# DEEP LEARNING

## Recurrent Neural Networks

EXPORT CONTROLLED - This technology or software is subject to the U.S. Export Administration Regulations (EAR), (15 C.F.R. Parts 730-774). No authorization from the U.S. Department of Commerce is required for export, re-export, in-country transfer, or access EXCEPT to country group E:1 or E:2 countries/persons per Supp.1 to Part 740 of the EAR. ECCN: EAR99

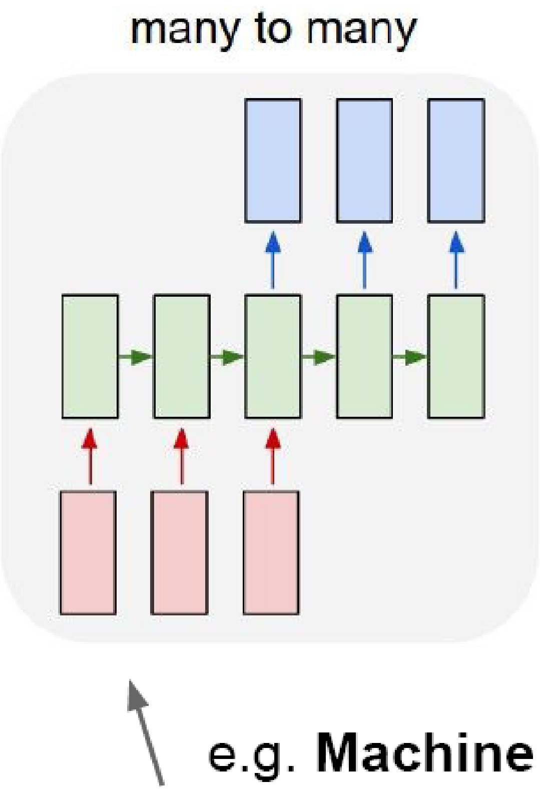
# Recurrent Neural Networks

- 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)

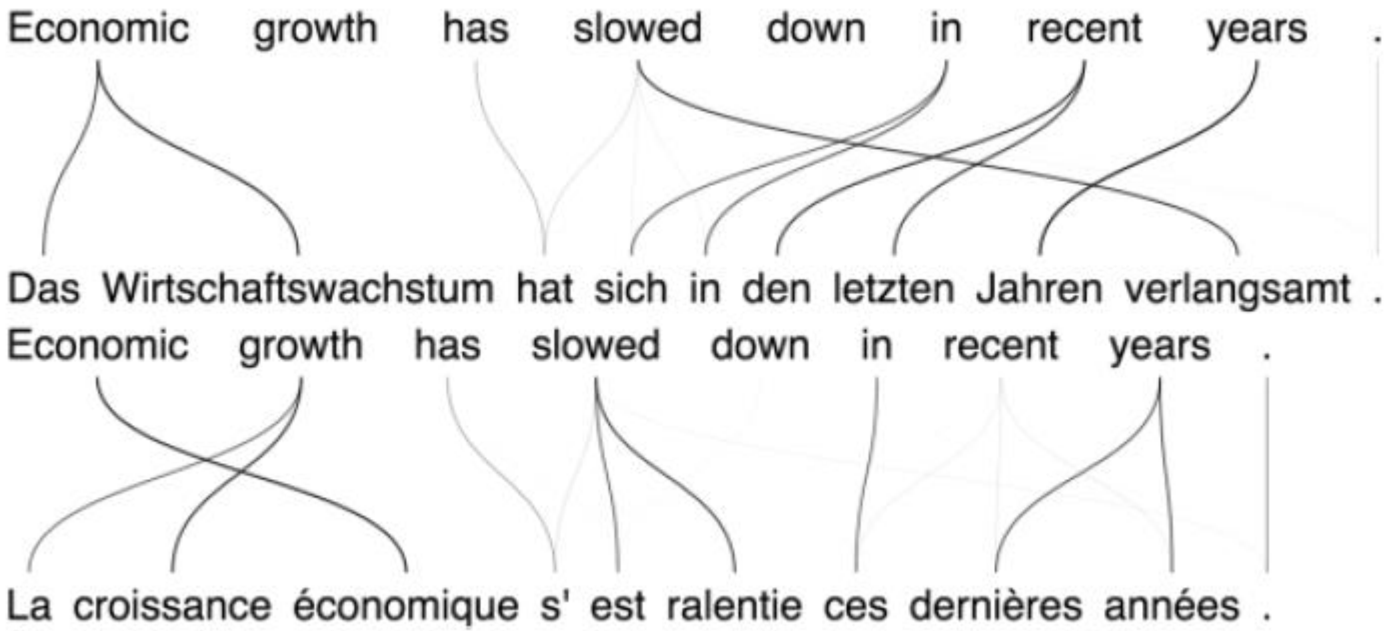
# Sequential Data

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

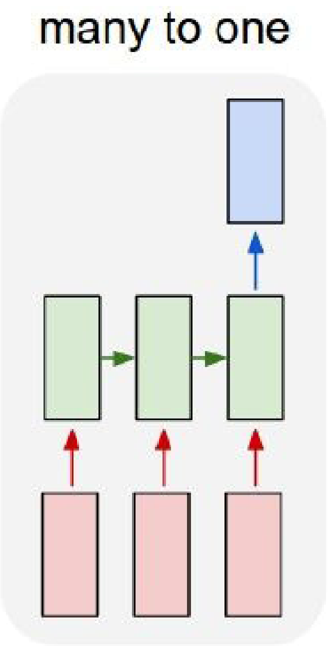
# Sequential Data: Machine Translation



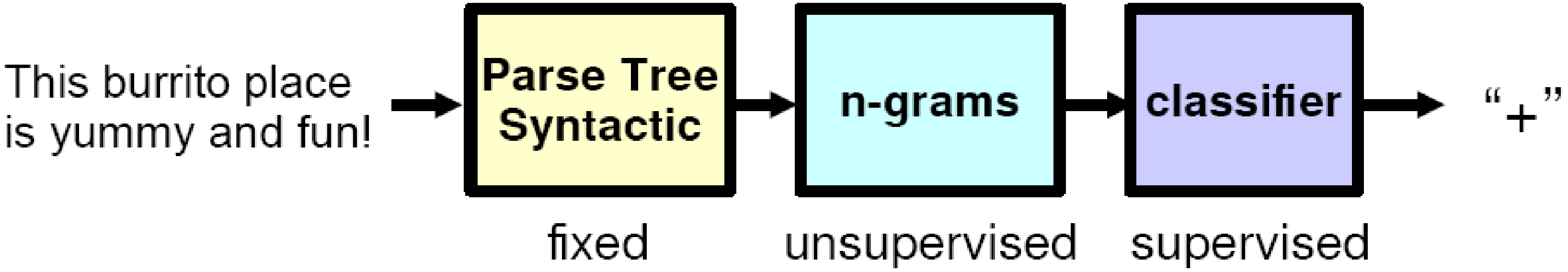
e.g. **Machine Translation**
seq of words -> seq of words


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 .

# Sequential Data: Sentiment Analysis



NLP



This burrito place is yummy and fun! → Parse Tree Syntactic → n-grams → classifier → "+"

fixed

unsupervised

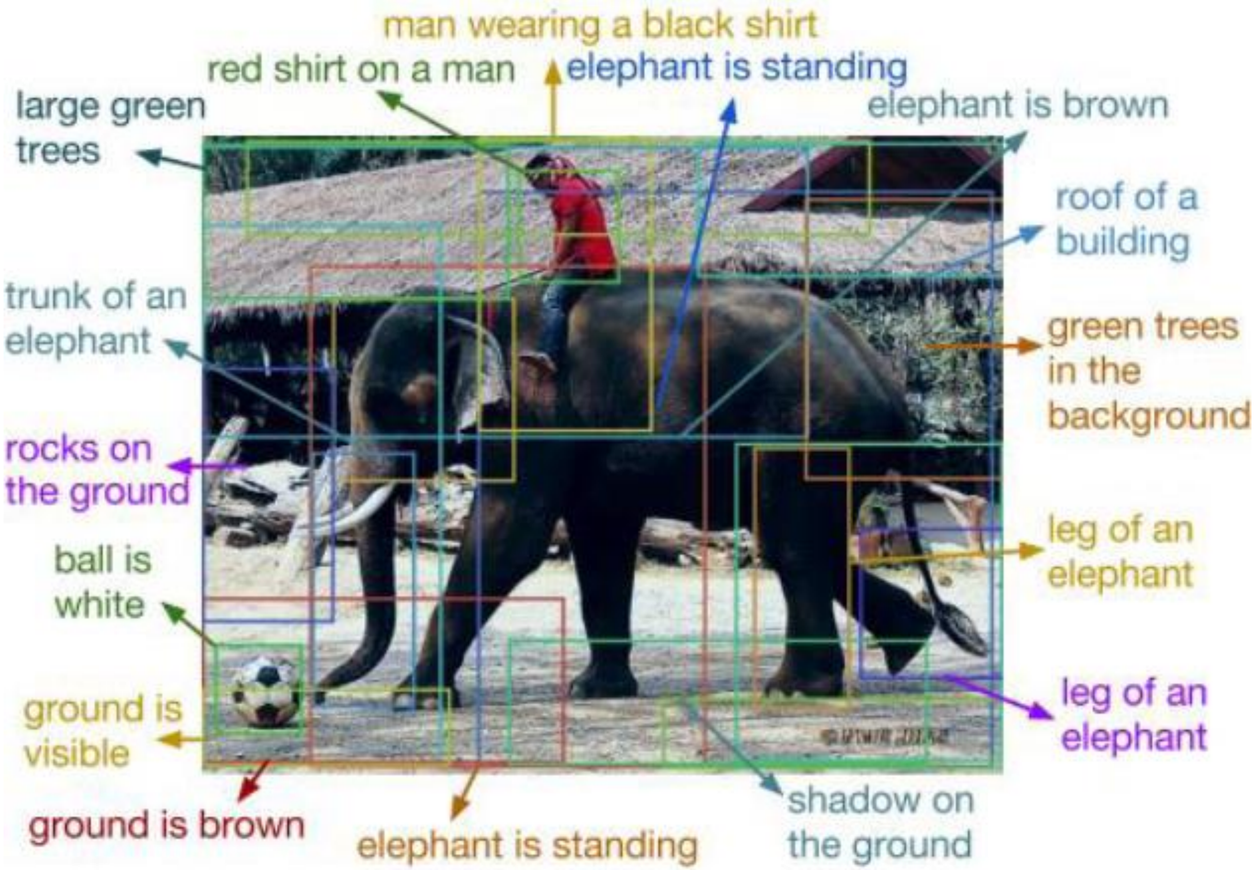
supervised

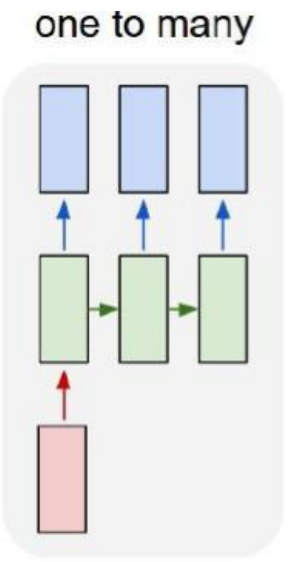
e.g. **Sentiment Classification**
sequence of words -> sentiment

# More Complicated Sequential Data

- **Data point:** two dimensional sequences like images
- **Label:** different type of sequences like text sentences
- Example: image captioning

# Image Captioning

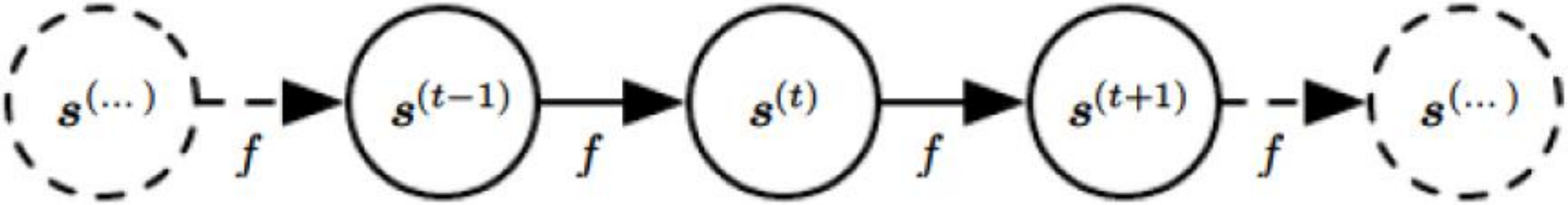




e.g. **Image Captioning**
image -> sequence of words

Image source: “DenseCap: Fully Convolutional Localization Networks for Dense Captioning,” by Justin Johnson, Andrej Karpathy, Li Fei-Fei

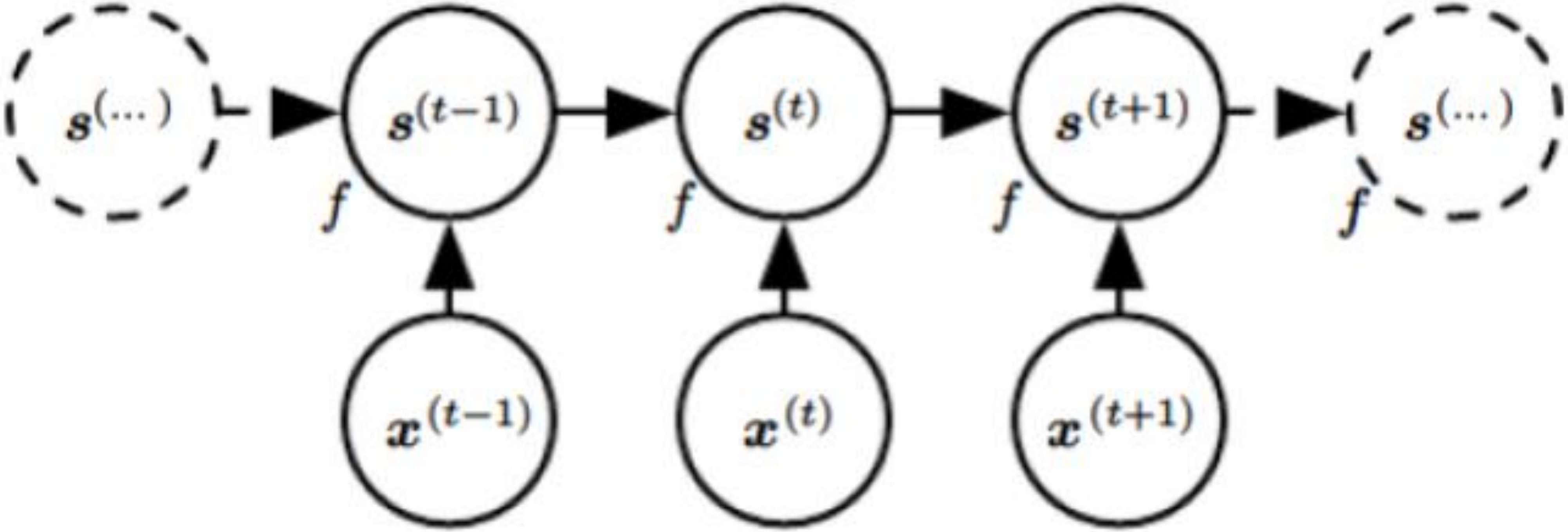
# A Typical Dynamic System



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

Figure from Deep Learning, Goodfellow, Bengio and Courville

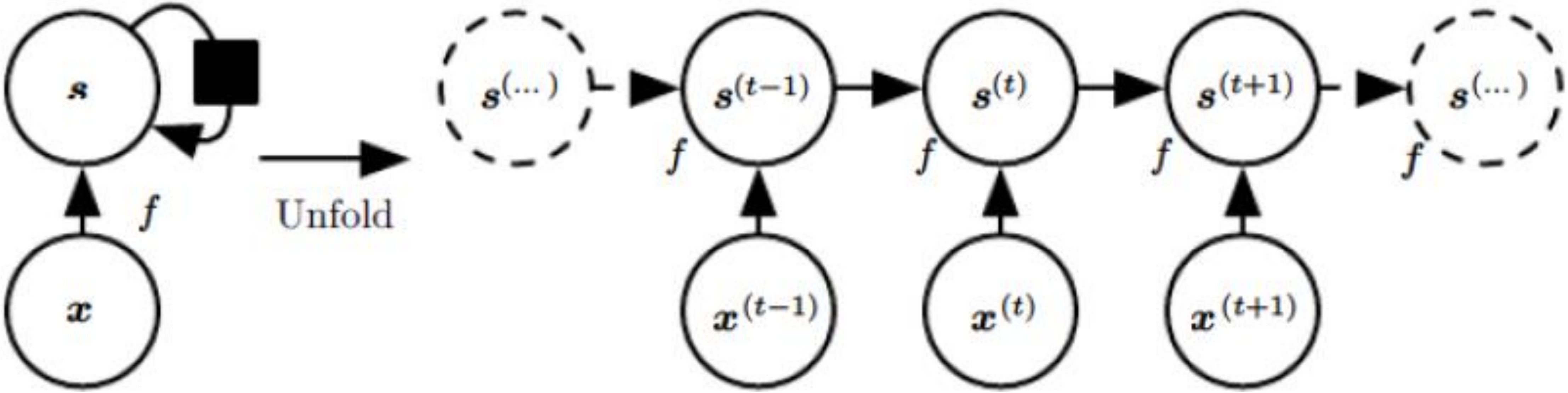
# A System Driven By External Data



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

Figure from Deep Learning, Goodfellow, Bengio and Courville

# Compact View (1/2)

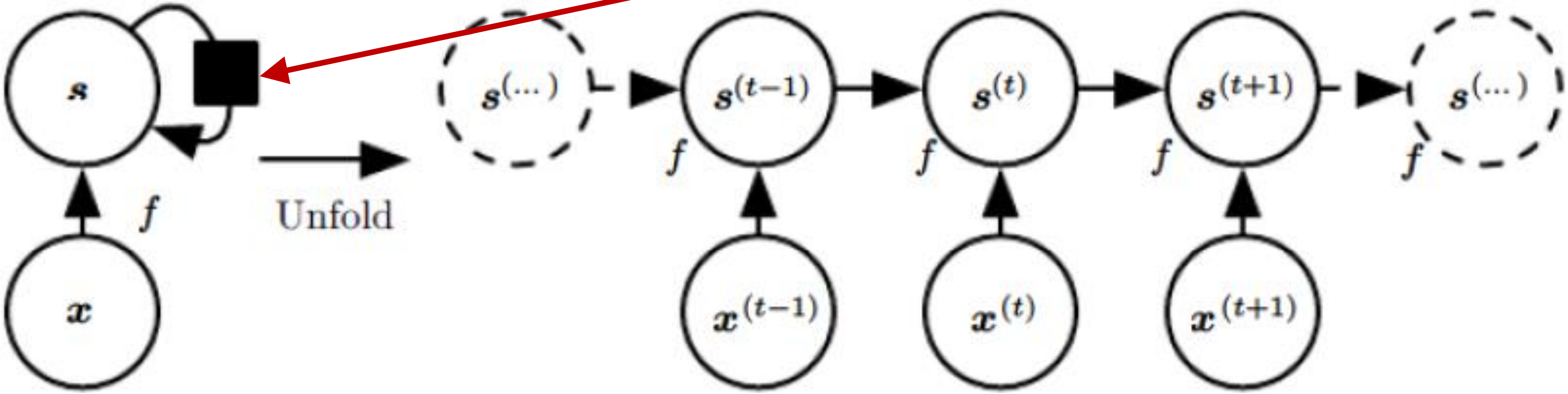


$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

Figure from Deep Learning, Goodfellow, Bengio and Courville

# Compact View (2/2)

Square: One step time delay



Key: the same $f$ and $\theta$ for all time steps

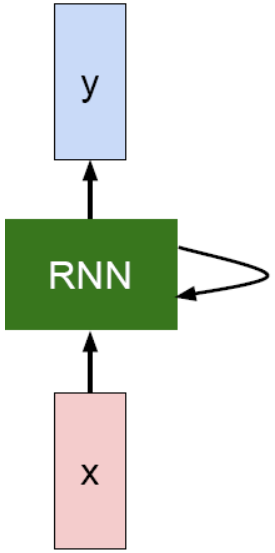
$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

Figure from Deep Learning, Goodfellow, Bengio and Courville

# Compact View

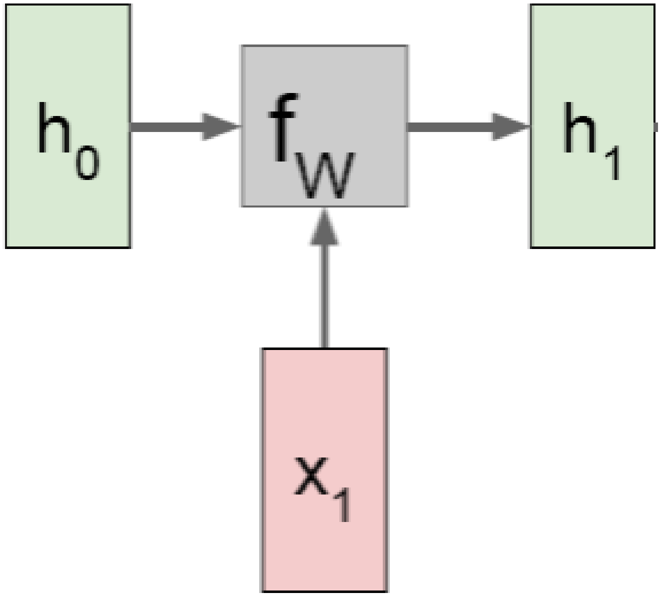
- Other forms
  - We can process a sequence of vectors x by applying a **recurrence formula** at every time step.
  - The same function and the same set of parameters are used at every time step.
  - CNNs share parameters across space; RNNs share across time.

$$\underset{\text{new state}}{\boxed{h_t}} = \underset{\text{some function with parameters W}}{\boxed{f_W}}(\underset{\text{old state}}{\boxed{h_{t-1}}}, \underset{\text{input vector at some time step}}{\boxed{x_t}})$$



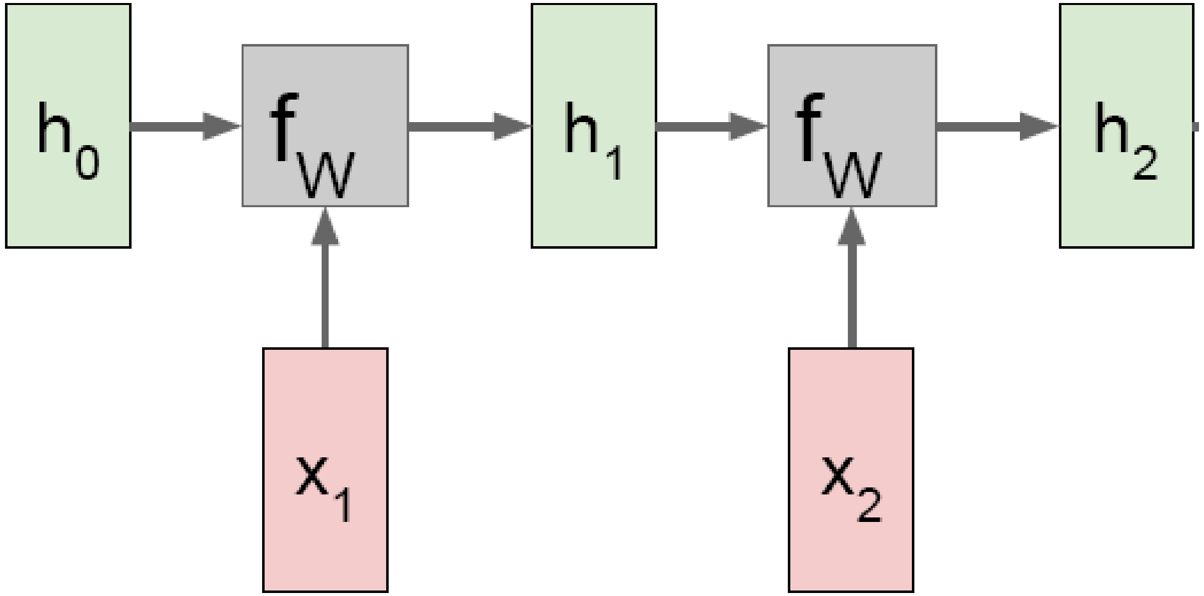
# A System Driven By External Data (1/7)

- Computational graph



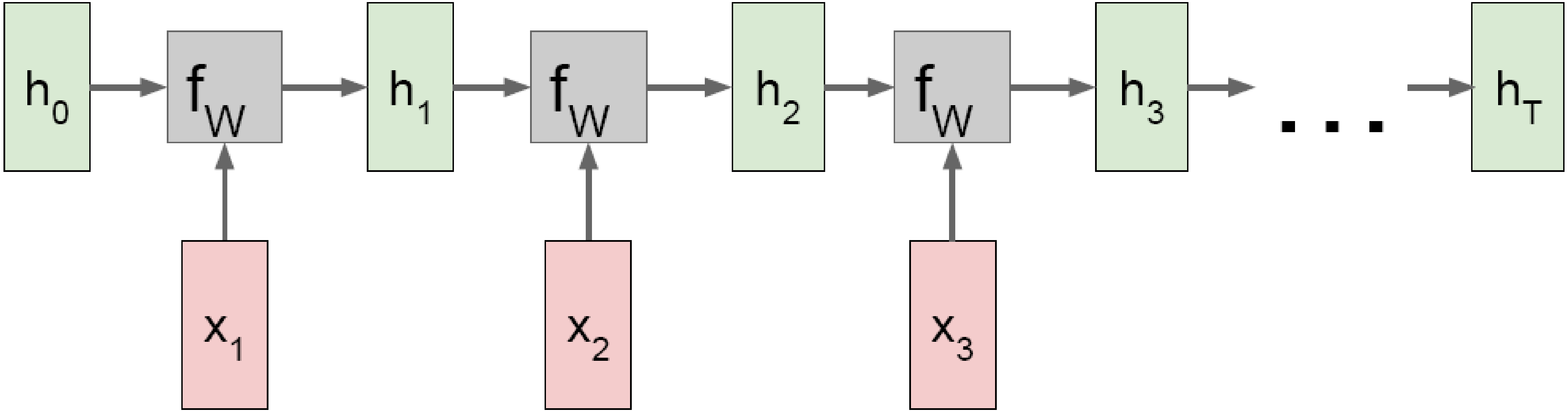
# A System Driven By External Data (2/7)

- Computational graph



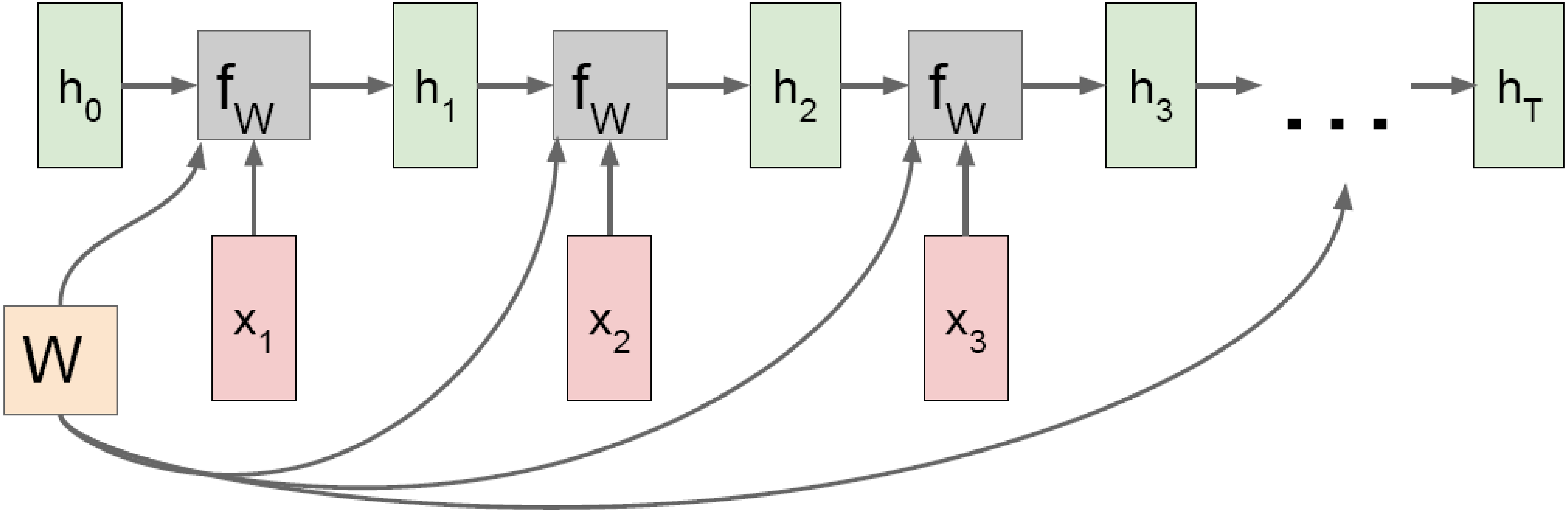
# A System Driven By External Data (3/7)

- Computational graph



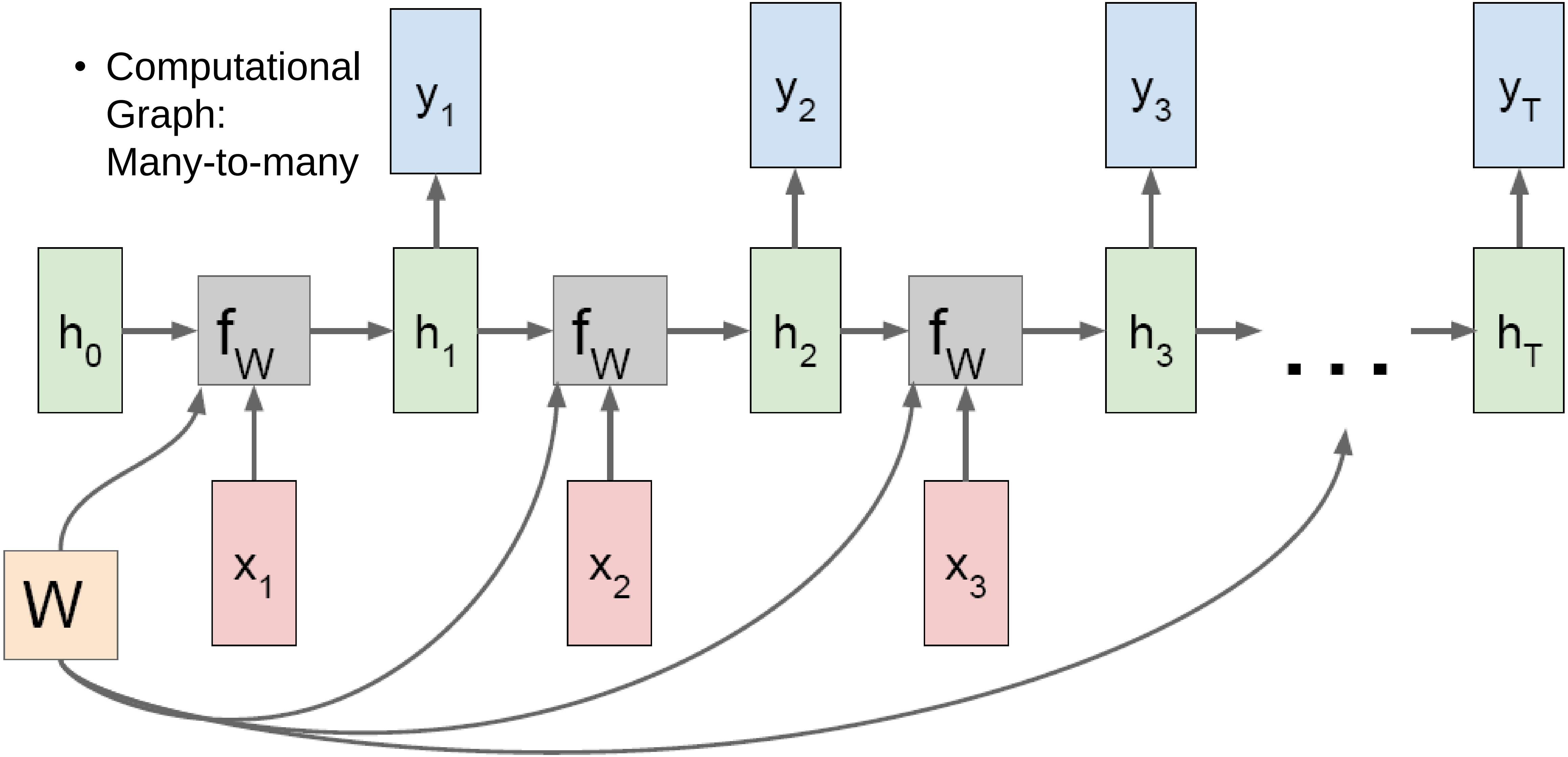
# A System Driven By External Data (4/7)

- Computational graph
    - Re-use the same weight matrix at every time-step



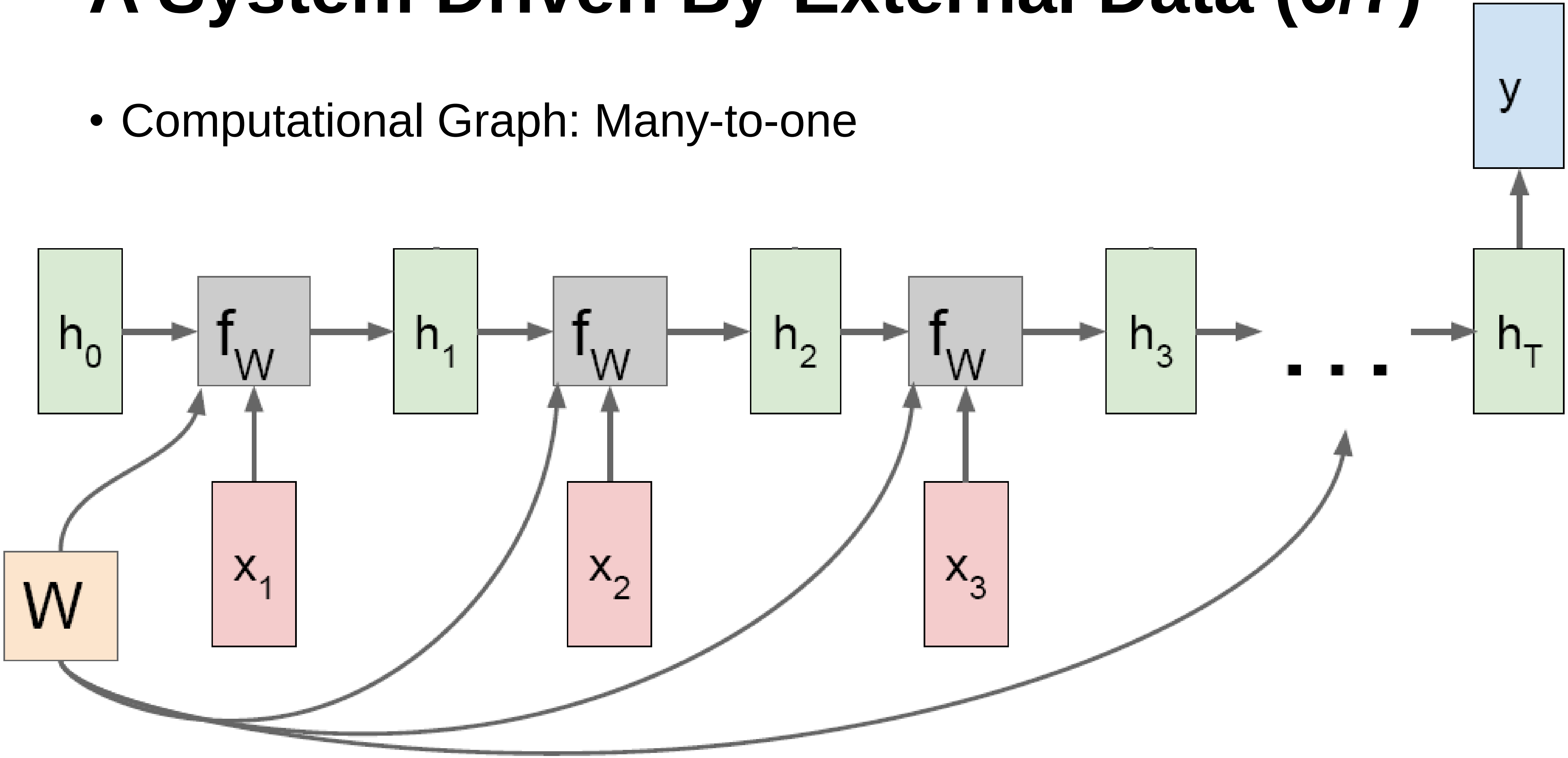
# A System Driven By External Data (5/7)

- Computational Graph: Many-to-many



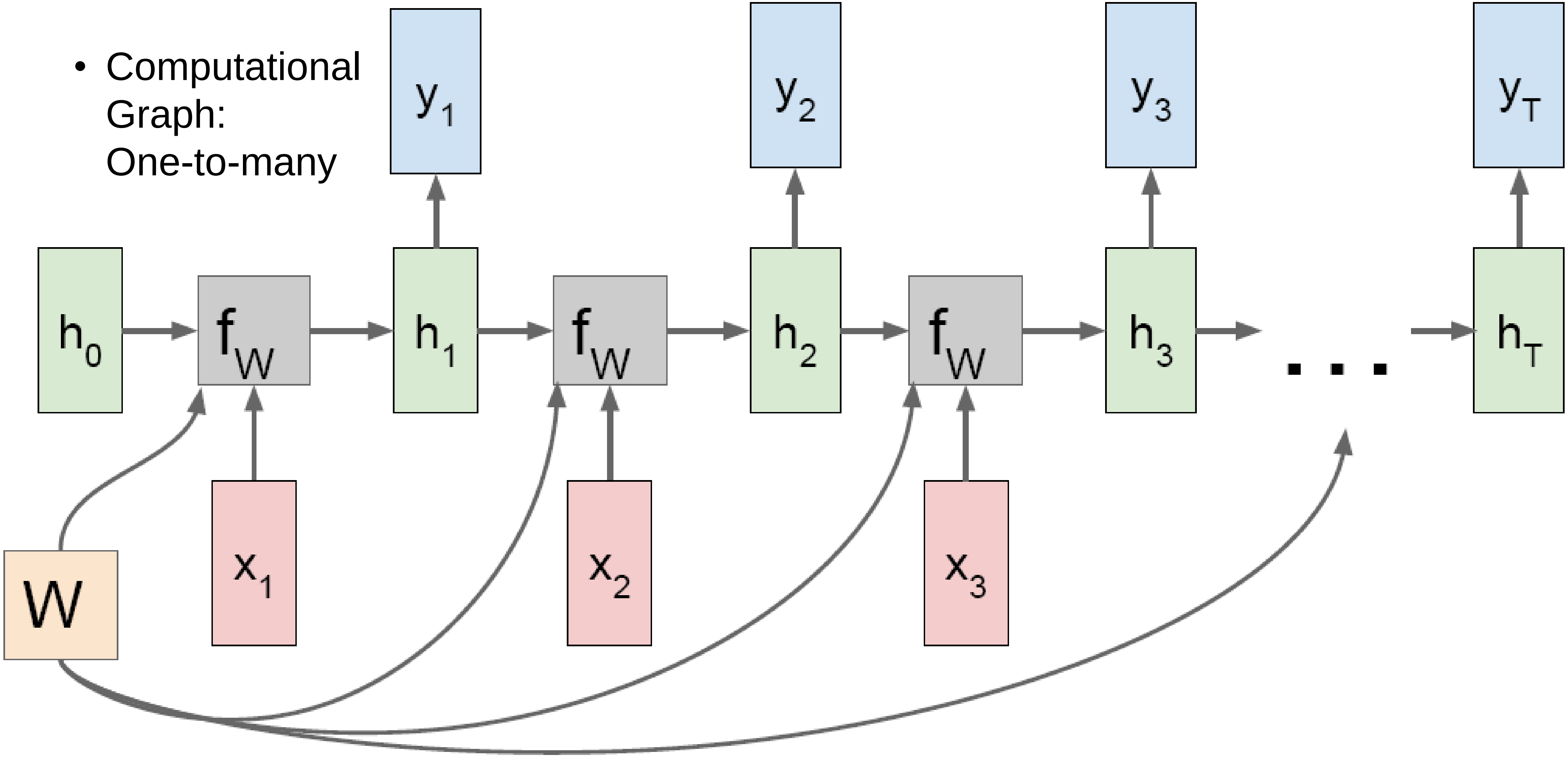
# A System Driven By External Data (6/7)

- Computational Graph: Many-to-one



# A System Driven By External Data (7/7)

- Computational Graph: One-to-many



# Recurrent Neural Networks (1/8)

- 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

# Recurrent Neural Networks (2/8)

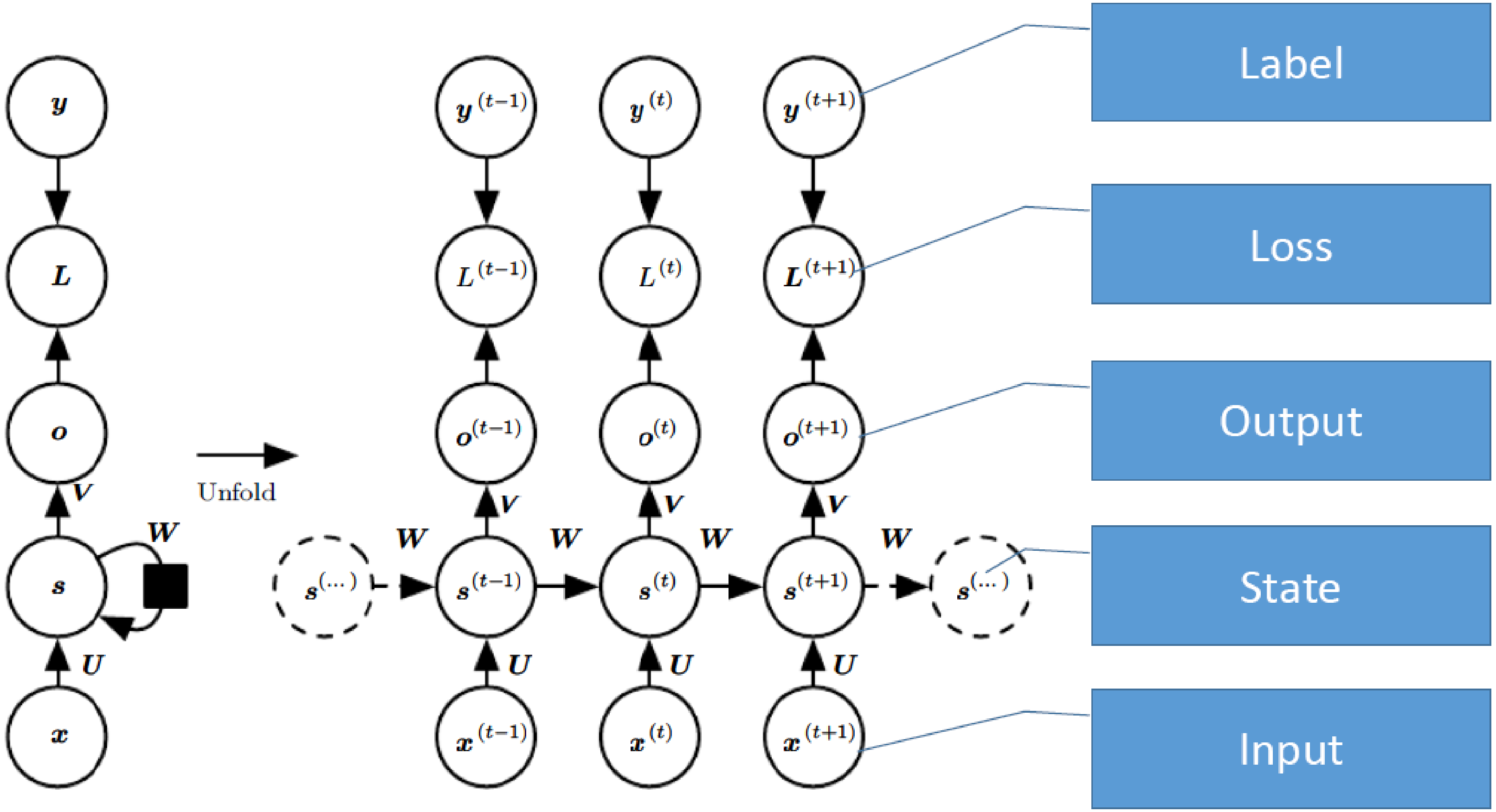
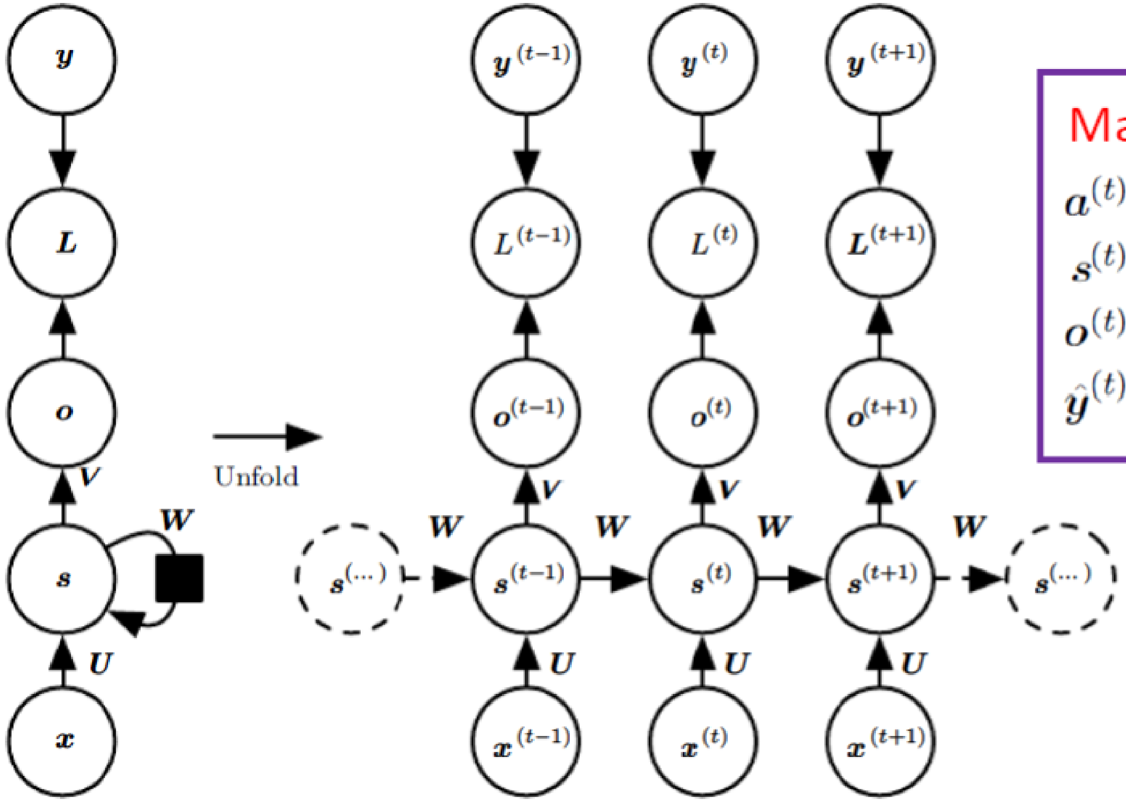


Figure from Deep Learning, Goodfellow, Bengio and Courville

# Recurrent Neural Networks (3/8)



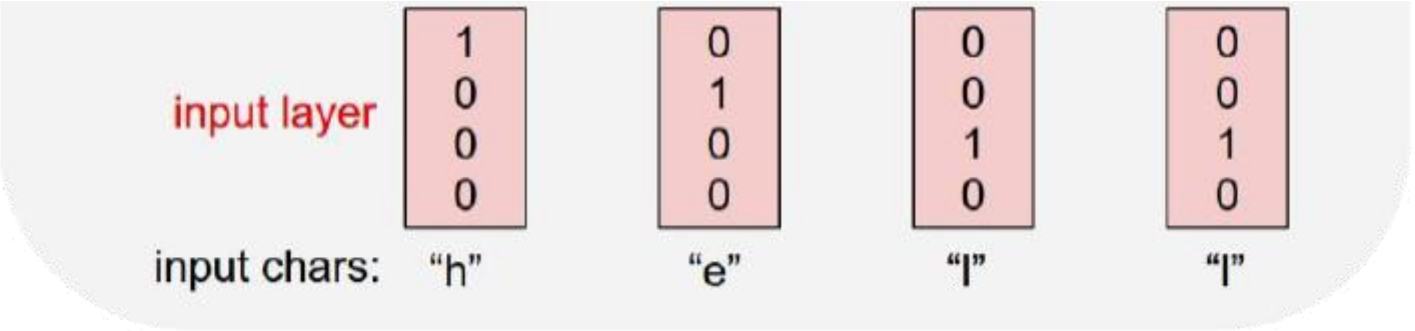
Math formula:

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

Figure from Deep Learning, Goodfellow, Bengio and Courville

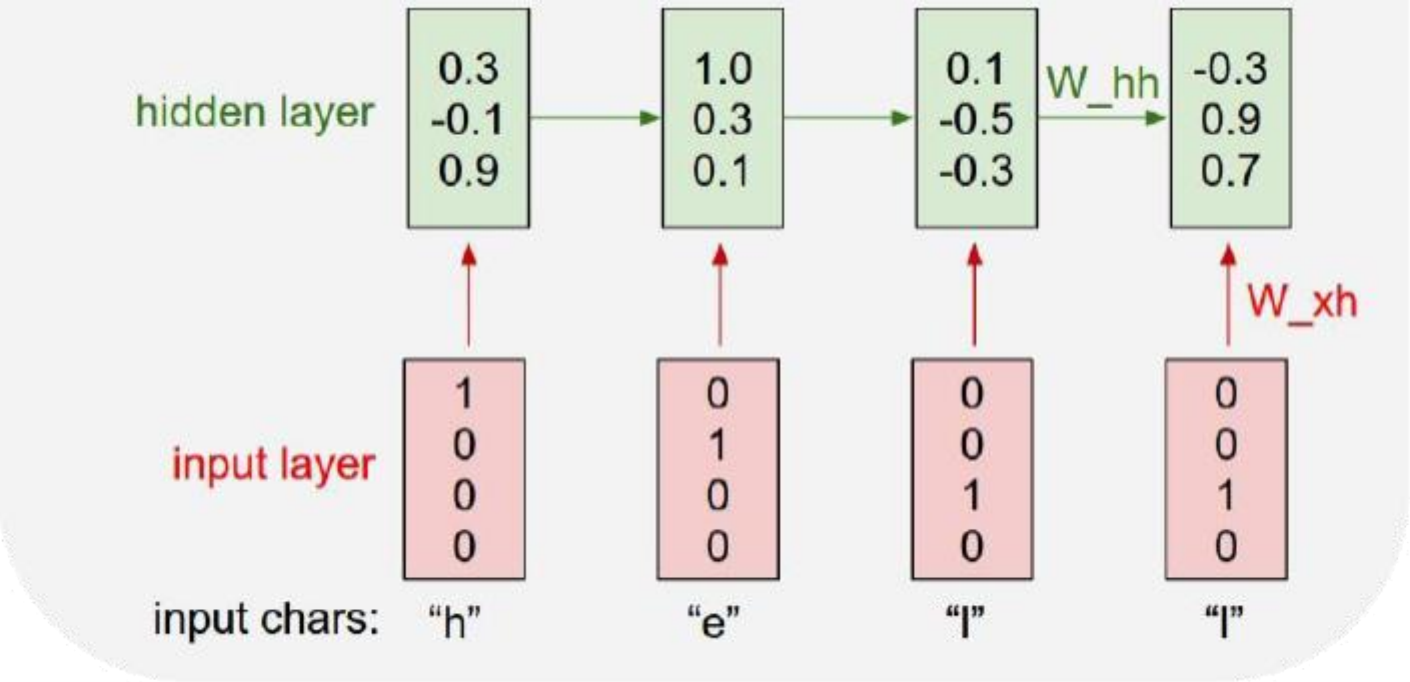
# Recurrent Neural Networks (4/8)

- **Example:**
  - Character-level language model
  - Vocabulary: [h,e,l,o]
  - Training sample: “hello”



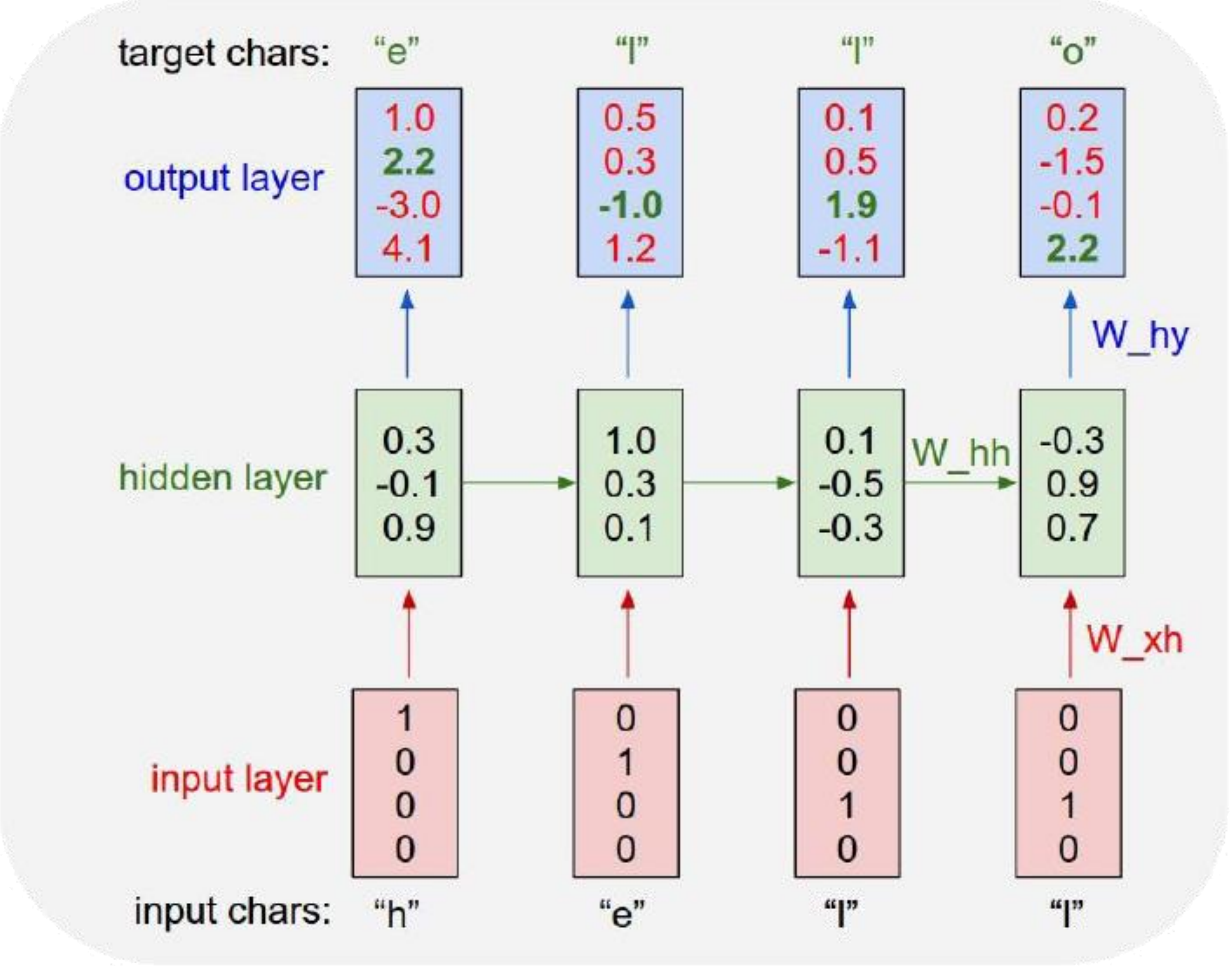
# Recurrent Neural Networks (5/8)

- **Example:**
  - Character-level language model
  - Vocabulary: [h,e,l,o]
  - Training sample: “hello”



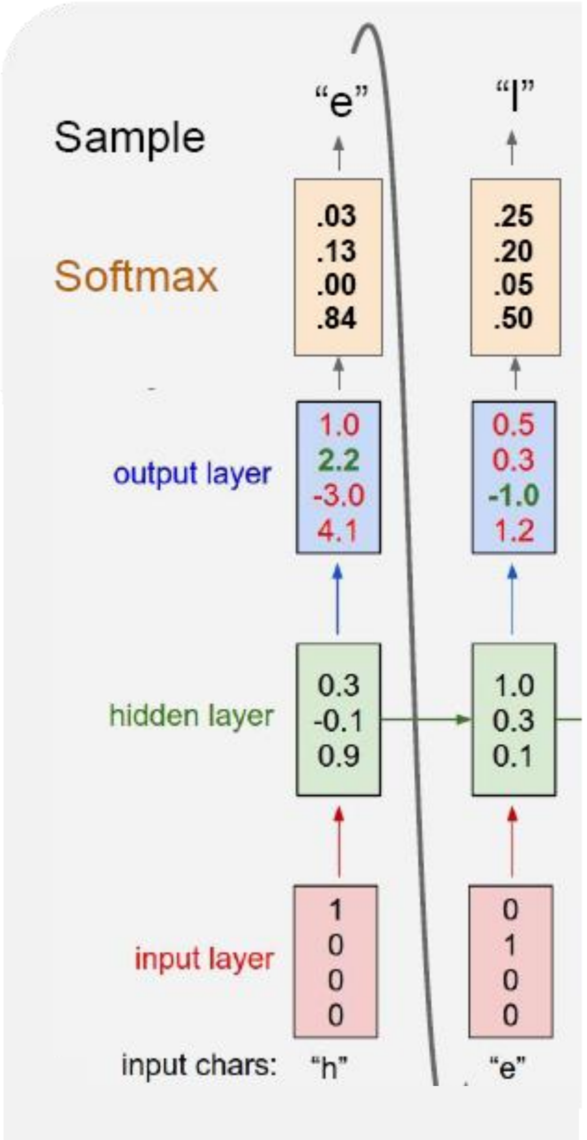
# Recurrent Neural Networks (6/8)

- **Example:**
  - Character-level language model
  - Vocabulary: [h,e,l,o]
  - Training sample: “hello”



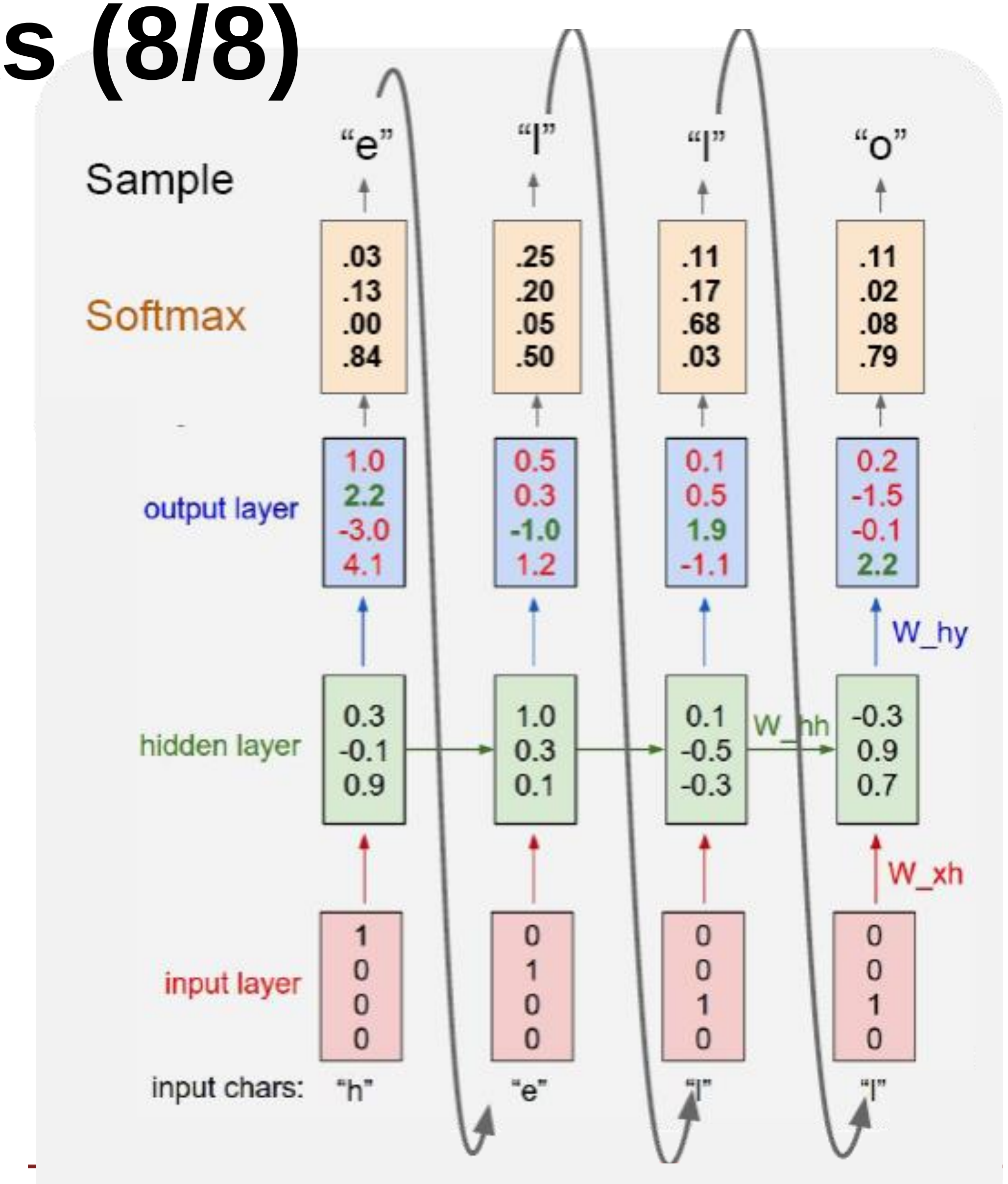
# Recurrent Neural Networks (7/8)

- **Example:**
  - Character-level language model
  - Vocabulary: [h,e,l,o]
  - Testing: character one at a time, feed back to model



# Recurrent Neural Networks (8/8)

- **Example:**
  - Character-level language model
  - Vocabulary: [h,e,l,o]
  - Testing: character one at a time, feed back to model



# Advantages

- **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])

# Example Implementation

- Given past, predict future; let’s implement an MLP model

# LSTM: Long Short Term Memory

- The basic structure of LSTM and some symbols to aid understanding

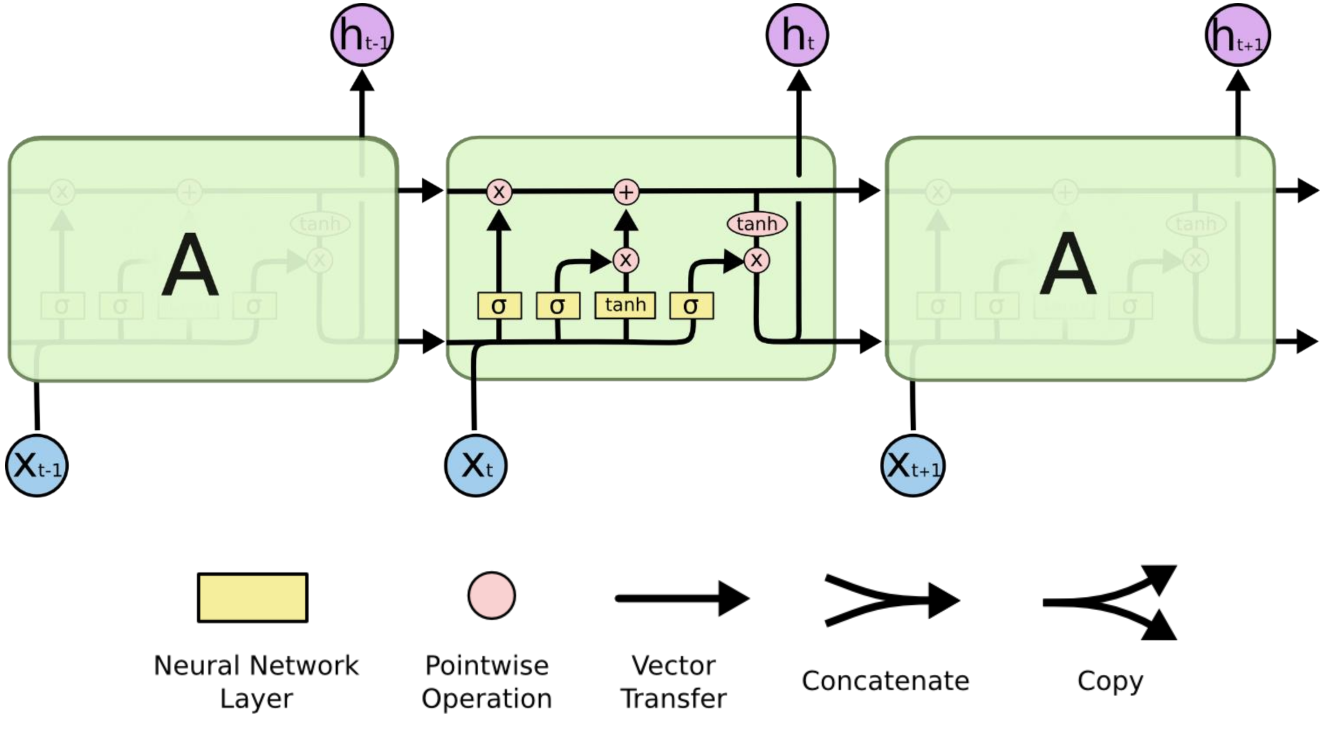
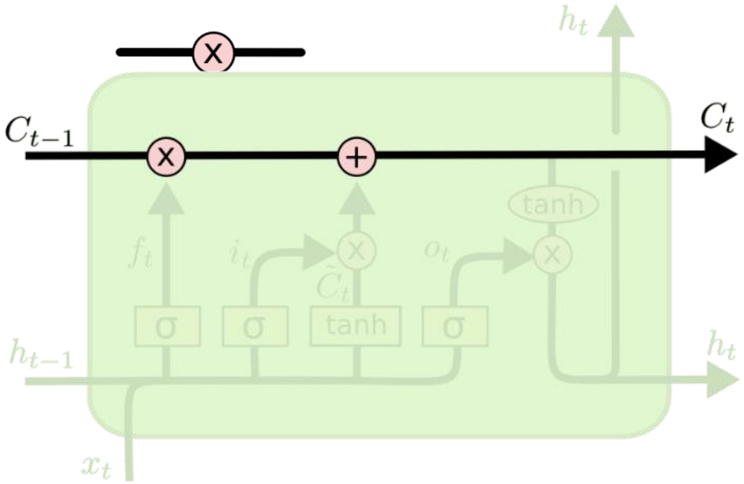


Image source: http://colah.github.io/posts/2015-08-Understanding-LSTMs

# LSTM Core Ideas

- Two key ideas of LSTM:
  - A backbone to carry state forward and gradients backward.

- Gating (pointwise multiplication) to modulate information flow. Sigmoid makes 0 < gate < 1.



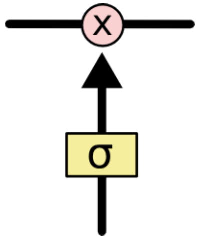
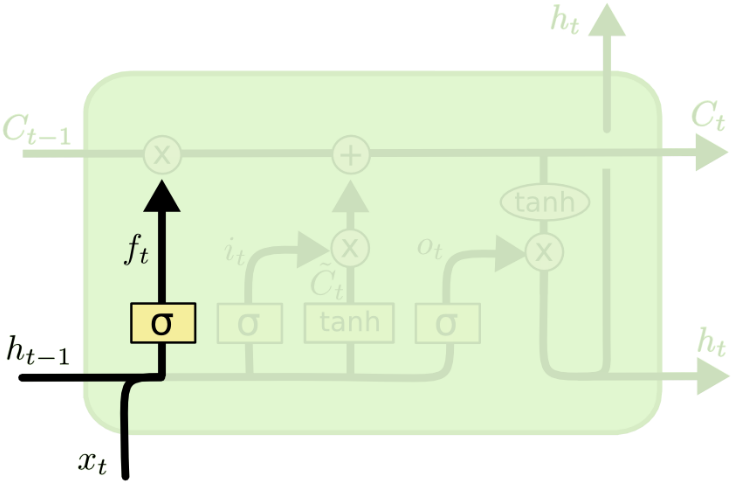


Image source: http://colah.github.io/posts/2015-08-Understanding-LSTMs

# LSTM Gating: Forget

- The *f* gate is ‘forgetting.’ Use previous state, *C,* previous output, *h*, and current input, *x*, to determine how much to suppress previous state.

- E.g., *C* might encode the fact that we have a subject and need a verb. Forget that when verb found.

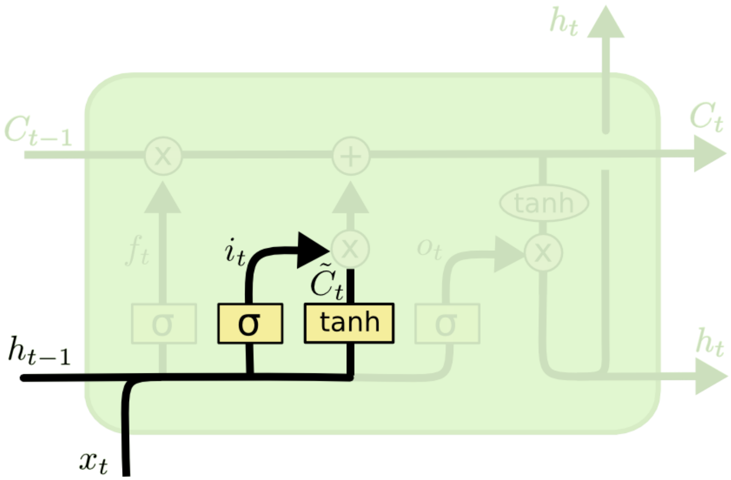


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

Image source: http://colah.github.io/posts/2015-08-Understanding-LSTMs

# LSTM Gating: Input Gate

- Input gate *i* determines which values of *C* to update
- Separate tanh layer produces new state to add to *C*



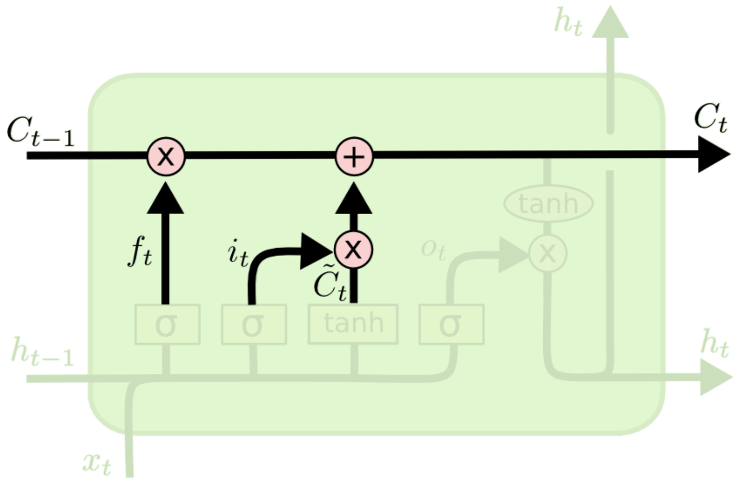
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \;+\; b_i\right)$$

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

Image source: http://colah.github.io/posts/2015-08-Understanding-LSTMs

# LSTM Gating: Update to C

- Forget gate does pointwise modulation of *C*.
- Input gate modulates the tanh layer – this is added to *C*.

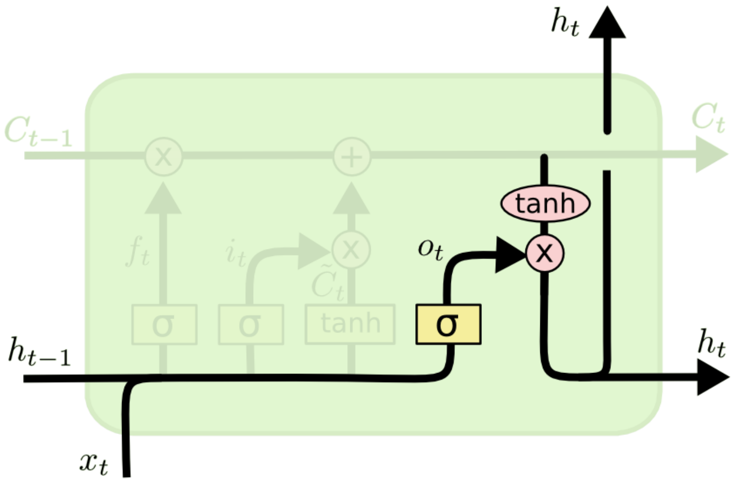


$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Image source: http://colah.github.io/posts/2015-08-Understanding-LSTMs

# LSTM Gating: Output

- *o* is the **output gate**: modulates what part of the state *C* gets passed (via tanh) to current output *h*



- E.g., could encode whether a noun is singular or plural to prepare for a verb
- But the real features are learned, not engineered.

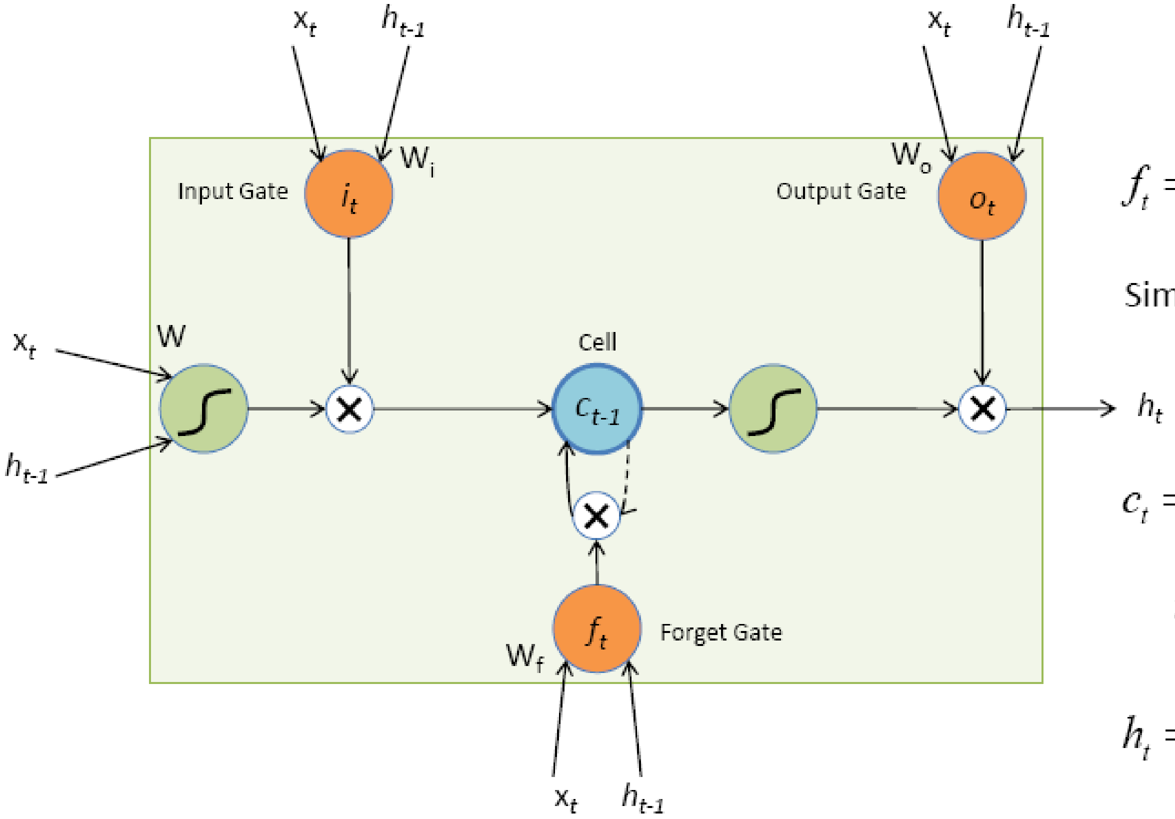
$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$

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

Image source: http://colah.github.io/posts/2015-08-Understanding-LSTMs

# The Popular LSTM Cell

- Another demonstration



$$f_t = \sigma\left( W_f \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_f \right)$$

Similarly for $i_t$, $o_t$

$$c_t = f_t \otimes c_{t-1} + i_t \otimes \tanh W \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix}$$

$$h_t = o_t \square \tanh c_t$$

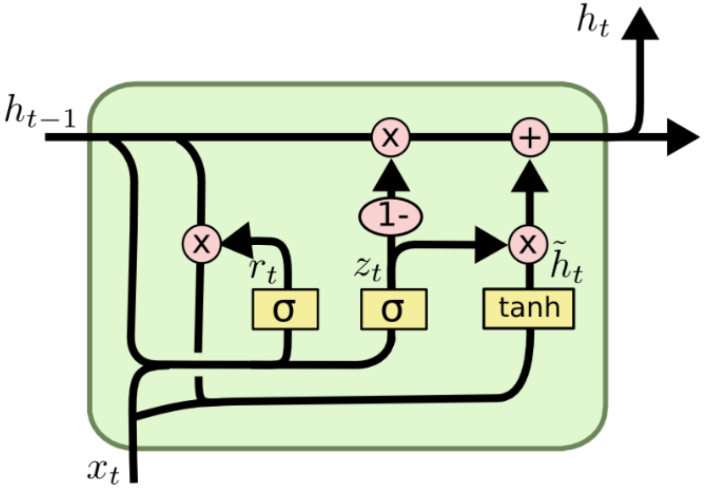
**Dashed line indicates time-lag**

# Example Implementation

- Given past, predict future; let’s implement a simple LSTM model
- Discuss pros/cons, compare with MLP model

# GRU: Gated Recurrent Unit

- Combine *C* and *h* into a single state/output
- Combine forget and input gates into update gate, *z*



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Image source: colah.github.io/posts/2015-08-Understanding-LSTMs

# Gated Recurrent Units (GRUs) (1/3)

- Main idea:
  - Keep around memory to capture **long dependencies**
  - Allow error messages to flow at **different strengths** depending on the inputs
- Standard RNN computes hidden layer at next time step directly: $h_t = \sigma(W^{(hh)}h_{t-1} + W^{(hx)}x_t)$
- Compute an update gate based on current input word vector and hidden state $z_t = \sigma(U^{(z)}h_{t-1} + W^{(z)}x_t)$
  - Controls how much of past state should matter now
  - If z close to 1, then we can copy information in that unit through many steps!

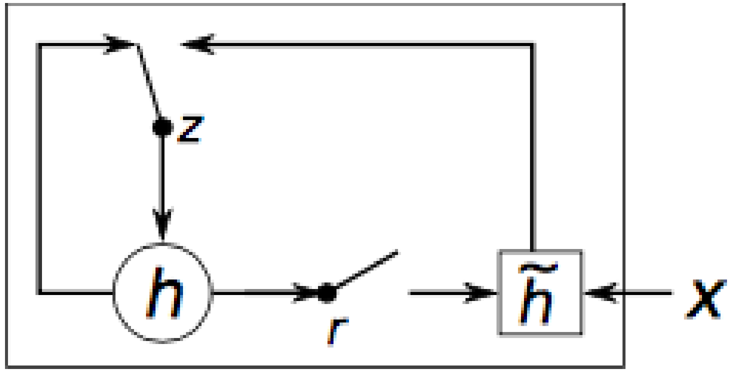
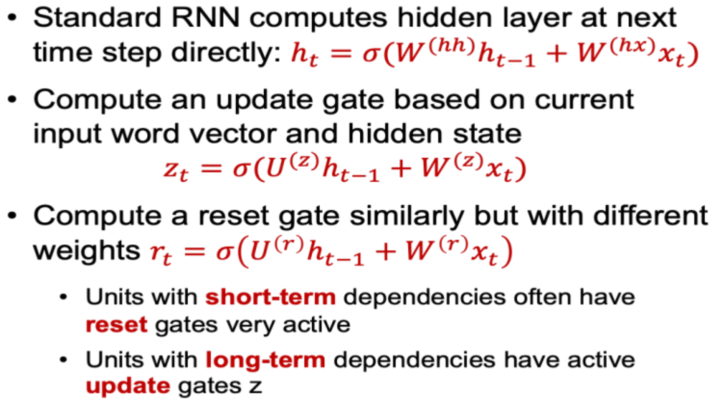


*Image source: www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4-implementing-a-grulstm-rnn-with-python-and-theano*

# Gated Recurrent Units (GRUs) (2/3)

- Standard RNN computes hidden layer at next time step directly: $h_t = \sigma(W^{(hh)}h_{t-1} + W^{(hx)}x_t)$
- Compute an update gate based on current input word vector and hidden state
  $$z_t = \sigma(U^{(z)}h_{t-1} + W^{(z)}x_t)$$
- Compute a reset gate similarly but with different weights $r_t = \sigma\left(U^{(r)}h_{t-1} + W^{(r)}x_t\right)$
  - Units with **short-term** dependencies often have **reset** gates very active
  - Units with **long-term** dependencies have active **update** gates z



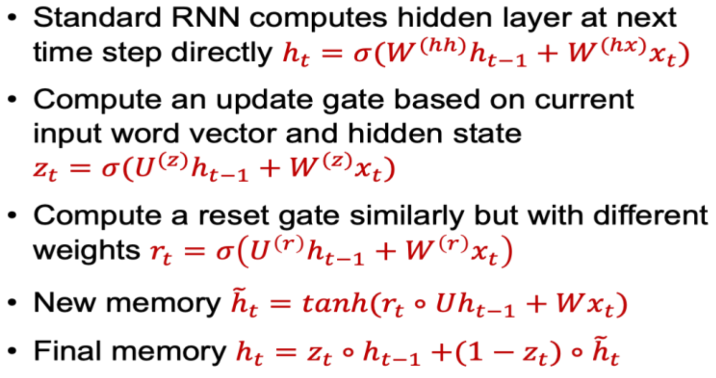
- Standard RNN computes hidden layer at next time step directly: $h_t = \sigma(W^{(hh)}h_{t-1} + W^{(hx)}x_t)$
- Compute an update gate based on current input word vector and hidden state
  $$z_t = \sigma(U^{(z)}h_{t-1} + W^{(z)}x_t)$$
- Compute a reset gate similarly but with different weights $r_t = \sigma\left(U^{(r)}h_{t-1} + W^{(r)}x_t\right)$
  - Units with **short-term** dependencies often have **reset** gates very active
  - Units with **long-term** dependencies have active **update** gates z

Image source: www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4-implementing-a-grulstm-rnn-with-python-and-theano

If reset close to 0, ignore previous hidden state (allows model to drop information that is irrelevant in the future)

# Gated Recurrent Units (GRUs) (3/3)

- Standard RNN computes hidden layer at next time step directly $h_t = \sigma(W^{(hh)}h_{t-1} + W^{(hx)}x_t)$
- Compute an update gate based on current input word vector and hidden state
  $$z_t = \sigma(U^{(z)}h_{t-1} + W^{(z)}x_t)$$
- Compute a reset gate similarly but with different weights $r_t = \sigma\left(U^{(r)}h_{t-1} + W^{(r)}x_t\right)$
- New memory $\tilde{h}_t = tanh(r_t \circ Uh_{t-1} + Wx_t)$
- Final memory $h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$



- Standard RNN computes hidden layer at next time step directly $h_t = \sigma(W^{(hh)}h_{t-1} + W^{(hx)}x_t)$
- Compute an update gate based on current input word vector and hidden state $z_t = \sigma(U^{(z)}h_{t-1} + W^{(z)}x_t)$
- Compute a reset gate similarly but with different weights $r_t = \sigma\left(U^{(r)}h_{t-1} + W^{(r)}x_t\right)$
- New memory $\tilde{h}_t = tanh(r_t \circ Uh_{t-1} + Wx_t)$
- Final memory $h_t = z_t \circ h_{t-1} + (1 - z_t) \circ \tilde{h}_t$

Image source: www.wildml.com/2015/10/recurrent-neural-network-tutorial-part-4-implementing-a-grulstm-rnn-with-python-and-theano

- LSTMs are a more complex form, but basically same intuition
- GRUs are often more preferred than LSTMs

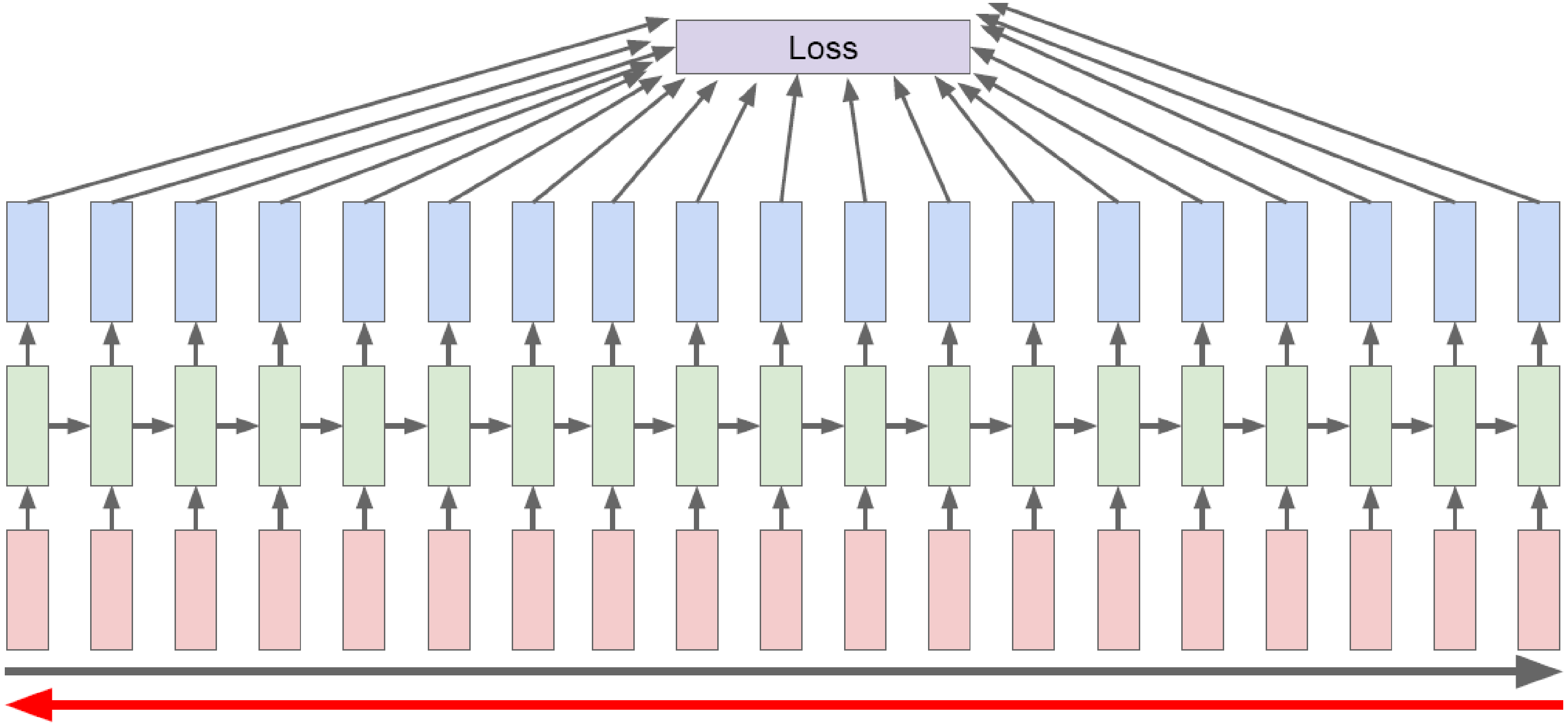
combines current & previous time steps

# Training RNN (1/7)

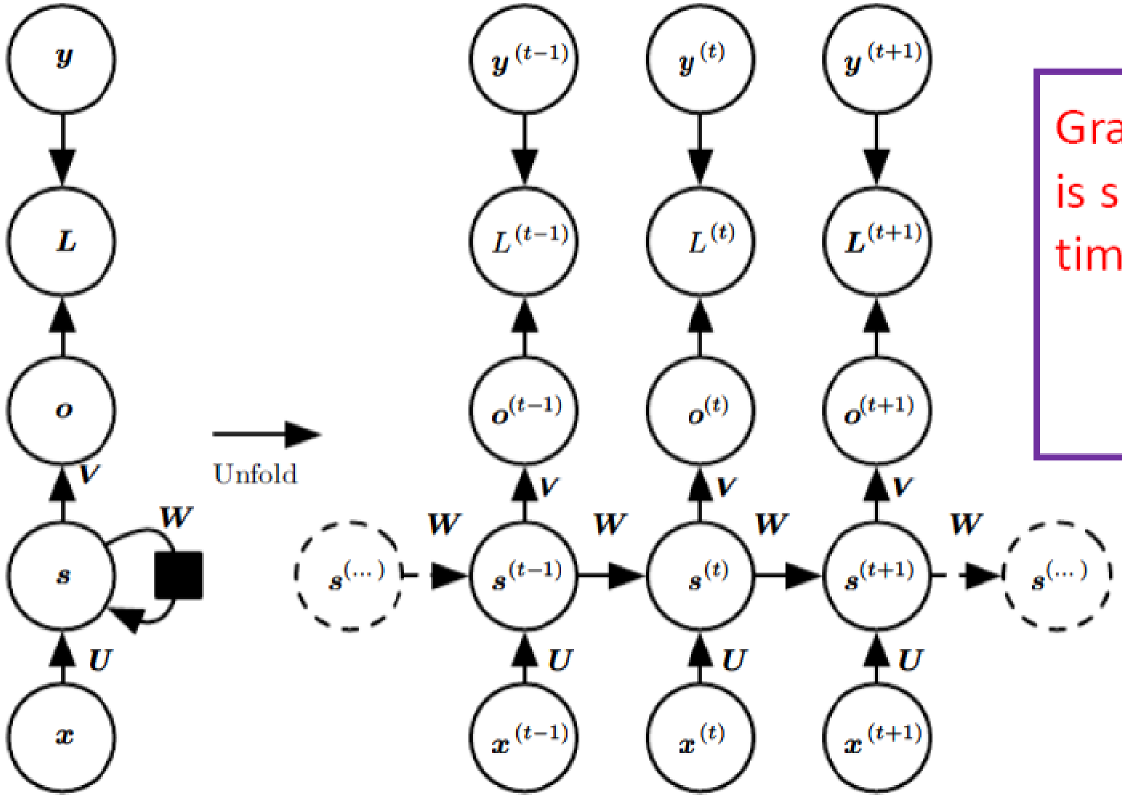
- Principle: unfold the computational graph, and use backpropagation
- Called **back-propagation through time (BPTT) algorithm**
- Can then apply any general-purpose gradient-based techniques
- Conceptually: first compute the gradients of **the internal nodes**, then compute the gradients of **the parameters**

# Training RNN (2/7)

- Backpropagation through time



# Training RNN (3/7)

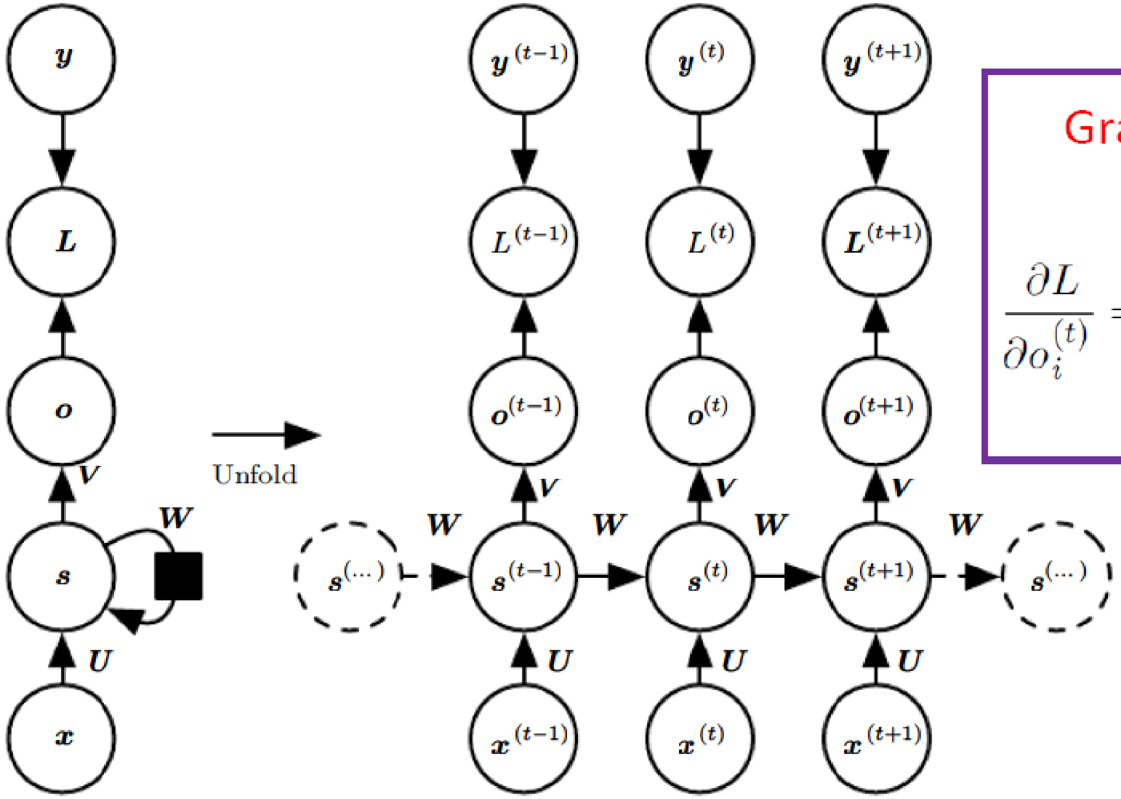


Gradient at $L^{(t)}$: (total loss is sum of those at different time steps)

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

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

# Training RNN (4/7)

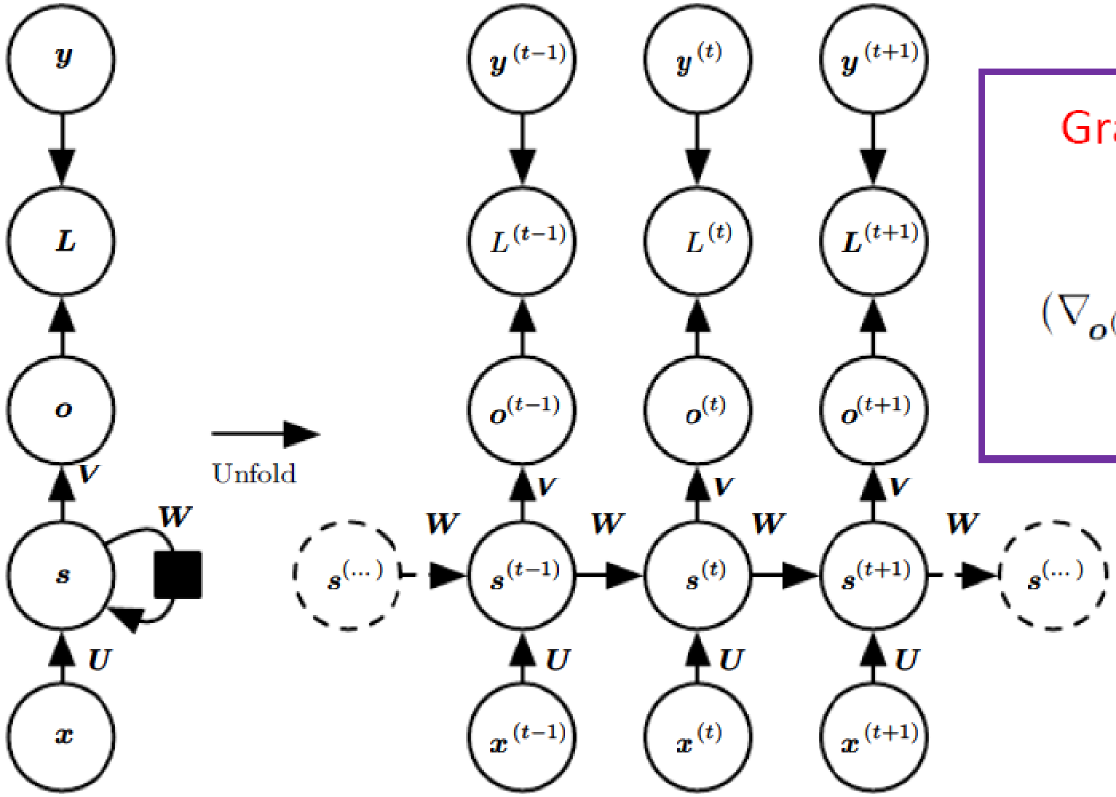


Gradient at $o^{(t)}$:

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

Figure from *Deep Learning,* Goodfellow, Bengio and Courville

# Training RNN (5/7)

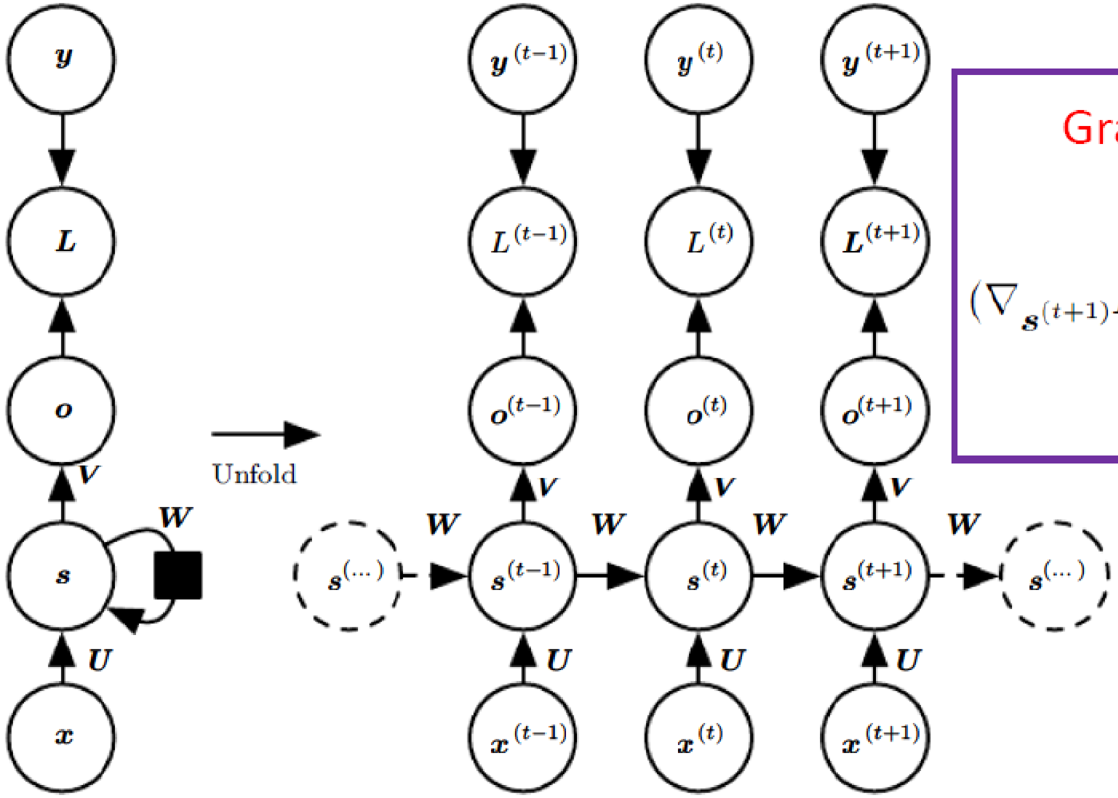


Gradient at $s^{(\tau)}$:

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

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

# Training RNN (6/7)

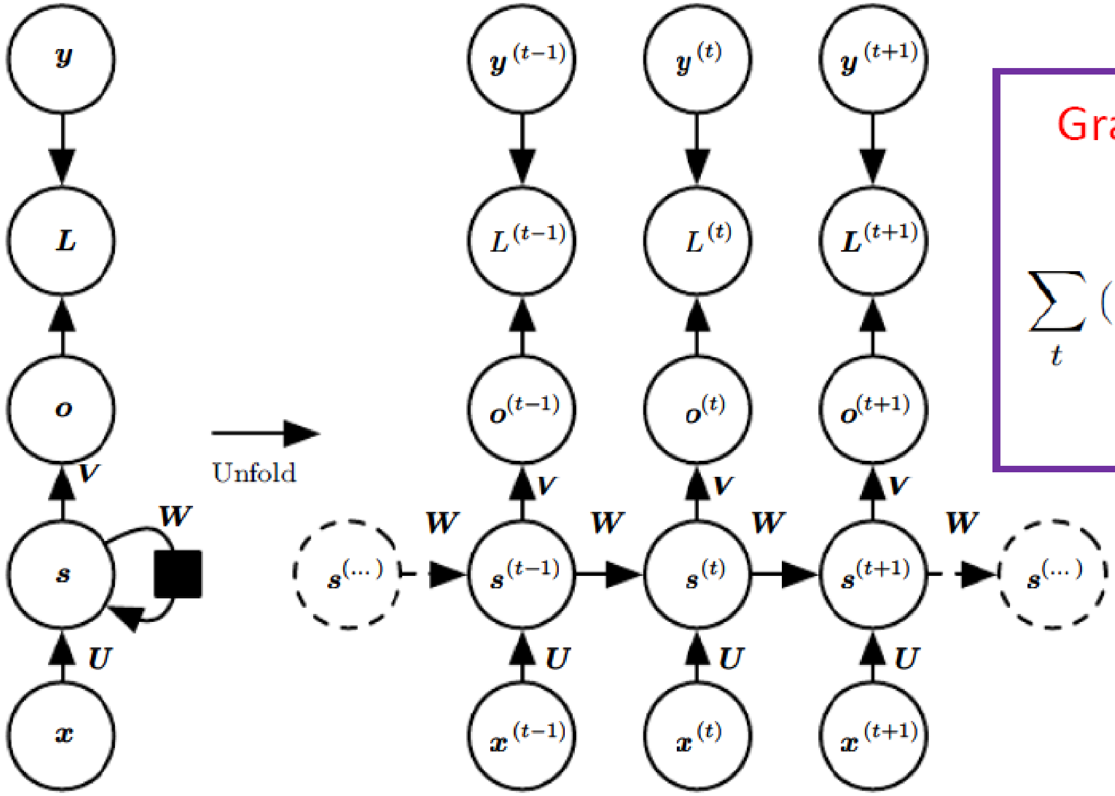


Gradient at $s^{(t)}$:

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

Figure from *Deep Learning,* Goodfellow, Bengio and Courville

# Training RNN (7/7)



Gradient at parameter $V$:

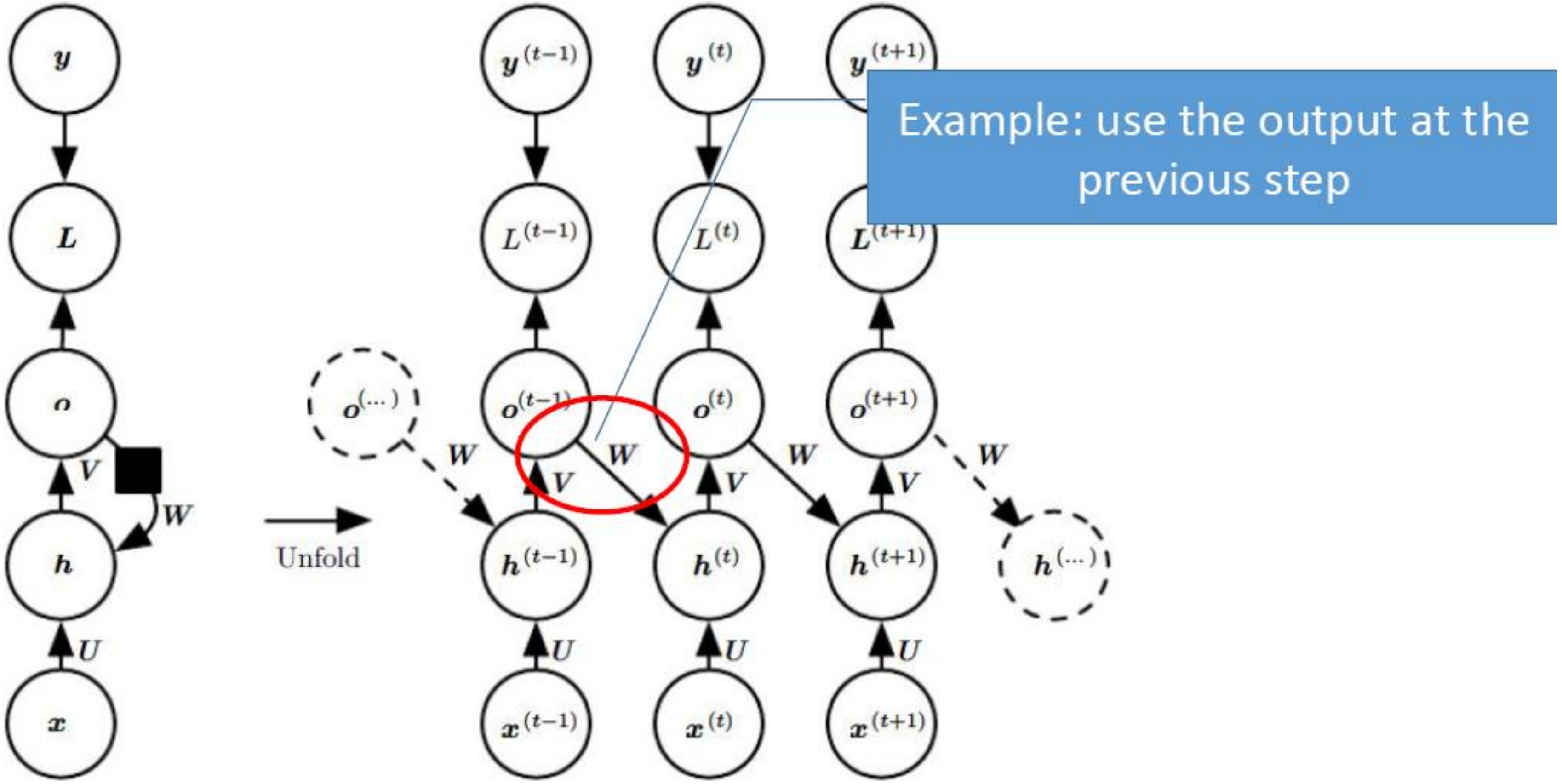
$$\sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \frac{\partial \boldsymbol{o}^{(t)}}{\partial V} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \boldsymbol{s}^{(t)\top}$$

Figure from *Deep Learning*, Goodfellow, Bengio and Courville

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

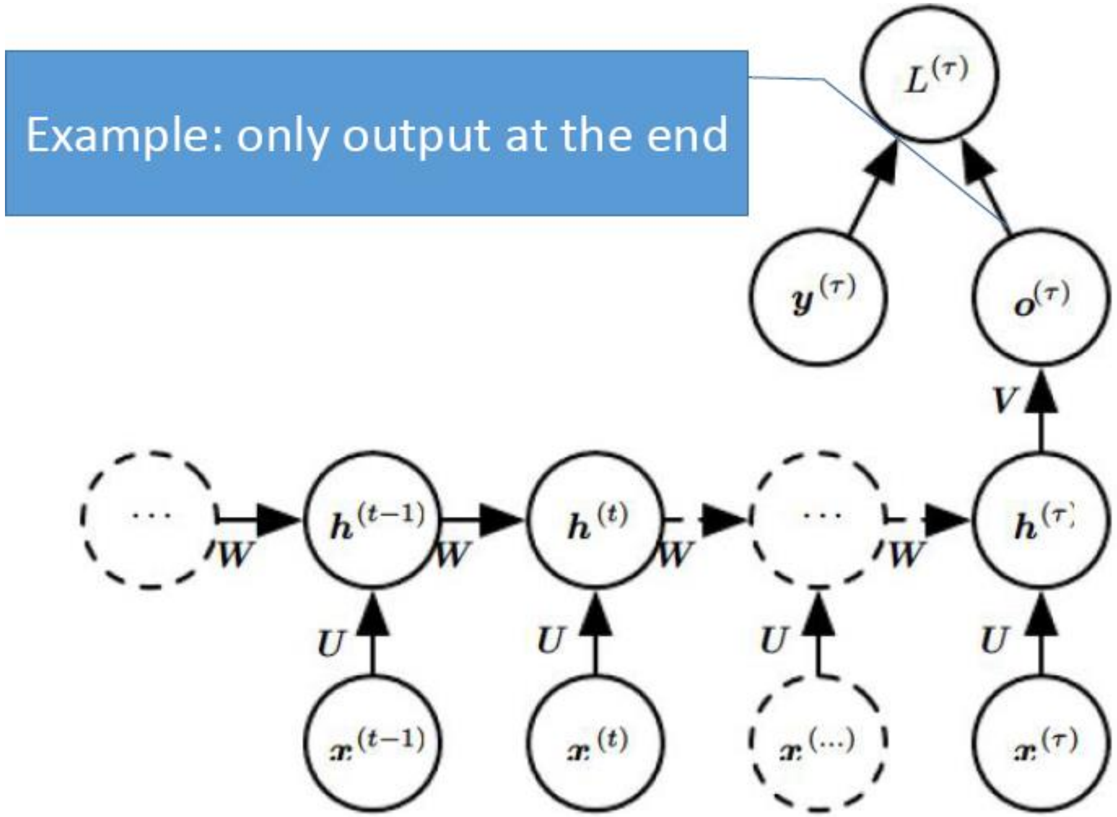
# RNN Variations (1/2)



Example: use the output at the previous step

Figure from *Deep Learning,* Goodfellow, Bengio and Courville

# RNN Variations (2/2)



Example: only output at the end

Figure from *Deep Learning,* Goodfellow, Bengio and Courville

# Bidirectional RNNs (1/2)

- 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 (2/2)

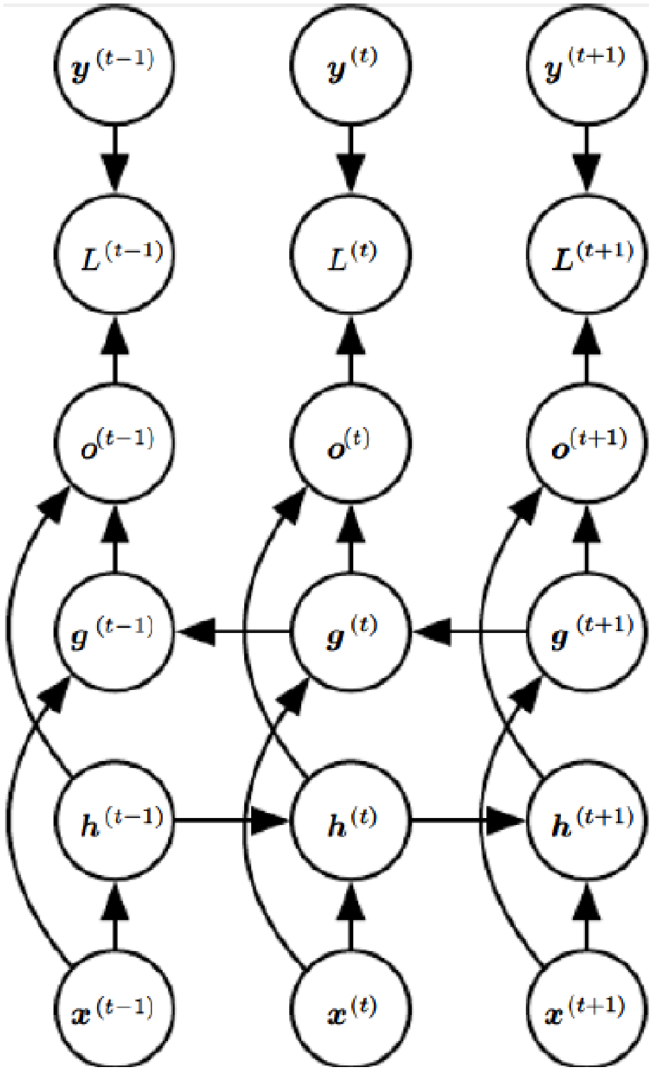
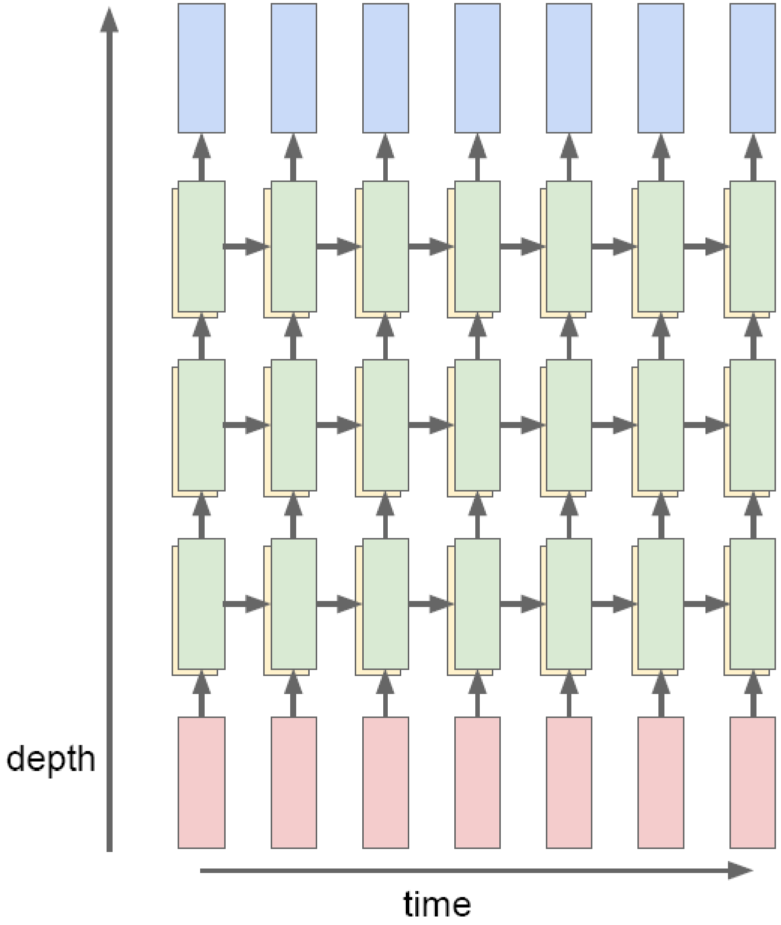


Image source: *Deep Learning,* Goodfellow, Bengio and Courville

# Multilayer RNNs



- Multilayer RNNs

$$\boxed{h_t^l = \tanh W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}}$$

$h \in \mathbb{R}^n$ $\qquad$ $W^l \ [n \times 2n]$

- LSTM

$W^l \ [4n \times 2n]$

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \tanh \end{pmatrix} W^l \begin{pmatrix} h_t^{l-1} \\ h_{t-1}^l \end{pmatrix}$$

$$c_t^l = f \odot c_{t-1}^l + i \odot g$$

$$h_t^l = o \odot \tanh(c_t^l)$$

# Encoder-decoder RNNs (1/2)

- 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-decoder RNNs (2/2)

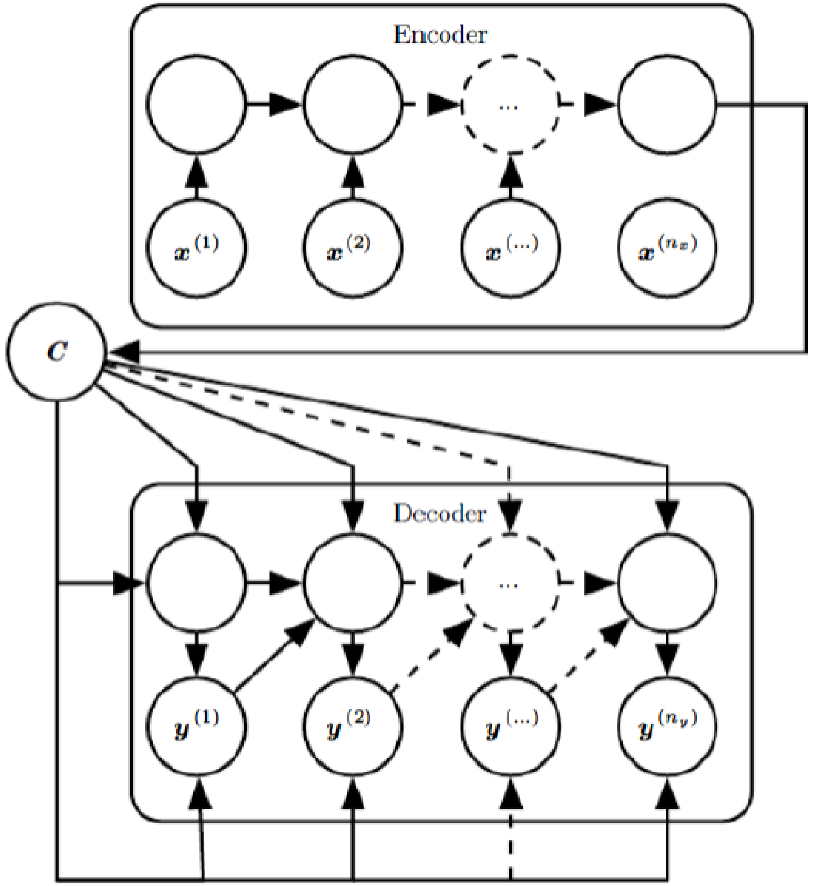


Image source: *Deep Learning*, Goodfellow, Bengio and Courville