

Session 7: Recurrent Neural Networks

Today's Menu

- Working with sequences
- Feed-forward networks and sequences
- Recurrent neural networks
 - “vanilla” RNN
 - Long short-term memory networks (LSTM)

Sequences!

- Previously: $y = f(x)$, where
 - f : (deep) neural network
 - x : independent sample
 - y : single output (class label)
- For the rest of the week:
 - $x = x_1, x_2, \dots, x_M$: sequence of M observations
 - $y = y_1, y_2, \dots, y_N$: sequence of N outputs (class labels)
- ...this will add another dimension to our tensors!

Sequences!

- $M > 1, N = 1$: “many-to-one”
 - speaker identification
 - sentiment classification
 - fraudulent transaction
- $M > 1, N > 1$: “many-to-many”
 - $M \equiv N$: “time-synchronous”, strict 1:1 alignment $x_t \rightarrow y_t$
 - speech recognition, part-of-speech tagging, ...
 - $M \neq N$: flexible alignment (if any)
 - machine translation, summarization, ...

Many-to-One

Sentiment
Classification

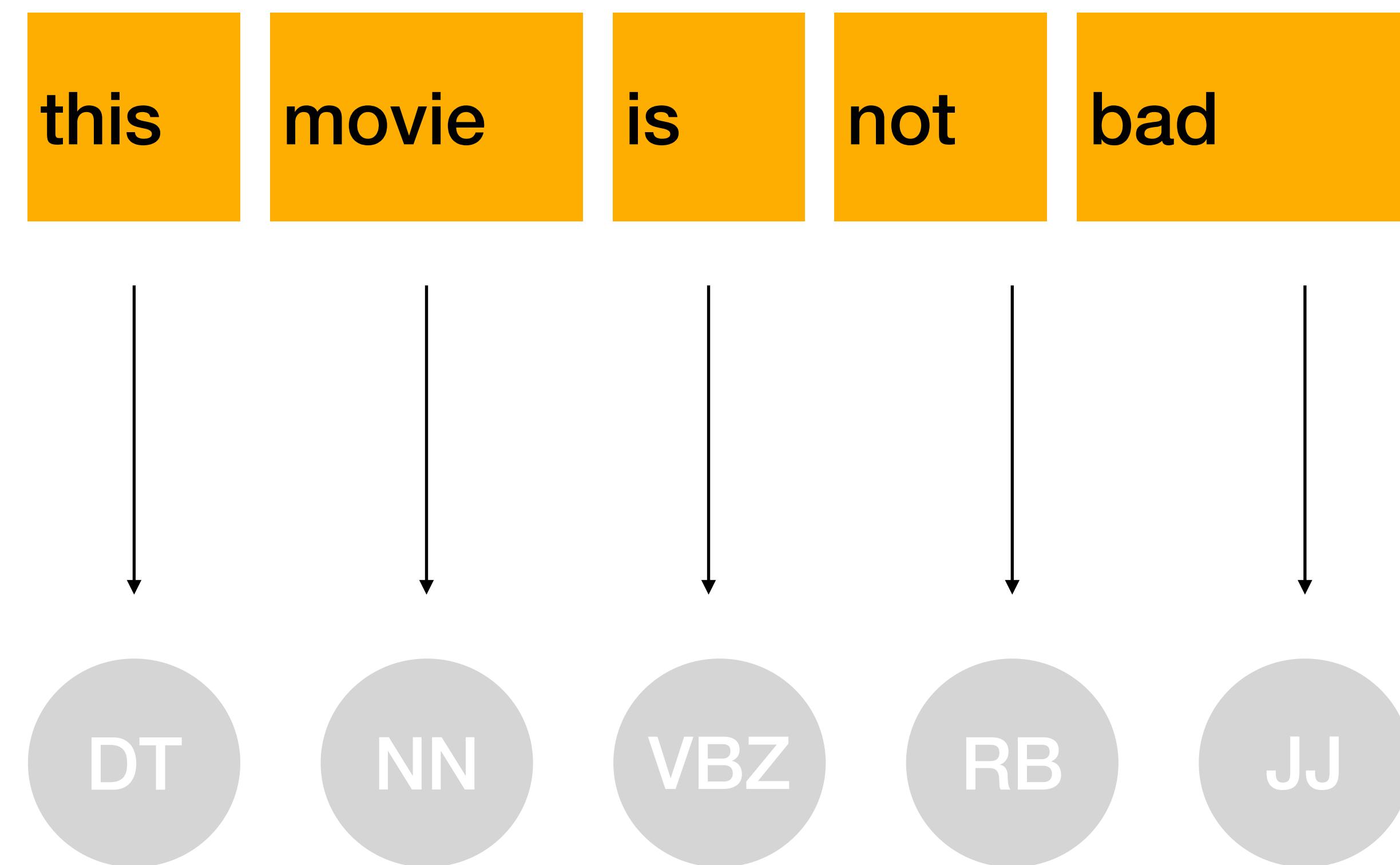
this movie is not bad



neutral

Many-to-Many, $M \equiv N$

Part-of-Speech
Tagging



Many-to-Many, $M \neq N$

Machine
Translation

this movie is not bad



der Film passt schon

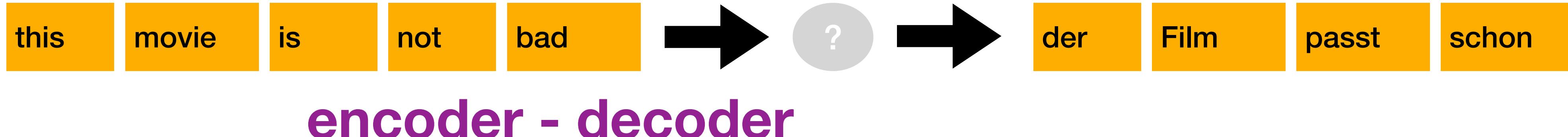
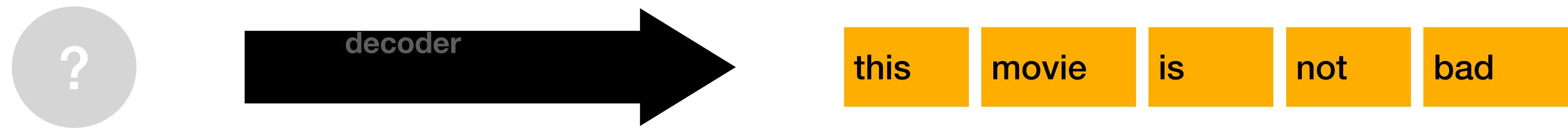
One-to-Many

Image Captioning



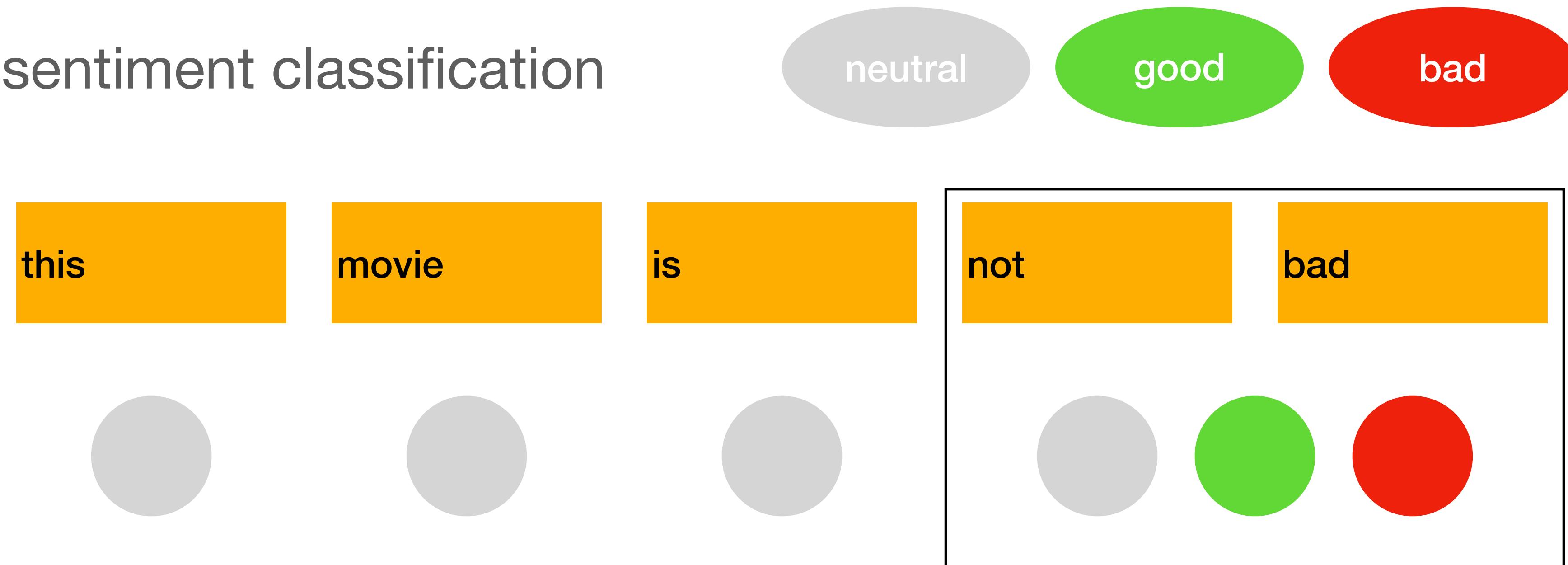
old man with woolen mittens

Terminology (revisited)



Context is Crucial

Example: sentiment classification

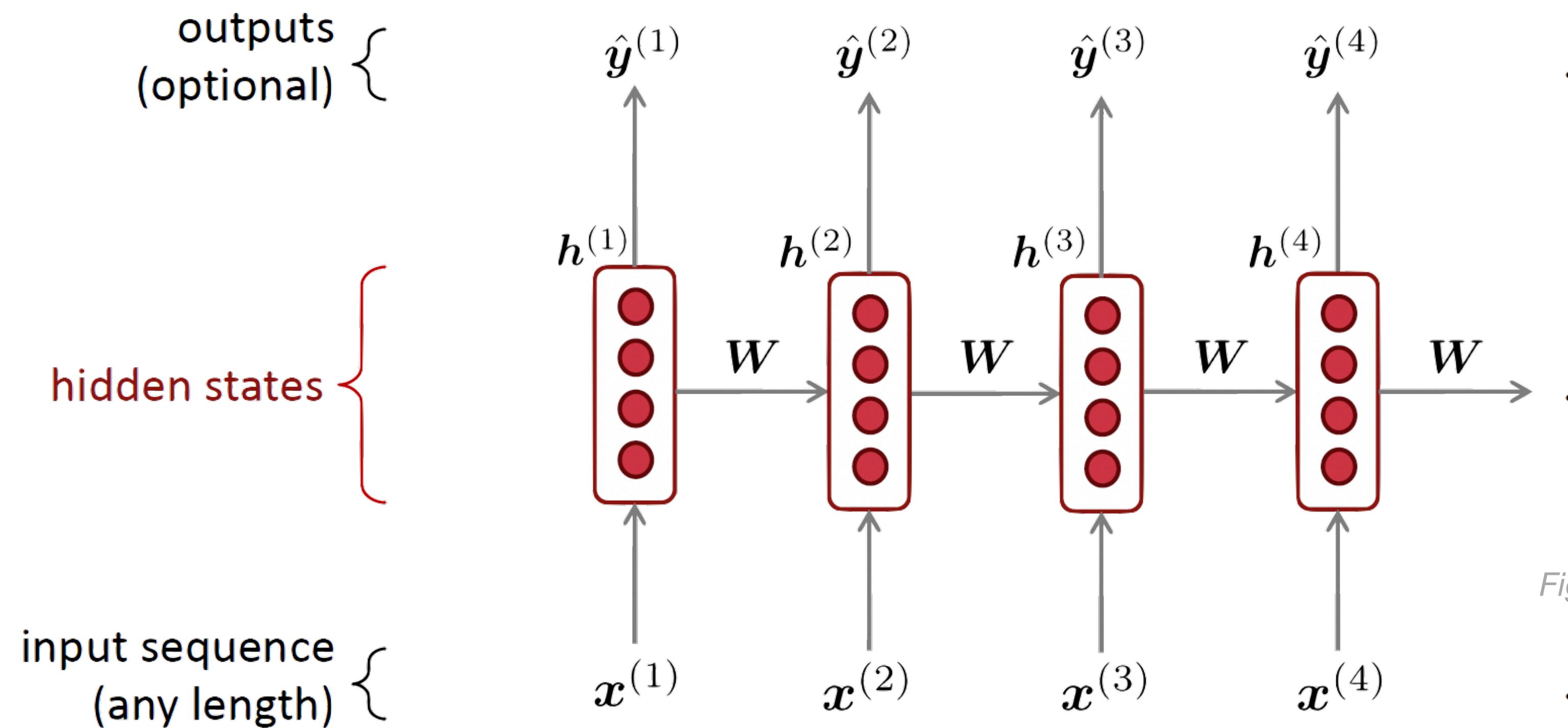


Solution: Use context windows to learn temporal relations

Drawbacks

- Context (or filter) size is fixed → no long-term dependencies
- Context is always used → large input or intermediate layers
- What happens at beginning/end of sequence?
- Is there another (better?) way to encode sequential information?

Recurrent Neural Networks



Figures: Manning et al.,
Stanford cs224n

...

Recurrent Neural Networks

output distribution

$$\hat{y}^{(t)} = \text{softmax} \left(U h^{(t)} + b_2 \right) \in \mathbb{R}^{|V|}$$

hidden states

$$h^{(t)} = \sigma \left(W_h h^{(t-1)} + W_e e^{(t)} + b_1 \right)$$

$h^{(0)}$ is the initial hidden state

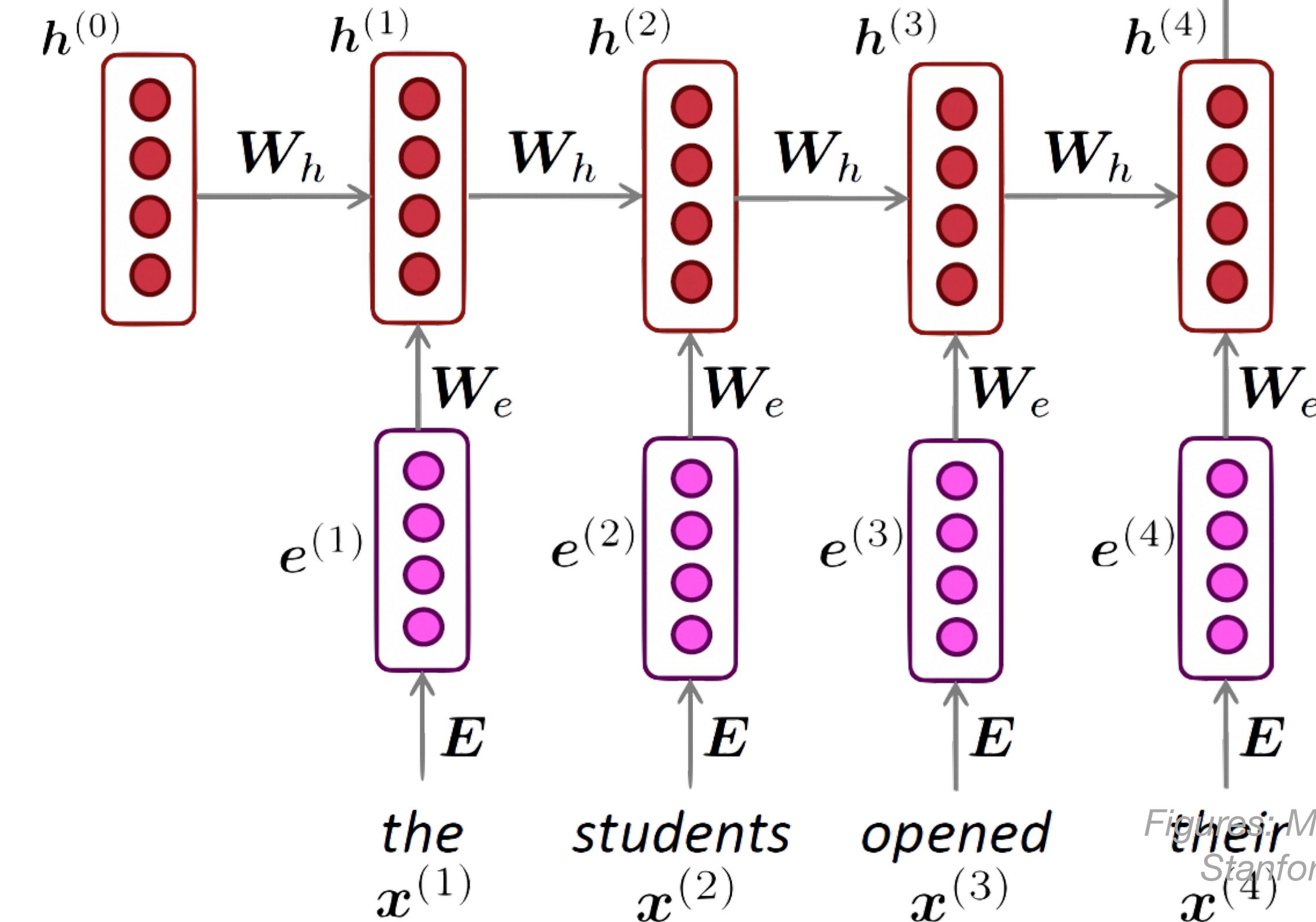
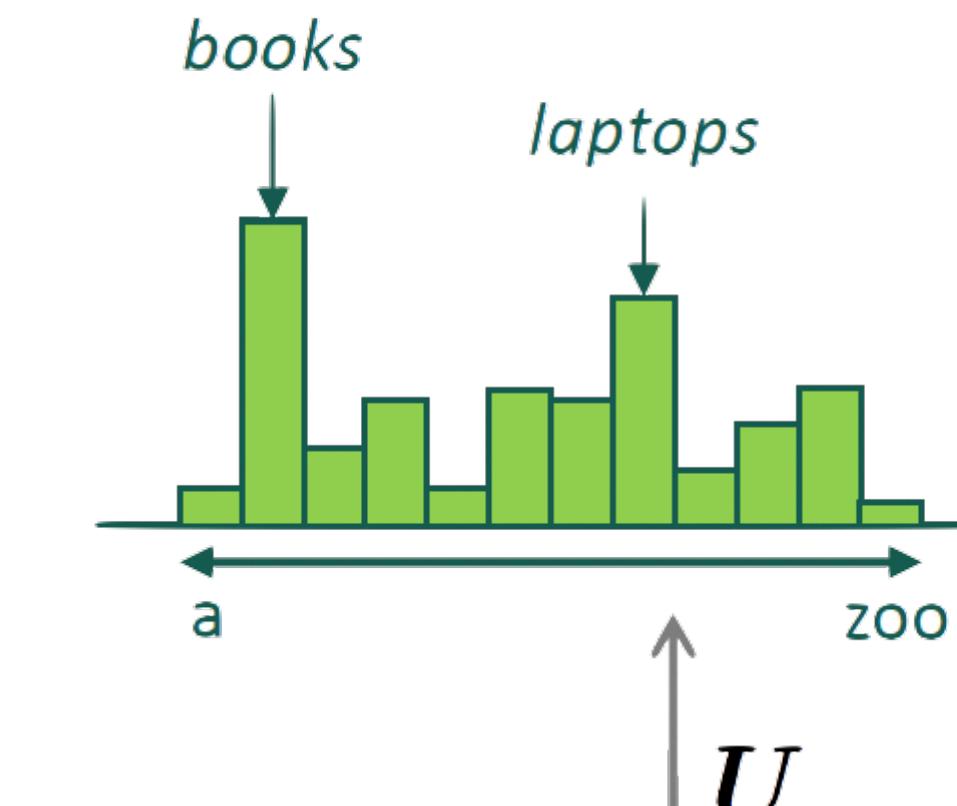
word embeddings

$$e^{(t)} = E x^{(t)}$$

words / one-hot vectors

$$x^{(t)} \in \mathbb{R}^{|V|}$$

$$\hat{y}^{(4)} = P(x^{(5)} | \text{the students opened their})$$



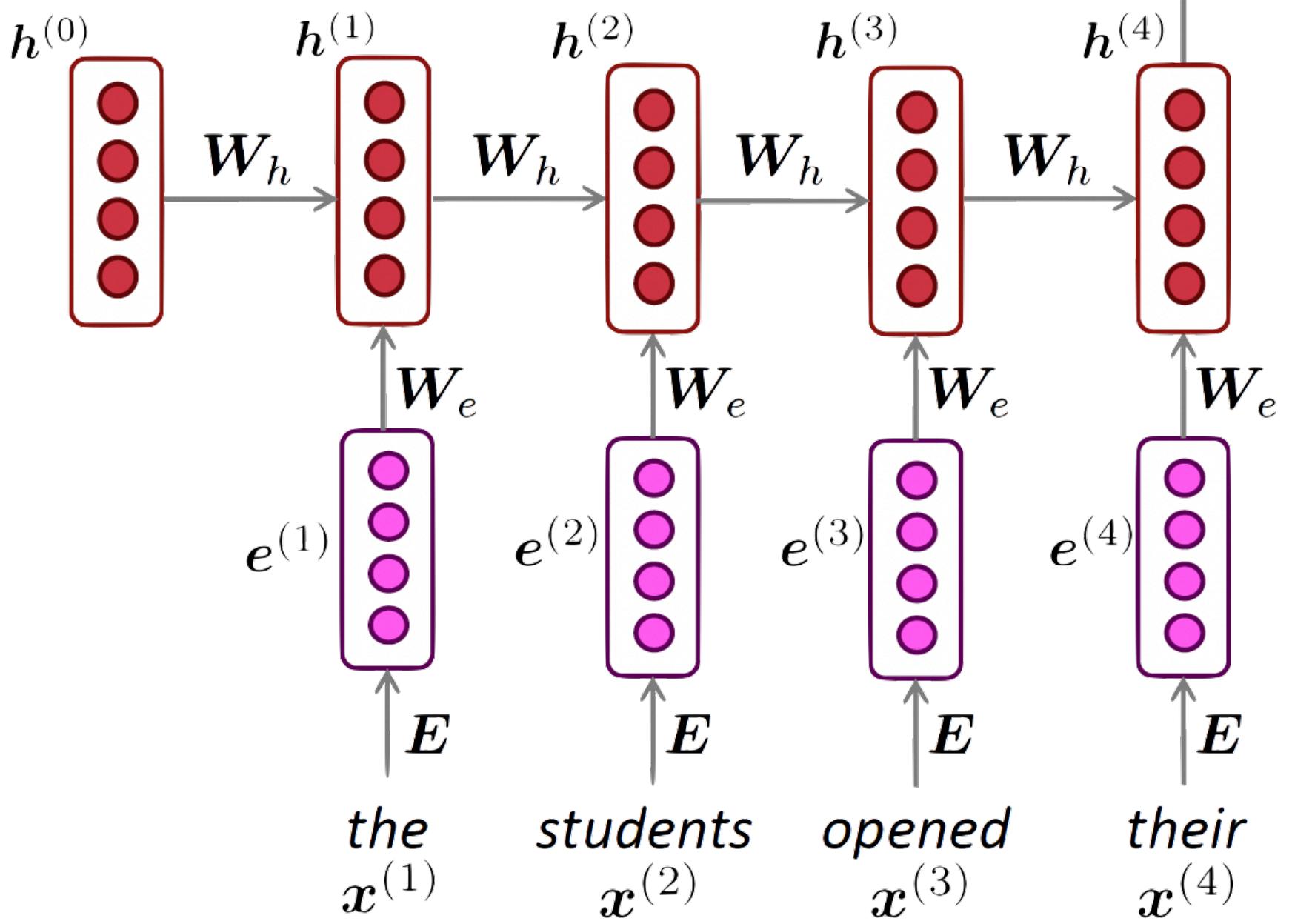
Figures: Manning et al.,
Stanford cs224n

Recurrent Neural Networks

- Advantages:
 - Can process **any length** input
 - Computation scheme allows information to remain “in the loop”
 - Model size doesn’t increase for longer input context
 - Same weights applied at every timestep → “symmetry” in compute
- Disadvantages
 - Recurrence is **slow**: can’t parallelize over time
 - In practice, difficult to access history

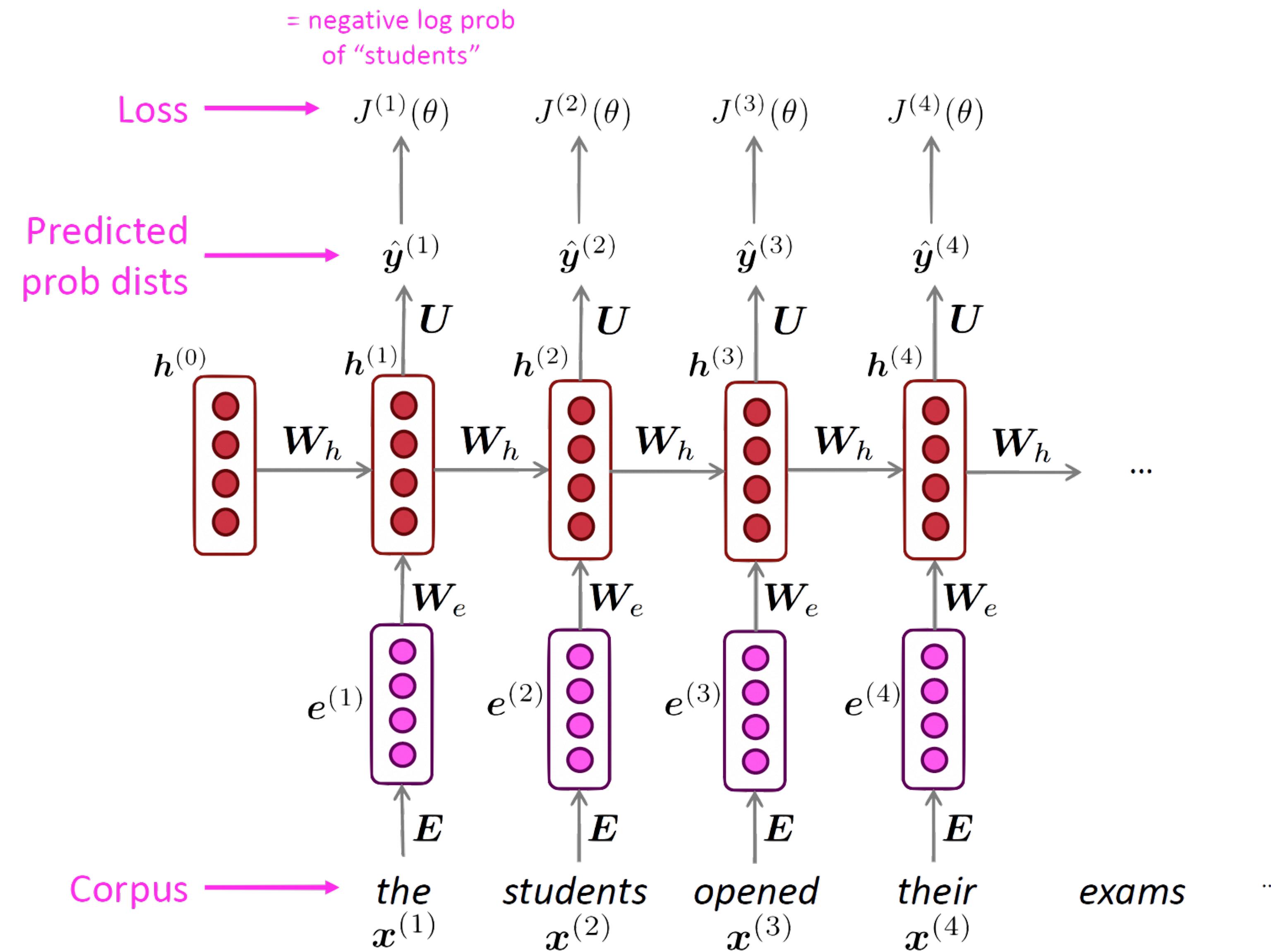
Training RNNs

- Forward pass = “unrolling”
- Loss computation:
 - many-to-one: at last observation/state (ignore previous outputs)
 - many-to-many: at every time step
- Gradient computation:
 - back-propagation through time (BPTT)
 - shared weights = shared gradients



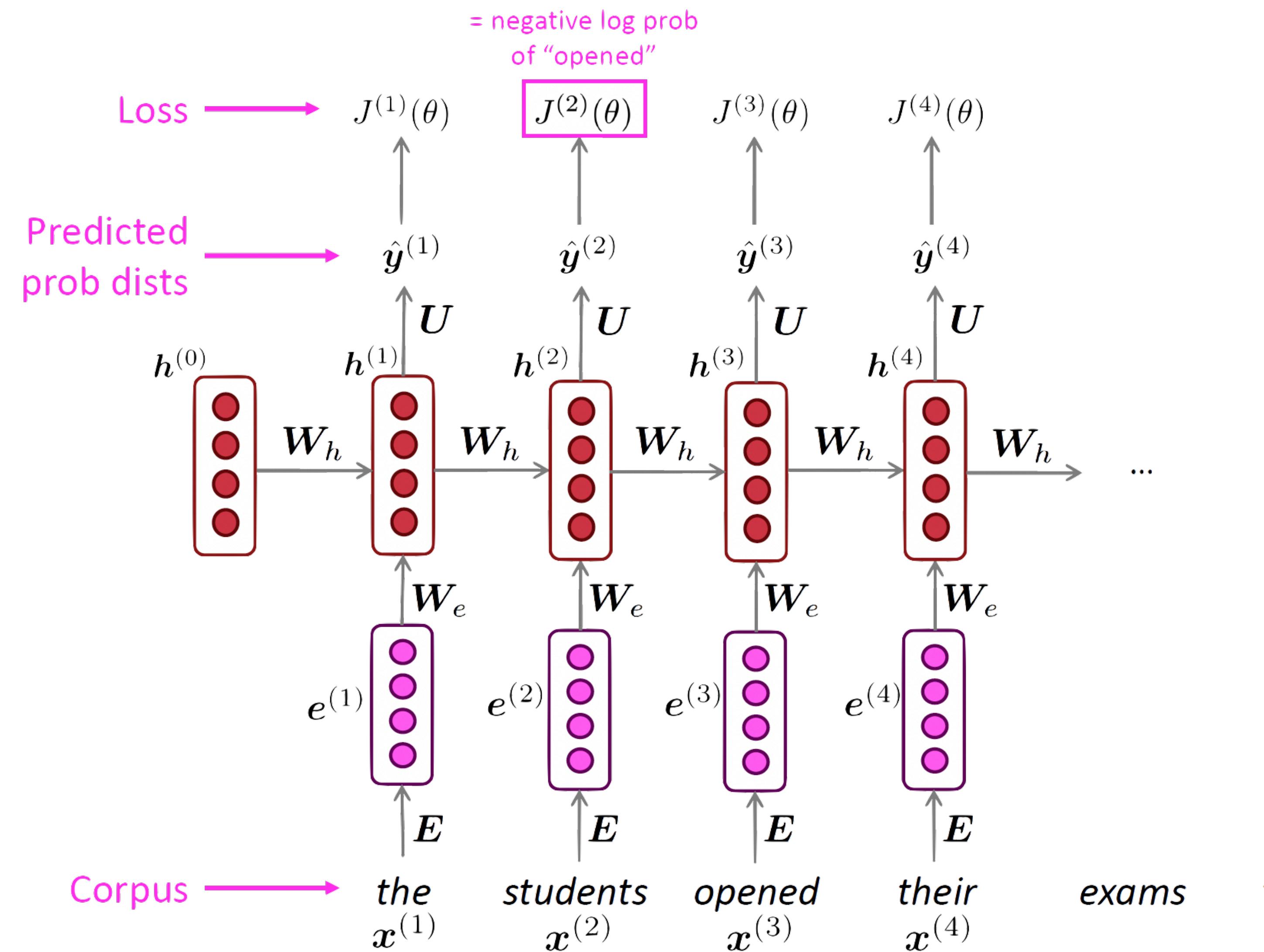
Figures: Manning et al.,
Stanford cs224n

Example: RNN-LM



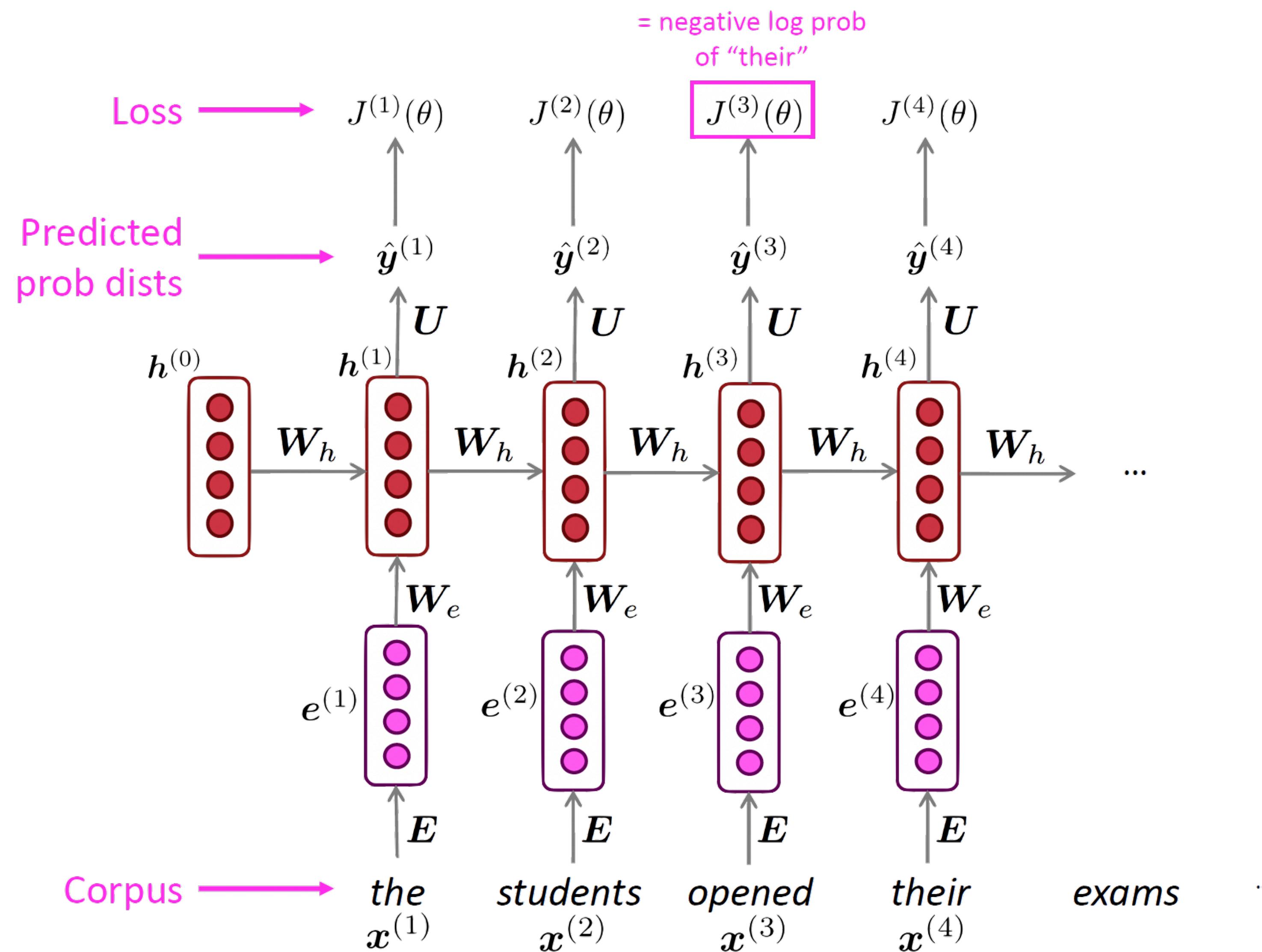
Figures: Manning et al.,
Stanford cs224n

Example: RNN-LM



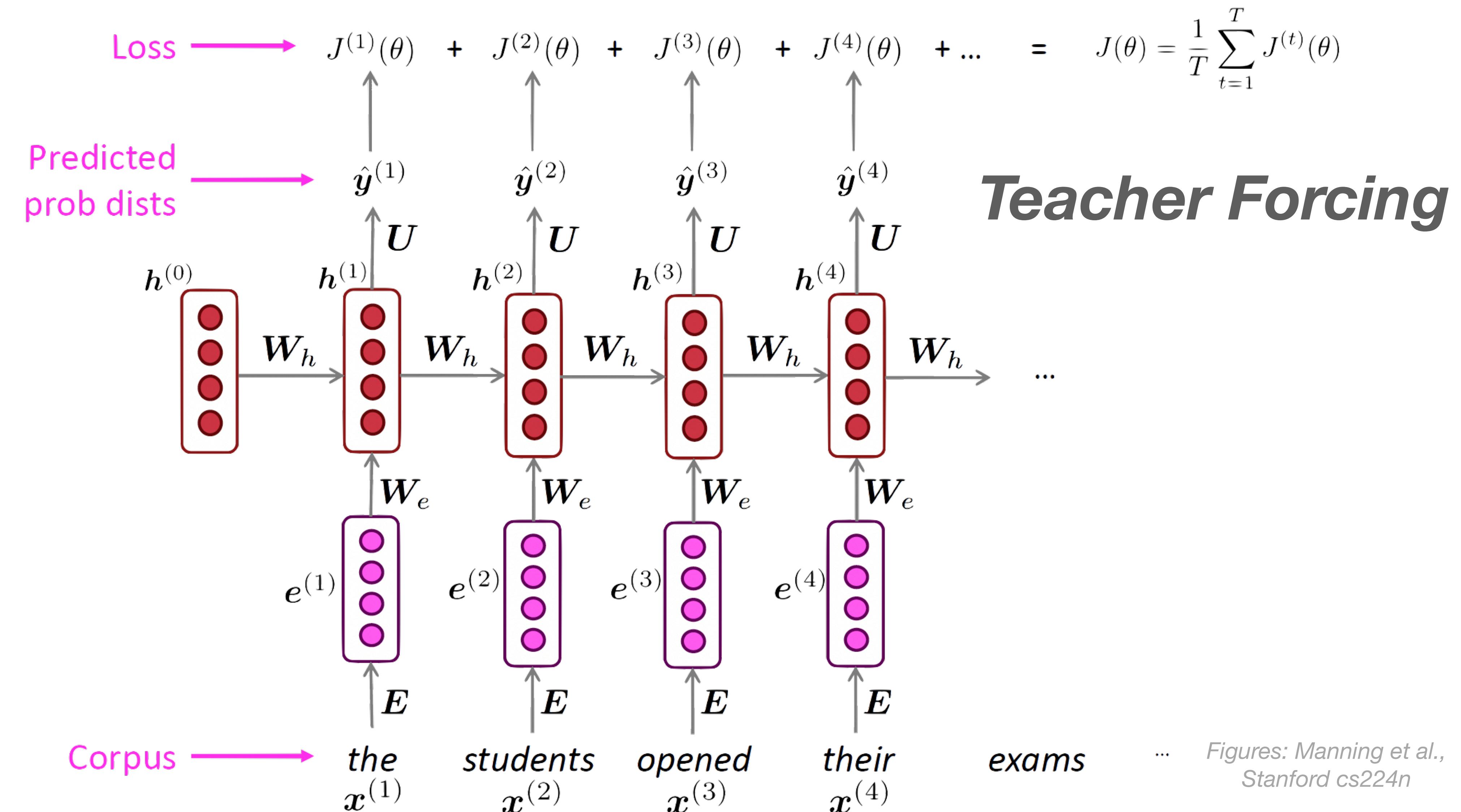
Figures: Manning et al., Stanford cs224n

Example: RNN-LM

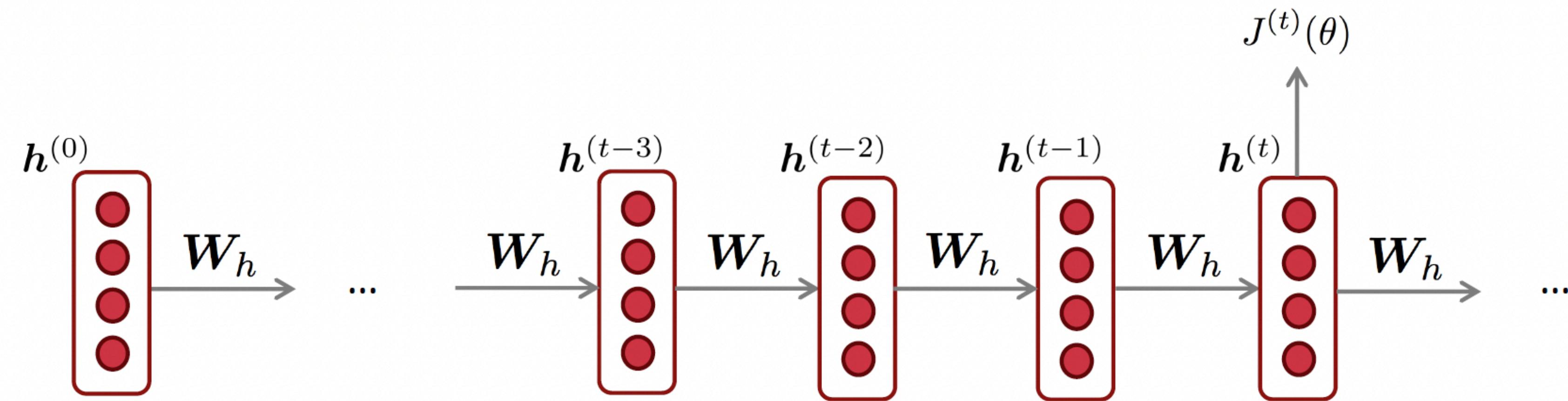


Figures: Manning et al.,
Stanford cs224n

Example: RNN-LM

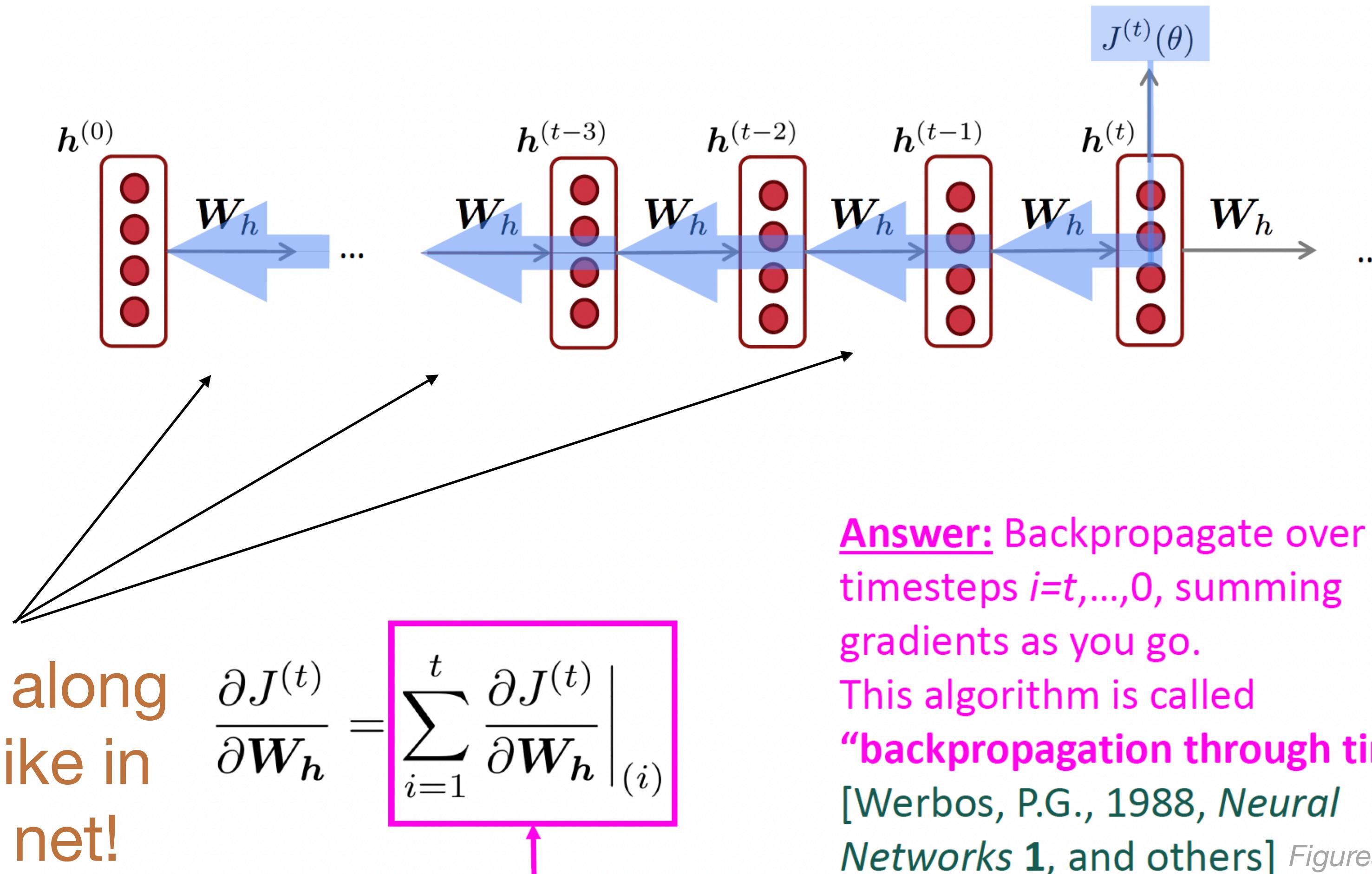


Back-Propagation for RNNs



Figures: Manning et al.,
Stanford cs224n

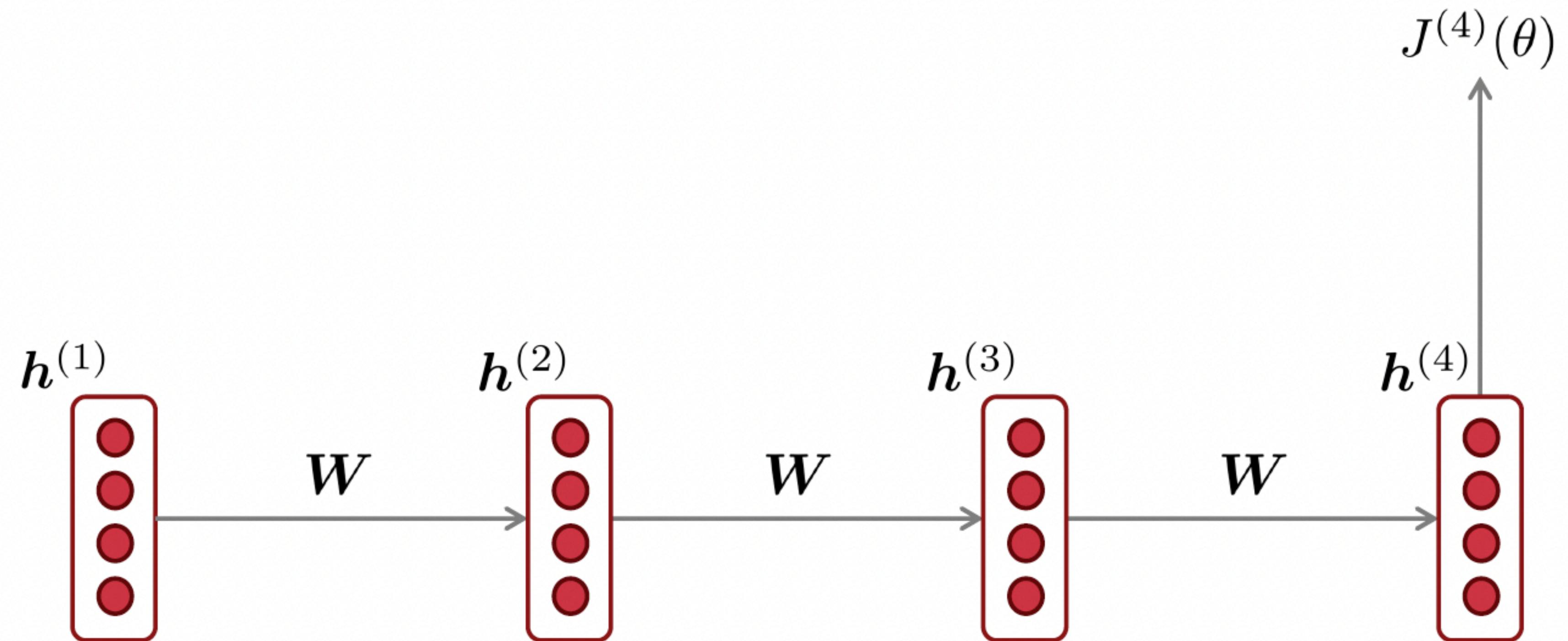
Back-Propagation Through Time (BPTT)



Answer: Backpropagate over timesteps $i=t, \dots, 0$, summing gradients as you go.
This algorithm is called “backpropagation through time”
[Werbos, P.G., 1988, *Neural Networks 1*, and others]

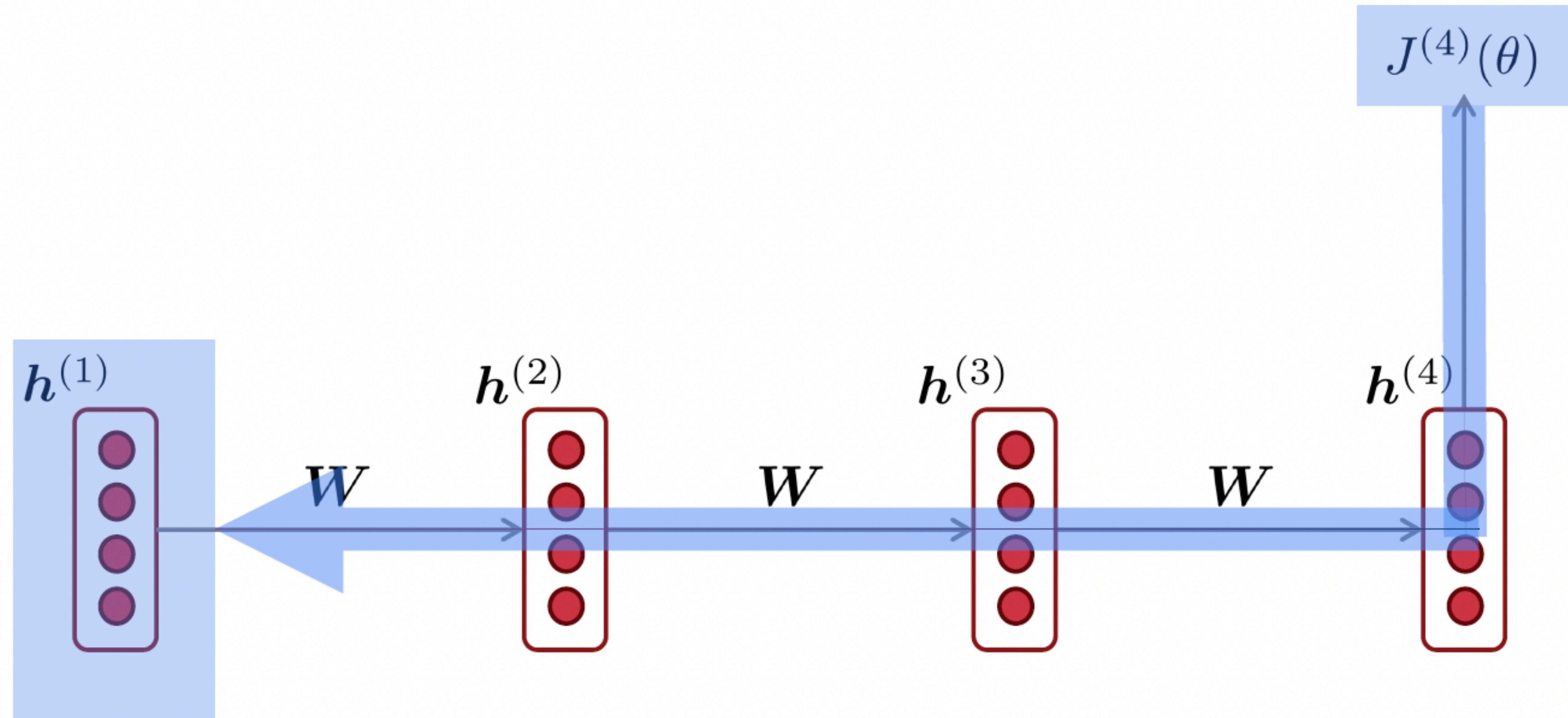
Figures: Manning et al.,
Stanford cs224n

Gradient Flow Revisited



Figures: Manning et al.,
Stanford cs224n

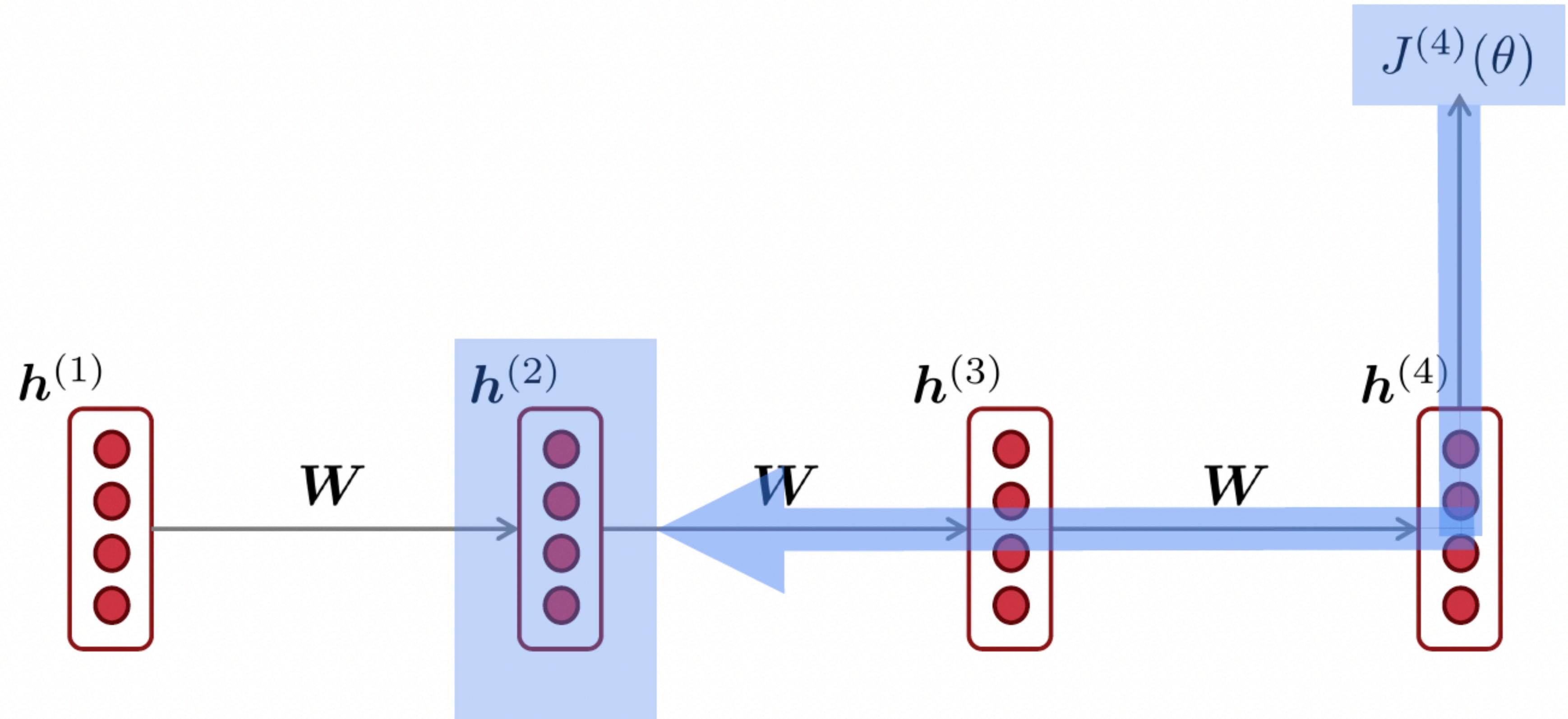
Gradient Flow Revisited



$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = ?$$

Figures: Manning et al.,
Stanford cs224n

Gradient Flow Revisited

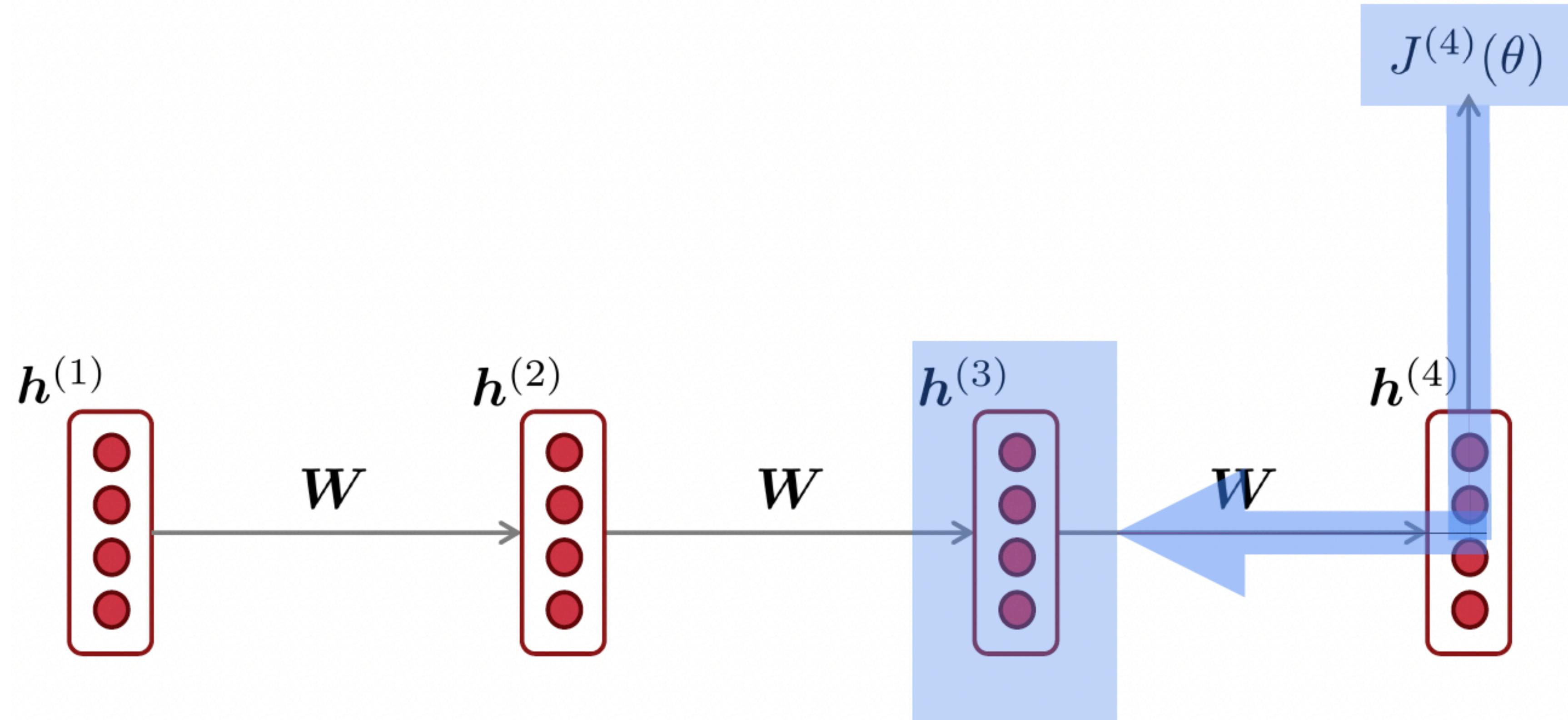


$$\frac{\partial J^{(4)}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}} \times \frac{\partial J^{(4)}}{\partial \mathbf{h}^{(2)}}$$

chain rule!

Figures: Manning et al.,
Stanford cs224n

Gradient Flow Revisited



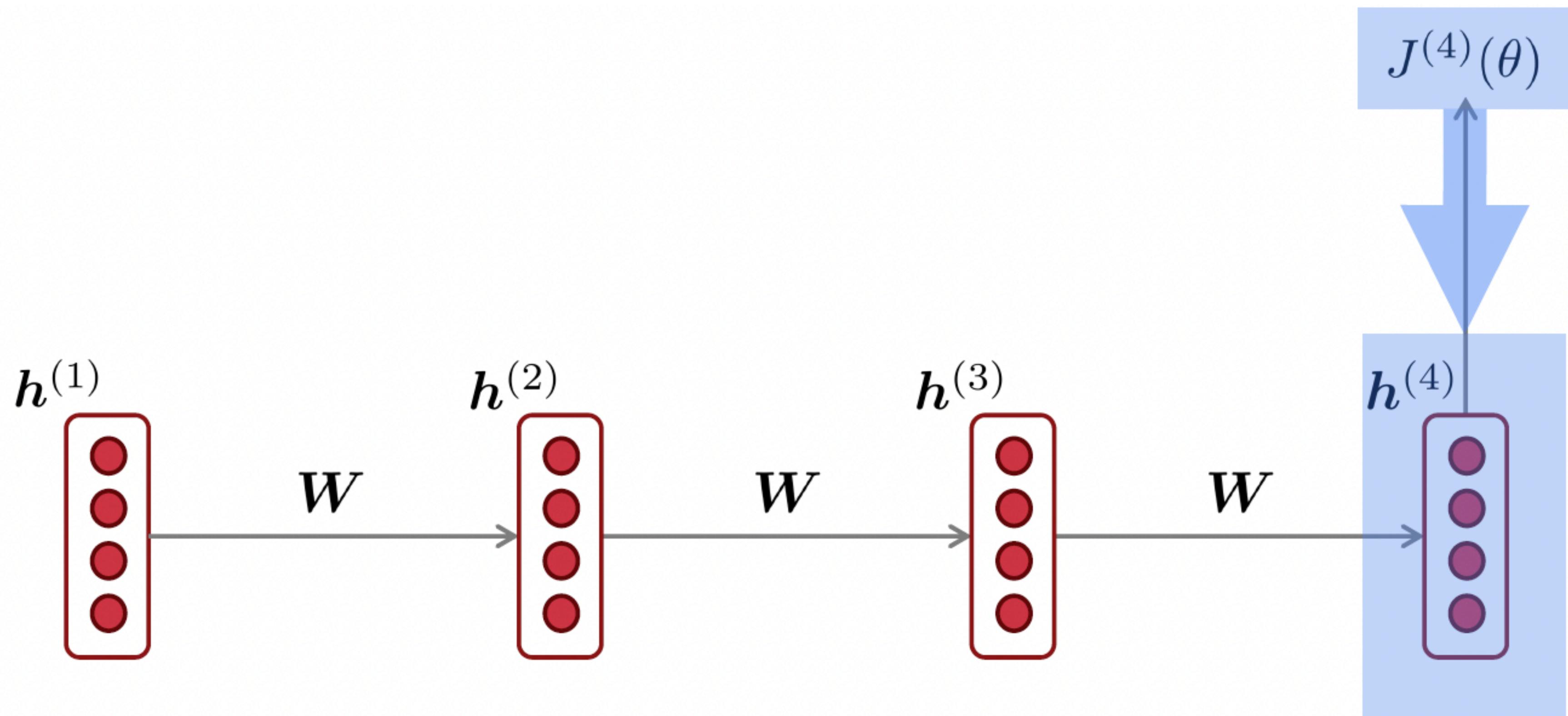
$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times$$

$$\frac{\partial h^{(3)}}{\partial h^{(2)}} \times \frac{\partial J^{(4)}}{\partial h^{(3)}}$$

chain rule!

Figures: Manning et al.,
Stanford cs224n

Gradient Flow Revisited



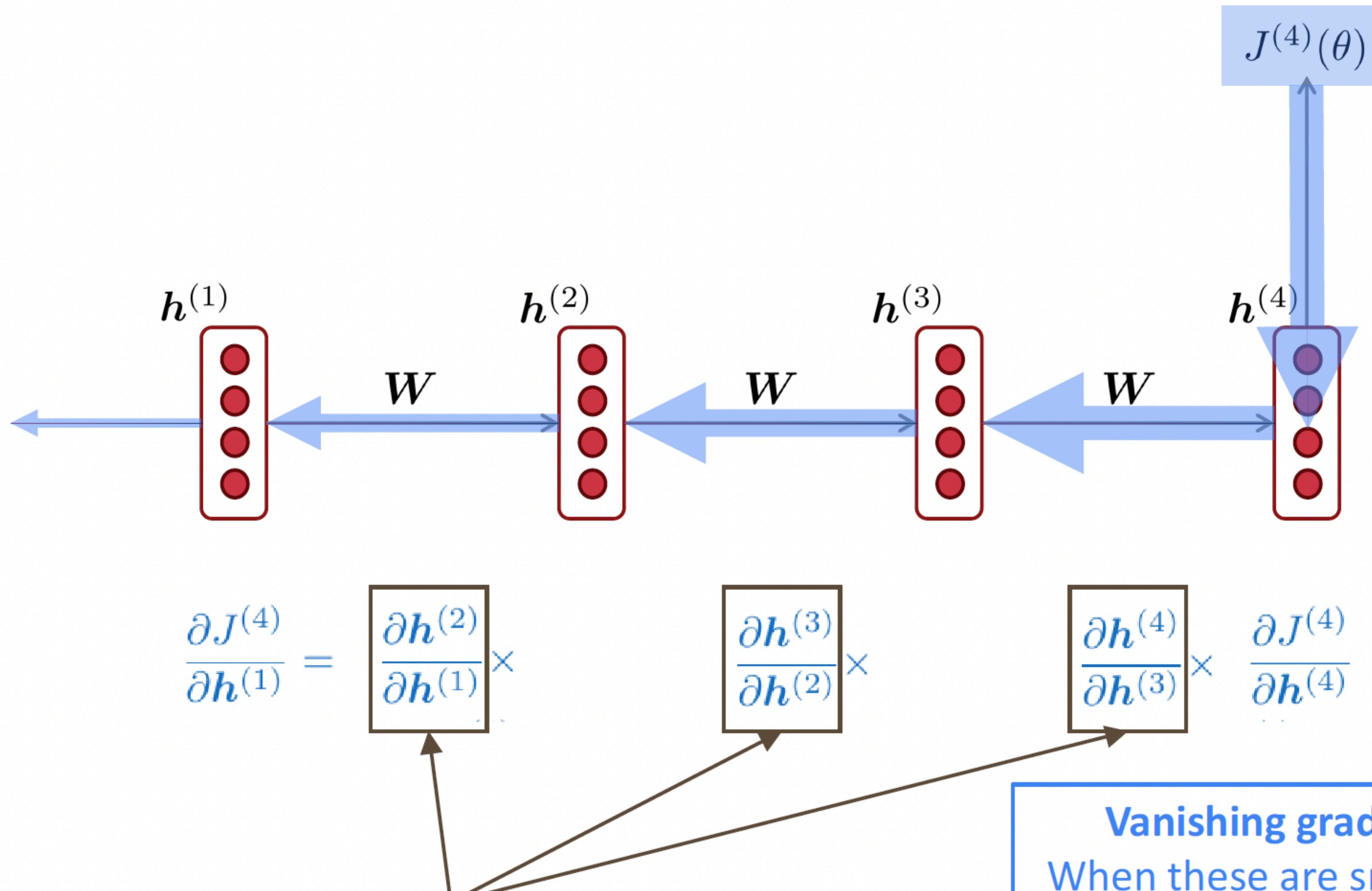
$$\frac{\partial J^{(4)}}{\partial \mathbf{h}^{(1)}} = \frac{\partial \mathbf{h}^{(2)}}{\partial \mathbf{h}^{(1)}} \times$$

$$\frac{\partial \mathbf{h}^{(3)}}{\partial \mathbf{h}^{(2)}} \times$$

$$\frac{\partial \mathbf{h}^{(4)}}{\partial \mathbf{h}^{(3)}} \times \frac{\partial J^{(4)}}{\partial \mathbf{h}^{(4)}}$$

chain rule!

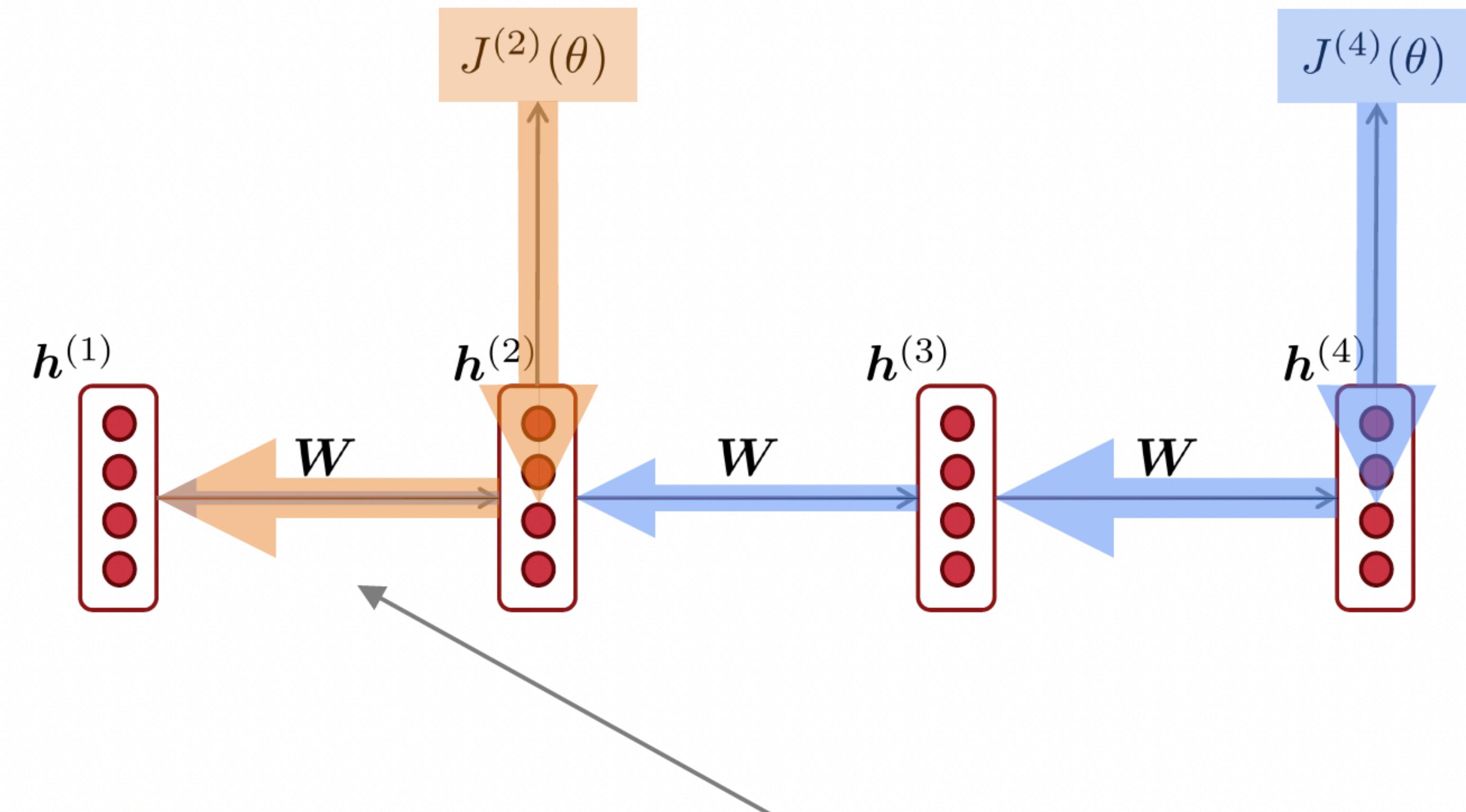
Gradient Flow Revisited



What happens if these are small?

Vanishing gradient problem:
When these are small, the gradient signal gets smaller and smaller as it backpropagates further

Gradient Flow Revisited



Gradient signal from far away is lost because it's much smaller than gradient signal from close-by.

So, model weights are updated only with respect to **near effects**, not **long-term effects**.

Exploding Gradient?

$$\theta^{new} = \theta^{old} - \overbrace{\alpha \nabla_{\theta} J(\theta)}^{\text{gradient}}$$

learning rate

What about the *vanishing* gradient?

$$\mathbf{h}^{(t)} = \sigma \left(\mathbf{W}_h \mathbf{h}^{(t-1)} + \mathbf{W}_x \mathbf{x}^{(t)} + \mathbf{b} \right)$$

Long Short-Term Memory RNNs (LSTMs)

- Hochreiter & Schmidhuber, 1997 (and Gers et al., 2000)
- At each time t , there is a hidden state $\mathbf{h}^{(t)}$ and cell state $\mathbf{c}^{(t)}$
 - Both are vectors length n
 - Cell stores long-term information in \mathbf{c}
 - LSTM can read, erase and write information from/to the cell
 - “read” etc. is metaphorically... it’s all matrix math ops

LSTM: the math

Forget gate: controls what is kept vs forgotten, from previous cell state

Input gate: controls what parts of the new cell content are written to cell

Output gate: controls what parts of cell are output to hidden state

New cell content: this is the new content to be written to the cell

Cell state: erase (“forget”) some content from last cell state, and write (“input”) some new cell content

Hidden state: read (“output”) some content from the cell

Sigmoid function: all gate values are between 0 and 1

$$\begin{aligned} f^{(t)} &= \sigma(W_f h^{(t-1)} + U_f x^{(t)} + b_f) \\ i^{(t)} &= \sigma(W_i h^{(t-1)} + U_i x^{(t)} + b_i) \\ o^{(t)} &= \sigma(W_o h^{(t-1)} + U_o x^{(t)} + b_o) \end{aligned}$$

$$\tilde{c}^{(t)} = \tanh(W_c h^{(t-1)} + U_c x^{(t)} + b_c)$$

$$c^{(t)} = f^{(t)} \circ c^{(t-1)} + i^{(t)} \circ \tilde{c}^{(t)}$$

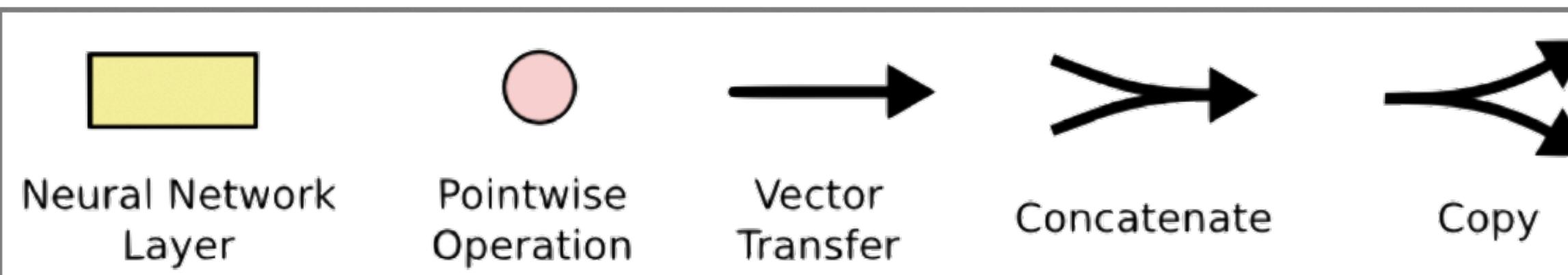
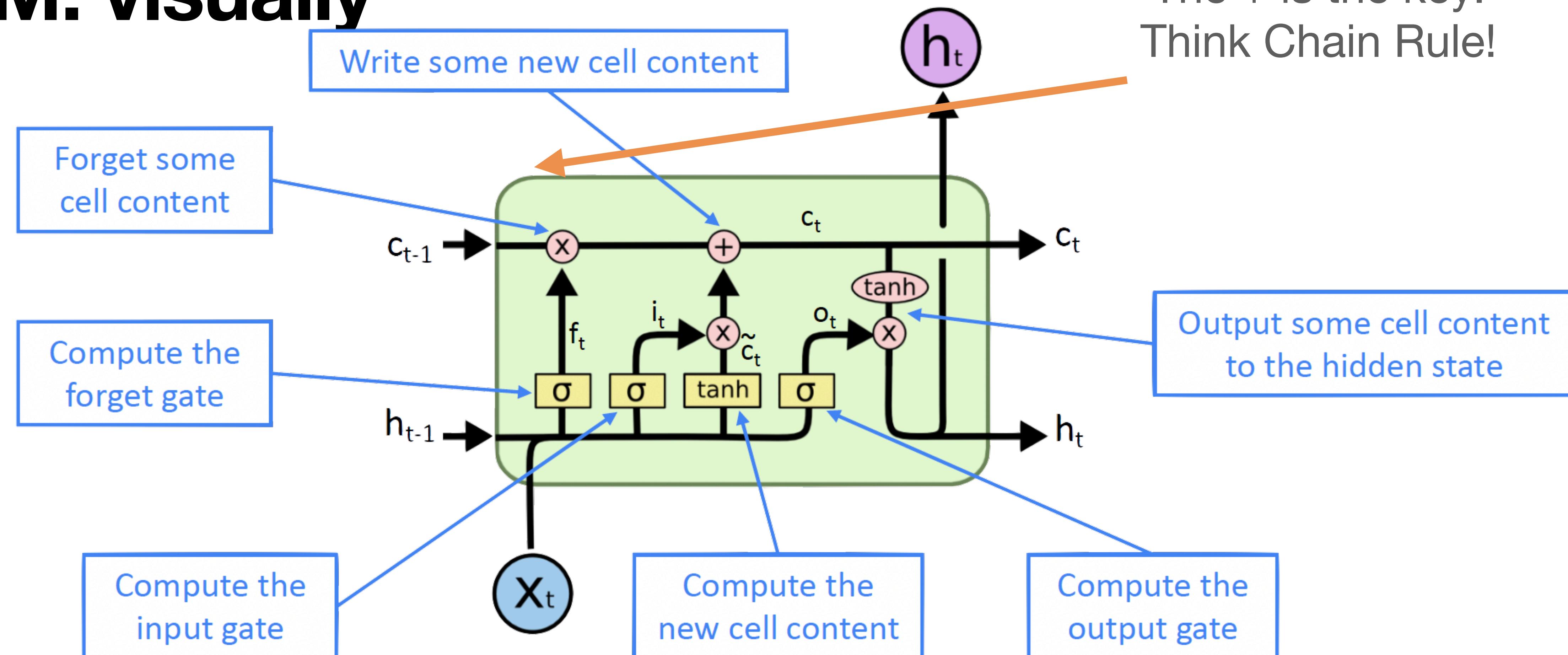
$$h^{(t)} = o^{(t)} \circ \tanh c^{(t)}$$

Gates are applied using element-wise (or Hadamard) product: \odot

All these are vectors of same length n

LSTM: visually

The + is the key!
Think Chain Rule!



Why does it work (or at least help)?

- The cell state is based on (scalar) multiplication and addition
→ numerically stable
- Gate values at 1 or 0 can help to preserve/delete information
- ...no winner takes all: LSTM does better, but has similar issues like RNNs
- In 2013-2015, LSTMs became SOTA for many sequence tasks (and predominant for NLP tasks)
- (now pretty much replaced by transformers)

Vanishing Gradient (revisited)

- VG is not just an RNN problem!
 - The deeper the (FF) net, the more likely it is (chain rule)
 - Choice of non-linearity is crucial
 - Lower layers learn slow/hard to train
- Solution for FF-DNNs:
 - residual connections
(aka skip-connections, “ResNets”)
 - In principle similar to LSTM approach
- The main issue/risk is repeated multiplication of same weight matrix

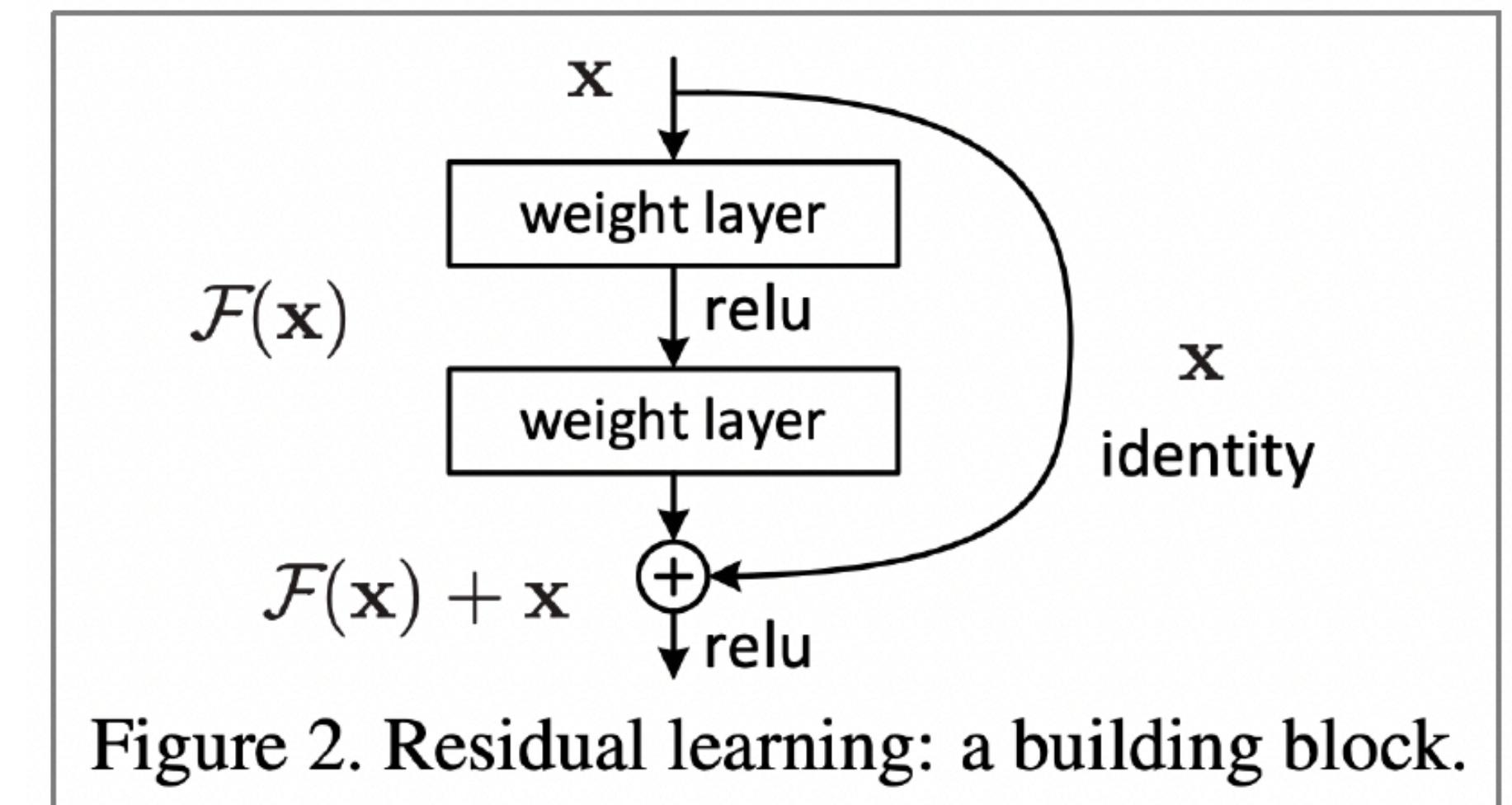
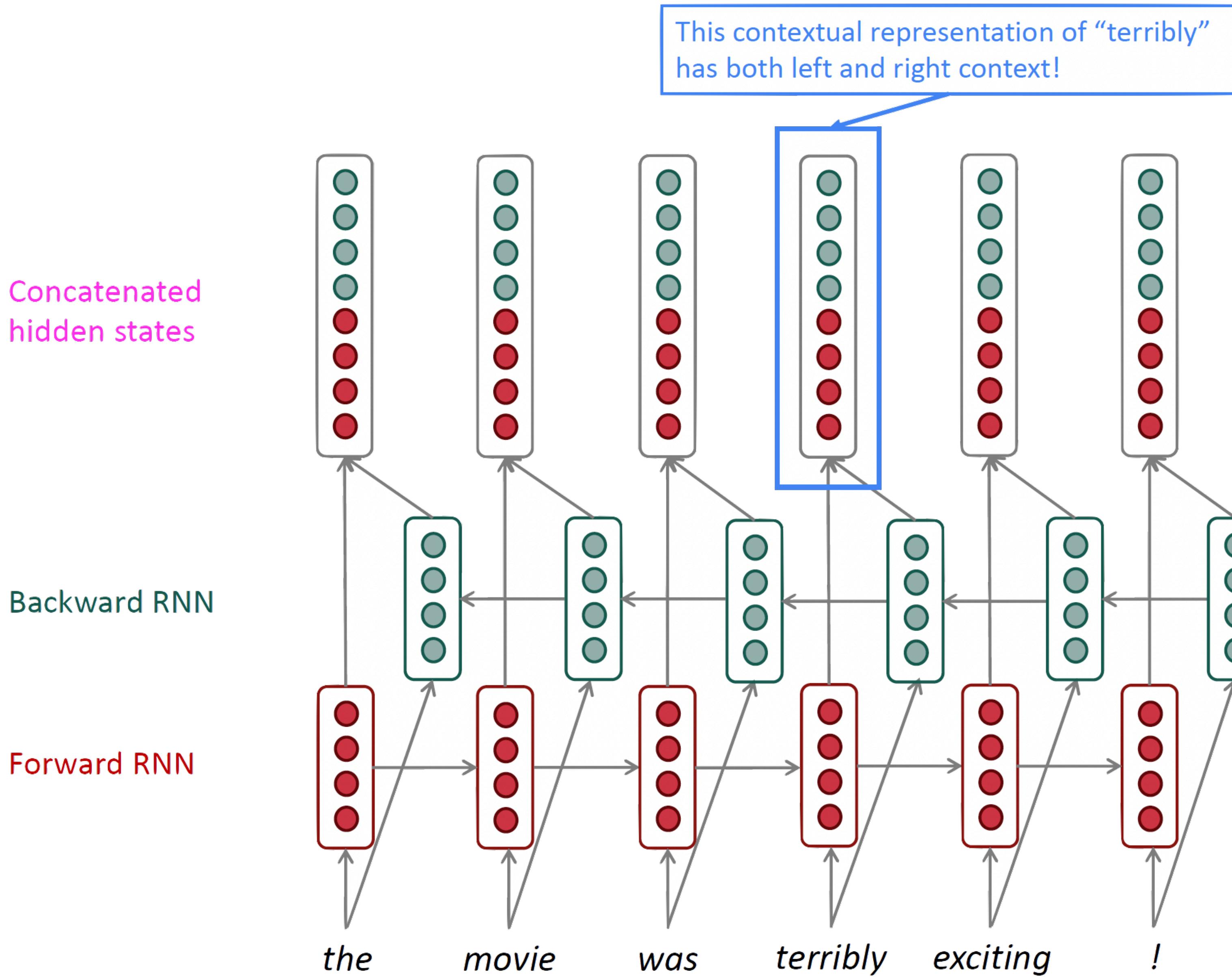


Figure 2. Residual learning: a building block.

“Deep Residual Learning for Image Recognition”,
He et al, 2015. <https://arxiv.org/pdf/1512.03385.pdf>

Extensions: Bi-directional LSTM



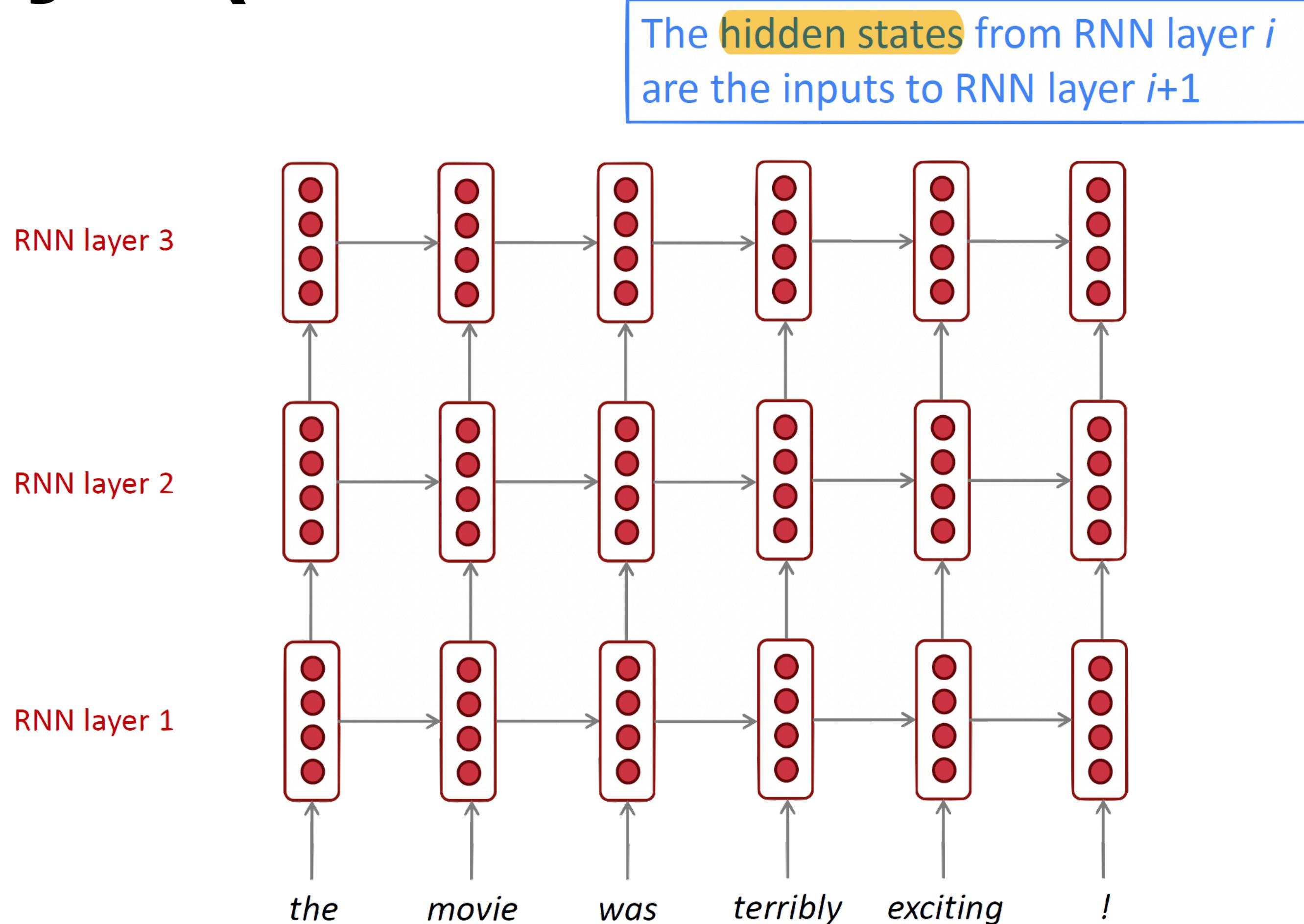
$$\begin{aligned}\vec{h}^{(t)} &= \text{RNN}_{\text{FW}}(\vec{h}^{(t-1)}, \mathbf{x}^{(t)}) \\ \overleftarrow{h}^{(t)} &= \text{RNN}_{\text{BW}}(\overleftarrow{h}^{(t+1)}, \mathbf{x}^{(t)}) \\ h^{(t)} &= [\vec{h}^{(t)}; \overleftarrow{h}^{(t)}]\end{aligned}$$

Requires access to full input sequence!

Figures: Manning et al., Stanford cs224n

Extensions: Multi-layer (stacked) LSTM

- RNNs are already “deep” in one dimension (unrolling)
- Make them “deeper” by applying multiple RNNs (“stacking”)
- Motivation similar to stacked filters: learn different sets of representations



Figures: Manning et al., Stanford cs224n

Summary

- RNNs encode sequence “history” in a hidden state
 - The repeated multiplication of the shared weight matrix is prone to VG!
- LSTMs improve over RNNs, conceptually & numerically
- Use bi-directionality if you have access to the full sequence
- RNN/LSTM help with several sequence problems:
 - *many-to-one*: apply linear layer on last output/hidden state
 - *many-to-many*: apply linear layer at each time step
 - *one-to-many*: yes, you can just keep sampling from an RNN :-)
 - ...how about many-to-many with $M \neq N$?