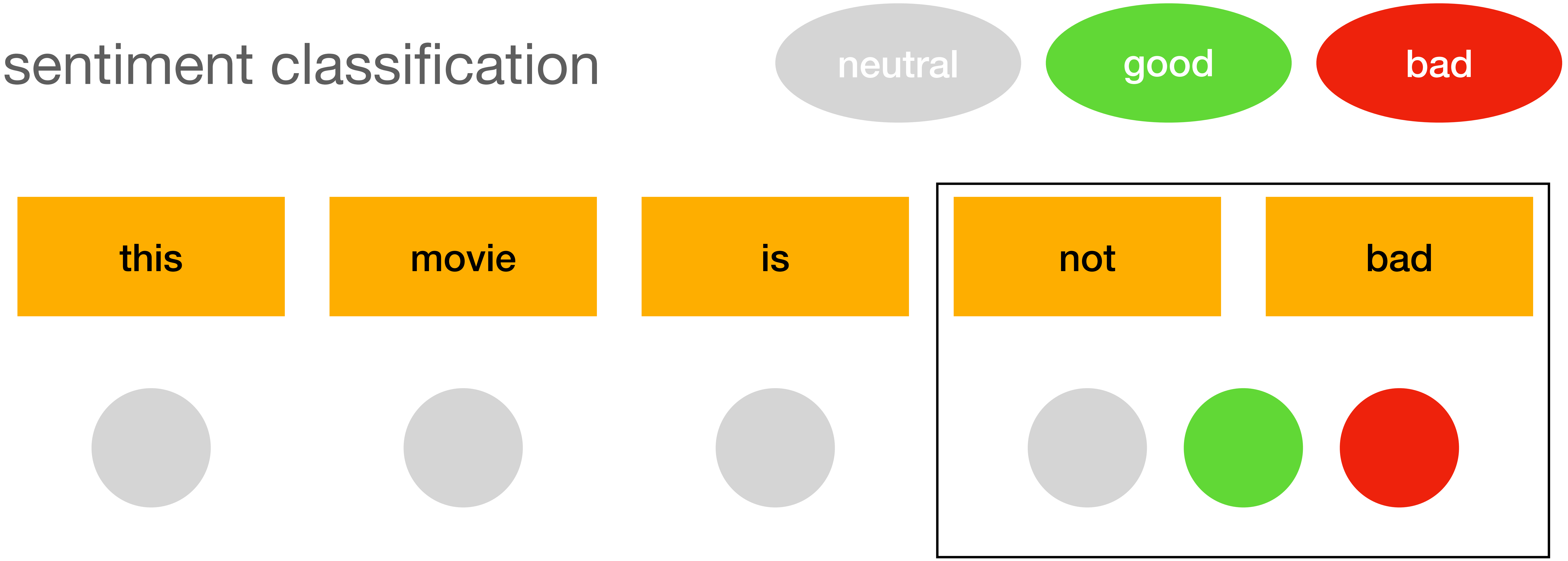
Session 7: Recurrent Neural Networks

Korbinian Riedhammer

TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

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?

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:
  - $\boldsymbol{x} = x_1, x_2, \ldots, x_M$: sequence of M observations
  - $\boldsymbol{y} = y_1, y_2, \ldots, y_N$: sequence of N outputs (class labels)
- …this will add another dimension to our tensors!

Sequences!

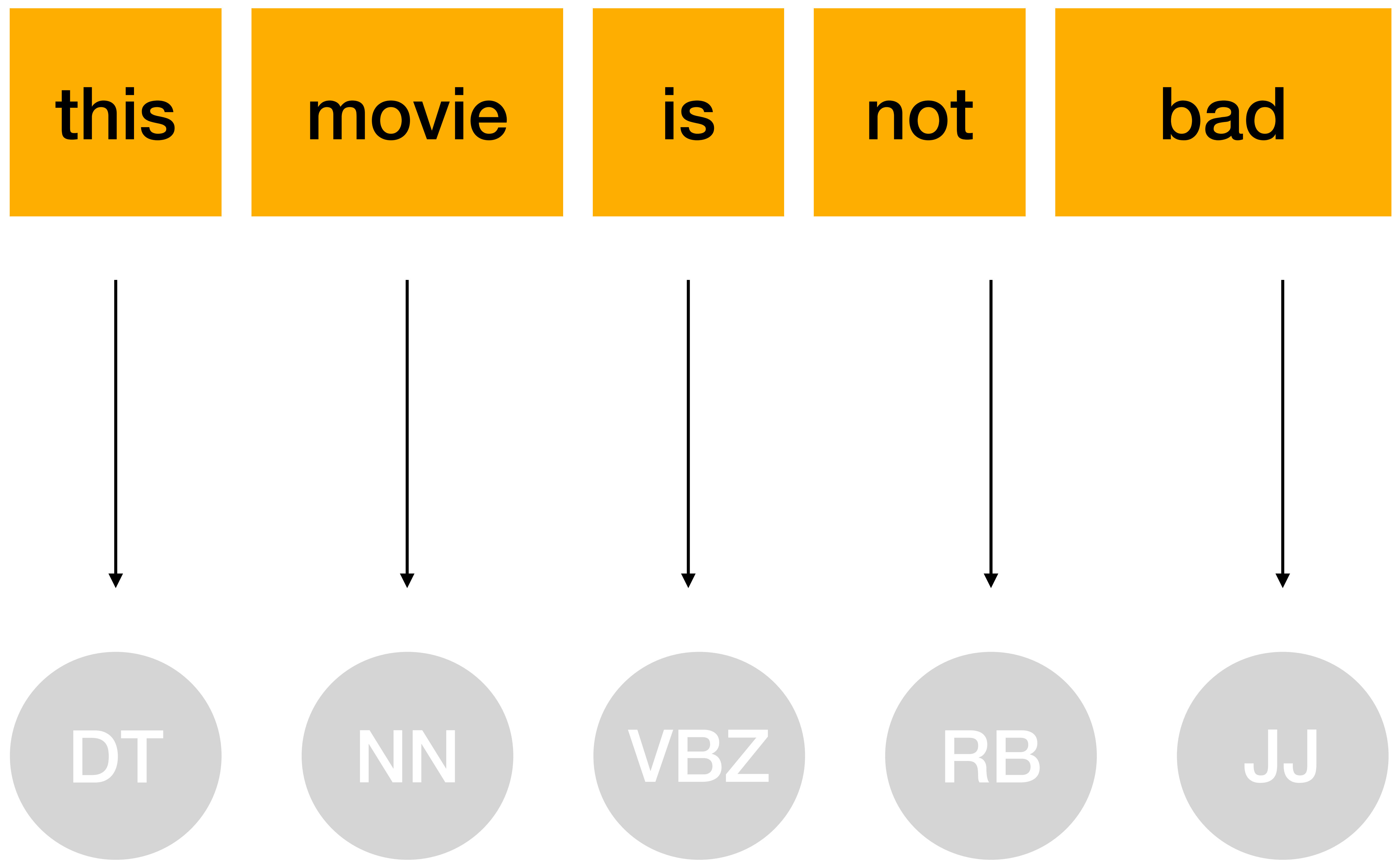
- $M > 1, \quad N = 1$: “many-to-one”
  - speaker identification
  - sentiment classification
  - fraudulent transaction
- $M > 1, \quad N > 1$: “many-to-many”
  - $M \equiv N$: “time-synchronous”, strict 1:1 alignment $x_t \rightarrow y_t$
    - speech recognition, part-of-speech tagging, …
  - $M \neq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

Many-to-One

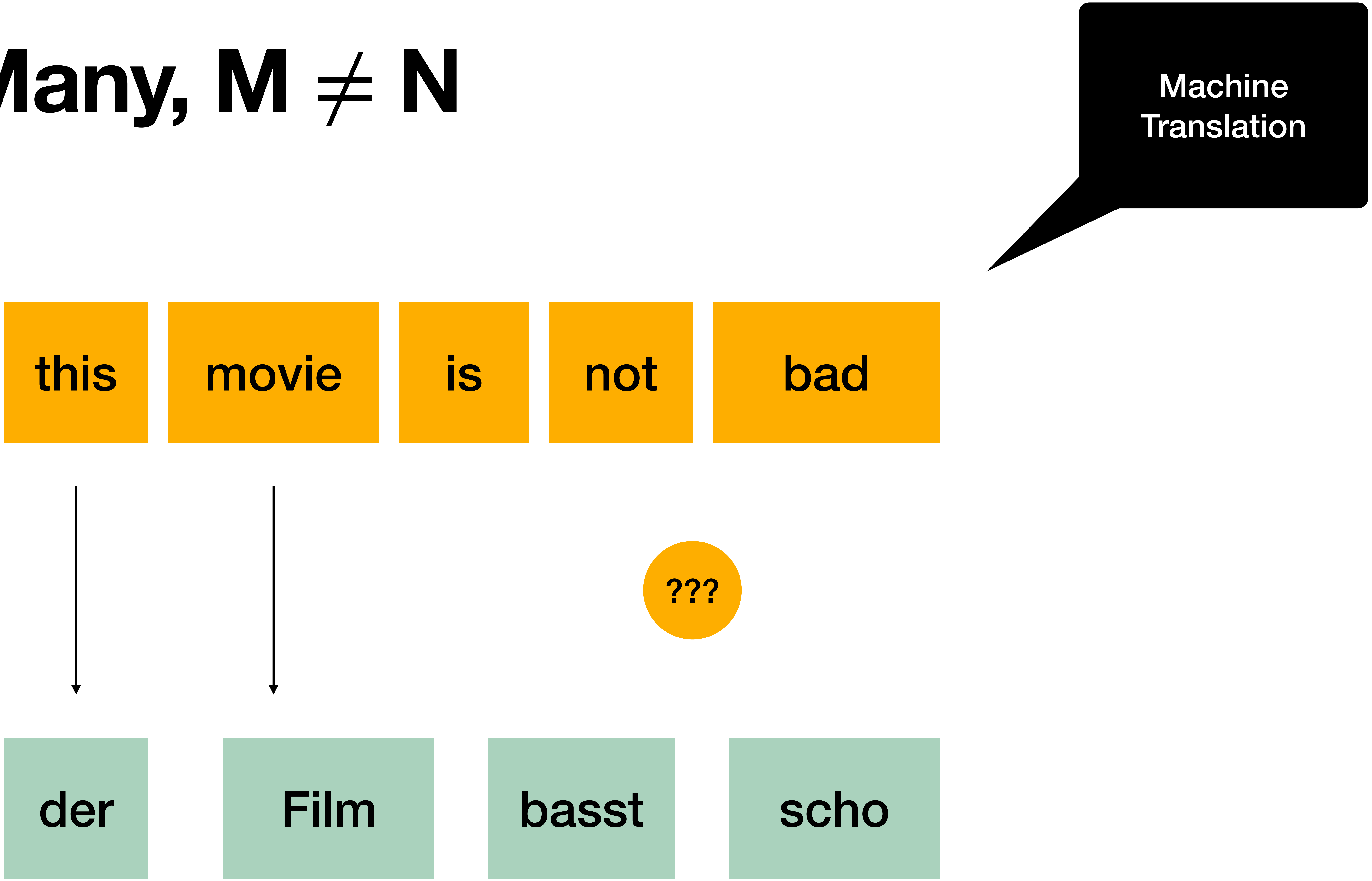
Sentiment Classification

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

Part-of-Speech Tagging



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

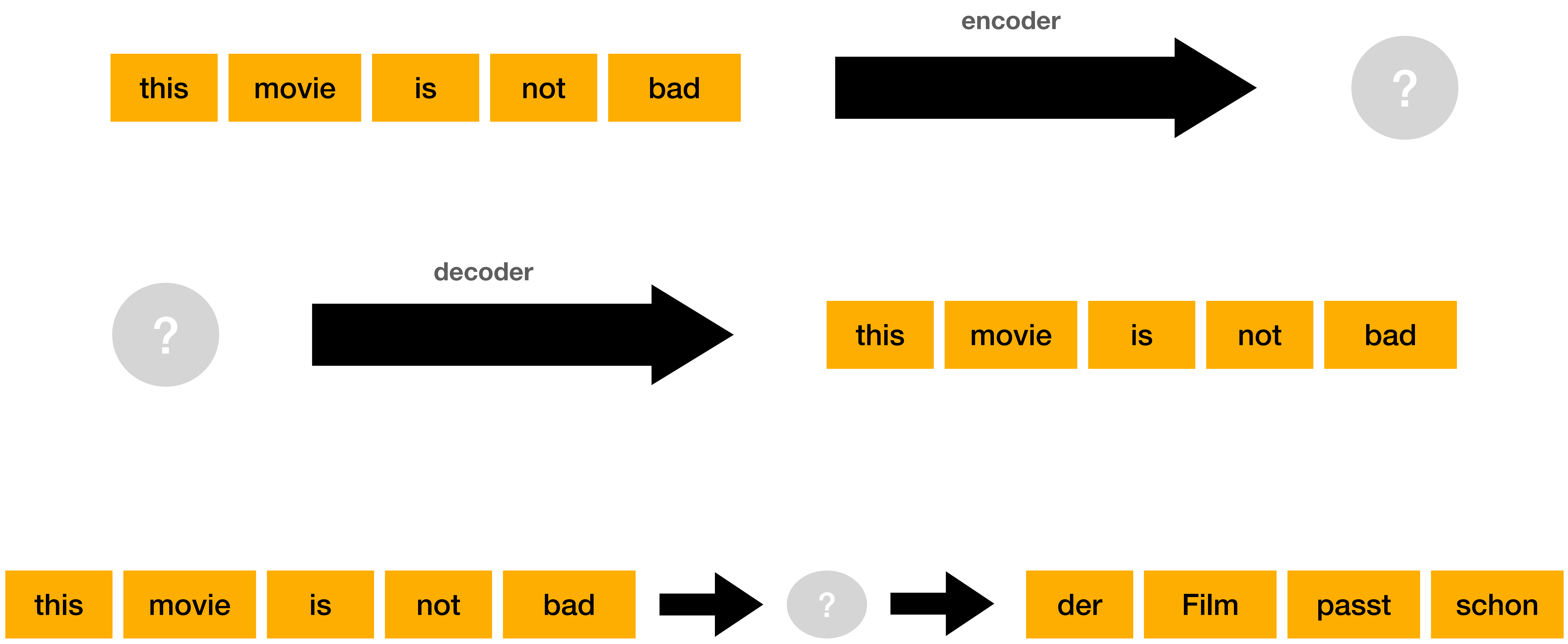


One-to-Many

Image Captioning

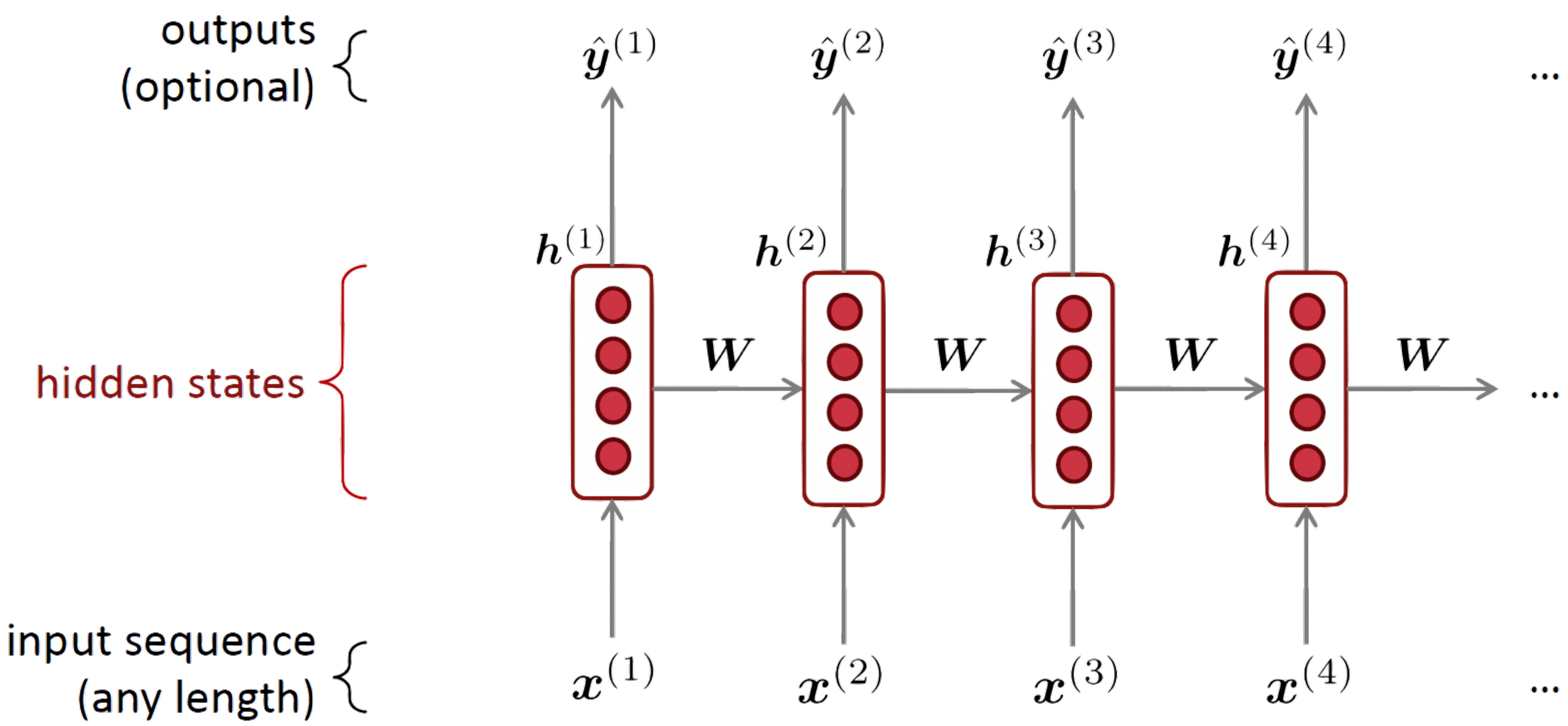
old man with woolen mittens

Terminology (revisited)



encoder - decoder

Recurrent Neural Networks



Figures: Manning et al., Stanford cs224n

Recurrent Neural Networks

output distribution

$$\hat{\boldsymbol{y}}^{(t)} = \text{softmax}\left(\boldsymbol{U}\boldsymbol{h}^{(t)} + \boldsymbol{b}_2\right) \in \mathbb{R}^{|V|}$$

hidden states

$$\boldsymbol{h}^{(t)} = \sigma\left(\boldsymbol{W}_h\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_e\boldsymbol{e}^{(t)} + \boldsymbol{b}_1\right)$$

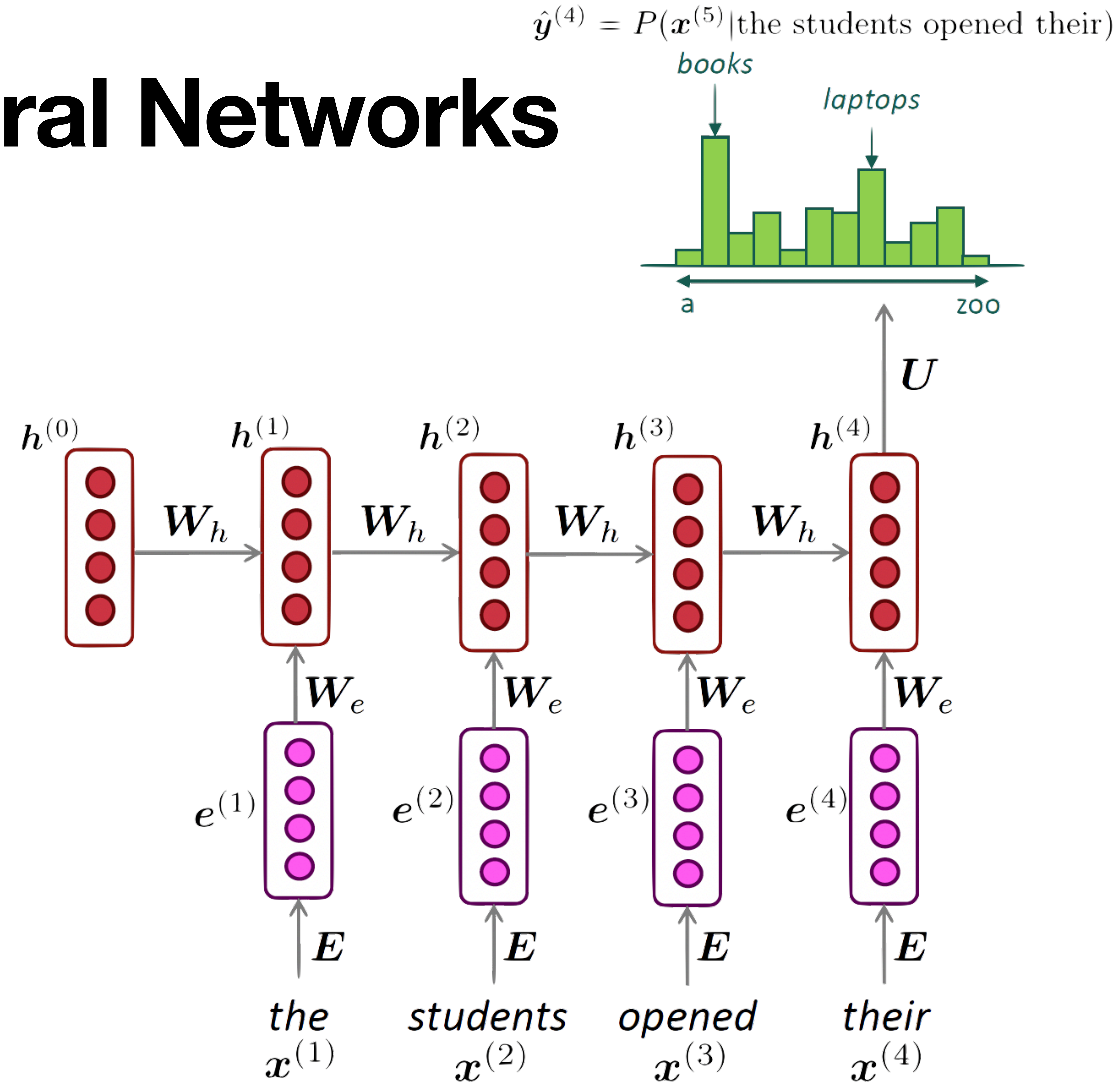
$\boldsymbol{h}^{(0)}$ is the initial hidden state

word embeddings

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

words / one-hot vectors

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

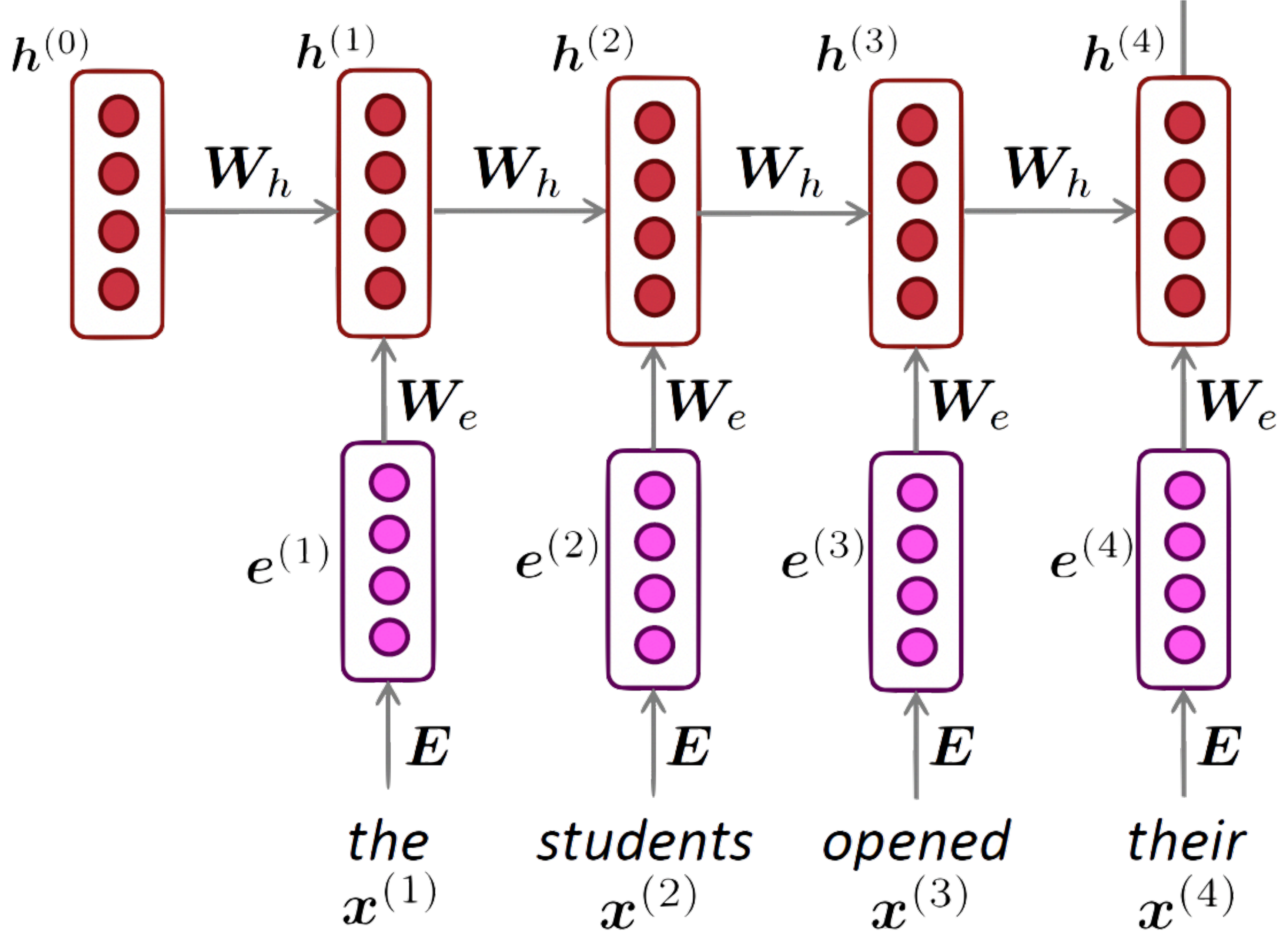


Figures: Manning et al., Stanford cs224n

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

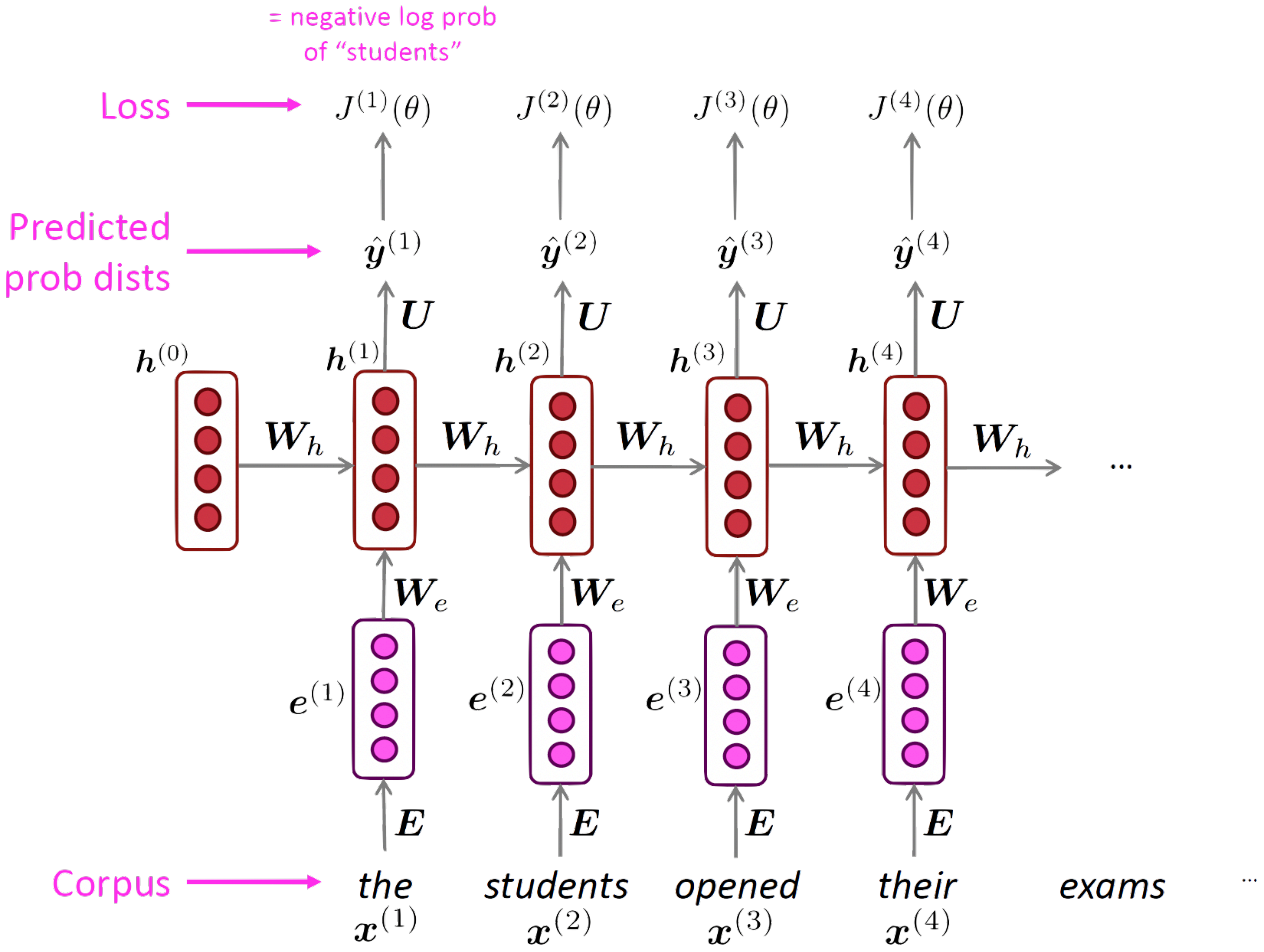
Training RNNs



- 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

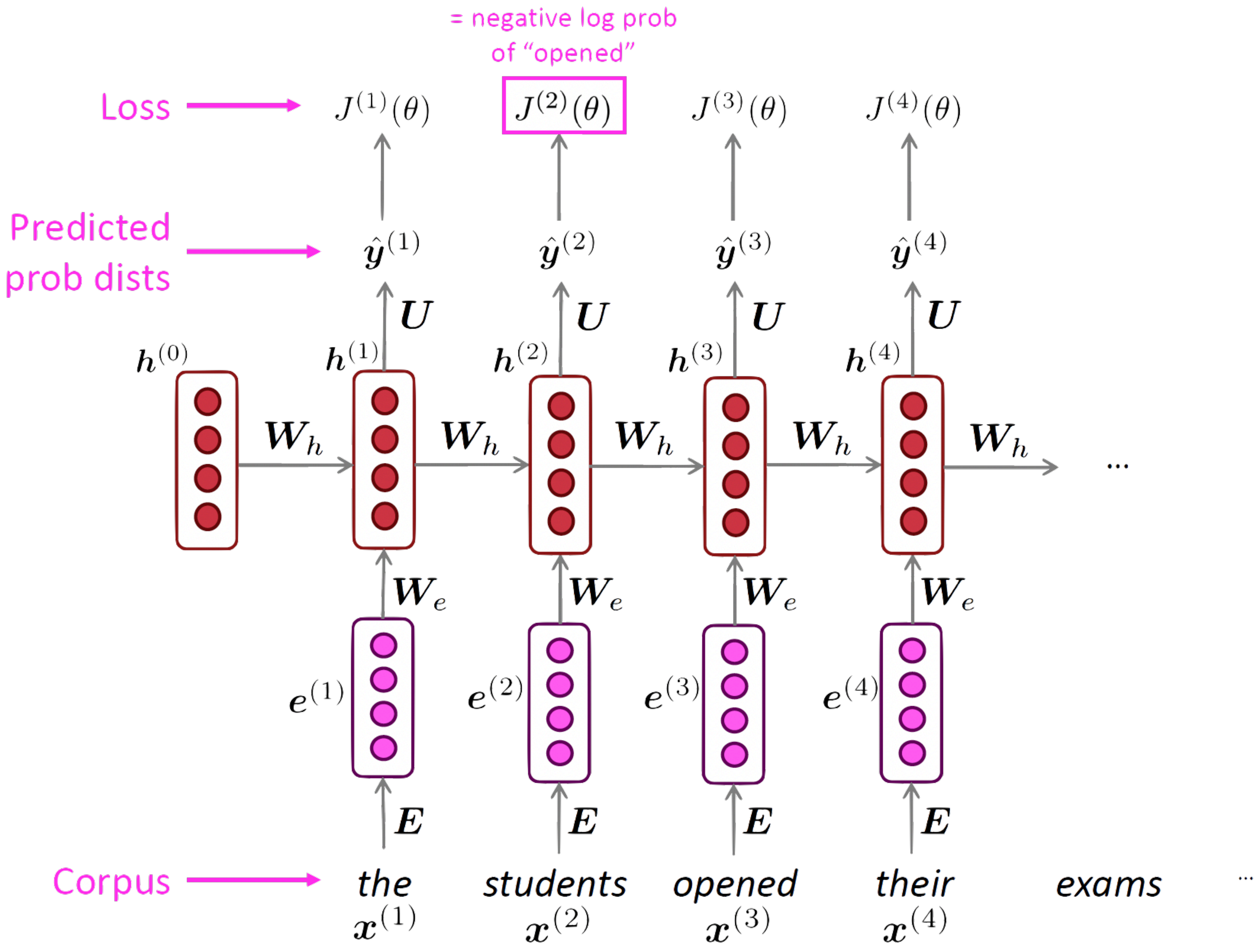
Figures: Manning et al., Stanford cs224n

Example: RNN-LM



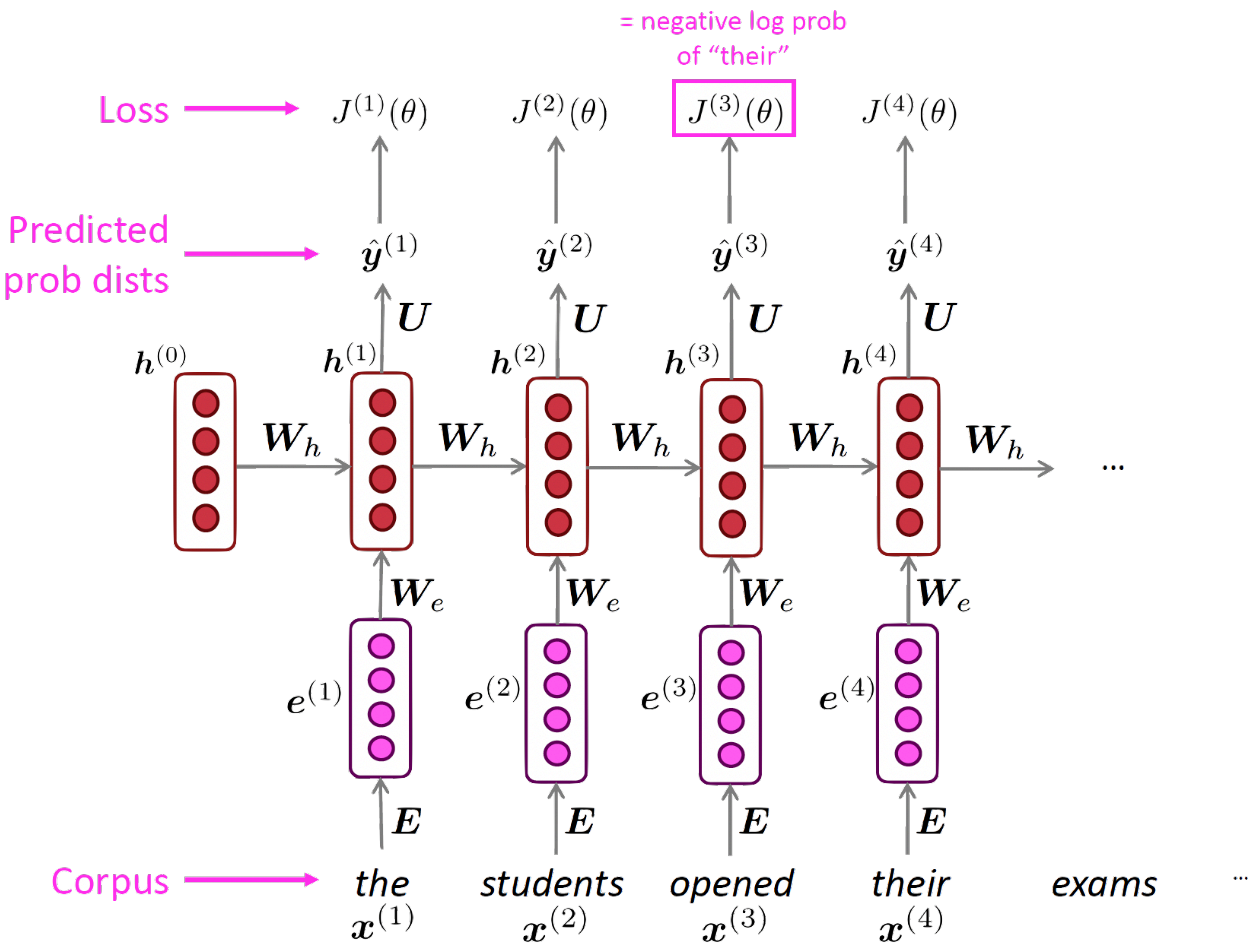
Figures: Manning et al., Stanford cs224n

Example: RNN-LM



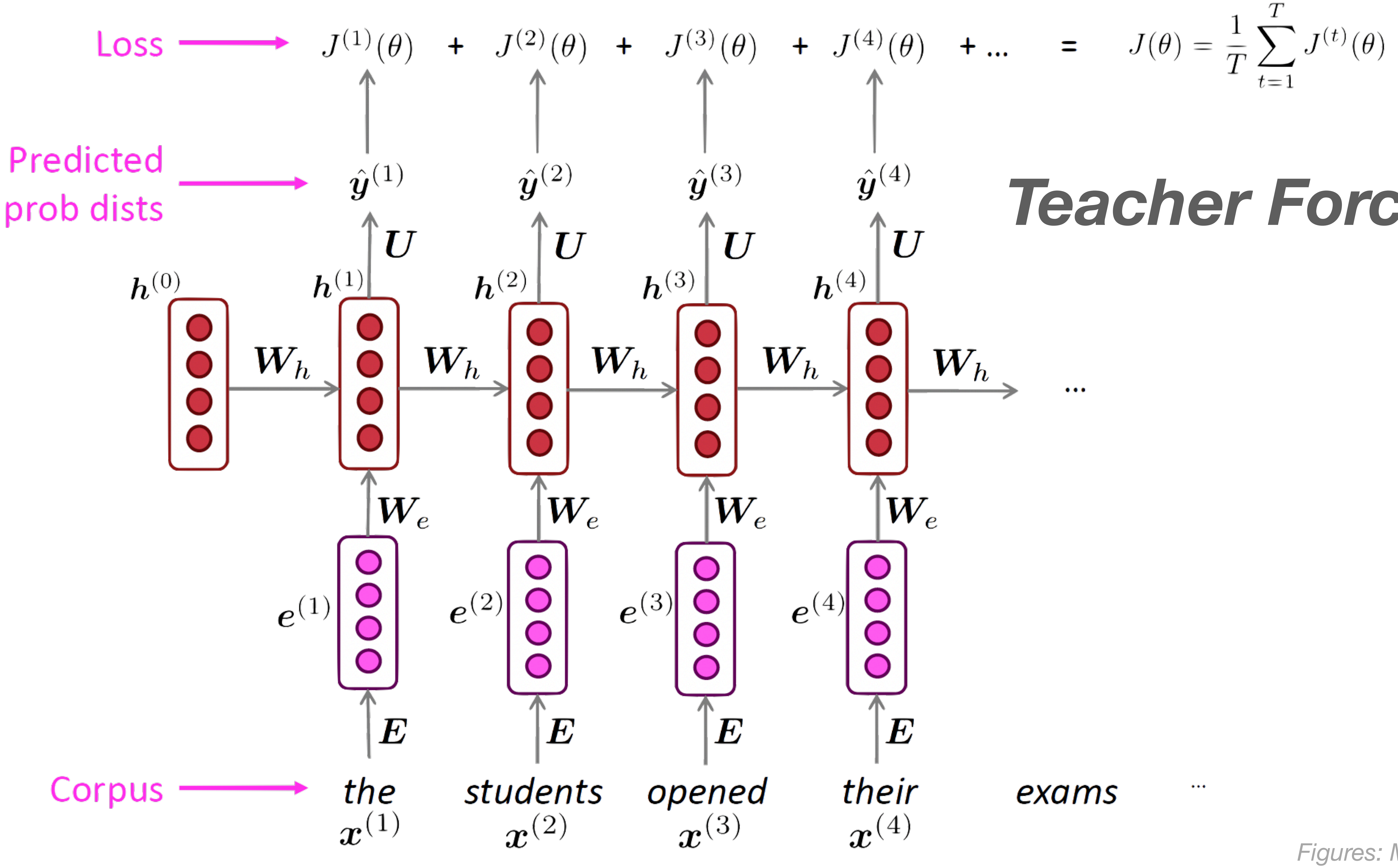
Figures: Manning et al., Stanford cs224n

Example: RNN-LM



Figures: Manning et al., Stanford cs224n

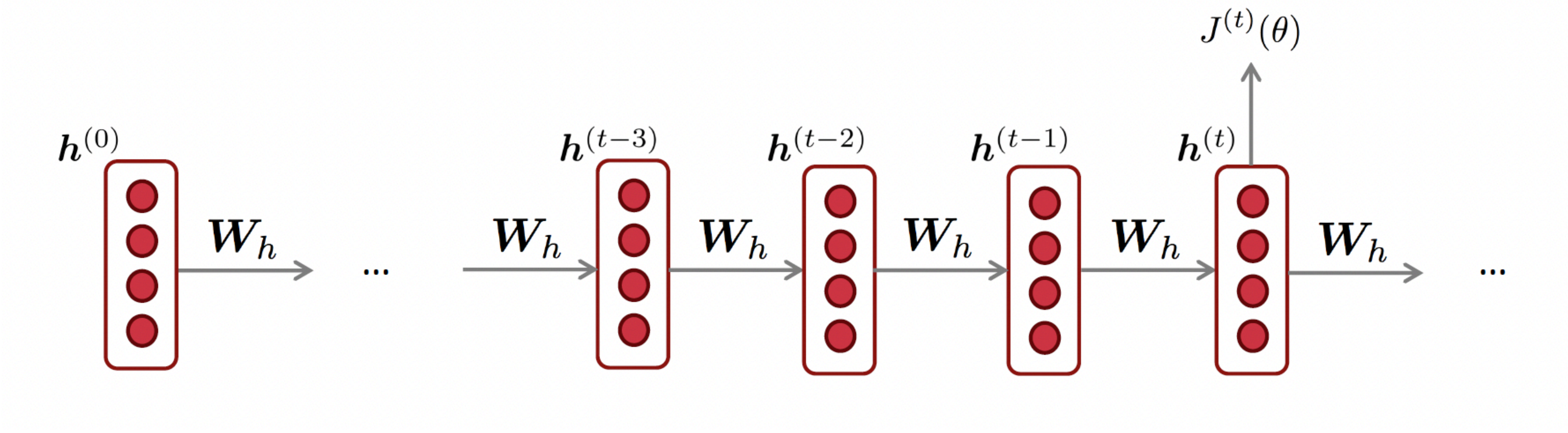
Example: RNN-LM



Teacher Forcing

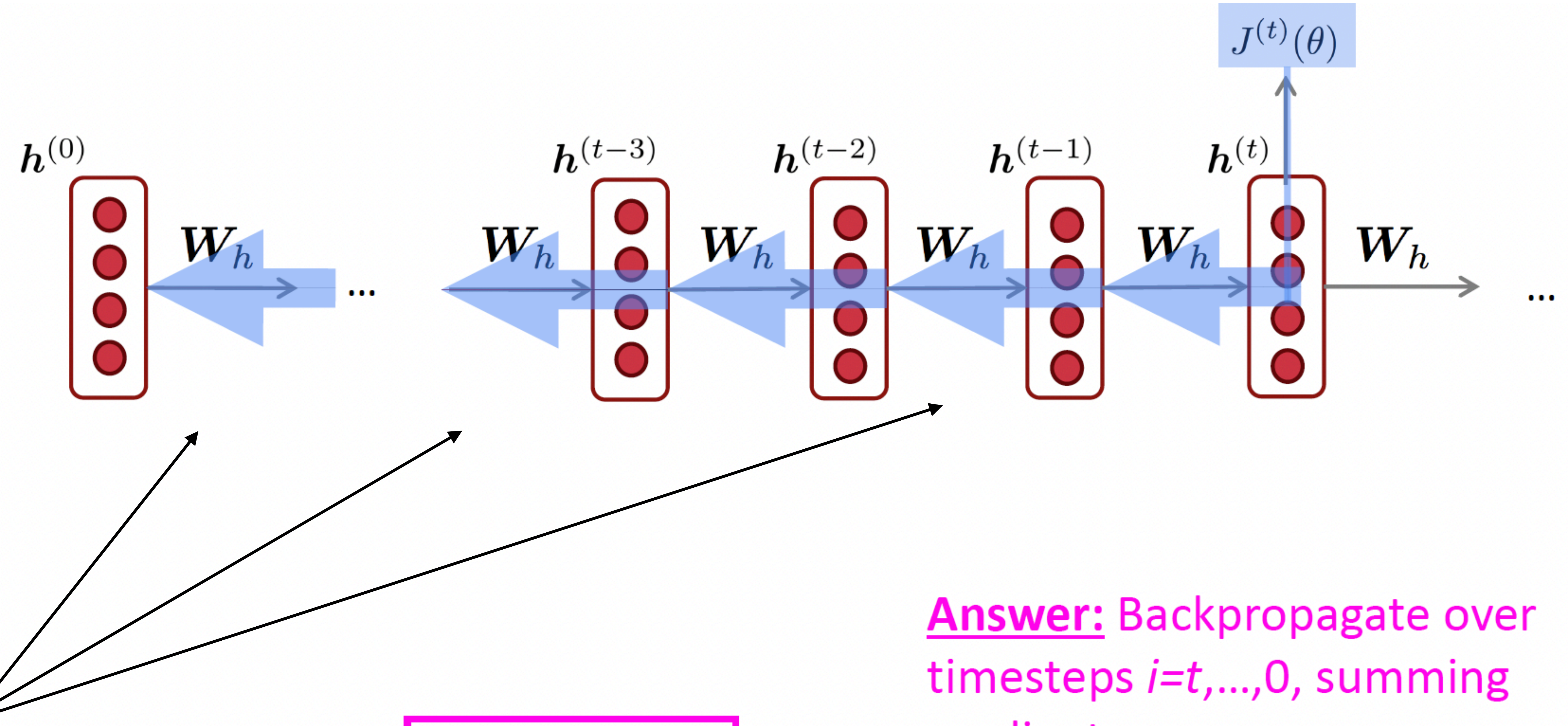
Figures: Manning et al., Stanford cs224n

Back-Propagation for RNNs



Figures: Manning et al., Stanford cs224n

Back-Propagation Through Time (BPTT)



Multiplications along the path, just like in a (very) deep net!

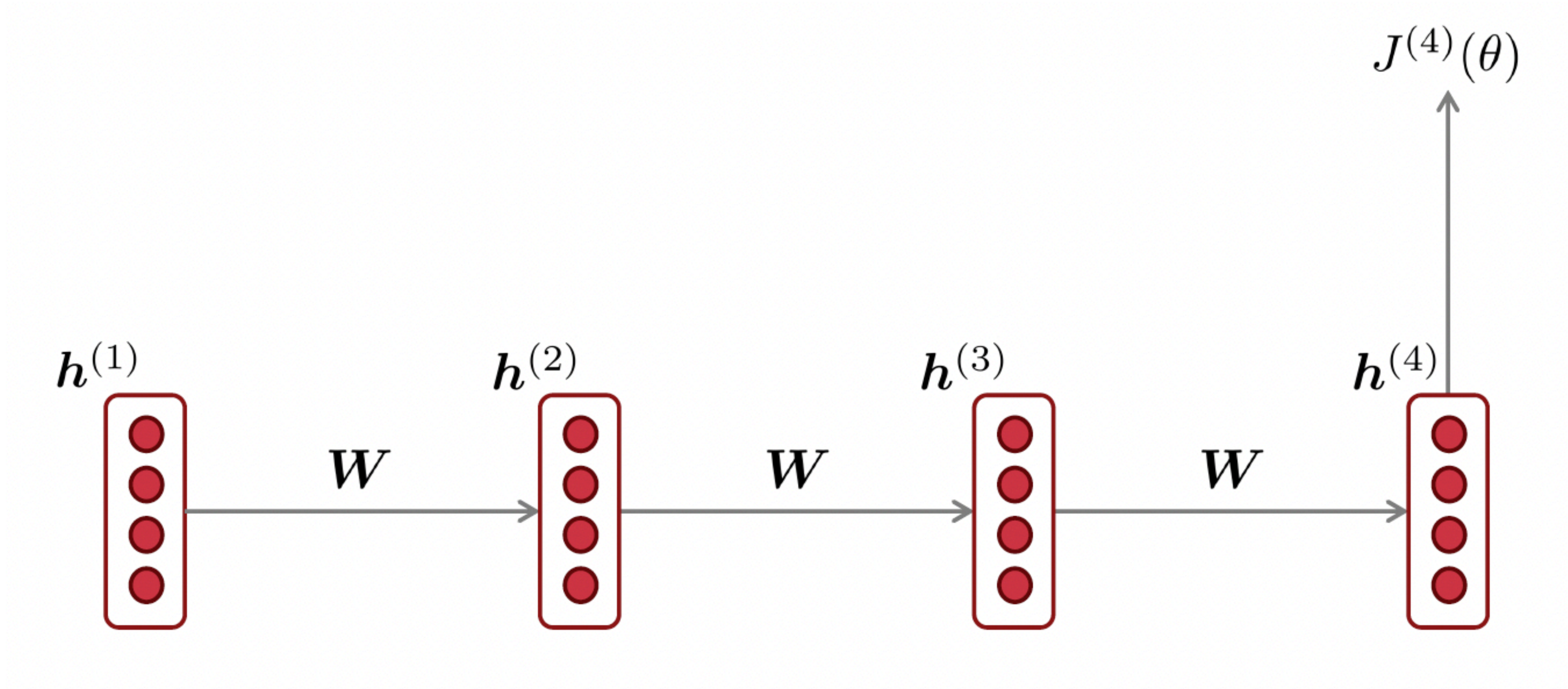
$$\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}} = \boxed{\sum_{i=1}^{t} \left.\frac{\partial J^{(t)}}{\partial \boldsymbol{W_h}}\right|_{(i)}}$$

Question: How do we calculate this?

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

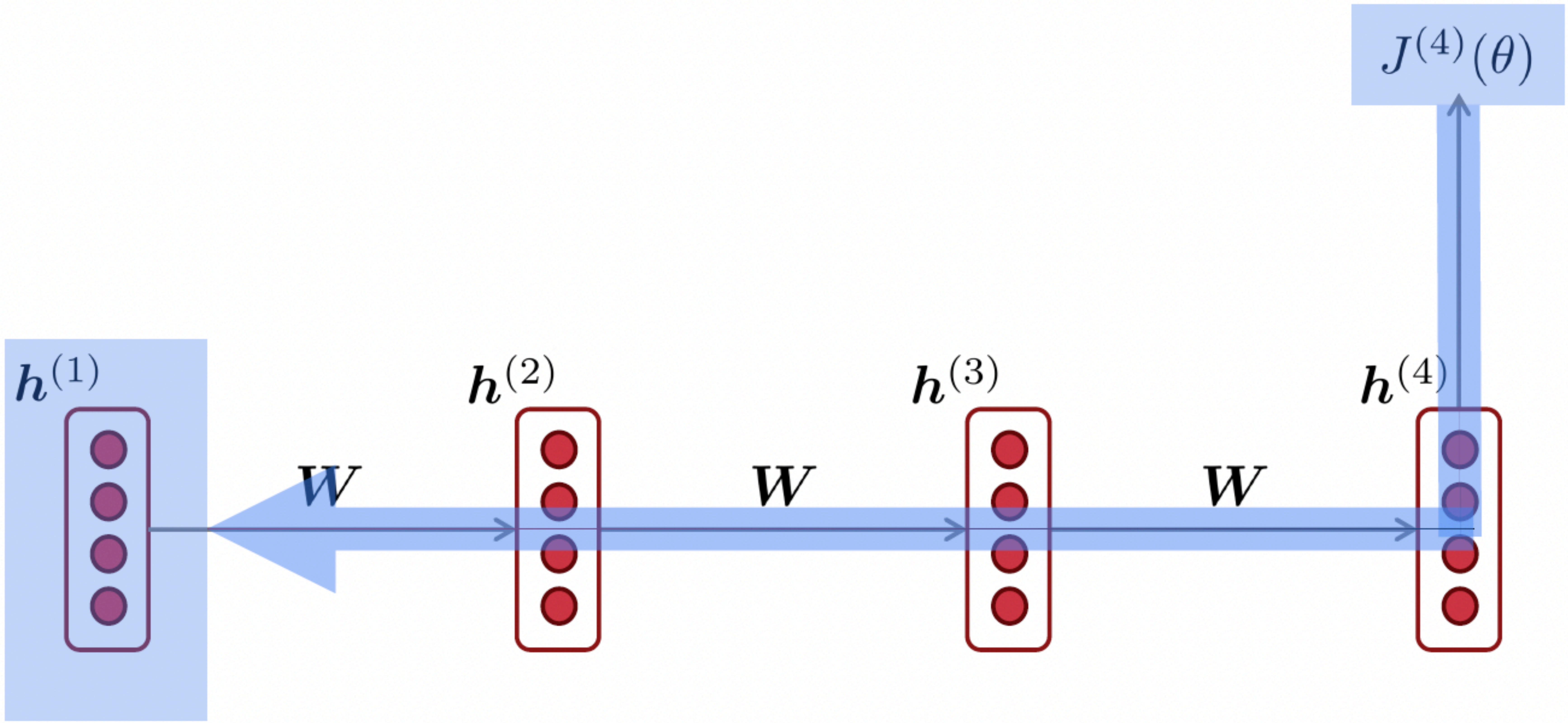
Figures: Manning et al., Stanford cs224n

Gradient Flow Revisited



Figures: Manning et al., Stanford cs224n

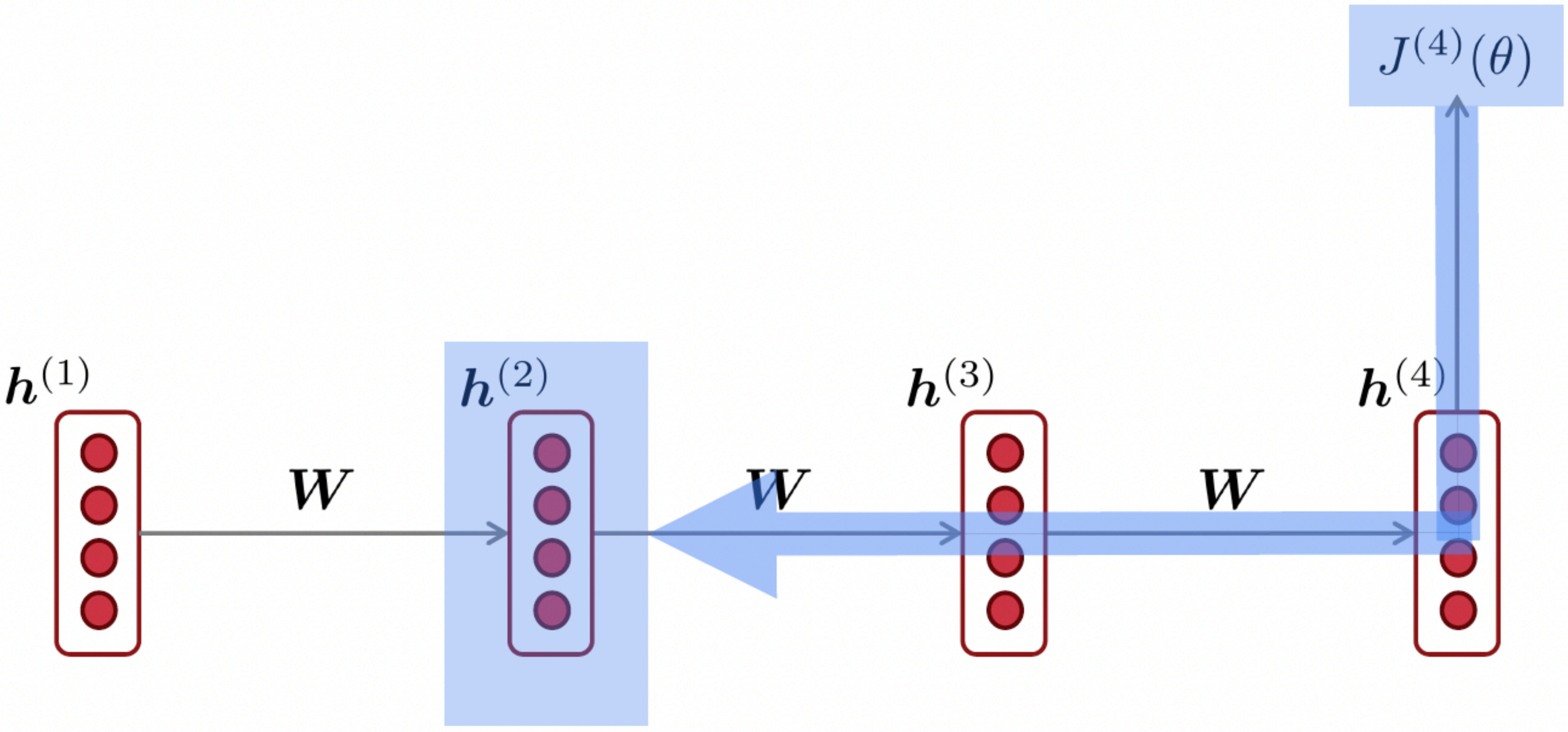
Gradient Flow Revisited



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

Figures: Manning et al., Stanford cs224n

Gradient Flow Revisited

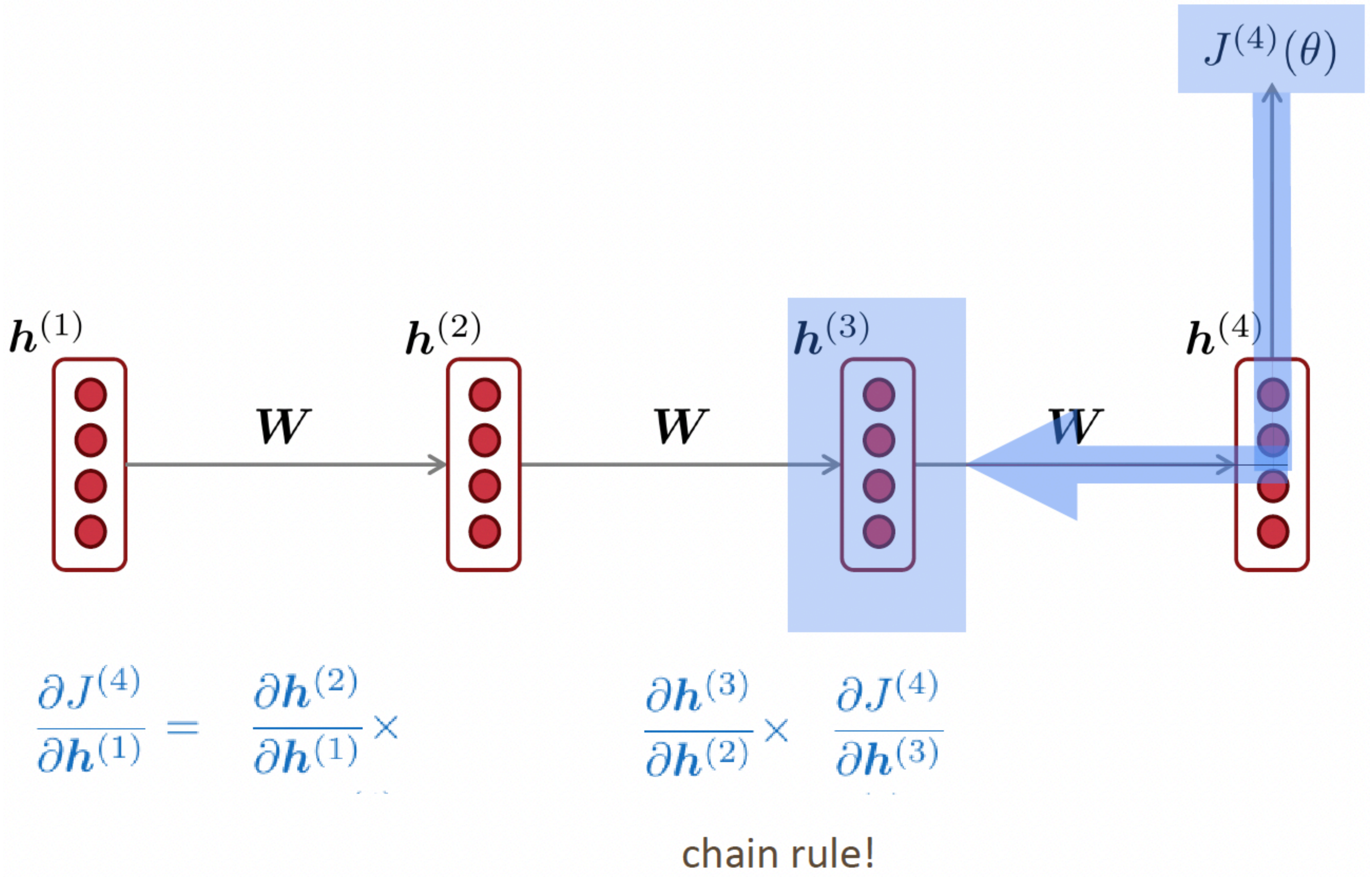


$$\frac{\partial J^{(4)}}{\partial h^{(1)}} = \frac{\partial h^{(2)}}{\partial h^{(1)}} \times \frac{\partial J^{(4)}}{\partial h^{(2)}}$$

chain rule!

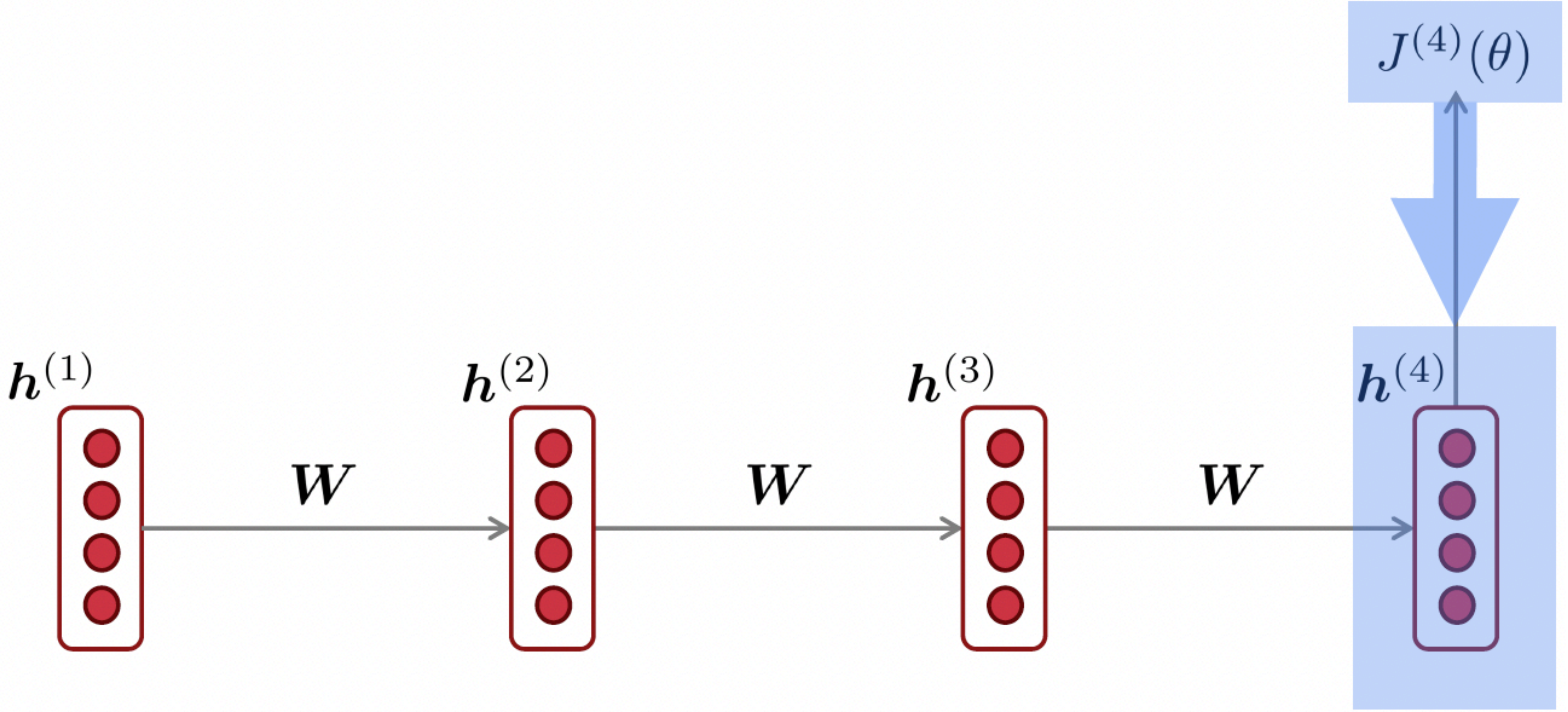
Figures: Manning et al., Stanford cs224n

Gradient Flow Revisited



Figures: Manning et al., Stanford cs224n

Gradient Flow Revisited

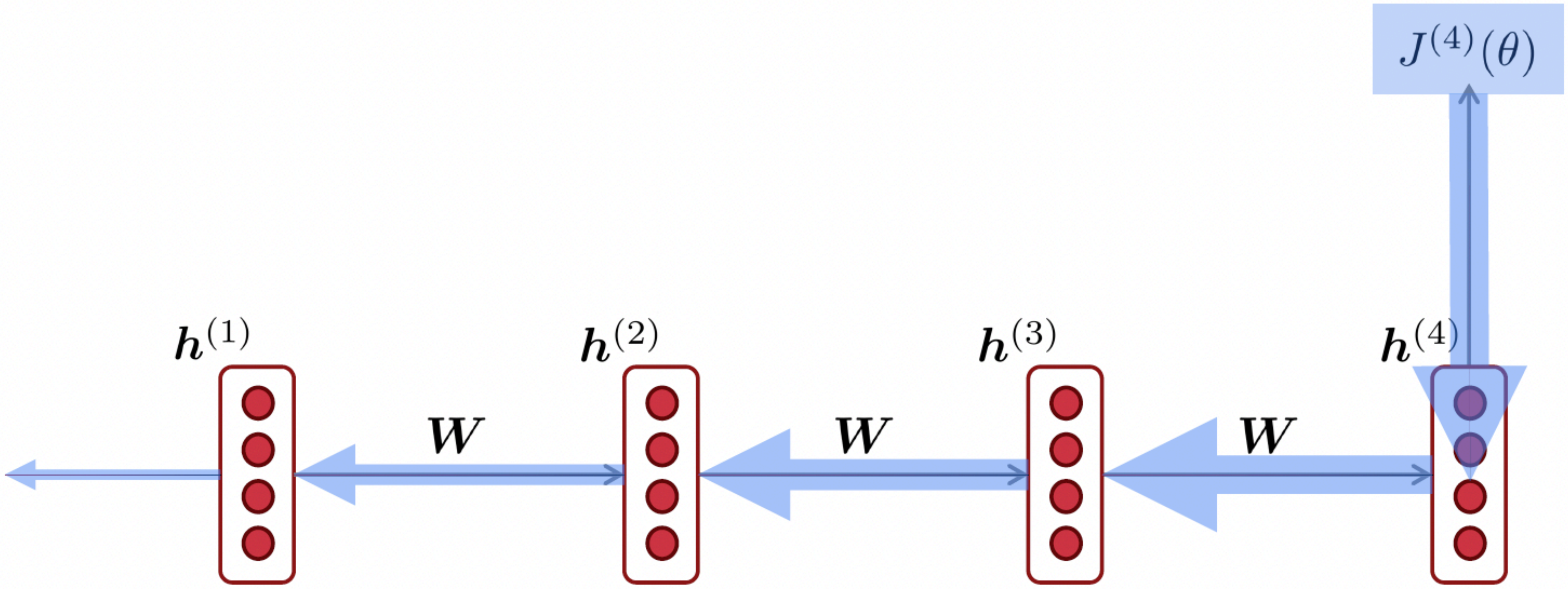


$$\frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(1)}} = \frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}} \times \frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(2)}} \times \frac{\partial \boldsymbol{h}^{(4)}}{\partial \boldsymbol{h}^{(3)}} \times \frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(4)}}$$

chain rule!

Figures: Manning et al., Stanford cs224n

Gradient Flow Revisited



$$\frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(1)}} = \boxed{\frac{\partial \boldsymbol{h}^{(2)}}{\partial \boldsymbol{h}^{(1)}}} \times \boxed{\frac{\partial \boldsymbol{h}^{(3)}}{\partial \boldsymbol{h}^{(2)}}} \times \boxed{\frac{\partial \boldsymbol{h}^{(4)}}{\partial \boldsymbol{h}^{(3)}}} \times \frac{\partial J^{(4)}}{\partial \boldsymbol{h}^{(4)}}$$

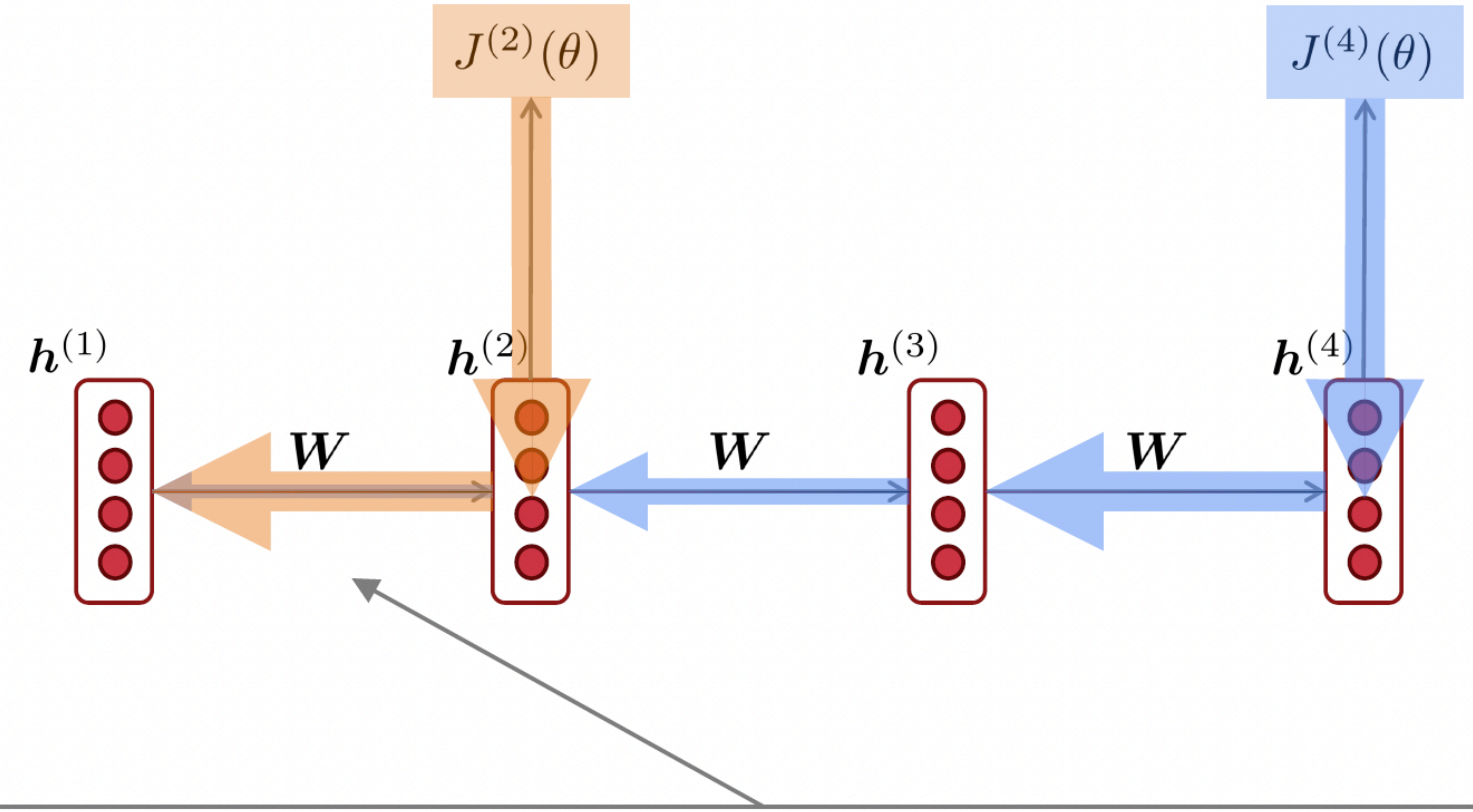
What happens if these are small?

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

Figures: Manning et al., Stanford cs224n

Gradient Flow Revisited

Figures: Manning et al., Stanford cs224n



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?

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

What about the *vanishing* gradient?

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

Long Short-Term Memory RNNs (LSTMs)

- Hochreiter & Schmidhuber, 1997 (and Gers et al., 2000)
- At each time t, there is a hidden state $\boldsymbol{h}^{(t)}$ and cell state $\boldsymbol{c}^{(t)}$
  - Both are vectors length n
  - Cell stores long-term information in $\boldsymbol{c}$
  - 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

Sigmoid function: all gate values are between 0 and 1

$$\boldsymbol{f}^{(t)} = \boxed{\sigma}\left(\boldsymbol{W}_f \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_f \boldsymbol{x}^{(t)} + \boldsymbol{b}_f\right)$$

$$\boldsymbol{i}^{(t)} = \boxed{\sigma}\left(\boldsymbol{W}_i \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_i \boldsymbol{x}^{(t)} + \boldsymbol{b}_i\right)$$

$$\boldsymbol{o}^{(t)} = \boxed{\sigma}\left(\boldsymbol{W}_o \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_o \boldsymbol{x}^{(t)} + \boldsymbol{b}_o\right)$$

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

$$\tilde{\boldsymbol{c}}^{(t)} = \tanh\left(\boldsymbol{W}_c \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_c \boldsymbol{x}^{(t)} + \boldsymbol{b}_c\right)$$

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

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

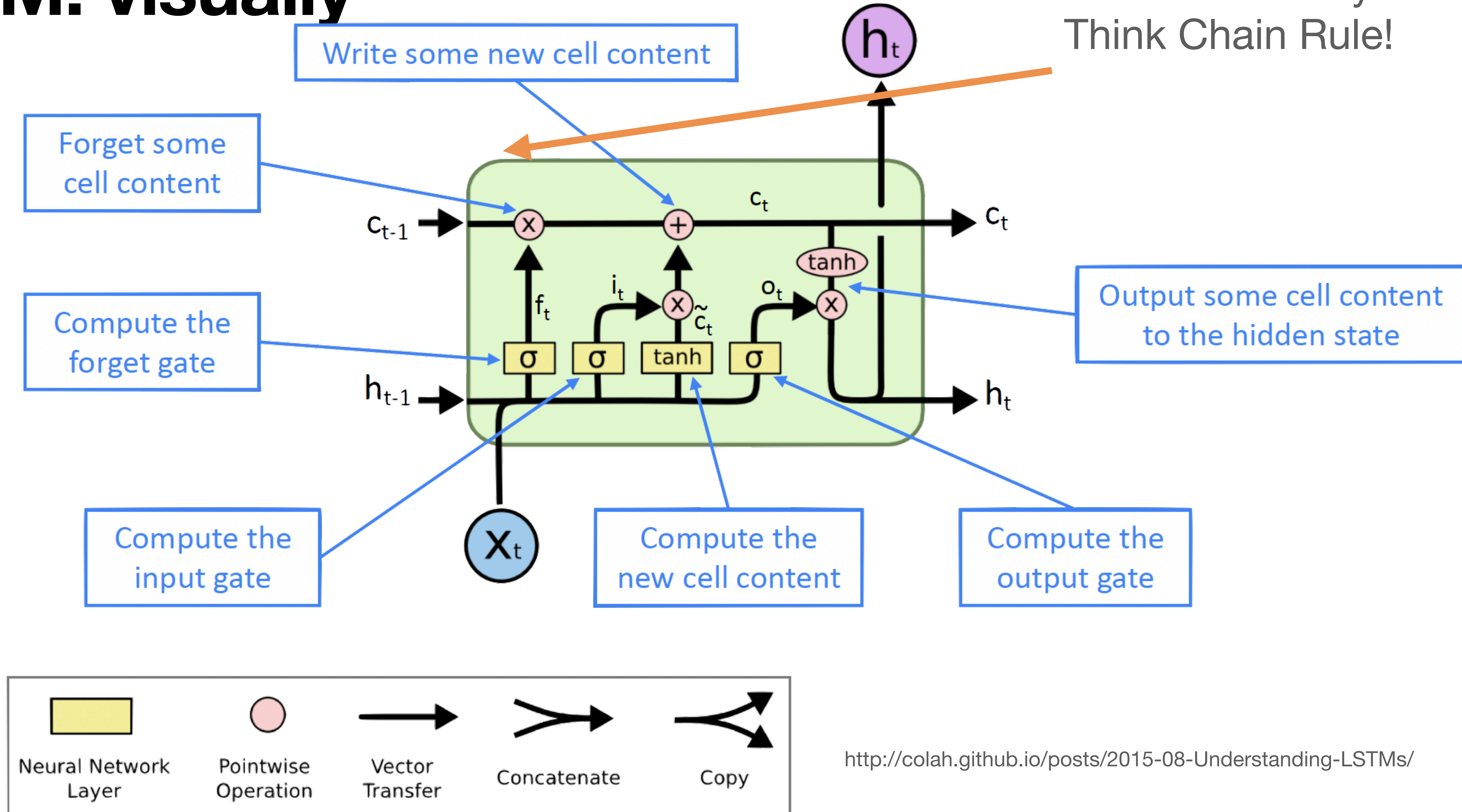
All these are vectors of same length n

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

Figures: Manning et al., Stanford cs224n

LSTM: visually

The + is the key!
Think Chain Rule!



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

Why does it work (or at least help)?

- The cell state is based on (scalar) multiplication and addition
  $\rightarrow$ 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

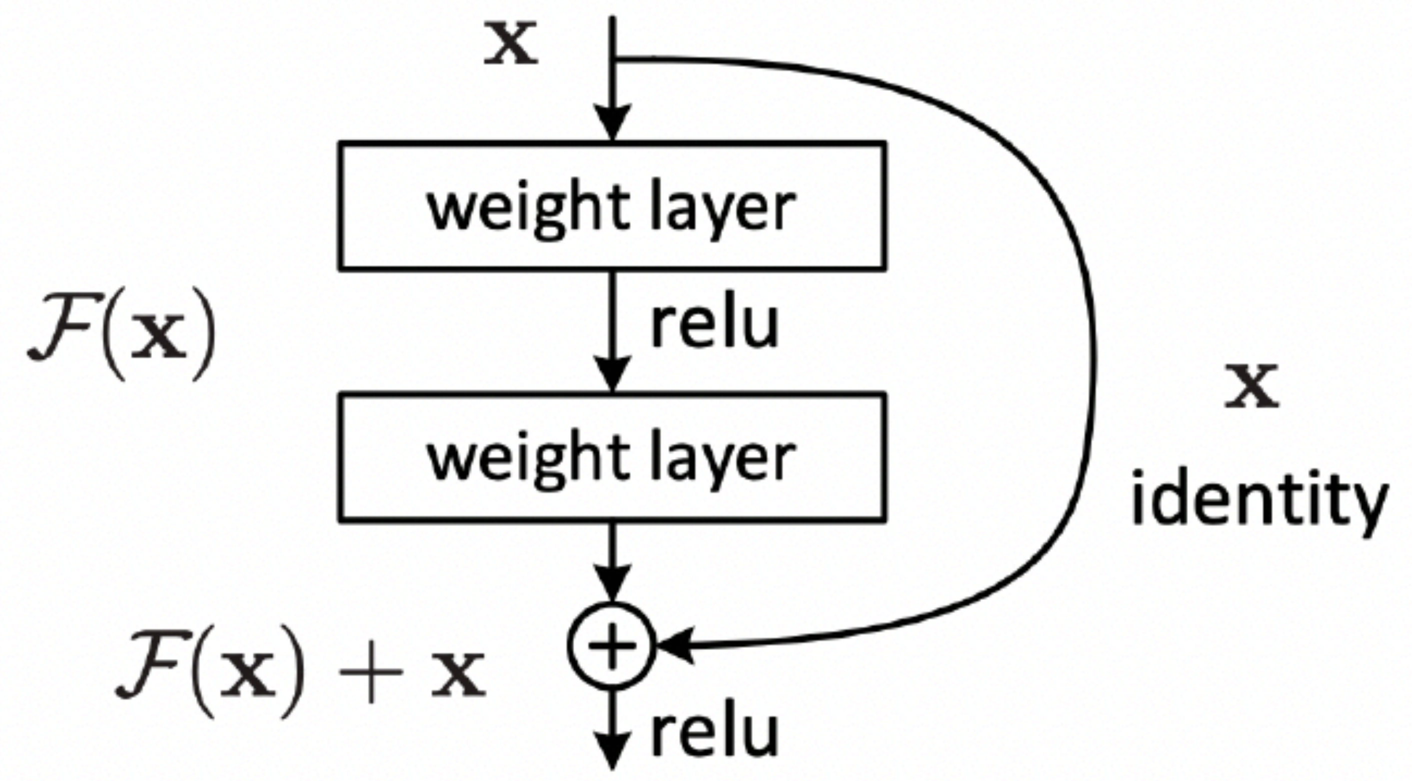


Figure 2. Residual learning: a building block.

"Deep Residual Learning for Image Recognition",
He et al, 2015. https://arxiv.org/pdf/1512.03385.pdf

”Learning Long-Term Dependencies with Gradient Descent is Difficult", Bengio et al. 1994, http://ai.dinfo.unifi.it/paolo//ps/tnn-94-gradient.pdf

Simplified: Gated Recurrent Unit (GRU)

- similar to an LSTM
- has a forget gate
- fewer parameters than an LSTM, because it has no output gate
- similar performance on certain tasks (polyphonic music modeling, speech signal modeling, natural language processing)
- sometimes better performance on (certain) smaller and less frequent datasets

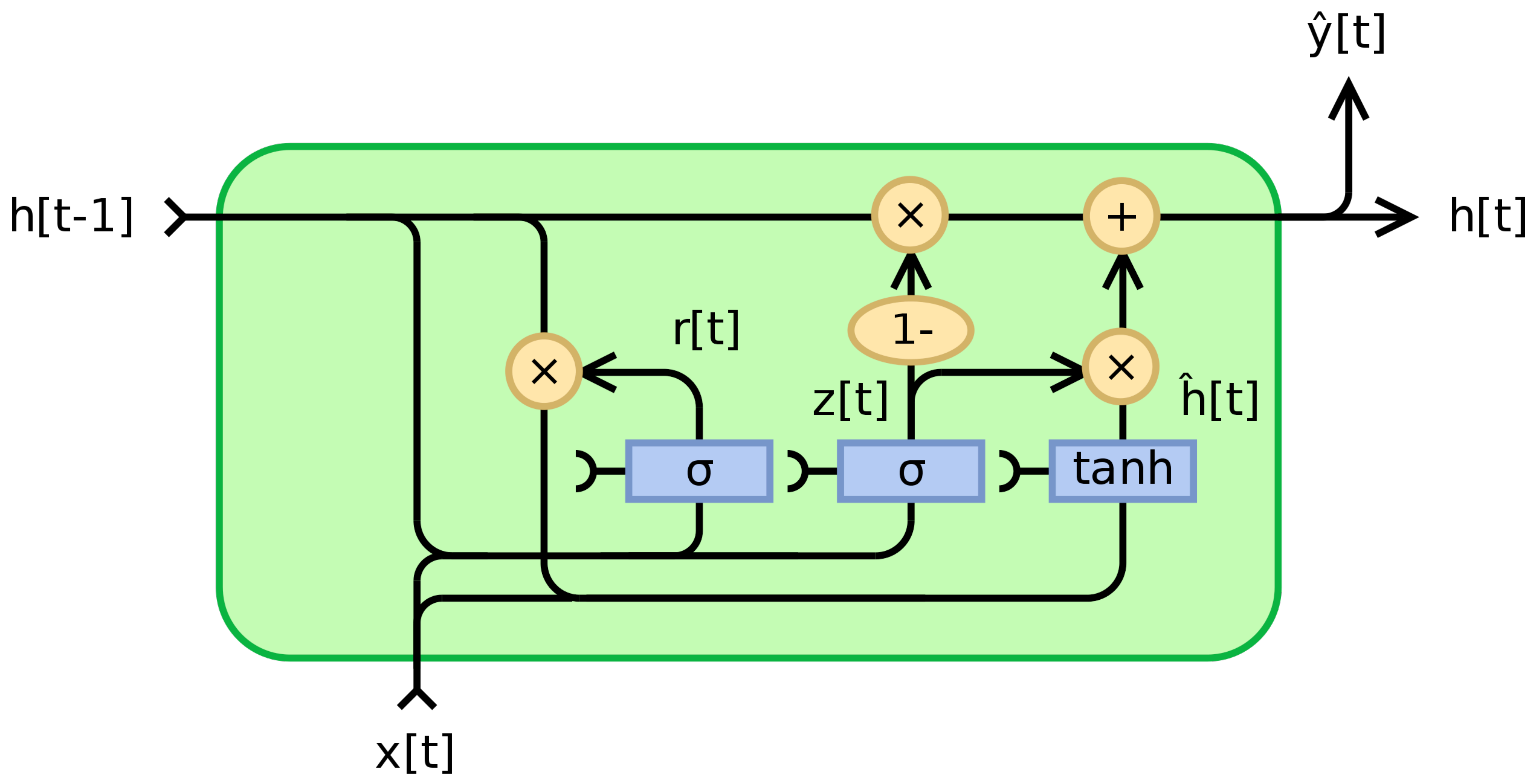


image: https://en.wikipedia.org/wiki/Gated_recurrent_unit

GRU visualised

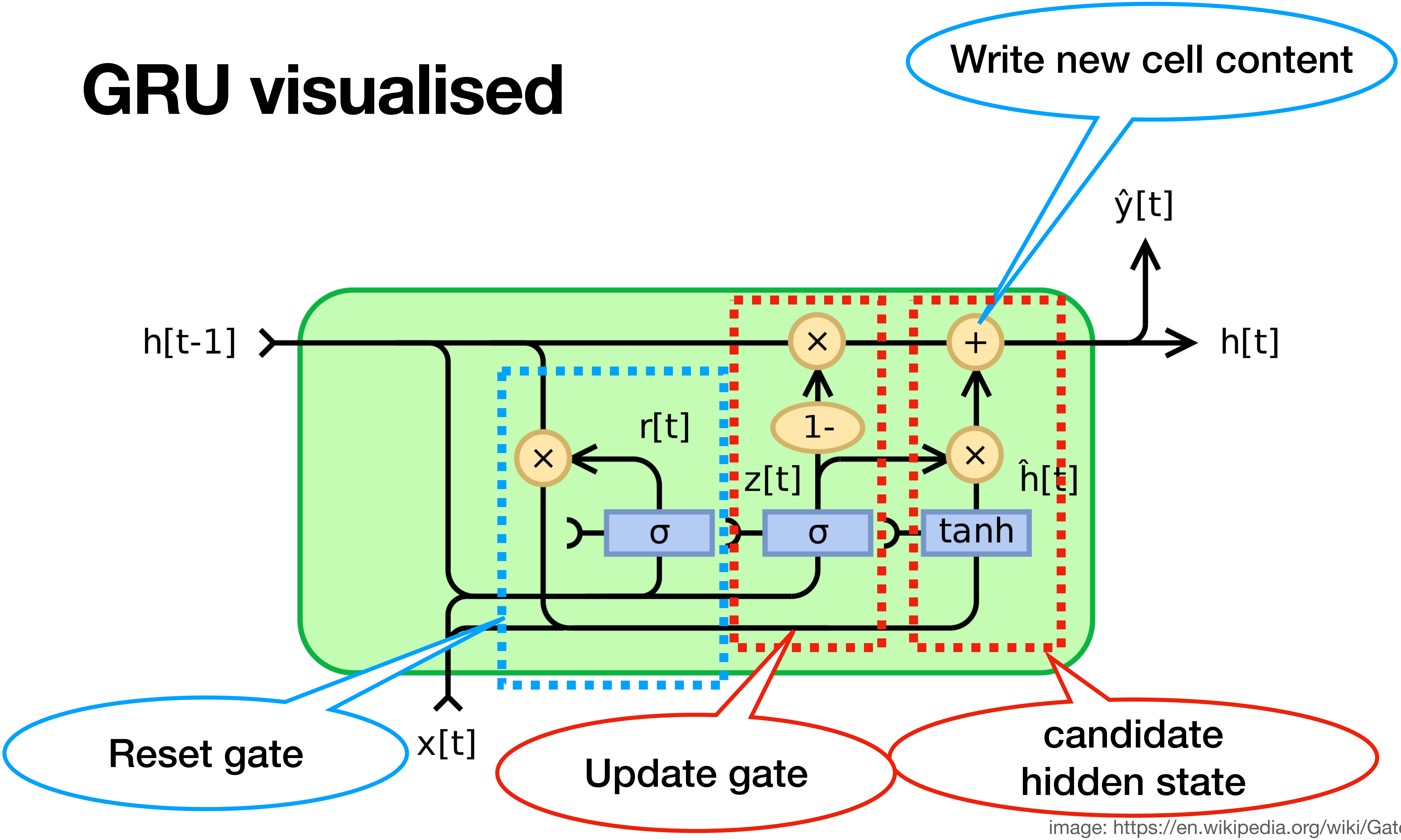


image: https://en.wikipedia.org/wiki/Gated_recurrent_unit

GRU

W_z for x_t,
U_z weights for h_t-1

$$z_t = \boxed{\sigma_g(W_z x_t + U_z h_{t-1} + b_z)}$$
$$r_t = \sigma_g(W_r x_t + U_r h_{t-1} + b_r)$$
$$\hat{h}_t = \phi_h(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$
$$h_t = z_t \odot \hat{h}_t + (1 - z_t) \odot h_{t-1}$$

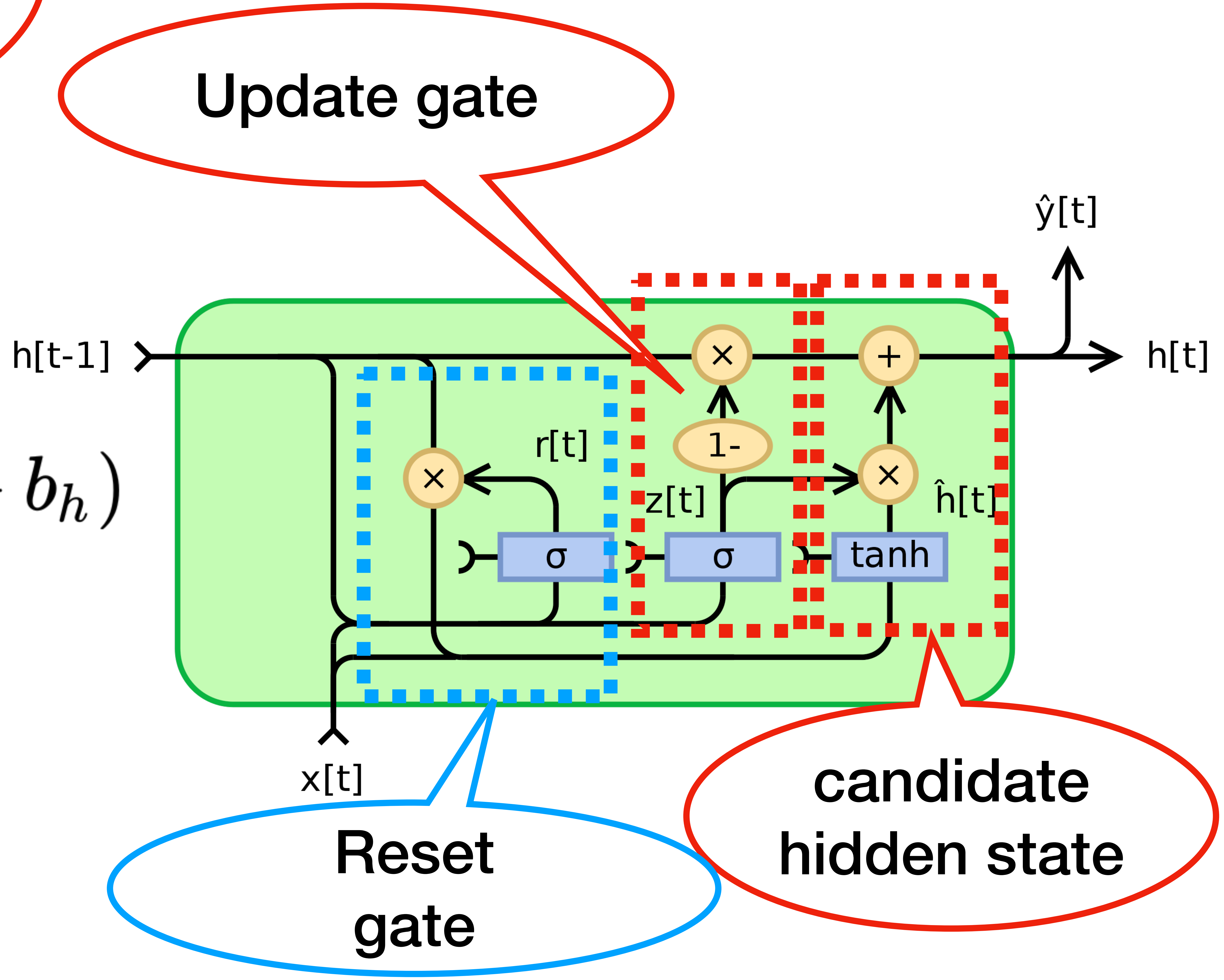


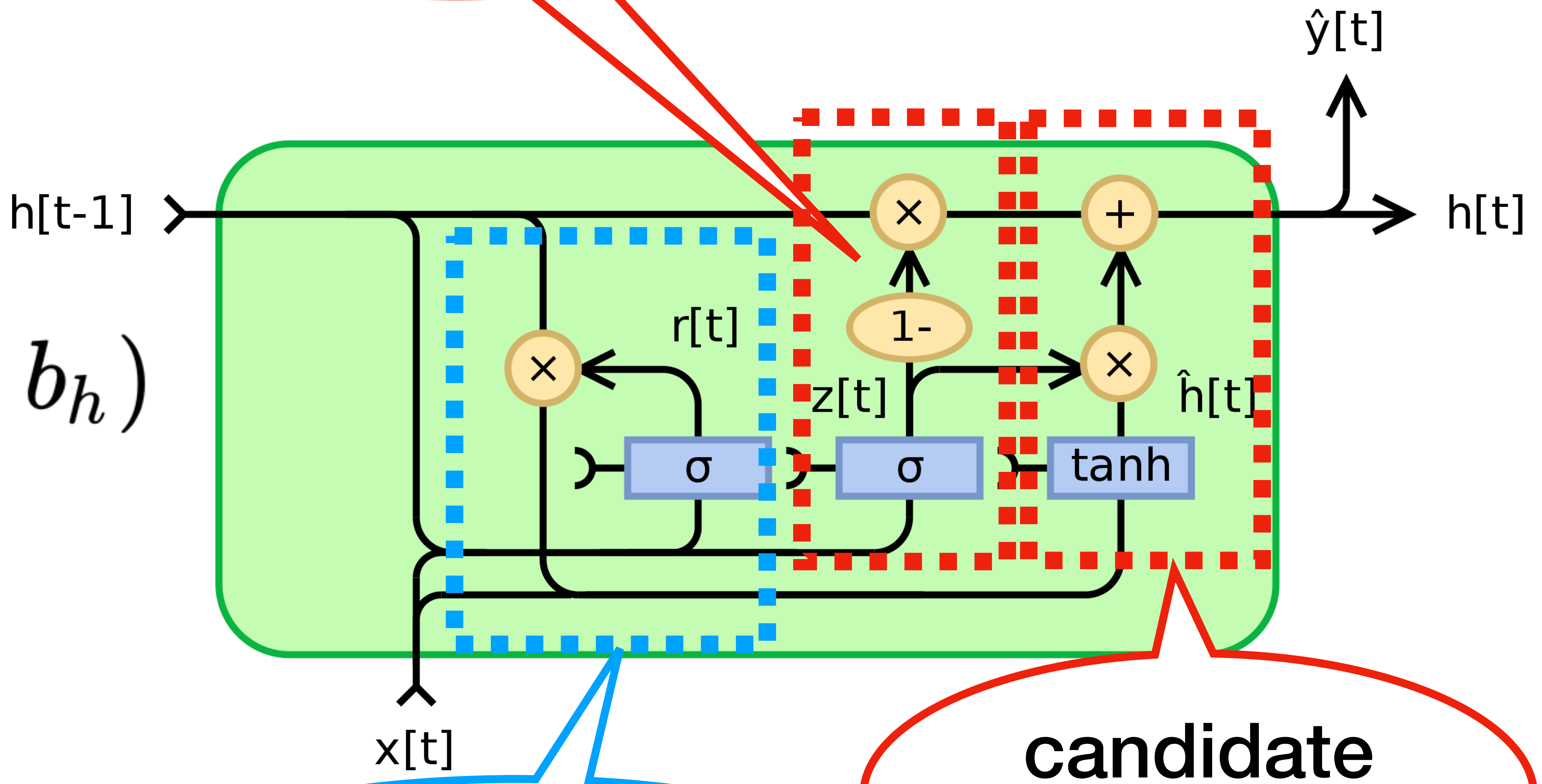
image: https://en.wikipedia.org/wiki/Gated_recurrent_unit

GRU

W_r for x_t,
U_r weights for h_t-1

$$z_t = \sigma_g(W_z x_t + U_z h_{t-1} + b_z)$$
$$r_t = \boxed{\sigma_g(W_r x_t + U_r h_{t-1} + b_r)}$$
$$\hat{h}_t = \phi_h(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$
$$h_t = z_t \odot \hat{h}_t + (1 - z_t) \odot h_{t-1}$$

Update gate



Reset gate

candidate hidden state

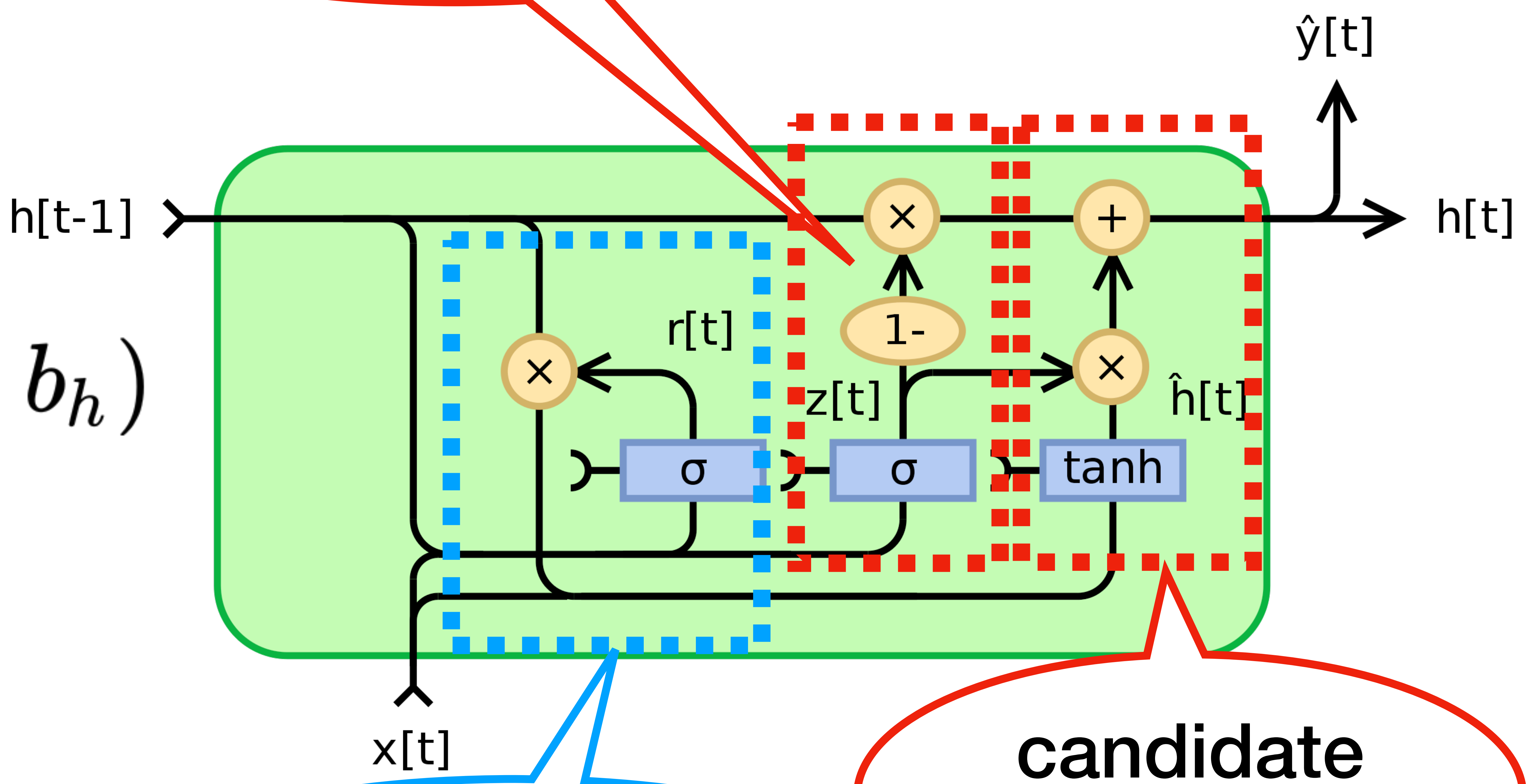
image: https://en.wikipedia.org/wiki/Gated_recurrent_unit

GRU

Update and forget gate process the same inputs (x_t, h_t-1)

$$z_t = \sigma_g(W_z x_t + U_z h_{t-1} + b_z)$$
$$r_t = \sigma_g(W_r x_t + U_r h_{t-1} + b_r)$$
$$\hat{h}_t = \phi_h(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$
$$h_t = z_t \odot \hat{h}_t + (1 - z_t) \odot h_{t-1}$$

Update gate



Reset gate

candidate hidden state

image: https://en.wikipedia.org/wiki/Gated_recurrent_unit

GRU

$$z_t = \sigma_g(W_z x_t + U_z h_{t-1} + b_z)$$

$$r_t = \sigma_g(W_r x_t + U_r h_{t-1} + b_r)$$

$$\hat{h}_t = \boxed{\phi_h(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)}$$

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

candidate hidden state

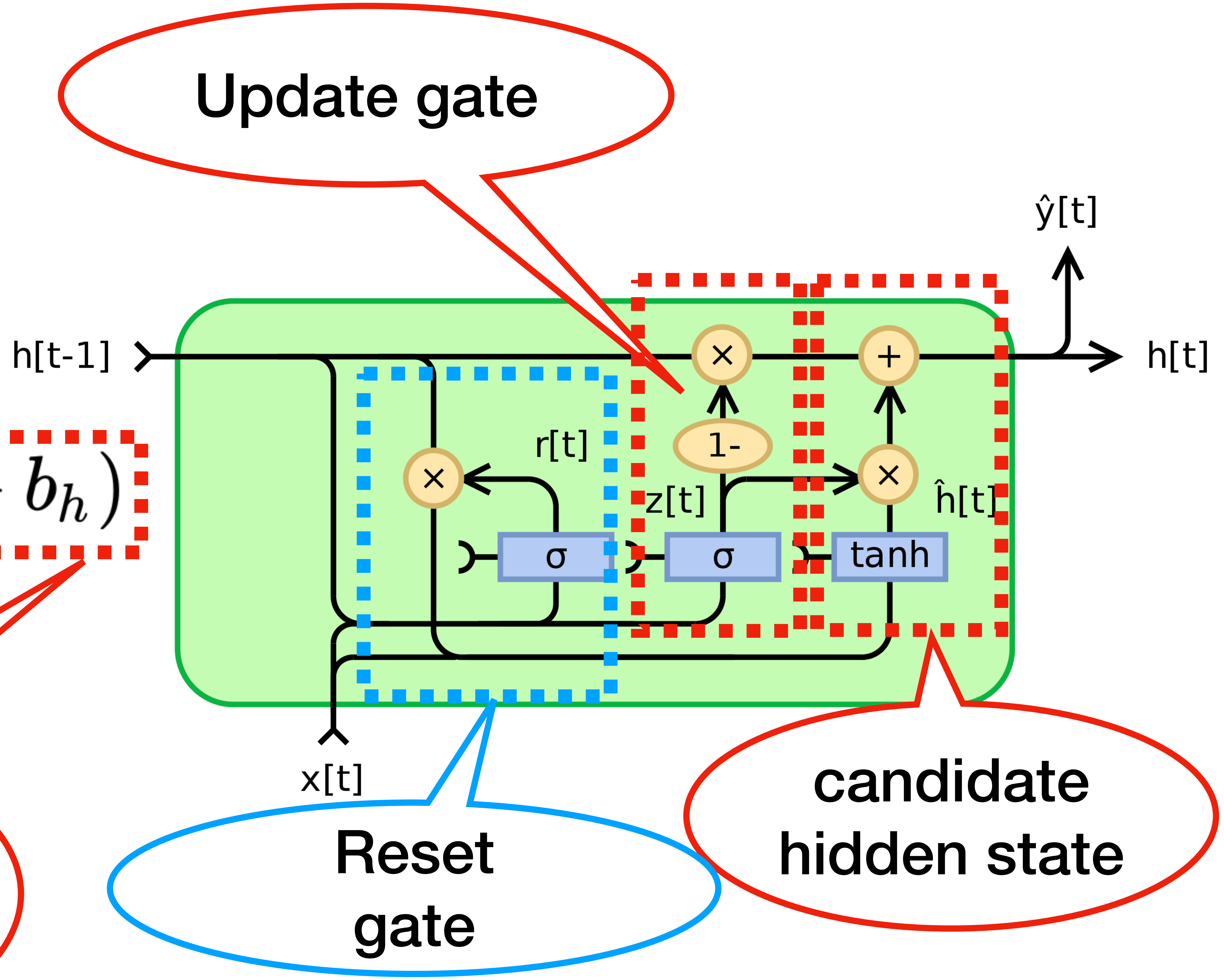


image: https://en.wikipedia.org/wiki/Gated_recurrent_unit

GRU

$$z_t = \sigma_g(W_z x_t + U_z h_{t-1} + b_z)$$
$$r_t = \sigma_g(W_r x_t + U_r h_{t-1} + b_r)$$
$$\hat{h}_t = \phi_h(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$
$$h_t = \boxed{z_t \odot \hat{h}_t + (1 - z_t) \odot h_{t-1}}$$

hidden state at time t

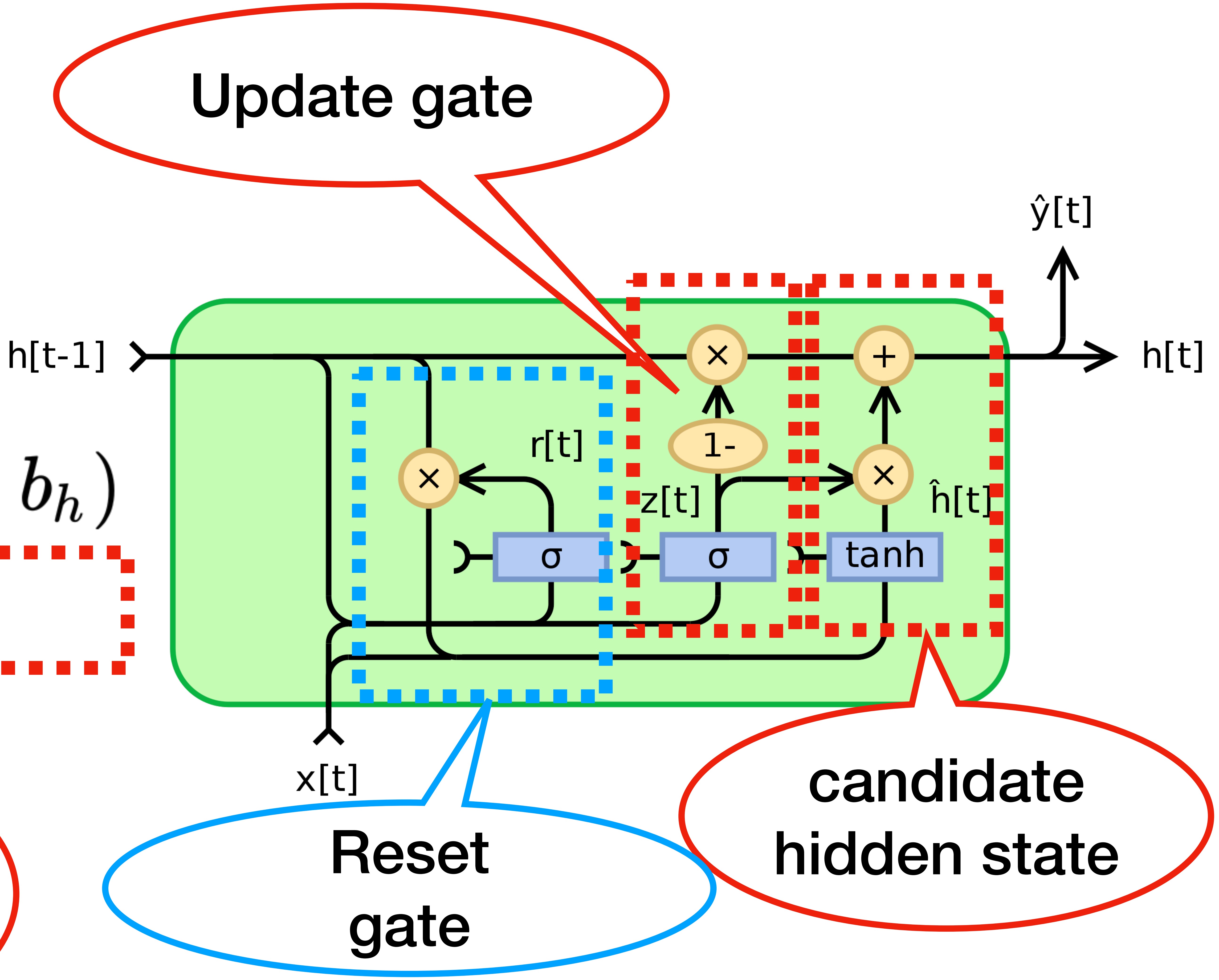


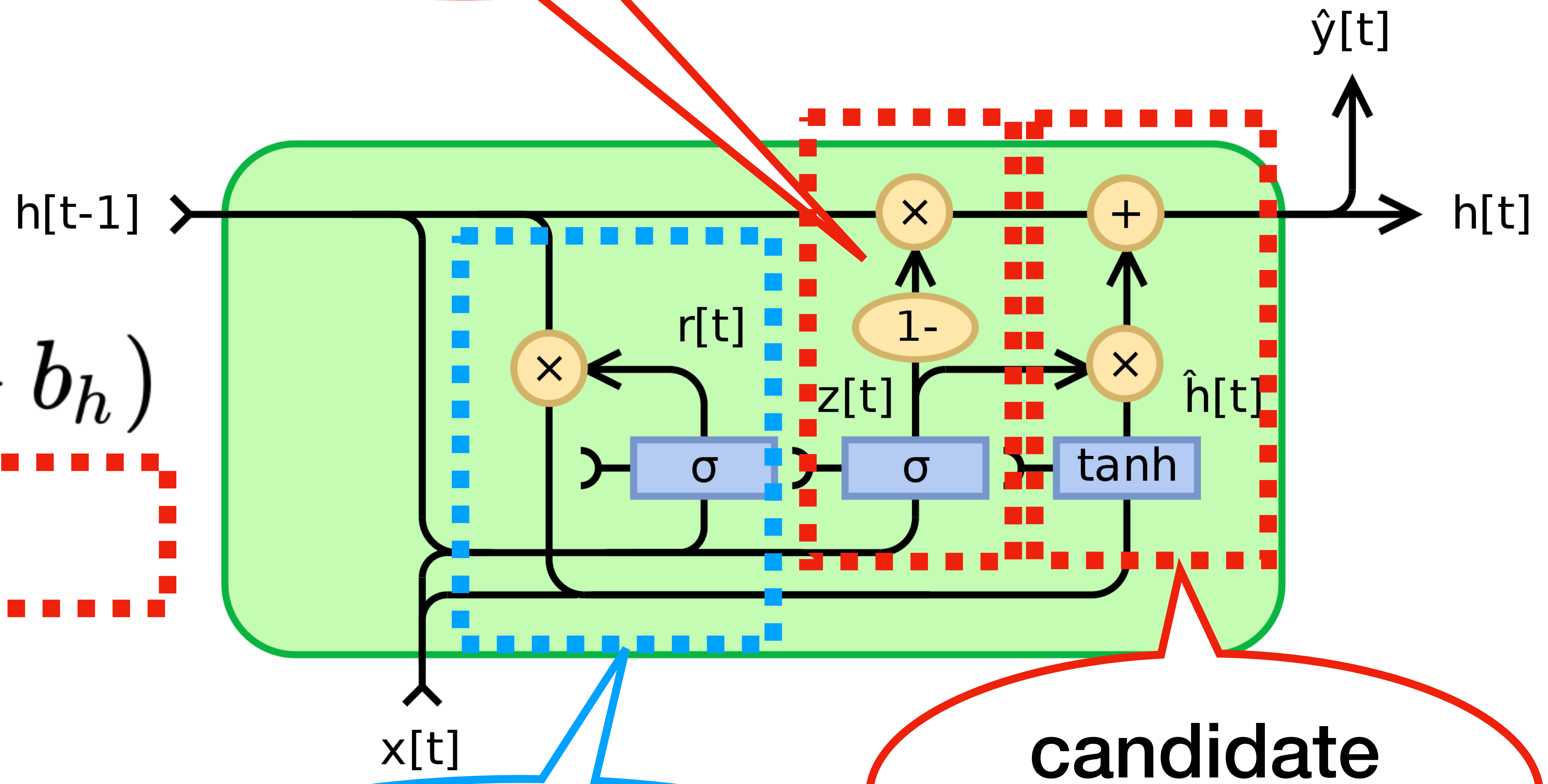
image: https://en.wikipedia.org/wiki/Gated_recurrent_unit

GRU

$$z_t = \sigma_g(W_z x_t + U_z h_{t-1} + b_z)$$
$$r_t = \sigma_g(W_r x_t + U_r h_{t-1} + b_r)$$
$$\hat{h}_t = \phi_h(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$
$$h_t = \boxed{z_t \odot \hat{h}_t + (1 - z_t) \odot h_{t-1}}$$

hidden state at time t

Update gate



Reset gate

candidate hidden state

image: https://en.wikipedia.org/wiki/Gated_recurrent_unit

GRU flavours

- simplified / slightly alternated gating mechanisms
  - Type 1, each gate depends only on the previous hidden state and the bias

$$z_t = \sigma_g(U_z h_{t-1} + b_z)$$
$$r_t = \sigma_g(U_r h_{t-1} + b_r)$$

  - Type 2, each gate depends only on the previous hidden state.

$$z_t = \sigma_g(U_z h_{t-1})$$
$$r_t = \sigma_g(U_r h_{t-1})$$

  - Type 3, each gate is computed using only the bias.

$$z_t = \sigma_g(b_z)$$
$$r_t = \sigma_g(b_r)$$

image: https://en.wikipedia.org/wiki/Gated_recurrent_unit

GRU flavours

- **Minimal gated unit**
  - similar to the fully gated unit
  - update and reset gate vector is merged into a forget gate.

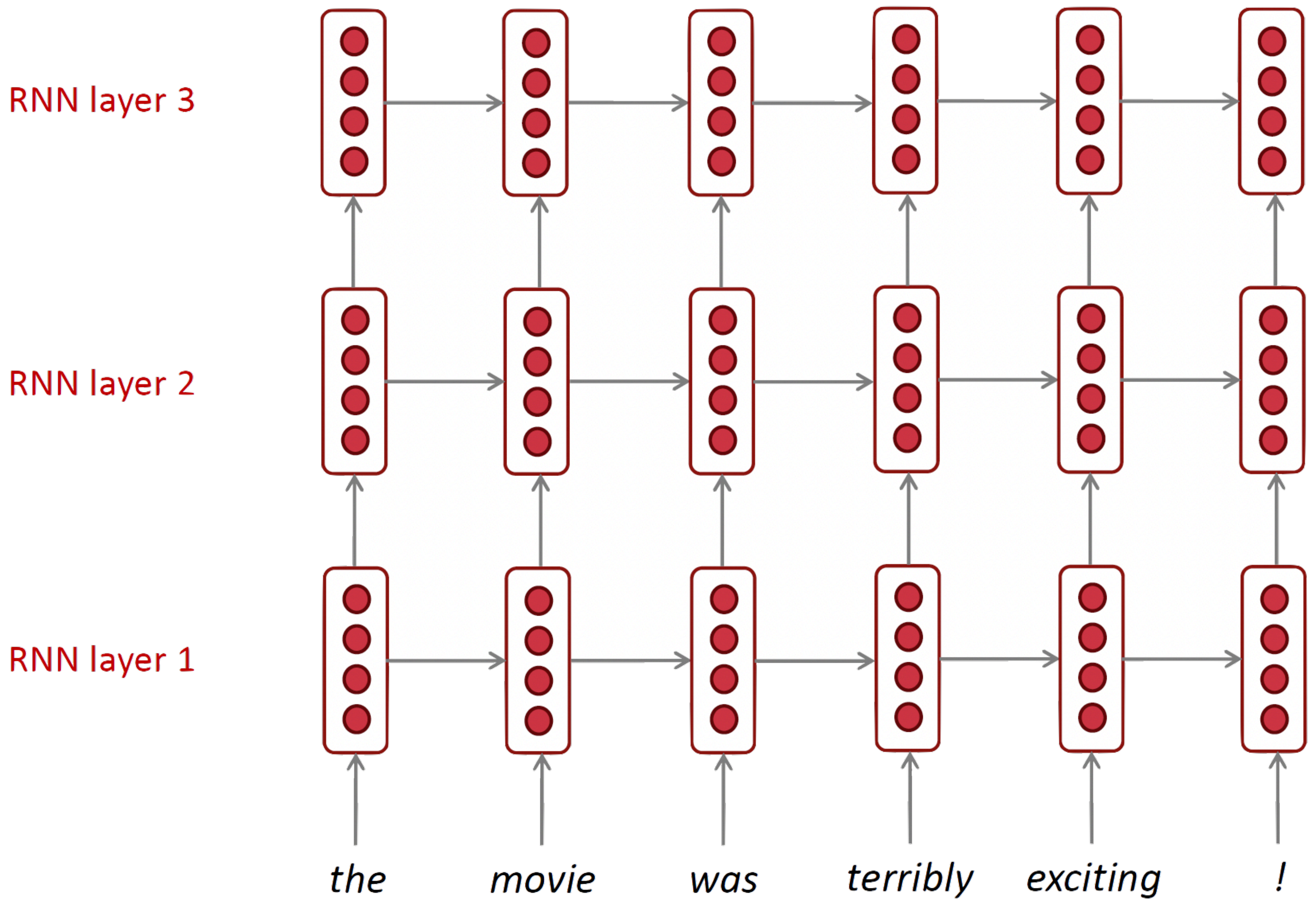
$$f_t = \sigma_g(W_f x_t + U_f h_{t-1} + b_f)$$
$$\hat{h}_t = \phi_h(W_h x_t + U_h(f_t \odot h_{t-1}) + b_h)$$
$$h_t = (1 - f_t) \odot h_{t-1} + f_t \odot \hat{h}_t$$

image: https://en.wikipedia.org/wiki/Gated_recurrent_unit

Extensions: Multi-layer (stacked) LSTM/GRU/RNN

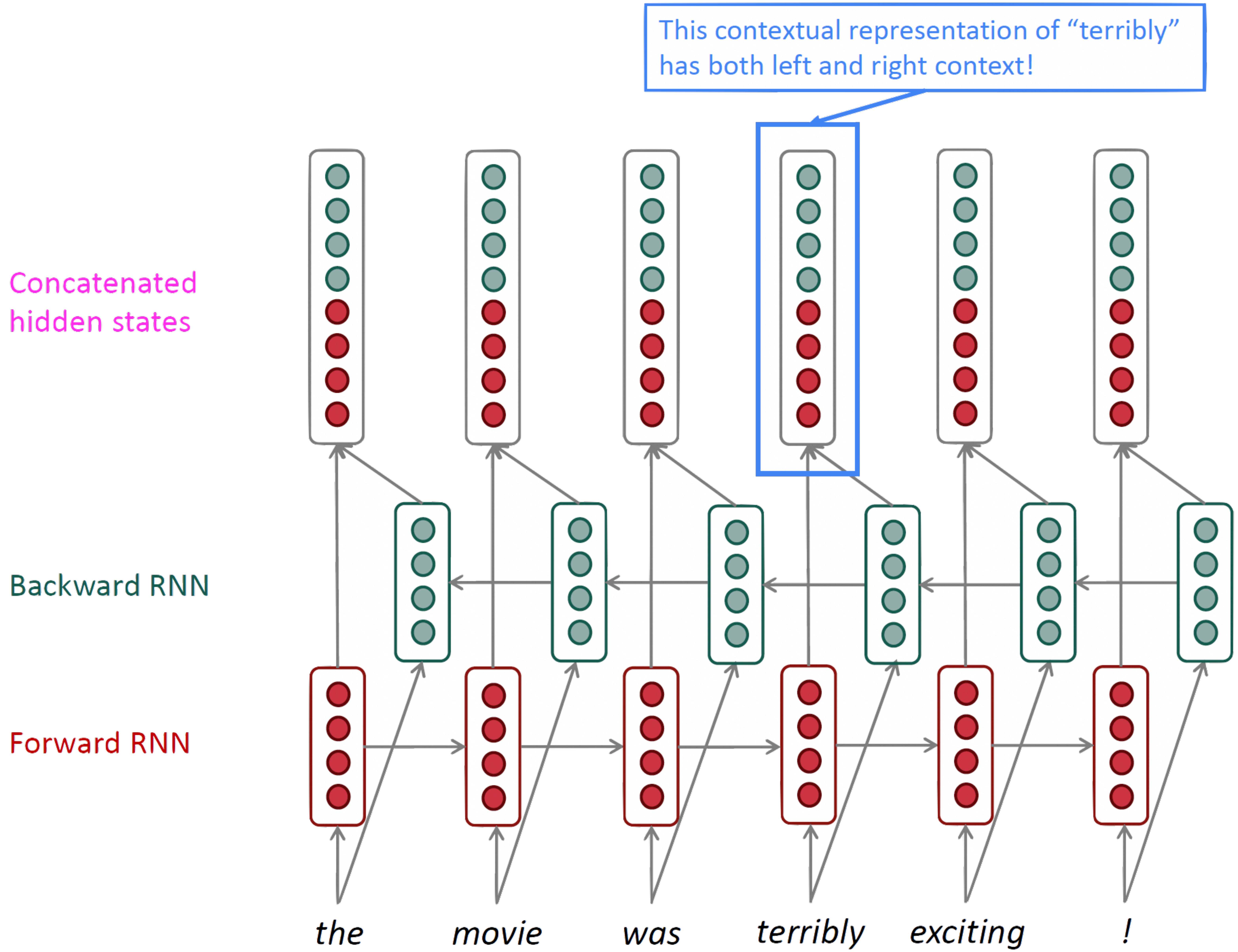
The hidden states from RNN layer *i* are the inputs to RNN layer *i*+1

- 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

Extensions: Bi-directional LSTM/GRU/RNN



$$\overrightarrow{\boldsymbol{h}}^{(t)} = \mathrm{RNN}_{\mathrm{FW}}(\overrightarrow{\boldsymbol{h}}^{(t-1)}, \boldsymbol{x}^{(t)})$$

$$\overleftarrow{\boldsymbol{h}}^{(t)} = \mathrm{RNN}_{\mathrm{BW}}(\overleftarrow{\boldsymbol{h}}^{(t+1)}, \boldsymbol{x}^{(t)})$$

$$\boldsymbol{h}^{(t)} = [\overrightarrow{\boldsymbol{h}}^{(t)}; \overleftarrow{\boldsymbol{h}}^{(t)}]$$

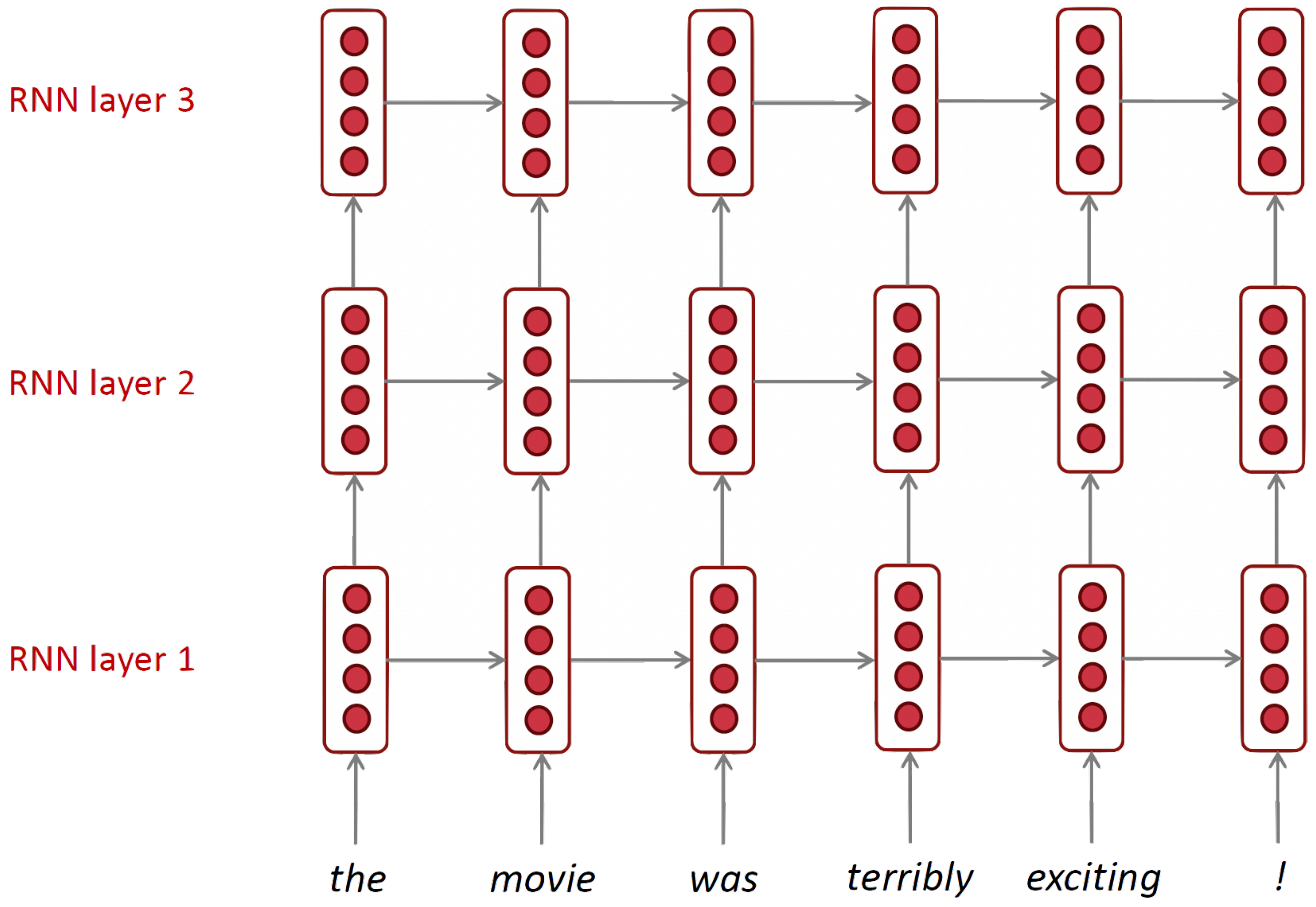
Requires access to full input sequence!

Figures: Manning et al., Stanford cs224n

Extensions: Multi-layer (stacked) LSTM/GRU/RNN

The hidden states from RNN layer *i* are the inputs to RNN layer *i*+1

- 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
- GRUs have less parameters than LSTMS but have similar performance
- Use bi-directionality if you have access to the full sequence
- RNN/LSTM/GRU 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 $M \neq N$?