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
- training recurrent neural networks
- LSTM and GRU
- encoder-decoder RNNs

Introduction

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 τ
- Label: can be a scalar, a vector, or even a sequence

- Example
  - Sentiment analysis
  - Machine translation

Example: machine translation

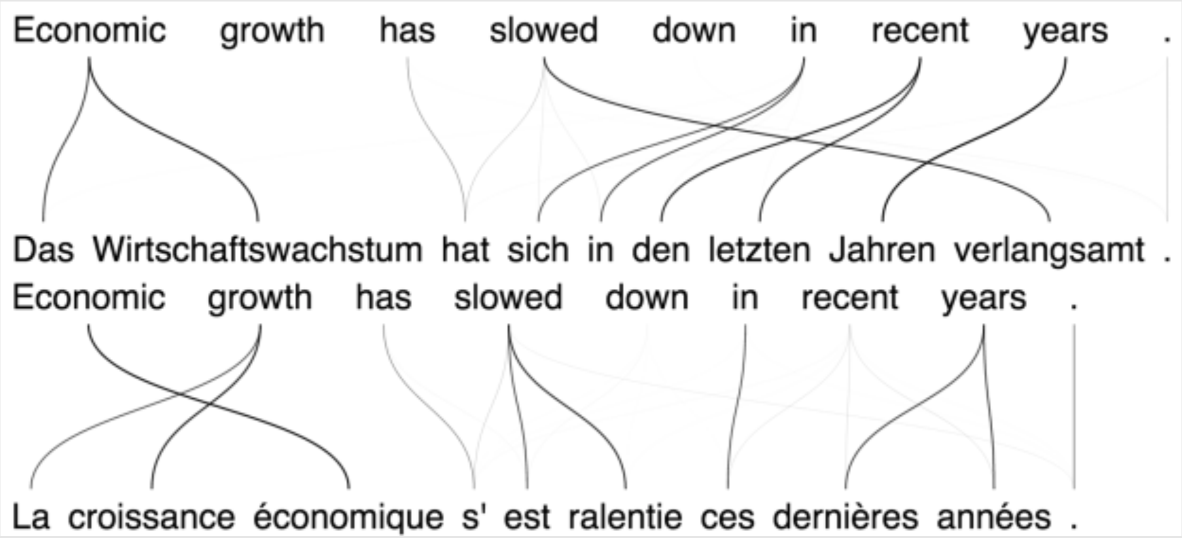


Figure from: devblogs.nvidia.com

More complicated sequential data

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

- Example: image captioning

Image captioning

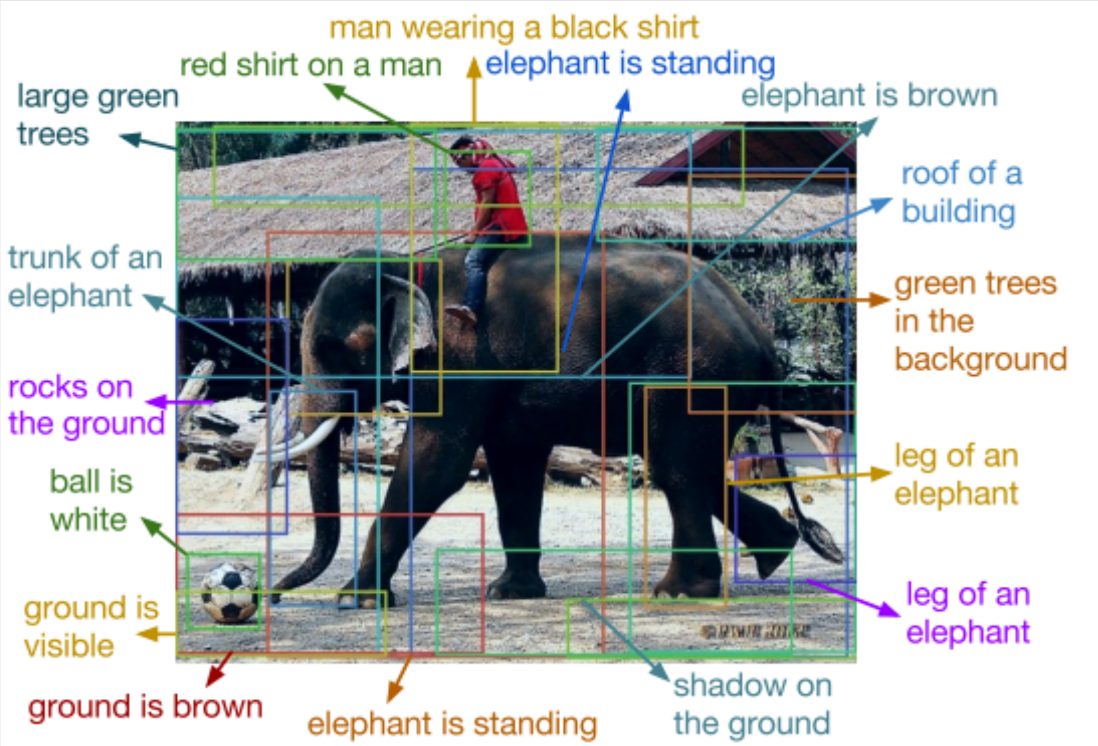
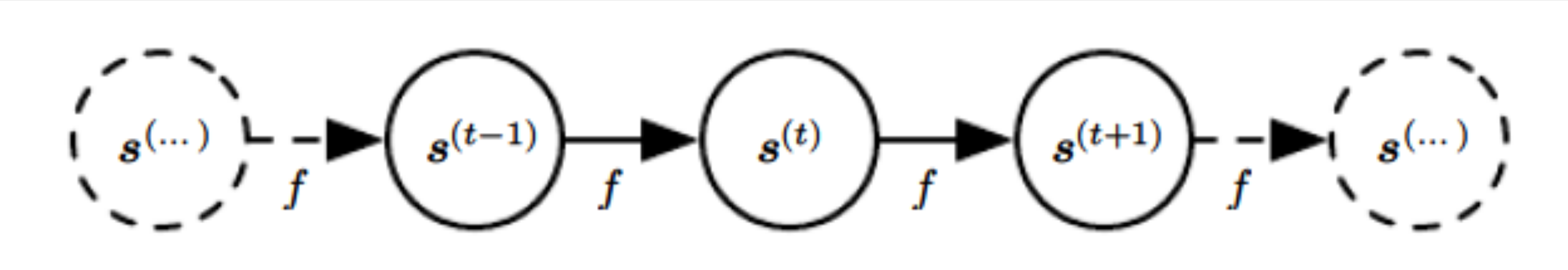


Figure from the paper “DenseCap: Fully Convolutional Localization Networks for Dense Captioning”, by Justin Johnson, Andrej Karpathy, Li Fei-Fei

Computational graphs

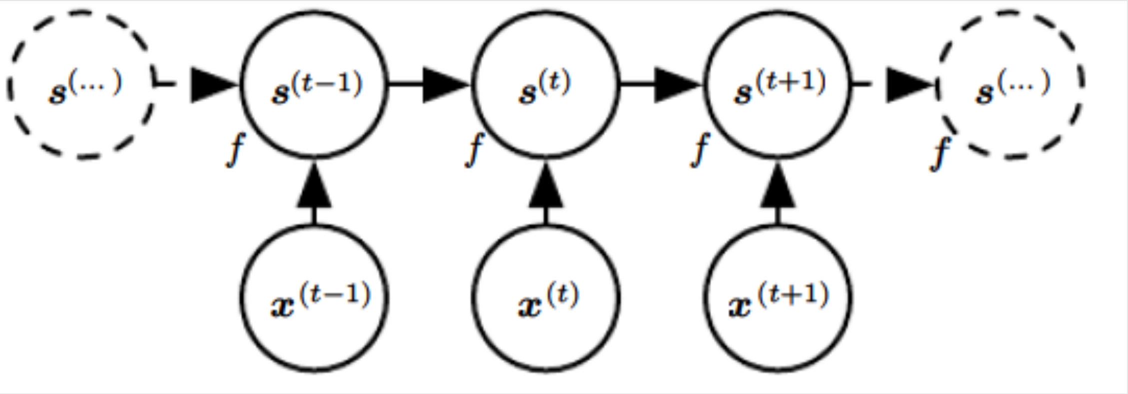
A typical dynamic system



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

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

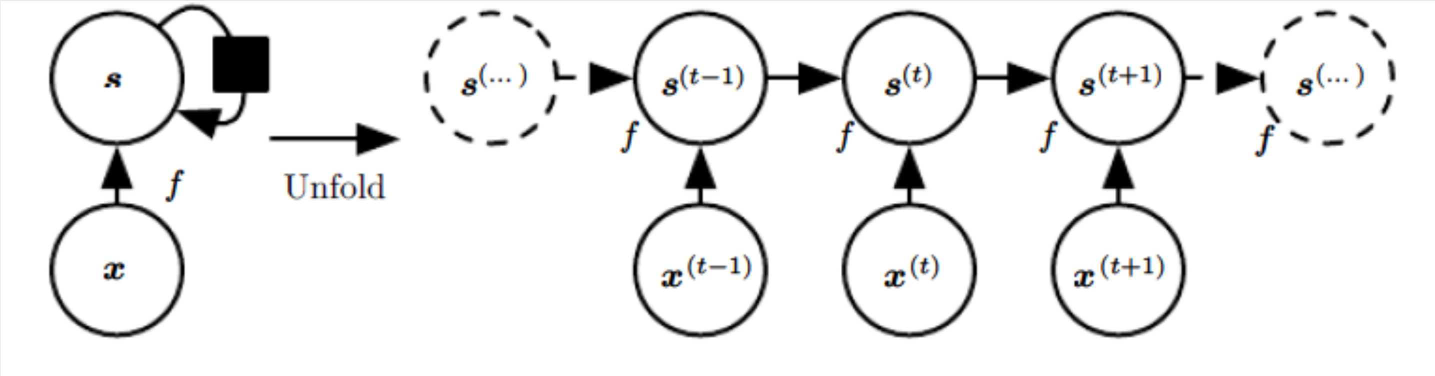
A system driven by external data



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

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

Compact view

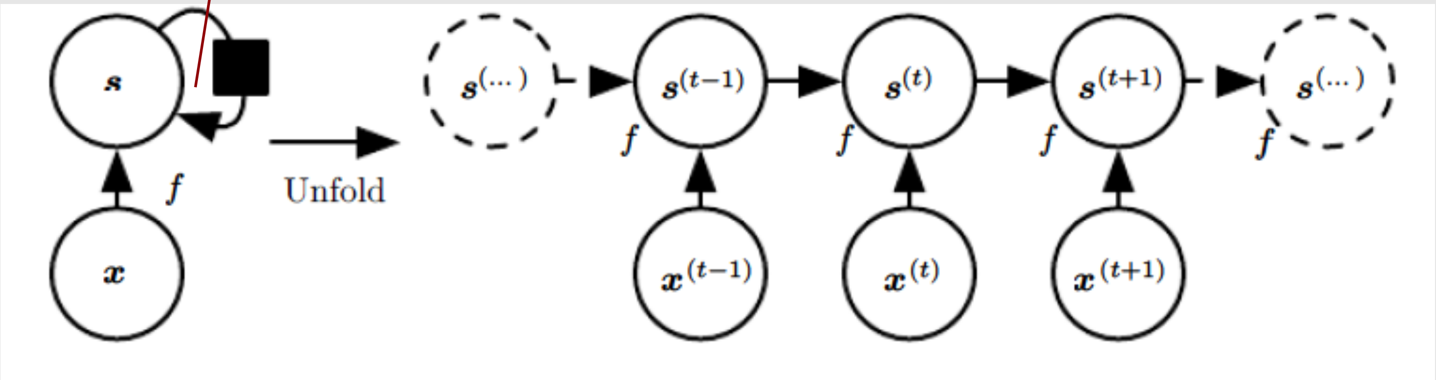


$$s^{(t+1)} = f(s^{(t)}, x^{(t+1)}; \theta)$$

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

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

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

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

Recurrent neural networks

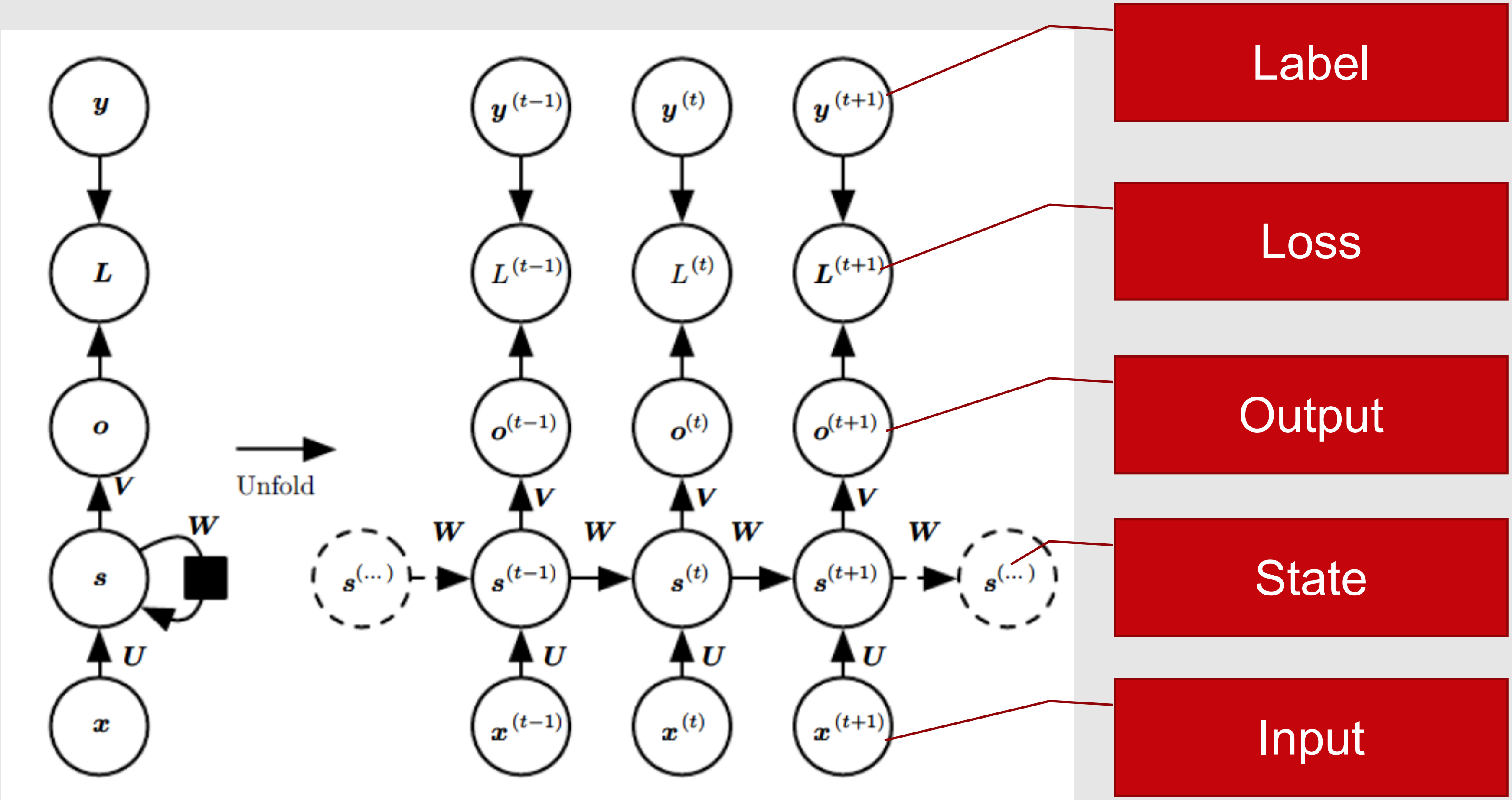
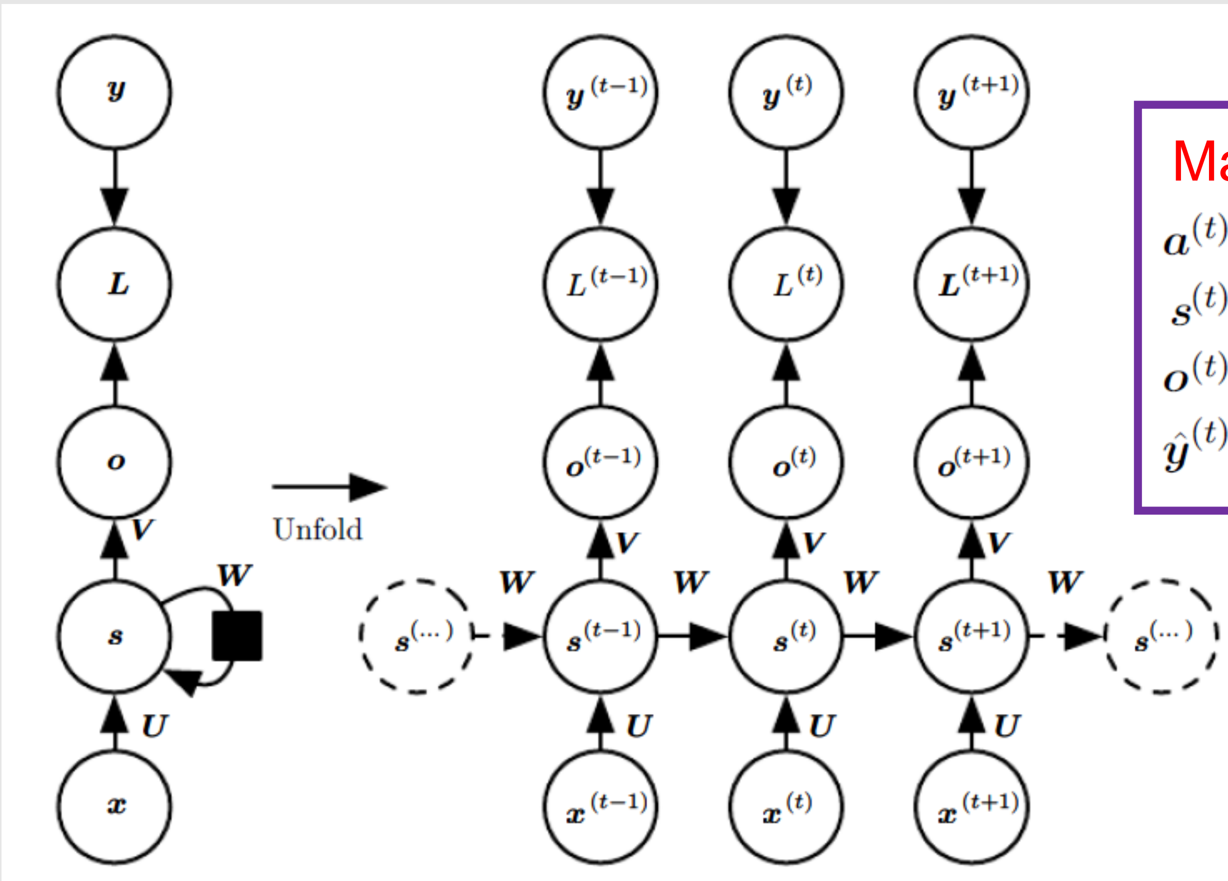


Figure from *Deep Learning*, by Goodfellow, Bengio and Courville

Recurrent neural networks



Math formula:

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

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

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)

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))

Training RNN

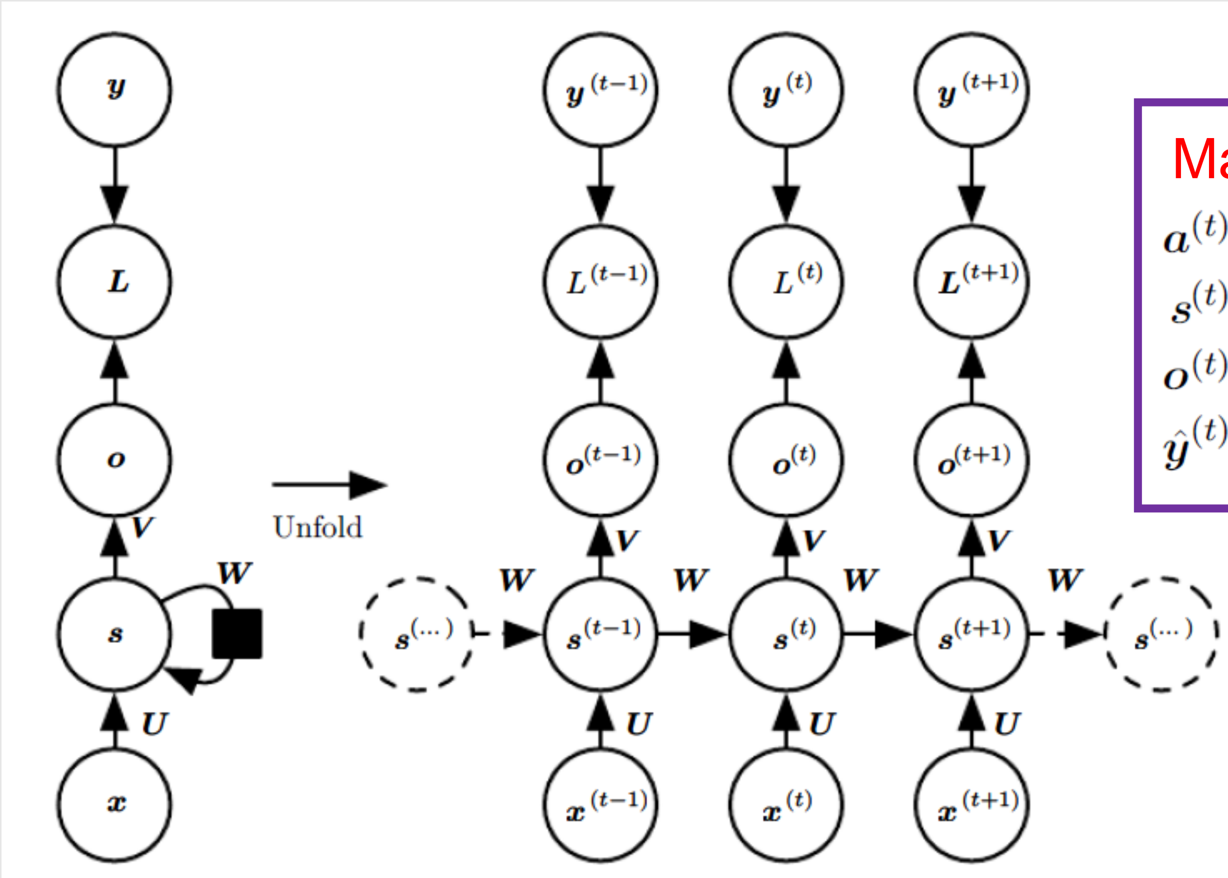
- Principle: unfold the computational graph, and use backpropagation
- Called back-propagation through time (BPTT) algorithm
- Can then apply any general-purpose gradient-based techniques

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

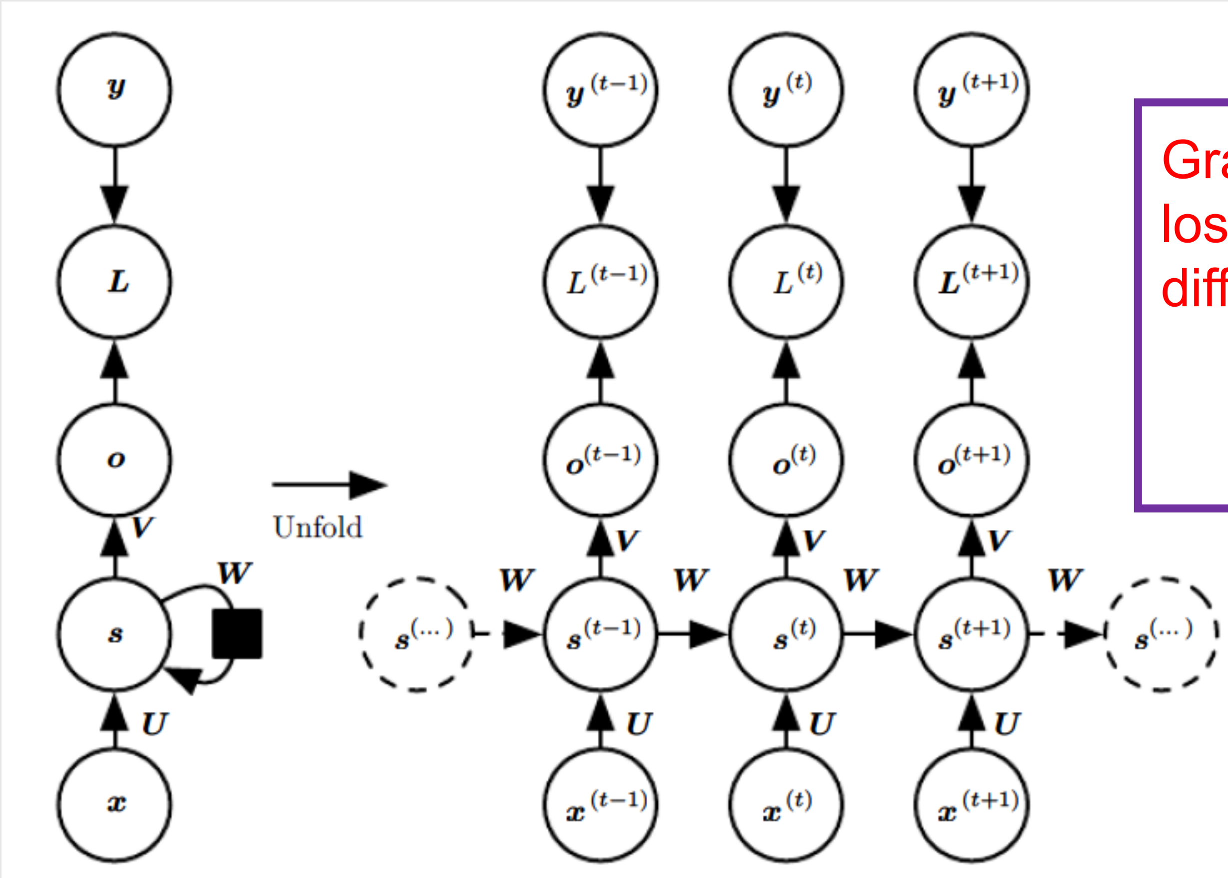


Math formula:

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

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

Recurrent neural networks

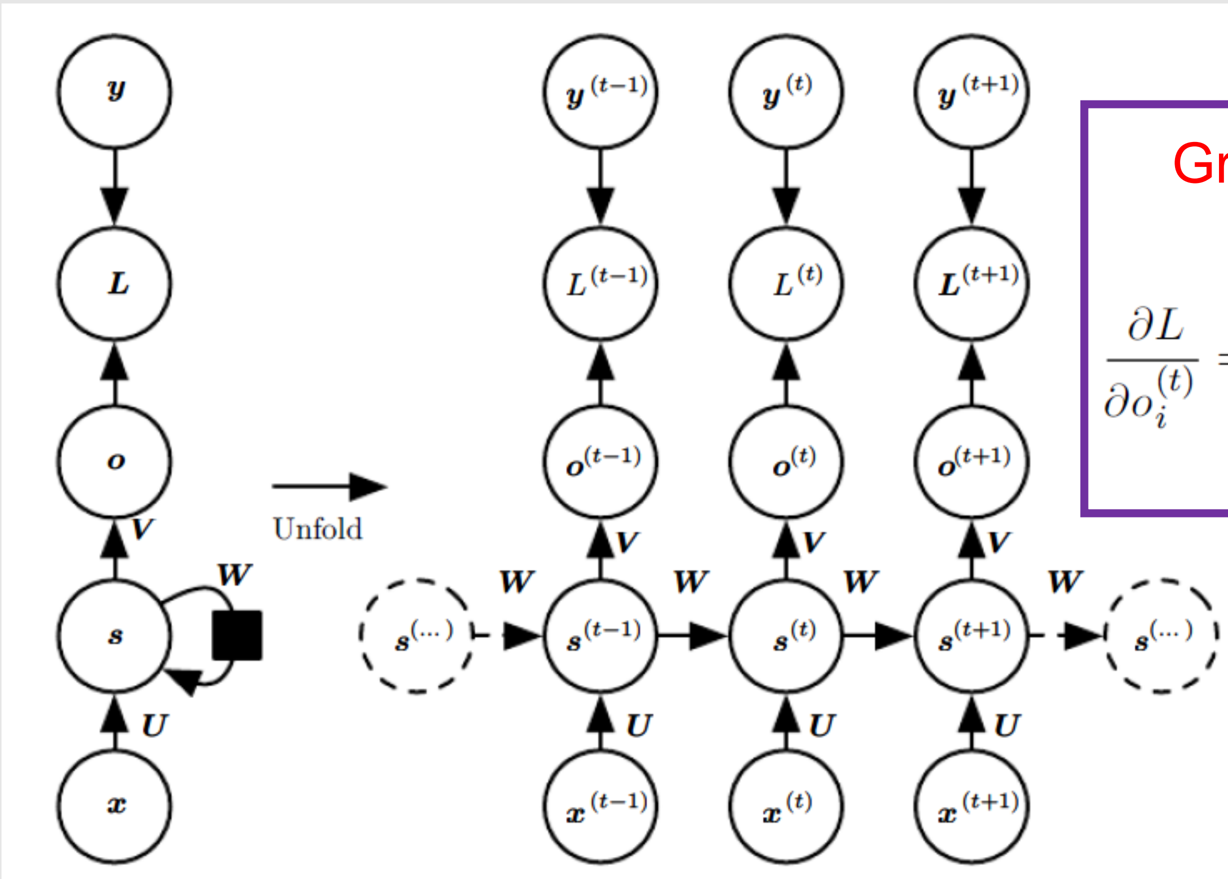


Gradient at $L^{(t)}$: (total loss is sum of those at different time steps)

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

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

Recurrent neural networks



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

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

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

How to derive this:

We ignore all time superscript (t) for clarity. For output vector o, the softmax probability prediction vector is $\hat{y}$ where $\hat{y}_i = \frac{e^{o_i}}{\sum_j e^{o_j}}$. Given label $y \in [C]$, the loss is negative log likelihood:

$$L = -\log p(y \mid o) = -\log \frac{e^{o_y}}{\sum_j e^{o_j}} = \log \frac{\sum_j e^{o_j}}{e^{o_y}}.$$

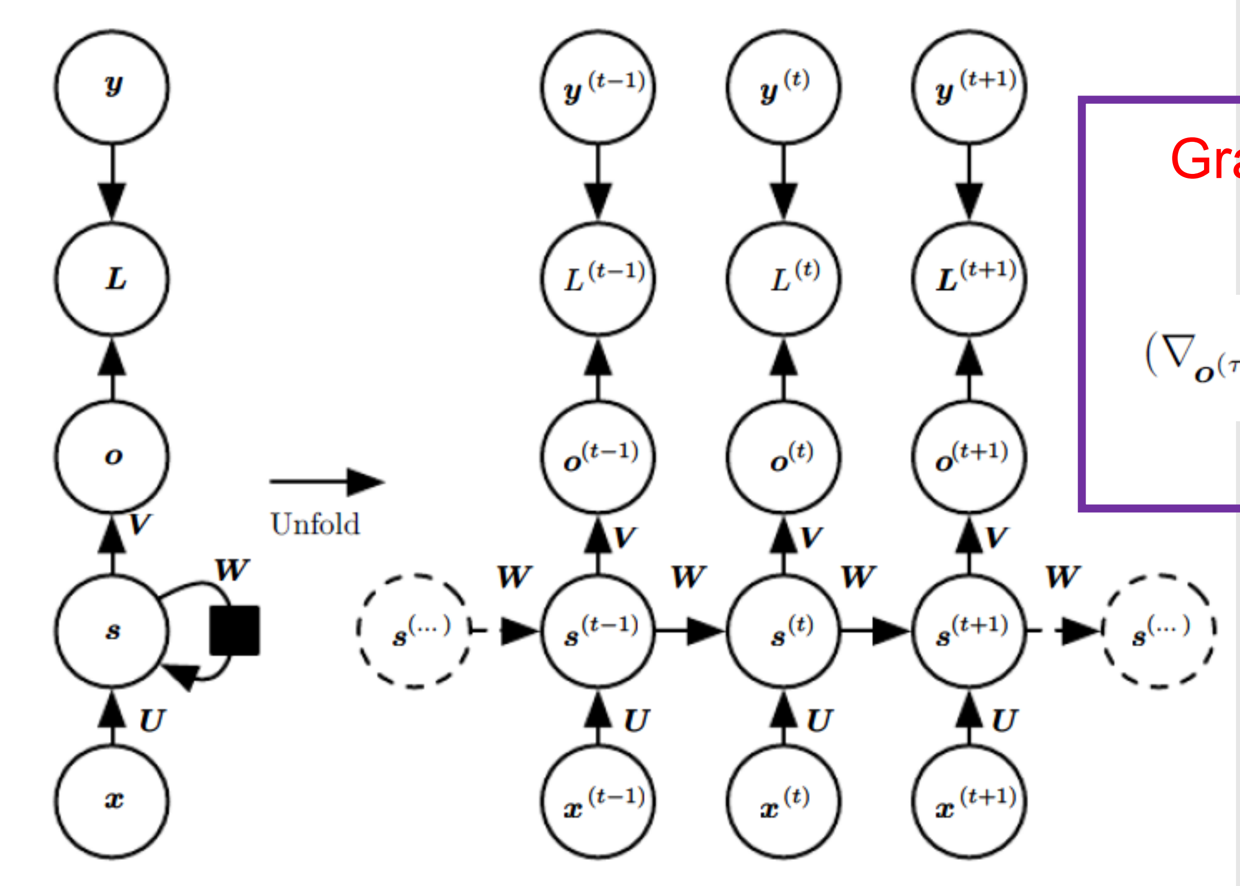
Thus

$$\begin{aligned} \partial L / \partial o_i &= \frac{e^{o_y}}{\sum_j e^{o_j}} \left(\frac{e^{o_i}}{e^{o_y}} + \left(\sum_j e^{o_j} \right) \frac{\partial}{\partial o_i} \left(\frac{1}{e^{o_y}} \right) \right) && (1) \\ &= \hat{y}_i + e^{o_y} \frac{\partial}{\partial o_i} \left(\frac{1}{e^{o_y}} \right). && (2) \end{aligned}$$

Note the partial derivative is 0 if $i \neq y$, and $-\frac{1}{e^{o_y}}$ if $i = y$. Hence

$$\partial L / \partial o_i = \hat{y}_i - 1_{[i=y]}.$$

Recurrent neural networks

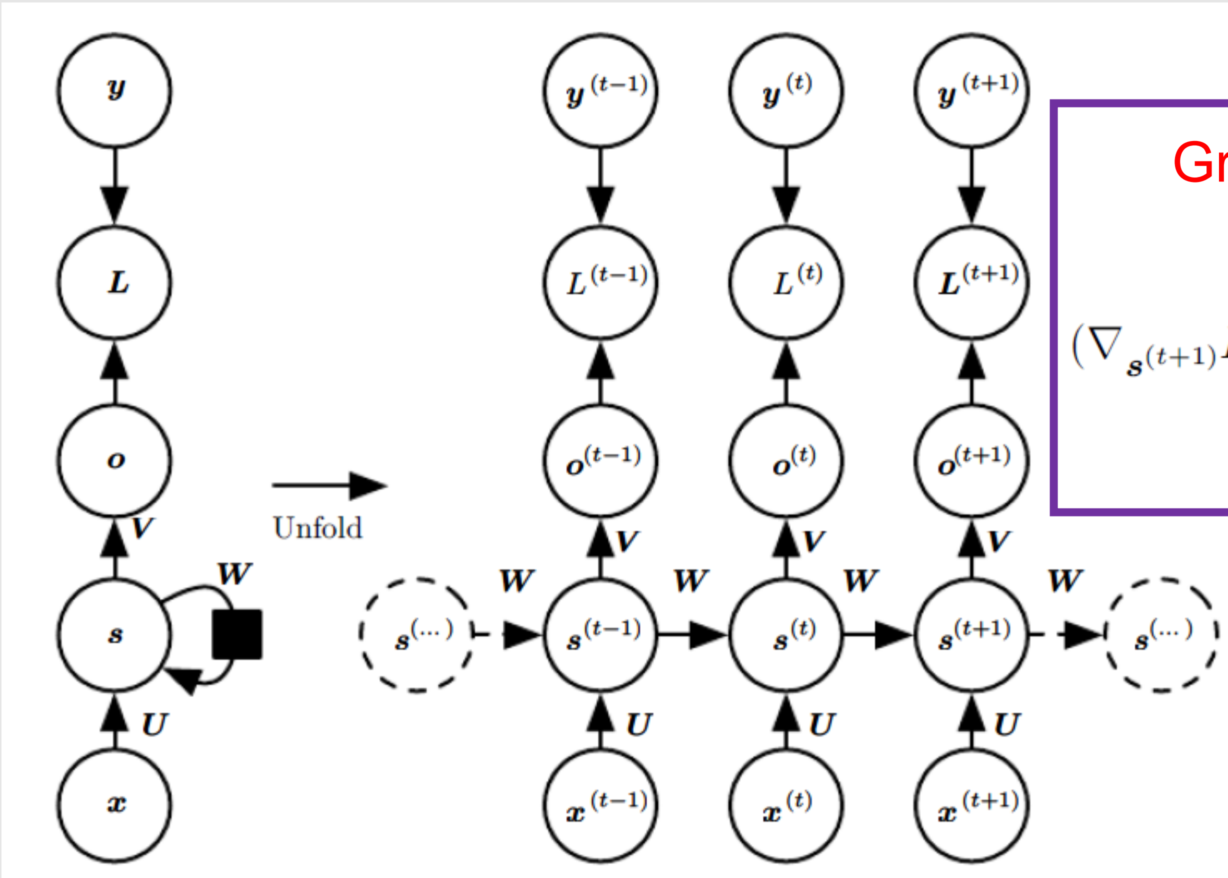


Gradient at $s^{(\tau)}$:

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

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

Recurrent neural networks

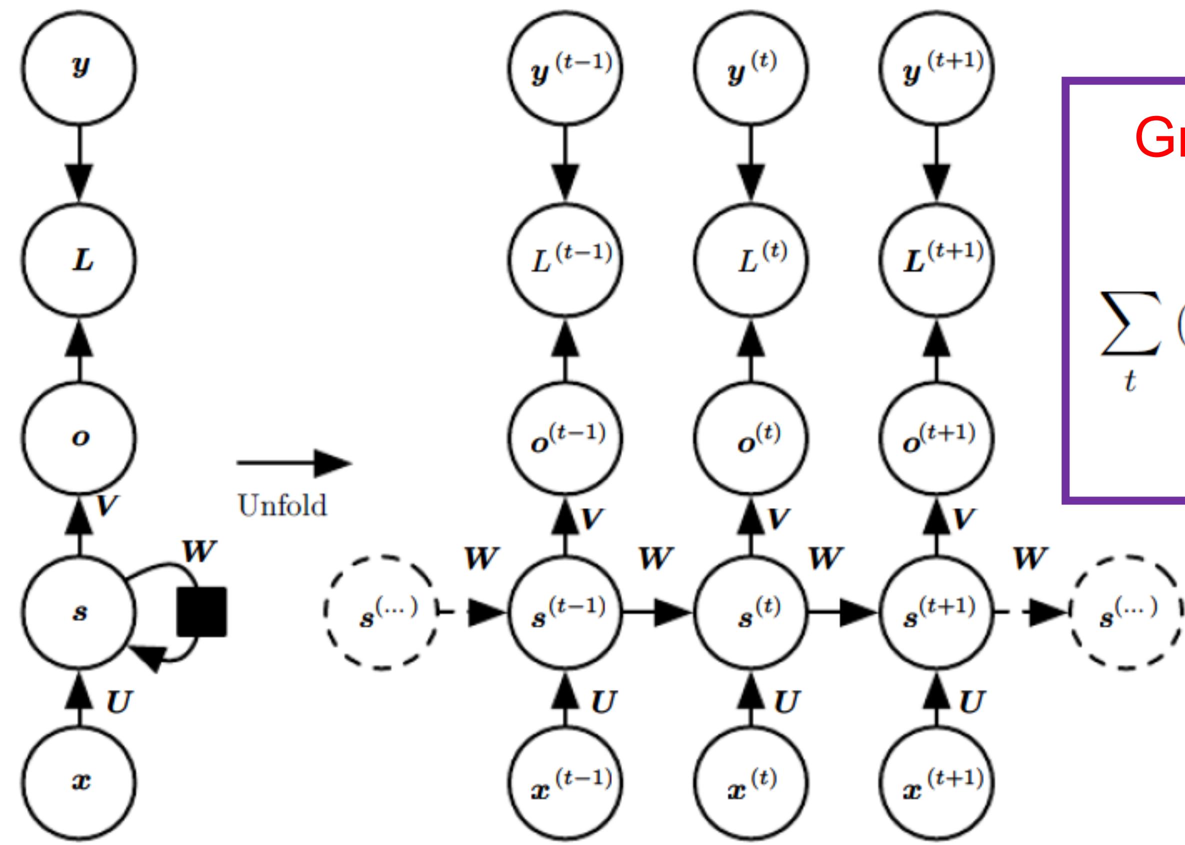


Gradient at $s^{(t)}$:

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

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

Recurrent neural networks



Gradient at parameter V:

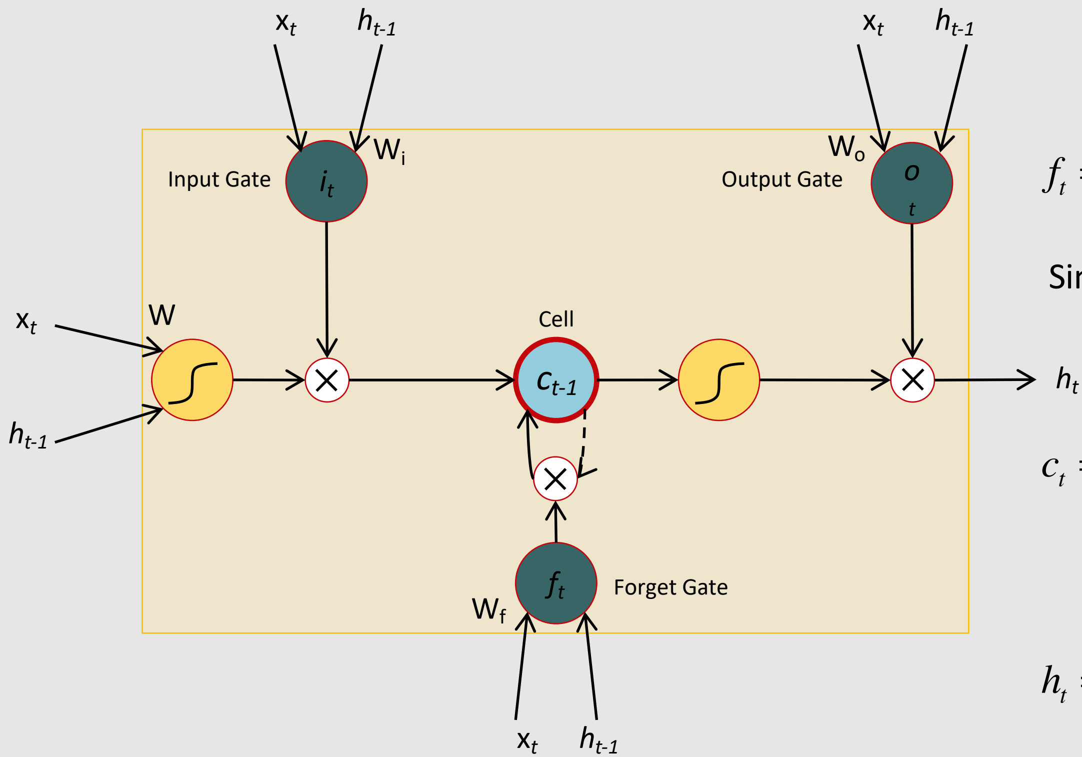
$$\sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{V}} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \boldsymbol{s}^{(t)\top}$$

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

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.

Long Short-Term Memory (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

Gated Recurrent Unit (GRU) Celll

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + (1 - u_i^{(t-1)}) \sigma_{any} \left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)} \right)$$

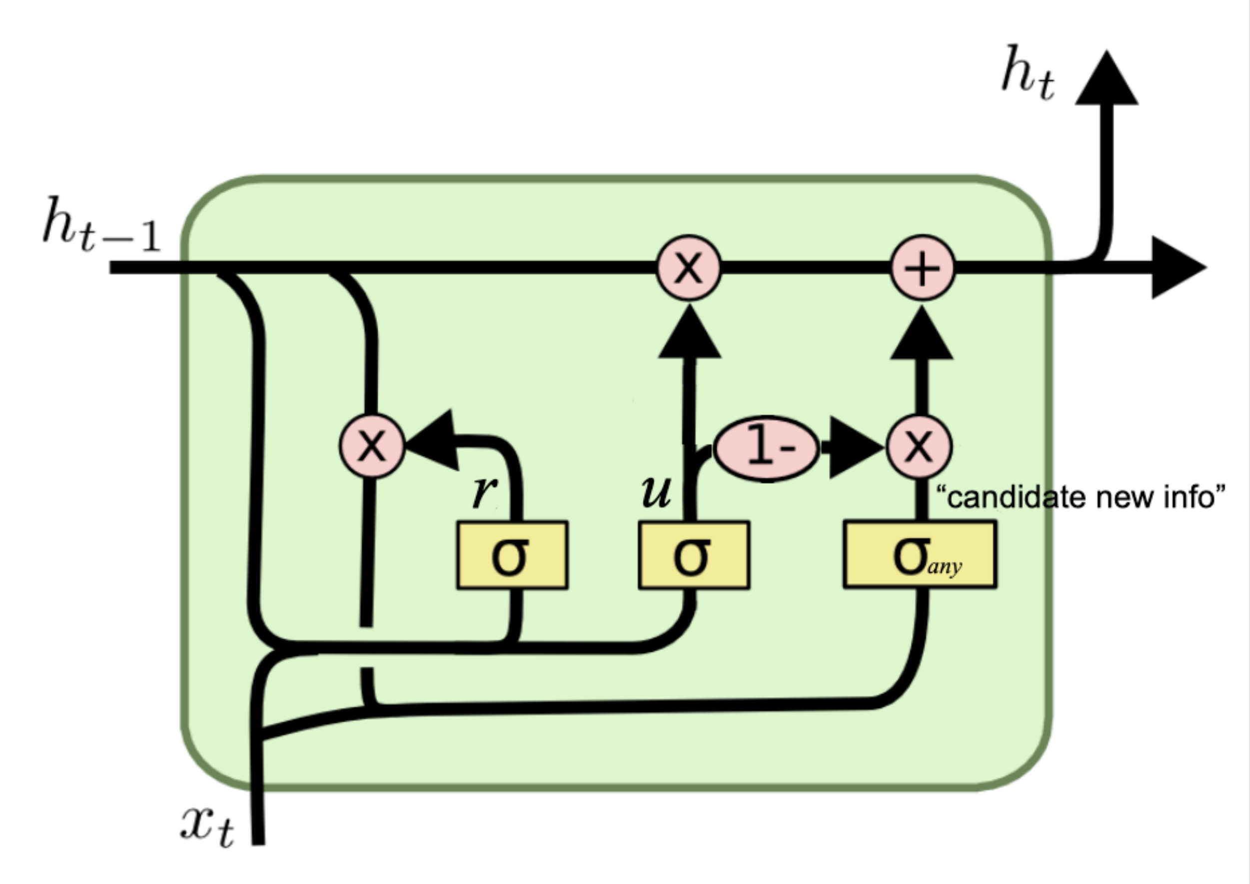


Figure **modified** from Christopher Olah
https://colah.github.io/posts/2015-08-Understanding-LSTMs/

Seq2seq (Encoder Decoder)

Good for machine translation

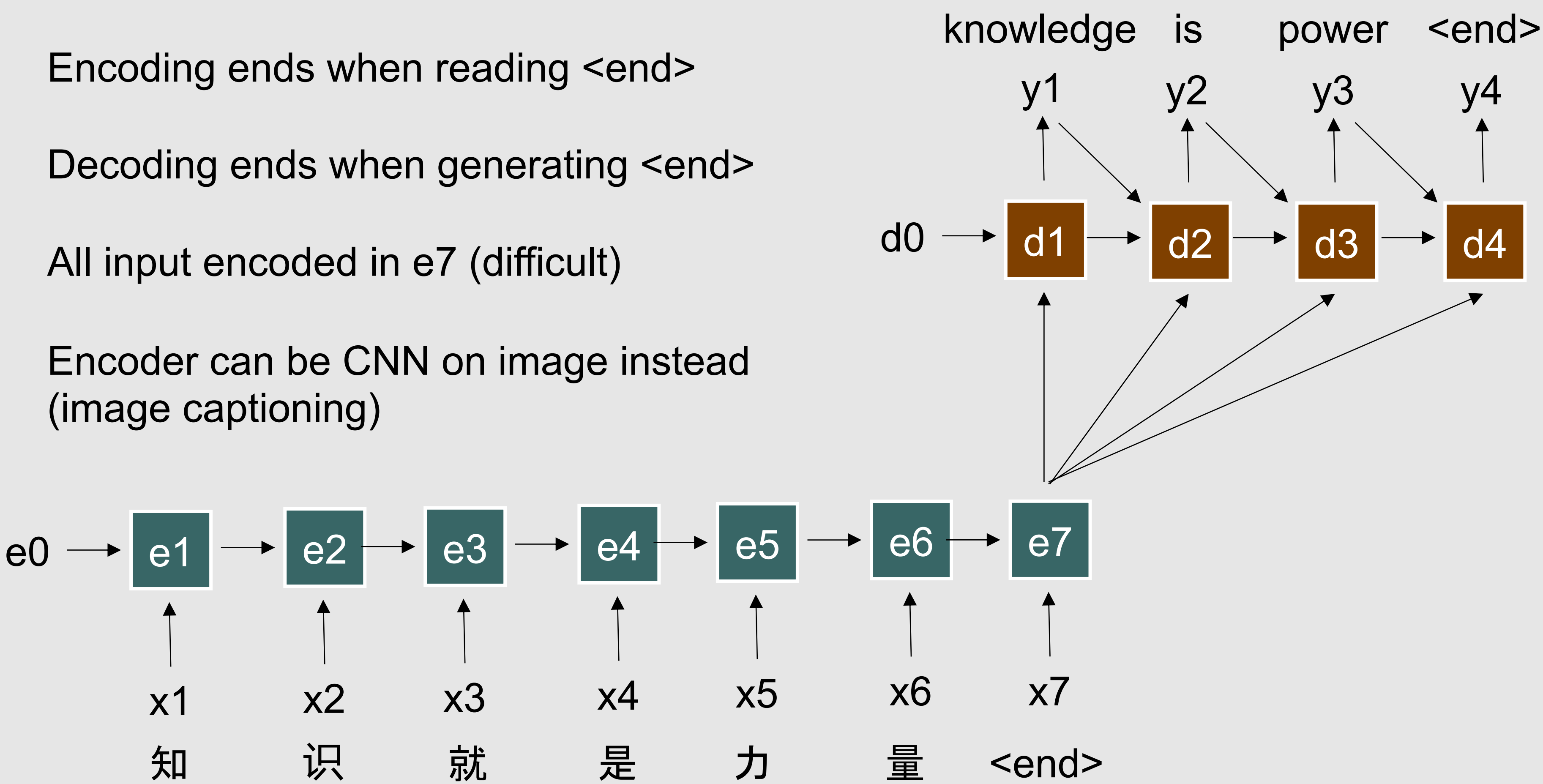
Two RNNs

Encoding ends when reading <end>

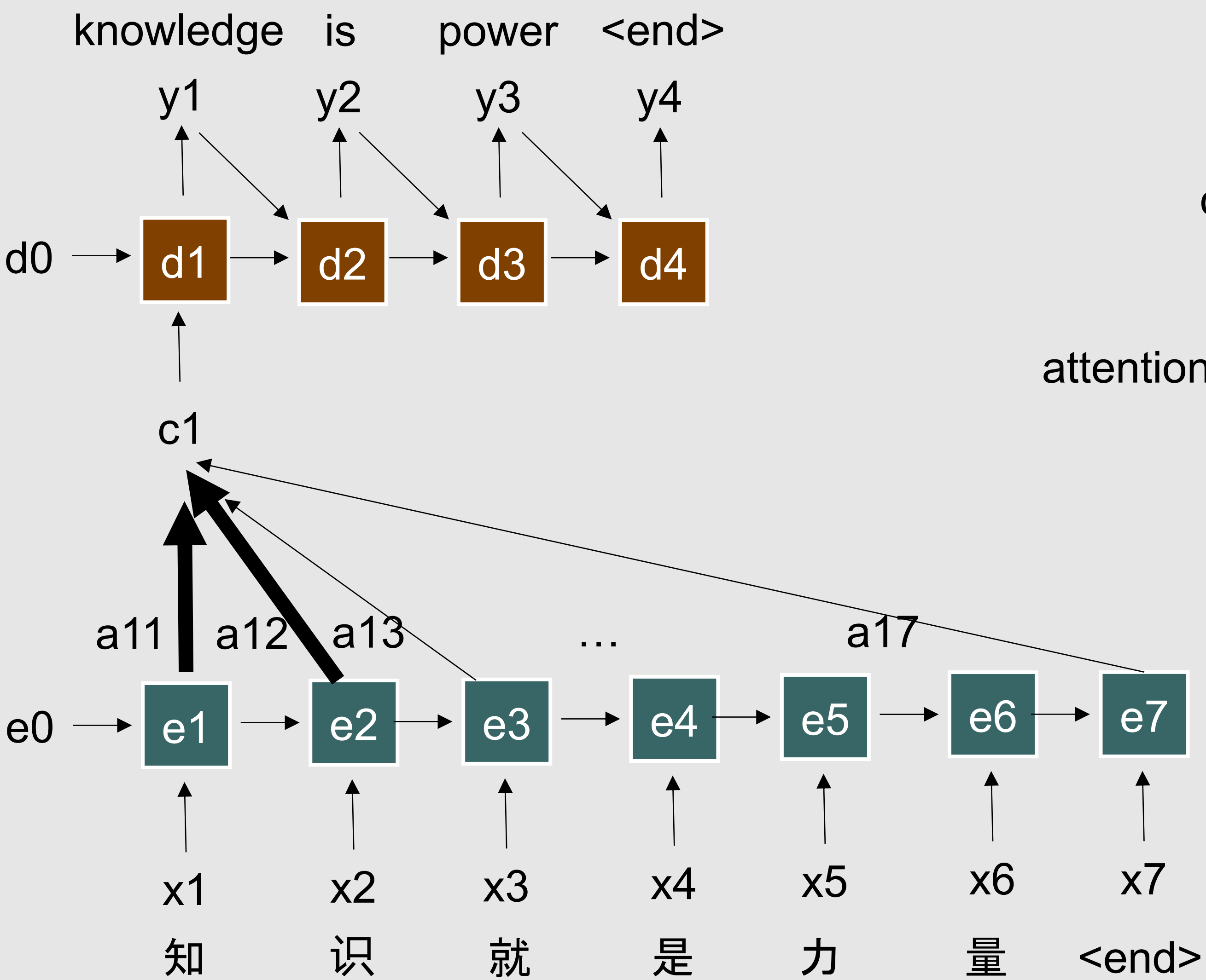
Decoding ends when generating <end>

All input encoded in e7 (difficult)

Encoder can be CNN on image instead (image captioning)



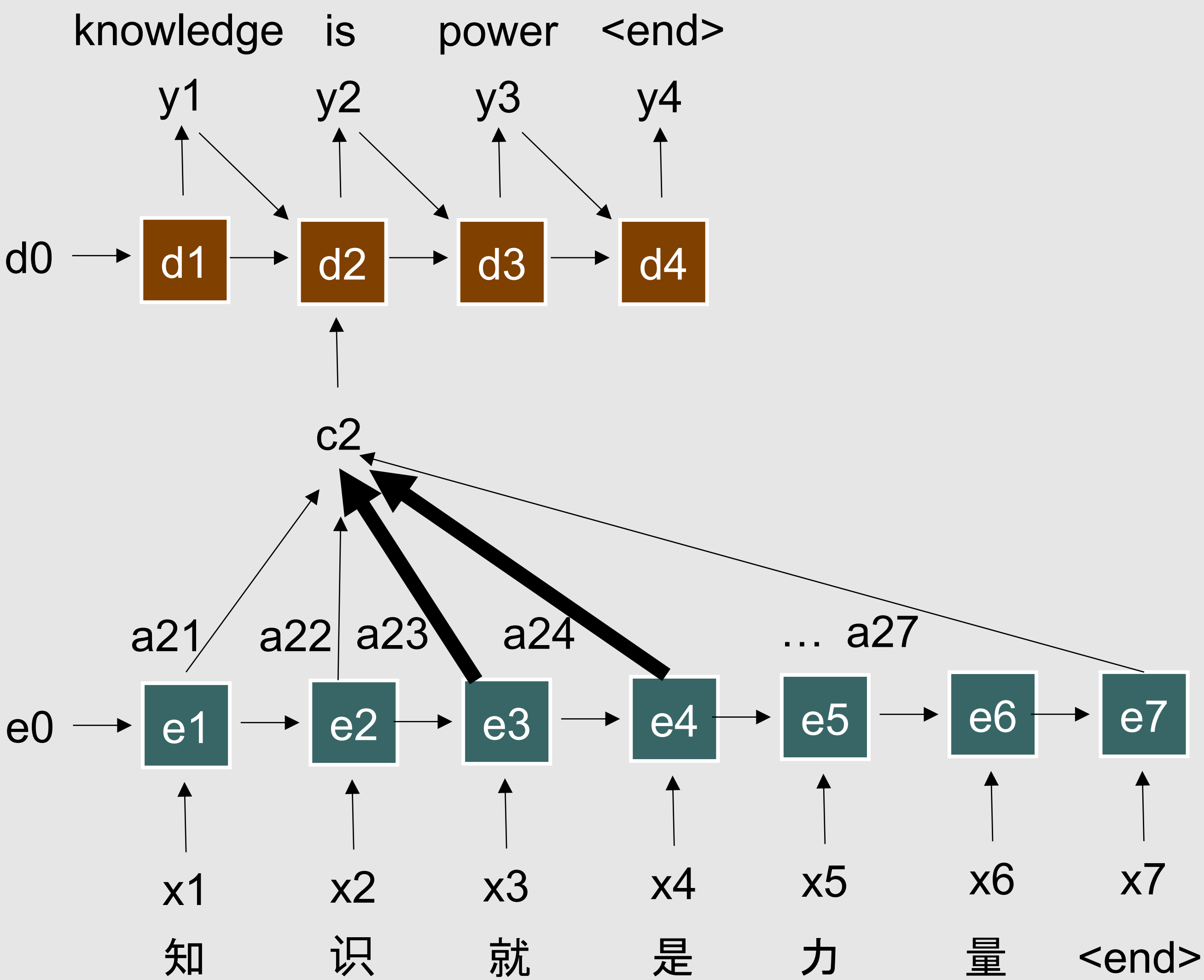
Seq2seq with Attention



context $c_1 = \sum_{t=1}^{7} a_{1t} e_t$

attention weights $\sum_{t=1}^{7} a_{1t} = 1$

Seq2seq with Attention



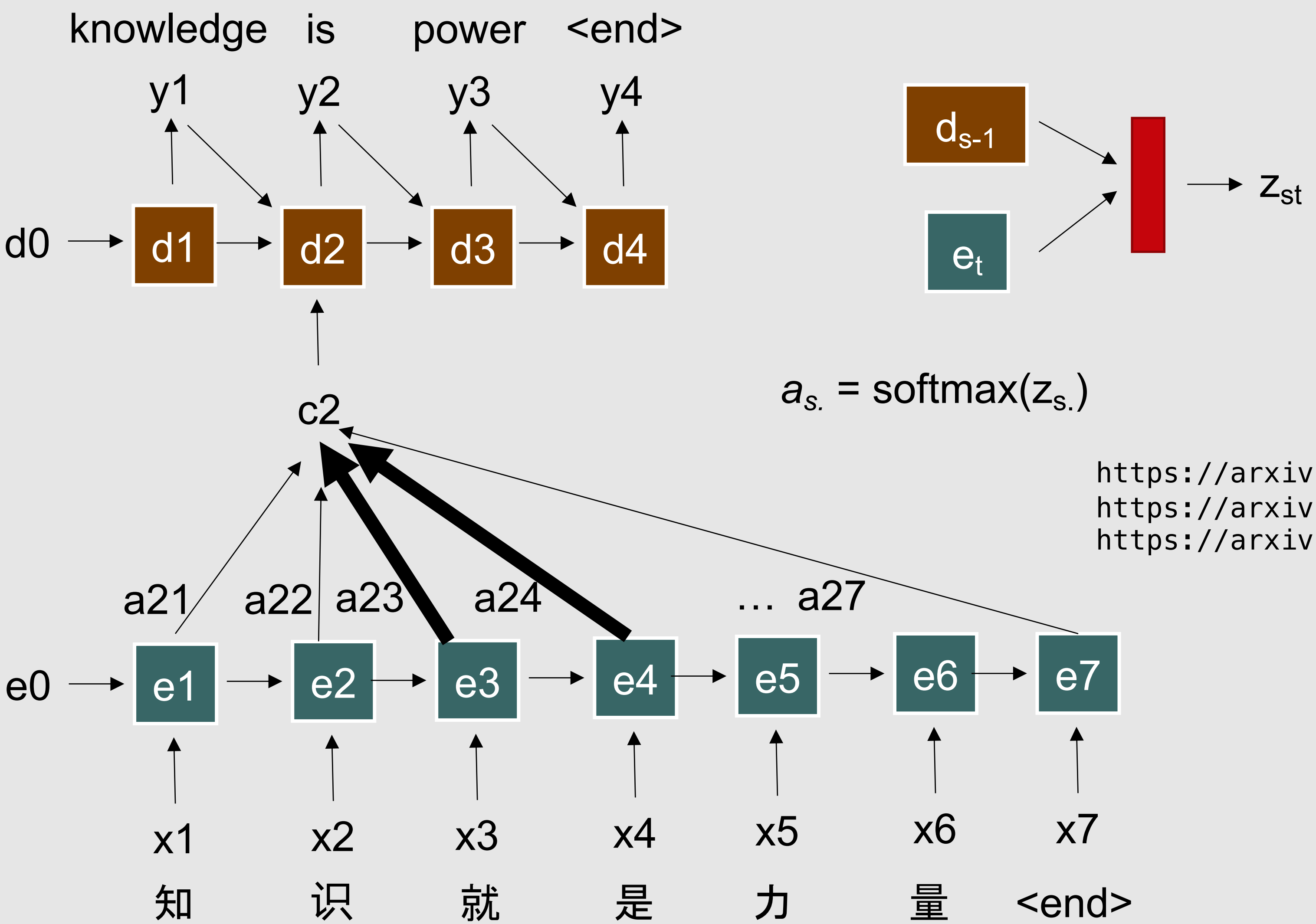
$$c_2 = \sum_{t=1}^{7} a_{2t} e_t$$

$$\sum_{t=1}^{7} a_{2t} = 1$$

… and so on

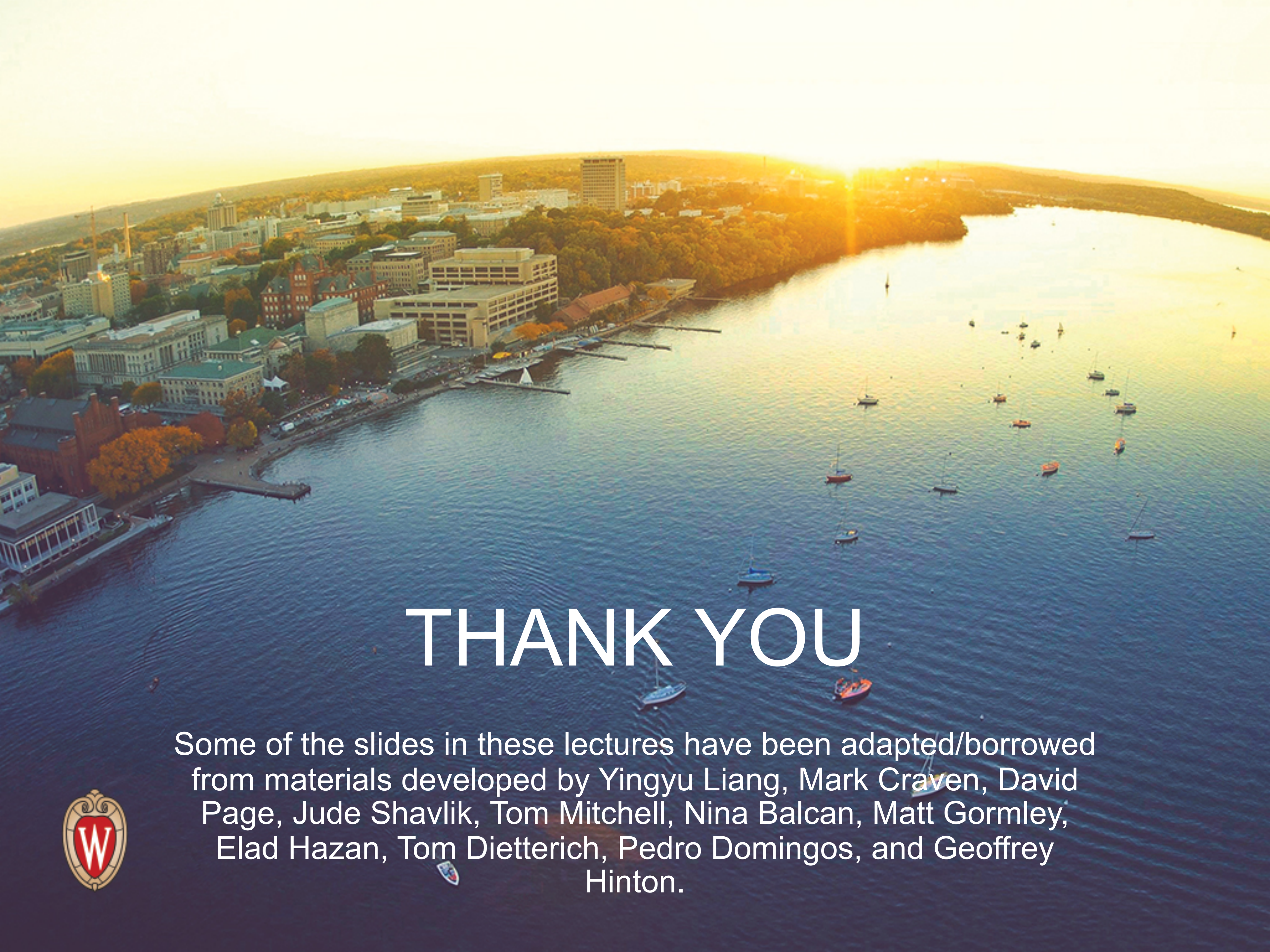
Attention weights

a_{st}: the amount of attention y_s should pay to e_t



$a_{s.} = \text{softmax}(z_{s.})$

https://arxiv.org/abs/1409.0473
https://arxiv.org/abs/1409.3215
https://arxiv.org/abs/1406.1078



THANK YOU

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