Lecture 9: Recurrent Neural Nets

Instructor: Swabha Swayamdipta

USC CSCI 499 LMs in NLP

Feb 14, 2024 Spring

Slides mostly adapted from Dan Jurafsky, and Chris Manning

Lecture Outline

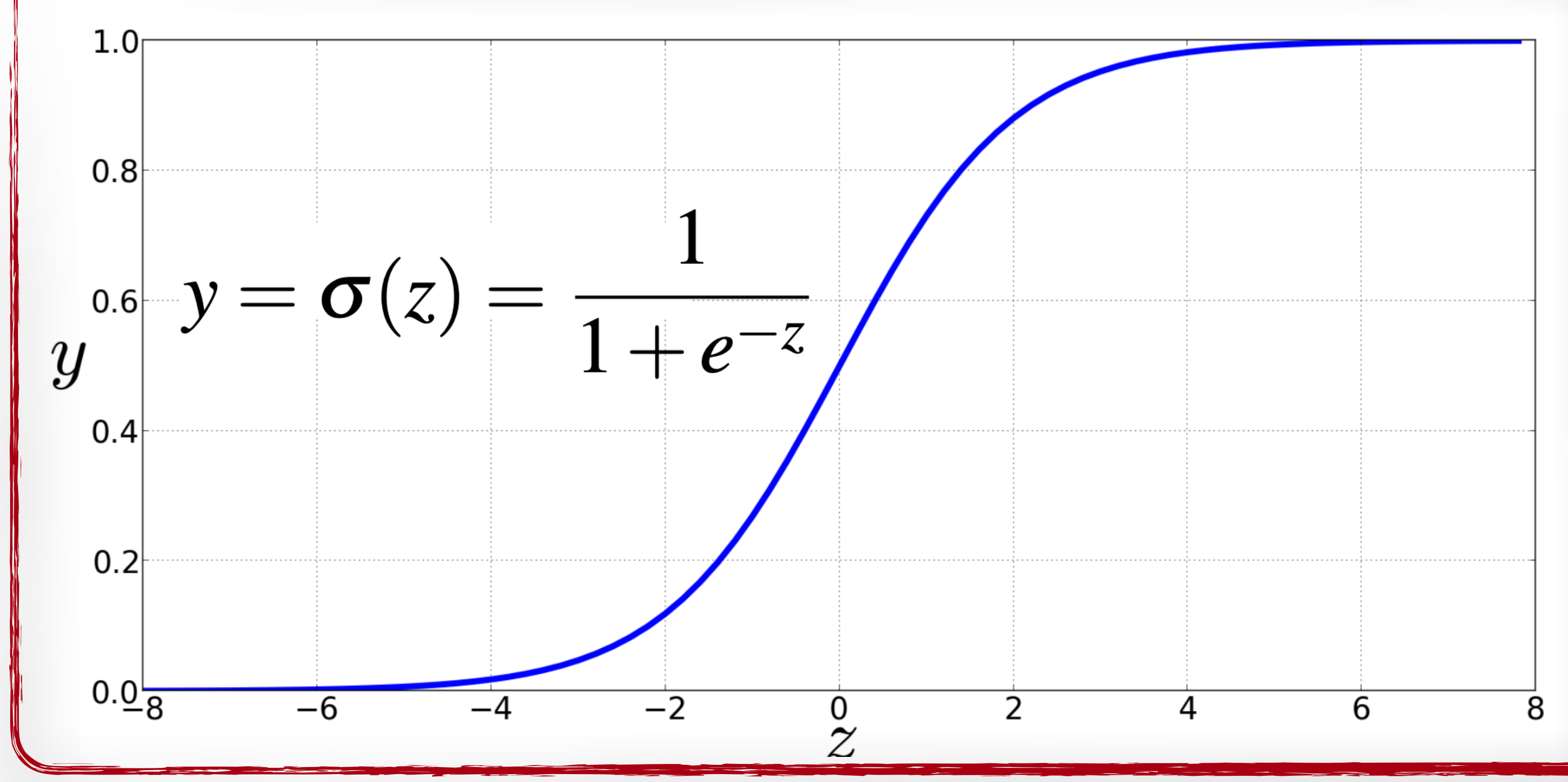
- Recap: Feedforward Neural Nets
- Recurrent Neural Nets
  - Language Models
- Training RNNLMs
- The Vanishing Gradient Problem
- LSTMs

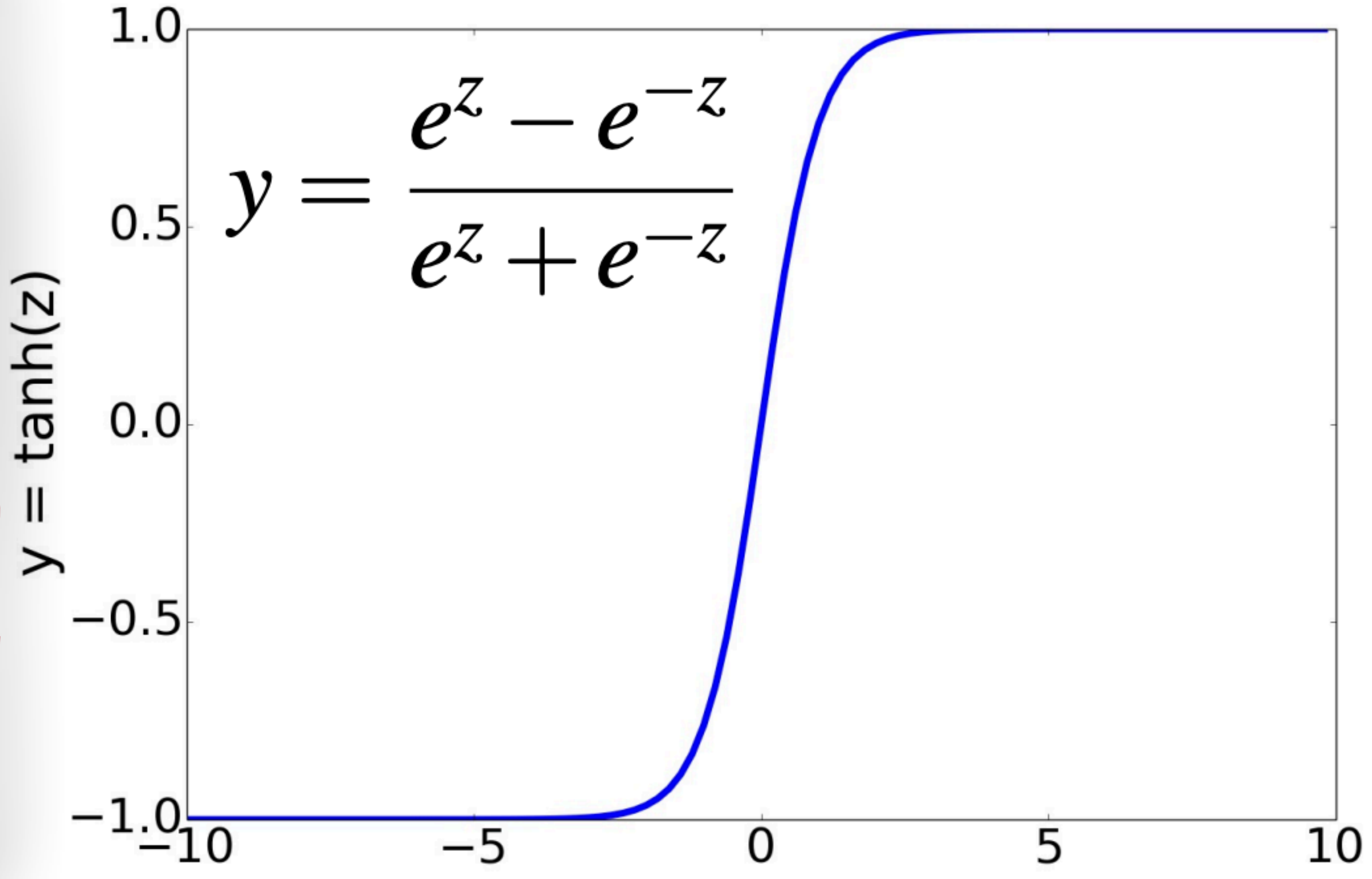
Recap:
Feedforward Neural Nets

Non-Linear Activation Functions

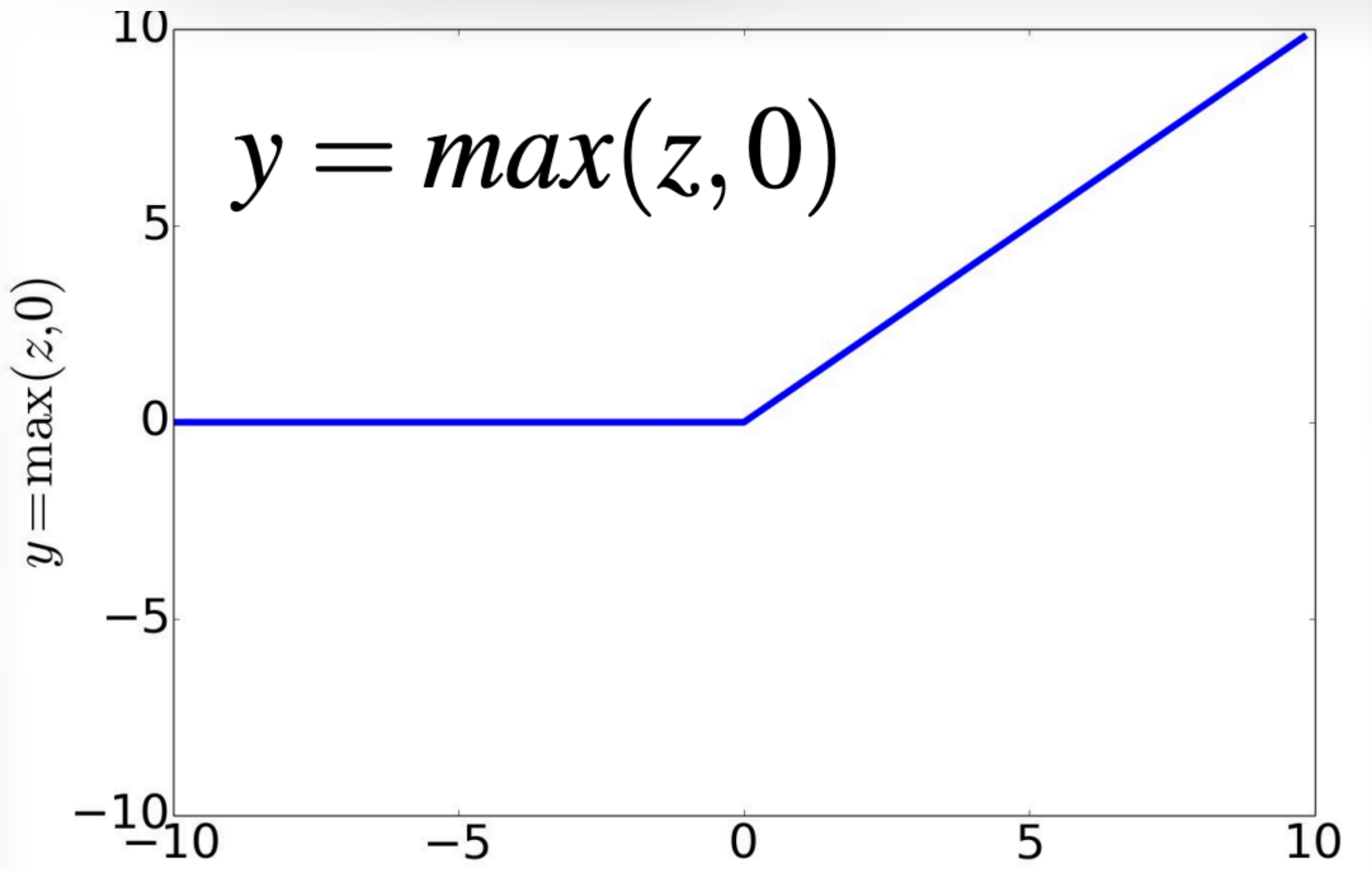
The key ingredient of a neural network is the non-linear activation function

Most common in the output layers





Most common in the hidden layers



sigmoid

softmax

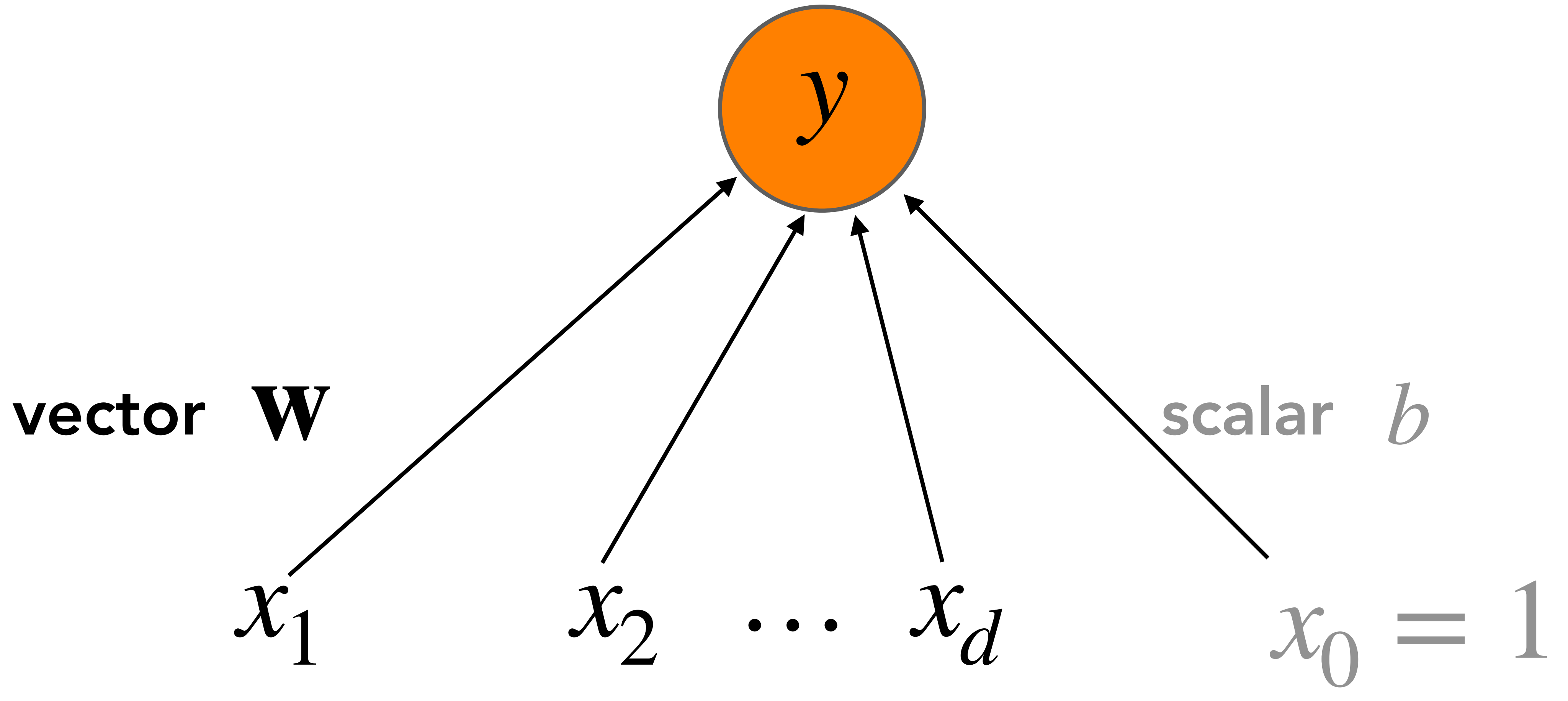
tanh

relu (Rectified Linear Unit)

USC Viterbi

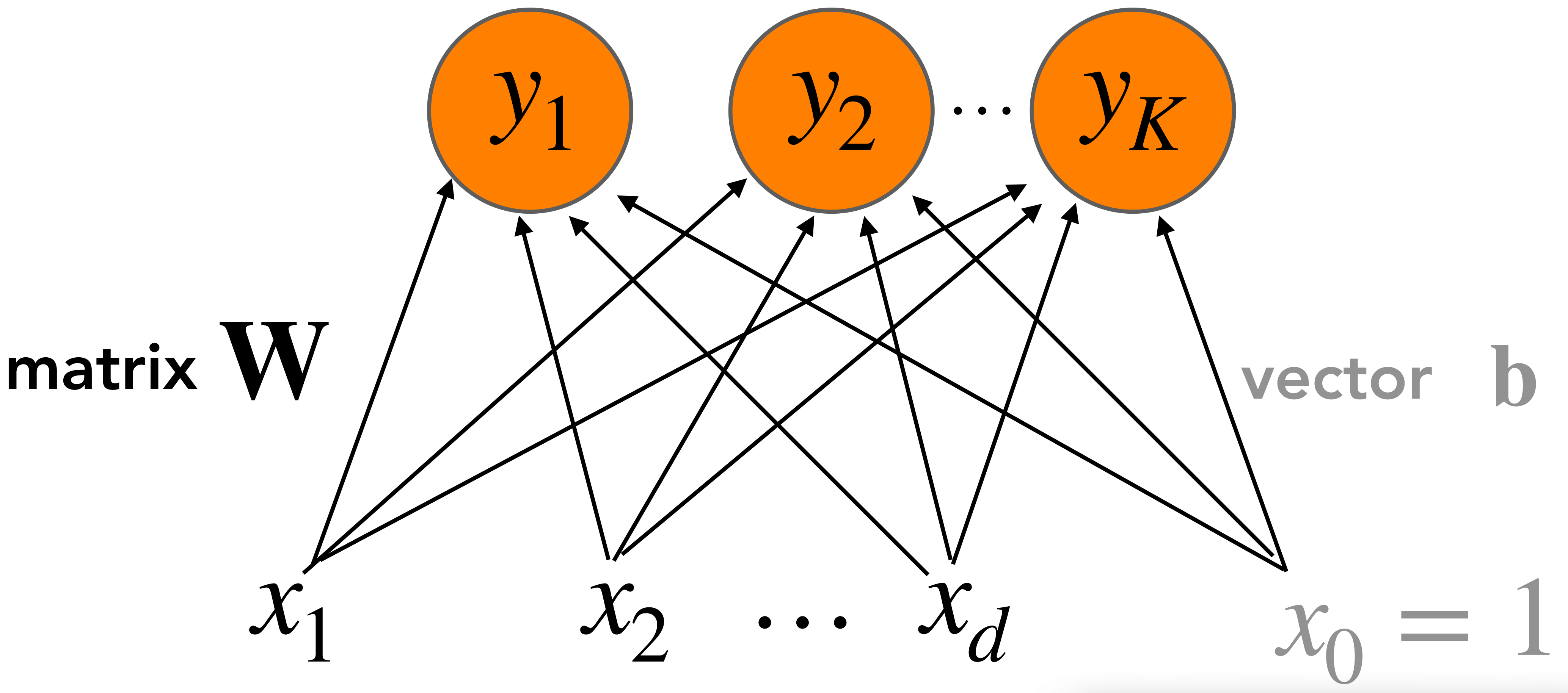
Logistic Regression

Output layer: $y = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$



Binary

Output layer: $y = \text{softmax}(\mathbf{w} \cdot \mathbf{x} + b)$



Multinomial

Input layer: vector $\mathbf{x}$

1-layer Network

Weighted sum of all incoming, followed by a non-linear activation

USC Viterbi

Two-layer Feedforward Network

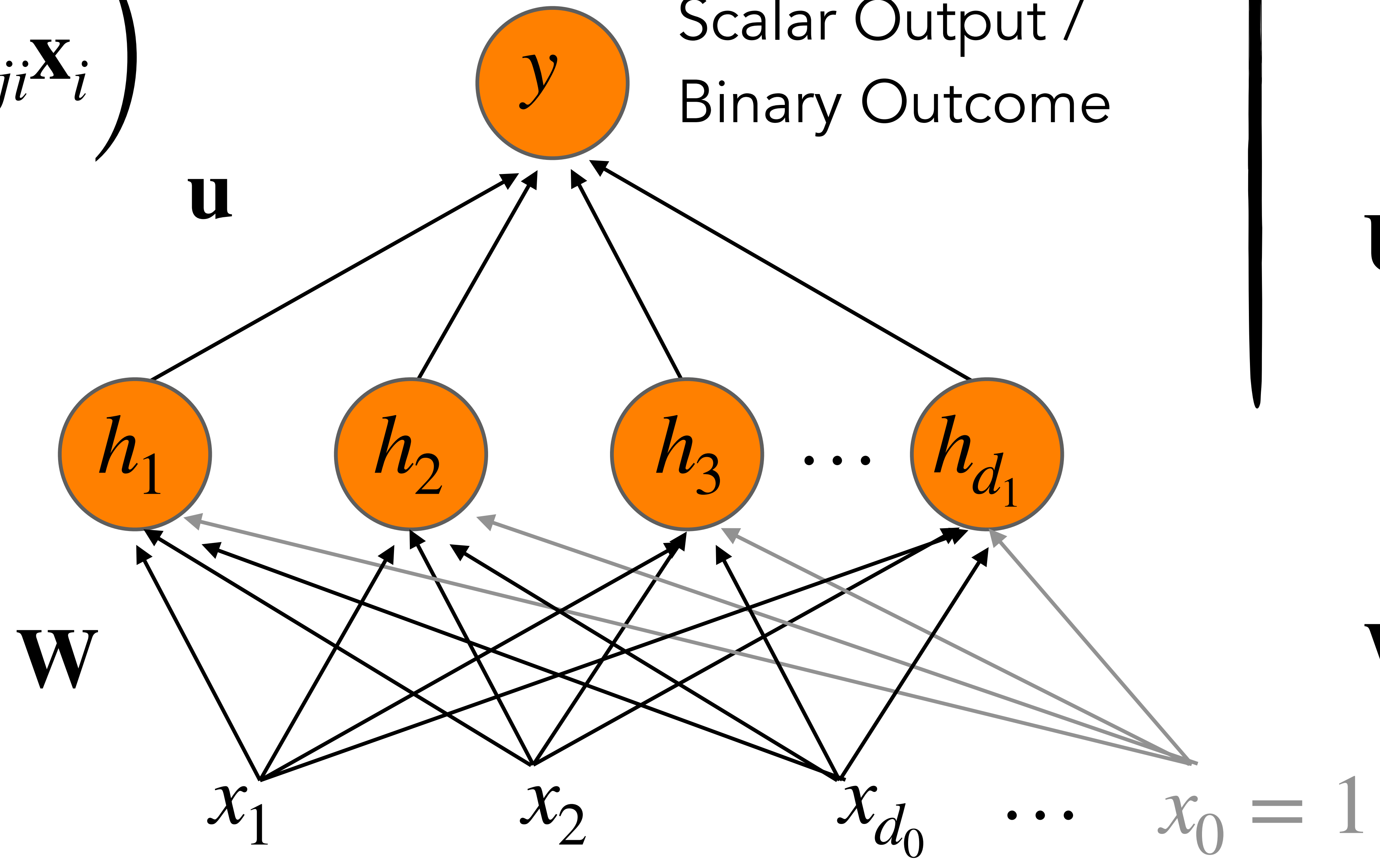
Hidden layer:

$$\mathbf{h} = g(\mathbf{W}\mathbf{x}) = g\left(\sum_{i=0}^{d_0} \mathbf{W}_{ji}\mathbf{x}_i\right)$$

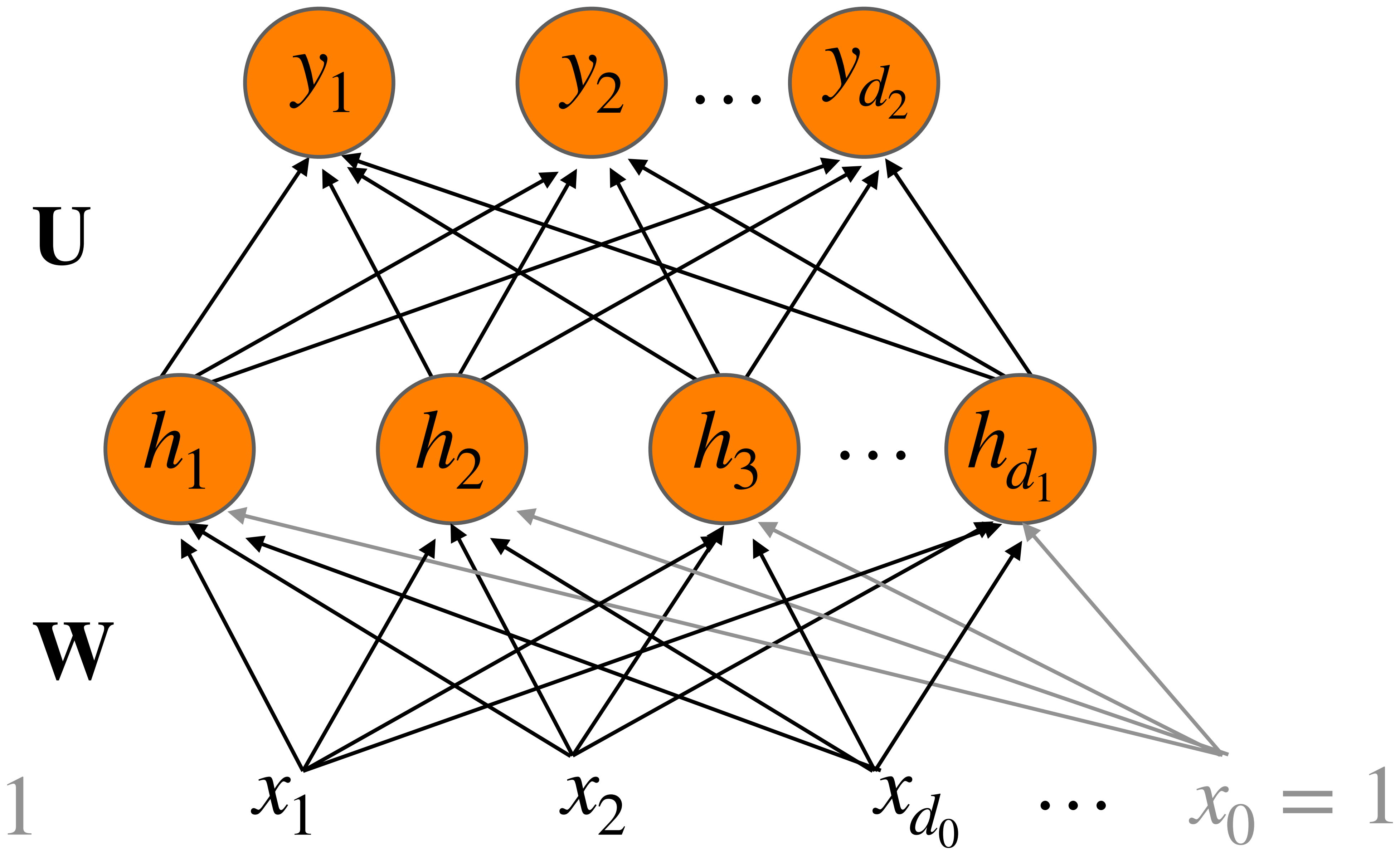
Usually ReLU or tanh

Output layer: $y = \sigma(\mathbf{uh})$

Scalar Output / Binary Outcome



Output layer: $\mathbf{y} = \text{softmax}(\mathbf{U} \cdot \mathbf{h})$



Input layer: vector $\mathbf{x}$

Fully connected single layer network

Simple Feedforward Neural LMs

Task: predict next word w_t given prior words $w_{t-1}, w_{t-2}, w_{t-3}, \ldots$

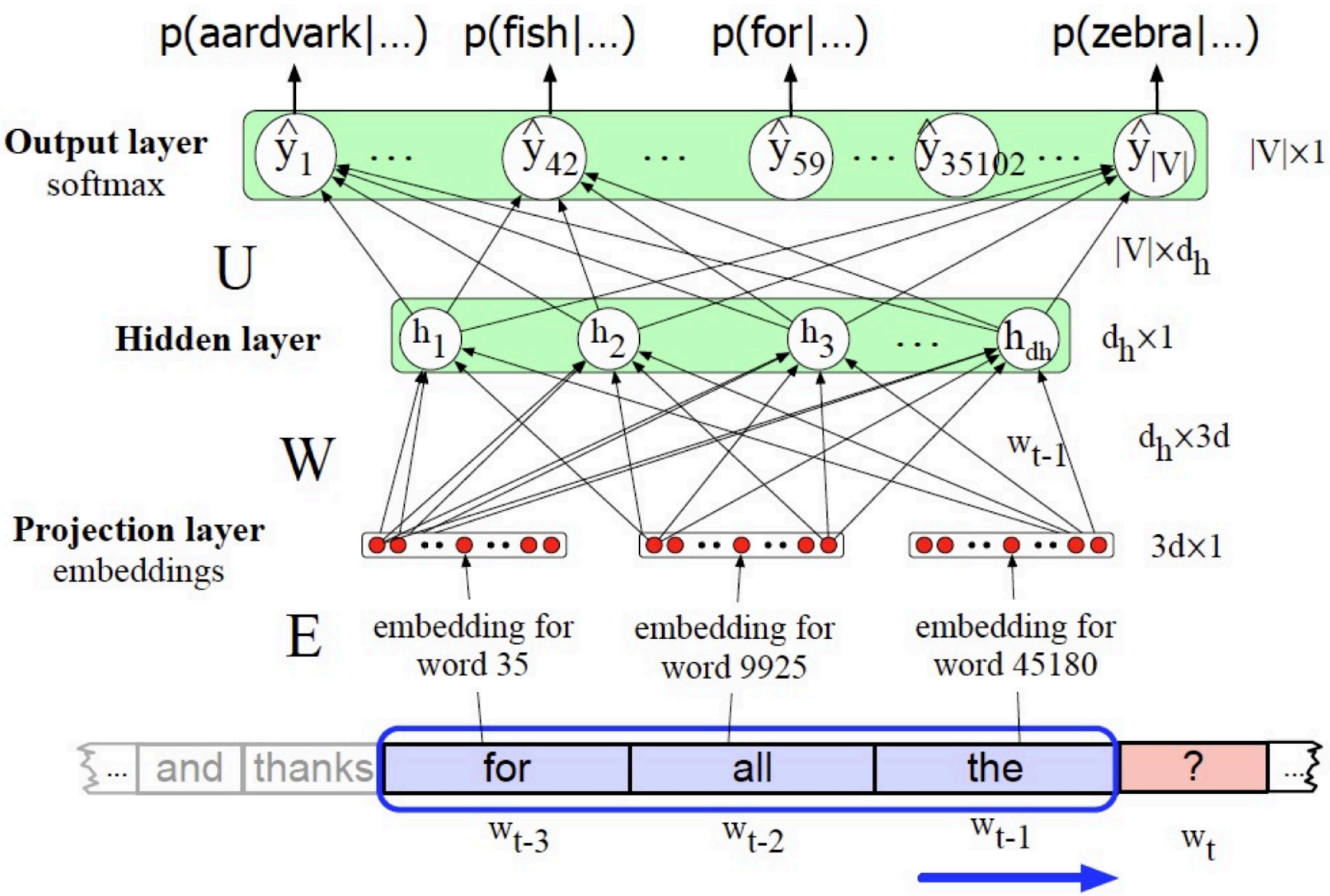
Problem: Dealing with sequences of arbitrary length….

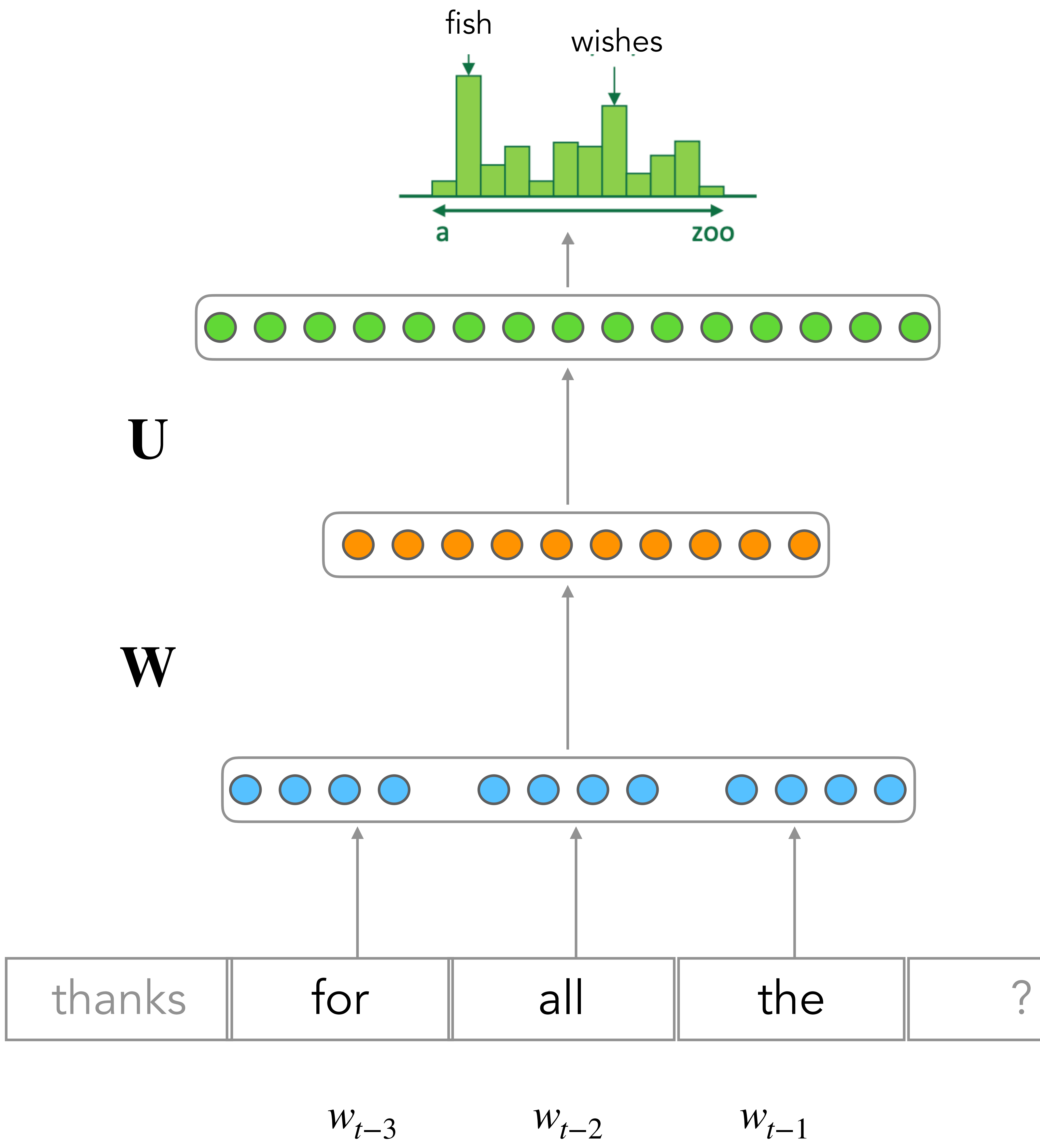
Solution: Sliding windows (of fixed length of size M)

$$P(w_t \mid w_{t-1:t-M+1}) \approx P(w_t \mid w_{t-1:1})$$

USC Viterbi

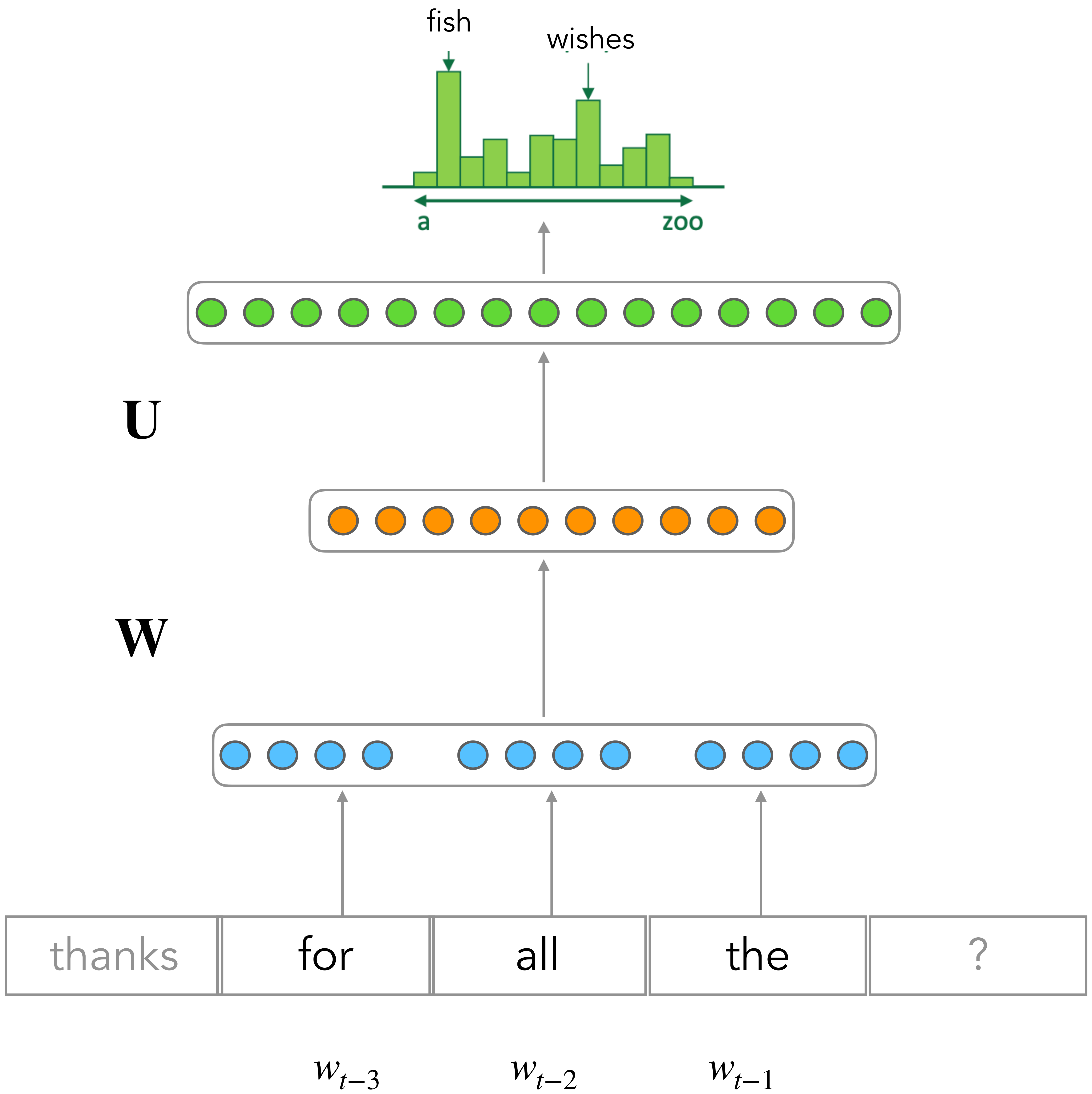
Simplified Representation





Feedforward LMs: Windows

- The goodness of the language model depends on the size of the sliding window!
- Fixed window can be too small
- Enlarging window enlarges $\mathbf{W}$
- Each word uses different rows of $\mathbf{W}$. We don't share weights across the window.
- Window can never be large enough!



Training FFNNs with Backprop and Computation Graphs

USC Viterbi

Feedforward Nets: Loss Function

$$L_{CE}(y, \hat{y}) = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$

- Cross Entropy Again!
- But now we may have many more classes, so we will use the multinomial LR loss
  - Replace sigmoid with softmax
- Now both $\mathbf{y}$ and $\hat{\mathbf{y}}$ are vectors of size K, for the total #classes
- At any time step, only one class is correct
- The true label $\mathbf{y}$ has $\mathbf{y}_c = 1$ if the correct class is c, with all other elements of y being 0
- Classifier will produce an estimate vector $\hat{\mathbf{y}}$, each element represents estimated probability, $\hat{\mathbf{y}}_k = p_\theta(y_k = 1|x)$

What is K for language modeling?

Hard Classification

$$L_{CE}(y, \hat{y}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k \quad = -\log \hat{y}_c, c \text{ being the correct class}$$

$$= -\log \frac{\exp(z_c)}{\sum_{j=1}^{K} \exp(z_j)}, c \text{ being the correct class}$$

USC Viterbi

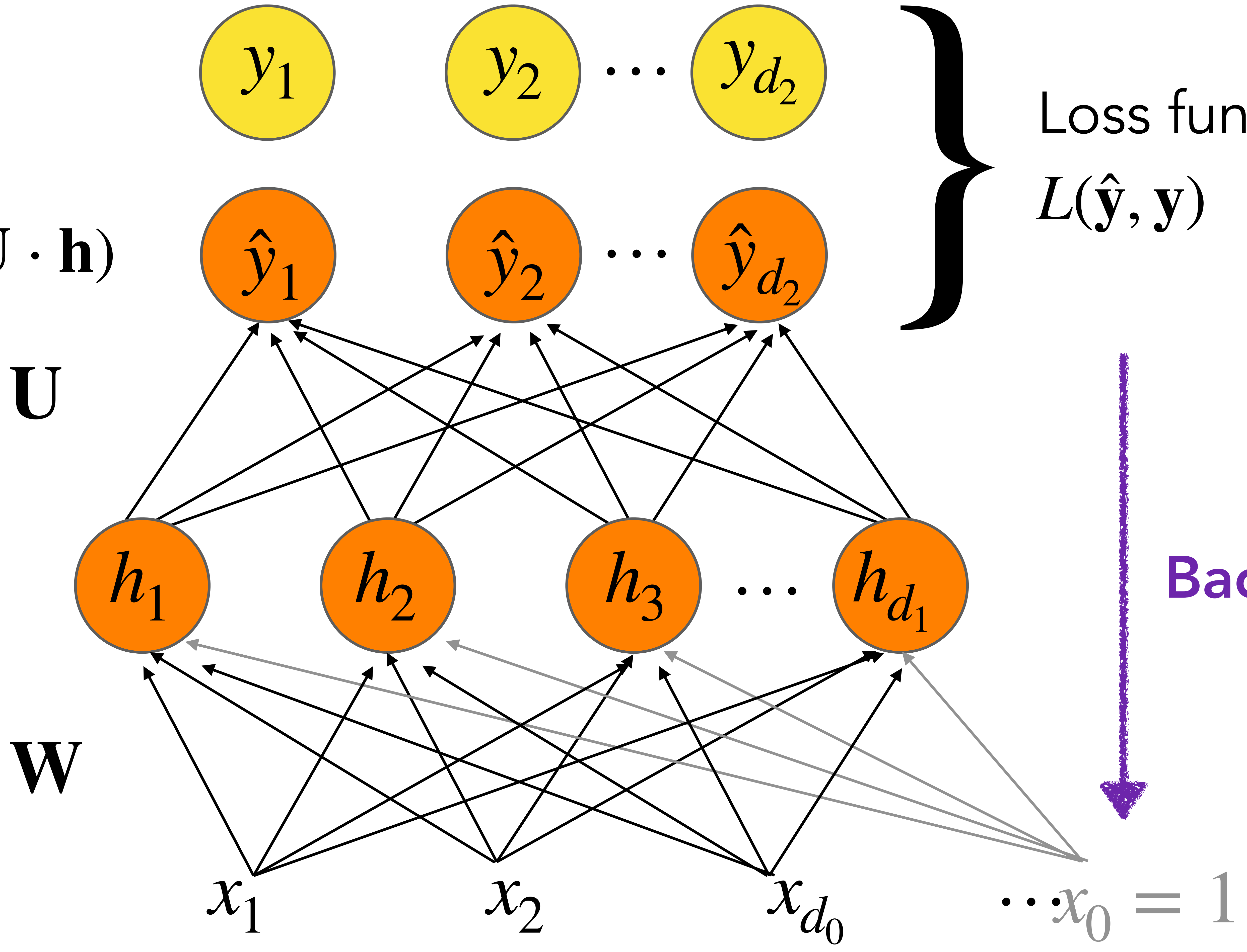
Training a 2-layer Network

Training instance $\mathbf{y}$

Model Output $\hat{\mathbf{y}} = \text{softmax}(\mathbf{U} \cdot \mathbf{h})$

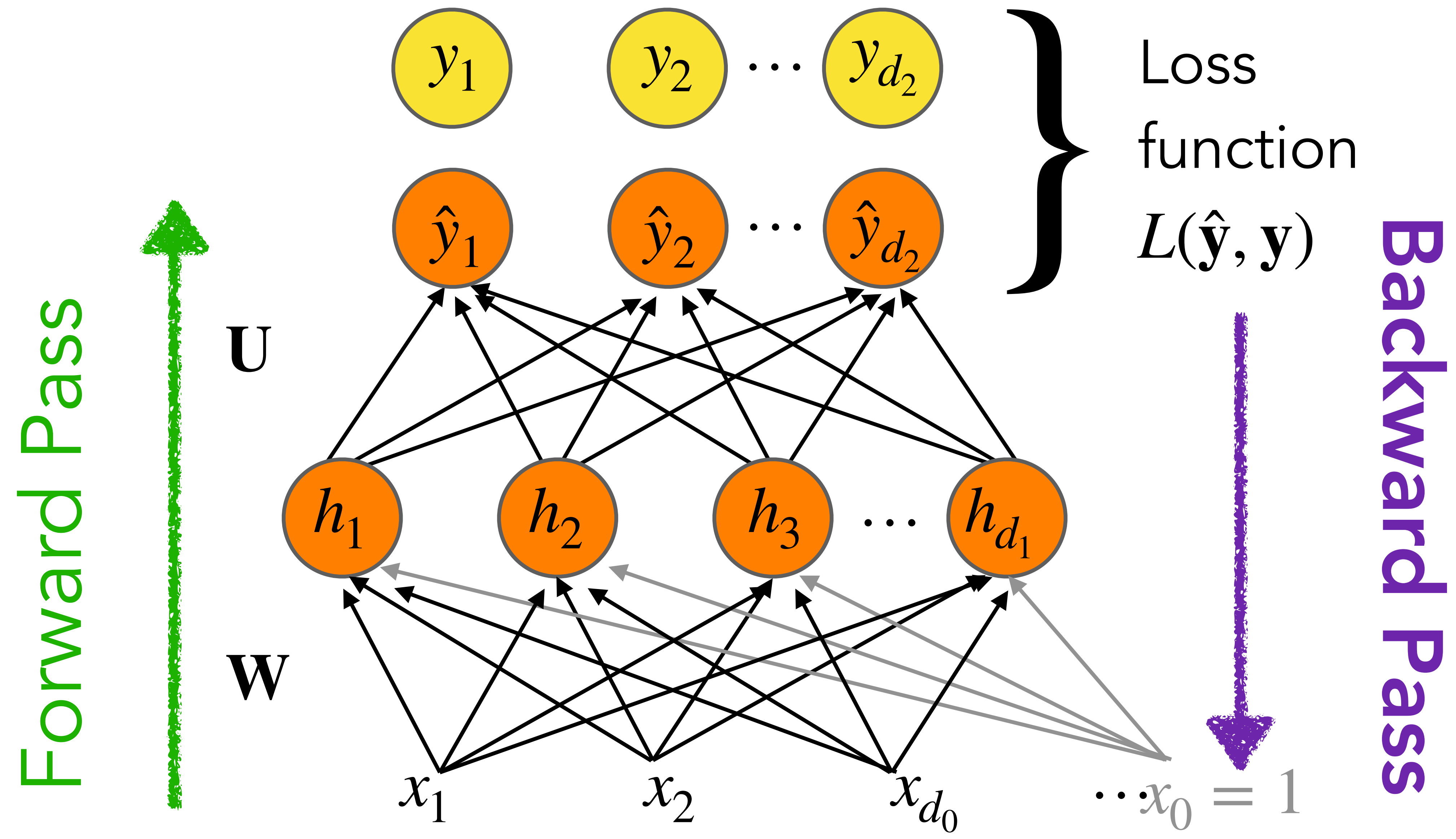
Forward Pass

Training instance $\mathbf{x}$



Loss function
$L(\hat{\mathbf{y}}, \mathbf{y})$

Backward Pass



For every training tuple (x, y)

- Run forward computation to estimate $\hat{y}$ and compute loss L between true y and $\hat{y}$
- Run backward computation to update weights:
  - Output layer: For every weight $\mathbf{U}_{ij}$ from hidden layer to the output layer
    - Update the weight by computing gradient $\dfrac{\partial L}{\partial \mathbf{U}_{ij}}$
  - Hidden layer: For every weight w from input layer to the hidden layer
    - Update the weight by computing gradient $\dfrac{\partial L}{\partial \mathbf{U}_{ij}}$

USC Viterbi

Computation Graphs

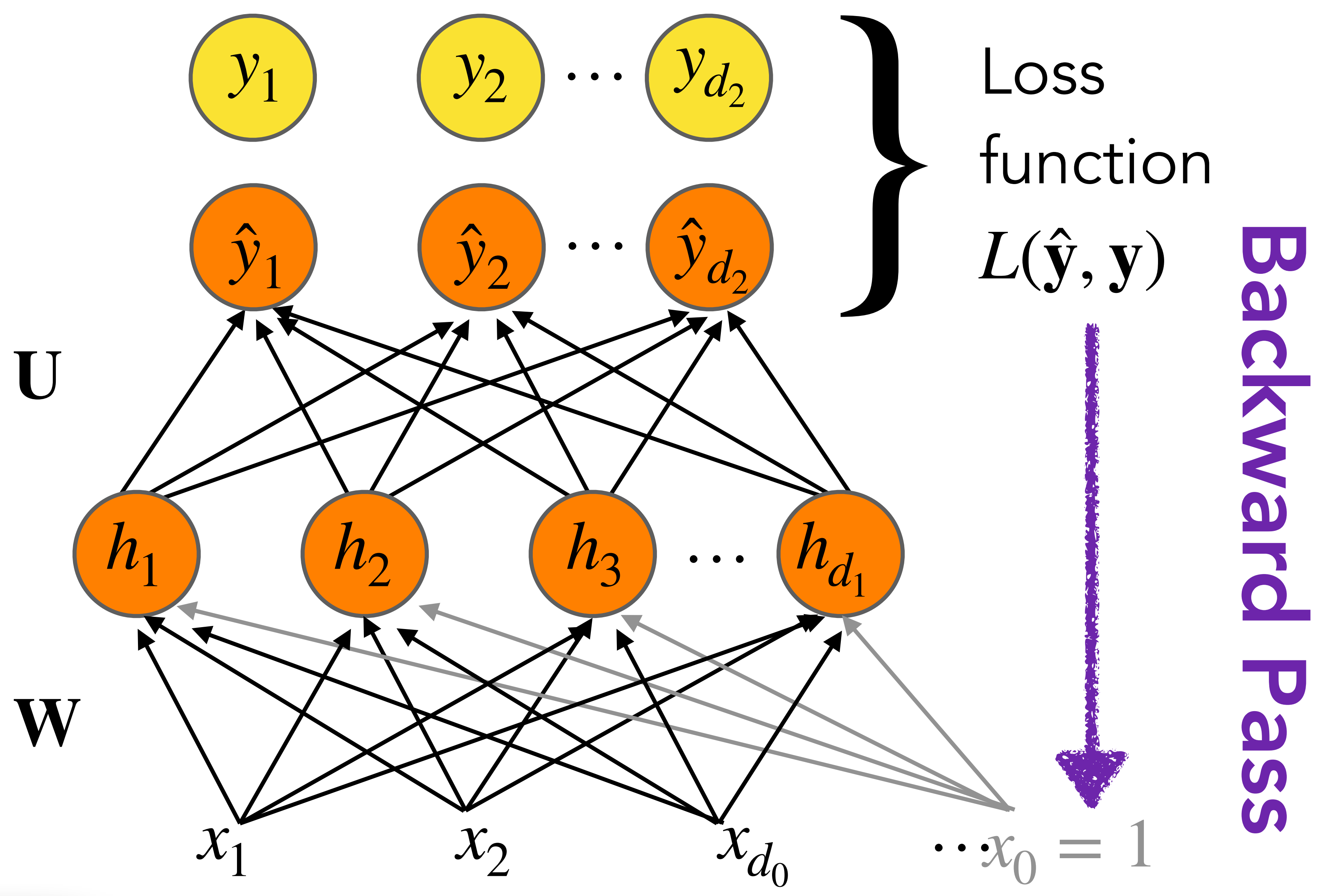
Graph representing the process of computing a mathematical expression

For training, we need the derivative of the loss with respect to each weight in every layer of the network

- But the loss is computed only at the very end of the network!

Solution: error backpropagation or backward differentiation

- Backprop is a special case of backward differentiation which relies on computation graphs



Backprop

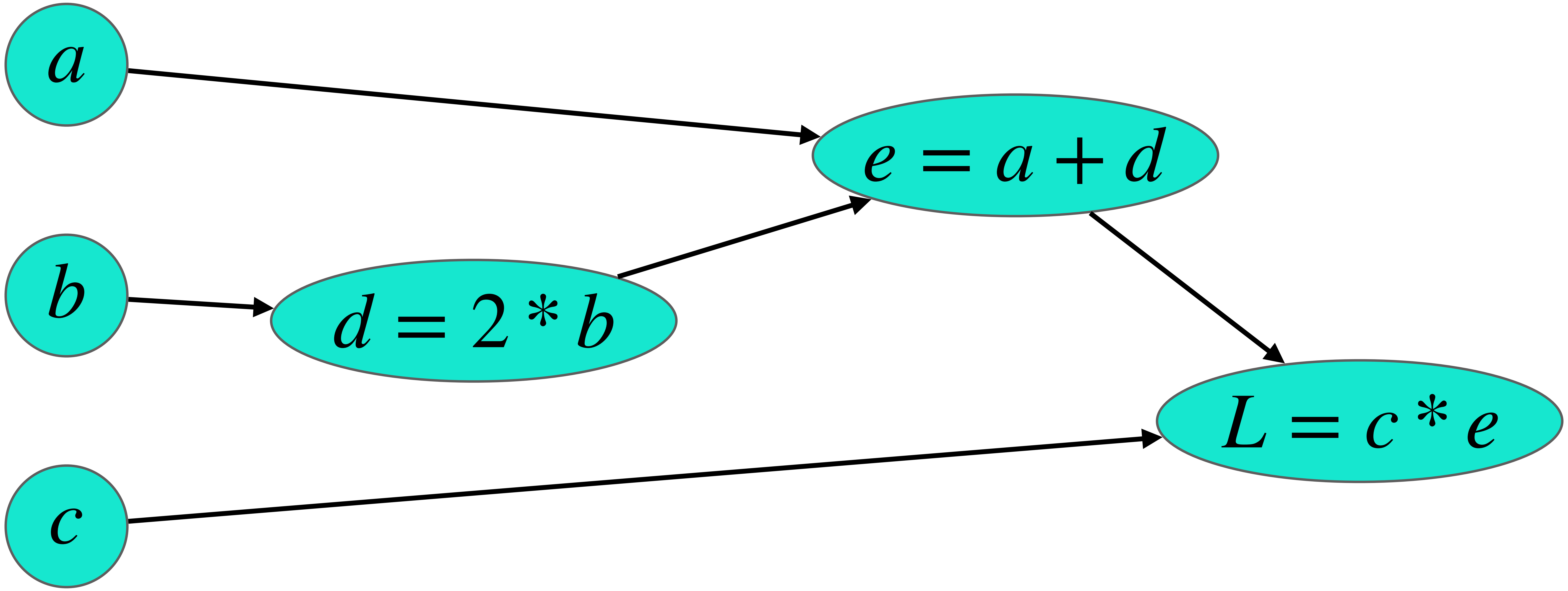
Rumelhart, Hinton, Williams, 1986

Example: Computation Graph

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

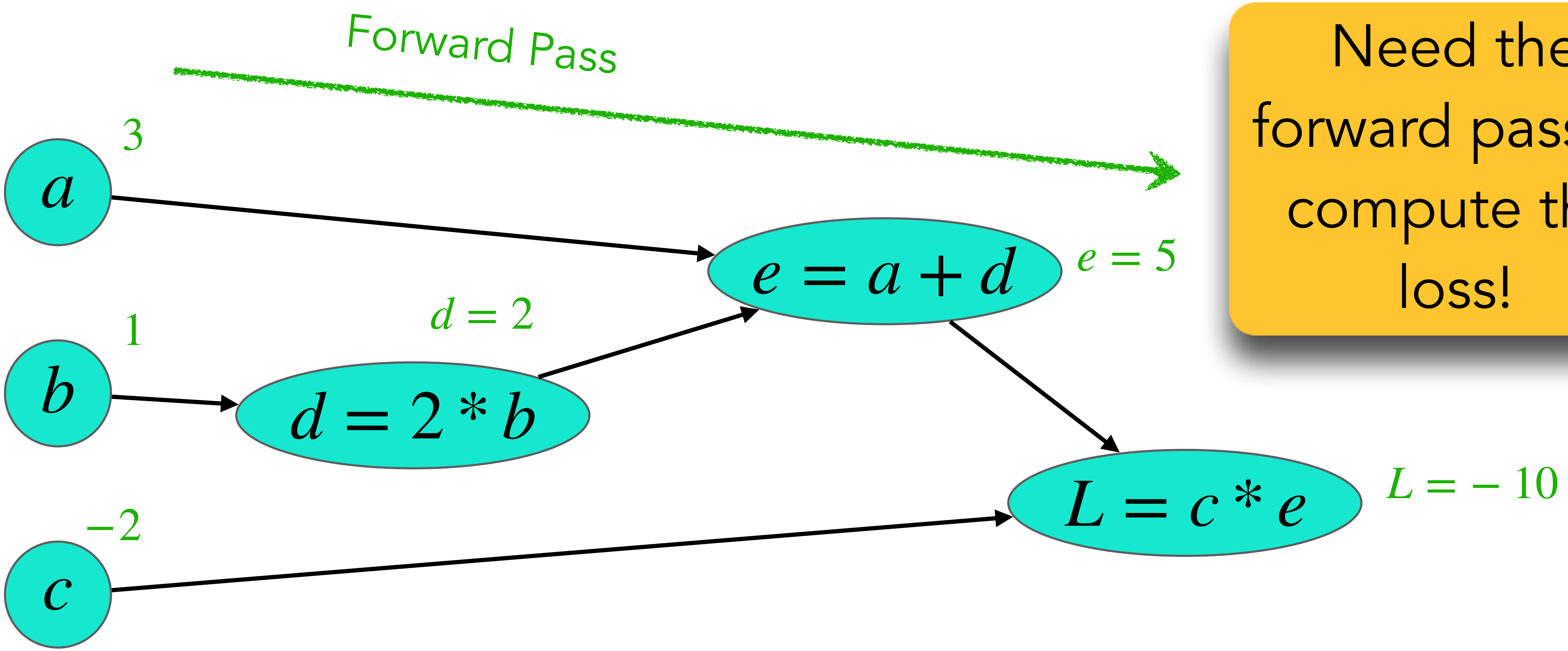


Example: Forward Pass

$d = 2 * b$

$e = a + d$

$L = c * e$



Need the forward pass to compute the loss!

Example: Backward Pass Intuition

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

- The importance of the computation graph comes from the **backward pass**
- Used to compute the derivatives needed for the weight updates

$$\frac{\partial L}{\partial a} = ? \qquad \frac{\partial L}{\partial b} = ? \qquad \frac{\partial L}{\partial c} = ?$$

Input Layer Gradients

Hidden Layer Gradients

$$\frac{\partial L}{\partial d} = ? \qquad \frac{\partial L}{\partial e} = ?$$

Chain Rule of Differentiation!

USC Viterbi

Example: Applying the chain rule

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

$$\frac{\partial L}{\partial c} = e$$

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e} \frac{\partial e}{\partial a}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e} \frac{\partial e}{\partial d} \frac{\partial d}{\partial b}$$

$$\frac{\partial L}{\partial e} = c$$

$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e} \frac{\partial e}{\partial d}$$

Cannot do all at once, need to follow an order…

USC Viterbi

Example: Backward Pass

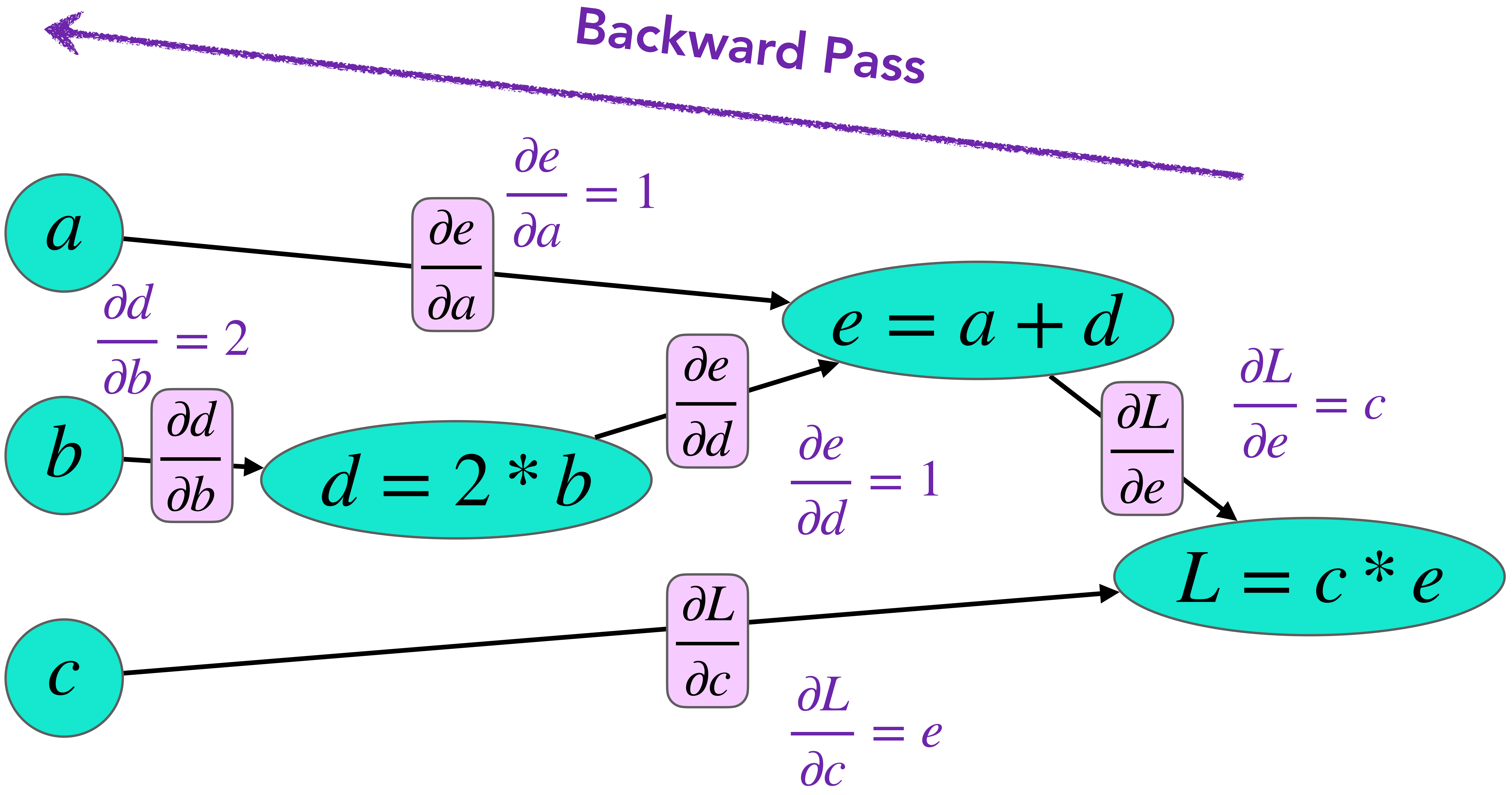
But we need the gradients of the loss with respect to parameters…

$$\frac{\partial L}{\partial c} = e \qquad \frac{\partial L}{\partial e} = c$$

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$



Example

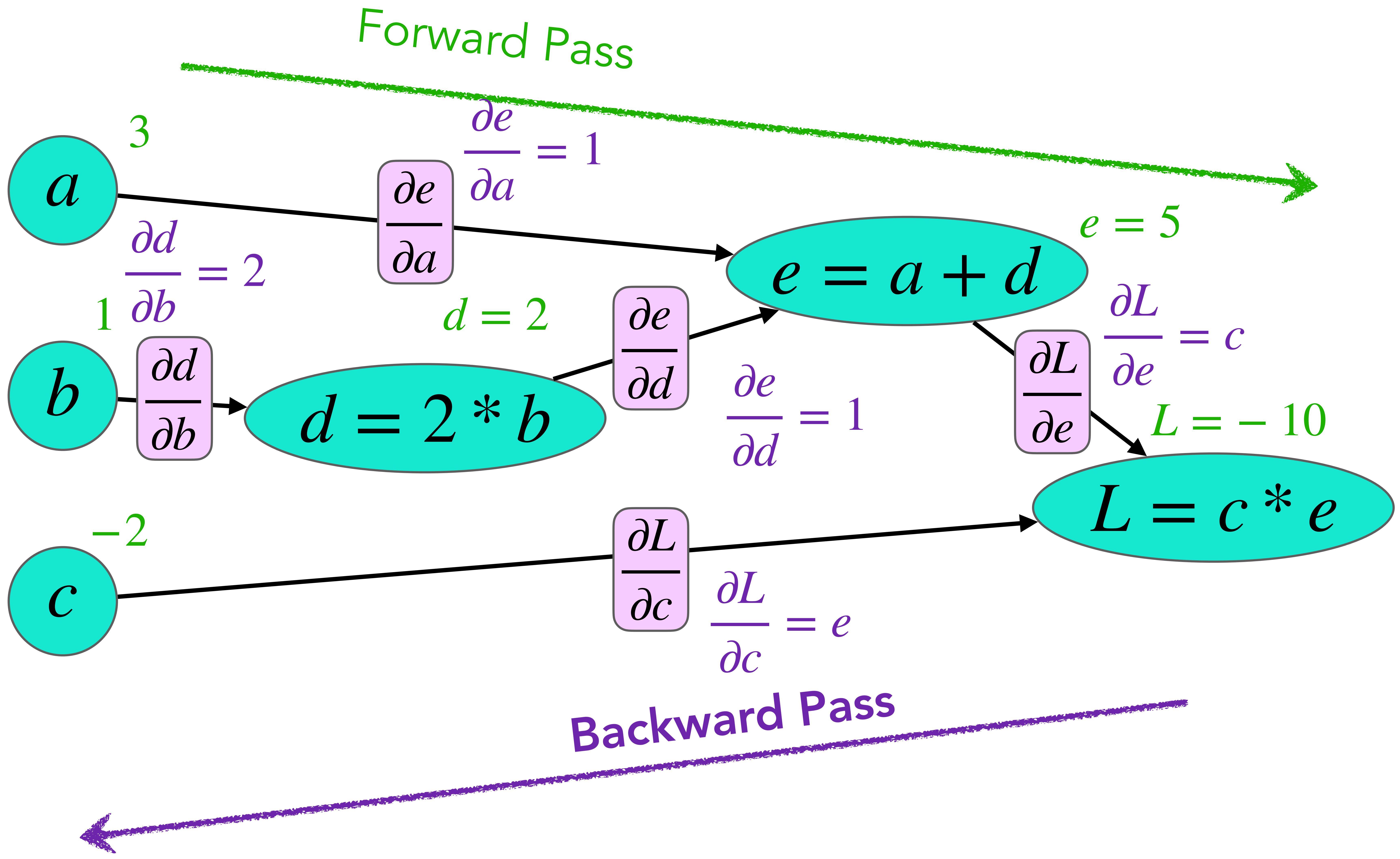
$$\frac{\partial L}{\partial e} = c = -2$$

$$\frac{\partial L}{\partial c} = e = 5$$

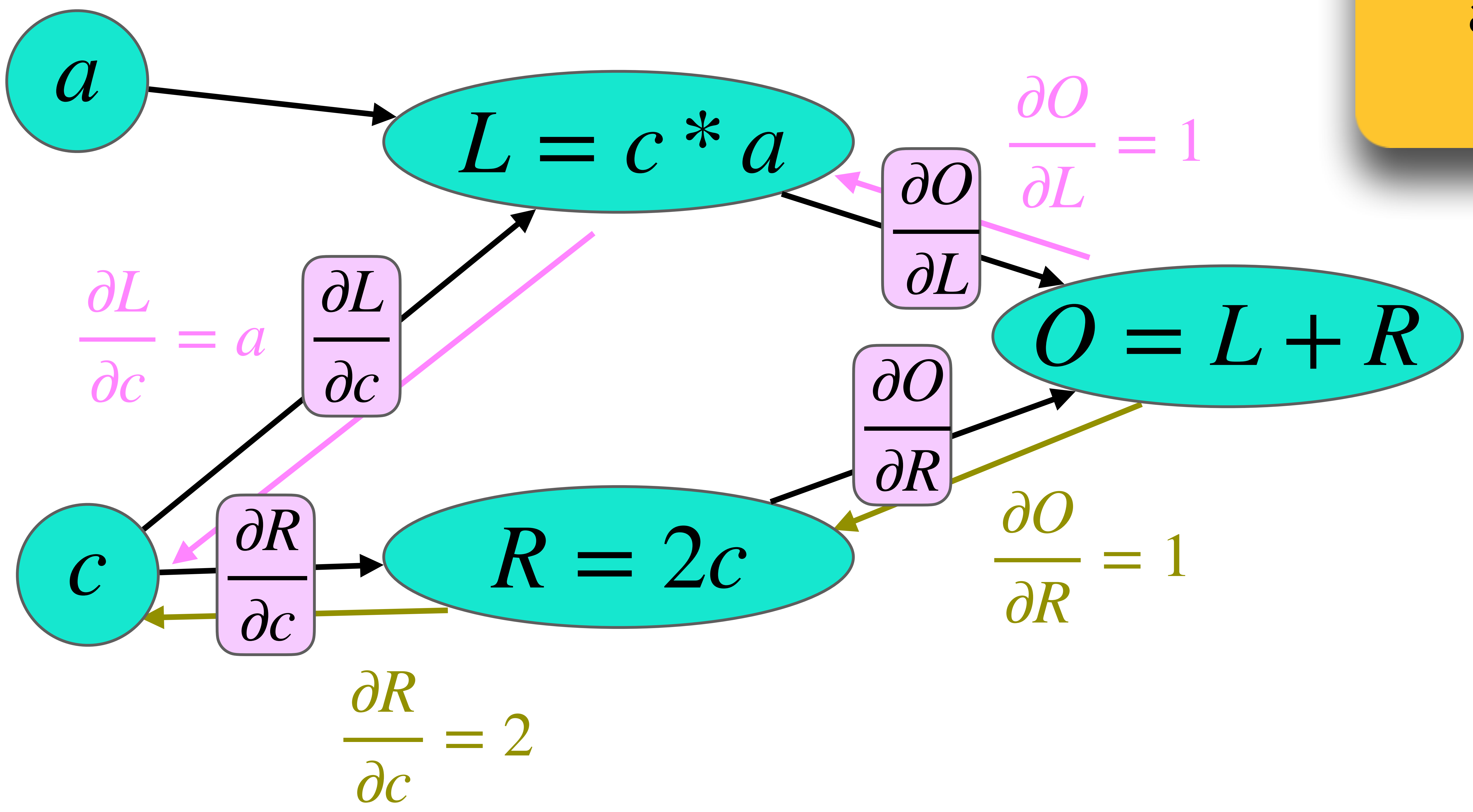
$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} = -2$$

$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d} = -2$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b} = -4$$



Example: Two Paths



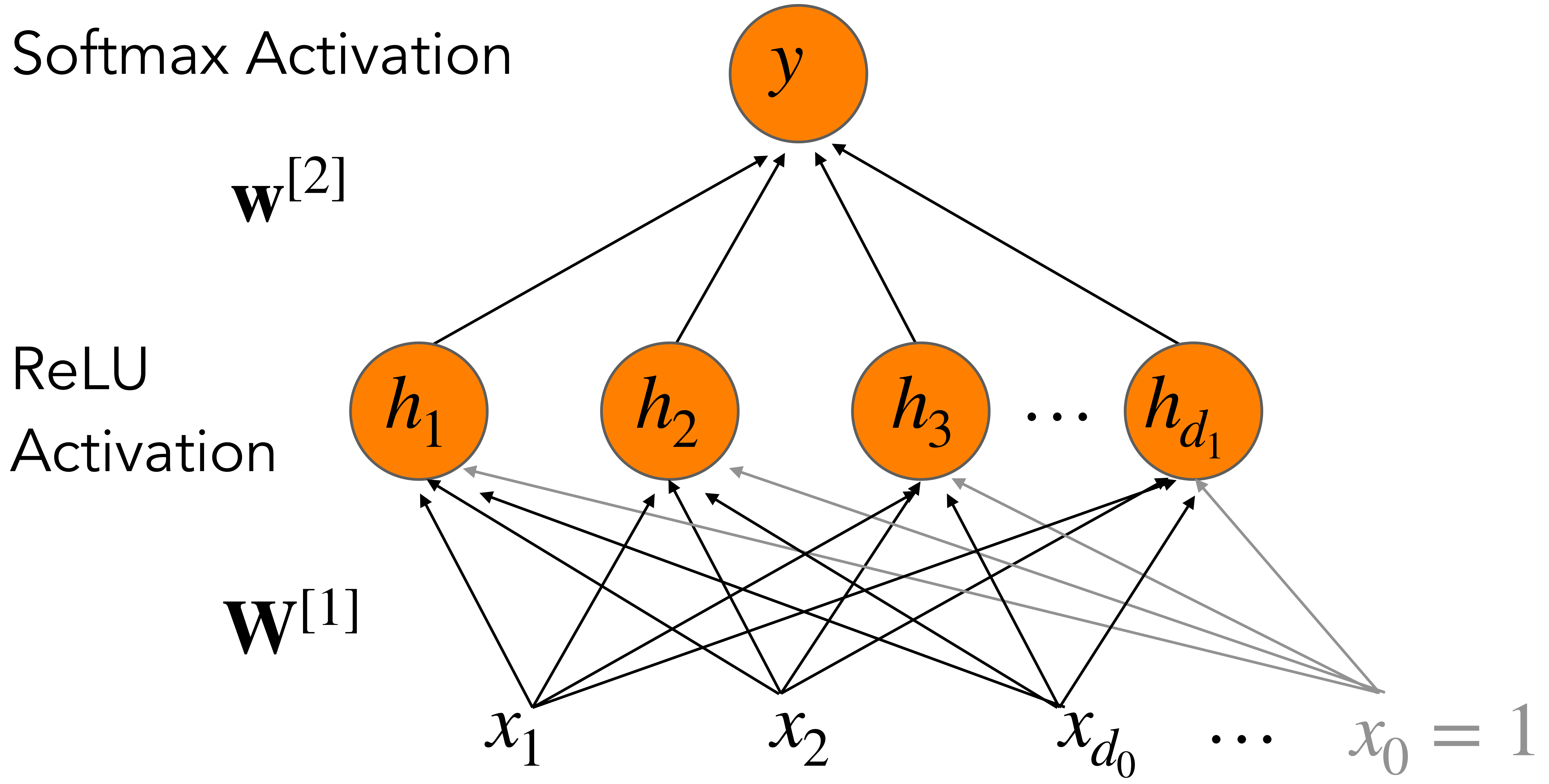
When multiple branches converge on a single node we will add these branches

$$\frac{\partial O}{\partial c} = \frac{\partial O}{\partial L}\frac{\partial L}{\partial c} + \frac{\partial O}{\partial R}\frac{\partial R}{\partial c}$$

Such cases arise when considering regularized loss functions

USC Viterbi

Backward Differentiation on a 2-layer MLP



$$\hat{y} = \sigma(z^{[2]})$$

$$z^{[2]} = \mathbf{w}^{[2]} \cdot \mathbf{h}^{[1]}$$

$$\mathbf{h}^{[1]} = \mathbf{ReLU}(\mathbf{z}^{[1]})$$ Element-wise

$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]}\mathbf{x}$$

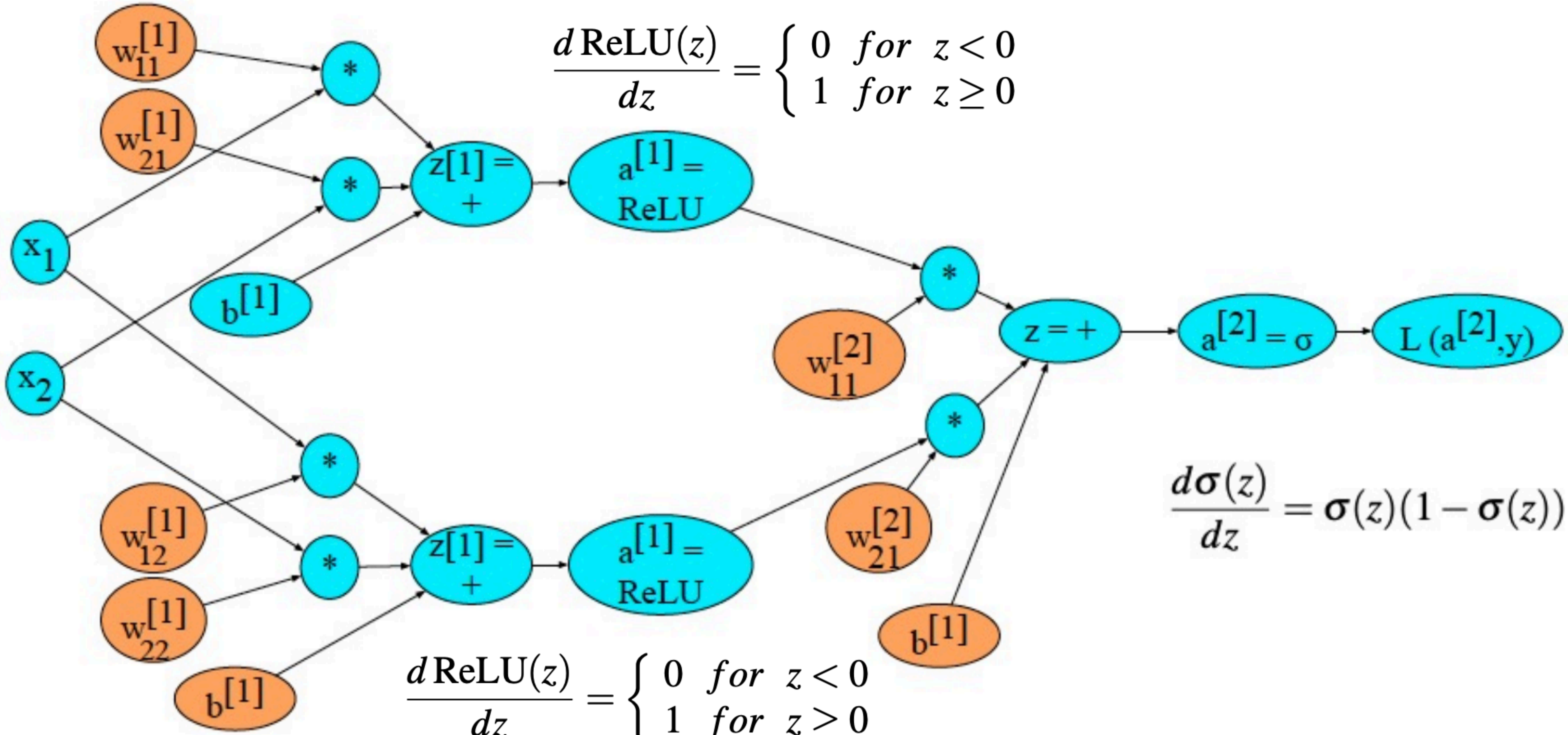
$$\boxed{\frac{\partial \sigma(z)}{\partial z} = \sigma(z)\sigma(-z) = \sigma(z)(1-\sigma(z))}$$

$$\boxed{\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \quad z < 0 \\ 1 & for \quad z \geq 0 \end{cases}}$$

USC Viterbi

2 layer MLP with 2 input features

$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \;\; z < 0 \\ 1 & for \;\; z \geq 0 \end{cases}$$



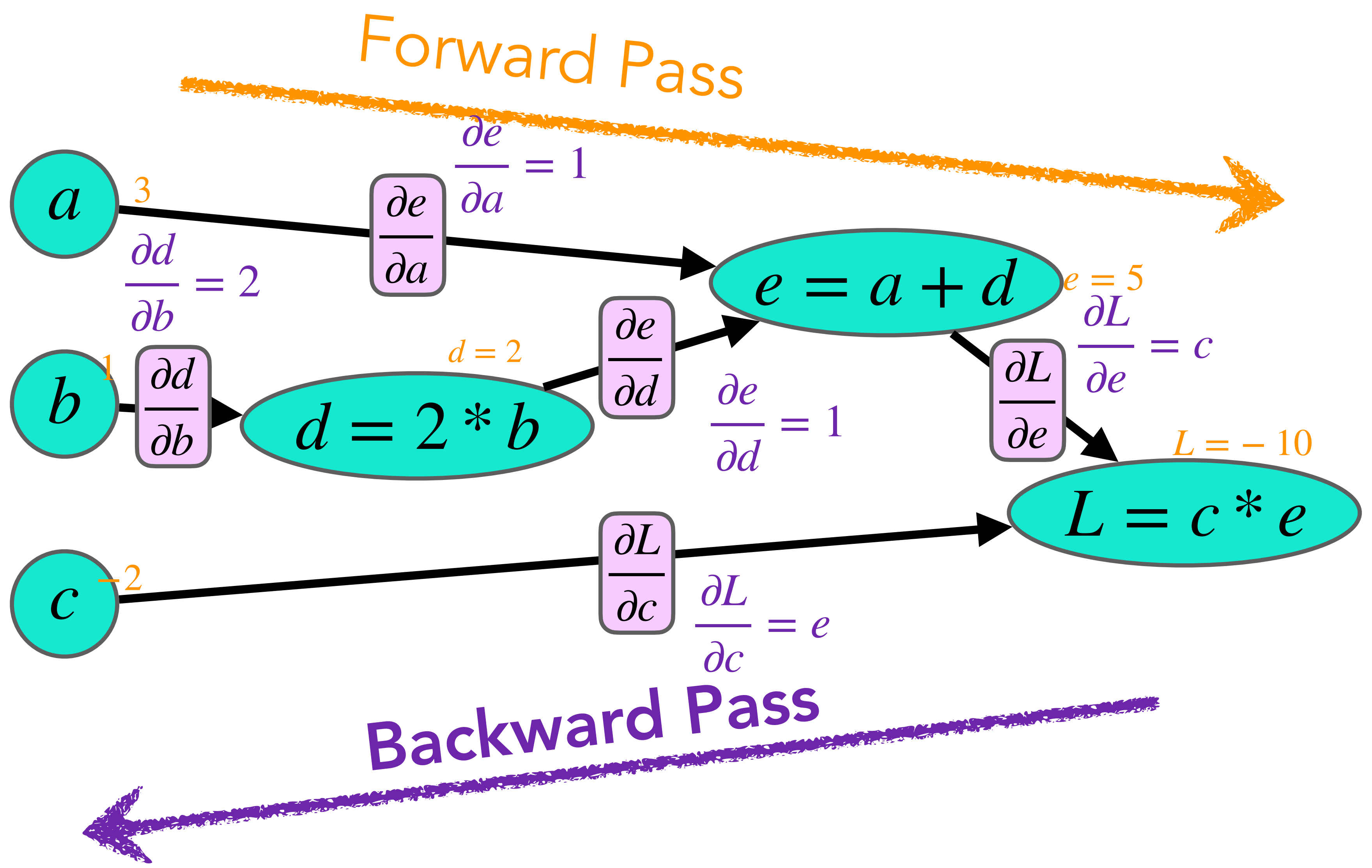
$$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$$

$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \;\; z < 0 \\ 1 & for \;\; z \geq 0 \end{cases}$$

USC Viterbi

Summary: Backprop / Backward Differentiation

- For training, we need the derivative of the loss with respect to weights in early layers of the network
  - But loss is computed only at the very end of the network!
- Solution: **backward differentiation**



Given a computation graph and the derivatives of all the functions in it we can automatically compute the derivative of the loss with respect to these early weights.

Libraries such as PyTorch do this for you in a single line: `model.backward()`

Recurrent Neural Nets

USC Viterbi

Recurrent Neural Networks

- Recurrent Neural Networks processes sequences one element at a time:
  - Contains one hidden layer $\mathbf{h}_t$ per time step! Serves as a memory of the entire history…
  - Output of each neural unit at time t based both on
    - the current input at t and
    - the hidden layer from time $t - 1$
- As the name implies, RNNs have a recursive formulation
    - dependent on its own earlier outputs as an input!
- RNNs thus don’t have
  - the limited context problem that n-gram models have, or
  - the fixed context that feedforward language models have,
  - since the hidden state can *in principle* represent information about all of the preceding words all the way back to the beginning of the sequence

Recurrent Neural Net Language Models

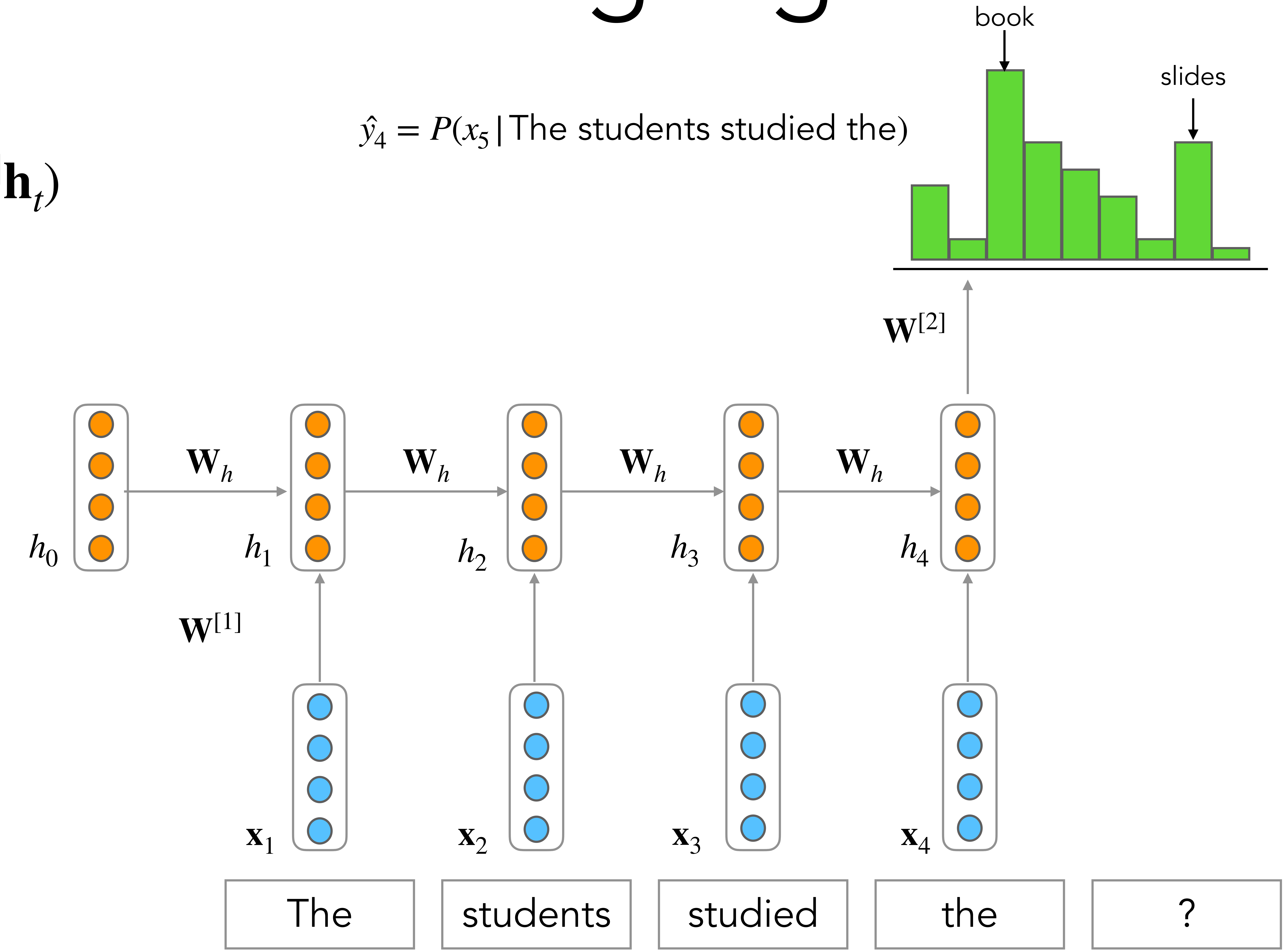
Output layer: $\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{W}^{[2]}\mathbf{h}_t)$

Hidden layer:

$\mathbf{h}_t = g(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}^{[1]}\mathbf{c}_t)$

Initial hidden state: $\mathbf{h}_0$

Word Embeddings, $\mathbf{x}_i$

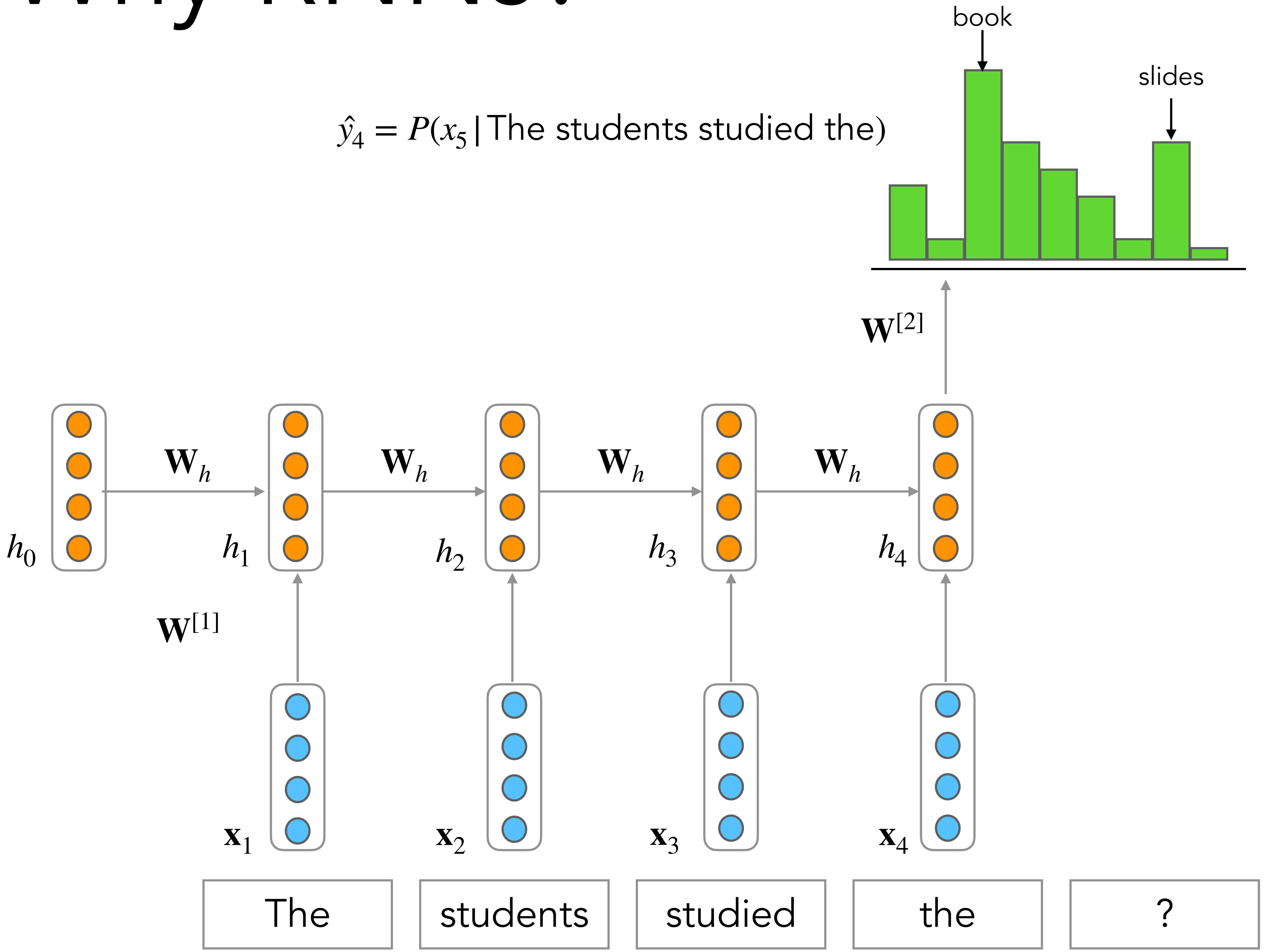


USC Viterbi

Why RNNs?

RNN Advantages:

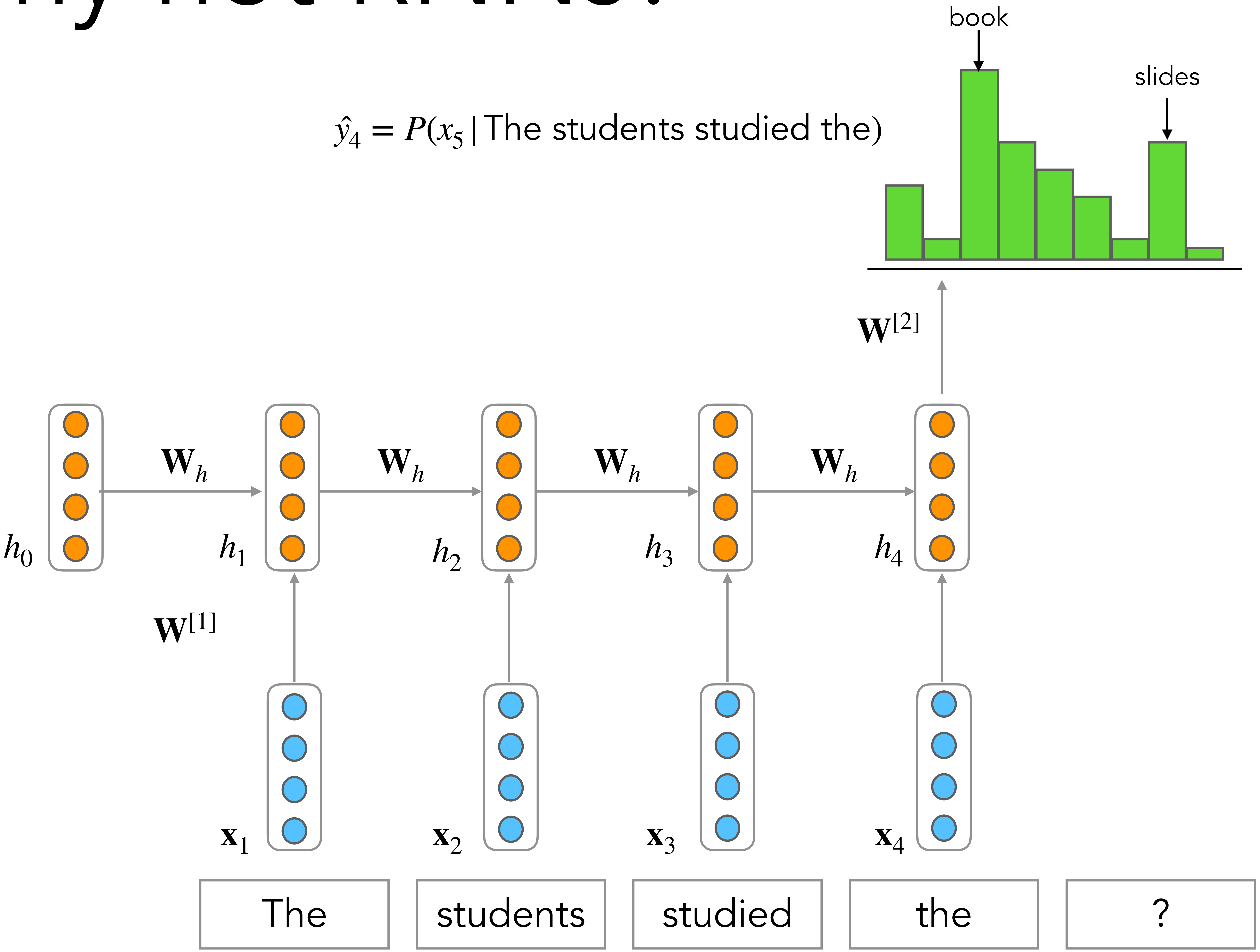
- Can process any length input
- Model size doesn't increase for longer input
- Computation for step t can (in theory) use information from many steps back
- Weights $\mathbf{W}^{[1]}$ are shared (tied) across timesteps → Condition the neural network on all previous words



Why not RNNs?

RNN Disadvantages:

- Recurrent computation is slow
- In practice, difficult to access information from many steps back



Training RNNLMs

Training Outline

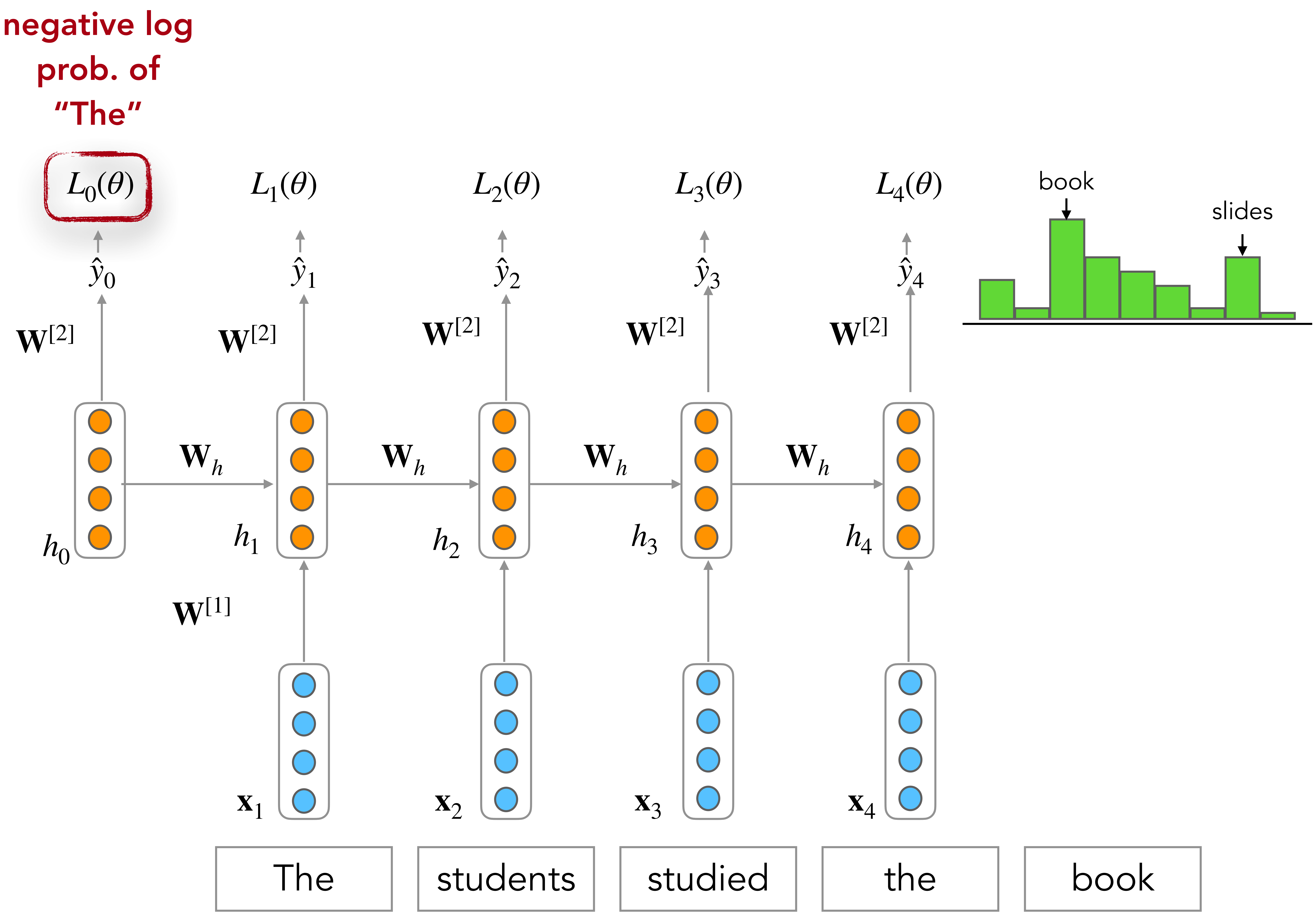
- Get a big corpus of text which is a sequence of words $x_1, x_2, \ldots x_T$
- Feed into RNN-LM; compute output distribution $\hat{y}_t$ for every step t
  - i.e. predict probability distribution of every word, given words so far
- Loss function on step t is usual cross-entropy between our predicted probability distribution $\hat{y}_t$, and the true next word $y_t = x_{t+1}$:

$$L_{CE}(\hat{y}_t, y_t; \theta) = -\sum_{v \in V} \mathbb{I}[y_t = v] \log \hat{y}_t = -\log p_\theta(x_{t+1} \mid x_{\leq t})$$

- Average this to get overall loss for entire training set:

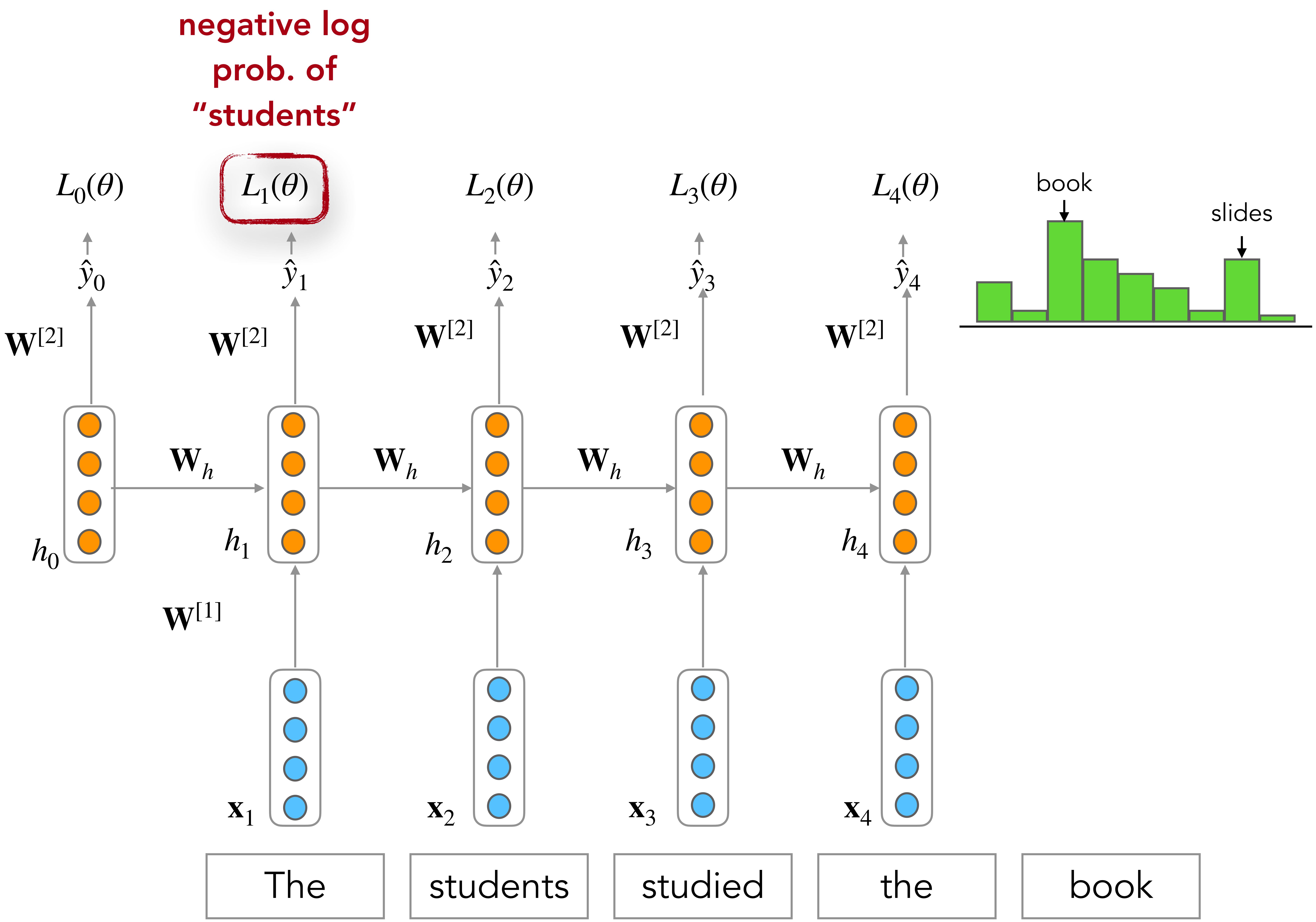
$$L(\theta) = \frac{1}{T} \sum_{t=1}^{T} L_{CE}(\hat{y}_t, y_t)$$

Loss



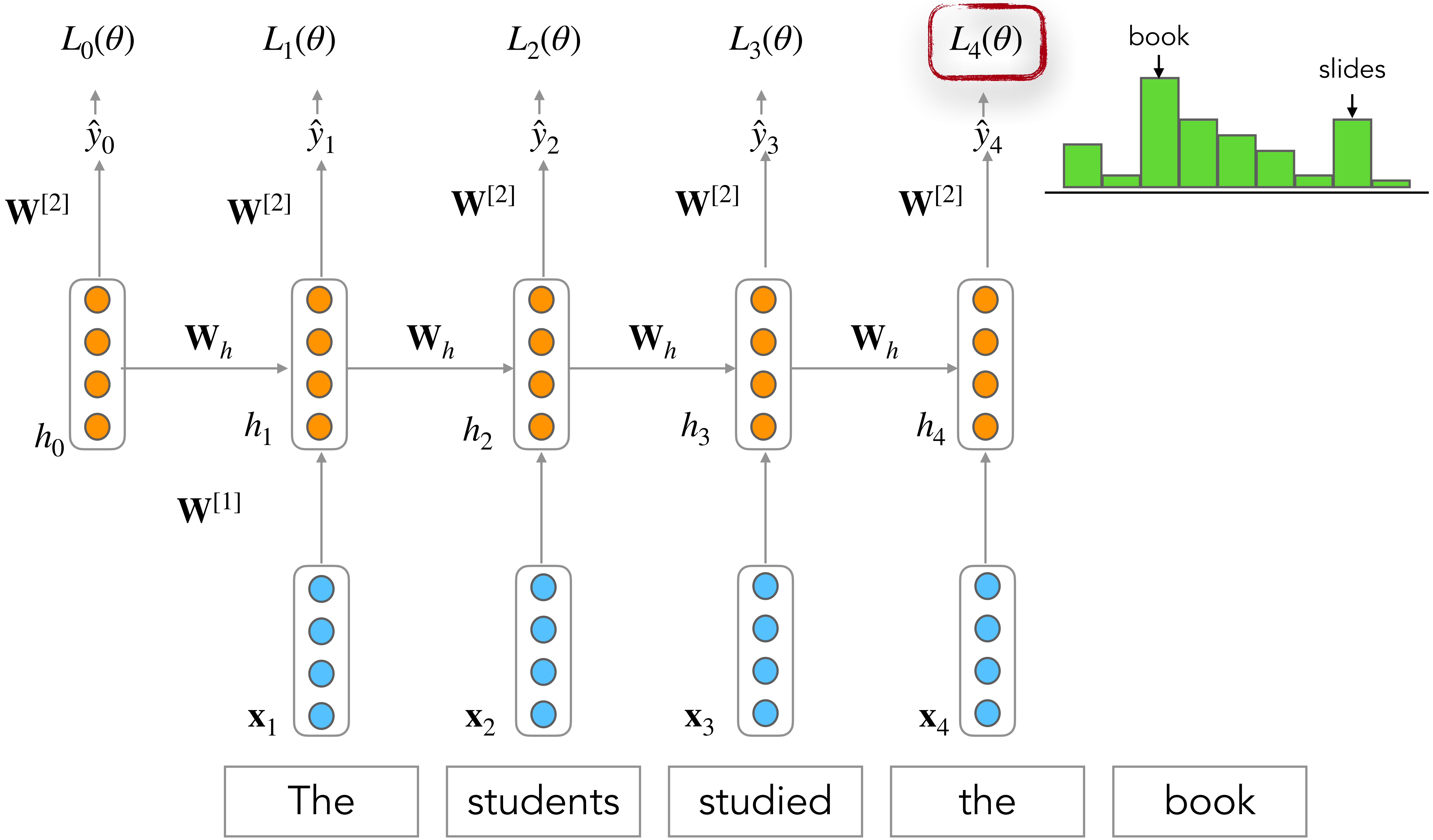
negative log prob. of "The"
$L_0(\theta)$
$L_1(\theta)$
$L_2(\theta)$
$L_3(\theta)$
$L_4(\theta)$
book
slides
$\hat{y}_0$
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
$\hat{y}_4$
$\mathbf{W}^{[2]}$
$\mathbf{W}_h$
h_0
h_1
h_2
h_3
h_4
$\mathbf{W}^{[1]}$
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_3$
$\mathbf{x}_4$
The
students
studied
the
book

Loss


negative log prob. of "students"
$L_0(\theta)$
$L_1(\theta)$
$L_2(\theta)$
$L_3(\theta)$
$L_4(\theta)$
book
slides
$\hat{y}_0$
$\hat{y}_1$
$\hat{y}_2$
$\hat{y}_3$
$\hat{y}_4$
$\mathbf{W}^{[2]}$
$\mathbf{W}_h$
h_0
h_1
h_2
h_3
h_4
$\mathbf{W}^{[1]}$
$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_3$
$\mathbf{x}_4$
The
students
studied
the
book

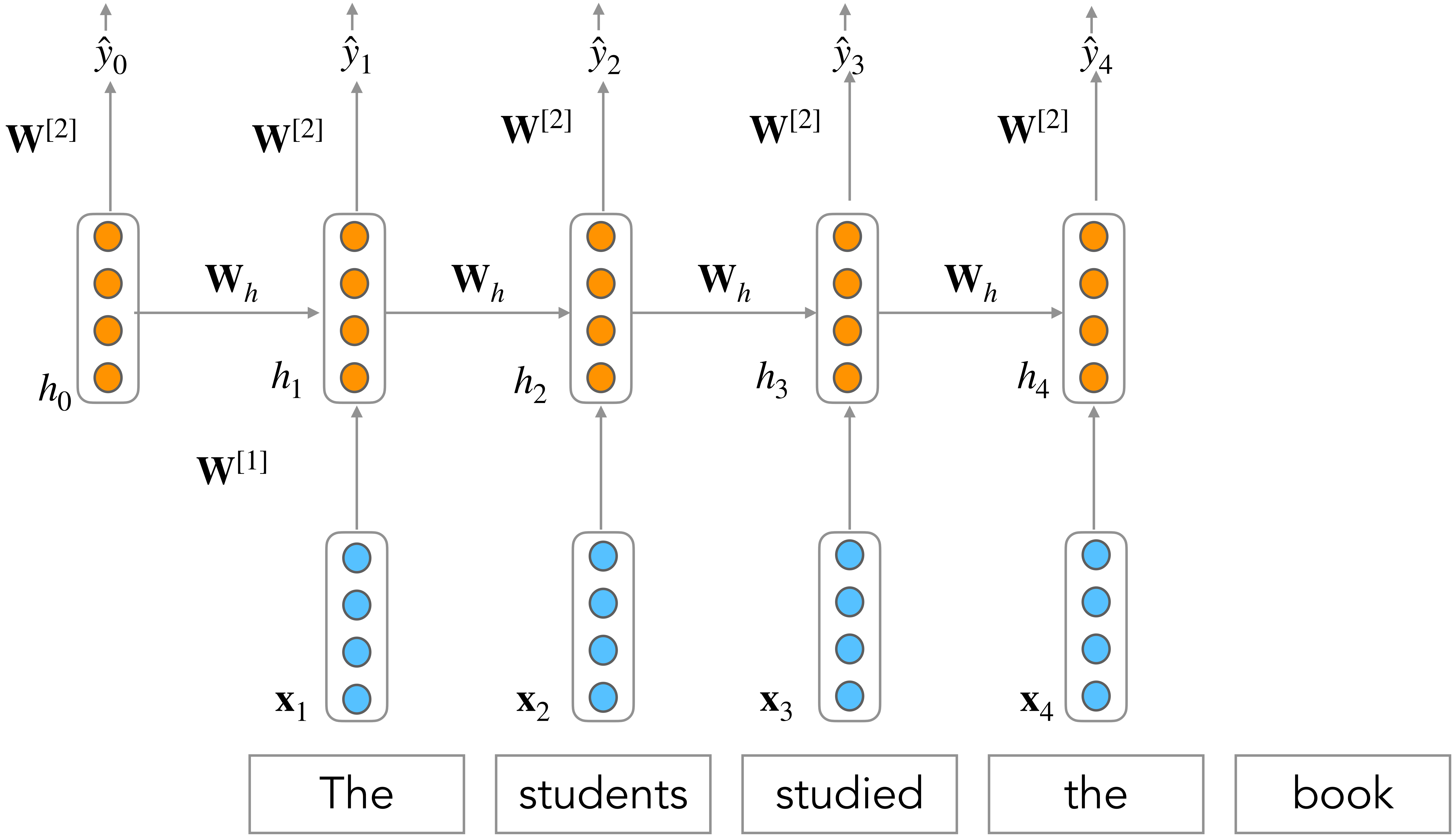
Loss

negative log prob. of "book"



Loss

$$L_0(\theta) + L_1(\theta) + L_2(\theta) + L_3(\theta) + L_4(\theta) + \ldots = L(\theta) = \frac{1}{T}\sum_{t=1}^{T} L_t(\theta)$$



RNNs vs. Other LMs

Table 2. *Comparison of different neural network architectures on Penn Corpus (1M words) and Switchboard (4M words).*

	Penn Corpus		Switchboard	
Model	NN	NN+KN	NN	NN+KN
KN5 (baseline)	-	141	-	92.9
feedforward NN	141	118	85.1	77.5
RNN trained by BP	137	113	81.3	75.4
RNN trained by BPTT	123	106	77.5	72.5

T. **Mikolov**, S. Kombrink, L. Burget, J. Černocký and S. Khudanpur, "Extensions of recurrent neural network language model," *2011 IEEE ICASSP*, doi: 10.1109/ICASSP.2011.5947611.

Practical Issues with training RNNs

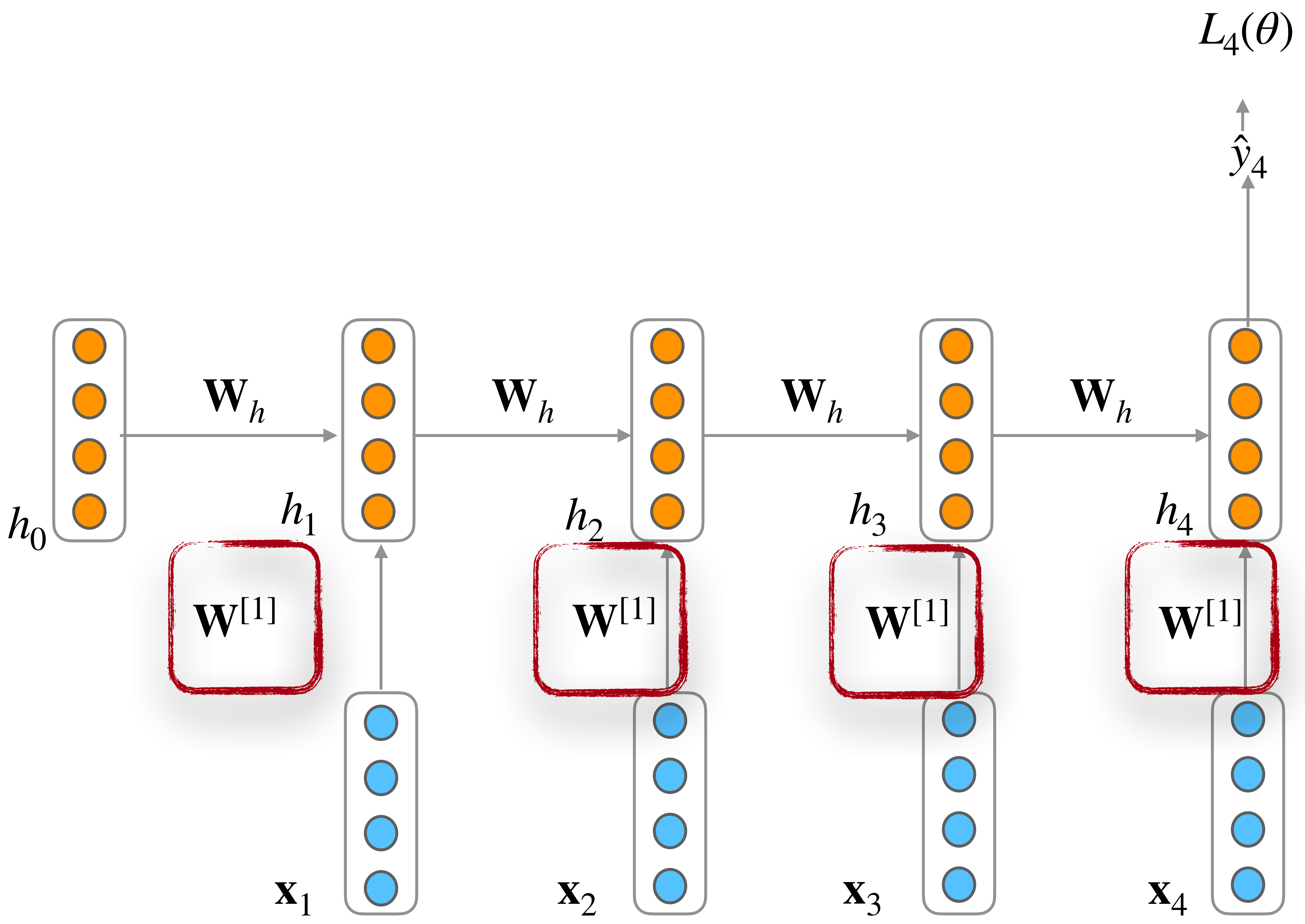
- Computing loss and gradients across entire corpus is too expensive!
- Recall: mini-batch Stochastic Gradient Descent allows us to compute loss and gradients for small chunk of data, and update.
- Solution: consider chunks of text.
  - In practice, consider $x_1, x_2, \ldots x_T$ for some T as a "sentence" or "single data instance"

$$L(\theta) = \frac{1}{T}\sum_{t=1}^{T} L_{CE}(\hat{y}_t, y_t)$$

- Compute loss for a sentence (actually usually a batch of sentences), compute gradients and update weights. Repeat.

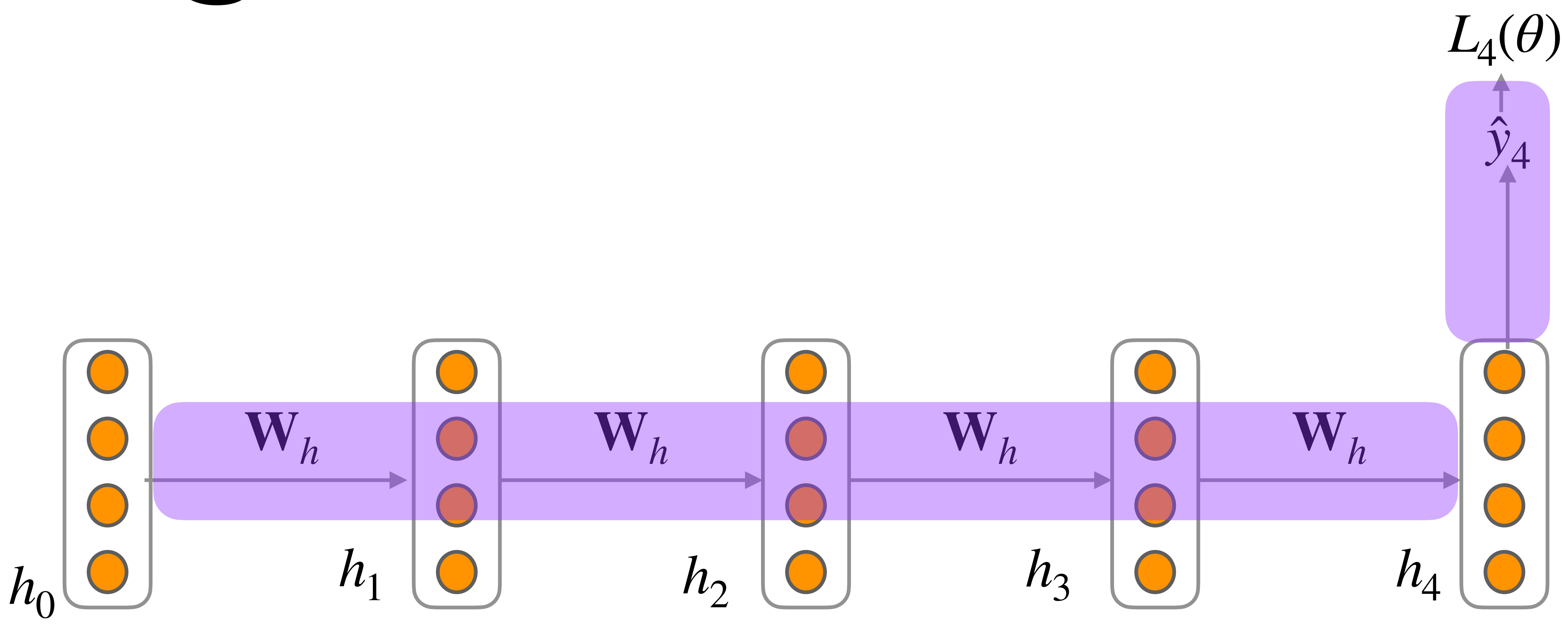
Training RNNs is hard

- Multiply the same matrix at each time step during forward propagation
- Ideally inputs from many time steps ago can modify output y
- This leads to something called the **vanishing gradient problem**



The Vanishing Gradient Problem and LSTMs

The Vanishing Gradient Problem: Intuition



When these gradients are small, the gradient signal gets smaller and smaller as it backpropagates further…

$$\frac{\partial L_4}{\partial h_0} = \frac{\partial h_1}{\partial h_0} \times \frac{\partial L_4}{\partial h_1}$$

$$= \frac{\partial h_1}{\partial h_0} \times \frac{\partial h_2}{\partial h_1} \times \frac{\partial L_4}{\partial h_2}$$

$$= \frac{\partial h_1}{\partial h_0} \times \frac{\partial h_2}{\partial h_1} \times \frac{\partial h_3}{\partial h_2} \times \frac{\partial L_4}{\partial h_3}$$

$$= \frac{\partial h_1}{\partial h_0} \times \frac{\partial h_2}{\partial h_1} \times \frac{\partial h_3}{\partial h_2} \times \frac{\partial h_4}{\partial h_3} \times \frac{\partial L_4}{\partial h_4}$$

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

The Vanishing Gradient Problem: Effects

- In practice, no long-term / long-range effects, contrary to the RNN promise
- Example language modeling task
  - To learn from this training example, the RNN-LM needs to model the dependency between "tickets" on the 7th step and the target word "tickets" at the end
- But if the gradient is small, the model can't learn this dependency
  - So, the model is unable to predict using similar long-distance dependencies at test time
- In practice a simple RNN will only condition ~7 tokens back [vague rule-of-thumb]

When she tried to print her tickets, she found that the printer was out of toner. She went to the stationery store to buy more toner. It was very overpriced. After installing the toner into the printer, she finally printed her ________

USC Viterbi

The Vanishing Gradient Problem: Fixes

- The main problem is that it is too difficult for the RNN to learn to preserve information over many timesteps
- In a vanilla RNN, the hidden state is constantly being rewritten

$$\mathbf{h}_t = reLU(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}^{[1]} \mathbf{x}_t)$$

New design: equip an RNN with separate memory which is added to

Solution? Think data structures…

USC Viterbi

Long Short-Term Memory RNNs (LSTMs)

- At time step t, introduces a new cell state $\mathbf{c}_t \in \mathbb{R}^d$
    - In addition to a hidden state $\mathbf{h}_t \in \mathbb{R}^d$
    - The cell stores long-term information (memory)
    - The LSTM can read, erase, and write information from the cell!
        - The cell becomes conceptually rather like RAM in a computer

- The selection of which information is erased/written/read is controlled by three corresponding gates:
    - Input gate $\mathbf{i}_t \in \mathbb{R}^d$, Output gate $\mathbf{o}_t \in \mathbb{R}^d$ and Forget gate $\mathbf{f}_t \in \mathbb{R}^d$
    - Each *element* of the gates can be open (1), closed (0), or somewhere in between
    - The gates are dynamic: their value is computed based on the current context

USC Viterbi

LSTMs

Given a sequence of inputs x_t , we will compute a sequence of hidden states h_t and cell states c_t

At timestep t :

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

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

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

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

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

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

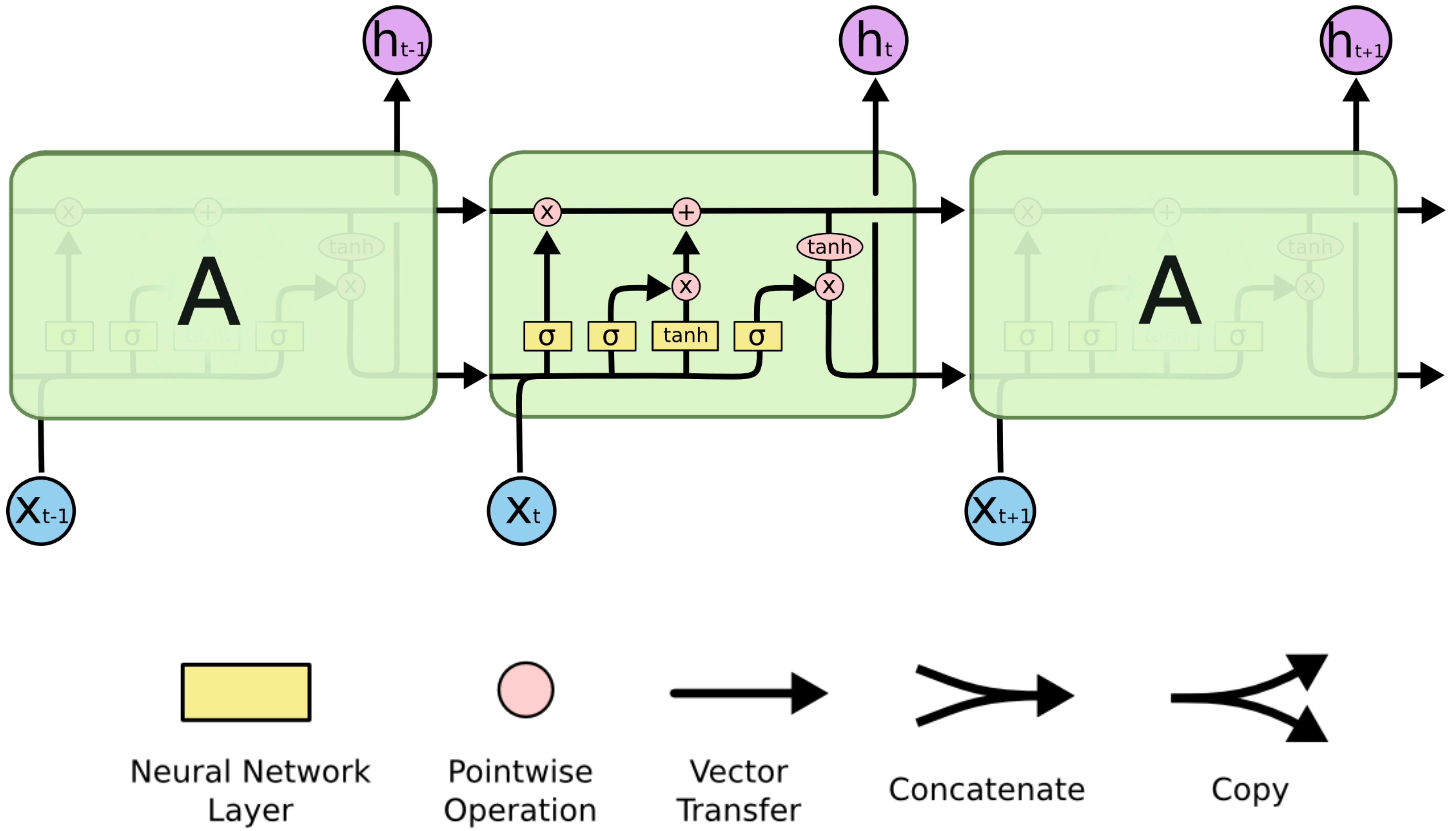
$$\boldsymbol{c}^{(t)} = \boldsymbol{f}^{(t)} \circ \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \circ \tilde{\boldsymbol{c}}^{(t)}$$

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

All these are vectors of same length *n*

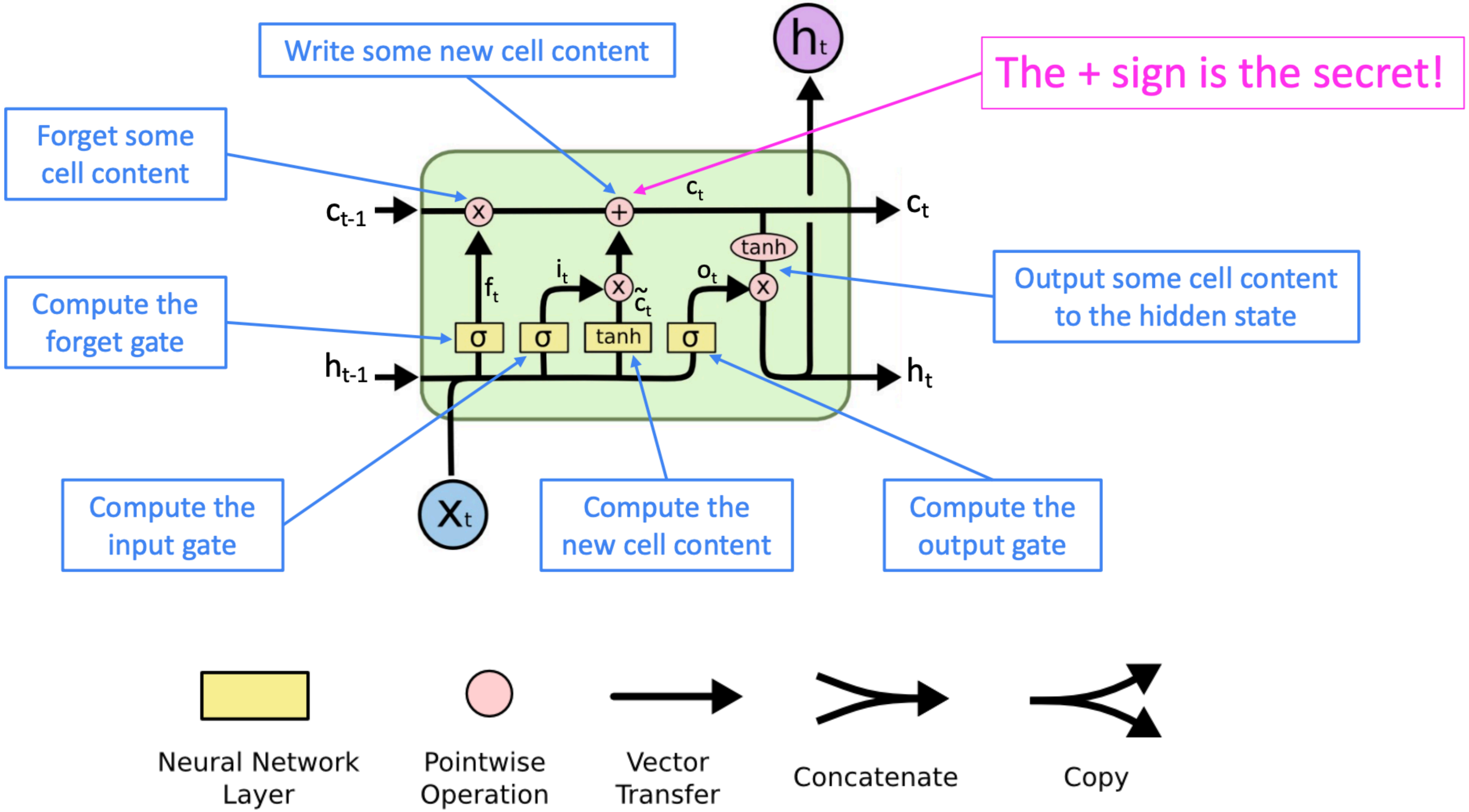
Gates are applied using element-wise (or Hadamard) product: $\odot$

LSTMs: A Visual Representation



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

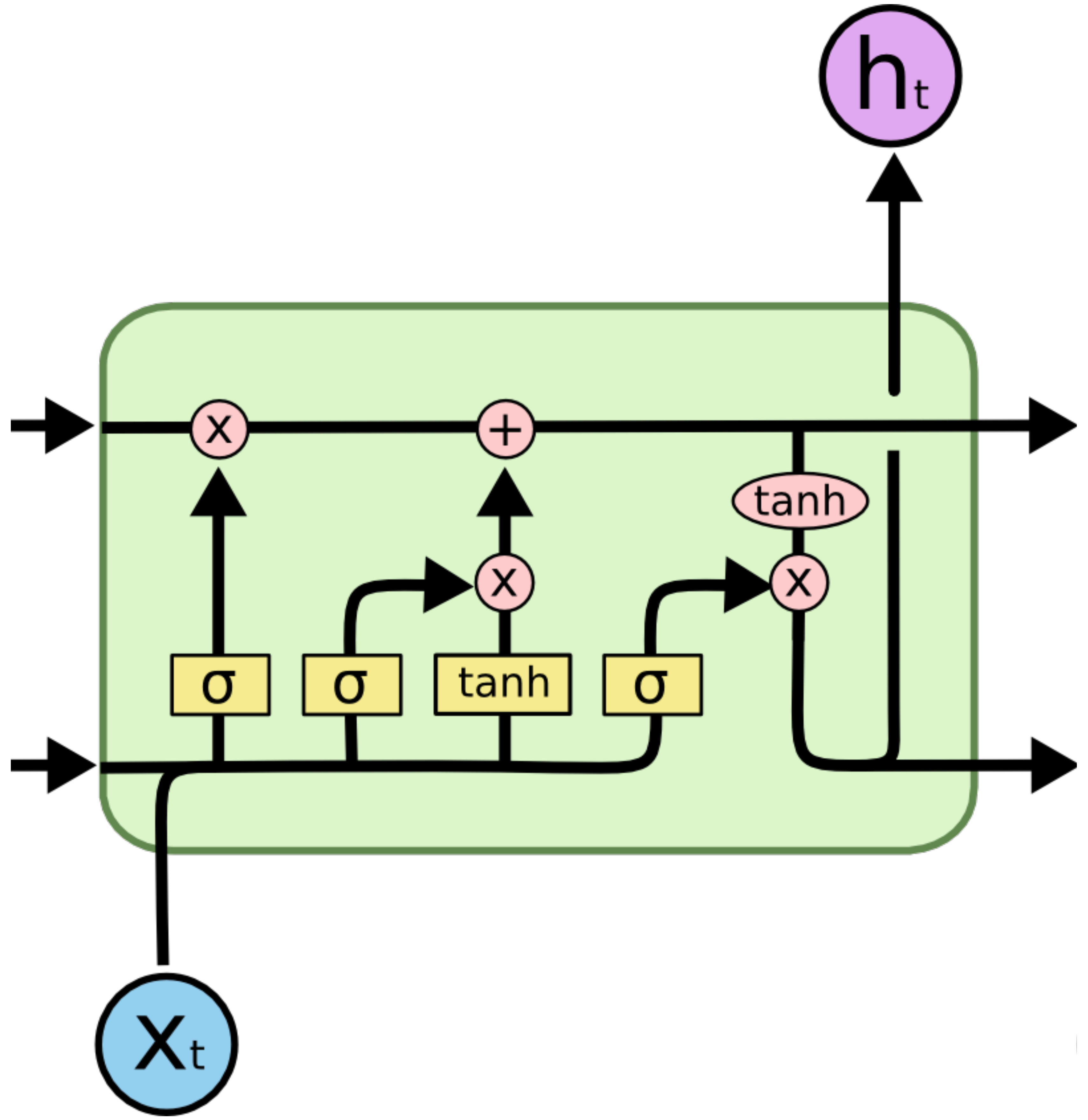
LSTMs: A Visual Representation



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

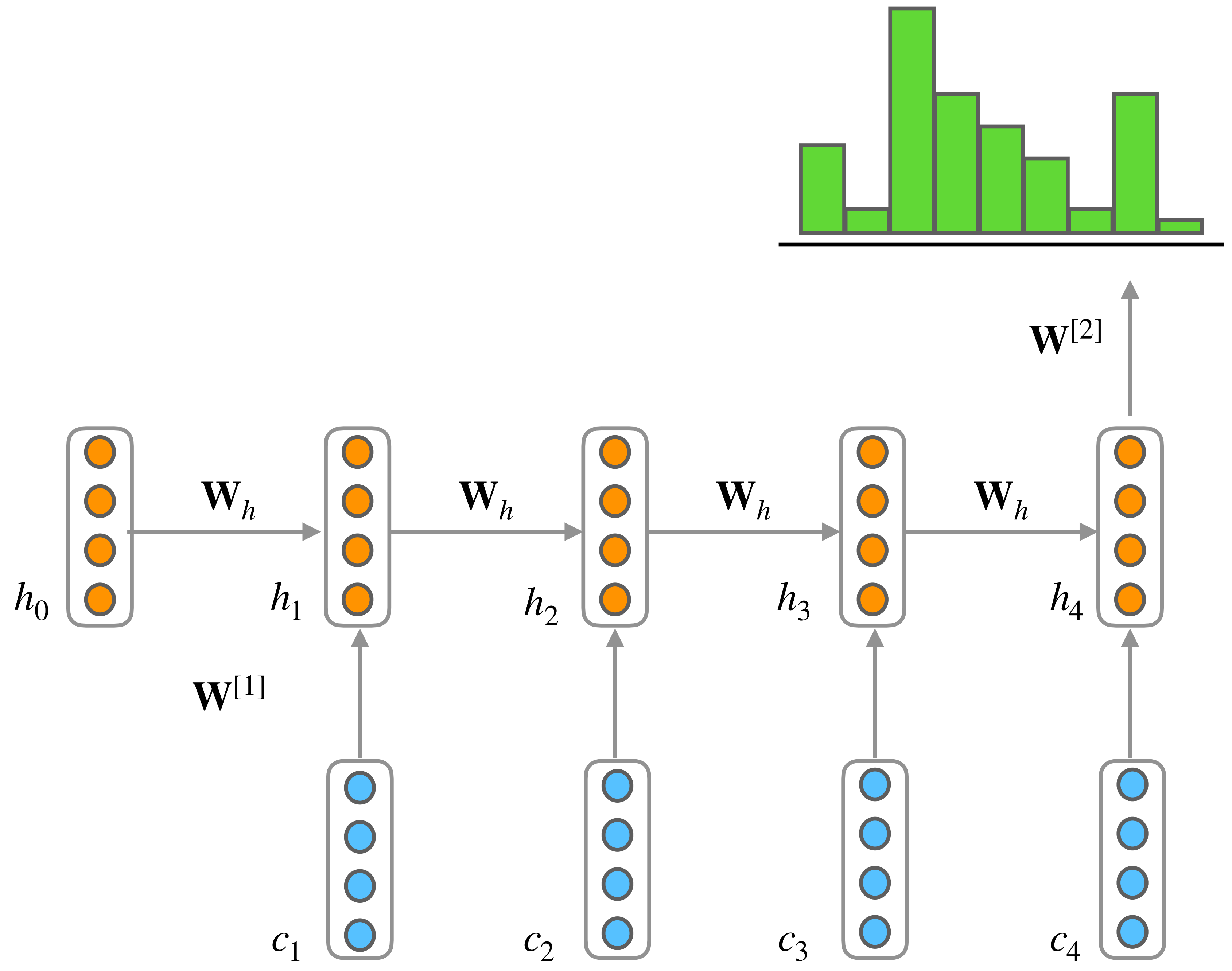
LSTMs: Summary

- The LSTM architecture makes it much easier for an RNN to preserve information over many timesteps
  - e.g., if the forget gate is set to 1 for a cell dimension and the input gate set to 0, then the information of that cell is preserved indefinitely
- In 2013–2015, LSTMs started achieving state-of-the-art results
  - Successful tasks include handwriting recognition, speech recognition, machine translation, parsing, and image captioning, as well as language models
  - LSTMs became the dominant approach for most NLP tasks
  - We'll look into machine translation next!



Summarizing RNNs

- Recurrent Neural Networks processes sequences one element at a time
- RNNs do not have
  - the limited context problem of n-gram models
  - the fixed context limitation of feedforward LMs
  - since the hidden state can *in principle* represent information about all of the preceding words all the way back to the beginning of the sequence
- But training RNNs is hard
  - Vanishing gradient problem
  - LSTMs address it by incorporating a memory



Next Class: Transformer Language Models!