CS 2731
Introduction to Natural Language Processing

Session 8: Feedforward neural networks

Michael Miller Yoder

September 23, 2024

School of Computing and Information

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 post on Blodgett et al. 2020

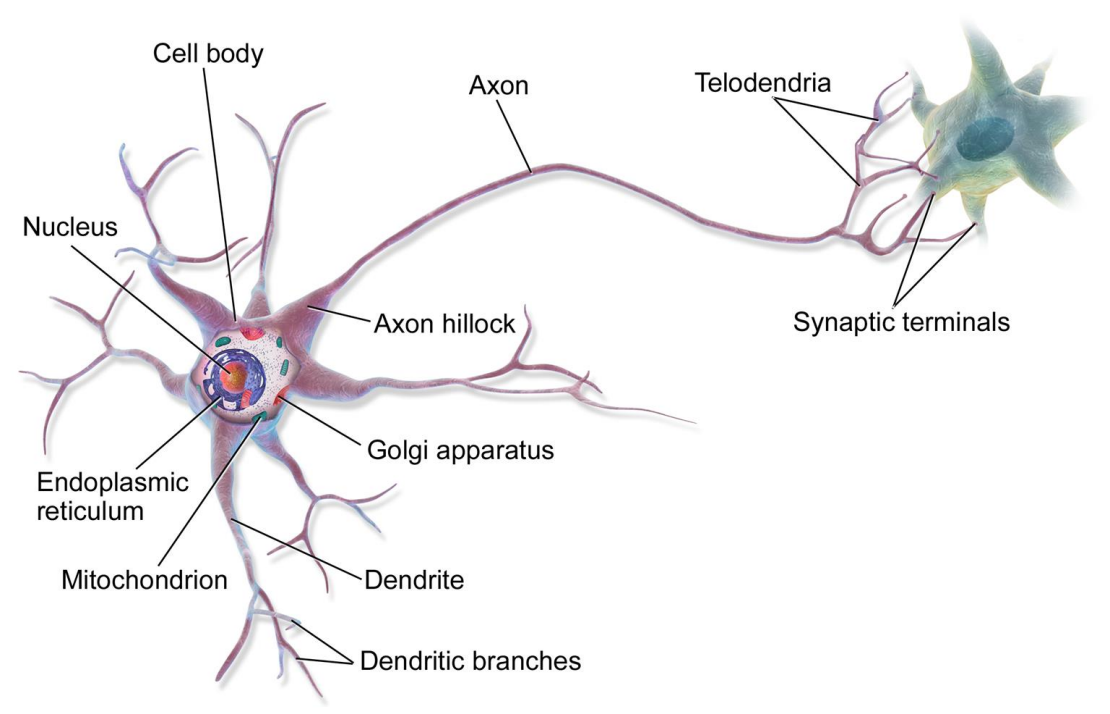
- Recommendations from Blodgett et al. for better work on bias
  1. Ground work analyzing bias in relevant literature outside of NLP that explores relationships between language and social hierarchies. Treat representational harms as harmful in their own right
  2. Explicitly state why “bias” in systems 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.

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

Slide adapted from Jurafsky & Martin

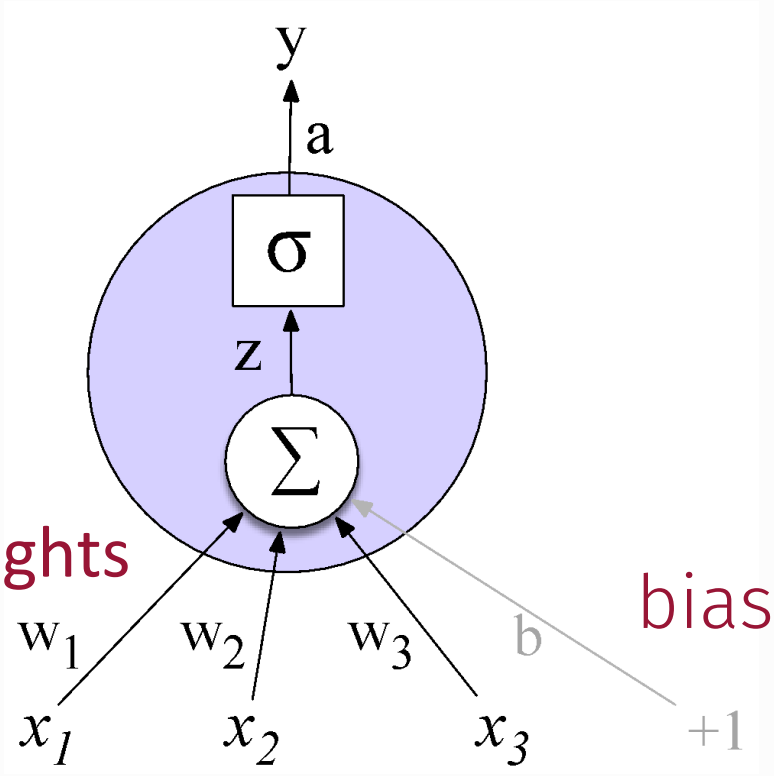
Neural Network Unit: This is not in your brain

Output value

Non-linear transform

Weighted sum

Input layer



Slide adapted from Jurafsky & Martin

The Variables in Our Very Important Formula

- $\mathbf{x}$ A vector of features of n dimensions (like number of positive sentiment words, length of document, etc.)
- $\mathbf{w}$ 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

Slide adapted from David Mortensen

The Fundamentals

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

$$z = b + \sum_i w_i x_i \tag{1}$$

Which we will represent as

$$z = \mathbf{w} \cdot \mathbf{x} + b \tag{2}$$

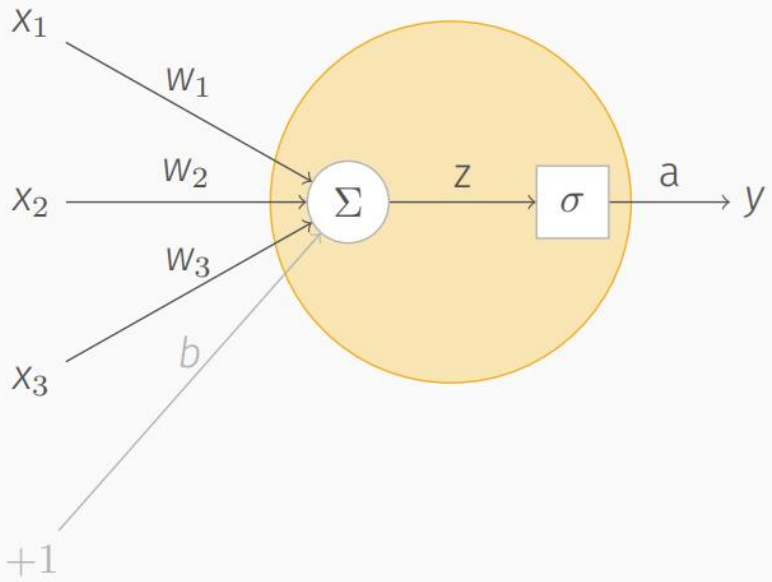
But we do not use z directly. Instead, we pass it through a non-linear function, like the sigmoid function:

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

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

Slide adapted from David Mortensen

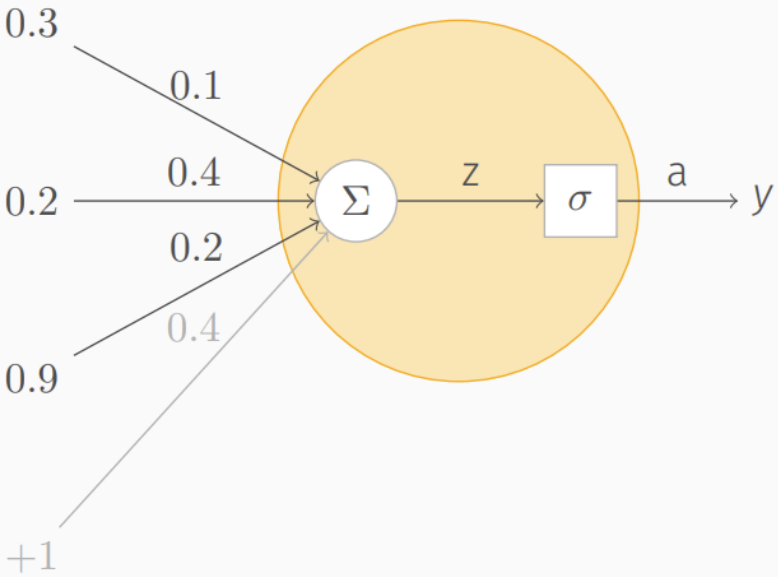
A Unit Illustrated



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].

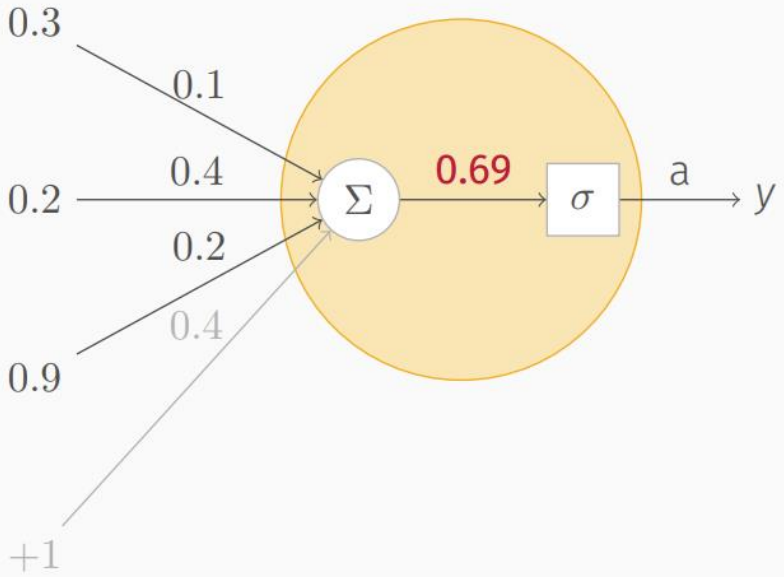
Slide adapted from David Mortensen

Filling in the Input Values and Weights



Slide adapted from David Mortensen

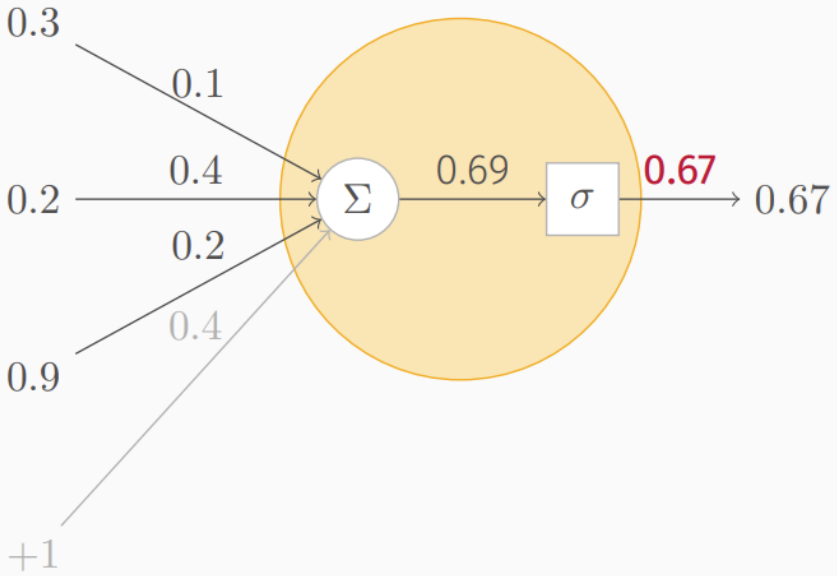
Multiplying the Input Values and Weights and Summing Them (with the Bias Term)



$$z = x_1 w_1 + x_2 w_2 + x_3 w_3 + b = 0.1(0.3) + 0.4(0.2) + 0.2(0.9) + 0.4 = 0.69 \quad (4)$$

Slide adapted from David Mortensen

Applying the Activation Function (Sigmoid)



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

Slide adapted from David Mortensen

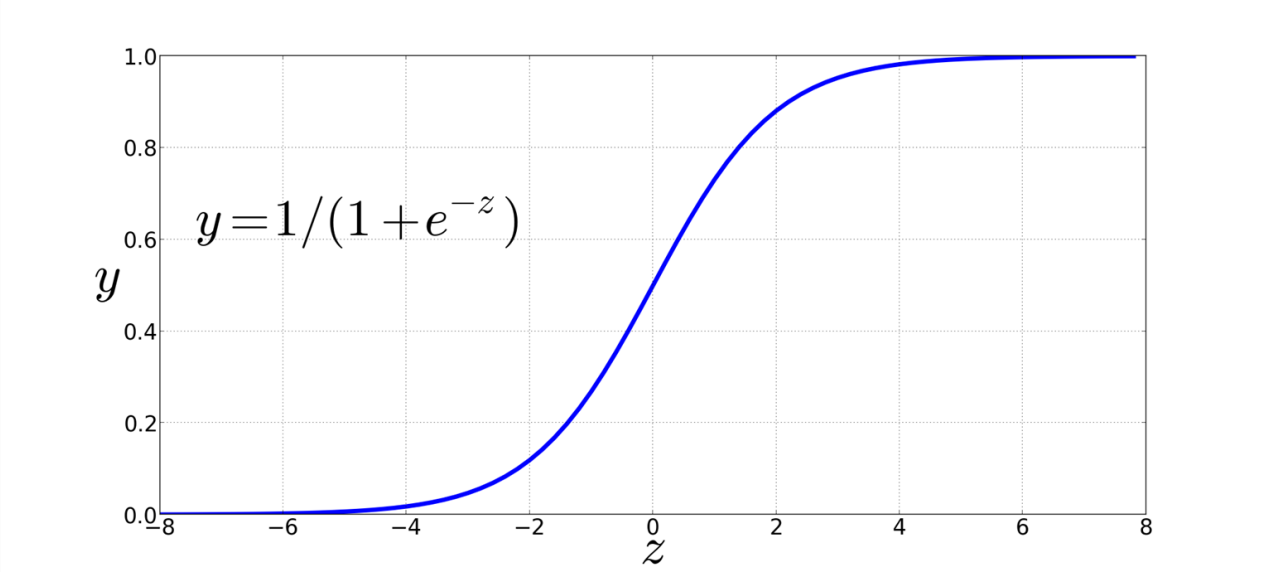
Non-linear activation functions

Non-Linear Activation Functions

We're already seen the sigmoid for logistic regression:

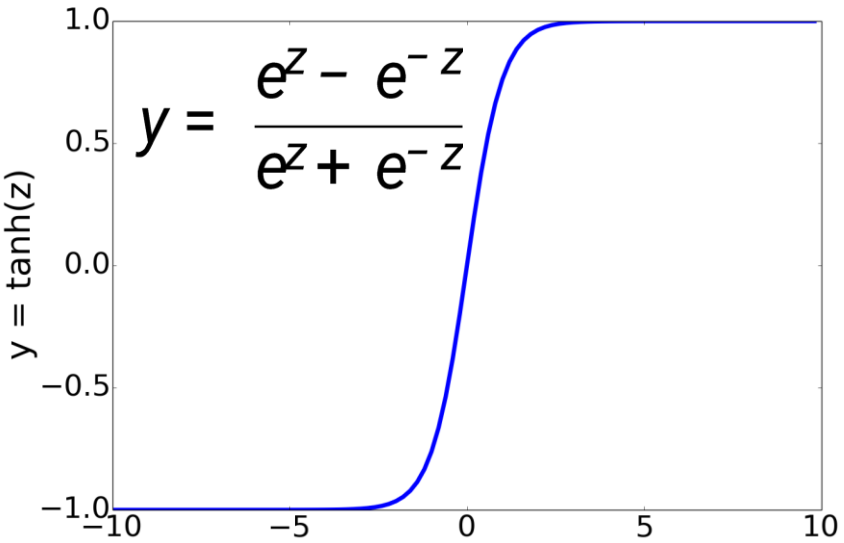
Sigmoid

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



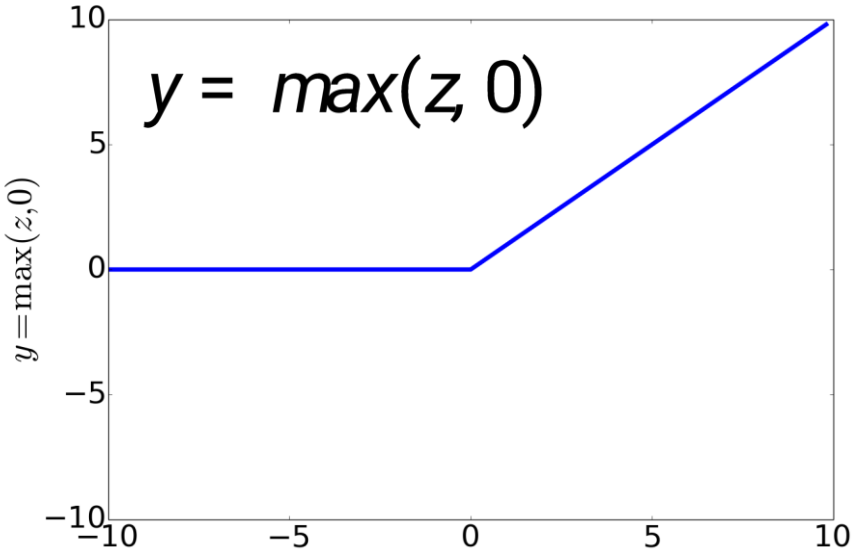
Slide adapted from Jurafsky & Martin

Non-Linear Activation Functions besides sigmoid



tanh

Most Common:



ReLU

Rectified Linear Unit

Slide adapted from Jurafsky & Martin

A little linear algebra

So Far, We Have Assume You Know Dot Products

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

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

$$\boldsymbol{a} \cdot \boldsymbol{b} = a_1 b_1 + a_2 b_2 + a_3 b_3$$

Slide adapted from David Mortensen

Now, You Need to Multiply Matrices

A matrix is an array of numbers

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

Two rows, three columns.

Slide adapted from David Mortensen

It's Easy to Multiple a Matrix by a Scalar

$$2 \cdot \begin{bmatrix} 5 & 2 \\ 3 & 1 \end{bmatrix} = \begin{bmatrix} 2 \cdot 5 & 2 \cdot 2 \\ 2 \cdot 3 & 2 \cdot 1 \end{bmatrix} = \begin{bmatrix} 10 & 7 \\ 2 & 4 \end{bmatrix}$$

Slide adapted from David Mortensen

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.

Slide adapted from David Mortensen

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.

Slide adapted from David Mortensen

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

Slide adapted from David Mortensen

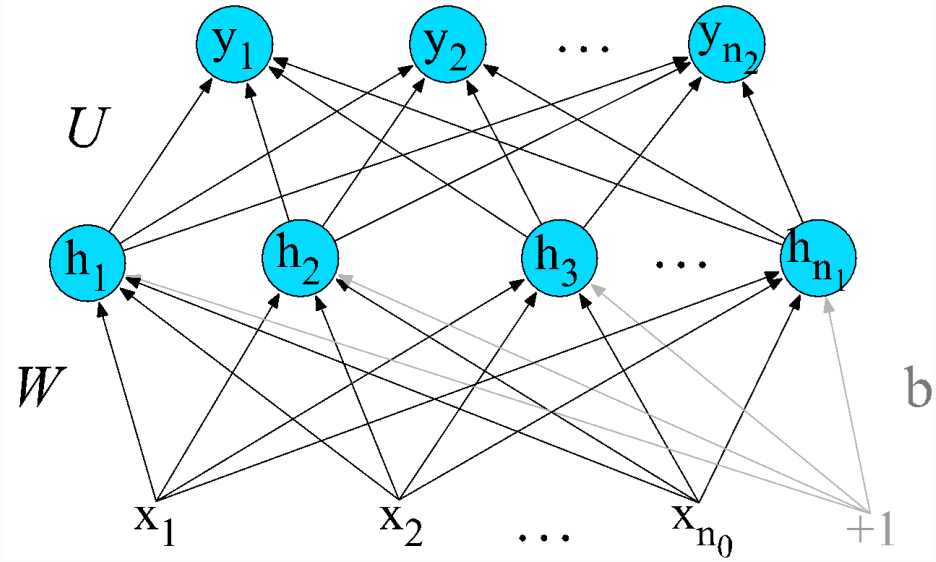
Feedforward neural networks

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

Slide adapted from David Mortensen

Feedforward Neural Networks

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



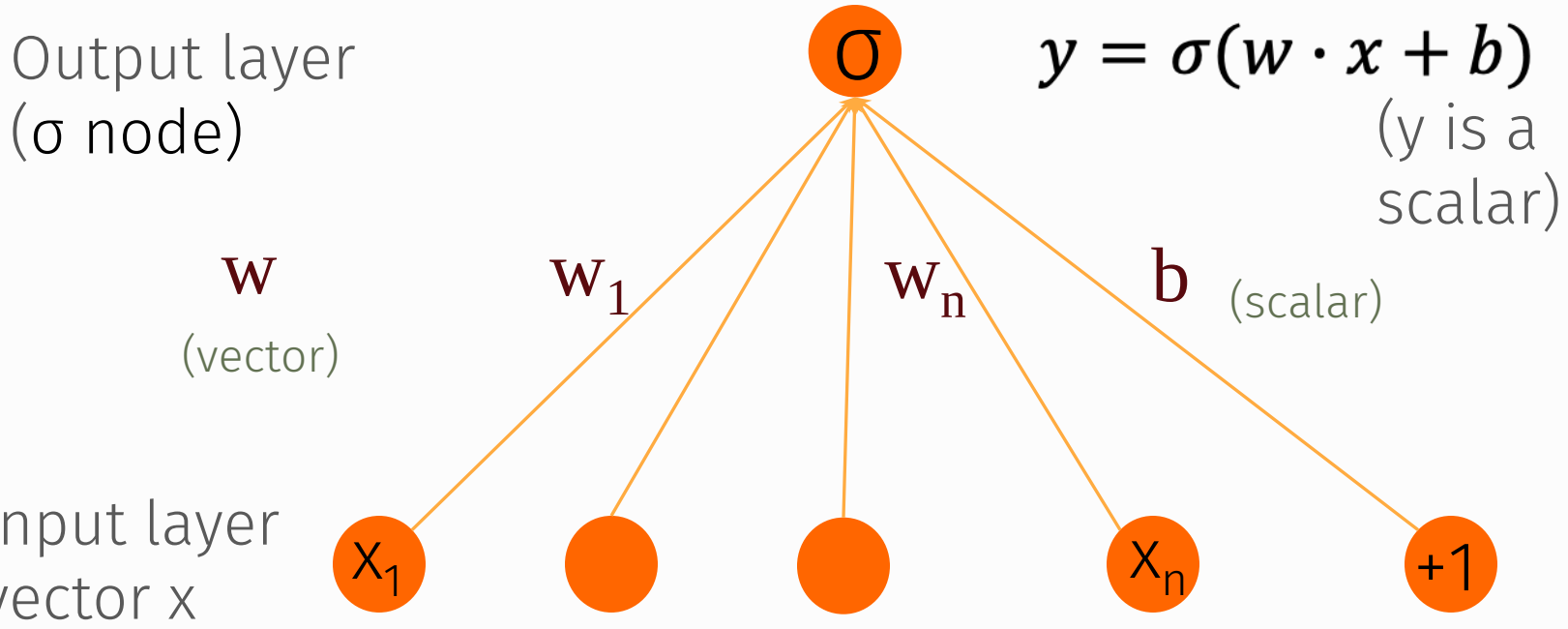
Slide adapted from Jurafsky & Martin

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

Slide adapted from David Mortensen

Binary Logistic Regression as a 1-layer Network

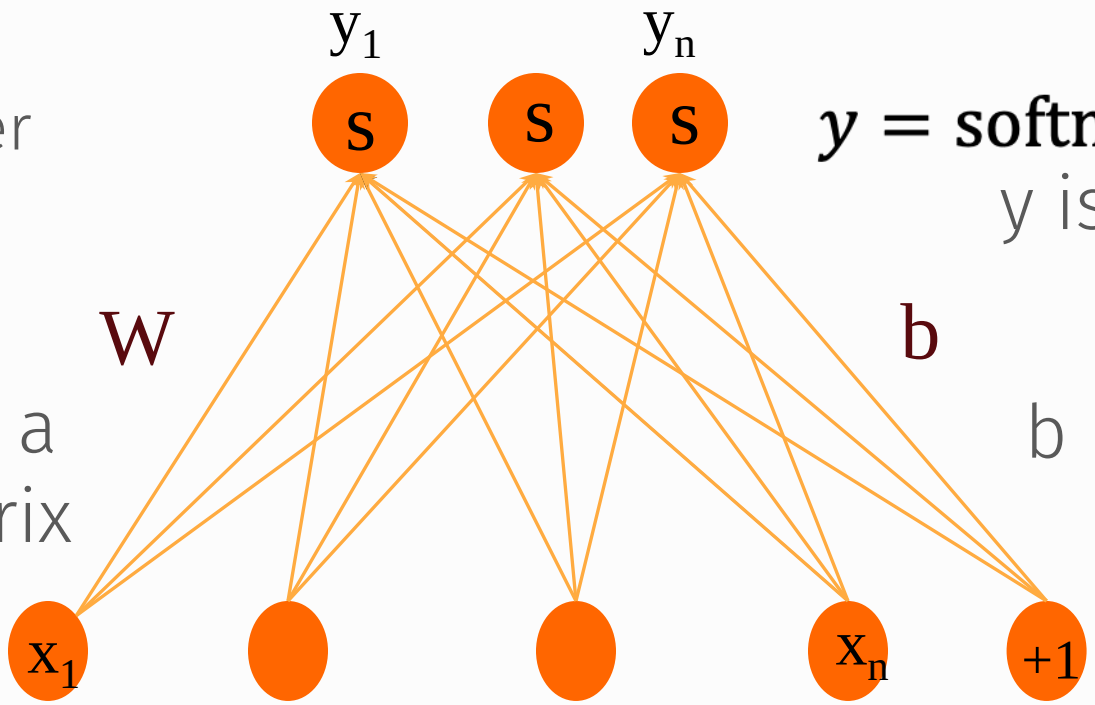
(we don't count the input layer in counting layers!)



Slide adapted from Jurafsky & Martin

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

Slide adapted from Jurafsky & Martin

The real power comes when multiple layers are added

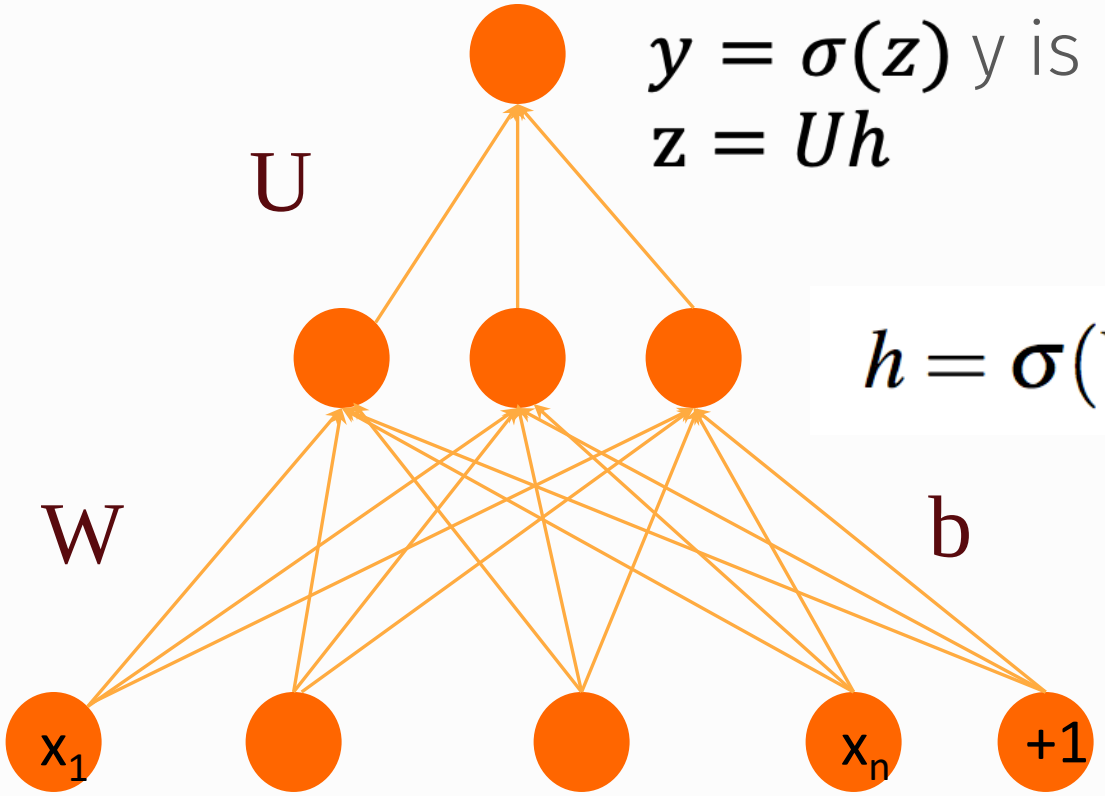
Slide adapted from David Mortensen

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

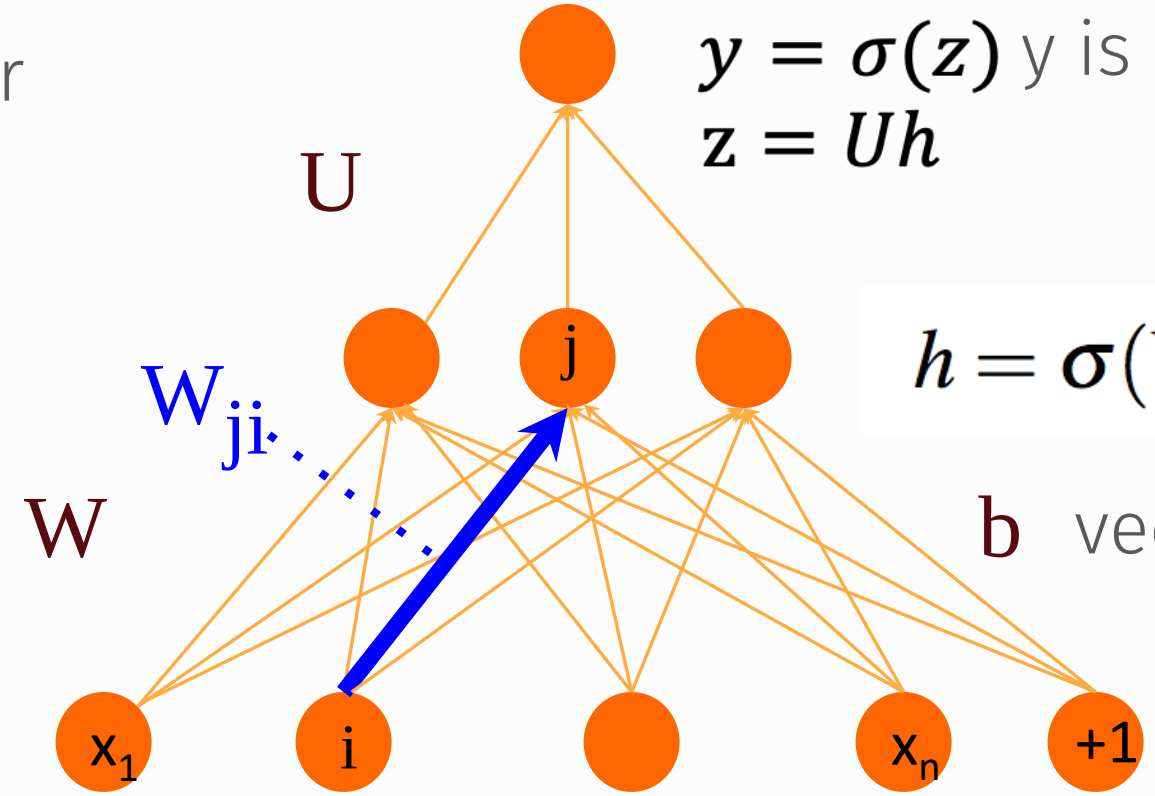
Slide adapted from Jurafsky & Martin

Two-Layer Network with scalar output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)



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

$z = Uh$

$h = \sigma(Wx + b)$

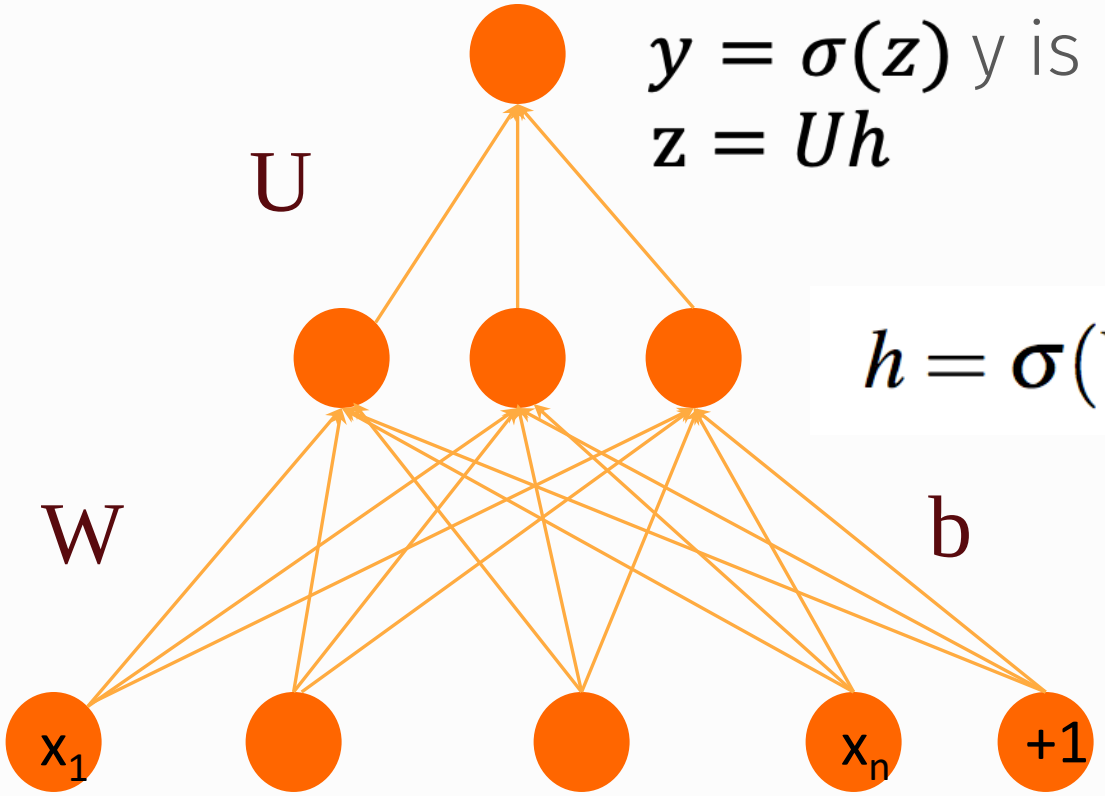
b vector

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

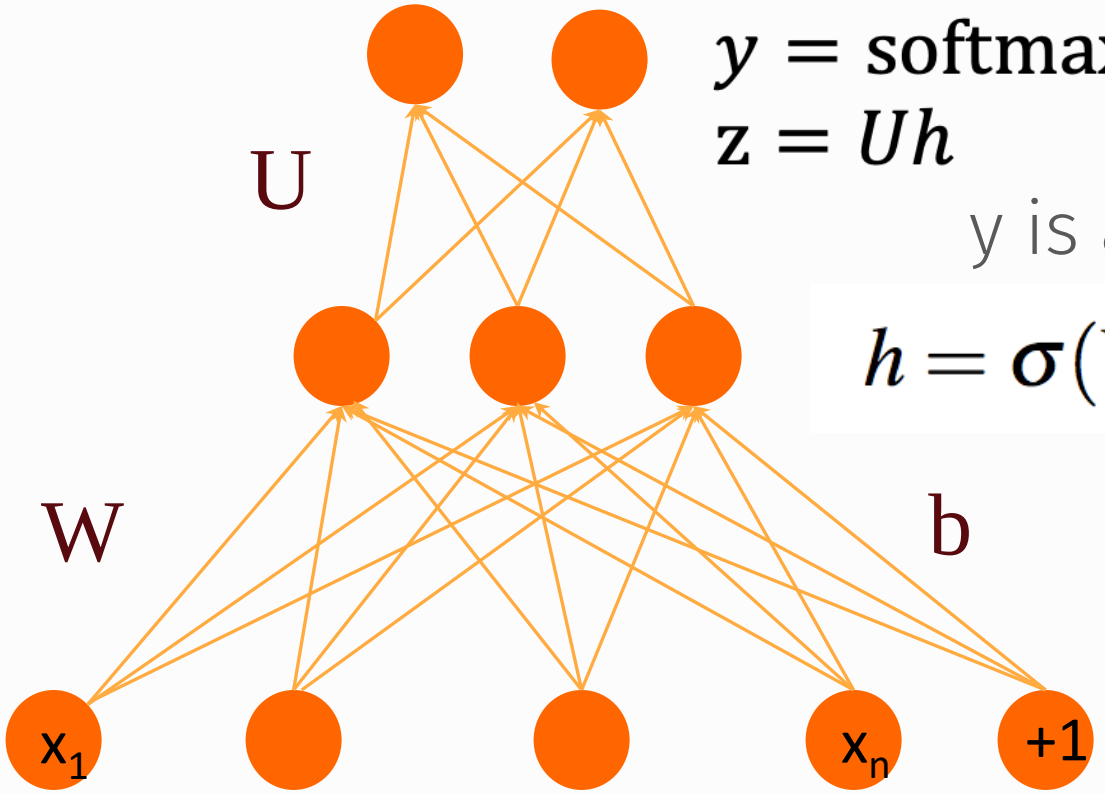
Slide adapted from Jurafsky & Martin

Two-Layer Network with softmax output

Output layer (σ node)

hidden units (σ node)

Input layer (vector)



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

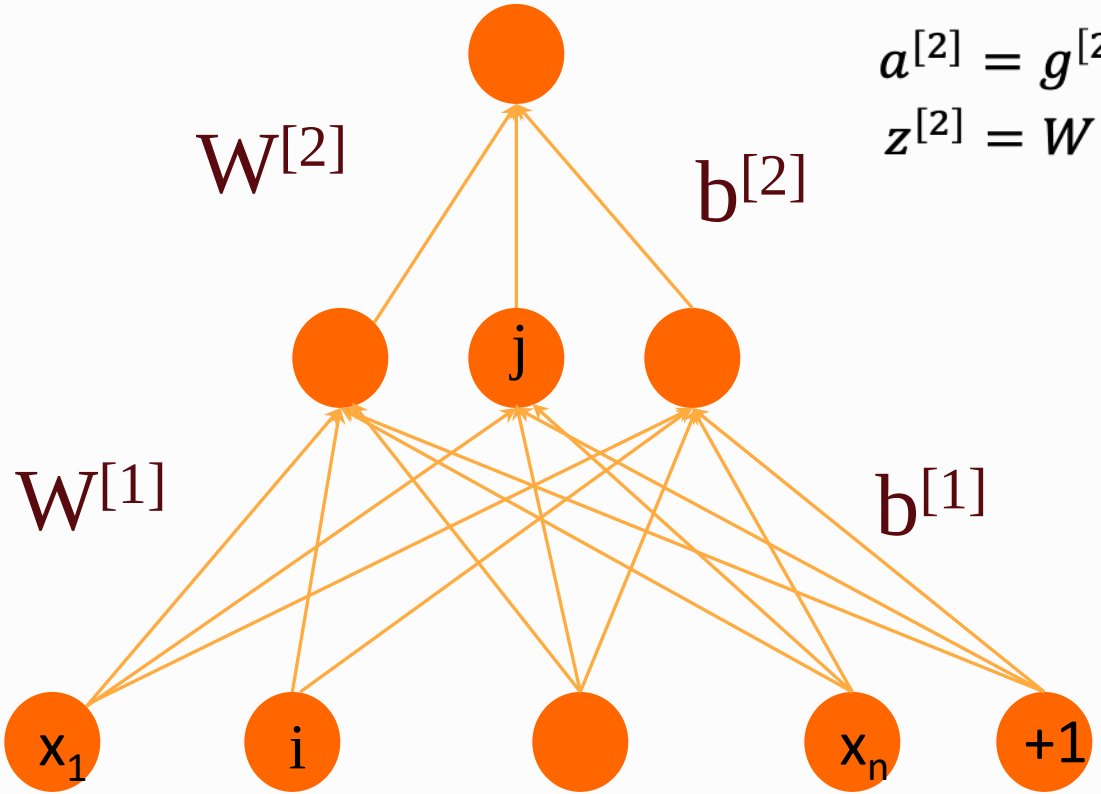
y is a vector

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

Could be
ReLU
Or tanh

Slide adapted from Jurafsky & Martin

Multi-layer Notation



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

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

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

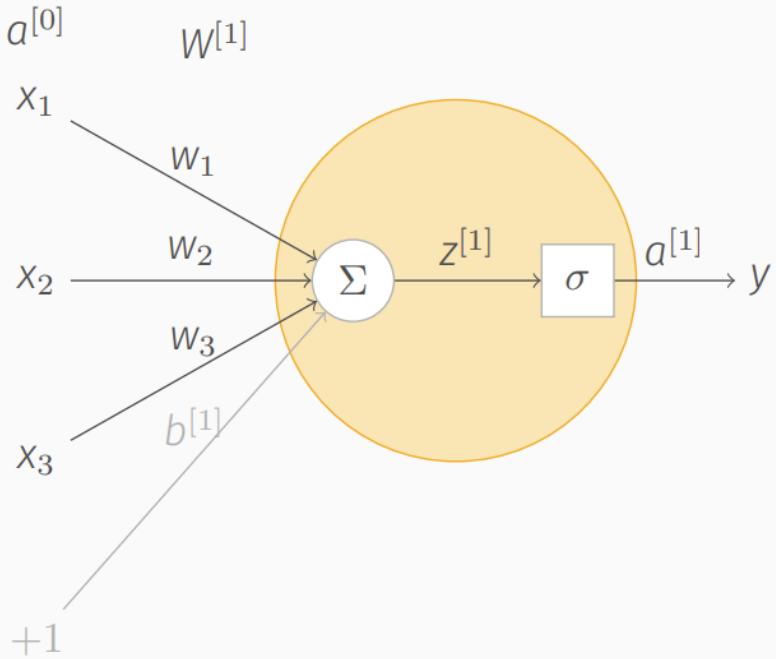
$$a^{[1]} = g^{[1]}(z^{[1]})$$ ReLU

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

$a^{[0]}$

Slide adapted from Jurafsky & Martin

A Forward Pass in Terms of Multi-Layer Notation



for each $i \in 1..n$ **do**

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

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

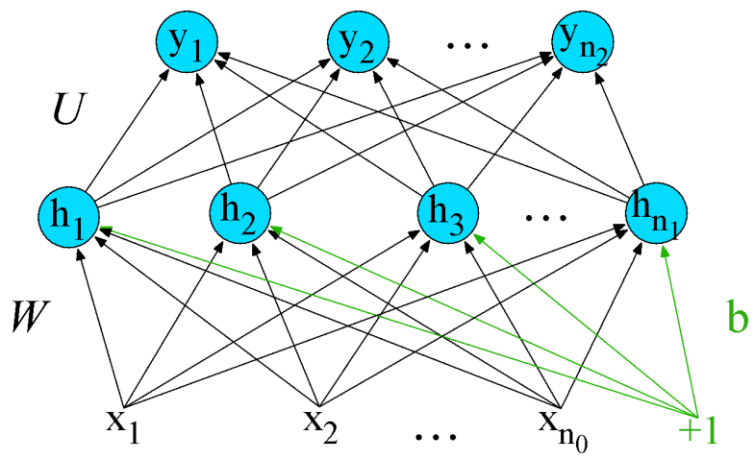
end for

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

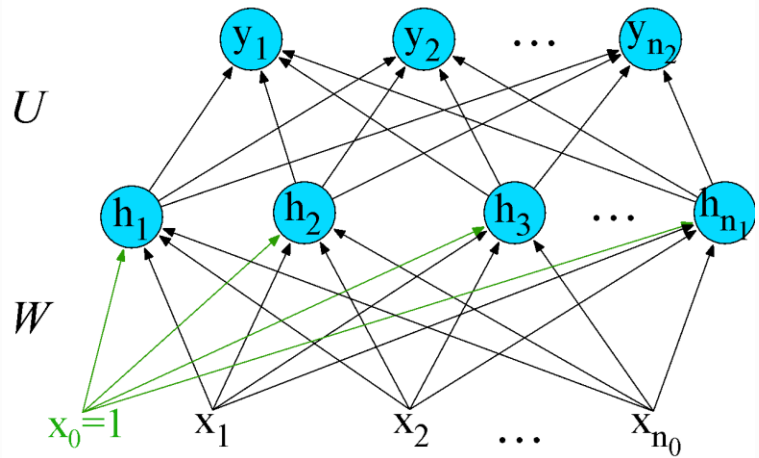
Slide credit: David Mortensen

Replacing the bias unit

Instead of:



We'll do this:



Slide adapted from Jurafsky & Martin

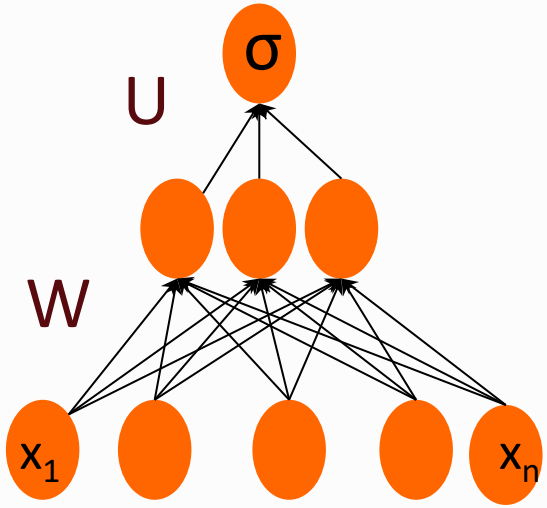
Feedforward neural nets as classifiers

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



Slide adapted from Jurafsky & Martin

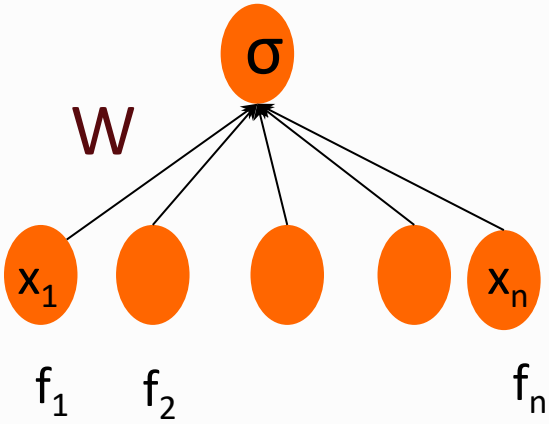
Sentiment Features

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)

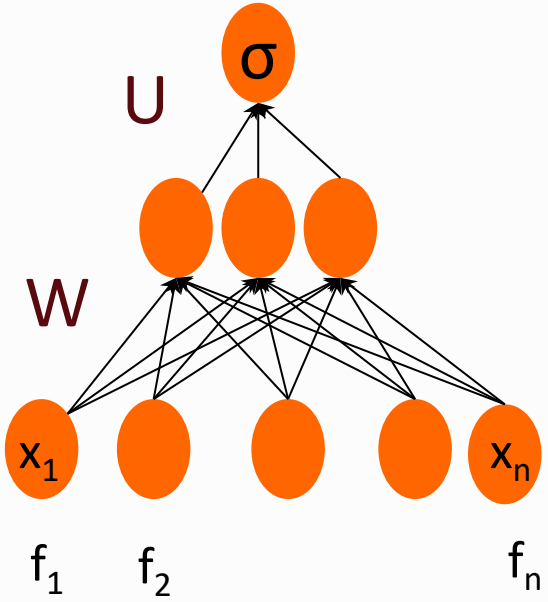
Slide adapted from Jurafsky & Martin

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.

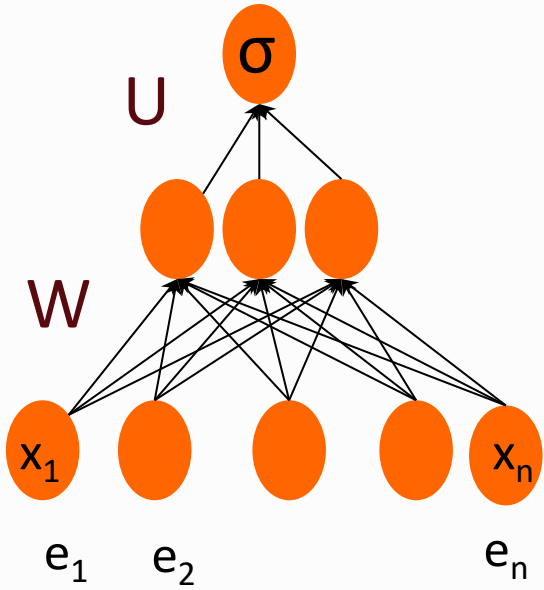
Slide adapted from Jurafsky & Martin

Even better: representation learning

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

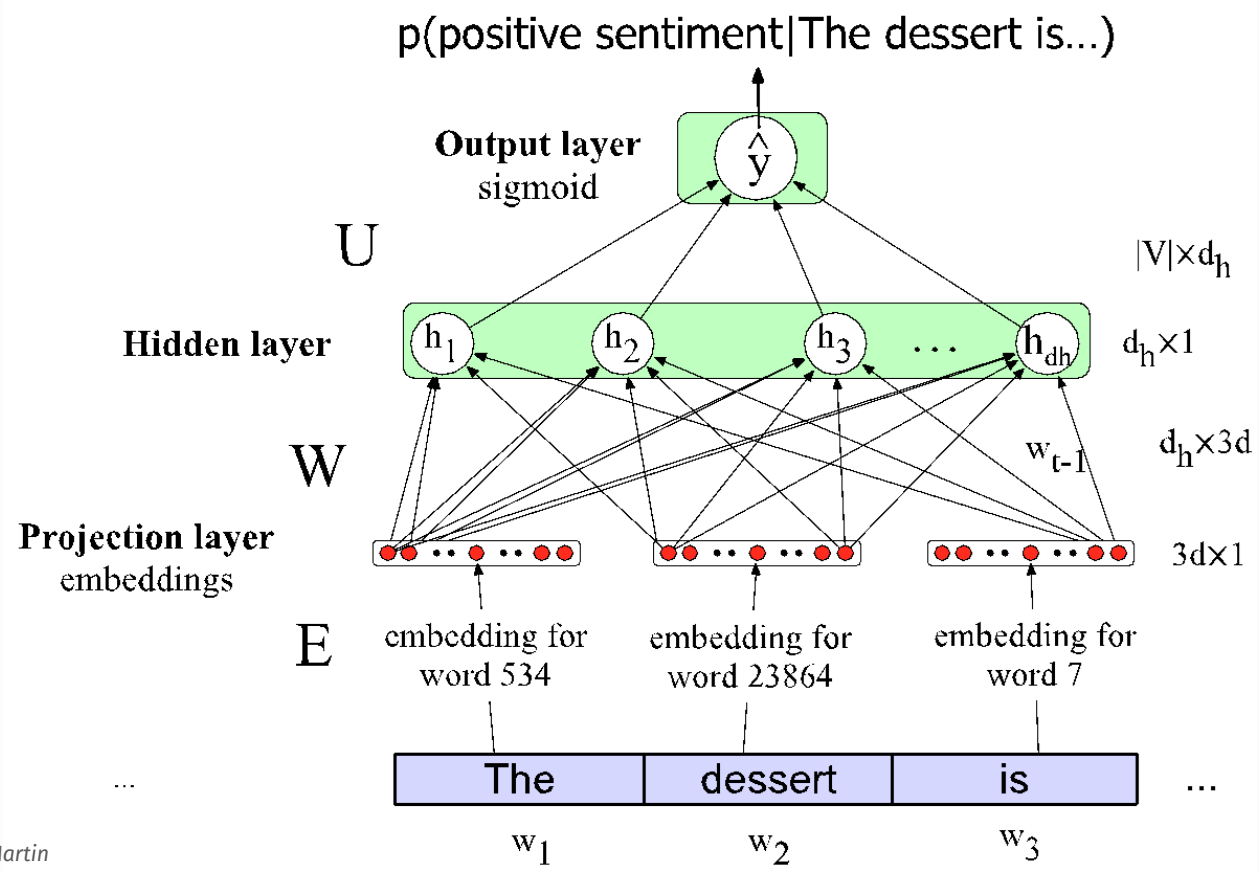
Instead of using hand-built human-engineered features for classification

Use learned representations like embeddings!



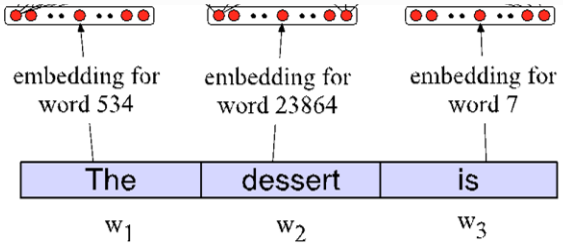
Slide adapted from Jurafsky & Martin

Neural net classification with embeddings as input features!



Slide adapted from Jurafsky & Martin

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

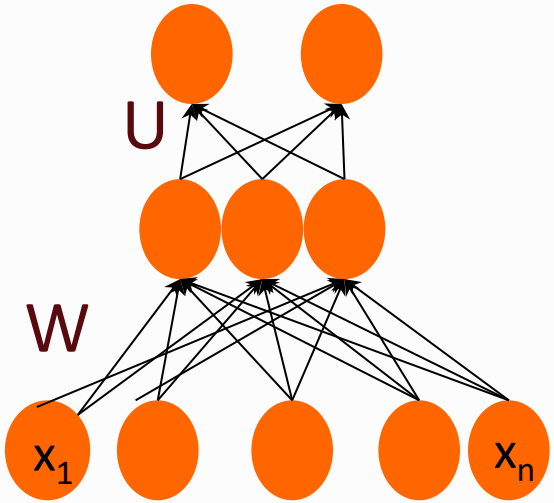
Slide adapted from Jurafsky & Martin

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”

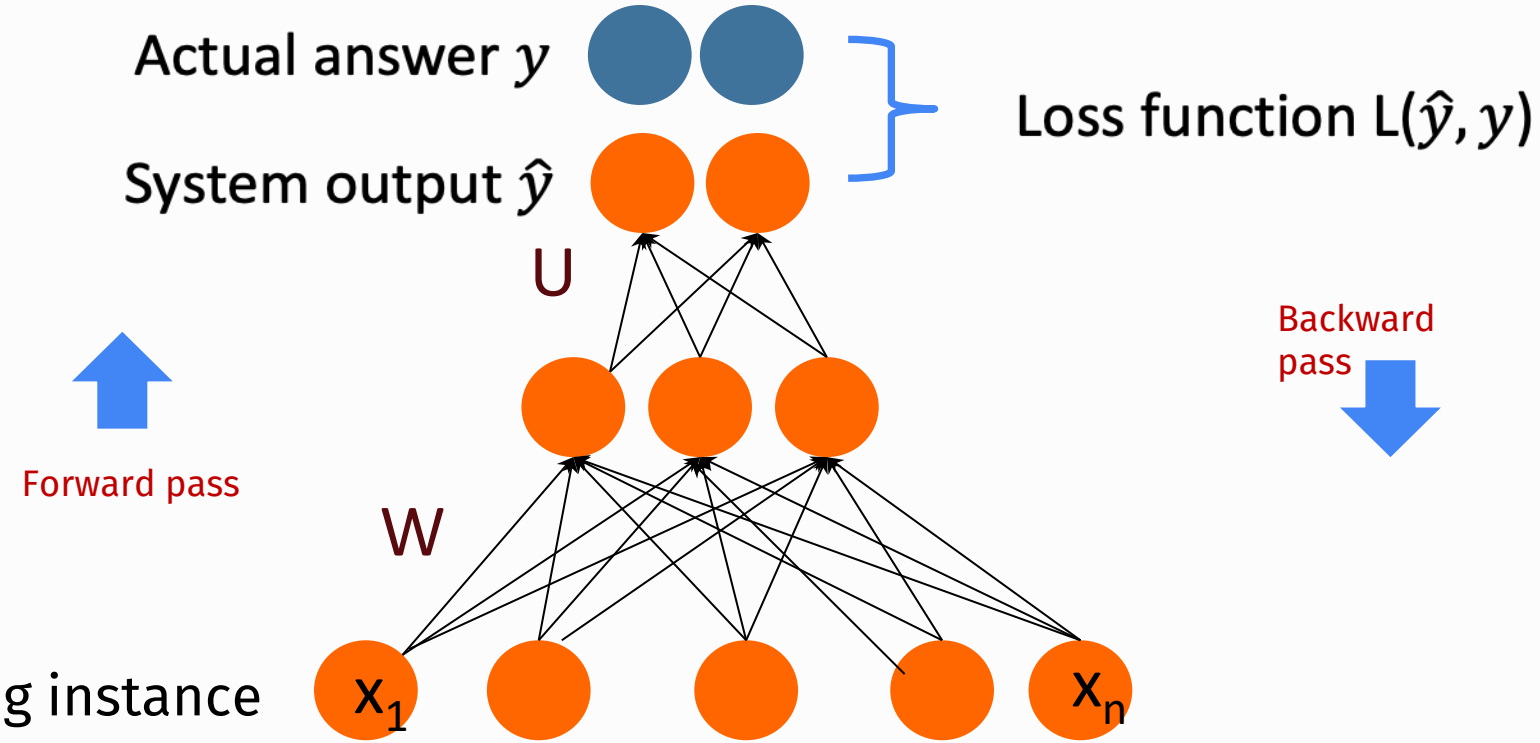
$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \quad 1 \leq i \leq D$$



Slide adapted from Jurafsky & Martin

Training feedforward neural networks

Intuition: training a 2-layer Network



Slide adapted from Jurafsky & Martin

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

Slide credit: David Mortensen

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

Slide credit: David Mortensen

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.

Slide credit: David Mortensen

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

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

- For logistic regression

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

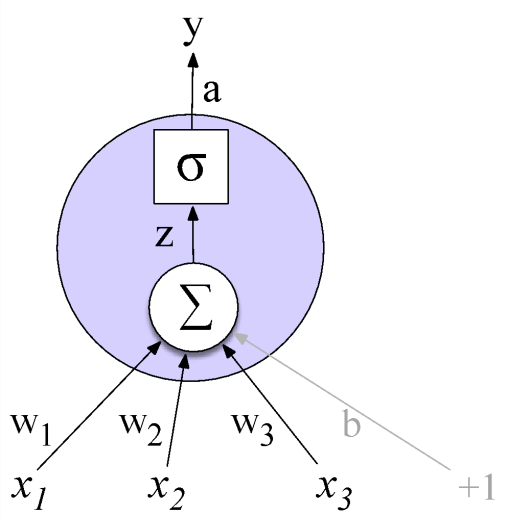
Slide adapted from Jurafsky & Martin

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

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

Slide adapted from Jurafsky & Martin

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!

Slide adapted from Jurafsky & Martin

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**

Slide adapted from Jurafsky & Martin

Computation Graphs

A computation graph represents the process of computing a mathematical expression

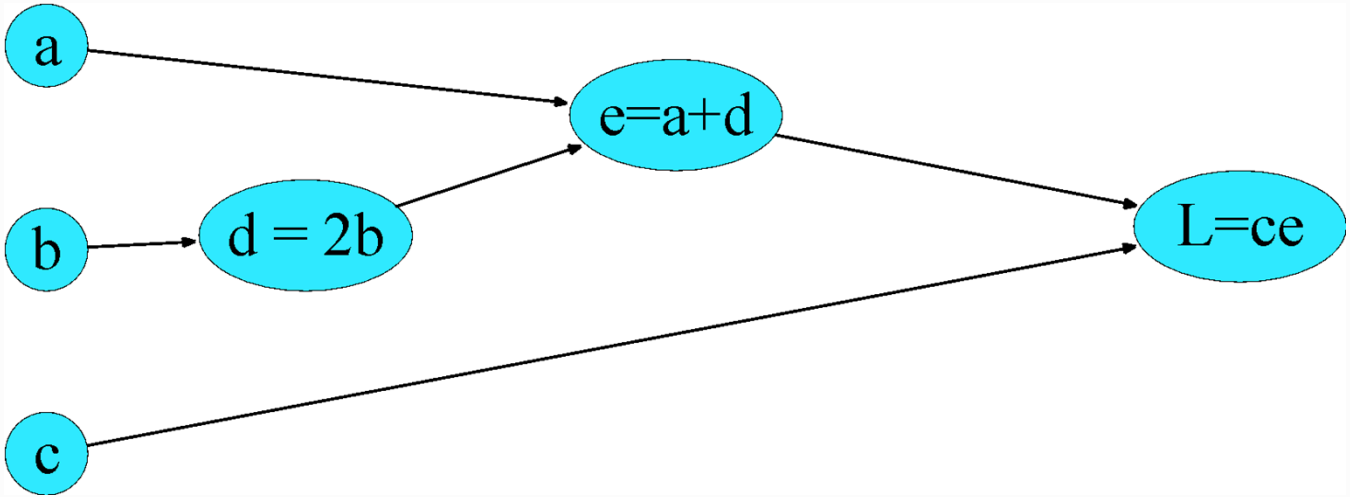
Slide adapted from Jurafsky & Martin

Example:

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

Computations:

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

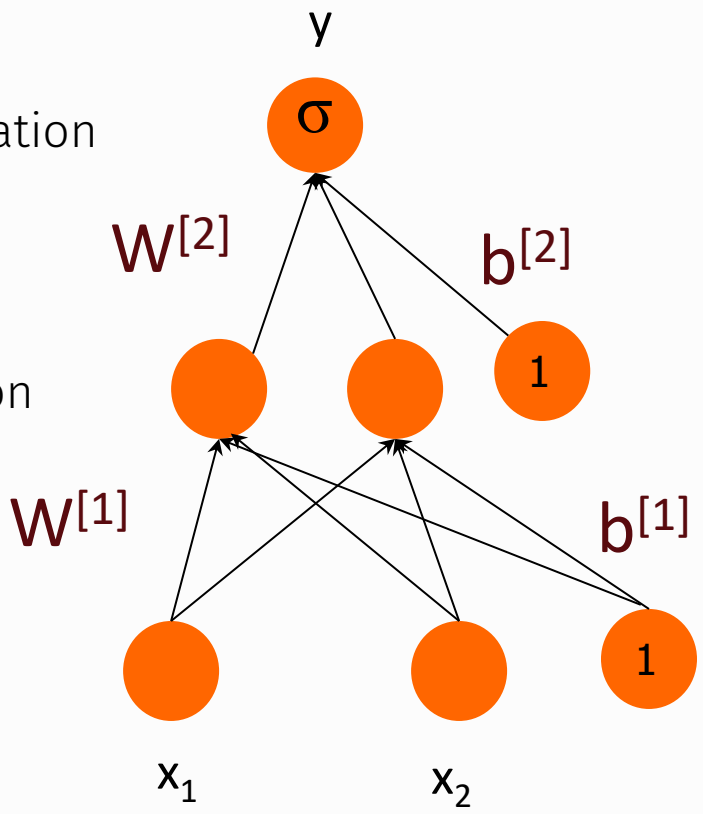


Slide adapted from Jurafsky & Martin

Backward differentiation on a two layer network

Sigmoid activation

ReLU activation



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

Slide adapted from Jurafsky & Martin

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?