Lecture 7: Backpropagation

Instructor: Swabha Swayamdipta
USC CSCI 444 NLP
Sep 22, 2025

Some slides adapted from Dan Jurafsky and Chris Manning

Announcements + Logistics

- Wed: Project Proposal Due
  - See instructions on website, please do not break format
- HW2 out yesterday
- HW1 grades will be out by next week
- Quiz 2 postponed
  - Oct 1, Wed after next

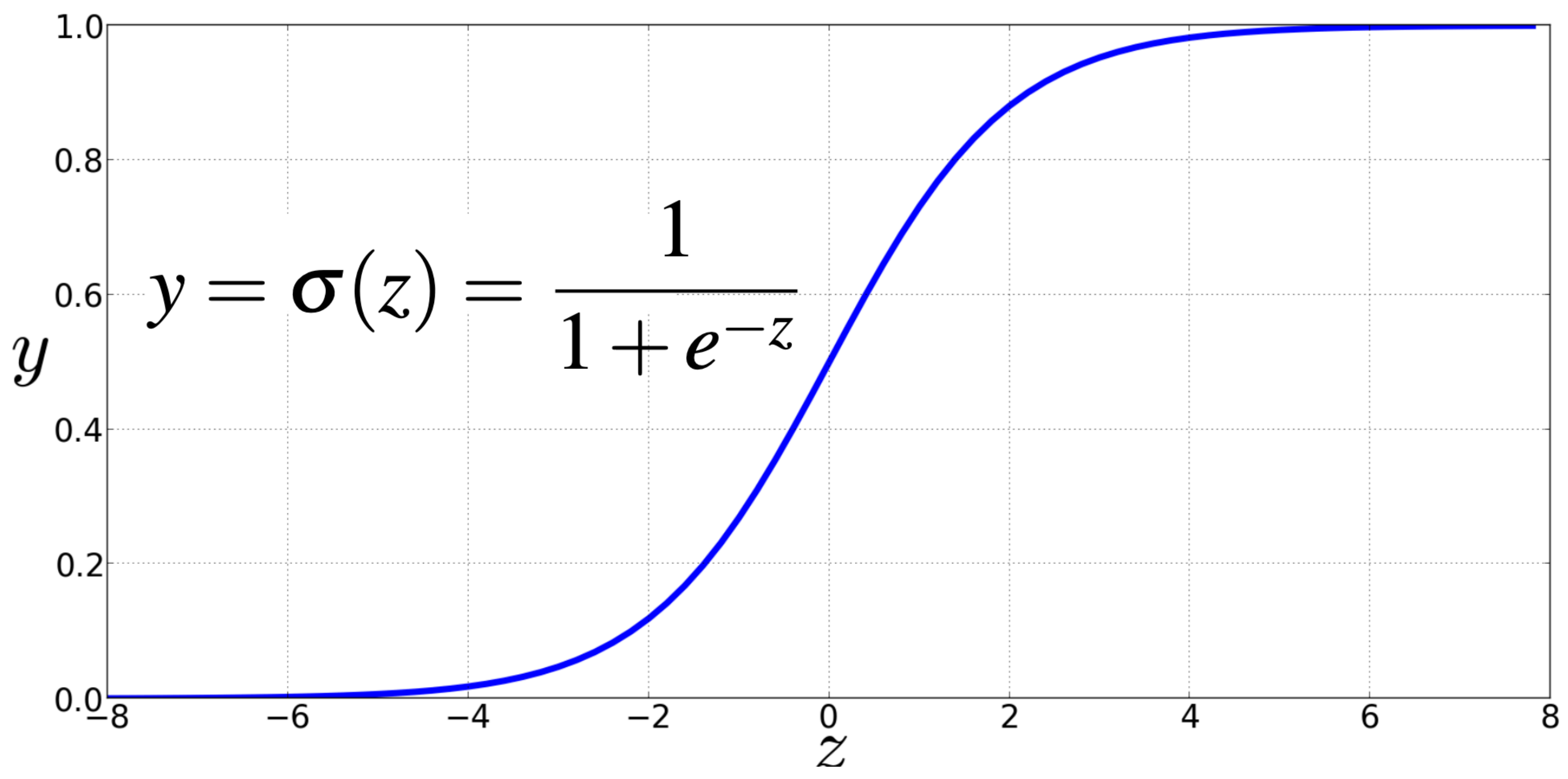
Lecture Outline

- Recap: Feedforward Neural Nets
- Feedforward Net Language Models
- Feedforward Nets for Classification
- Training Feedforward Nets
- Computation Graphs and Backprop
- Next: Recurrent Neural Nets (RNNs)

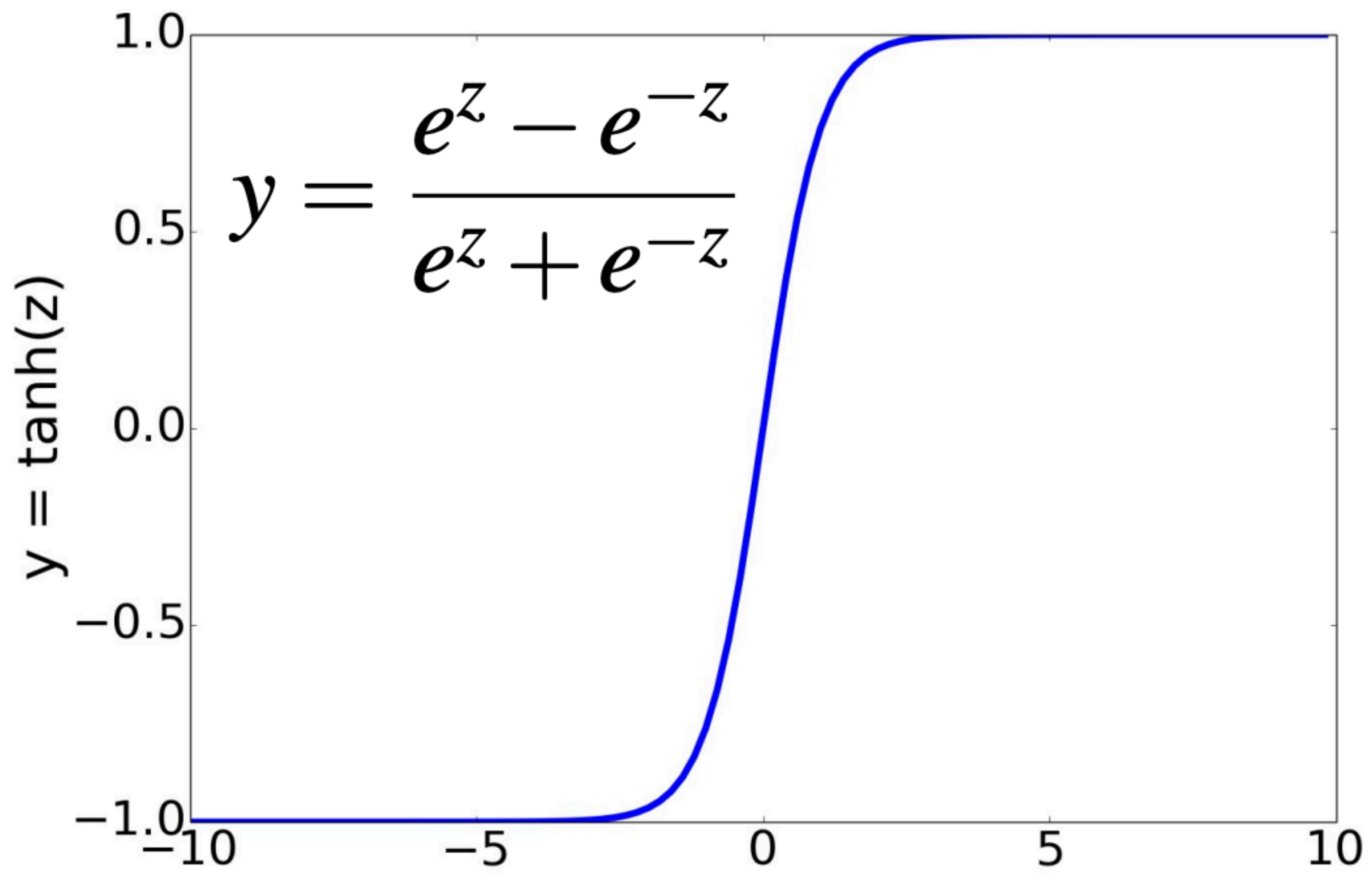
Recap: Feed-Forward Neural Networks

Non-Linear Activation Functions

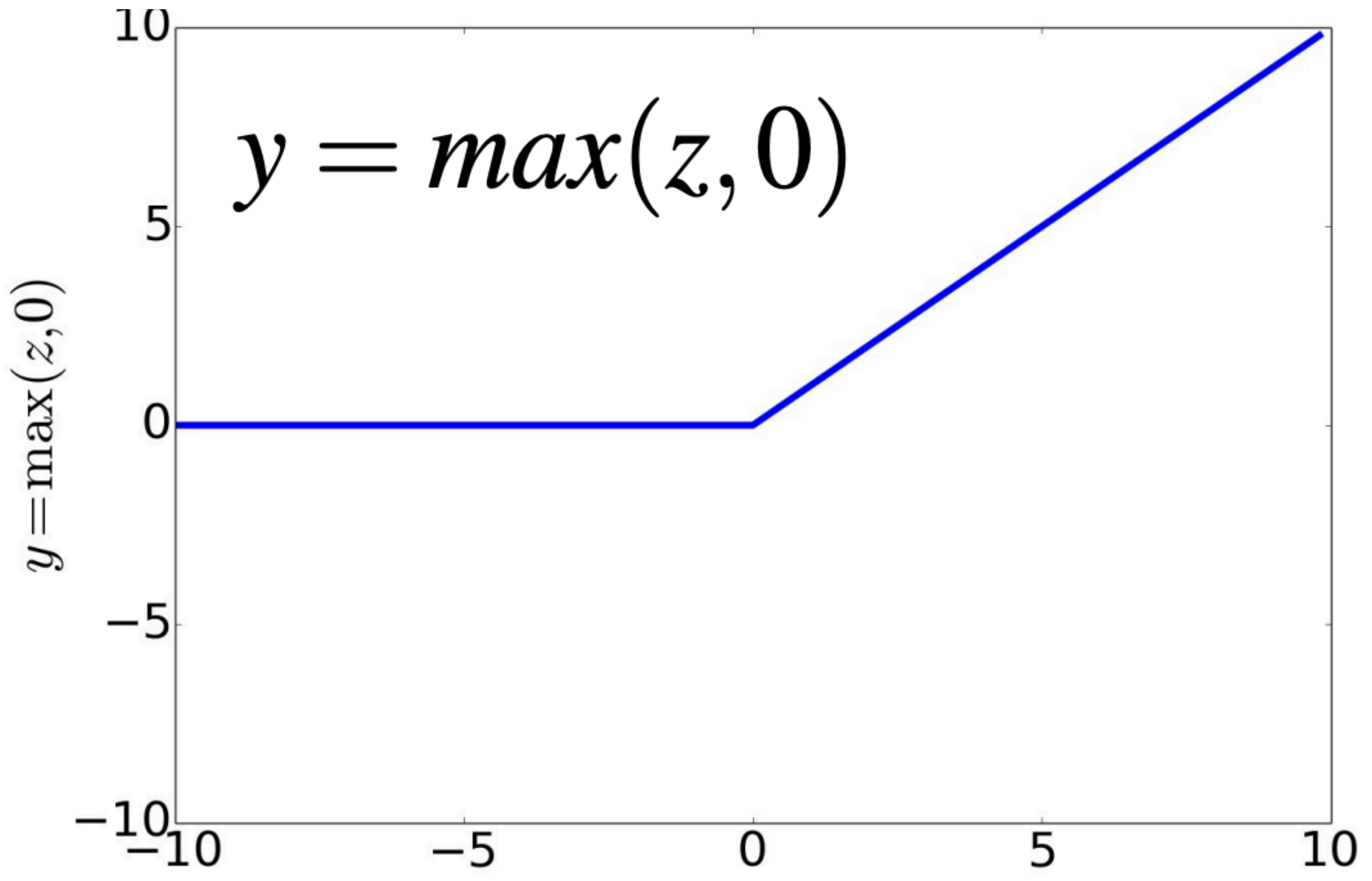
Most common!



sigmoid



tanh

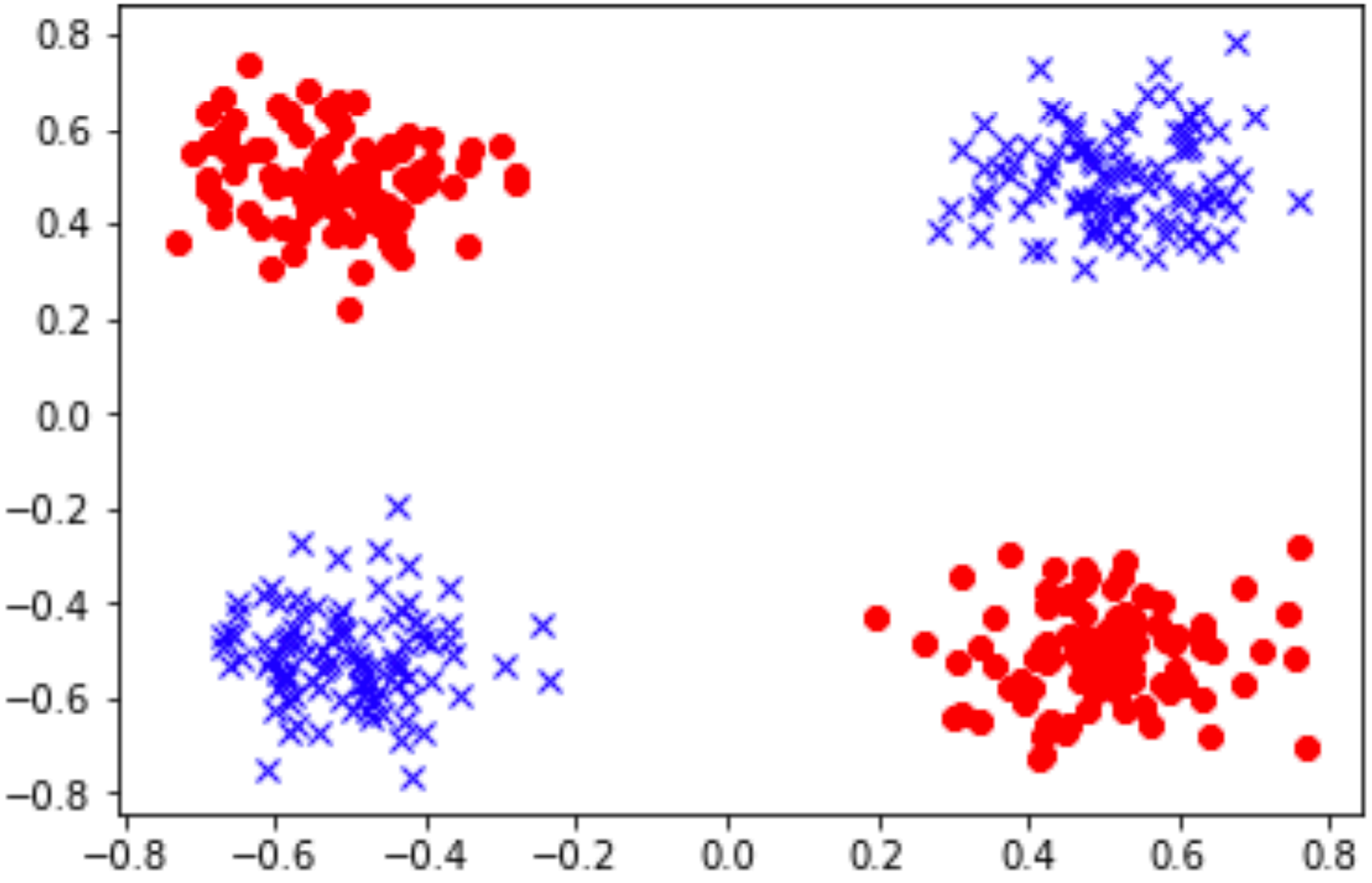


relu (Rectified Linear Unit)

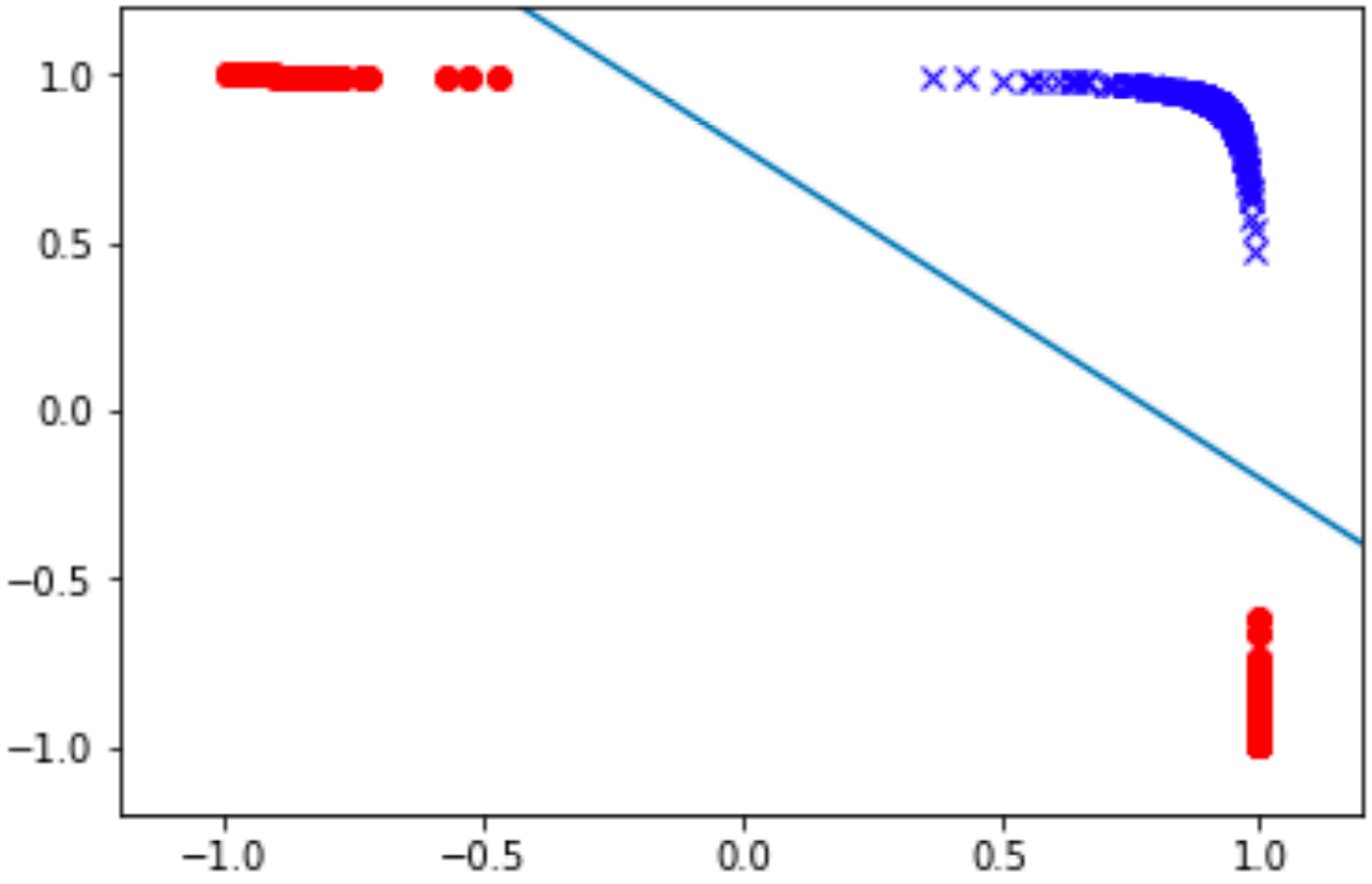
The key ingredient of a neural network is the non-linear activation function

Power of non-linearity

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



After a $\tanh(\cdot)$ transformation:



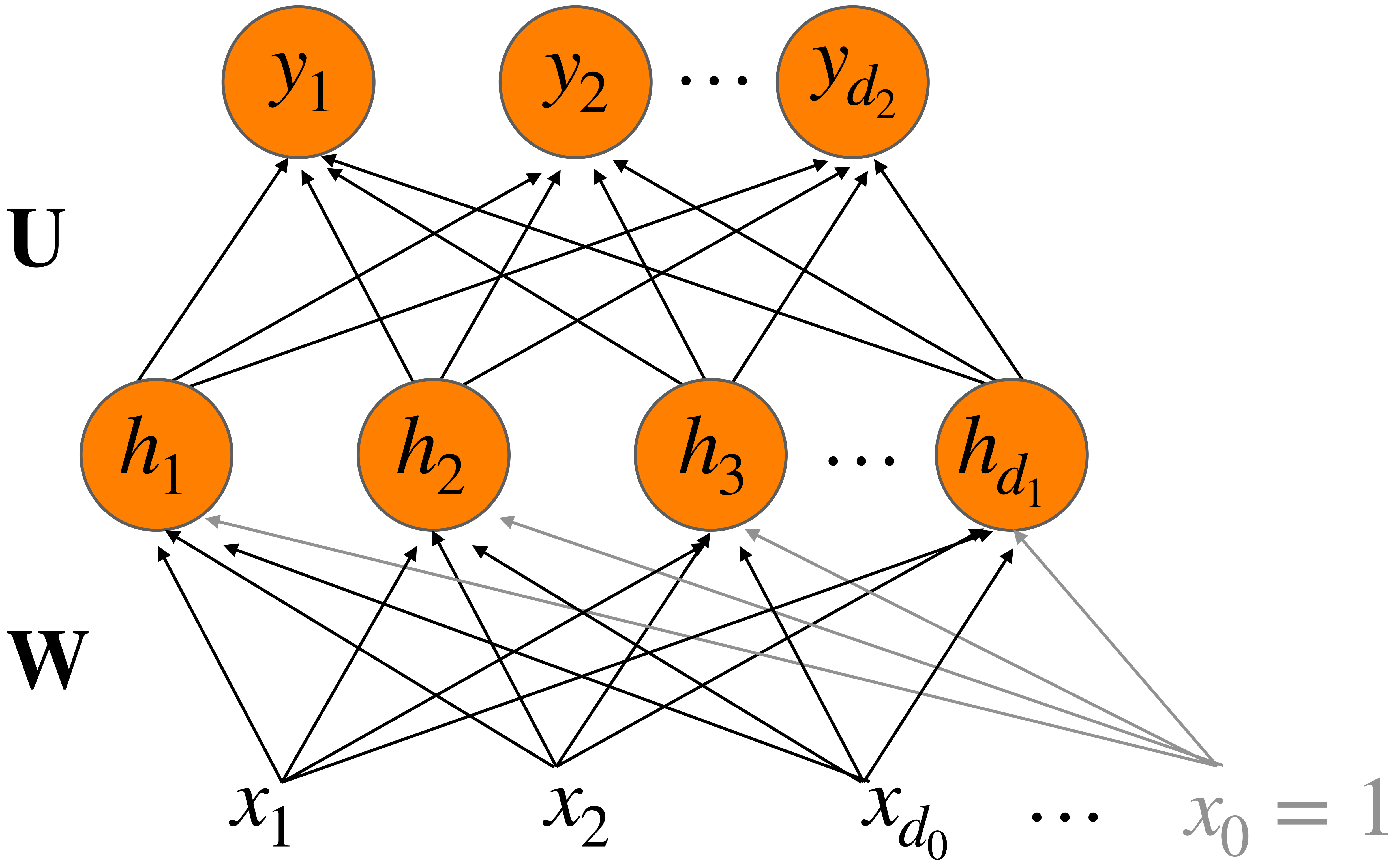
Two-layer FFNN: Notation

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

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

Input layer: vector $\mathbf{x}$



We usually drop the $\mathbf{b}$ and add one dimension to the $\mathbf{W}$ matrix

Lecture Outline

- Recap: Feedforward Neural Nets
- Feedforward Net Language Models
- Feedforward Nets for Classification
- Training Feedforward Nets
- Computation Graphs and Backprop
- Next: Recurrent Neural Nets (RNNs)

FFNN Language Models

Feedforward Neural Language Models

- Language Modeling: Calculating the probability of the next word in a sequence given some history.
- Compared to n-gram language models, neural network LMs achieve much higher performance
  - In general, count-based methods can never do as well as optimization-based ones
- State-of-the-art neural LMs are based on more powerful neural network technology like Transformers
- But **simple feedforward LMs** work well too!

Why?

Can neural LMs overcome the overfitting problem in n-gram LMs?

Simple Feedforward Neural LMs

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

Problem: Now we are dealing with sequences of arbitrary length….

Solution: Sliding windows (of fixed length)

Basis of word embedding models!

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

First introduced by Yoshua Bengio and colleagues in 2003

USC Viterbi

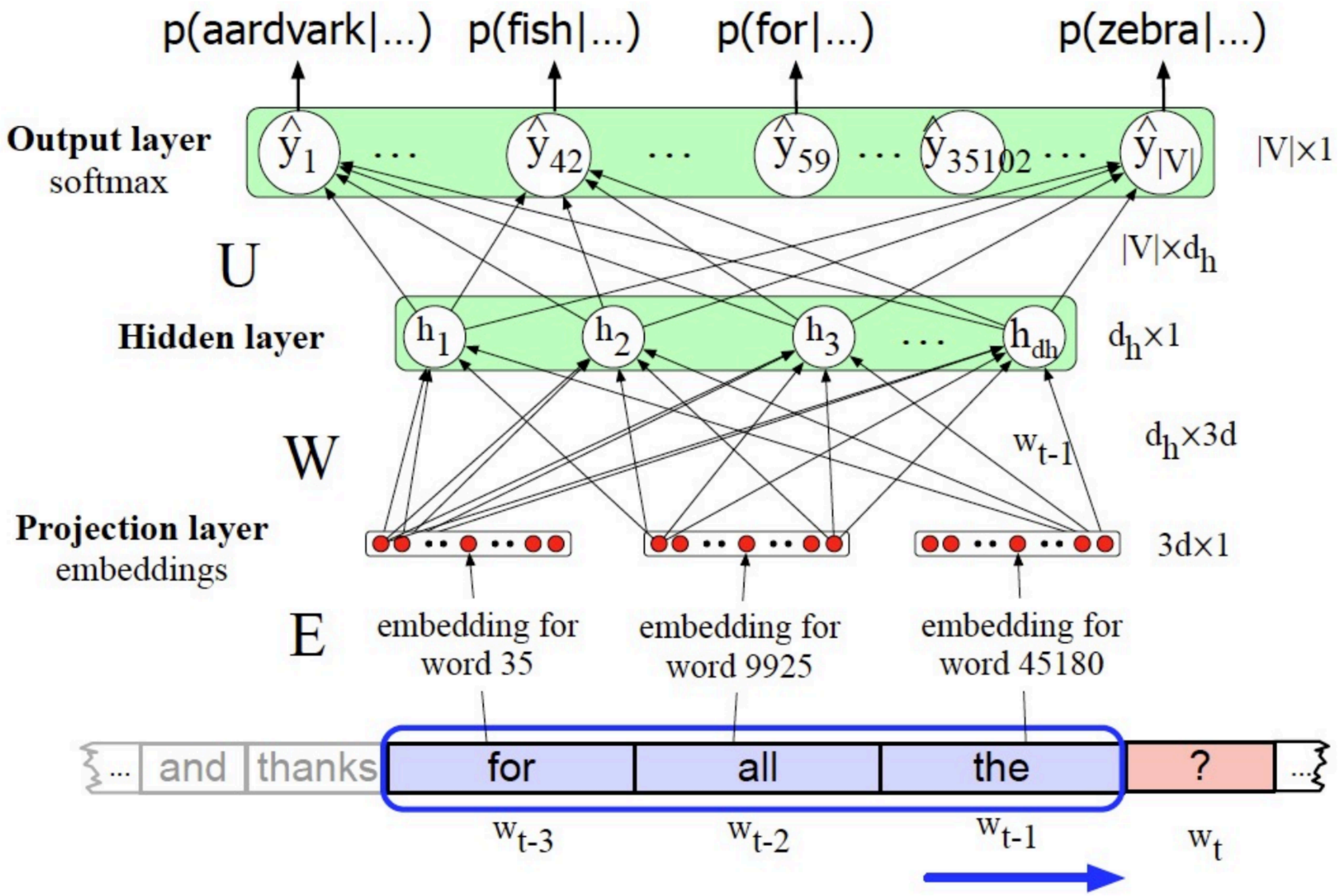
Data: Feedforward Language Model

- Self-supervised
- Computation is divided into time steps t, where different sliding windows are considered
- $x_t = (w_{t-1}, \ldots, w_{t-M+1})$ for the context
    - represent words in this prior context by their embeddings, rather than just by their word identity as in n-gram LMs
    - allows neural LMs to generalize better to unseen data / similar data
    - All embeddings in the context are concatenated
- $y_t = w_t$ for the next word
    - Represented as a one hot vector of vocabulary size where only the ground truth gets a value of 1 and every other element is a 0

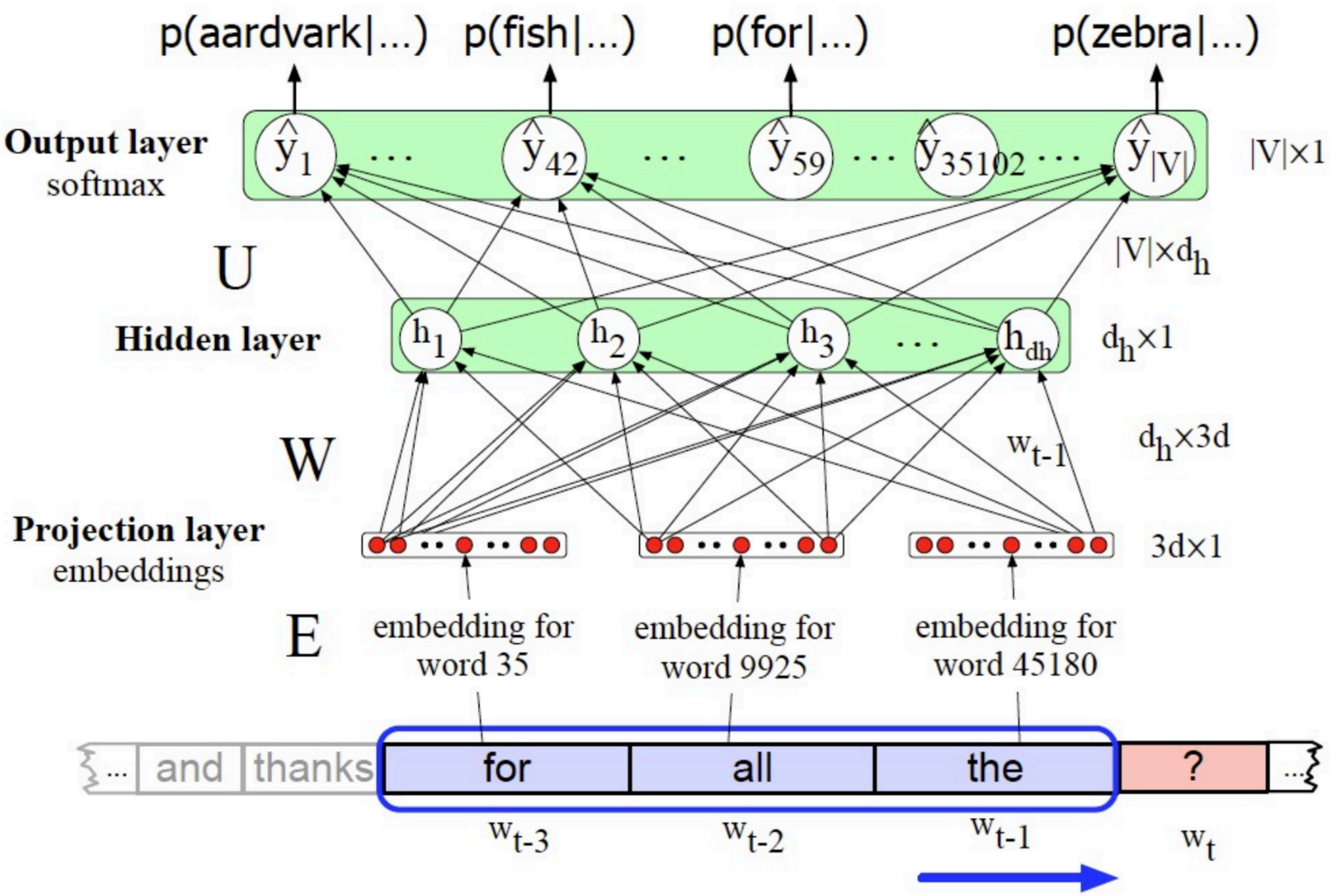
One-hot vector

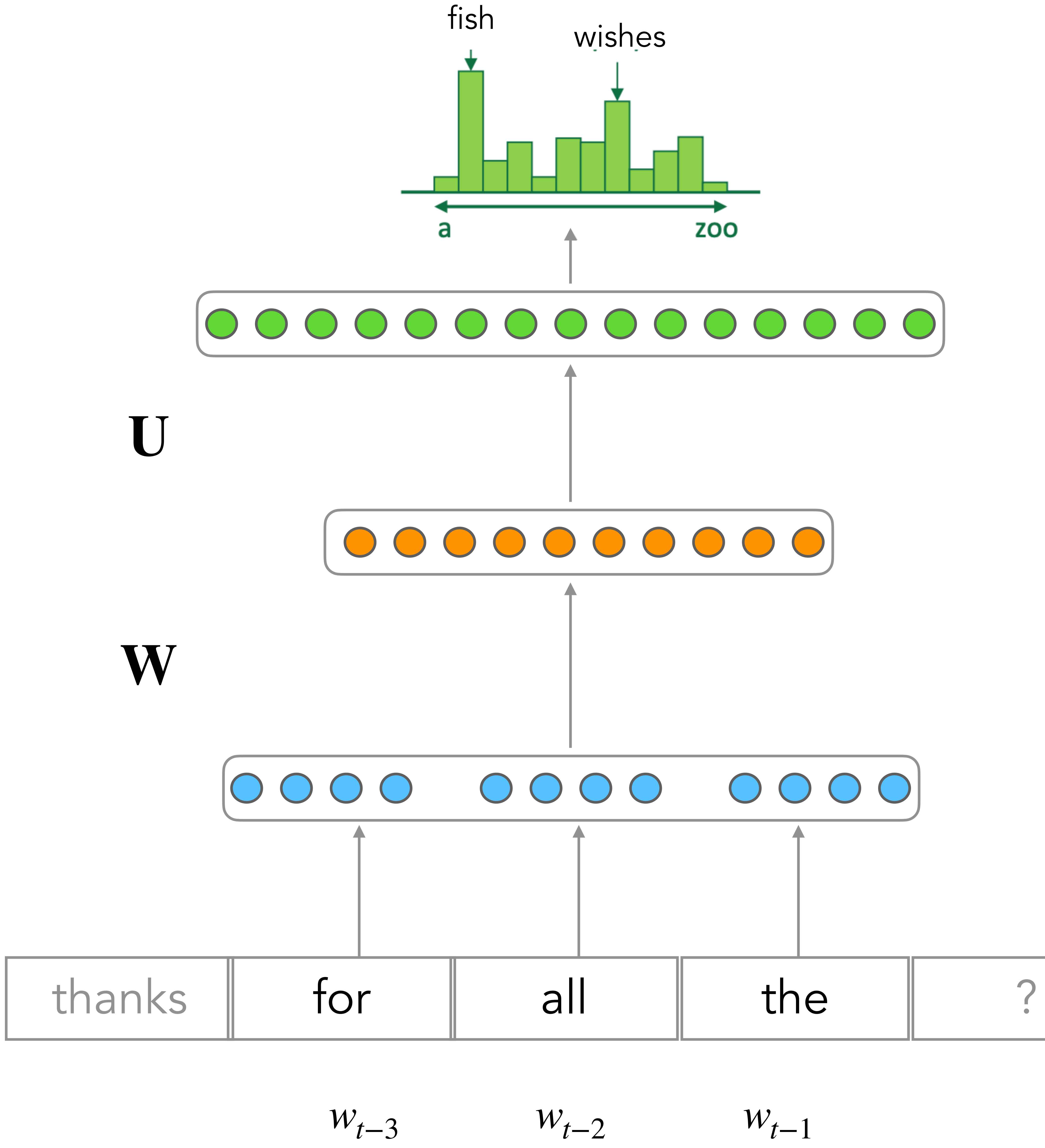
Feedforward Neural LM

- Sliding window of size 4 (including the target word)
- Every feature in the embedding vector connected to every single hidden unit
- Projection / embedding layer is a kind of input layer
  - This is where we plug in our word2vec embeddings
  - May or may not update embedding weights



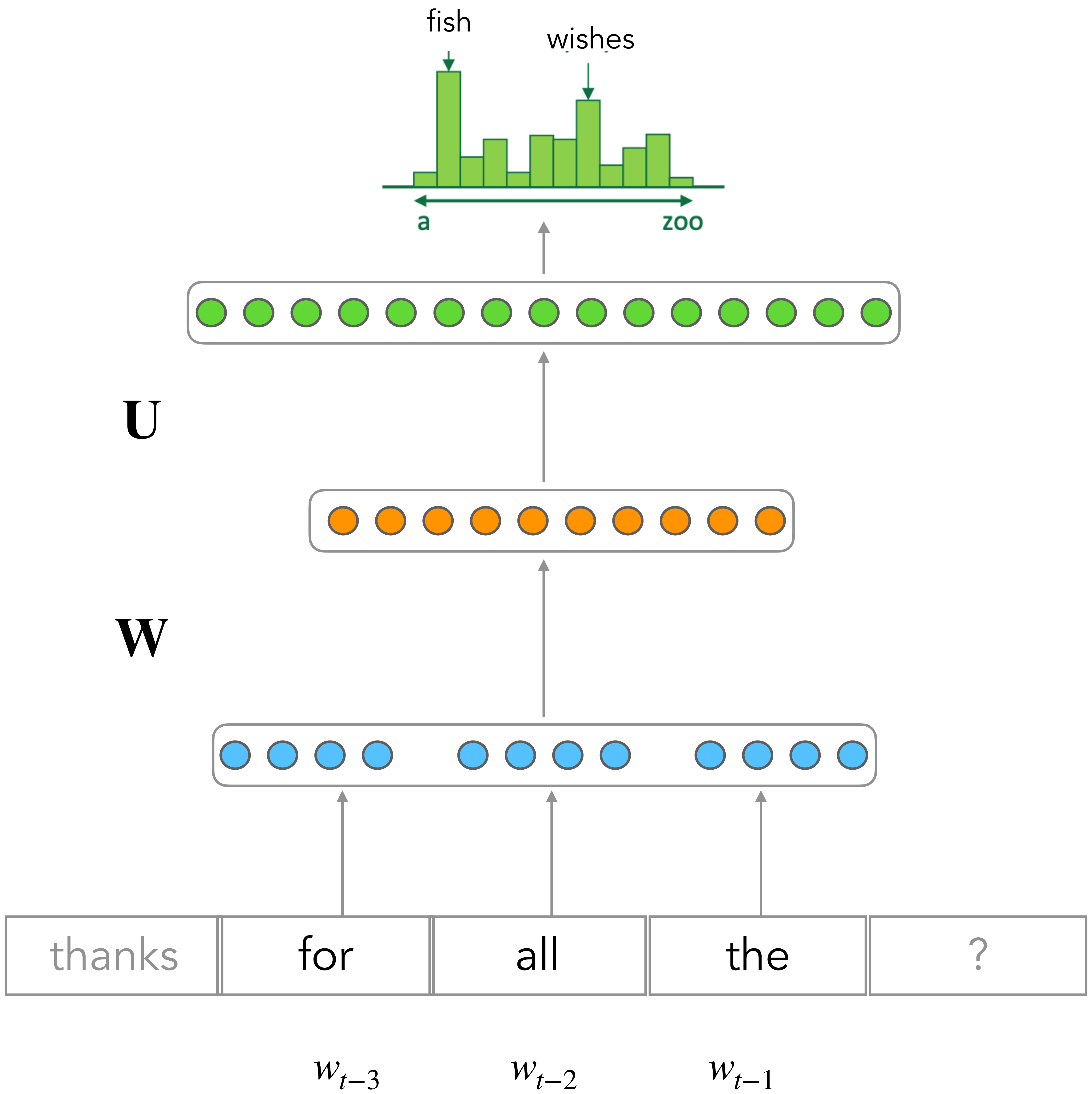
Simplified Representation





Feedforward LMs: Windows

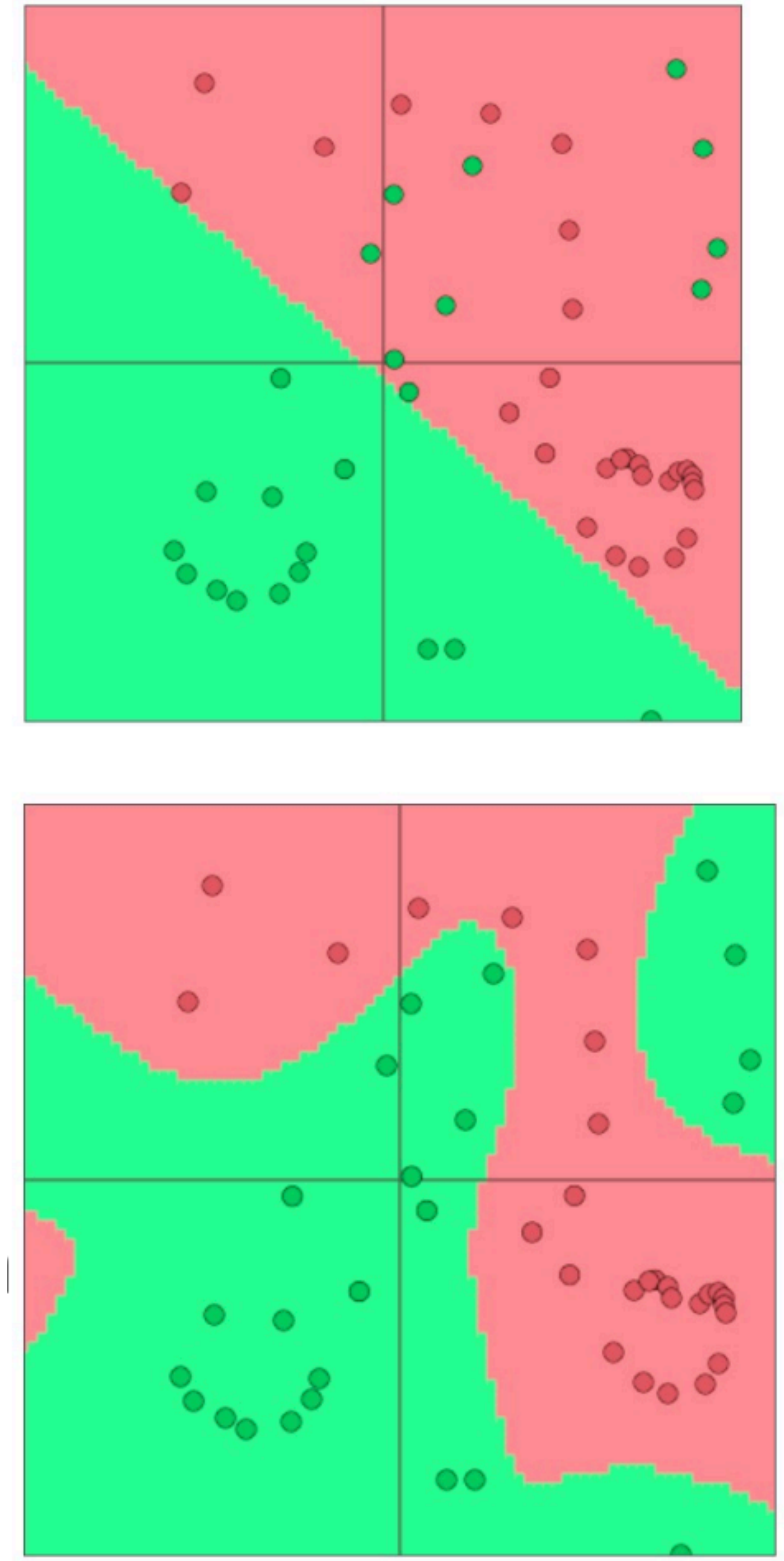
- The goodness of the language model depends on the size of the sliding window!
- Fixed window can be too small
- Enlarging window enlarges **W**
- Each word uses different rows of **W**. We don't share weights within the window.
- Window can never be large enough!



FFNN for Classification

FFNN and Classification

- Learn both FFNN parameters, $\mathbf{W}$ and word embeddings!!
- Conceptually, we have an embedding layer: $\mathbf{x}_i$ for the ith input word in the window
- We use deep networks—more layers—that let us compose our data multiple times, giving a non-linear classifier

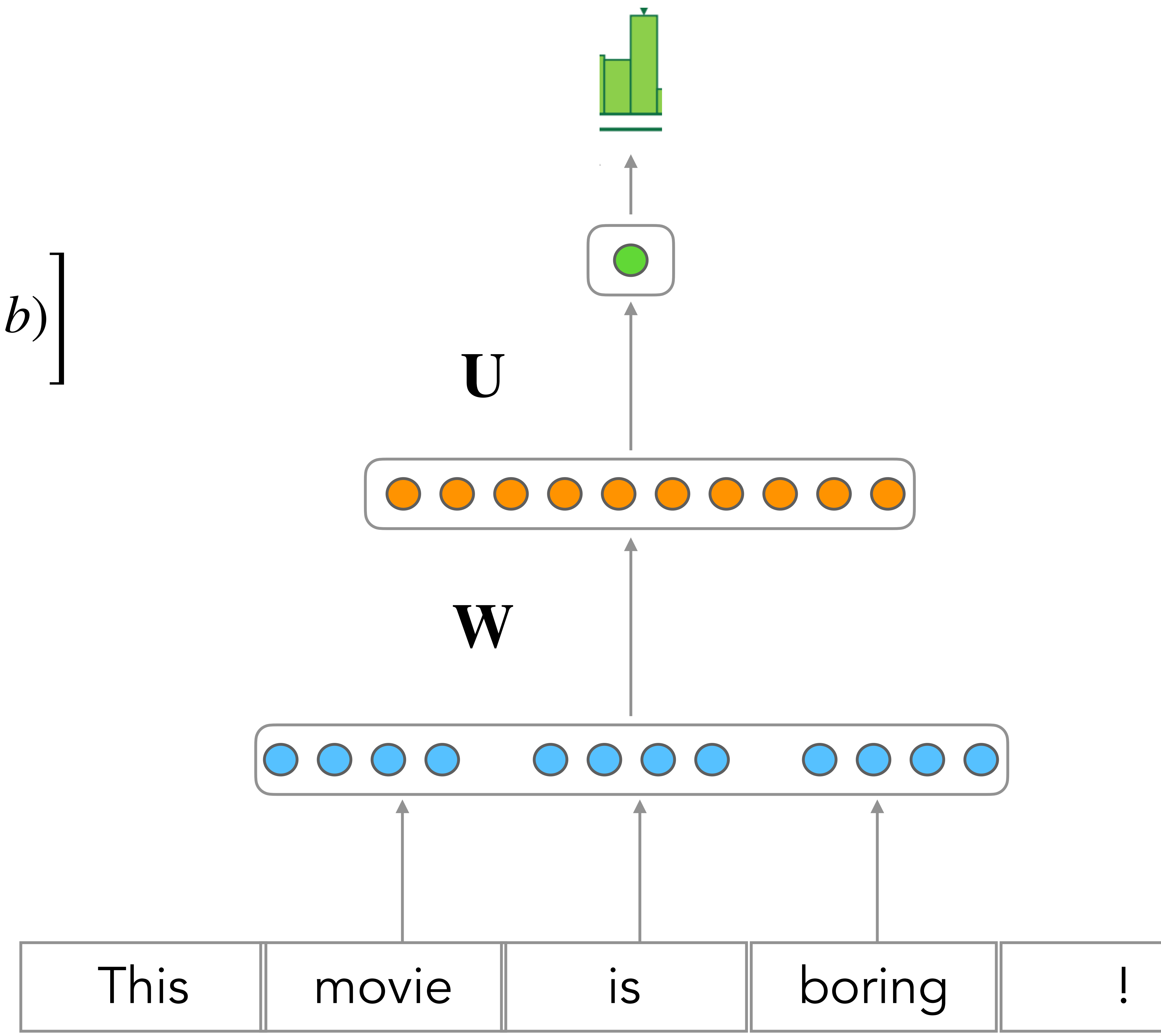


FNN and Classification

- Training Objective: For each training example $(\mathbf{x}, y)$, our objective is to maximize the probability of the correct class y or we can minimize the negative log probability of that class:

$$L_{CE} = -\log P(y = c \mid \mathbf{x}; \theta) = -(\mathbf{w}_c \cdot \mathbf{x} + b) + \log \left[\sum_{j=1}^{K} \exp(\mathbf{w}_j \cdot \mathbf{x} + b) \right]$$

- Loss as Cross entropy: $H(q, p) = -\sum_{j=1}^{K} q_j \log p_j$
  - ground truth (or true or gold or target) is a 1-hot vector of size K, where $q_j = 1; q_i = 0 \forall i \neq j$
  - hence, the only term left is the negative log probability of the true class, $-\log p(y_j \mid \mathbf{x})$
- True for both language modeling and classification



Lecture Outline

- Recap: Feedforward Neural Nets
- Feedforward Net Language Models
- Feedforward Nets for Classification
- Training Feedforward Nets
- Computation Graphs and Backprop
- Next: Recurrent Neural Nets (RNNs)

Training FFNNs

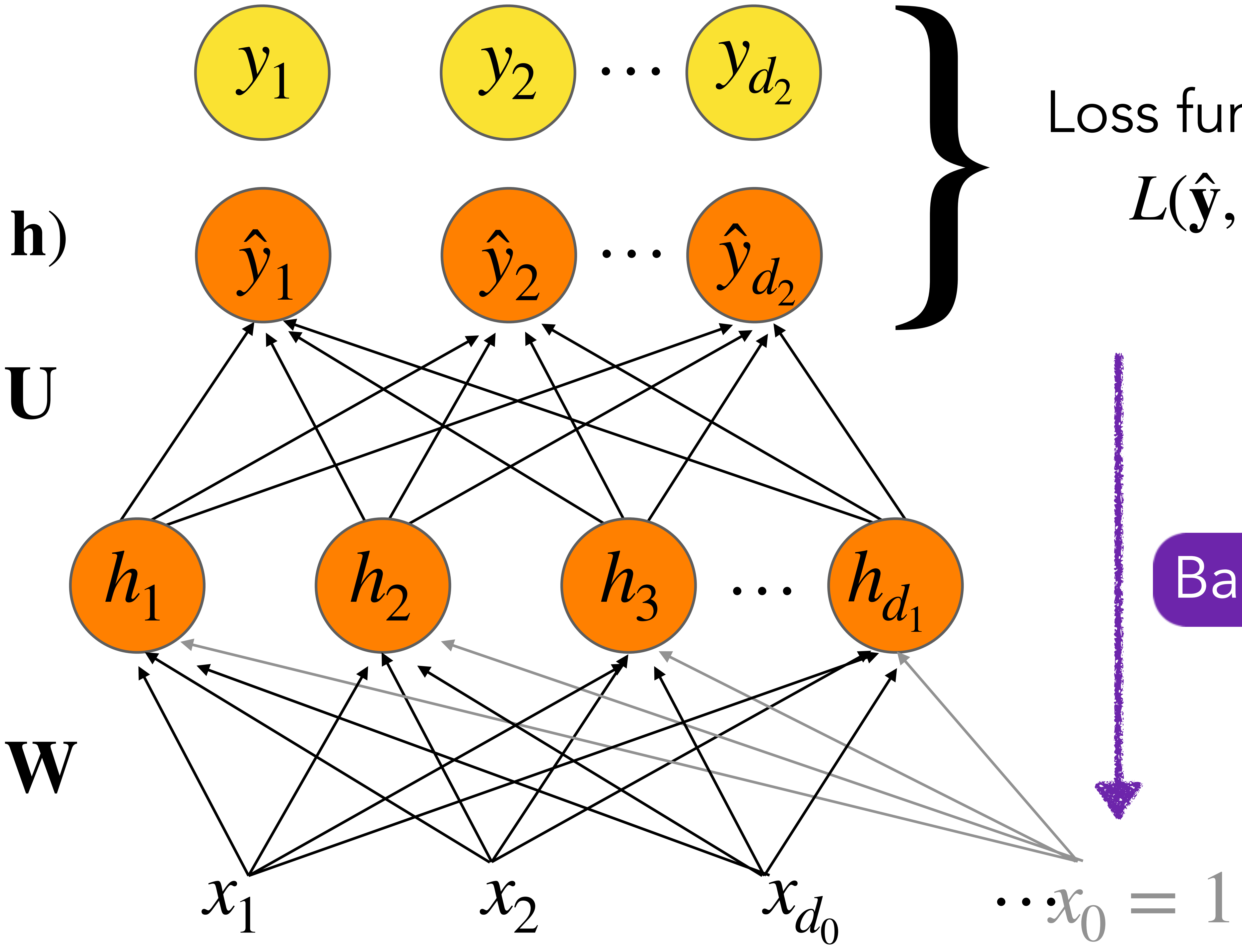
Intuition: Training a 2-layer Network

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

Intuition: Training a 2-layer network

For every training tuple (x, y)

- Run forward computation to find our estimate $\hat{y}$
- Run backward computation to update weights:
  - For every output node
    - Compute loss L between true y and the estimated $\hat{y}$
    - For every weight w from hidden layer to the output layer
      - Compute the gradient of L w.r.t. $\mathbf{w}$ and update $\mathbf{w}$
    - For every hidden node
      - Assess how much blame it deserves for the current answer
      - For every weight w from input layer to the hidden layer
        - Compute the gradient of L w.r.t. $\mathbf{w}$ and update $\mathbf{w}$

LR and FFNN: Similarities and Differences

Cross Entropy Loss again!

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

Gradient Update

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

Only one parameter! Remember the bias parameter is just another dimension

Computation Graphs

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

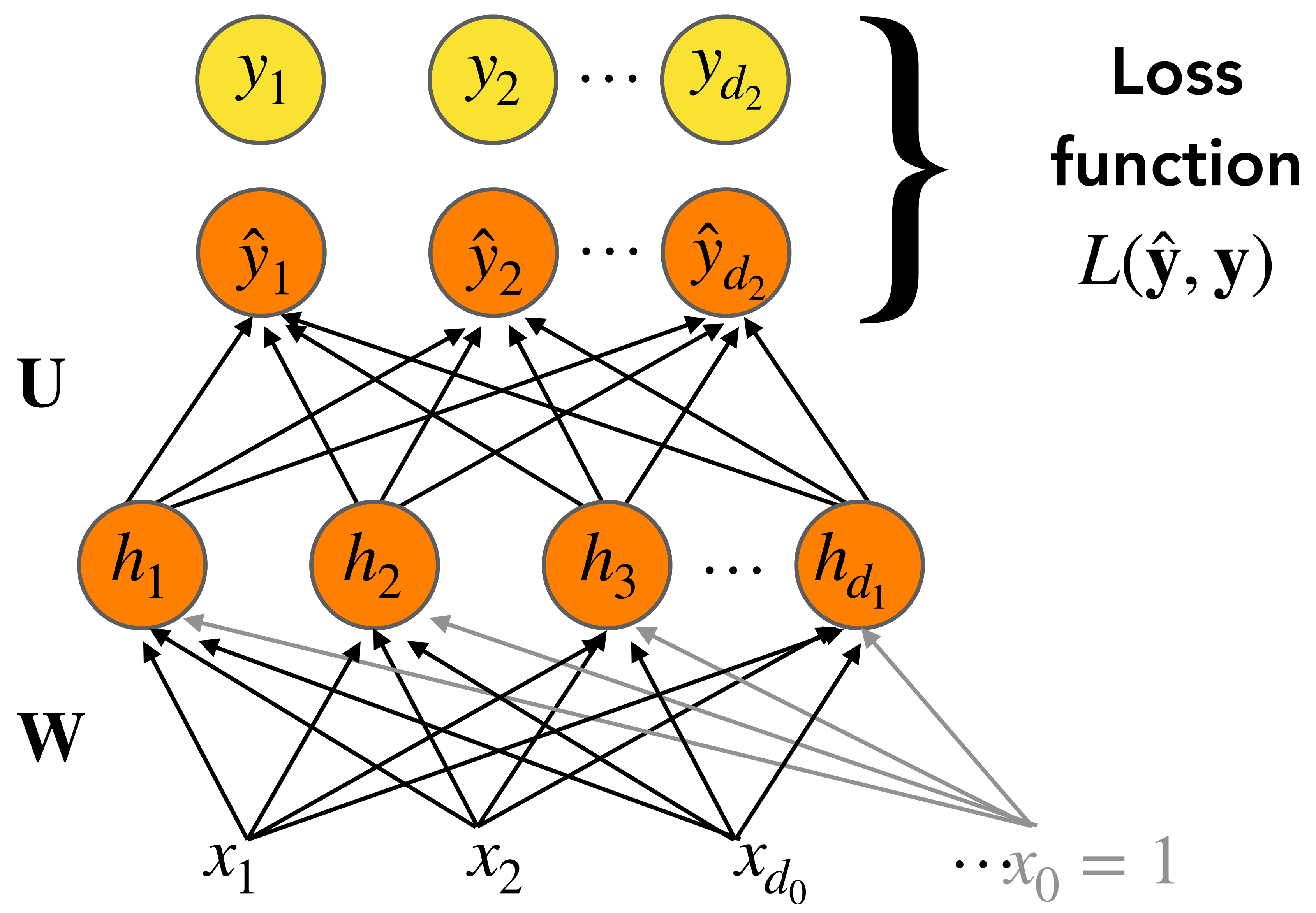
Lecture Outline

- Recap: Feedforward Neural Nets
- Feedforward Net Language Models
- Feedforward Nets for Classification
- Training Feedforward Nets
- Computation Graphs and Backprop
- Next: Recurrent Neural Nets (RNNs)

Computation Graphs and Backprop

Why Computation Graphs?

- 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
    - Backprop relies on computation graphs



Backprop

Graph representing the process of computing a mathematical expression

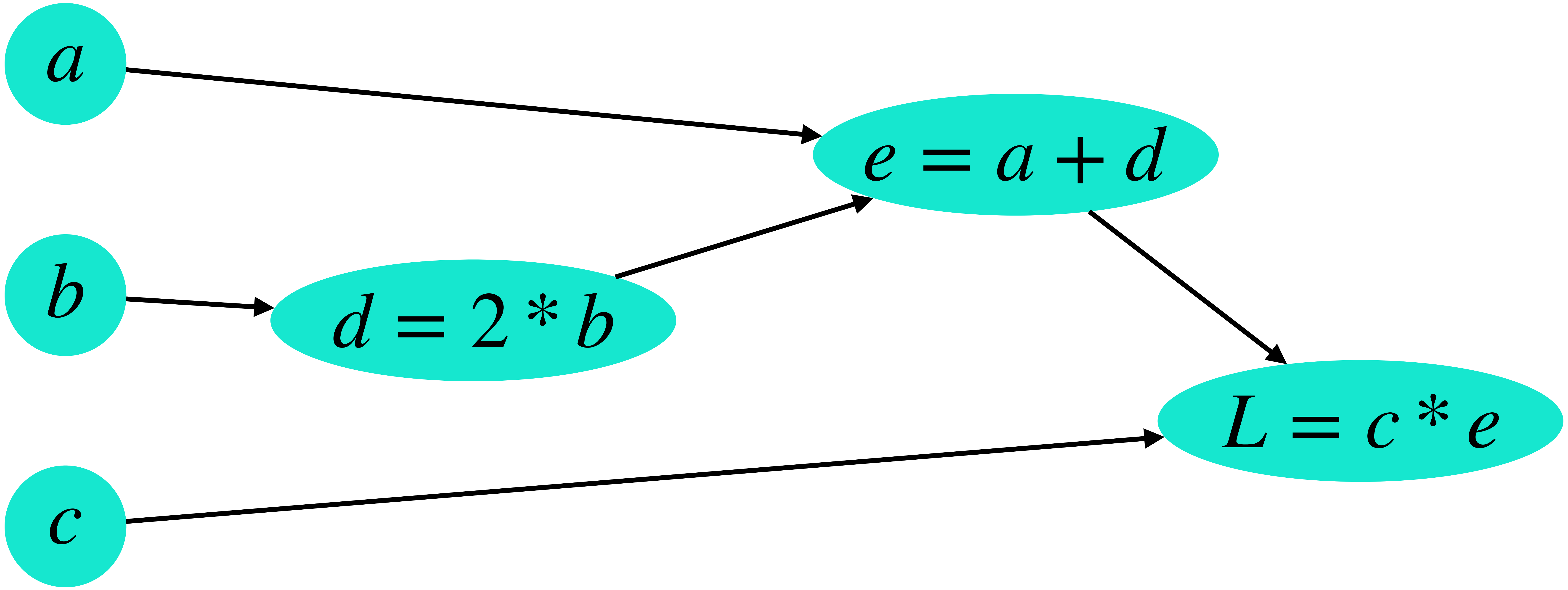
Rumelhart, Hinton, Williams, 1986

Example: Computation Graph

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

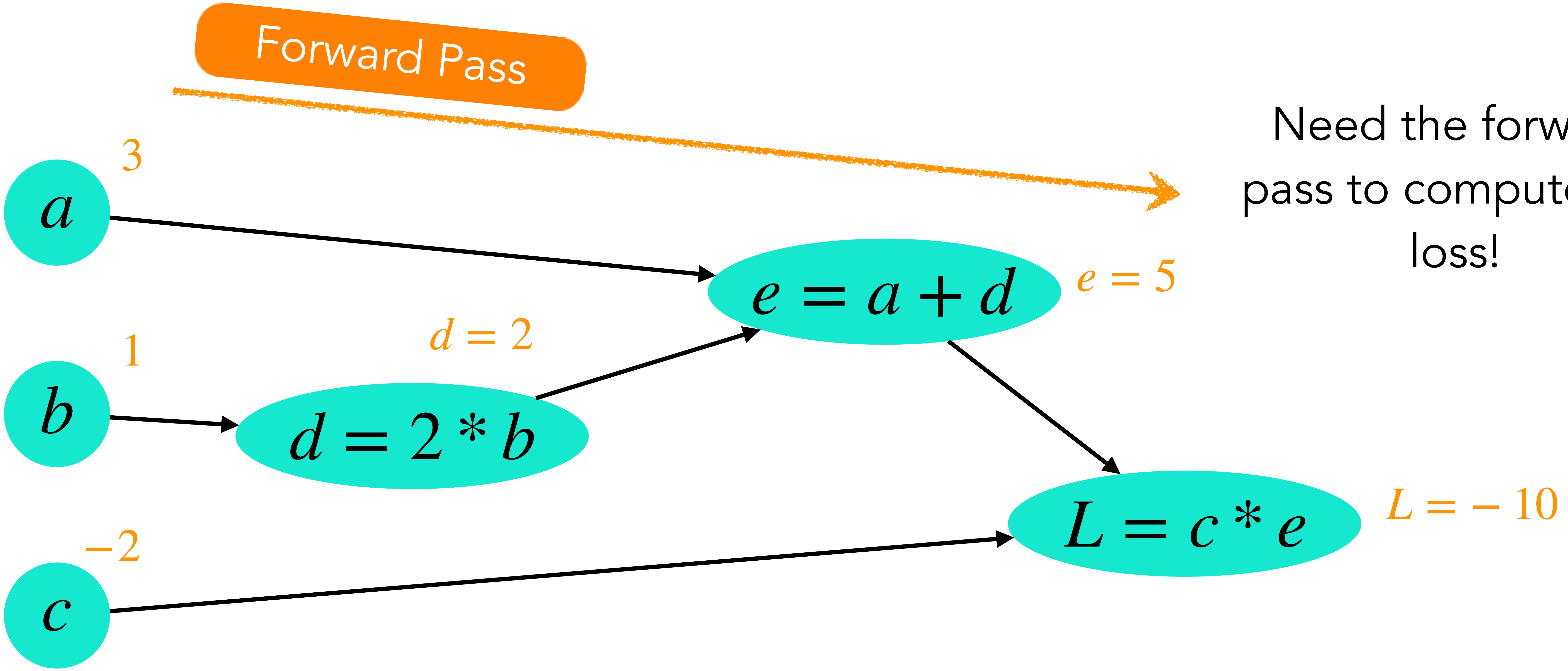


Example: Forward Pass

$d = 2 * b$

$e = a + d$

$L = c * e$



Need the forward pass to compute the loss!

But how to compute parameter updates?

Example: Backward Pass Intuition

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

$$d = 2 * b$$
$$e = a + d$$
$$L = c * e$$

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

Input Layer Gradients

Hidden Layer Gradients

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

Chain Rule of Differentiation!

USC Viterbi

The Chain Rule

Computing the derivative of a composite function:

$$f(x) = u(v(x)) \qquad \frac{\partial f}{\partial x} = \frac{\partial u}{\partial v}\frac{\partial v}{\partial x}$$

$$f(x) = u(v(w(x))) \qquad \frac{\partial f}{\partial x} = \frac{\partial u}{\partial v}\frac{\partial v}{\partial w}\frac{\partial w}{\partial x}$$

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…

Example: Backward Pass

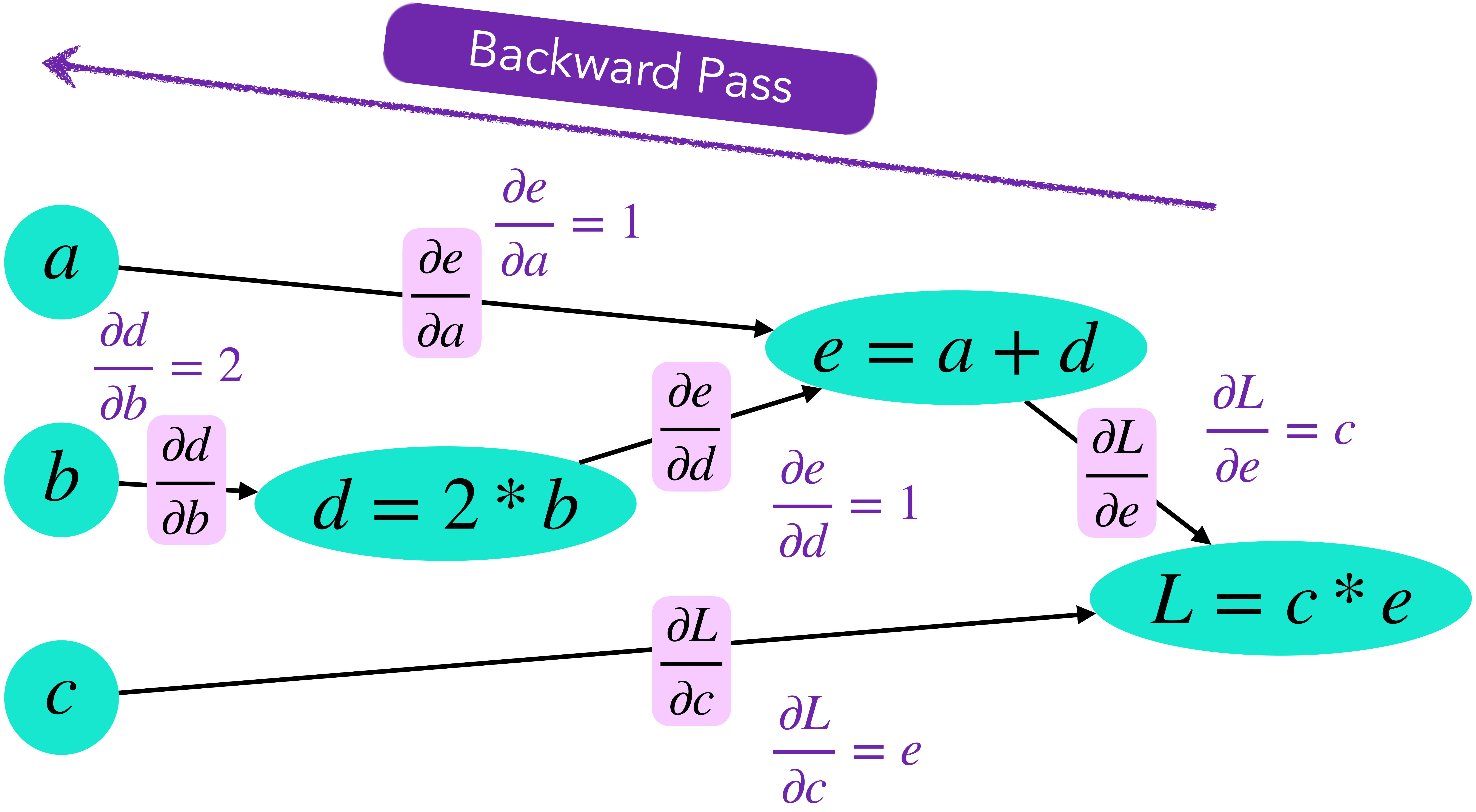
But we need the gradients of the loss with respect to parameters…

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

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

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

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



Example

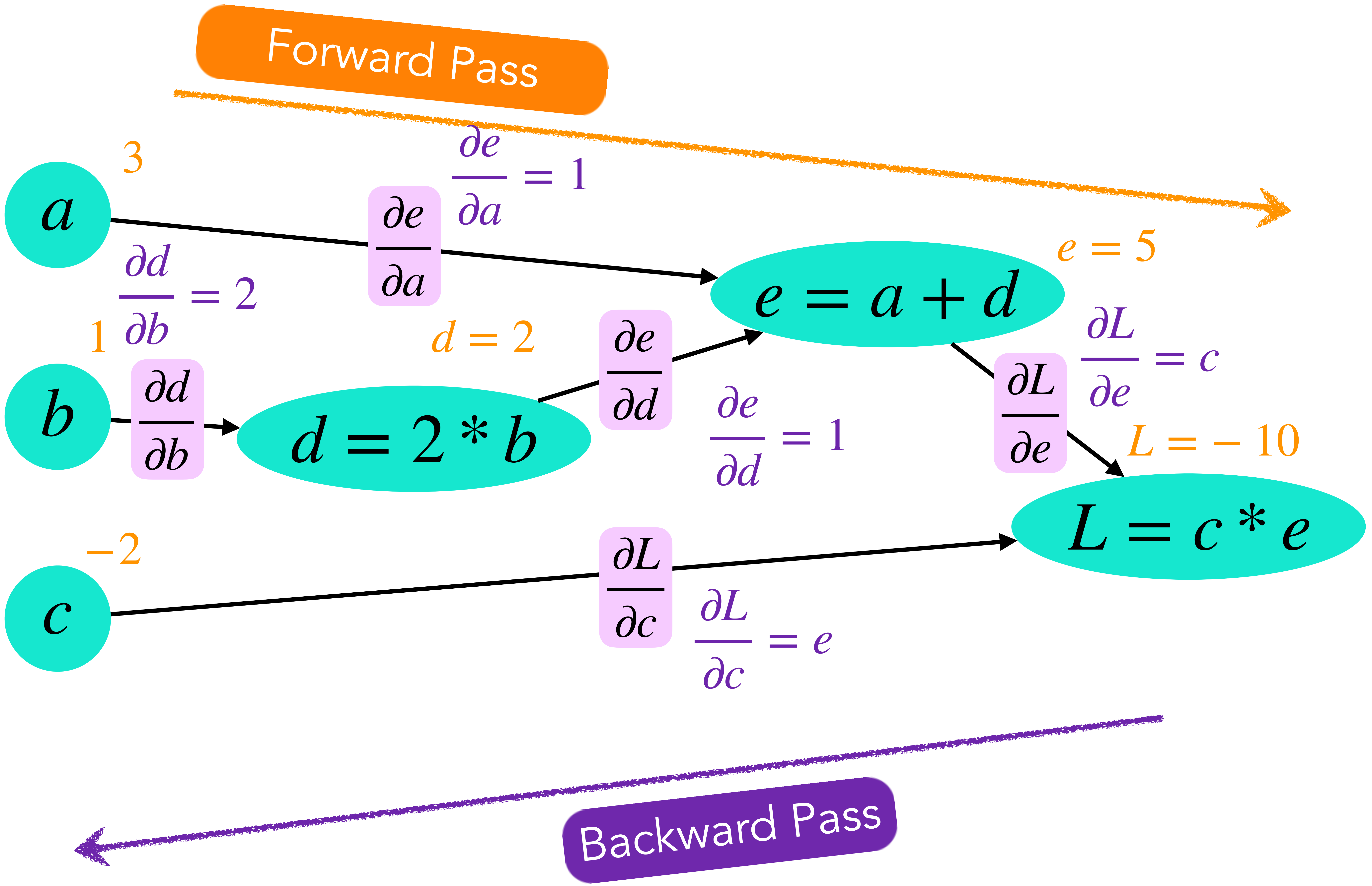
$$\frac{\partial L}{\partial e} = c = -2$$

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

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

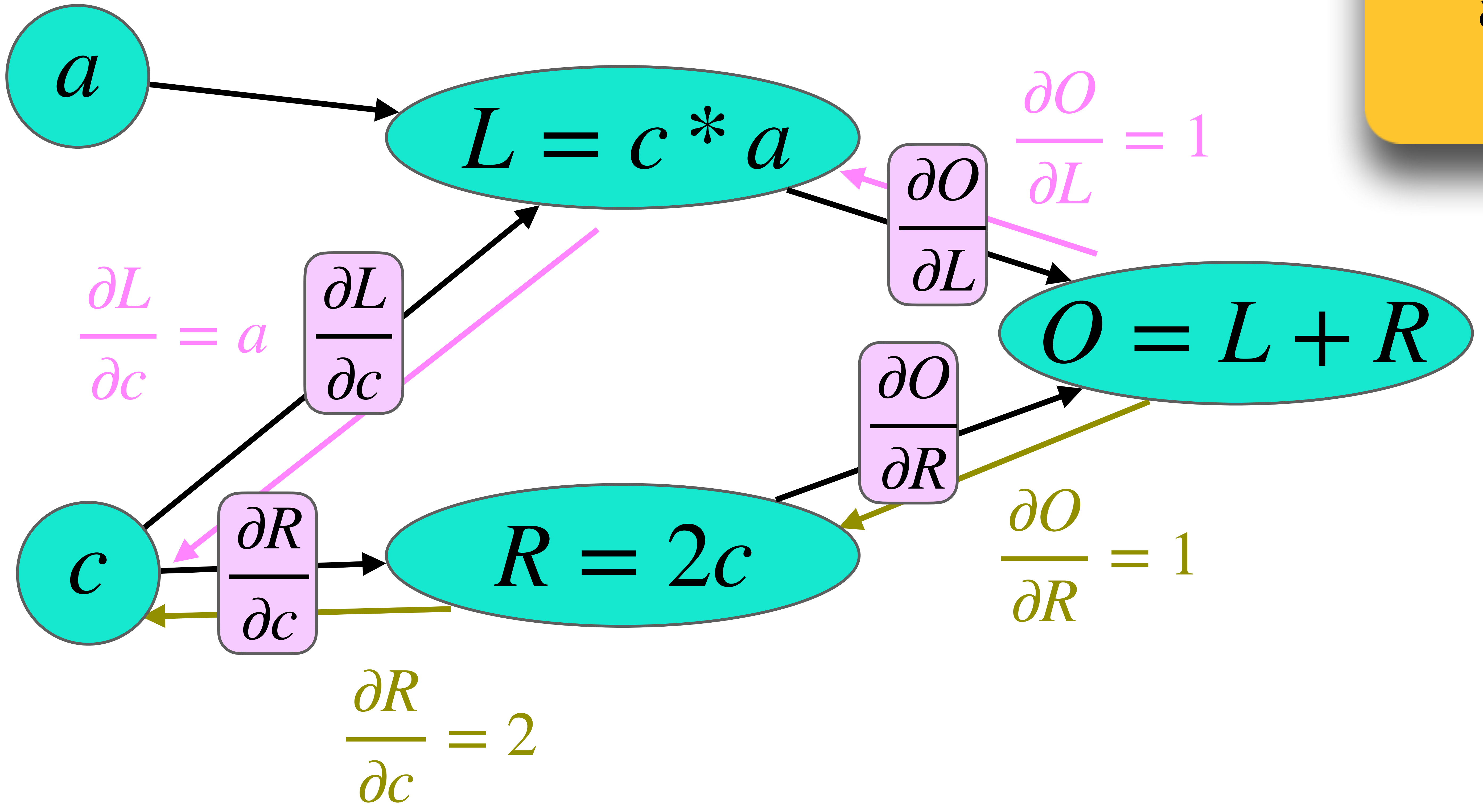
$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d} = -2$$

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



Example: Two Paths

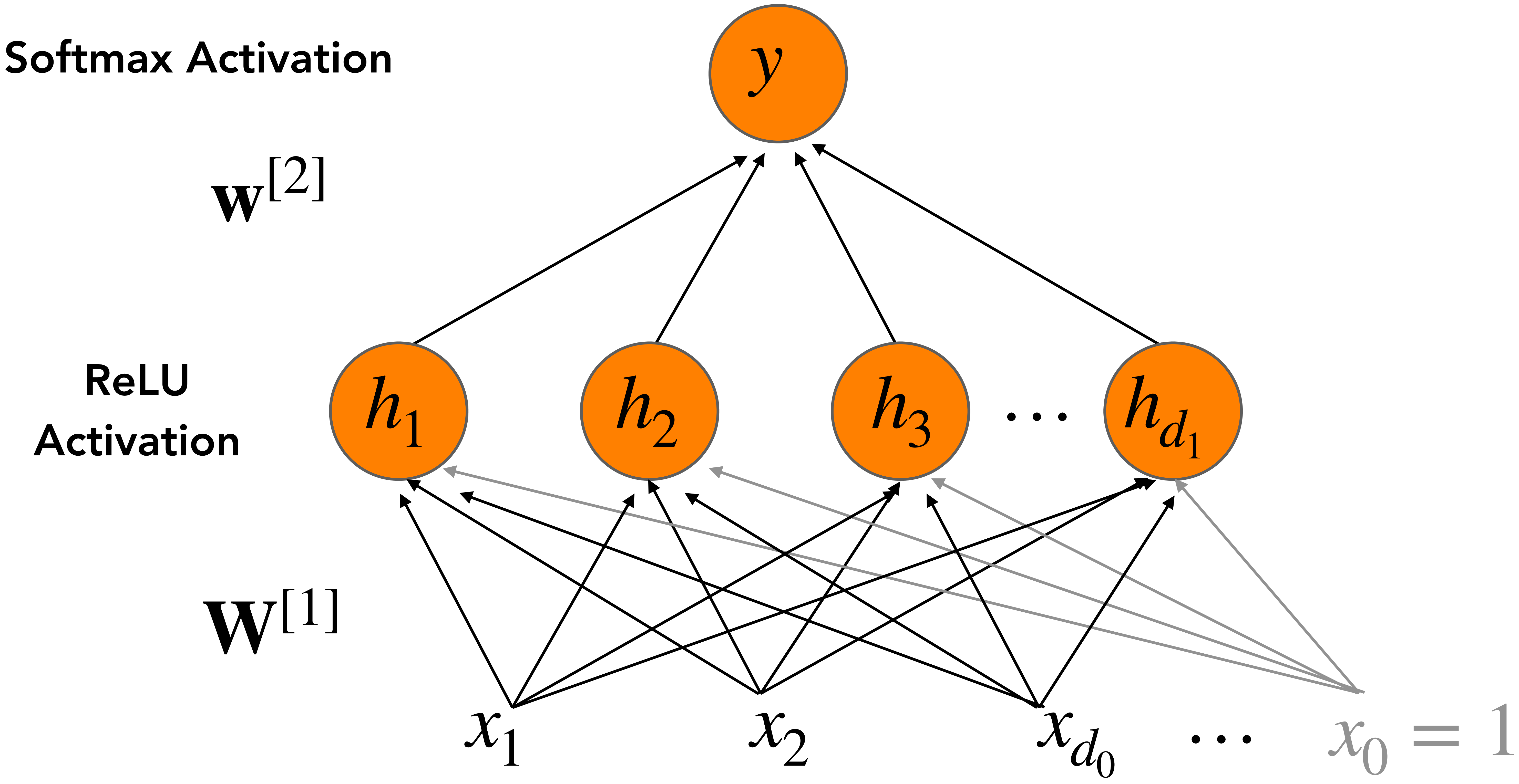
When multiple branches converge on a single node we will add these branches



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

Such cases arise when considering regularized loss functions

Backward Differentiation on a 2-layer MLP



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

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

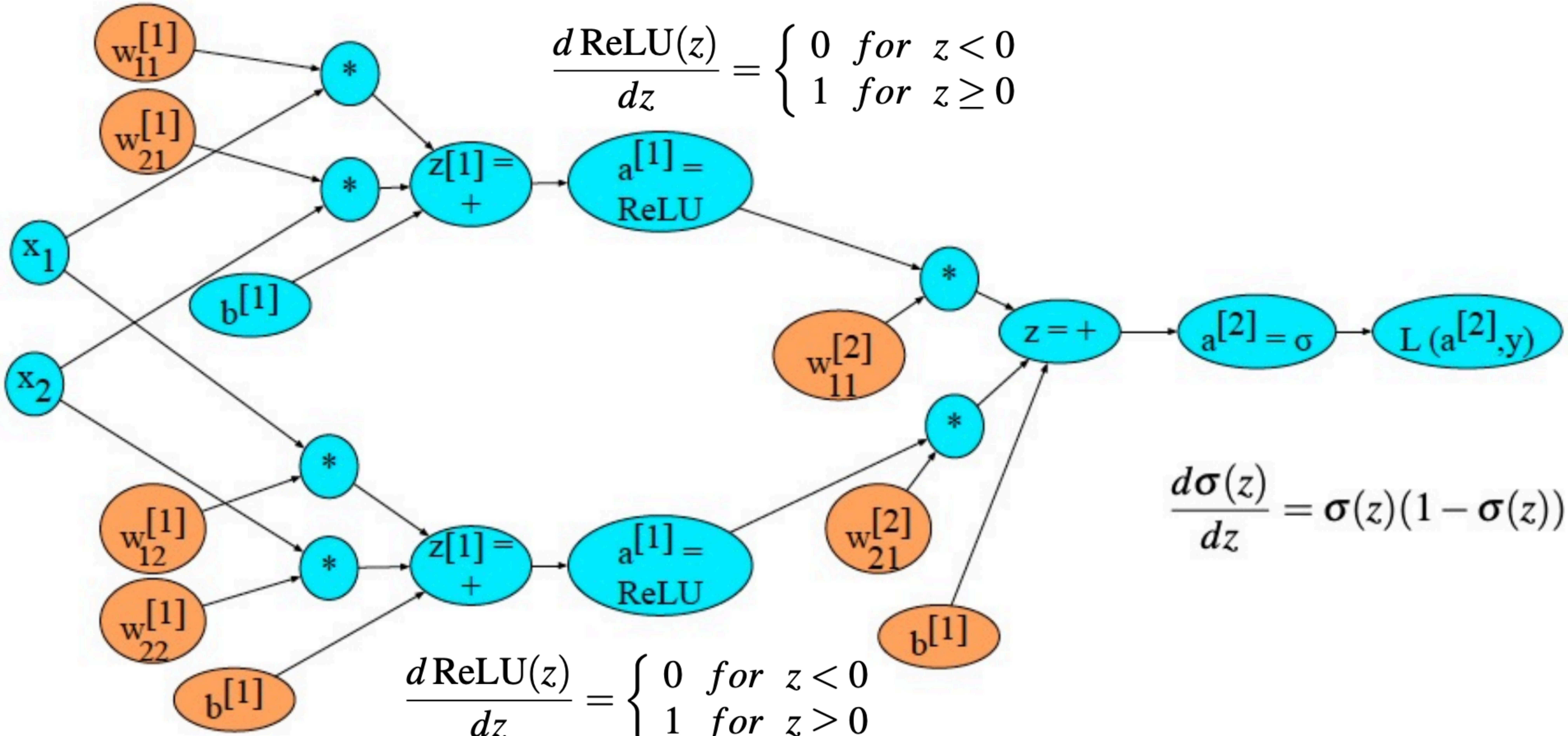
$$\mathbf{h}^{[1]} = \text{ReLU}(\mathbf{z}^{[1]})$$ **Element-wise**

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

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

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

2 layer MLP with 2 input features



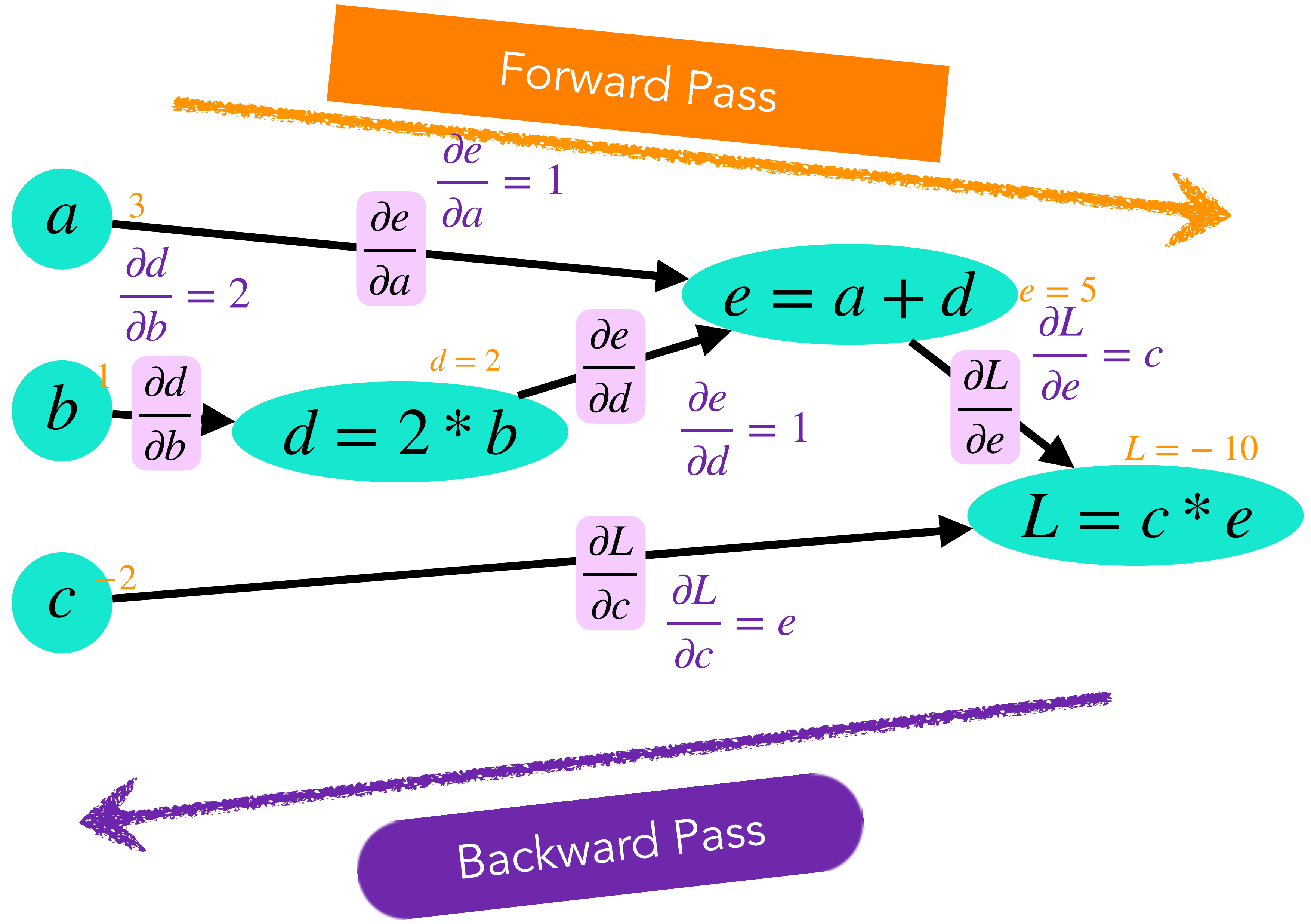
$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \quad z < 0 \\ 1 & for \quad z \geq 0 \end{cases}$$

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

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

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**
- 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

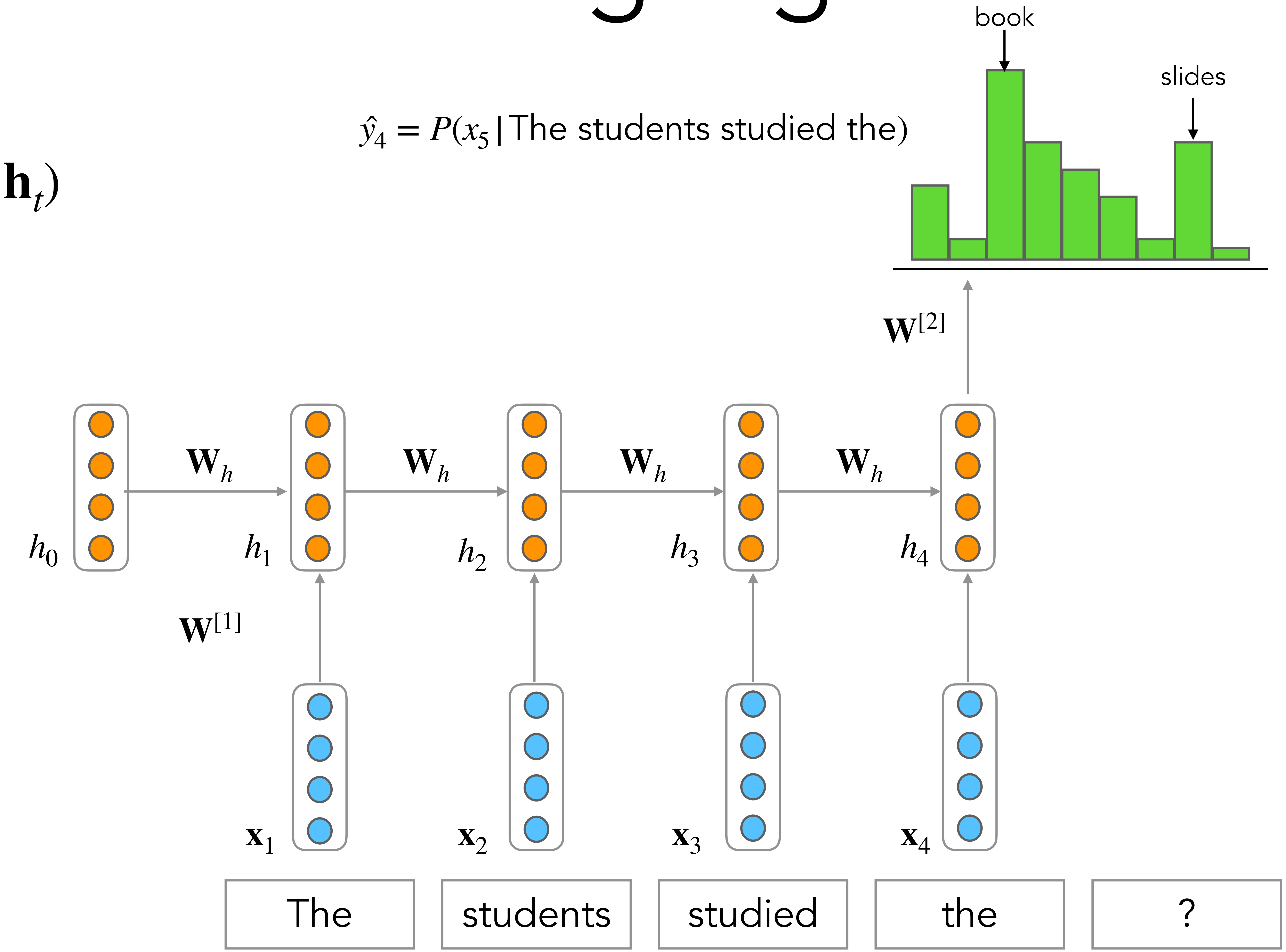
Output layer: $\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{W}^{[2]}\mathbf{h}_t)$

Hidden layer:

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

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

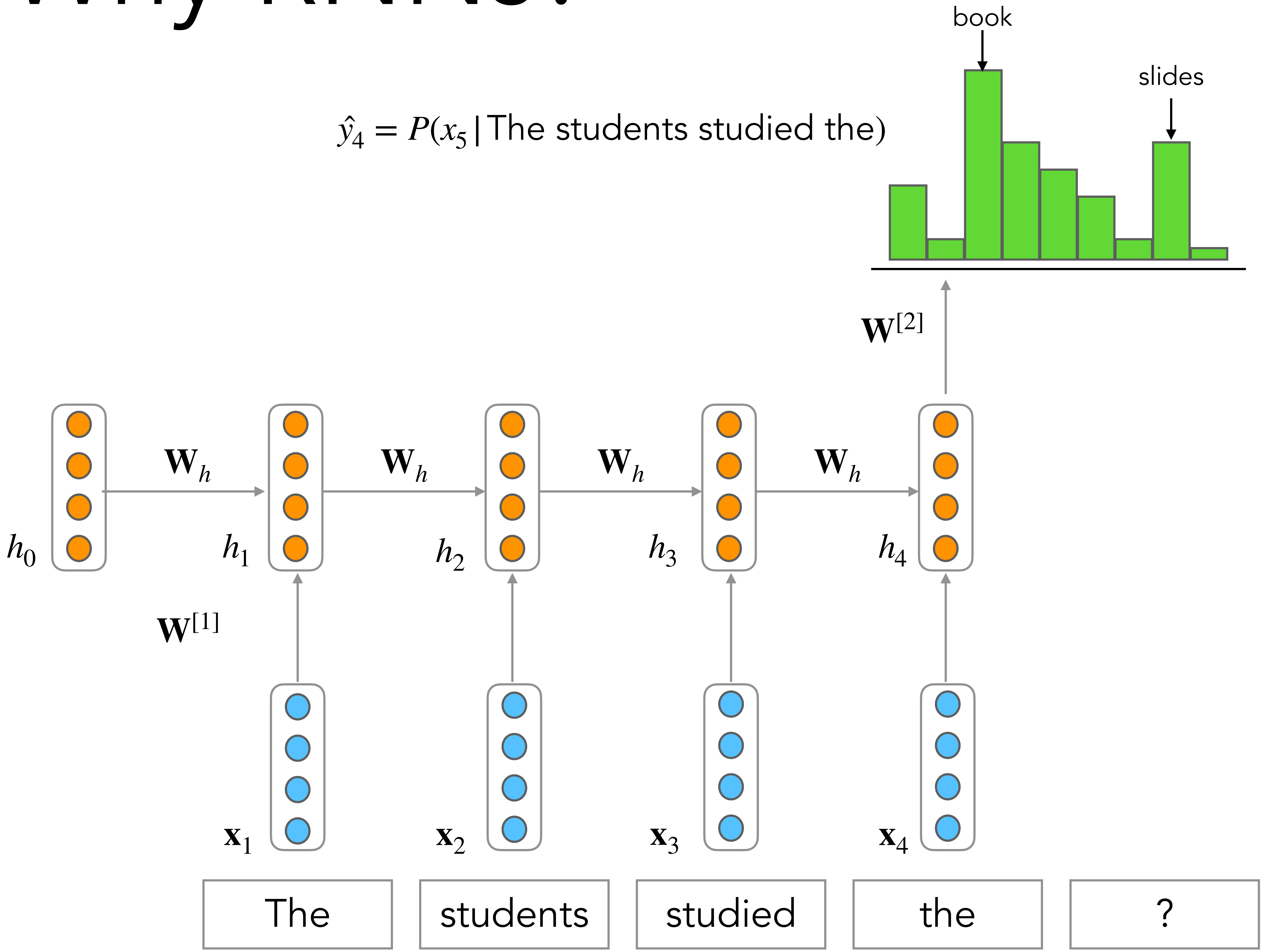
Word Embeddings, $\mathbf{x}_i$



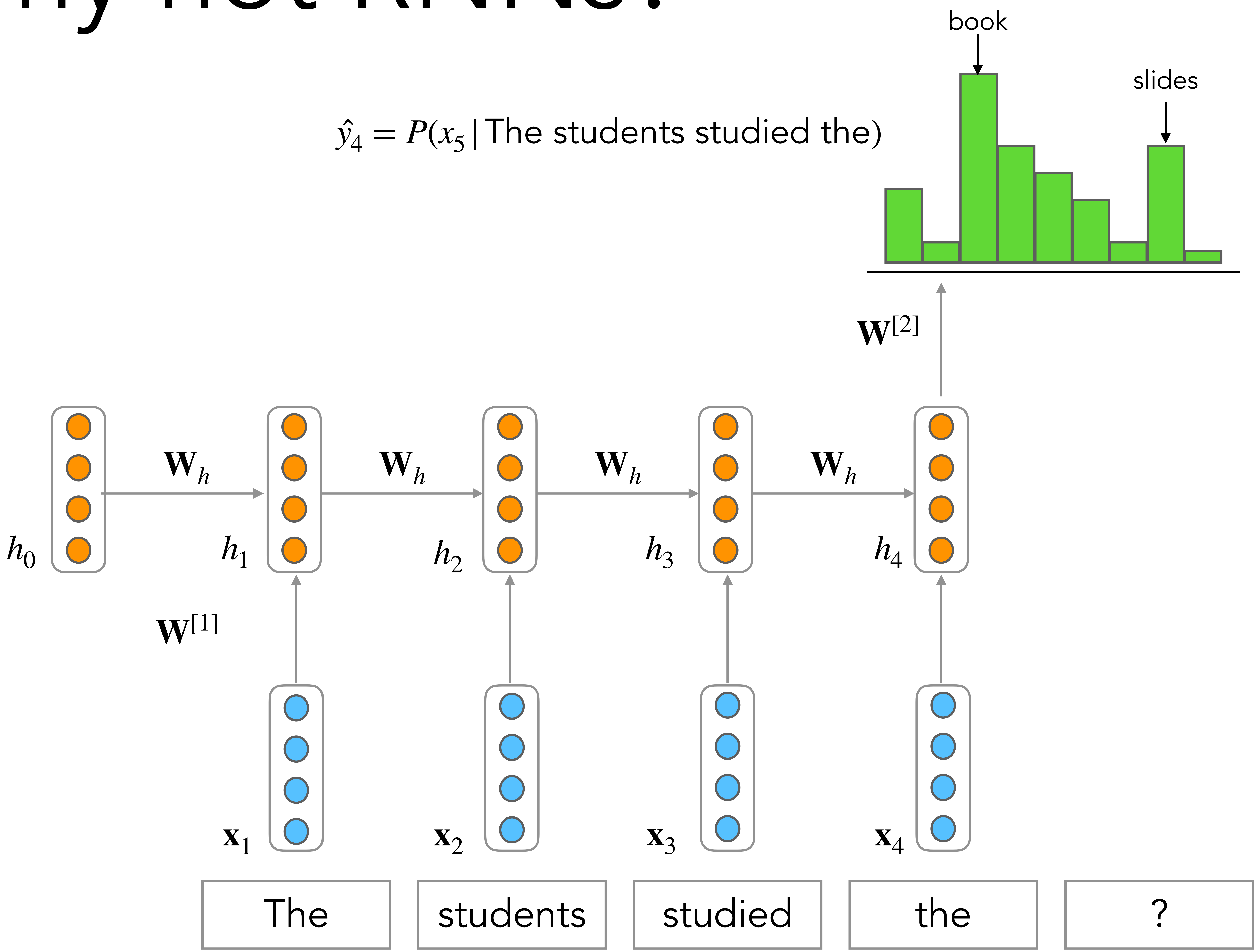
Why RNNs?

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

USC Viterbi

Concluding Thoughts

Next Class:

- More on Recurrent Neural Nets