# Neural Network Part 4: Recurrent Neural Networks

CS 760@UW-Madison

# Goals for the lecture

You should understand the following concepts

- sequential data
- computational graph
- recurrent neural networks (RNN) and the advantage
- encoder-decoder RNNs

Optional:

- training recurrent neural networks

# 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

Standard setting:

- Each data point: A sequence of vectors $x^{(t)}$, for $1 \leq t \leq \tau$
  - corresponding sequence of labels $y^{(t)}$, for $1 \leq t \leq \tau$
- Batch data: many sequences with different lengths $\tau$

Other settings:

- Label: can be a scalar, a vector, or even a sequence
- Examples
  - Sentiment analysis
  - Machine translation

# Example: machine translation

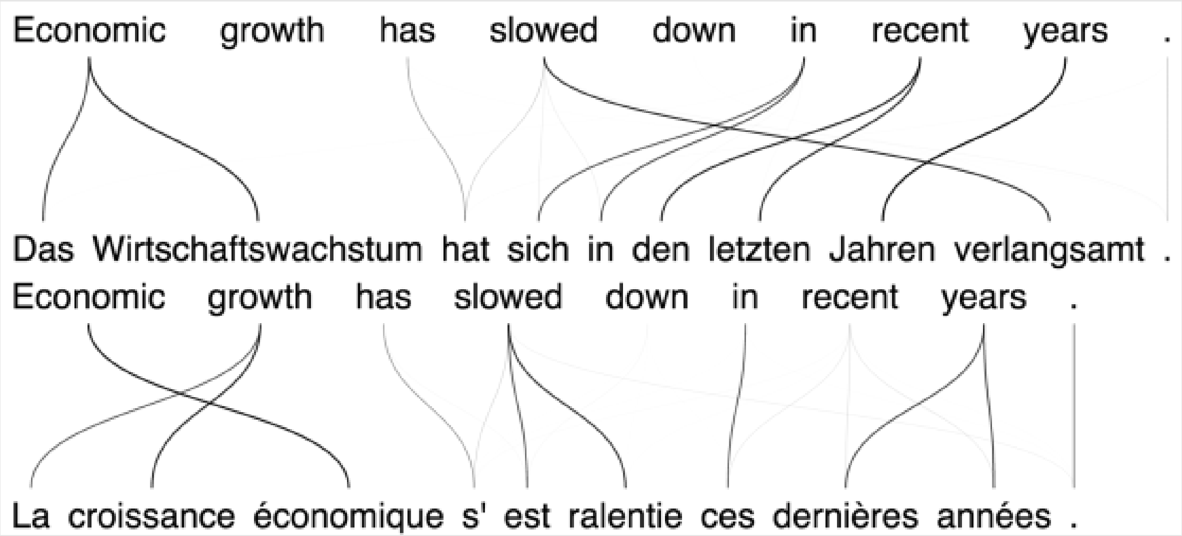



Figure from: devblogs.nvidia.com

# More complicated sequential data

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

# Image captioning

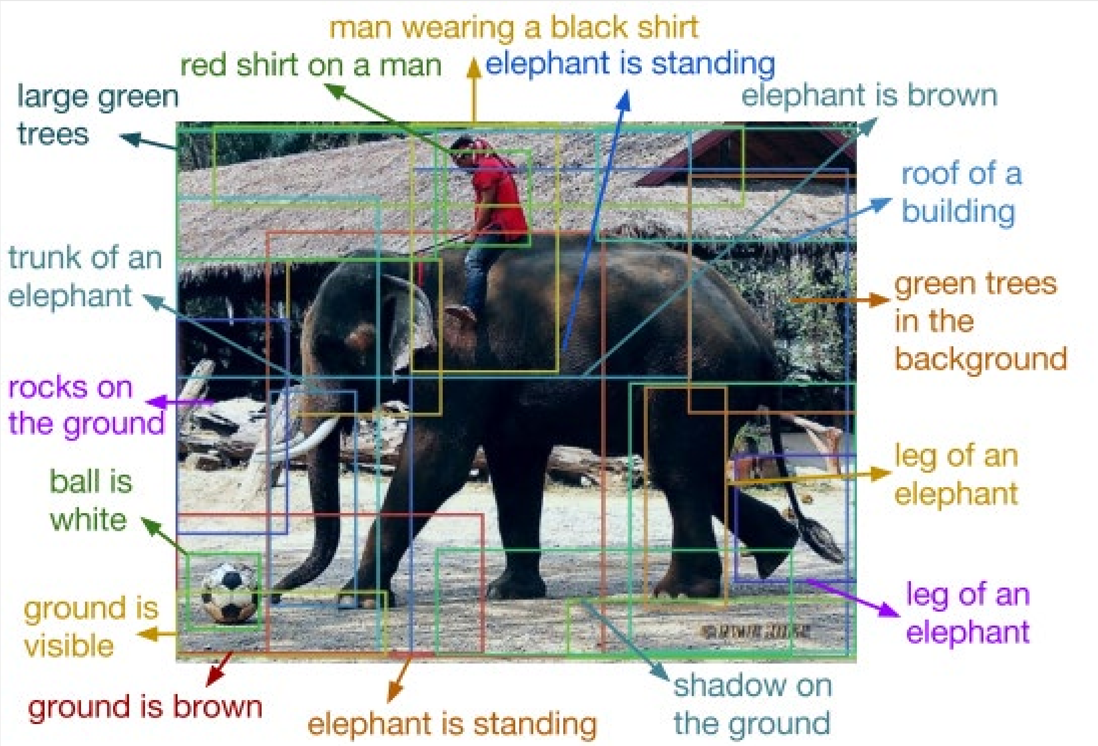
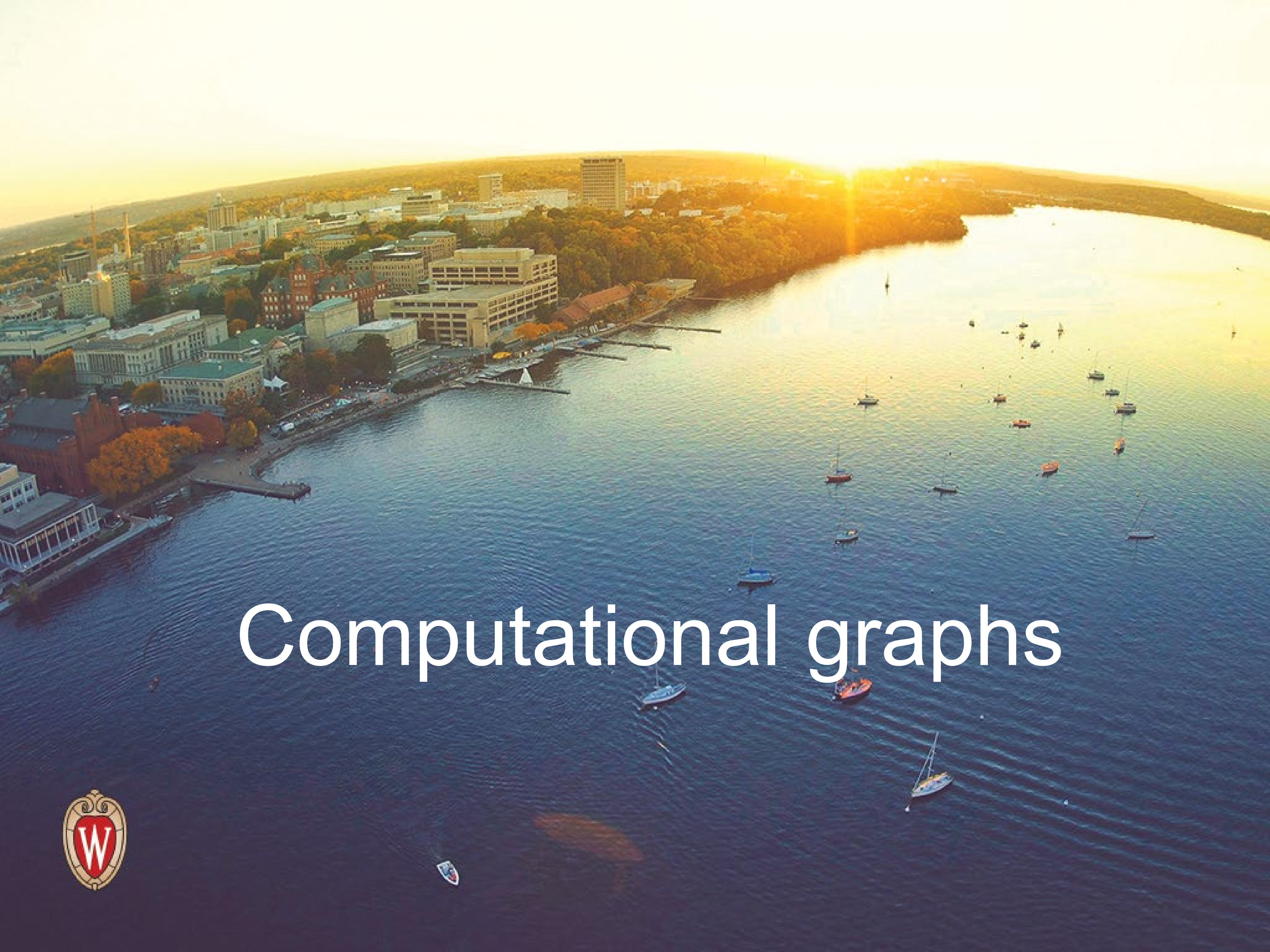
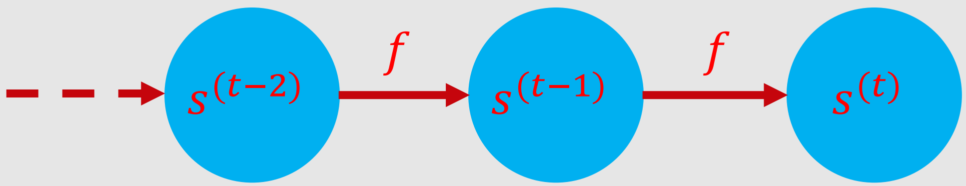



Figure from the paper “DenseCap: Fully Convolutional Localization Networks for Dense Captioning”, by Justin Johnson, Andrej Karpathy, Li Fei-Fei

# Computational graphs

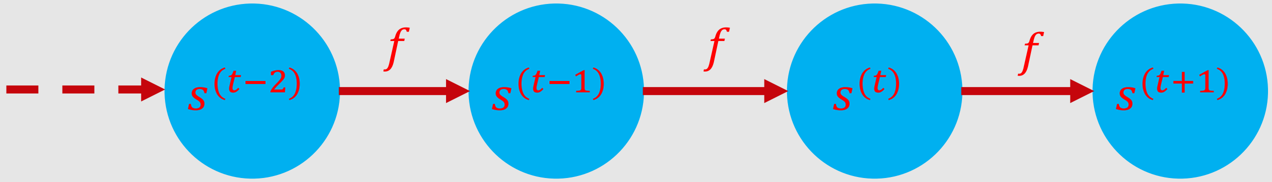


# A typical dynamic system



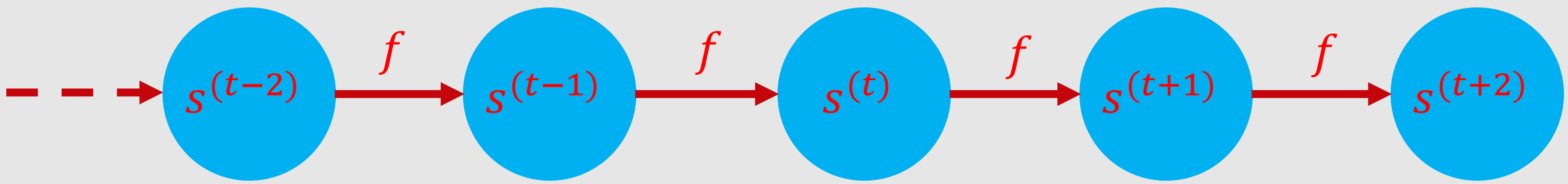

$$s^{(t+1)} = f(s^{(t)}; \theta)$$

# A typical dynamic system



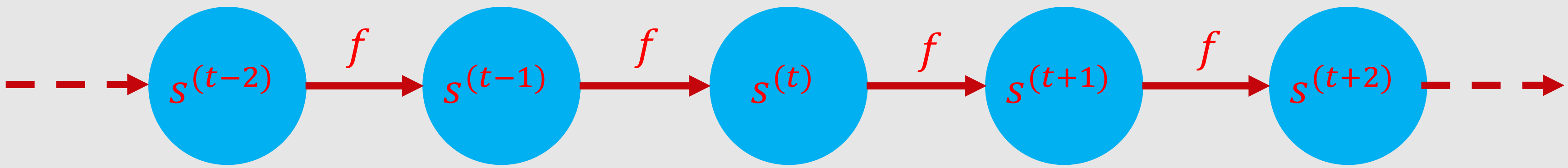

$$s^{(t+1)} = f(s^{(t)}; \theta)$$

# A typical dynamic system



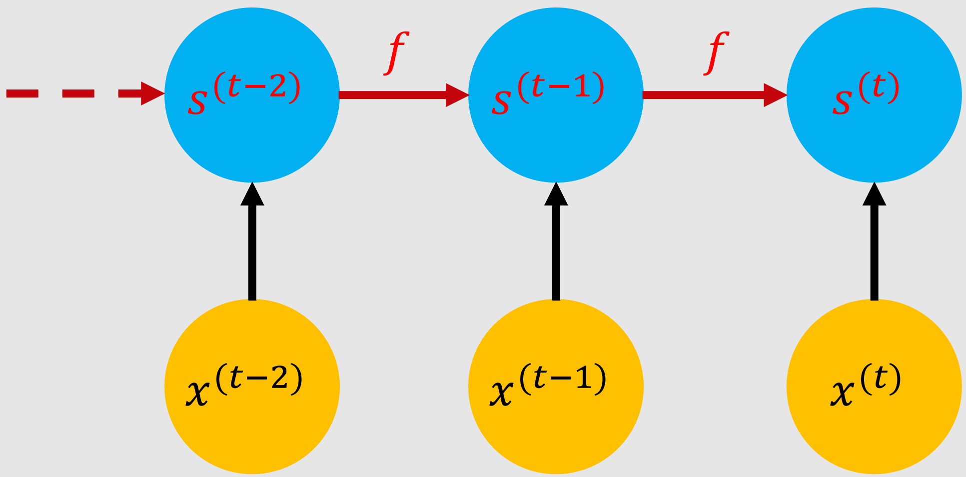

$$s^{(t+2)} = f(s^{(t+1)}; \theta)$$

# A typical dynamic system



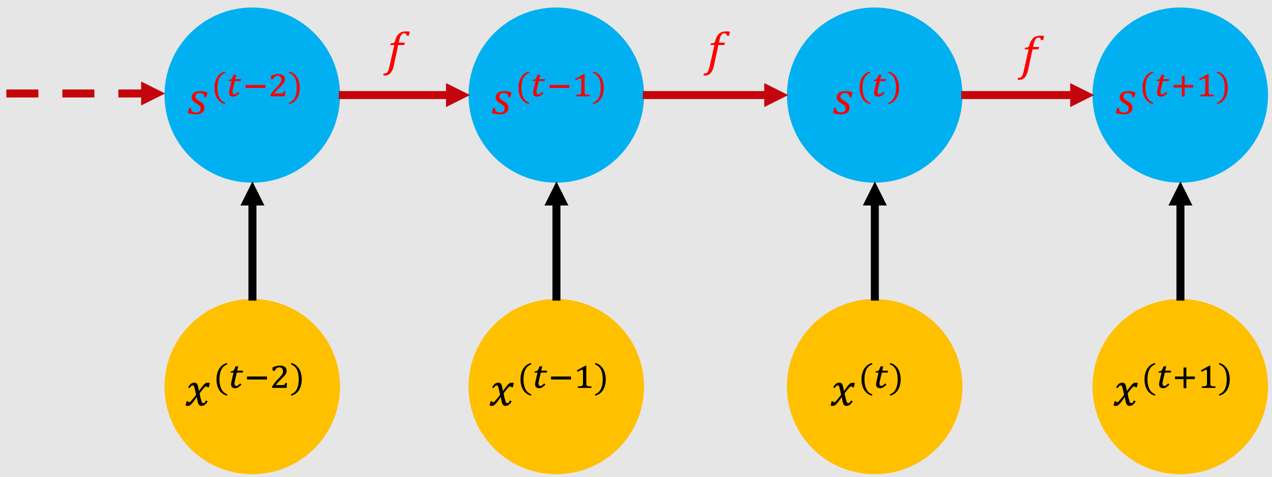

$$s^{(t+3)} = f\left(s^{(t+2)}; \theta\right), \ldots$$

# A dynamic system driven by external data



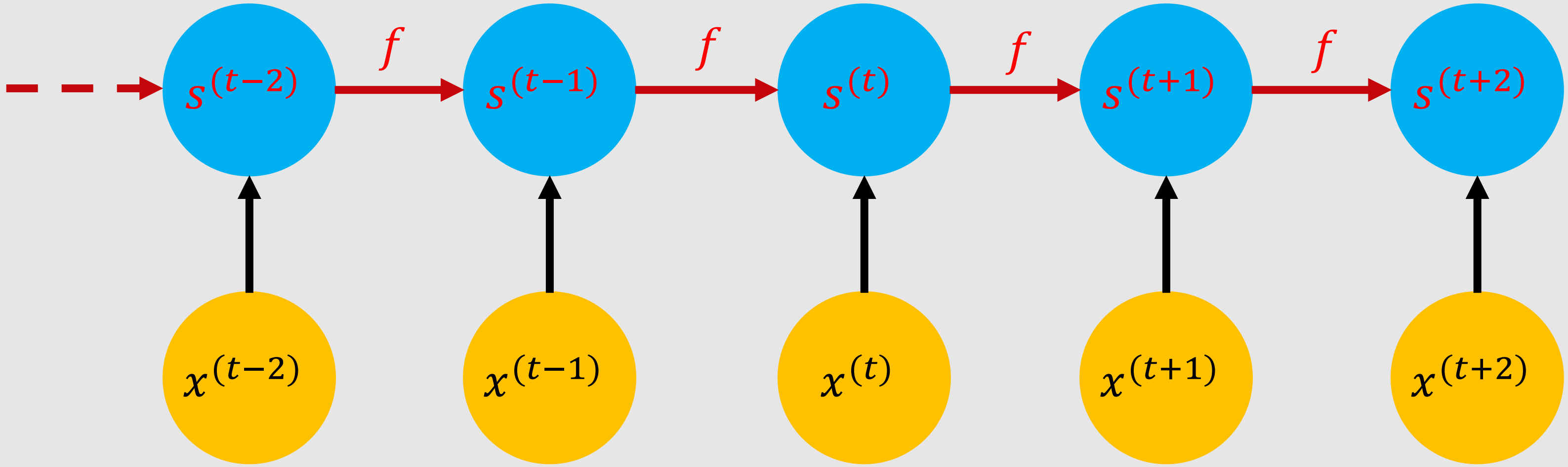

$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

# A dynamic system driven by external data



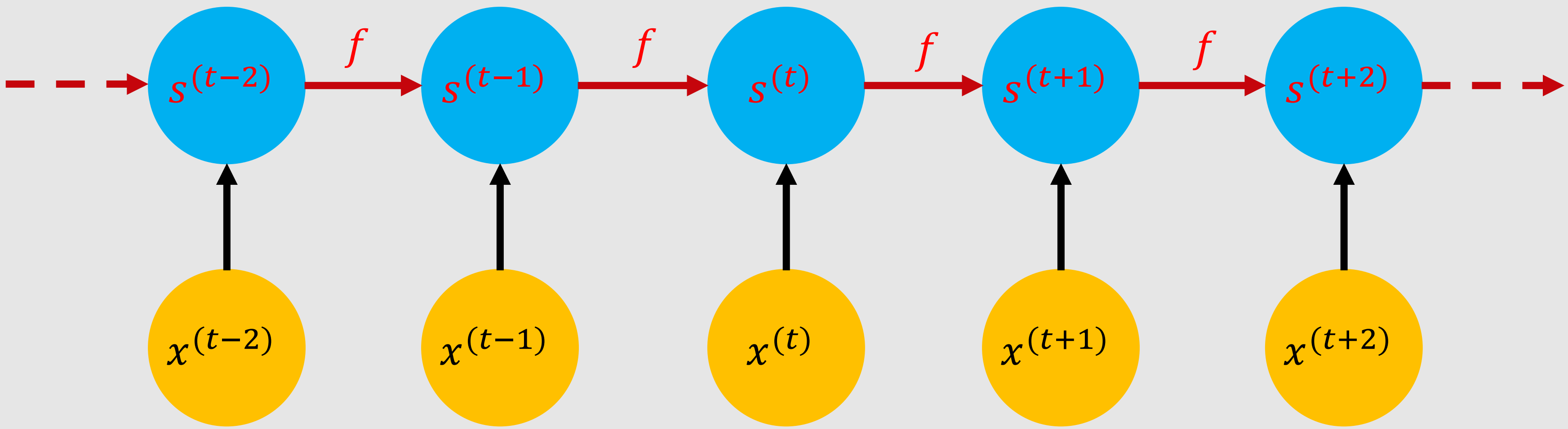

$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

# A dynamic system driven by external data



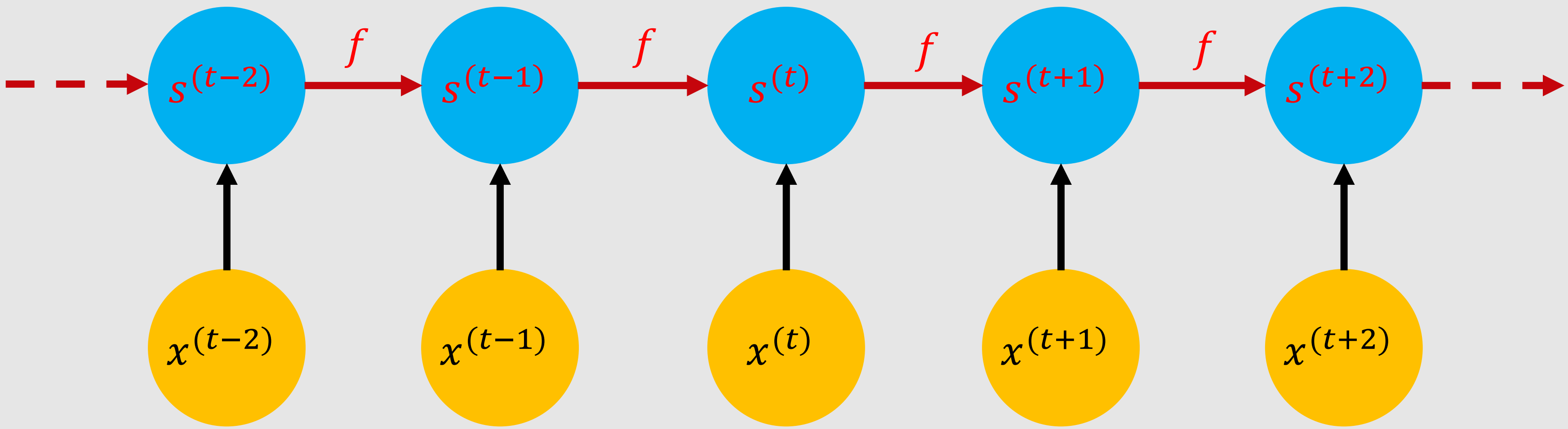

$$s^{(t+2)} = f(s^{(t+1)}, x^{(t+2)}; \theta)$$

# A dynamic system driven by external data



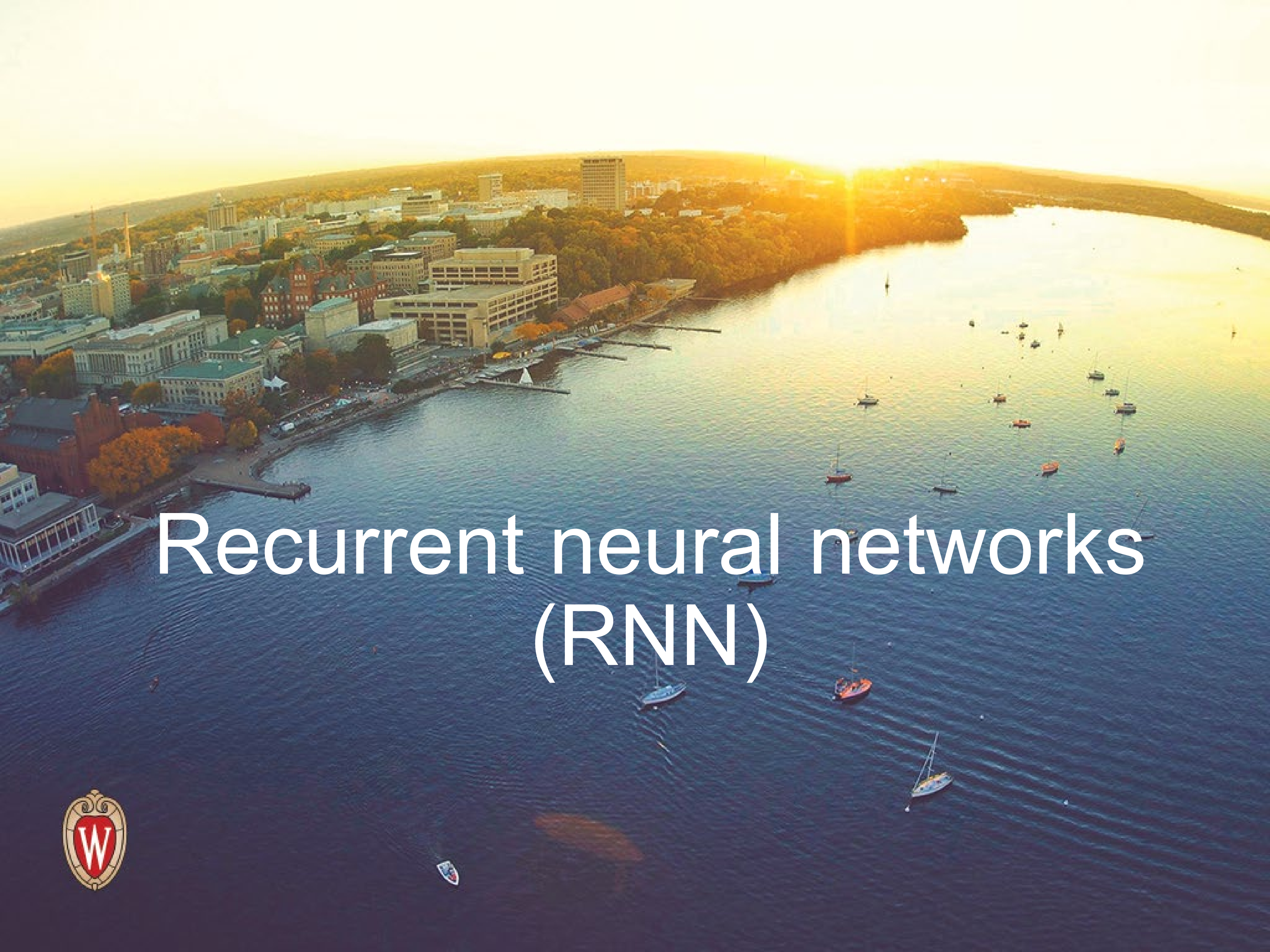

$$s^{(t+3)} = f\left(s^{(t+2)}, x^{(t+3)}; \theta\right), \ldots$$

# A dynamic system driven by external data




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

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

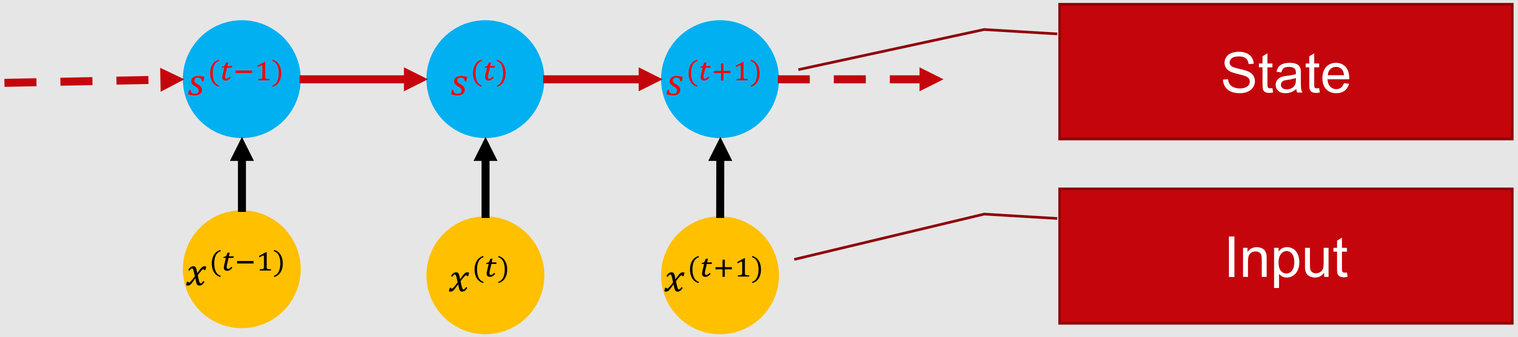


# Recurrent neural networks (RNN)

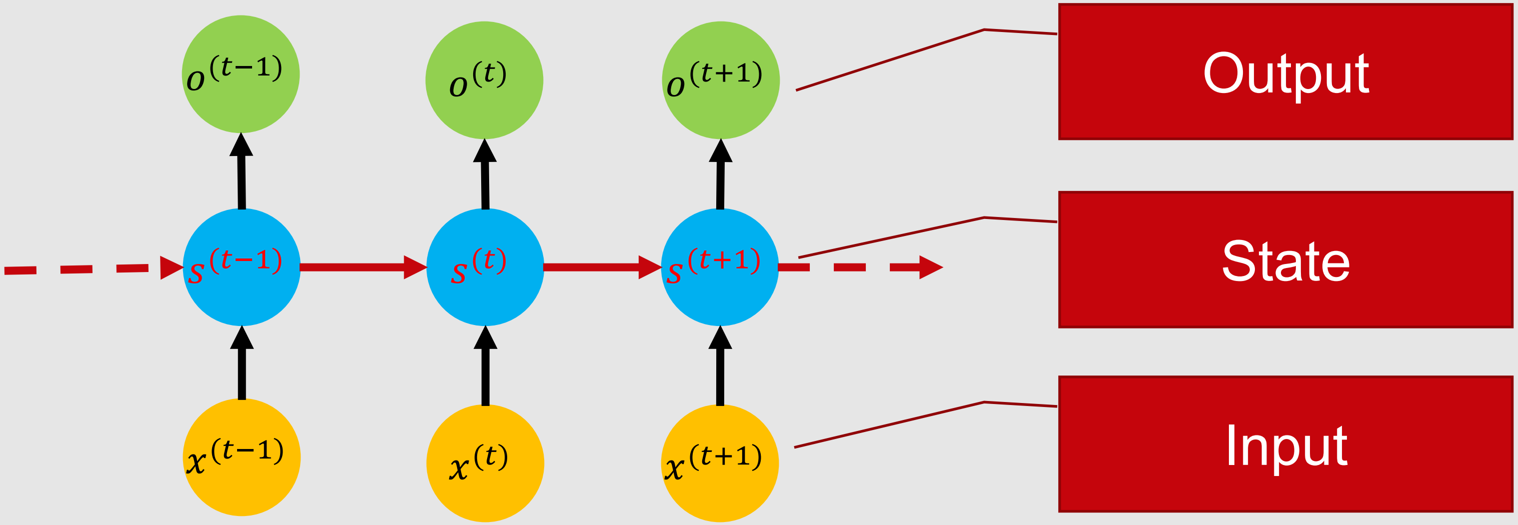
# Recurrent neural networks

- 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 current hidden state and the output entry
- Loss: typically computed at every time step

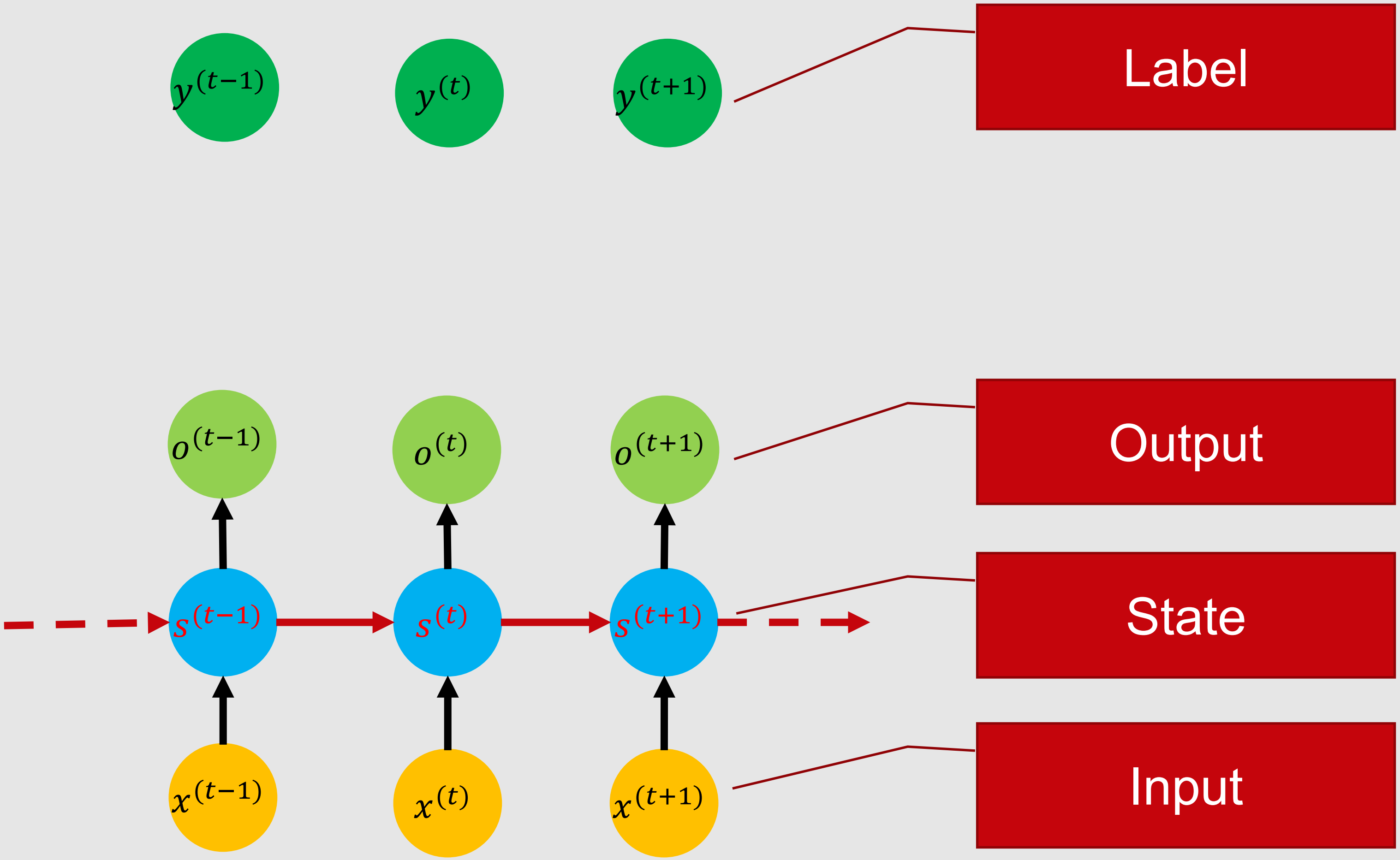
# Recurrent neural networks



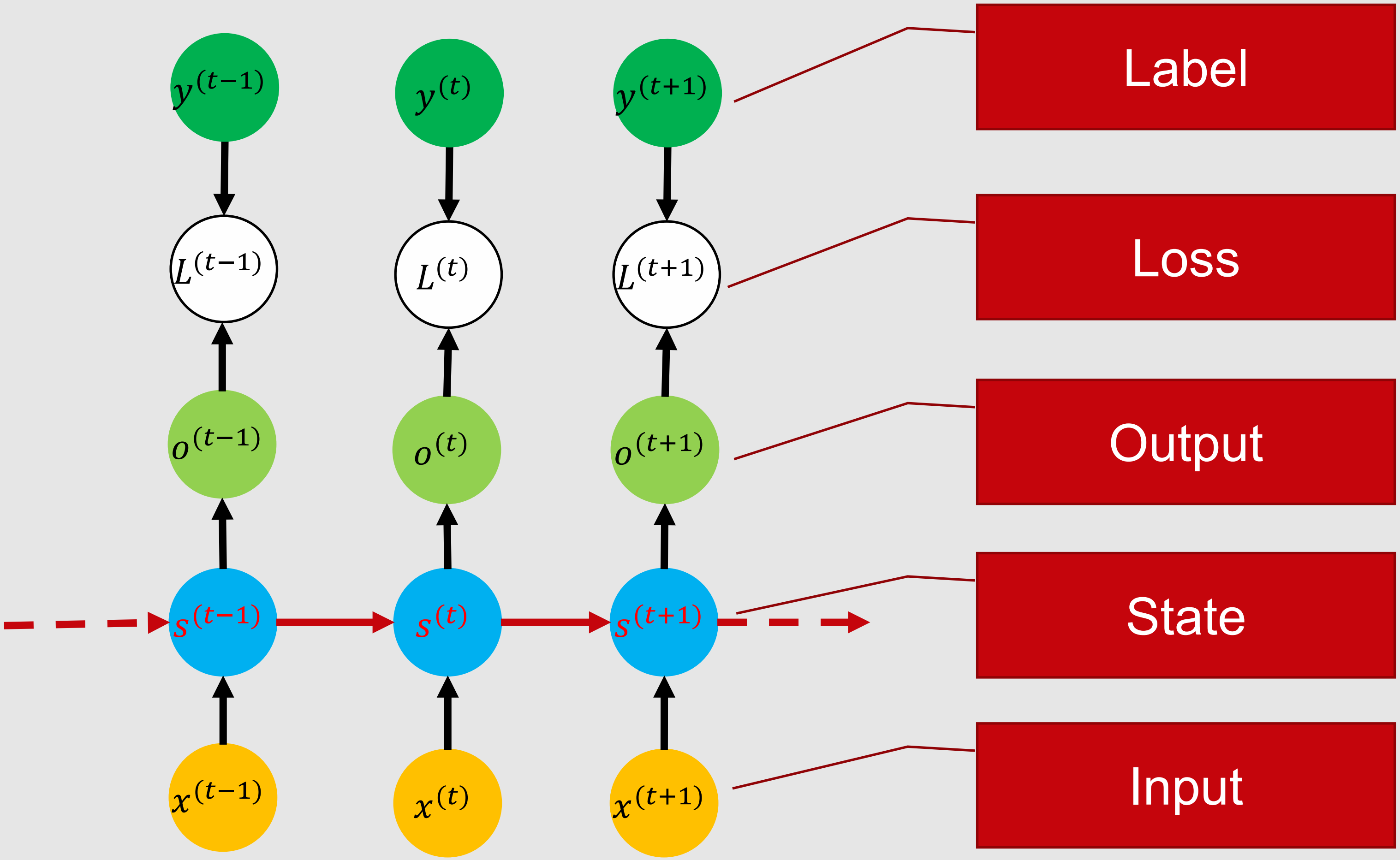
# Recurrent neural networks



# Recurrent neural networks

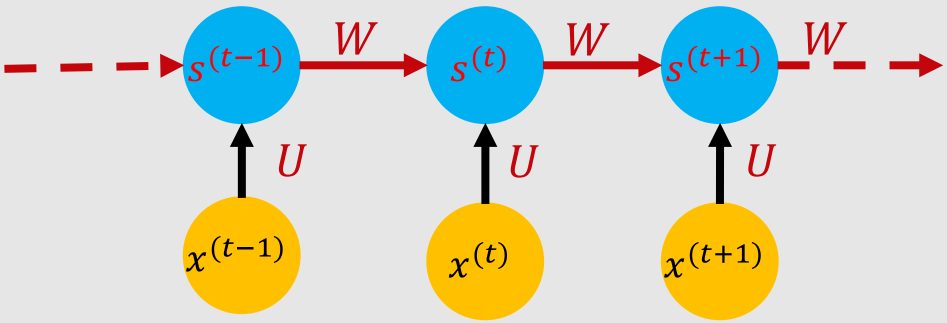


# Recurrent neural networks

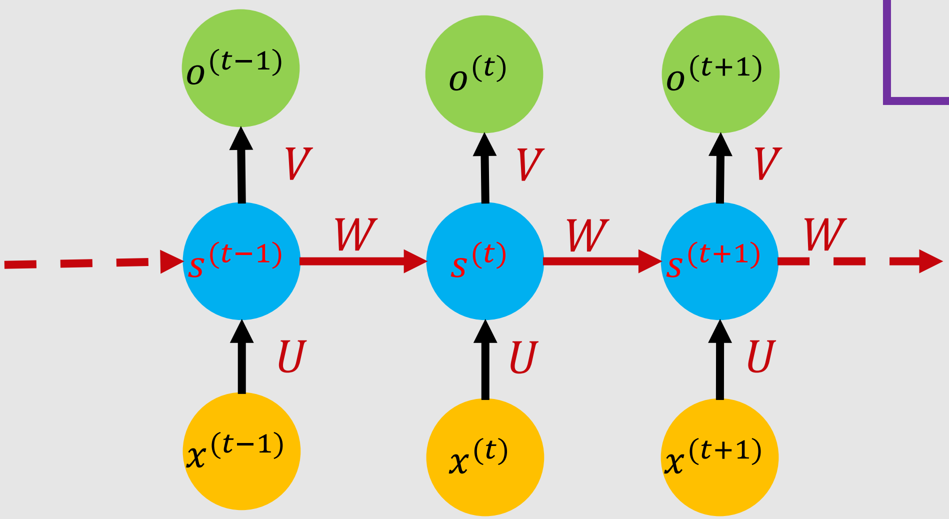


# Recurrent neural networks: standard version

$$a^{(t)} = b + Ws^{(t-1)} + Ux^{(t)}$$
$$s^{(t)} = \tanh\left(a^{(t)}\right)$$

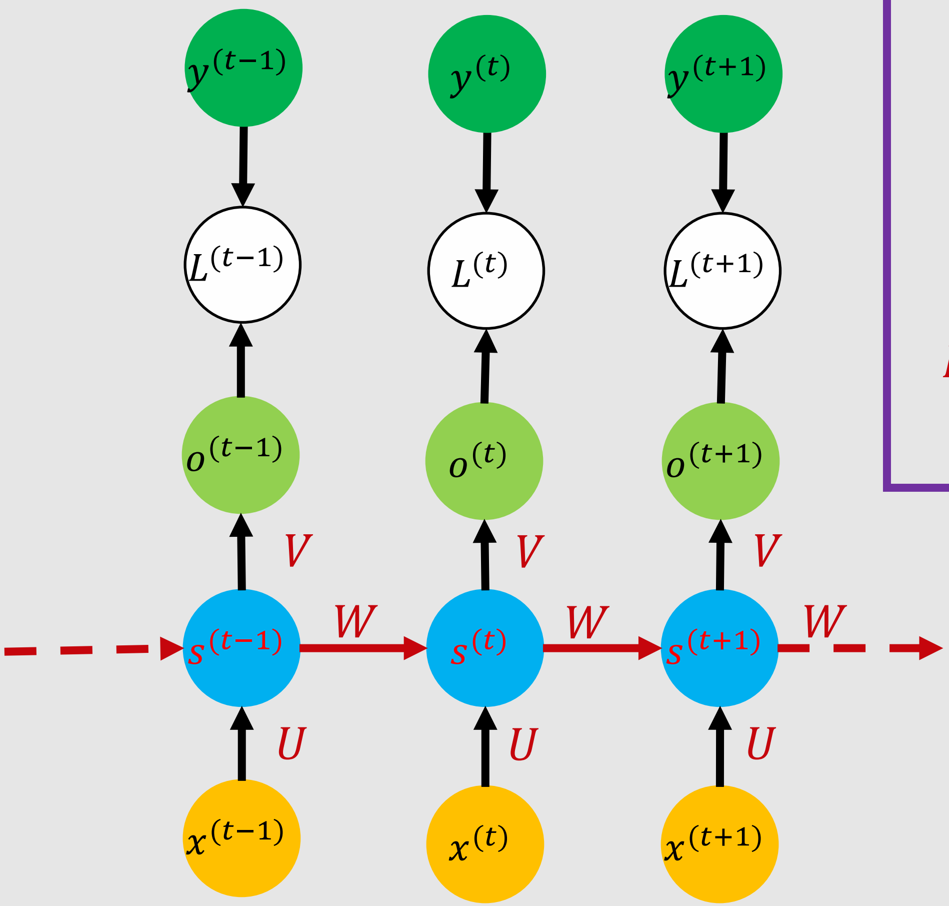


# Recurrent neural networks: standard version




$$a^{(t)} = b + Ws^{(t-1)} + Ux^{(t)}$$
$$s^{(t)} = \tanh\left(a^{(t)}\right)$$
$$o^{(t)} = c + Vs^{(t)}$$

# Recurrent neural networks: standard version




$$a^{(t)} = b + Ws^{(t-1)} + Ux^{(t)}$$
$$s^{(t)} = \tanh\left(a^{(t)}\right)$$
$$o^{(t)} = c + Vs^{(t)}$$
$$\hat{y}^{(t)} = \mathrm{softmax}\left(o^{(t)}\right)$$
$$L^{(t)} = \mathrm{CrossEntropy}(y^{(t)}, \hat{y}^{(t)})$$

# 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 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))

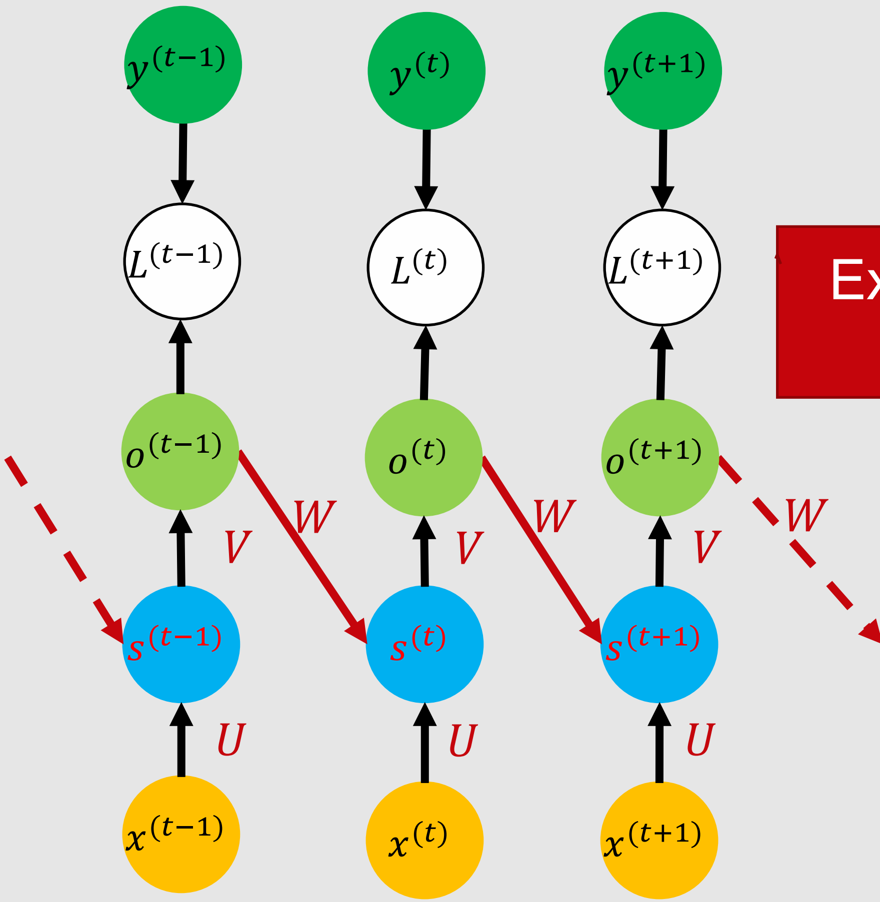
# Variants of RNN

# 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 current hidden state and 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

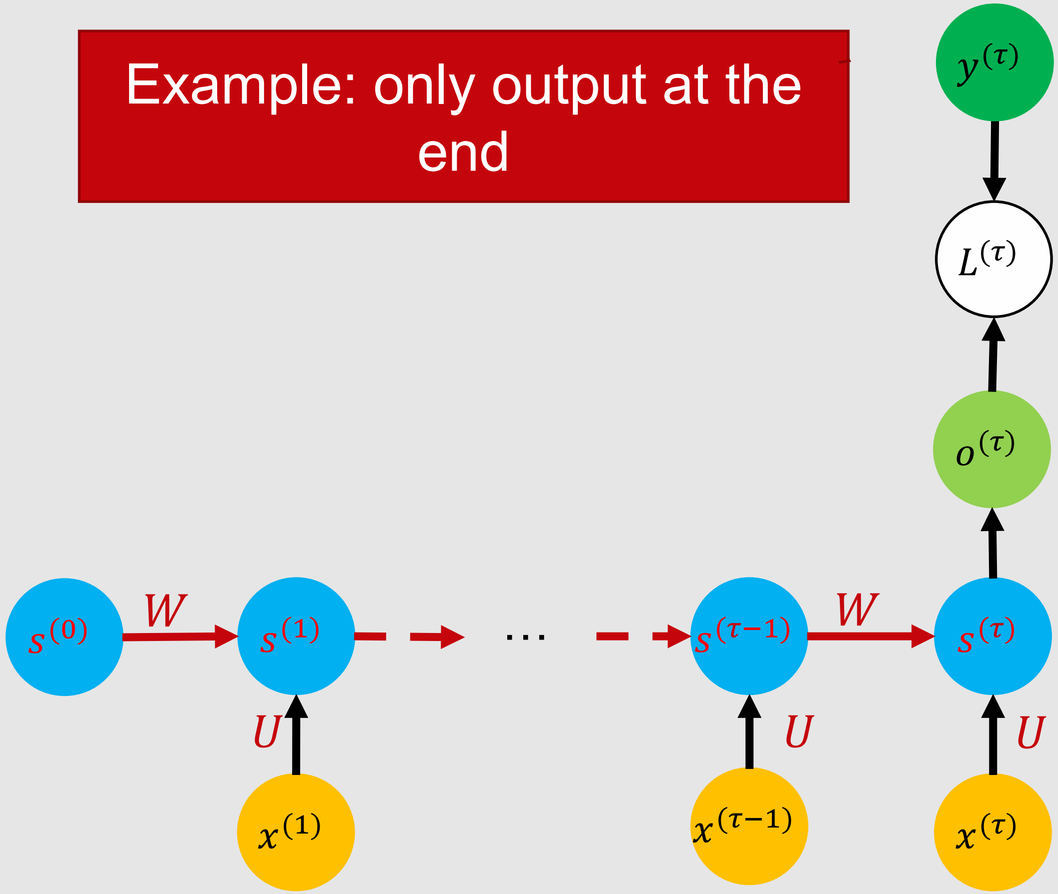
# Recurrent neural network variant




Example: use the output at the previous step

# Recurrent neural network variant

Example: only output at the end



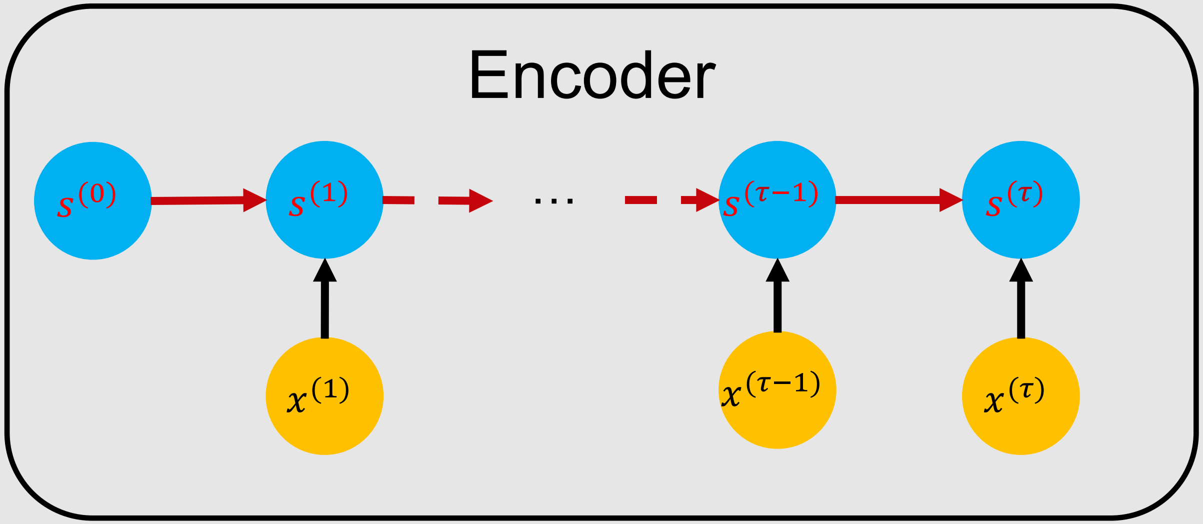
# Encoder-decoder RNNs

- RNNs: can map sequence to one vector; or to sequence of same length

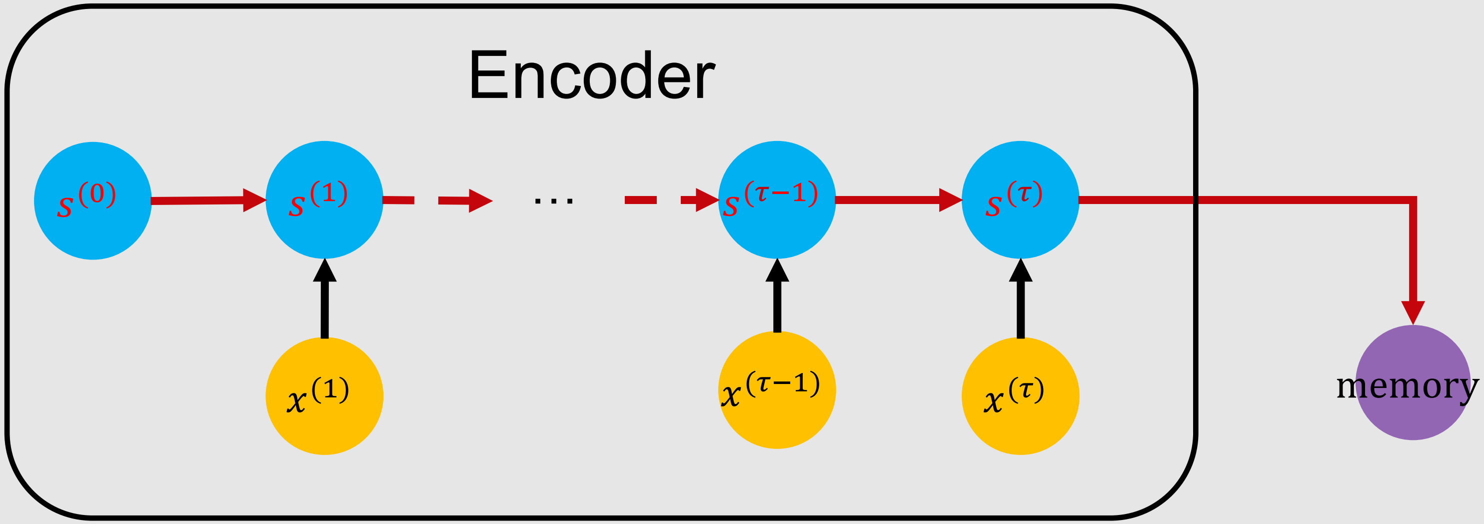
- What about mapping sequence to sequence of different length?

- Example: speech recognition, machine translation, question answering, etc.

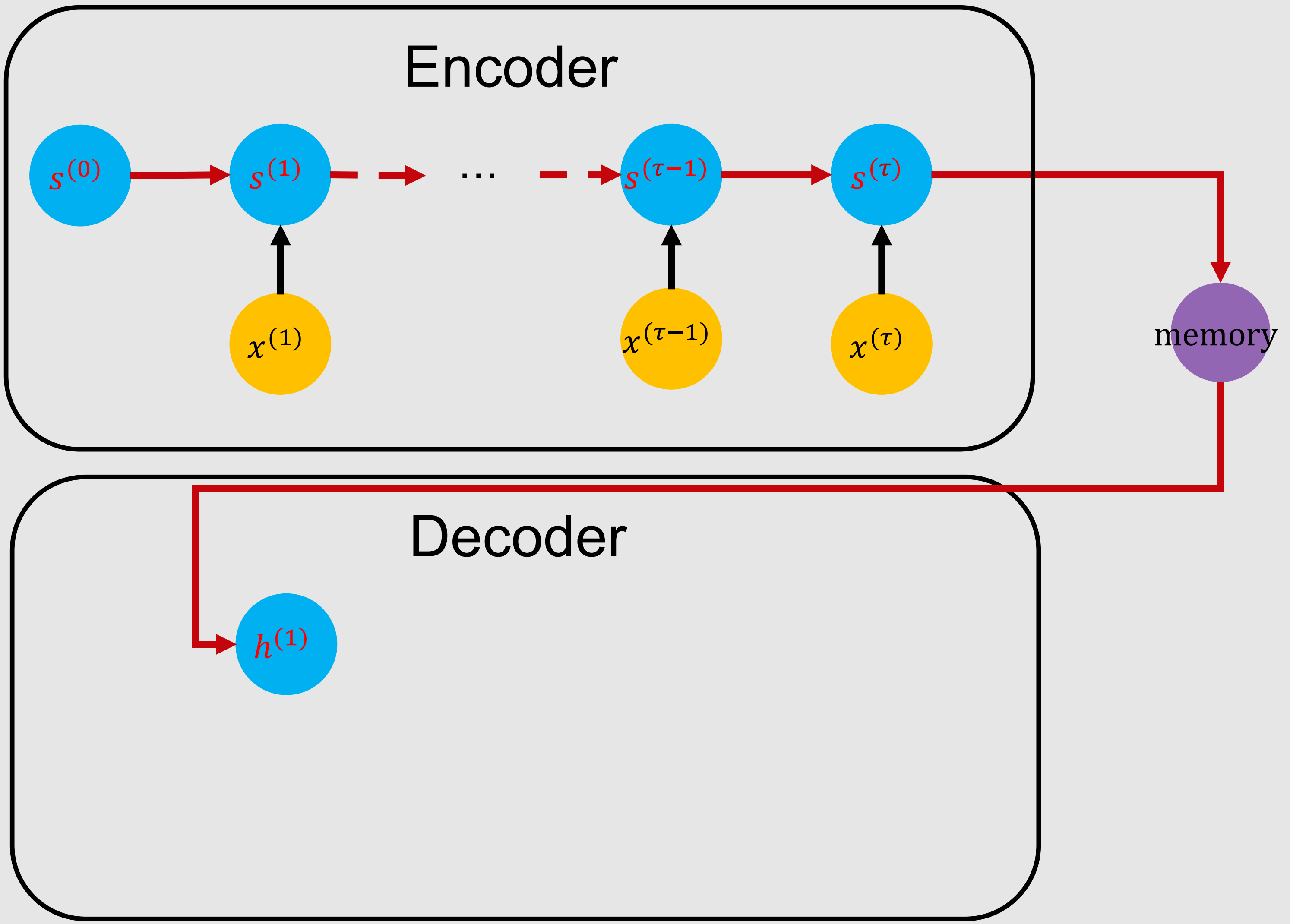
# Encoder-decoder RNNs



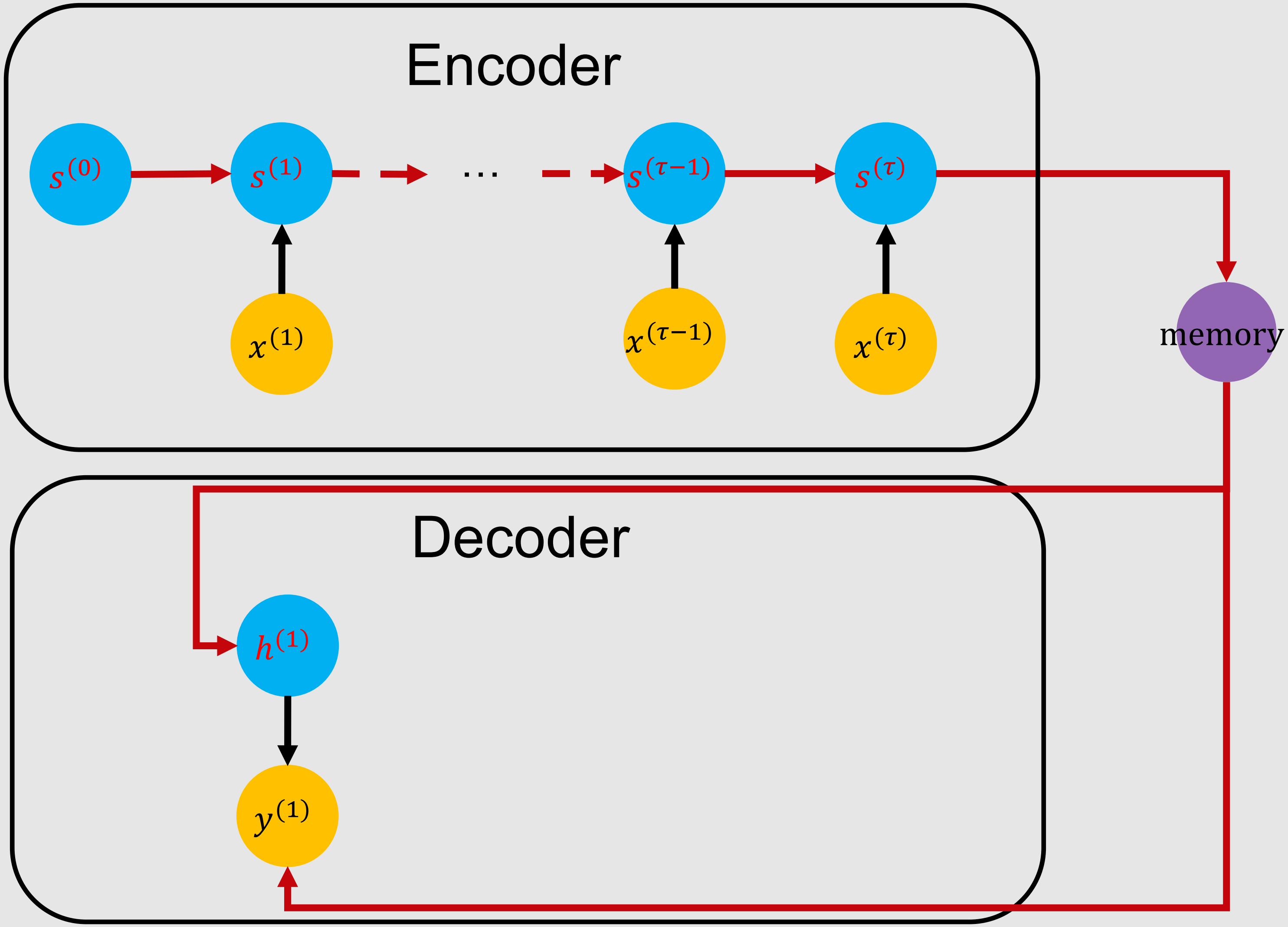
# Encoder-decoder RNNs



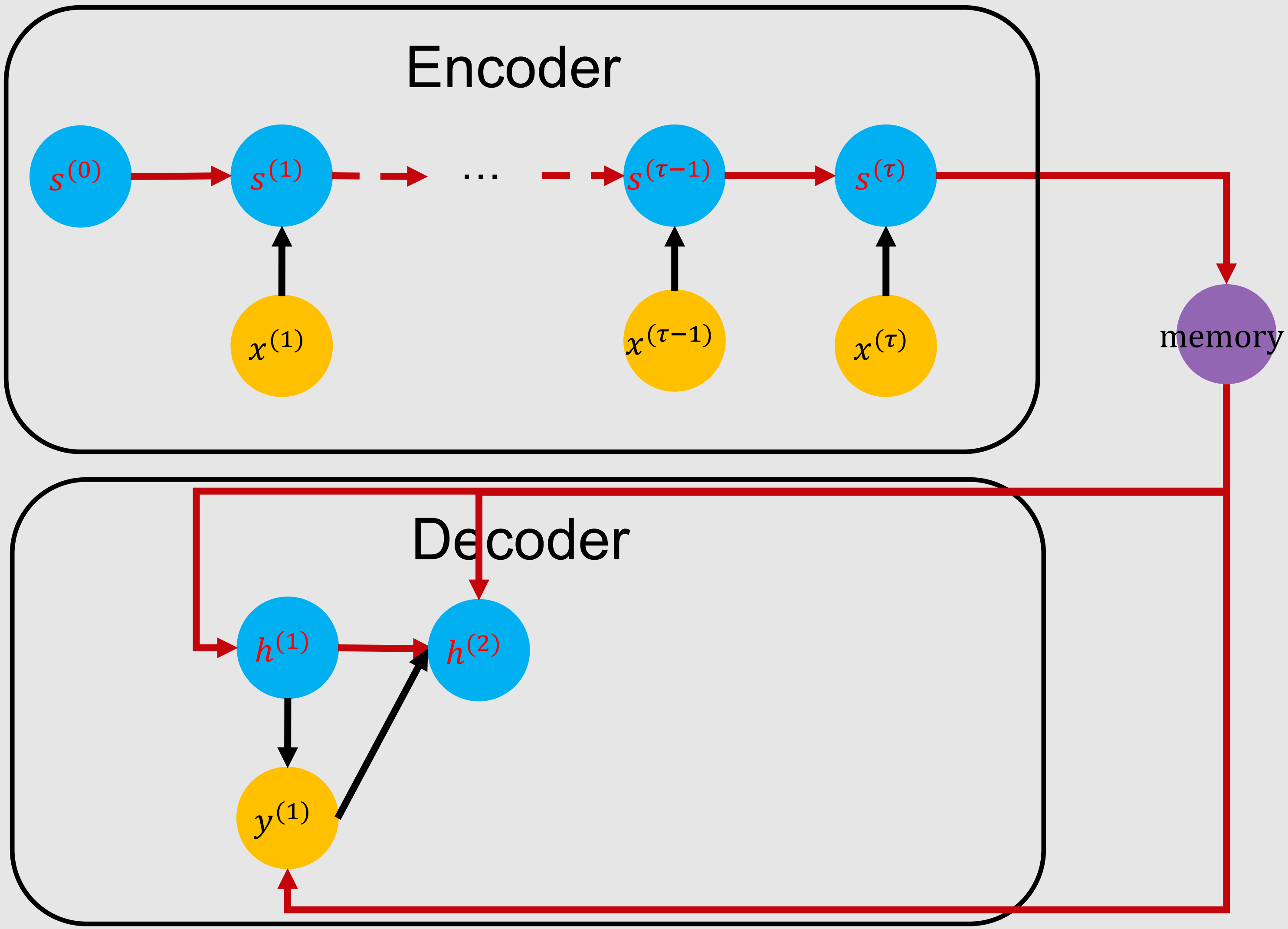
# Encoder-decoder RNNs



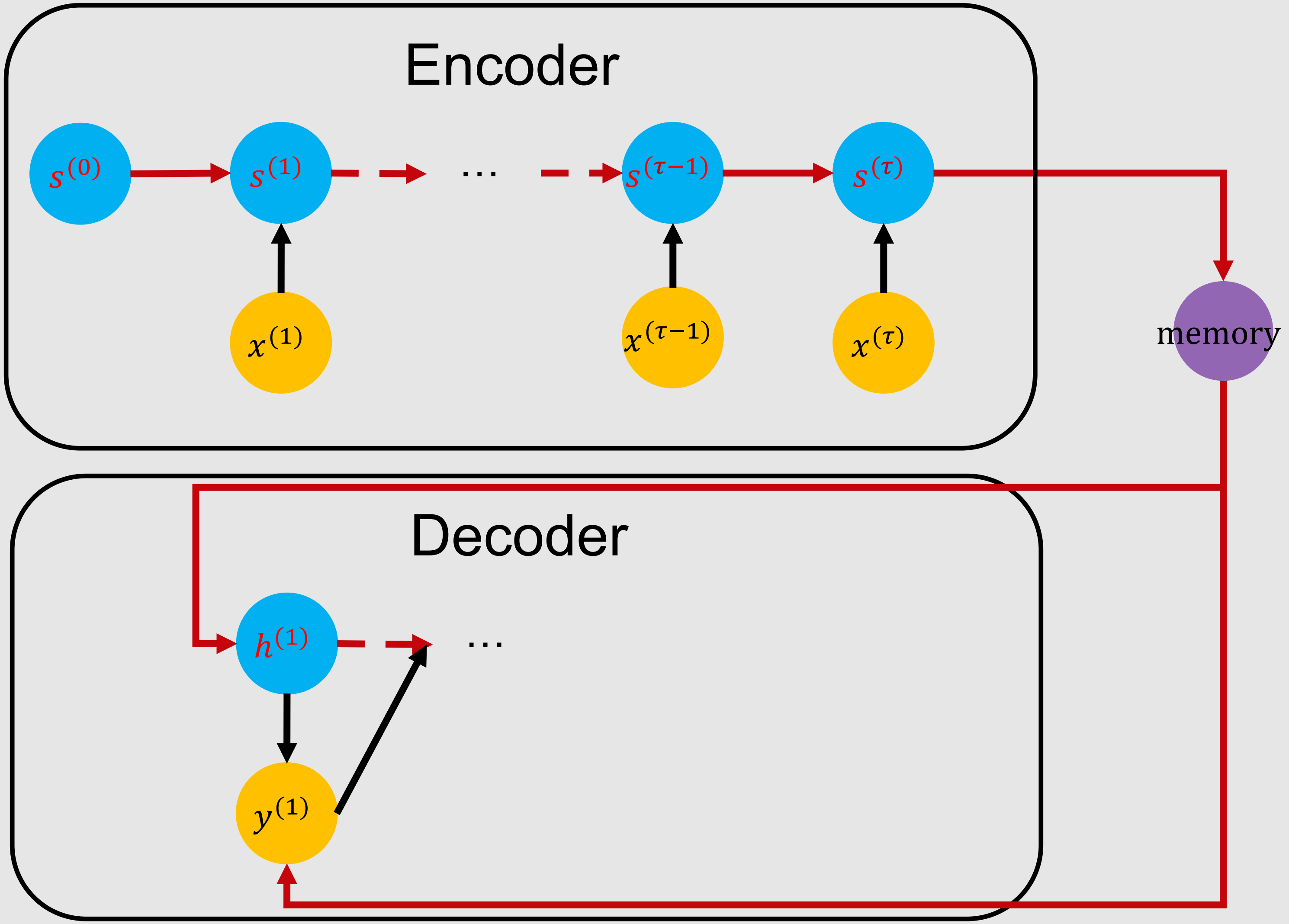
# Encoder-decoder RNNs



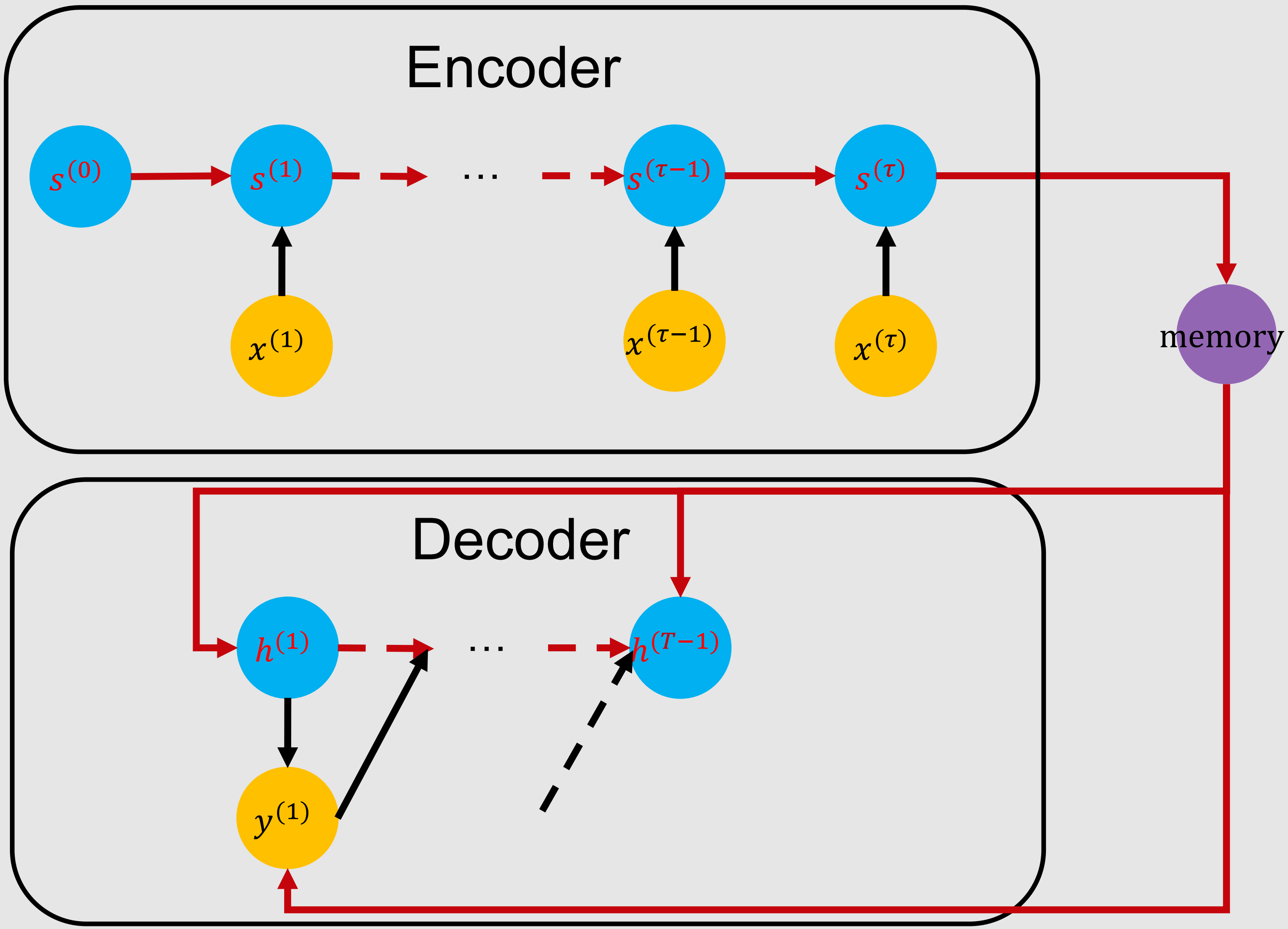
# Encoder-decoder RNNs



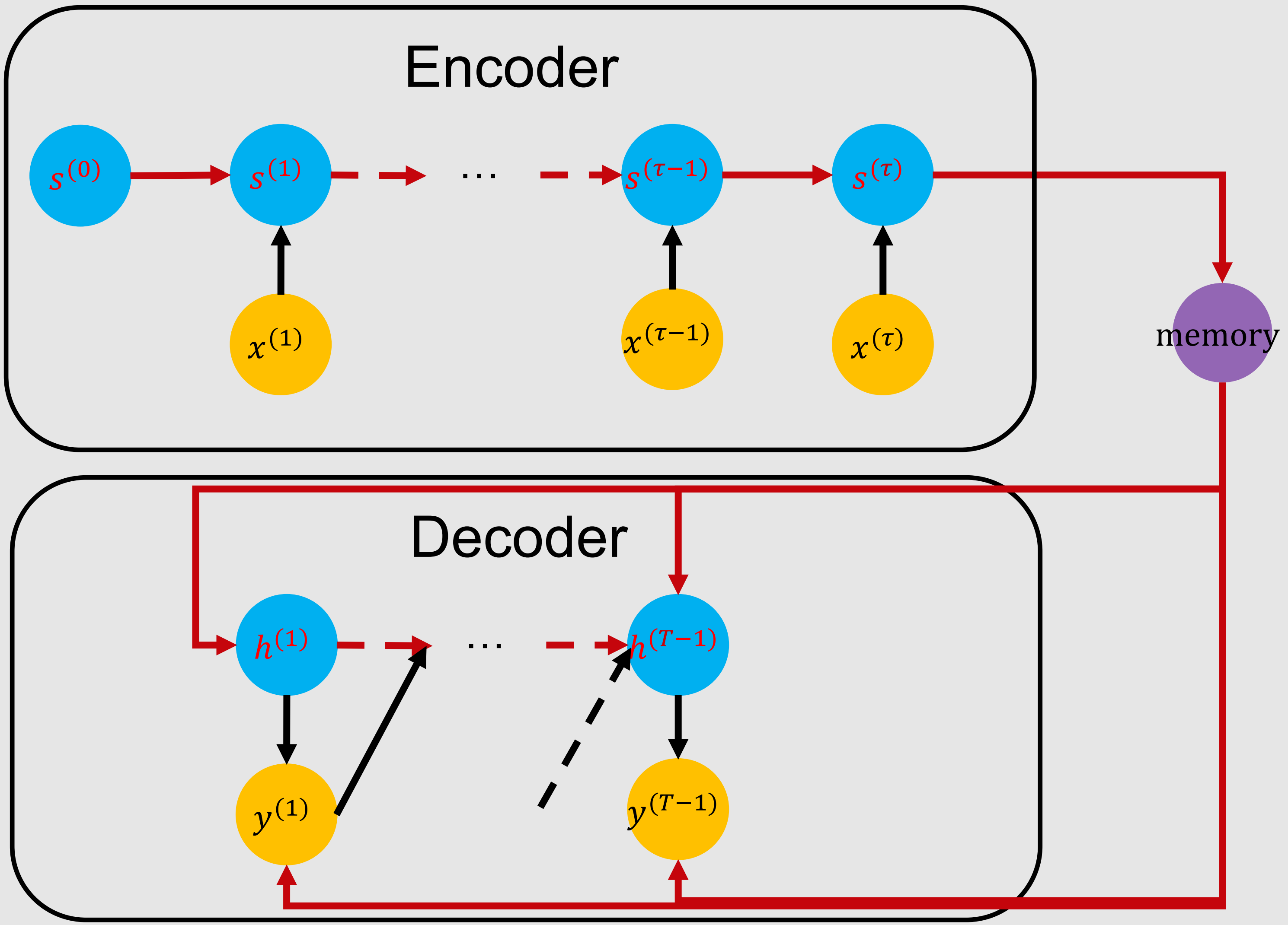
# Encoder-decoder RNNs



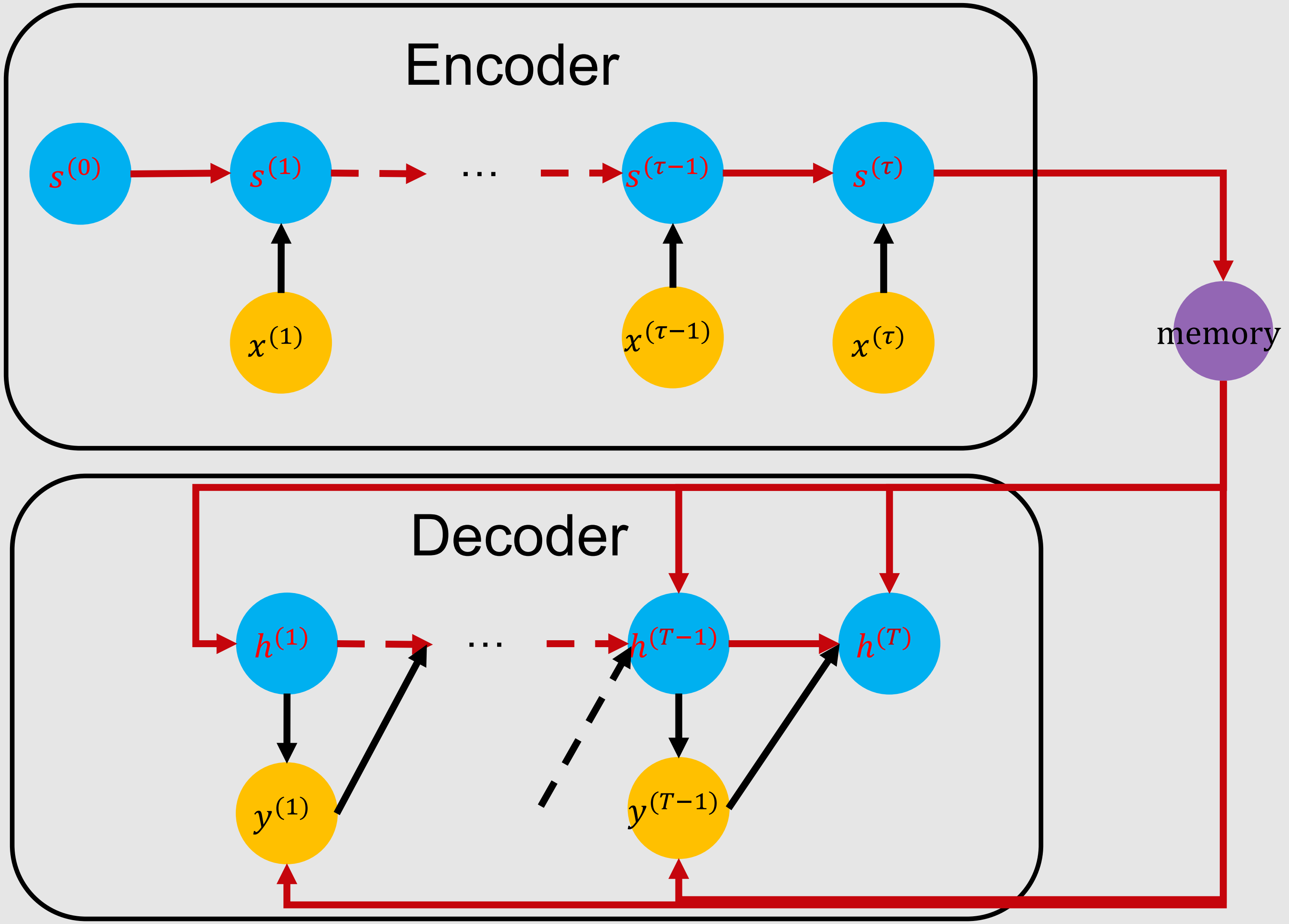
# Encoder-decoder RNNs



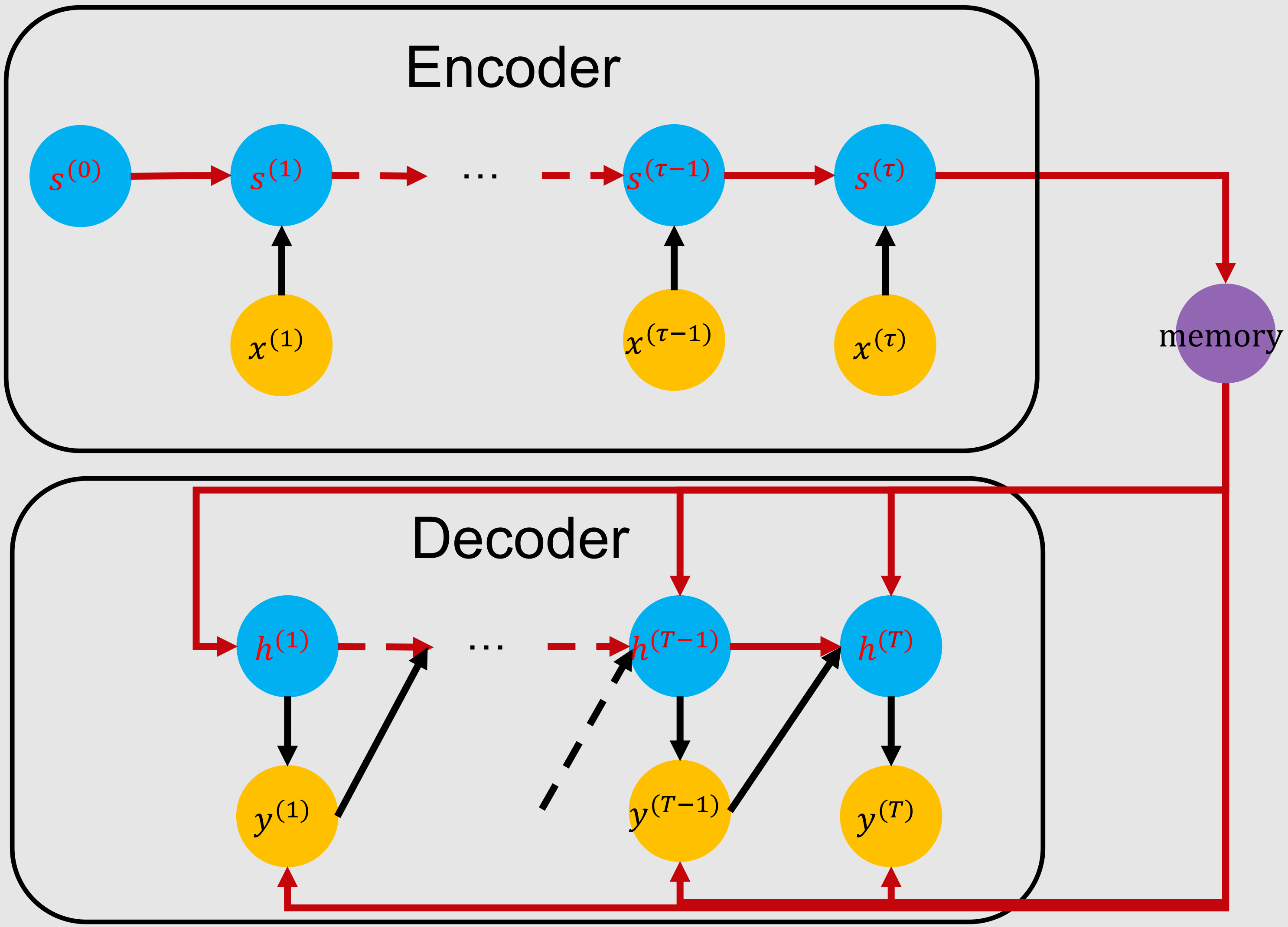
# Encoder-decoder RNNs



# Encoder-decoder RNNs



# Encoder-decoder RNNs



# Optional: Training RNN

# Training RNN

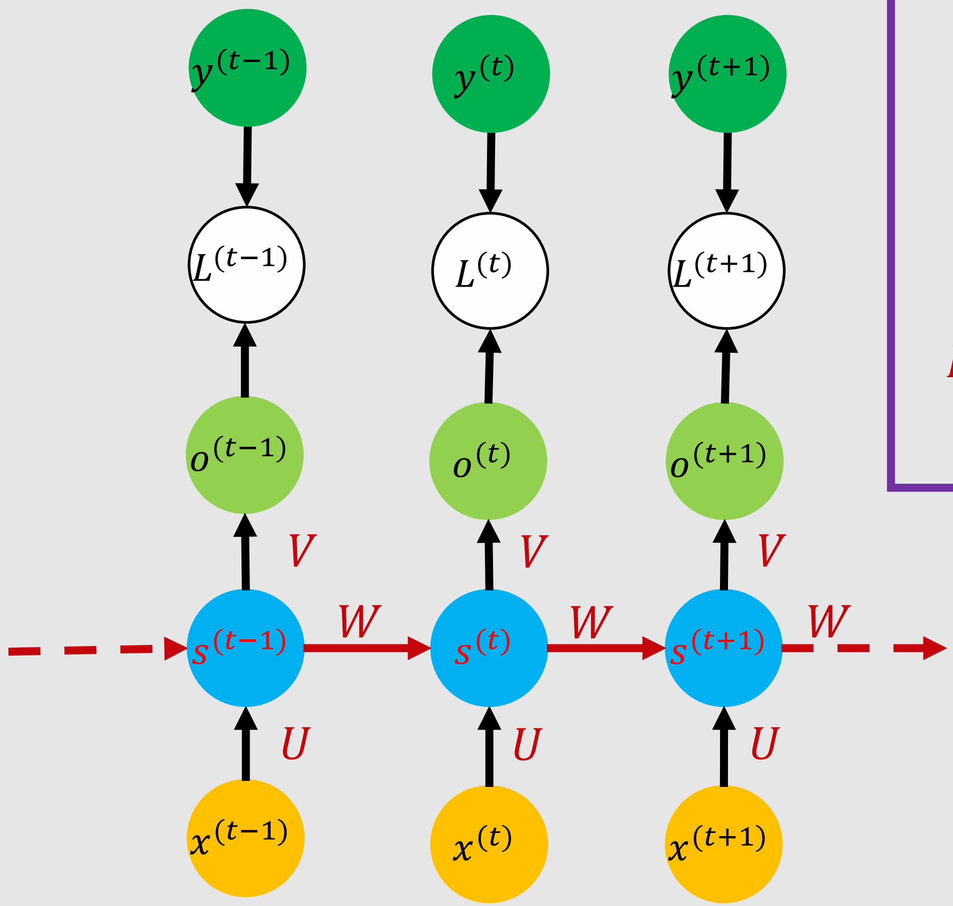
- Principle: unfold the computational graph, and use backpropagation
- Called back-propagation through time (BPTT) algorithm
- Can then apply any general-purpose gradient-based techniques

# 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

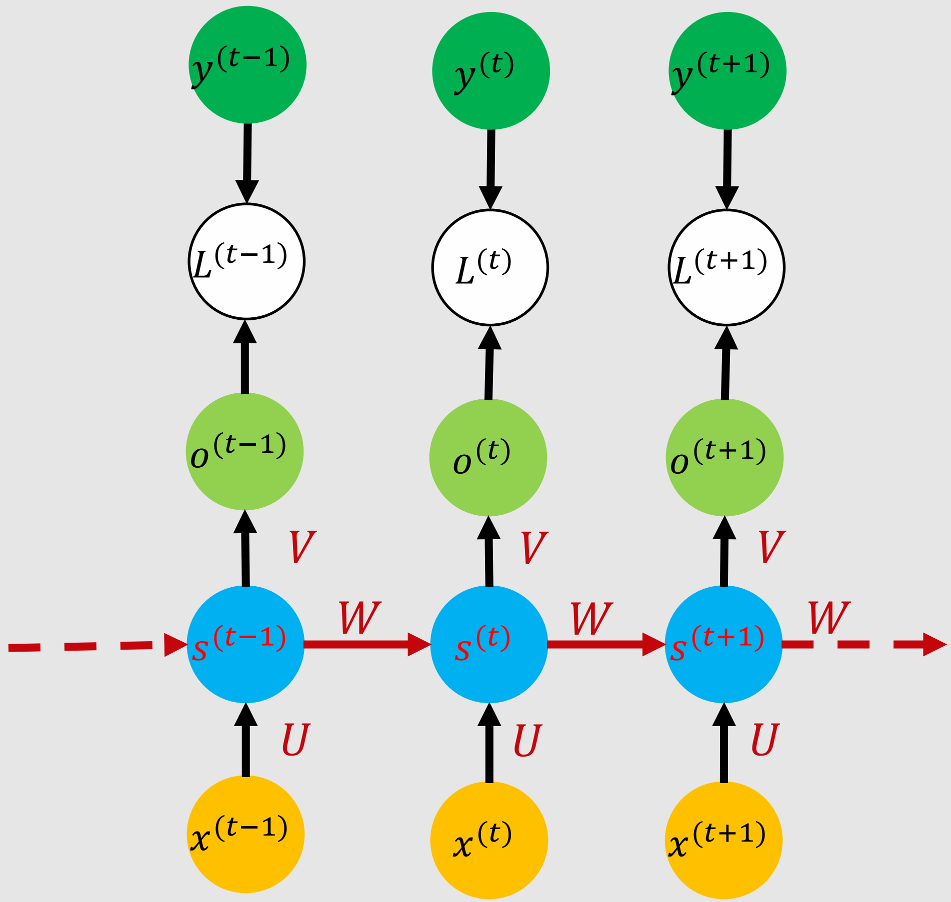
- Conceptually: first compute the gradients of the internal nodes, then compute the gradients of the parameters

# Recurrent neural networks




$$a^{(t)} = b + Ws^{(t-1)} + Ux^{(t)}$$
$$s^{(t)} = \tanh\left(a^{(t)}\right)$$
$$o^{(t)} = c + Vs^{(t)}$$
$$\hat{y}^{(t)} = \mathrm{softmax}\left(o^{(t)}\right)$$
$$L^{(t)} = \mathrm{CrossEntropy}(y^{(t)}, \hat{y}^{(t)})$$

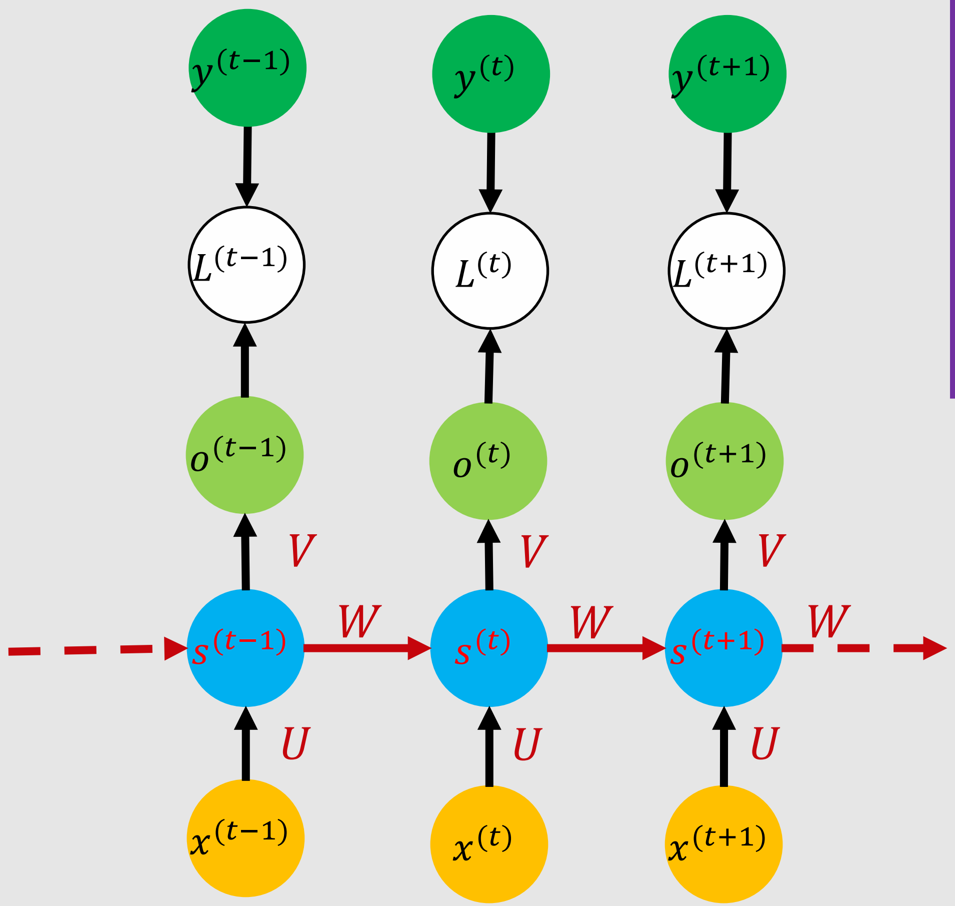
# Recurrent neural networks




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

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

# Recurrent neural networks

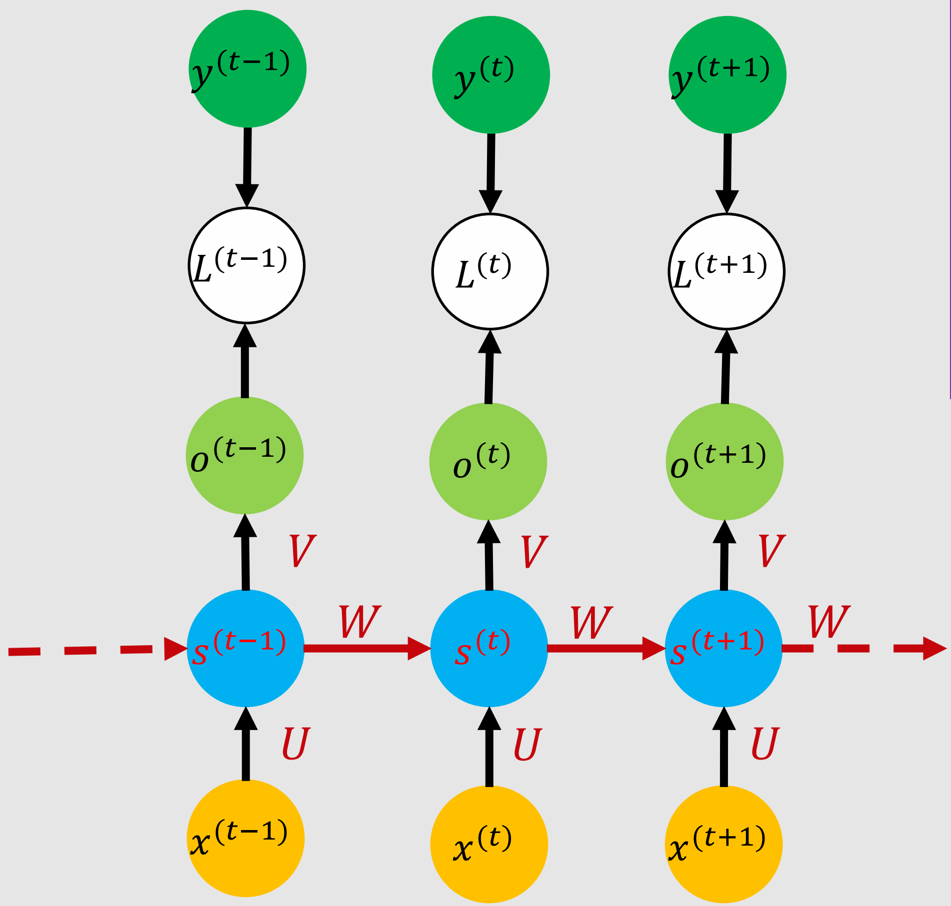



Gradient at $o^{(t)}$:

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

$$1_{i,y^{(t)}} = \begin{cases} 1, & y^{(t)} = i \\ 0, & \text{otherwise} \end{cases}$$

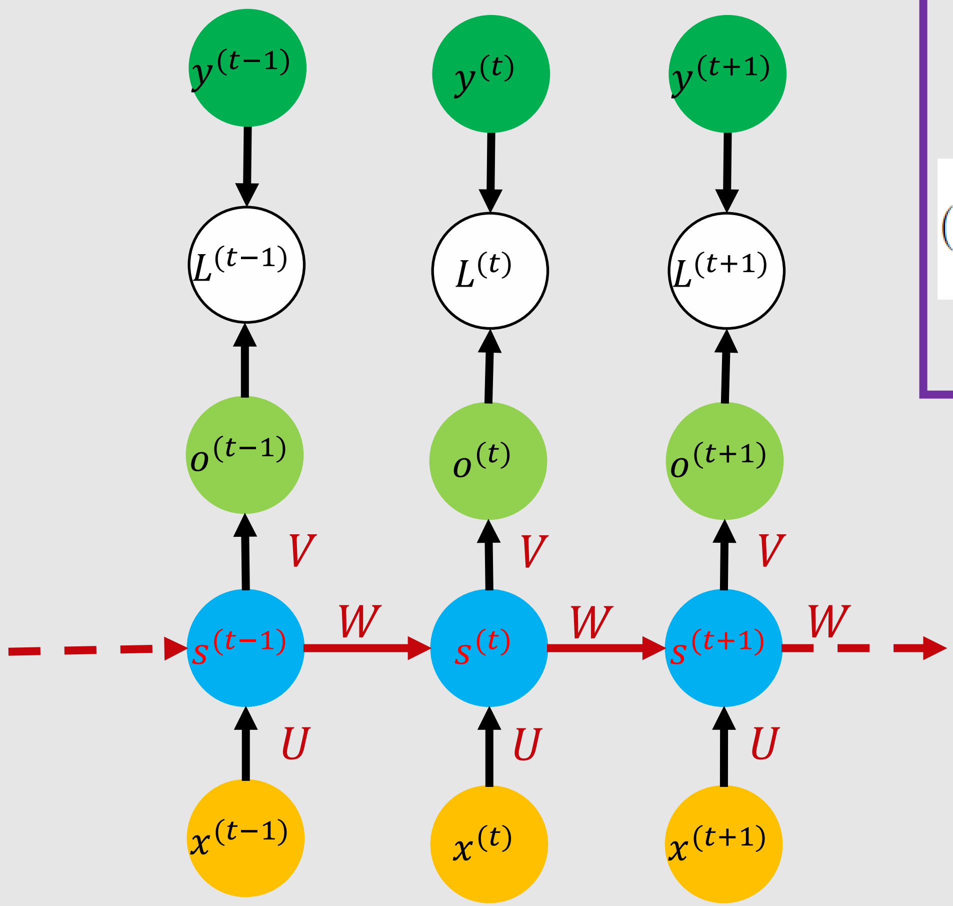
# Recurrent neural networks




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

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

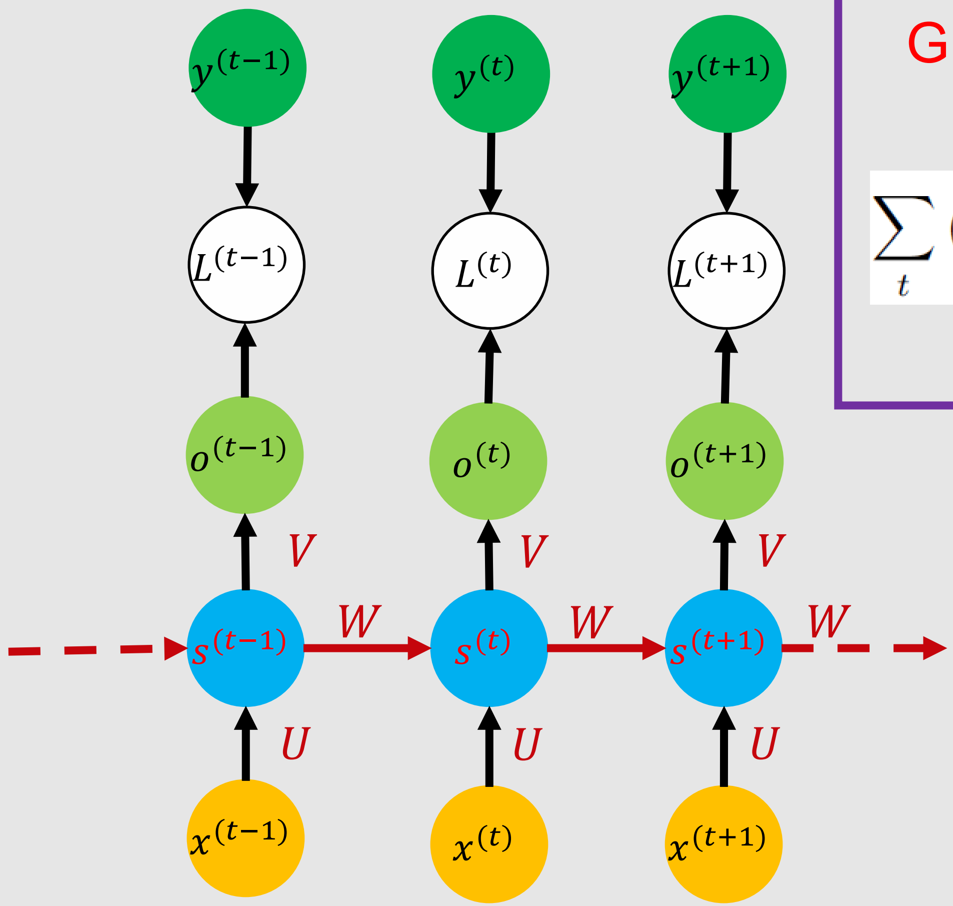
# Recurrent neural networks




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

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

# Recurrent neural networks



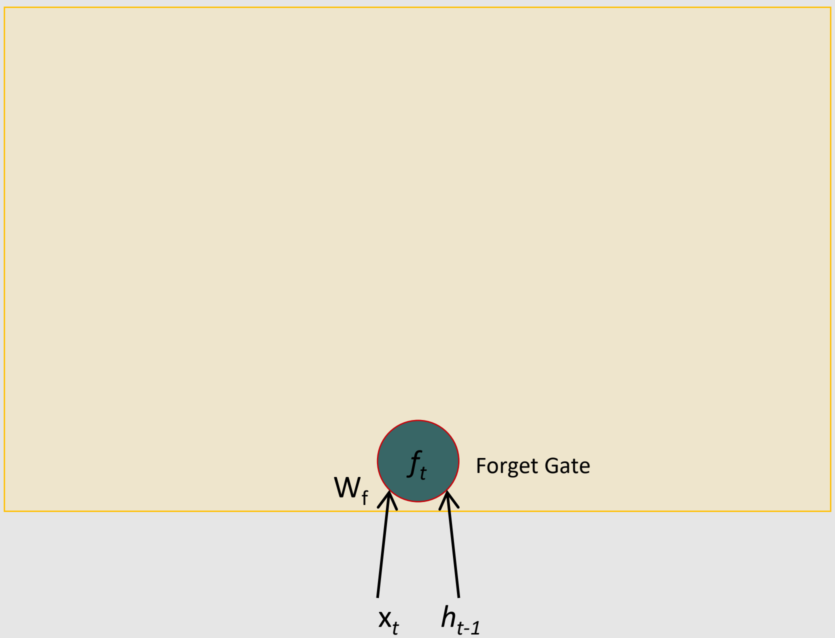

Gradient at parameter $V$:

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

# 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.

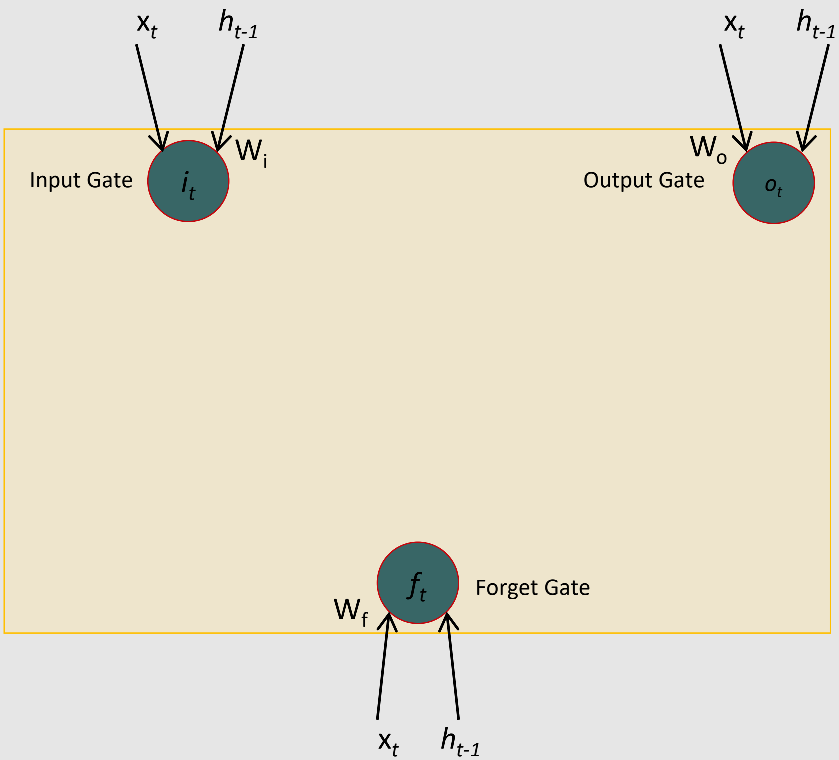
# The Popular LSTM Cell




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

* Dashed line indicates time-lag, and ⊗ is element-wise multiplication

# The Popular LSTM Cell

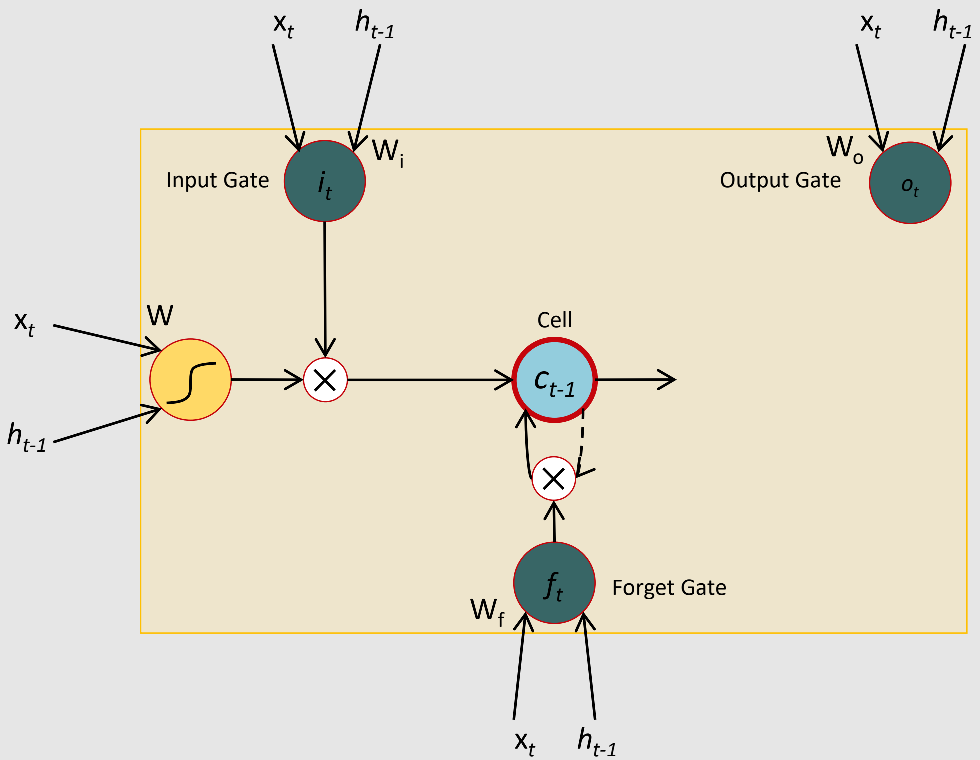



$$f_t = \sigma\left( W_f \begin{pmatrix} x_t \\ h_{t-1} \end{pmatrix} + b_f \right)$$

Similarly for $i_t$, $o_t$

* Dashed line indicates time-lag, and ⊗ is element-wise multiplication

# The Popular LSTM Cell



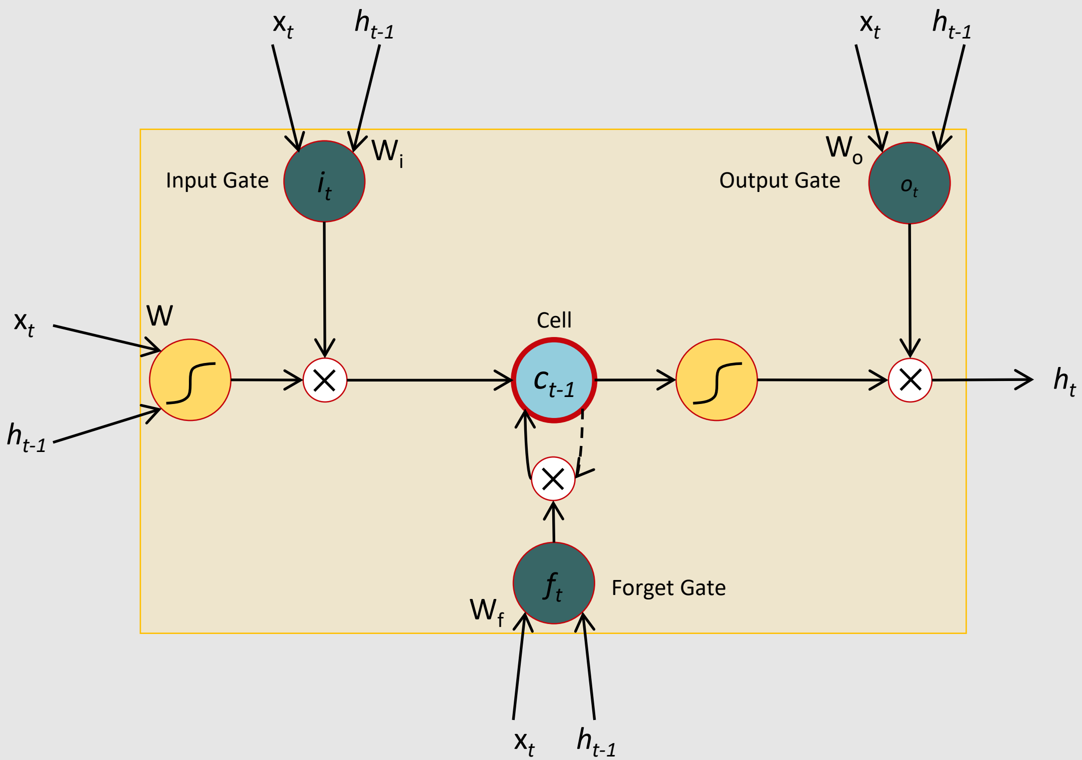

$$f_t = \sigma\left(W_f\begin{pmatrix}x_t\\h_{t-1}\end{pmatrix} + b_f\right)$$

Similarly for $i_t$, $o_t$

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

* Dashed line indicates time-lag, and ⊗ is element-wise multiplication

# The Popular LSTM Cell




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

Similarly for $i_t$, $o_t$

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

$$h_t = o_t \otimes \tanh c_t$$

* Dashed line indicates time-lag, and ⊗ is element-wise multiplication

# THANK YOU

Some of the slides in these lectures have been adapted/borrowed from materials developed by Mark Craven, David Page, Jude Shavlik, Tom Mitchell, Nina Balcan, Matt Gormley, Elad Hazan, Tom Dietterich, Pedro Domingos, and Geoffrey Hinton.