

# Simple neural networks and logistic regression

NLP Week 5

Thanks to Dan Jurafsky for slides and inspiration!

# Plan for today

1. Text classification
2. Neural Units
3. Simple neural network architectures and logistic regression
4. Training models and Back-propagation
5. *Group exercises*

# Text classification

## Positive or negative movie review?



unbelievably disappointing



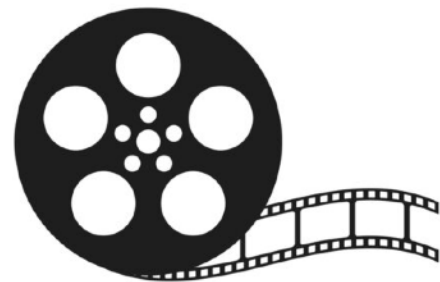
Full of zany characters and richly applied satire, and some great plot twists



this is the greatest screwball comedy ever filmed



It was pathetic. The worst part about it was the boxing scenes.



# Text classification

- **Other examples:**

- Review classification
- Spam detection
- Sentiment analysis
- Author detection
- Topic/subject assignment

**... and the list goes on!**

# Text classification

**Text classification is...**

**The task of assigning a class label to a document or text.**

*Input:*

- a document  $d$
- a fixed set of classes  $C = \{c_1, c_2, \dots, c_J\}$

*Output:* a predicted class  $c \in C$

# Text classification

Text classification is...

The task of assigning a class label to a document or text.

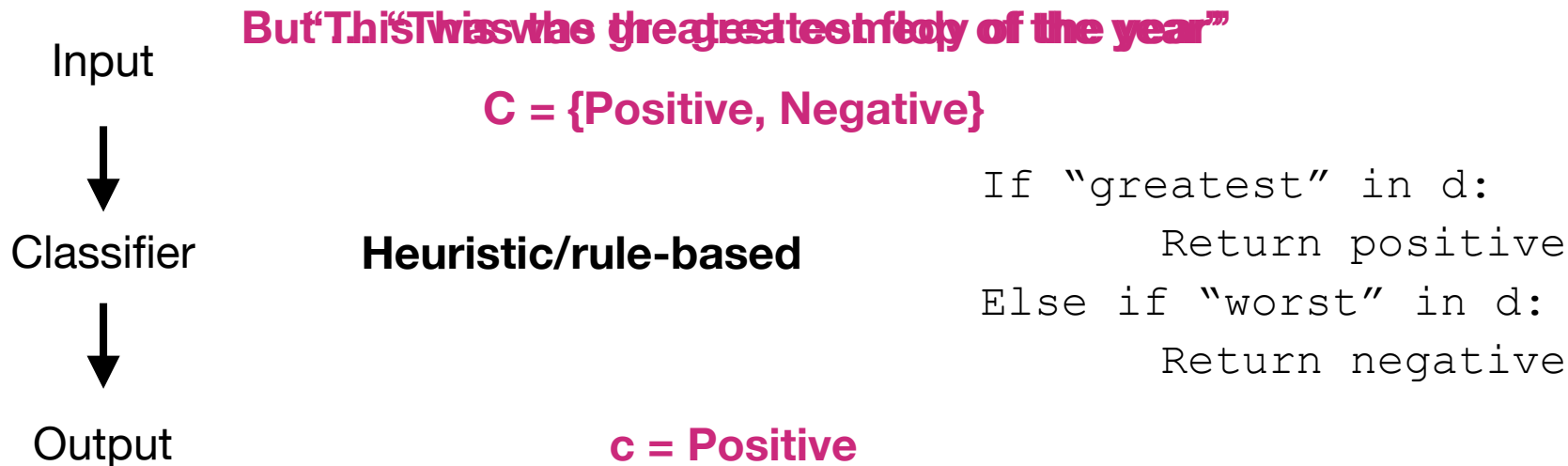
*Input:*

- a document  $d \leftarrow$  “This was the greatest comedy of the year”
- a fixed set of classes  $C \leftarrow \{c_1, \dots, c_n\} = \{\text{Positive, Neutral, Negative}\}$

*Output:* a predicted class  $c \in C$

# Text classification

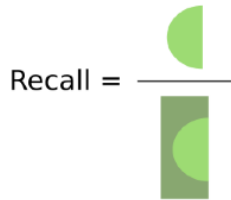
What should our classification algorithm look like?



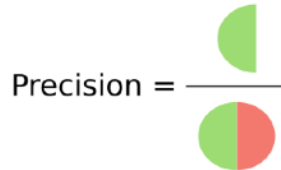
# Precision and Recall

- Heuristic classifiers tend to have high **precision** but very low **recall**.
- **Classifier accuracy is measured with precision and recall:**

How many relevant items are retrieved?

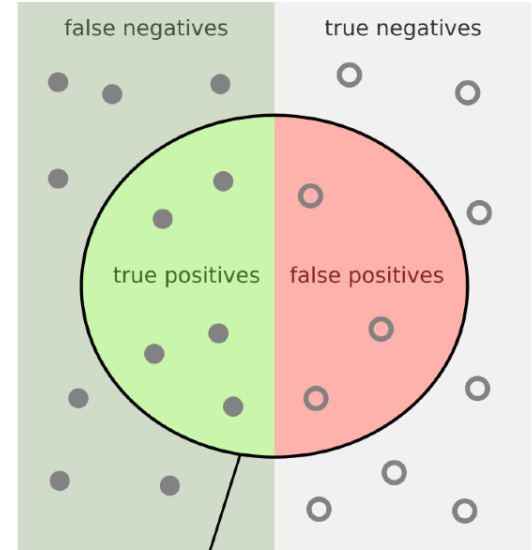


How many retrieved items are relevant?



$$\text{Recall} = \text{TP} / (\text{FN} + \text{TP})$$

$$\text{Precision} = \text{TP} / (\text{FP} + \text{TP})$$



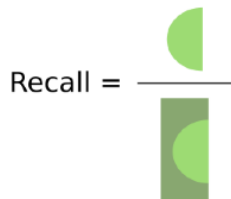


# Precision and Recall

**F-score** is the harmonic mean of the two:

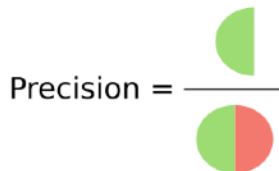
$$F_1 = \frac{2}{\text{recall}^{-1} + \text{precision}^{-1}} = 2 \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{2TP}{2TP + FP + FN}$$

How many relevant items are retrieved?

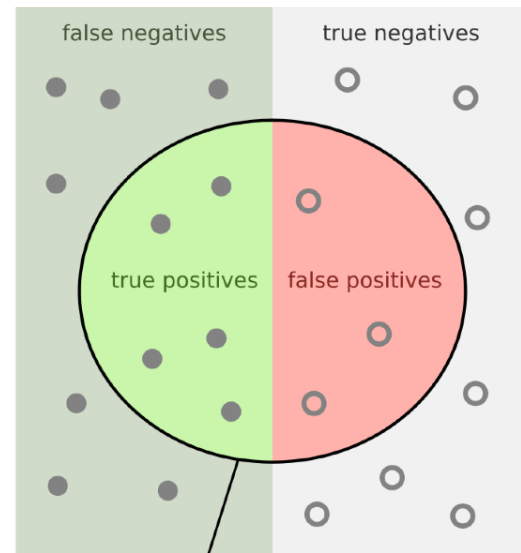


$$\text{Recall} = TP / (\mathbf{FN} + TP)$$

How many retrieved items are relevant?



$$\text{Precision} = TP / (\mathbf{FP} + TP)$$



# Supervised learning

An alternative to heuristic classifiers is to use machine learning to ***learn an algorithm rather than define an algorithm*** that will correctly classify texts.

We do this using ***supervised learning***. Supervised learning using a training data which is correctly ***labelled*** with the desired classes. We will then try to learn a set of shared features across documents for each class.

*Input:*

- a document  $d$
- a fixed set of classes  $C = \{c_1, c_2, \dots, c_J\}$
- A training set of  $m$  hand-labeled documents  $(d_1, c_1), \dots, (d_m, c_m)$

*Output:*

- a learned classifier  $\gamma: d \rightarrow c$

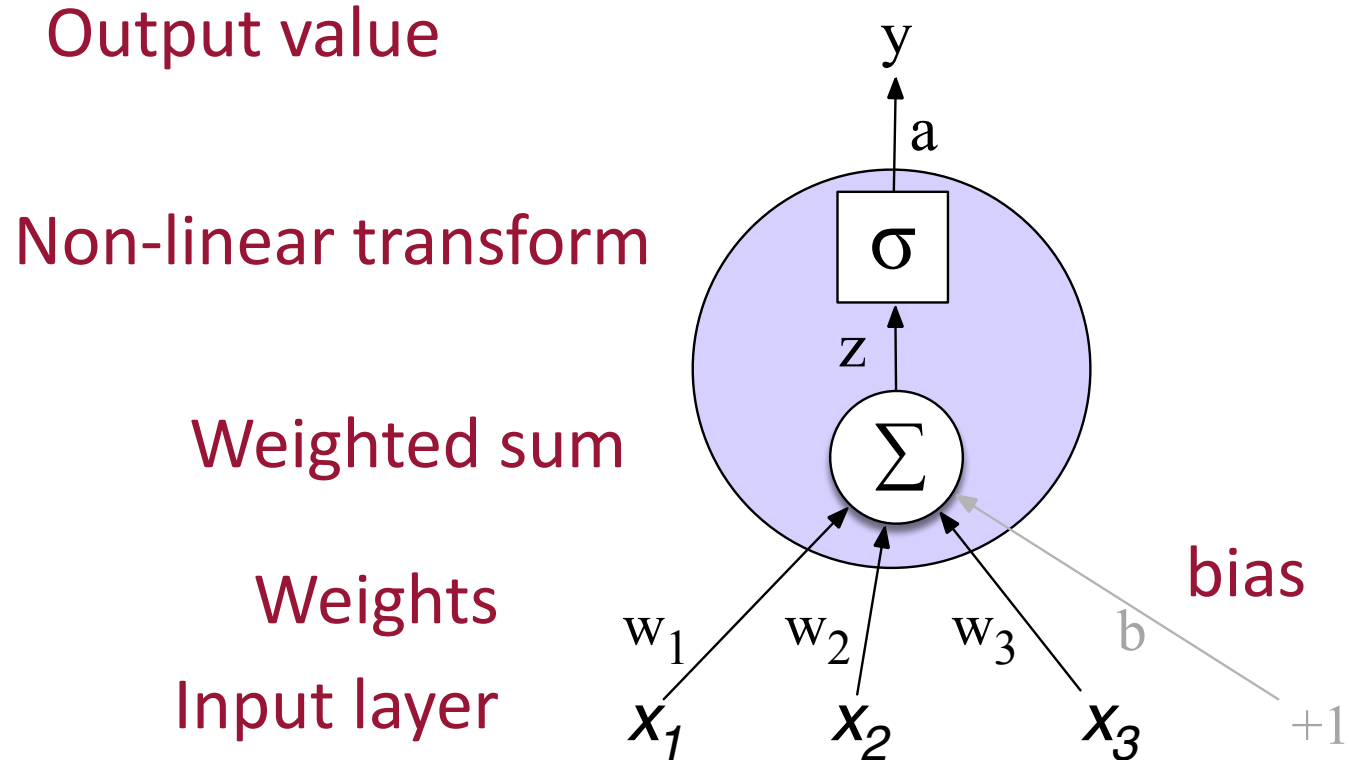
# Supervised learning

**There are many types of learned classifiers:**

- Naïve Bayes
- Logistic regression (this lecture)
- Neural networks (this lecture)
- ...
- Even finetuned LLMs or prompted LLMs

**All require labelled training data to do classification**

# Neural Network Unit



# 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  $f$ :

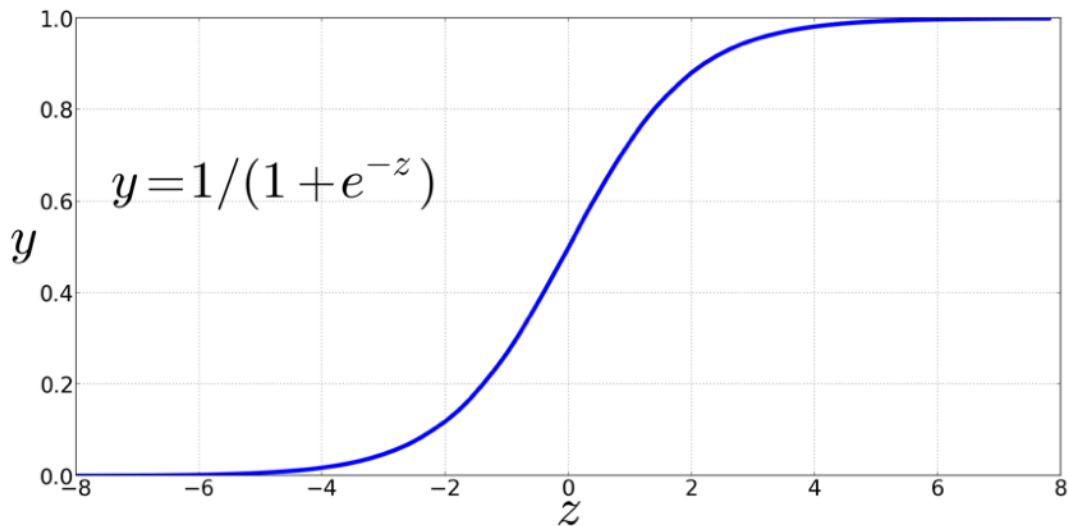
$$y = a = f(z)$$

# Non-Linear Activation Functions

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

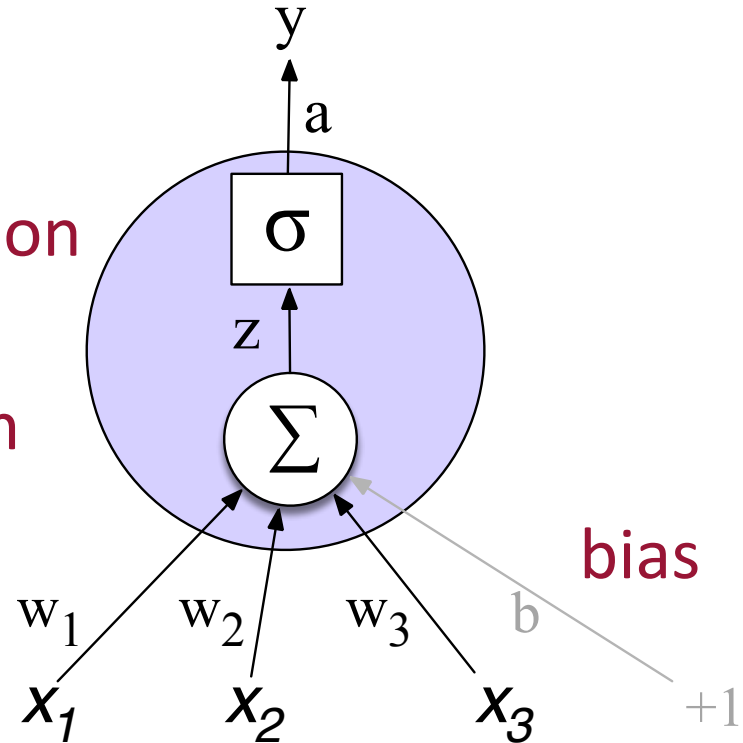
Output value

Non-linear activation function

Weighted sum

Weights

Input layer





# An example

*Suppose a unit has:*

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

$$b = 0.5$$

What happens with input  $x$ :

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

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

# An example

*Suppose a unit has:*

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

$$b = 0.5$$

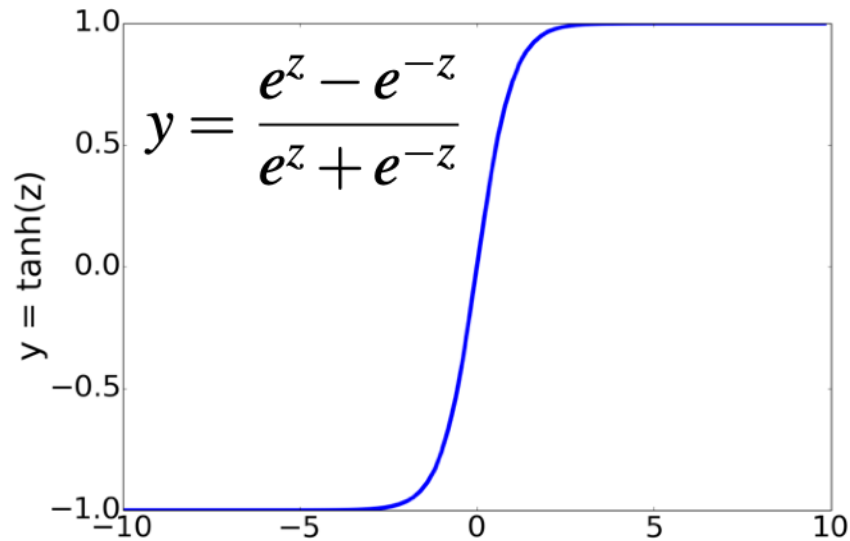
What happens with input  $x$ :

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

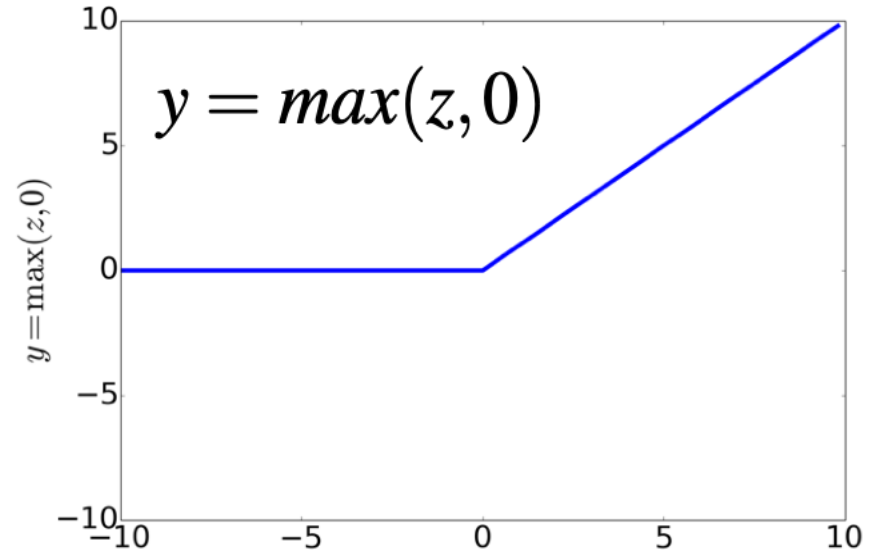
$$y = \sigma(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}} = \frac{1}{1 + e^{-(.5 \cdot .2 + .6 \cdot .3 + .1 \cdot .9 + .5)}} = \frac{1}{1 + e^{-0.87}} = .70$$

# Non-Linear Activation Functions besides sigmoid

Most Common:



tanh

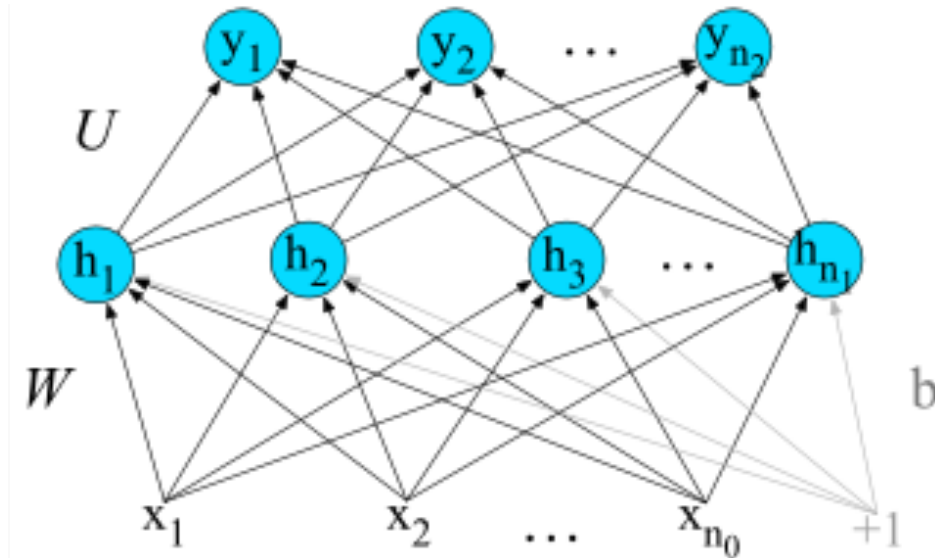


ReLU

Rectified Linear Unit

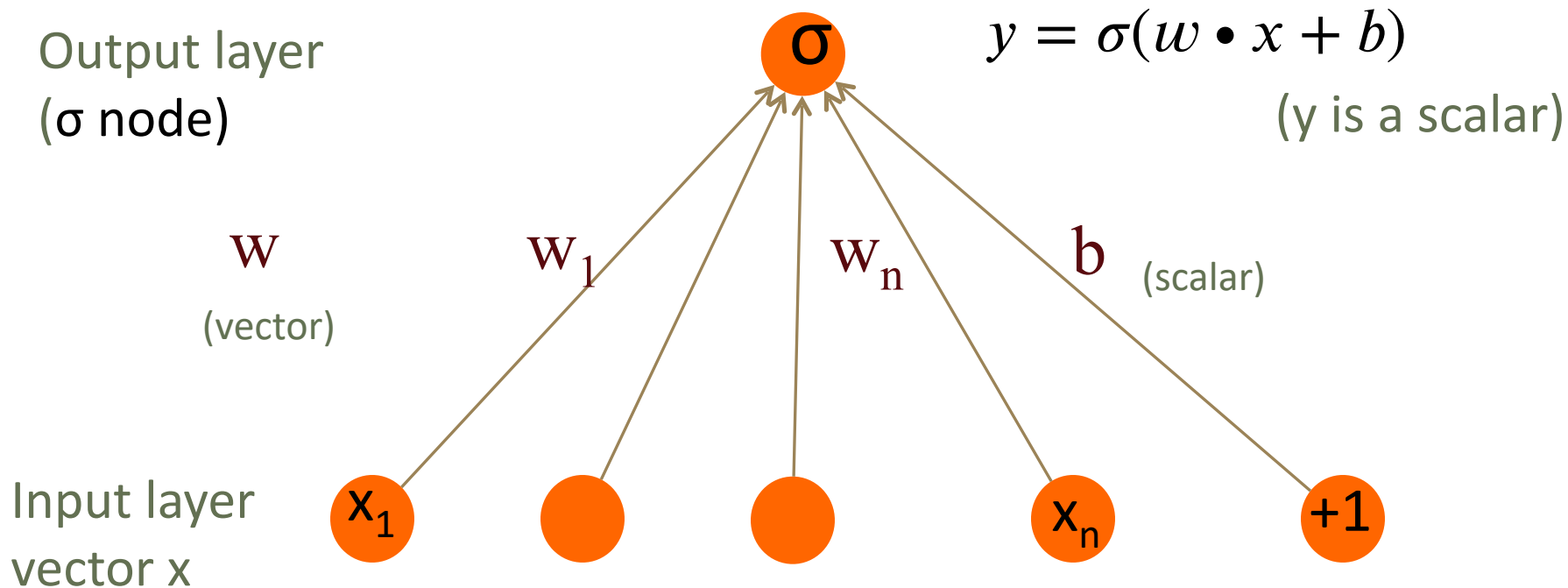
# Feedforward Neural Networks

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



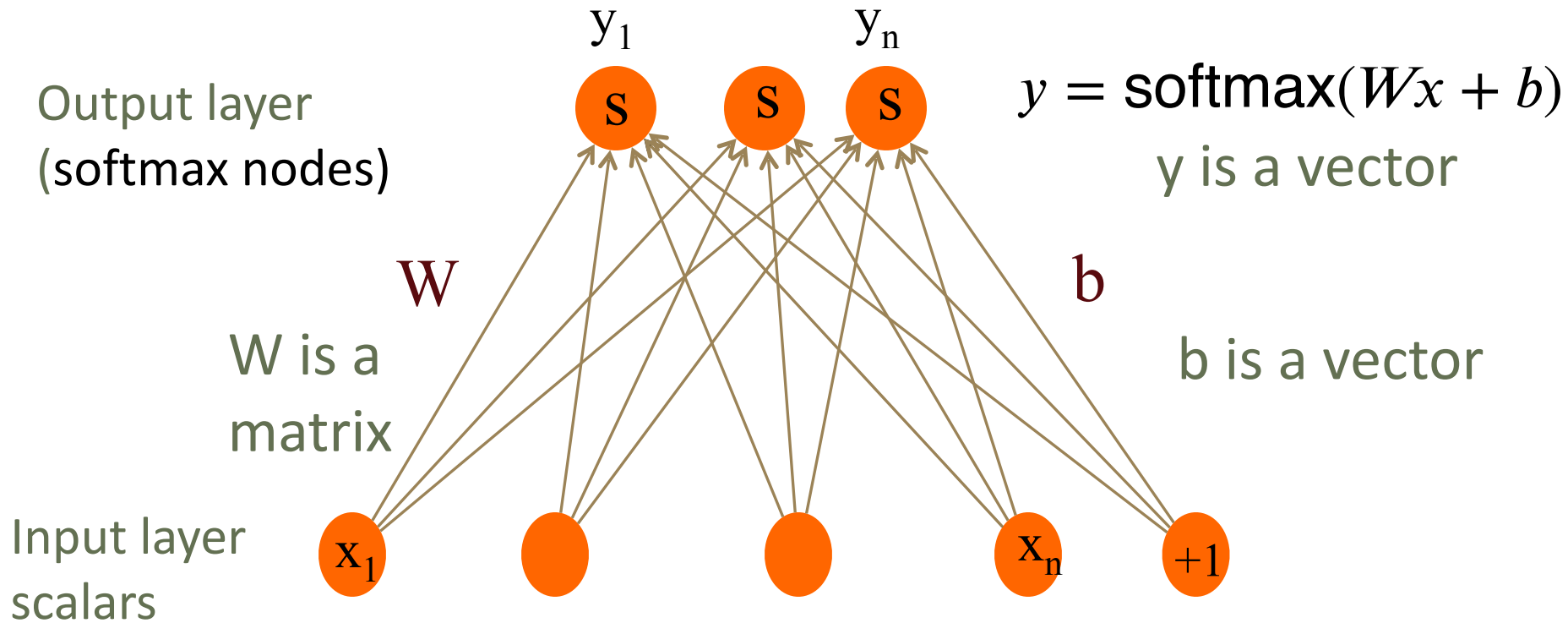
# Binary Logistic Regression as a 1-layer Network

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



# Multinomial Logistic Regression as a 1-layer Network

Fully connected single layer network



# Softmax: a generalization of sigmoid

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

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

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

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  
( $\sigma$  node)

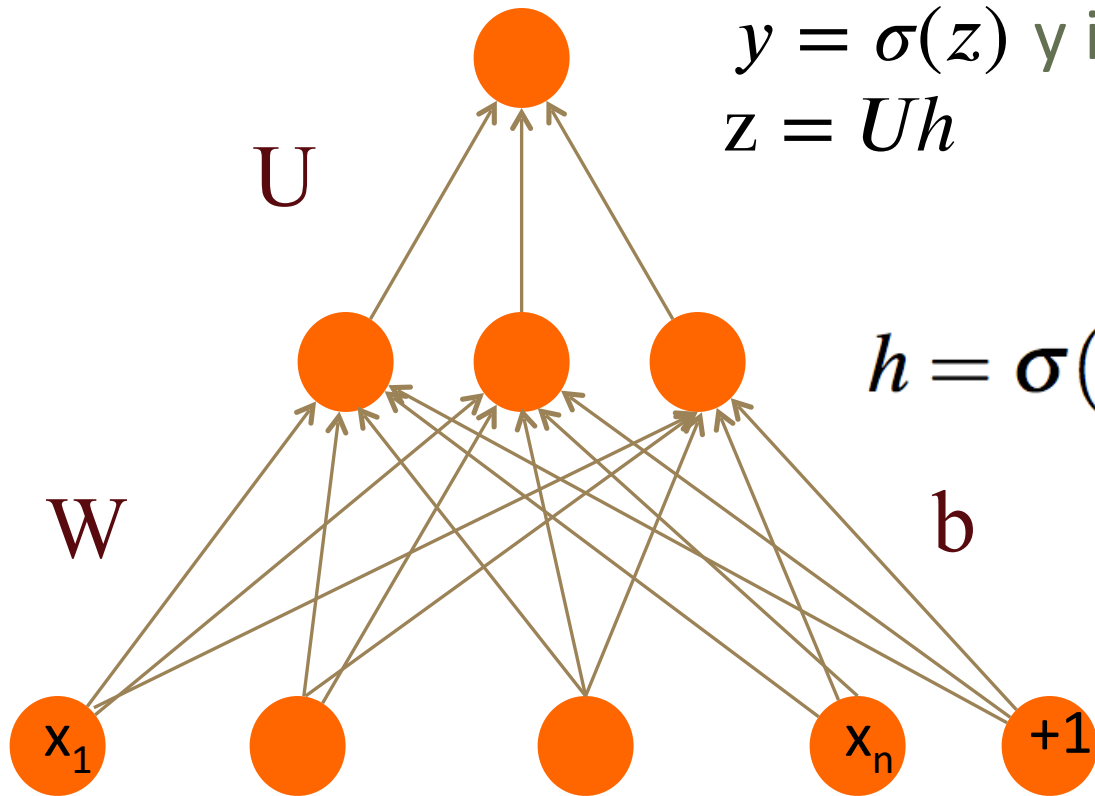
$$y = \sigma(z) \text{ } y \text{ is a scalar}$$
$$z = Uh$$

hidden units  
( $\sigma$  node)

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

Could be ReLU  
Or tanh

Input layer  
(vector)





# Two-Layer Network with scalar output

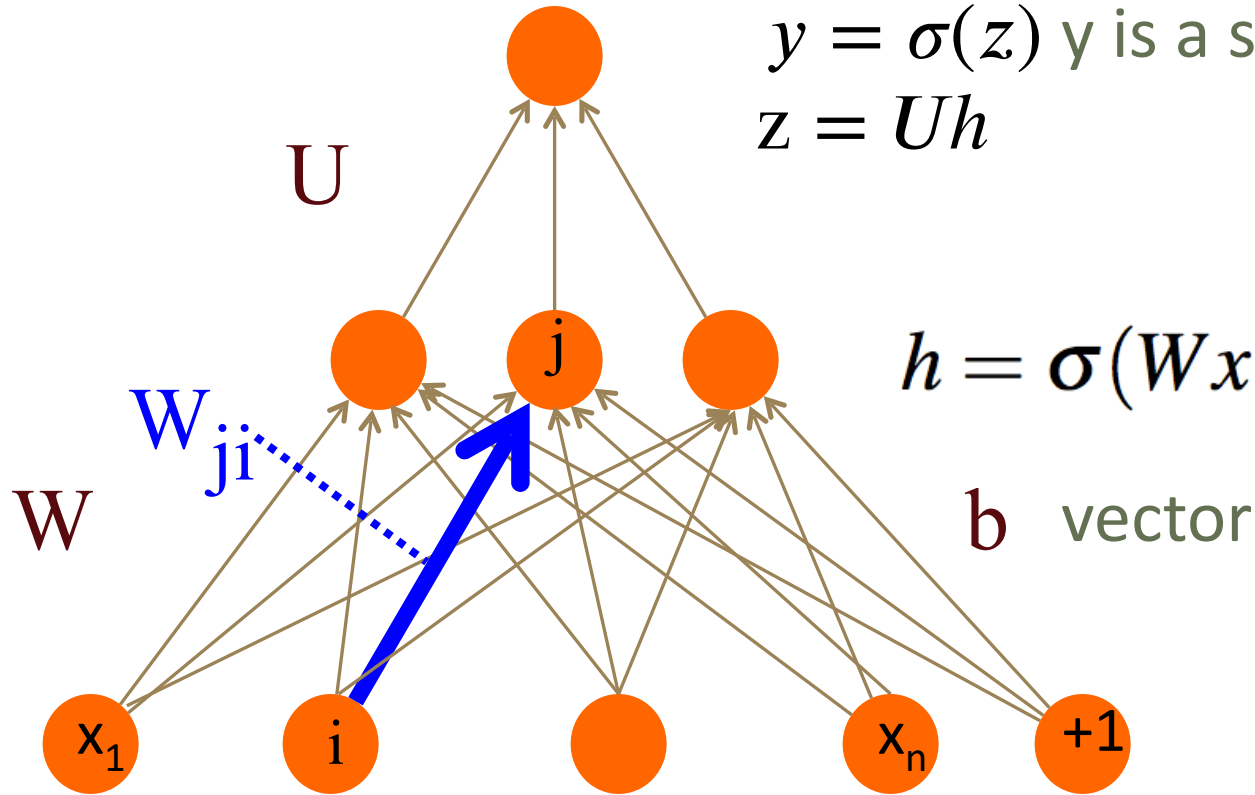
Output layer  
( $\sigma$  node)

$$y = \sigma(z) \text{ } y \text{ is a scalar}$$
$$z = Uh$$

hidden units  
( $\sigma$  node)

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

Input layer  
(vector)



# Two-Layer Network with scalar output

Output layer  
( $\sigma$  node)

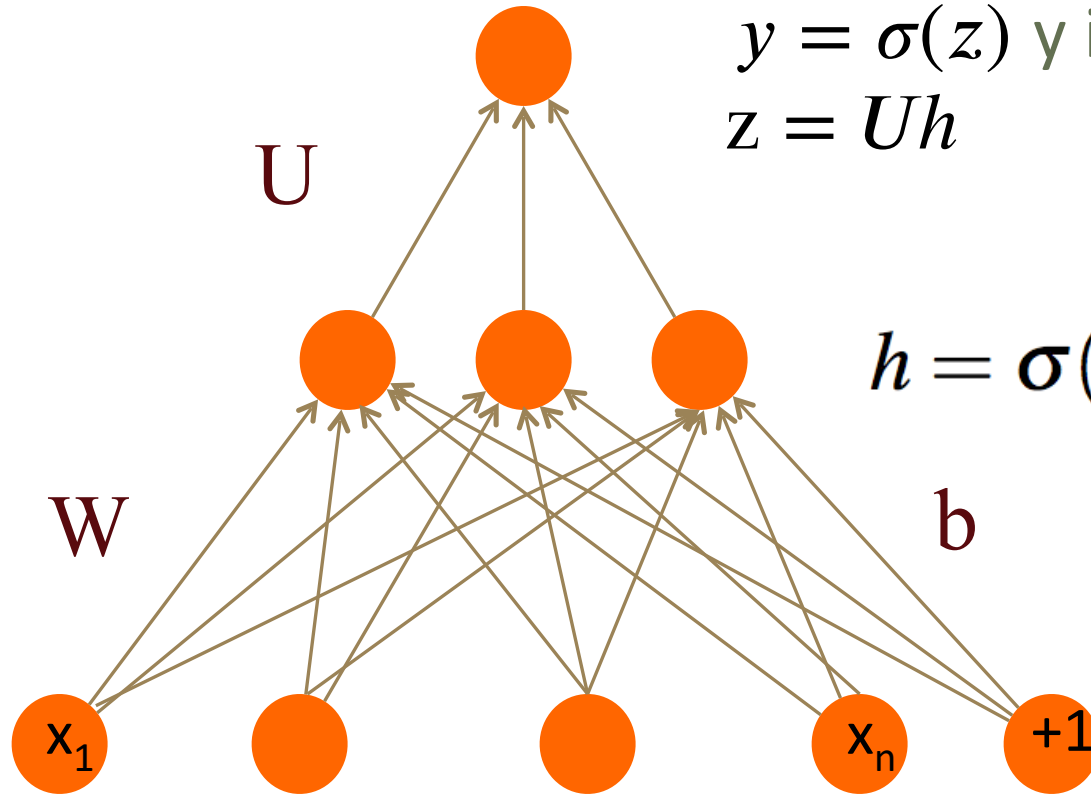
$$y = \sigma(z) \text{ } y \text{ is a scalar}$$
$$z = Uh$$

hidden units  
( $\sigma$  node)

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

Could be ReLU  
Or tanh

Input layer  
(vector)



# Two-Layer Network with softmax output

Output layer  
( $\sigma$  node)

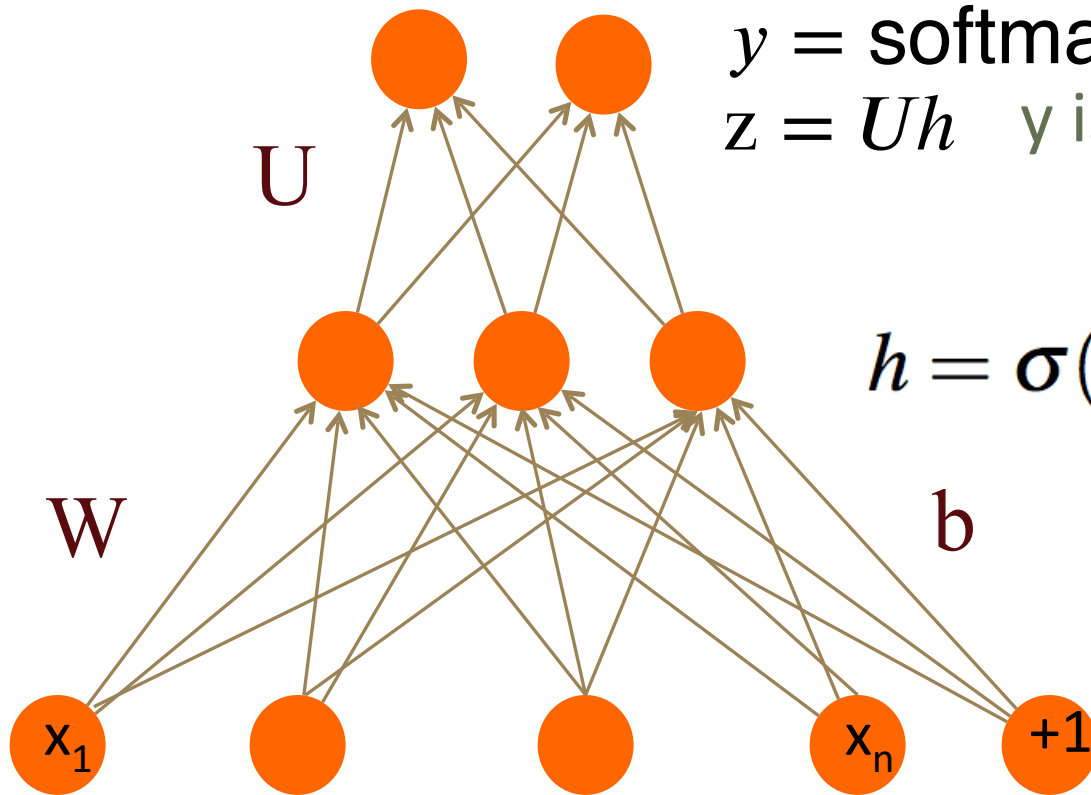
$$y = \text{softmax}(z)$$
$$z = Uh \quad y \text{ is a vector}$$

hidden units  
( $\sigma$  node)

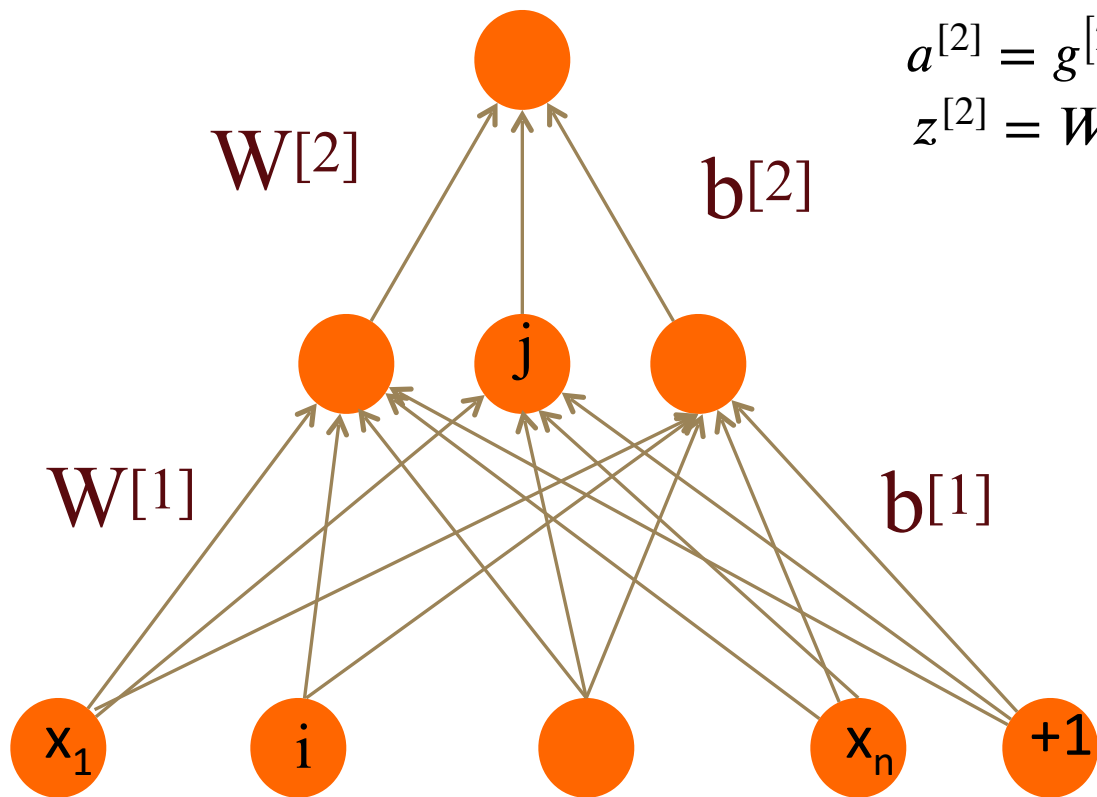
$$h = \sigma(Wx + b)$$

Could be ReLU  
Or tanh

Input layer  
(vector)



# Multi-layer Notation



$$y = a^{[2]}$$

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

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

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

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

$$a^{[0]}$$

# Multi Layer Notation

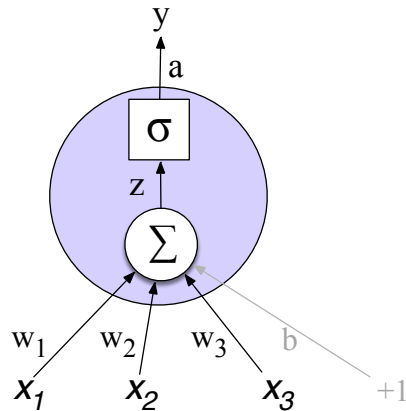
$$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$$

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

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

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

$$\hat{y} = a^{[2]}$$



**for  $i$  in  $1..n$**

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

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

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

# 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  $a^{[1]}_0=1$  ,  $a^{[2]}_0=1, \dots$

# Replacing the bias unit

Instead of:

$$x = x_1, x_2, \dots, x_{n_0}$$

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

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

We'll do this:

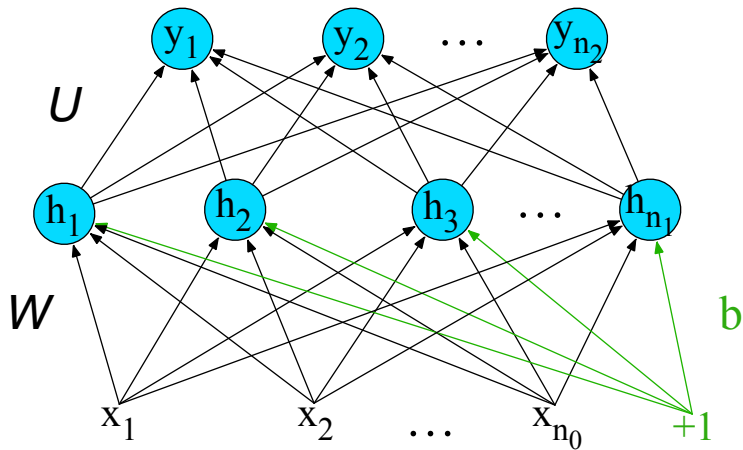
$$x = x_0, x_1, x_2, \dots, x_{n_0}$$

$$h = \sigma(Wx)$$

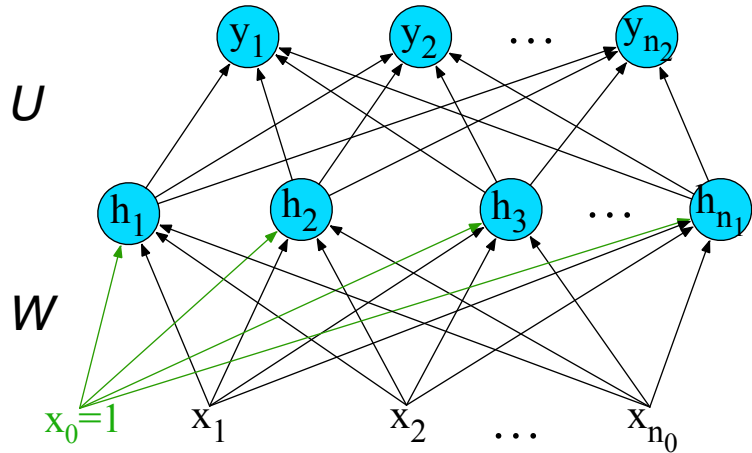
$$\sigma \left( \sum_{i=0}^{n_0} W_{ji} x_i \right)$$

# Replacing the bias unit

Instead of:



We'll do this:





# Use cases for feedforward networks

Let's consider 2 (simplified) sample tasks:

1. Text classification
2. Language modelling

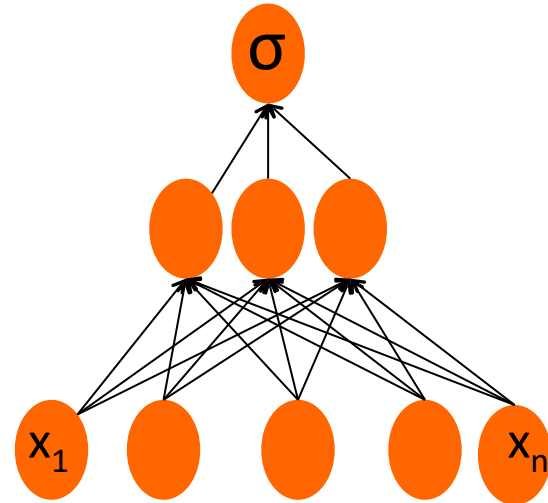
State of the art systems use more powerful neural architectures, but simple models are useful to consider!

# 1. Classification: Sentiment Analysis

We could do exactly what we did with logistic regression

Input layer are binary features as before, ex.  
(ex. Contains the word “great”)

Output layer is 0 (neg) or 1 (pos)

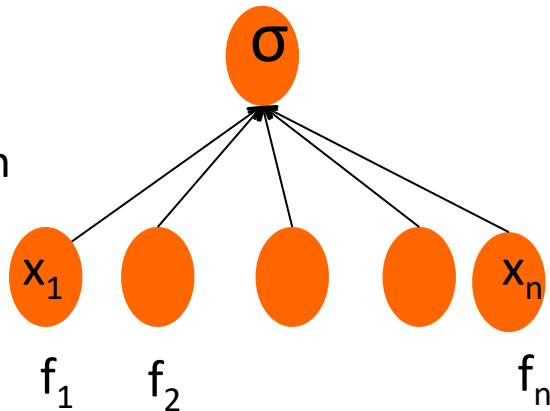


# Example sentiment input features

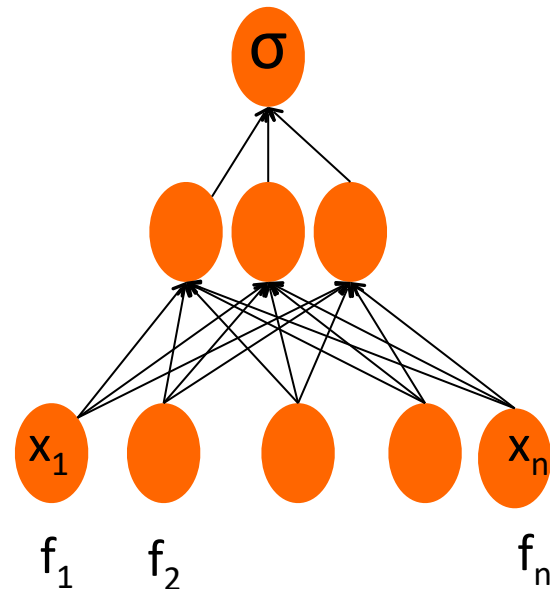
Var	Definition
$x_1$	count(positive lexicon) $\in$ doc)
$x_2$	count(negative lexicon) $\in$ doc)
$x_3$	$\begin{cases} 1 & \text{if "no"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$
$x_4$	count(1st and 2nd pronouns $\in$ doc)
$x_5$	$\begin{cases} 1 & \text{if "!"} \in \text{doc} \\ 0 & \text{otherwise}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

# Feedforward nets for simple classification

Logistic  
Regression



2-layer  
feedforward  
network



Just adding a hidden layer to logistic regression

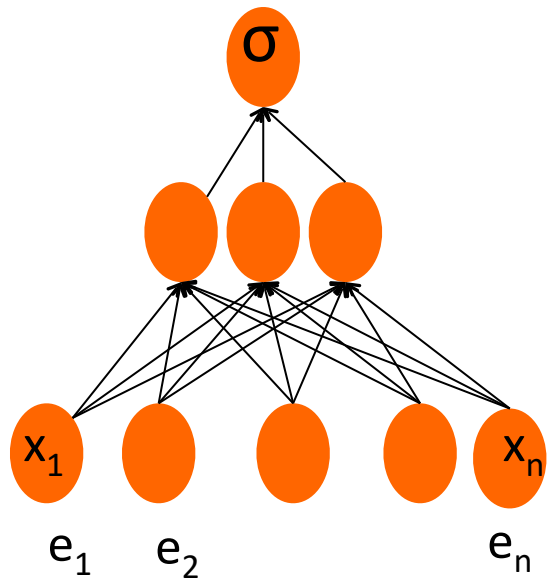
- allows the network to use non-linear interactions between features which may (or may not) improve performance.

# Even better: representation learning

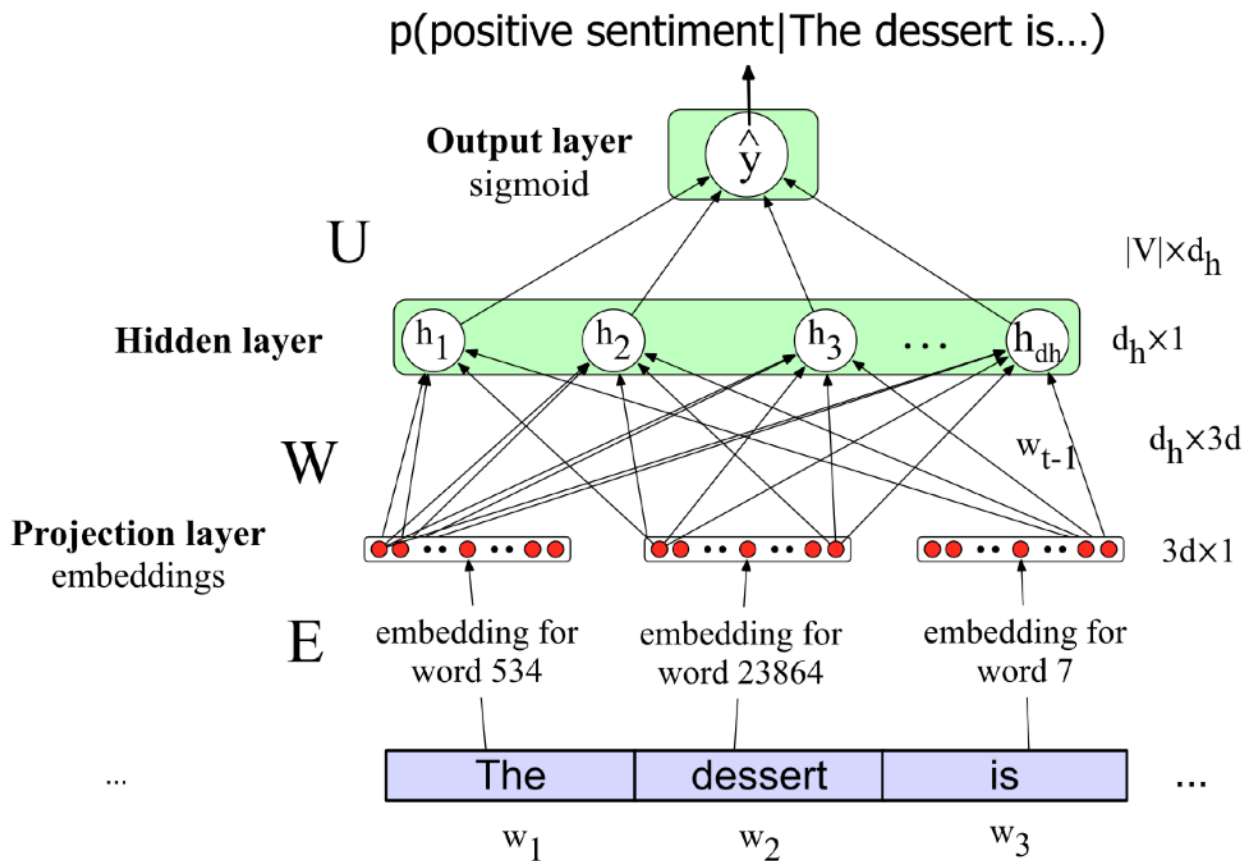
The real power of deep learning comes from the ability to **learn** features from the data

Instead of using hand-built human-engineered features for classification

Use learned representations like embeddings!



# Neural Net Classification with embeddings as input features!

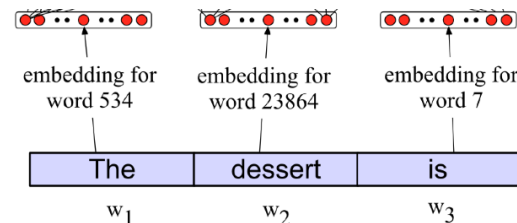


# Issue: texts come in different sizes

This assumes a fixed size length (3)!

Kind of unrealistic.

Some simple solutions (more sophisticated solutions later)



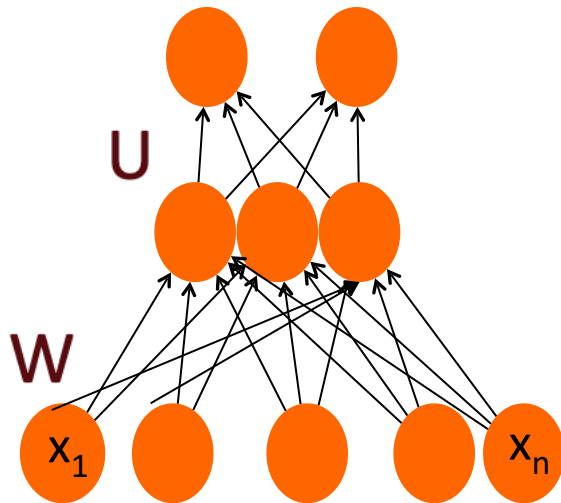
1. Make the input the length of the longest review
  - If shorter then pad with zero embeddings
  - Truncate if you get longer reviews at test time
2. Create a single "sentence embedding" (the same dimensionality as a word) to represent all the words
  - Take the mean of all the word embeddings
  - Take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

# 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”

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





## 2. Language modelling with FNNs

**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** can do almost as well!

# Simple feedforward Neural Language Models

**Task:** predict next word  $w_t$

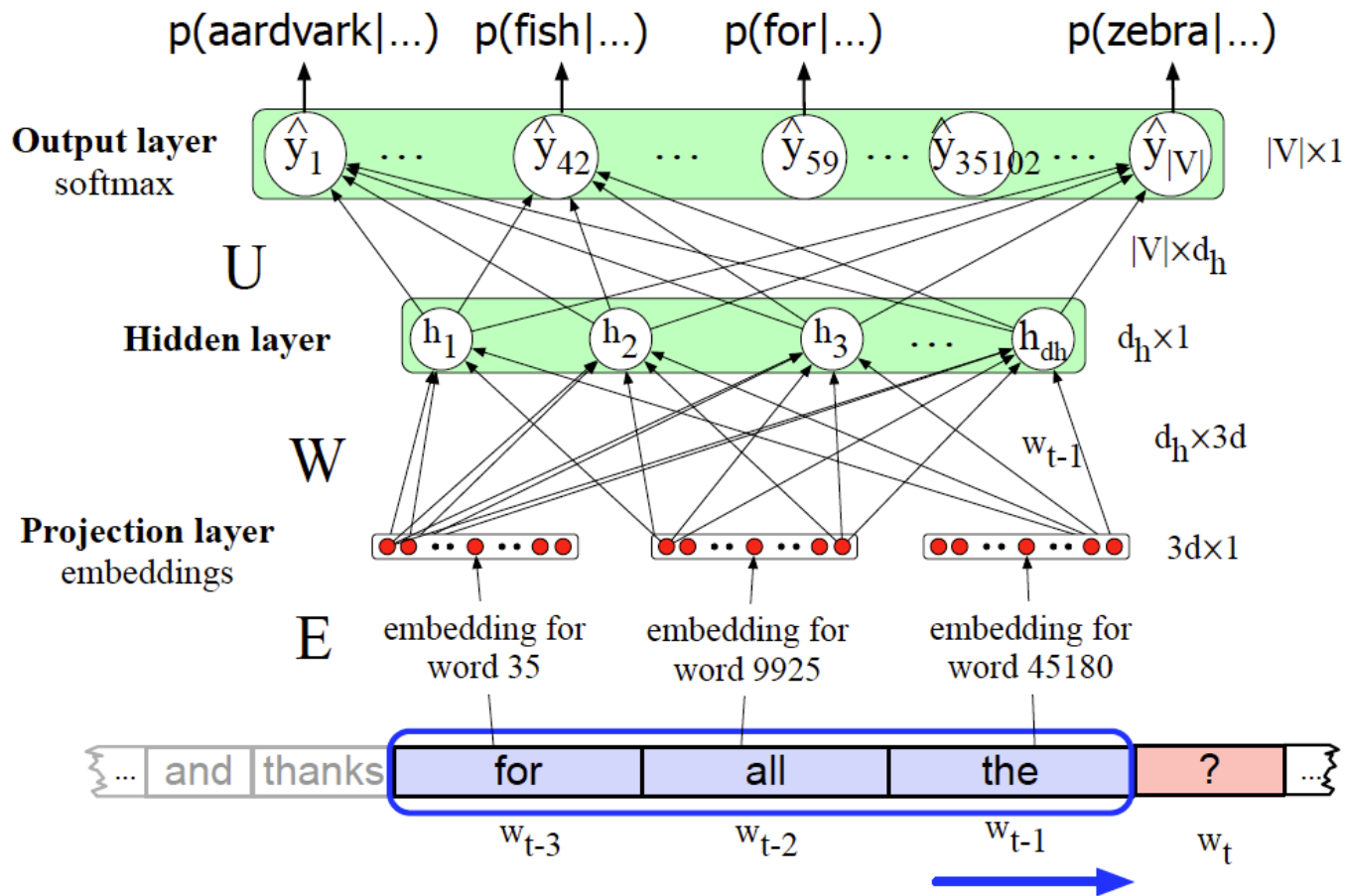
given prior words  $w_{t-1}, w_{t-2}, w_{t-3}, \dots$

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

**Solution:** Sliding windows (of fixed length)

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

# 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:

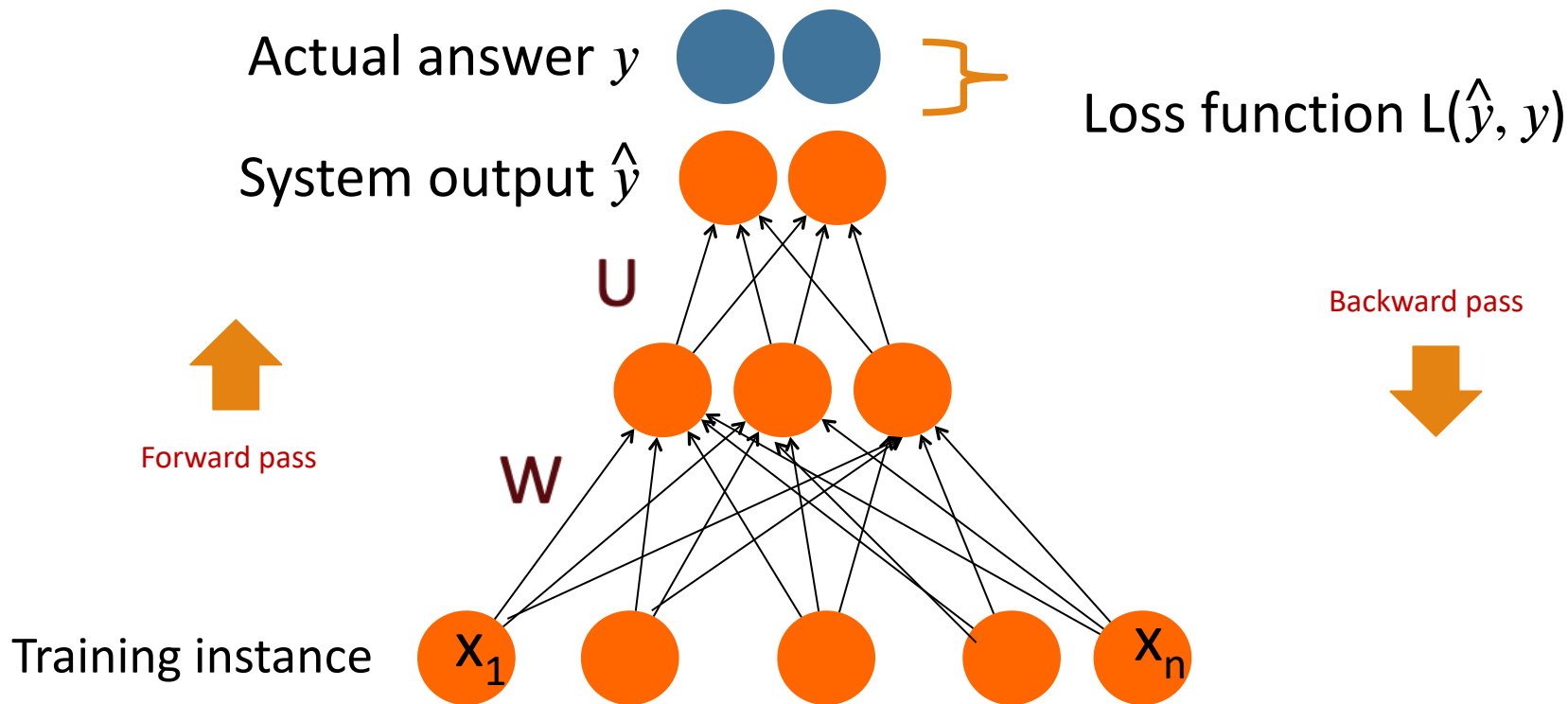
I forgot to make sure that the dog gets \_\_\_\_

N-gram LM can't predict "fed"!

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

**Great, but how do we train, or fit, a neural network model to data?**

# 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

# 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:

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



# Gradient descent for weight updates

Use the derivative of the loss function with respect to weights  $\frac{d}{dw}L(f(x; w), y)$

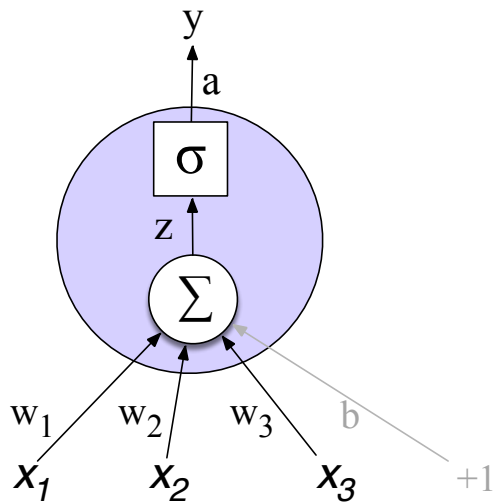
To tell us how to adjust weights for each training item

- Move them in the opposite direction of the gradient

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

# Where did that derivative come from?

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



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:

- 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

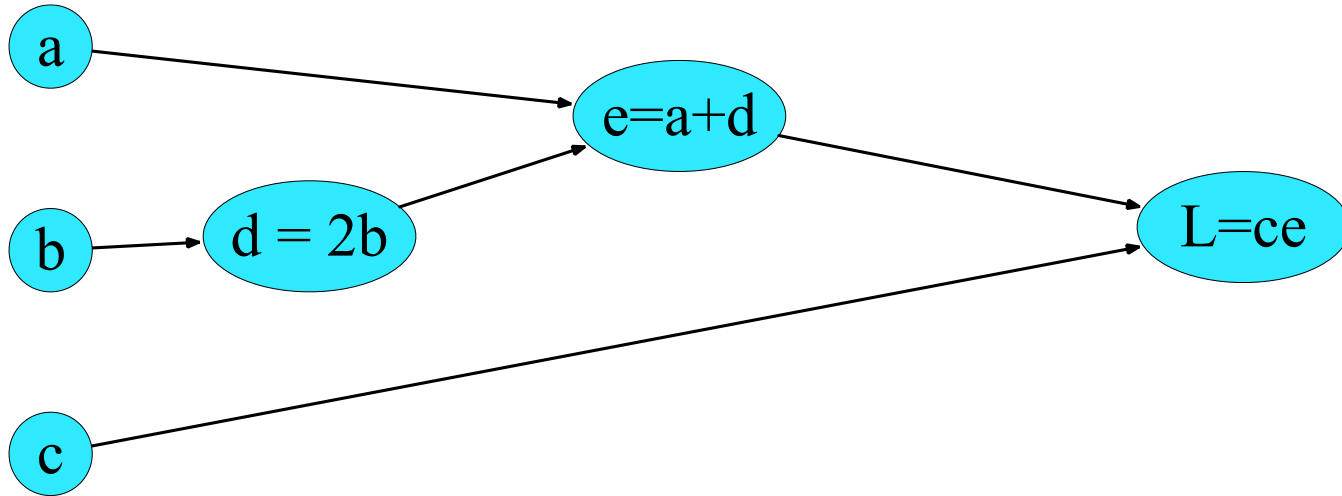
**Example:**  $L(a, b, c) = c(a + 2b)$

Computations:

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$



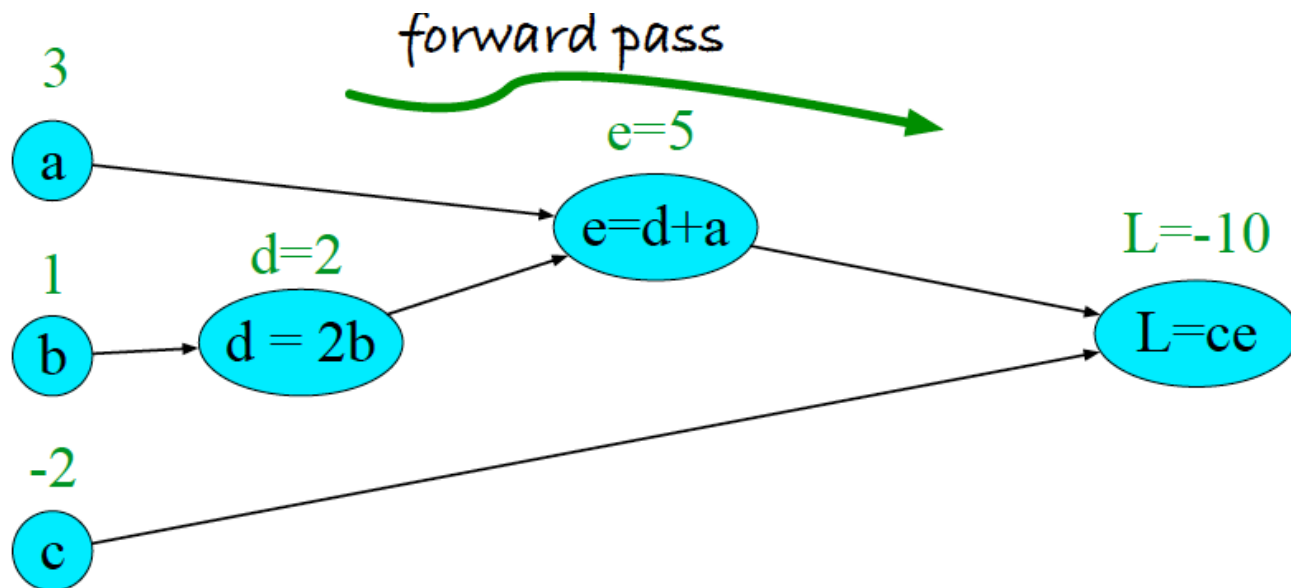
**Example:**  $L(a,b,c) = c(a + 2b)$

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

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

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  $L$ .

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

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

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

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

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

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

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

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

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

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

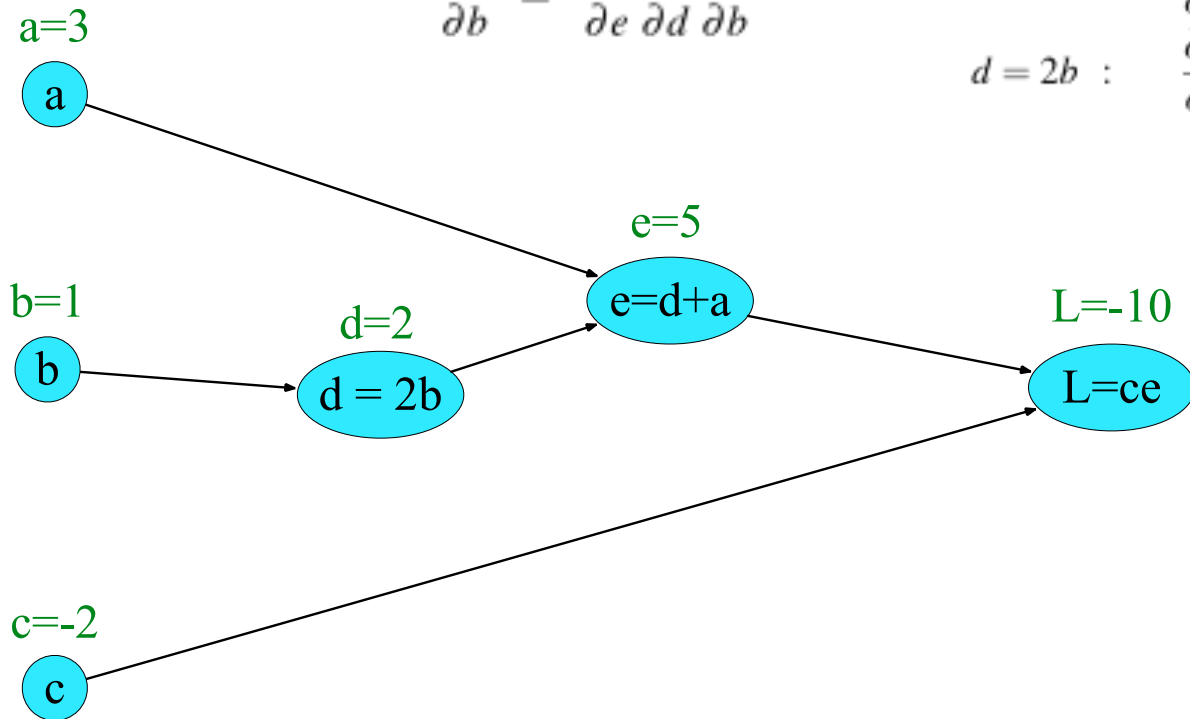
$$e = a + d : \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

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

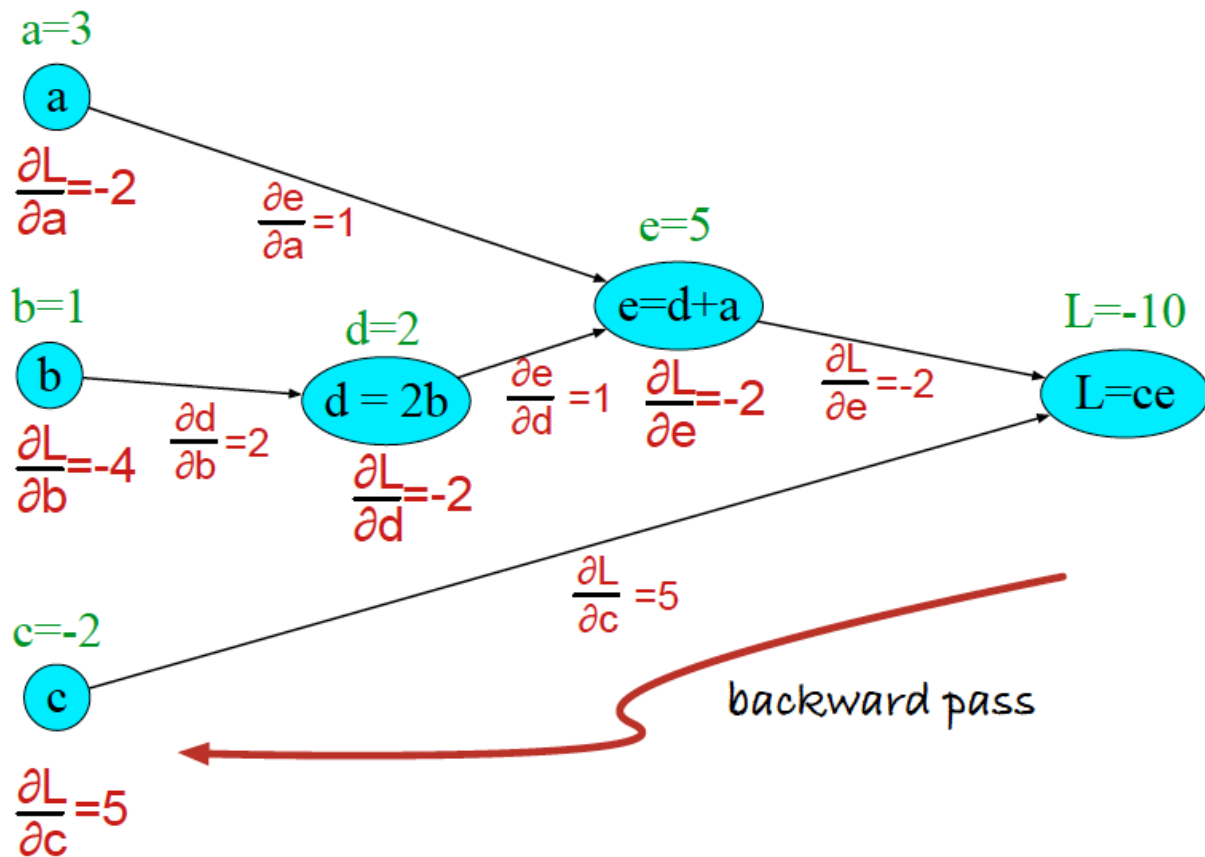
# Example

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

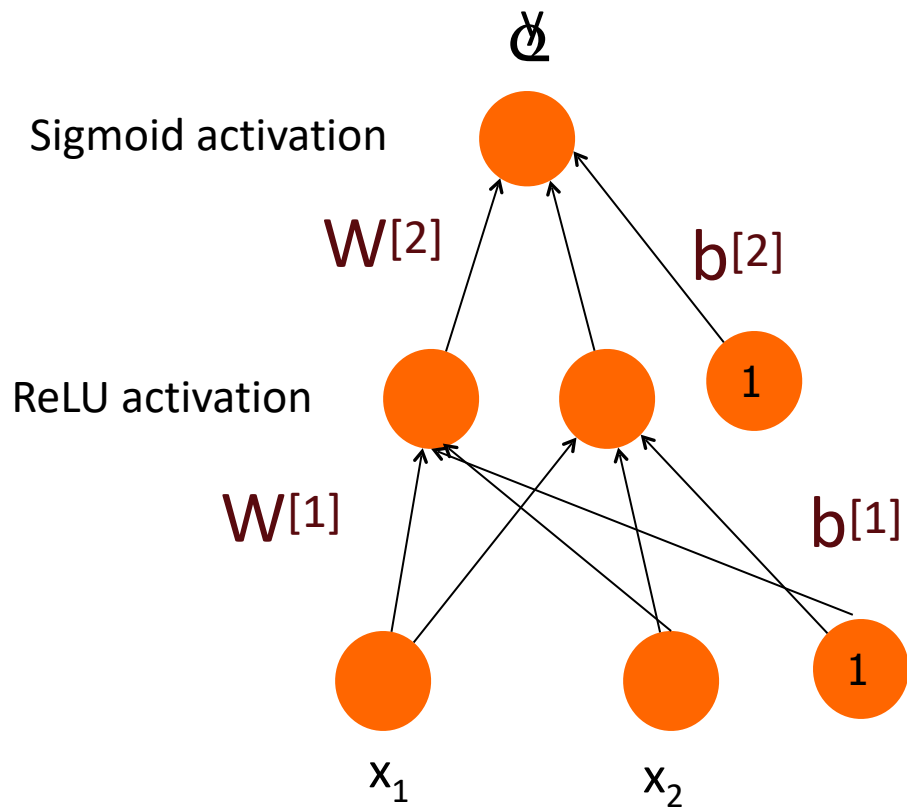
$$L = ce : \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$
$$e = a + d : \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$
$$d = 2b : \quad \frac{\partial d}{\partial b} = 2$$



# Example



# Backward differentiation on a two layer network



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

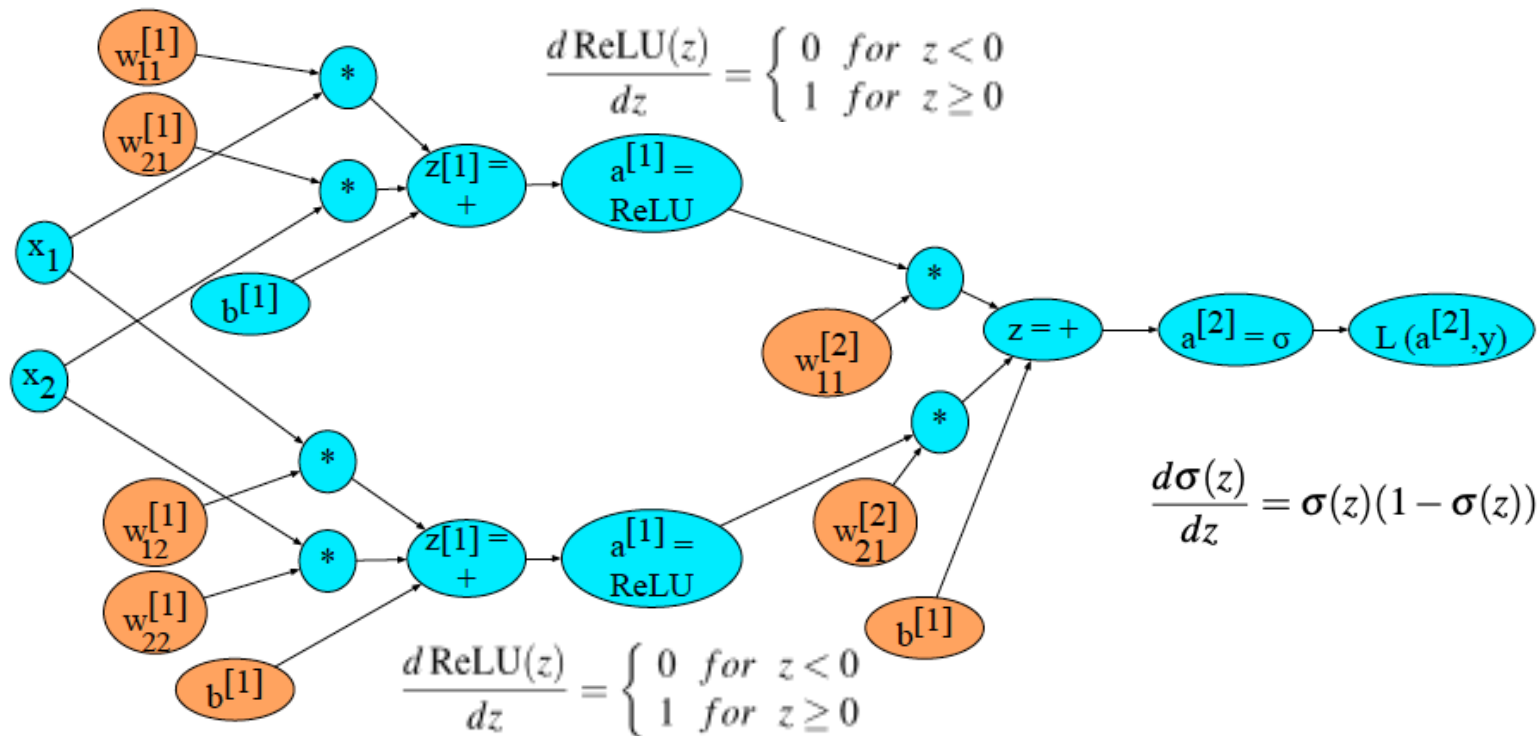
$$a^{[1]} = \text{ReLU}(z^{[1]})$$

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

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

$$\hat{y} = a^{[2]}$$

# Backward differentiation on a two layer network





# 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.

# Reminder: Gradient descent for weight updates

Use the derivative of the loss function with respect to weights  $\frac{d}{dw}L(f(x; w), y)$

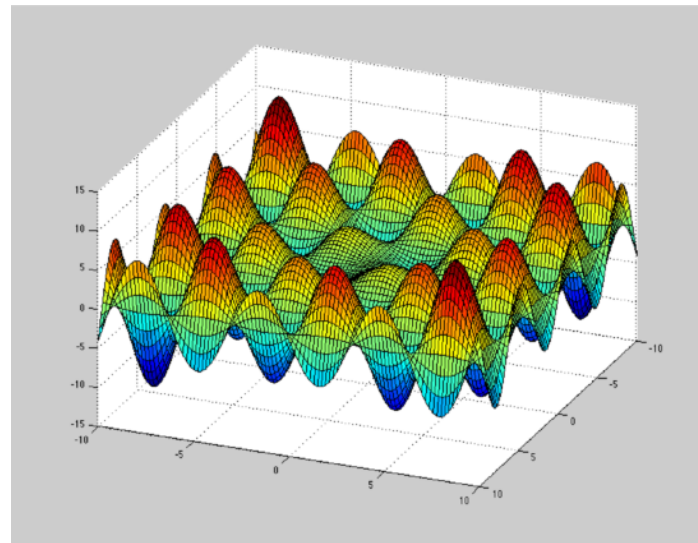
To tell us how to adjust weights for each training item

- Move them in the opposite direction of the gradient

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

# Optimization

- NN optimization is a non-convex problem (i.e. there are many local optima)
- So how do we try to find the best one and avoid overfitting?



# Hyperparameters

Weights (**W**, **b**) are the model parameters which will update during training, hyperparameters are those which we set from the start and do not update but affect the computational graph.

# Hyperparameters

Here are some hyperparameters which will affect model optimization:

- Learning rate
- Number of epochs
- Minibatch size
- Number of layers
- Hidden layer sizes
- ...

**[15 minute break]**

# Working with NN models!

**Team up!**

**Open exercises/week 5 in your course folder and start writing/running code!**