



# Lecture 9: Recurrent Neural Nets

*Instructor: Swabha Swayamdipta*  
*USC CSCI 544 Applied NLP*  
*Sep 24, Fall 2024*



# Announcements

- This Thu: Project Proposal Due
  - Once you propose an idea, you're NOT allowed to completely change it
  - Allowed to make modifications, based on our recommendations
- Quiz 2 grades are released
- Deadlines from now on: 11:59 PM PT
  - for assignments and project deliverables
  - NOT project final report
- Next week: Quiz 3
- Based on your requests: HW2 deadline is now Oct 8
- Quiz 1: No more regrade requests

# Lecture Outline

- Quiz 2 Answers
- Recap: Feedforward Neural Nets and Backprop
- Recurrent Neural Nets
  - Language Models
- Training RNNLMs
- The Vanishing Gradient Problem
- LSTMs
- Applications of RNNs

# Quiz 2 Answers (Redacted)

# Recap: Feedforward Neural Nets

# Feedforward Nets: Loss Function

- 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  $y_c = 1$  if the correct class is  $c$ , with all other elements of  $\mathbf{y}$  being 0
- Classifier will produce an estimate vector  $\hat{\mathbf{y}}$ , each element represents estimated probability,  
 $\hat{y}_k = p_{\theta}(y_k = 1 | x)$

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

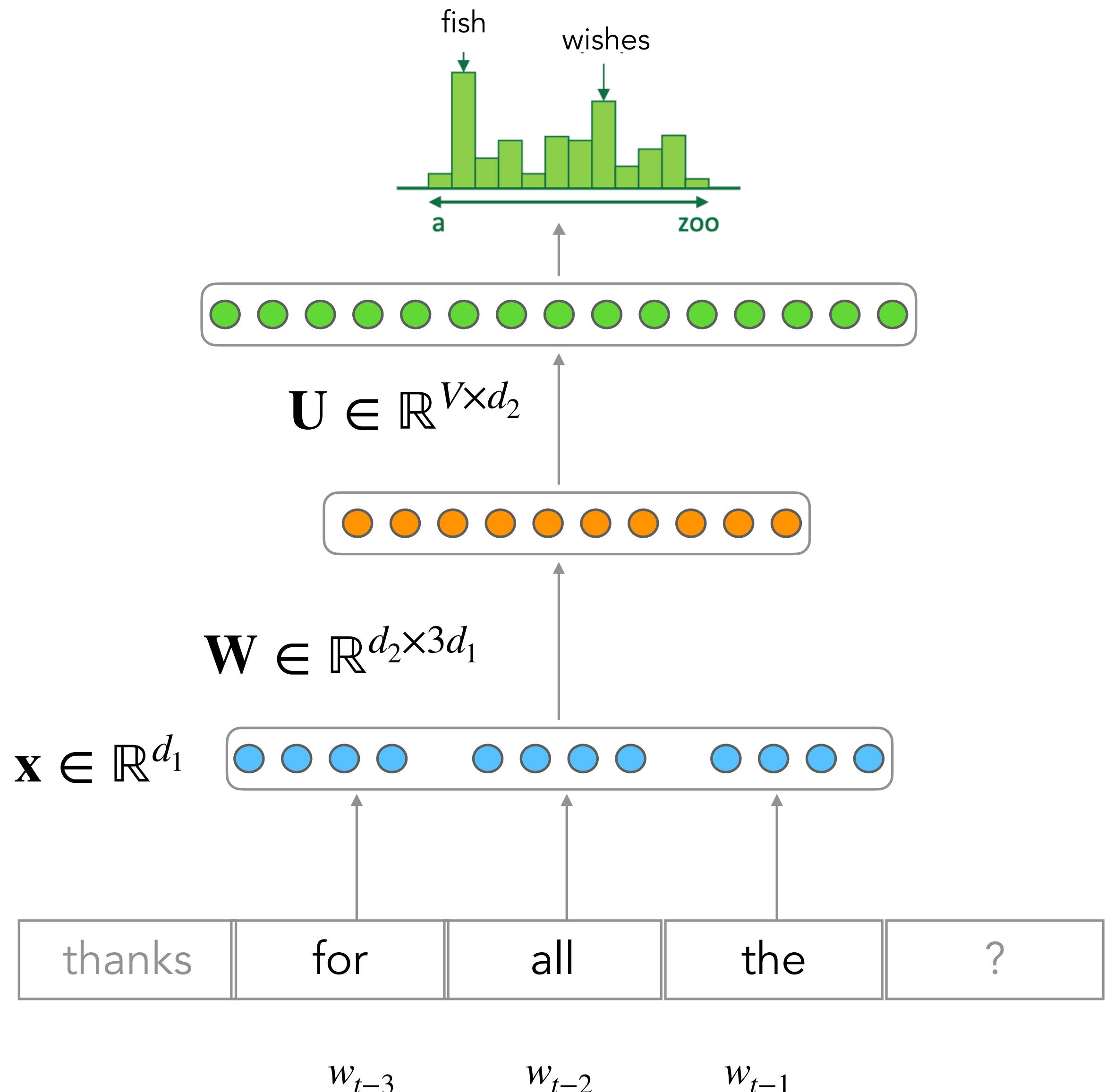
What is  $K$  for language modeling?

Hard Classification

$$\begin{aligned} L_{CE}(y, \hat{y}) &= - \sum_{k=1}^K y_k \log \hat{y}_k &= - \log \hat{y}_c, c \text{ being the correct class} \\ &&= - \log \frac{\exp(z_c)}{\sum_{j=1}^K \exp(z_j)}, c \text{ being the correct class} \end{aligned}$$

# Feedforward LMs: Windows

- Word embeddings  $\mathbf{x}$  could either be looked up (from a dictionary of pretrained embeddings) or learned from scratch (usually with a random initialization)
- The goodness of the language model depends on the size of the sliding window
  - Fixed window can be too small
  - Enlarging window enlarges  $\mathbf{W}$
  - Windows can never be large enough!
- Each word uses different rows of  $\mathbf{W}$ . We don't share weights *within* the window.



# From LR to FFNN: Gradients

Cross Entropy Loss for LR

$$\begin{aligned} L_{CE}(y, \hat{y}) &= -\log p(y|x) = -[y \log \hat{y} + (1-y) \log(1-\hat{y})] \\ &= -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1-y) \log(\sigma(-\mathbf{w} \cdot \mathbf{x} + b))] \end{aligned}$$

Gradient Update for LR

$$\frac{\partial L_{CE}(\hat{y}, y)}{\partial w_j} = [\sigma(\mathbf{w} \cdot \mathbf{x} + b) - y]x_j$$

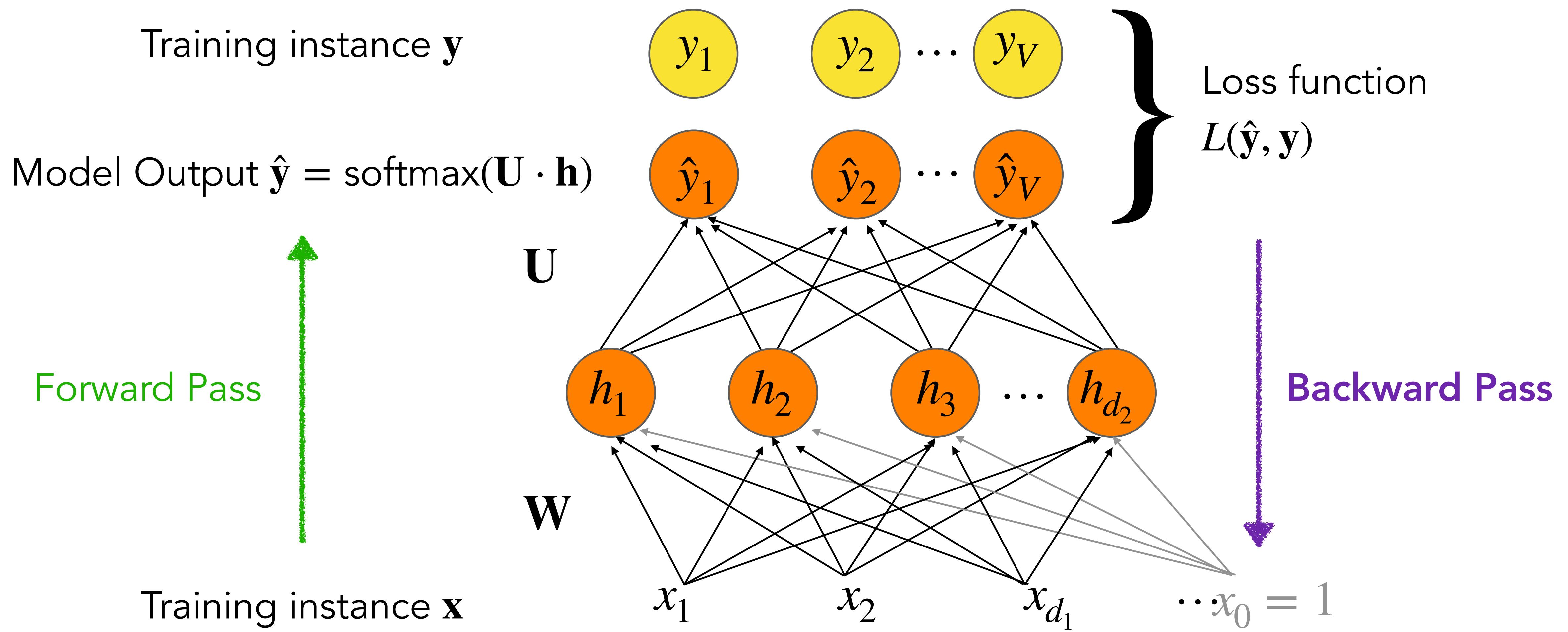
Only one parameter!

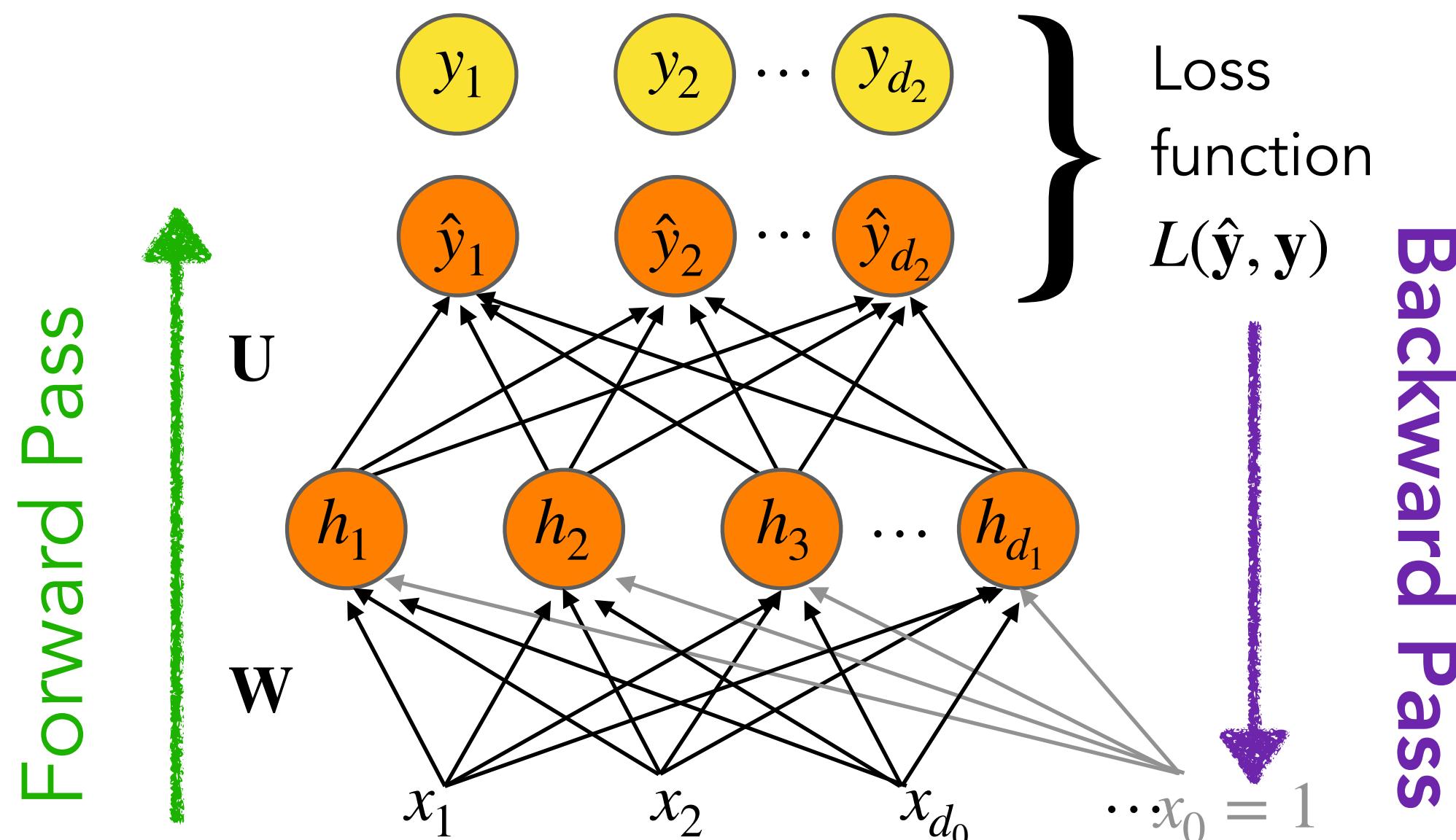
computation graphs

Typically, consider the Jacobian:  $J_L = \nabla_{\mathbf{w}} L$

As (multiple) hidden layers are introduced, there will be many more parameters to consider, not to mention activation functions!

# Training a 2-layer Network





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  $U_{ij}$  from hidden layer to the output layer
    - Update the weight by computing gradient  $\frac{\partial L}{\partial U_{ij}}$
  - Hidden layer: For every weight  $W_{ij}$  from input layer to the hidden layer
    - Update the weight by computing gradient  $\frac{\partial L}{\partial W_{ij}}$

# Computation Graphs

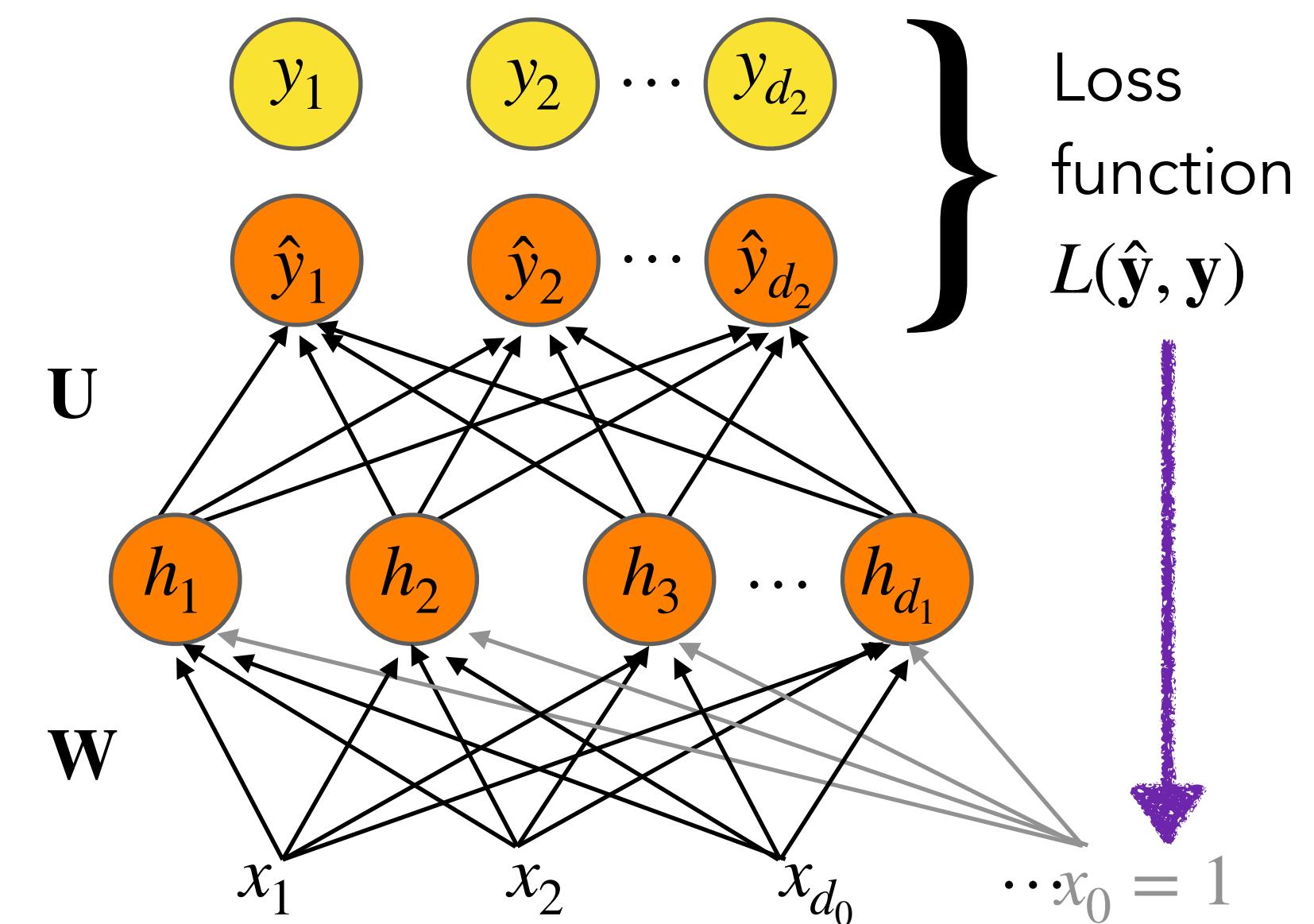
Graph representing the process of computing a mathematical expression

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



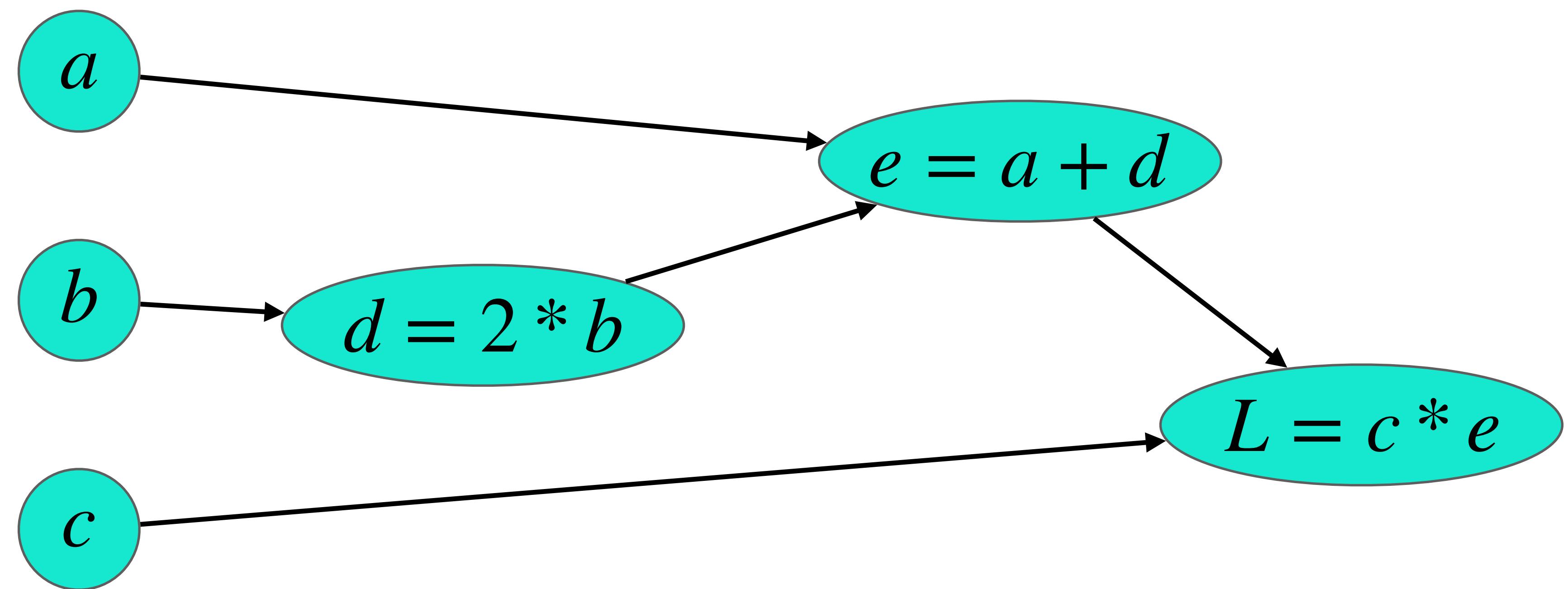
Backprop

# Example: Computation Graph

$$d = 2 * b$$

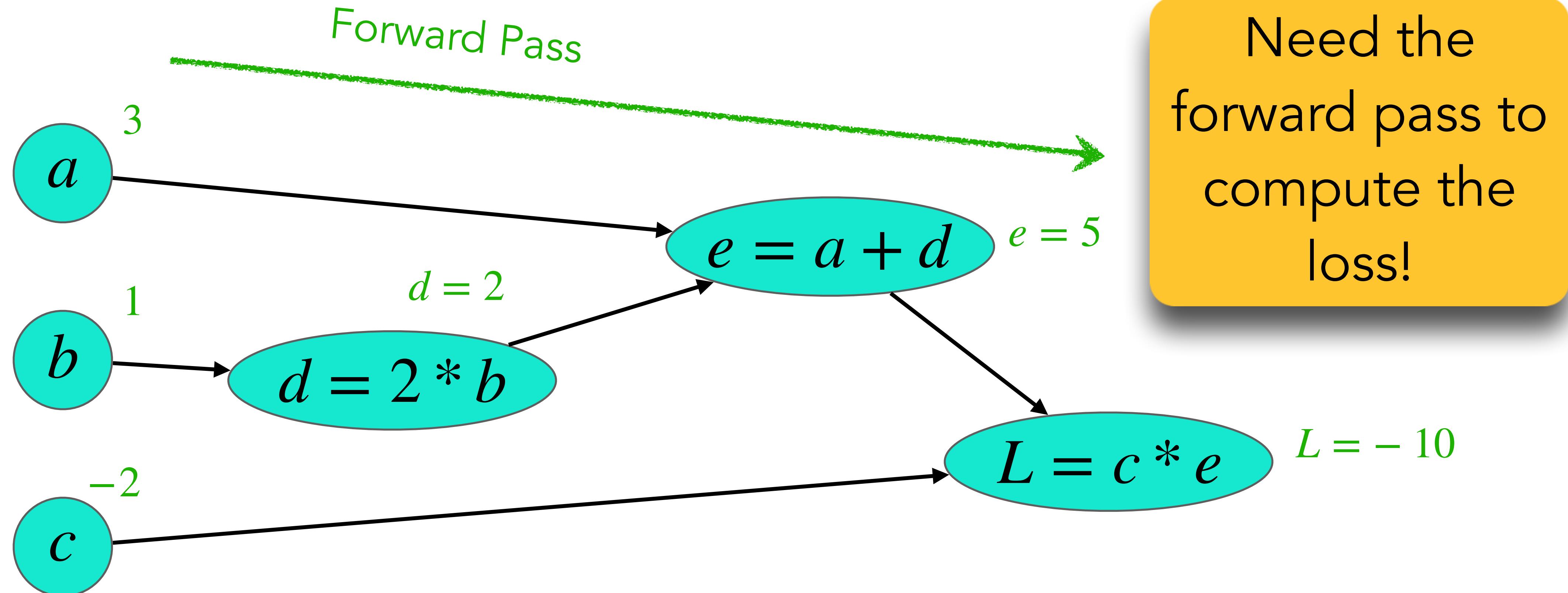
$$e = a + d$$

$$L = c * e$$



# Example: Forward Pass

$$\begin{aligned}d &= 2 * b \\e &= a + d \\L &= c * e\end{aligned}$$



# 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

$$\frac{\partial L}{\partial a} = ? \quad \frac{\partial L}{\partial b} = ? \quad \frac{\partial L}{\partial c} = ? \quad \left. \right\} \text{Input Layer Gradients}$$
$$\left\{ \begin{array}{ll} \frac{\partial L}{\partial d} = ? & \frac{\partial L}{\partial e} = ? \\ \text{Hidden Layer Gradients} & \end{array} \right.$$

Chain Rule of Differentiation!

# 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...

# Example: Backward Pass

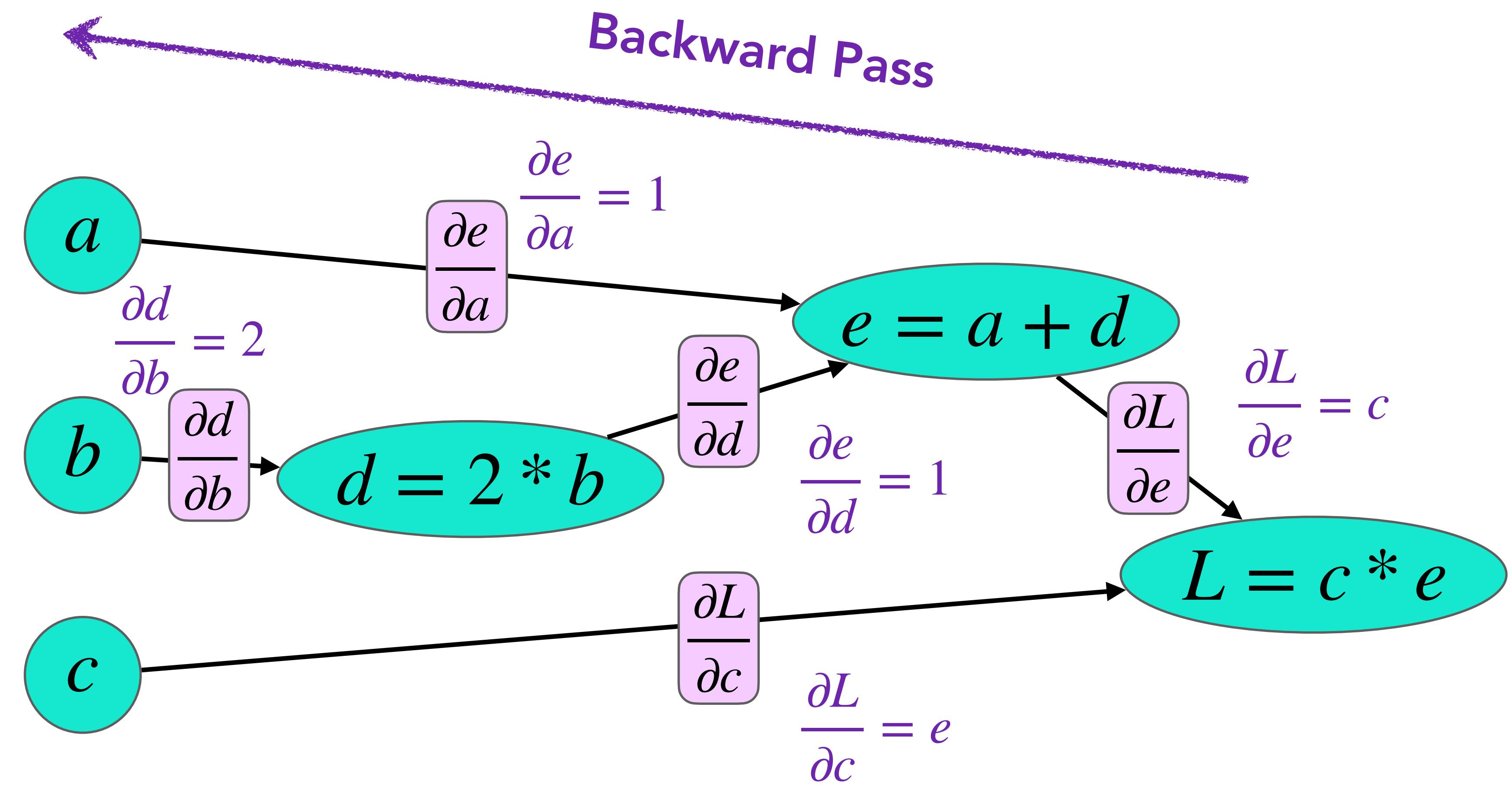
But we need the gradients of the loss with respect to parameters...

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

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

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

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



# Example

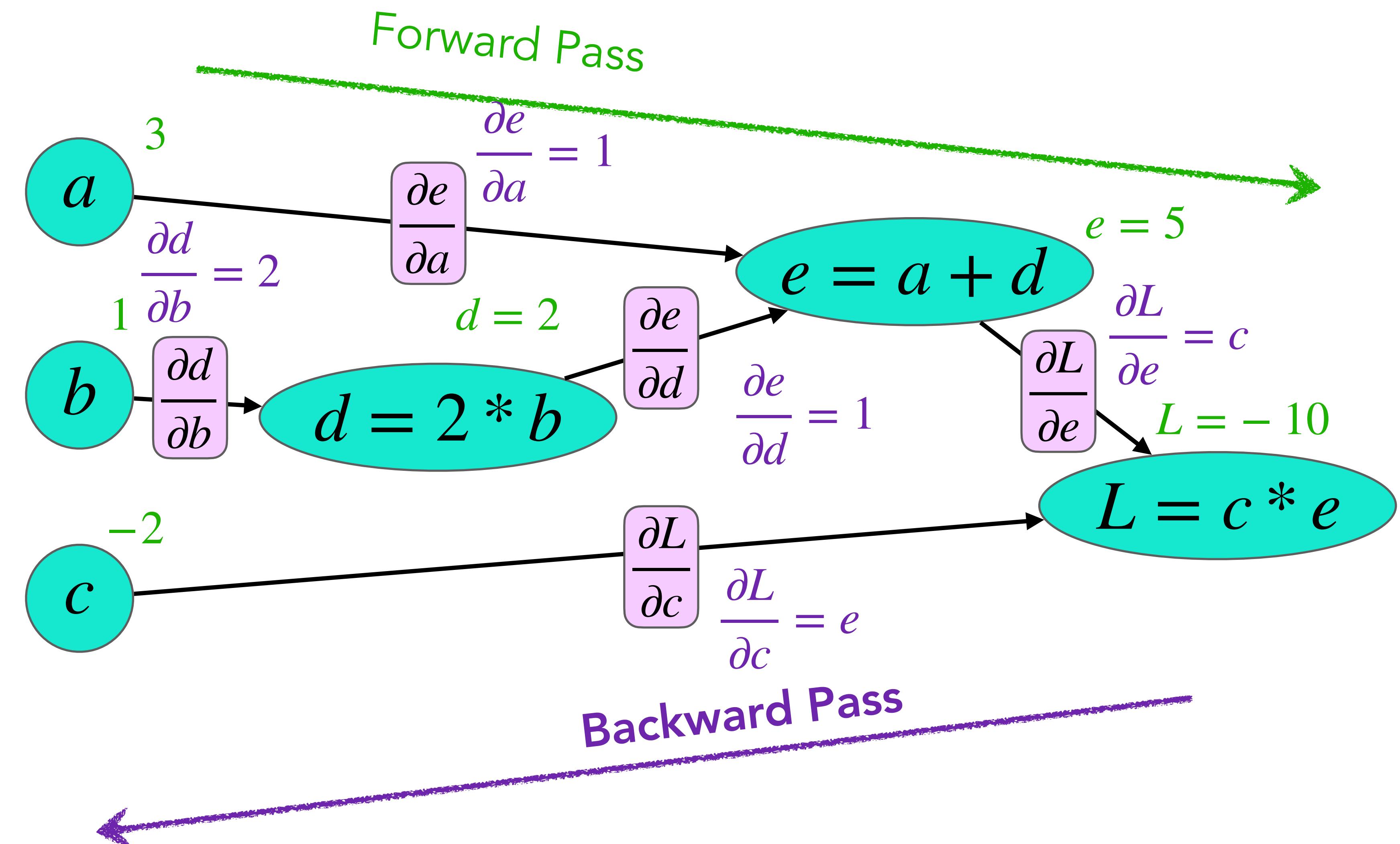
$$\frac{\partial L}{\partial e} = c = -2$$

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

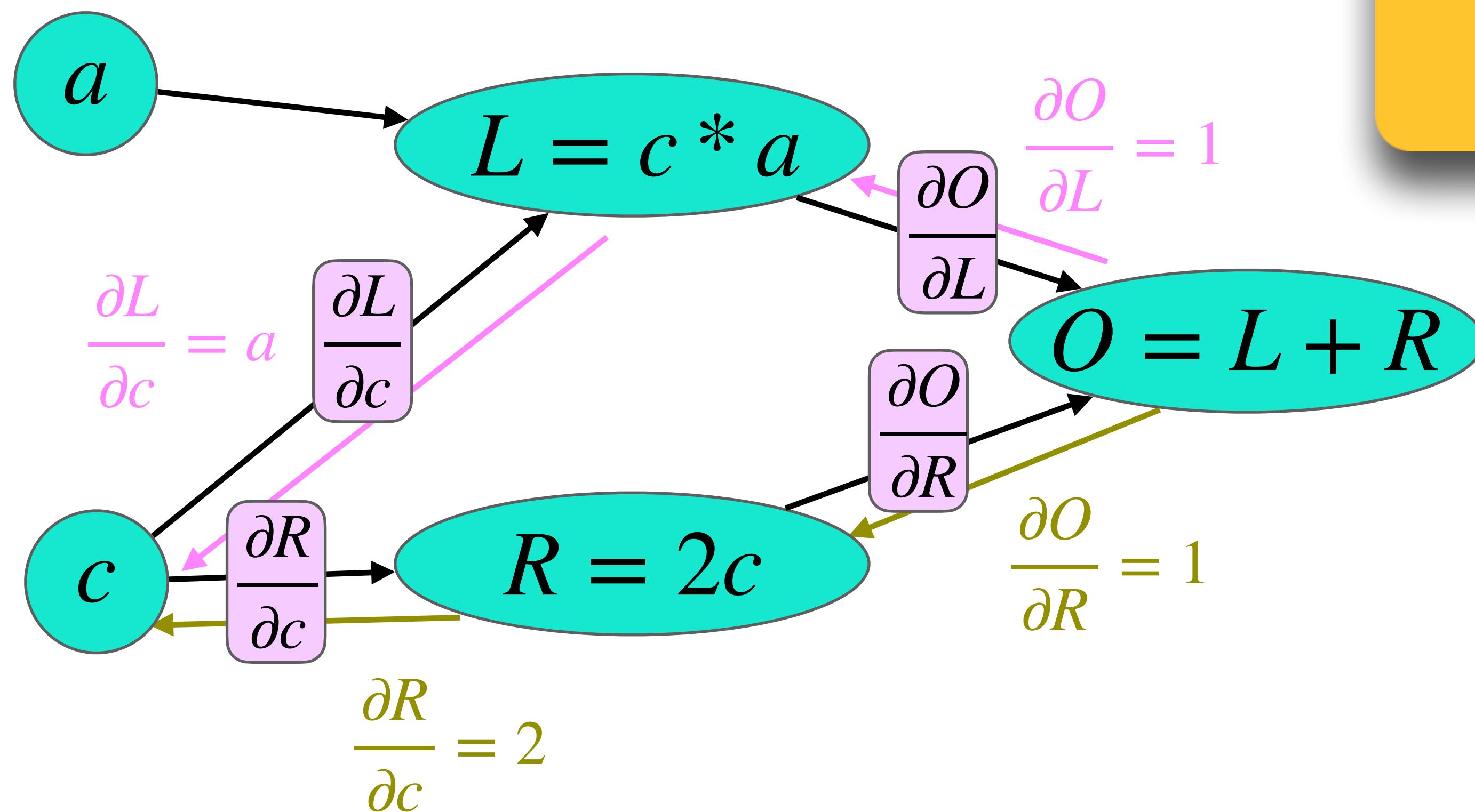
$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e} \frac{\partial e}{\partial a} = -2$$

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

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



# Example: Two Paths



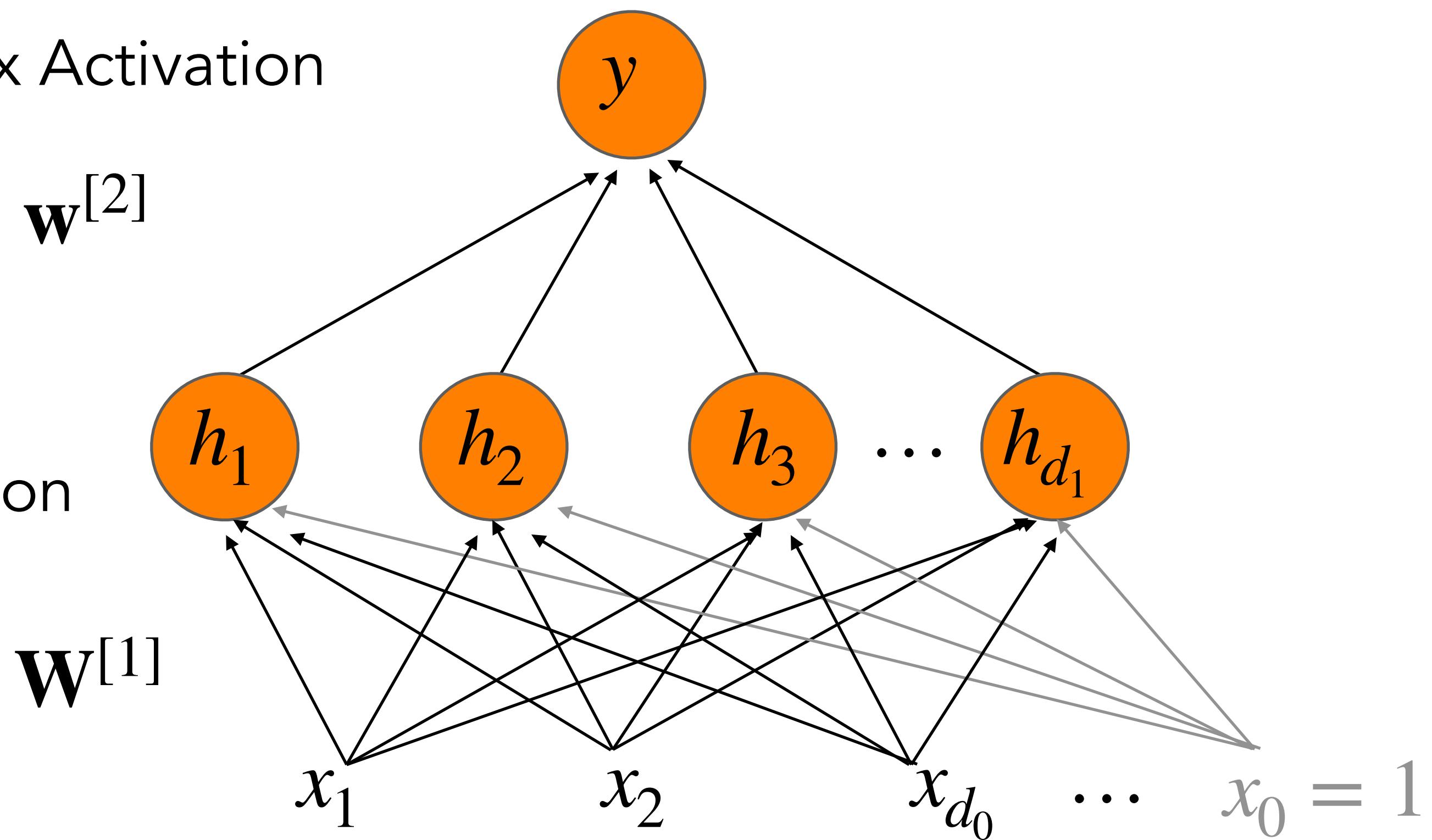
When multiple branches converge on a single node we will add these branches

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

Such cases arise when considering regularized loss functions

# Backward Differentiation on a 2-layer MLP

Softmax Activation



ReLU  
Activation

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

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

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

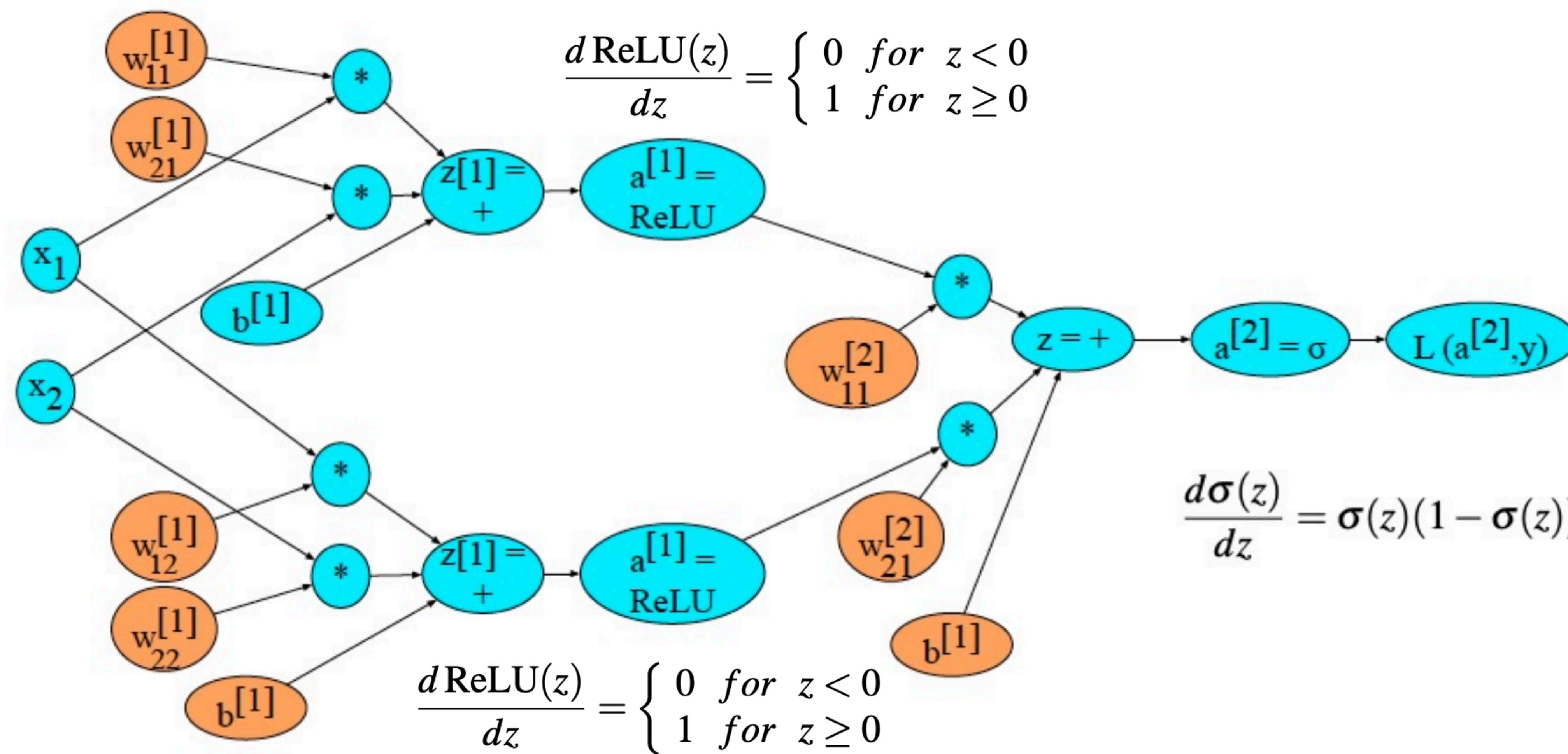
Element-wise

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

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

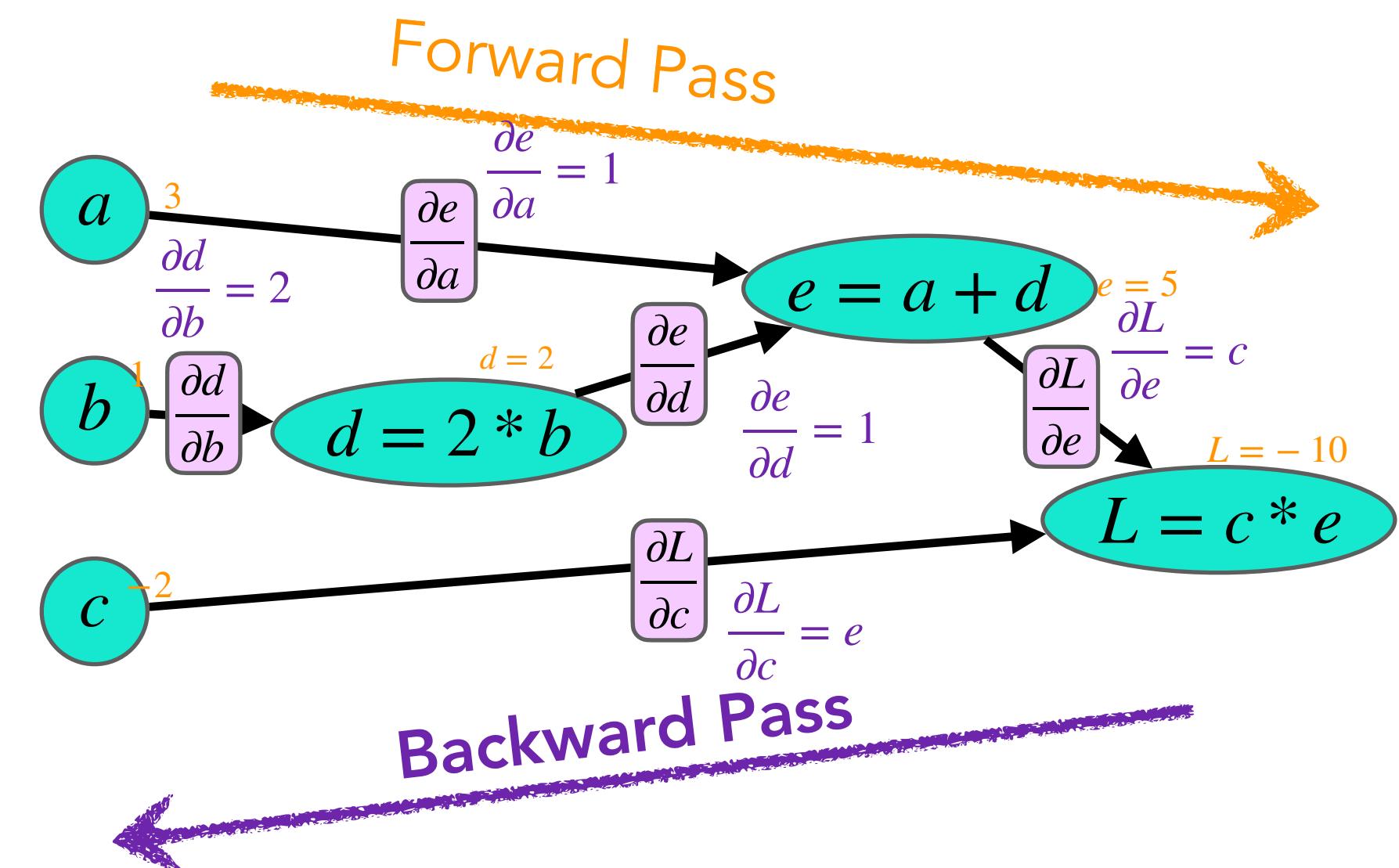
$$\frac{d \text{ReLU}(z)}{dz} = \begin{cases} 0 & \text{for } z < 0 \\ 1 & \text{for } z \geq 0 \end{cases}$$

# 2 layer MLP with 2 input features



# 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**
- Backprop is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient
  - Storing repeated subexpressions, employing recursion



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

# 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

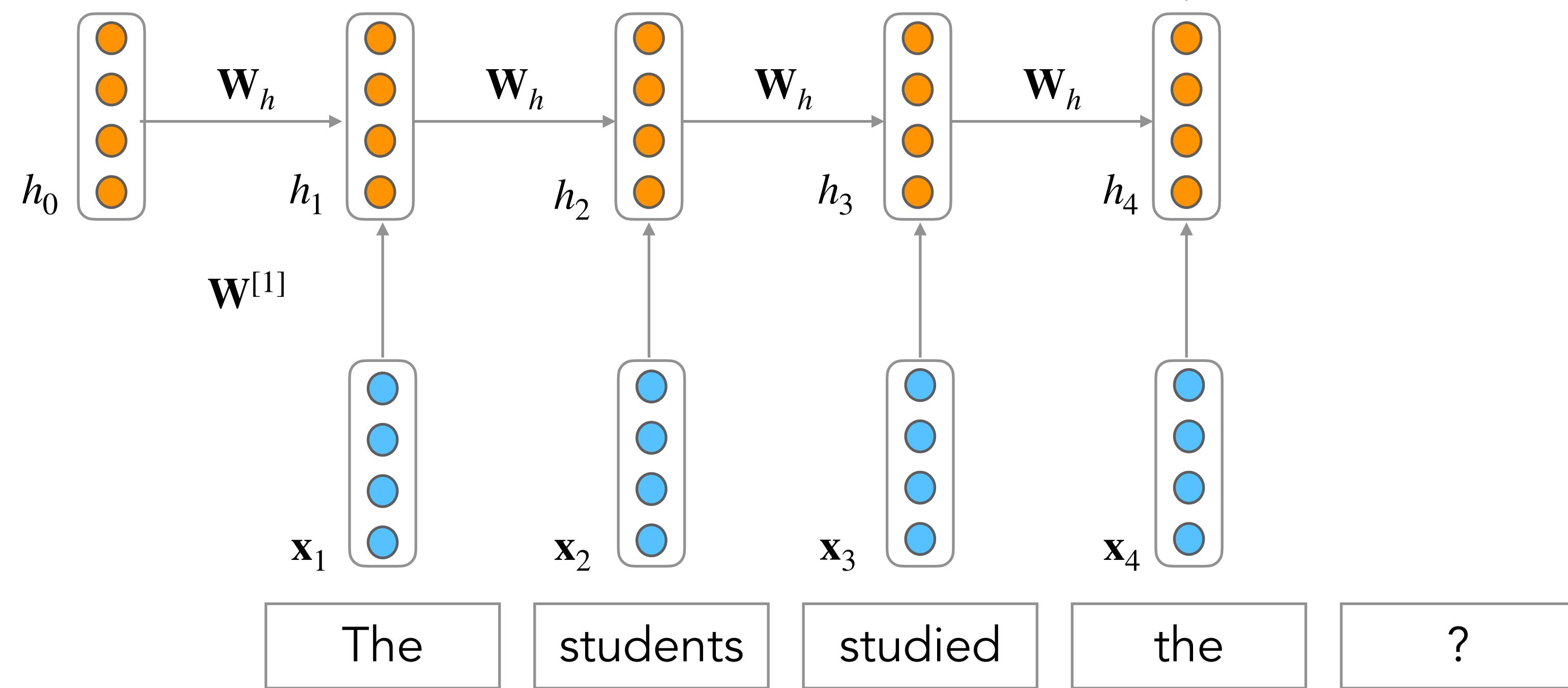
Output layer:  $\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{W}^{[2]}\mathbf{h}_t)$

Hidden layer:

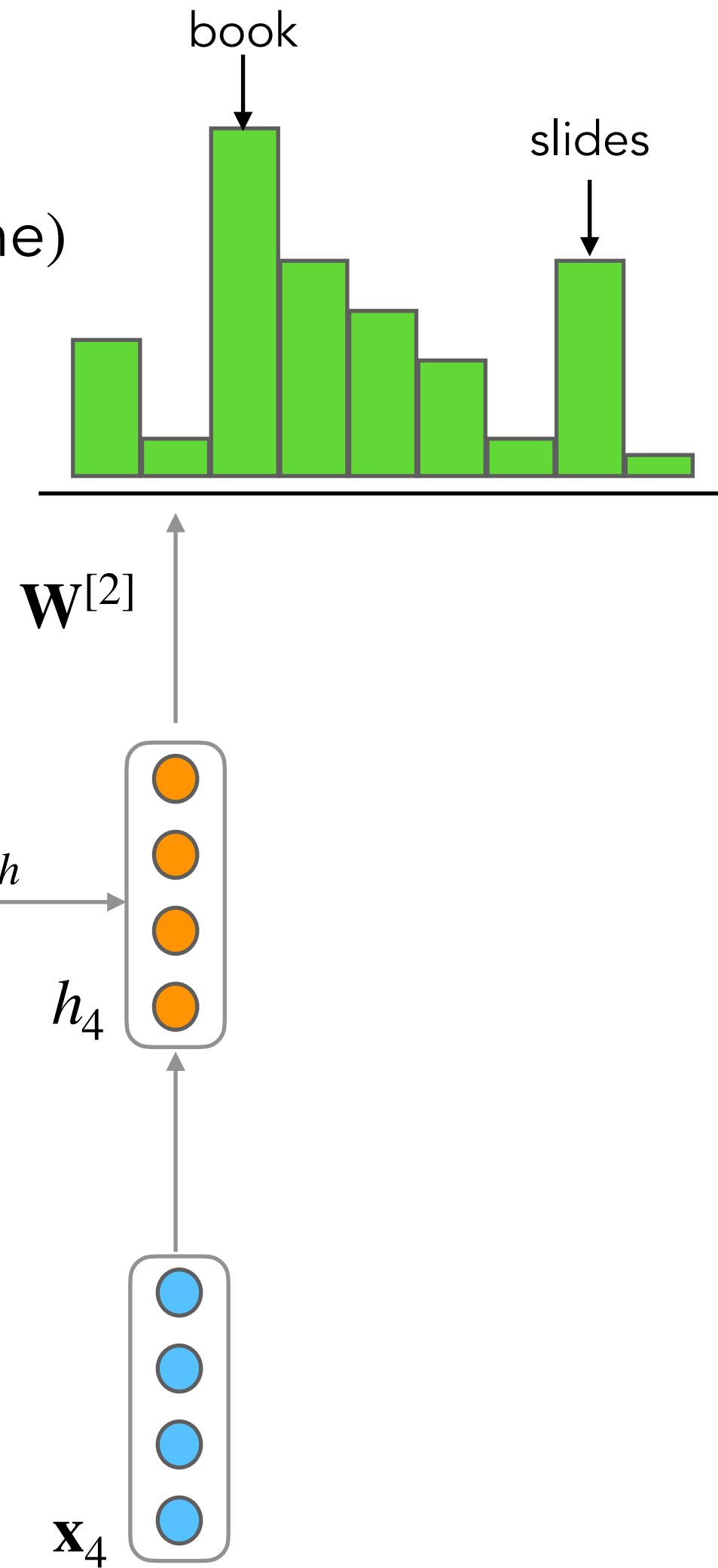
$$\mathbf{h}_t = g(\mathbf{W}_h\mathbf{h}_{t-1} + \mathbf{W}^{[1]}\mathbf{x}_t)$$

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

Word Embeddings,  $\mathbf{x}_i$



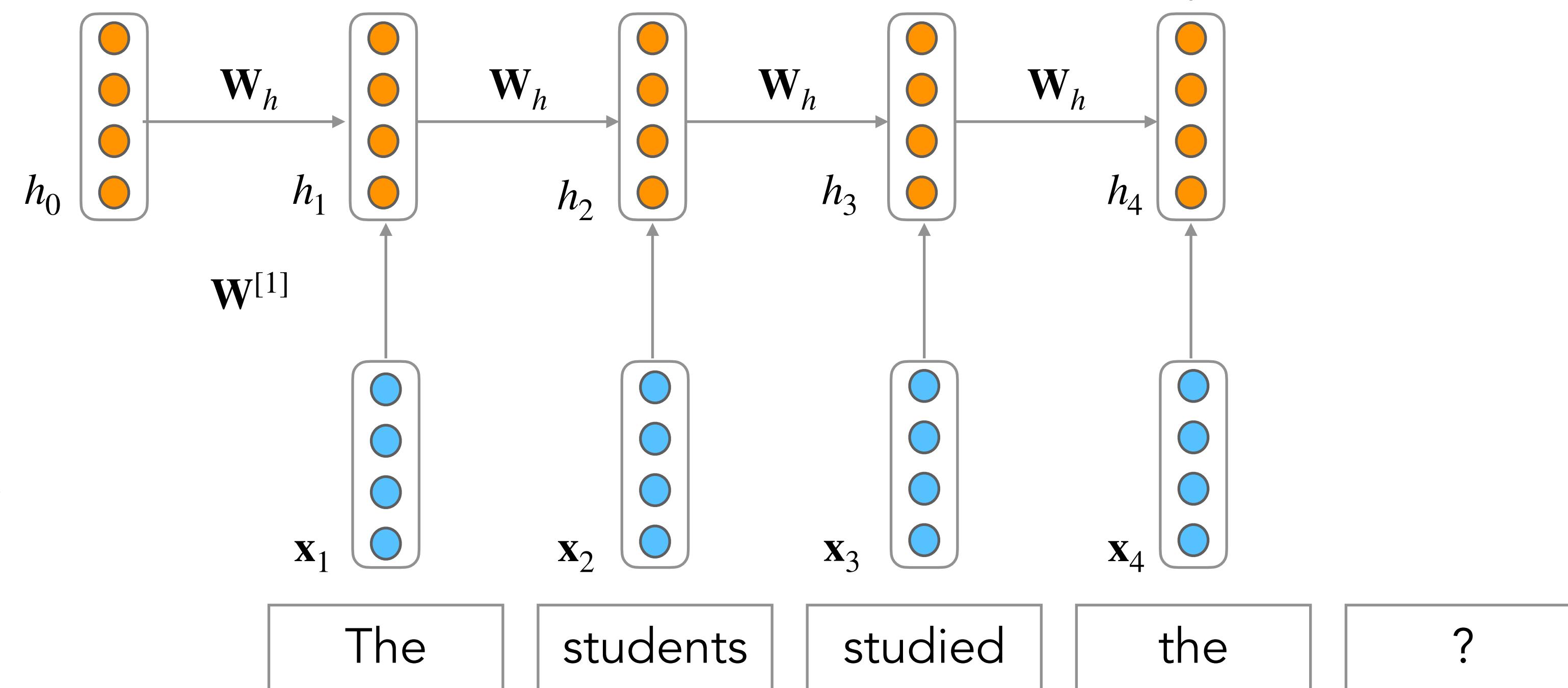
$$\hat{y}_4 = P(x_5 | \text{The students studied the})$$



# Why RNNs?

## RNN Advantages:

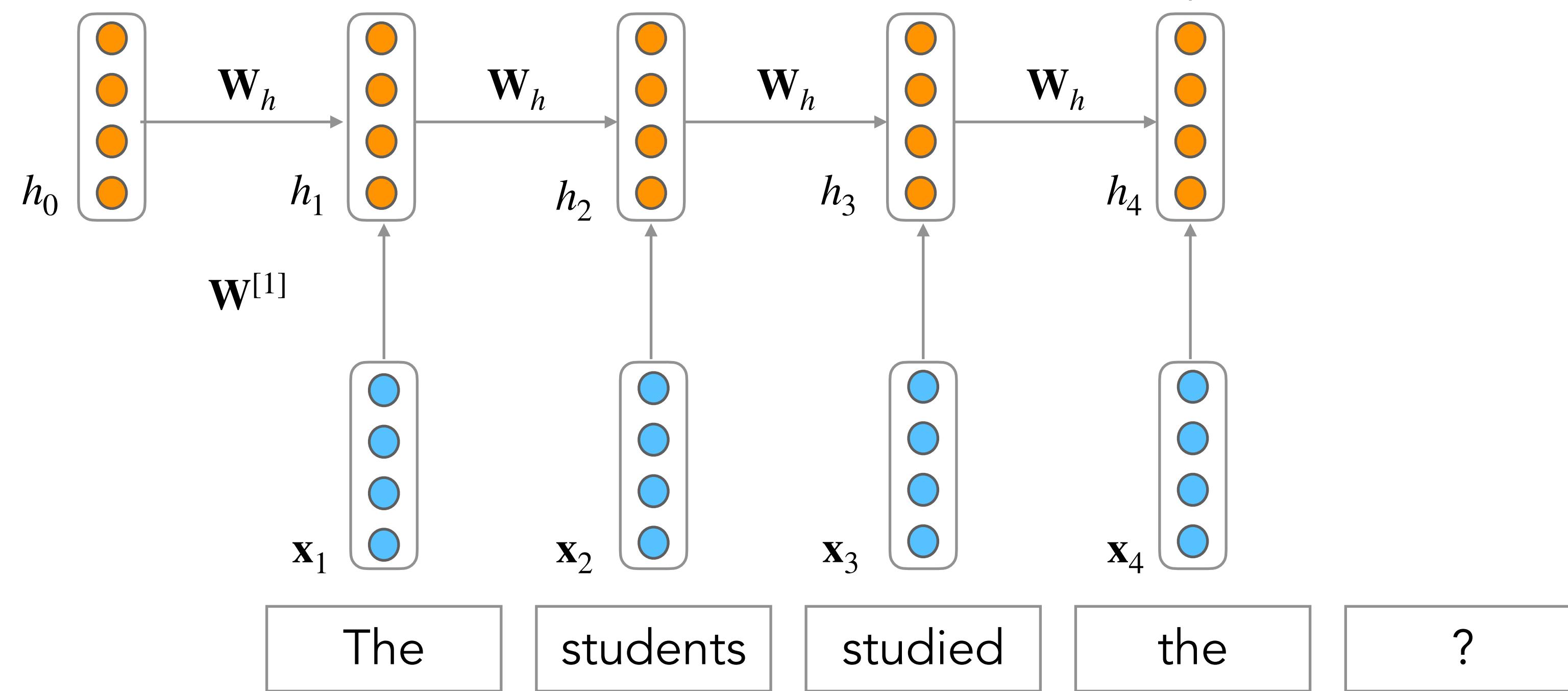
- Can process any length input
- Model size doesn't increase for longer input
- Computation for step  $t$  can (in theory) use information from many steps back
- Weights  $\mathbf{W}^{[1]}$  are shared (tied) across timesteps → Condition the neural network on all previous words



# Why not RNNs?

RNN Disadvantages:

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



# Training RNNLMs

# Training Outline

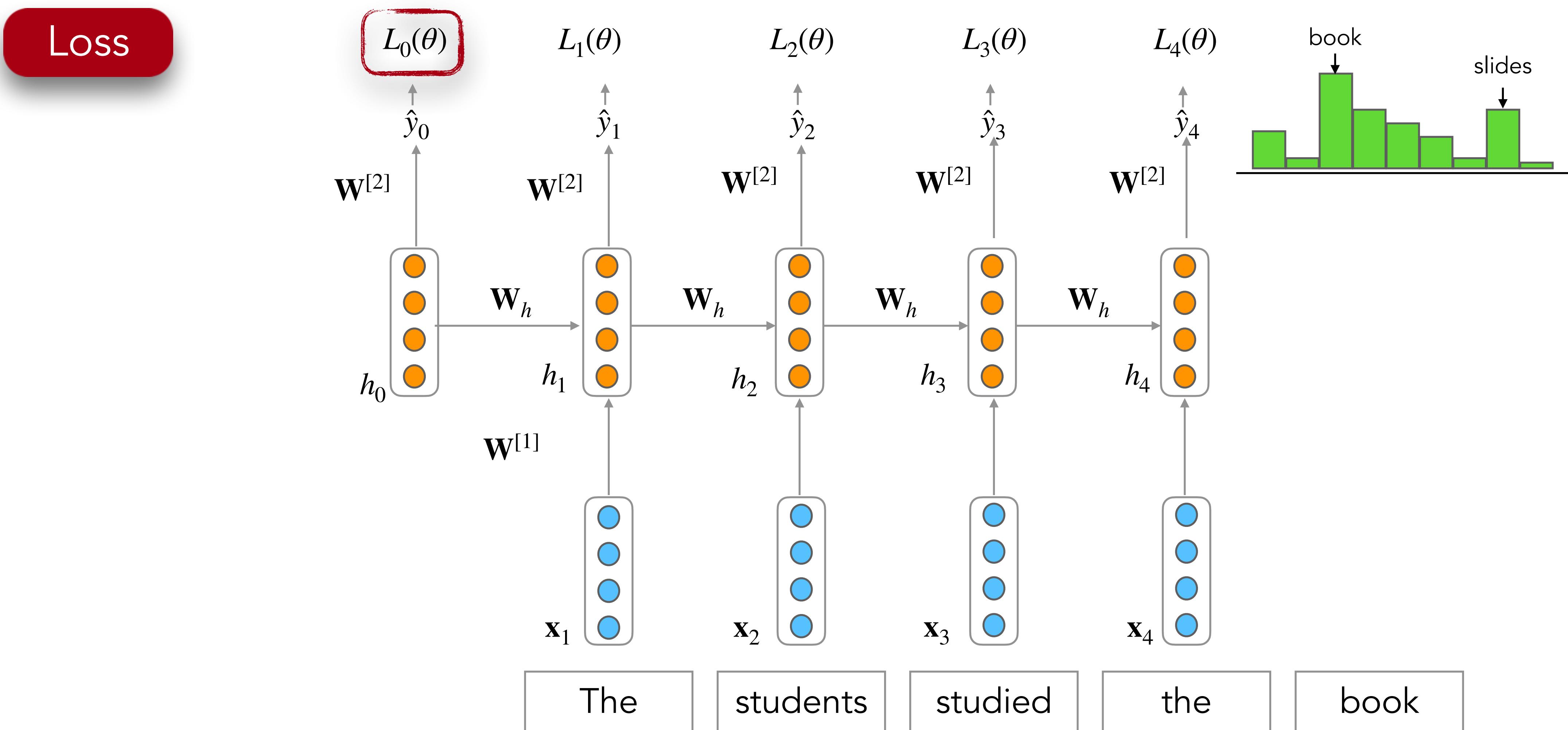
- Get a big corpus of text which is a sequence of words  $x_1, x_2, \dots, x_T$
- Feed into RNN-LM; compute output distribution  $\hat{y}_t$  for every step  $t$ 
  - i.e. predict probability distribution of every word, given words so far
- Loss function on step  $t$  is usual cross-entropy between our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

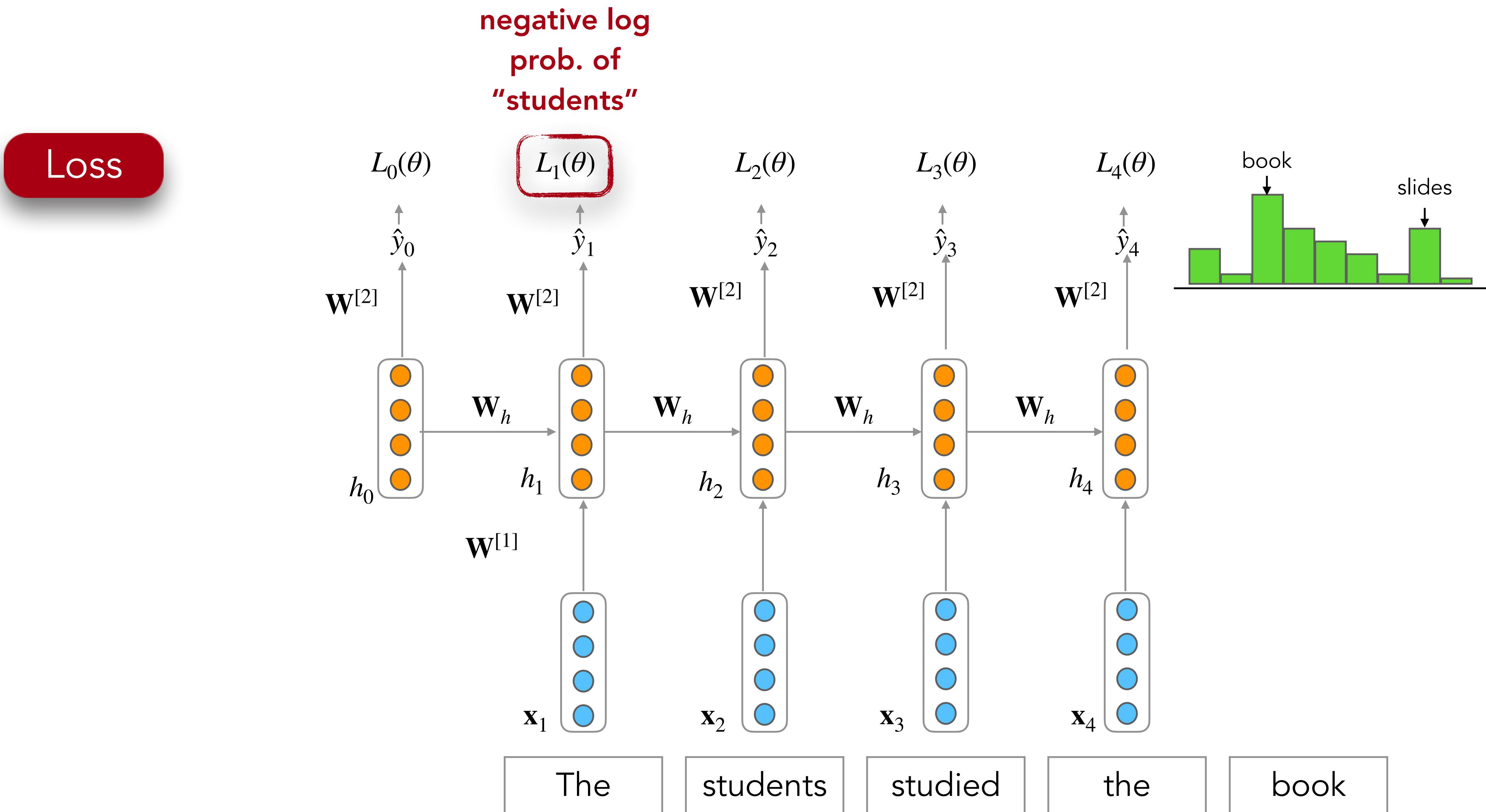
$$L_{CE}(\hat{y}_t, y_t; \theta) = - \sum_{v \in V} \mathbb{I}[y_t = v] \log \hat{y}_t = - \log p_\theta(x_{t+1} | x_{\leq t})$$

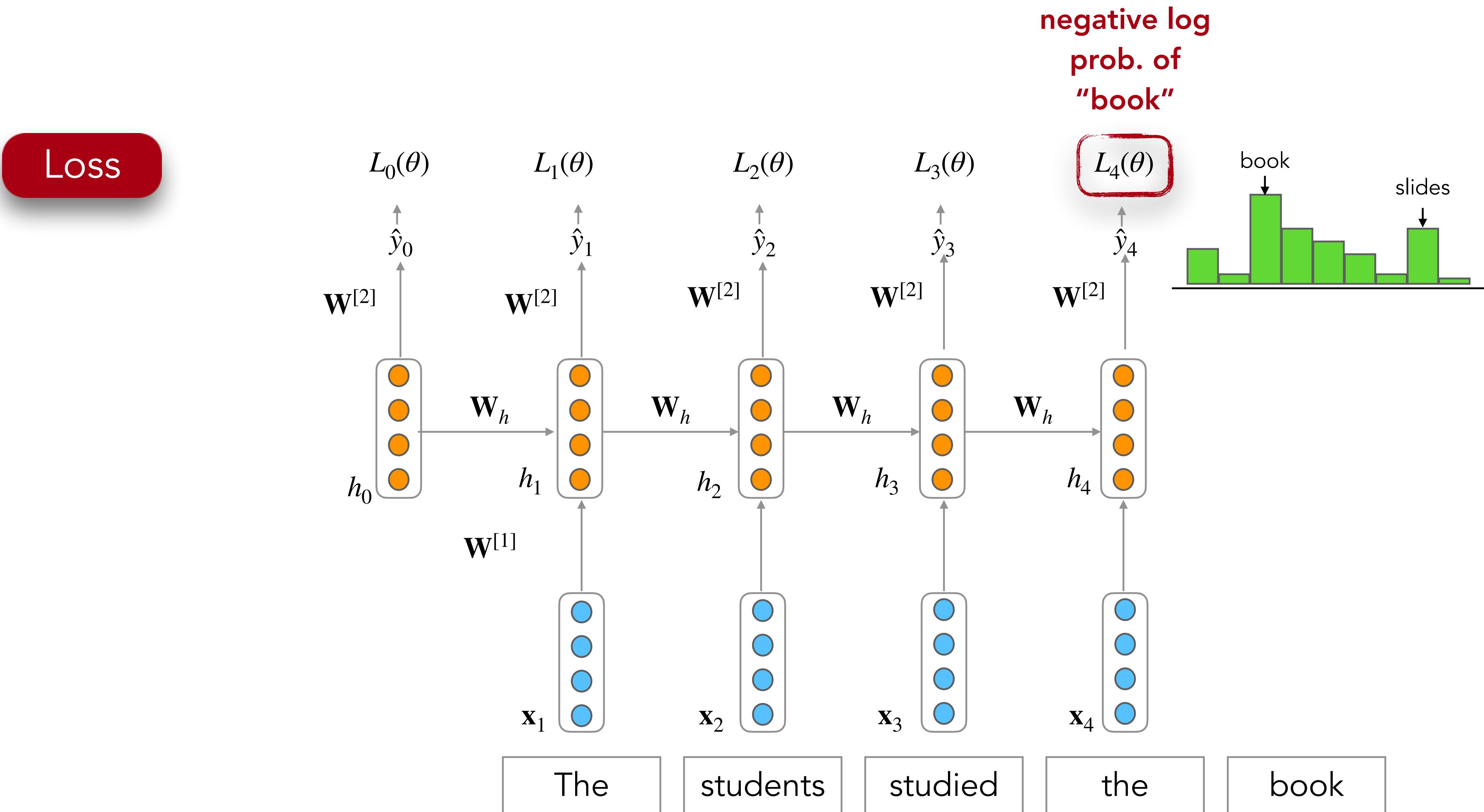
- Average this to get overall loss for entire training set:

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

**negative log  
prob. of  
"The"**

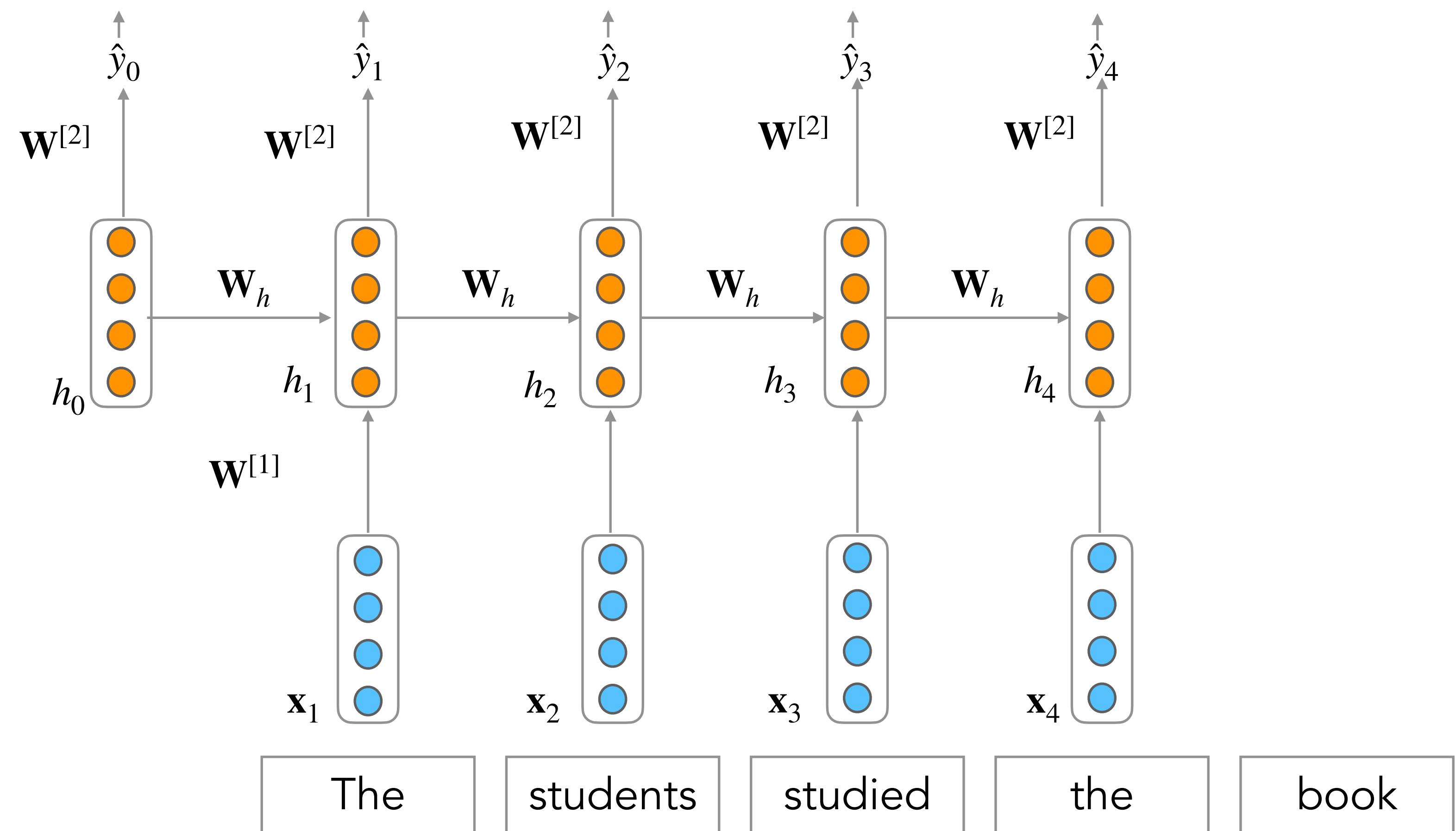






## Loss

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



# RNNs vs. Other LMs

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

Model	Penn Corpus		Switchboard	
	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

# 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  $x_1, x_2, \dots, x_T$  for some  $T$  as a “sentence” or “single data instance”

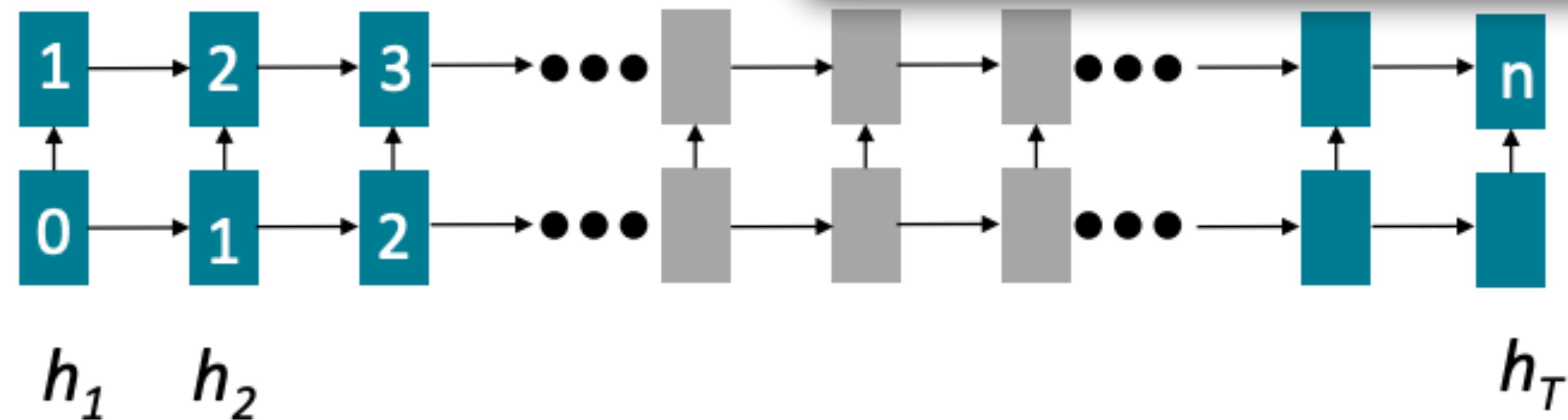
$$L(\theta) = \frac{1}{T} \sum_{t=1}^T L_{CE}(\hat{y}_t, y_t)$$

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

# Training RNNs is hard: Parallelizability

- Forward and backward passes have **O(sequence length)** unparallelizable operations!
  - GPUs can perform a bunch of independent computations at once!
  - But future RNN hidden states can't be computed in full before past RNN hidden states have been computed

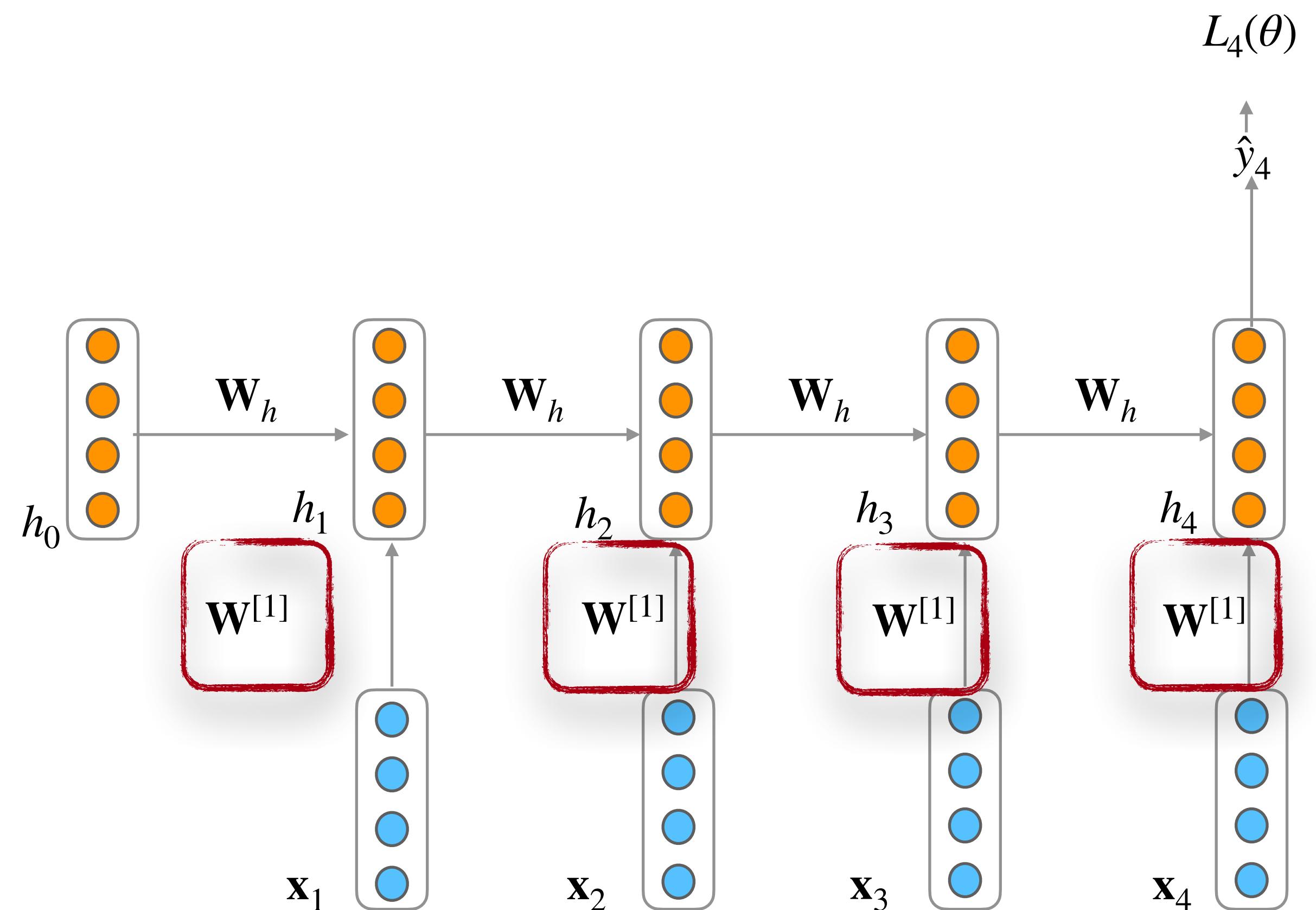
Inhibits training on very large datasets!



Numbers indicate min # of steps before a state can be computed

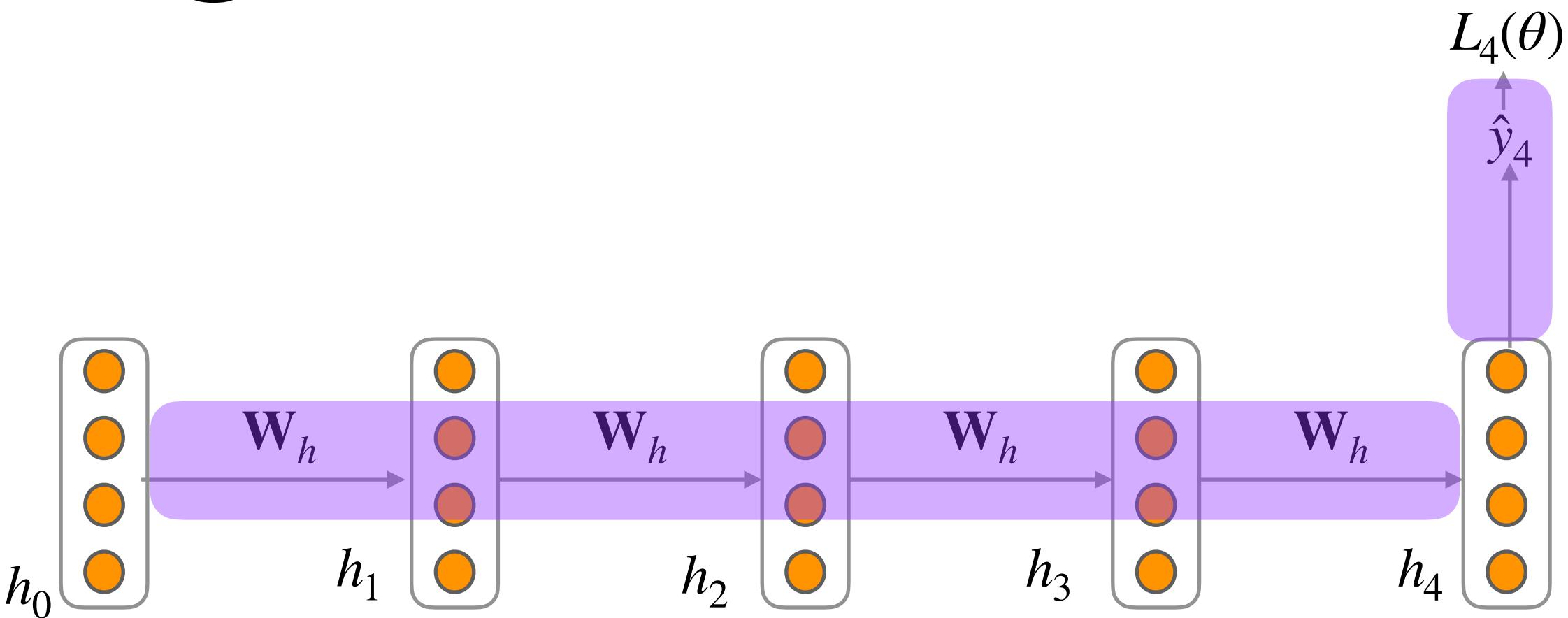
# Training RNNs is hard: Gradients

- 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...

$$\begin{aligned}
 \frac{\partial L_4}{\partial h_0} &= \frac{\partial h_1}{\partial h_0} \times \frac{\partial L_4}{\partial h_1} & = \frac{\partial h_1}{\partial h_0} \times \frac{\partial h_2}{\partial h_1} \times \frac{\partial L_4}{\partial h_2} \\
 & & = \frac{\partial h_1}{\partial h_0} \times \frac{\partial h_2}{\partial h_1} \times \frac{\partial h_3}{\partial h_2} \times \frac{\partial L_4}{\partial h_3} \\
 & & = \frac{\partial h_1}{\partial h_0} \times \frac{\partial h_2}{\partial h_1} \times \frac{\partial h_3}{\partial h_2} \times \frac{\partial h_4}{\partial h_3} \times \frac{\partial L_4}{\partial h_4}
 \end{aligned}$$

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

---

# 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

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

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

Solution? Think data structures...

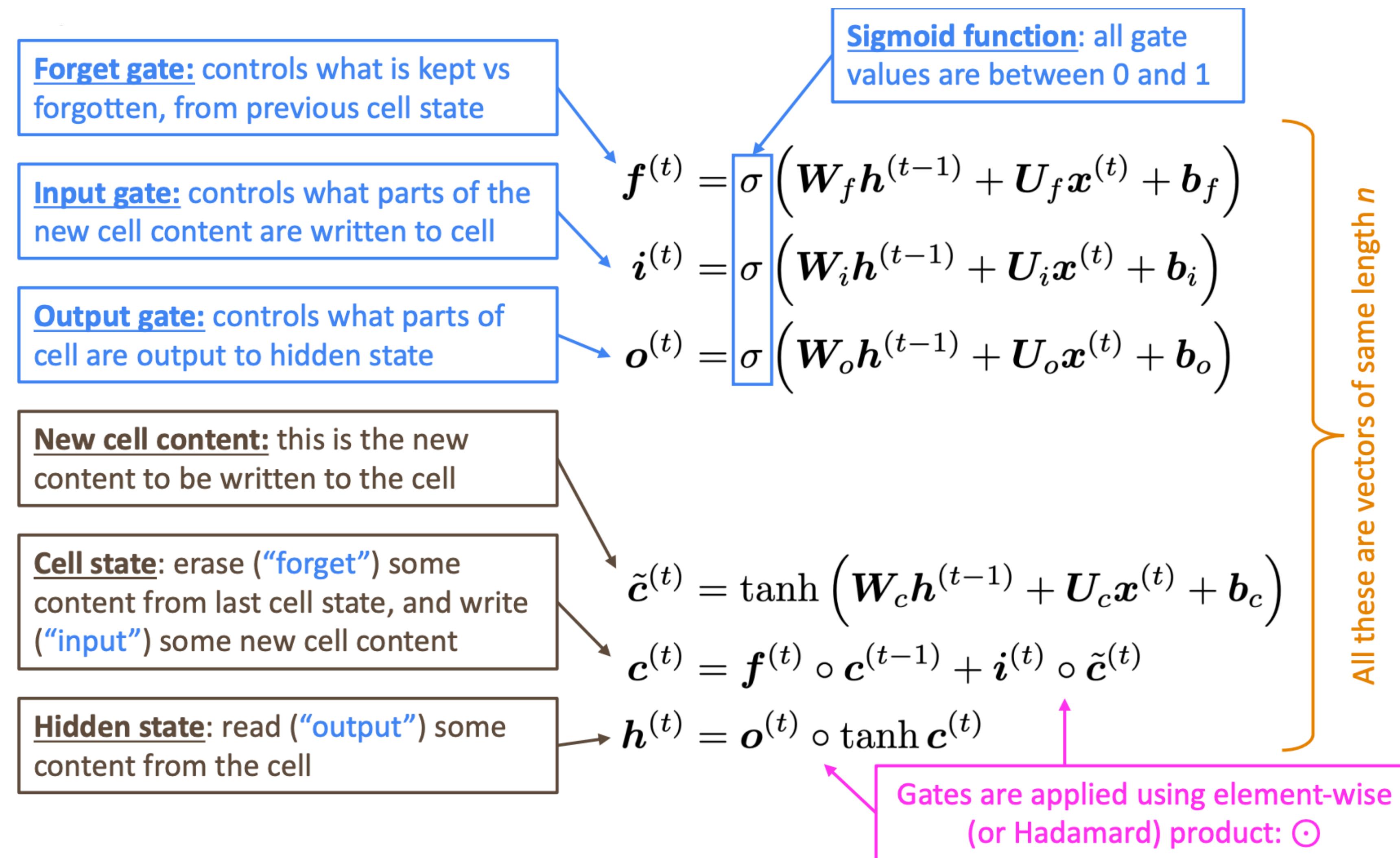
# Long Short-Term Memory RNNs (LSTMs)

- 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

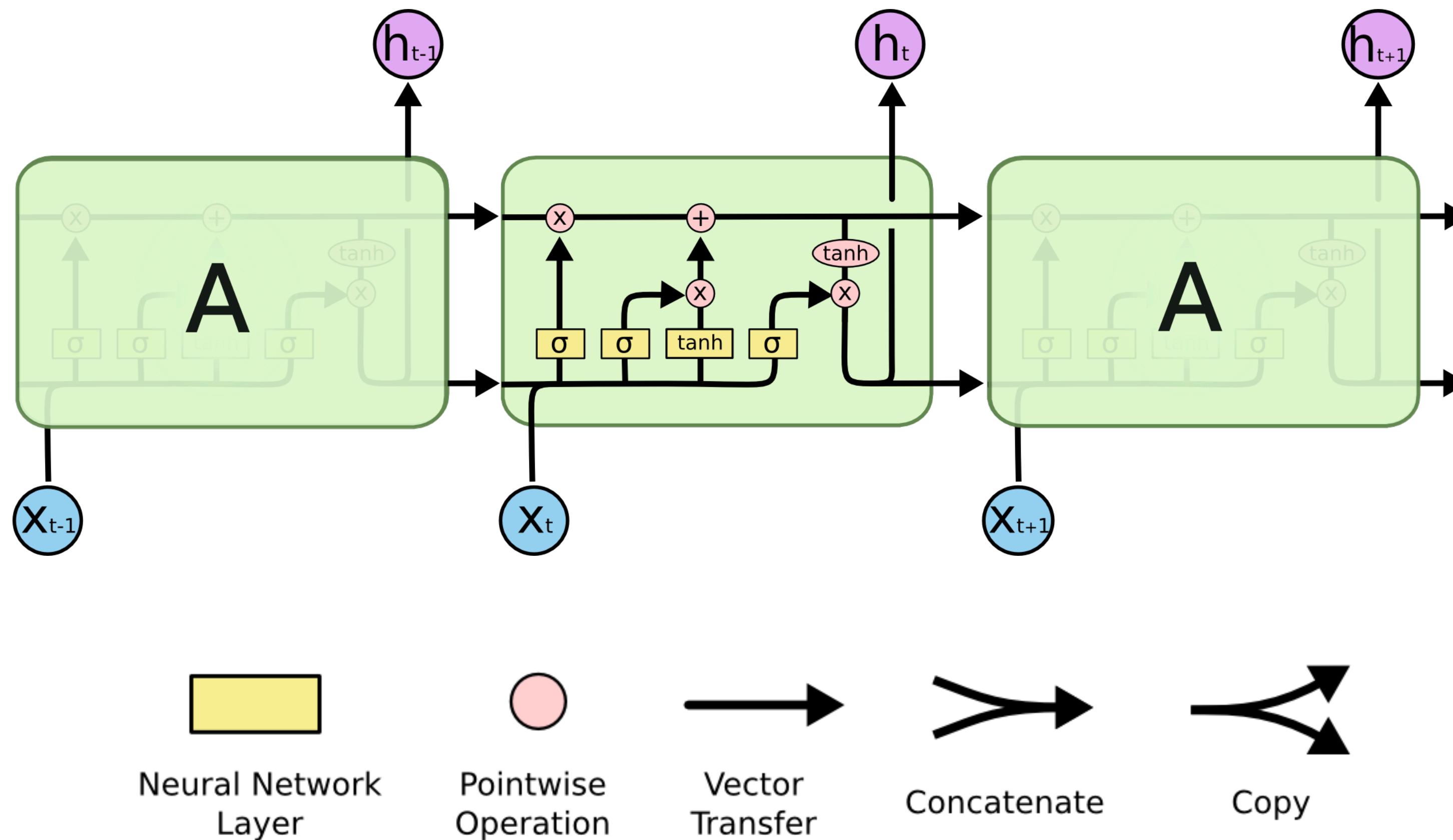
# LSTMs

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

At timestep  $t$ :

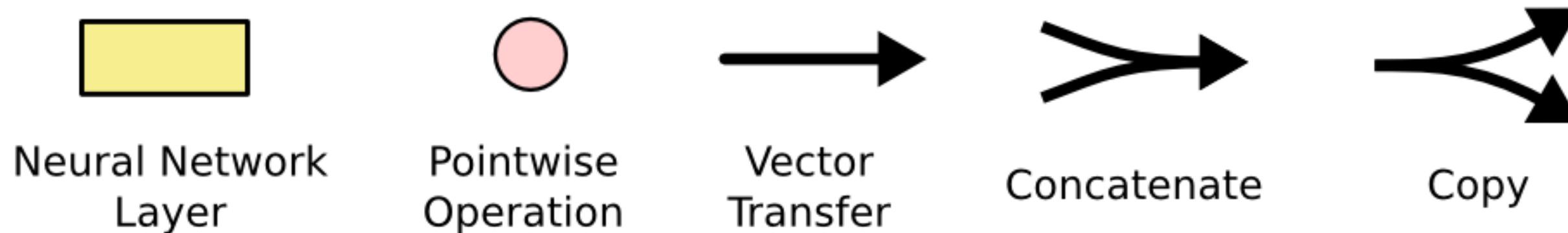
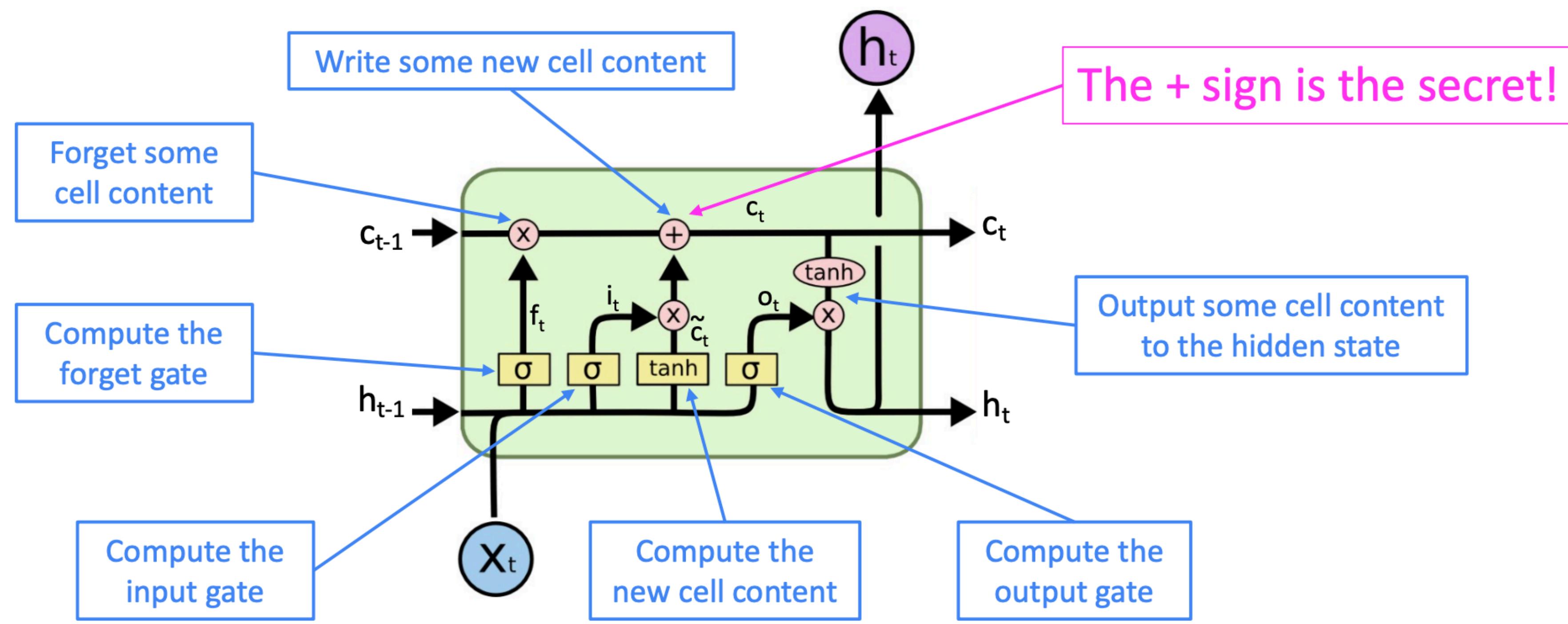


# LSTMs: A Visual Representation



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

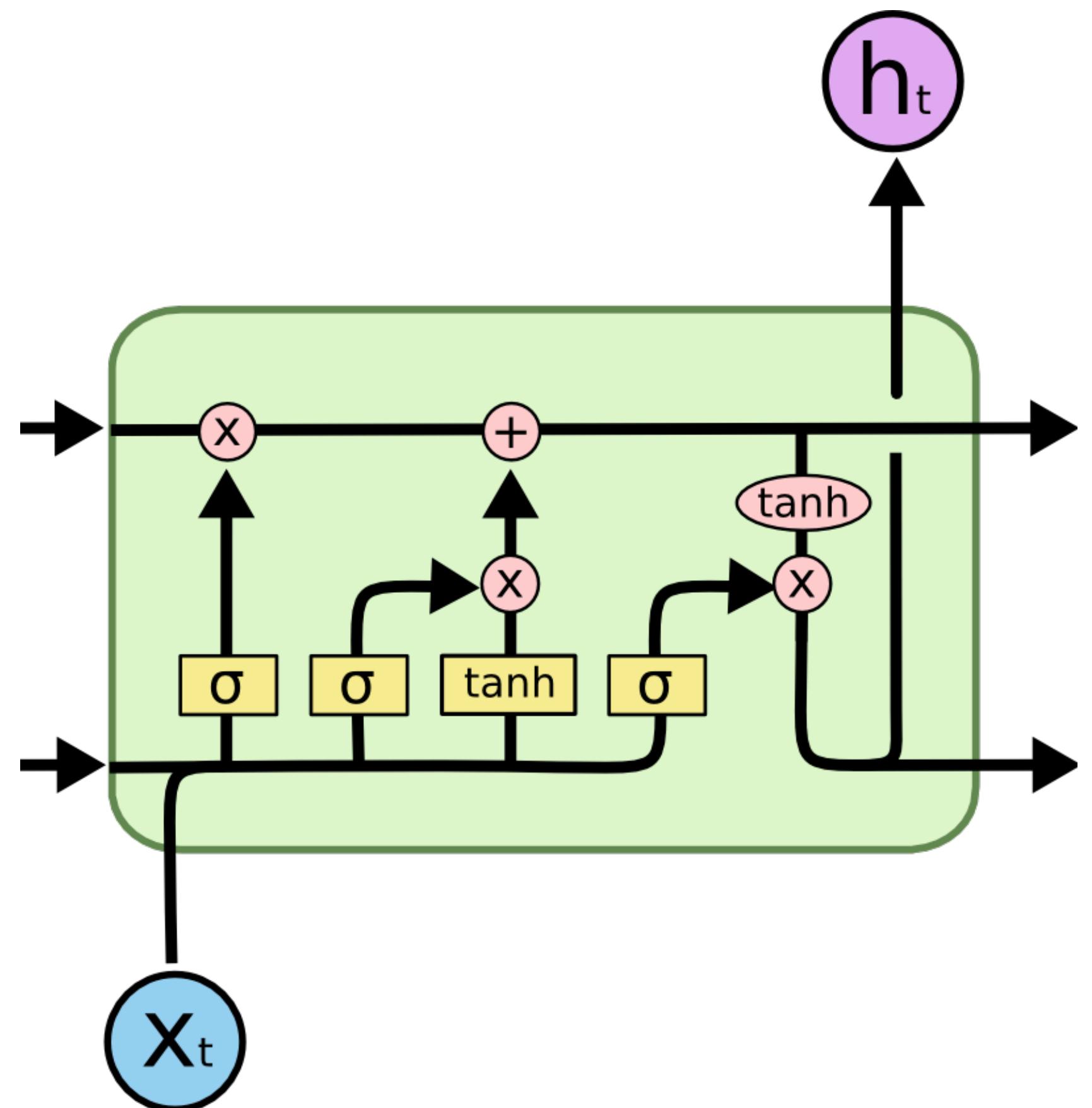
# LSTMs: A Visual Representation



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

