Lecture 6:
Feedforward
Neural Network & RNN

Instructor: Xiang Ren

USC CSCI 444 NLP

2026 Spring

Slides mostly adapted from Dan Jurafsky, some from Mohit Iyyer

Ingredients of Supervised Machine Learning

I. Data as pairs $(x^{(i)}, y^{(i)}) \quad s.t \;\; i \in \{1 \ldots N\}$

- $x^{(i)}$ usually represented by a feature vector $\mathbf{x}^{(i)} = [x_1, x_2, \ldots, x_d]$,
  - e.g. word embeddings

II. Model

- A classification function that computes $\hat{y}$, the estimated class, via $p(y|x)$
  - e.g. sigmoid function: $\sigma(z) = 1/(1 + \exp(-z))$

III. Loss

- An objective function for learning
  - e.g. cross-entropy loss, L_{CE}

IV. Optimization

- An algorithm for optimizing the objective function
  - e.g. stochastic gradient descent

V. Inference / Evaluation

Learning Phase

IV. Optimization: Stochastic Gradient Descent

Our goal: minimize the loss

- Loss function is parameterized by weights: $\theta = [\mathbf{w}; b]$
- We will represent $\hat{y}$ as $f(x; \theta)$ to make the dependence on θ more obvious
- We want the weights that minimize the loss, averaged over all examples:

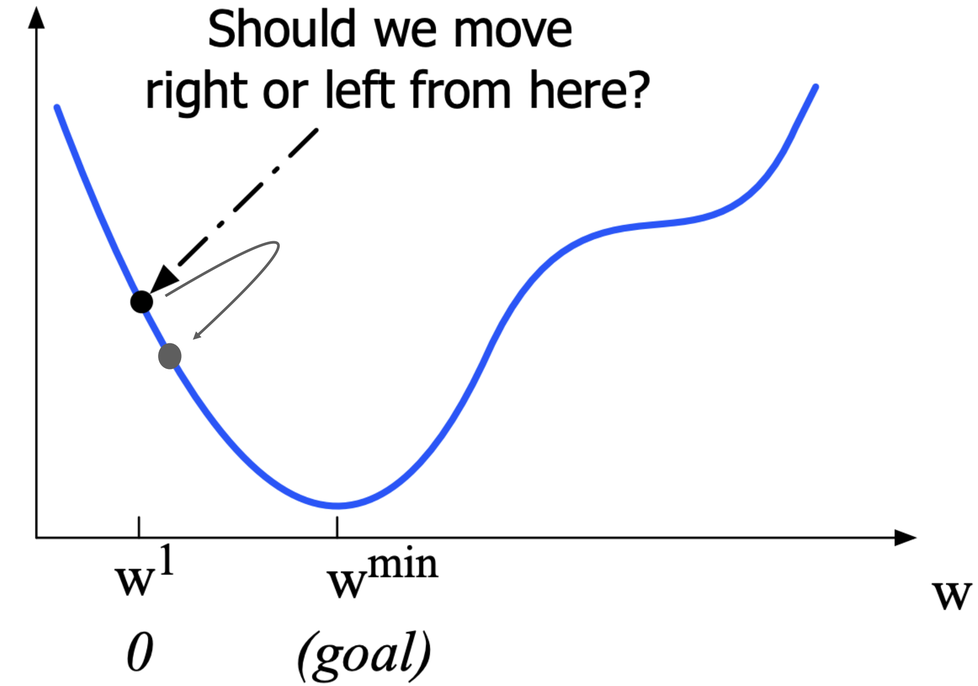
$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{\text{CE}}(f(x^{(i)}; \theta), y^{(i)})$$

Gradients

- The gradient of a function of many variables is a vector pointing in the direction of the greatest increase in a function.

Gradient Descent

- Find the gradient of the loss function at the current point and move in the opposite direction.



But by how much?

Gradient Updates

- Move the value of the gradient $\frac{\partial}{\partial w} L(f(x; w), y^*)$, weighted by a learning rate η
- Higher learning rate means move w faster

Too high: the learner will take big steps and overshoot

$$w_{t+1} = w_t - \eta \frac{\partial}{\partial w} L(f(x; w), y^*)$$

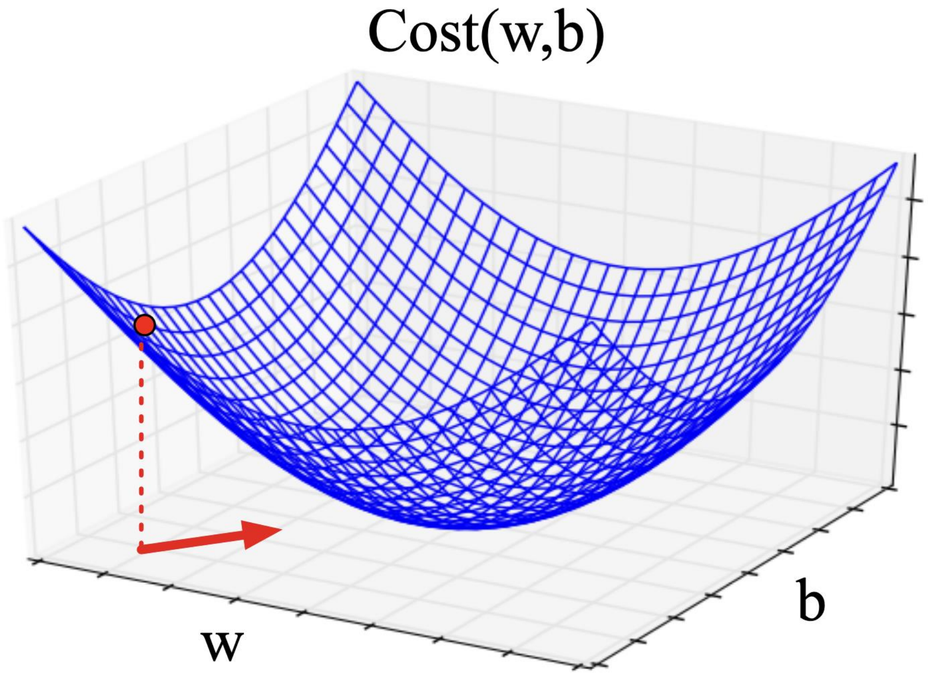
Too low: the learner will take too long

If parameter θ is a vector of d dimensions:

The gradient is just such a vector; it expresses the directional components of the sharpest slope along each of the d dimensions.

Under 2 dimensions

- Consider 2 dimensions, w and b:
- Visualizing the gradient vector at the red point
- It has two dimensions shown in the x-y plane



Real-life gradients

- Are much longer; lots and lots of weights!
- For each dimension θ_i the gradient component i tells us the slope with respect to that variable
  - "How much would a small change in θ_i influence the total loss function L?"
  - We express the slope as a partial derivative ∂ of the loss $\partial\theta_i$
  - The gradient is then defined as a vector of these partials

Real-life gradients

We will represent $\hat{y}$ as $f(x;\theta)$ to make the dependence on θ more obvious

$$\nabla_\theta L(f(x;\boldsymbol{\theta}),y)) = \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(x;\boldsymbol{\theta}),y) \\ \frac{\partial}{\partial w_2} L(f(x;\boldsymbol{\theta}),y) \\ \vdots \\ \frac{\partial}{\partial w_n} L(f(x;\boldsymbol{\theta}),y) \end{bmatrix}$$

The final equation for updating θ at time step $t+1$ based on the gradient is thus

$$\theta_{t+1} = \theta_t - \eta \frac{\partial}{\partial \theta} L(f(x;\theta),y)$$

Gradients for Logistic Regression

Case 1: Sentiment Analysis

Recall: the cross-entropy loss for logistic regression

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

Derivatives have a closed form solution:

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

Gradients: word2vec

Case 2: Word2Vec

$$L_{CE} = -[\log\sigma(\mathbf{w} \cdot \mathbf{c}_{pos}) + \sum_{j=1}^{K} \log\sigma(-\mathbf{w} \cdot \mathbf{c}_{neg_j})]$$

3 different parameters

$$\frac{\partial L_{CE}}{\partial \mathbf{c}_{pos}} = [\sigma(\mathbf{c}_{pos} \cdot \mathbf{w}) - 1]\mathbf{w}$$

$$\frac{\partial L_{CE}}{\partial \mathbf{c}_{neg_j}} = [\sigma(\mathbf{c}_{neg_j} \cdot \mathbf{w})]\mathbf{w}$$

$$\frac{\partial L_{CE}}{\partial w} = [\sigma(\mathbf{c}_{pos} \cdot \mathbf{w}) - 1]\mathbf{c}_{pos} + \sum_{j=1}^{K} [\sigma(\mathbf{c}_{neg_j} \cdot \mathbf{w})]\mathbf{c}_{neg_j}$$

Update the parameters by subtracting respective η-weighted gradients

$$w_{t+1} = w_t - \eta[[\sigma(\mathbf{c}_{pos} \cdot \mathbf{w}) - 1]\mathbf{c}_{pos} + \sum_{j=1}^{K} [\sigma(\mathbf{c}_{neg_j} \cdot \mathbf{w})]\mathbf{c}_{neg_j}]$$

Pseudocode

- function STOCHASTIC GRADIENT DESCENT ($L(), f(), x, y$) returns θ
  - # where: L is the loss function
  - # f is a function parameterized by θ
  - # x is the set of training inputs $x^{(1)}, x^{(2)}, \ldots x^{(N)}$
  - # y is the set of training outputs (labels) $y^{(1)}, y^{(2)}, \ldots y^{(N)}$
  - $\theta \leftarrow 0$ (or randomly initialized)
  - repeat till done
  - for each training tuple $(x^{(i)}, y^{(i)})$: (in random order)

Stochastic Gradient Descent

1. Compute $\hat{y}^{(i)} = f(x^{(i)}; \theta)$ # What is our estimated output $\hat{y}^{(i)}$?
2. Compute the loss $L(\hat{y}^{(i)}, y^{(i)})$ # How far off is $\hat{y}^{(i)}$ from the true output $y^{(i)}$?
3. $g \leftarrow \nabla L(f(x^{(i)}; \theta), y^{(i)})$ # How should we move θ to maximize loss?
4. $\theta \leftarrow \theta - \eta g$ # Go the other way instead

  - return θ

Mini-Batching

function STOCHASTIC GRADIENT DESCENT $(L(), f(), x, y, m)$ returns θ

where: L is the loss function

f is a function parameterized by θ

x is the set of training inputs $x^{(1)}, x^{(2)}, \ldots x^{(N)}$

y is the set of training outputs (labels) $y^{(1)}, y^{(2)}, \ldots y^{(N)}$ and m is the mini-batch size

$\theta \leftarrow 0$ (or randomly initialized)

repeat till done

for each randomly sampled minibatch of size m:

1. for each training tuple $(x^{(i)}, y^{(i)})$ in the minibatch: (in random order)
   i. Compute $\hat{y}^{(i)} = f(x^{(i)}; \theta)$ # What is our estimated output $\hat{y}^{(i)}$?
   ii. Compute the loss $L_{mini} \leftarrow L_{mini} + L(\hat{y}^{(i)}, y^{(i)})$ # How far off is $\hat{y}^{(i)}$ from the true output $y^{(i)}$?
2. $g \leftarrow \frac{1}{m} \nabla L_{mini}(f(x^{(i)}; \theta), y^{(i)})$ # How should we move θ to maximize loss?
3. $\theta \leftarrow \theta - \eta g$ # Go the other way instead

return θ

Multinomial Logistic Regression

Multinomial Logistic Regression

- Often we need more than 2 classes
  - Positive / negative / neutral sentiment of a document
  - Parts of speech of a word (noun, verb, adjective, adverb, preposition, etc.)
  - Actionable classes for emergency SMSs
- If >2 classes we use **multinomial** logistic regression
  - = Softmax regression
  - So "logistic regression" will just mean binary (2 output classes)

Multinomial Logistic Regression

The probability of everything must still sum to 1

$$P(+|x) + P(-|x) + P(\sim |x) = 1$$

Need a generalization of the sigmoid called the softmax

- Takes a vector $\mathbf{z} = [z_1, z_2, \ldots, z_K]$ of K arbitrary values
- Outputs a probability distribution
  - each value in the range $[0,1]$
  - all the values summing to 1

Softmax

The Softmax Function

Turns a vector $\mathbf{z} = [z_1, z_2, \ldots, z_K]$ of K arbitrary values into probabilities

$$softmax(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{K} \exp(z_j)} \qquad 1 \leq i \leq K$$

The denominator $\sum_{i=1}^{K} \exp(z_i)$ is used to normalize all the values into probabilities.

$$softmax(\mathbf{z}) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{K} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{K} \exp(z_i)}, \ldots, \frac{\exp(z_K)}{\sum_{i=1}^{K} \exp(z_i)}\right]$$

Softmax: Example

Turns a vector $\mathbf{z} = [z_1, z_2, \ldots, z_K]$ of K arbitrary values into probabilities

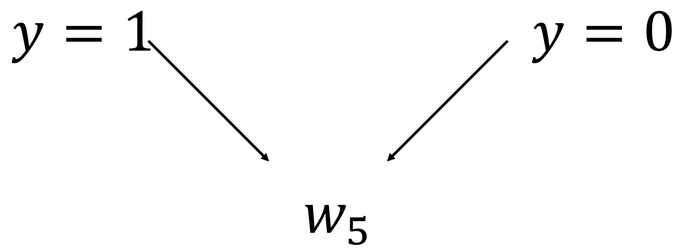
$$\mathbf{z} = [0.6,1.1,1.5,1.2,3.2,1.1]$$

$$softmax(\mathbf{z}) = [\frac{\exp(z_1)}{\sum_{i=1}^{K}\exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{K}\exp(z_i)}, \ldots, \frac{\exp(z_K)}{\sum_{i=1}^{K}\exp(z_i)}]$$

$$softmax(\mathbf{z}) = [0.055,0.090,0.0067,0.10,0.74,0.010]$$

Binary versus Multinomial

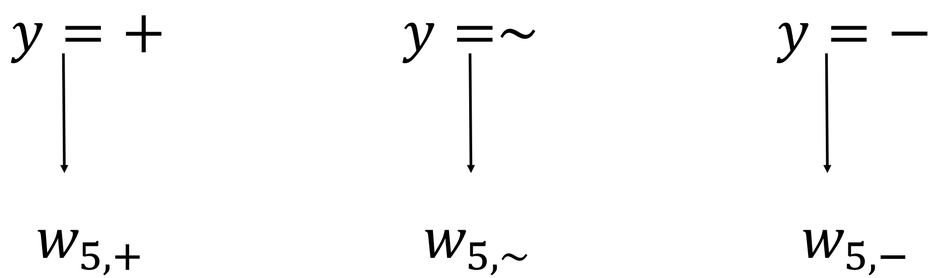
Binary Logistic Regression



$$x_5 = \begin{cases} 1 & \text{if } \text{“!”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases} \qquad w_5 = 3.0$$

Why do we not need a different weight for each class in binary logistic regression?

Multinomial Logistic Regression



separate weights for each class

Feature	Definition	$w_{5,+}$	$w_{5,-}$	$w_{5,0}$
$f_5(x)$	$\begin{cases} 1 & \text{if } \text{“!”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	3.5	3.1	−5.3

Softmax in multinomial logistic regression

Parameters are now a matrix $\mathbf{W} \in \mathbb{R}^{d \times K}$ and $b \in \mathbb{R}^1$

$$P(y = c|\mathbf{x}; \theta) = \frac{\exp(\mathbf{w}_c \cdot \mathbf{x} + b)}{\sum_{j=1}^{K} \exp(\mathbf{w}_j \cdot \mathbf{x} + b)}$$

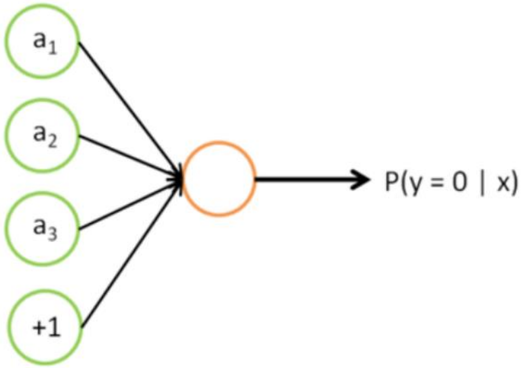
- Input is still the dot product between weight vector $\mathbf{w}_c$ and input vector $\mathbf{x}$, offset by b
- But separate weight vectors for each of the K classes, each of dimension d

Multinomial LR Loss:

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

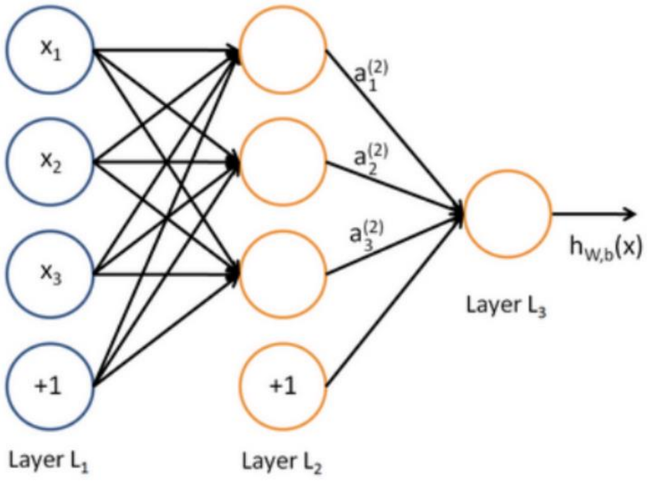
Concluding Thoughts

- Logistic Regression
  - A very simple neural network



Logistic Regression

- Next: Let's add some hidden layers
  - Feed-Forward Neural Nets

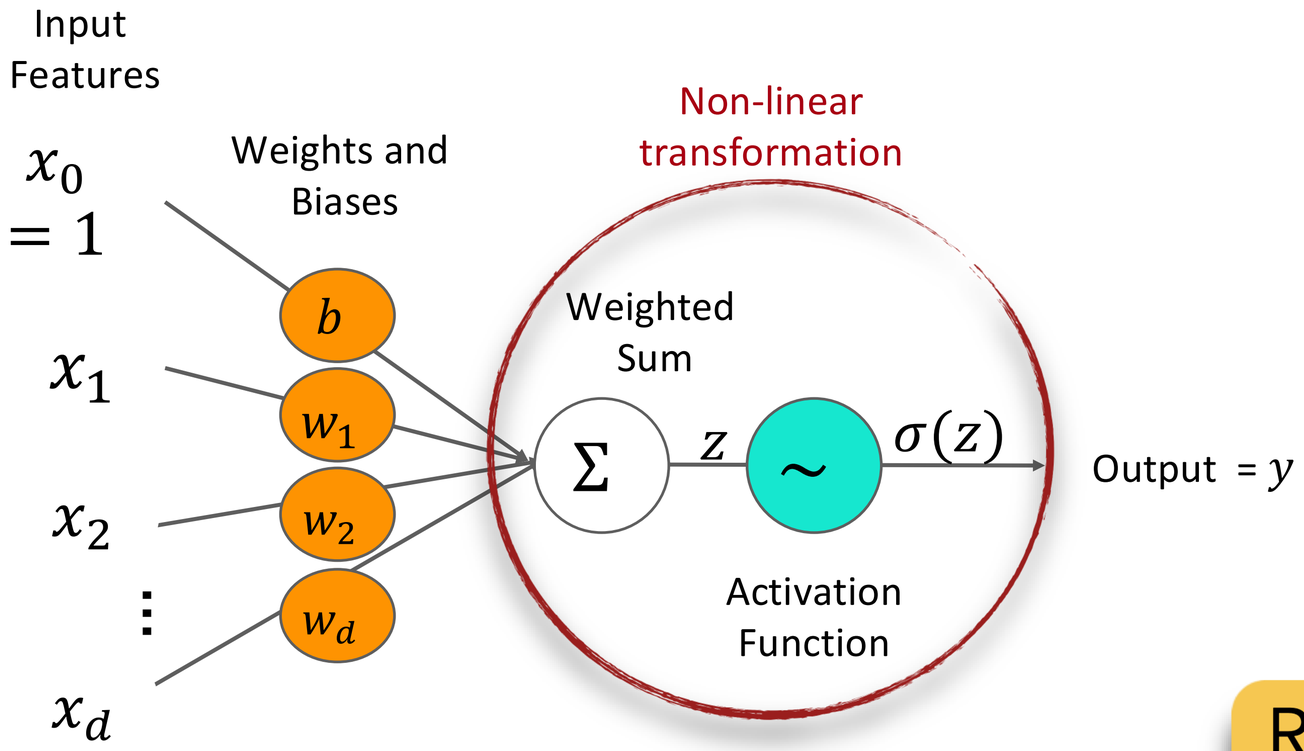


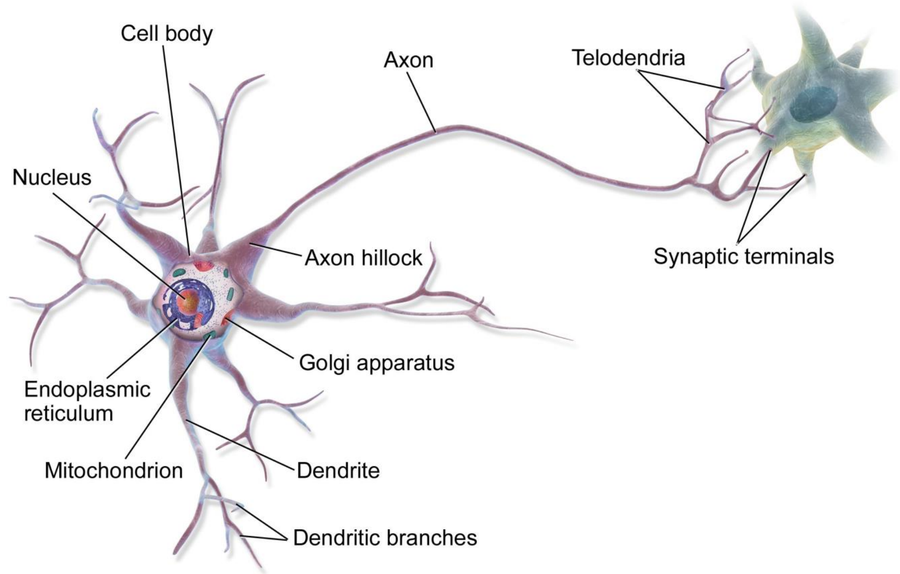
Neural Network

Feed-Forward Neural Networks

Neural Network Unit

Logistic Regression is a very simple neural network

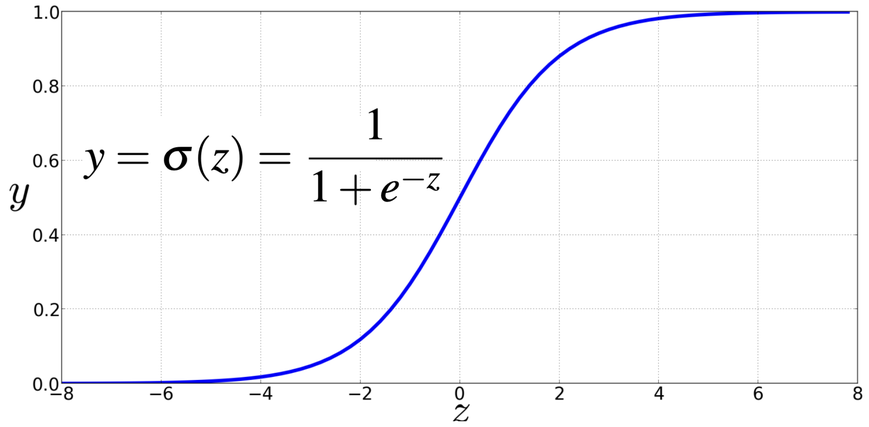


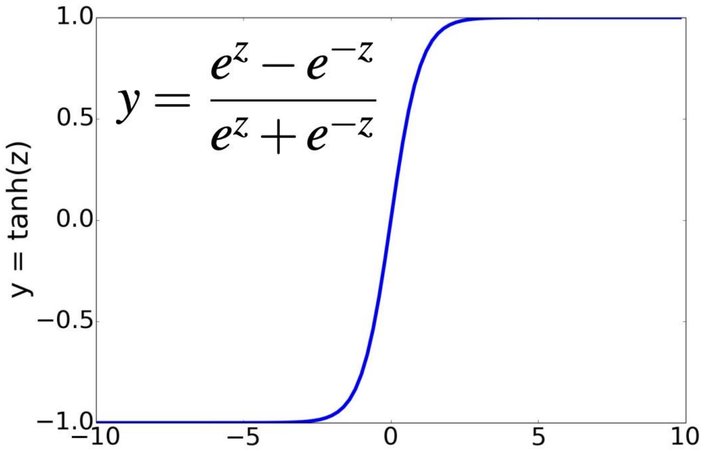


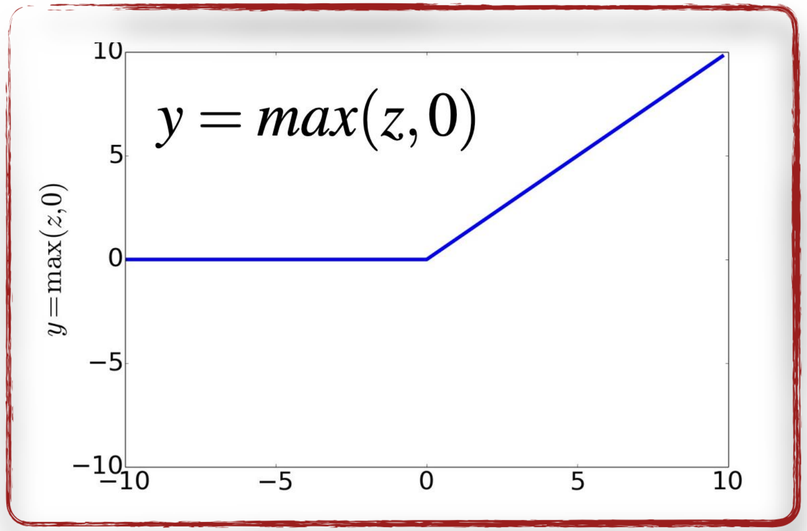
Resembles a neuron in the brain!

Non-Linear Activation Functions

Most common!







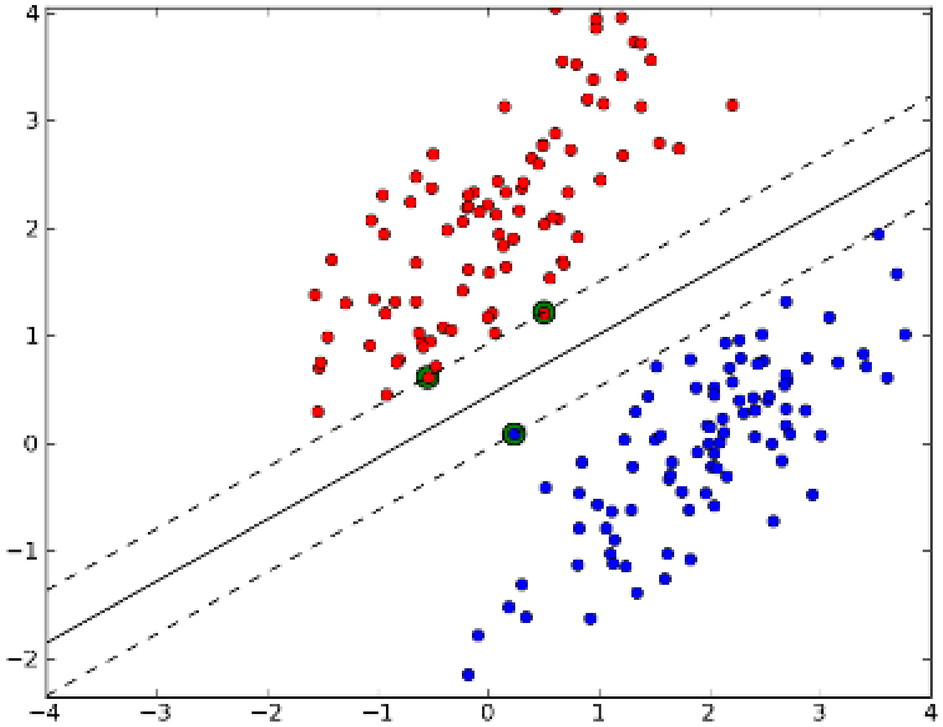
sigmoid

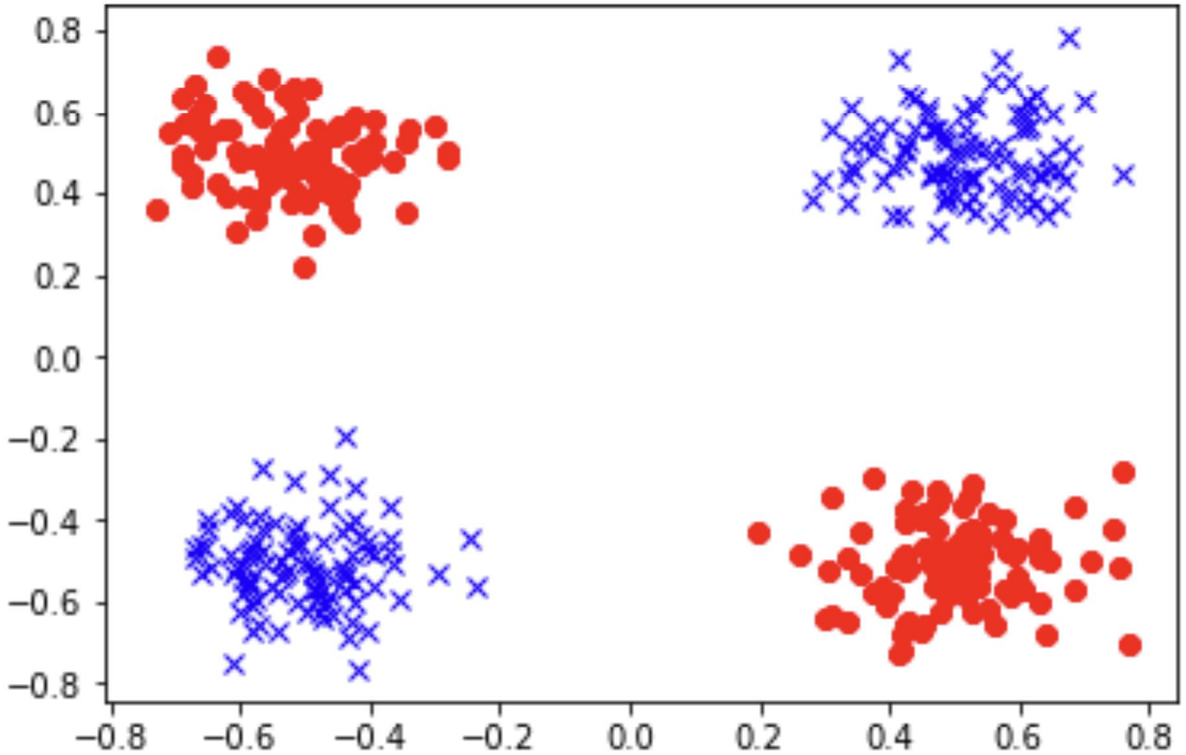
tanh

relu (Rectified Linear Unit)

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

Linear vs. Non-linear Functions

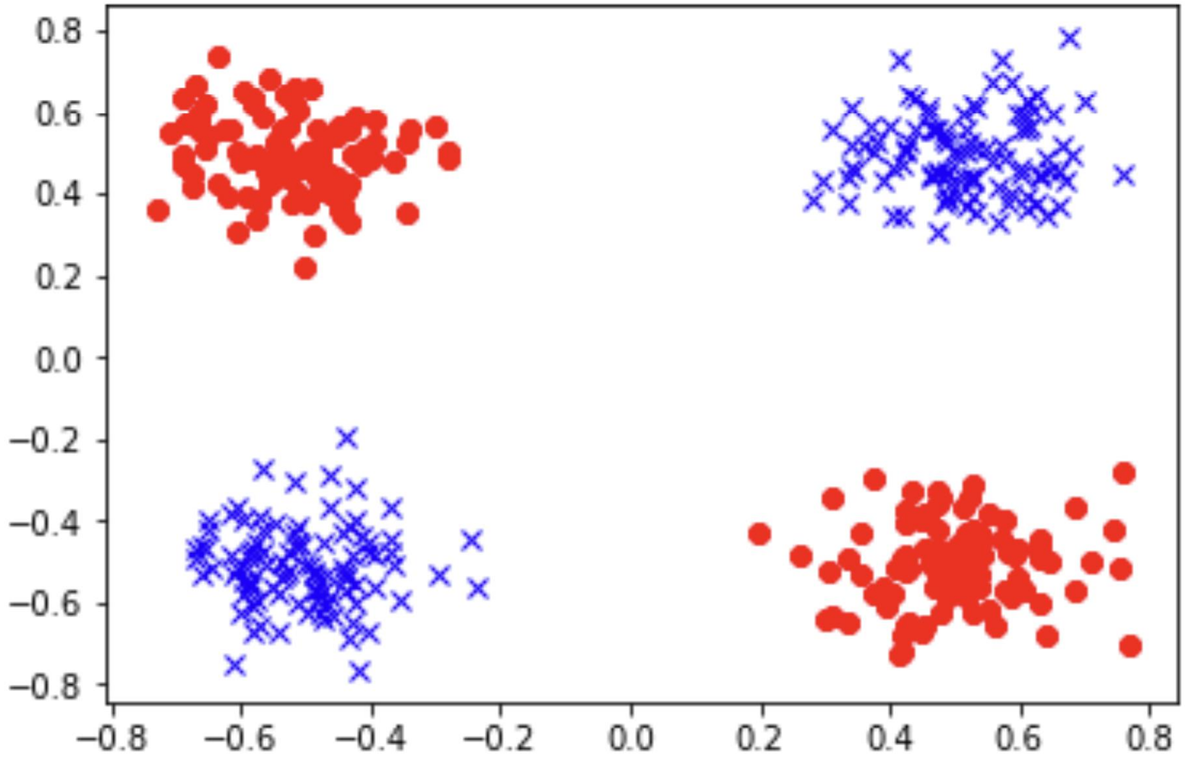




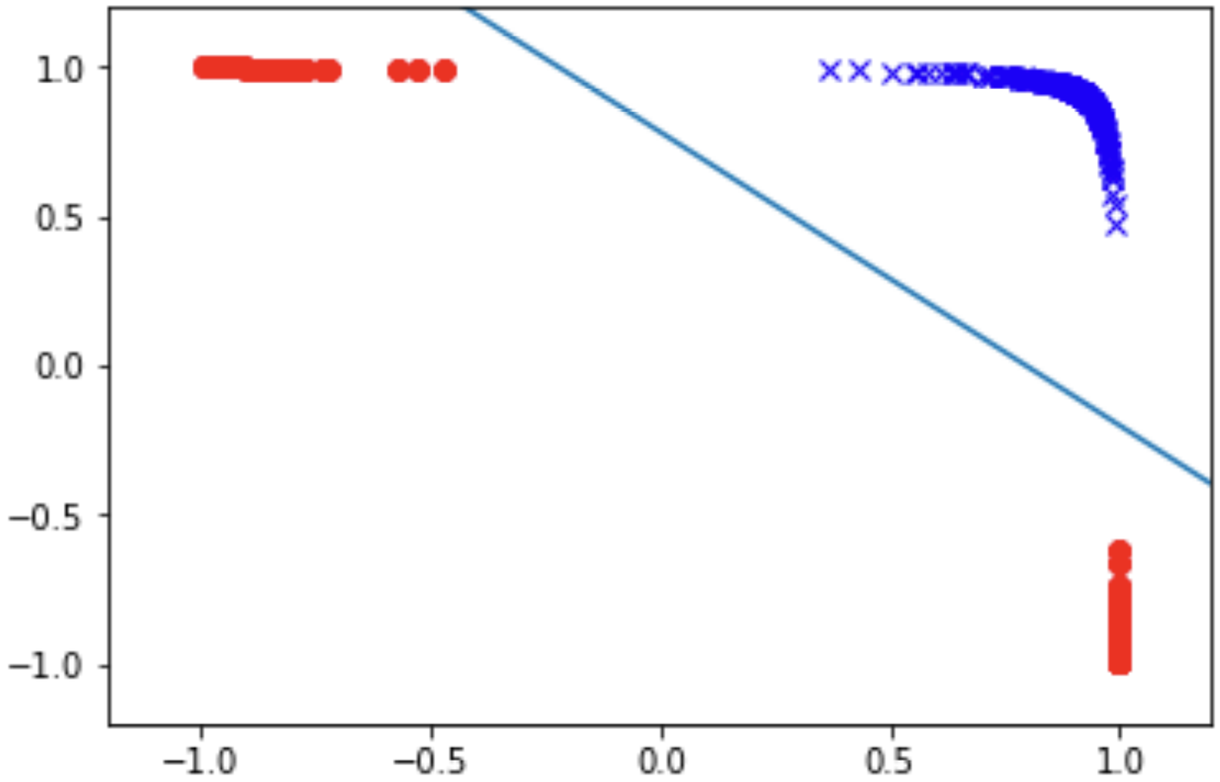
Linearly inseparable

Power of non-linearity

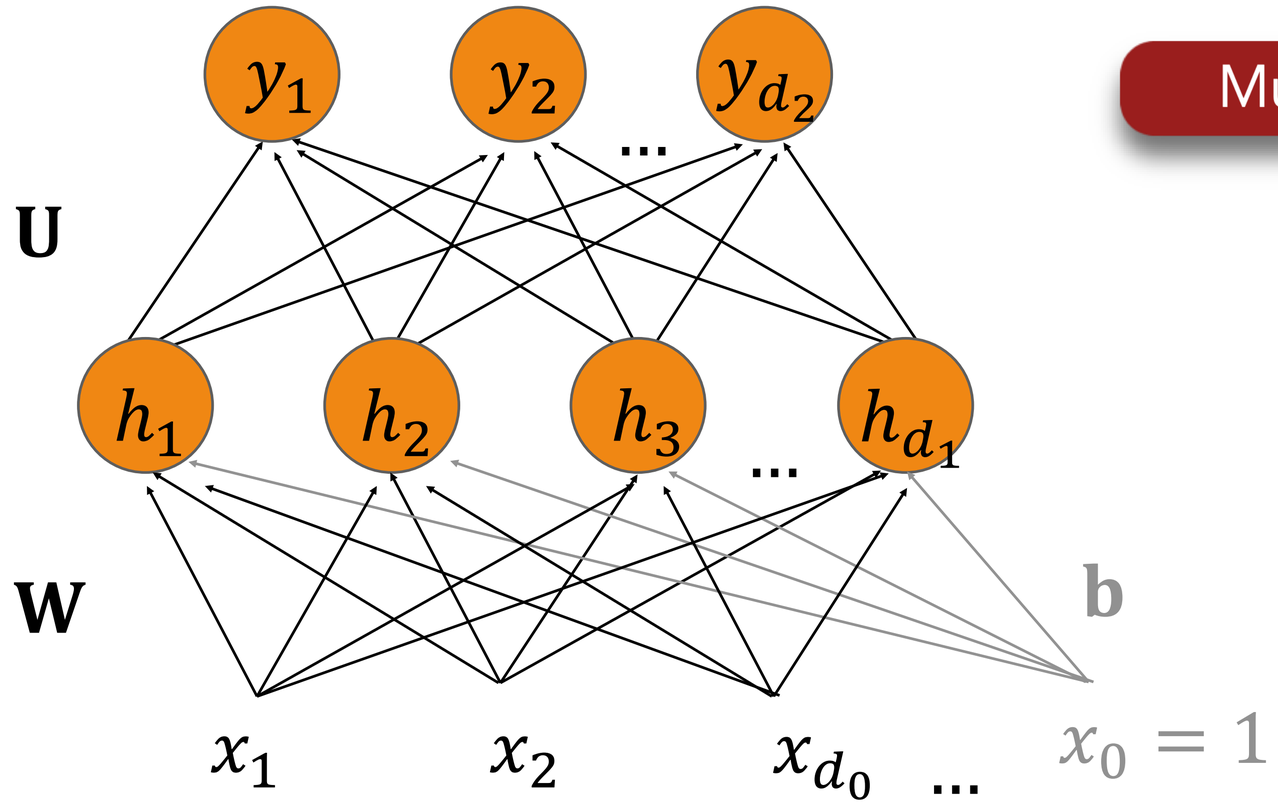
$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



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



Feedforward Neural Nets

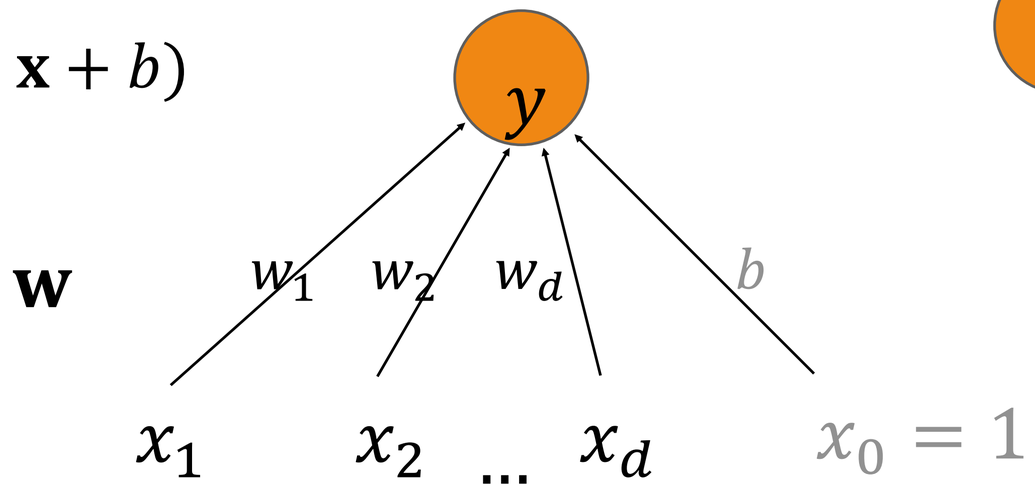


Multilayer Perceptron

Let's break it down by revisiting our logistic regression model

Binary Logistic Regression

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



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

1-layer Network

we don't count the input layer in counting layers!

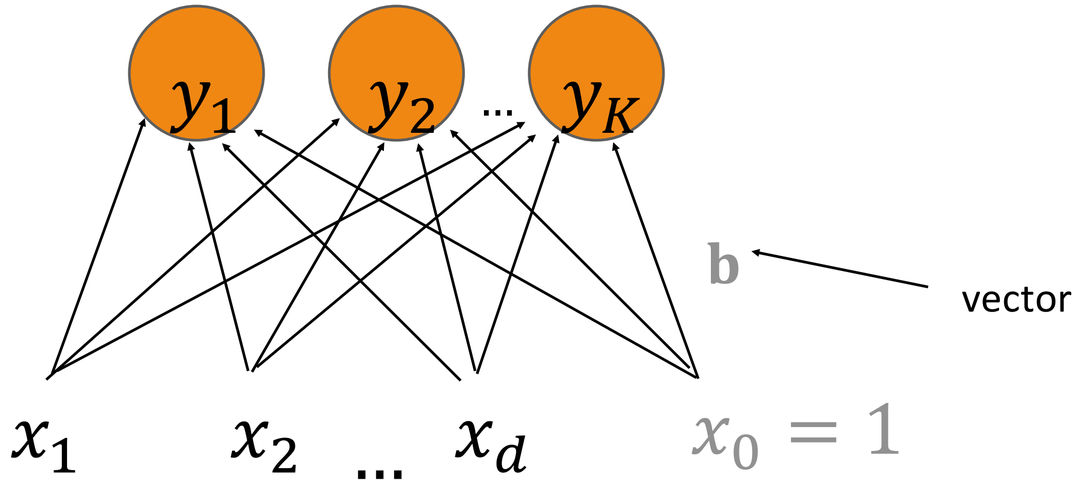
Multinomial Logistic Regression

$$softmax(\mathbf{z}) = [\frac{\exp(z_1)}{\sum_{i=1}^{K}\exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{K}\exp(z_i)}, \ldots, \frac{\exp(z_K)}{\sum_{i=1}^{K}\exp(z_i)}]$$

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

matrix → $\mathbf{W}$

Input layer: vector $\mathbf{x}$



1-layer Network

Fully connected single layer network

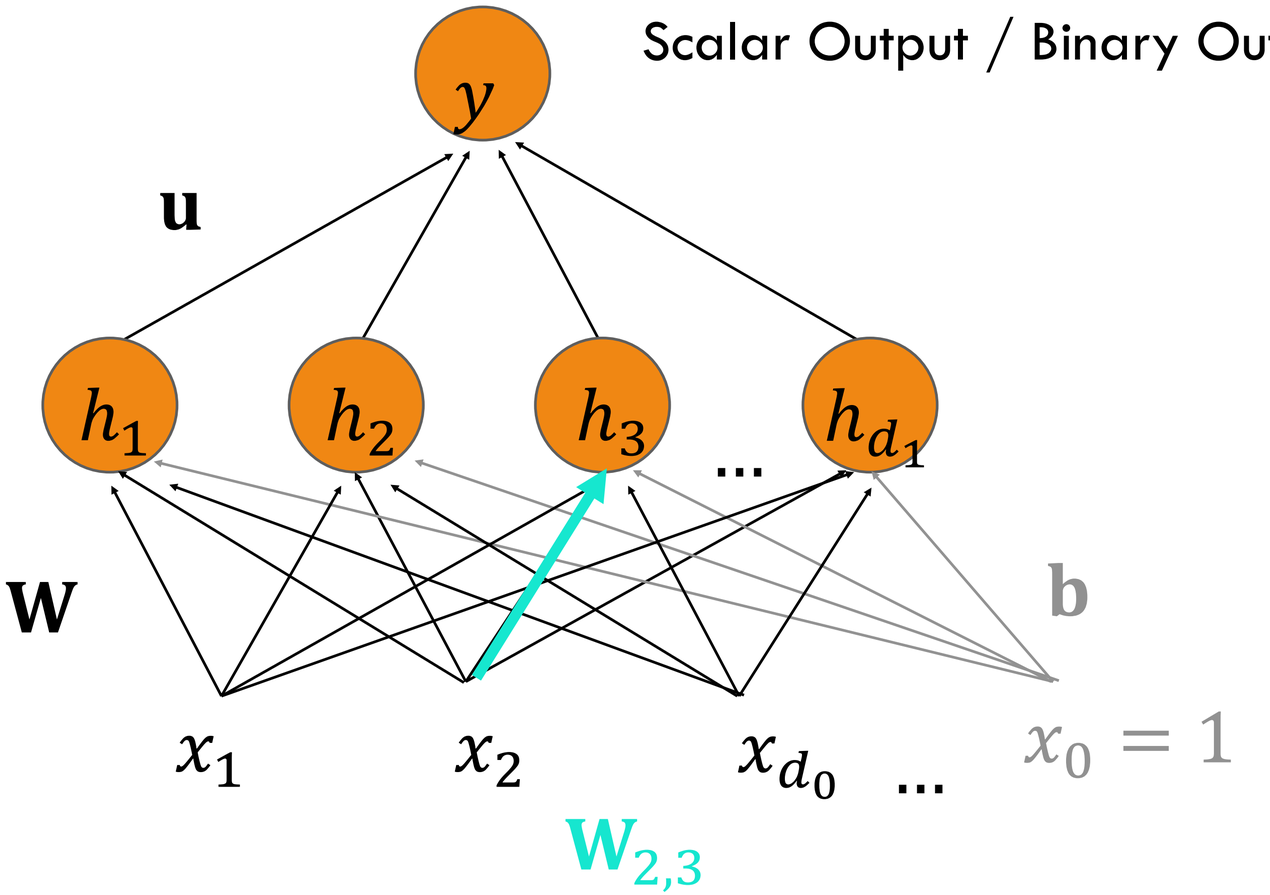
Two-layer Feedforward Network

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

Hidden layer: $\mathbf{h} = g(\mathbf{Wx} + \mathbf{b})$

Usually ReLU or tanh

Input layer: vector $\mathbf{x}$



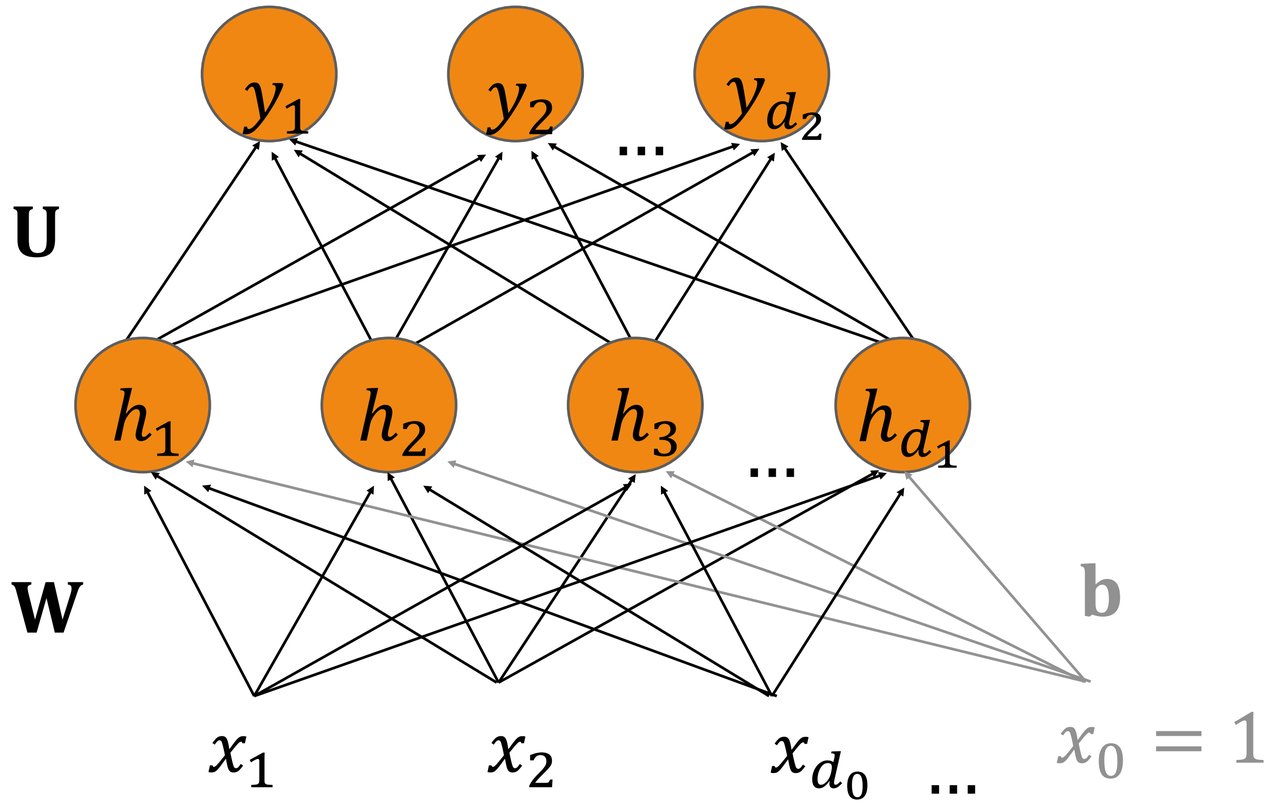
Two-layer Feedforward Network with Softmax Output

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

Hidden layer: $\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$

Usually ReLU or tanh

Input layer: vector $\mathbf{x}$



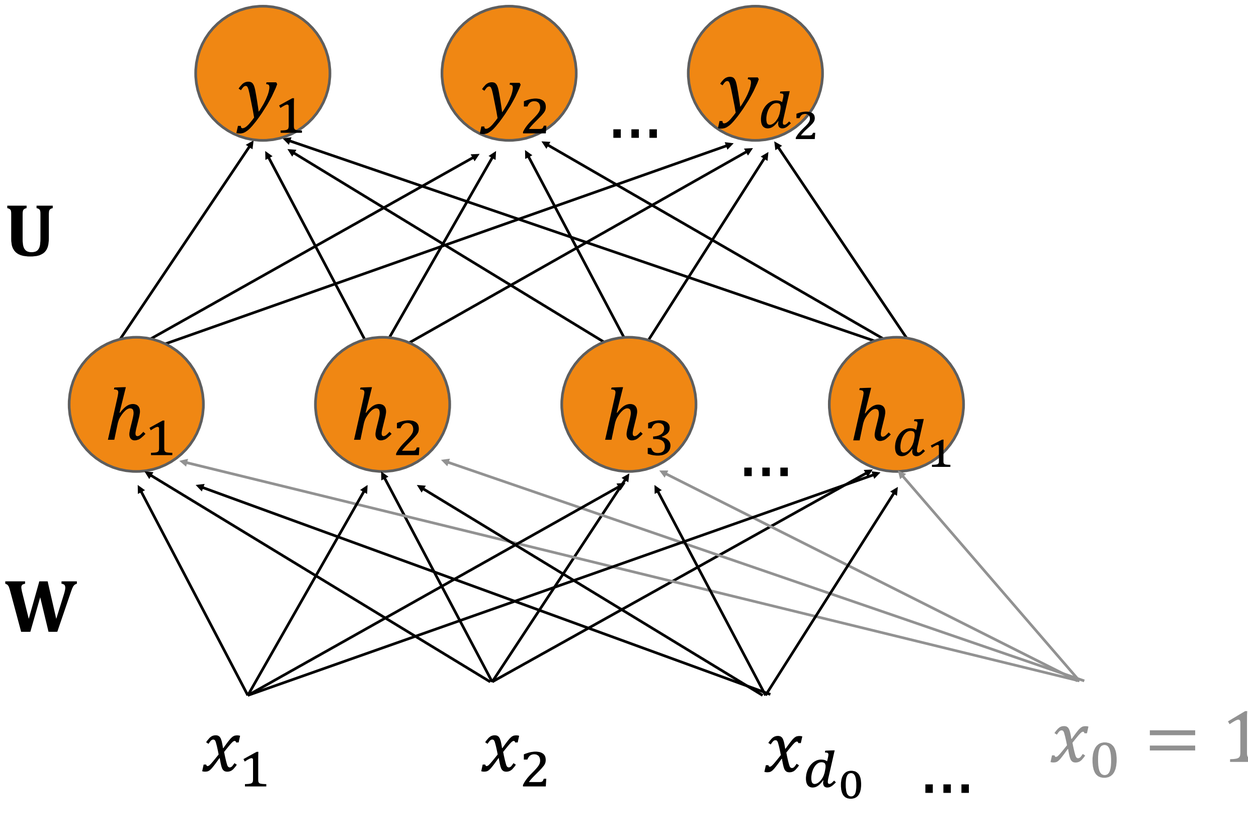
Two-layer FFNN: Notation

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

Hidden layer: $\mathbf{h} = g(\mathbf{W}\mathbf{x}) = g(\sum_{i=0}^{d_0} W_{ji} x_i)$

Usually ReLU or tanh

Input layer: vector $\mathbf{x}$



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

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
  - Remember the overfitting problem in n-gram LMs?
  - In general, count-based methods can never do as well as optimization-based ones

Why?

- State-of-the-art neural LMs are based on more powerful neural network technology like Transformers
- But simple feedforward LMs can do almost as well!

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….

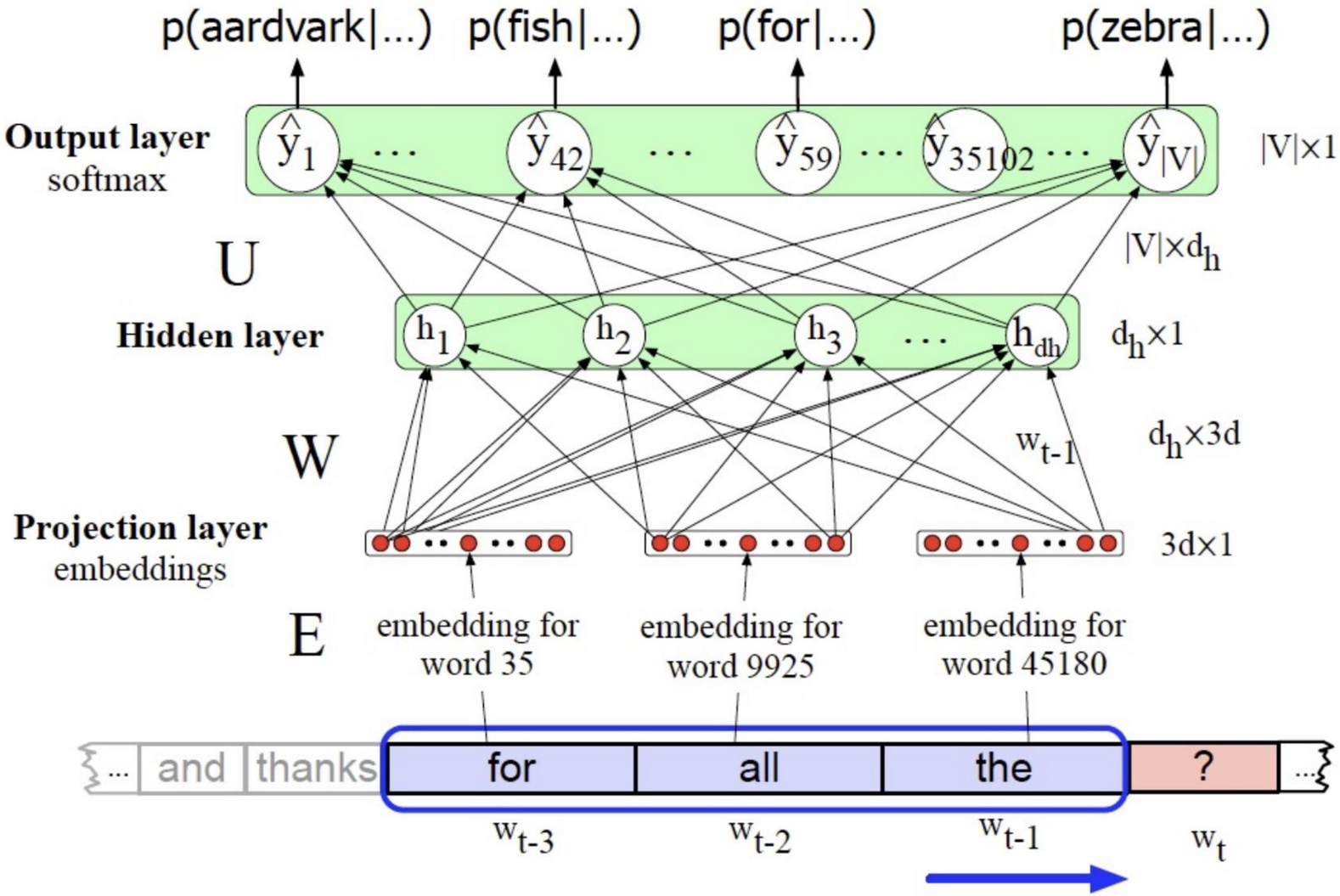
Solution: Sliding windows (of fixed length)

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

Where does this come from?

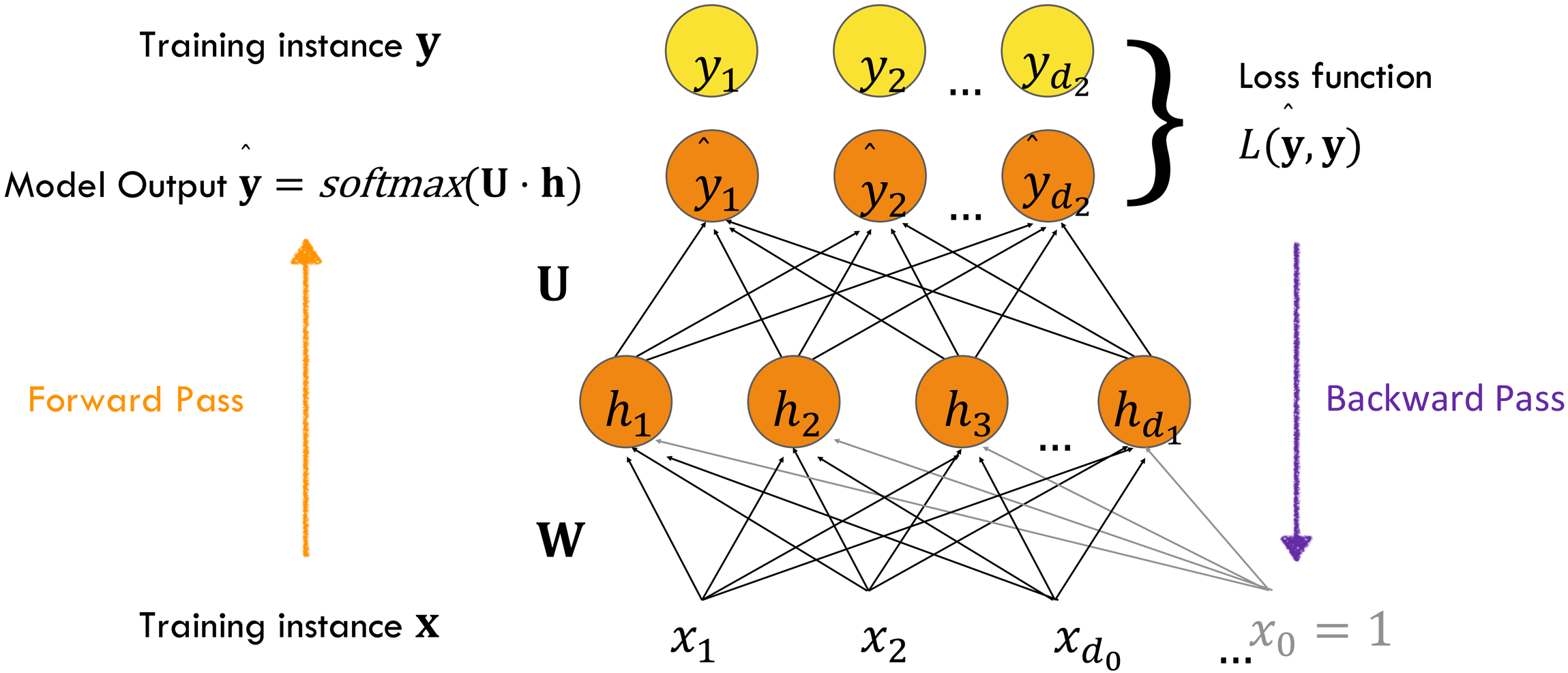
Feedforward Neural LM

- Sliding window of size 3
- Every feature in the embedding vector connected to every single hidden unit



Training FFNNs and Backprop

Intuition: Training a 2-layer Network



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
      - Update the weight
    - 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
        - Update the weight

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!

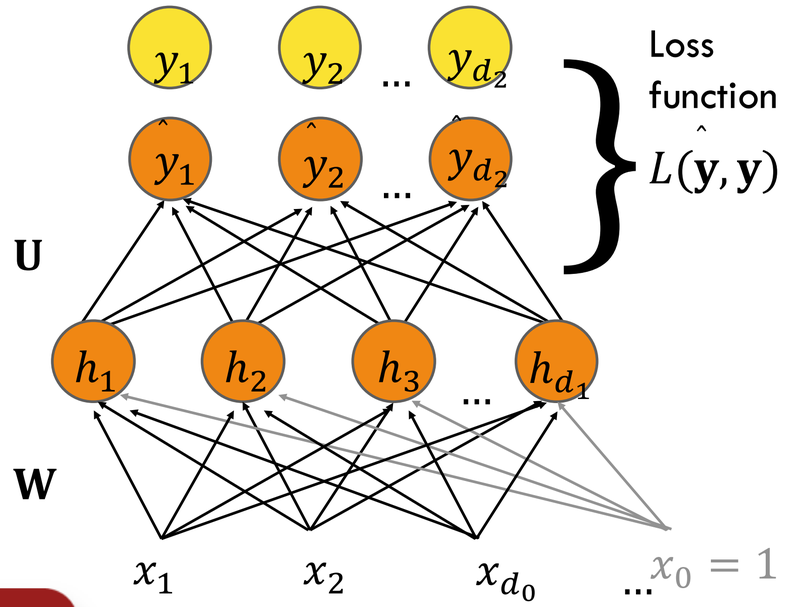
computation graphs

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

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
    - Which relies on computation graphs



Backprop

Graph representing the process of computing a mathematical expression

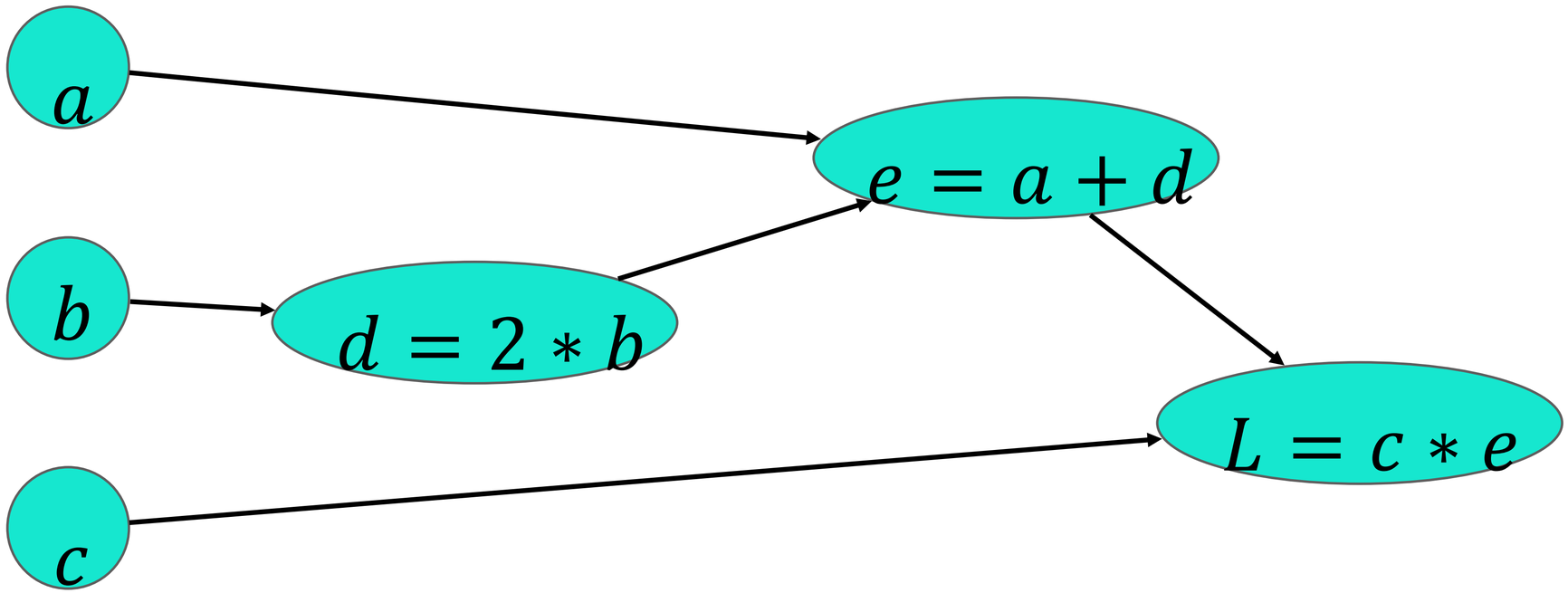
Rumelhart, Hinton, Williams, 1986

Example: Computation Graph

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

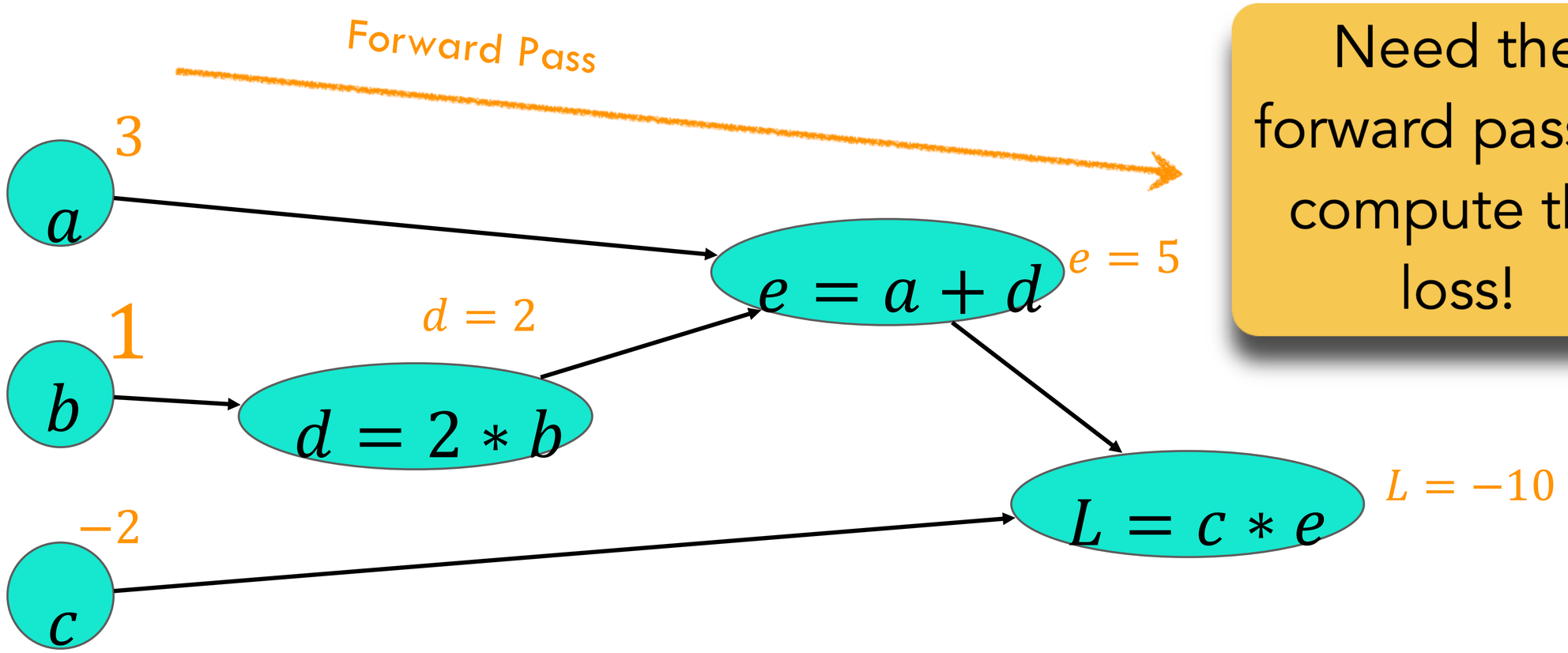


Example: Forward Pass

$d = 2 * b$

$e = a + d$

$L = c * e$



Need the forward pass to compute the loss!

But how to compute parameter updates?

Example: Backward Pass Intuition

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

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

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

$$\frac{\partial L}{\partial b} = ?$$

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

} Input Layer Gradients

Hidden Layer Gradients }

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

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

Chain Rule of Differentiation!

The Chain Rule

Computing the derivative of a composite function:

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

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

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

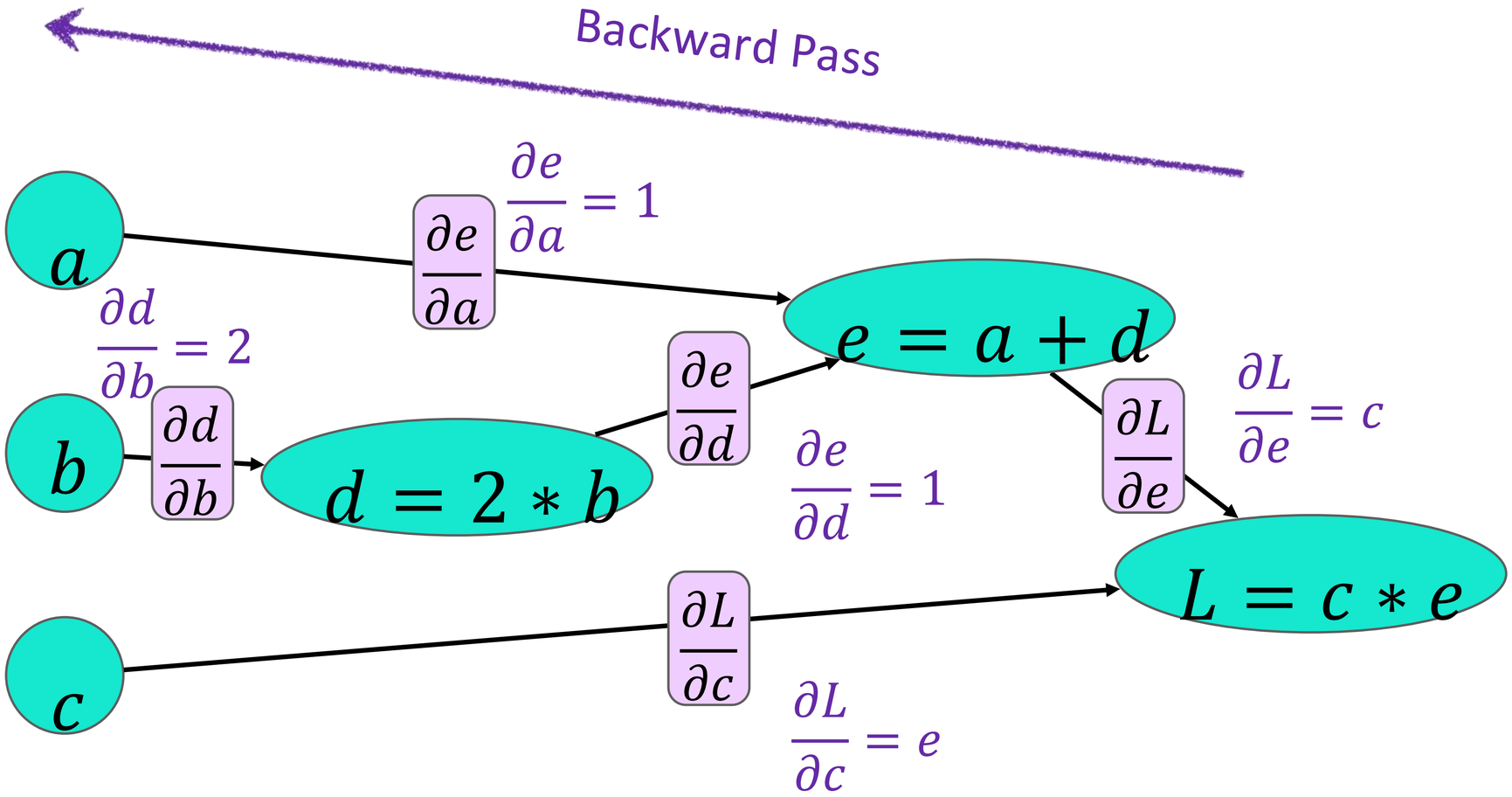
But we need the gradients of the loss with respect to parameters…

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

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

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

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



Example

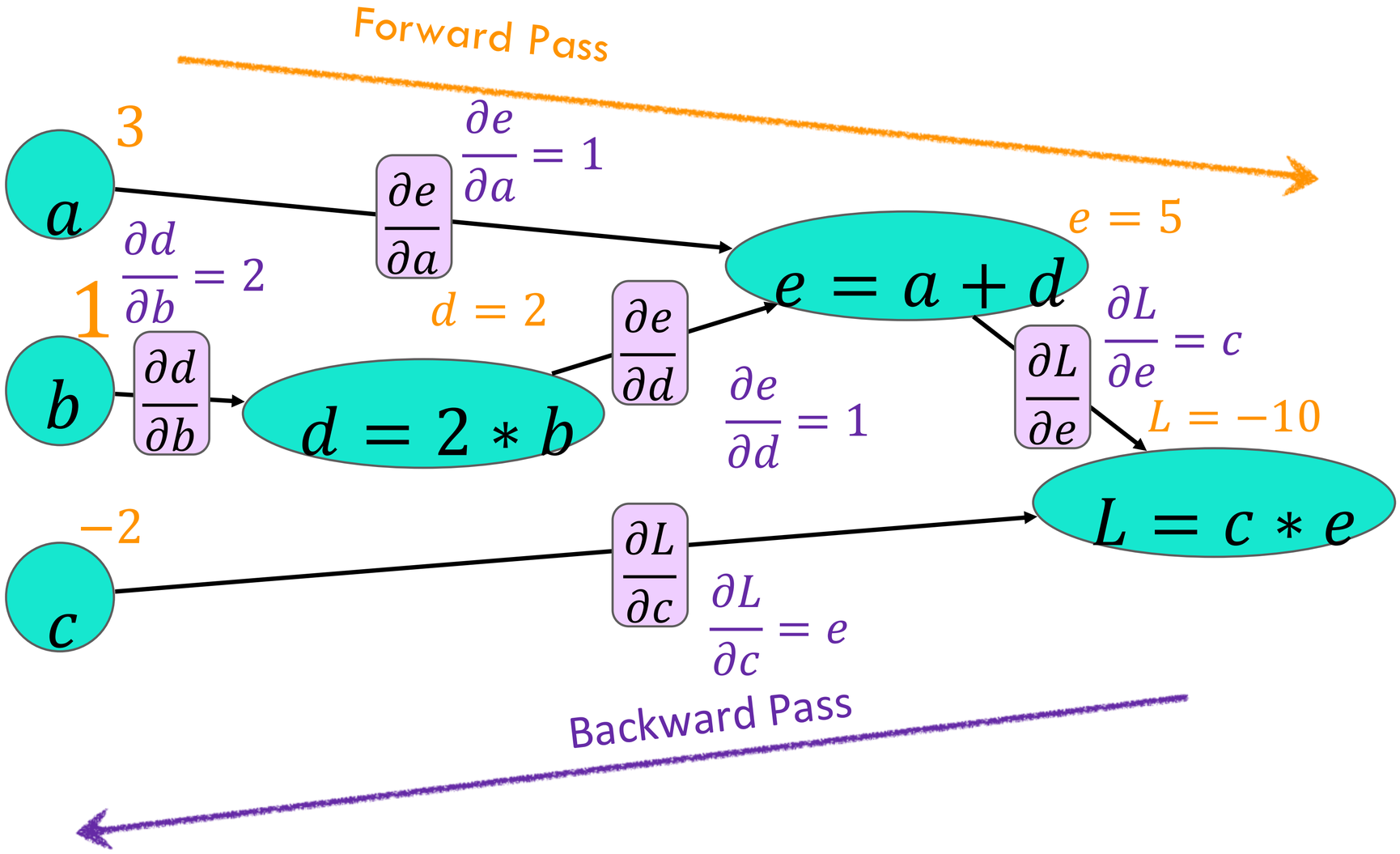
$$\frac{\partial L}{\partial e} = c = -2$$

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

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

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

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



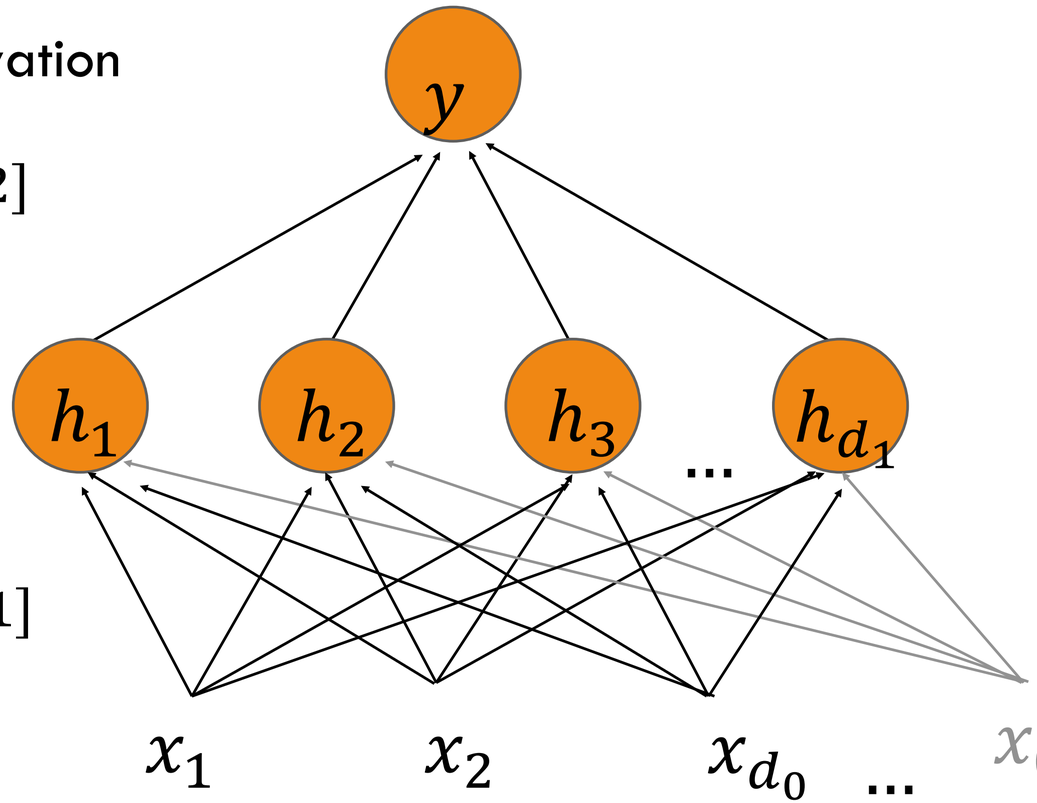
Backward Differentiation on a 2-layer MLP

Softmax Activation

$\mathbf{w}^{[2]}$

ReLU Activation

$\mathbf{W}^{[1]}$



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

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

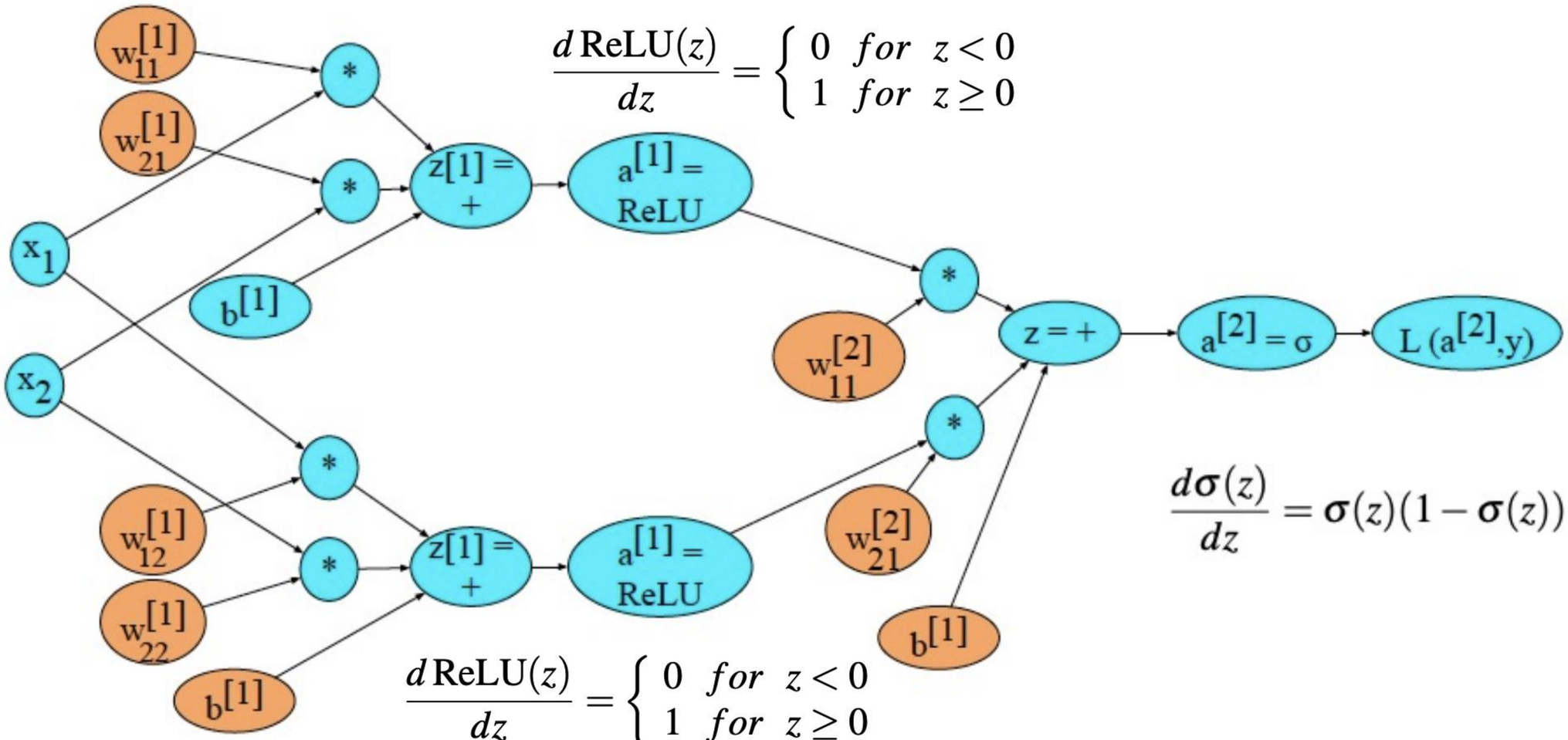
$$\mathbf{h}^{[1]} = ReLU(\mathbf{z}^{[1]})$$ Element-wise

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

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

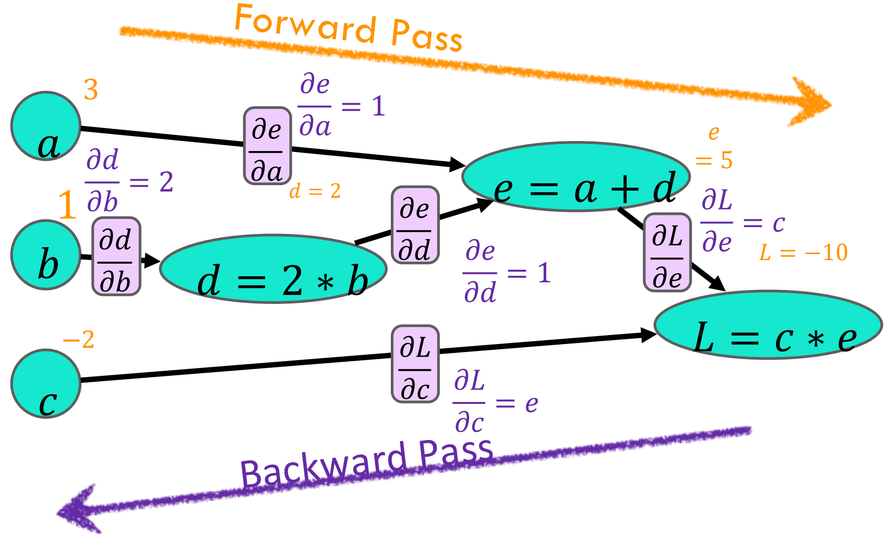
$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \quad z < 0 \\ 1 & for \quad z \geq 0 \end{cases}$$

2 layer MLP with 2 input features



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()`

Announcements + Logistics

- HW1 is due by Feb 4, 11:59 PM PT
- Project proposal is due by Feb 11, 11:59 PM PT