Recurrent Neural Networks

Dr. Xiaowei Huang

https://cgi.csc.liv.ac.uk/~xiaowei/

Up to now,

- Overview of Machine Learning
- Traditional Machine Learning Algorithms
- Deep learning
  - Introduction to Tensorflow
  - Introduction to Deep Learning
  - Functional view and features
  - Backward and forward computation
  - Convolutional neural networks (various layers, regularization, etc)

Topics

- Sequential data
- Computational graph
- Recurrent neural networks (RNN)
- Example: LSTM
- CNN + RNN
- Training RNN
- Some other variants of RNNs

Sequential Data

Recurrent neural networks

- Dates back to (Rumelhart *et al.*, 1986)
- A family of neural networks for handling sequential data, which involves variable length inputs or outputs

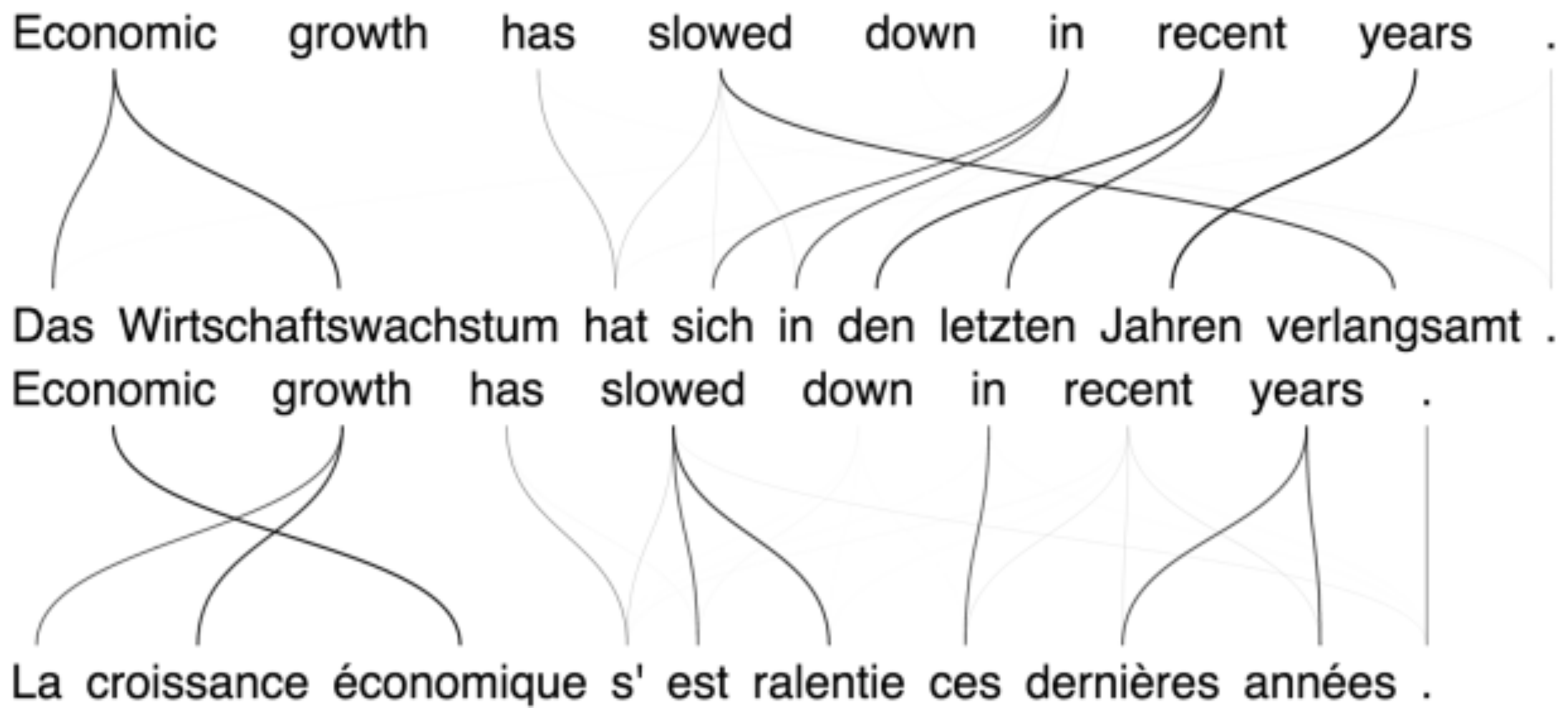
- Especially, for natural language processing (NLP) and video recognition

Sequential data

- Each data point: A sequence of vectors $x(t)$, for $1 \leq t \leq \tau$
- Batch data: many sequences with different lengths τ
- Label: can be a scalar, a vector, or even a sequence

- Example
  - Sentiment analysis
  - Machine translation
  - Video content analysis

Example: machine translation

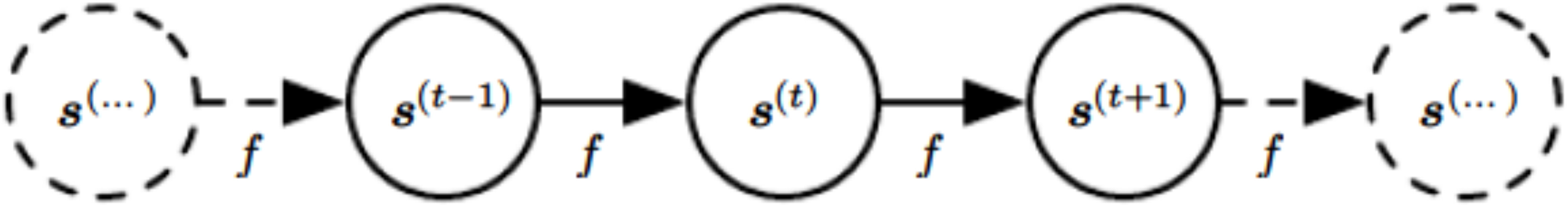


Economic growth has slowed down in recent years .
Das Wirtschaftswachstum hat sich in den letzten Jahren verlangsamt .
Economic growth has slowed down in recent years .
La croissance économique s' est ralentie ces dernières années .

Example: Video Stream

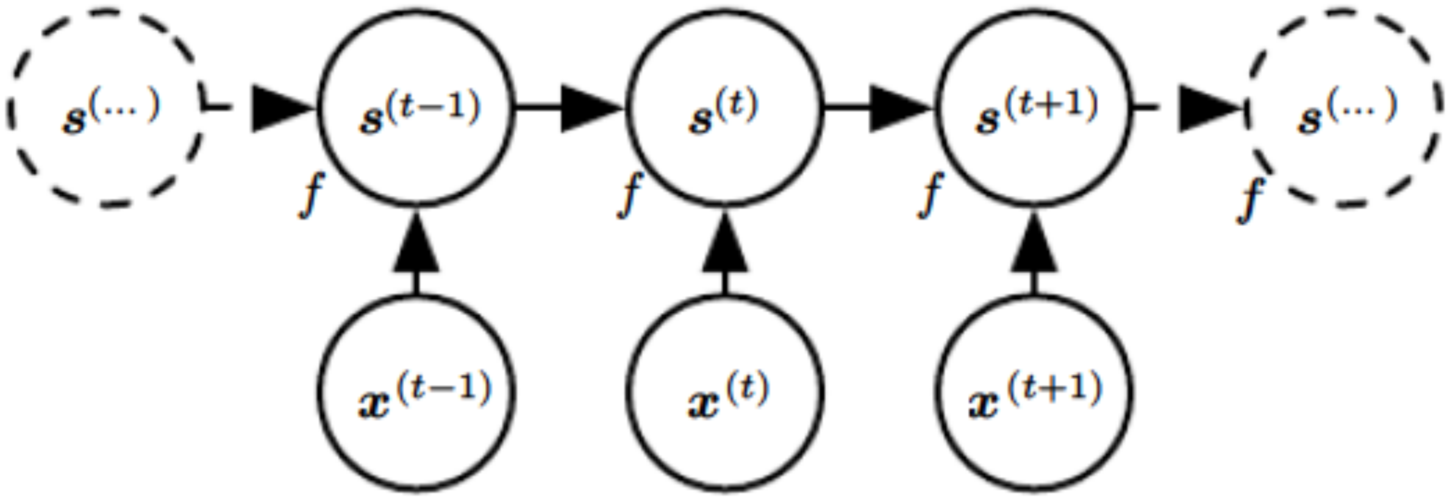
Computational graphs

A typical dynamic system



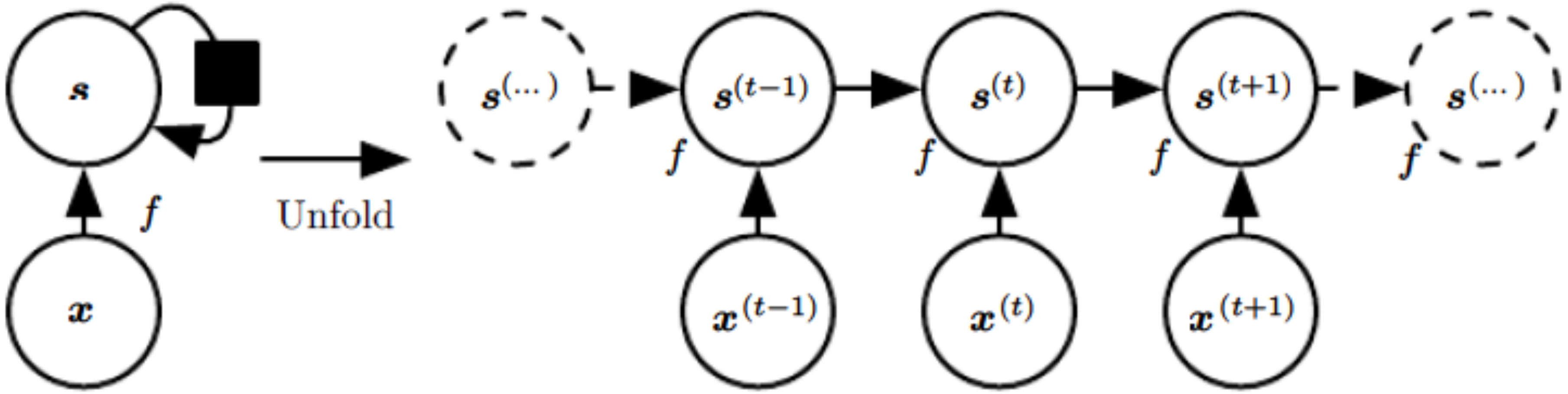
$$s^{(t+1)} = f(s^{(t)}\,;\,\theta)$$

A system driven by external data



$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

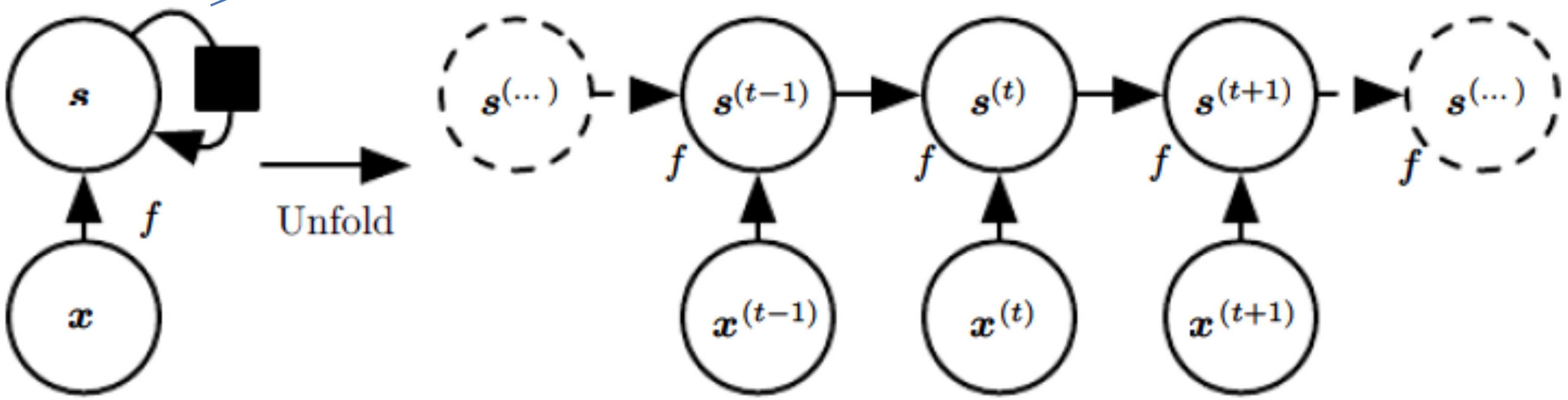
Compact view



$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

Compact view

square: one step time delay



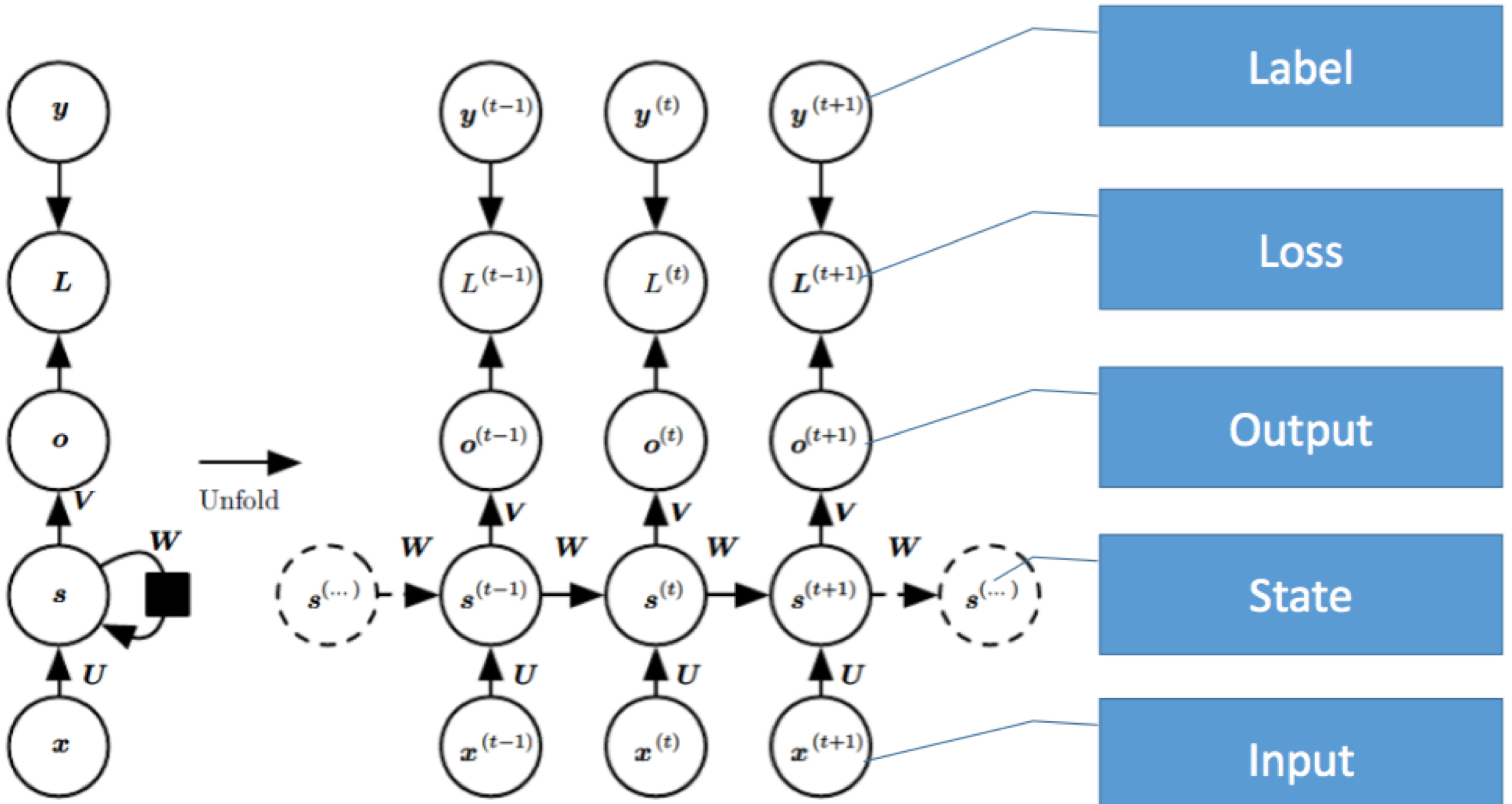
$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

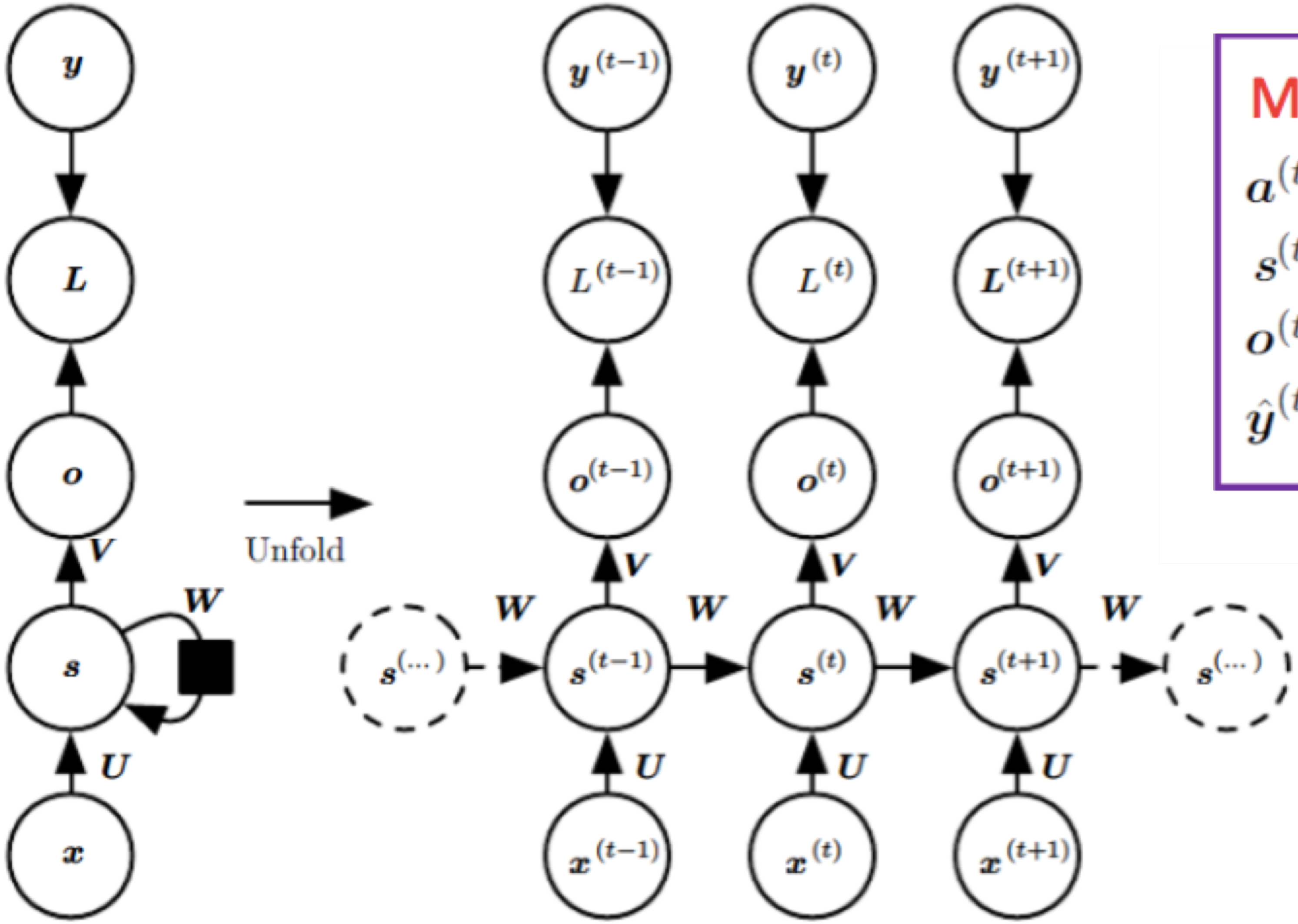
Key: the same f and θ for all time steps

Recurrent neural networks (RNN)

Recurrent neural networks

- Use the same computational function and parameters across different time steps of the sequence
- Each time step: takes the input entry and the previous hidden state to compute the output entry
- Loss: typically computed at every time step


y
L
o
V
W
s
U
x
Unfold
$y^{(t-1)}$
$y^{(t)}$
$y^{(t+1)}$
$L^{(t-1)}$
$L^{(t)}$
$L^{(t+1)}$
$o^{(t-1)}$
$o^{(t)}$
$o^{(t+1)}$
$s^{(\dots)}$
$s^{(t-1)}$
$s^{(t)}$
$s^{(t+1)}$
$x^{(t-1)}$
$x^{(t)}$
$x^{(t+1)}$
Label
Loss
Output
State
Input



Math formula:

$$\begin{aligned} \boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{s}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\ \boldsymbol{s}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\ \boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{s}^{(t)} \\ \hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)}) \end{aligned}$$

Advantage

- Hidden state: a lossy summary of the past
- Shared functions and parameters: greatly reduce the capacity and good for generalization in learning
- Explicitly use the prior knowledge that the sequential data can be processed in the same way at different time step (e.g., NLP)

Advantage

- Hidden state: a lossy summary of the past
- Shared functions and parameters: greatly reduce the capacity and good for generalization in learning
- Explicitly use the prior knowledge that the sequential data can be processed by in the same way at different time step (e.g., NLP)

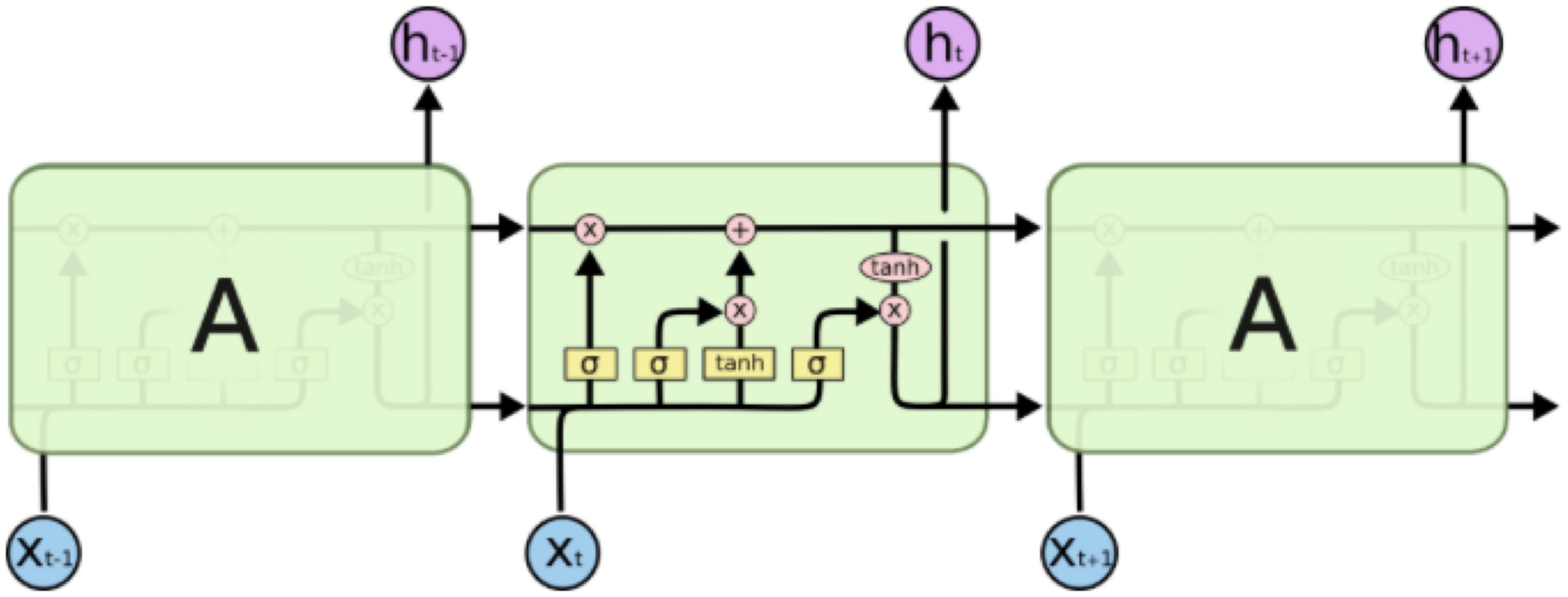
- Yet still powerful (actually universal): any function computable by a Turing machine can be computed by such a recurrent network of a finite size (see, e.g., Siegelmann and Sontag (1995))

The problem of exploding/vanishing gradient

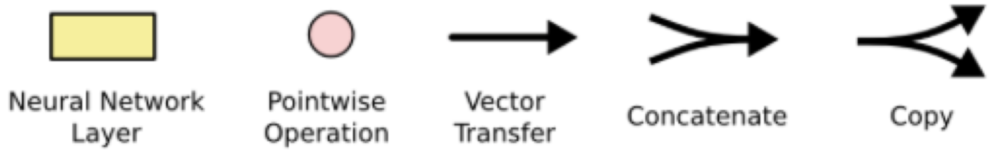
- What happens to the magnitude of the gradients as we backpropagate through many layers?
  - If the weights are small, the gradients shrink exponentially.
  - If the weights are big the gradients grow exponentially.
- Typical feed-forward neural nets can cope with these exponential effects because they only have a few hidden layers.
- In an RNN trained on long sequences (*e.g.* 100 time steps) the gradients can easily explode or vanish.
  - We can avoid this by initializing the weights very carefully.
- 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.

Example: LSTM

General Diagram



The repeating module in an LSTM contains four interacting layers.



each line carries an entire vector, from the output of one node to the inputs of others

pink circles represent pointwise operations, like vector addition

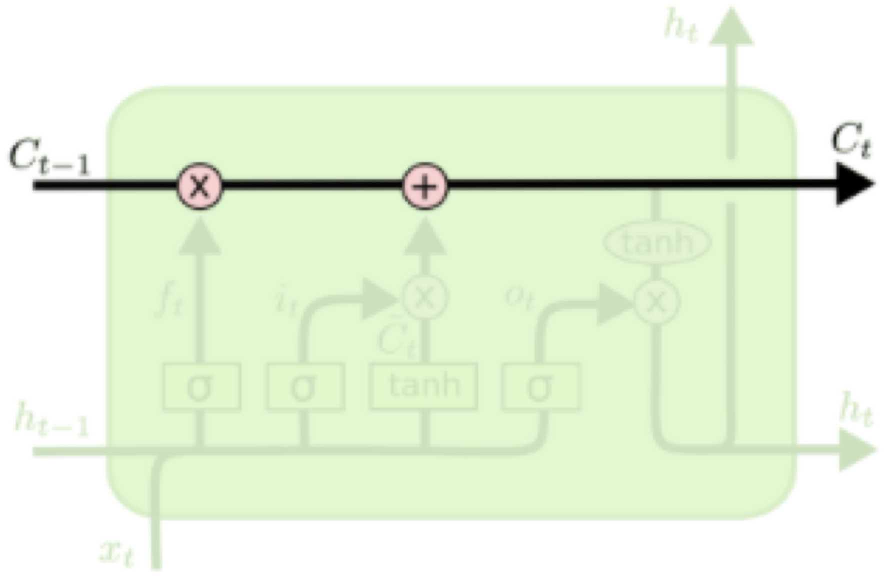
yellow boxes are learned neural network layers

Lines merging denote concatenation

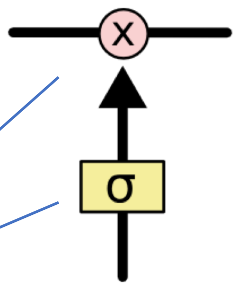
line forking denote its content being copied and the copies going to different locations

The Core Idea Behind LSTMs

- The key to LSTMs is the cell state, the horizontal line running through the top of the diagram.
- The cell state is kind of like a conveyor belt. It runs straight down the entire chain, with only some minor linear interactions. It's very easy for information to just flow along it unchanged.



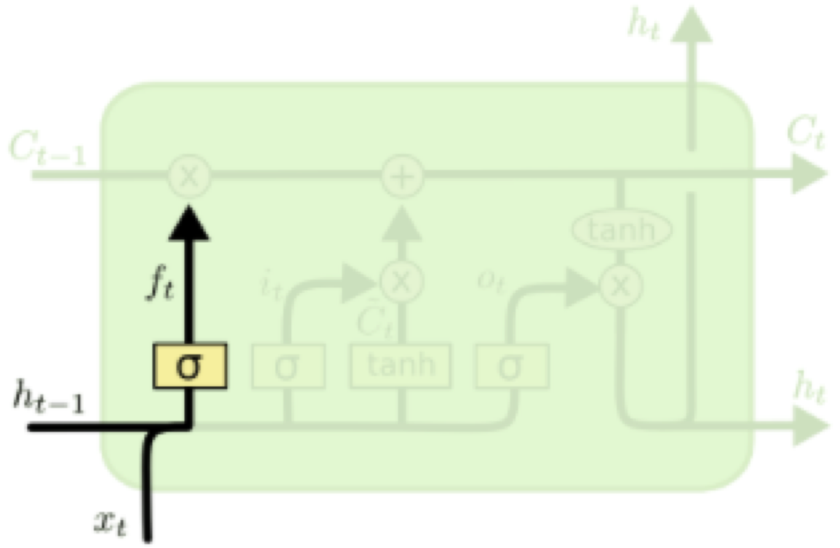
The Core Idea Behind LSTMs



- The LSTM does have the ability to remove or add information to the cell state, carefully regulated by structures called gates.
- Gates are a way to optionally let information through. They are composed out of
  - a sigmoid neural net layer and
  - a pointwise multiplication operation.
- 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,”
  - A value of one means “let everything through!”
- An LSTM has three of these gates, to protect and control the cell state.

forget gate layer

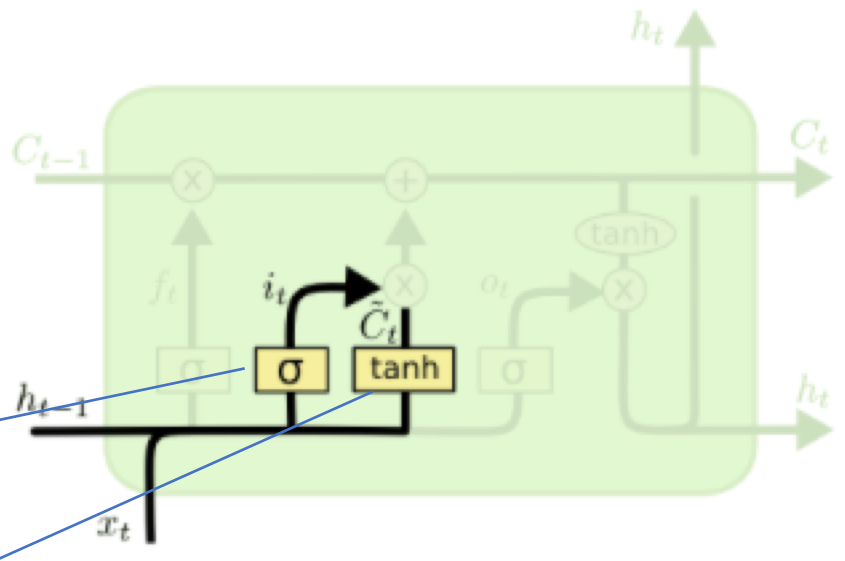
- The first step in our LSTM is to decide what information we’re going to throw away from the cell state.
  - 1 represents “completely keep this”
  - 0 represents “completely get rid of this.”



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

input gate layer

- decide what new information we’re going to store in the cell state.
  - First, a sigmoid layer called the “input gate layer” decides which values we’ll update.
  - Next, a tanh layer creates a vector of new candidate values that could be added to the state.
- In the next step, we’ll combine these two to create an update to the state.

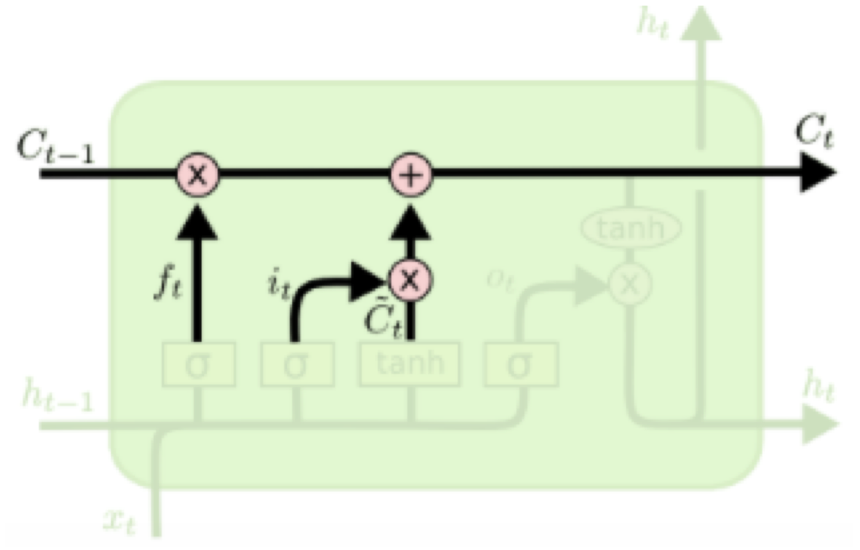


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

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

update cell state C_{t-1} into C_t

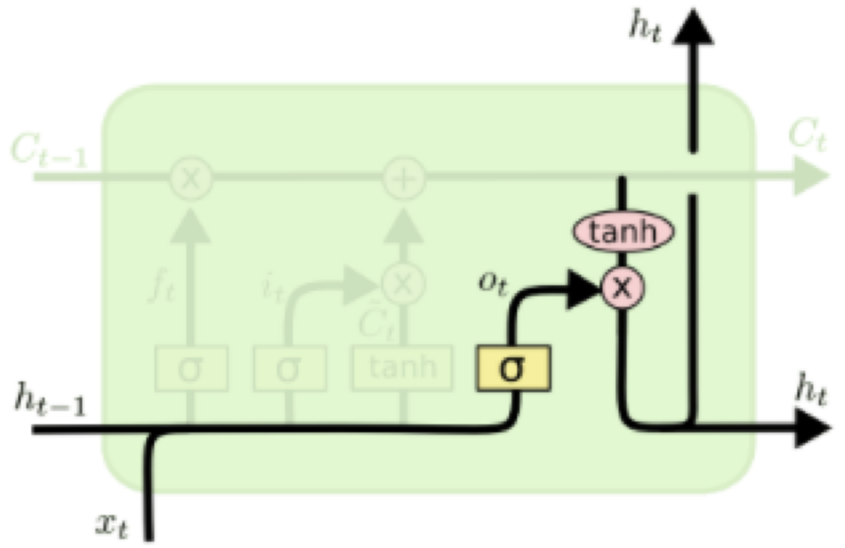
- We multiply the old state by f_t, forgetting the things we decided to forget earlier.
- Then we add $i_t * \hat{C}_t$. This is the new candidate values, scaled by how much we decided to update each state value.



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

decide what we’re going to output

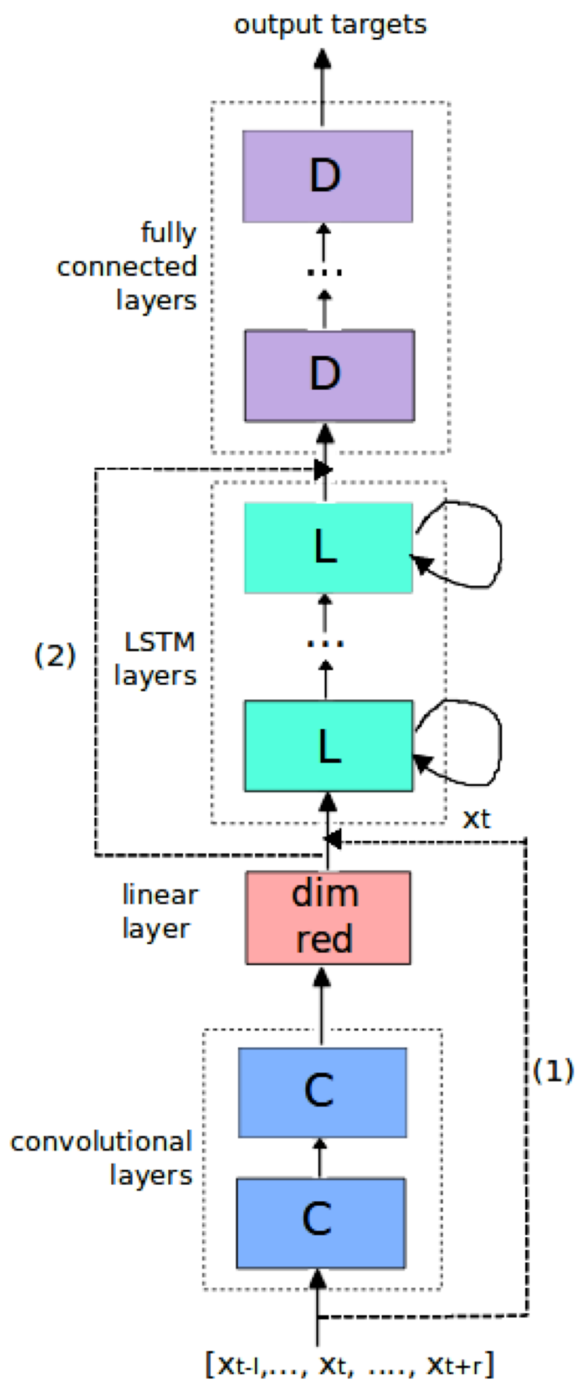
- First, we run a sigmoid layer which decides what parts of the cell state we’re going to output.
- Then, we put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.



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

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

CNN + RNN



LSTMs are used in modelling tasks related to sequences and do predictions based on it.

CNNs are used in modelling problems related to spatial inputs like images.

CNN + RNN

- CNN-LSTMs are generally used when their inputs have spatial structure in their input such as the 2D structure or pixels in an image or the 1D structure of words in a sentence, paragraph, or document and also have a temporal structure in their input such as the order of images in a video or words in text, or require the generation of output with temporal structure such as words in a textual description.

Training RNN

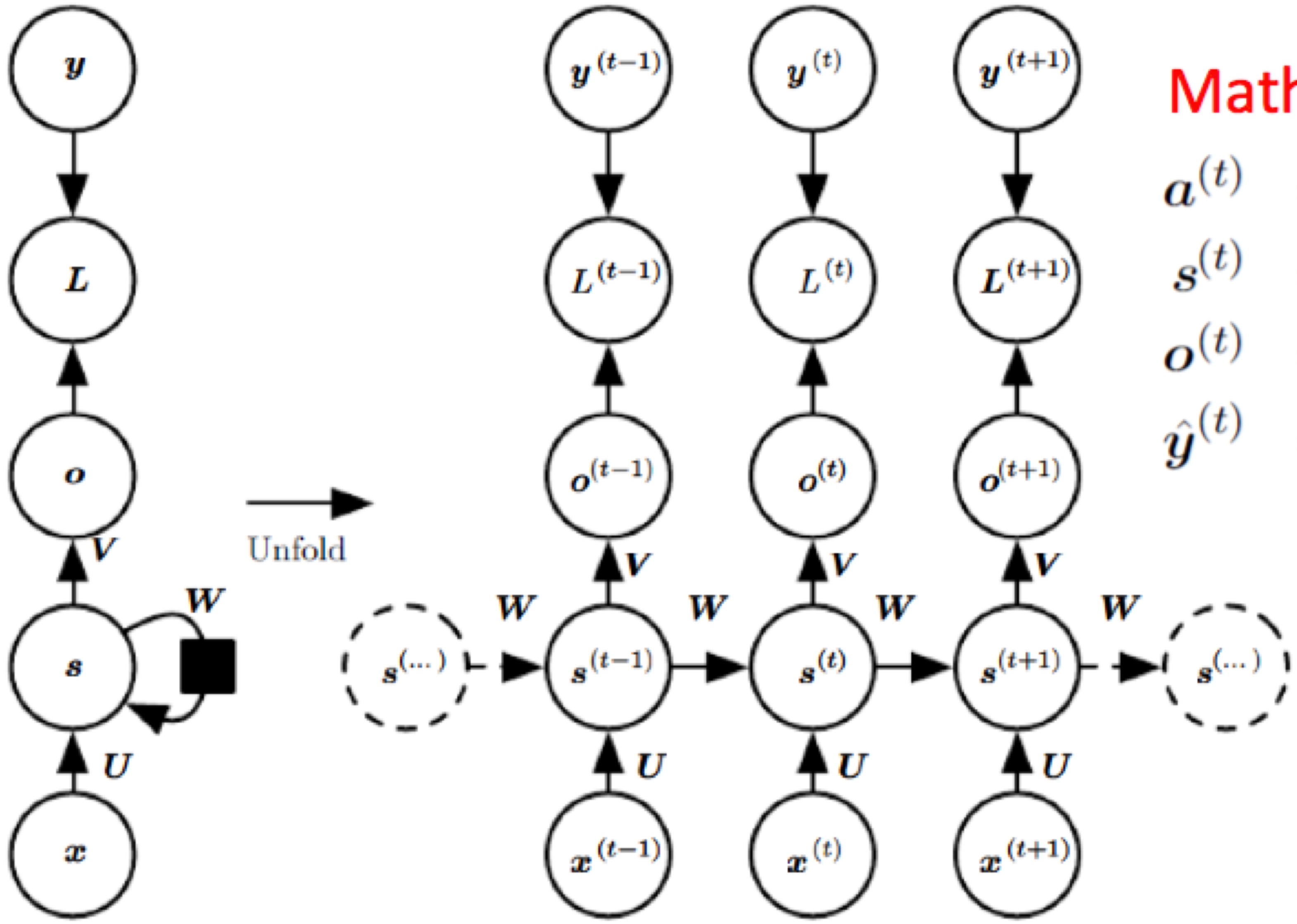
Training RNN

- Principle: unfold the computational graph, and use backpropagation
- Called back-propagation through time (BPTT) algorithm
- Can then apply any general-purpose gradient-based techniques

Training RNN

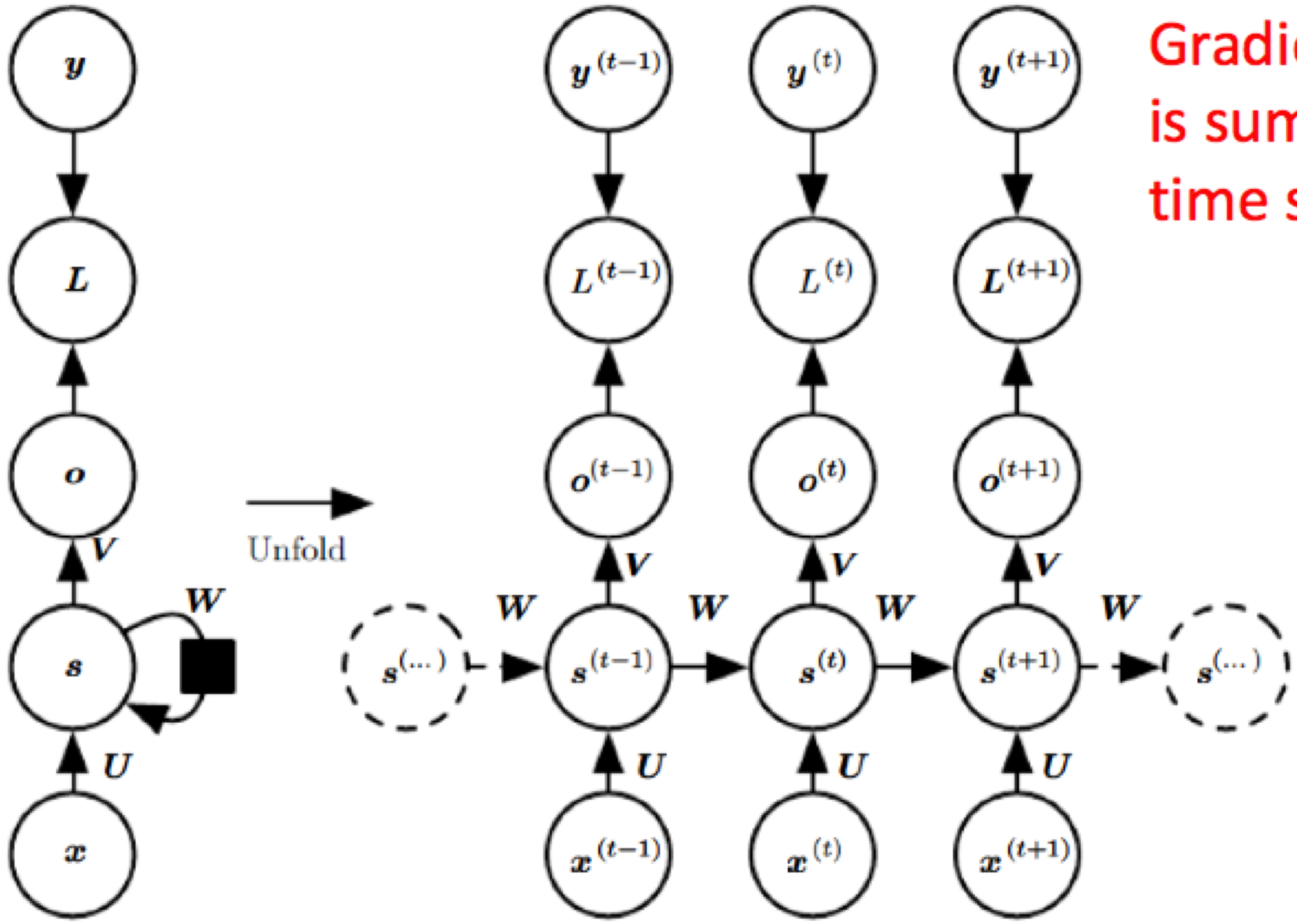
- Principle: unfold the computational graph, and use backpropagation
- Called back-propagation through time (BPTT) algorithm
- Can then apply any general-purpose gradient-based techniques

- Conceptually: first compute the gradients of the internal nodes, then compute the gradients of the parameters



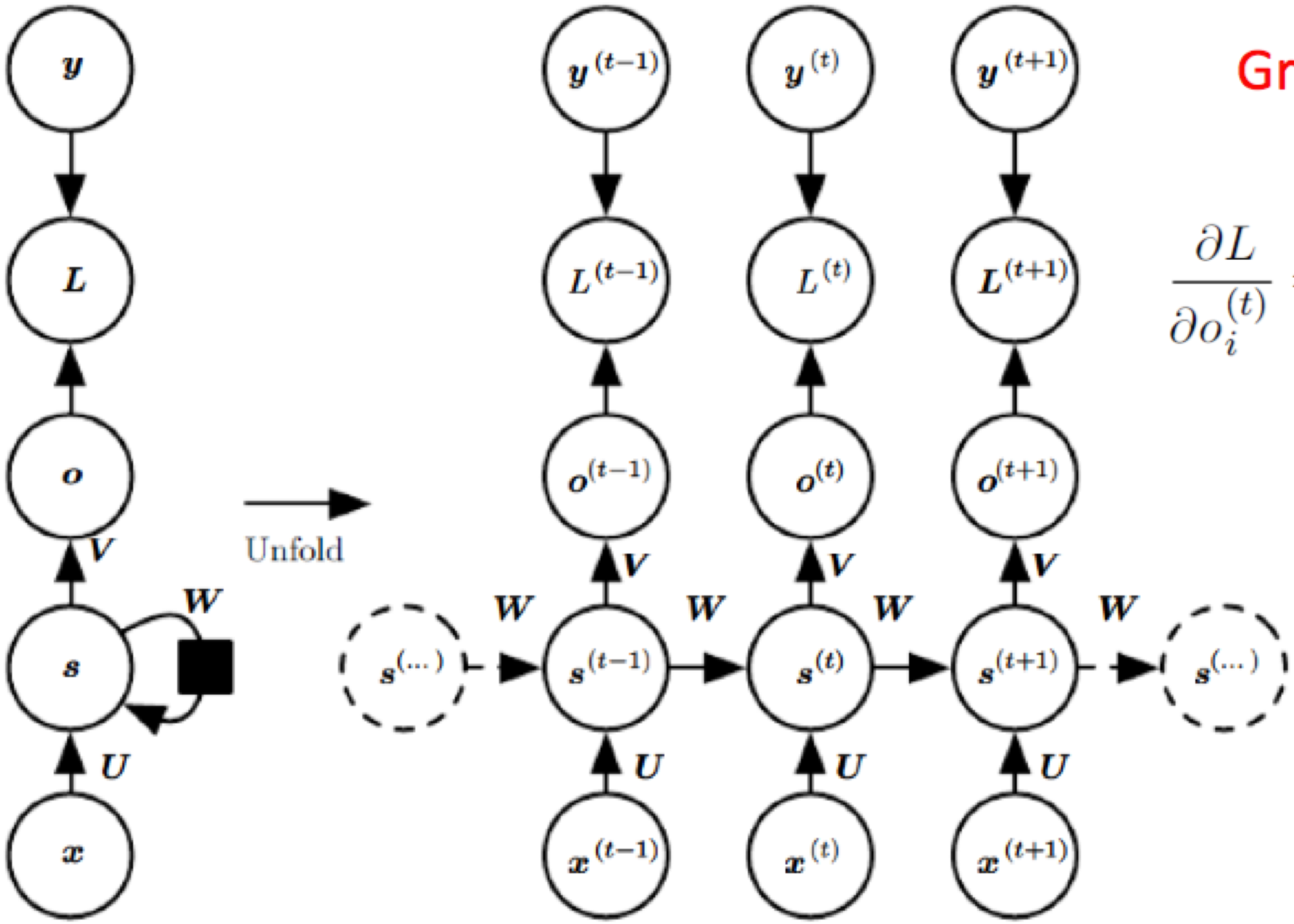
Math formula:

$$
\begin{aligned}
\boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{s}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\
\boldsymbol{s}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\
\boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{s}^{(t)} \\
\hat{\boldsymbol{y}}^{(t)} &= \operatorname{softmax}(\boldsymbol{o}^{(t)})
\end{aligned}
$$



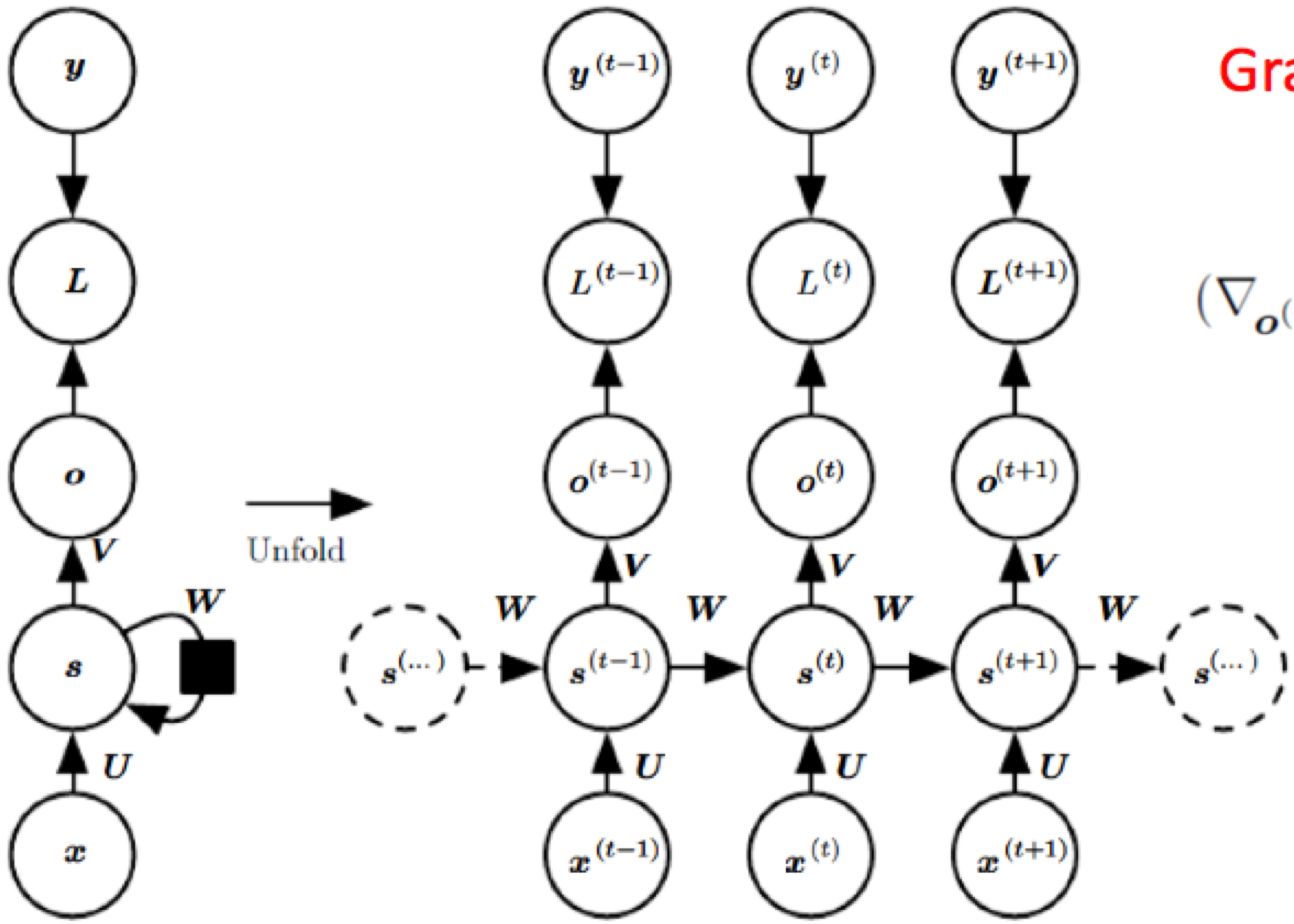
Gradient at $L^{(t)}$: (total loss is sum of those at different time steps)

$$\frac{\partial L}{\partial L^{(t)}} = 1.$$



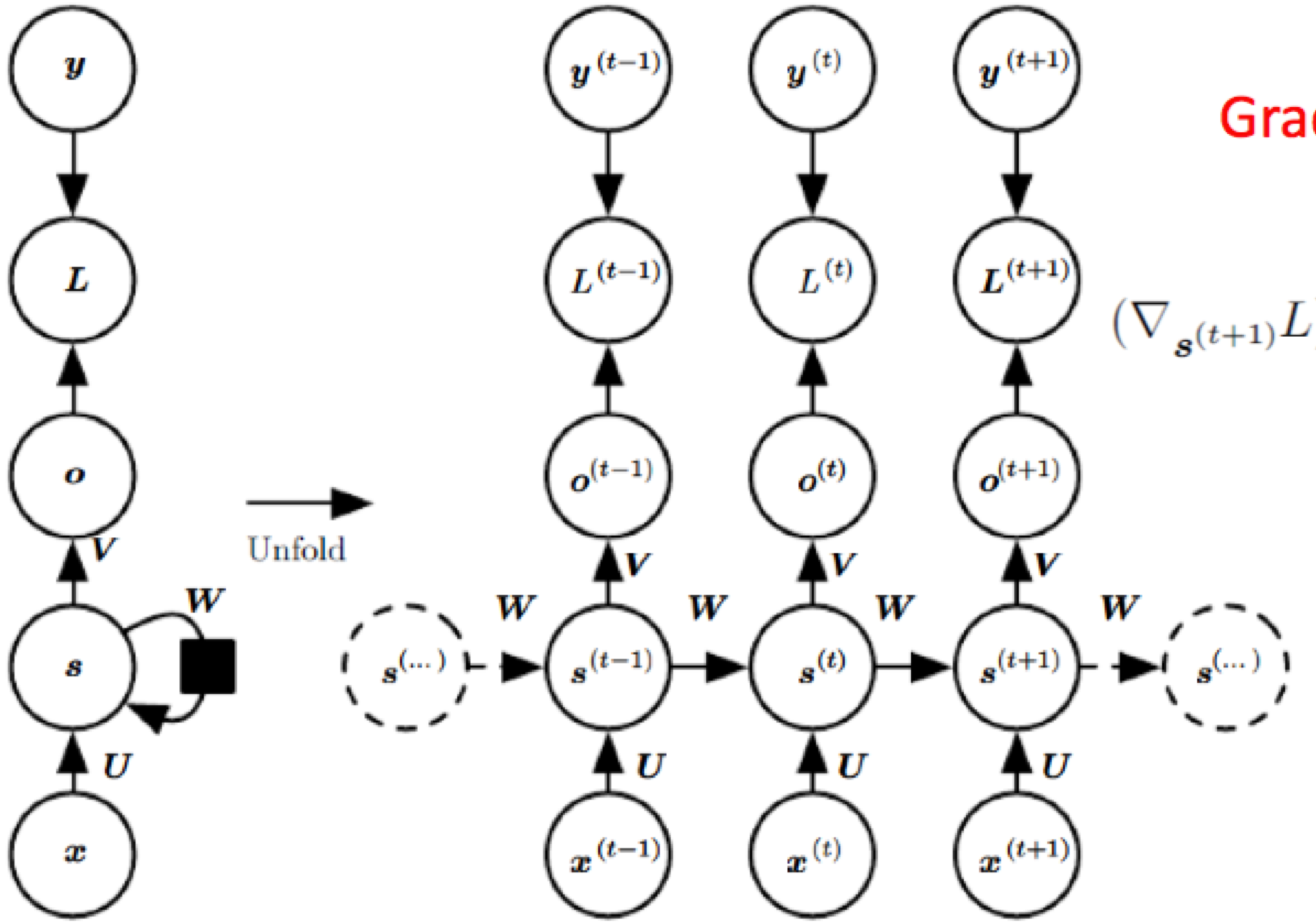
Gradient at $o^{(t)}$:

$$\frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i,y^{(t)}}$$



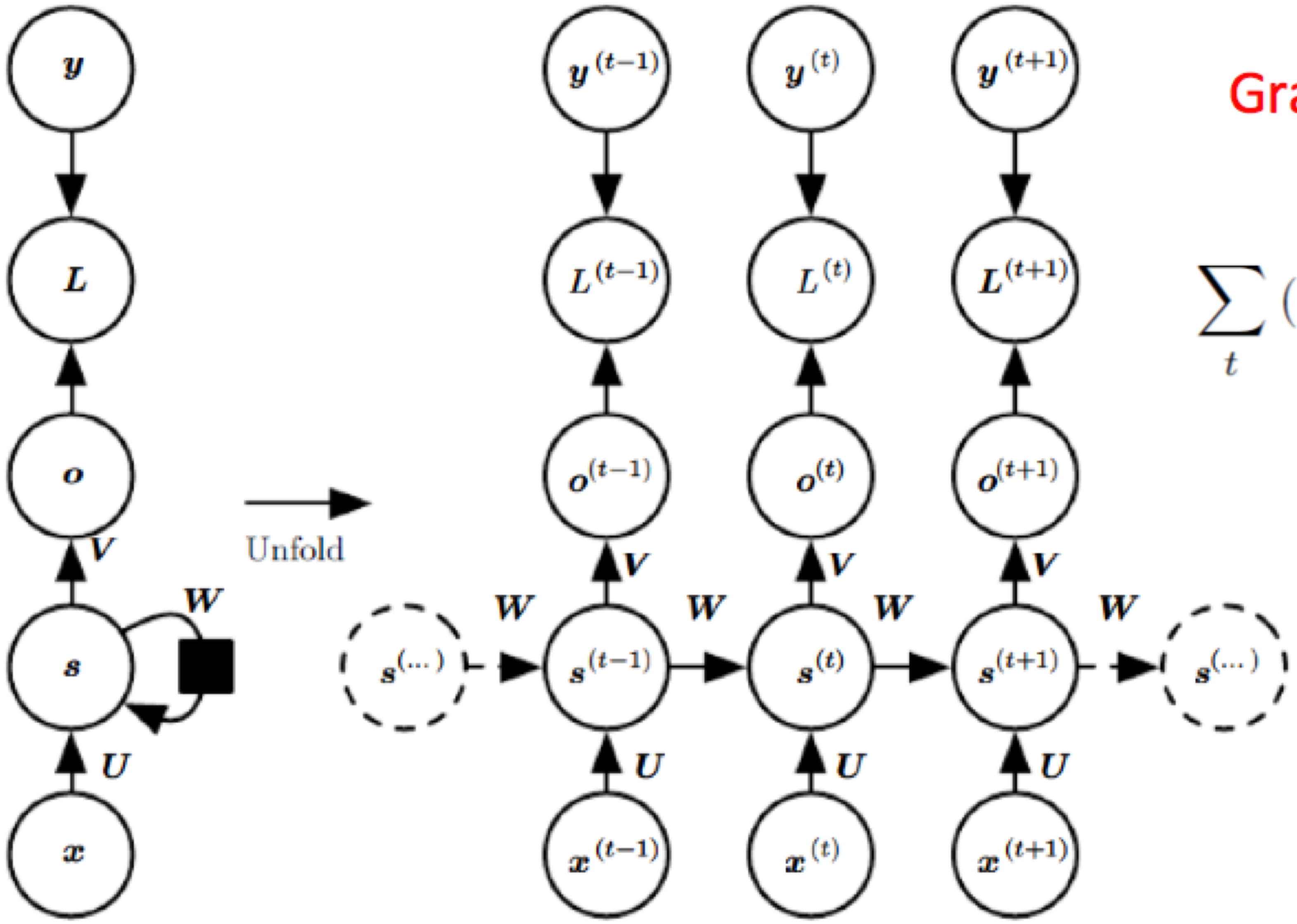
Gradient at $s^{(\tau)}$:

$$(\nabla_{\boldsymbol{o}^{(\tau)}} L) \frac{\partial \boldsymbol{o}^{(\tau)}}{\partial \boldsymbol{s}^{(\tau)}} = (\nabla_{\boldsymbol{o}^{(\tau)}} L) \boldsymbol{V}$$



Gradient at $s^{(t)}$:

$$(\nabla_{\boldsymbol{s}^{(t+1)}} L) \frac{\partial \boldsymbol{s}^{(t+1)}}{\partial \boldsymbol{s}^{(t)}} + (\nabla_{\boldsymbol{o}^{(t)}} L) \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{s}^{(t)}}$$



Gradient at parameter V:

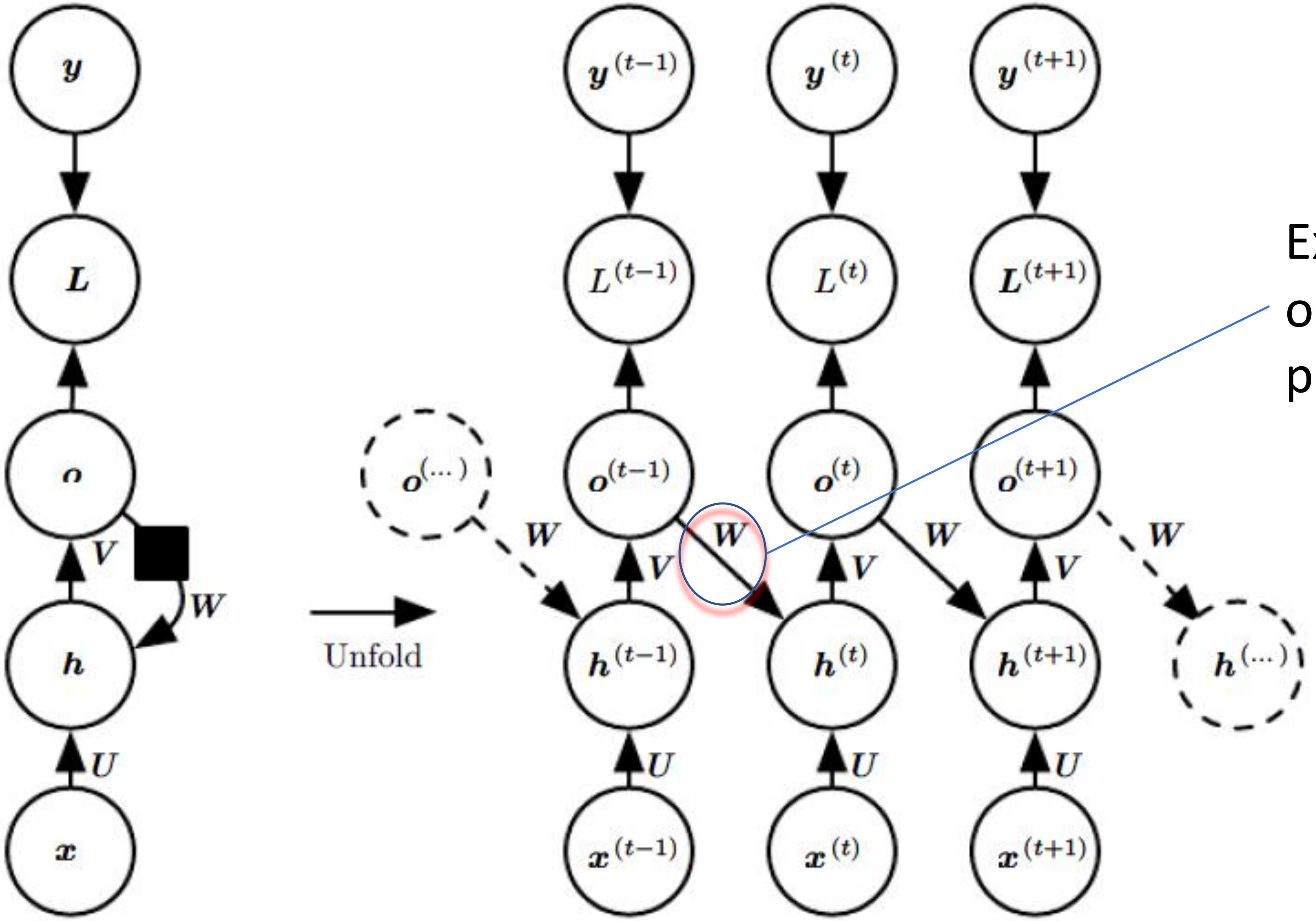
$$\sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{V}} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \boldsymbol{s}^{(t)\top}$$

Some Other Variants of RNN

RNN

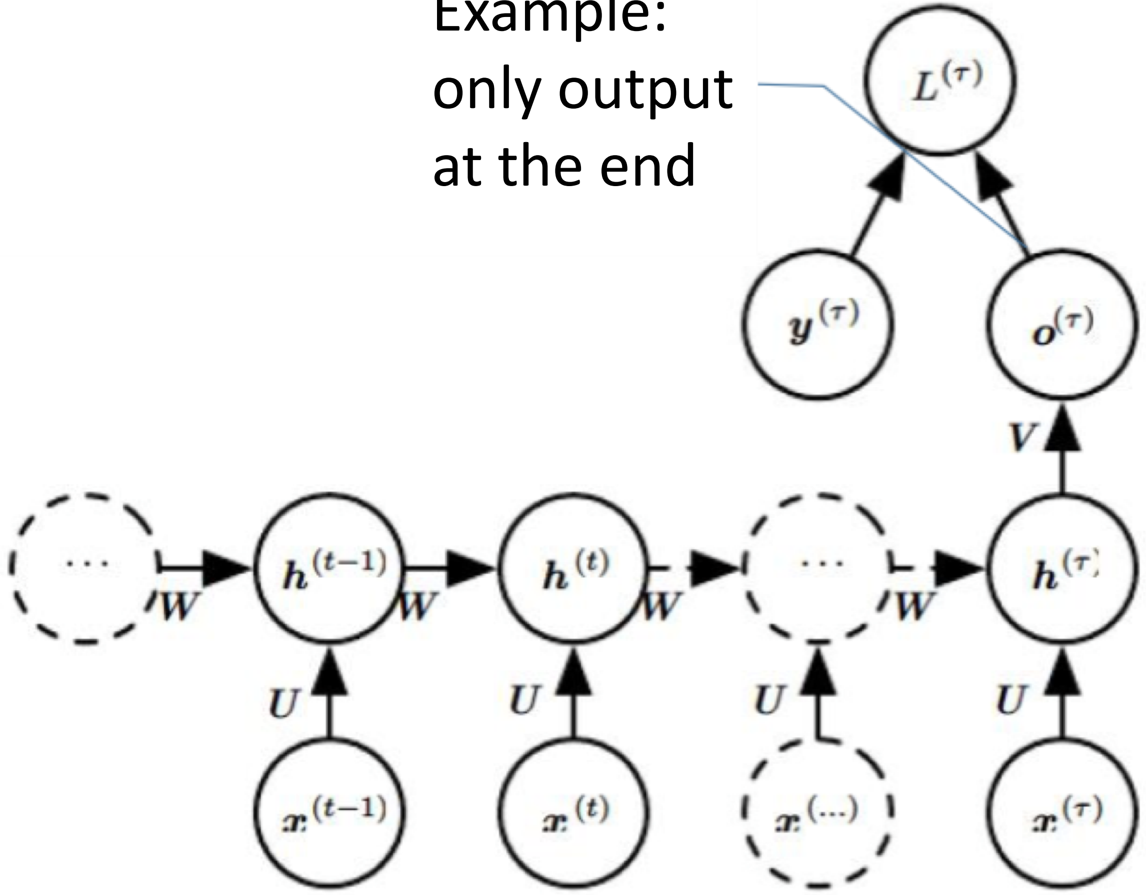
- Use the same computational function and parameters across different time steps of the sequence
- Each time step: takes the input entry and the previous hidden state to compute the output entry
- Loss: typically computed every time step

- Many variants
  - Information about the past can be in many other forms
  - Only output at the end of the sequence



Example: use the output at the previous step

Example:
only output
at the end

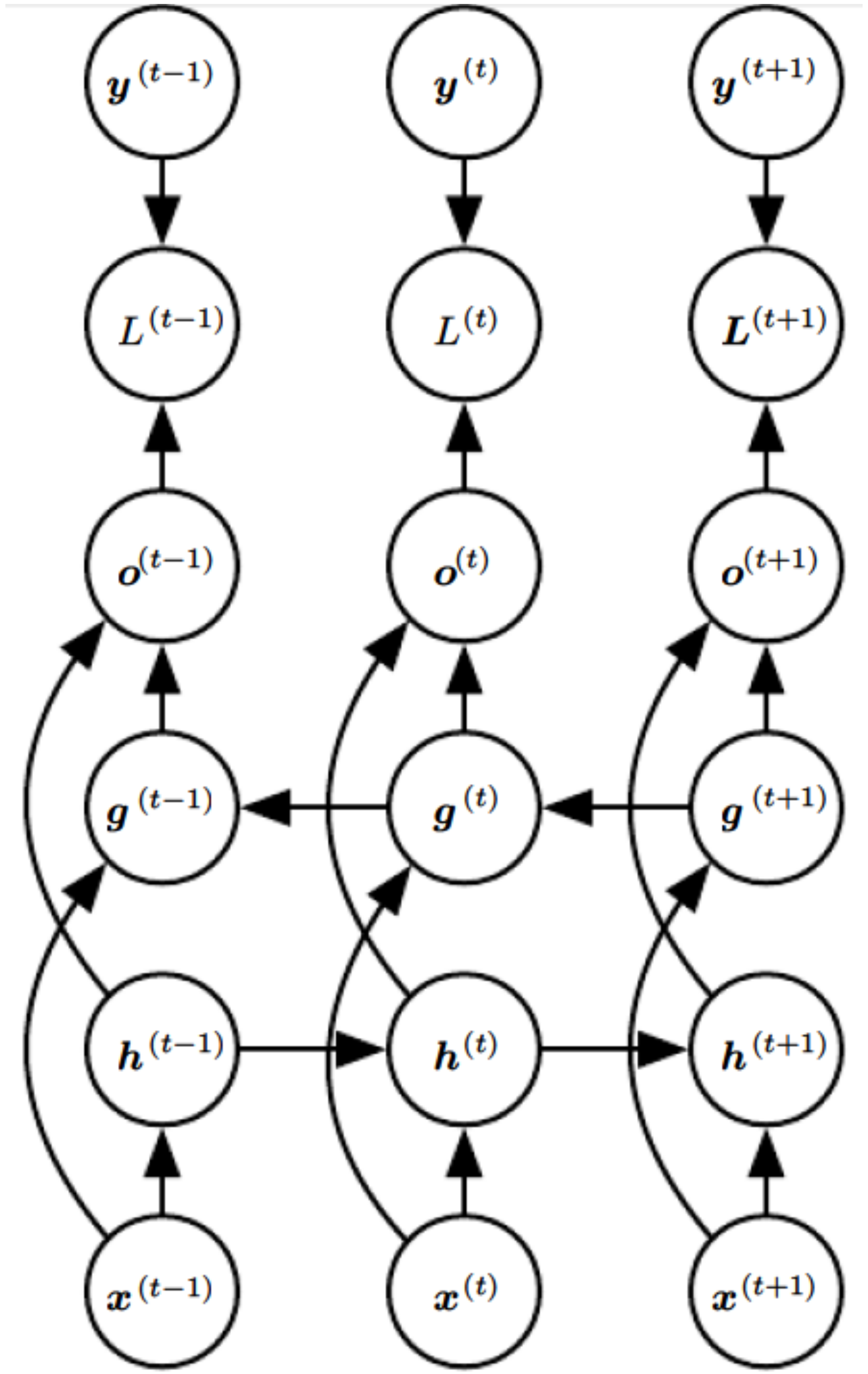


Bidirectional RNNs

- Many applications: output at time t may depend on the whole input sequence
- Example in speech recognition: correct interpretation of the current sound may depend on the next few phonemes, potentially even the next few words

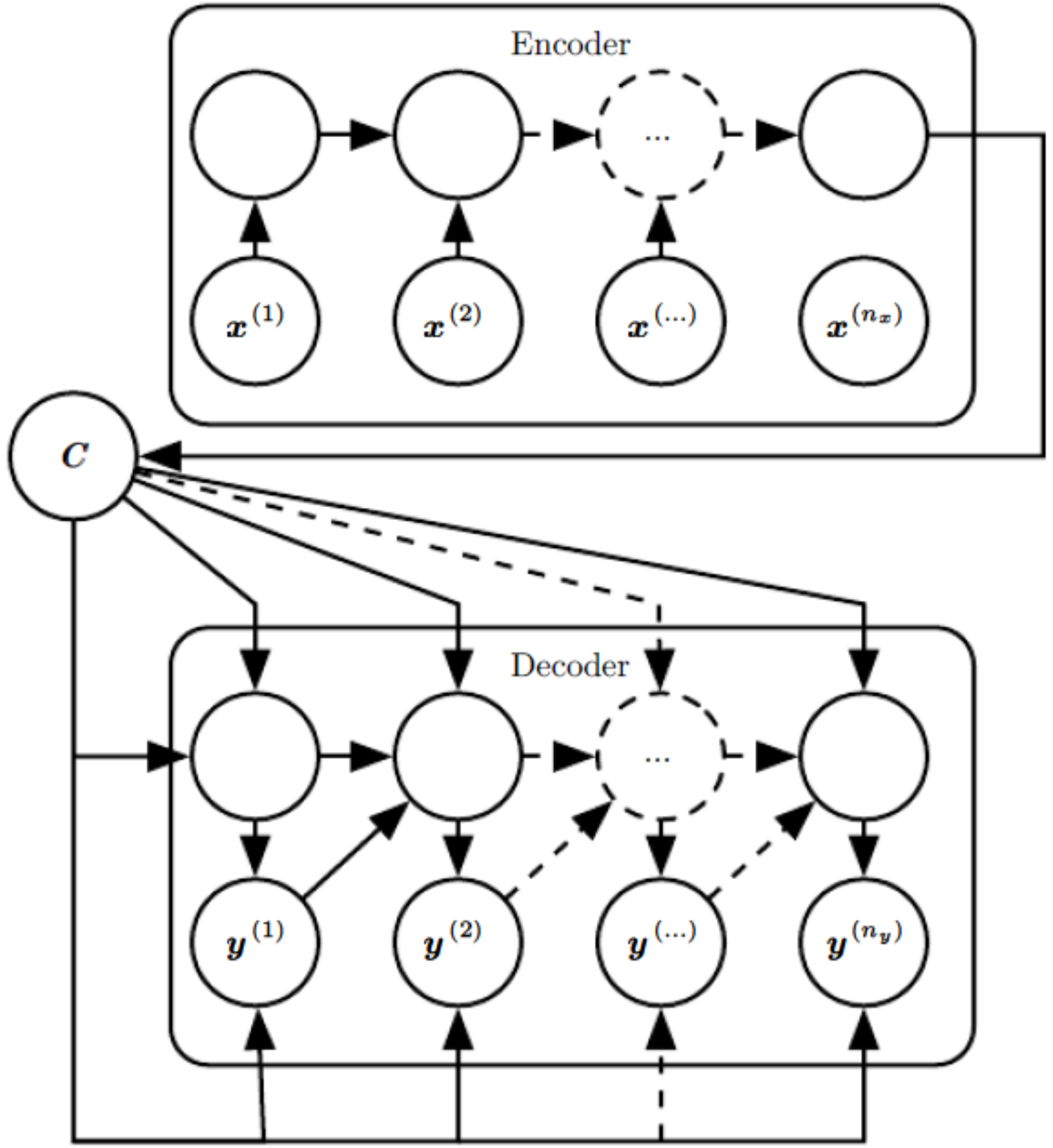
- Bidirectional RNNs are introduced to address this

BiRNNs



Encoder-decoder RNNs

- RNNs: can map sequence to one vector; or to sequence of same length
- What about mapping sequence to sequence of different length?
- Example: speech recognition, machine translation, question answering, etc


Encoder
...
$x^{(1)}$
$x^{(2)}$
$x^{(...)}$
$x^{(n_x)}$
C
Decoder
...
$y^{(1)}$
$y^{(2)}$
$y^{(...)}$
$y^{(n_y)}$