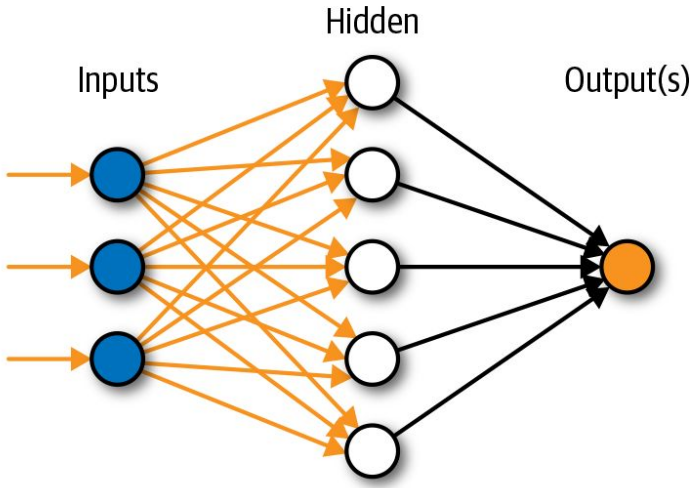
Recurrent Neural Networks (RNN)

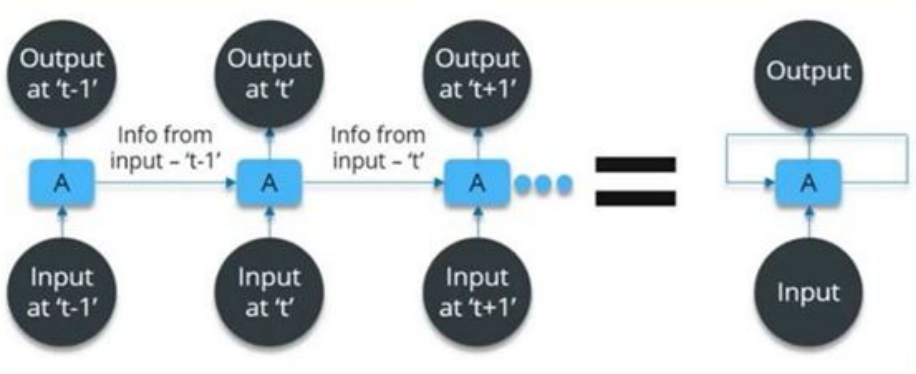
Neural Networks

- Output depends on Current Input

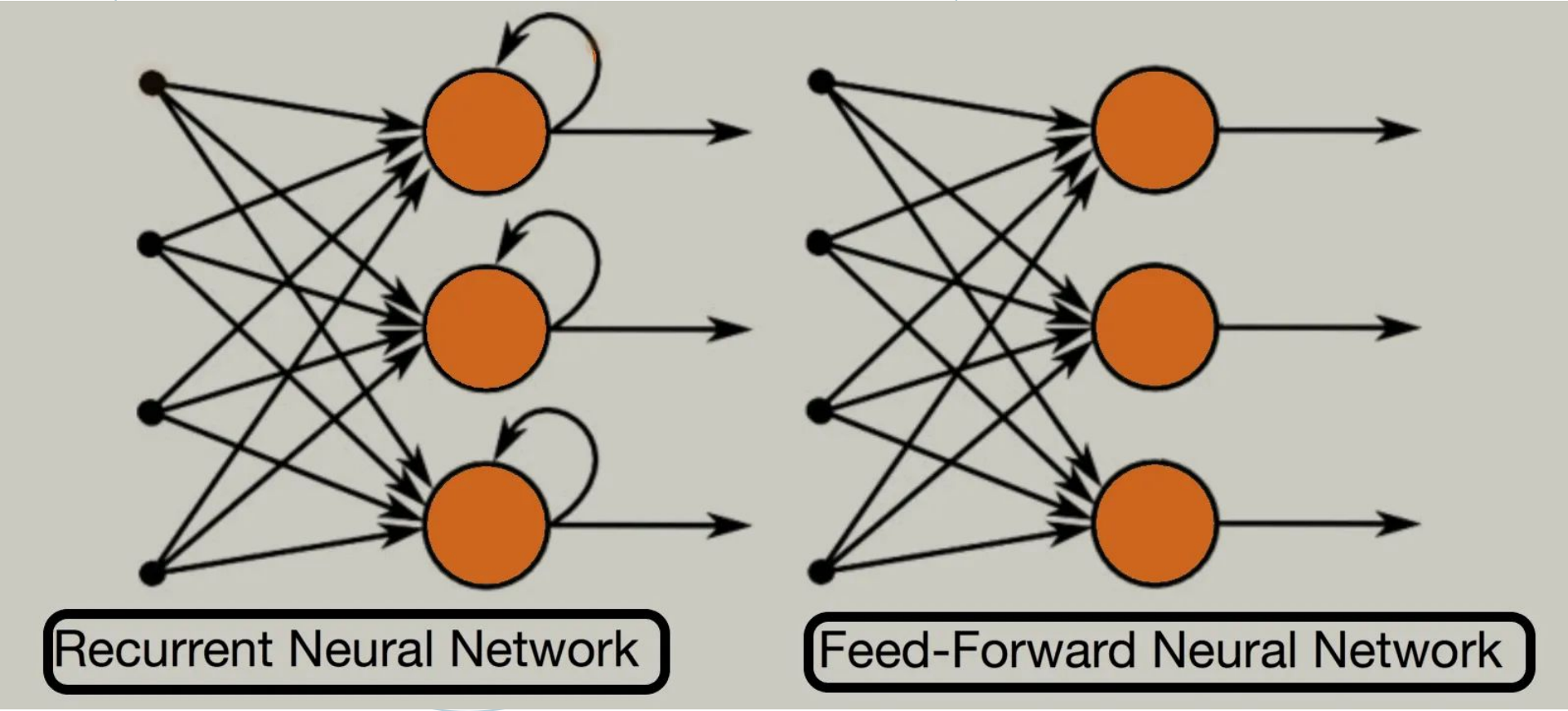


- No Cycles or loop in Network
- No Memory about the past

Recurrent Neural Networks



- Can handle sequential data
- Considers the current input and also the previously received inputs
- Can memorize inputs due to its internal memory

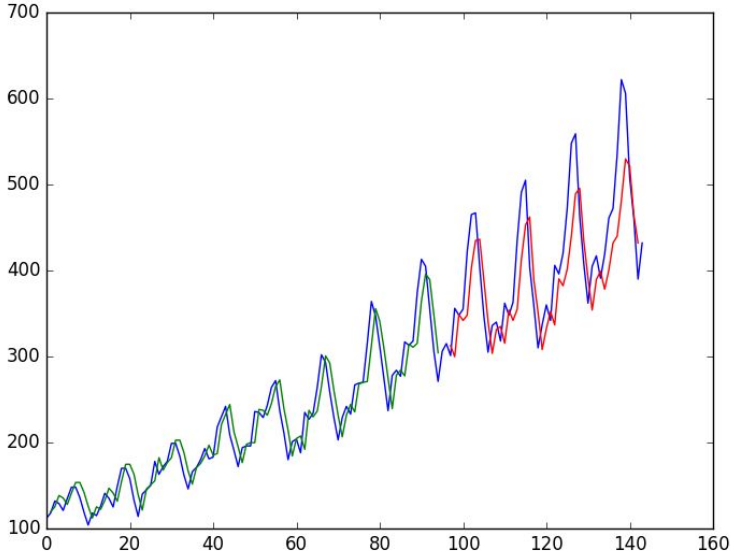

Recurrent Neural Network
Feed-Forward Neural Network

WHY RNN?

Application of RNN

1

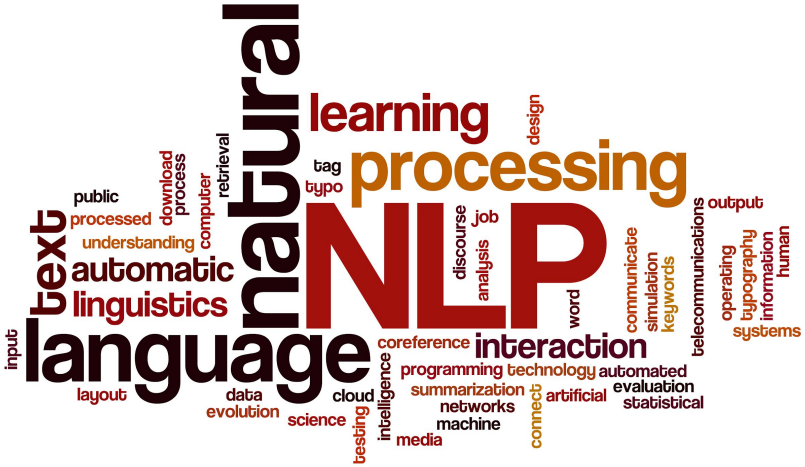
Time Series

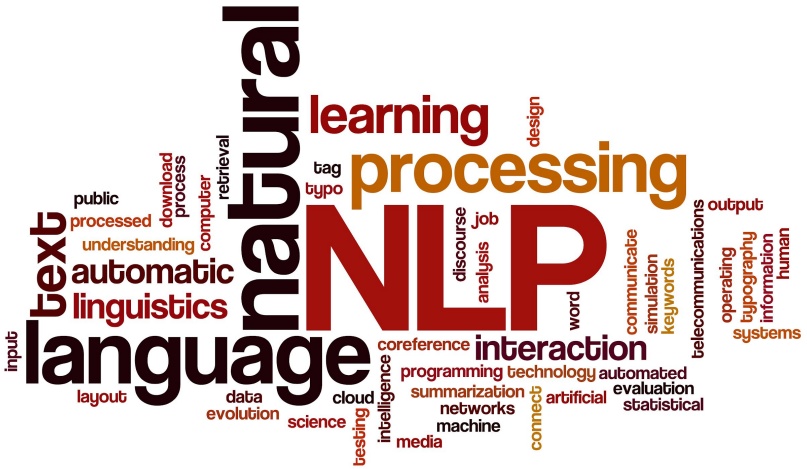


2

NLP

Text Classification, Sentiment Analysis,
Document Summary, Answer prediction





Machine Translation

Translate the input into different language

4

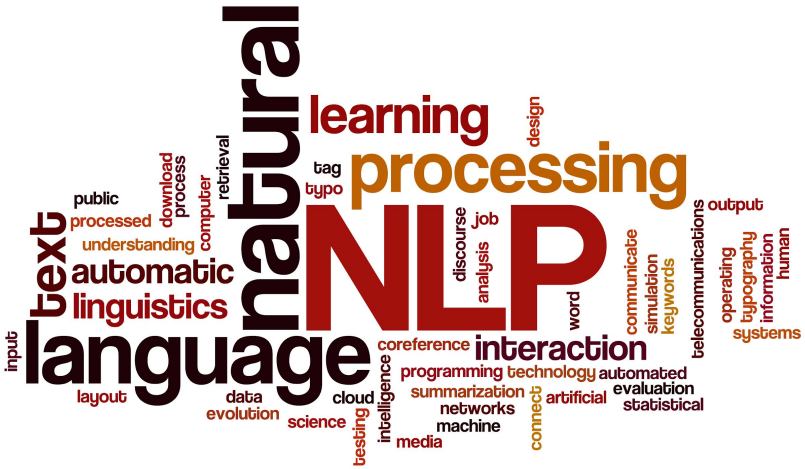
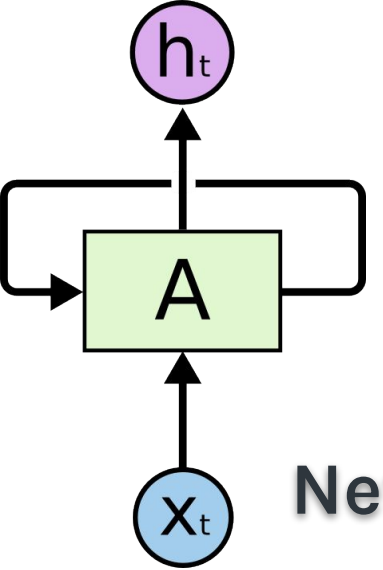


Image Captioning

Caption the image by analysing the activities

5

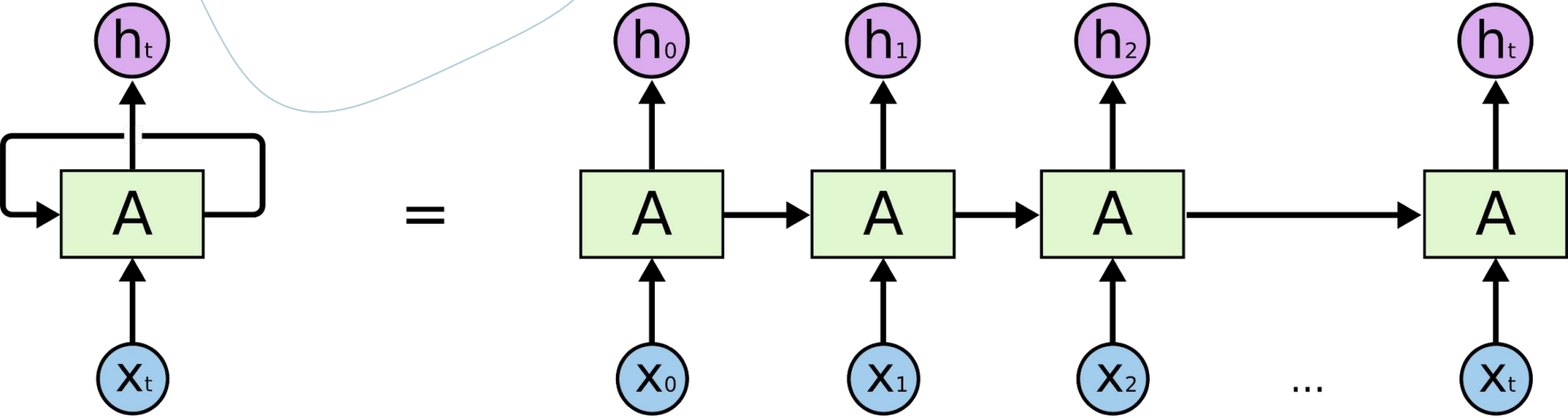
Speech Recognition



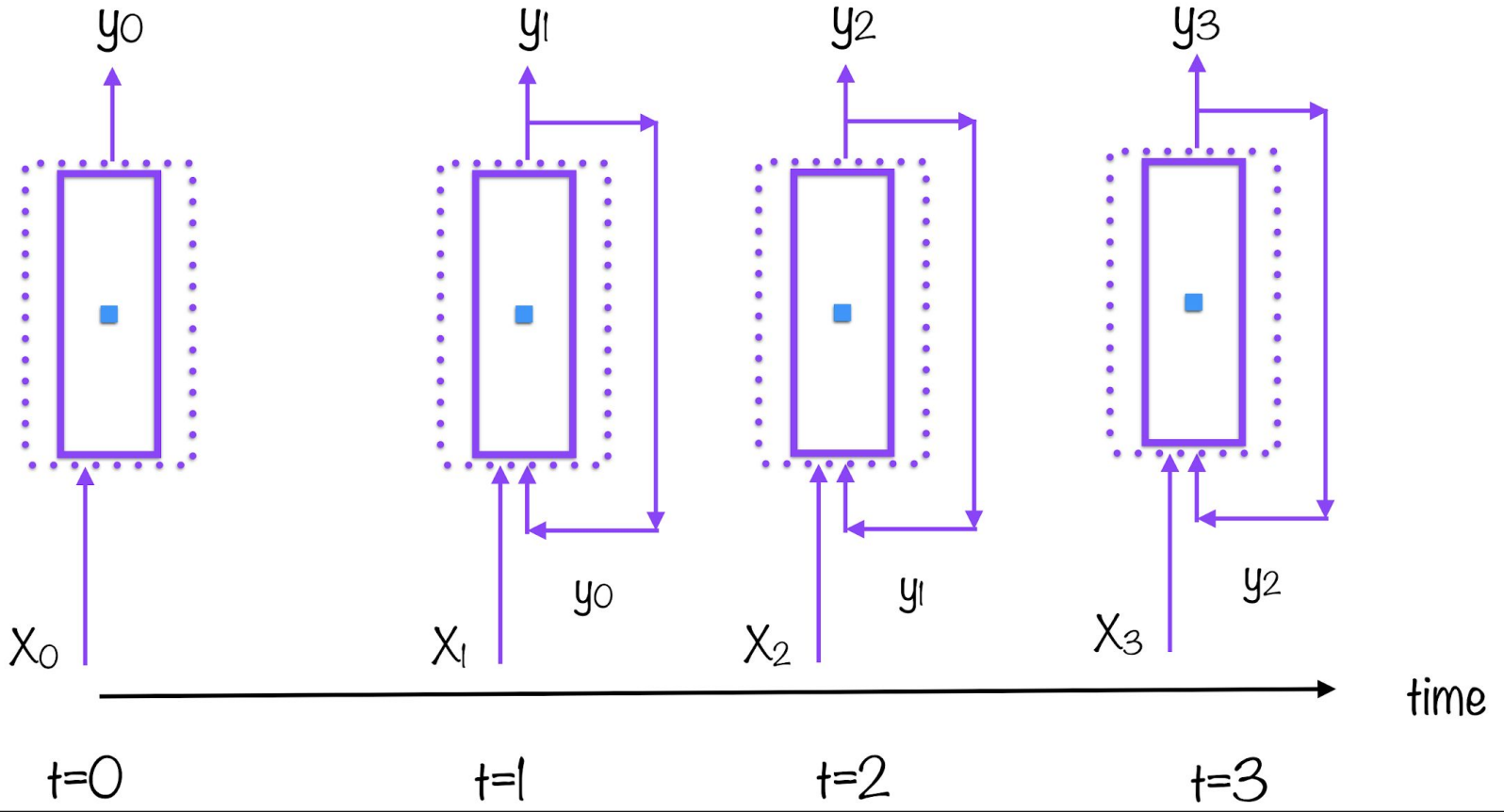
“

Networks with loops in them, allowing information to persist

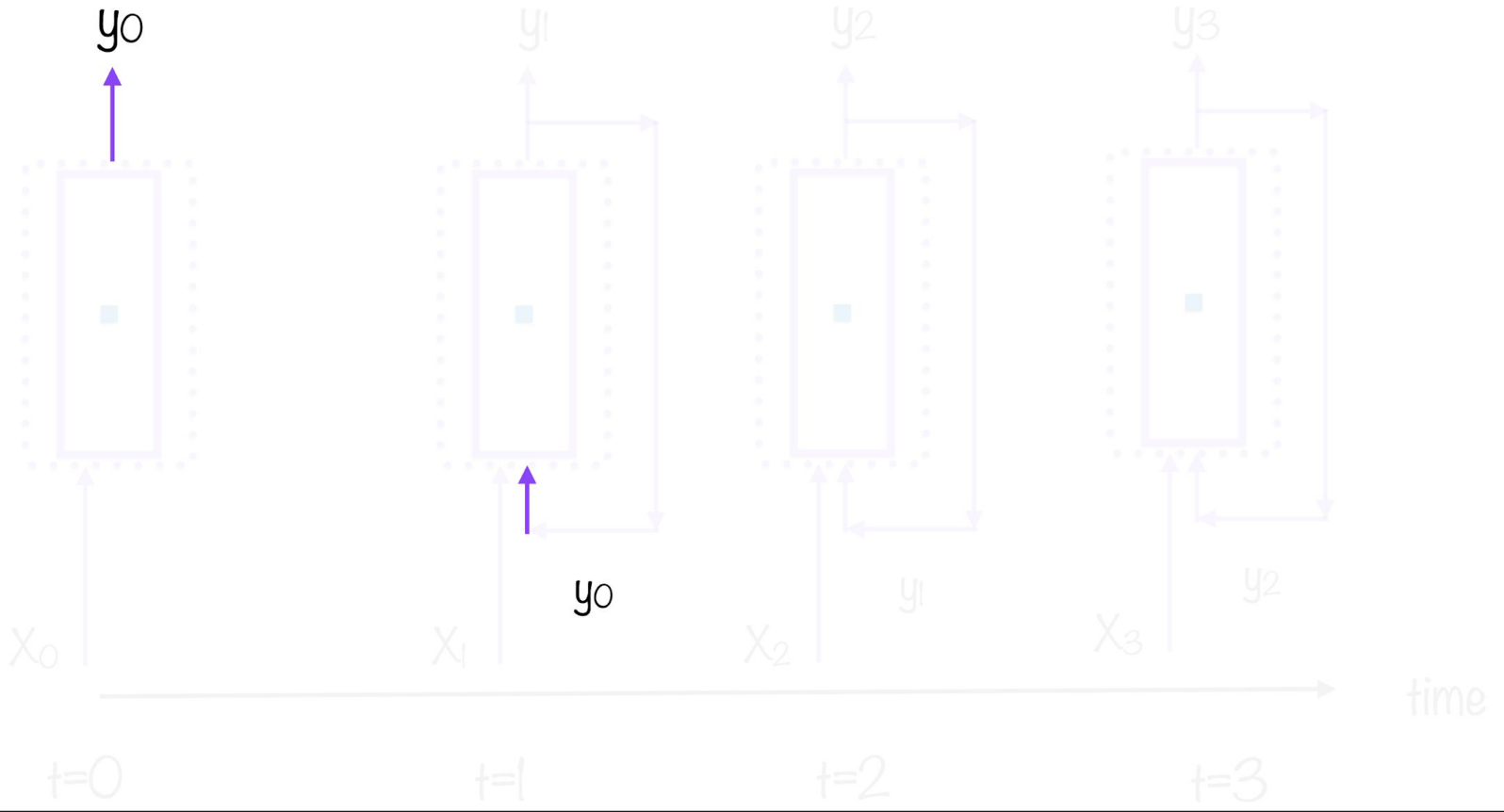
An unrolled recurrent neural network



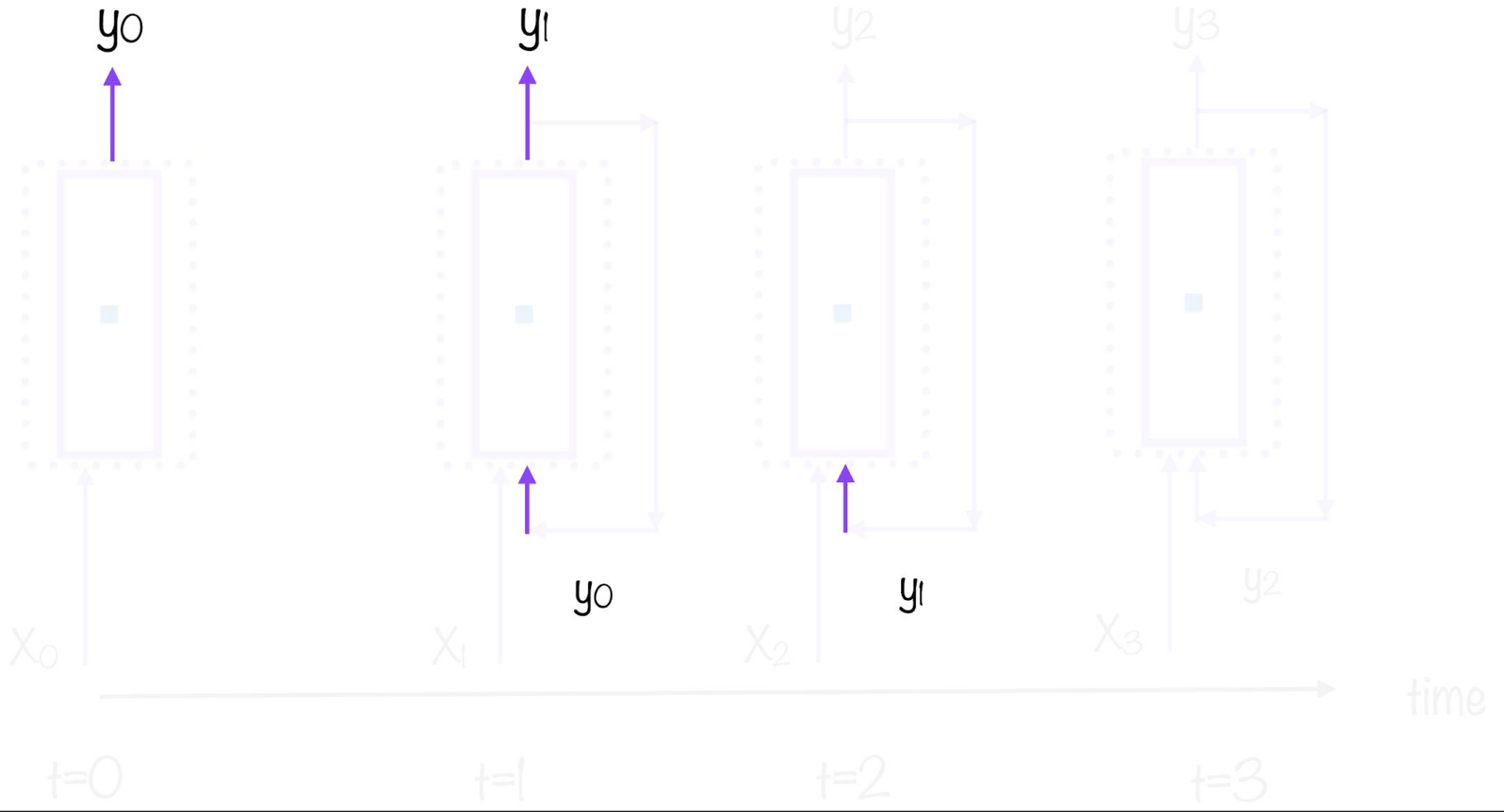
Unrolling Through Time



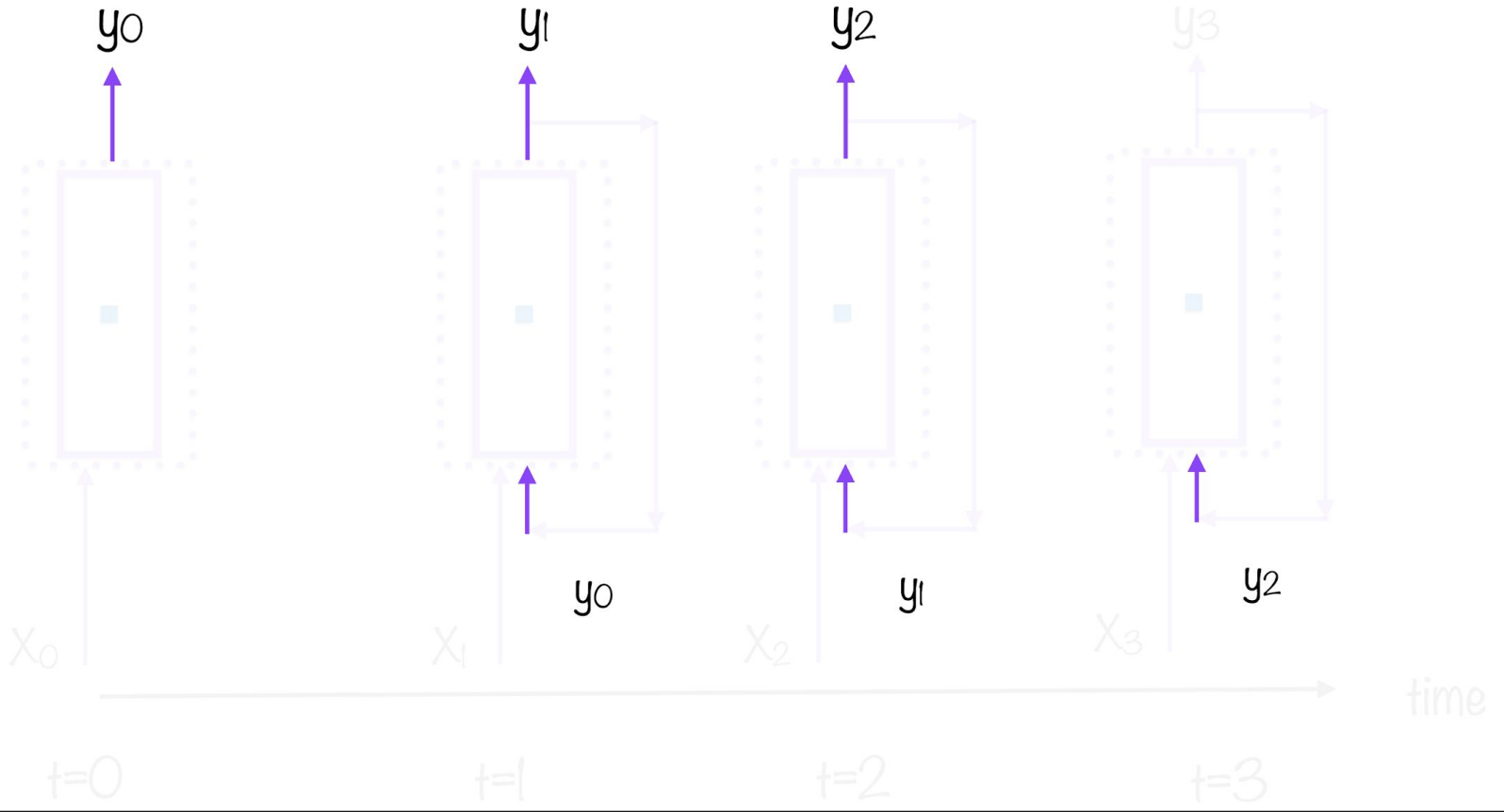
Unrolling Through Time



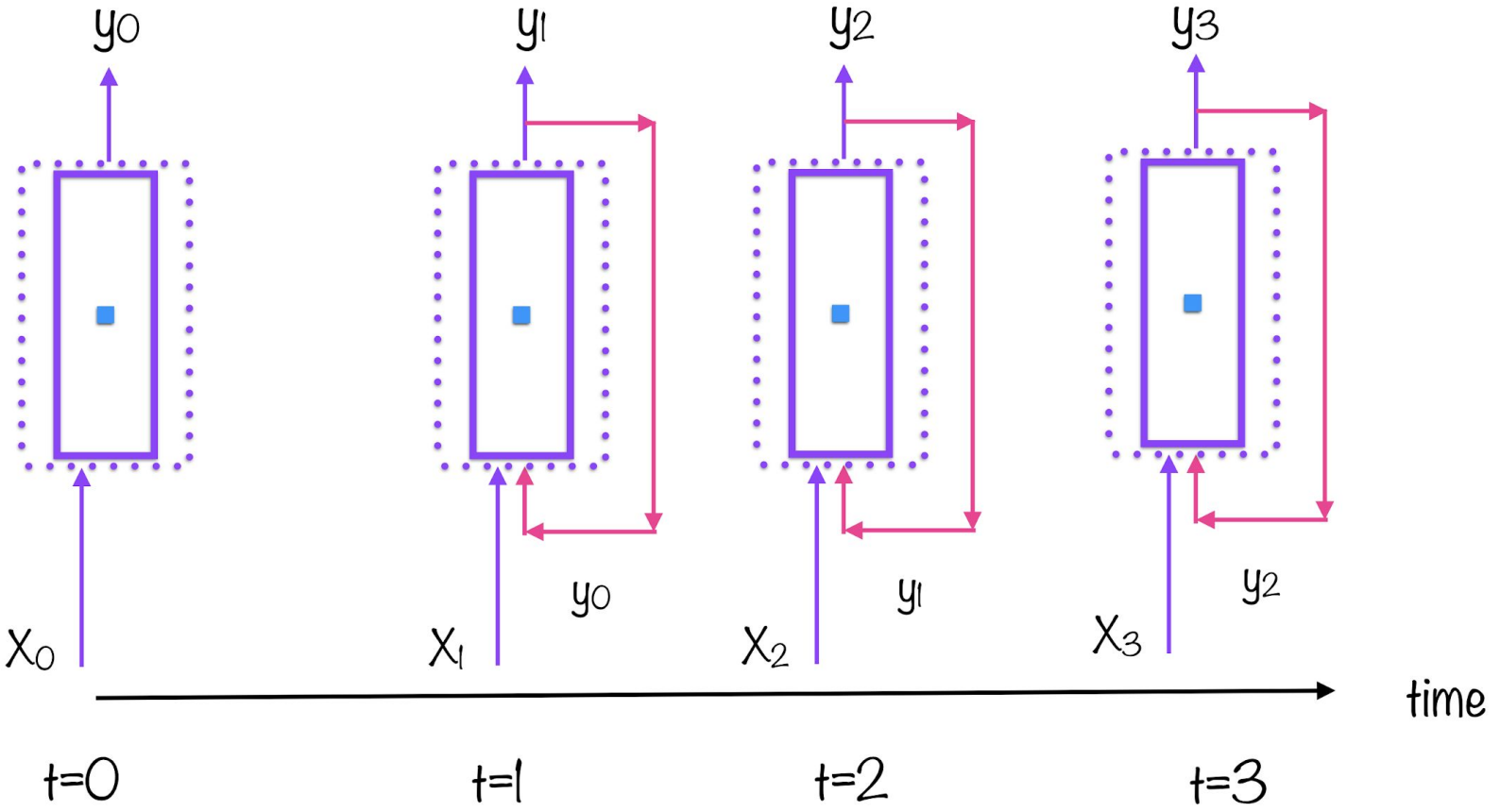
Unrolling Through Time



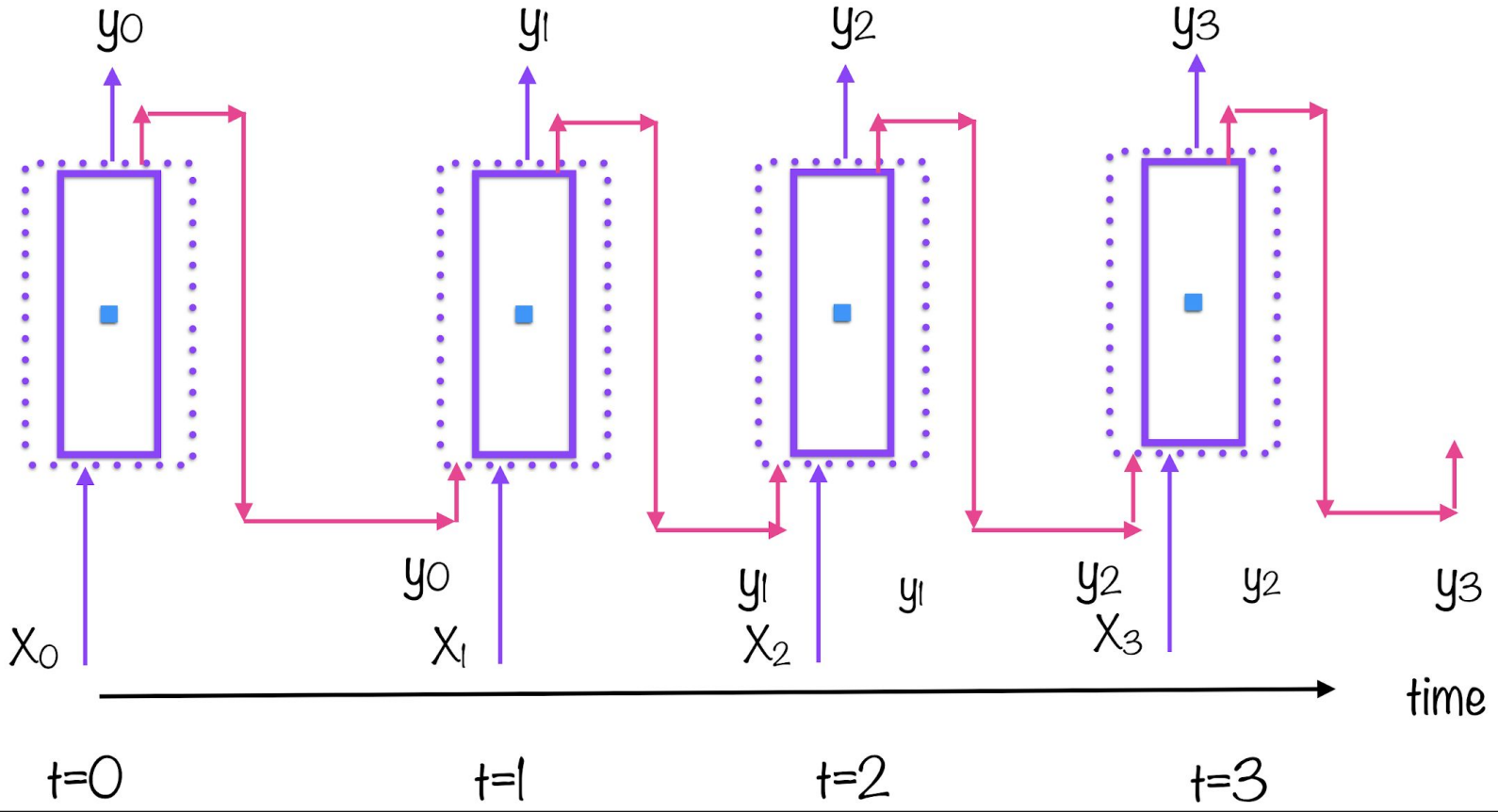
Unrolling Through Time



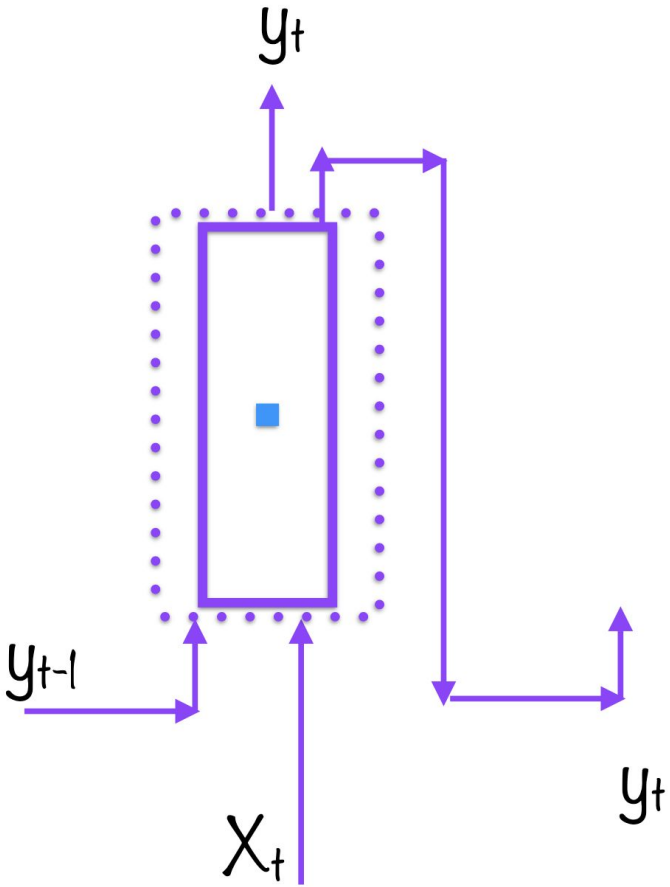
Unrolling Through Time



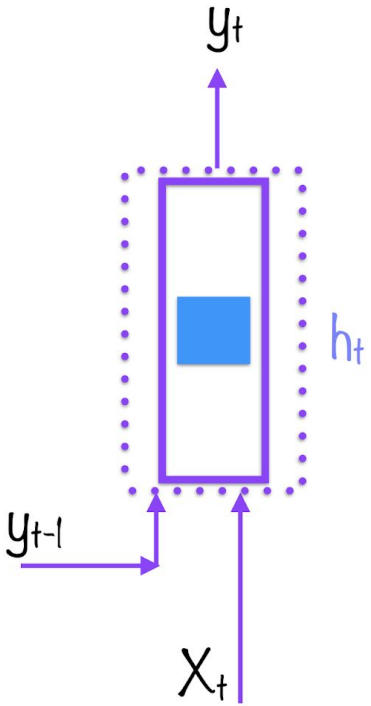
Unrolling Through Time



Recurrent Neuron



Memory and State



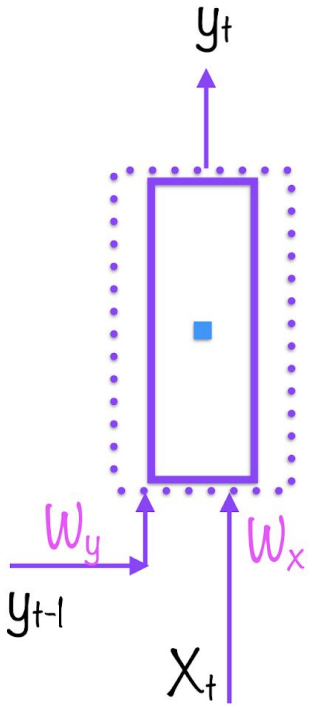
Recurrent neurons remember the past

They possess 'memory'

The stored state could be more complex than simply y_{t-1}

The internal state is represented by h_t

Recurrent Neuron



Now, each neuron has two weight vectors

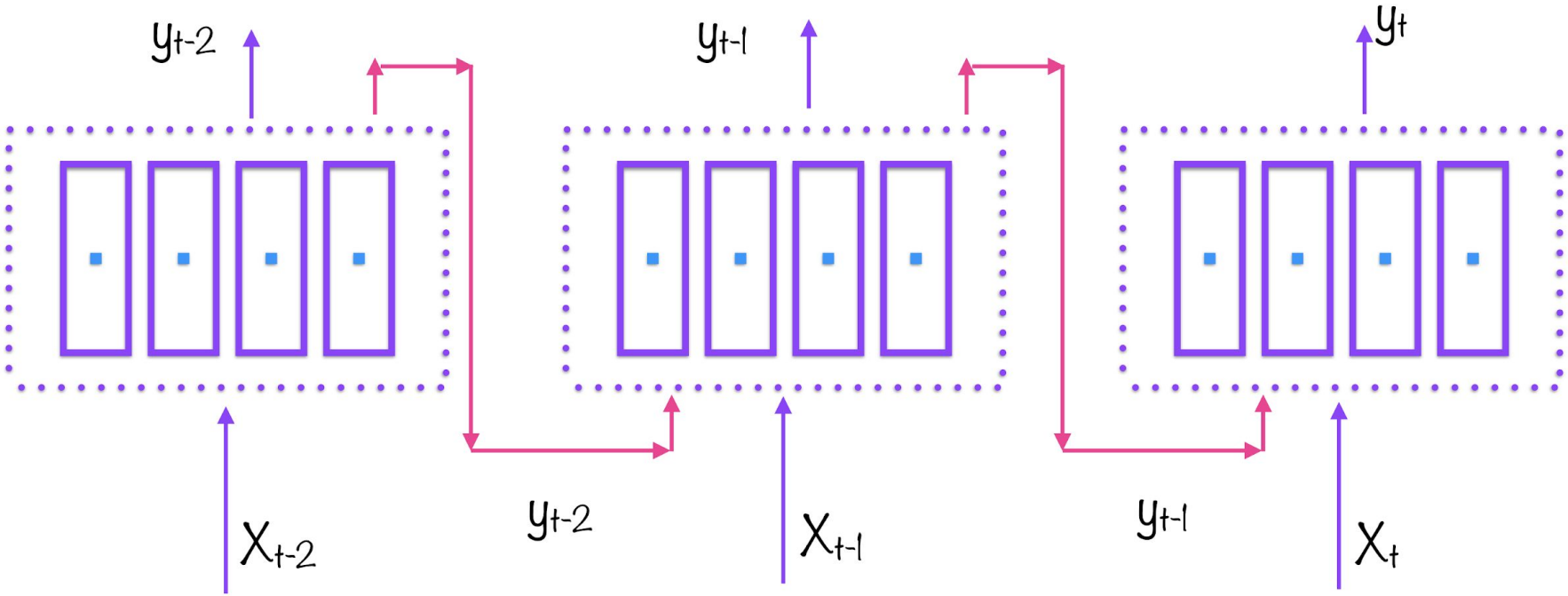
W_x, W_y

Output of neutron as a whole is given as

$$y_t = \Phi(X_t W_x + y_{t-1} W_y + b)$$

(Φ is the activation function)

Layer of Recurrent Neurons



The cell unrolled through time form the layers of the neural network

Back Propagation

Allows the weights and biases of the neurons to converge to their values

Recurrent neural networks may be unrolled very far back in time

They're prone to the vanishing and exploding gradients issue

RNN fails in real time

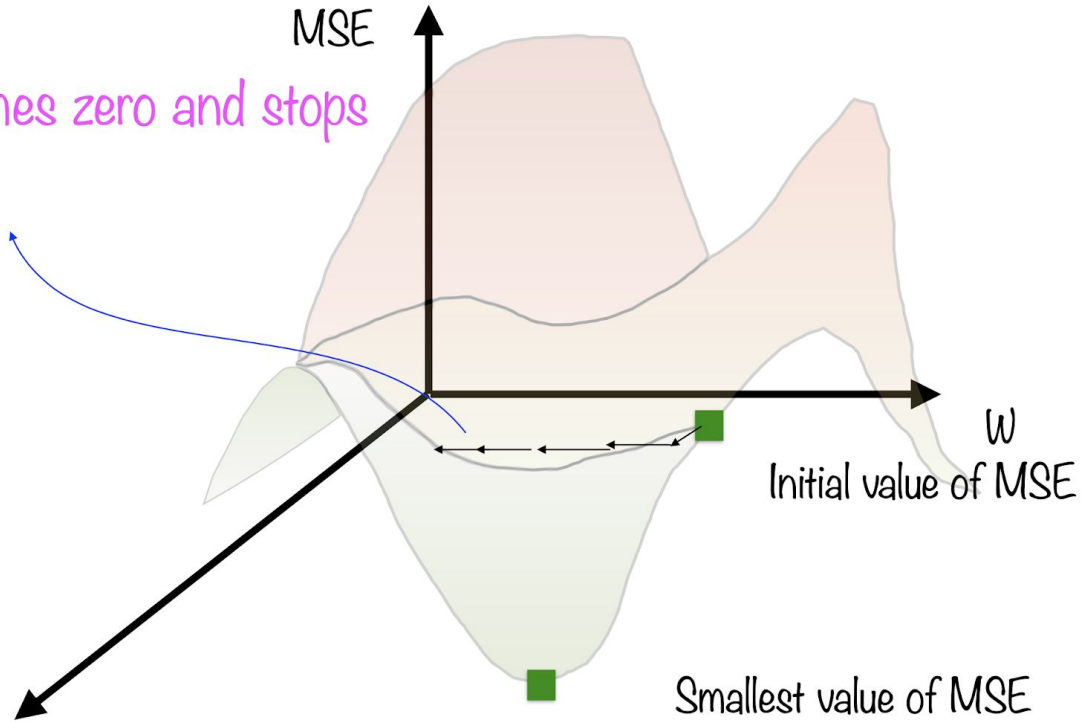
- Back Propagation in iterative process

Fails either if

- Gradients change at all
- Gradients changes too fast

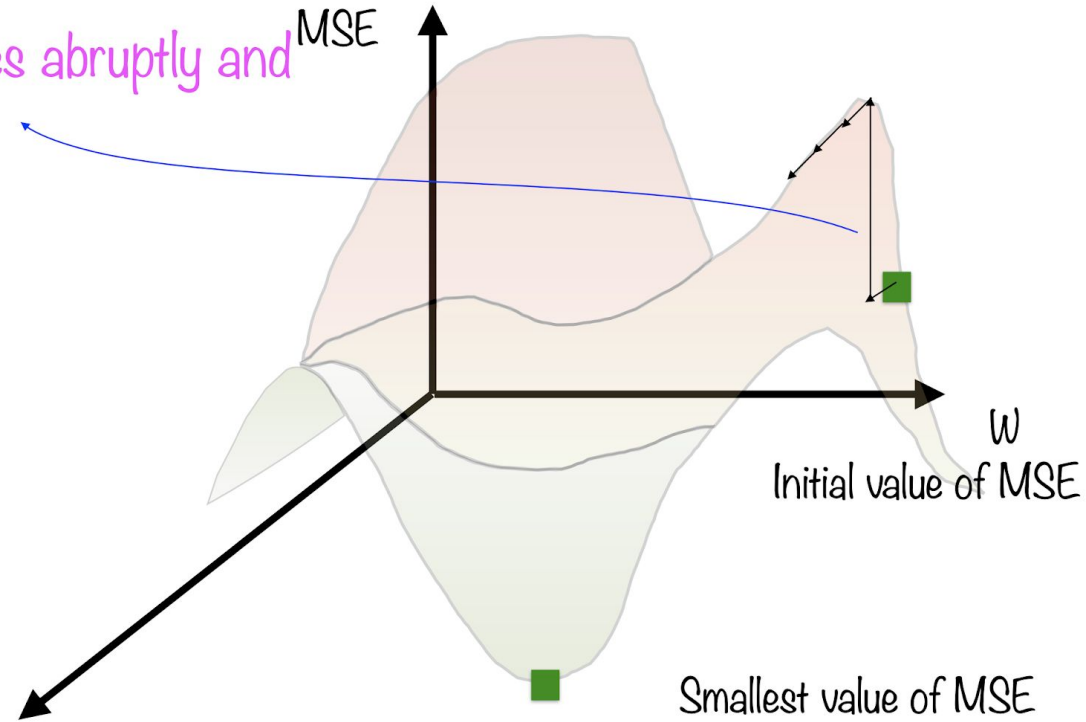
"Vanishing Gradient Problem"

Gradient becomes zero and stops changing

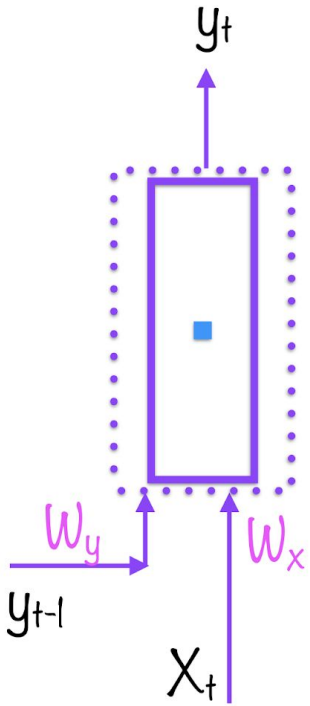


"Exploding Gradient Problem"

Gradient changes abruptly and "explodes"



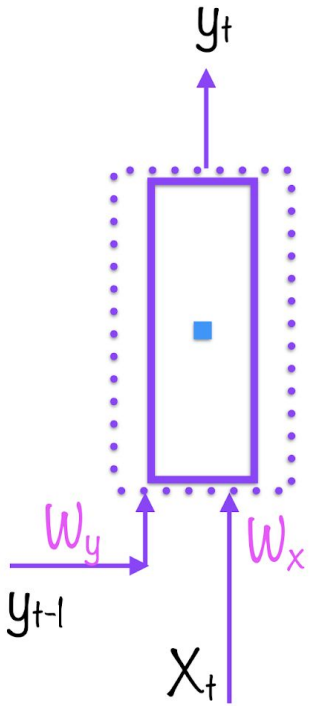
BPTT



Back Propagation Through Time is the tweaked version of Back Propagation for RNN training

Need as many layers as past time periods

BPTT



If output relies on distant past...

...Vanishing/exploding gradients very likely

One option - truncated BPTT

(Just discard distant past)

Not ideal - need to store long-term state

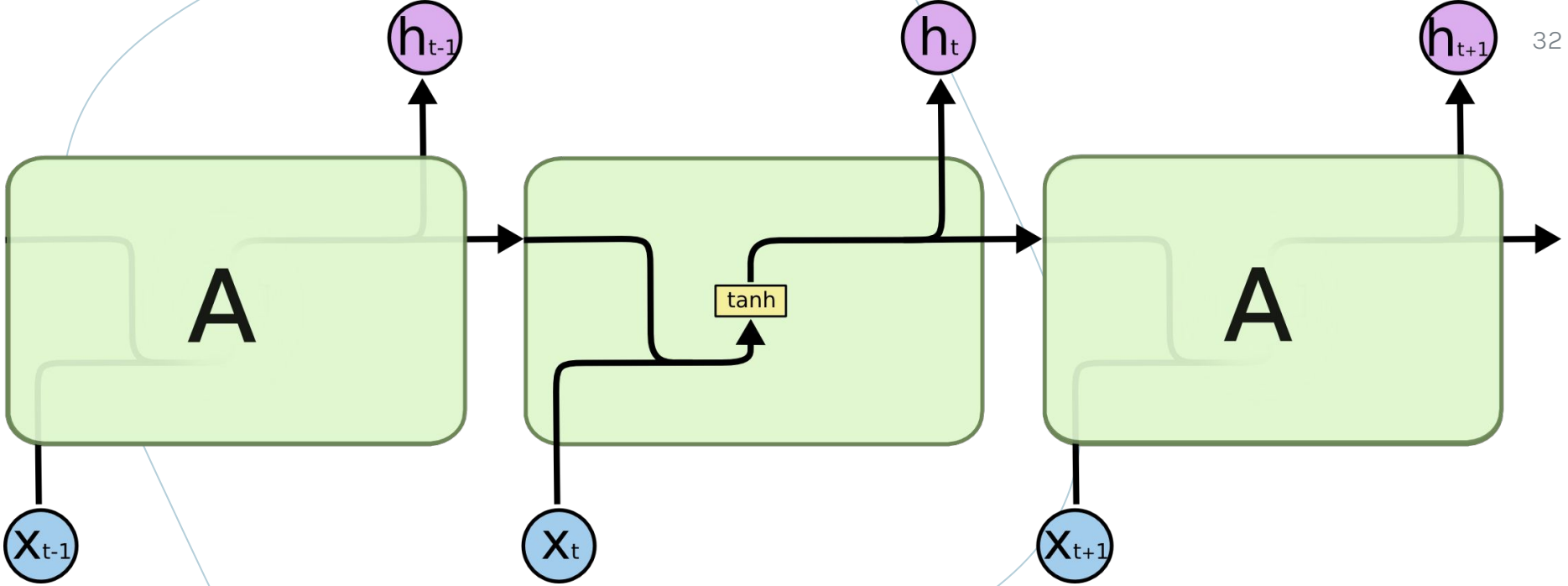
Solution of RNN Problem

LSTM

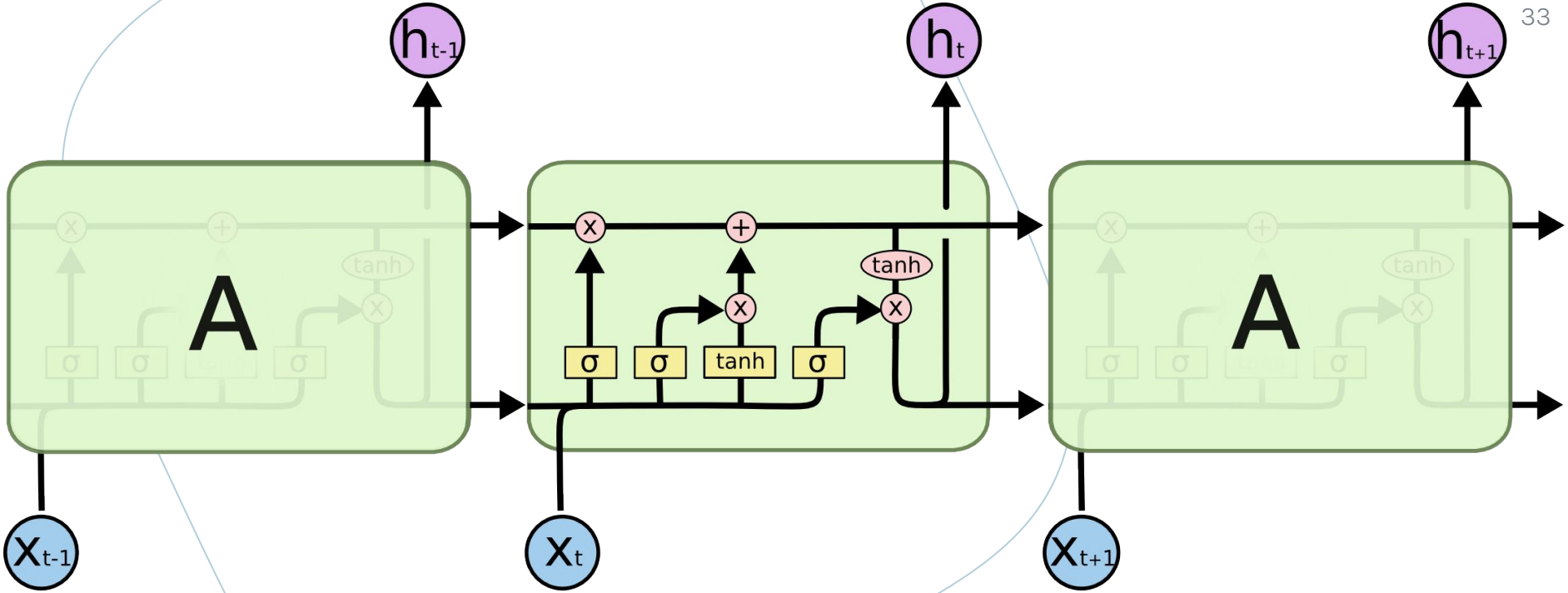
Long Short Term Memory networks - "**LSTM**"

- A Special kind of RNN, capable of learning long-term dependencies

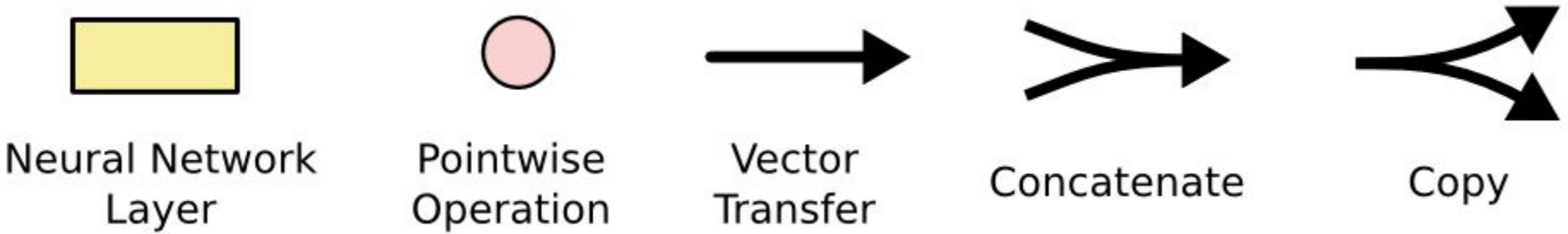
- Introduced by Hochreiter & Schmidhuber (1997)



In standard RNNs, this repeating module will have a very simple structure, such as a single **tanh** layer.



LSTMs also have this chain like structure, but the repeating module has a different structure. Instead of having a single neural network layer, there are **four, interacting in a very special way.**


Neural Network Layer
Pointwise Operation
Vector Transfer
Concatenate
Copy

Notation

LSTM Breakdown

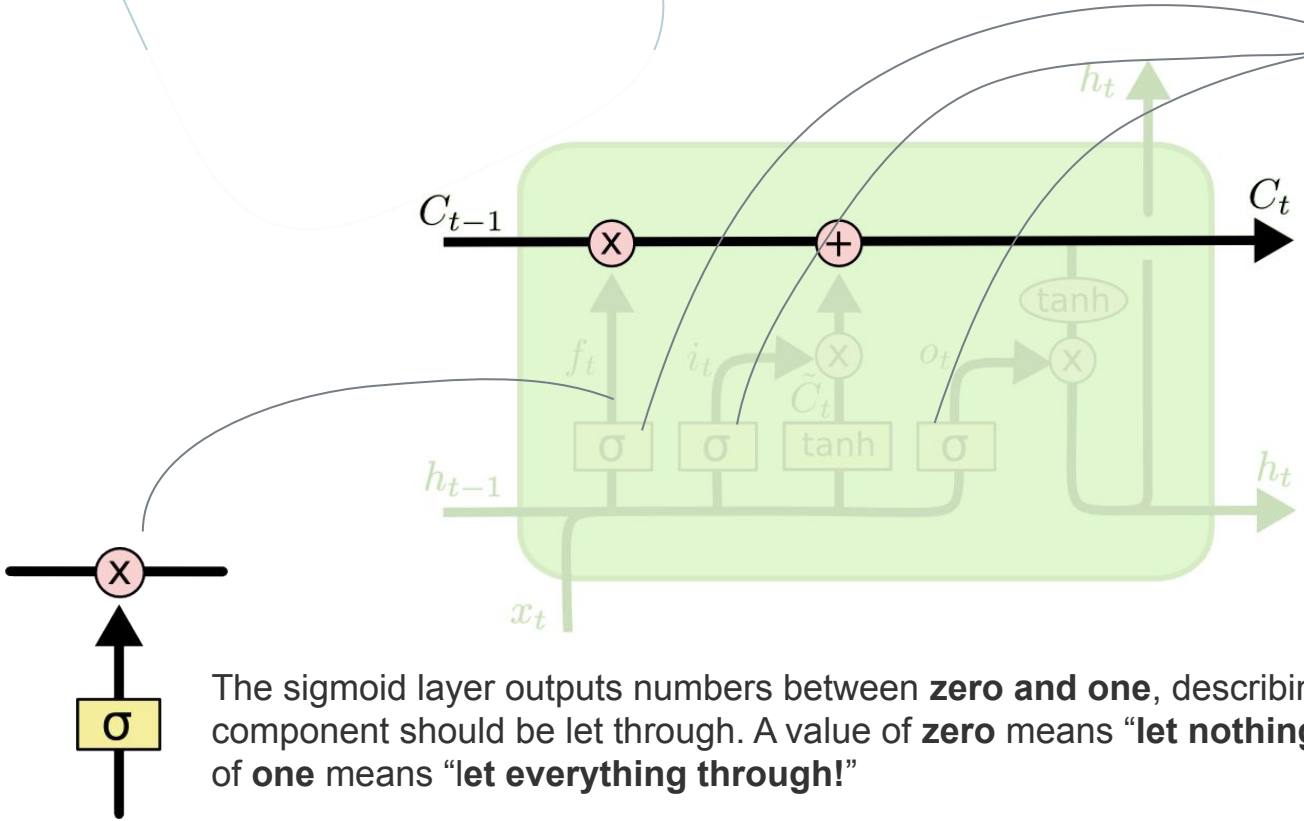
Input Gates

Hidden State

Forget Gates

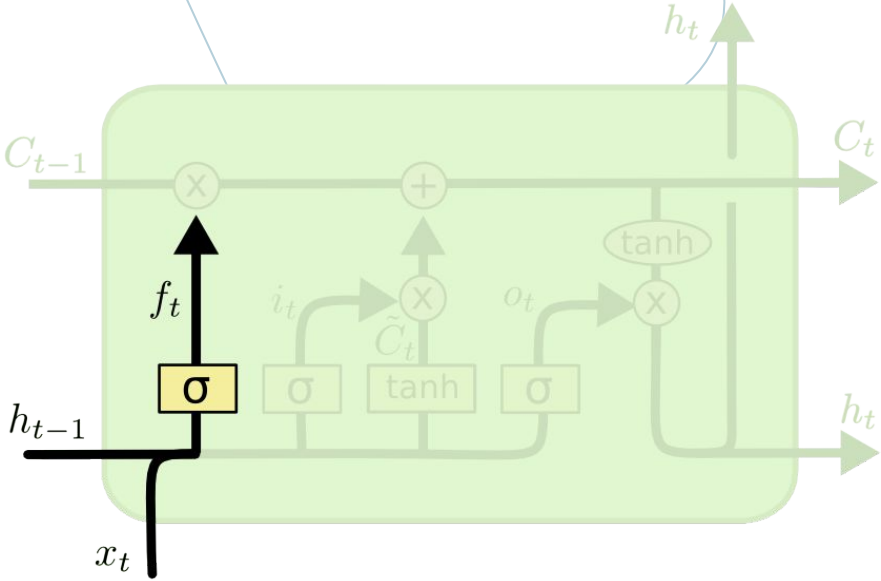
Output Gates

An LSTM has **three** of these gates, to protect and control the cell state.



The sigmoid layer outputs numbers between **zero and one**, describing how much of each component should be let through. A value of **zero** means "**let nothing through**," while a value of **one** means "l**et everything through!**"

Forget Gates

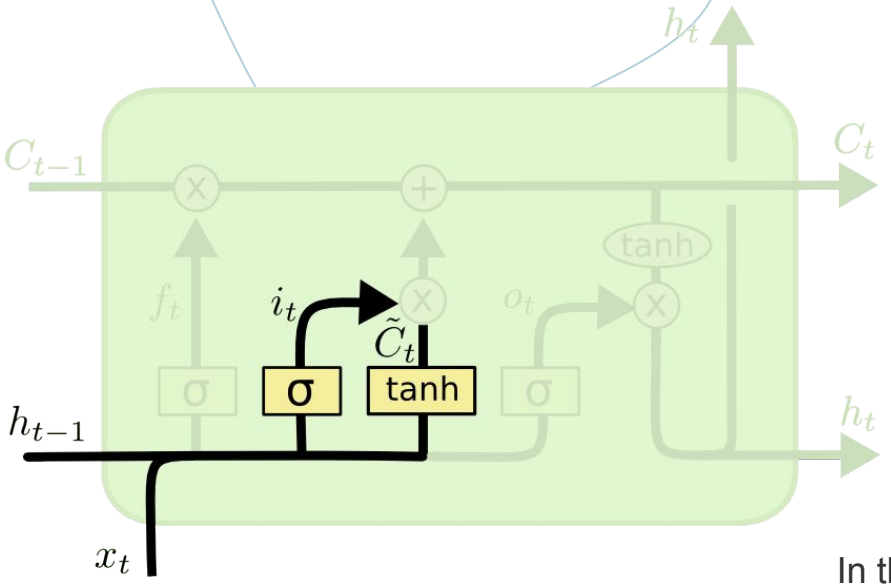


What is added to the hidden state

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Let's go to the example of a language model trying to predict the next word based on all the previous ones. In such a problem, the cell state might include the gender of the present subject, so that the correct pronouns can be used. When we see a new subject, we want to forget the gender of the old subject.

Input Gates



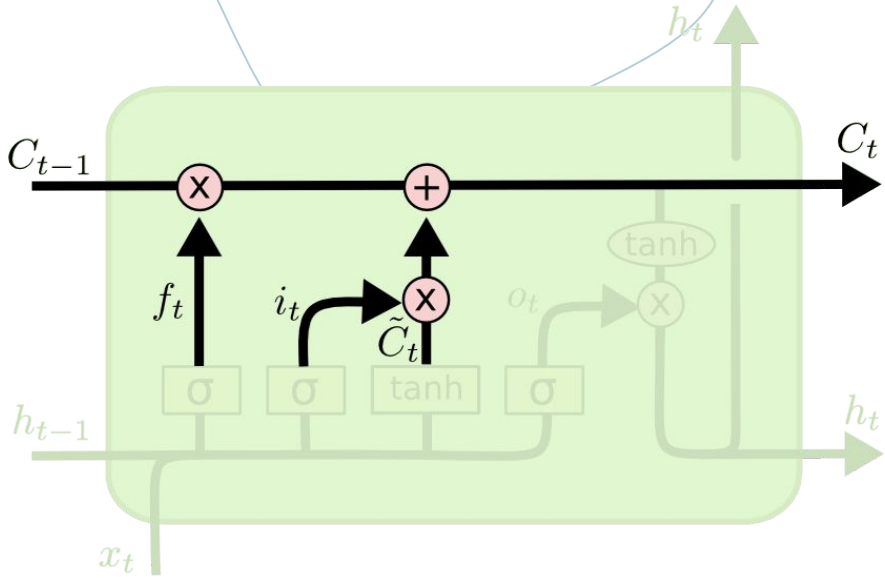
What is kept from previous states

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

In the example of our language model, we'd want to add the gender of the new subject to the cell state, to replace the old one we're forgetting.

Hidden State

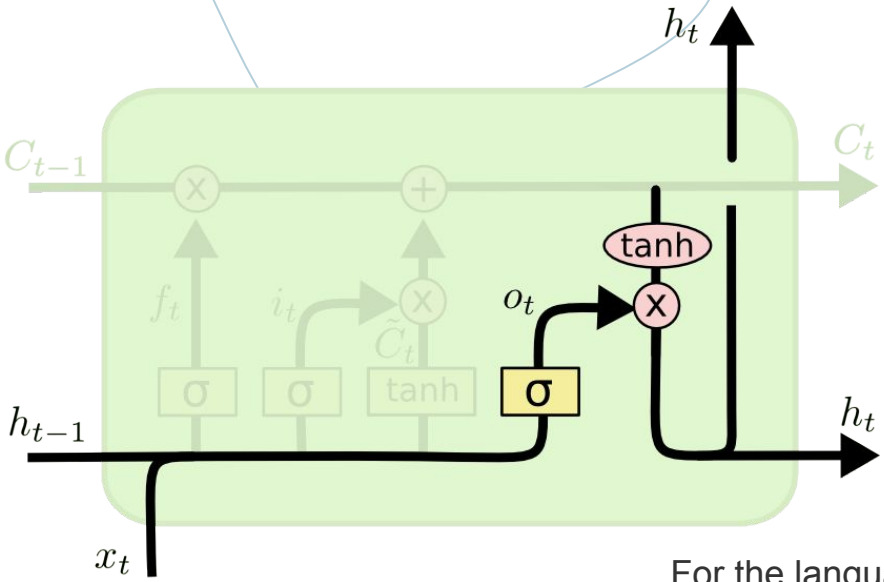


Carry previous information

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

In the case of the language model, this is where we'd **actually drop** the information about the old subject's gender and add the new information, as we decided in the previous steps.

Output Gates



What is reported as output

$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

For the language model example, since it just saw a subject, it might want to output information relevant to a verb, in case that's what is coming next. For example, it might output whether the subject is singular or plural, so that we know what form a verb should be conjugated into if that's what follows next.

```
def LSTMCELL(prev_ct, prev_ht, input):
    combine = prev_ht + input
    ft = forget_layer(combine)
    candidate = candidate_layer(combine)
    it = input_layer(combine)
    Ct = prev_ct * ft + candidate * it
    ot = output_layer(combine)
    ht = ot * tanh(Ct)
    return ht, Ct


ct = [0, 0, 0]
ht = [0, 0, 0]

for input in inputs:
    ct, ht = LSTMCELL(ct, ht, input)
```

1. First, the previous hidden state and the current input get concatenated. We'll call it *combine*.
2. *Combine* get's fed into the forget layer. This layer removes non-relevant data.
4. A candidate layer is created using *combine*. The candidate holds possible values to add to the cell state.
3. *Combine* also get's fed into the input layer. This layer decides what data from the candidate should be added to the new cell state.
5. After computing the forget layer, candidate layer, and the input layer, the cell state is calculated using those vectors and the previous cell state.
6. The output is then computed.
7. Pointwise multiplying the output and the new cell state gives us the new hidden state.