Overview of Machine Learning and Pattern Recognition

Dipartimento di Automatica e Informatica
Politecnico di Torino, Torino, ITALY

Outline

- Introduction
- Motivation
- RNN architectures
- RNN problems
- LSTM
- How LSTM solves the typical RNN problems
- Conclusions

Introduction

- RNN were introduced in the late 80's.
- Hochreiter discovers the 'vanishing gradients' problem in 1991.
- Long Short Term Memory published in 1997.
- LSTM a recurrent network to overcome these problems.

Motivation

- Feed forward networks accept a fixed-sized vector as input and produce a fixed-sized vector as output
- fixed amount of computational steps
- recurrent nets allow us to operate over *sequences* of vectors

Motivation

- Recurrent Neural Networks take the previous output or hidden states as inputs.
  The composite input at time t has some historical information about the happenings at time T < t

- RNNs are useful as their intermediate values (state) can store information about past inputs for a time that is not fixed a priori

Motivation

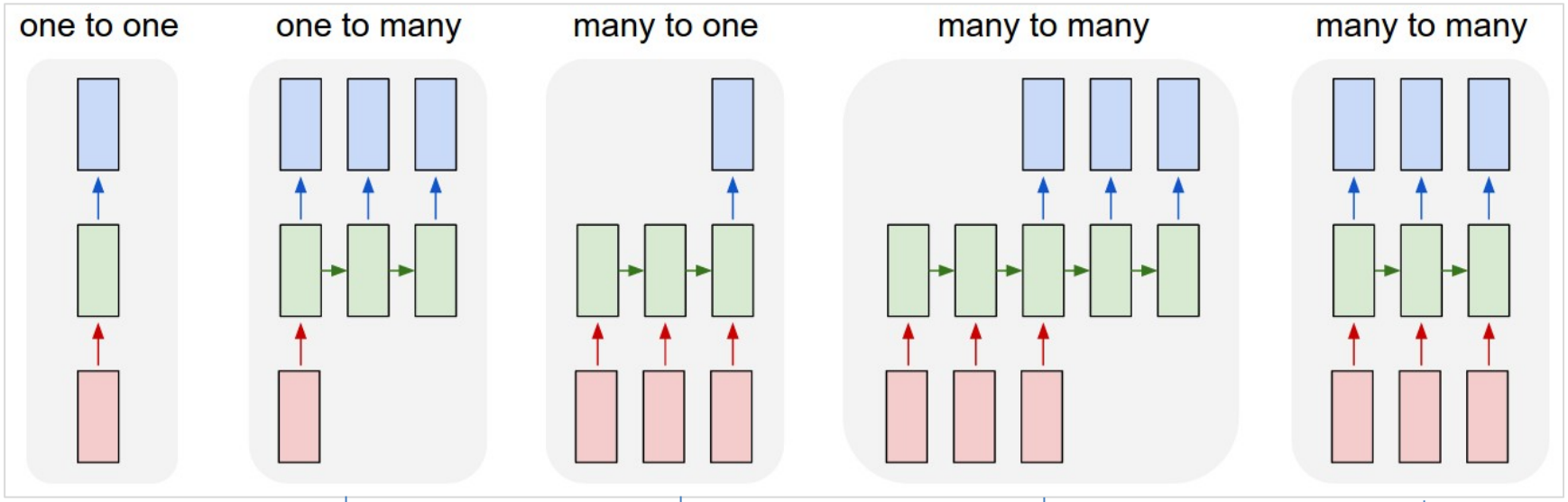


Image classification

Image captioning

Sentiment analysis

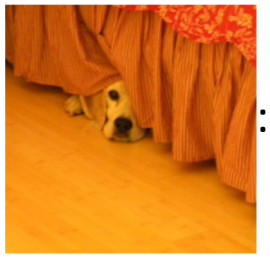
Machine translation

Synced sequence(video classification)

Image Captioning

- Given an image, produce a sentence describing its contents

- Inputs: Image feature (from a CNN)
- Outputs: Multiple words (let's consider one sentence)



: The dog is hiding

Image Captioning

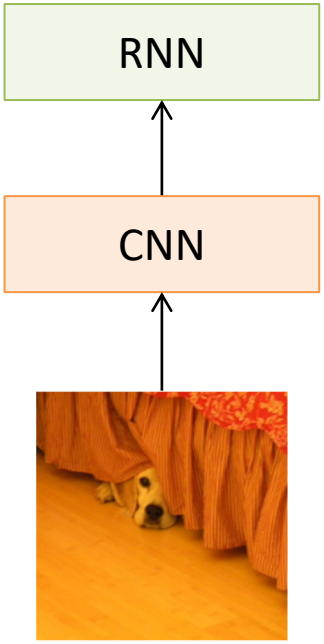


Image Captioning

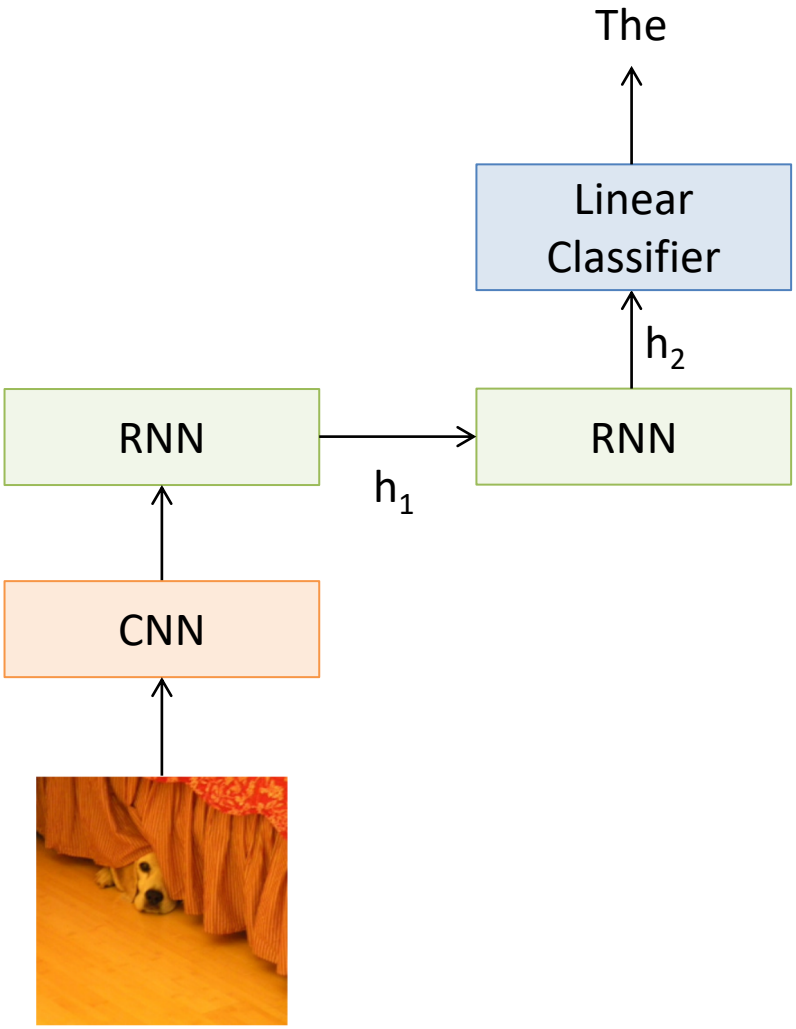
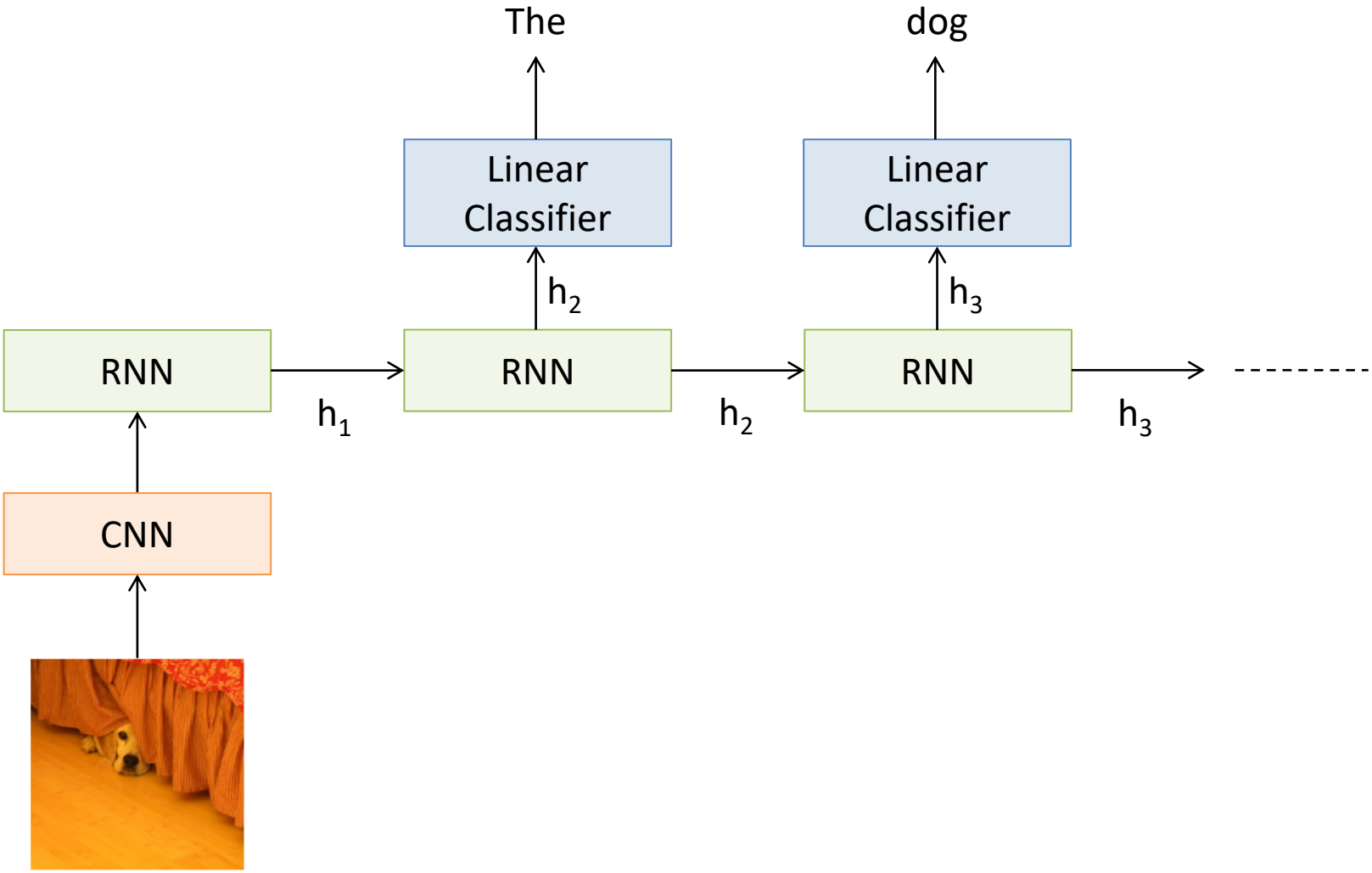


Image Captioning



Motivation

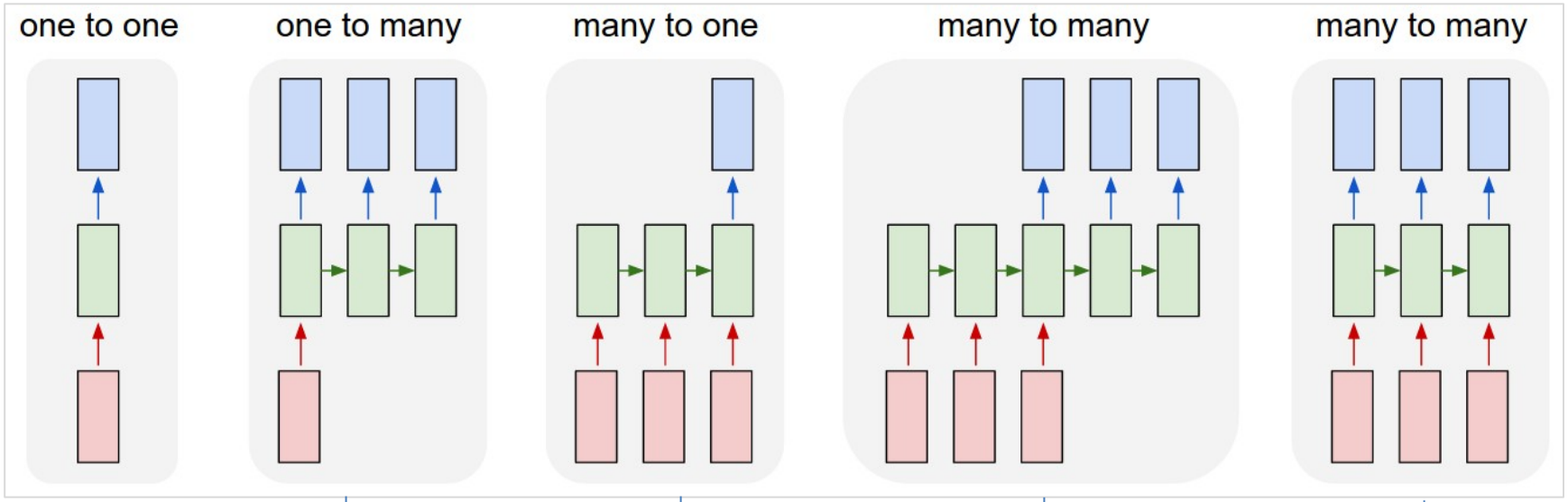


Image classification

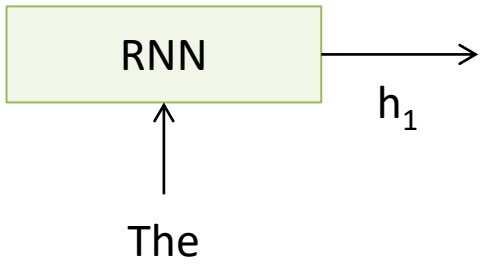
Image captioning

Sentiment analysis

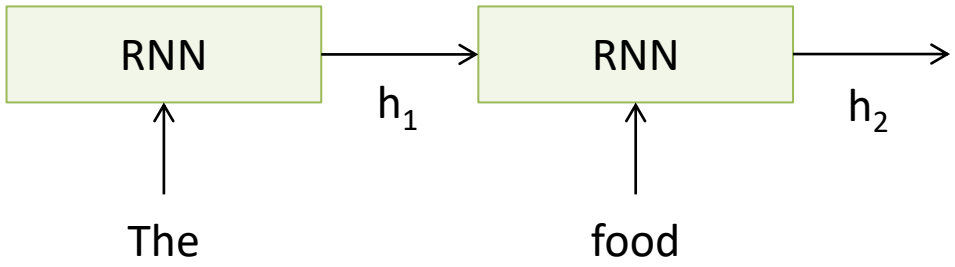
Machine translation

Synced sequence(video classification)

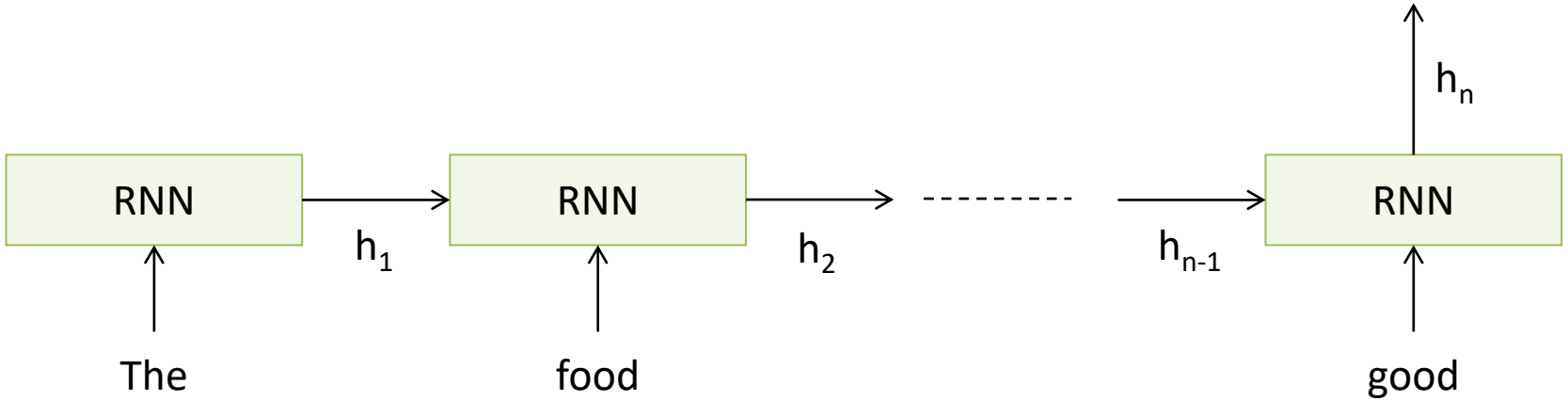
Sentiment Classification



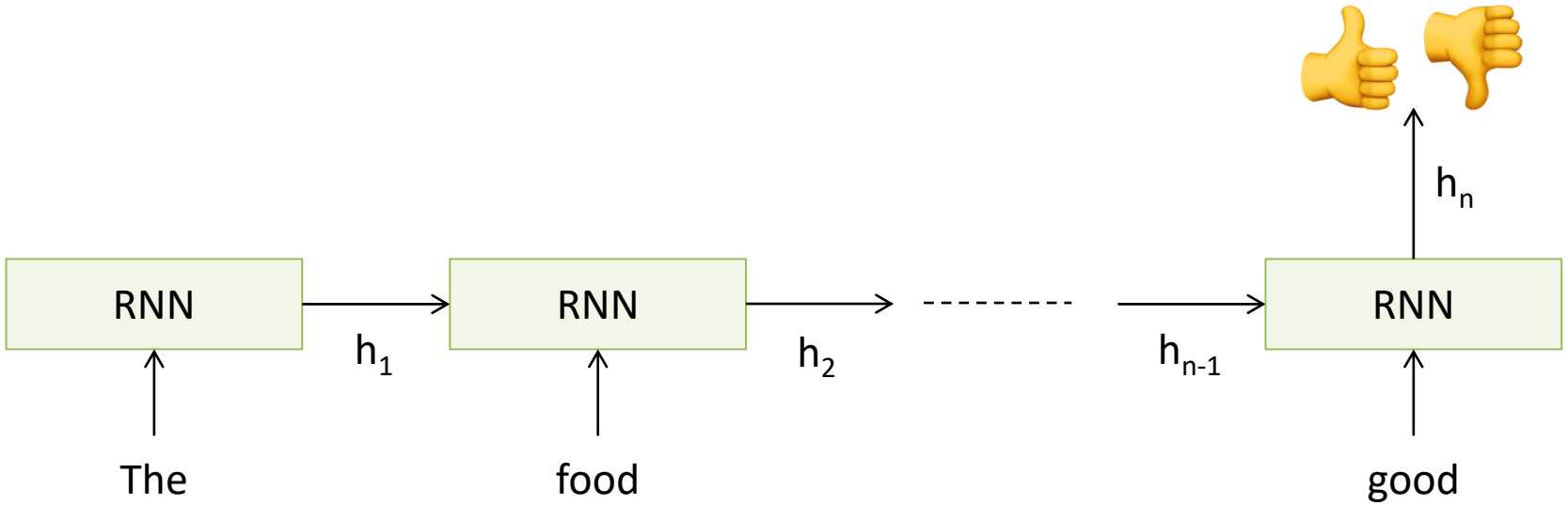
Sentiment Classification



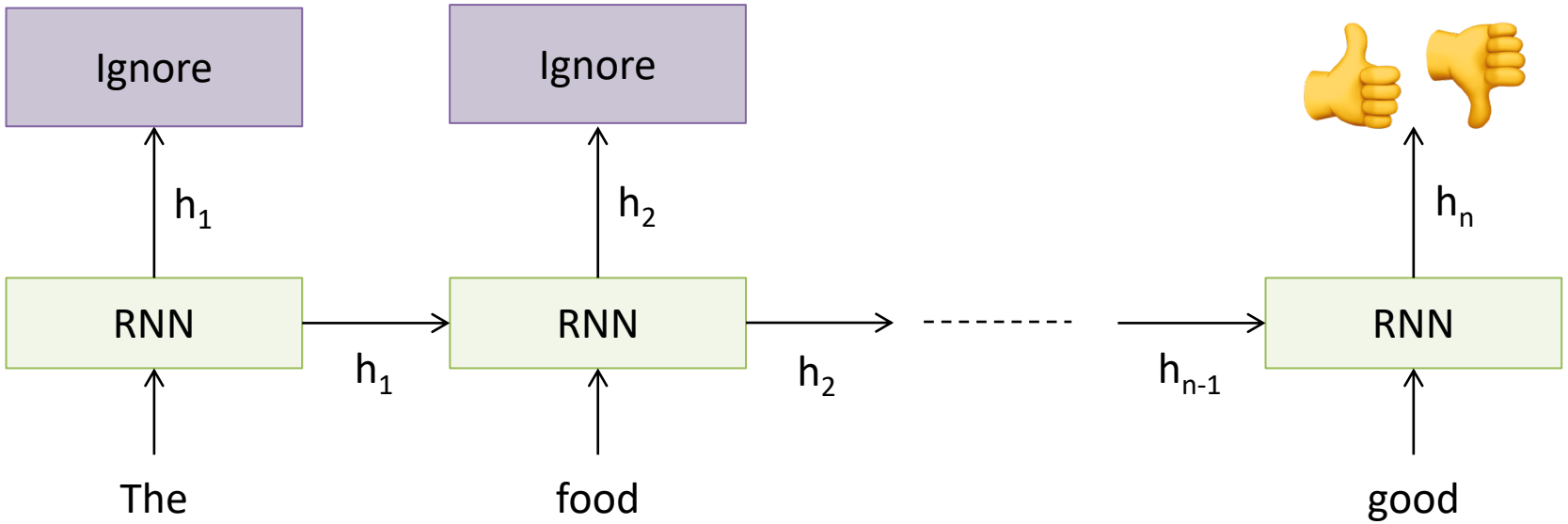
Sentiment Classification



Sentiment Classification



Sentiment Classification



Motivation

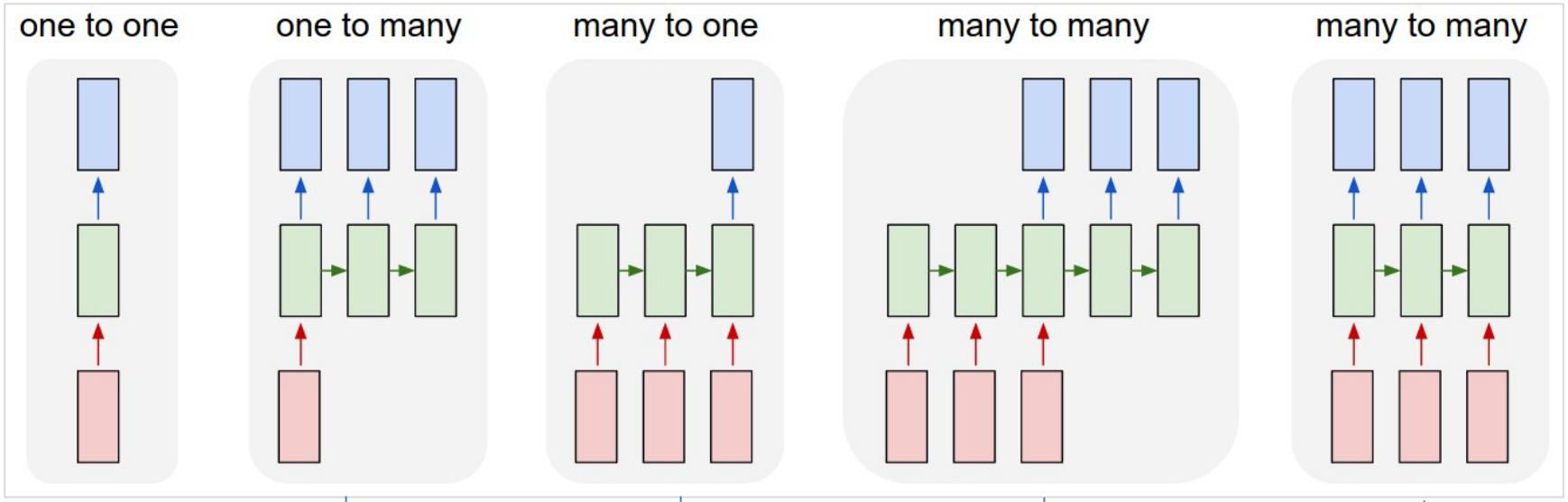


Image classification

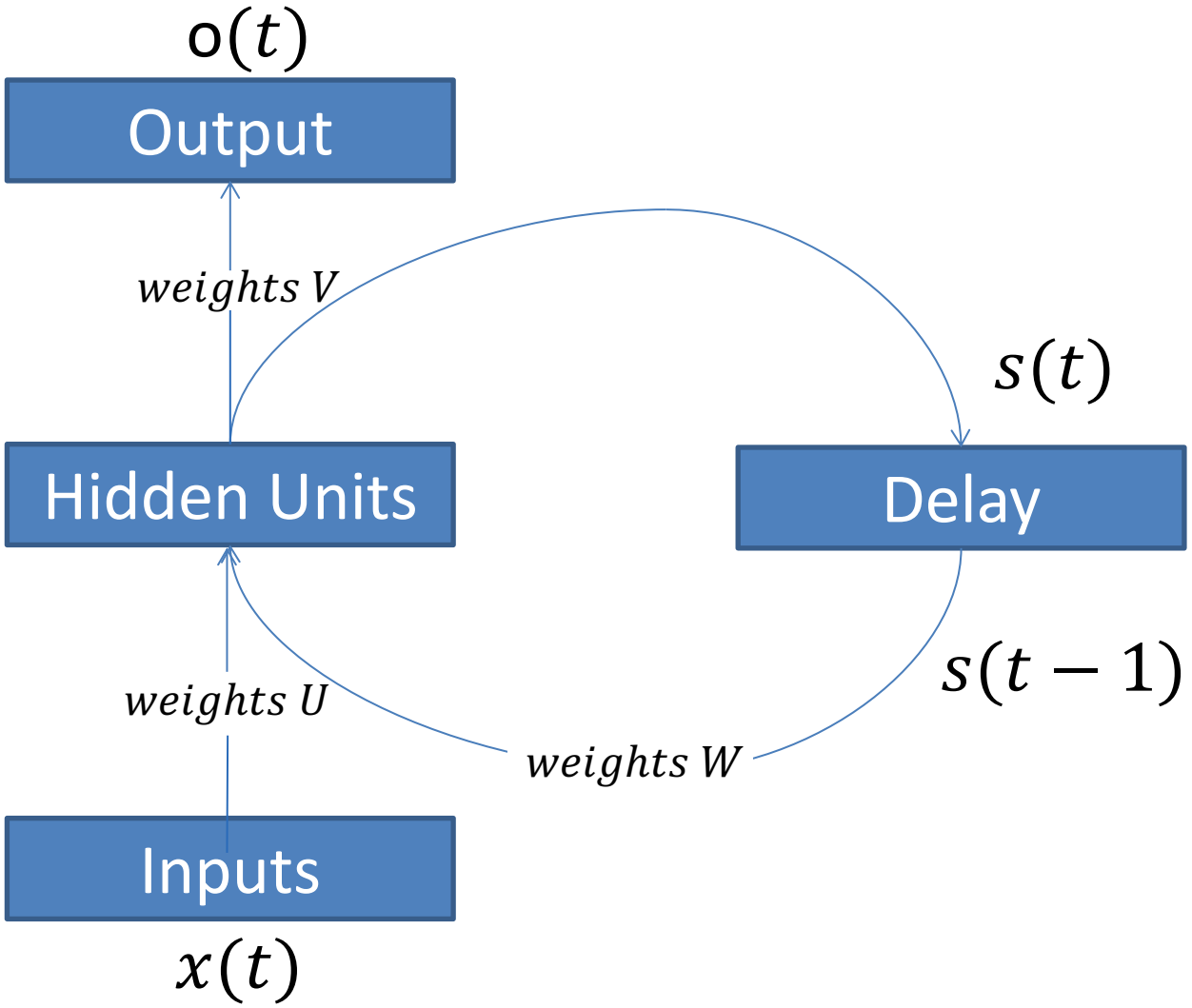
Image captioning

Sentiment analysis

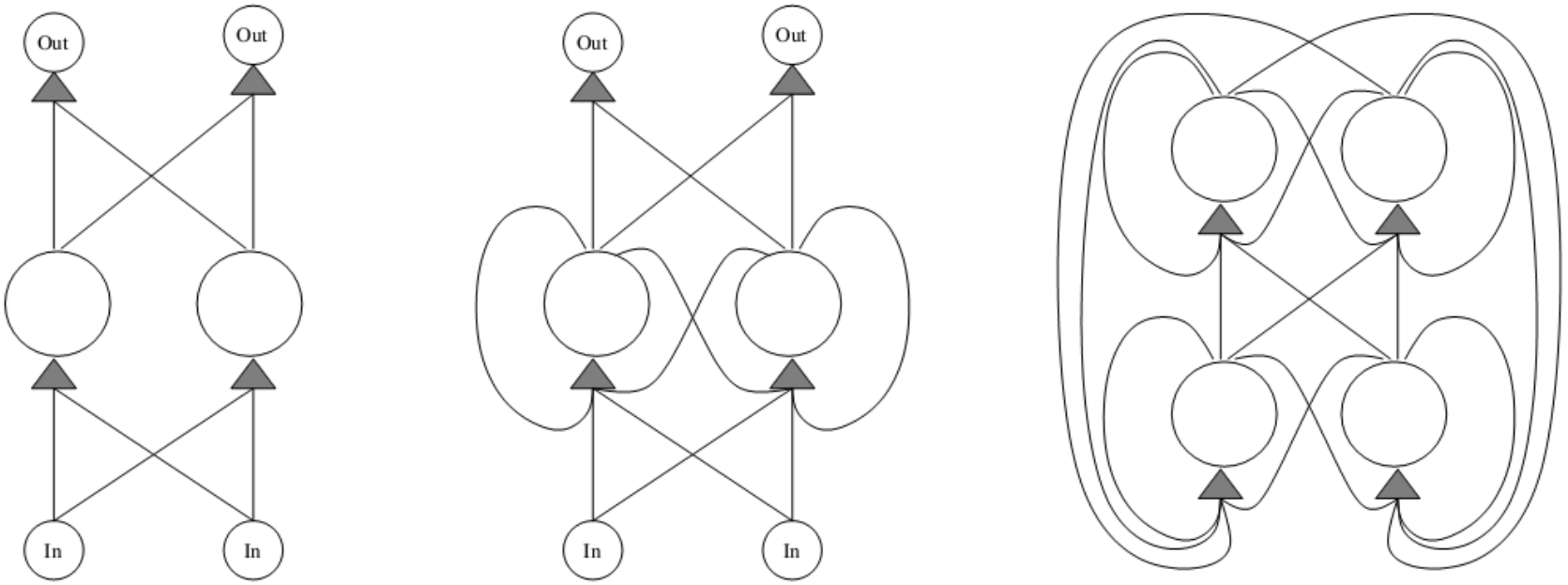
Machine translation

Synced sequence(video classification)

RNN Architecture



RNN Architecture



Left: feed forward neural network
Middle: a simple recurrent neural network
Right: Fully connected recurrent neural network

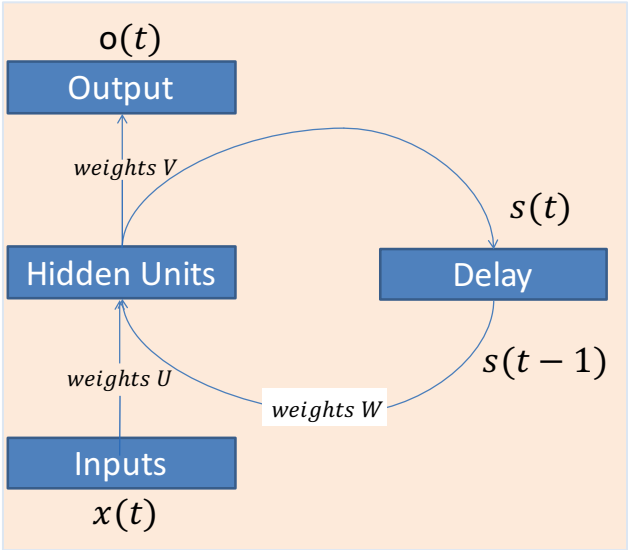
RNN Forward Pass

- The hidden input at time t:

$$a_h(t) = Ux(t) + Ws(t-1)$$

- The hidden activation at time t:

$$s(t) = f_h\left(a_h(t)\right)$$



- The network input to the output unit at time t:

$$a_o(t) = Vs(t)$$

- The output of the network at time t is:

$$o(t) = f_o\left(a_o(t)\right)$$

RNN Architecture

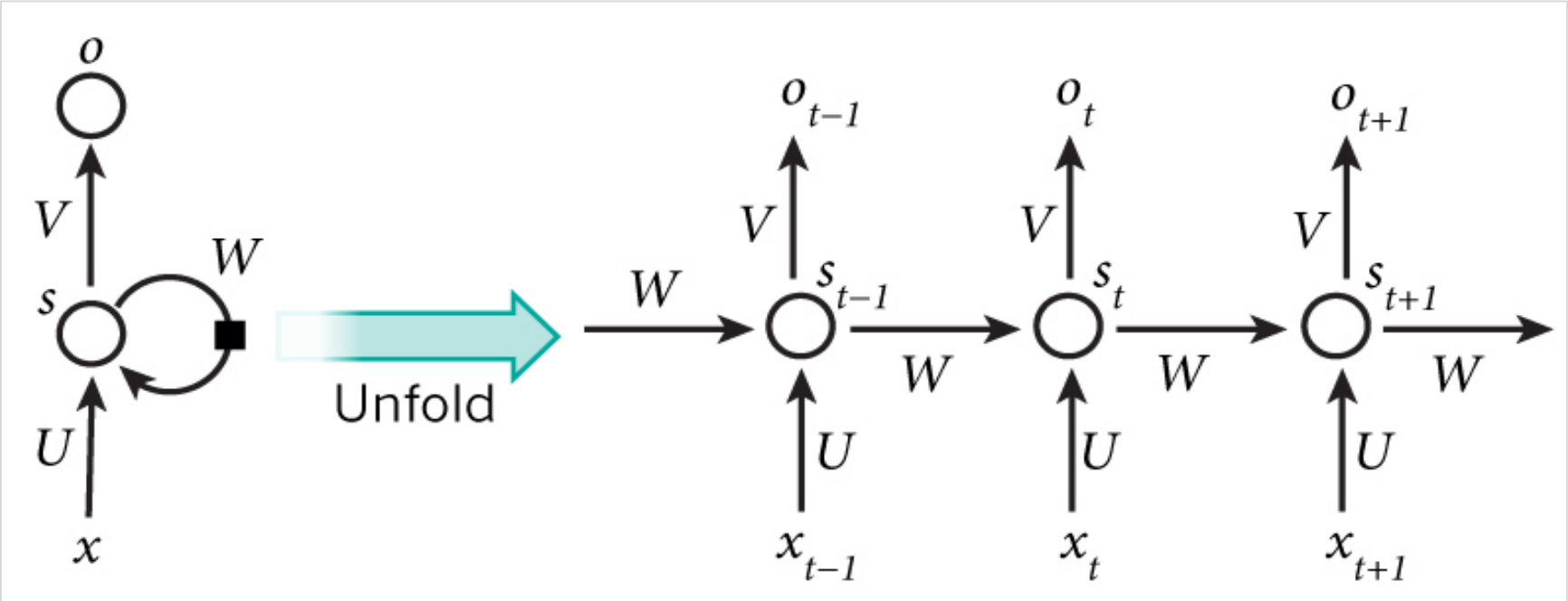
- If a network training sequence starts at time t_0 and ends at t_1, the total **loss function** is the sum over time of the square error function:

$$[e(t)]_k = [d(t)]_k - [o(t)]_k$$

$$E(t) = \frac{1}{2} e(t)^T e(t)$$

$$E_{total} = \sum_t E(t)$$

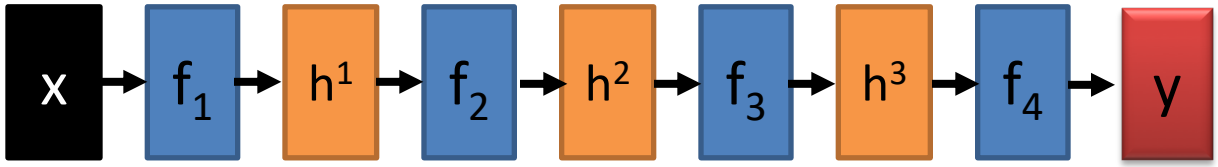
RNN Architecture



- The recurrent network can be converted into a feed forward network by **unfolding over time**

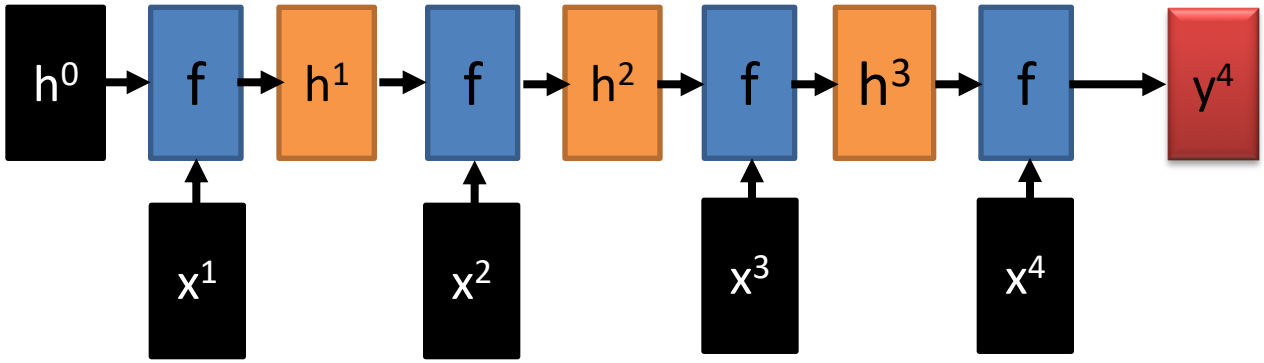
Feedforward vs recurrent nets

1. Feedforward network does not have input (or output) at each computation step
2. Feedforward network has different parameters for each layer



$h^t = f_t(h^{t-1}) = \sigma(W^t h^{t-1} + b^t)$

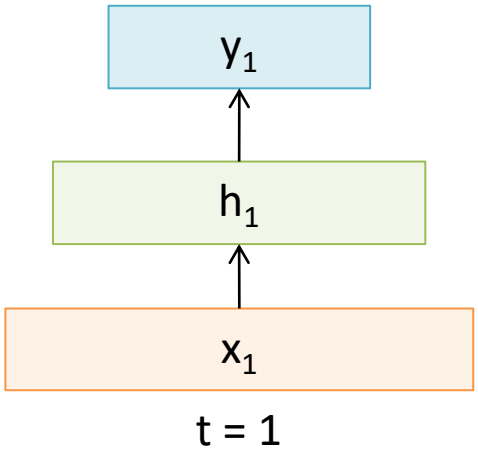
t is layer



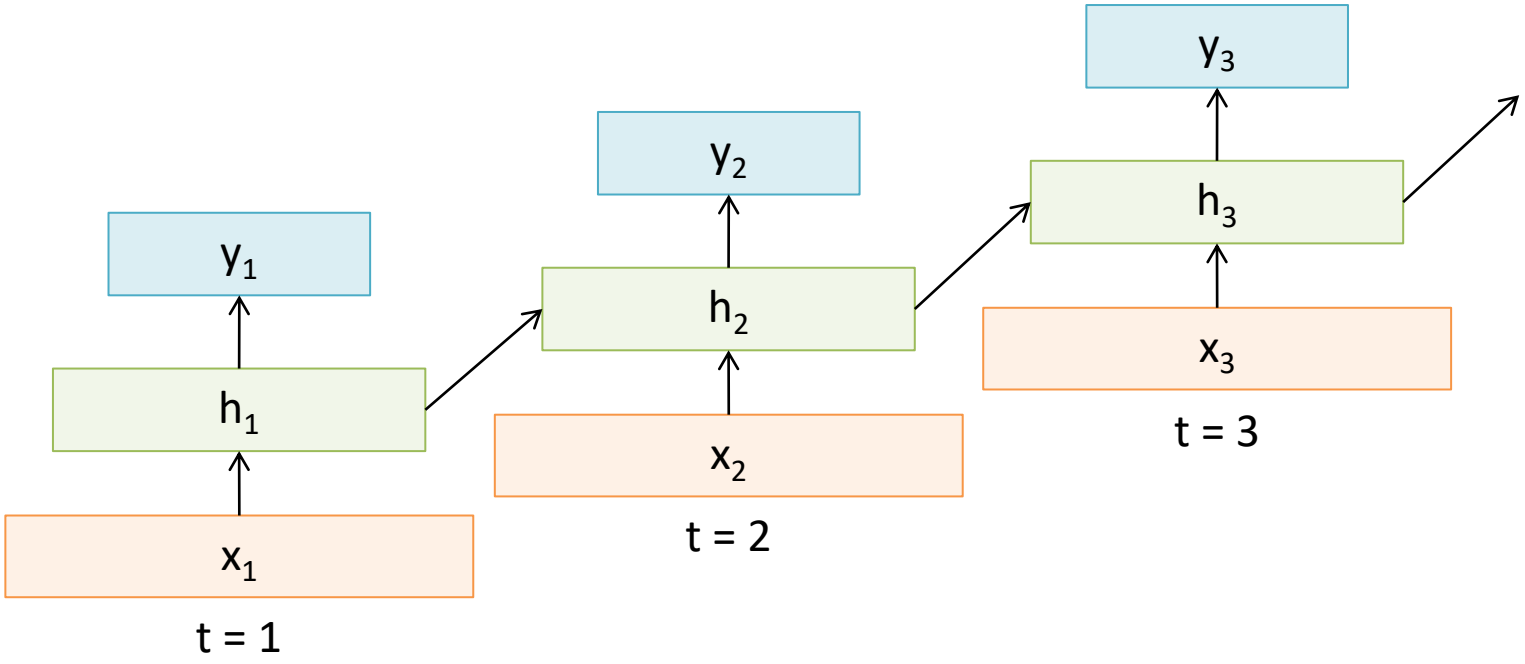
t is time step

$h^t = f(h^{t-1}, x^t) = \sigma(W^h h^{t-1} + W^i x^t + b^i)$

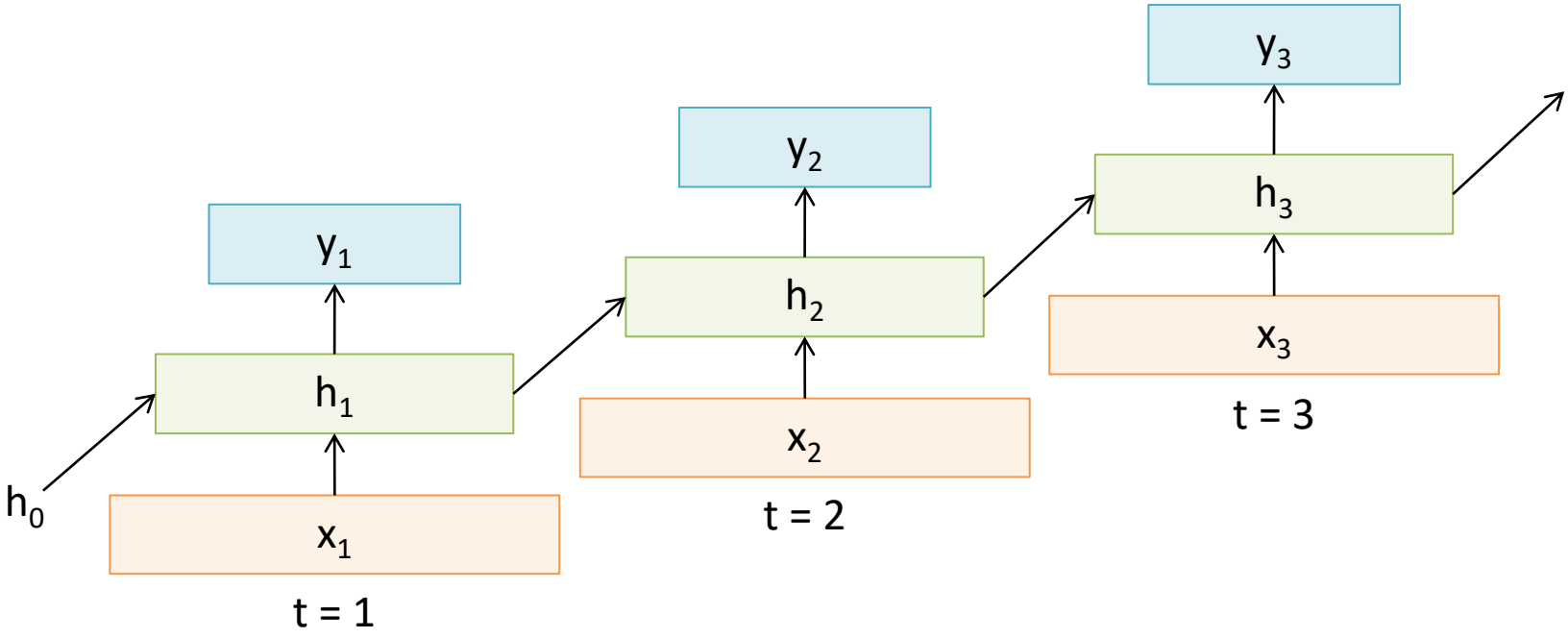
Sample Feed-forward Network



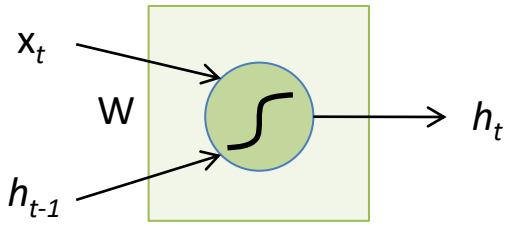
Sample RNN



Sample RNN

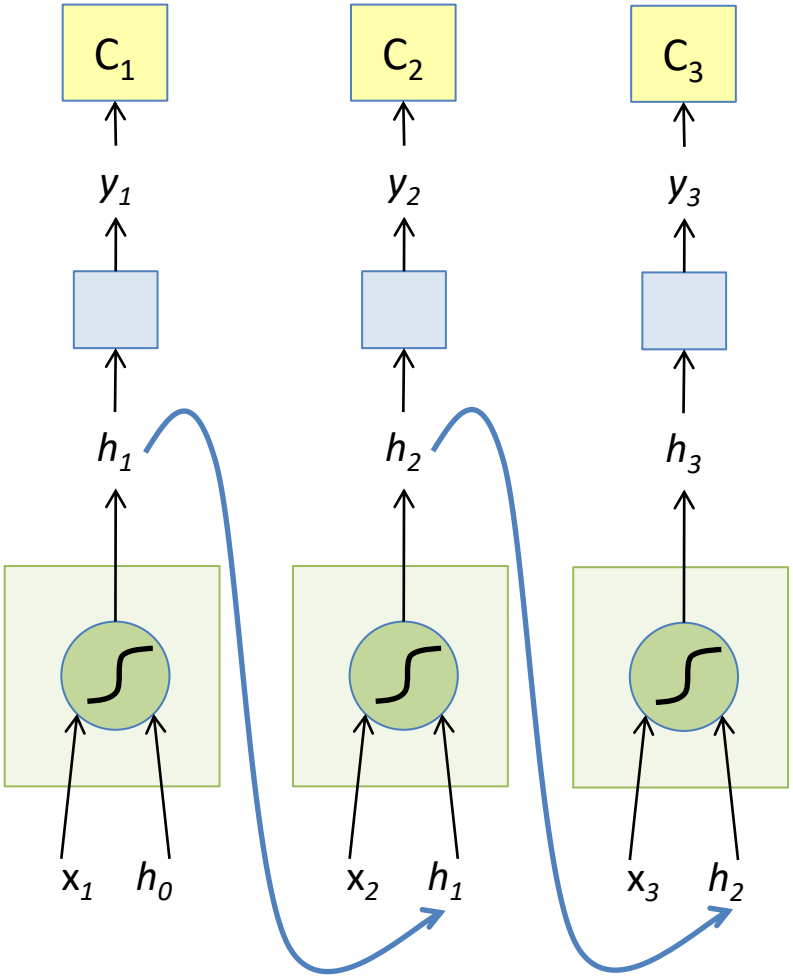


The Vanilla RNN Cell



$$h_t = \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

The Vanilla RNN Forward

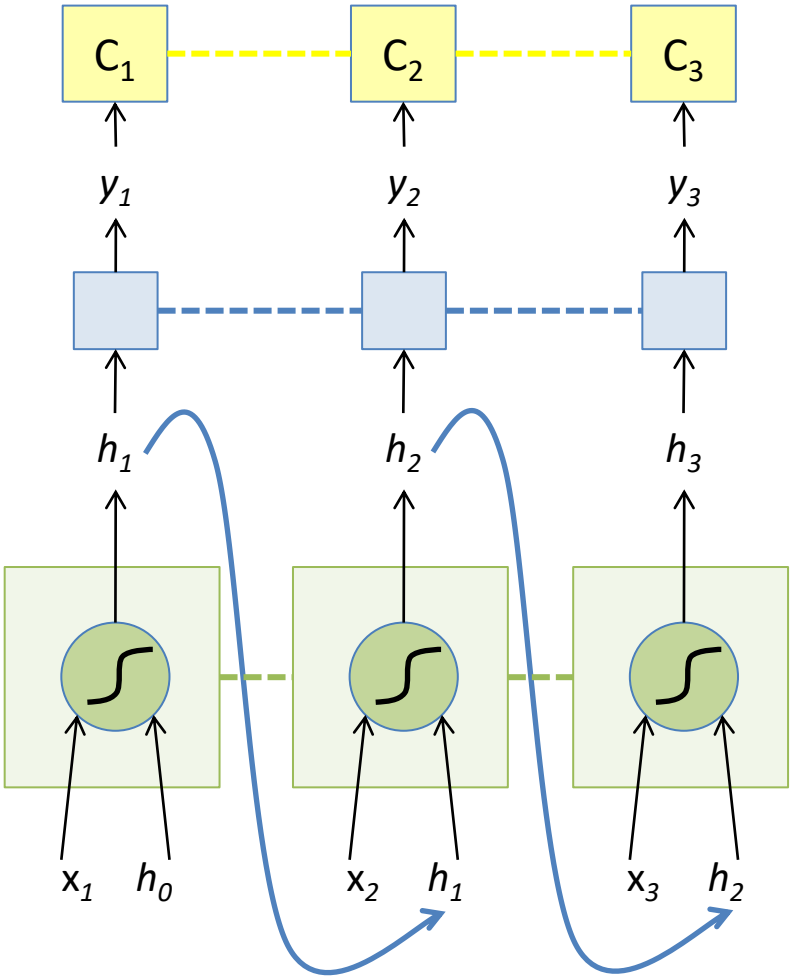


$$h_t = \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

$$y_t = \mathrm{F}(h_t)$$

$$C_t = \mathrm{Loss}(y_t, \mathrm{GT}_t)$$

The Vanilla RNN Forward



$$h_t = \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

$$y_t = \mathrm{F}(h_t)$$

$$C_t = \mathrm{Loss}(y_t, \mathrm{GT}_t)$$

------ indicates shared weights

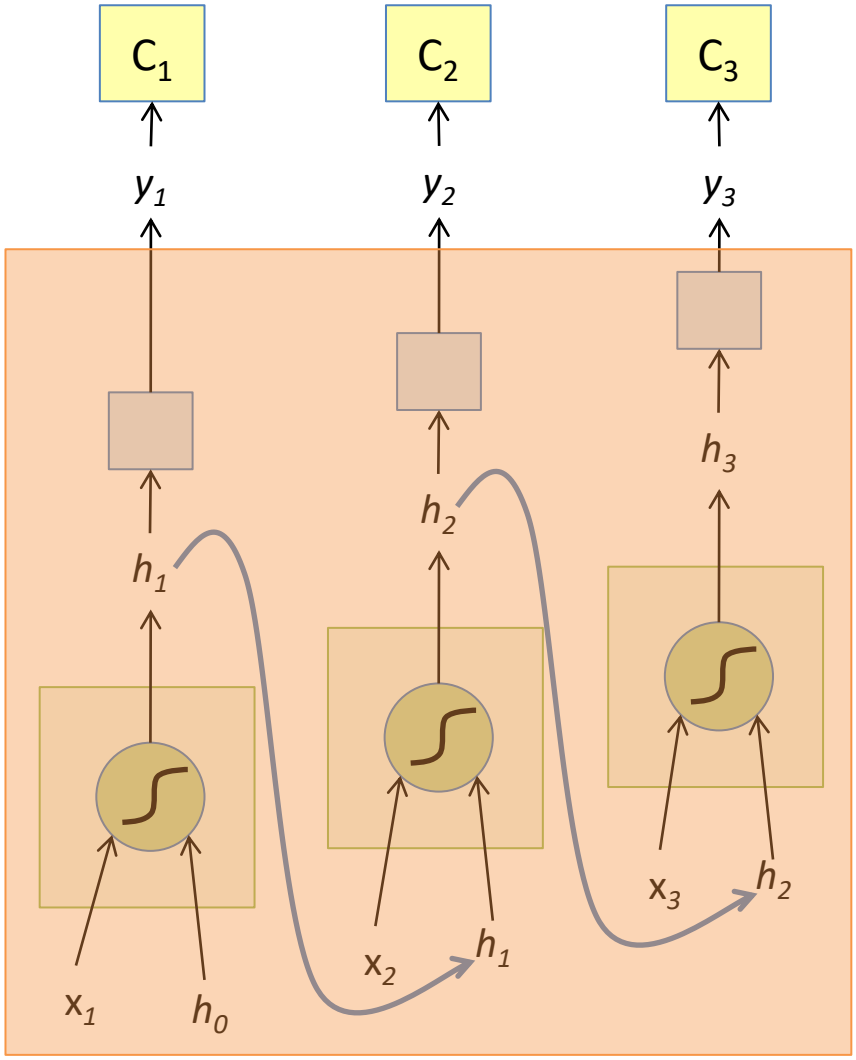
Note that the weights are shared over time

Essentially, copies of the RNN cell are made over time (unrolling/unfolding), with different inputs at different time steps

BackPropagation Through Time (BPTT)

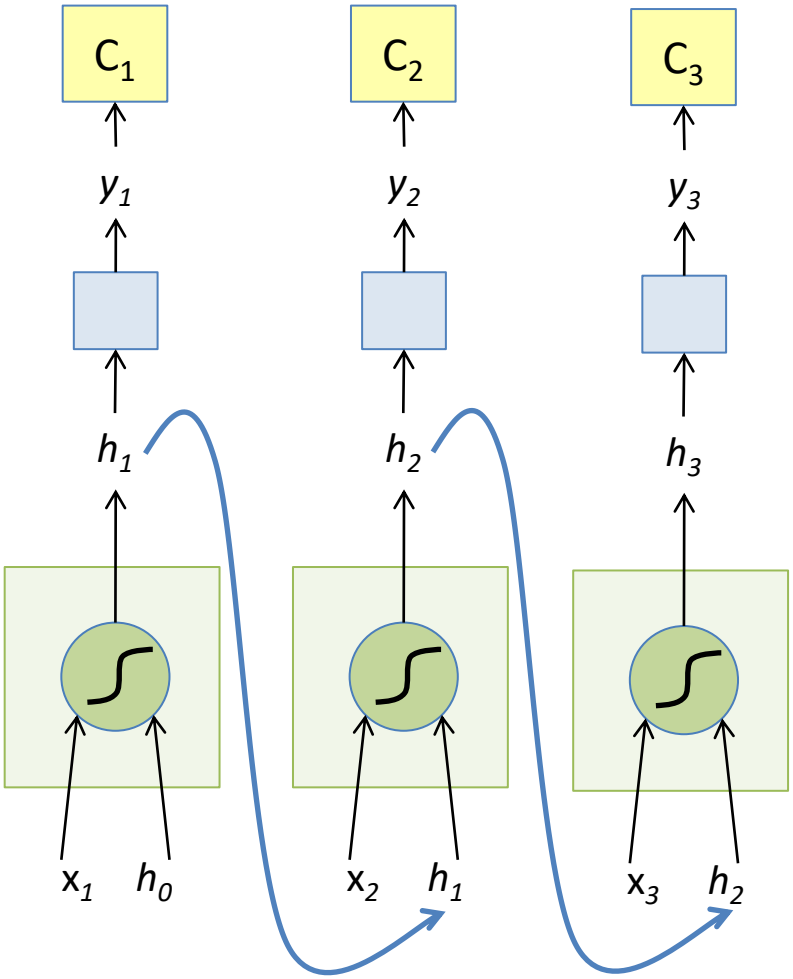
- One of the methods used to train RNNs
- The unfolded network (used during forward pass) is treated as one big feed-forward network
- This unfolded network accepts the whole time series as input
- The weight updates are computed for each copy in the unfolded network, then summed and then applied to the RNN weights

The Unfolded Vanilla RNN

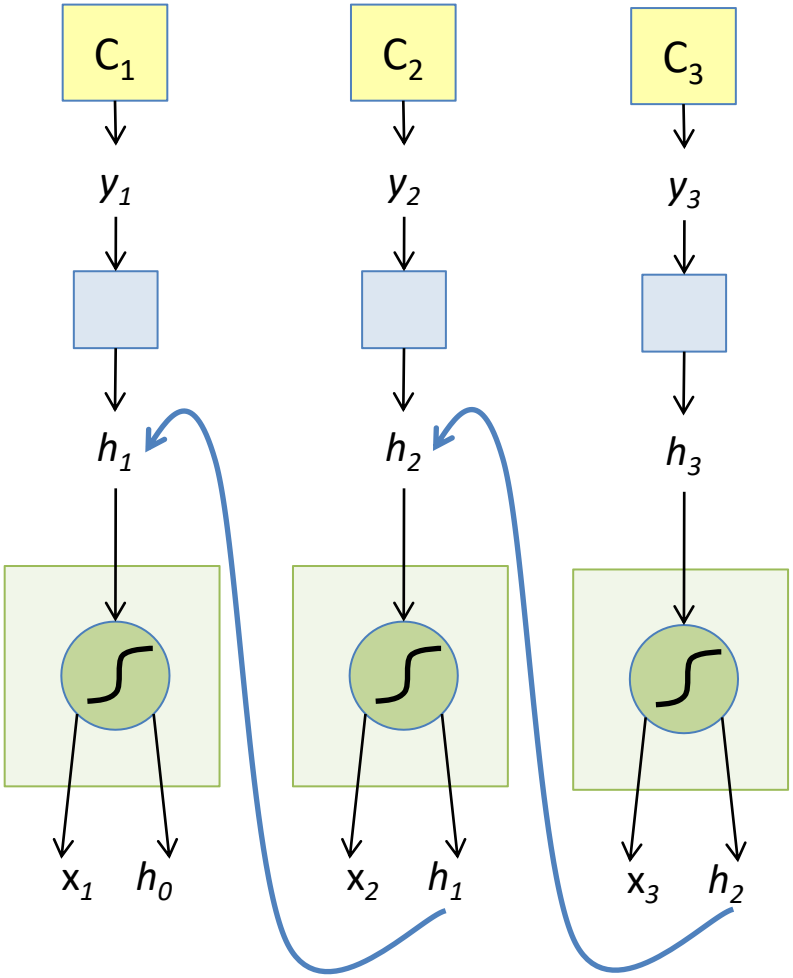


- Treat the unfolded network as one big feed-forward network!
- This big network takes in entire sequence as an input
- Compute gradients through the usual backpropagation
- Update shared weights

The Unfolded Vanilla RNN Forward

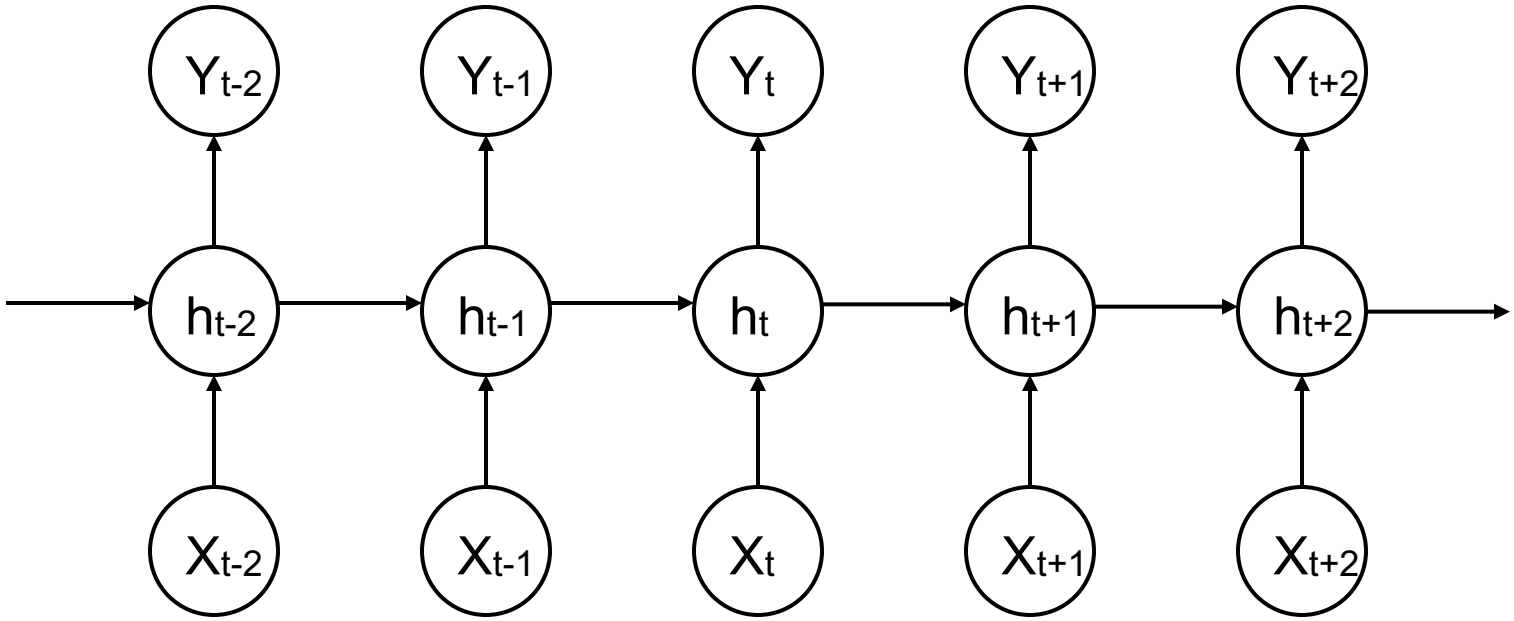


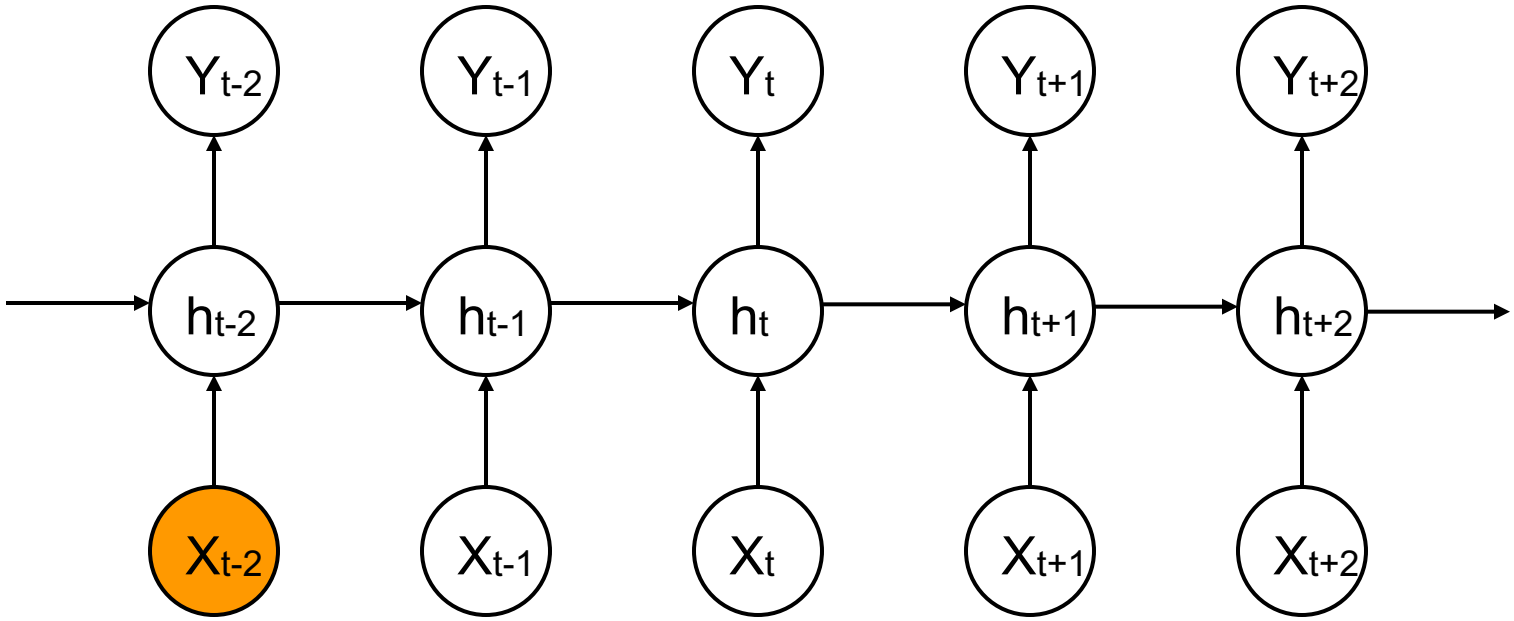
The Unfolded Vanilla RNN Forward

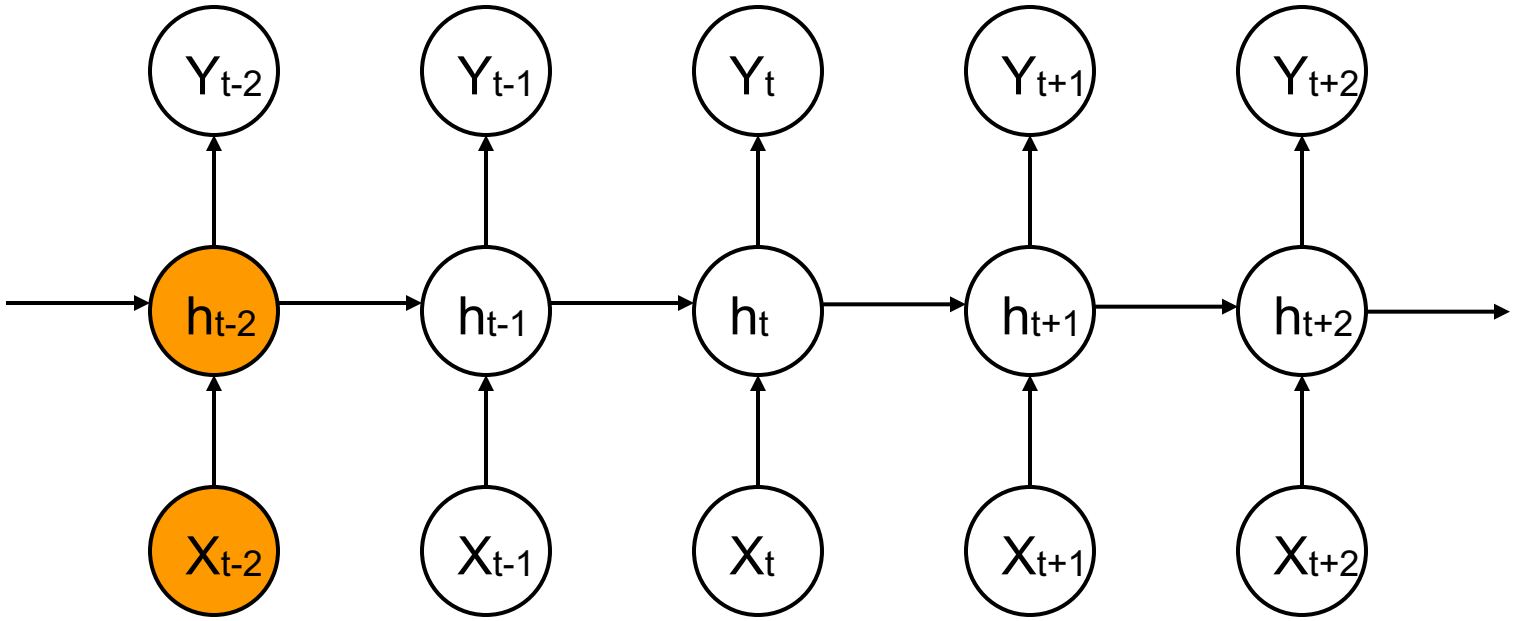


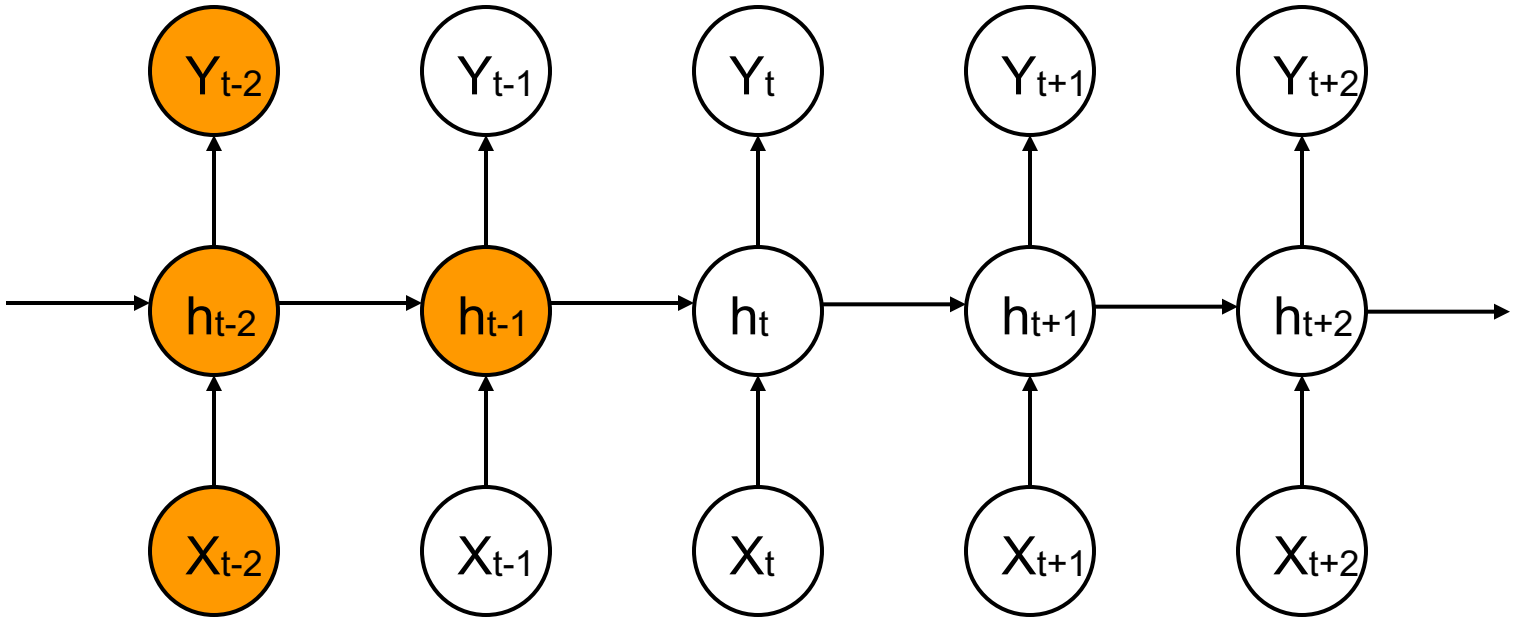
Back Propagation Through Time

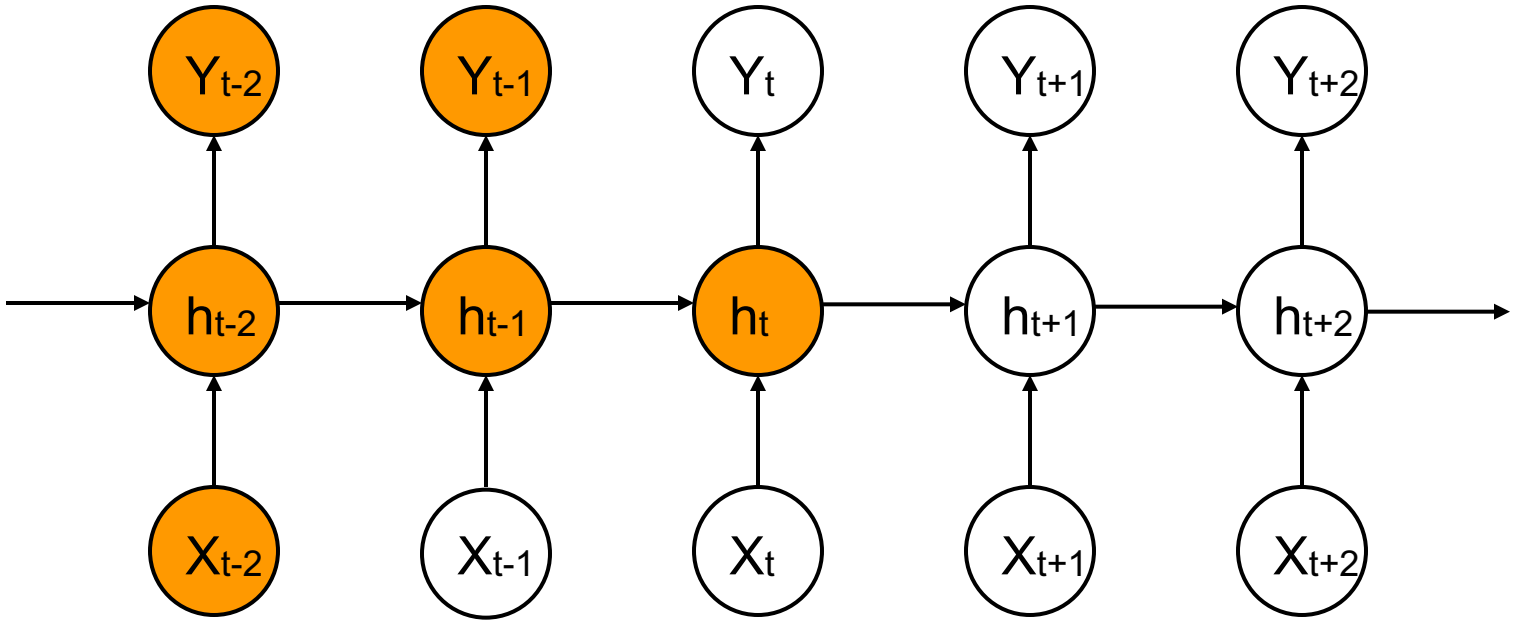
- BPTT learning algorithm is an extension of standard backpropagation that performs gradients descent on an unfolded network.
- The errors have to be back-propagated through time as well as through the network
- The gradient descent weight updates have contributions from each time step.
- The key difference with traditional FFNNs is that we sum up the gradients for at each time step, because the parameters are shared across time steps.

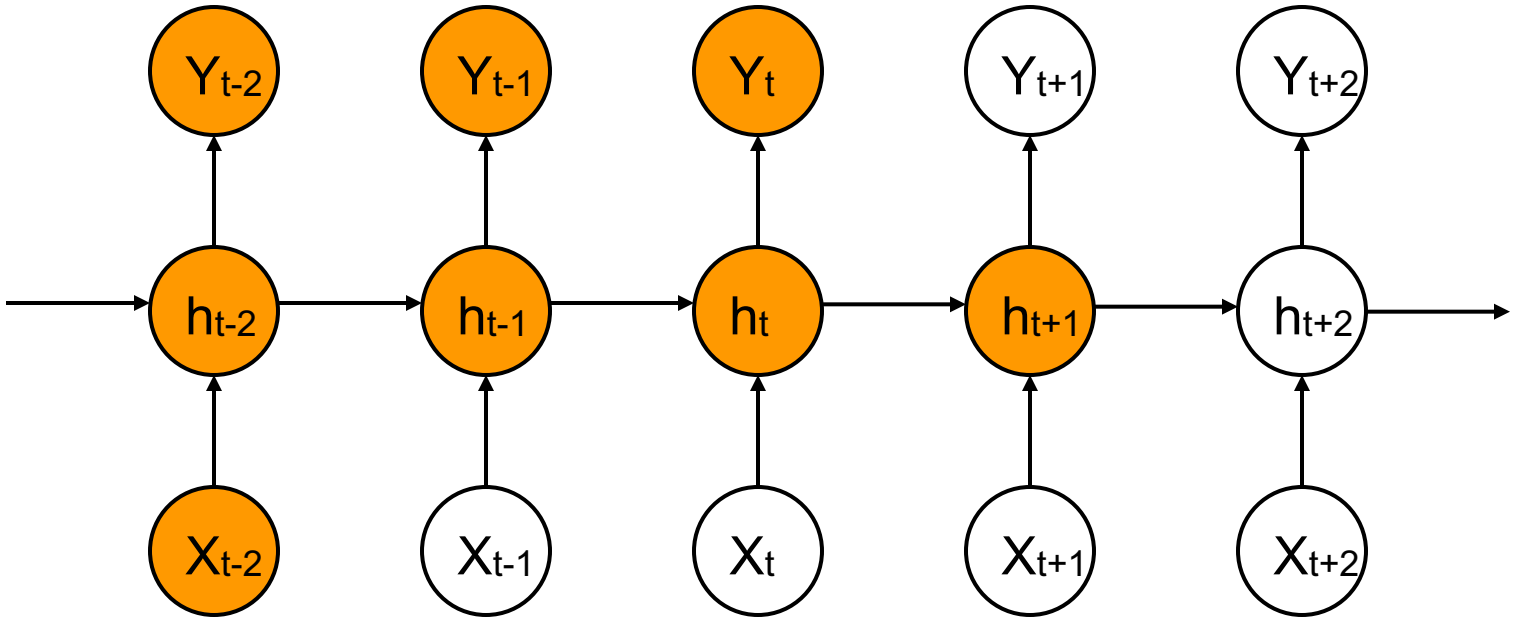

Yt-2
Yt-1
Yt
Yt+1
Yt+2
ht-2
ht-1
ht
ht+1
ht+2
Xt-2
Xt-1
Xt
Xt+1
Xt+2

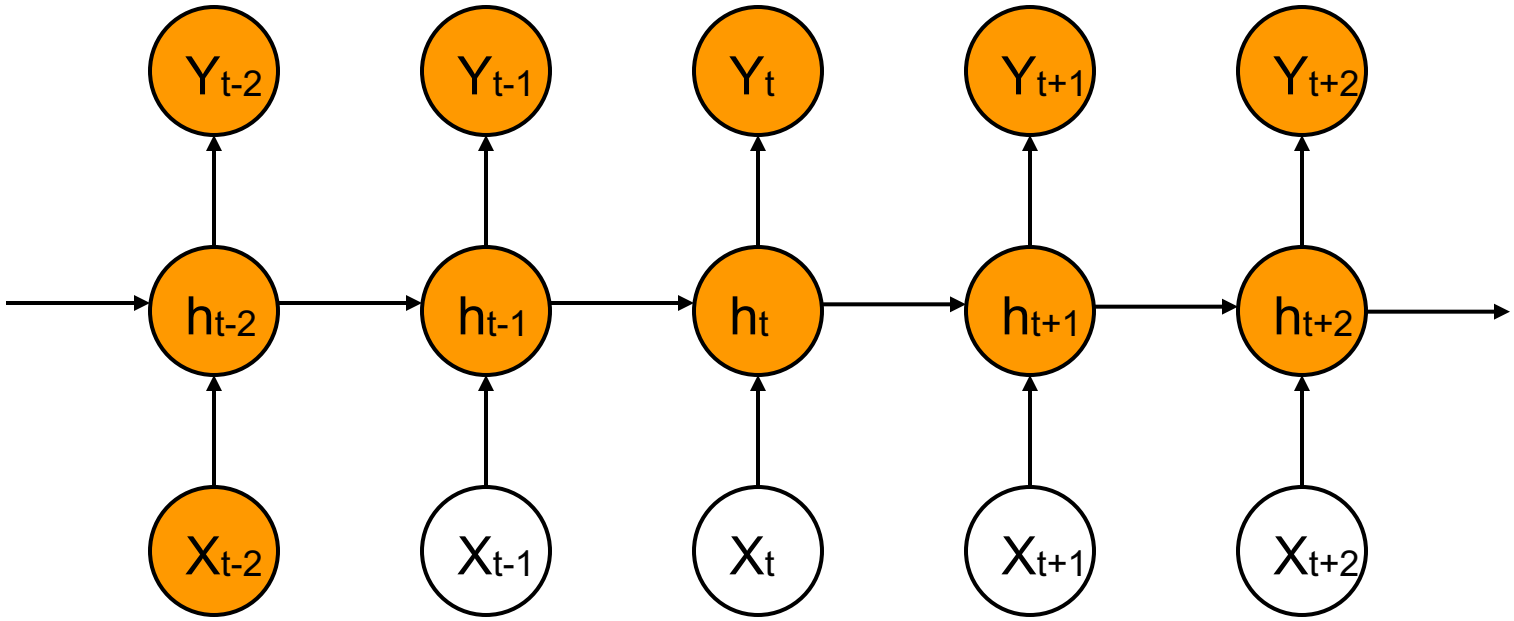

Yt-2
Yt-1
Yt
Yt+1
Yt+2
ht-2
ht-1
ht
ht+1
ht+2
Xt-2
Xt-1
Xt
Xt+1
Xt+2

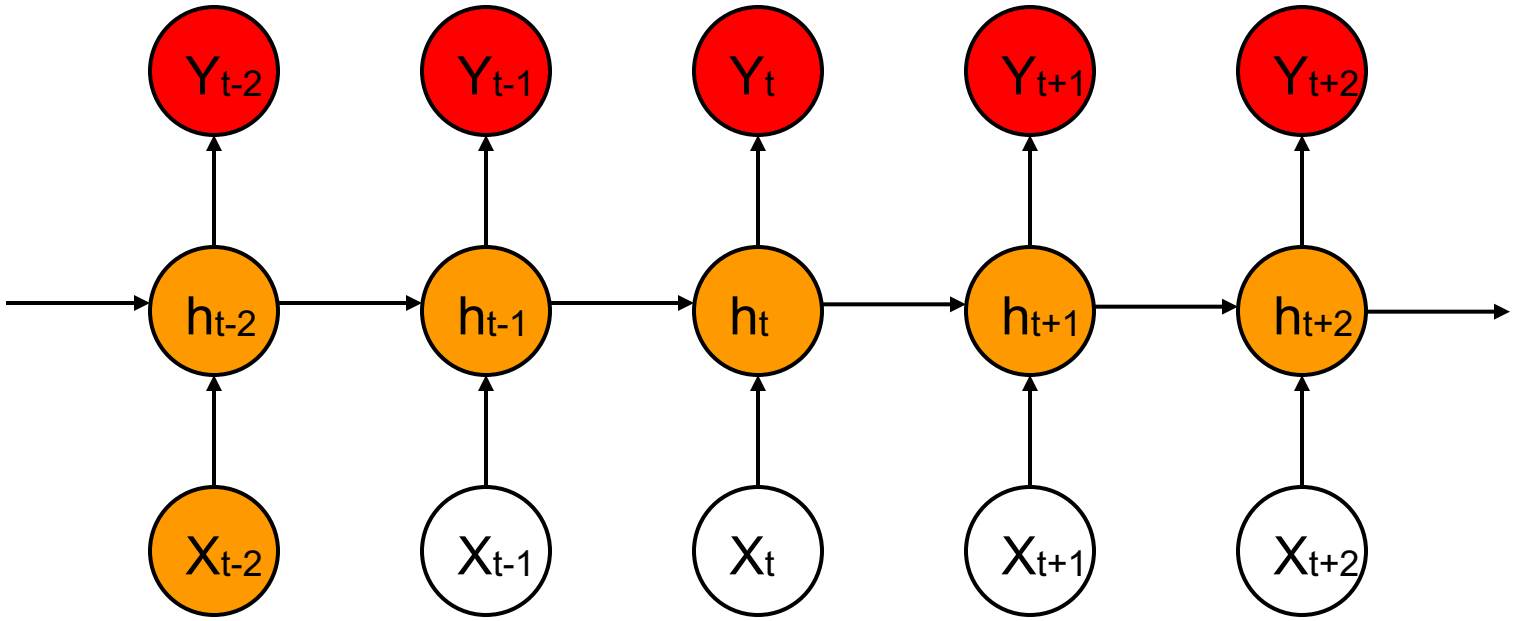

Yt-2
Yt-1
Yt
Yt+1
Yt+2
ht-2
ht-1
ht
ht+1
ht+2
Xt-2
Xt-1
Xt
Xt+1
Xt+2

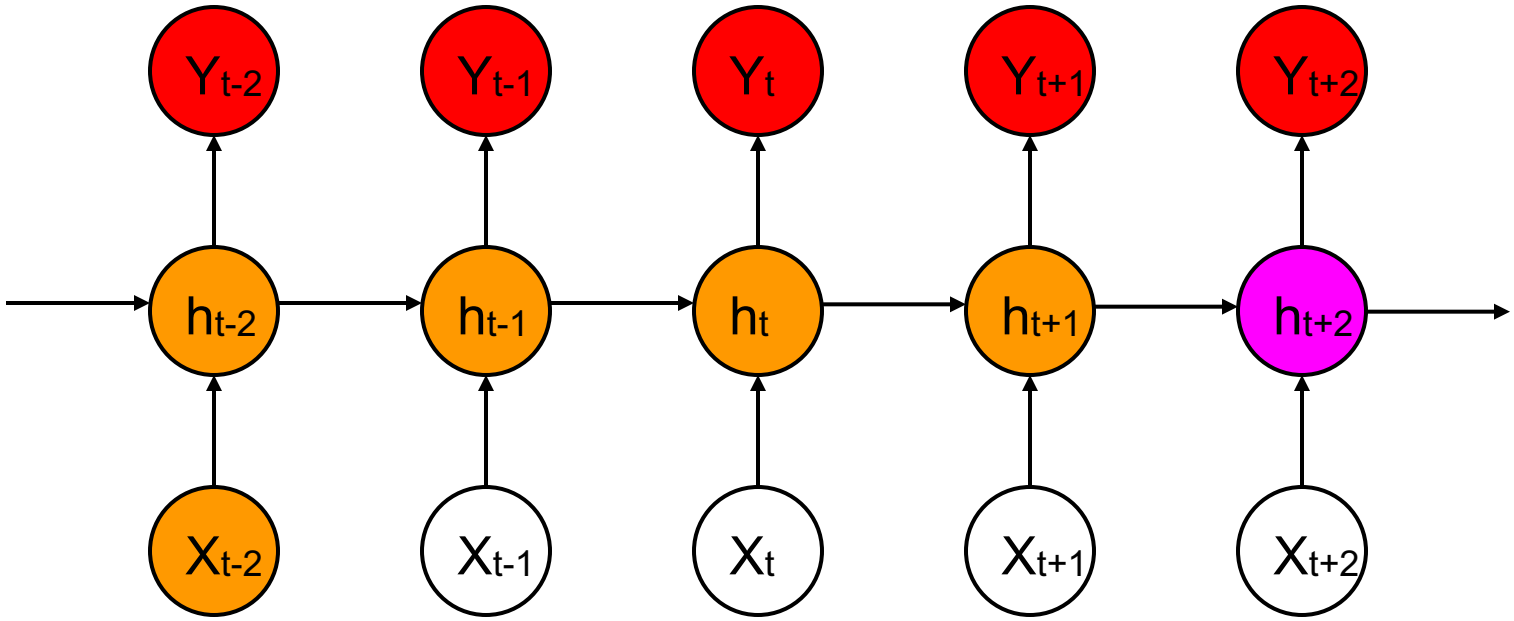

Yt-2
Yt-1
Yt
Yt+1
Yt+2
ht-2
ht-1
ht
ht+1
ht+2
Xt-2
Xt-1
Xt
Xt+1
Xt+2

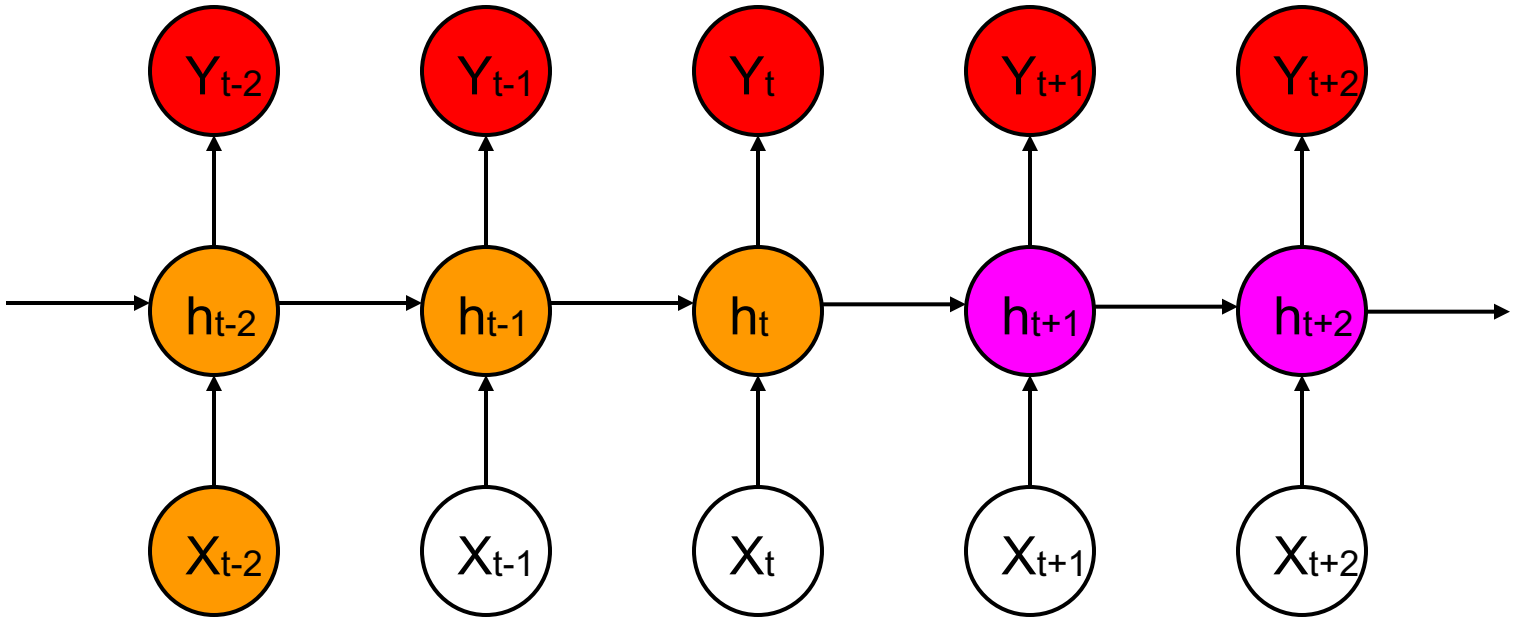

Yt-2
Yt-1
Yt
Yt+1
Yt+2
ht-2
ht-1
ht
ht+1
ht+2
Xt-2
Xt-1
Xt
Xt+1
Xt+2

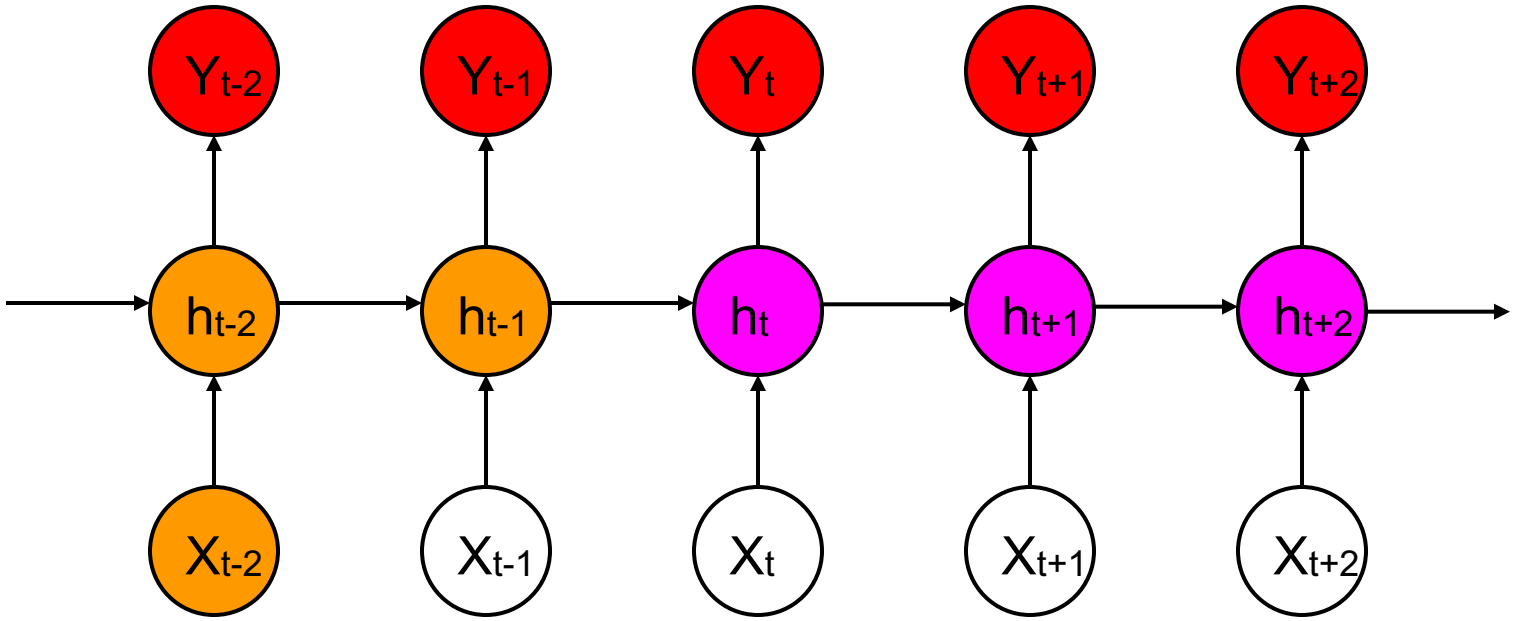

Yt-2
Yt-1
Yt
Yt+1
Yt+2
ht-2
ht-1
ht
ht+1
ht+2
Xt-2
Xt-1
Xt
Xt+1
Xt+2

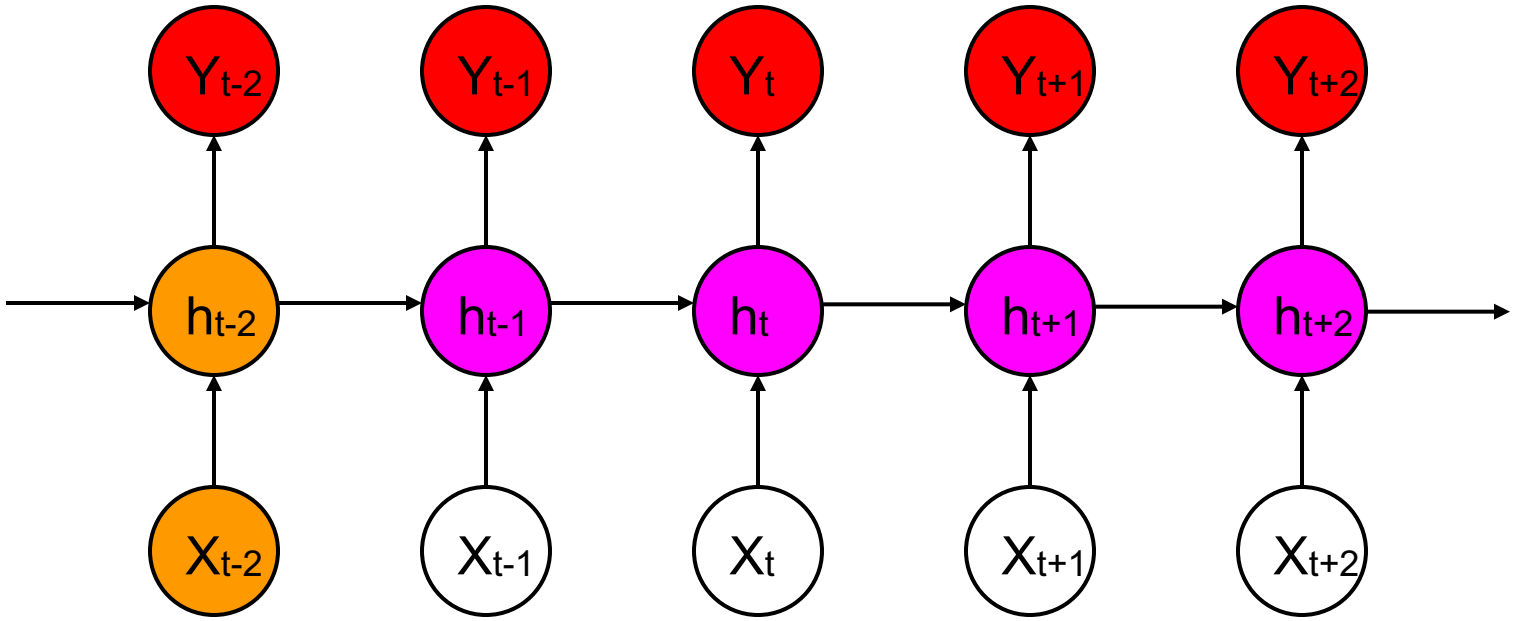

Yt-2
Yt-1
Yt
Yt+1
Yt+2
ht-2
ht-1
ht
ht+1
ht+2
Xt-2
Xt-1
Xt
Xt+1
Xt+2

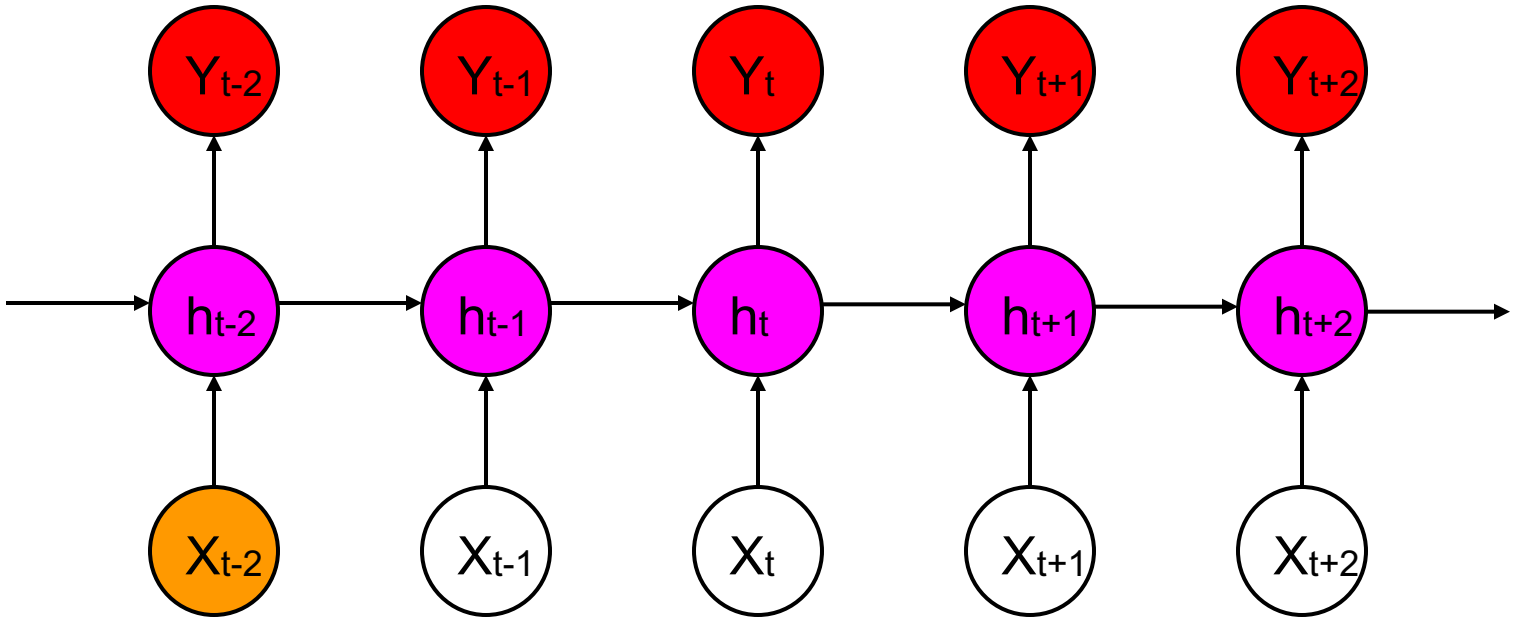

Yt-2
Yt-1
Yt
Yt+1
Yt+2
ht-2
ht-1
ht
ht+1
ht+2
Xt-2
Xt-1
Xt
Xt+1
Xt+2


Yt-2
Yt-1
Yt
Yt+1
Yt+2
ht-2
ht-1
ht
ht+1
ht+2
Xt-2
Xt-1
Xt
Xt+1
Xt+2


Yt-2
Yt-1
Yt
Yt+1
Yt+2
ht-2
ht-1
ht
ht+1
ht+2
Xt-2
Xt-1
Xt
Xt+1
Xt+2


Yt-2
Yt-1
Yt
Yt+1
Yt+2
ht-2
ht-1
ht
ht+1
ht+2
Xt-2
Xt-1
Xt
Xt+1
Xt+2


Yt-2
Yt-1
Yt
Yt+1
Yt+2
ht-2
ht-1
ht
ht+1
ht+2
Xt-2
Xt-1
Xt
Xt+1
Xt+2


Yt-2
Yt-1
Yt
Yt+1
Yt+2
ht-2
ht-1
ht
ht+1
ht+2
Xt-2
Xt-1
Xt
Xt+1
Xt+2

An irritating extra-issue

- We need to specify the initial activity state of all the hidden and output units.
- We could just fix these initial states to have some default value like 0.5.
- But it is better to treat the initial states as learned parameters.
- We learn them in the same way as we learn the weights.
  - Start off with an initial random guess for the initial states.
  - At the end of each training sequence, backpropagate through time all the way to the initial states to get the gradient of the error function with respect to each initial state.
  - Adjust the initial states by following the negative gradient.

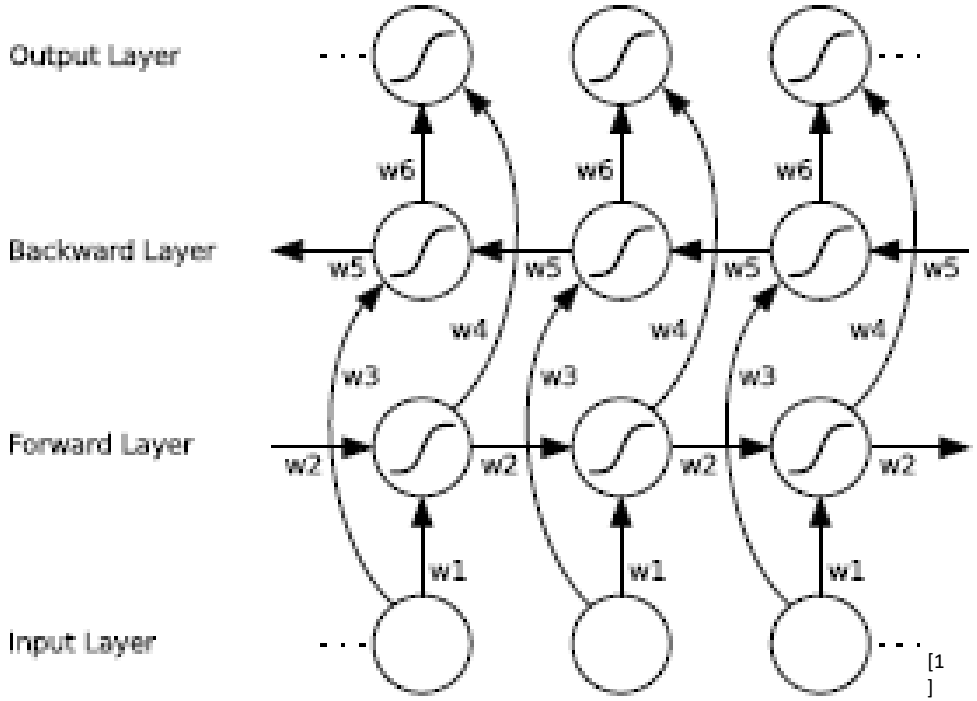
Bidirectional RNNs

- For many sequence labelling tasks it would be beneficial to have access to future as well as past context. However, since standard RNNs process sequences in temporal order, they ignore future context.

- Bidirectional recurrent neural networks solve this problem. The basic idea of BRNNs is to present each training sequence forwards and backwards to two separate recurrent hidden layers, both of which are connected to the same output layer.

- This structure provides the output layer with complete past and future context for every point in the input sequence, without displacing the inputs from the relevant targets.

Bidirectional RNNs



An unfolded bidirectional RNN

Forward pass for BRNN

for *t = 1* to *T* **do**
< forward pass for the forward hidden layer, storing activations at each timestep >

for *t = T* to *1* **do**
< forward pass for the backward hidden layer, storing activations at each timestep >

for all *t* **in any order do**
< forward pass for the output layer, using the stored activations from both the hidden layers >

Backward pass for BRNN

for all *t* **in any order do**

< backward pass for the output layer, storing δ terms at each timestep >

for *t = T* to *1* **do**

< BPTT backward pass for the forward hidden layer, using the stored δ terms from the output layer >

for *t = 1* to *T* **do**

< BPTT backward pass for the backward hidden layer, using the stored δ terms from the output layer >

Exploding and vanishing gradients

- What happens to the magnitude of the gradients as we backpropagate through many layers?
  - If the weights are small, the gradients shrink exponentially.
  - If the weights are big the gradients grow exponentially.

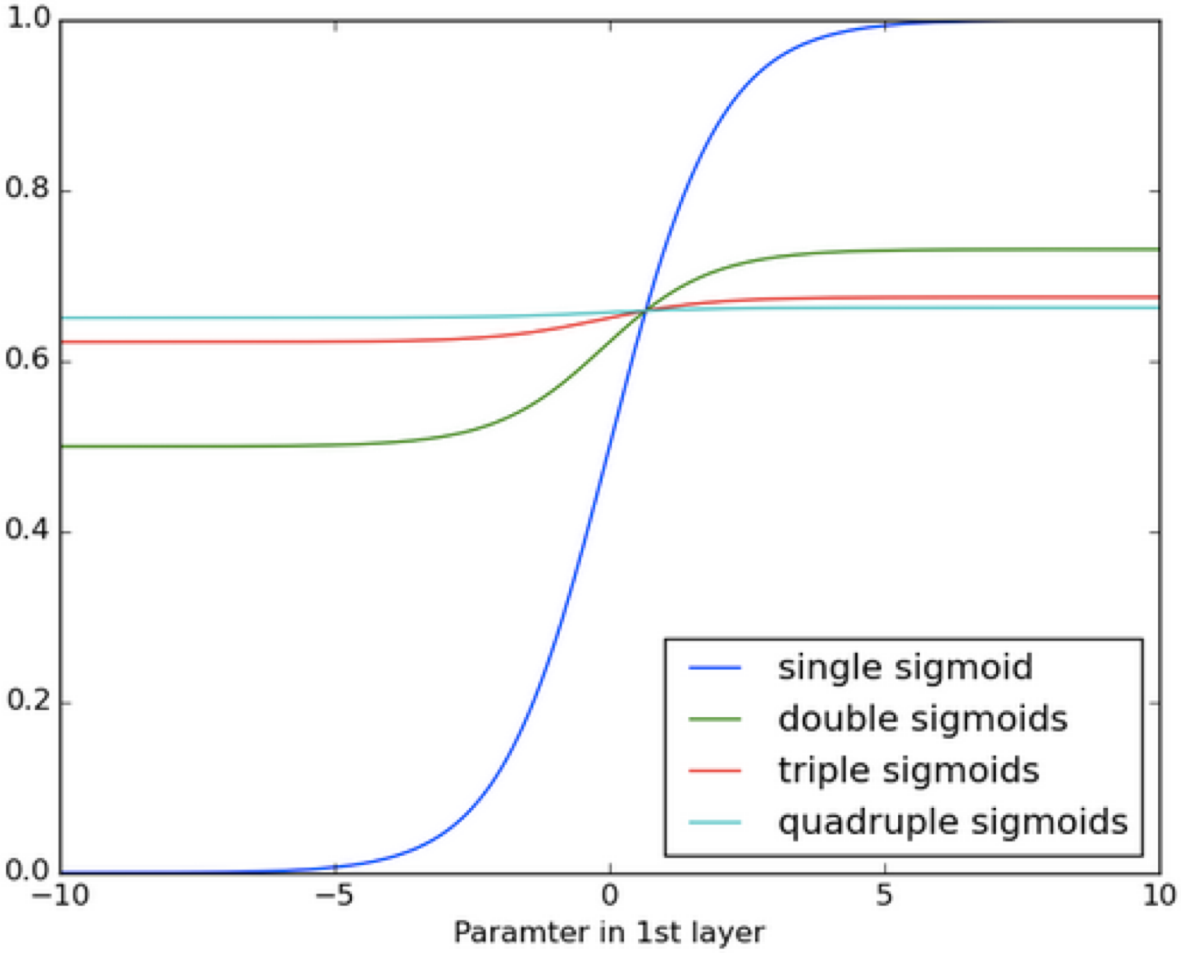
- Typical feed-forward neural nets can cope with these exponential effects because they only have a few hidden layers.

Exploding and vanishing gradients

- In an RNN trained on long sequences (*e.g.* 100 time steps) the gradients can easily explode or vanish.

- Even with good initial weights, its very hard to detect that the current target output depends on an input from many time-steps ago.
  - So RNNs have difficulty dealing with long-range dependencies.
  - No long-time memory allowed!

Vanishing Gradients



Possible solutions

In case of exploding gradient

- stop backpropagating after a certain point, which is usually not optimal because not all of the weights get updated
- penalize or artificially reduce gradient
- put a maximum limit on a gradient

Vanishing gradient happens more often, and it is a much bigger problem…

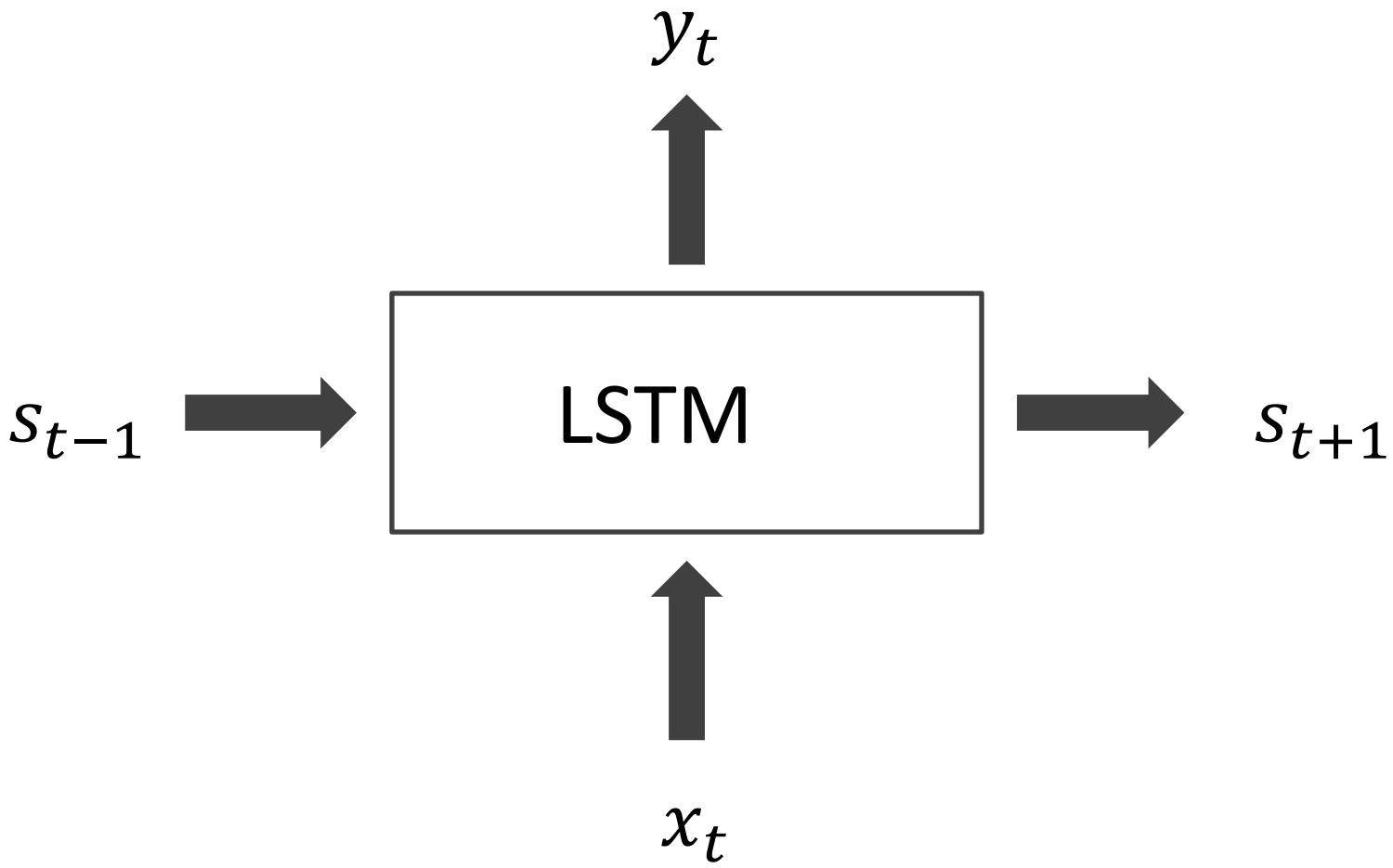
Outline - LSTM

- Introduction
- Motivation
- RNN architecture
- RNN problems
- Long Short Term Memory
- How LSTM solves the problem
- Conclusions

LSTM - introduction

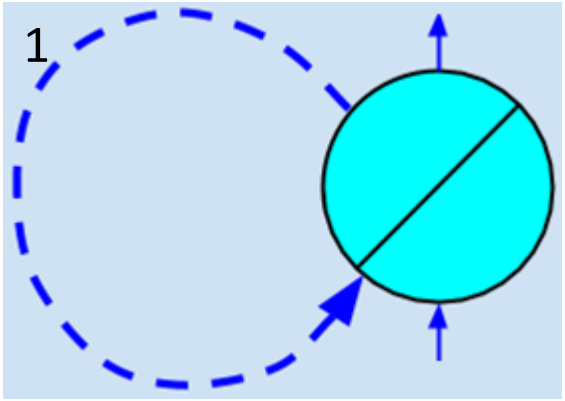
- LSTM was invented to solve the vanishing gradients problem.
- LSTM maintain a more constant error flow in the backpropogation process.
- LSTM can learn over more than 1000 time steps , and thus can handle large sequences that are linked remotely.

LSTM – same idea as RNN



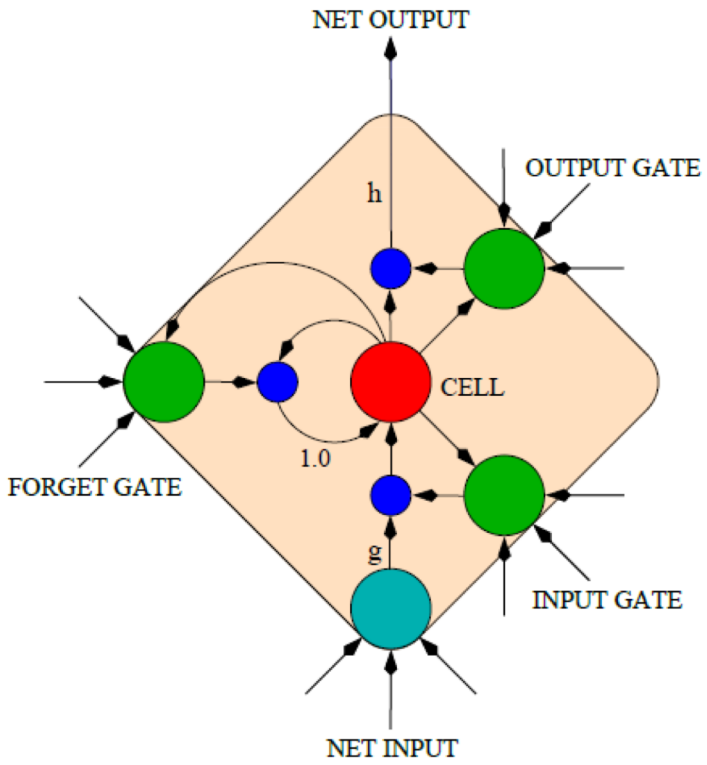
LSTM – memory cell

- LSTM networks introduce a new structure called a memory cell.
- Each memory cell contains a node with a self-connected recurrent edge of fixed weight one, ensuring that the gradient can pass across many time steps without Vanishing – which is called CEC (constant error carousel)



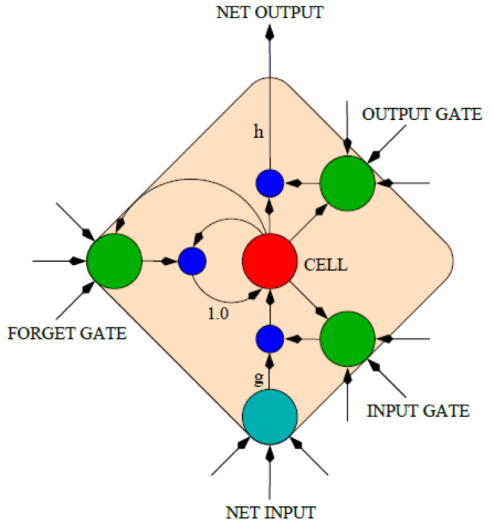
LSTM – memory cell

- Each memory cell contains four main elements:
  - Input gate
  - Forget gate
  - Output gate
  - Neuron with self-recurrent
- These gates allow the cells to keep and access information over long periods of time.



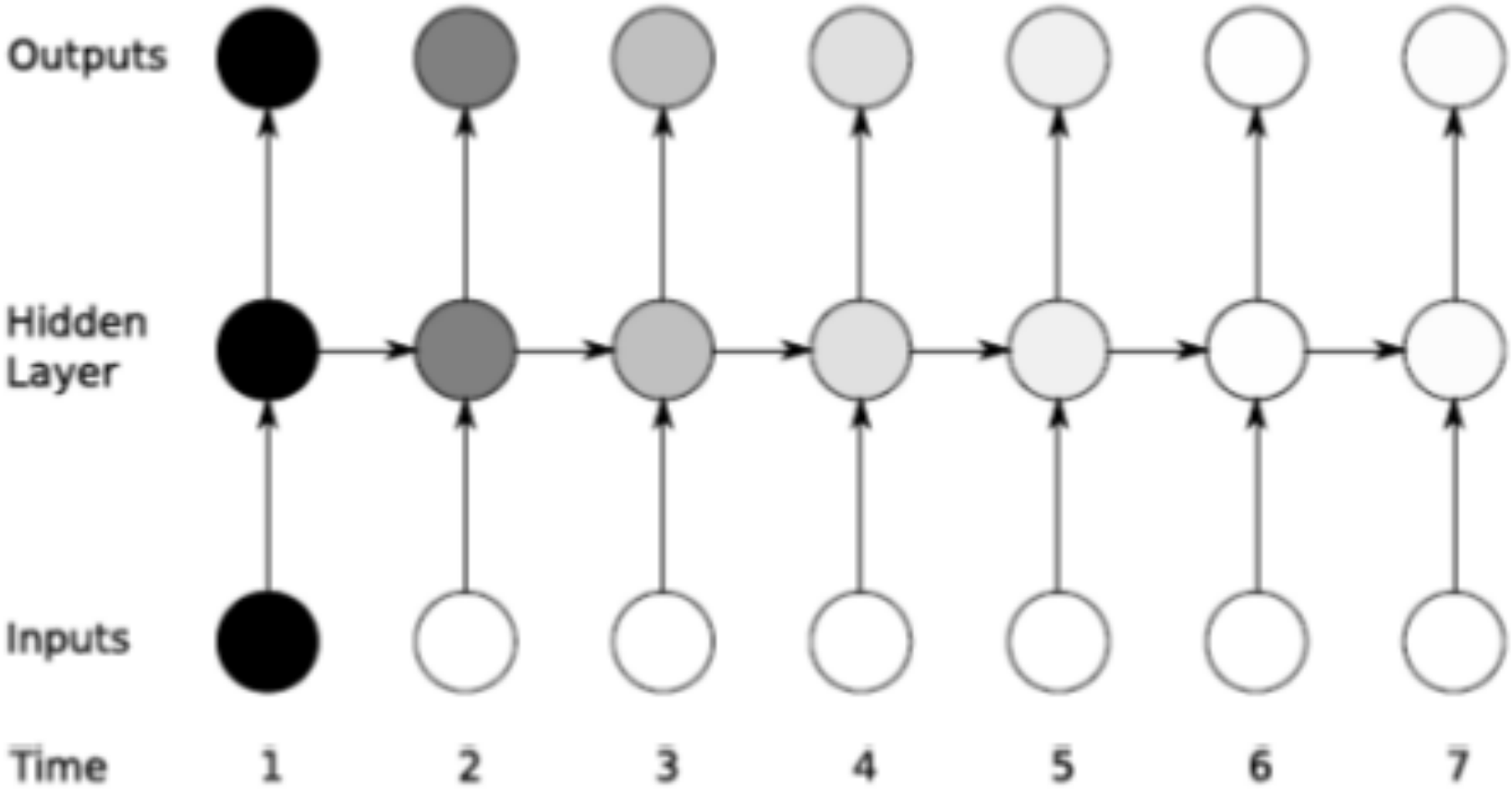
LSTM – memory cell

- i: input gate, how much of the new information will be let through the memory cell.
- f: forget gate, responsible for information should be thrown away from memory cell.
- o: output gate, how much of the information will be passed to expose to the next time step.
- g: self-recurrent which is equal to standard RNN
- c_t: internal memory of the memory cell
- s_t: hidden state
- y: final output

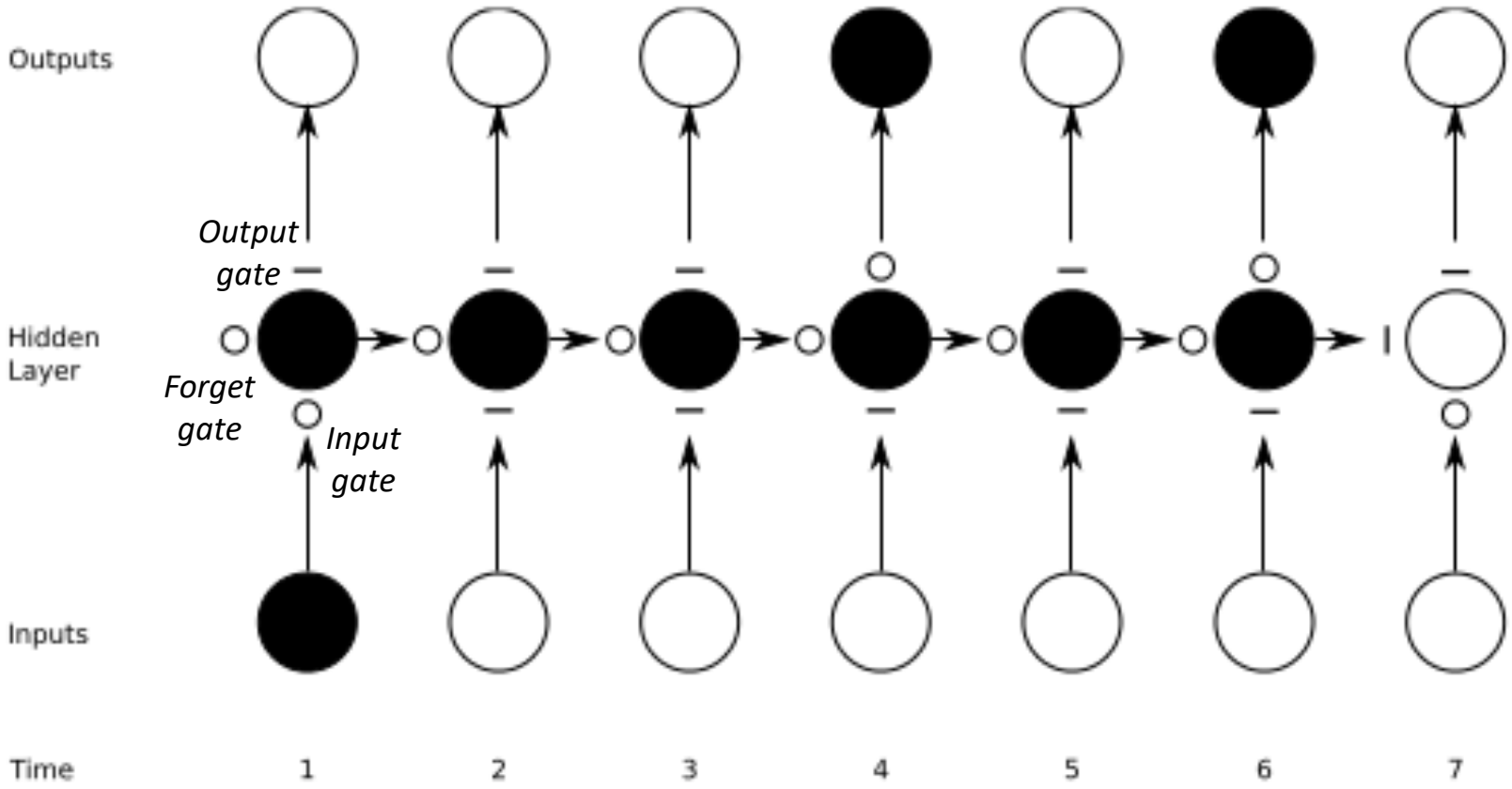


- $i = \sigma(x_t U^i + s_{t-1} W^i)$
- $f = \sigma(x_t U^f + s_{t-1} W^f)$
- $o = \sigma(x_t U^o + s_{t-1} W^o)$
- $g = \tanh(x_t U^g + s_{t-1} W^g)$
- $c_t = c_{t-1} \circ f + g \circ i$
- $s_t = \tanh(c_t) \circ o$
- $y = softmax(V s_t)$

RNN – vanishing information



LSTM – preserving information



- -> closed gate o -> open gate

LSTM – preserving information

- Traditional RNNs are a special case of LSTMs:
  - Set the input gate to all ones (passing all new information)
  - Set the forget gate to all zeros (forgetting all of the previous memory)
  - Set the output gate to all ones (exposing the entire memory).

LSTM Backward Pass

- Errors arriving at cell outputs are propagated to the CEC
- Errors can stay for a long time inside the CEC
- This ensures non-decaying error, solving the vanishing gradient problem

LSTM conclusions

- RNNs - self connected networks
- Vanishing gradients and long memory problems
- LSTM - solves the vanishing gradient and the long memory limitation problem
- LSTM can learn sequences with more than 1000 time steps.