UNIVERSITÉ DU LUXEMBOURG

Natural Language Processing
Lecture 7: Introduction to Neural Networks

Salima Lamsiyah
University Luxembourg, FSTM, DCS, MINE Research Group
Salima.lamsiyah@uni.lu

Lecture Plan

1. Units in Neural Networks
2. The XOR Problem
3. Feedforward Neural Networks
4. Training Feedforward Neural Networks
5. Computation Graphs and Backward Differentiation (Optional but Important)
6. Feedforward Neural Networks in NLP

Logistic Regression Recall

Data: Inputs are continuous vectors of length K. Outputs are discrete.

$$\mathcal{D} = \{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{N} \text{ where } \mathbf{x} \in \mathbb{R}^K \text{ and } y \in \{0, 1\}$$

Model: Logistic function applied to dot product of parameters with input vector.

$$p_{\boldsymbol{\theta}}(y = 1|\mathbf{x}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^T \mathbf{x})}$$

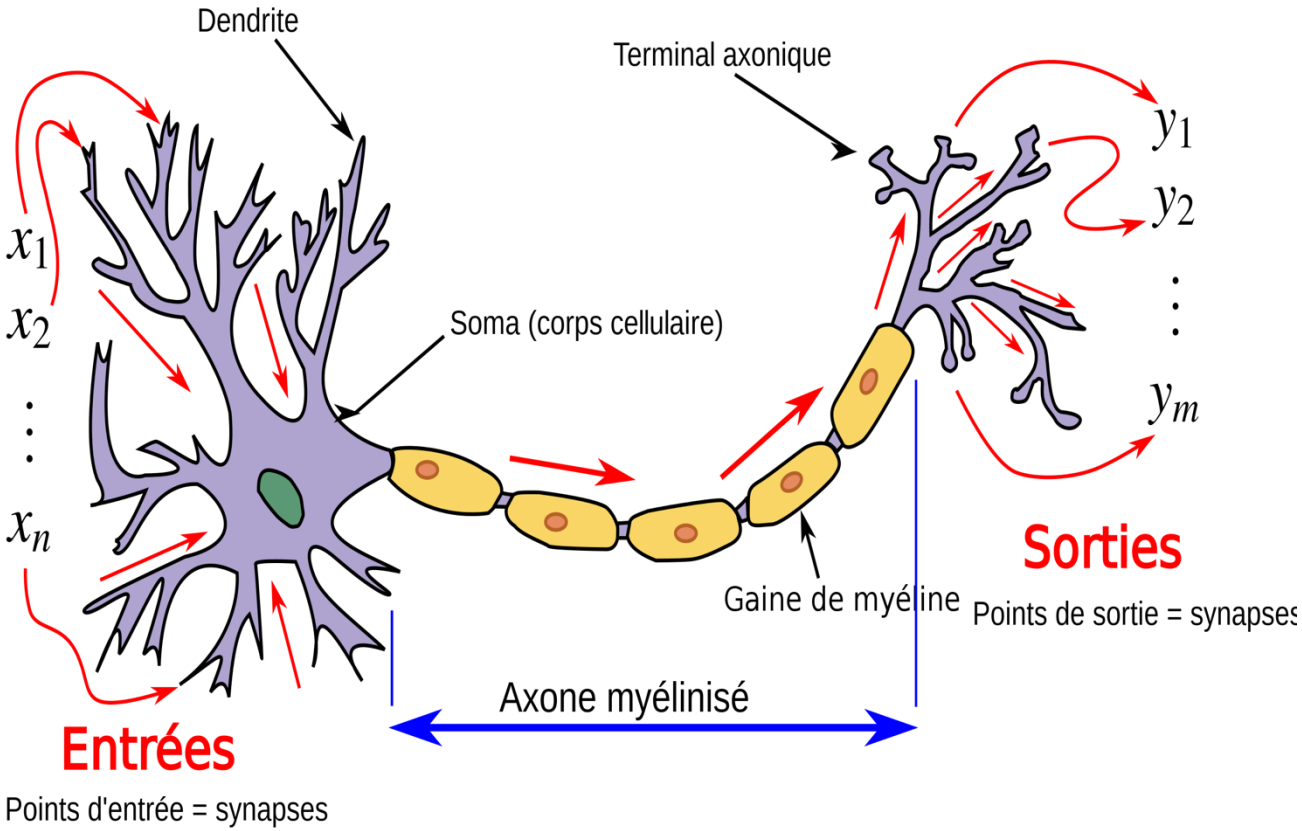
Learning: finds the parameters that minimize some objective function. $\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, J(\boldsymbol{\theta})$

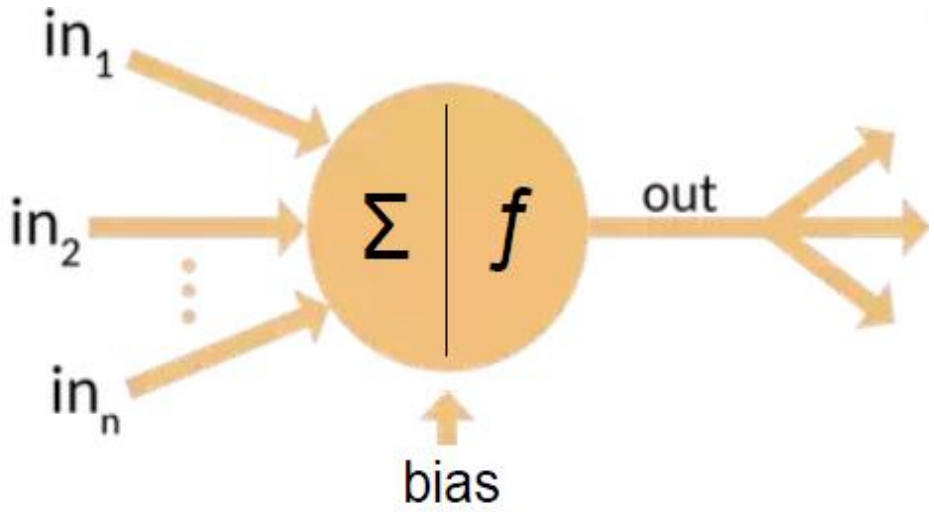
Prediction: Output is the most probable class.

$$\hat{y} = \underset{y \in \{0,1\}}{\operatorname{argmax}}\, p_{\boldsymbol{\theta}}(y|\mathbf{x})$$

Units in Neural Networks

Biological Neuron Vs Artificial Neuron





Perceptron (artificial neuron)

- The perceptron was invented in 1943 by Warren McCulloch and Walter Pitts.[3] The first implementation was a machine built in 1958 at the Cornell Aeronautical Laboratory by Frank Rosenblatt,[

Neural Network Unit: This is not in your brain
Perceptron

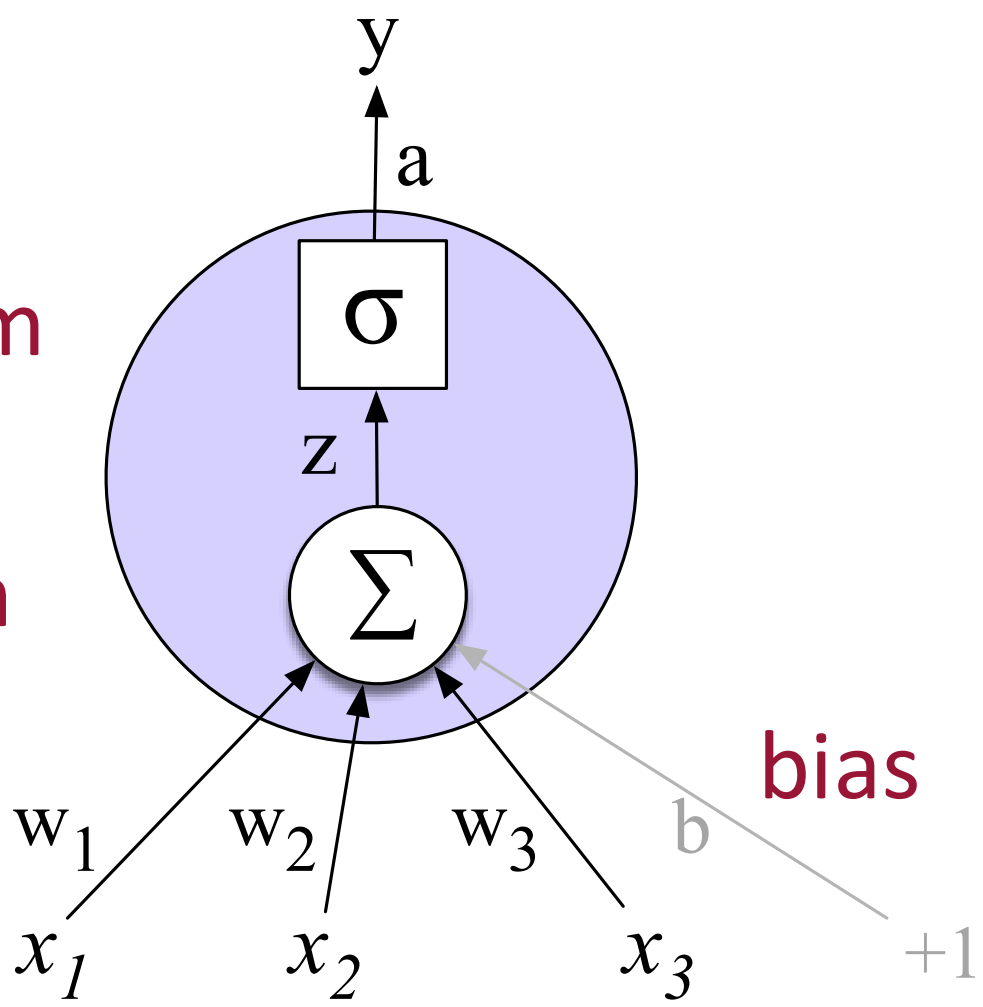
Output value

Non-linear transform

Weighted sum

Weights

Input layer



Neural unit

- Take weighted sum of inputs, plus a bias

$$z = b + \sum_i w_i x_i$$

$$z = w \cdot x + b$$

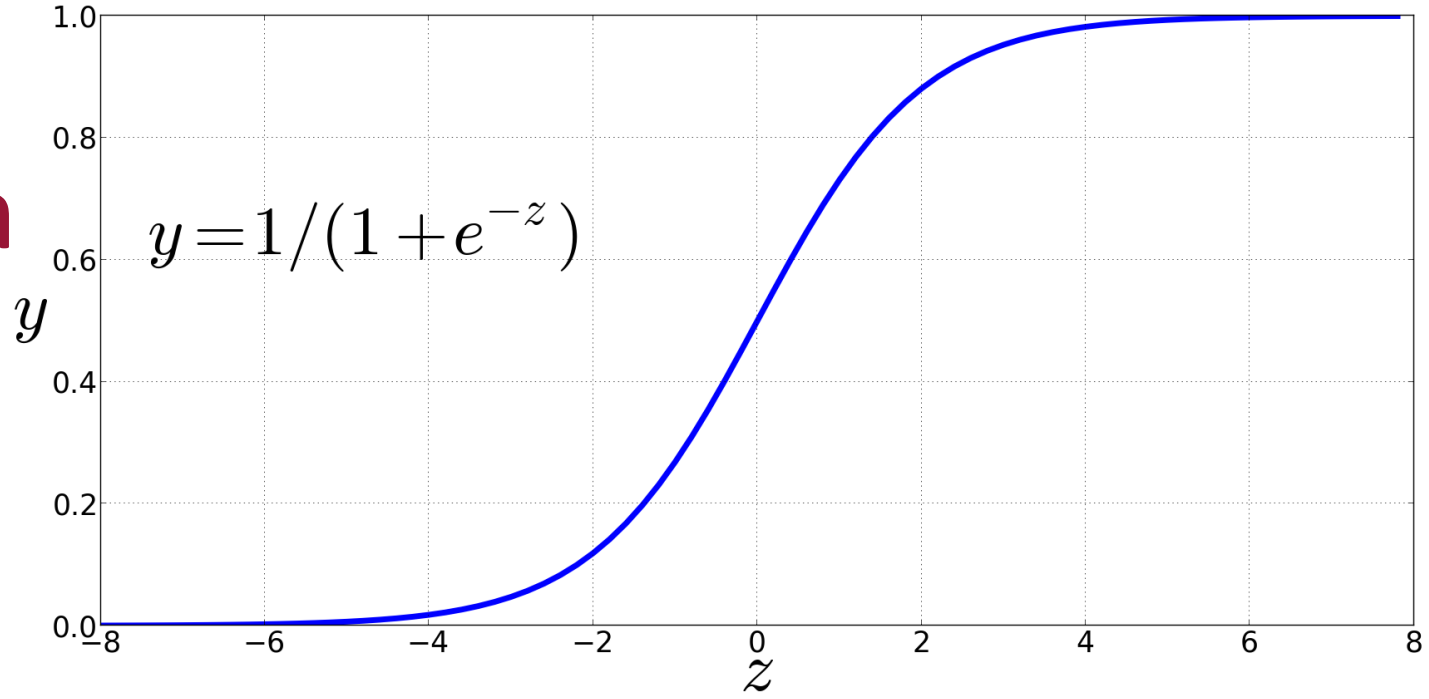
- Instead of just using z, we'll apply a nonlinear activation function g:

$$\hat{y} = a = g(z)$$

Non-Linear Activation Functions

Sigmoid Function

$$y = s(z) = \frac{1}{1 + e^{-z}}$$



Final function the unit is computing

$$y = s(w \cdot x + b) = \frac{1}{1 + \exp(-(w \cdot x + b))}$$

An example

- What happens with input x:

$$x = [0.5, 0.6, 0.1]$$

- *Suppose a unit has:*

$$w = [0.2, 0.3, 0.9]$$

$$b = 0.5$$

$$y = s(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}} =$$

$$\frac{1}{1 + e^{-(.5 \times .2 + .6 \times .3 + .1 \times .9 + .5)}} =$$

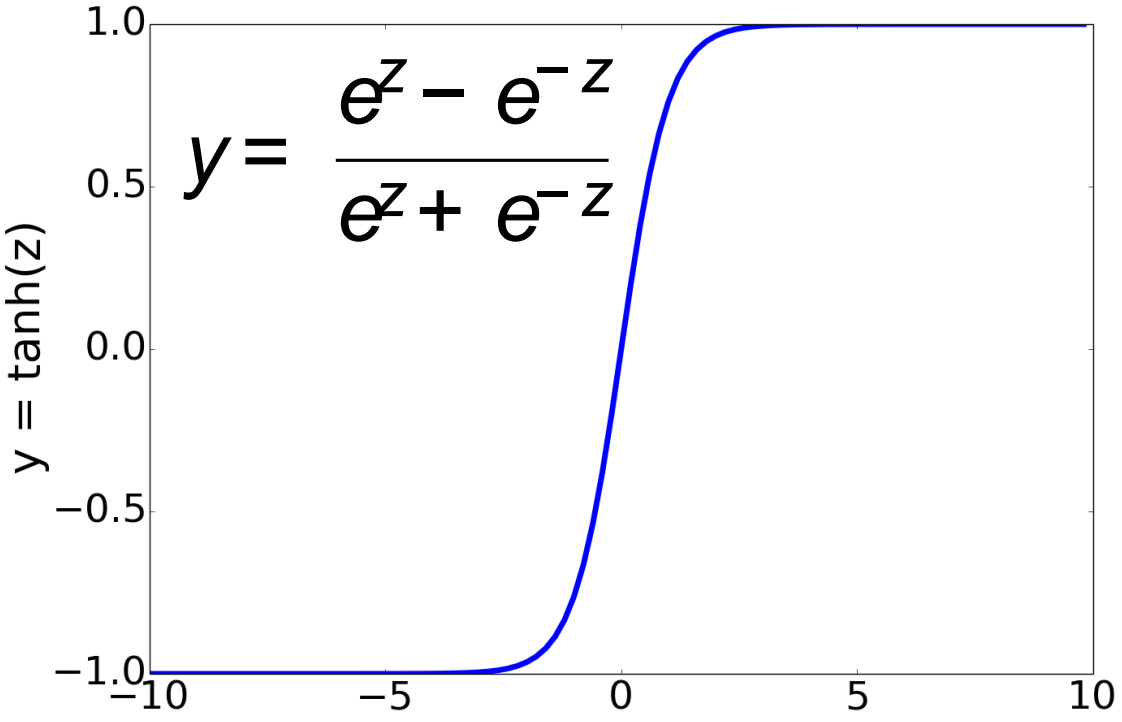
An example

$$y = s(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}} =$$

$$\frac{1}{1 + e^{-(.5 \times .2 + .6 \times .3 + .1 \times .9 + .5)}} =$$

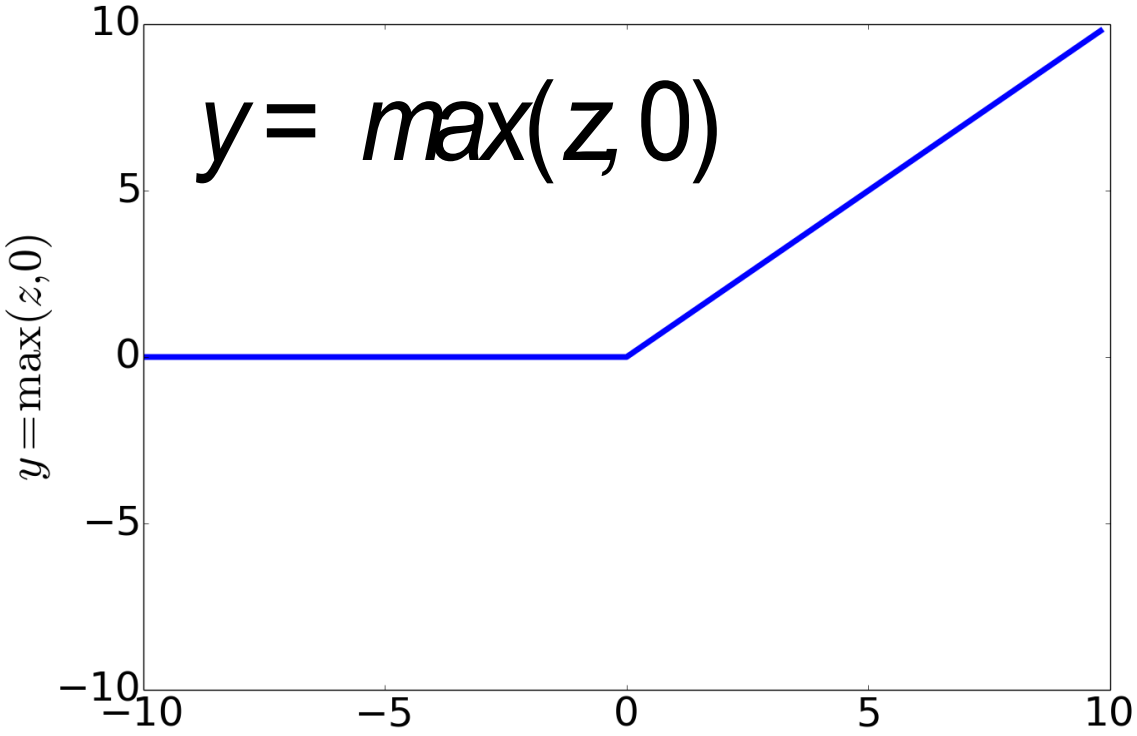
$$\frac{1}{1 + e^{-0.87}} = .70$$

Non-Linear Activation Functions besides sigmoid



tanh

Most Common:



ReLU

Rectified Linear Unit

Activation Functions: Smoothness and Saturation

- **Smoothly Differentiable**
  - Means the function's slope changes gradually (no sharp corners).
  - Sigmoid and tanh → smooth ReLU → not smooth at z = 0

- **Sigmoid and tanh Functions**
  - Sigmoid maps z → (0, 1); tanh maps z → (–1, 1).
  - For large |z|: output ≈ 1 or –1 → derivative ≈ 0 (saturation).
  - When the derivative is ≈ 0, the neuron stops learning.

- **Example (Sigmoid Saturation):**

$$\sigma(10) = \frac{1}{1 + e^{-10}} \approx 0.99995, \quad \sigma'(10) = \sigma(10)\,[1 - \sigma(10)] \approx 0.00005$$

=> Derivative almost zero ⇒ weights barely update. **(Vanishing Gradient Problem)**

Vanishing Gradient Problem and ReLU Solution

- Thw vanishing gradient problem:
  - The learning signal (gradient) becomes too weak to improve earlier layers.

- **ReLU: A Practical Solution**

$$ReLU(z) = \max(0, z)$$

- Passes positive inputs directly, blocks negatives.
- Derivative = 1 for z>0 → no vanishing effect.
- Simple, fast, and works well in deep networks.

- **Example:**

$$z = -3 \Rightarrow 0,\ z = 4 \Rightarrow 4$$

→ For positive inputs, neuron keeps learning effectively.

When ReLU dies!

- **Solution 1: Leaky ReLU**

$$\text{LeakyReLU}(z) = \begin{cases} z, & z > 0 \\ \alpha z, & z \leq 0 \end{cases}$$

- Instead of being **0** for negative values, it allows a **small slope** (e.g., $\alpha = 0.01$).
- This means gradients can still pass through even when $z < 0$.

=> Fixes "dead neurons" problem.

- **Example**:
  If $z = -4$and $\alpha = 0.01$:
  → output = -0.04, derivative = 0.01 (small but not 0).

The XOR Problem

The XOR problem

Minsky and Papert (1969)

- Can neural units compute simple functions of input?

AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

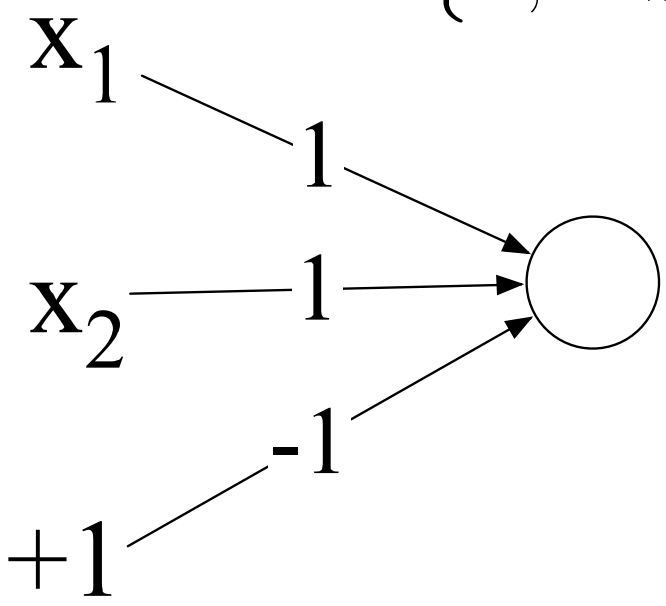
Perceptron

- A very simple neural unit
- Binary output (0 or 1)
- Linear activation function

$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$

Easy to build AND or OR with perceptron

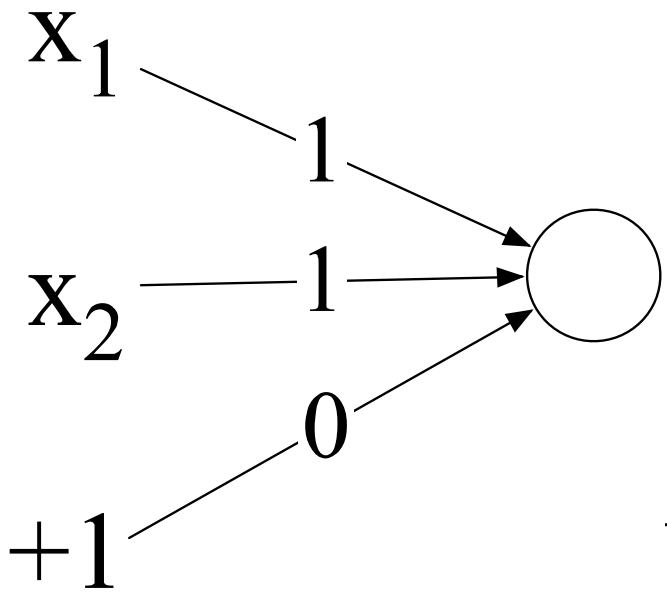
$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$



AND

AND

x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1



OR

OR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

Why? Perceptrons are linear classifiers

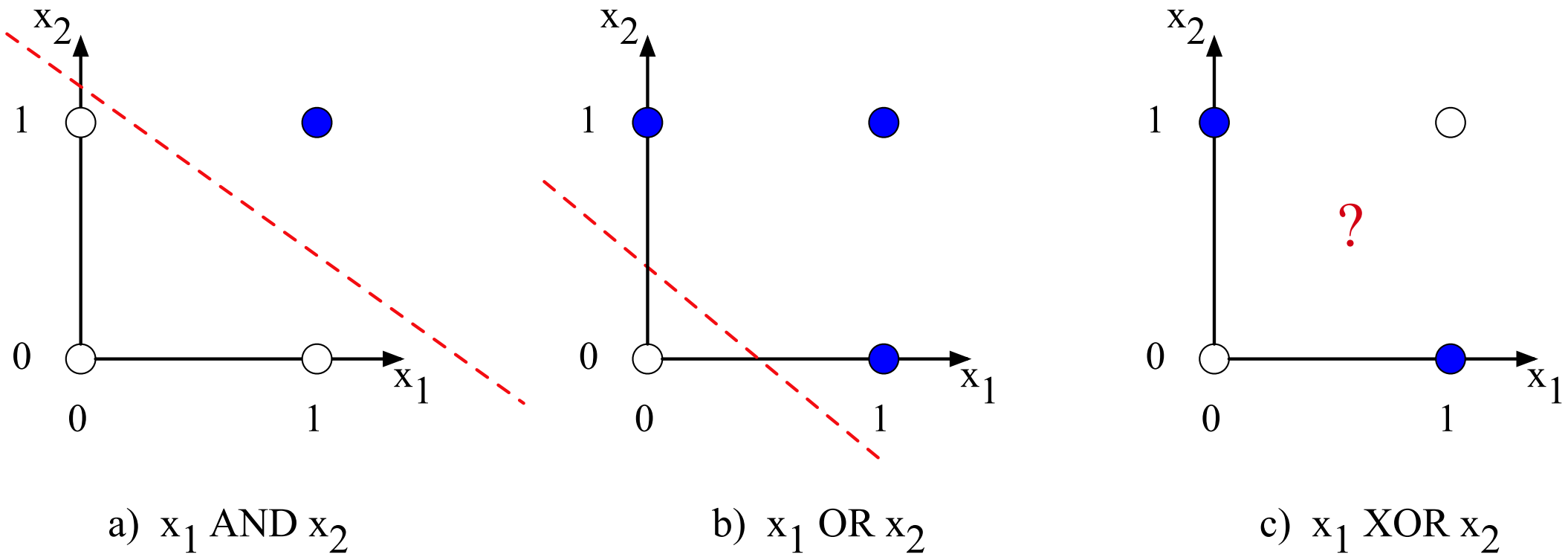
- Perceptron equation given x_1 and x_2, is the equation of a line

$$w_1 x_1 + w_2 x_2 + b = 0$$

- (in standard linear format: $x_2 = (-w_1/w_2)x_1 + (-b/w_2)$)

- This line acts as a **decision boundary**
- 0 if input is on one side of the line
- 1 if on the other side of the line

Decision boundaries



a) x_1 AND x_2

b) x_1 OR x_2

c) x_1 XOR x_2

XOR is not a **linearly separable** function!

Is it possible to capture XOR with perceptron?

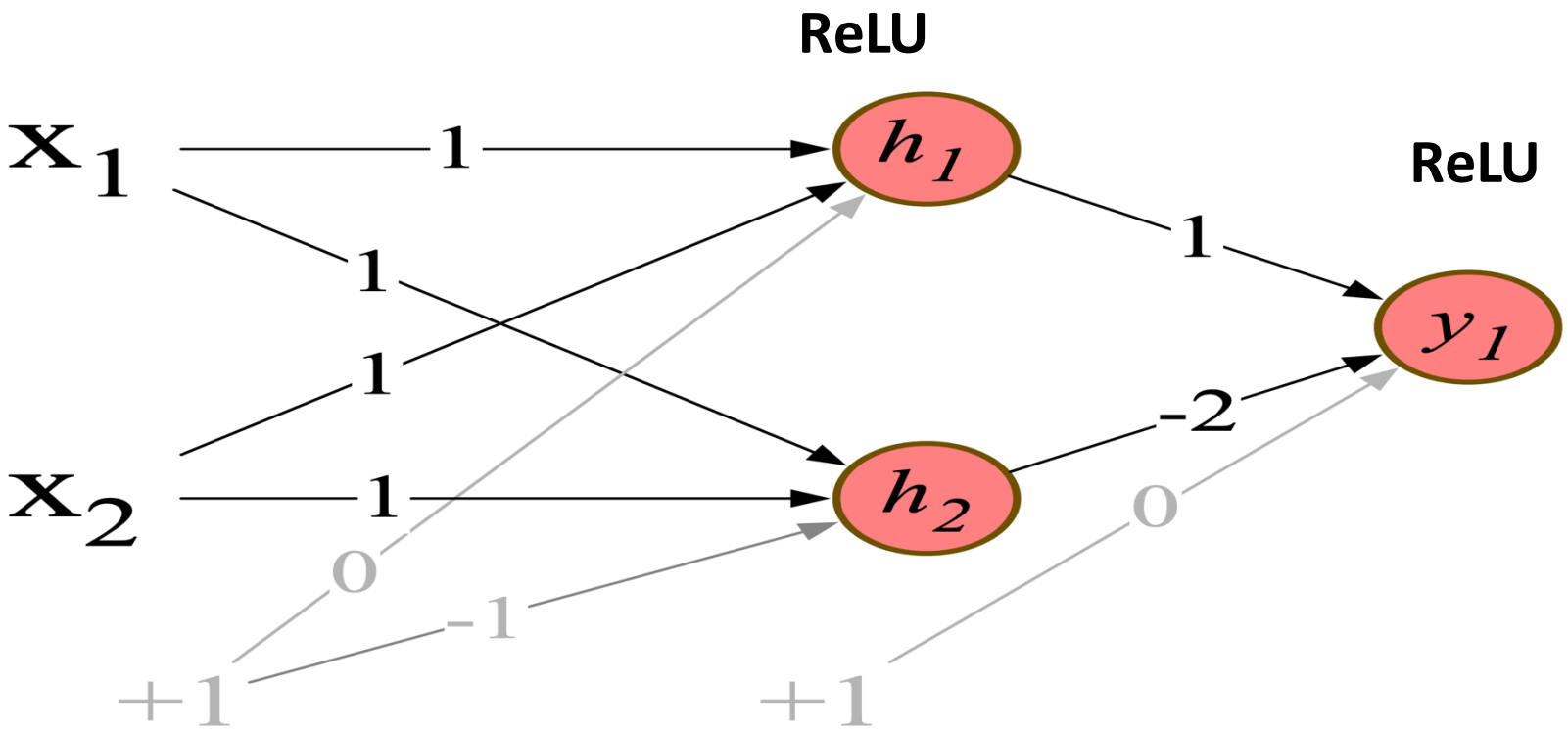
- Try for yourself!

Solution to the XOR problem

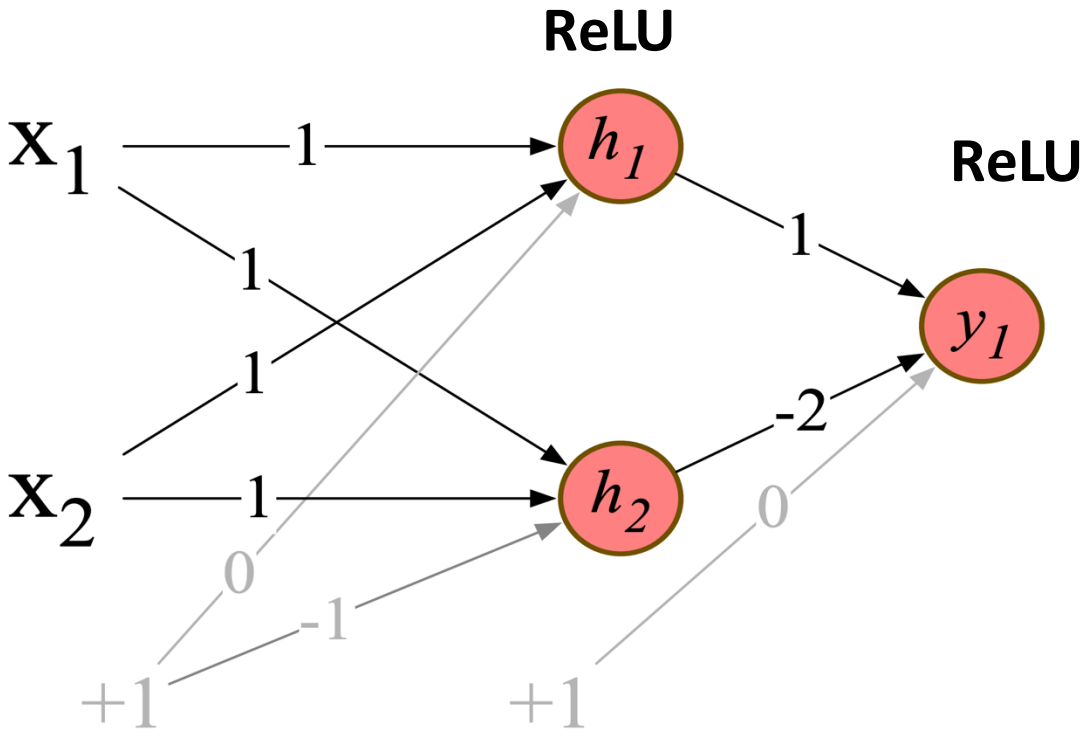
- XOR **can't** be calculated by a single perceptron
- XOR **can** be calculated by a layered network of units.

XOR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0



Solution to the XOR problem

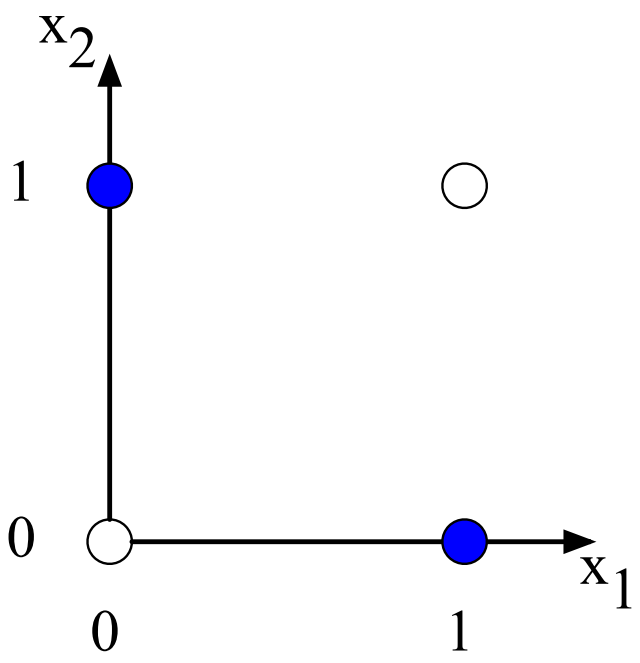


$$y = \begin{cases} 0, & \text{if } w \cdot x + b \leq 0 \\ 1, & \text{if } w \cdot x + b > 0 \end{cases}$$

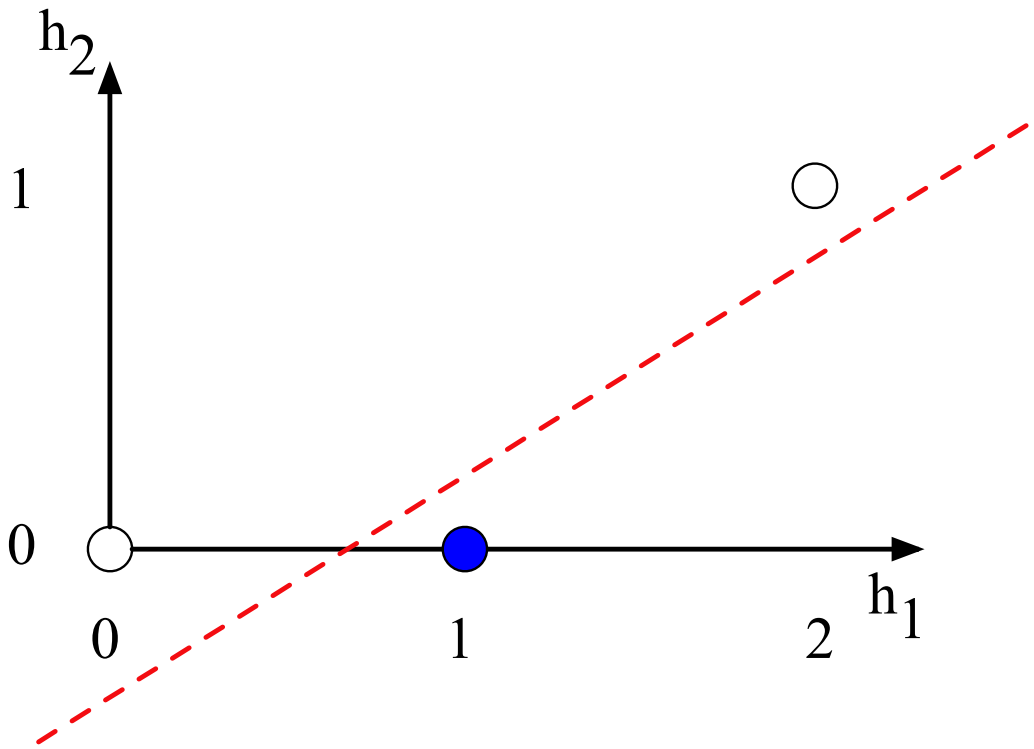
x_1	x_2	h_1	h_2	y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	2	1	0

The hidden representation h

x_1	x_2	h_1	h_2	y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	2	1	0



a) The original *x* space



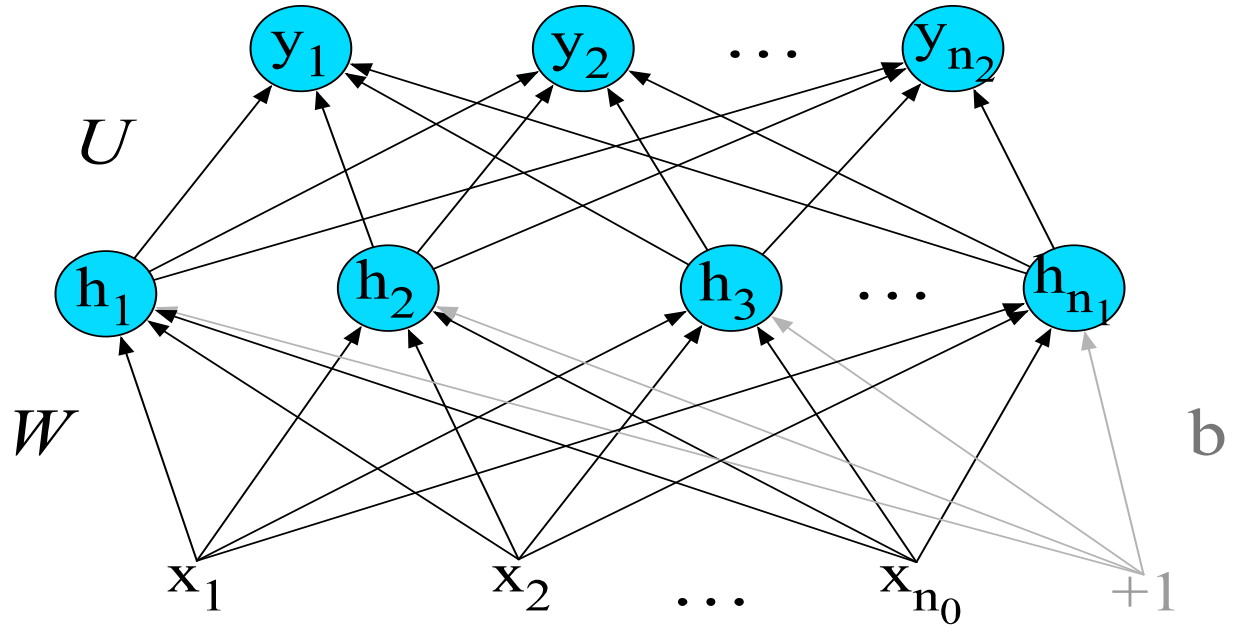
b) The new (linearly separable) *h* space

(With learning: hidden layers will learn to form useful representations)

Feedforward Neural Networks

Feedforward Neural Networks

- Can also be called **multi-layer perceptrons** (or **MLPs**) for historical reasons



A feedforward network is a multilayer network in which the units are connected with no cycles;

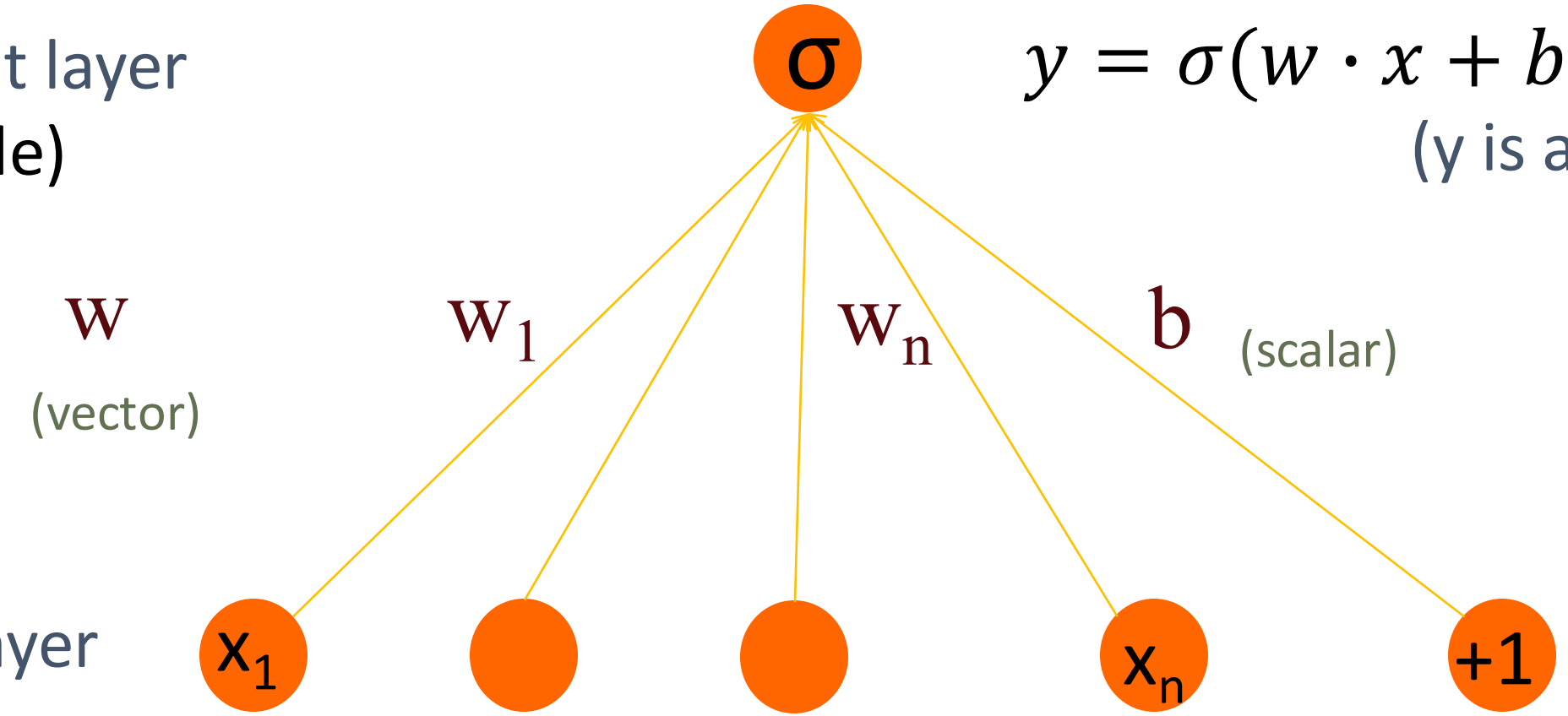
Binary Logistic Regression as a 1-layer Network

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

Output layer
(σ node)

$$y = \sigma(w \cdot x + b)$$

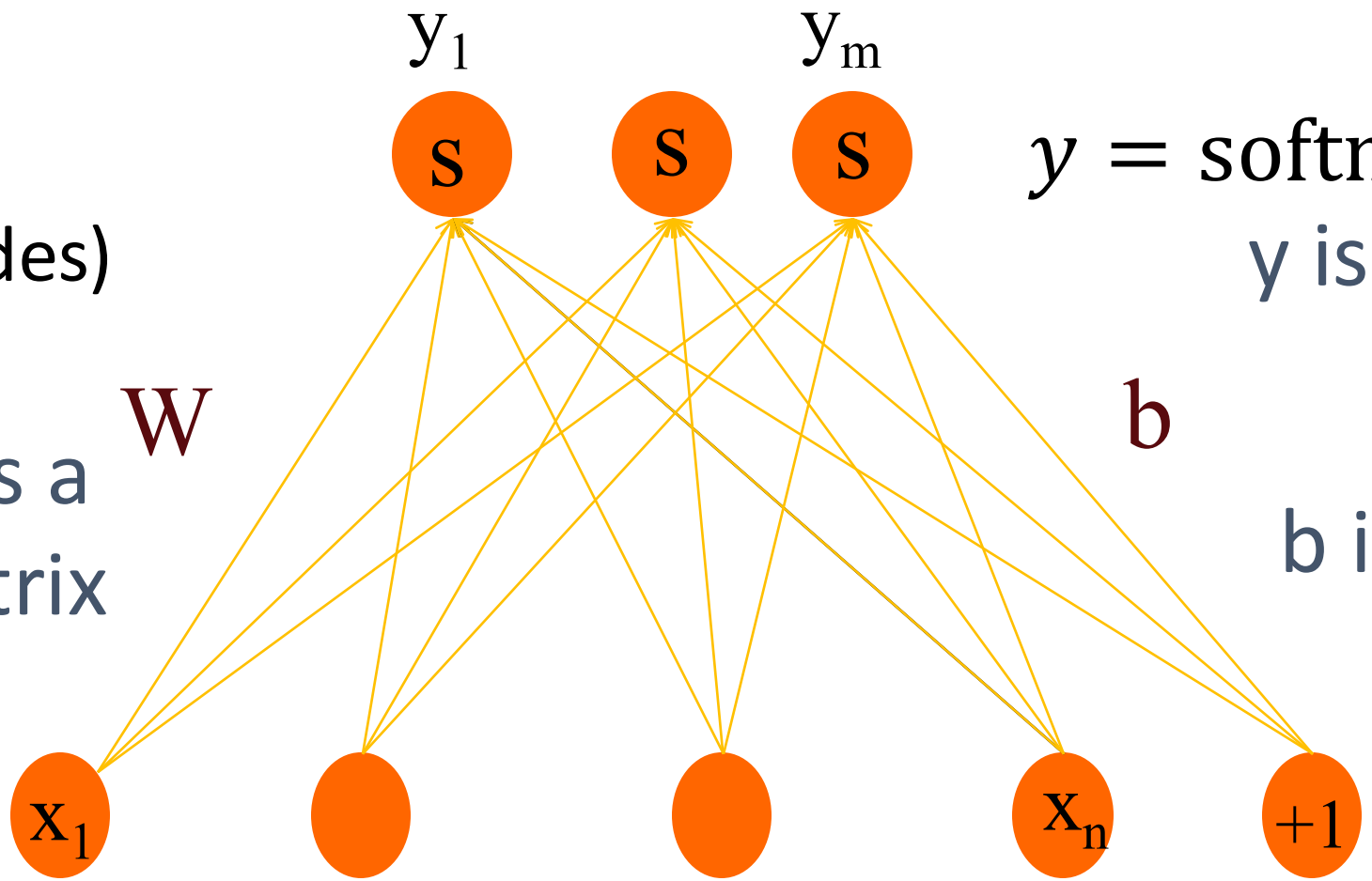
(y is a scalar)



Input layer
vector x

Multinomial Logistic Regression as a 1-layer Network

Fully connected single layer network



Output layer
(softmax nodes)

$$y = \text{softmax}(Wx + b)$$

y is a vector

W is a matrix

b is a vector

Input layer
scalars

Reminder: softmax: a generalization of sigmoid

- For a vector *z* of dimensionality *k*, the softmax is:

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

- Example:

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{k} \exp(z_j)} \quad 1 \leq i \leq k$$

$$z = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

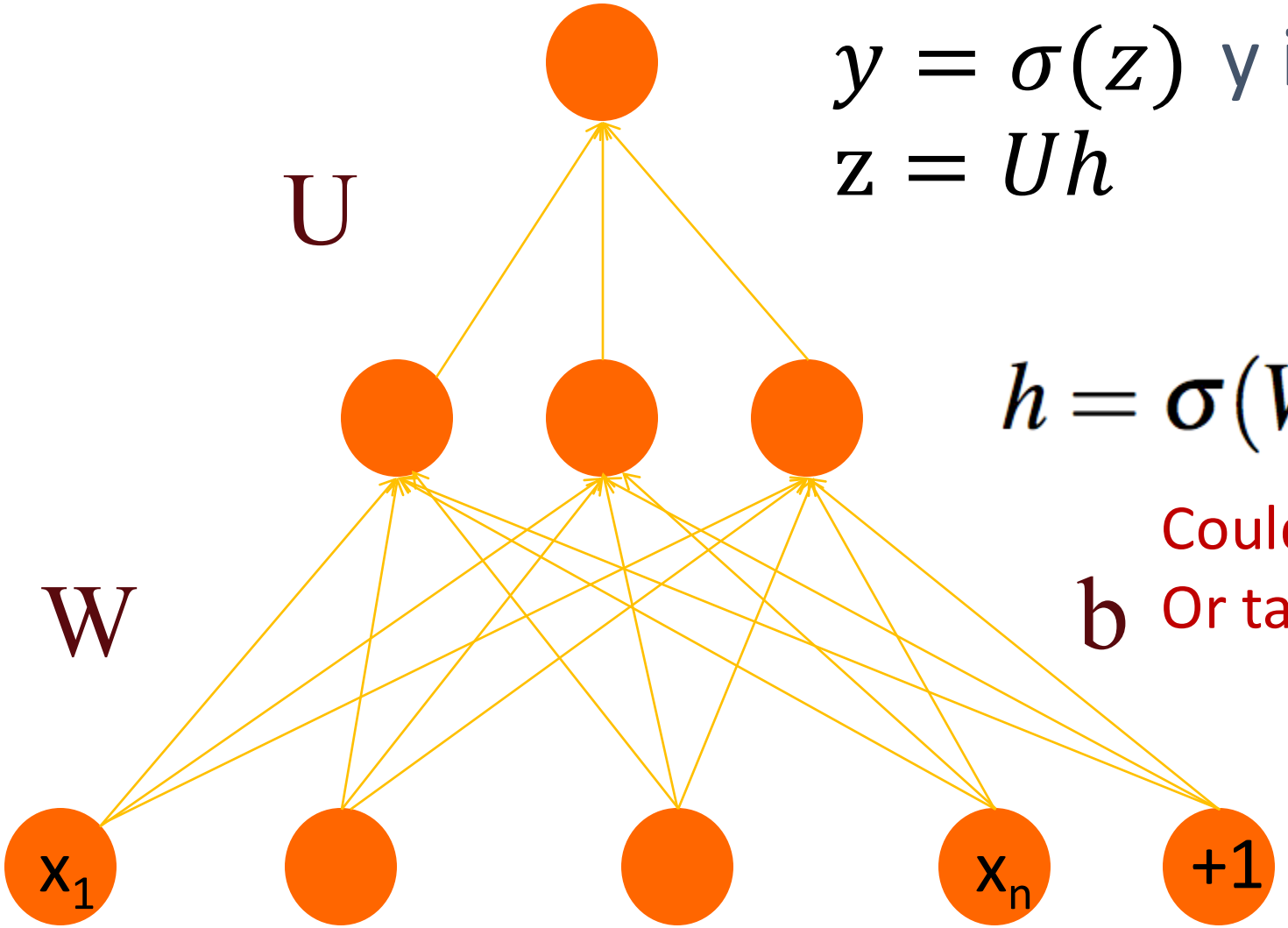
$$\text{softmax}(z) = [0.055, 0.090, 0.006, 0.099, 0.74, 0.010]$$

Two-Layer Network with one output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)



$y = \sigma(z)$ y is a scalar

$z = Uh$

$h = \sigma(Wx + b)$

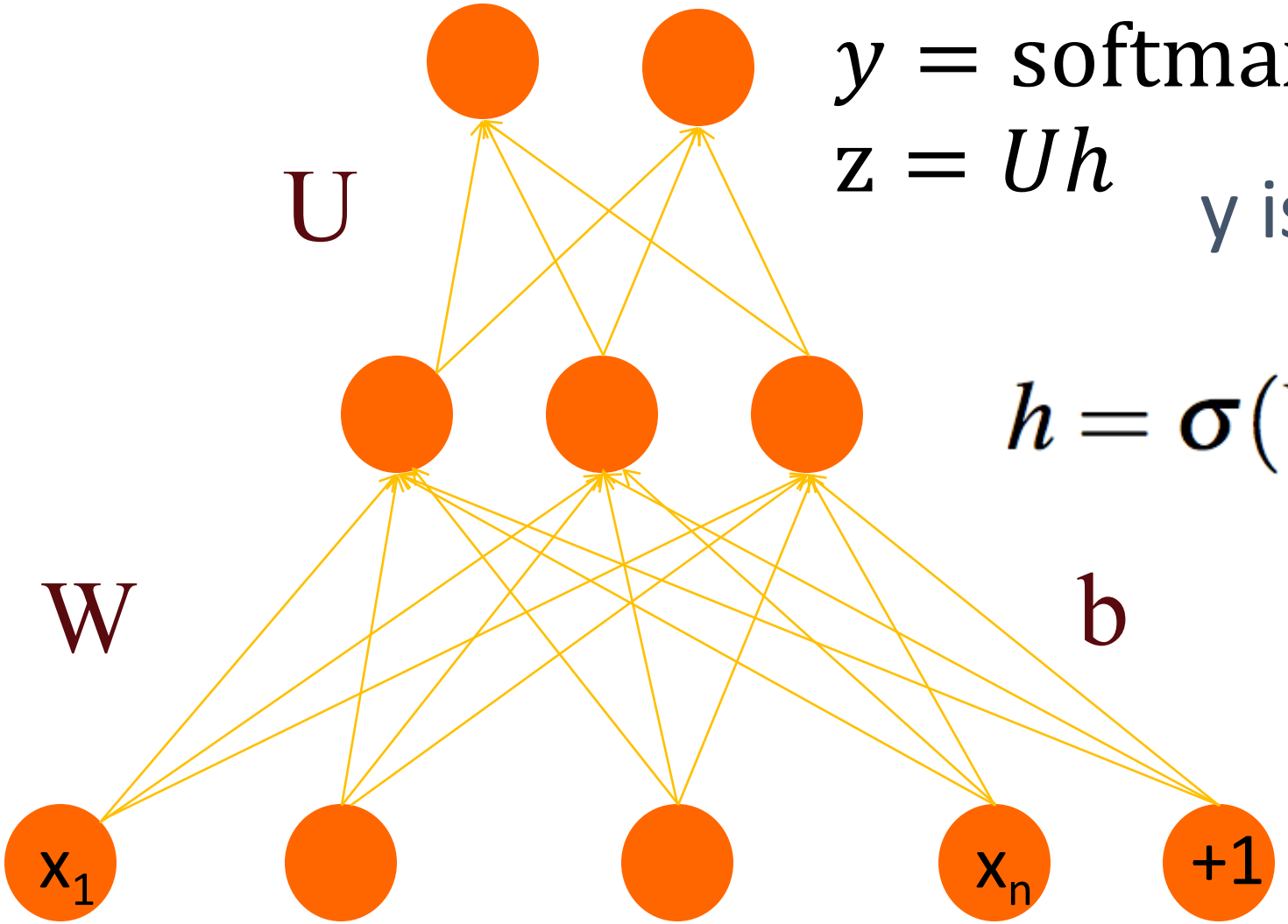
Could be ReLU
Or tanh

Two-Layer Network with softmax output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)



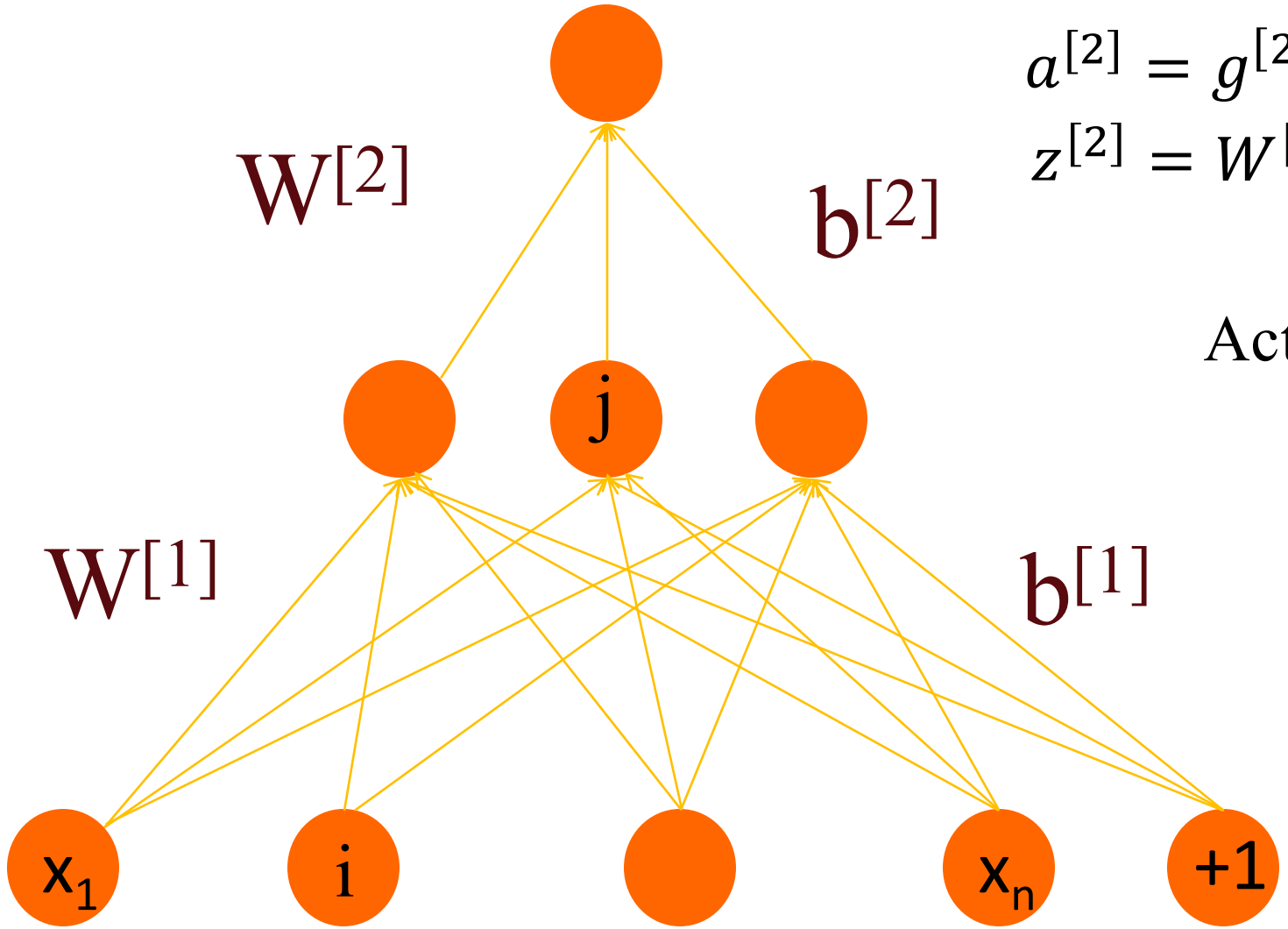
$$y = \text{softmax}(z)$$
$$z = Uh$$

y is a vector

$$h = \sigma(Wx + b)$$

Could be ReLU
Or tanh

Multi-layer Notation



$y = a^{[2]}$ sigmoid or softmax

$a^{[2]} = g^{[2]}(z^{[2]})$

$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$

Activation function e.g. ReLU

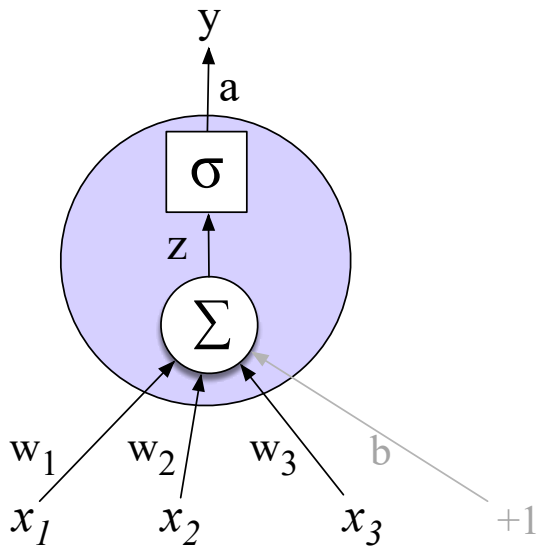
$a^{[1]} = g^{[1]}(z^{[1]})$

$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$

$a^{[0]}$

Multi Layer Notation

$$
\begin{aligned}
z^{[1]} &= W^{[1]} a^{[0]} + b^{[1]} \\
a^{[1]} &= g^{[1]}(z^{[1]}) \\
z^{[2]} &= W^{[2]} a^{[1]} + b^{[2]} \\
a^{[2]} &= g^{[2]}(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$



for i **in** 1..n

$$
\begin{aligned}
z^{[i]} &= W^{[i]} a^{[i-1]} + b^{[i]} \\
a^{[i]} &= g^{[i]}(z^{[i]}) \\
\hat{y} &= a^{[n]}
\end{aligned}
$$

Replacing the bias unit

- Instead of:

$$x = x_1, x_2, \ldots, x_{n0}$$

$$h = \sigma(Wx + b)$$

$$h_j = \sigma\left(\sum_{i=1}^{n_0} W_{ji} x_i + b_j\right)$$

We'll do this:

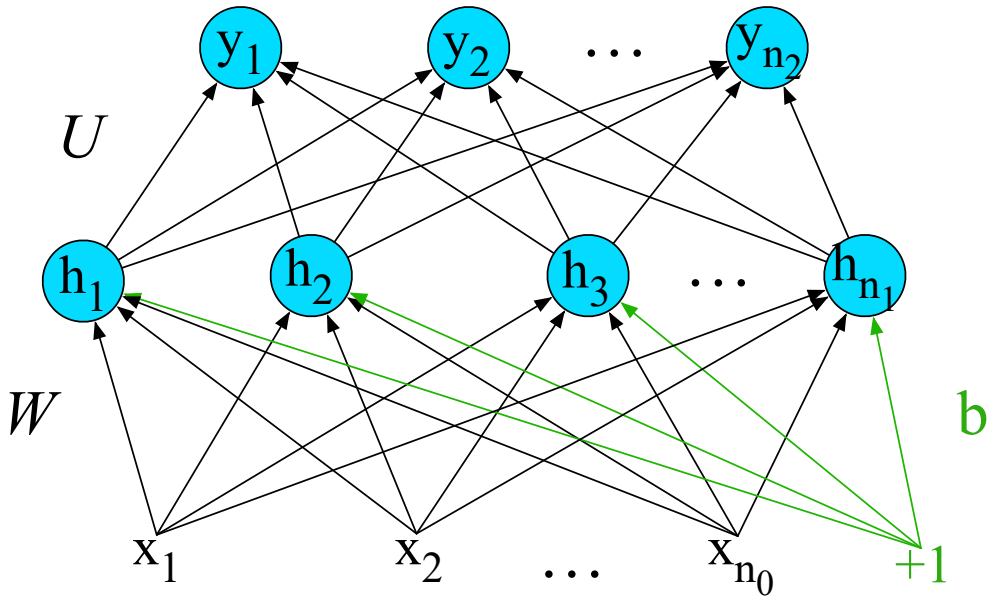
$$x = x_0, x_1, x_2, \ldots, x_{n0}$$

$$h = \sigma(Wx)$$

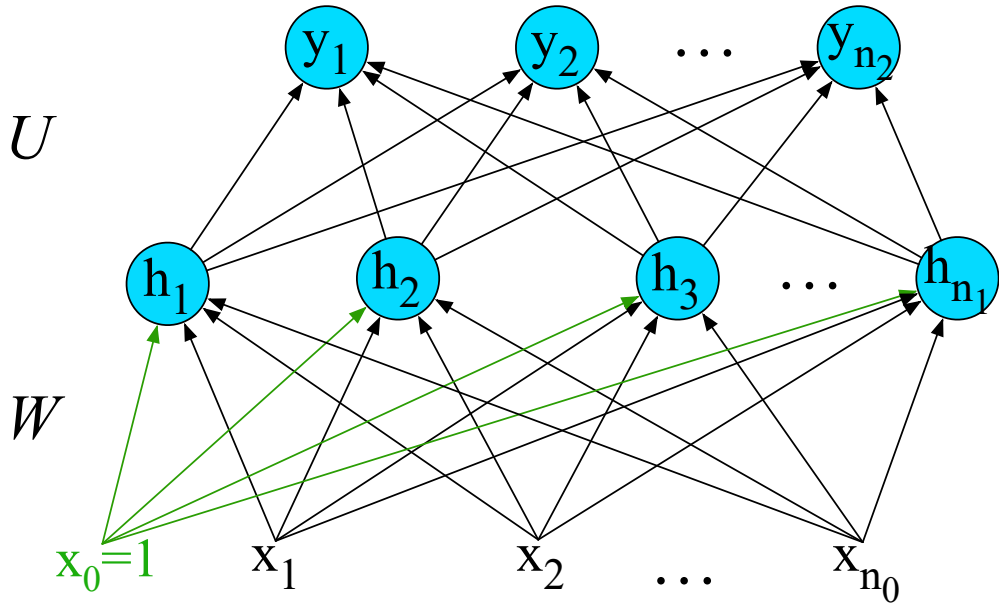
$$\sigma\left(\sum_{i=0}^{n_0} W_{ji} x_i\right)$$

Replacing the bias unit

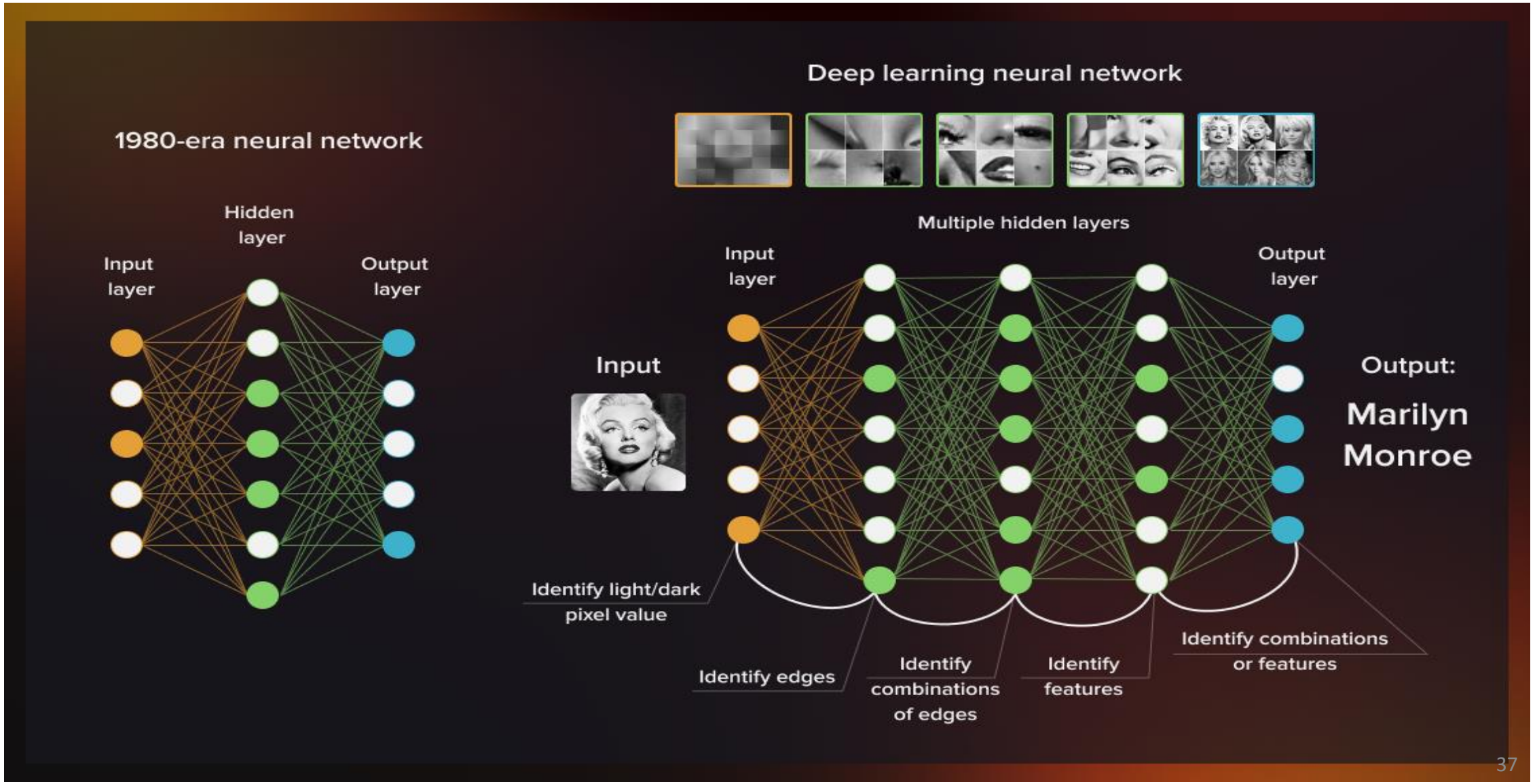
Instead of:



We'll do this:

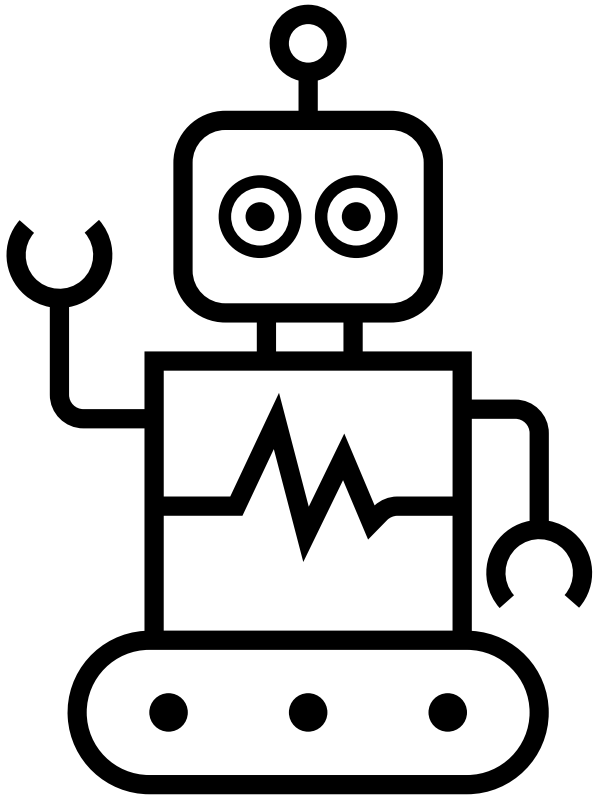


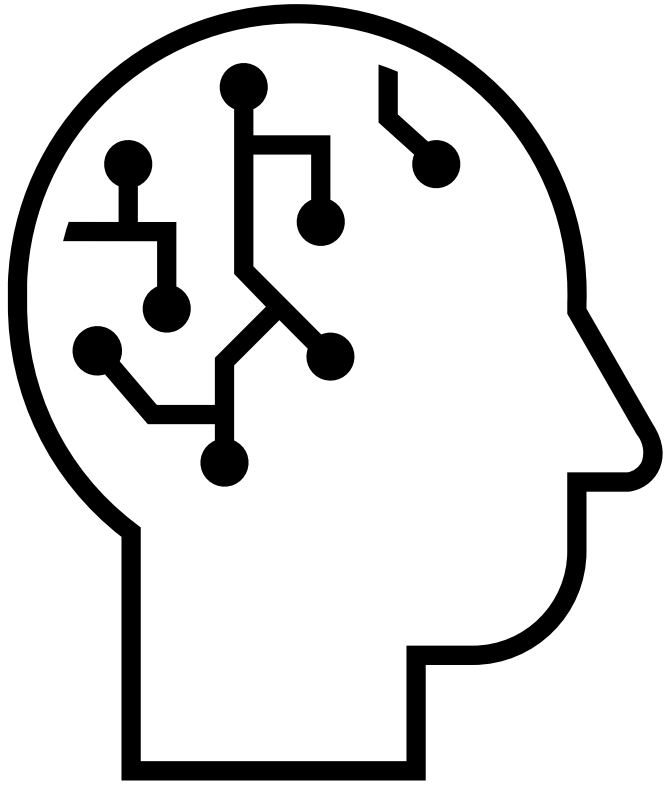
Non Deep vs Deep Neural Networks (Deep Learning)



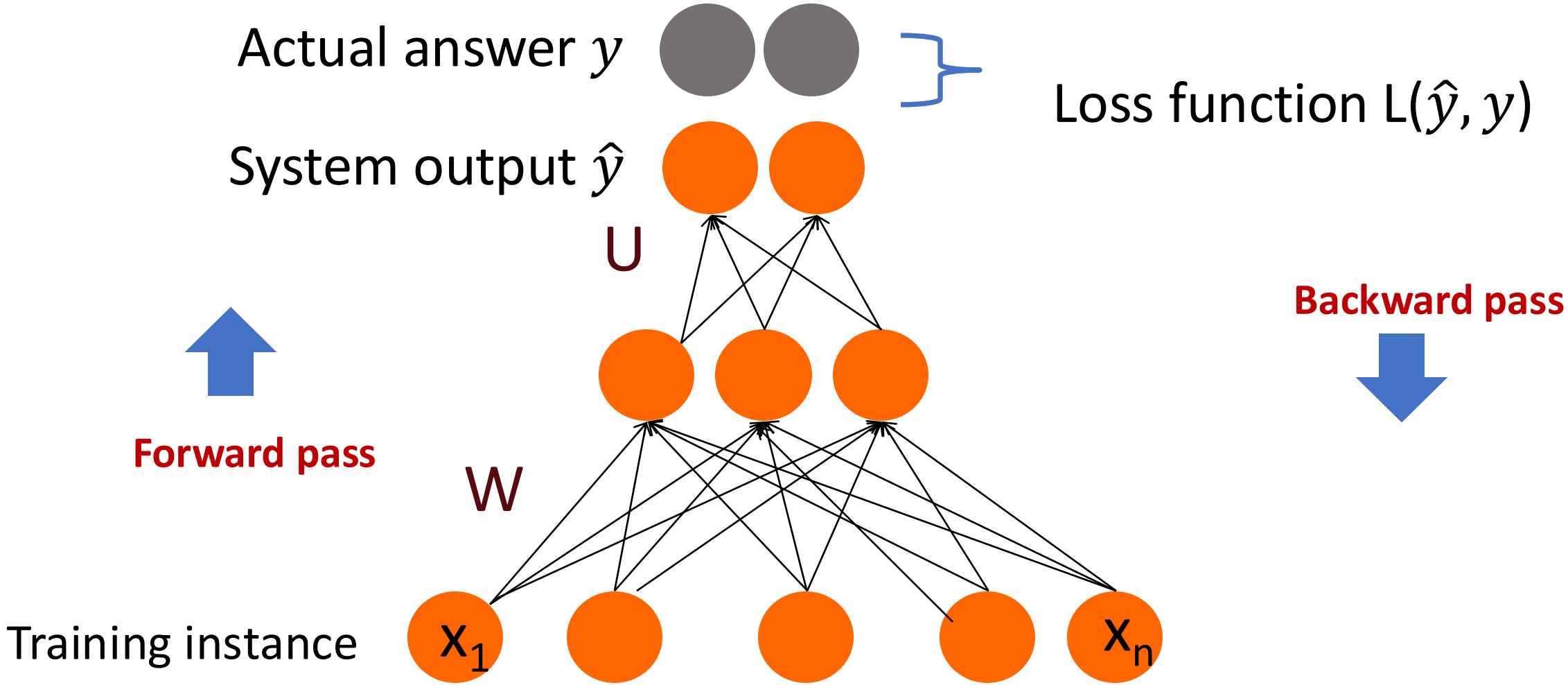
Training Neural Networks

How does the network “learn”?





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**
    - Determine how much it contributed to the prediction error
    - For every weight w from input layer to the hidden layer
      - Update the weight

Reminder: Loss Function for binary logistic regression

- A measure for how far off the current answer is to the right answer
- Cross entropy loss for logistic regression:

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

$$= -[y \log \sigma(w \cdot x + b) + (1-y)\log(1-\sigma(w \cdot x + b))]$$

Reminder: gradient descent for weight updates

- Use the derivative of the loss function with respect to weights
  $\frac{d}{dw} L(f(x; w), y)$
- To tell us how to adjust weights for each training item
  - Move them in the opposite direction of the gradient
  - For logistic regression

$$w^{t+1} = w^t - h \frac{d}{dw} L(f(x; w), y)$$

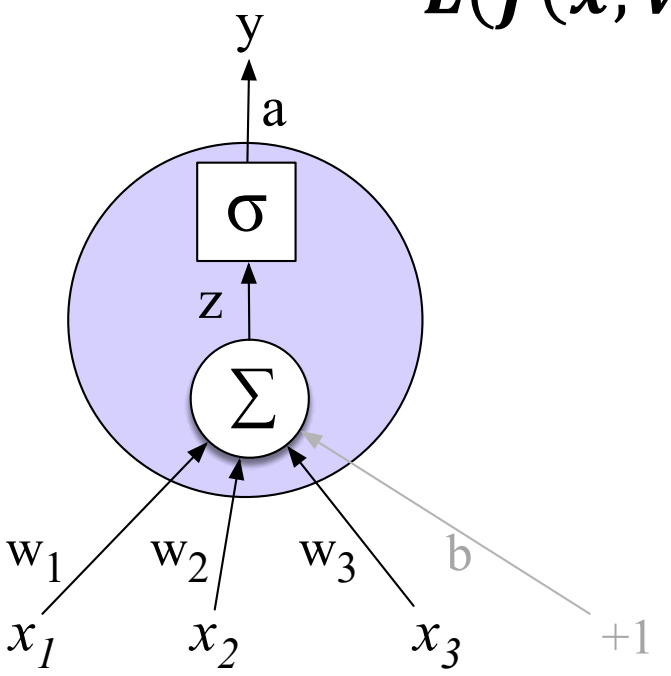
$$\frac{\partial L_{\text{CE}}(\hat{y}, y)}{\partial w_j} = [\sigma(w \cdot x + b) - y] x_j$$

Where did that derivative come from?

- Using the chain rule! $f(x) = u(v(x))$

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$$

$$\boldsymbol{L(f(x; w), y)}$$



Derivative of the weighted sum

Derivative of the Activation

Derivative of the Loss

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_i}$$

How can I find that gradient for every weight in the network?

- These derivatives on the prior slide only give the updates for one weight layer: the last one!

- What about deeper networks?
- Lots of layers, different activation functions?

- Solution in the next slide:
- Even more use of the chain rule!!
- Computation graphs and backward differentiation!

Computation Graphs and Backward Differentiation

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** (Rumelhart, Hinton, Williams, 1986)
  - **Backprop** is a special case of **backward differentiation**
  - Which relies on **computation graphs**.

Computation Graphs

A computation graph represents the process of computing a mathematical expression,

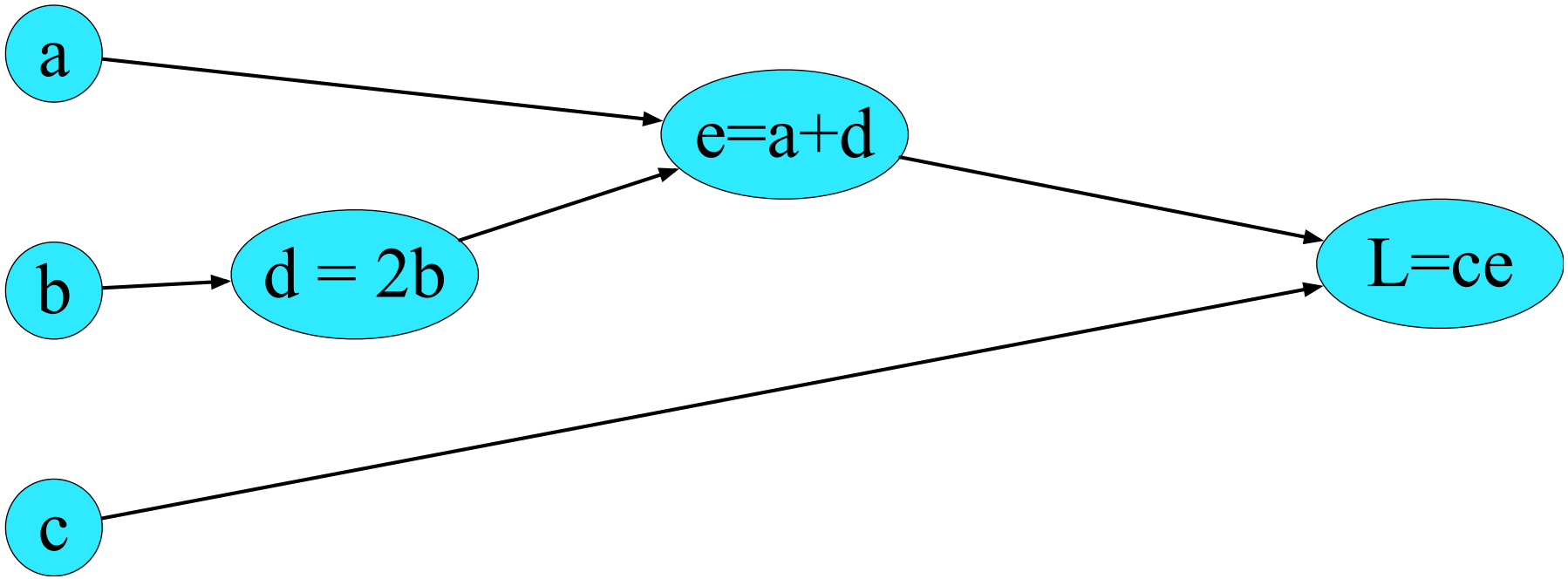
in which the computation is broken down into separate operations, each of which is modeled as a node in a graph.

Example:

$$L(a,b,c) = c(a+2b)$$

Computations:

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

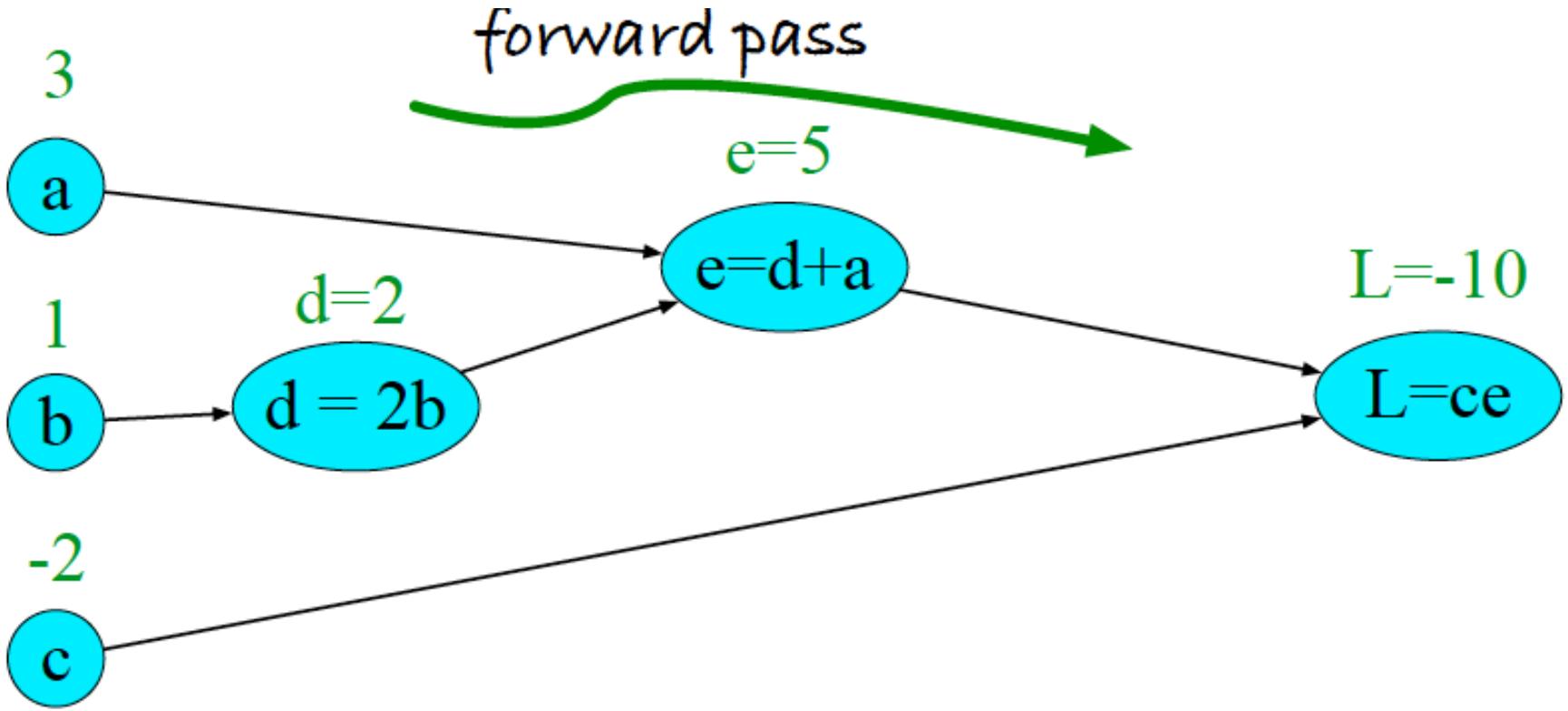


Example:

$$L(a,b,c) = c(a+2b)$$

Computations:

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

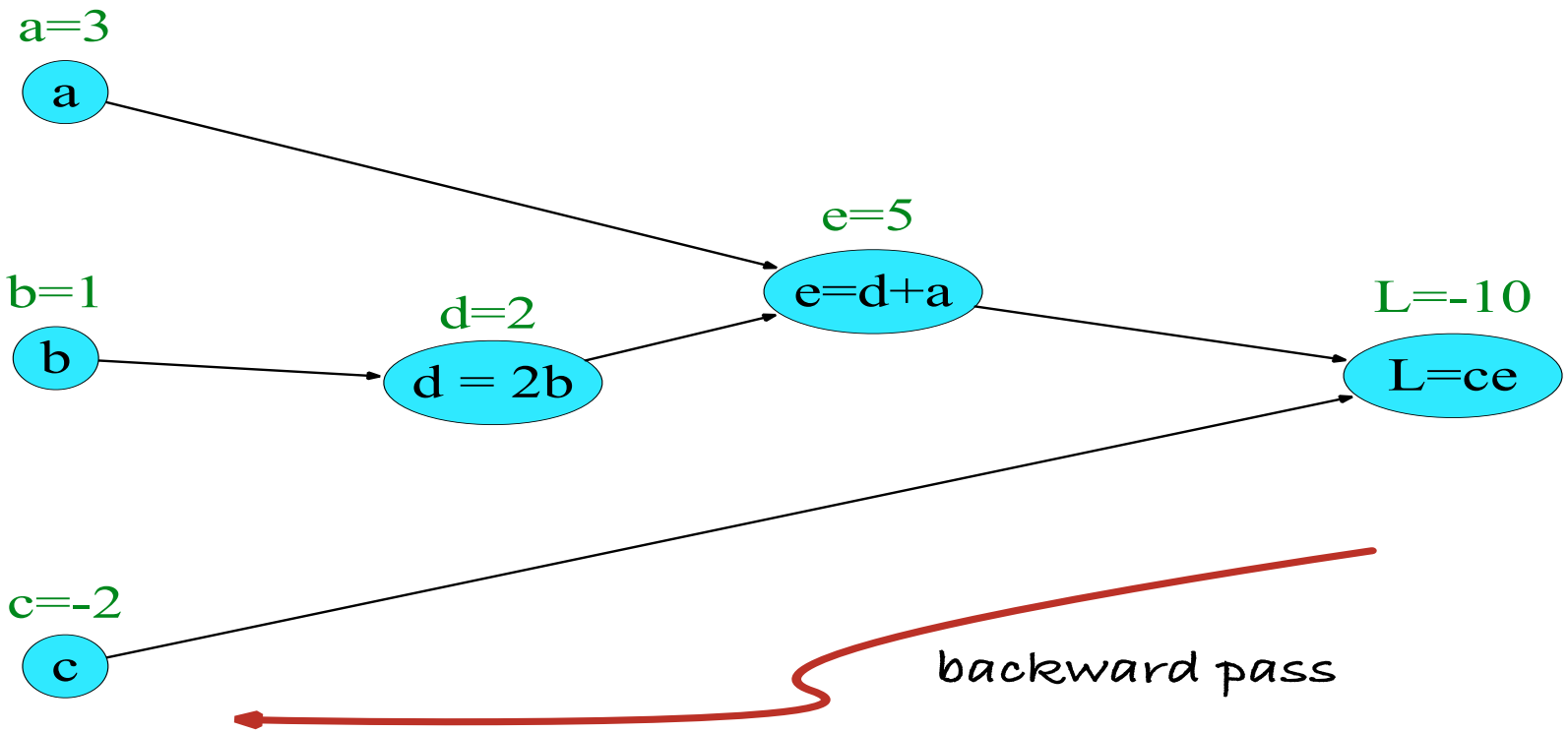


Example:

$$L(a,b,c) = c(a+2b)$$

Computations:

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



Backwards differentiation in computation graphs

- The importance of the computation graph comes from the backward pass
- This is used to compute the derivatives that we'll need for the weight update.

Example

$$L(a,b,c) = c(a+2b)$$

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

We want: $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$

The derivative $\frac{\partial L}{\partial a}$, tells us how much a small change in a affects L, while holding the others constant.

The chain rule

- Computing the derivative of a composite function:

- $f(x) = u(v(x))$

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$$

- $f(x) = u(v(w(x)))$

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx}$$

Example

$$L(a,b,c) = c(a+2b)$$

$$d = 2*b$$

$$e = a+d$$

$$L = c*e$$

$\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$

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

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

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

$$L = ce : \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

$$e = a+d : \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

$$d = 2b : \quad \frac{\partial d}{\partial b} = 2$$

Example

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

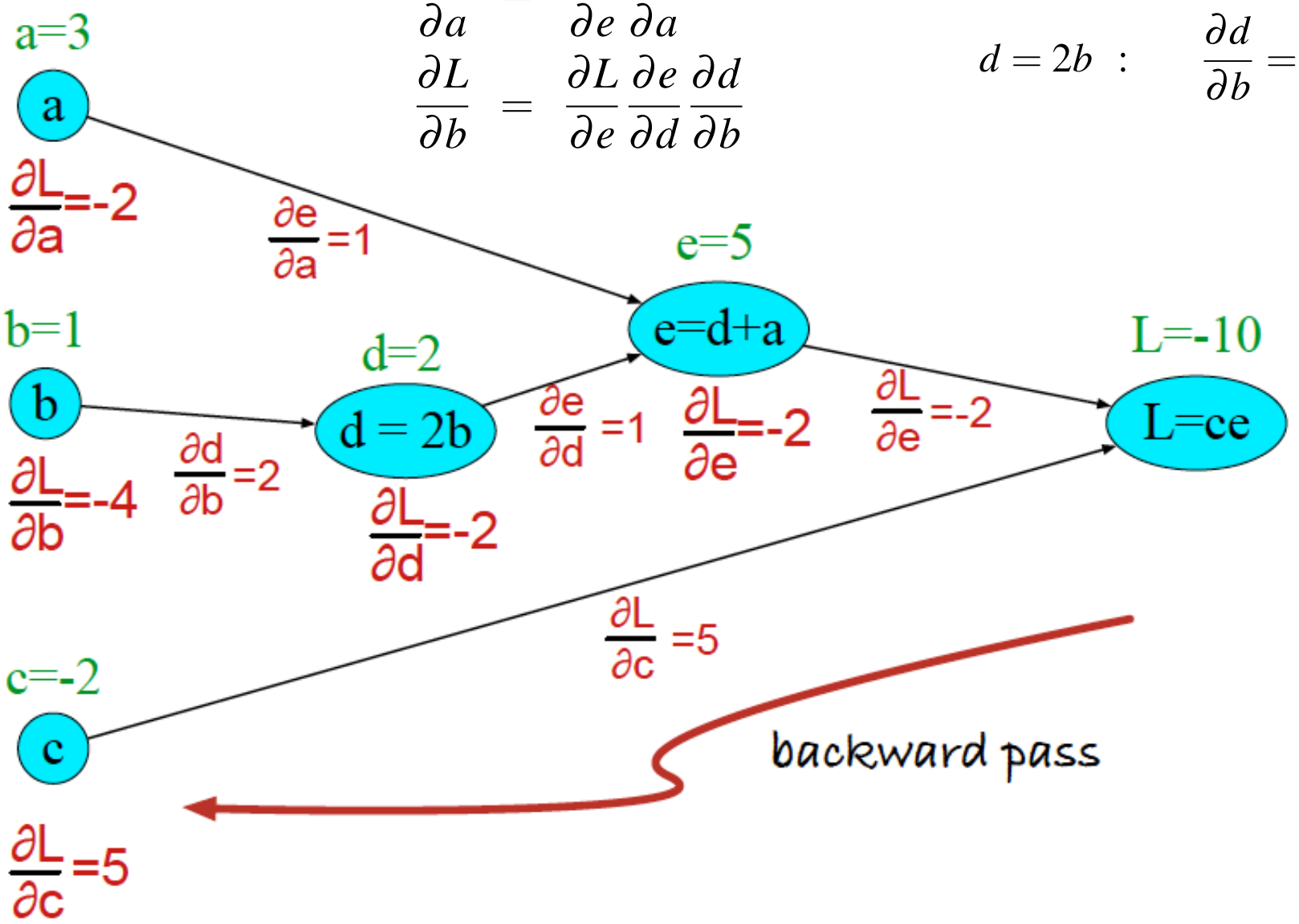
$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

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

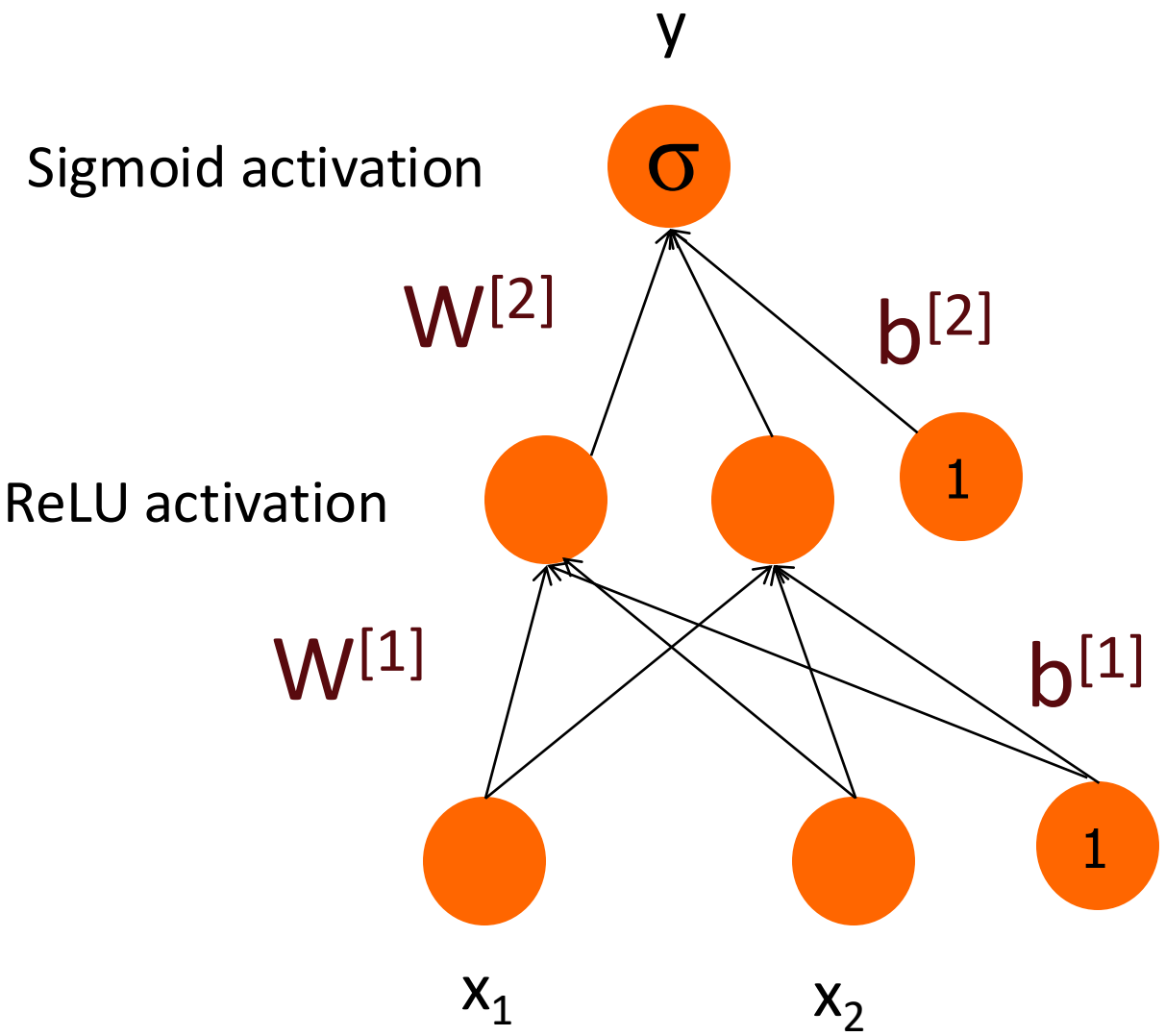
$$L = ce : \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

$$e = a + d : \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

$$d = 2b : \quad \frac{\partial d}{\partial b} = 2$$



Backward differentiation on a two layer network



$$
\begin{aligned}
z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\
a^{[1]} &= \text{ReLU}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

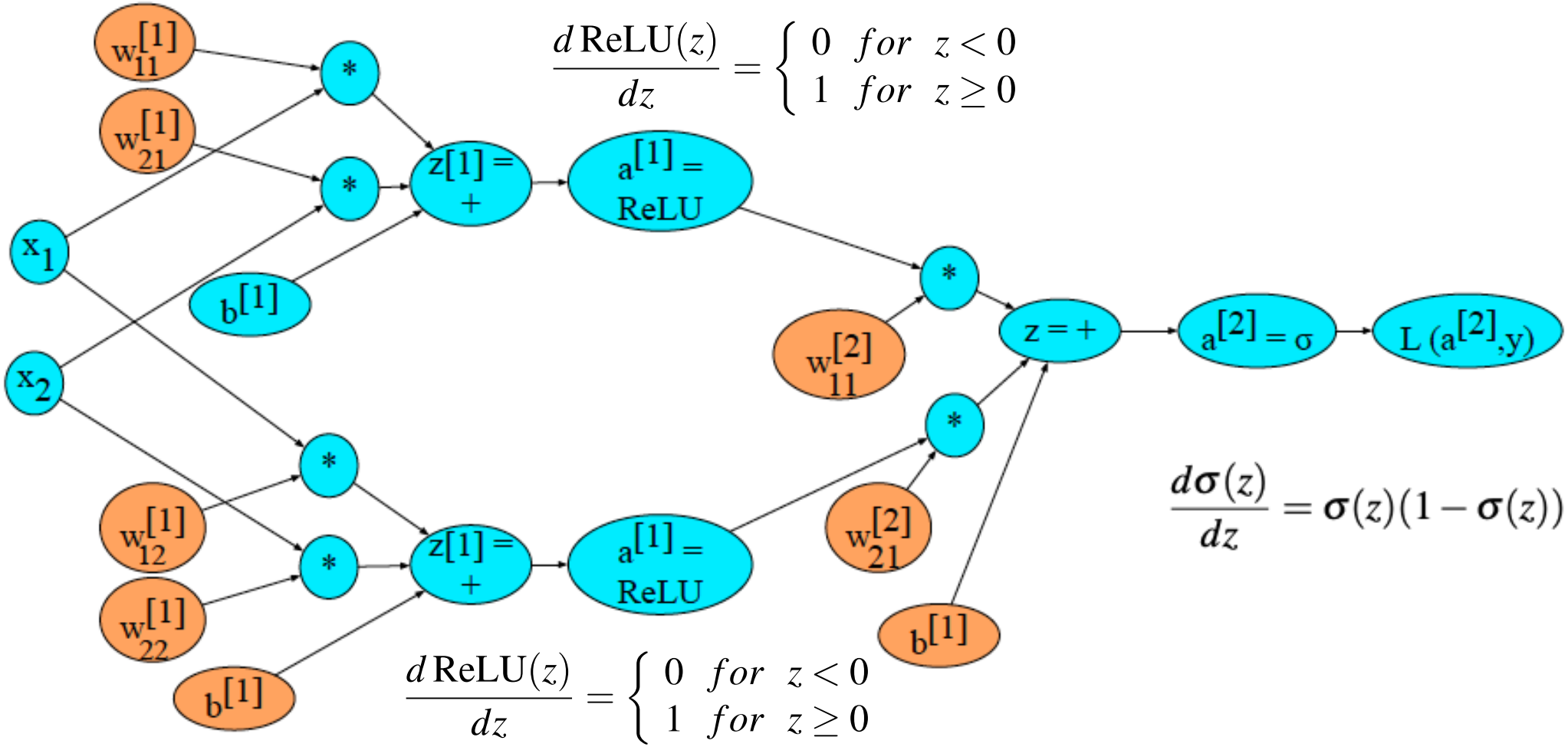
Backward differentiation on a two layer network

$$
\begin{aligned}
z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\
a^{[1]} &= \mathrm{ReLU}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

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

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

Backward differentiation on a 2-layer network



Starting off the backward pass: $\frac{\partial L}{\partial z}$

(I'll write a for $a^{[2]}$ and z for $z^{[2]}$)

$$
\begin{aligned}
z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\
a^{[1]} &= \mathbf{ReLU}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= \boldsymbol{\sigma}(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

$$L(\hat{y}, y) = -(\mathrm{y}\log(\hat{y}) + (1-y)\log(1-\hat{y}))$$

$$L(a, y) = -(\mathrm{y}\log a + (1-y)\log(1-a))$$

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

$$\frac{\partial L}{\partial a} = -\left(\left(y\frac{\partial \log(a)}{\partial a}\right) + (1-\mathrm{y})\ \frac{\partial \log(1-a)}{\partial a}\right)$$

$$= -\left(\left(y\frac{1}{a}\right) + (1-\mathrm{y})\ \frac{1}{1-a}(-1)\right) = -\left(\frac{y}{a} + \frac{y-1}{1-a}\right)$$

$$\frac{\partial a}{\partial z} = a(1-a) \qquad \frac{\partial L}{\partial z} = -\left(\frac{y}{a} + \frac{y-1}{1-a}\right)a(1-a) = \mathrm{a} - \mathrm{y}$$

Summary

- 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.

Use FeedForward Neural Networks in NLP

Use cases for feedforward networks

- Let's consider 2 (simplified) sample tasks:
  1. Text classification
  2. Language modeling

- State of the art systems use more powerful neural architectures, but simple models are useful to consider!

Classification: Sentiment Analysis

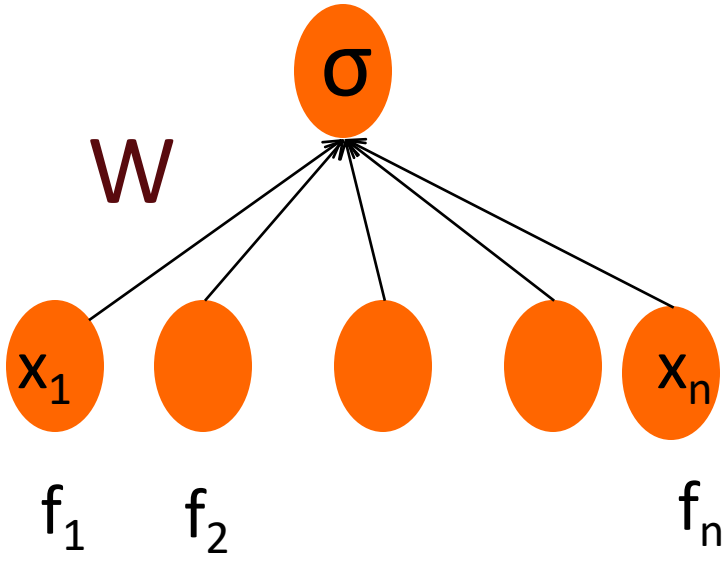
- We could do exactly what we did with logistic regression
- Input layer are binary features as before
- Output layer is 0 or 1

Sentiment Features

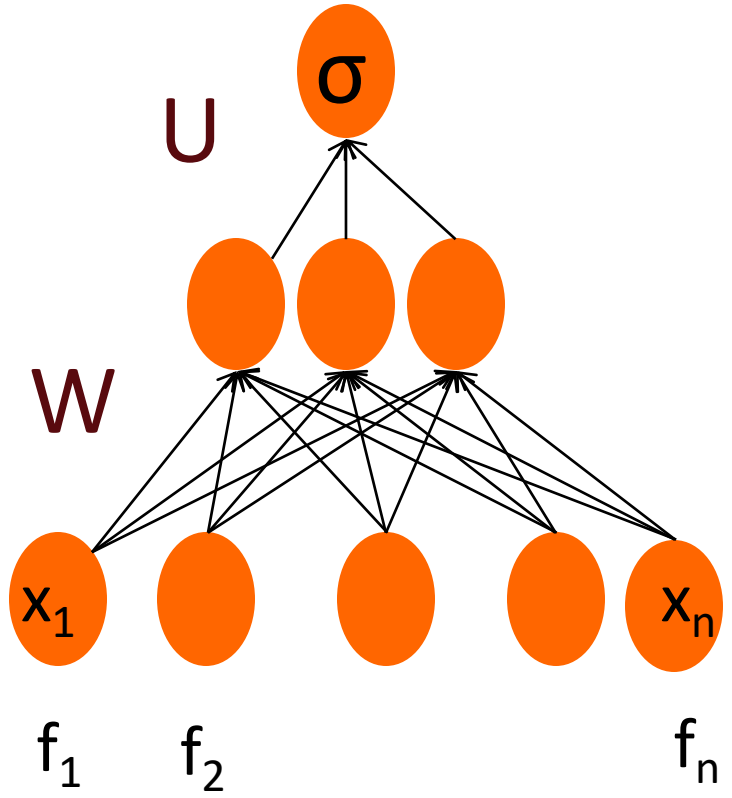
Var	Definition
x_1	count(positive lexicon) $\in$ doc)
x_2	count(negative lexicon) $\in$ doc)
x_3	$\begin{cases} 1 & \text{if } \text{“no”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$
x_4	count(1st and 2nd pronouns $\in$ doc)
x_5	$\begin{cases} 1 & \text{if } \text{“!”} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$
x_6	log(word count of doc)

Feedforward nets for simple classification

Logistic Regression



2-layer feedforward network

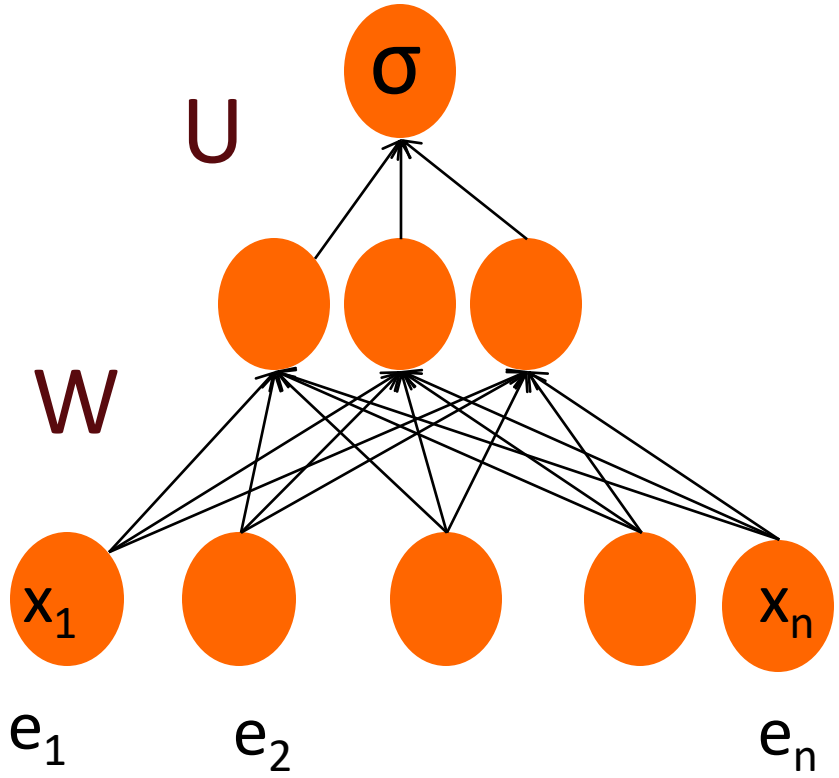


- Just adding a hidden layer to logistic regression
- allows the network to use non-linear interactions between features
- which may (or may not) improve performance.

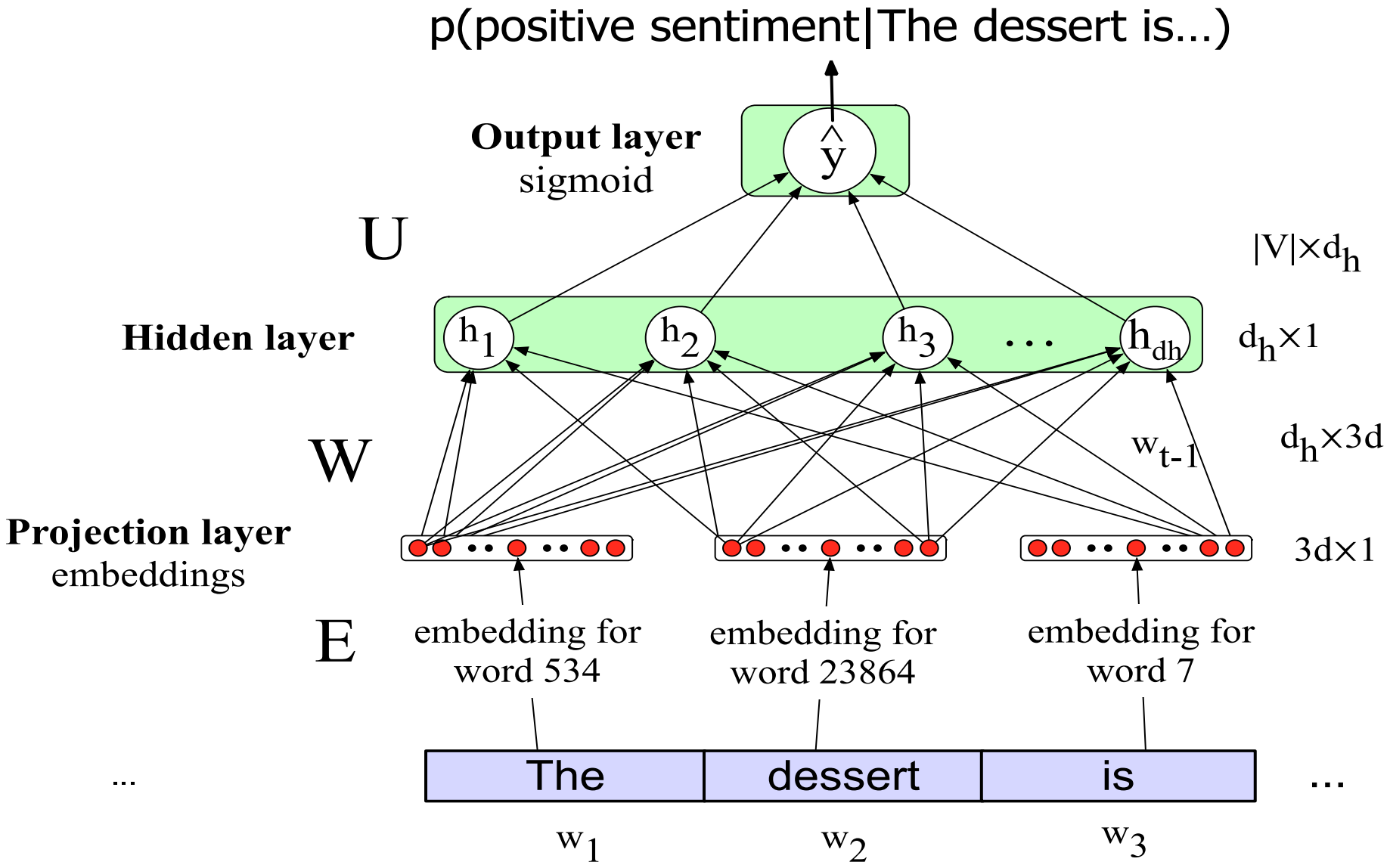
Even better: representation learning

- The real power of deep learning comes from the ability to **learn** features from the data
- Instead of using hand-built human-engineered features for classification
- Use learned representations like embeddings!

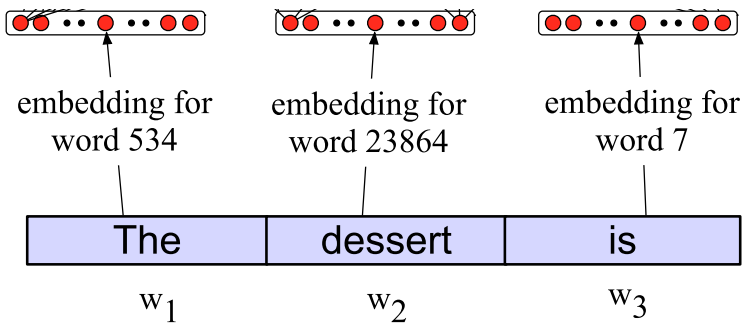
(e.g. word2vec, gloVe, FastText)



Neural Net Classification with embeddings as input features!



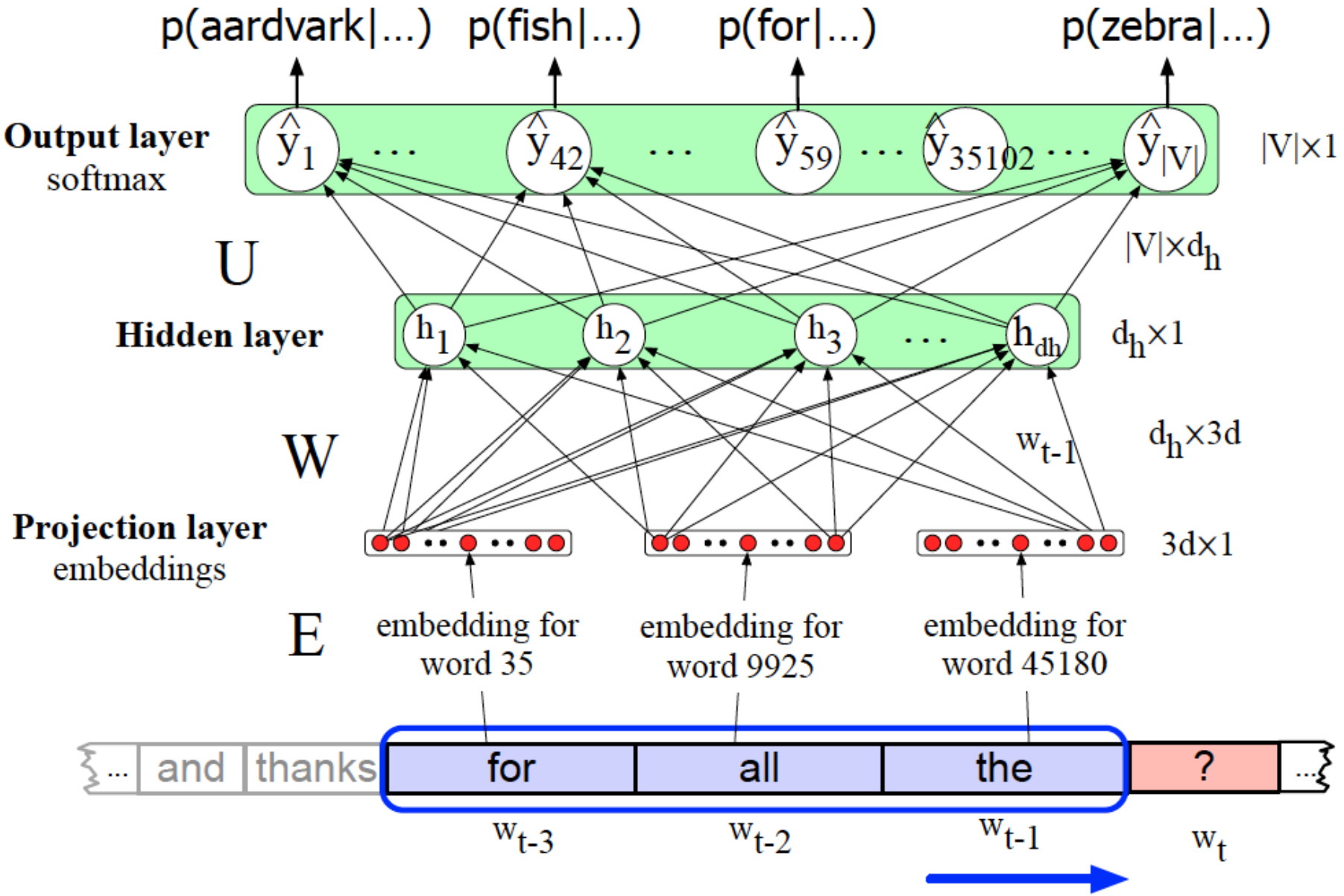
Issue: texts come in different sizes



- This assumes a fixed size length (3)!
- Kind of unrealistic.
- Some simple solutions (more sophisticated solutions later)

1. Make the input the length of the longest review
   - If shorter then pad with zero embeddings
   - Truncate if you get longer reviews at test time
2. Create a single "sentence embedding" (the same dimensionality as a word) to represent all the words
   - Take the mean of all the word embeddings
   - Take the element-wise max of all the word embeddings
     - For each dimension, pick the max value from all words

Neural Language Model



Why Neural LMs work better than N-gram LMs

- **Training data:**
- We've seen: I have to make sure that the cat gets fed.
- Never seen: dog gets fed
- **Test data:**
- I forgot to make sure that the dog gets ___
- N-gram LM can't predict "fed"!
- Neural LM can use similarity of "cat" and "dog" embeddings to generalize and predict “fed” after dog

Recap!

“Traditional or Conventional” Machine Learning:

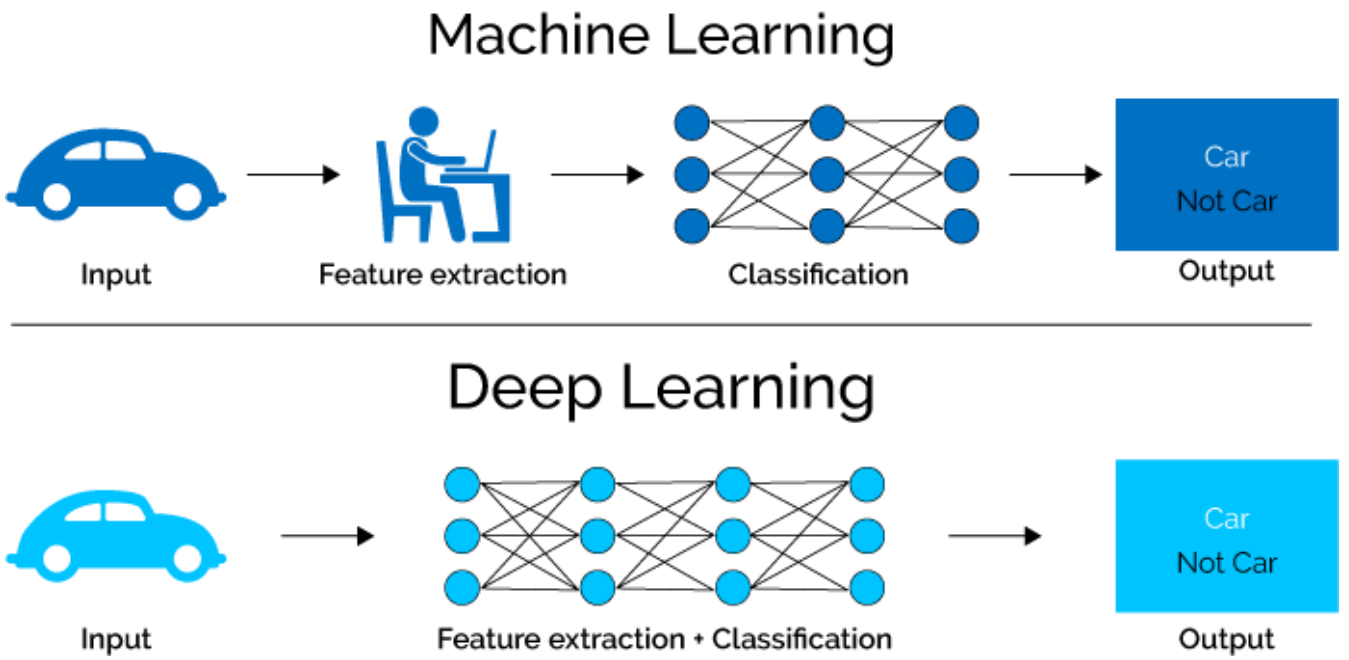
handcrafted features → learned classifier → **cat**

Deep, “end-to-end” learning:

learned low-level features → learned mid-level features → learned high-level features → learned classifier → **cat**

Conventional ML vs. Deep Learning

- *Conventional machine learning methods rely on human-designed feature representations (USED as inputs to ML models)*



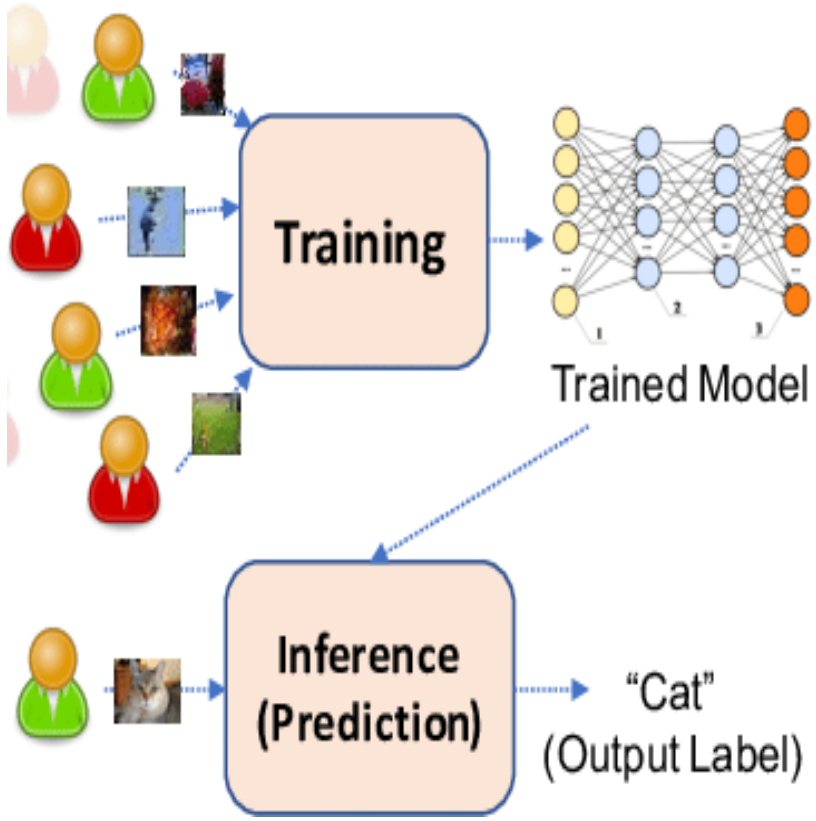
- *Deep learning, on the other hand, learns hierarchical representations directly from the data, eliminating the need for explicit feature engineering.*

Ref.https://www.xenonstack.com/blog/static/public/uploads/media/machine-learning-vs-deep-learning.png

1. **Model:** Represents the underlying structure or architecture that captures the relationship between input data and the desired output. It can be a mathematical or computational representation that maps inputs to outputs.

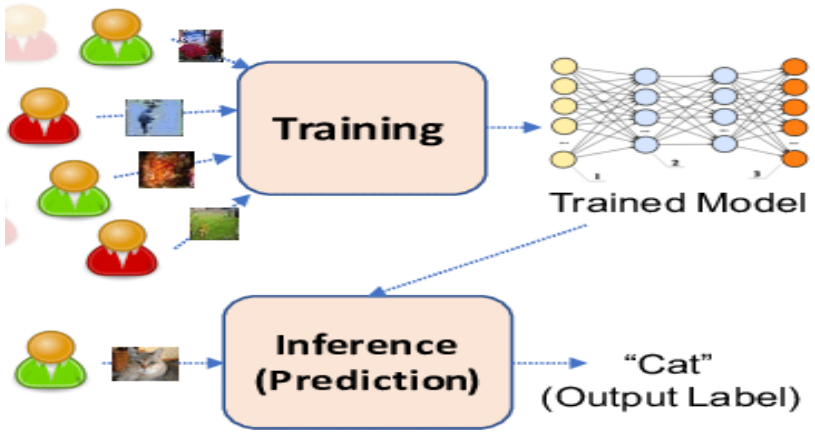
2. **Learning/Training**: The learning or training phase involves optimizing the model's parameters to minimize the difference between its predicted outputs and the true labels in the training data. The model learns from the available labeled data to capture the patterns and relationships necessary for accurate predictions.

3. **Inference:** The inference phase occurs after the model has been trained. It involves utilizing the learned model to make predictions or produce outputs on new, unseen data. In this phase, the model applies the learned patterns and relationships to make informed decisions or generate desired outcomes.



Optimization

- In the context of machine learning, learning can be seen as optimization problem that plays a critical role in finding the optimal values of model parameters or hyperparameters that minimize a specified cost or loss function
- Mathematical optimization:
  "the selection of a best element (with regard to some criterion) from some set of available alternatives" (Wikipedia)



- Learning as an optimization problem
  - cost function:

loss

regularization

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} L(f(\mathbf{x}_i;\theta), y_i) + R(\theta)$$

Some types of Deep Learning Models

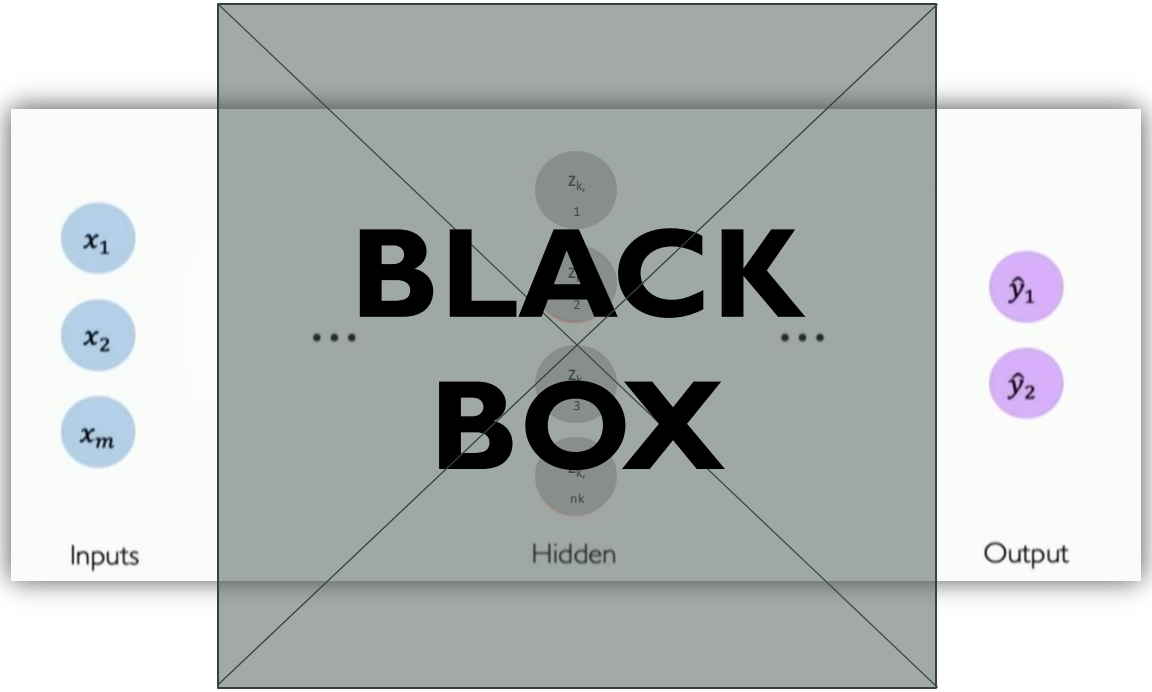
- **Multilayer Perceptrons (MLPs)**: A basic neural network architecture often used for classification tasks.
- **Convolutional Neural Networks (CNNs)**: Primarily used for image and video recognition tasks.
- **Recurrent Neural Networks (RNNs)**: Suitable for sequential data, like time series and natural language.
- **Transformer Models**: Known for their effectiveness in NLP tasks, especially with large-scale models like GPT.
- **Generative Adversarial Networks (GANs)**: Used for generating new, synthetic data resembling the training data.
- **Autoencoders**: Typically used for unsupervised learning tasks like dimensionality reduction and denoising.
- **Boltzmann Machines**: Utilized for pattern recognition and data representation tasks.
- ….

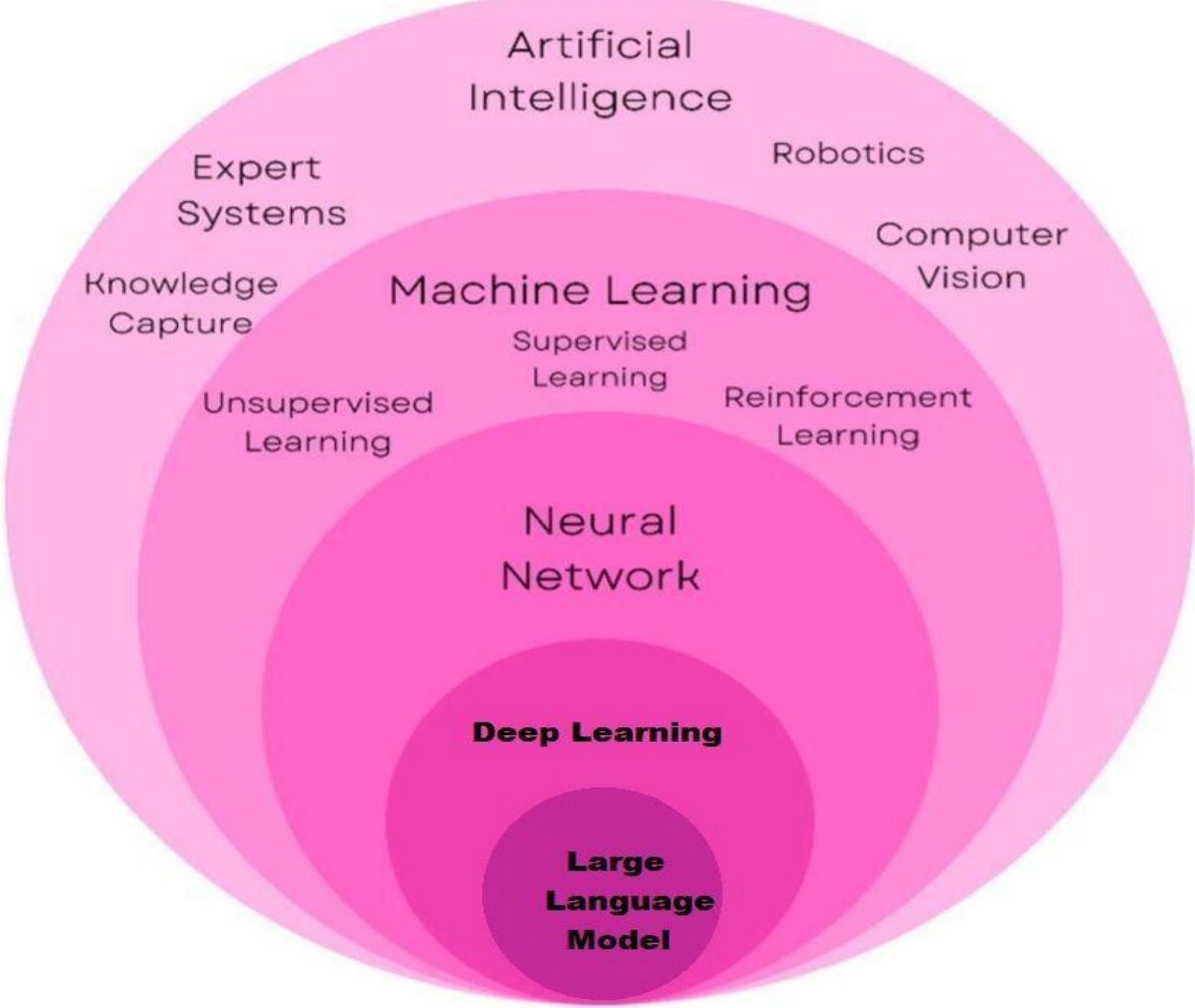
Deep Learning (Advantages and Disadvantages)

Advantages

1. Easily identifies trends and patterns
2. No human intervention needed
3. Continuous Improvement
4. Handling multi-dimensional and multi-variety data
5. Wide Applications

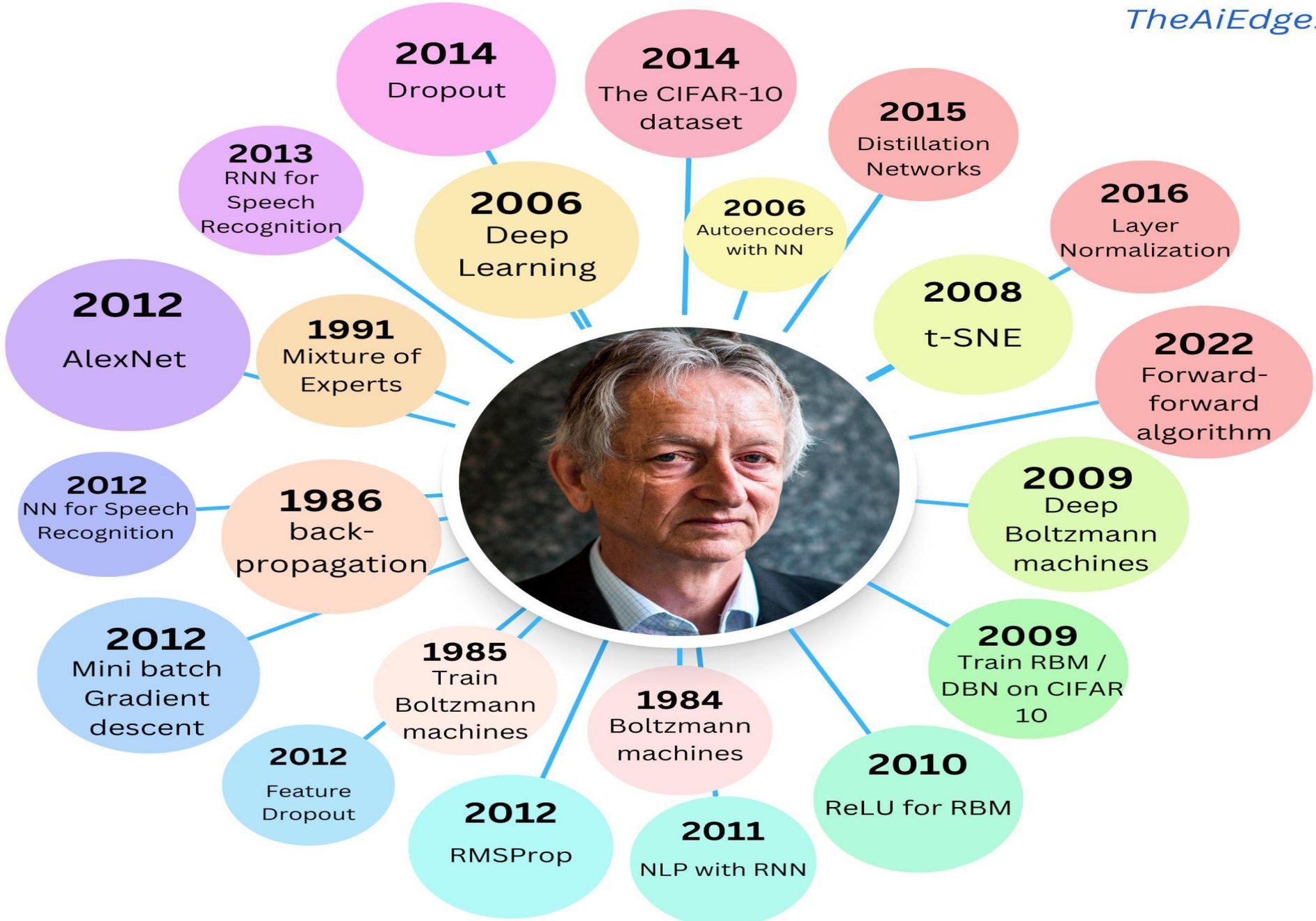
- Not easily explainable or interpretable




Artificial Intelligence
Expert Systems
Robotics
Computer Vision
Knowledge Capture
Machine Learning
Supervised Learning
Unsupervised Learning
Reinforcement Learning
Neural Network
Deep Learning
Large Language Model

Heroes of Machine Learning: Geoffrey Hinton

TheAiEdge.io



Reading

- **Important!.** Please refer to this chapter: Chapter 6
- https://medium.com/machine-learning-researcher/artificial-neural-network-ann-4481fa33d85a
- https://techvidvan.com/tutorials/artificial-neural-network/
- Deep Learning Specialization https://www.coursera.org/specializations/deep-learning

Exercise

- https://drive.google.com/file/d/1O2lUtoU3hiG3oYt6xNPkngYpct-C-lQF/view?usp=sharing

Next Class

- **Lecture 8: Text Representation (Part 2**