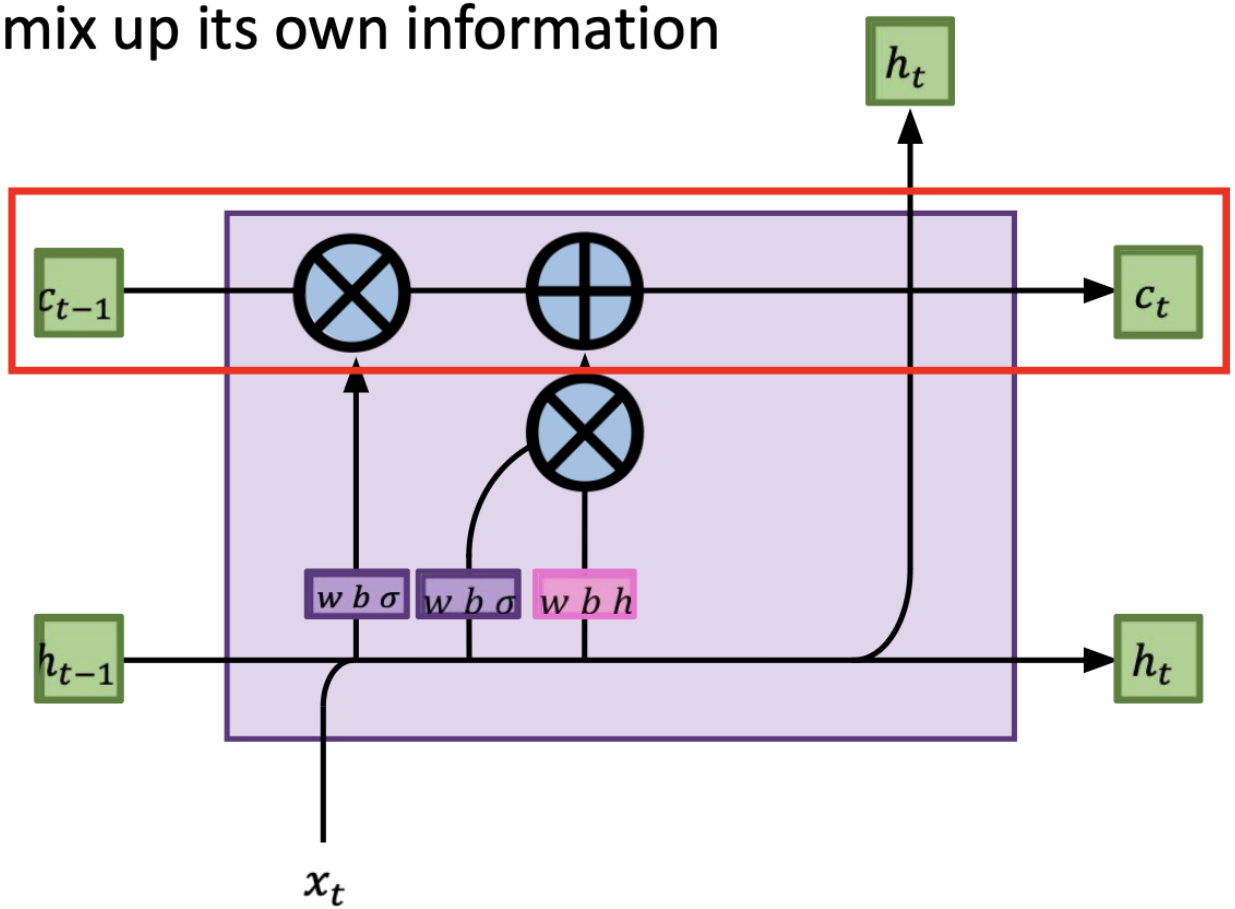
Remembering information

- Add what we didn't forget to what we did remember

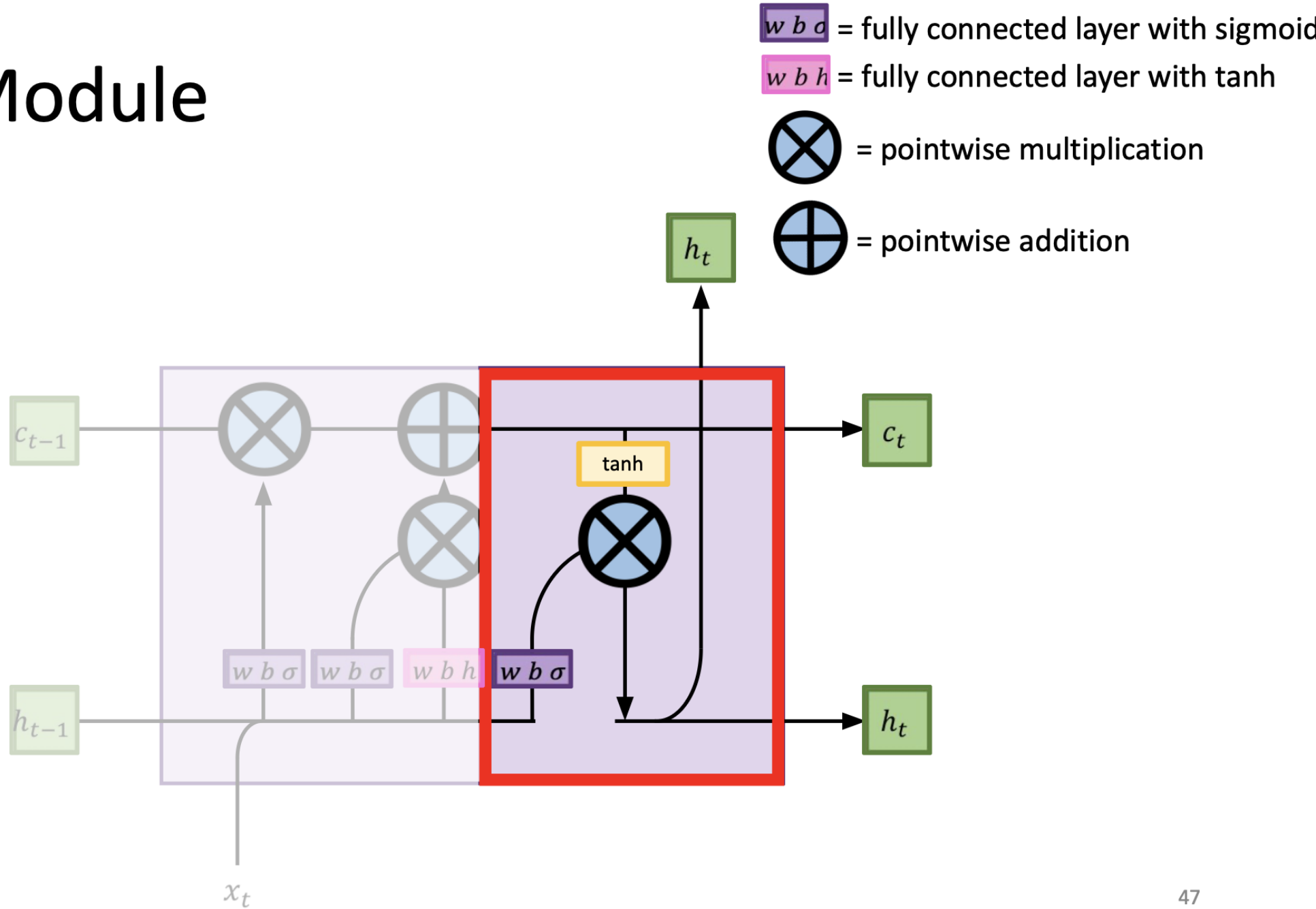


Why does this solve our problem?

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

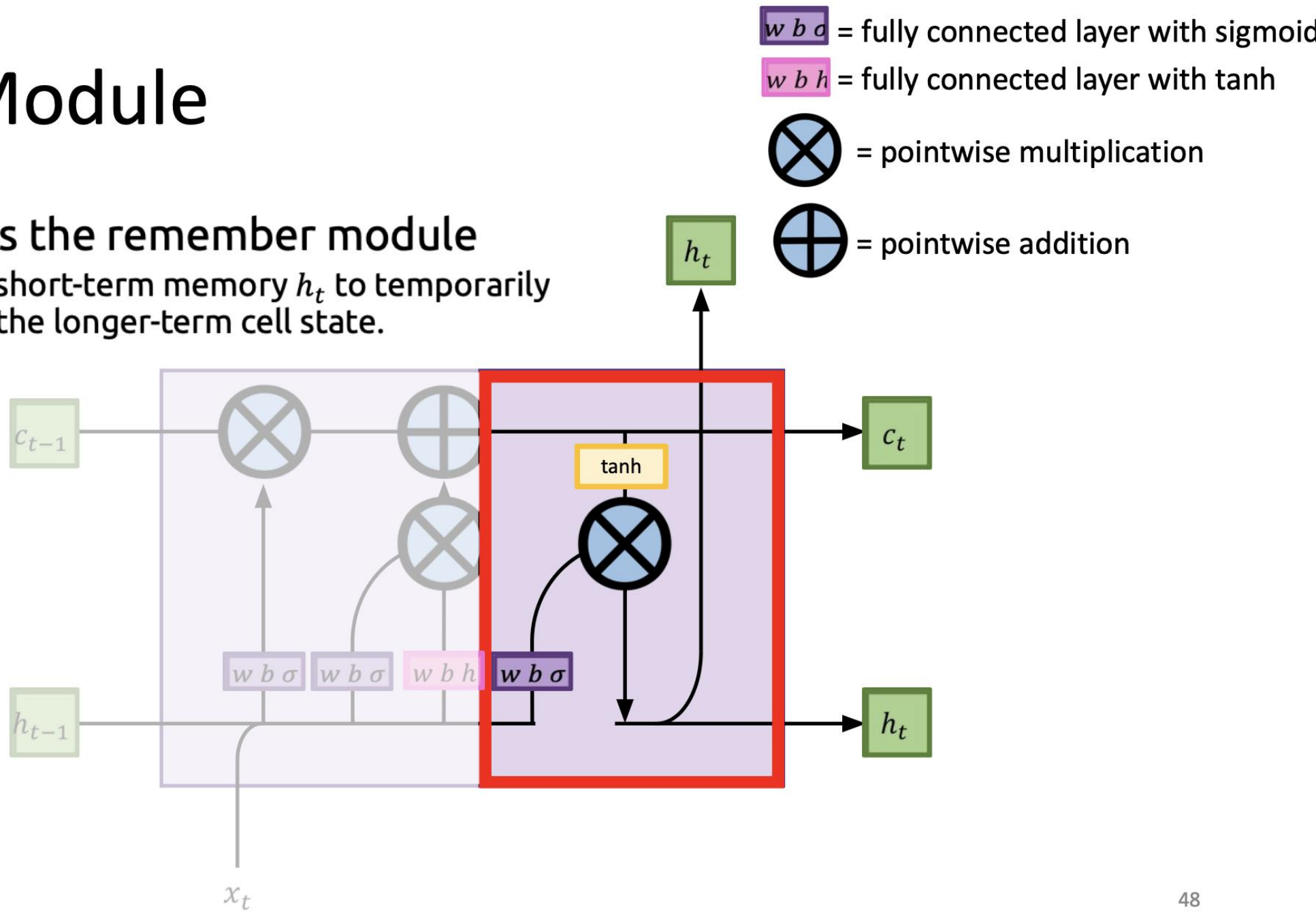


Output Module



Output Module

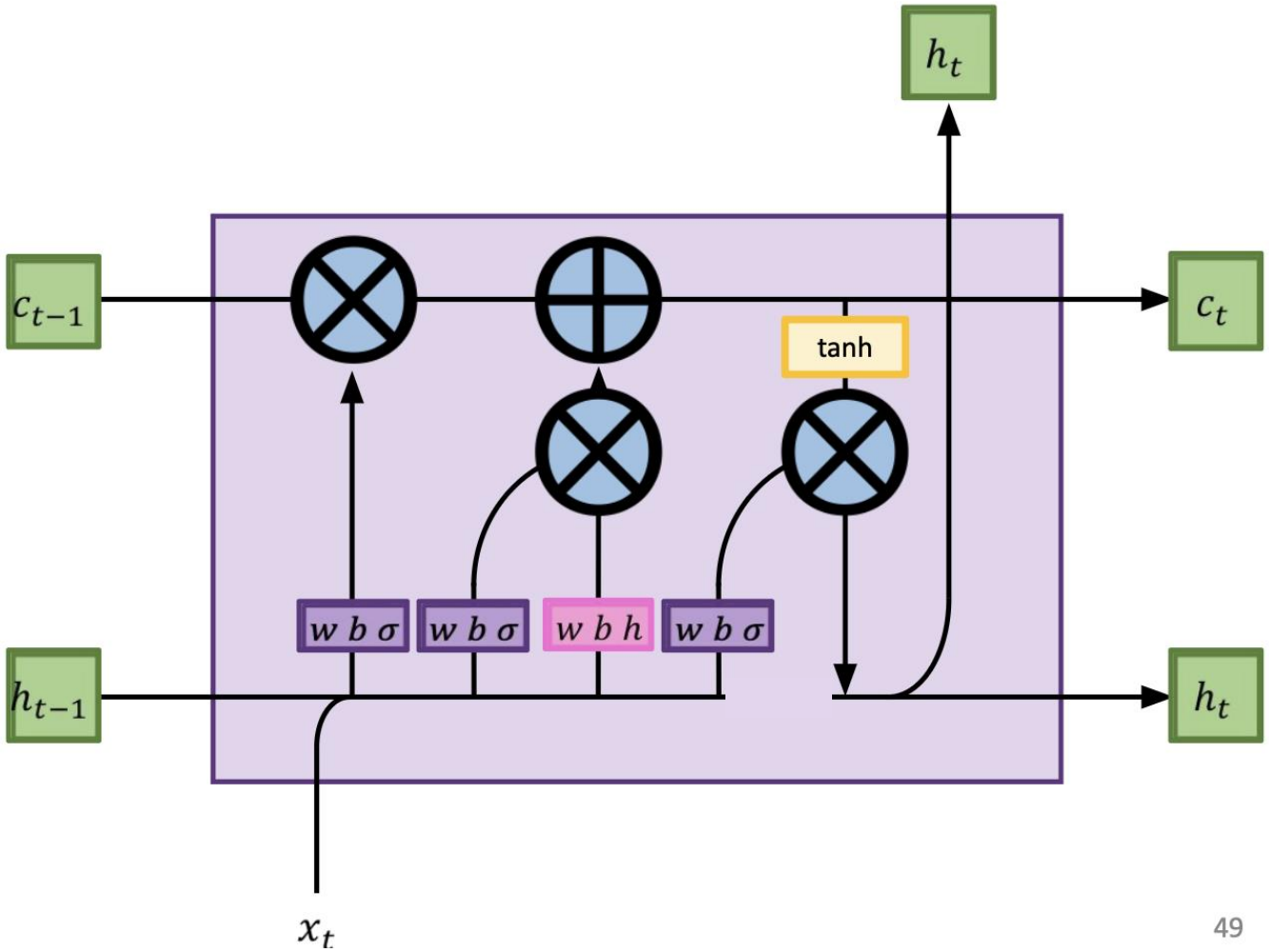
- Same structure as the remember module
  - Provides path for short-term memory h_t to temporarily acquire info from the longer-term cell state.



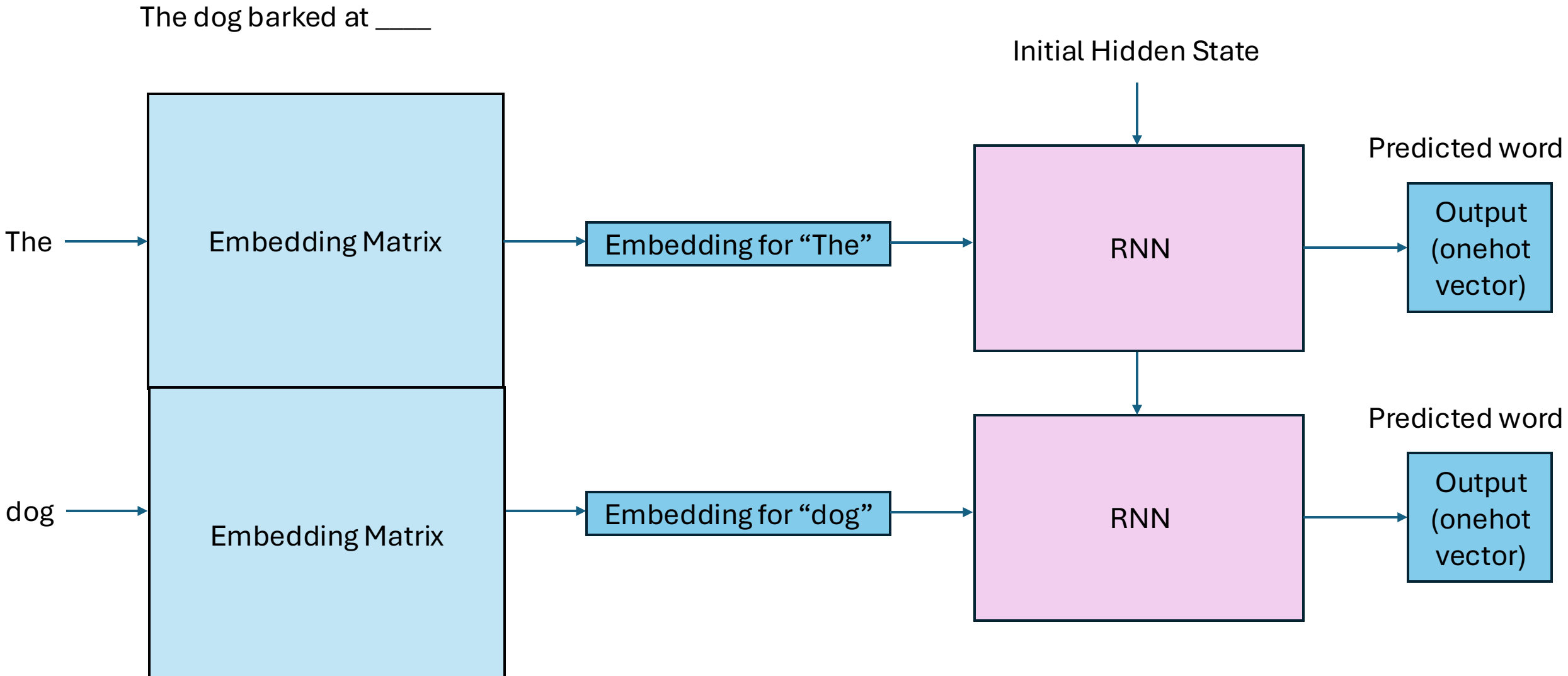
Any questions?

The Complete LSTM

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



Overview of RNN Sequence Prediction



When to compute loss

We have predictions and ground truths at every step, why not compute the loss after every word and backprop?

Should your model be penalized equally for incorrect predictions?

1) The ___

2) The dog barked at ___

Well… What task are you actually training it for?

Recap of LSTMs

Two “Outputs”: Cell and Hidden states

- Hidden state h_t is used for output (and short term memory)
- Cell State used for long term memory

Forget Module:

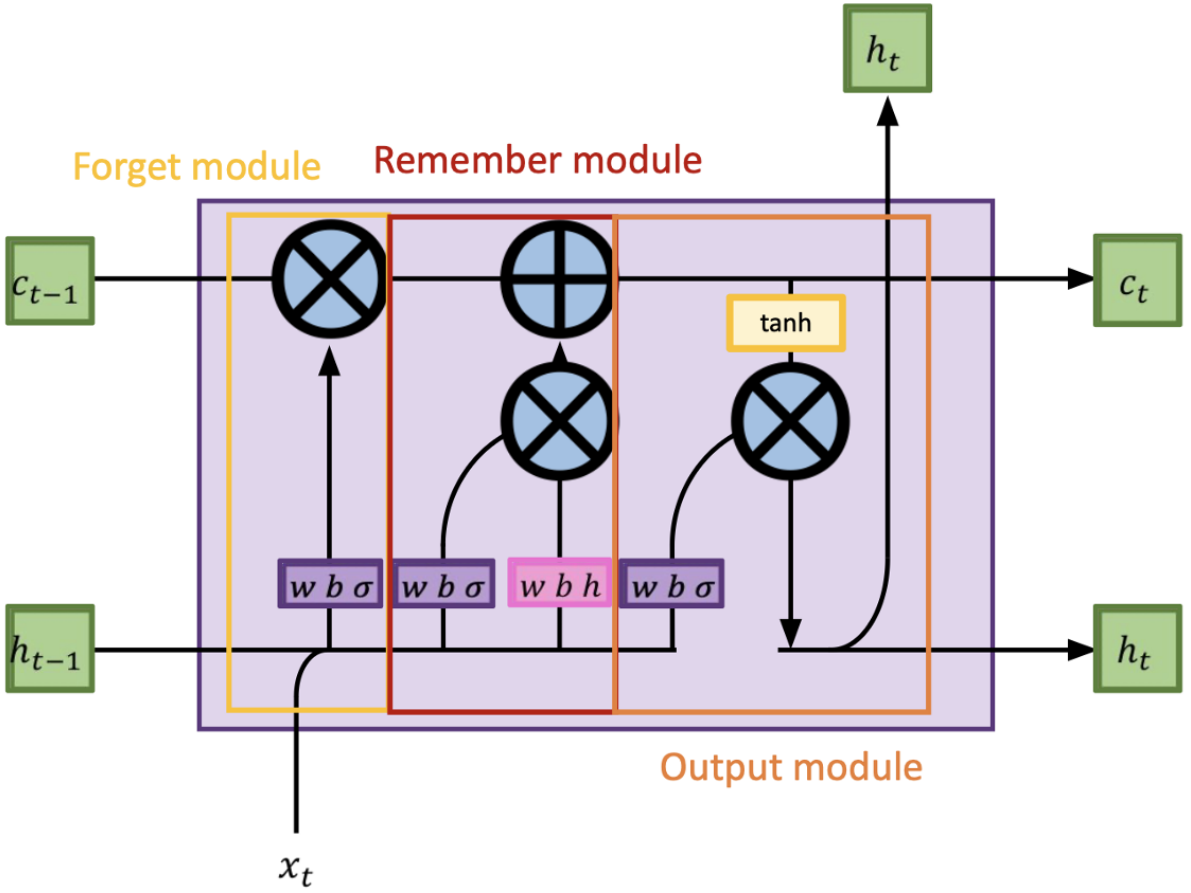
- Multiply cell state by numbers between 0 and 1 (0 forgets information, 1 keeps it)

Remember Module:

- Adds information from short term memory/input to long term memory

Output Module:

- Combines input (x_t), short term memory (h_t), and long term memory (c_t).
- Produces output
- Output is passed along as short term memory

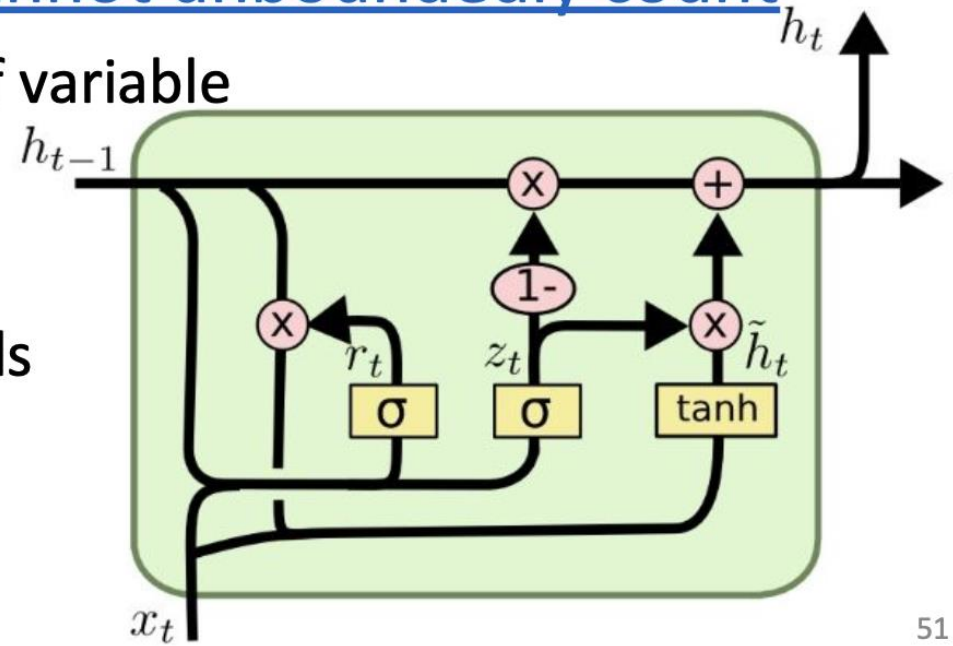


GRU

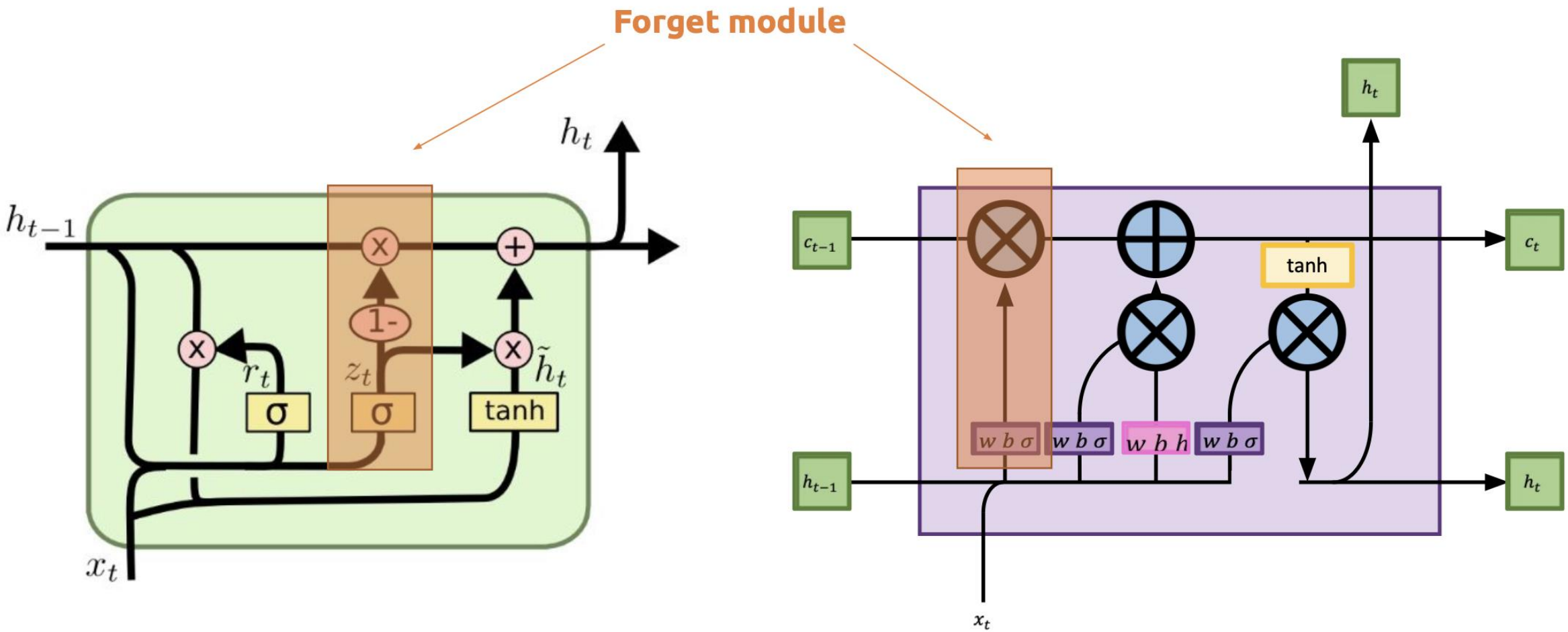
- ***G***ated ***R***ecurrent ***U***nit
- In practice, similar performance and may train faster
  - Removes cell state, computationally more efficient and less complex
- In theory, weaker than LSTMs since it cannot unboundedly count
  - Counting: track increment or decrement of variable
  - e.g. Validate brackets in code

    [… (… { … } …) …]

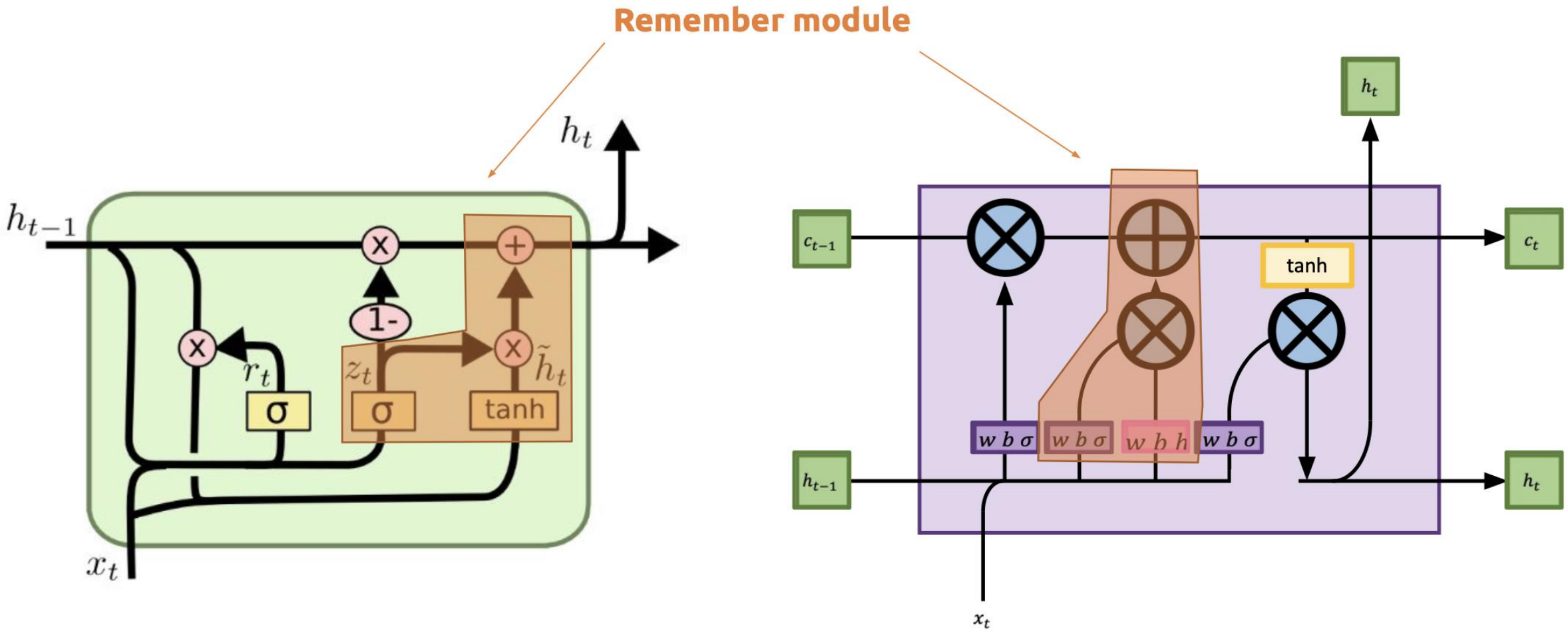
    Requires counting brackets & nesting levels



GRU vs LSTM


Forget module
h_t
h_{t-1}
x_t
r_t
z_t
$\tilde{h}_t$
σ
σ
tanh
1-
c_{t-1}
c_t
h_{t-1}
h_t
h_t
tanh
w b σ
w b σ
w b h
w b σ
x_t

GRU vs LSTM


Remember module
h_t
h_{t-1}
x_t
r_t
z_t
$\tilde{h}_t$
σ
σ
tanh
1-
c_{t-1}
c_t
h_t
h_{t-1}
x_t
tanh
w b σ
w b σ
w b h
w b σ

GRU vs LSTM

No direct analogue in LSTM
"Select the part of the memory that is relevant for the current prediction step"