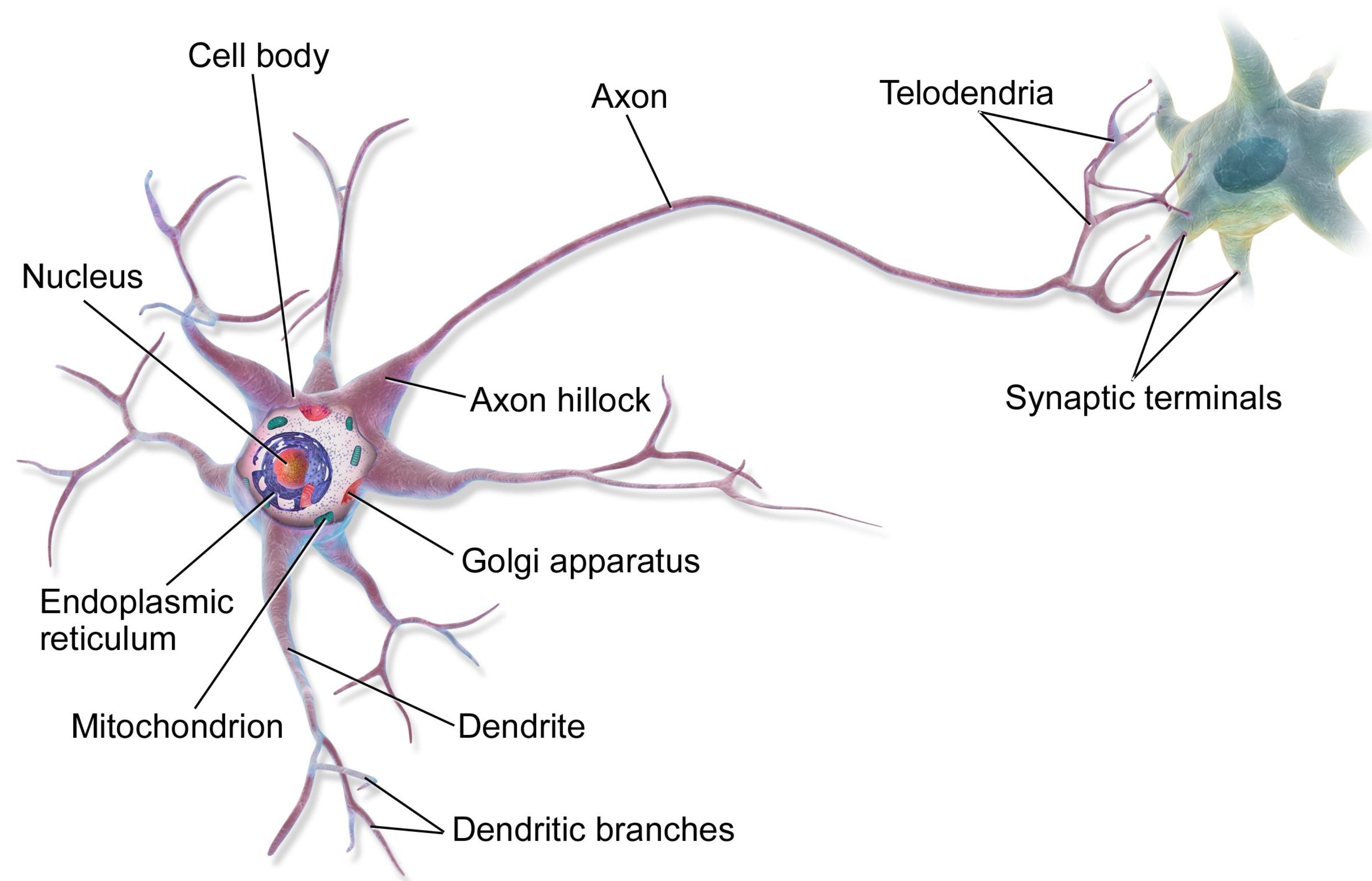
Simple Neural Networks and Neural Language Models

Units in Neural Networks

This is in your brain



By BruceBlaus - Own work, CC BY 3.0,
https://commons.wikimedia.org/w/index.php?curid=28761830

Neural Network Unit

This is not in your brain

Output value

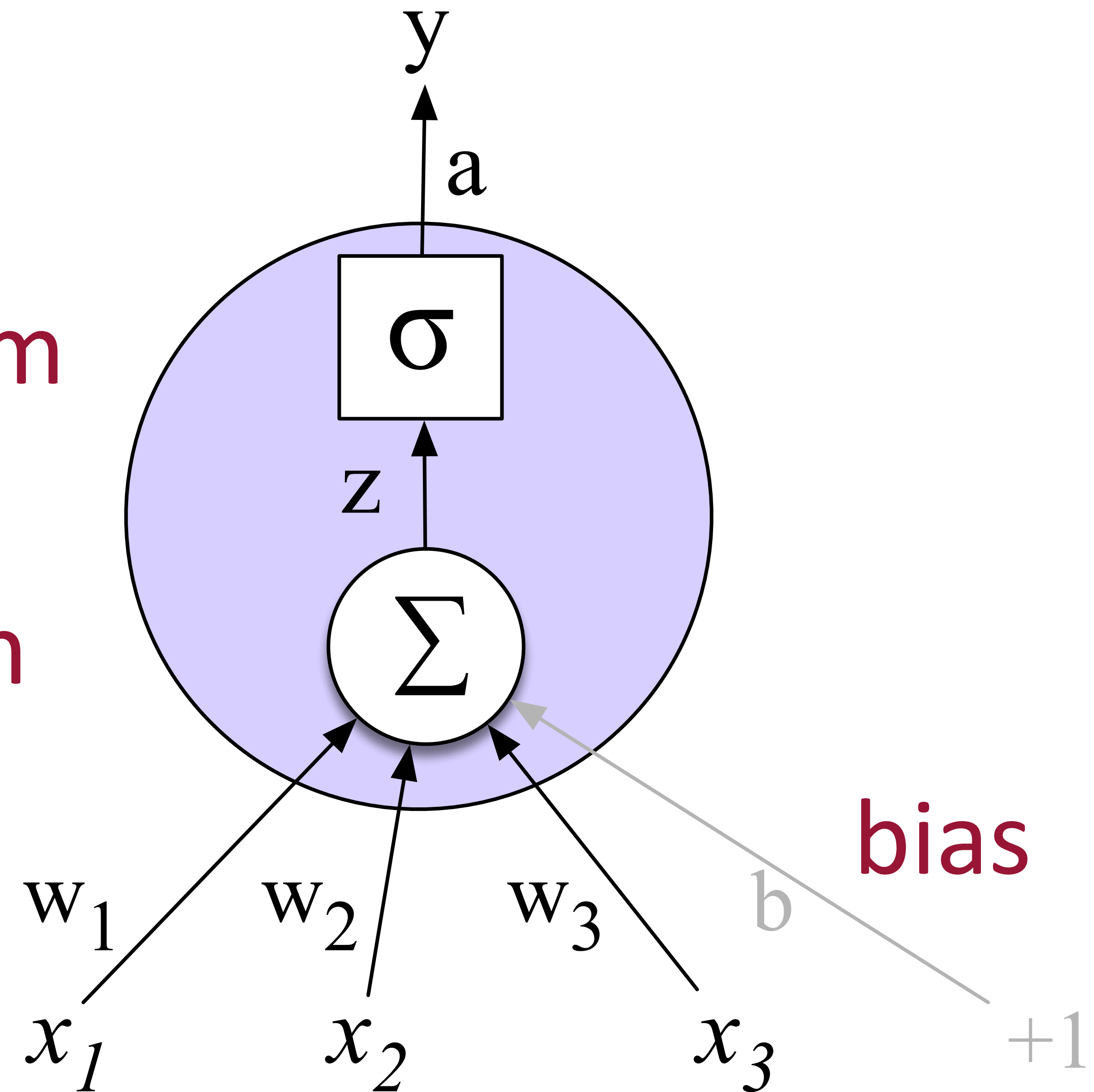
Non-linear transform

Weighted sum

Weights

Input layer

bias



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 f:

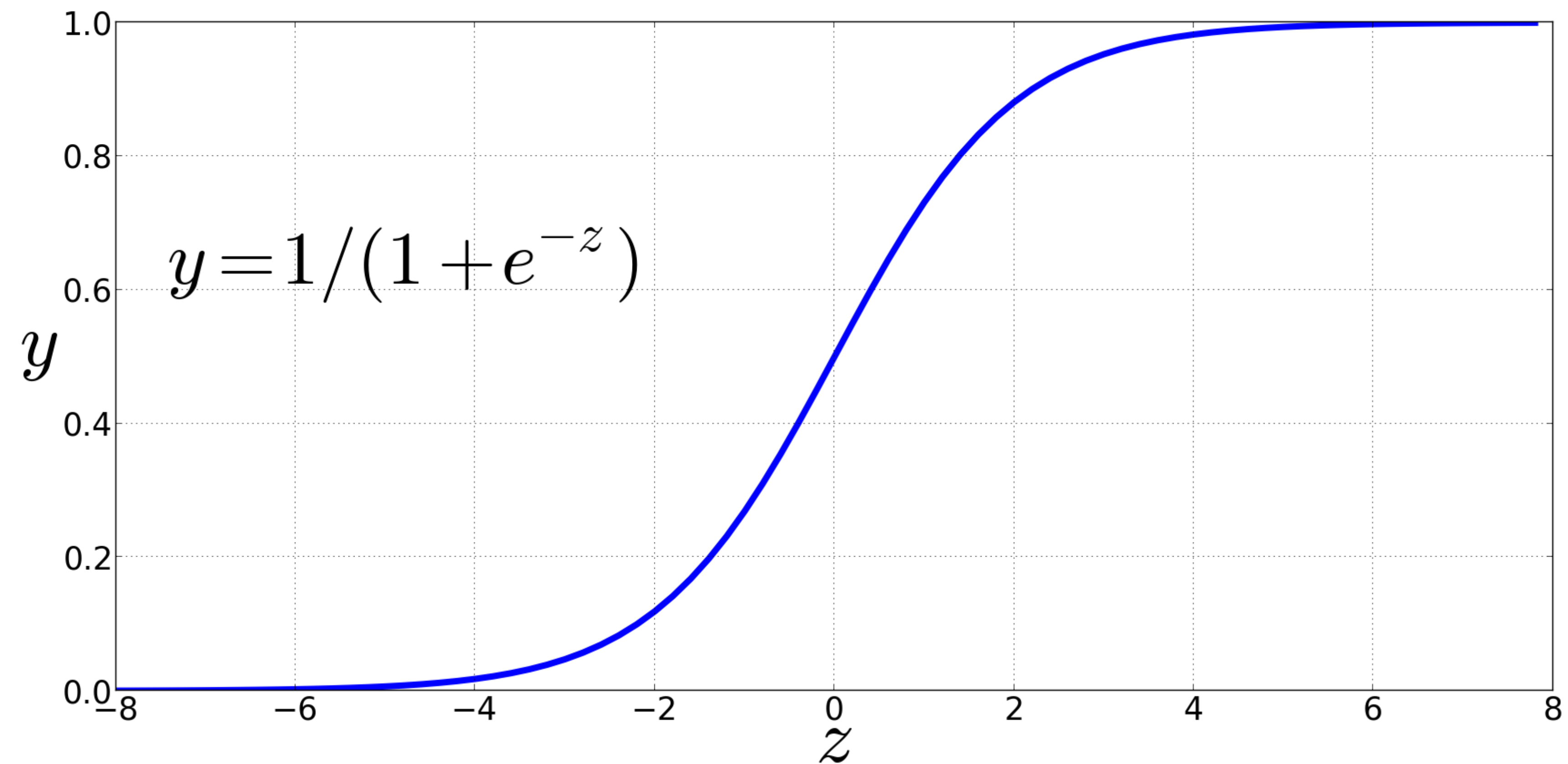
$$y = a = f(z)$$

Non-Linear Activation Functions

We're already seen the sigmoid for logistic regression:

Sigmoid

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



Final function the unit is computing

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

Final unit again

Output value

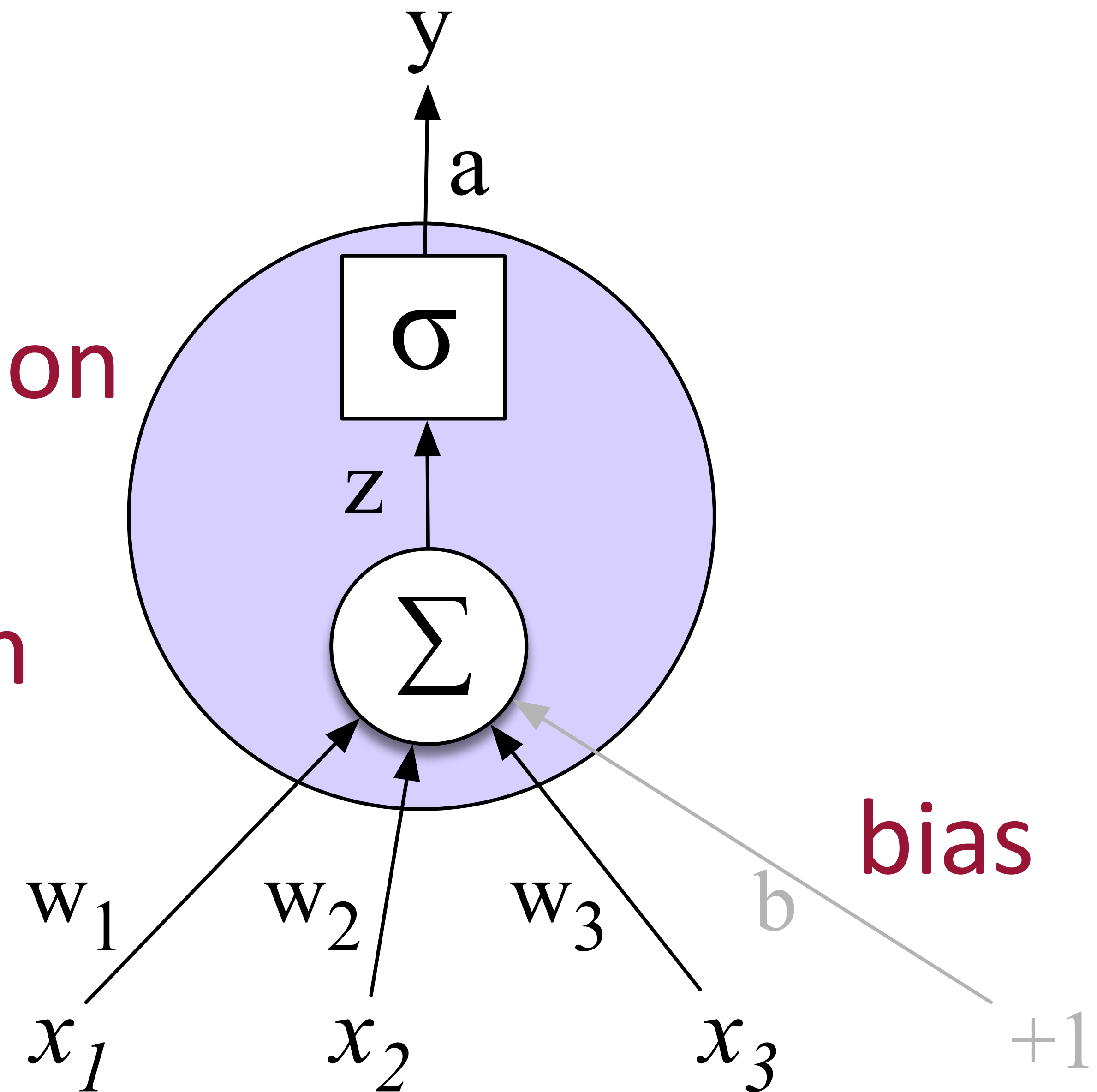
Non-linear activation function

Weighted sum

Weights

Input layer

bias



An example

Suppose a unit

$w = [0.2, 0.3, 0.9]$

$b = 0.5$

What happens with input x:

$x = [0.5, 0.6, 0.1]$

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

An example

Suppose a unit

$w = [0.2, 0.3, 0.9]$

$b = 0.5$

What happens with the following input x?

$x = [0.5, 0.6, 0.1]$

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

What happens with input x:

$x = [0.5, 0.6, 0.1]$

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

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

What happens with input x:

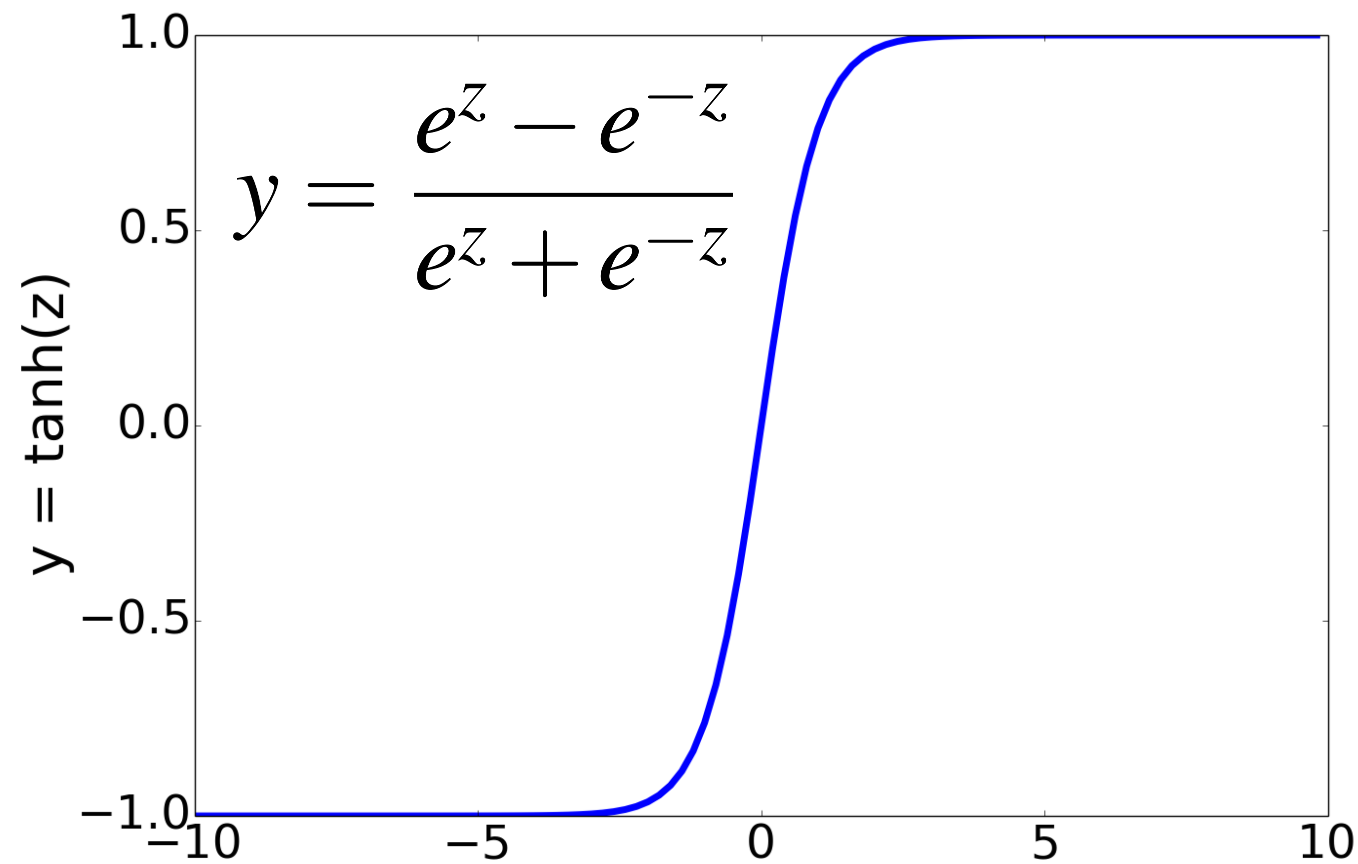
```
x = [0.5,0.6,0.1]
```

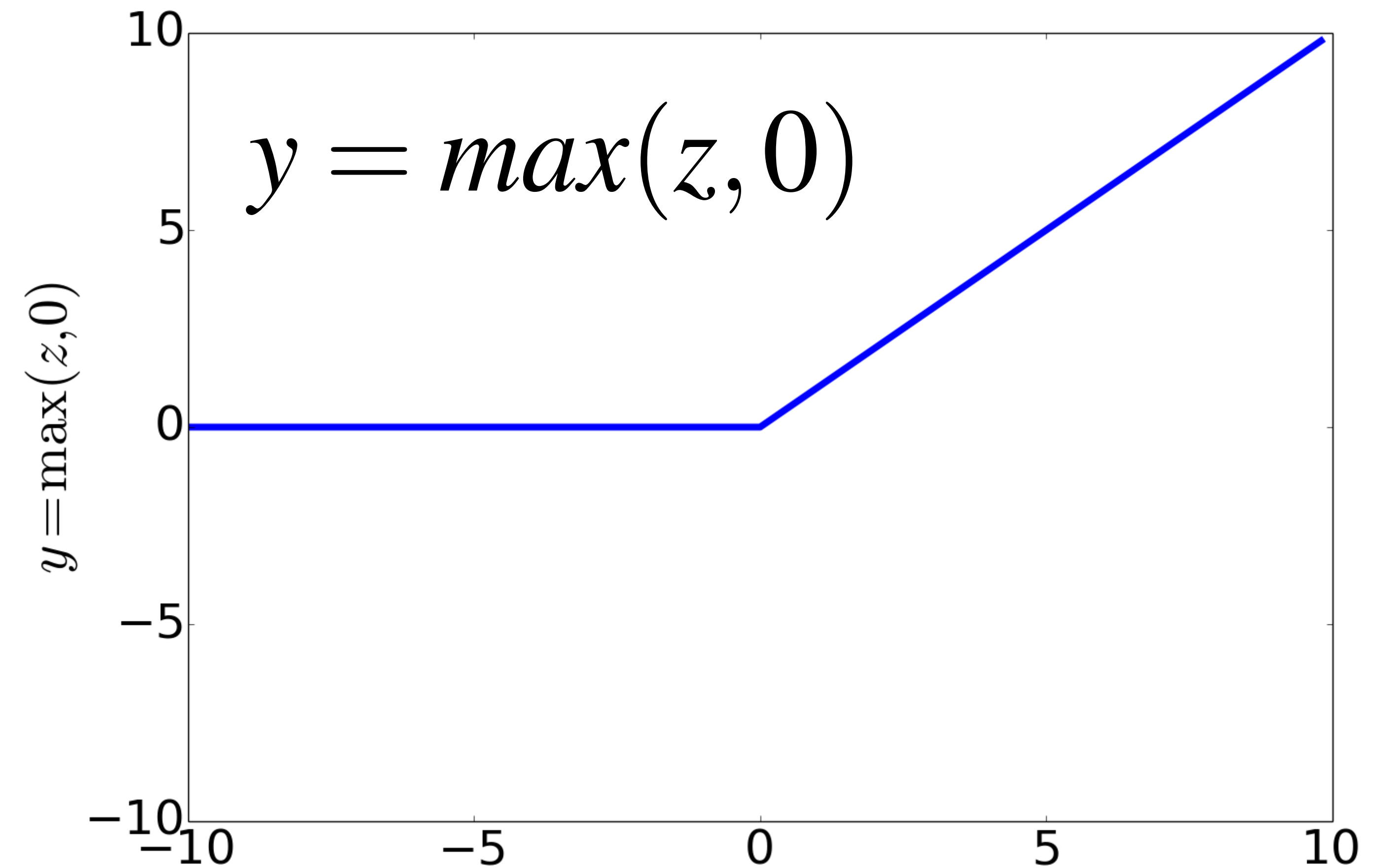
$$y = \sigma(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}} =$$

$$\frac{1}{1 + e^{-(.5*.2+.6*.3+.1*.9+.5)}} = \frac{1}{1 + e^{-0.87}} = .70$$

Non-Linear Activation Functions besides sigmoid

Most Common:





tanh

ReLU

Rectified Linear Unit

Simple Neural Networks and Neural Language Models

Units in Neural Networks

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

Perceptrons

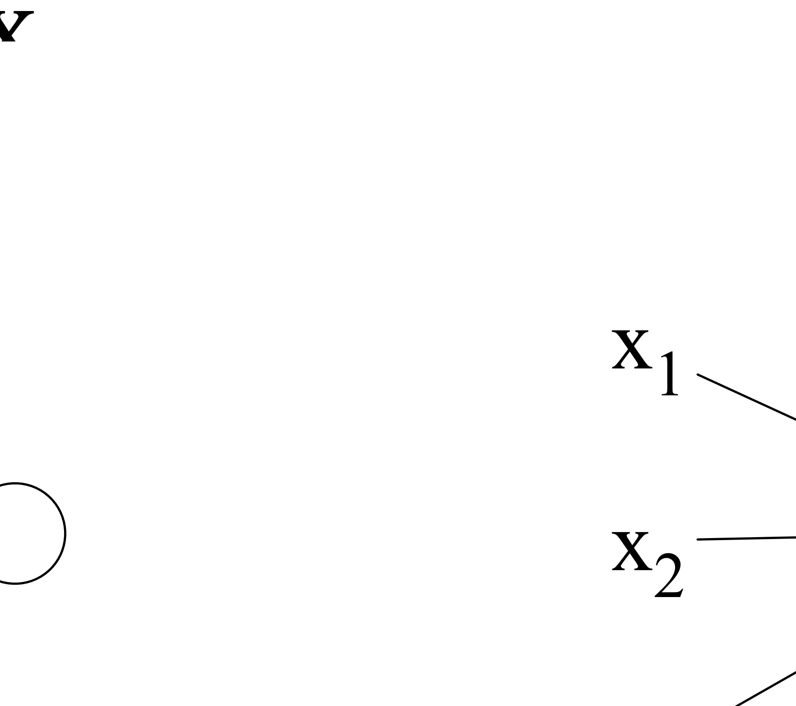
A very simple neural unit

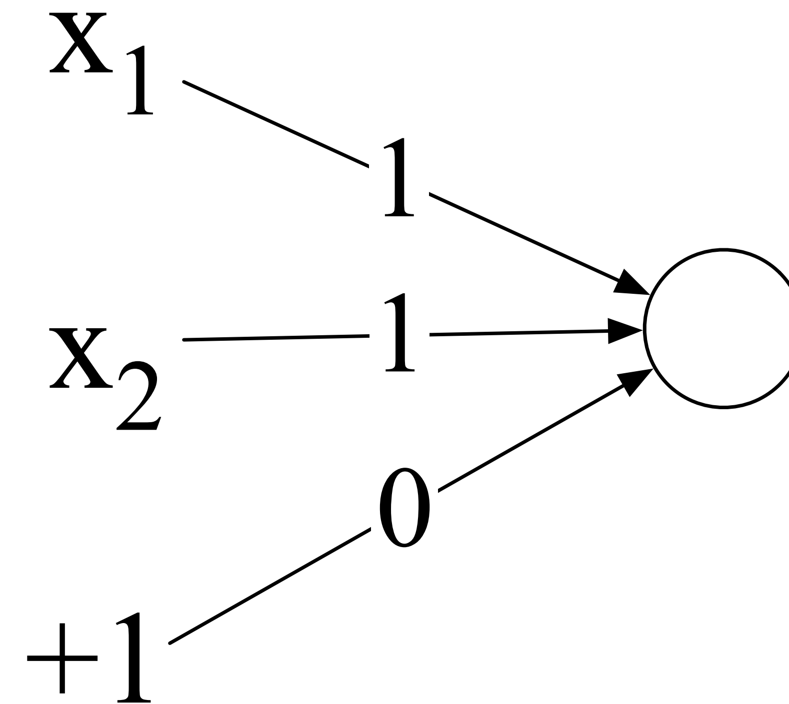
- Binary output (0 or 1)
- No non-linear activation function

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

erceptrons

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





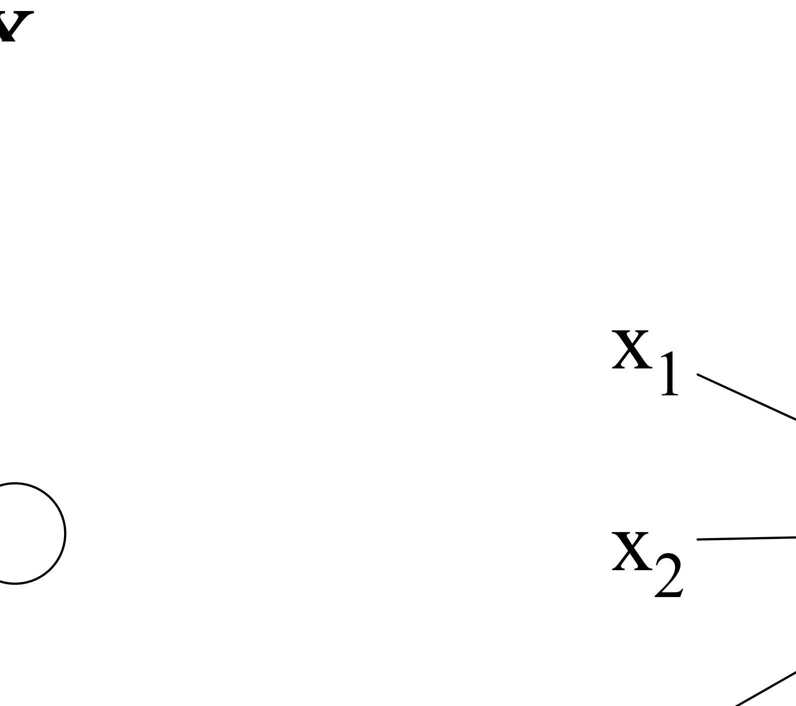
OR

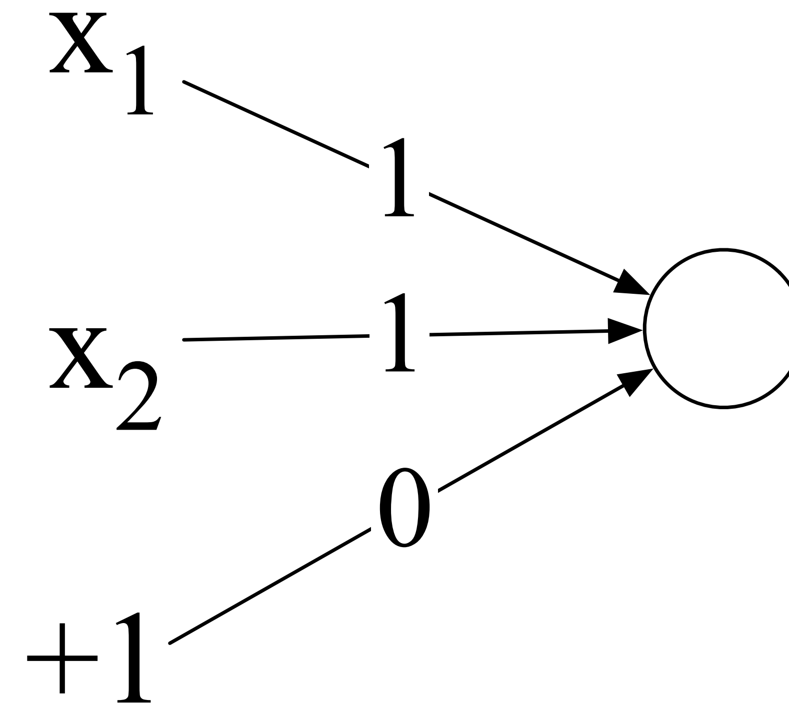
OR

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

erceptrons

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



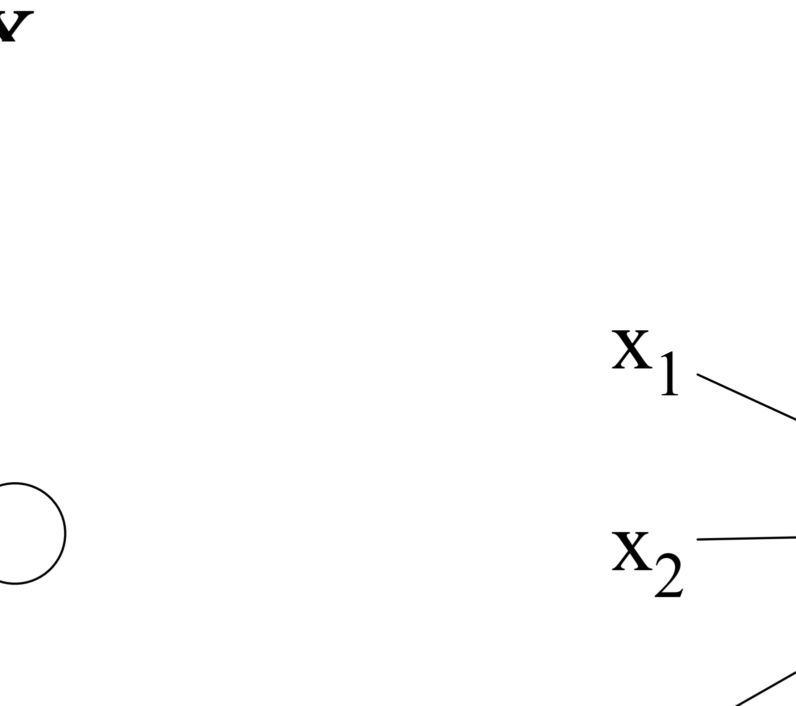


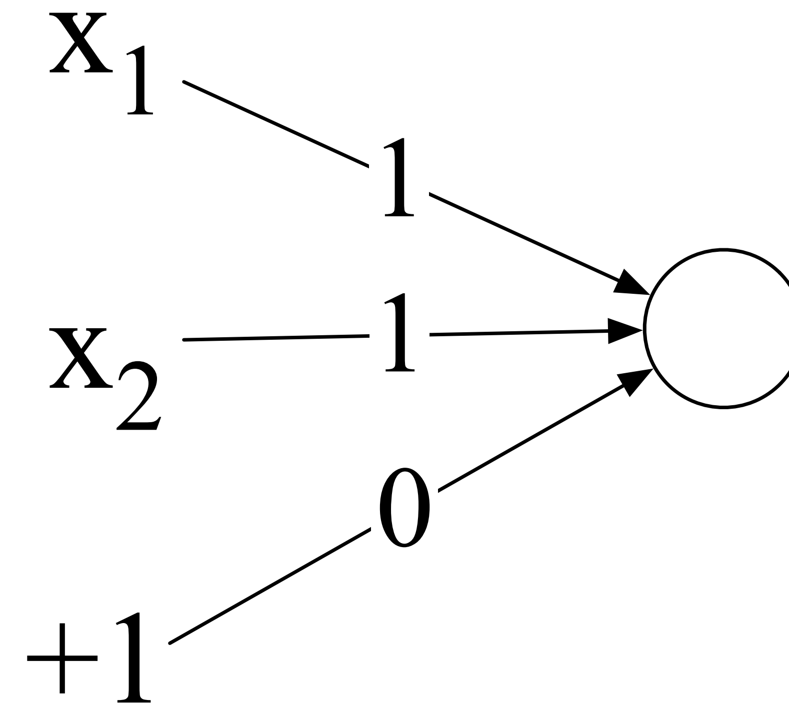
OR

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

erceptrons

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





OR

OR

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

Not possible to capture XOR with perceptrons

Pause the lecture and try for yourself!

Why? Perceptrons are linear classifiers

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

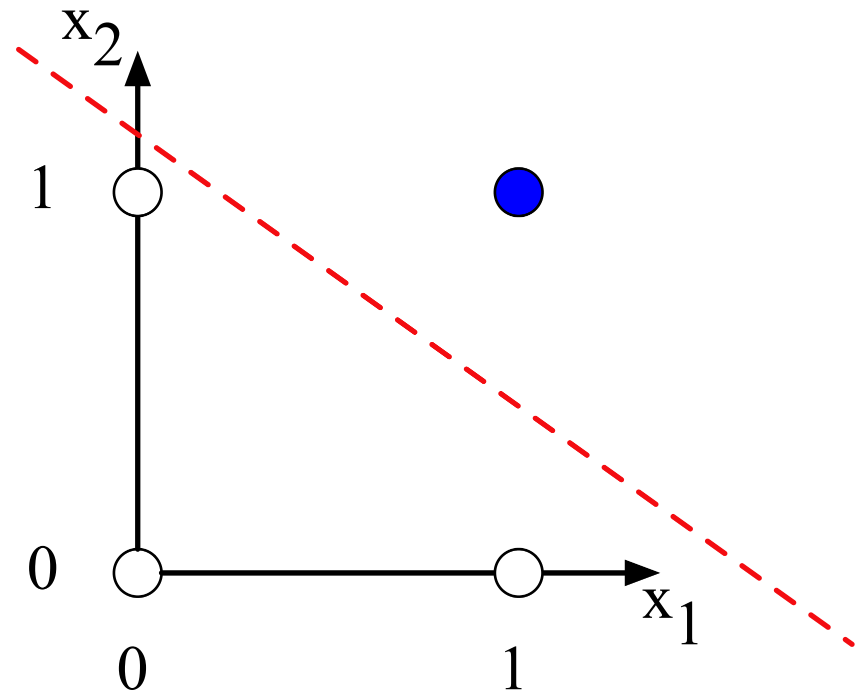
$$w_1x_1 + w_2x_2 + b = 0$$

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

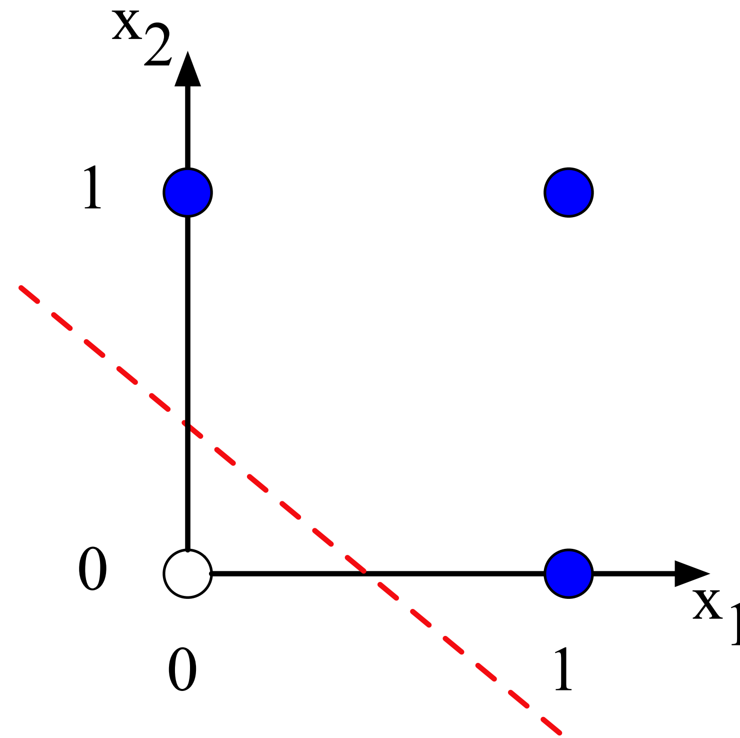
This line acts as a **decision boundary**

- 0 if input is on one side of the line
- 1 if on the other side of the line

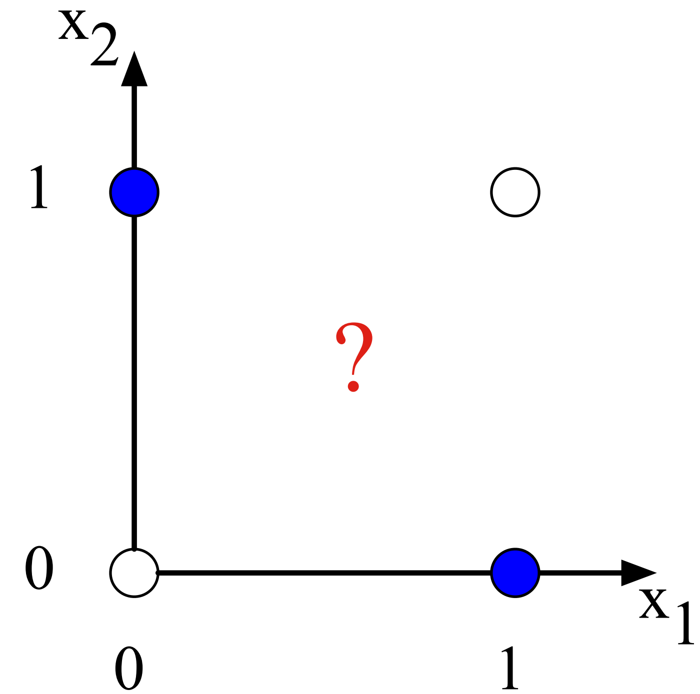
Decision boundaries



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!

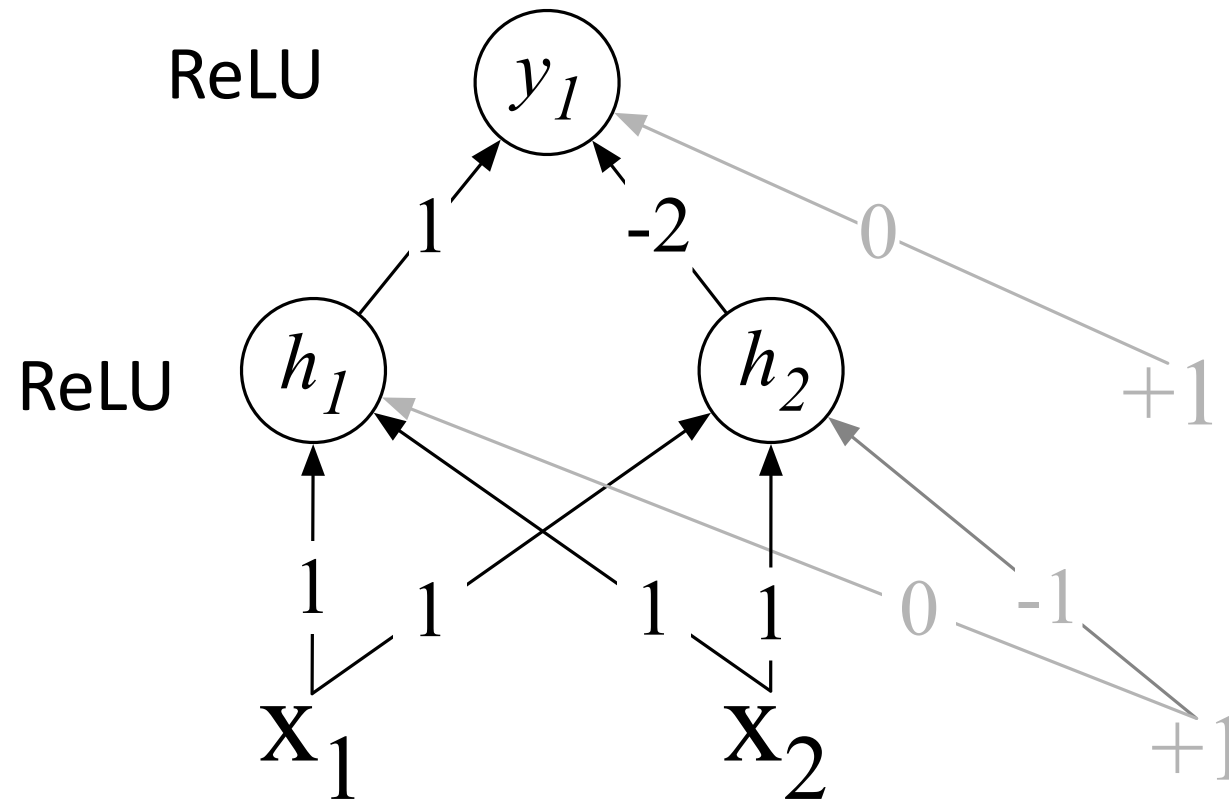
Solution to the XOR problem

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



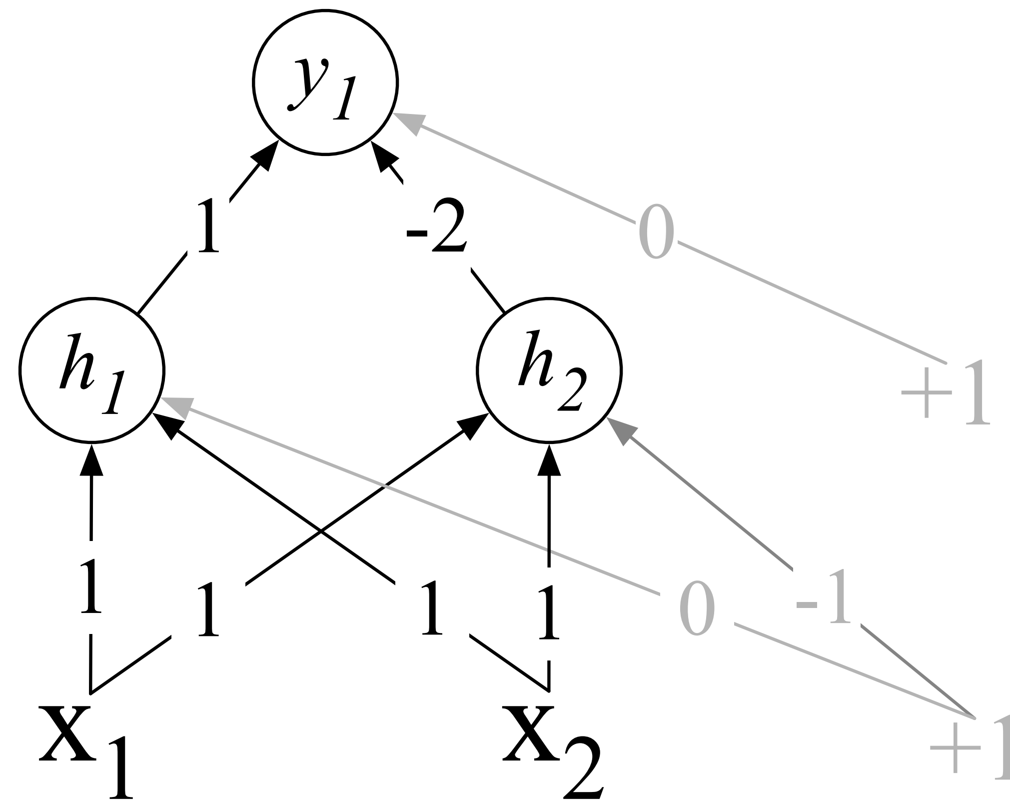
Solution to the XOR problem

XOR **can't** be calculated by a single perceptron

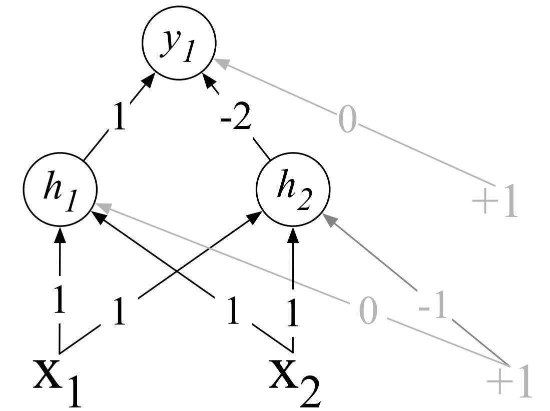
XOR **can** be calculated by a layered network of units.

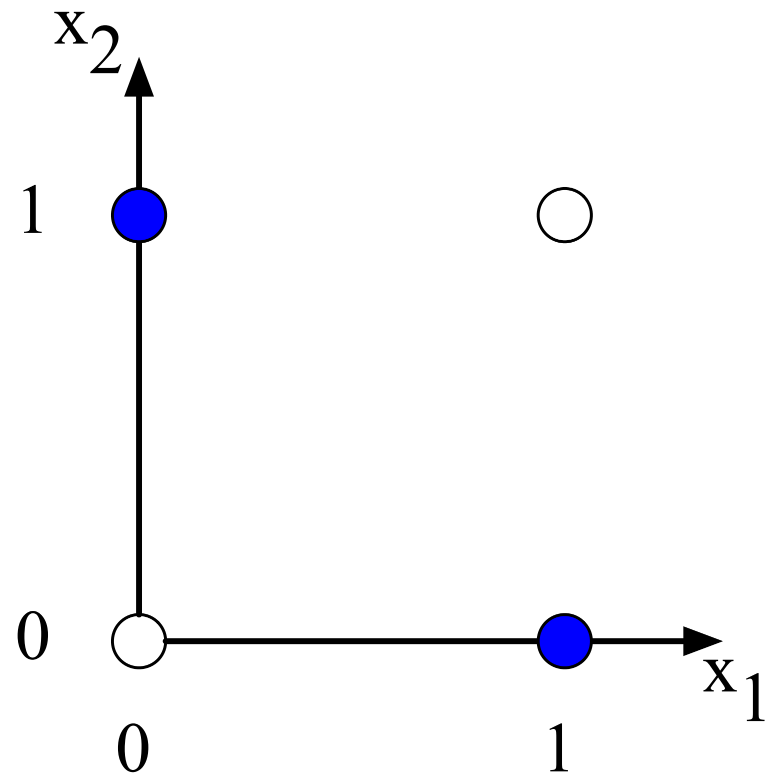
XOR

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

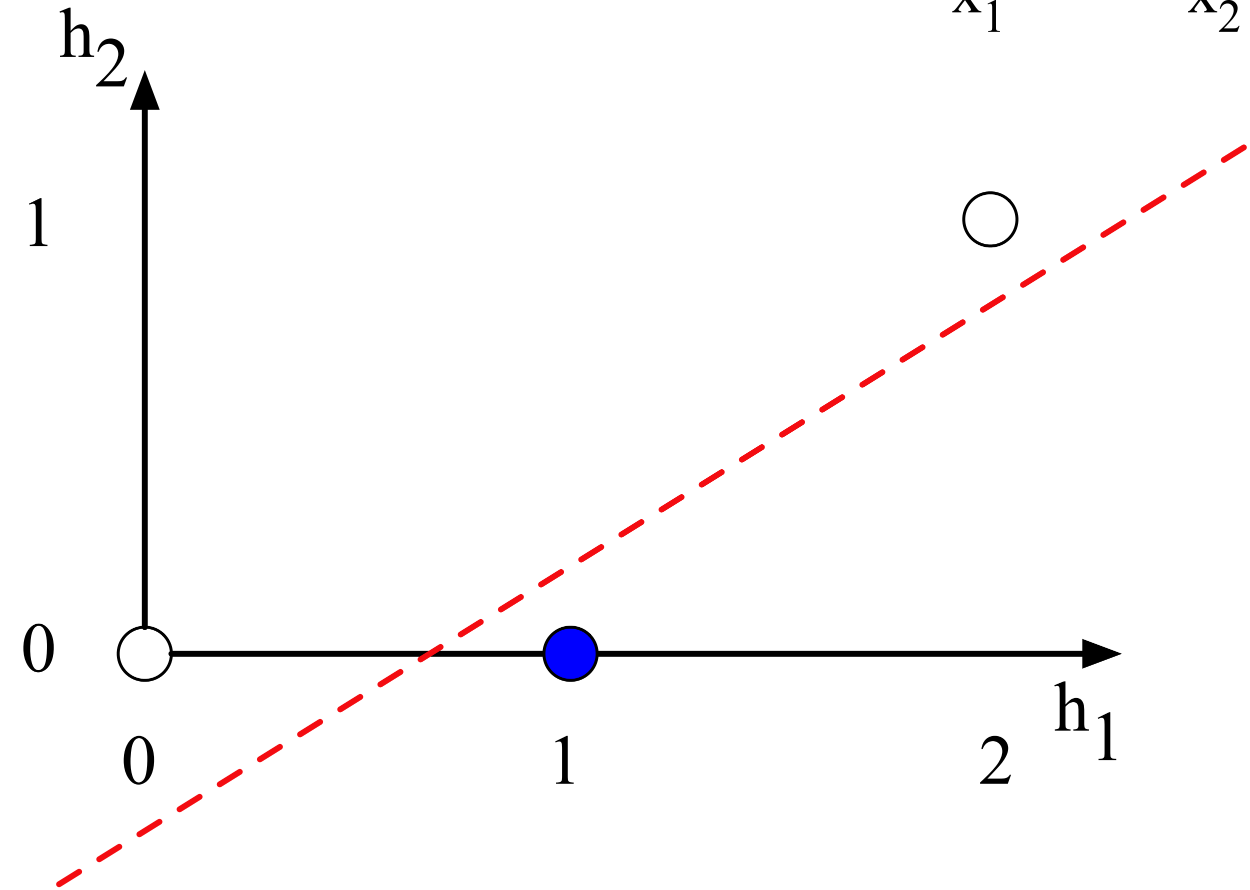


The hidden representation h





a) The original *x* space



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

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

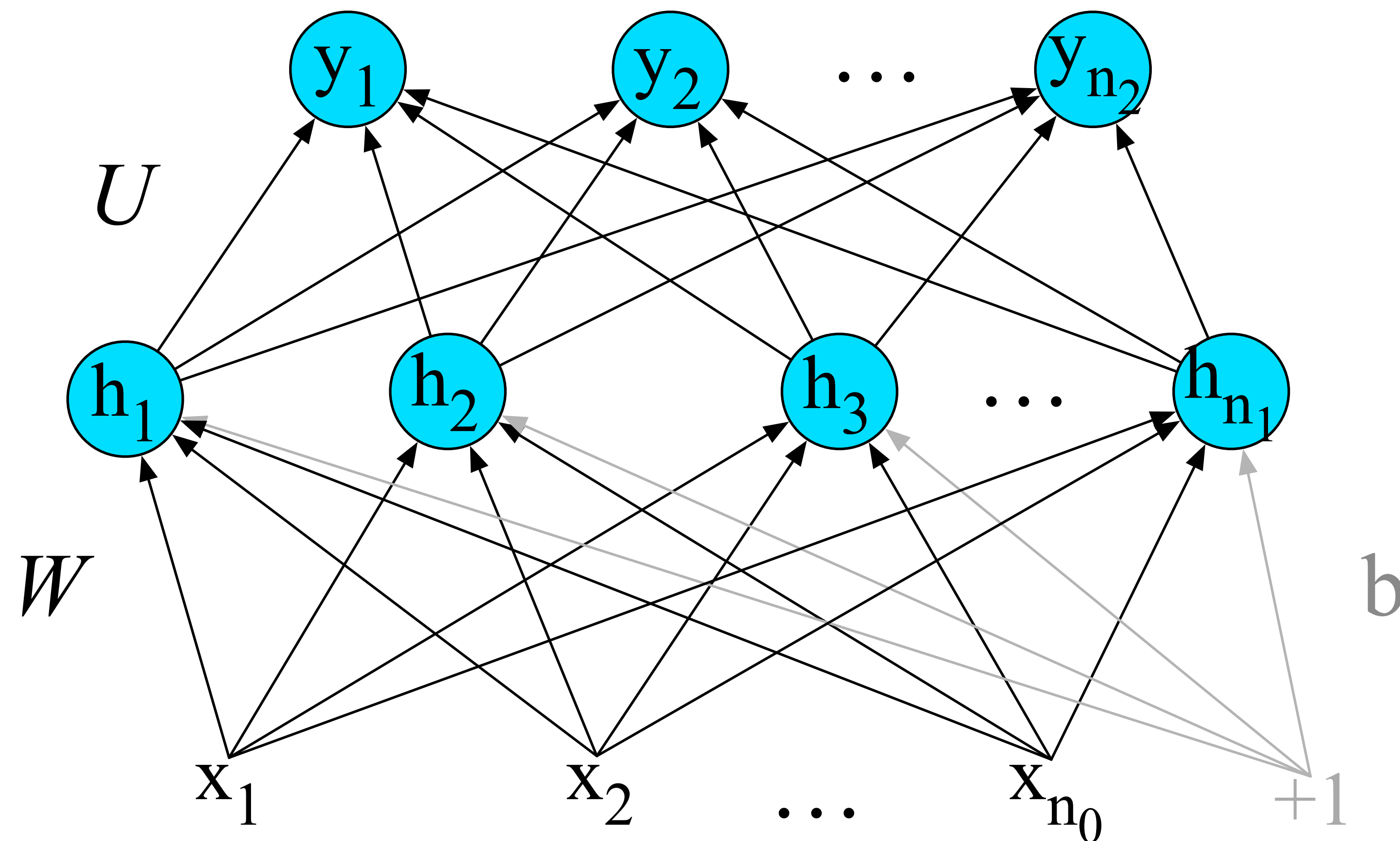
Simple Neural Networks and Neural Language Models

The XOR problem

Feedforward Neural Networks

Feedforward Neural Networks

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



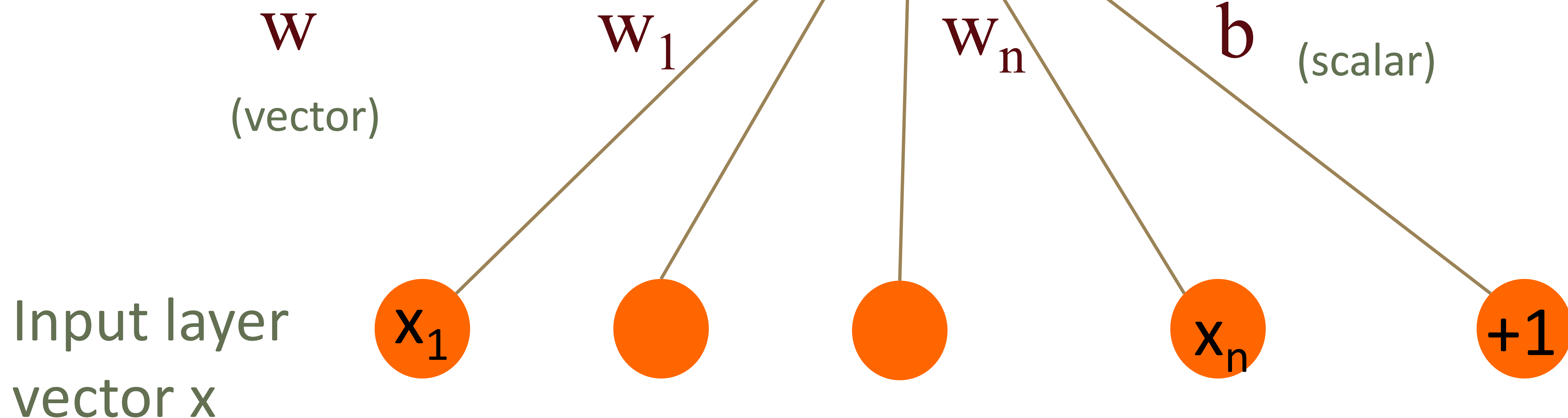
Binary Logistic Regression as a 1-layer Network

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

Output layer
(σ node)

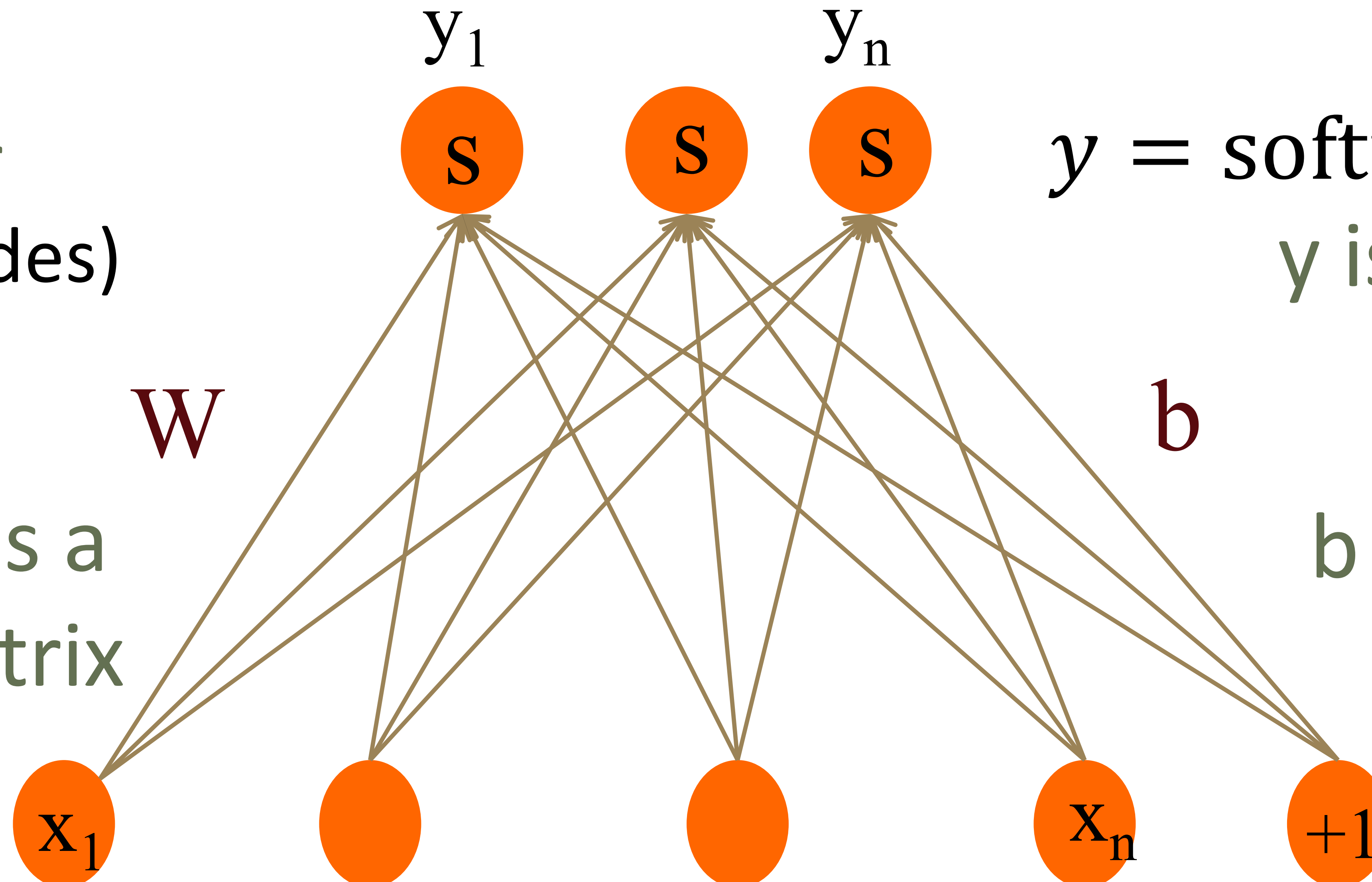
$y = \sigma(w \cdot x + b)$

(y is a scalar)



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:

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

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

Example:

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

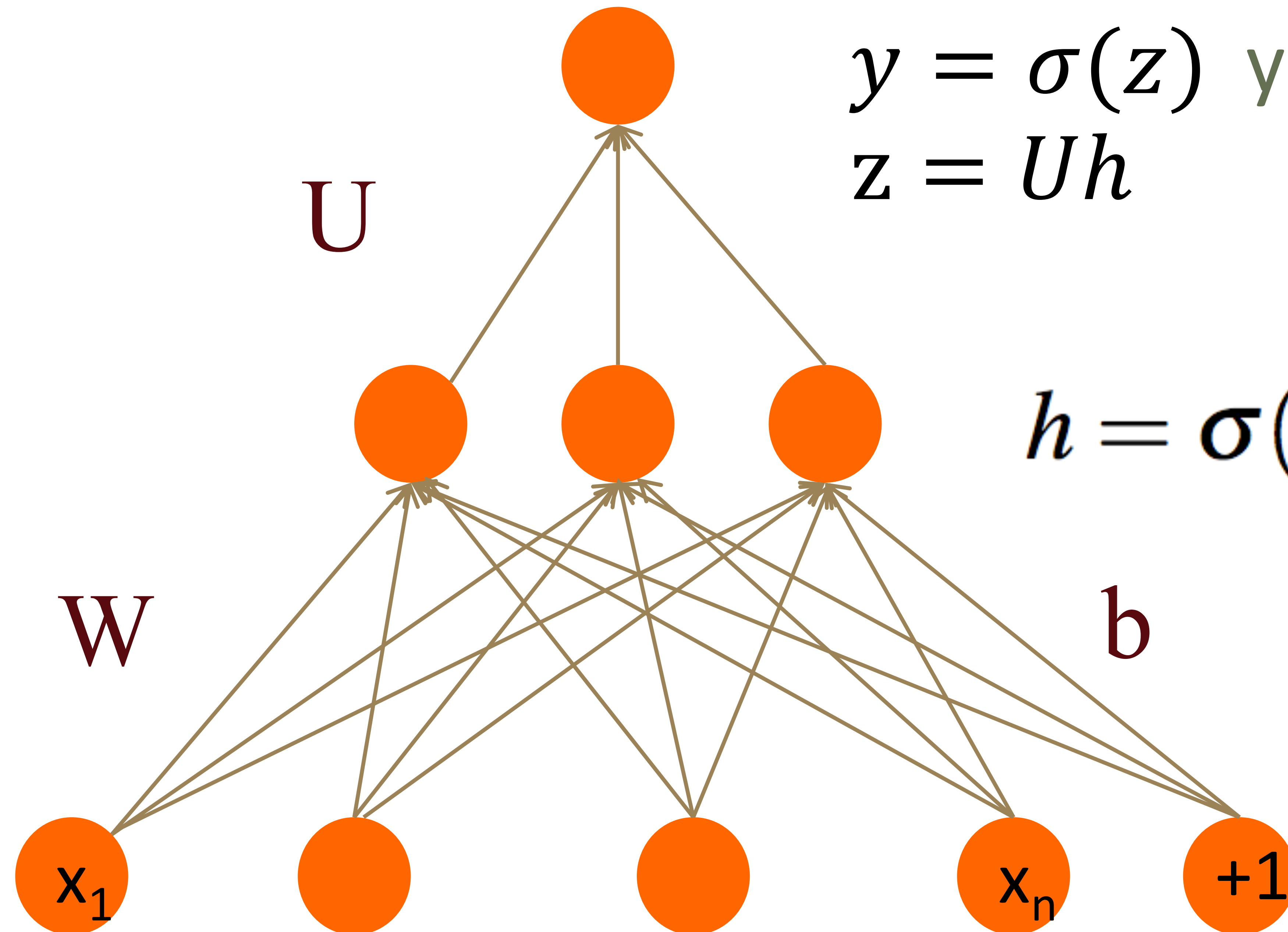
$$\text{softmax}(z) = [0.055, 0.090, 0.006, 0.099, 0.74, 0.010]$$

Two-Layer Network with scalar output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)



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

$z = Uh$

$h = \sigma(Wx + b)$

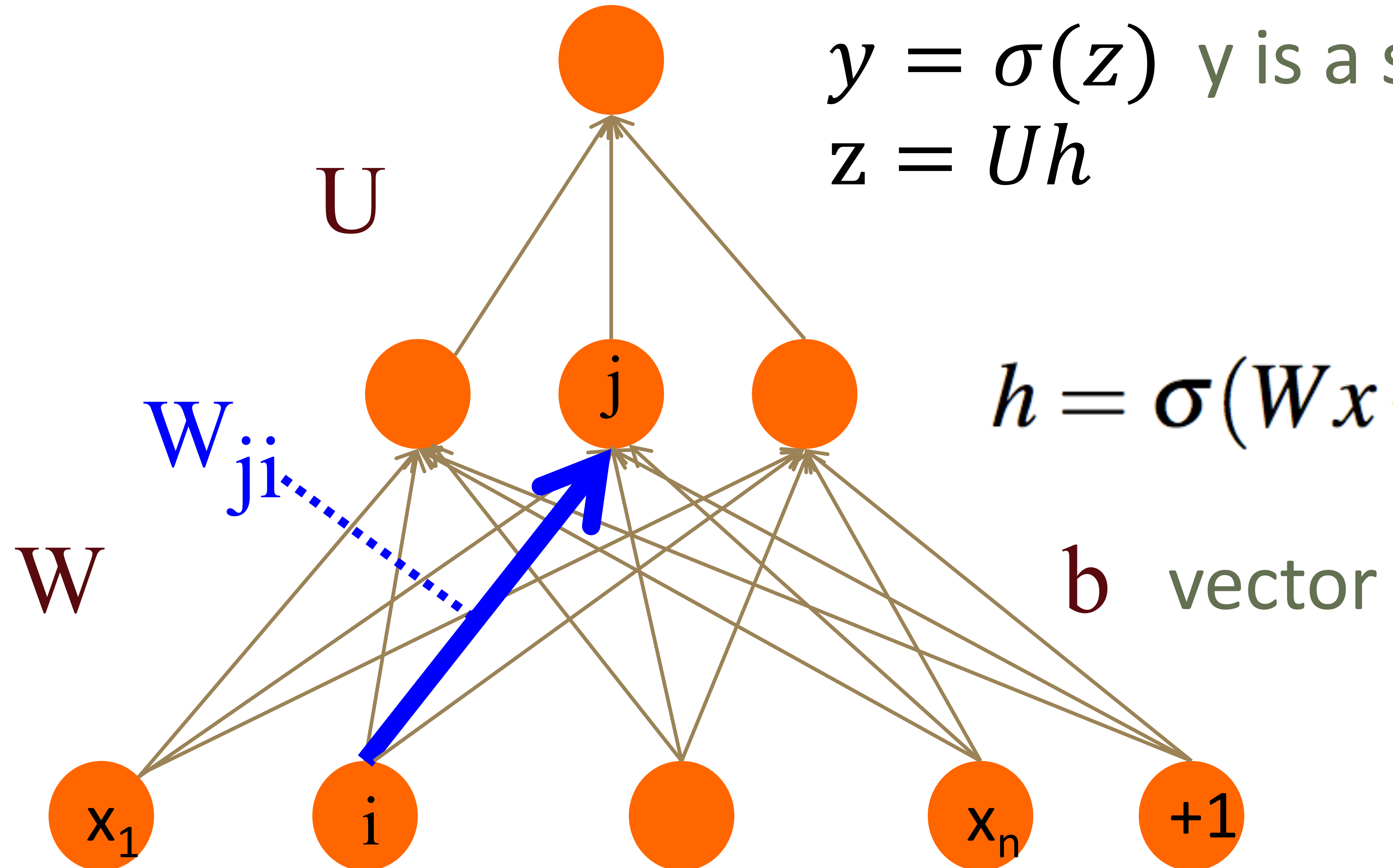
Could be ReLU
Or tanh

Two-Layer Network with scalar output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)



Two-Layer Network with scalar output

Output layer
(σ node)

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

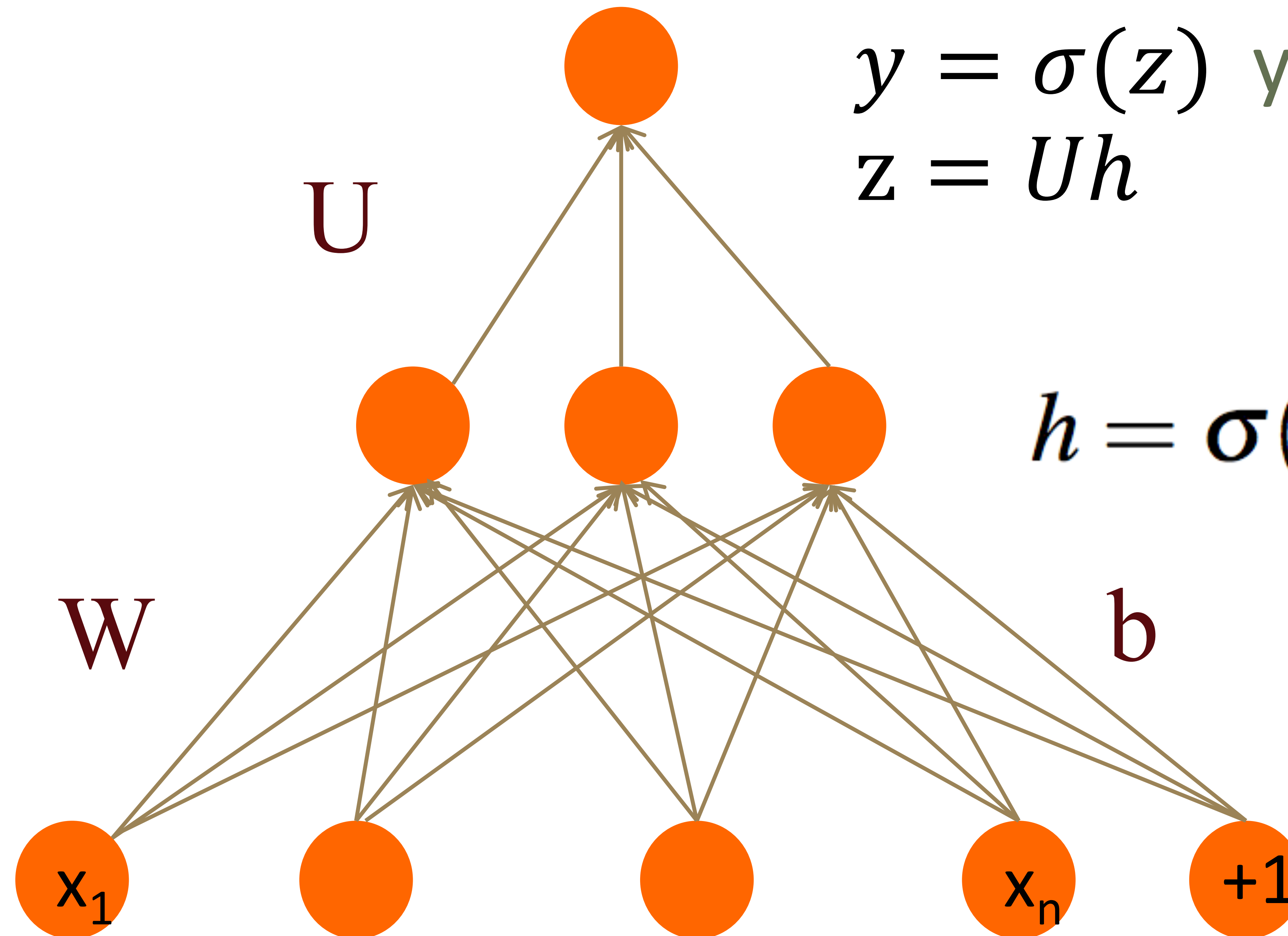
$z = Uh$

hidden units
(σ node)

$h = \boldsymbol{\sigma}(Wx + b)$

Could be ReLU
Or tanh

Input layer
(vector)

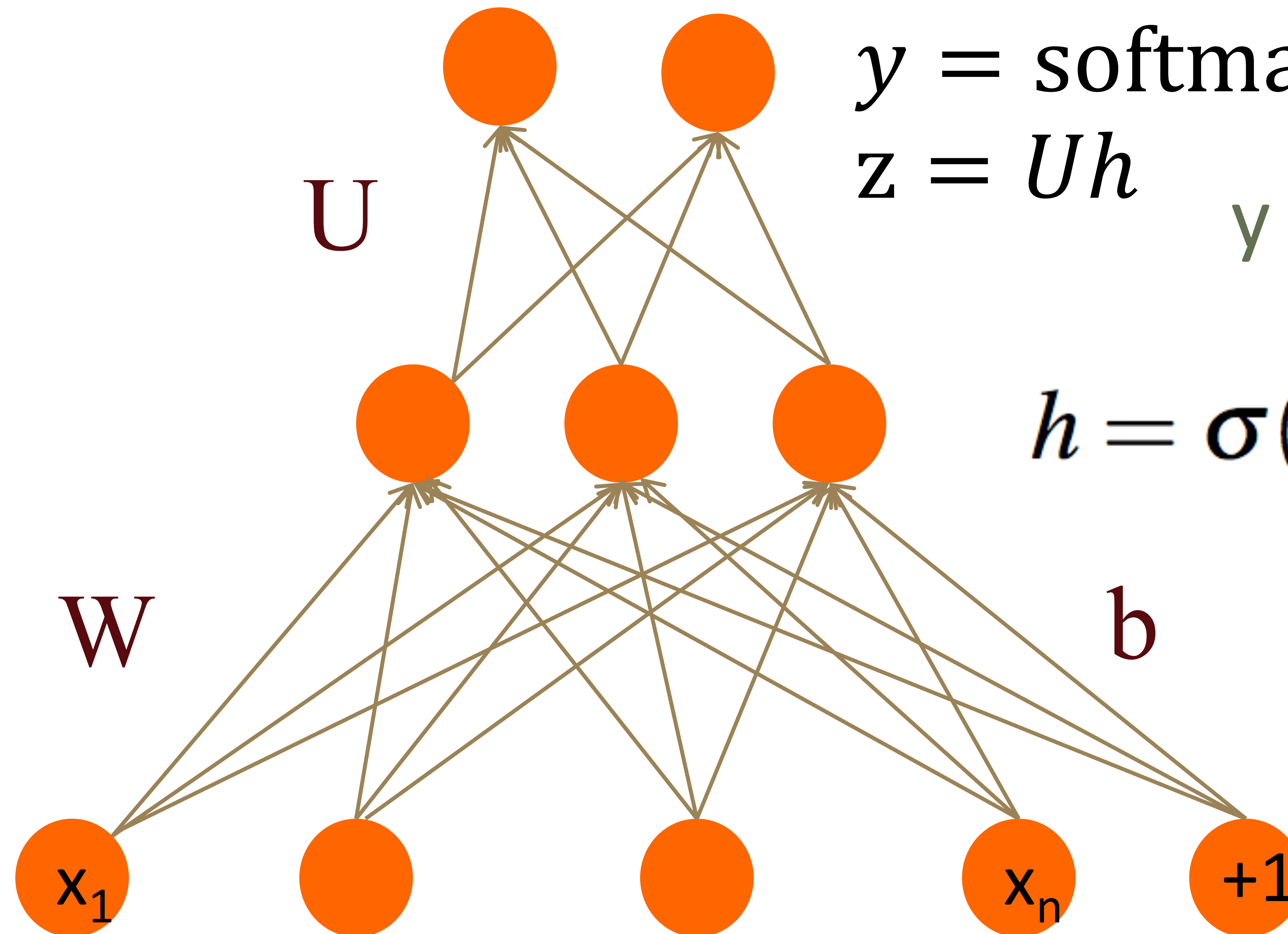


Two-Layer Network with softmax output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)



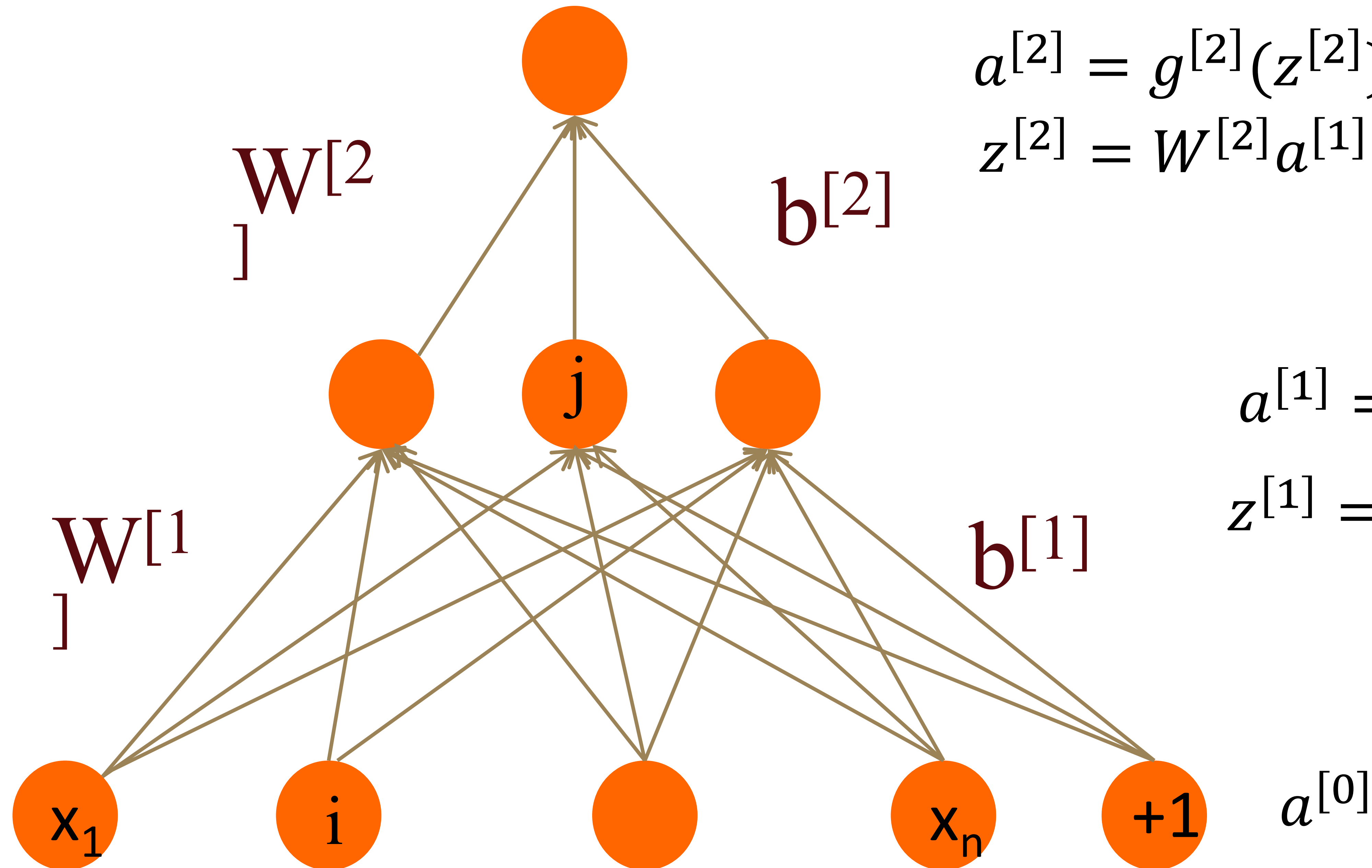
$$y = \text{softmax}(z)$$
$$z = Uh$$

y is a vector

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

Could be ReLU
Or tanh

Multi-layer Notation



$y = a^{[2]}$

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

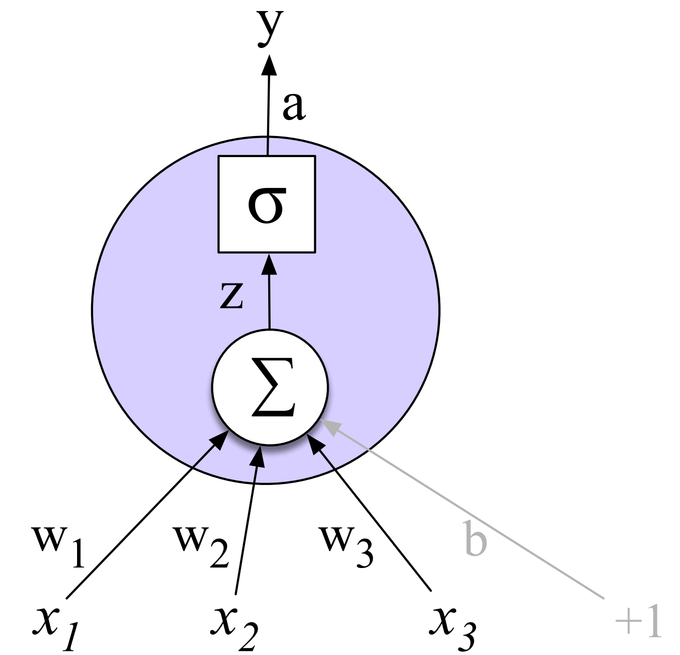
$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$

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

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

$a^{[0]}$

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

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

$$\hat{y} = a^{[n]}$$

Replacing the bias unit

Let's switch to a notation without the bias unit

Just a notational change

1. Add a dummy node a_0=1 to each layer
2. Its weight w_0 will be the bias
3. So input layer $a^{[0]}_0$=1,
   - And $a^{[1]}_0$=1 , $a^{[2]}_0$=1,…

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:

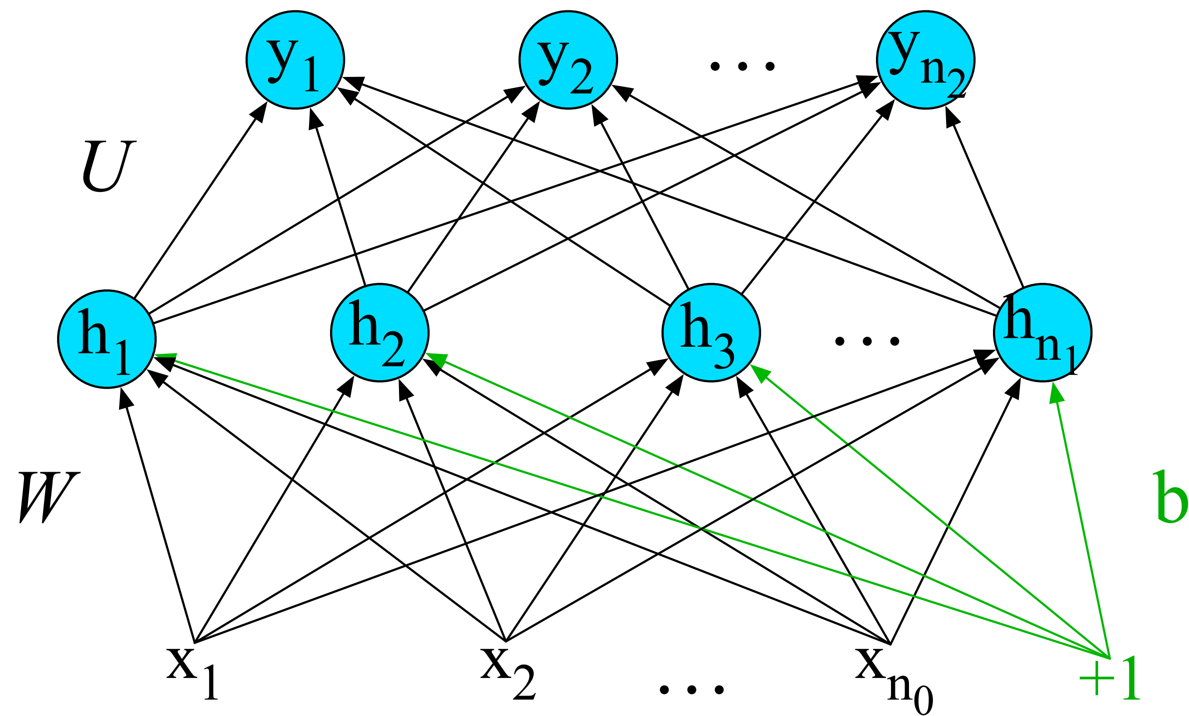
$$x = x_0, x_1, x_2, \ldots, x_{n0}$$

$$h = \sigma(Wx)$$

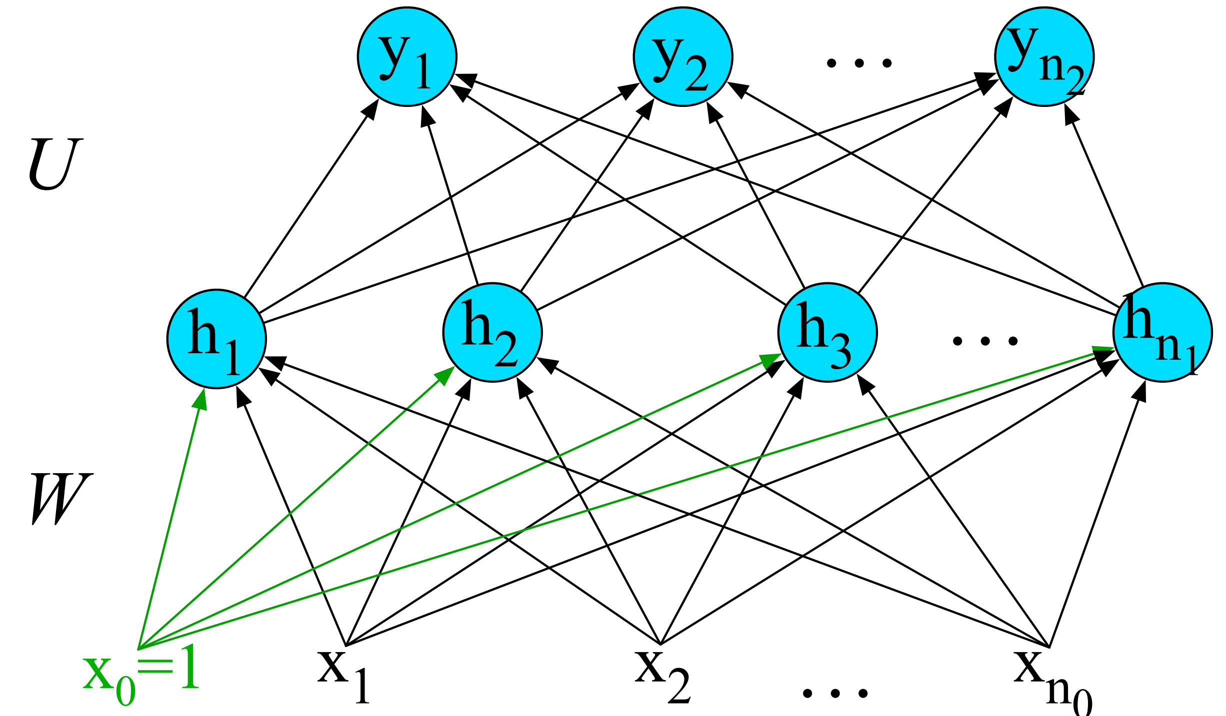
$$\sigma\left(\sum_{i=0}^{n_0} W_{ji} x_i\right)$$

Replacing the bias unit

Instead of:



We'll do this:



Simple Neural Networks and Neural Language Models

Feedforward Neural Networks

Simple Neural Networks and Neural Language Models

Feedforward networks for simple classification

Use cases for feedforward networks

Let's consider 2 (simplified) sample tasks:

1. Text classification
2. Language modeling

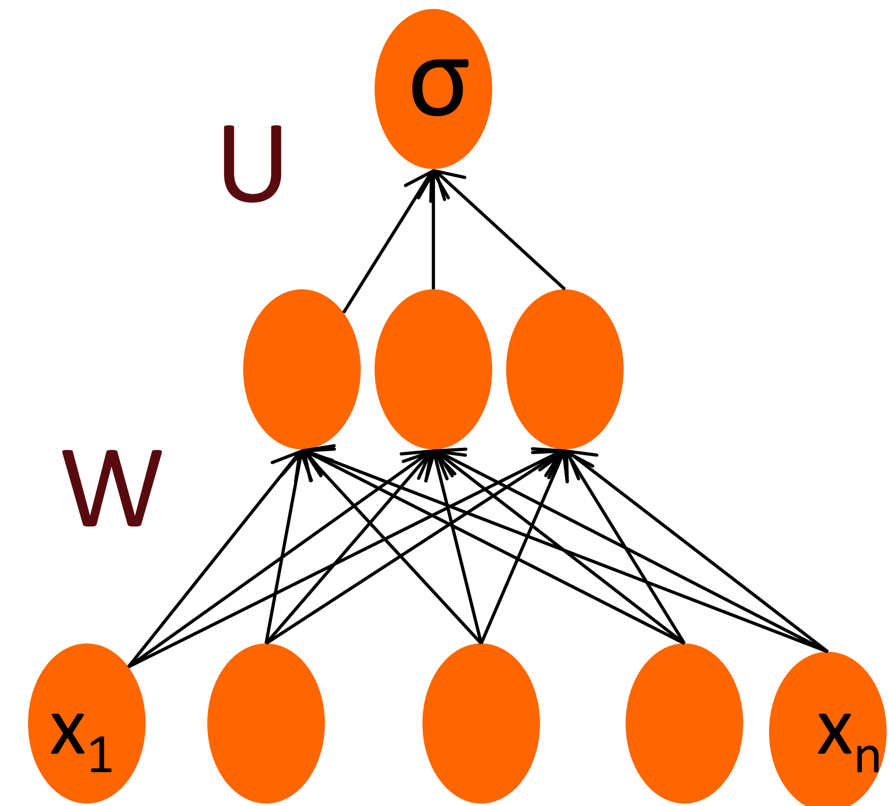
State of the art systems use more powerful neural classifiers like BERT/MLM classifiers, but simple models will introduce some important ideas

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

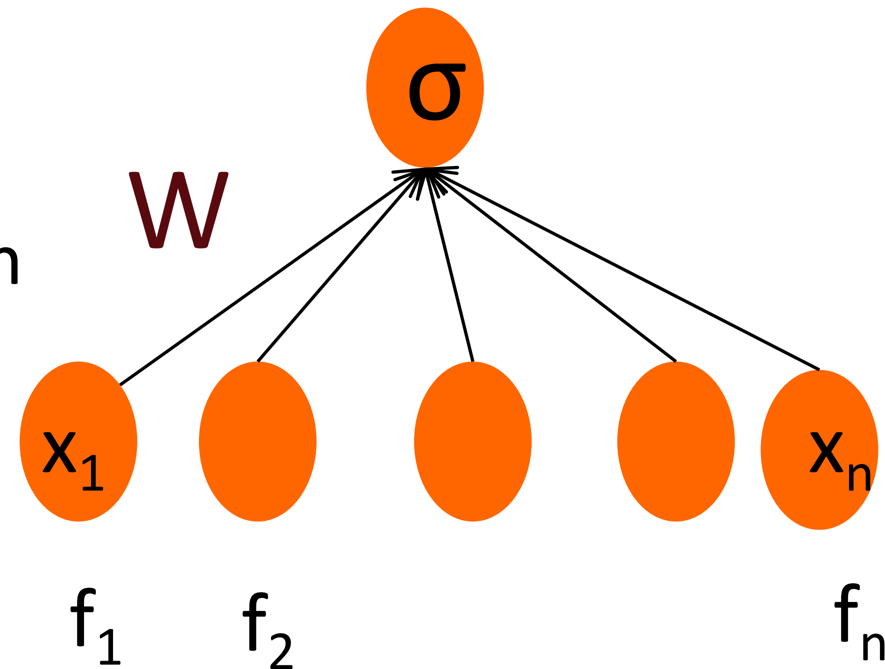


Sentiment Features

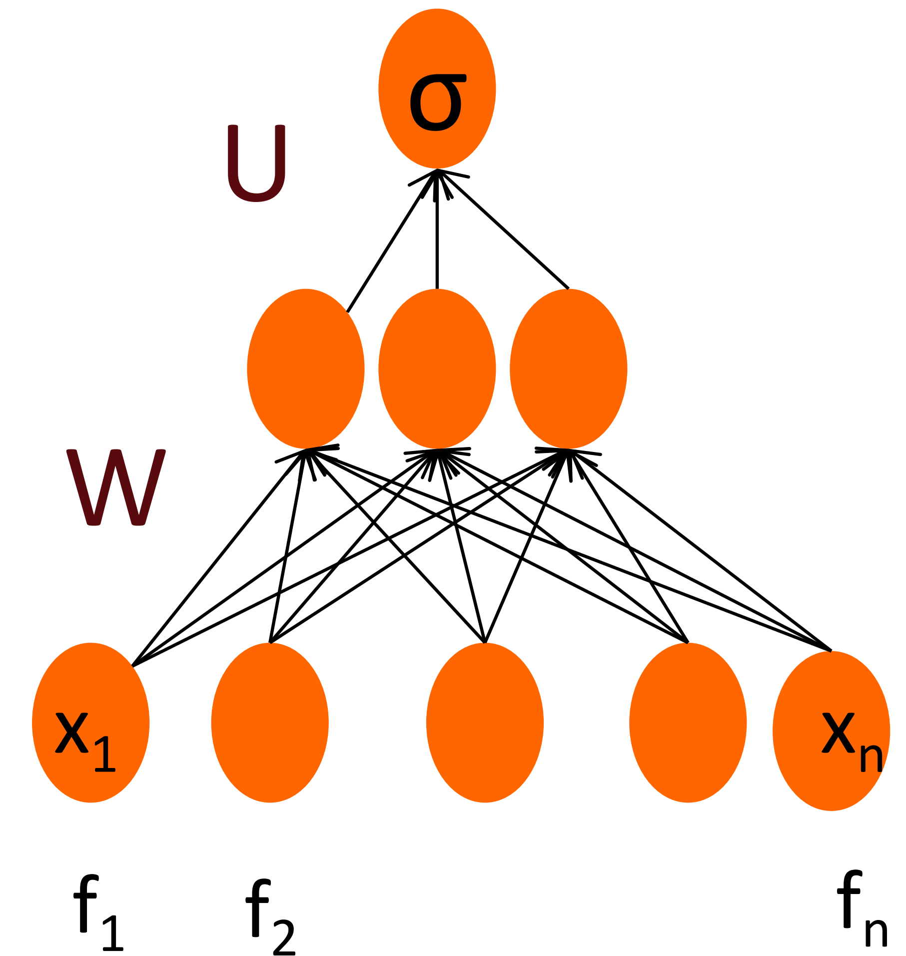
Var	Definition
x_1	count(positive lexicon) $\in$ doc)
x_2	count(negative lexicon) $\in$ doc)
x_3	$\begin{cases} 1 & \text{if "no"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$
x_4	count(1st and 2nd pronouns $\in$ doc)
x_5	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise}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

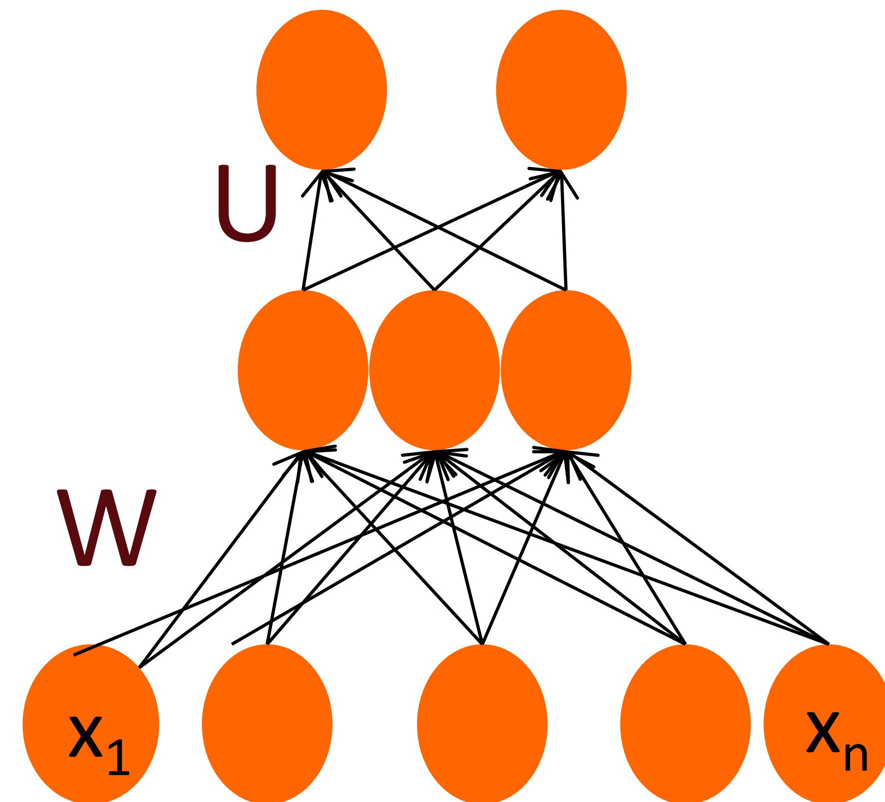
- allows the network to use non-linear interactions between features
- which may (or may not) improve performance.

Reminder: Multiclass Outputs

What if you have more than two output classes?

- Add more output units (one for each class)
- And use a "softmax layer"

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



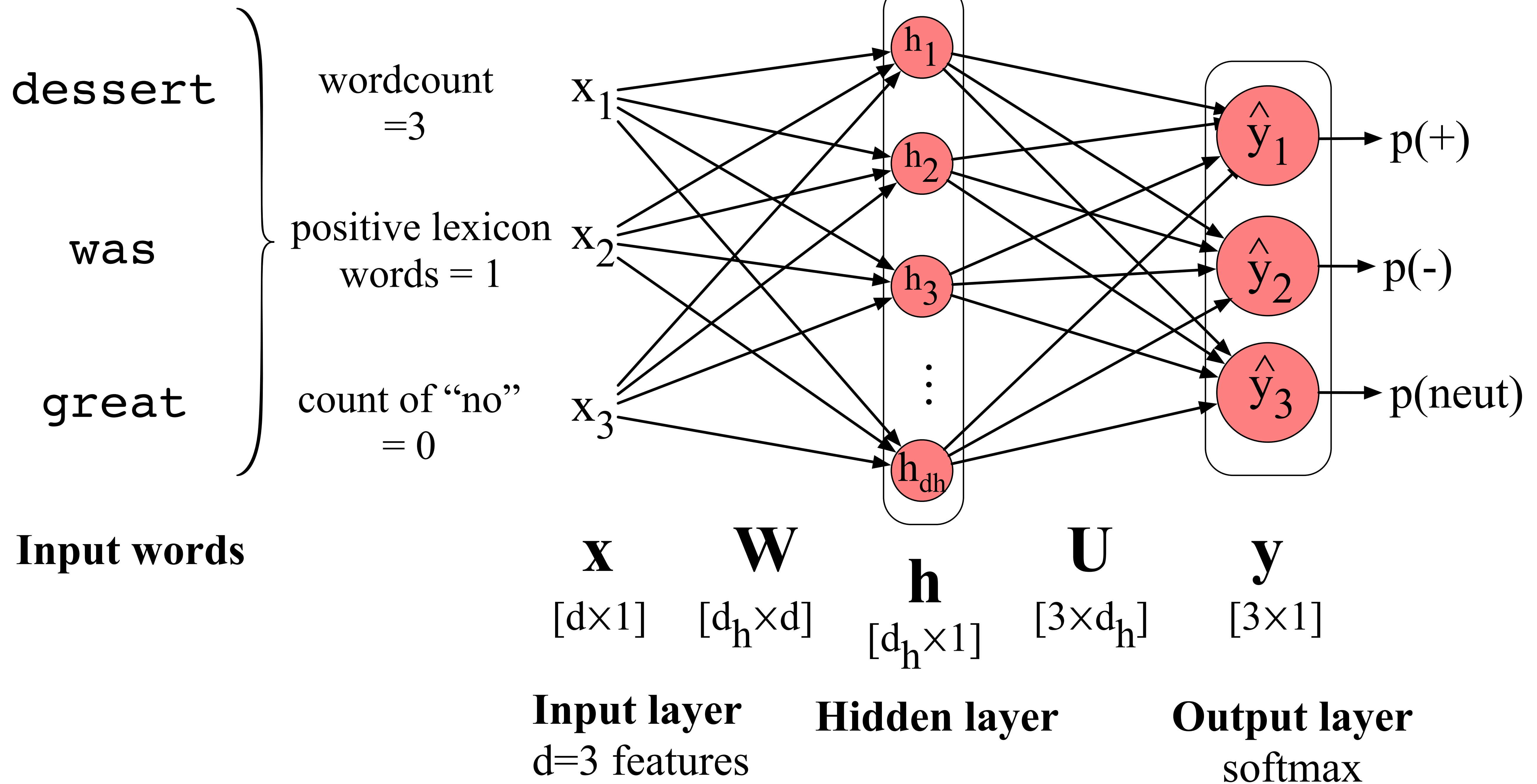
The output layer

$\hat{y}$ could have two nodes (one each for positive and negative), or 3 nodes (positive, negative, neutral).

- $\hat{y}_1$ estimated probability of positive sentiment
- $\hat{y}_2$ probability of negative
- $\hat{y}_3$ probability of neutral

Equations for NN classification with hand features

$$
\begin{aligned}
\mathbf{x} &= [\mathbf{x}_1, \mathbf{x}_2, \ldots \mathbf{x}_d] \quad \text{(each } \mathbf{x}_i \text{ is a hand-designed feature)} \\
\mathbf{h} &= \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \\
\mathbf{z} &= \mathbf{U}\mathbf{h} \\
\hat{\mathbf{y}} &= \text{softmax}(\mathbf{z})
\end{aligned}
$$


dessert
was
great
wordcount =3
positive lexicon words = 1
count of “no” = 0
x1
x2
x3
h1
h2
h3
hdh
ŷ1
ŷ2
ŷ3
p(+)
p(-)
p(neut)
Input words
x
[d×1]
W
[dh×d]
h
[dh×1]
U
[3×dh]
y
[3×1]
Input layer
d=3 features
Hidden layer
Output layer
softmax

Vectoring for parallelizing inference

We would like to efficiently classify the whole test set of m observations.

So we vectorize: pack all the input features into **X**

- Each row $\mathbf{x}^{(i)}$ of **X** is a row vector with all the features for example $x^{(i)}$
- Feature dimensionality is *d*, **X** is [*m* x *d*]

Slight changes to equations

Each input is now a row vector

- **X** is of shape [*m* x *d*]
- **W** is of shape [d_h x *d*]
- We'll need to do some reordering and transposing
- Bias vector **b** that used to be [1 x d_h] is now [*m* x d_h]

$$\mathbf{H} = \sigma(\mathbf{X}\mathbf{W}^\intercal + \mathbf{b})$$
$$\mathbf{Z} = \mathbf{H}\mathbf{U}^\intercal$$
$$\hat{\mathbf{Y}} = \text{softmax}(\mathbf{Z})$$

Feedforward networks for simple classification

Simple Neural Networks and Neural Language Models

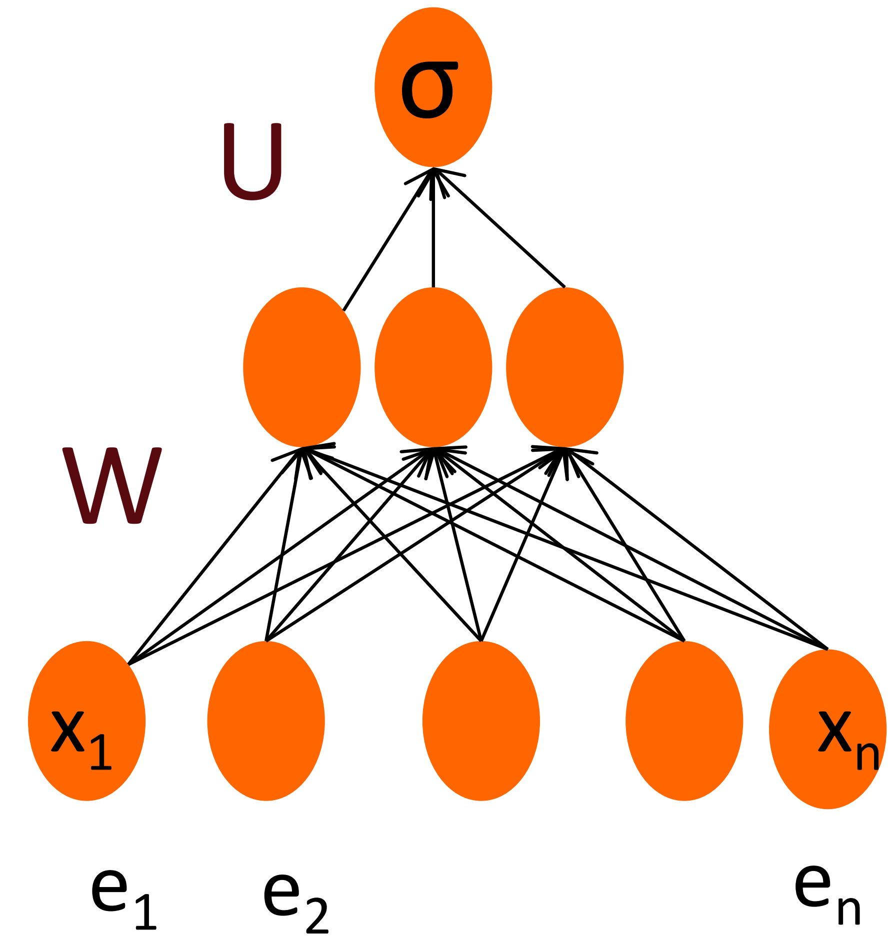
Embeddings as input to feedforward classifiers

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!



Embedding matrix **E**

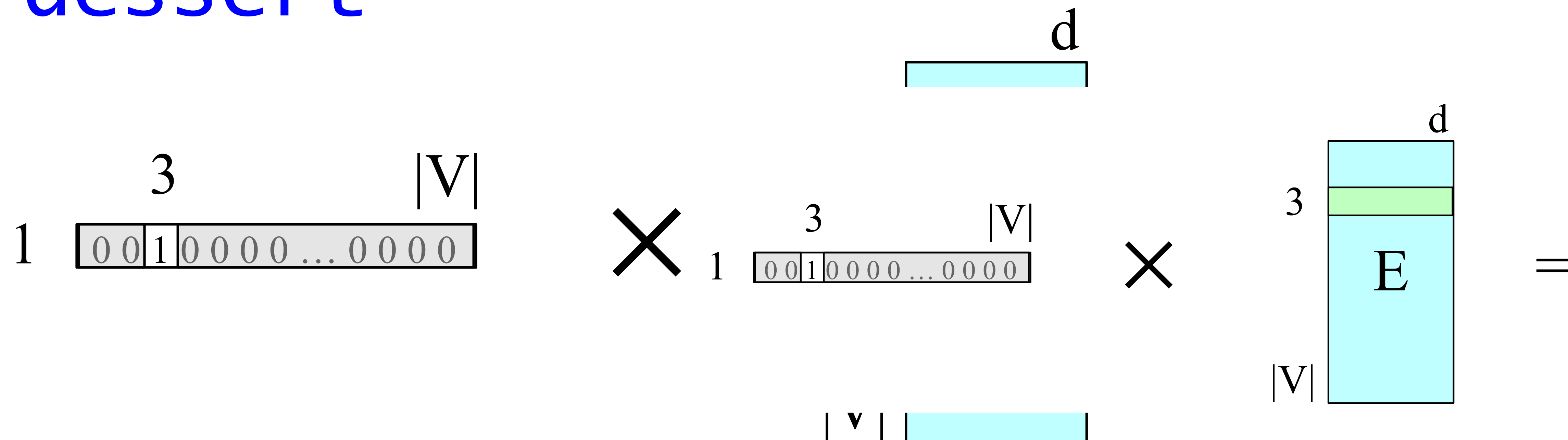
- An embedding is a vector of dimension [1 x *d]* that represents the input token.
- An embedding matrix **E** is a dictionary, one row per token of vocab *V*
- **E** has shape $[|V| \times d]$
- Embedding matrices are central to NLP; they represent input text in LLMs and all NLP tools

Text classification from embeddings

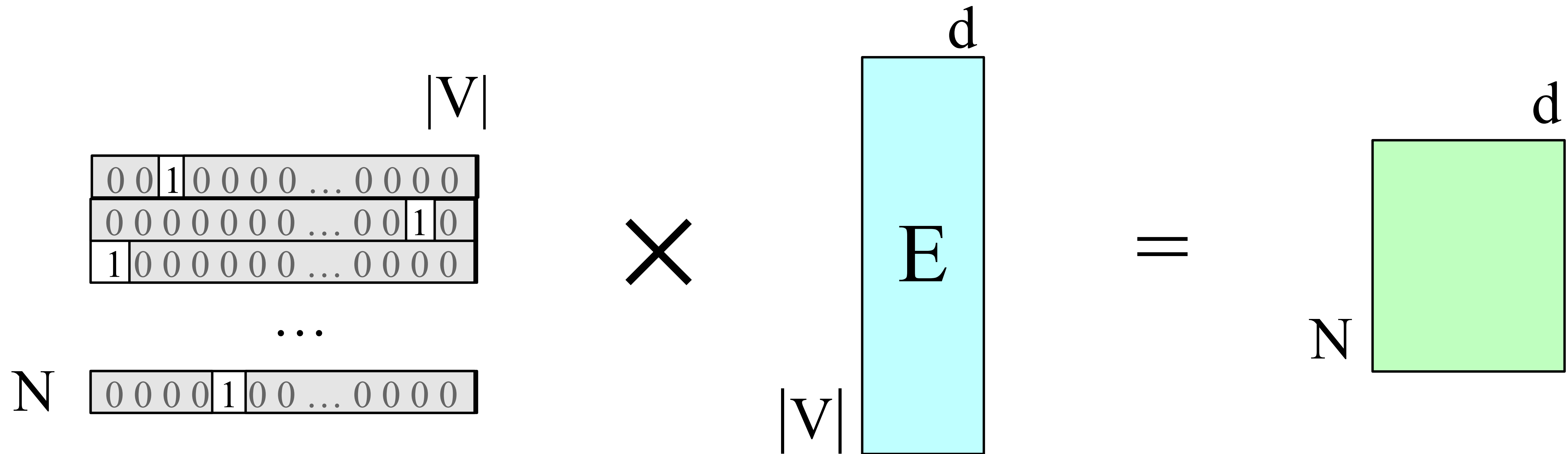
- Given tokenized input: dessert was great
- Select the embedding vectors from **E:**
  1. Convert BPE tokens into vocabulary indices
     - w = [3, 9824, 226]
  2. Use indexing to select the corresponding rows from **E**
     - row 3, row 4000, row 10532

Another way to think of indexing from **E**

- Treat each input word as one-hot vector
  - If `dessert` is index 3: `[0 0 1 0 0 0 0 ... 0 0 0 0]`
    `1 2 3 4 5 6 7 ... ... |V|`
- Multiply it by **E** to select out the embedding for `dessert`



Collecting embeddings from E for *N* words



Given window of N tokens, represented by *N* [1 × *d*] embeddings

Need to return a single class (e.g., pos or neg)

Text comes in different lengths

Given window of N tokens, represented by N $[1 \times d]$ embeddings, return a single class (e.g., pos or neg)

1. **Concatenate** all the inputs into one long vector of shape $[1 \times dN]$, i.e. input is length N of longest review
    - If shorter then pad with zero embeddings
    - Truncate if you get longer reviews at test time
2. **Pool** the inputs into a single short $[1 \times d]$ vector. A single "sentence embedding" (the same dimensionality as a word) to represent all the words. Less info, but very efficient and fast.

Pooling

Intuition: exact position not so important for sentiment.

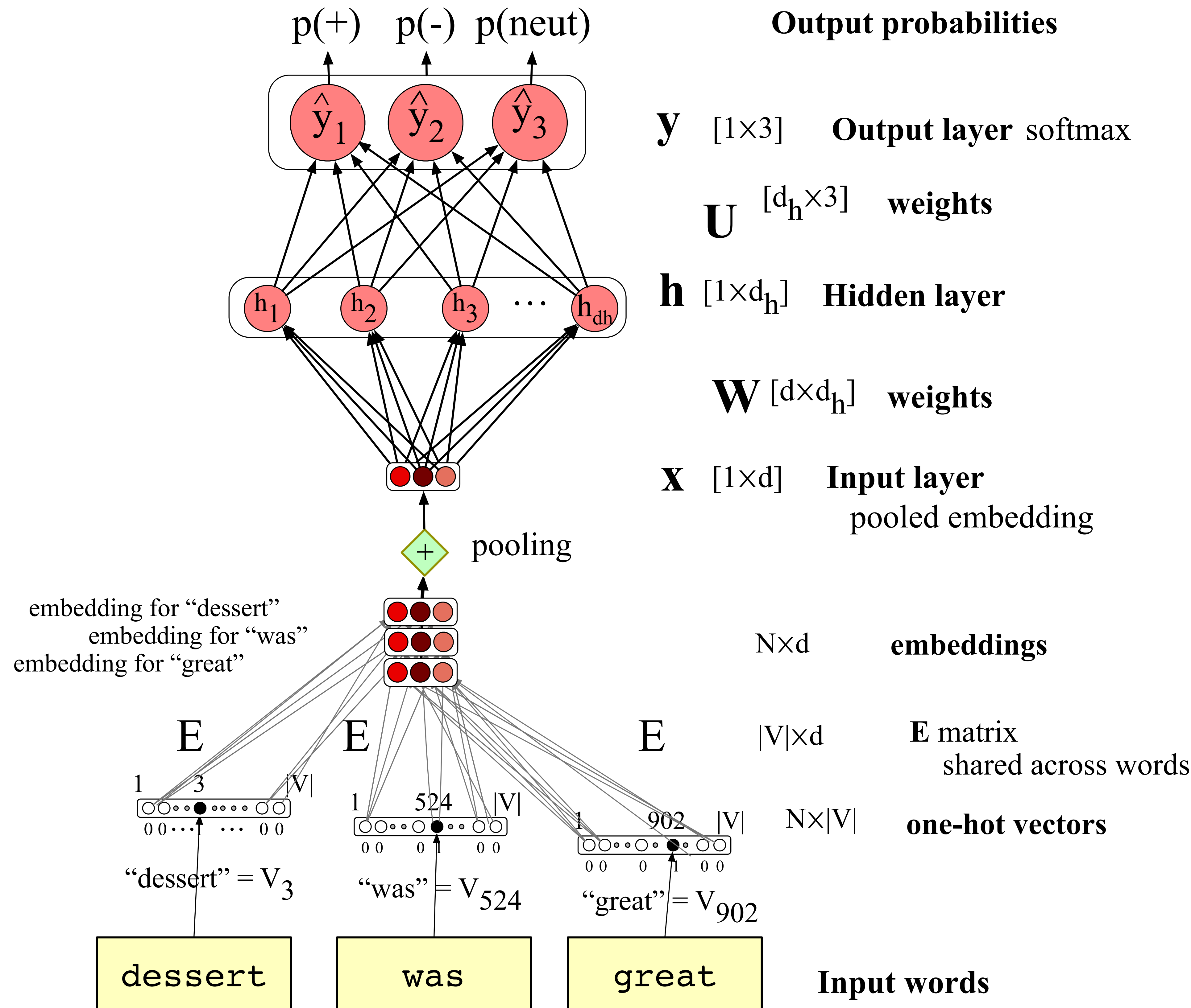
We'll just do some sort of averaging of all the vectors.

Mean pooling:

$$\mathbf{x}_{mean} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{e}(w_i)$$

$$\begin{aligned}
\mathbf{x} &= \text{mean}(\mathbf{e}(w_1), \mathbf{e}(w_2), \ldots, \mathbf{e}(w_n)) \\
\mathbf{h} &= \sigma(\mathbf{x}\mathbf{W} + \mathbf{b}) \\
\mathbf{z} &= \mathbf{h}\mathbf{U} \\
\hat{\mathbf{y}} &= \text{softmax}(\mathbf{z})
\end{aligned}$$

Pooling



Simple Neural Networks and Neural Language Models

Embeddings as input to feedforward classifiers

Simple Neural Networks and Neural Language Models

Neural language modeling with feedforward networks

Neural Language Models (LMs)

Language Modeling: Calculating the probability of the next word in a sequence given some history.

- We've seen N-gram based LMs
- But neural network LMs far outperform n-gram language models

State-of-the-art neural LMs are based on more powerful neural network technology like Transformers

But **simple feedforward LMs** introduce many of the important concepts

Simple feedforward Neural Language Models

Task: predict next word w_t

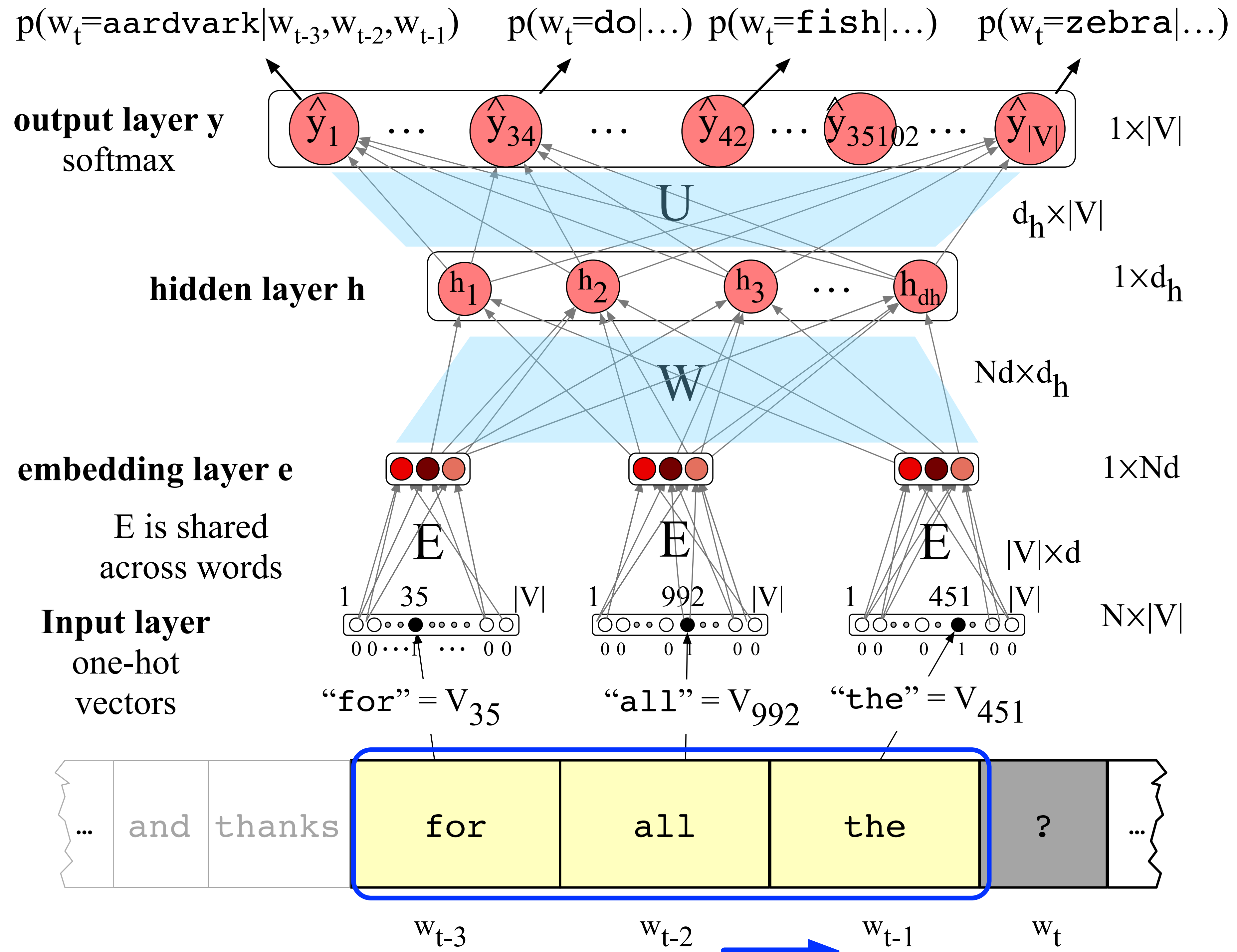
given prior words w_{t-1}, w_{t-2}, w_{t-3}, …

Problem: Now we're dealing with sequences of arbitrary length.

Solution: Sliding windows (of fixed length)

$$P(w_t|w_1,\ldots,w_{t-1}) \approx P(w_t|w_{t-N+1},\ldots,w_{t-1})$$

Inference in a Feedforward Language Model



Feedforward language model equations

$$
\begin{aligned}
\mathbf{e} &= [\mathbf{E}\mathbf{x}_{\mathbf{t-3}}; \mathbf{E}\mathbf{x}_{\mathbf{t-2}}; \mathbf{E}\mathbf{x}_{\mathbf{t-1}}] \\
\mathbf{h} &= \sigma(\mathbf{W}\mathbf{e} + \mathbf{b}) \\
\mathbf{z} &= \mathbf{U}\mathbf{h} \\
\hat{\mathbf{y}} &= \text{softmax}(\mathbf{z})
\end{aligned}
$$

So $\hat{y}_{42}$ is the probability of the next word w_t being V_{42} = `fish`

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"!

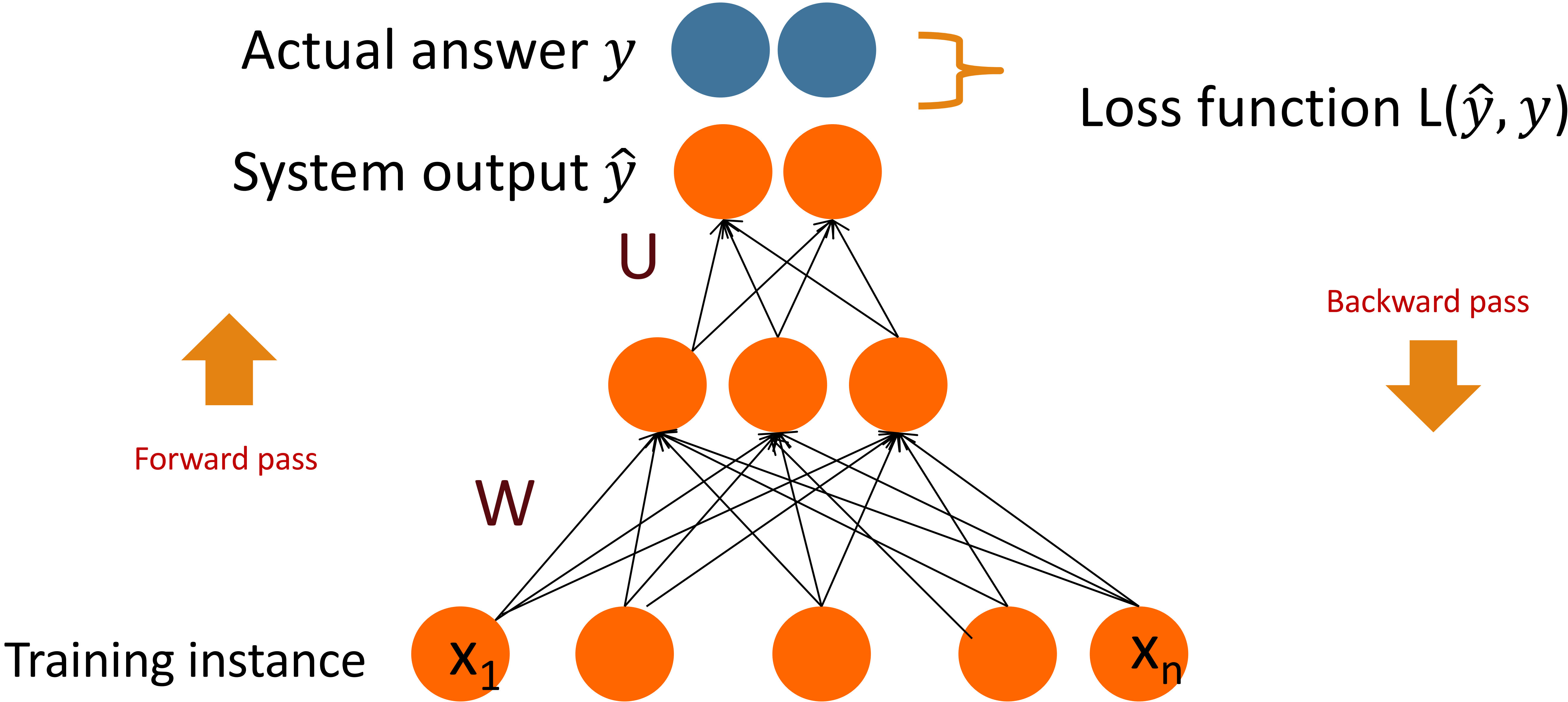
Neural LM can use similarity of "cat" and "dog" embeddings to generalize and predict “fed” after dog

Simple Neural Networks and Neural Language Models

Neural language modeling with feedforward networks

Training Neural Nets: Overview

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

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:

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

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

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

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

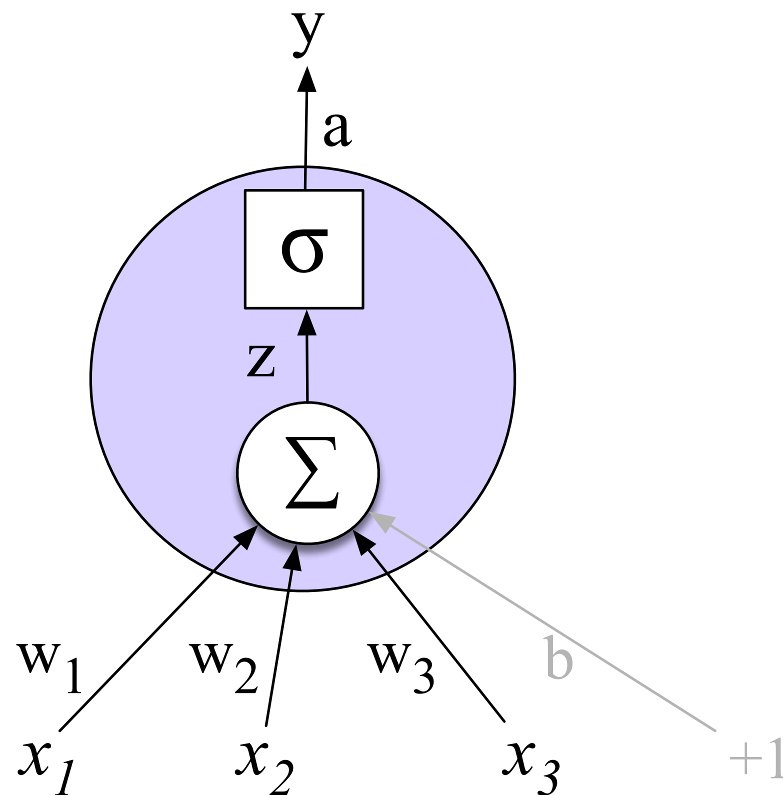
- For logistic regression

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

Where did that derivative come from?

Using the chain rule! $f(x) = u(v(x))$ $\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$

Intuition (see the text for details)



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 lecture:

- Even more use of the chain rule!!
- Computation graphs and backward differentiation!

Simple Neural Networks and Neural Language Models

Training Neural Nets: Overview

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**.

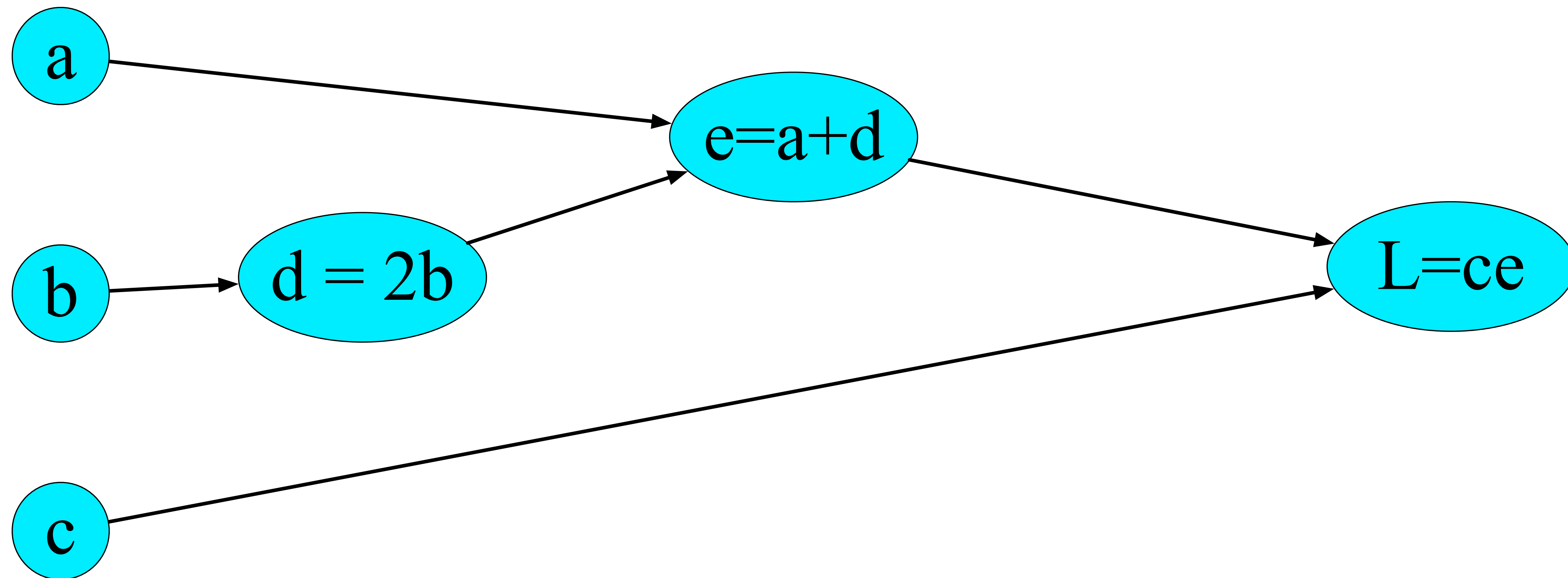
Computation Graphs

A computation graph represents the process of computing a mathematical expression

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

Computations:

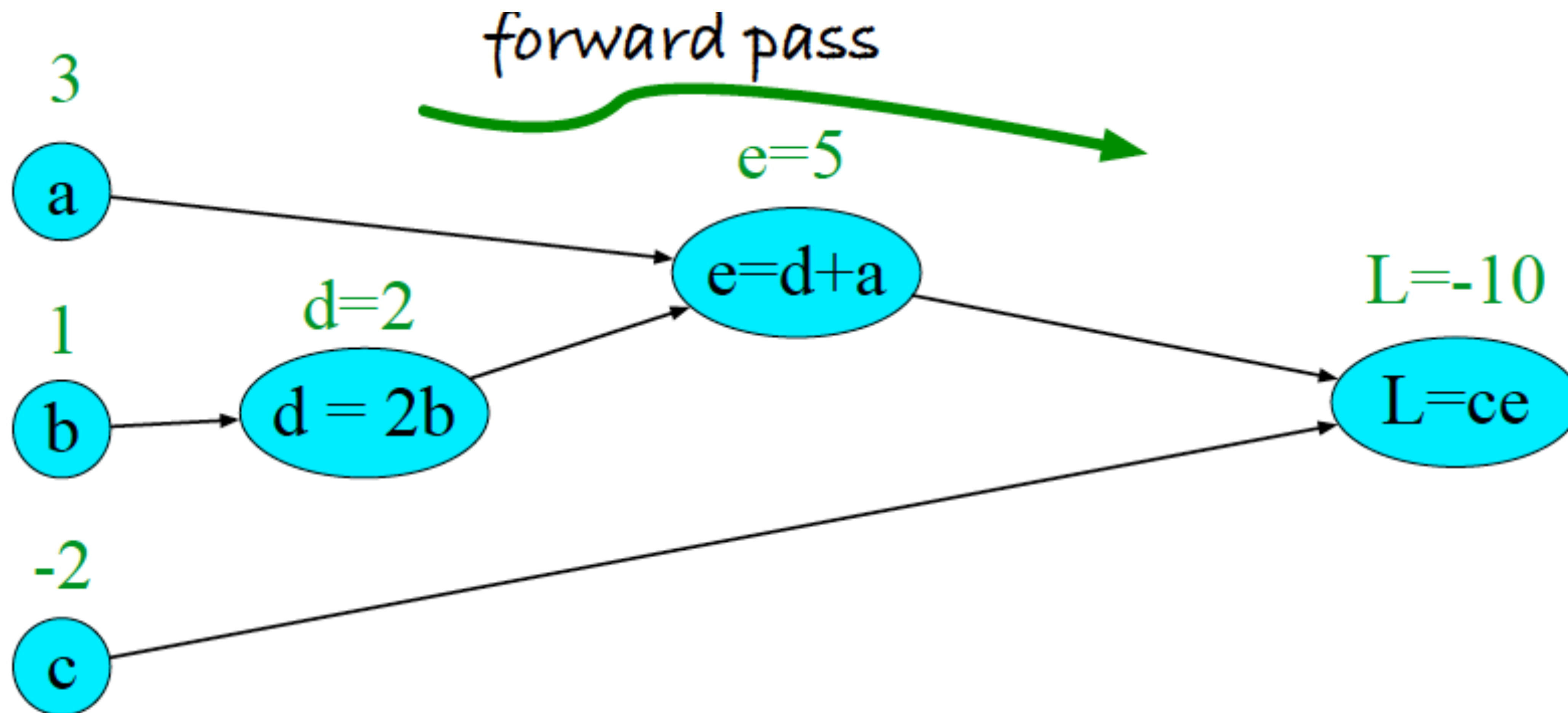
$$
\begin{aligned}
d &= 2 * b \\
e &= a + d \\
L &= c * e
\end{aligned}
$$



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

Computations:

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



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

The derivative $\frac{\partial L}{\partial a}$, tells us how much a small change in *a* affects *L*.

The chain rule

Computing the derivative of a composite function:

$$f(x) = u(v(x)) \qquad \frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$$

$$f(x) = u(v(w(x))) \qquad \frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx}$$

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

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

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

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

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

Example

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

$$d = 2*b$$

$$e = a+d$$

$$L = c*e$$

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

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

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

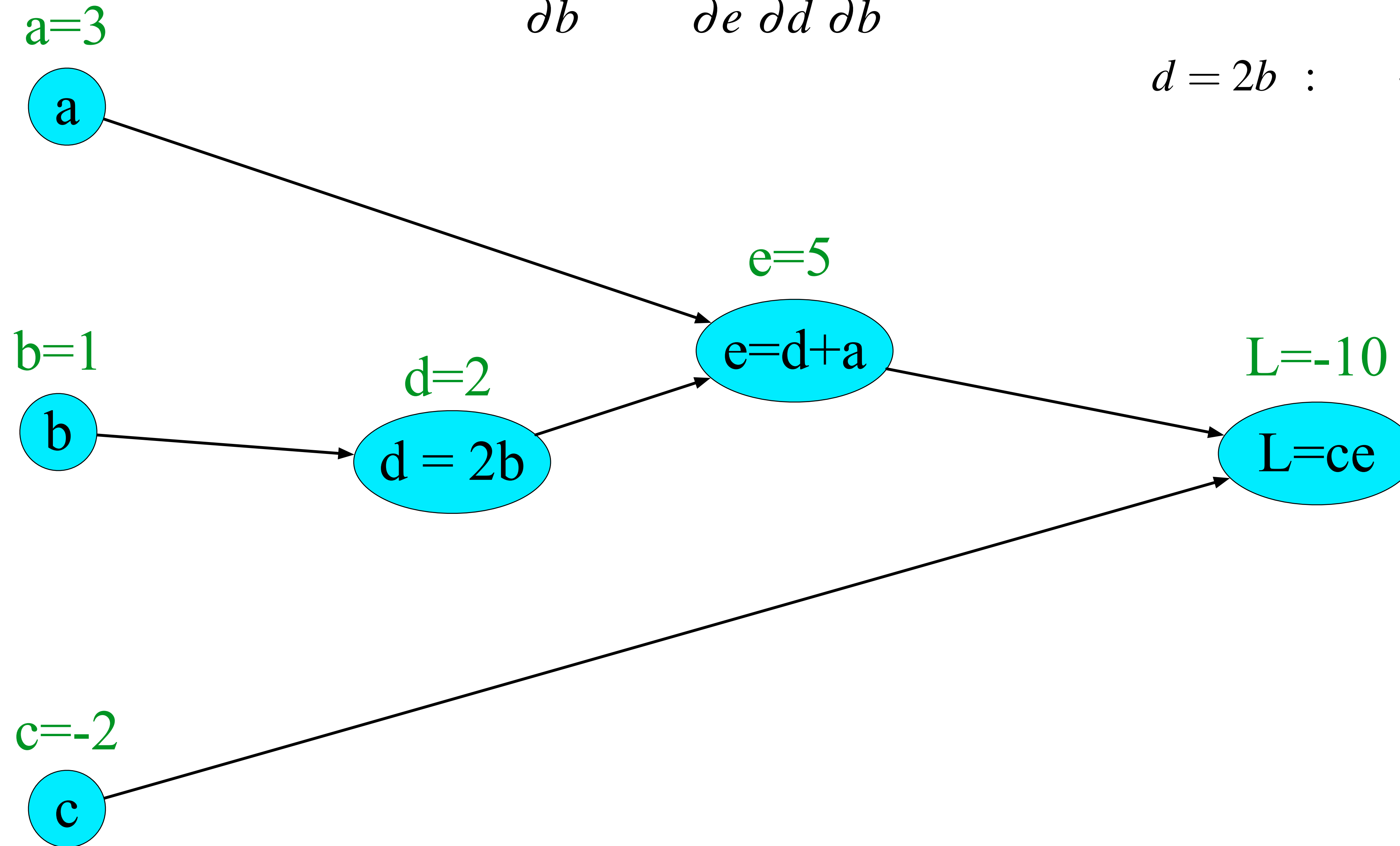
$$e = a+d \ : \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

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

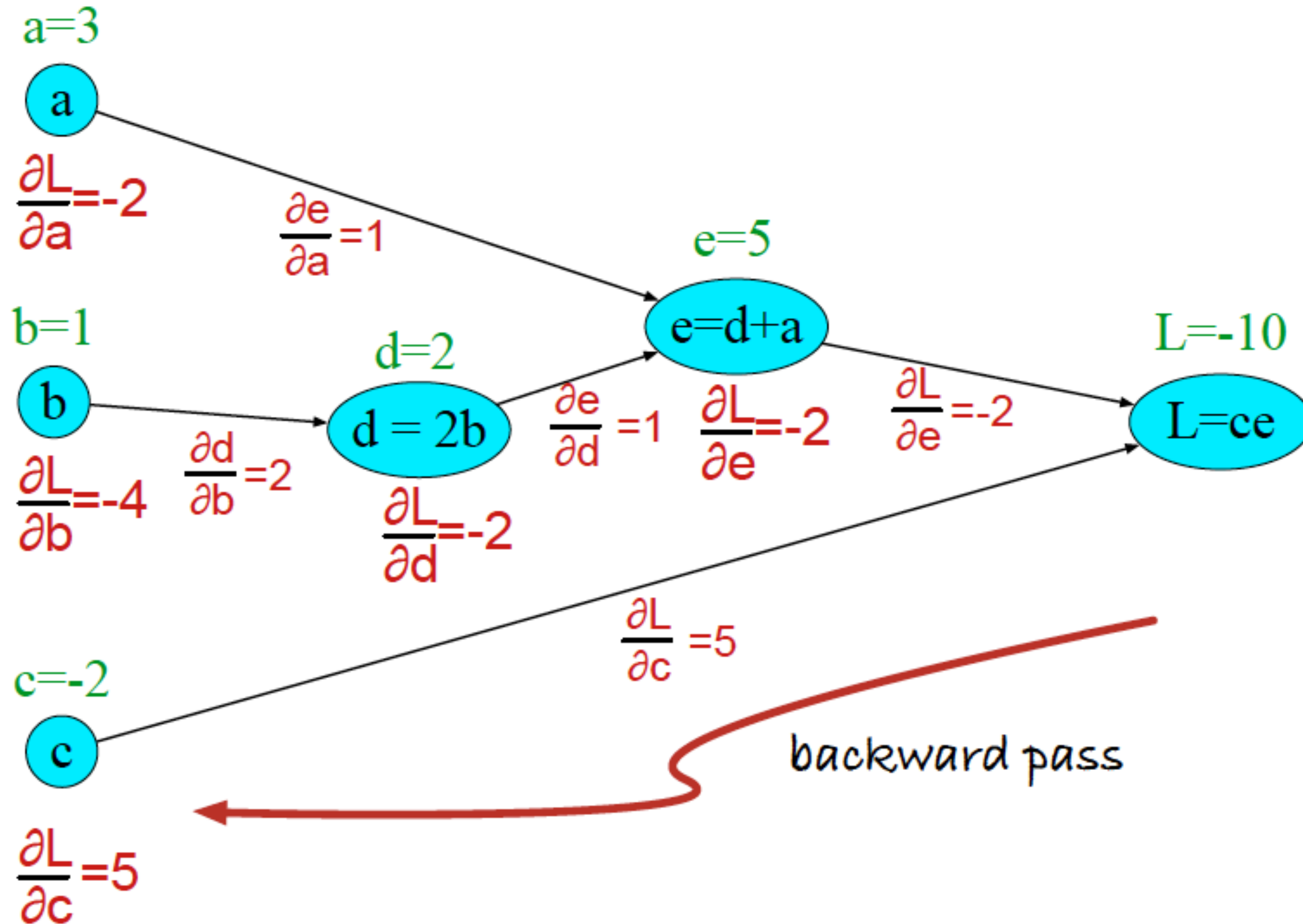
Example

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

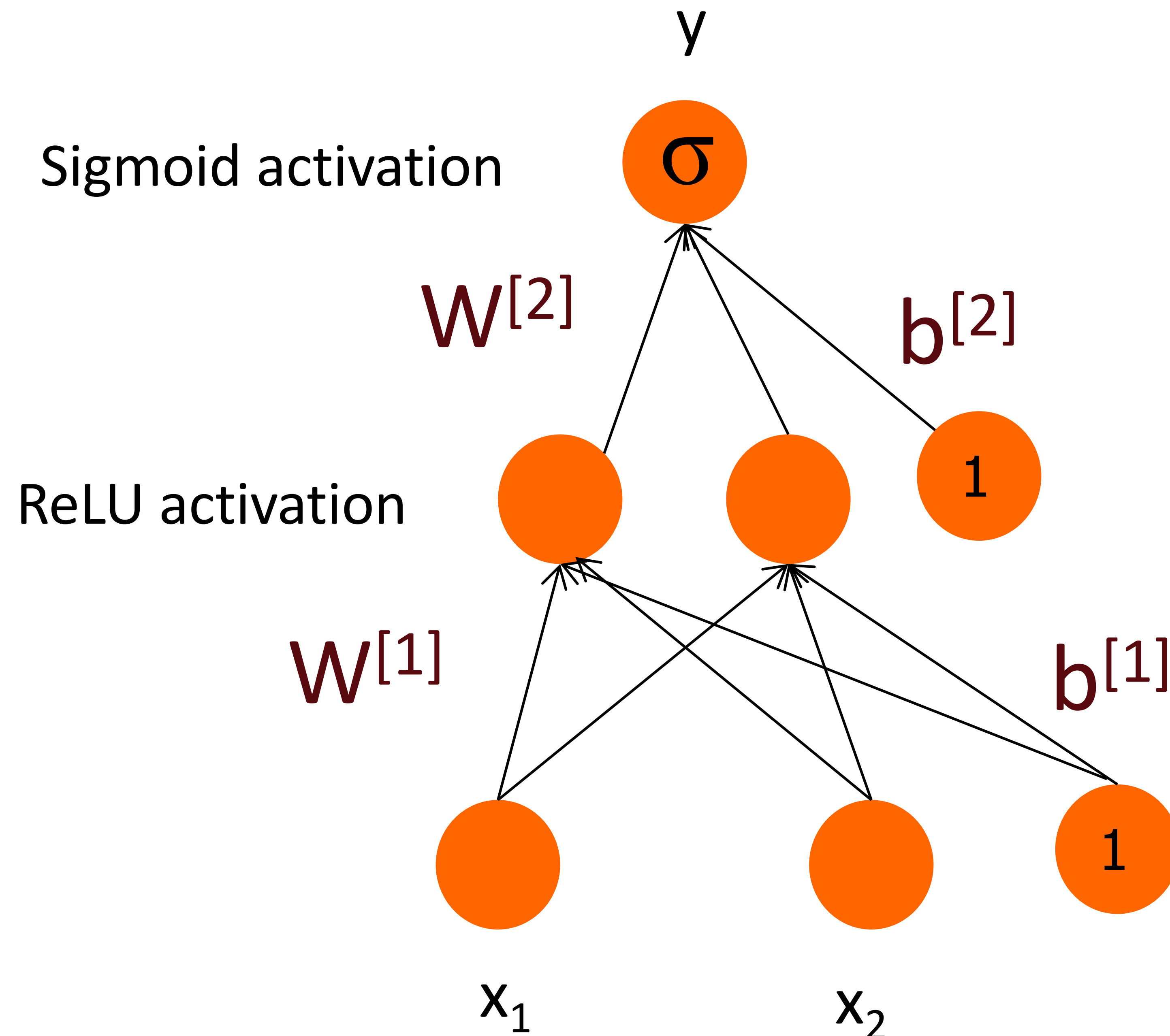
$$L = ce \; : \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$
$$e = a + d \; : \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$
$$d = 2b \; : \quad \frac{\partial d}{\partial b} = 2$$



Example



Backward differentiation on a two layer network



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

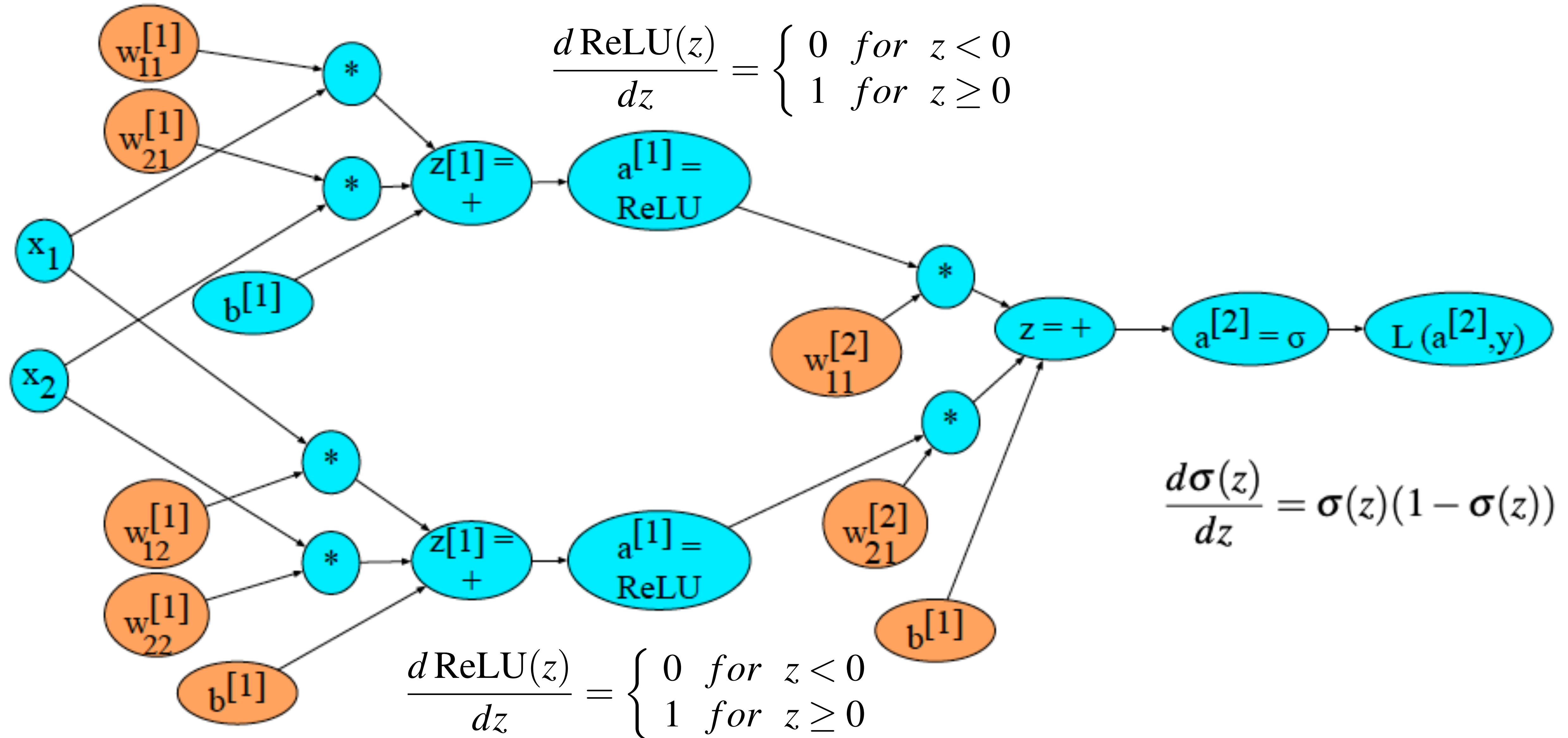
Backward differentiation on a two layer network

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

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

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

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 $z^{[2]}$)

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

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

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

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

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

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

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

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.

Simple Neural Networks and Neural Language Models

Computation Graphs and Backward Differentiation