

CS 2731

Introduction to Natural Language Processing

Session 8: Feedforward neural networks

Michael Miller Yoder

September 23, 2024

Course logistics

- [Homework 2](#) is due **next Thu Oct 3**
 - Text classification
 - Written and programming components
 - Optional Kaggle competition for best LR and NN politeness classifiers
- Projects
 - Thanks for submitting your project ideas
 - Look out for the project ranking form (released today), due **this Thu Sep 26**
- Thanks for the discussion posts!

Blodgett et al. 2020 summary

- Recommendations from Blodgett et al. for better work on bias
 1. Ground work analyzing bias in relevant literature outside of NLP that explores relationships between language and social hierarchies. Treat representational harms as harmful in their own right
 2. Explicitly state why “bias” in systems is harmful, in what ways, and to whom. Be explicit about normative reasoning behind these judgements.
 3. Engage with the lived experiences of members of communities affected by NLP systems. Reimagine power relations between technologists and such communities.

Discussion forum on Blodgett et al. 2020

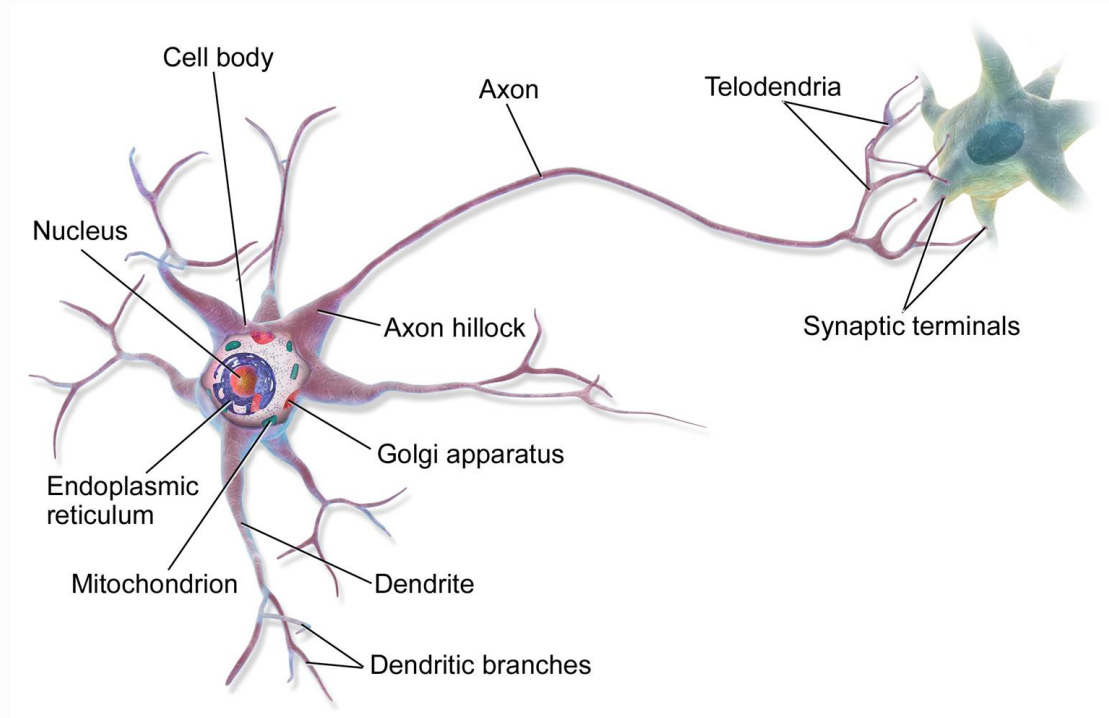
- Language ideologies
 - AAVE, Indian English (Dilip), Bhojpuri (Anveshika), regional Chinese dialects (Yifang), regional Korean dialects (Geonyeong)
 - 'illegals' (Hugh)
 - Can limit opportunities for speakers of non-standard dialects
 - Only “prestigious” dialects or languages in NLP training data (Xiaoyan)
- Allocational harms
 - Targeted emails for applications to MIT (John)
 - Underrepresentation of marginalized genders, ethnicities in EHR data, medical field (Akshat, Jiyang)
 - Resume filters (Rojin)
 - Loans (Zhuochun, Rojin)
 - Policing and sentencing (Rojin, Maanya)
 - Can more easily explore, “correct” biases in AI systems (Joel)
 - What can developers do to avoid biases? (Xianglong)
- CS researchers need to collaborate with others to address social issues (Yushui)

Lecture overview: feedforward neural networks

- Neural network fundamentals
- Non-linear activation functions
- Linear algebra review
- Feedforward neural networks as classifiers
- Training feedforward neural networks (backpropagation)

Neural network fundamentals

This is in your brain



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

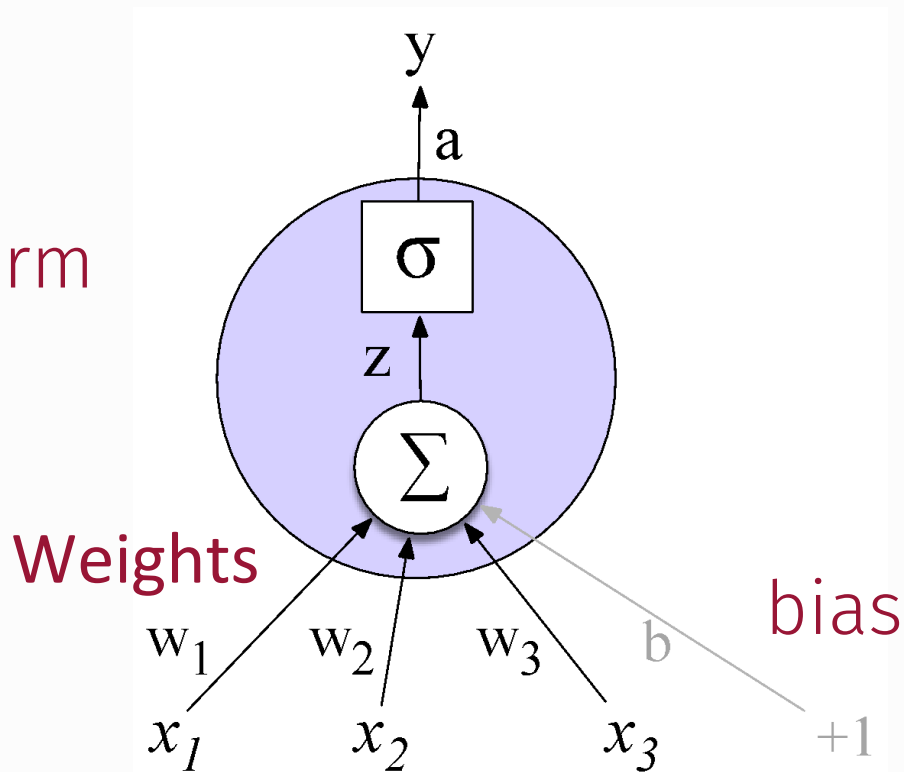
Neural Network Unit: This is not in your brain

Output value

Non-linear transform

Weighted sum

Input layer



Weights

bias

The Variables in Our Very Important Formula

- x** A vector of features of n dimensions (like number of positive sentiment words, length of document, etc.)
- w** A vector of weights of n dimensions specifying how discriminative each feature is
- b** A scalar bias term that shifts z
- z** The raw score
- y** A random variable (e.g., $y = 1$ means positive sentiment and $y = 0$ means negative sentiment)

The Fundamentals

The fundamental equation that describes a unit of a neural network should look very familiar:

$$z = b + \sum_i w_i x_i \quad (1)$$

Which we will represent as

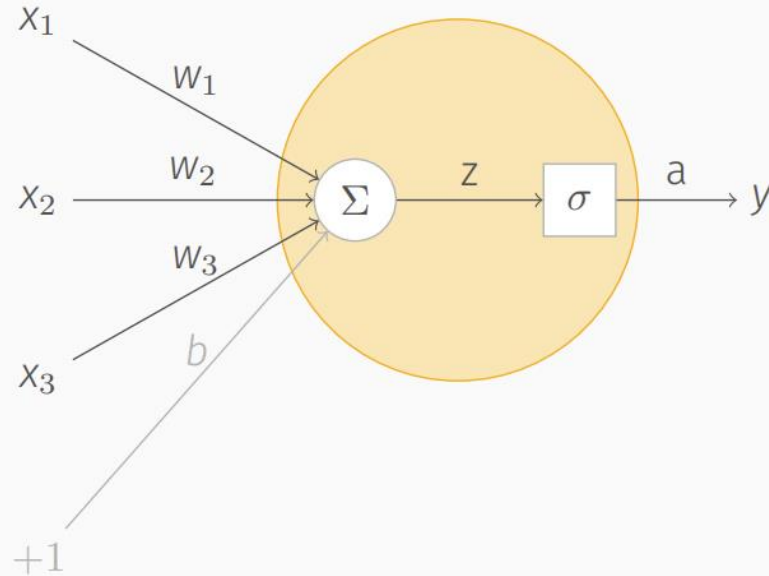
$$z = \mathbf{w} \cdot \mathbf{x} + b \quad (2)$$

But we do not use z directly. Instead, we pass it through a non-linear function, like the sigmoid function:

$$y = \sigma(z) = \frac{1}{1 + e^{-z}} \quad (3)$$

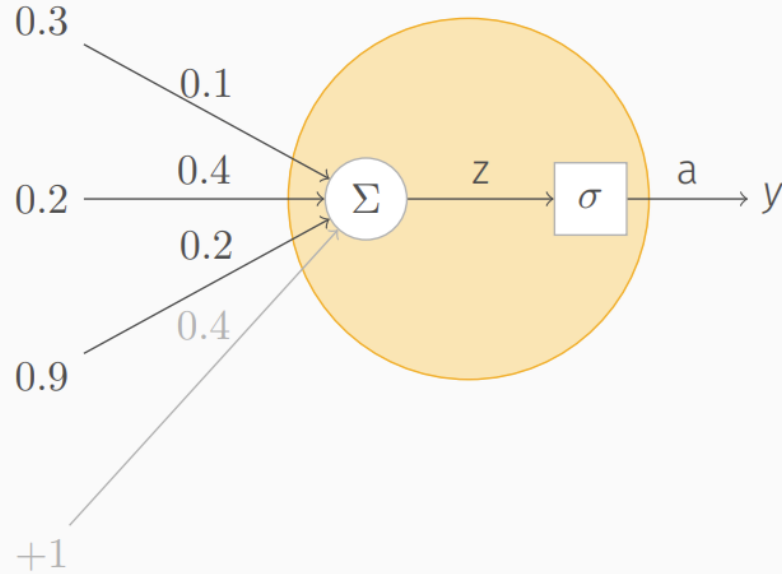
(which has some nice properties even though, in practice, we will prefer other functions like tanh and ReLU).

A Unit Illustrated

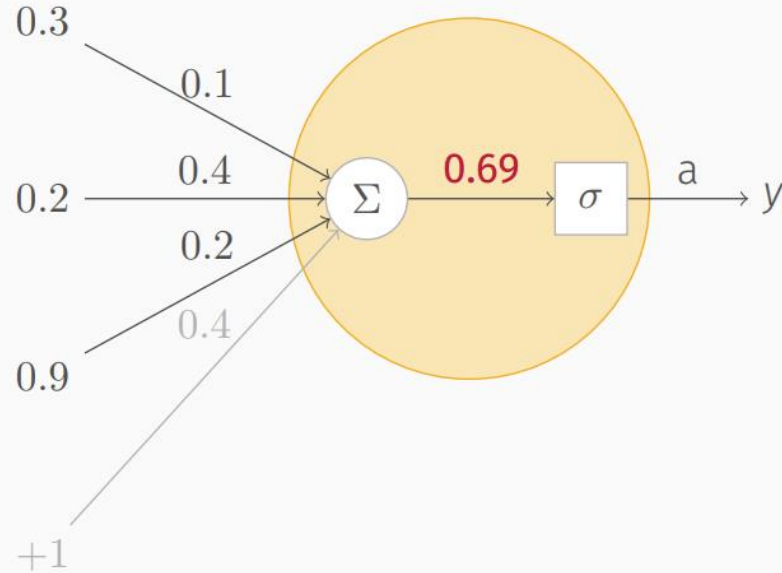


Take, for example, a scenario in which our unit has the weights $[0.1, 0.4, 0.2]$ and the bias term 0.4 and the input vector x has the values $[0.3, 0.2, 0.9]$.

Filling in the Input Values and Weights

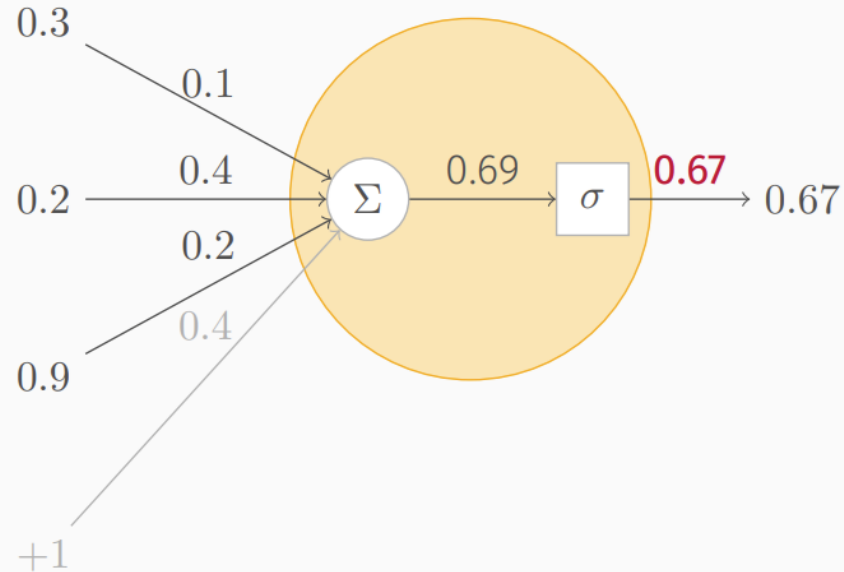


Multiplying the Input Values and Weights and Summing Them (with the Bias Term)



$$z = x_1w_1 + x_2w_2 + x_3w_3 + b = 0.1(0.3) + 0.4(0.2) + 0.2(0.9) + 0.4 = 0.69 \quad (4)$$

Applying the Activation Function (Sigmoid)



$$y = \sigma(0.69) = \frac{1}{1 + e^{-0.69}} = 0.67 \quad (5)$$

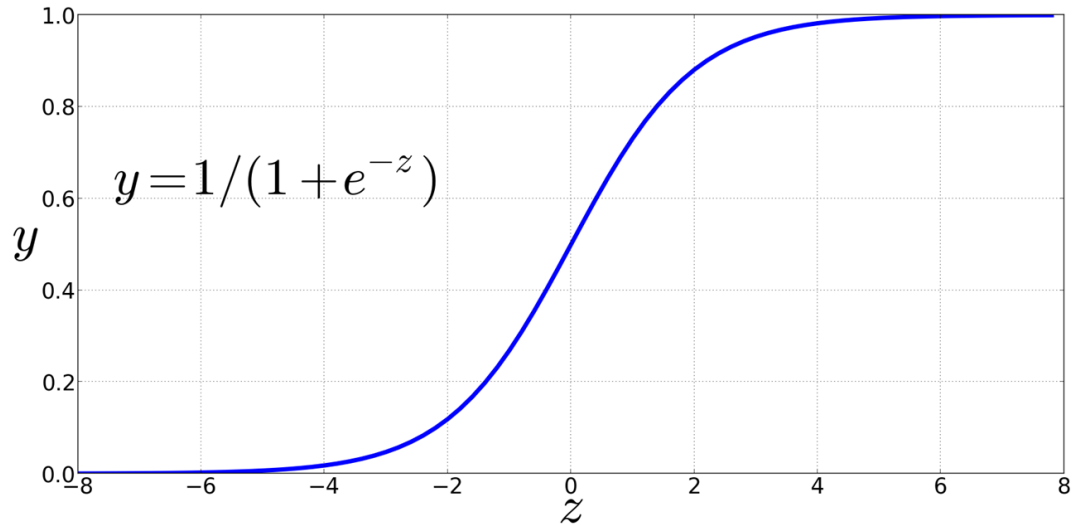
Non-linear activation functions

Non-Linear Activation Functions

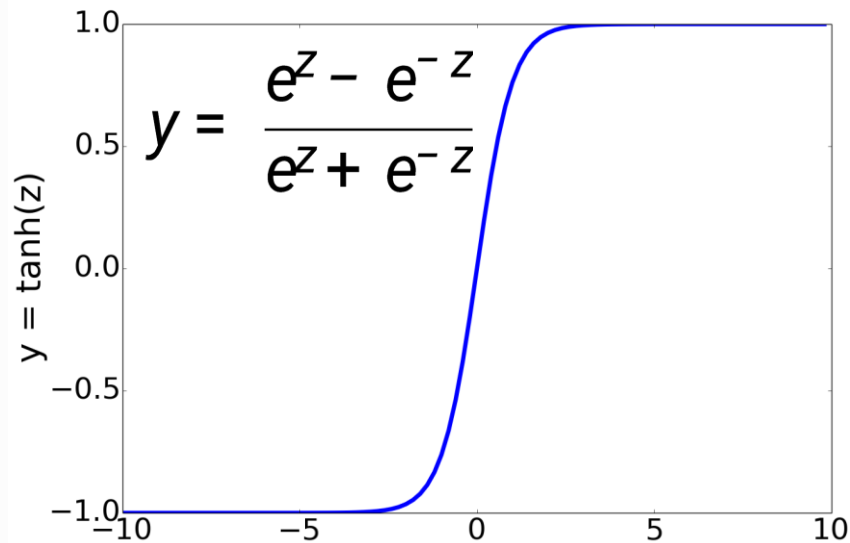
We're already seen the sigmoid for logistic regression:

Sigmoid

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

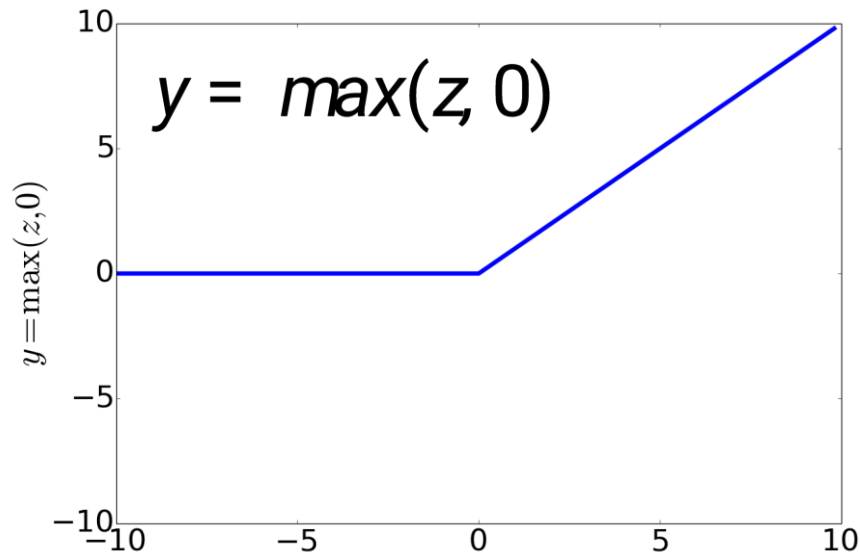


Non-Linear Activation Functions besides sigmoid



tanh

Most Common:



ReLU

Rectified Linear Unit

A little linear algebra

So Far, We Have Assume You Know Dot Products

$$\mathbf{a} = (a_1, a_2, a_3)$$

$$\mathbf{b} = (b_1, b_2, b_3)$$

$$\mathbf{a} \cdot \mathbf{b} = a_1b_1 + a_2b_2 + a_3b_3$$

A matrix is an array of numbers

$$\begin{bmatrix} 6 & 4 & 24 \\ 1 & -9 & 8 \end{bmatrix}$$

Two rows, three columns.

It's Easy to Multiply a Matrix by a Scalar

$$2 \cdot \begin{bmatrix} 5 & 2 \\ 3 & 1 \end{bmatrix} = \begin{bmatrix} 2 \cdot 5 & 2 \cdot 2 \\ 2 \cdot 3 & 2 \cdot 1 \end{bmatrix} = \begin{bmatrix} 10 & 4 \\ 6 & 2 \end{bmatrix}$$

Multiplying Matrices by Matrices Is Slightly Trickier

Let a_1 and a_2 be the row vectors of matrix A and b_1 and b_2 be the column vectors of a matrix B . Find $C = AB$

$$\begin{bmatrix} 1 & 7 \\ 2 & 4 \end{bmatrix} \cdot \begin{bmatrix} 3 & 3 \\ 5 & 2 \end{bmatrix} = \begin{bmatrix} a_1 \cdot b_1 & a_1 \cdot b_2 \\ a_2 \cdot b_1 & a_2 \cdot b_2 \end{bmatrix} = \begin{bmatrix} 38 & 17 \\ 26 & 14 \end{bmatrix}$$

A must have the same number of rows as B has columns.

Multiplying a Matrix by a Vector Is Roughly the Same

Multiplying a matrix by a vector is like multiply a matrix by a matrix with one column:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

The result is a vector.

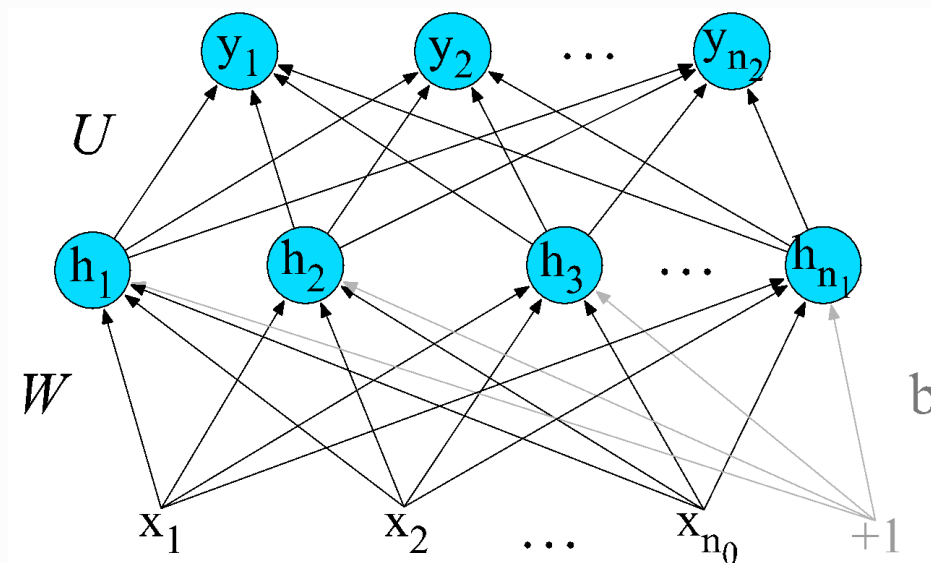
Matrix multiplication is not hard but inference with neural nets is mostly this (plus some non-linear functions)

Feedforward neural networks

Adding multiple units to a neural network increases its power to learn patterns in data. **Feedforward Neural Nets (FFNNs or MLPs)**

Feedforward Neural Networks

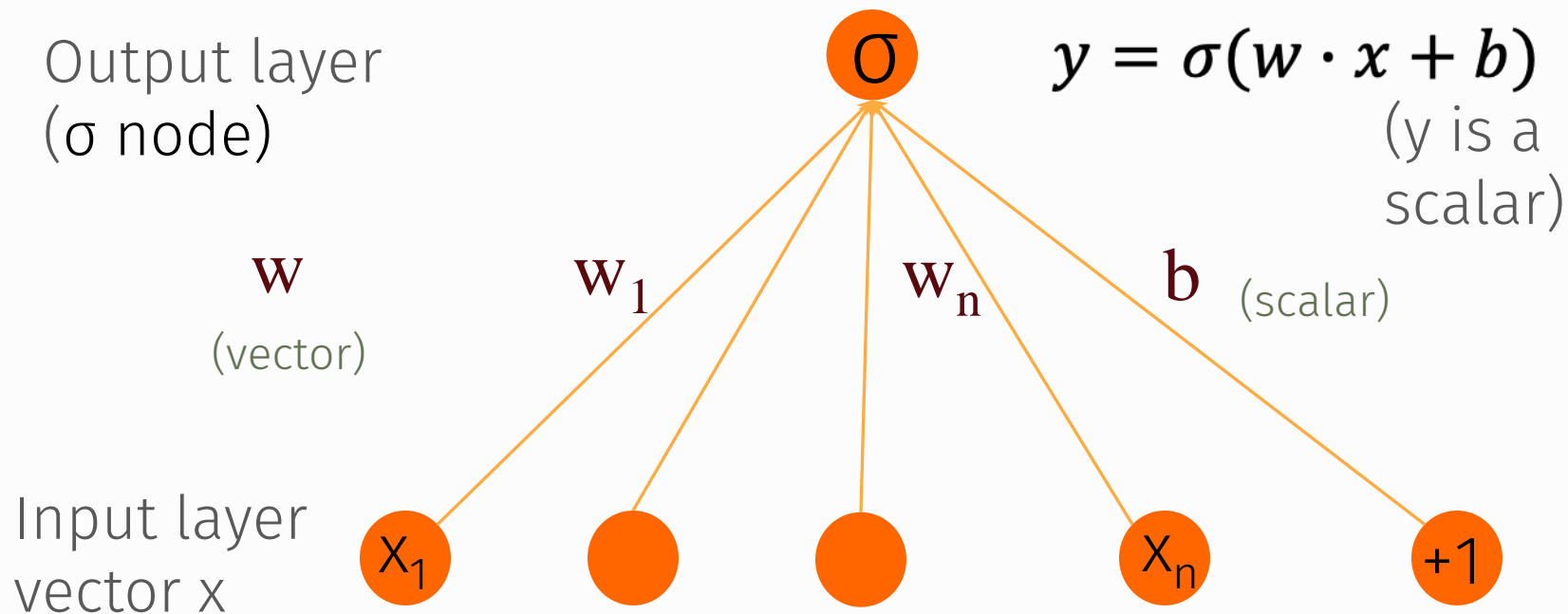
Can also be called **multi-layer perceptrons** (or **MLPs**) for historical reasons



The simplest FFNN is just binary logistic regression
(INPUT LAYER = feature vector)

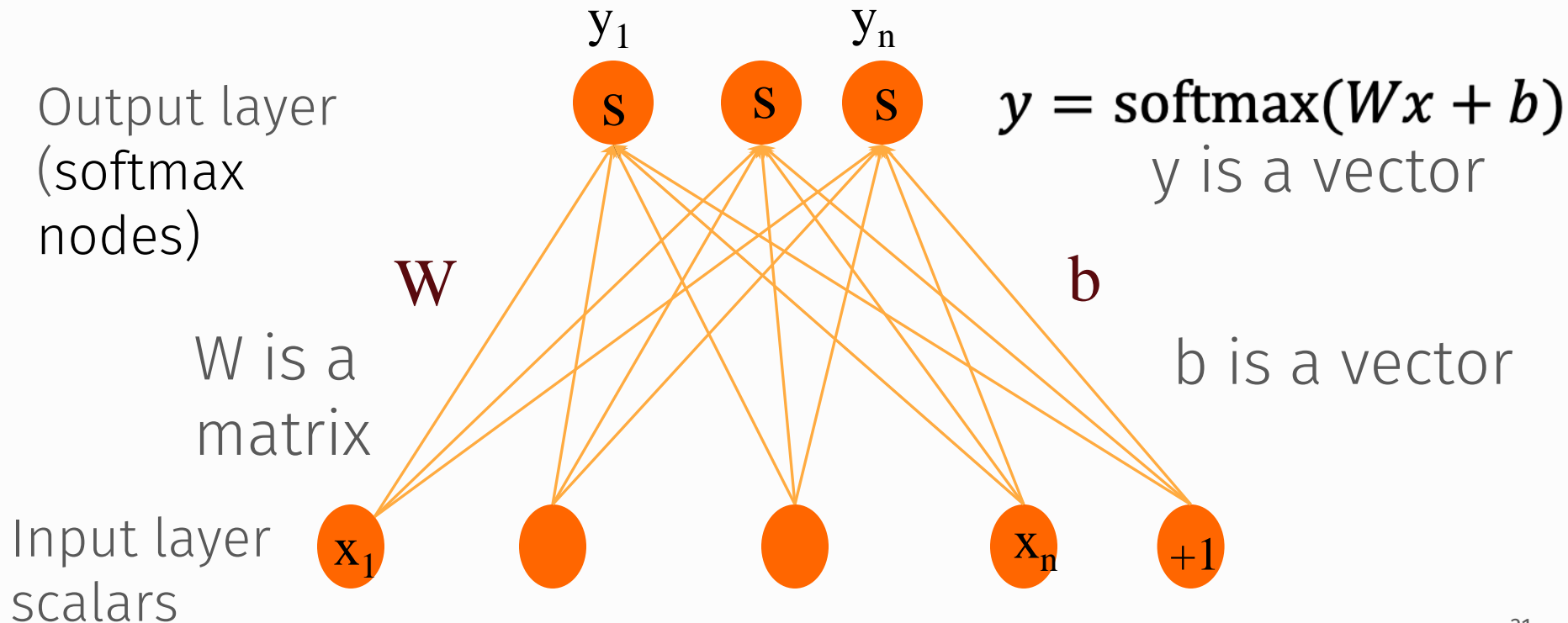
Binary Logistic Regression as a 1-layer Network

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



Multinomial Logistic Regression as a 1-layer Network

Fully connected single layer network



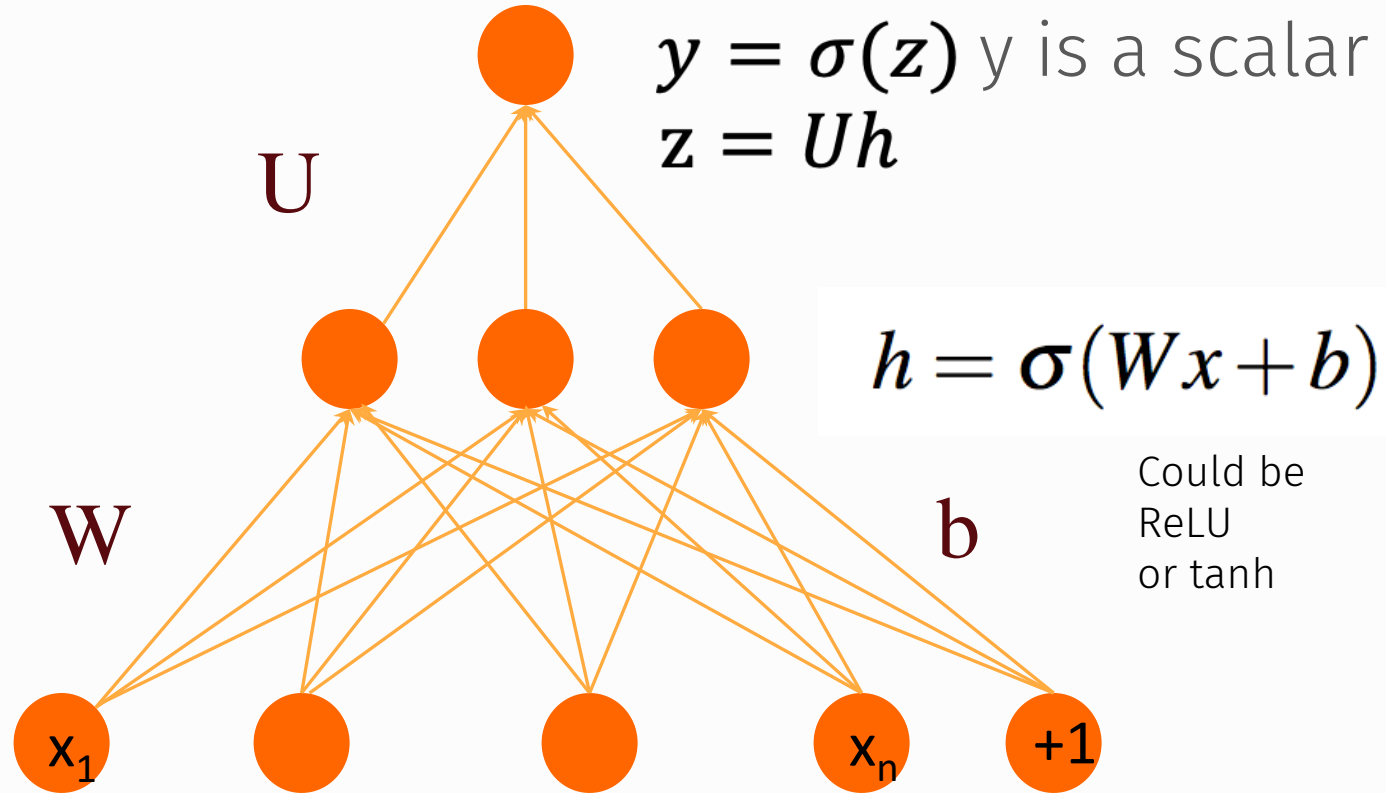
The real power comes when multiple layers are added

Two-Layer Network with scalar output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)

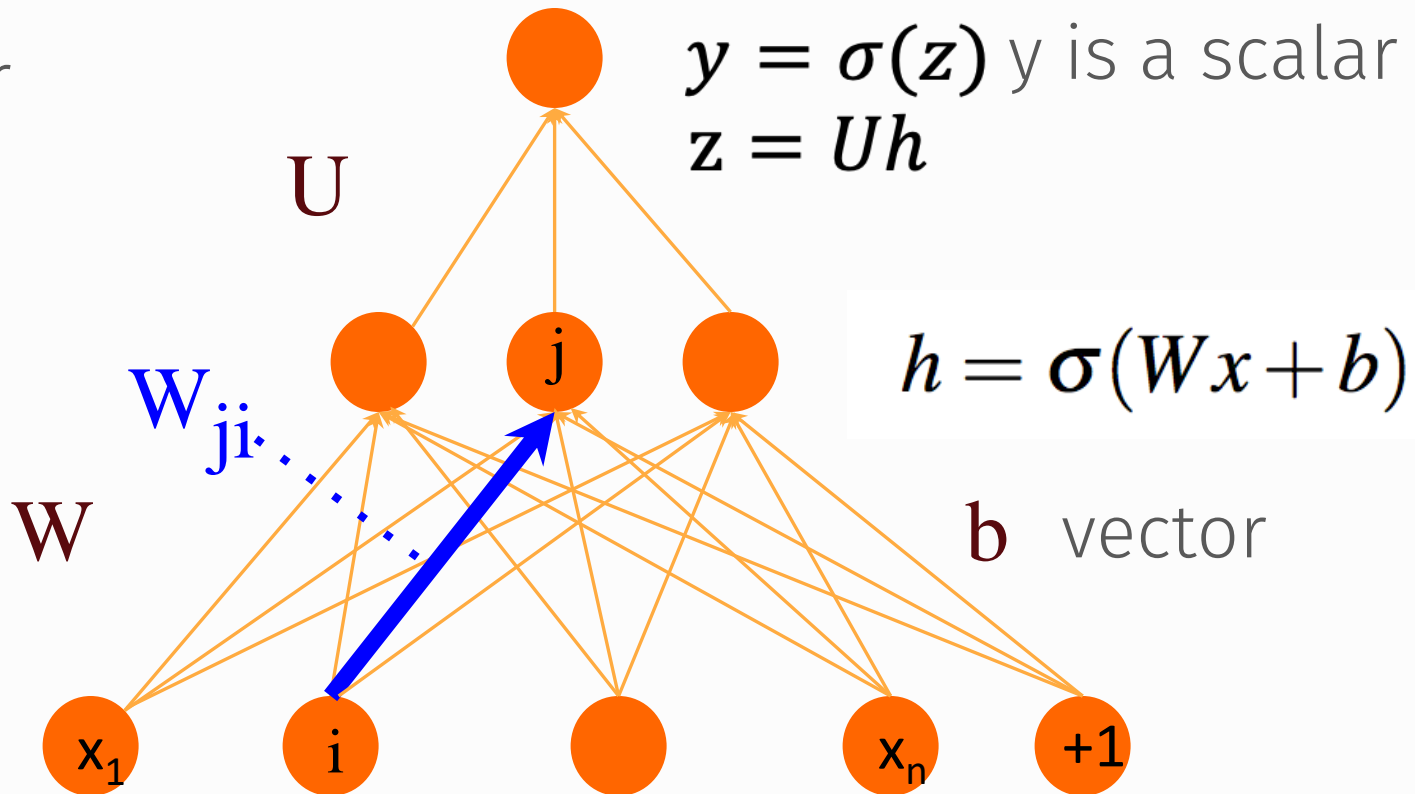


Two-Layer Network with scalar output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)

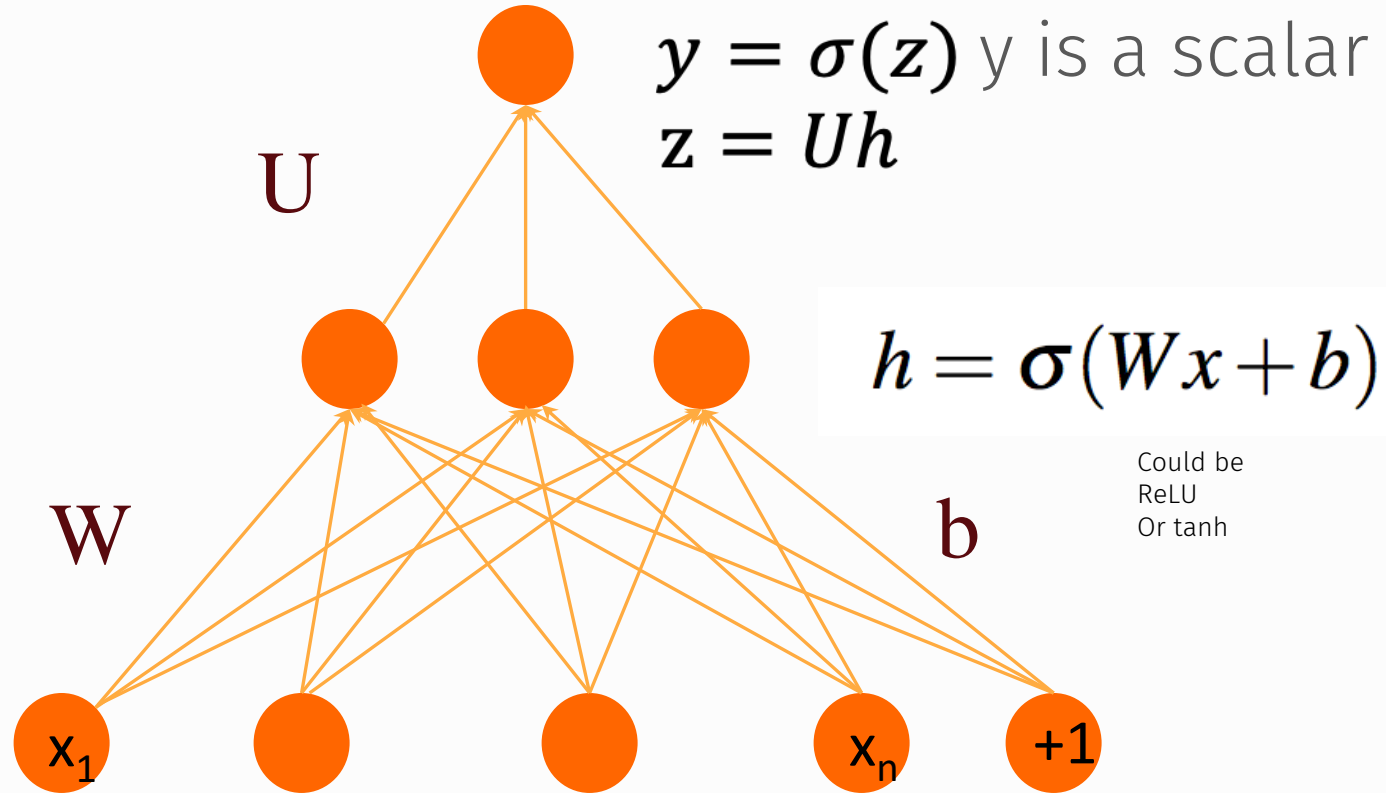


Two-Layer Network with scalar output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)

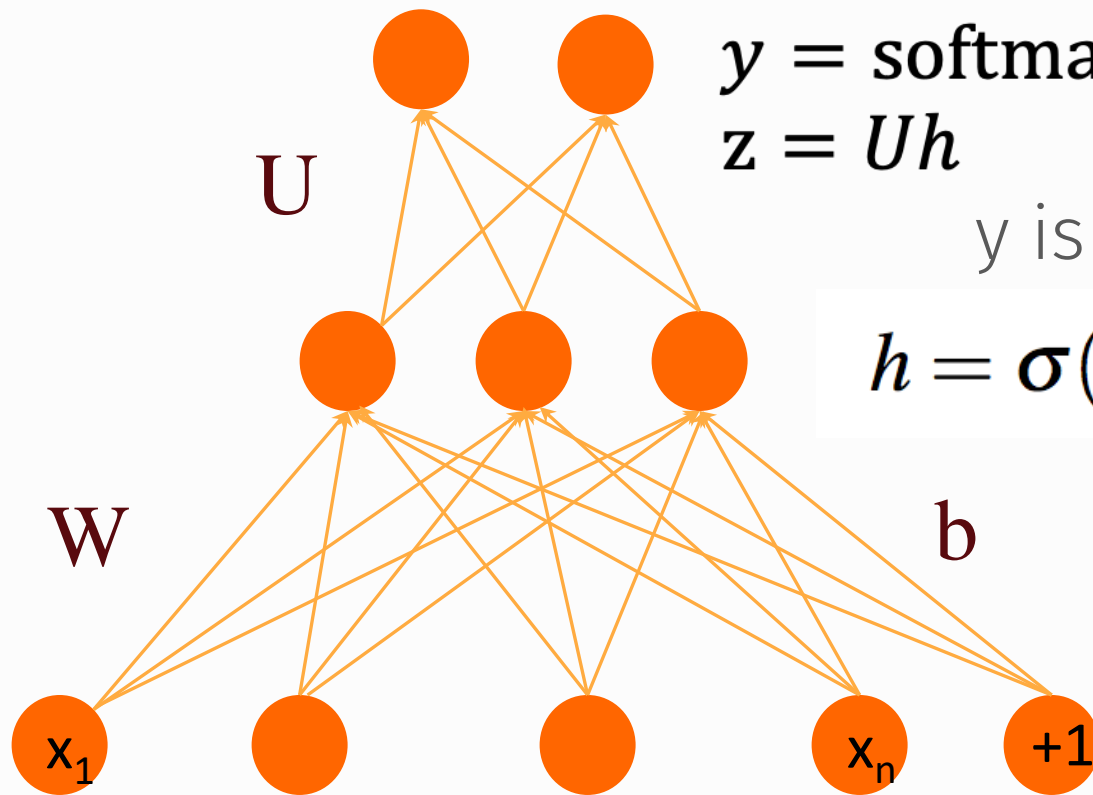


Two-Layer Network with softmax output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)



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

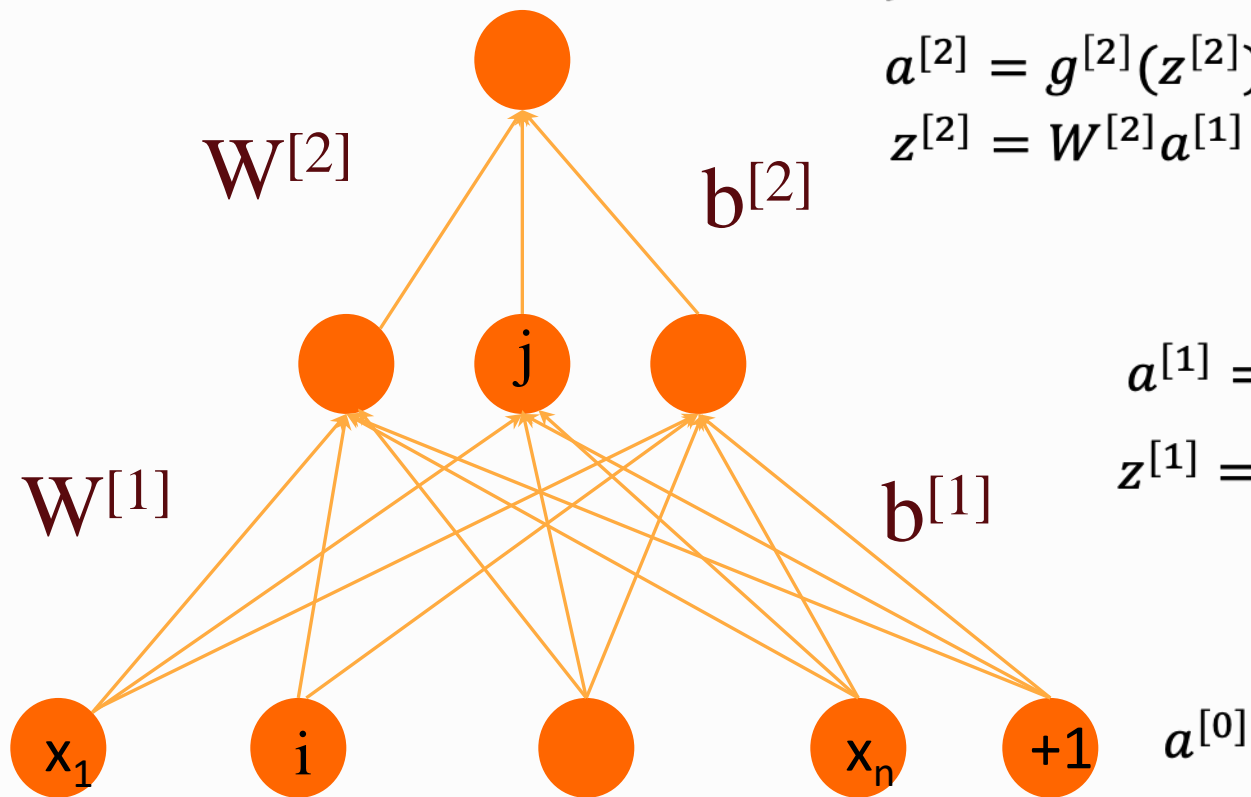
$$z = Uh$$

y is a vector

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

Could be
ReLU
Or tanh

Multi-layer Notation



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

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

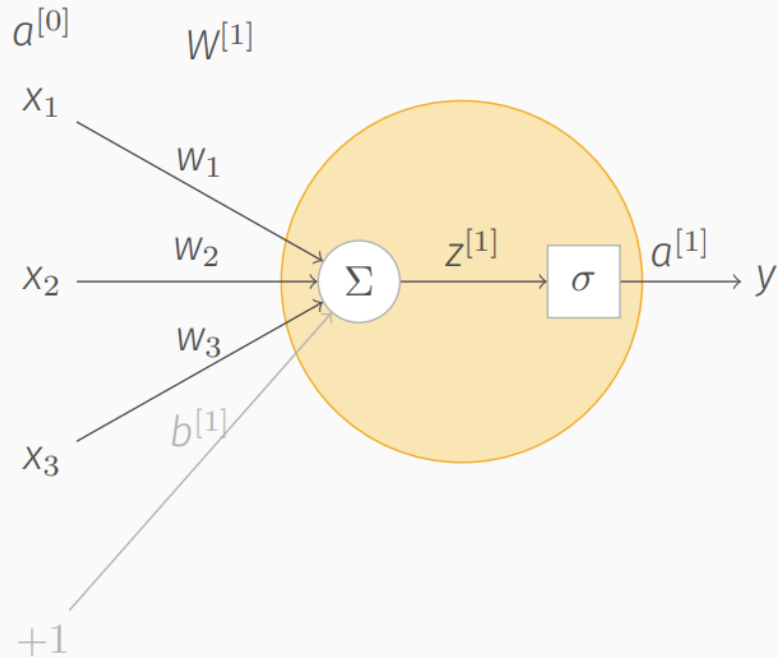
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

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

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

$$a^{[0]}$$

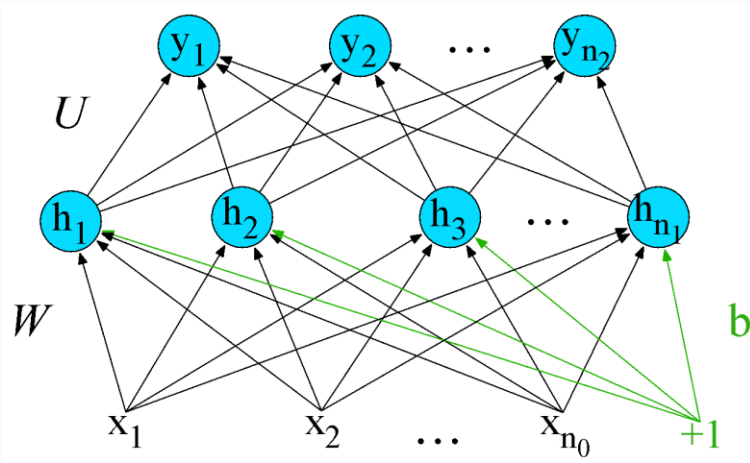
A Forward Pass in Terms of Multi-Layer Notation



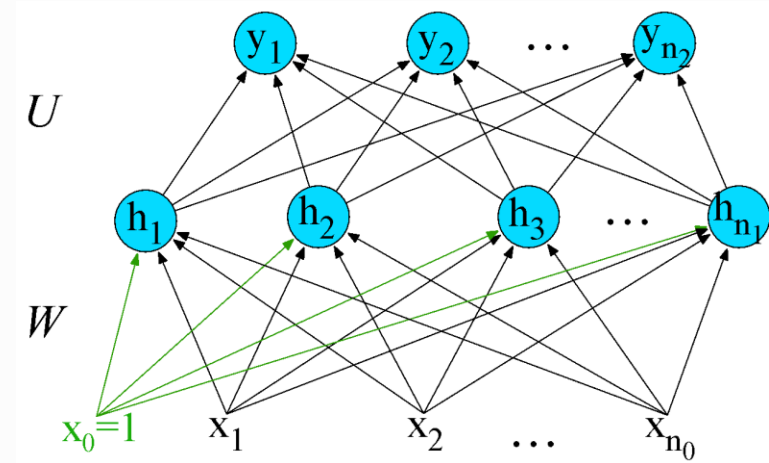
```
for each  $i \in 1..n$  do  
     $z^{[i]} \leftarrow W^{[i]}a^{[i-1]} + b^{[i]}$   
     $a^{[i]} \leftarrow g^{[i]}(z^{[i]})$   
end for  
 $\hat{y} \leftarrow a^{[n]}$ 
```

Replacing the bias unit

Instead of:



We'll do this:



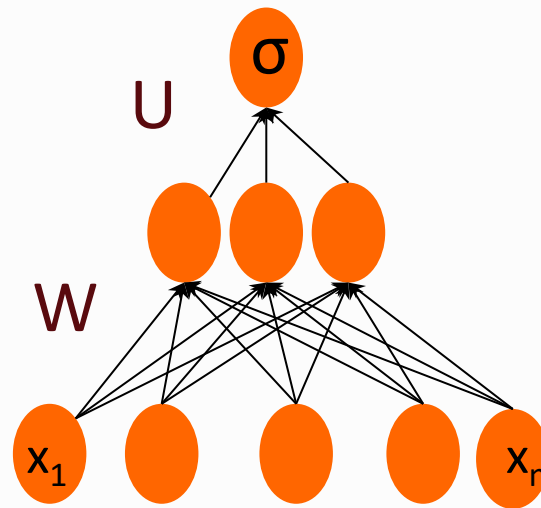
Feedforward neural nets as classifiers

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

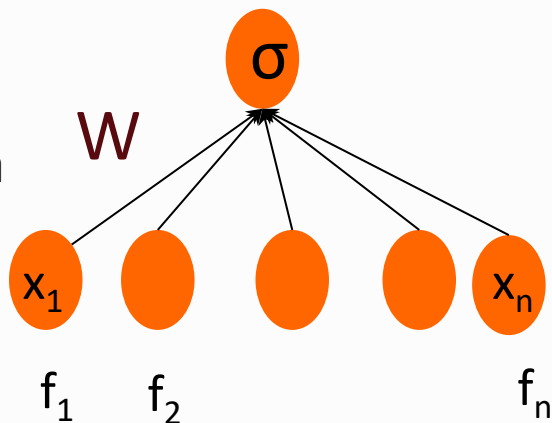


Sentiment Features

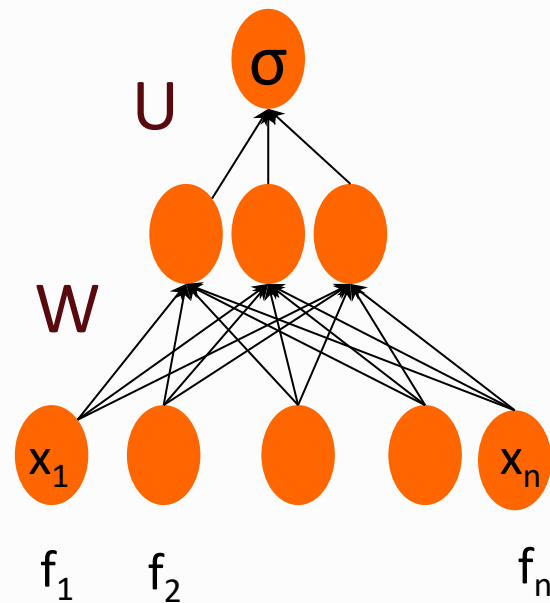
Var	Definition
x_1	$\text{count}(\text{positive lexicon}) \in \text{doc}$
x_2	$\text{count}(\text{negative lexicon}) \in \text{doc}$
x_3	$\begin{cases} 1 & \text{if "no"} \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$
x_4	$\text{count}(\text{1st and 2nd pronouns}) \in \text{doc}$
x_5	$\begin{cases}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

Feedforward nets for simple classification

Logistic
Regression



2-layer
feedforward
network



Just adding a hidden layer to logistic regression

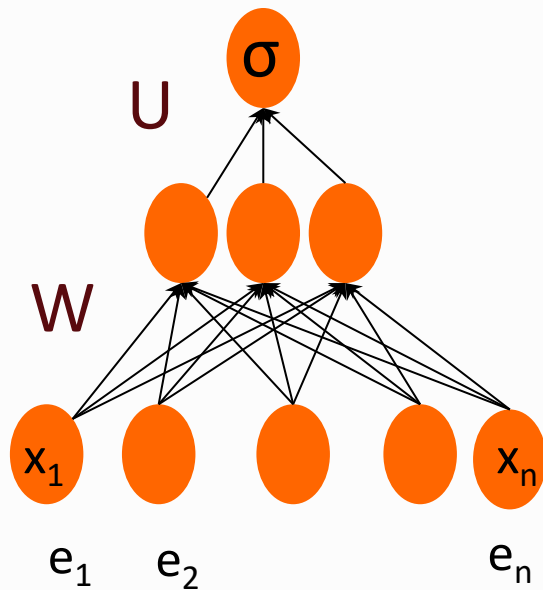
- allows the network to use non-linear interactions between features
- which may (or may not) improve performance.

Even better: representation learning

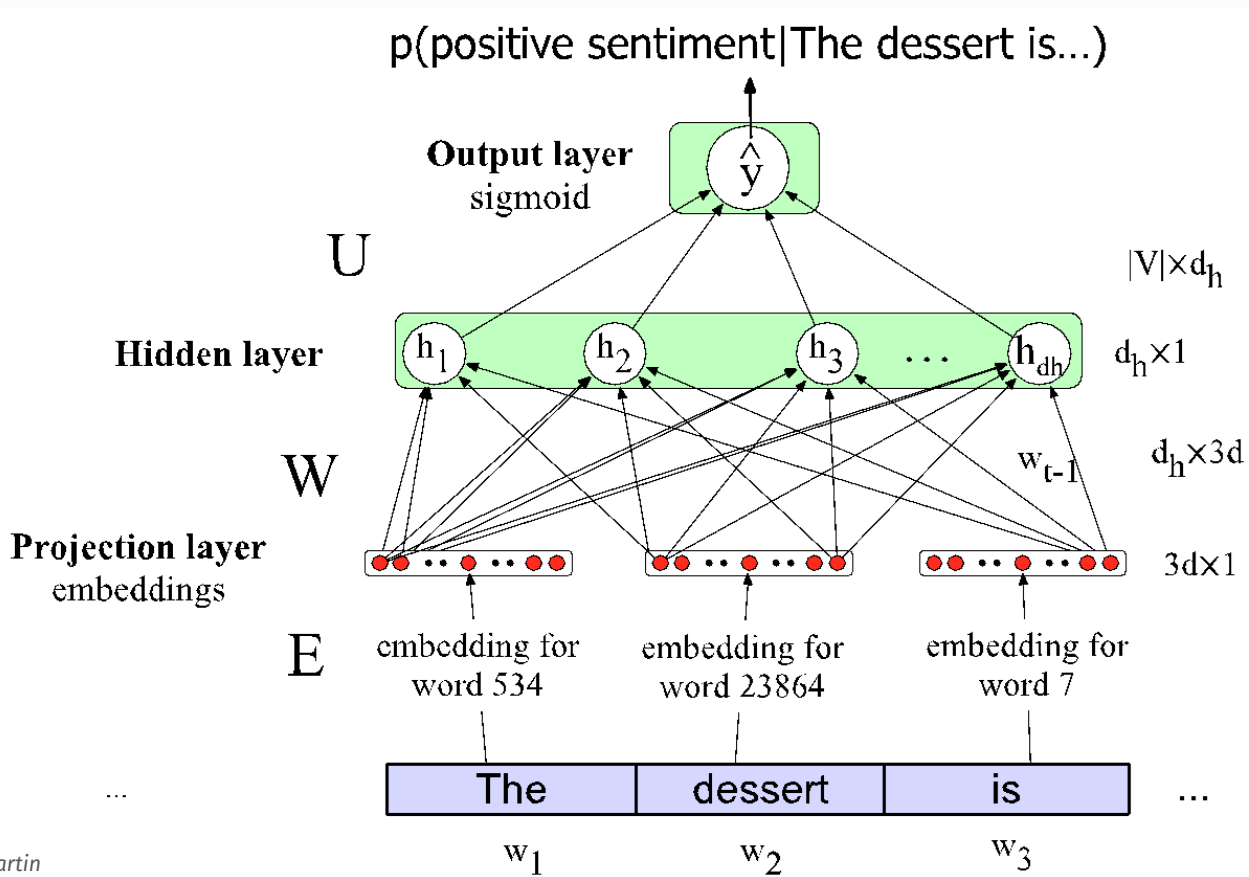
The real power of deep learning comes from the ability to **learn** features from the data

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

Use learned representations like embeddings!



Neural net classification with embeddings as input features!



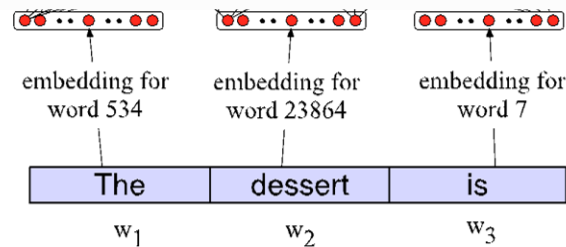
Issue: texts come in different sizes

This assumes a fixed size length (3)!

Kind of unrealistic.

Some simple solutions:

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

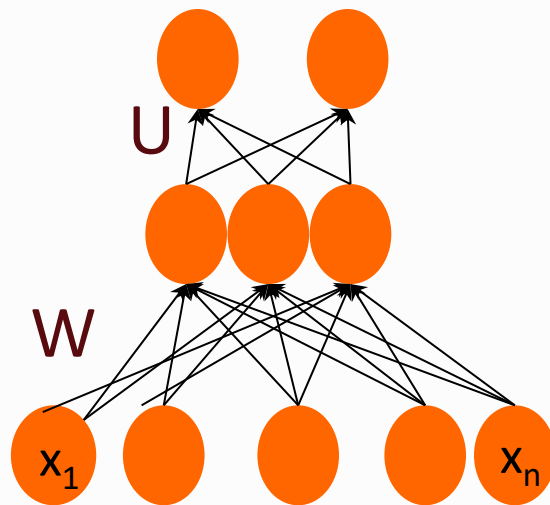


Reminder: Multiclass Outputs

What if you have more than two output classes?

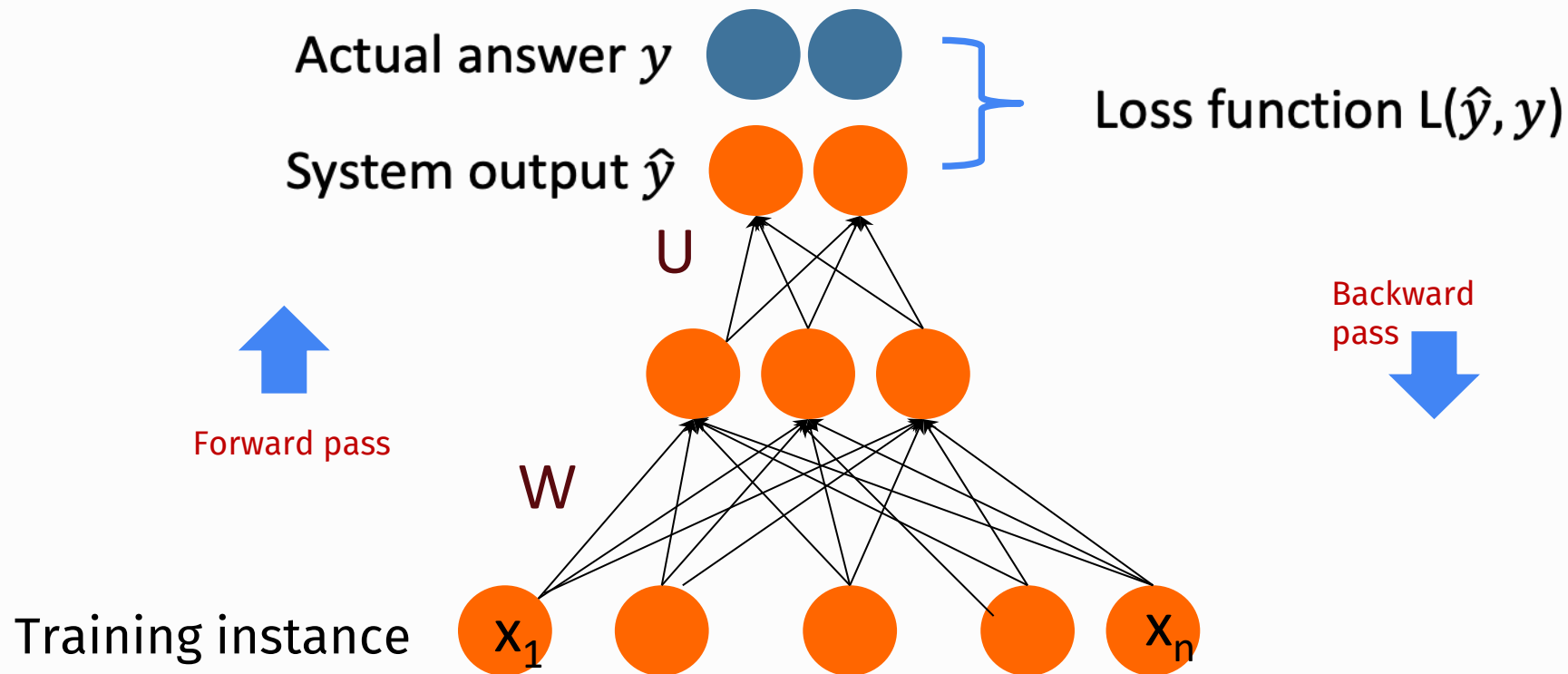
- Add more output units (one for each class)
- And use a “softmax layer”

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



Training feedforward neural networks

Intuition: training a 2-layer Network



Remember stochastic gradient descent
from the logistic regression lecture—find
gradient and optimize

The Intuition Behind Training a 2-Layer Network

For every training tuple (x, y)

1. Run **forward** computation to find the estimate \hat{y}
2. Run **backward** computation to update weights
 - For every output node
 - Compute the loss L between true y and estimated \hat{y}
 - For every weight w from the hidden layer to the output layer: update the weights
 - For every hidden node
 - Assess how much blame it deserves for the current answer
 - From every weight w from the input layer to the hidden layer
 - Update the weight

Computing the gradient requires finding the derivative of the loss with respect to each weight in every layer of the network.

Error backpropagation through computation graphs.

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

$$w_{t+1} = w_t - \eta \frac{d}{dw} L_{CE}(f(x; w), y)$$

- For logistic regression

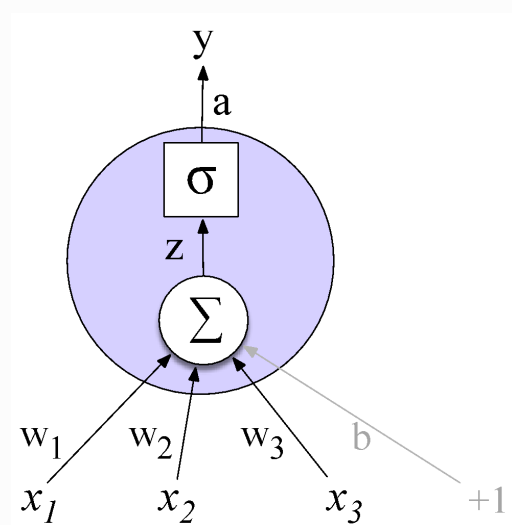
$$\frac{\partial L_{CE}(\hat{y}, y)}{\partial w_j} = [\sigma(w \cdot x + b) - y] x_j$$

Where did that derivative come from?

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

Intuition (see the text for details)

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



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:

- Even more use of the chain rule!!
- Computation graphs and error backpropagation!

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)

- Relies on **computation graphs**

Computation Graphs

A computation graph represents the process of computing a mathematical expression

Example:

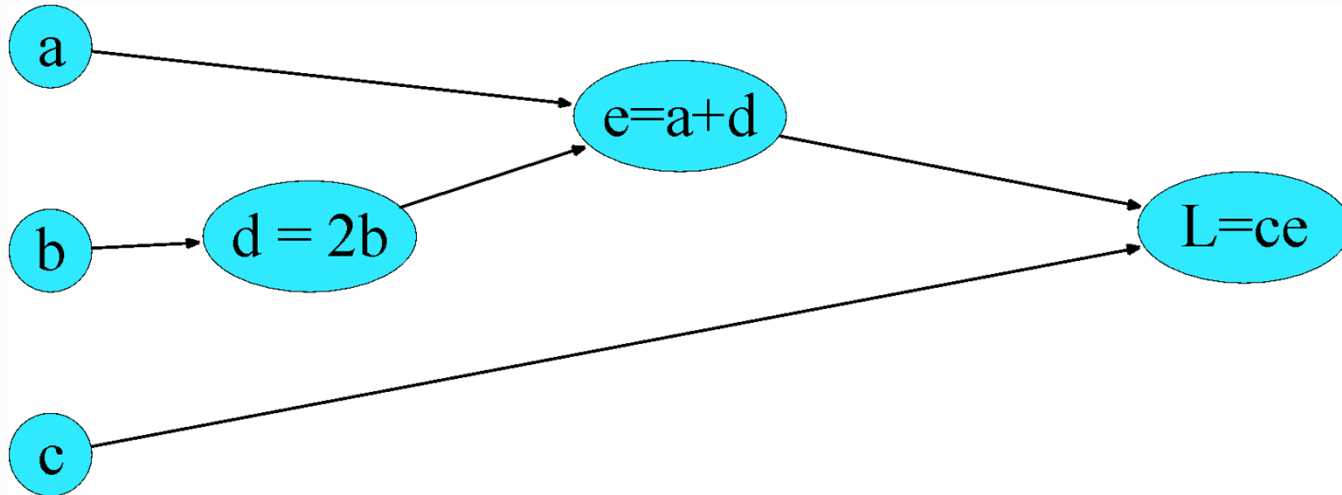
$$L(a, b, c) = c(a + 2b)$$

Computations:

$$d = 2 * b$$

$$e = a + d$$

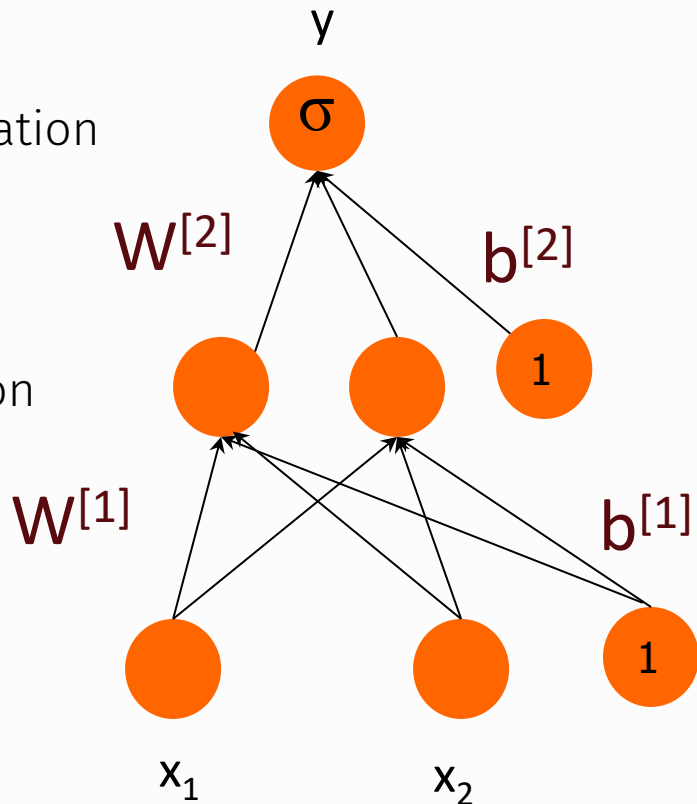
$$L = c * e$$



Backward differentiation on a two layer network

Sigmoid activation

ReLU activation



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

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

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

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

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

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: **backpropagation**

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.

Questions?