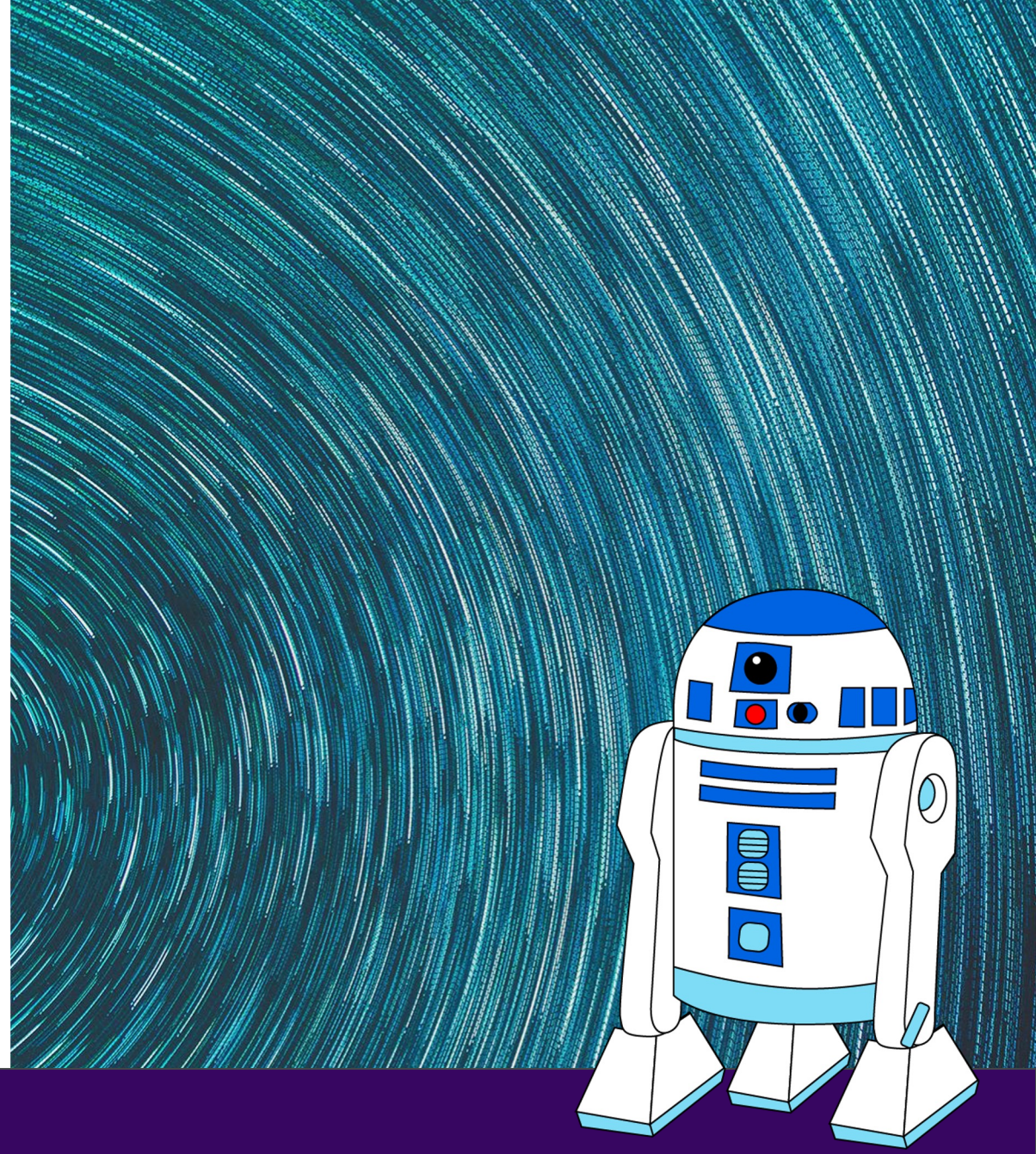
CIS 4210/5210:
ARTIFICIAL INTELLIGENCE

Module 13: Neural Networks

Jurafsky and Martin Chapter 7

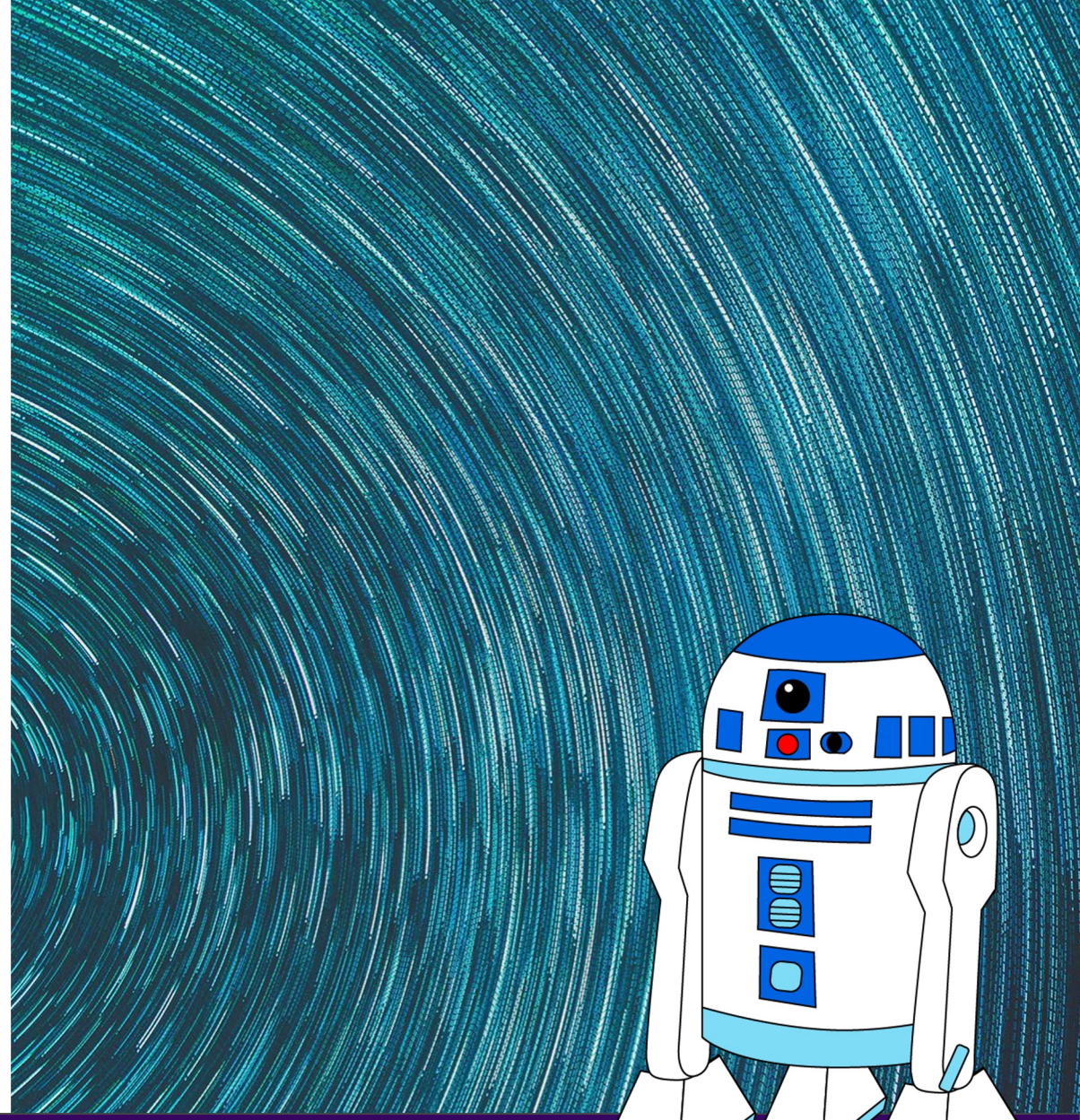


Penn Engineering
University of Pennsylvania

CIS 4210/5210:
ARTIFICIAL INTELLIGENCE

Logistic Regression

Jurafsky and Martin Chapter 5



Penn Engineering
University of Pennsylvania

Classifier components

Machine learning classifiers require a training corpus of M observations input/output pairs $(x^{(i)},y^{(i)})$.

1. A **feature representation** of the input. For each input observation $x^{(i)}$, this will be a vector of features $[x_1,x_2,...,x_n]$.
2. A **classification function** that computes the estimated class $\hat{y}$ via $p(y|x)$.
3. An **objective function** for learning, usually involving minimizing error on training examples.
4. An algorithm for **optimizing** the objective function.

Sentiment classifier

Input: "Spiraling away from narrative control as its first three episodes unreel, this series, about a post-apocalyptic future in which nearly everyone is blind, wastes the time of Jason Momoa and Alfre Woodard, among others, on a story that starts from a position of fun, giddy strangeness and drags itself forward at a lugubrious pace."

Output: positive (1) or **negative (0)**

Sentiment classifier

For sentiment classification, consider an input observation x, represented by a vector of **features** $[x_1,x_2,...,x_n]$. The classifier output y can be 1 (positive sentiment) or 0 (negative sentiment). We want to estimate $\boldsymbol{P(y = 1|x)}$.

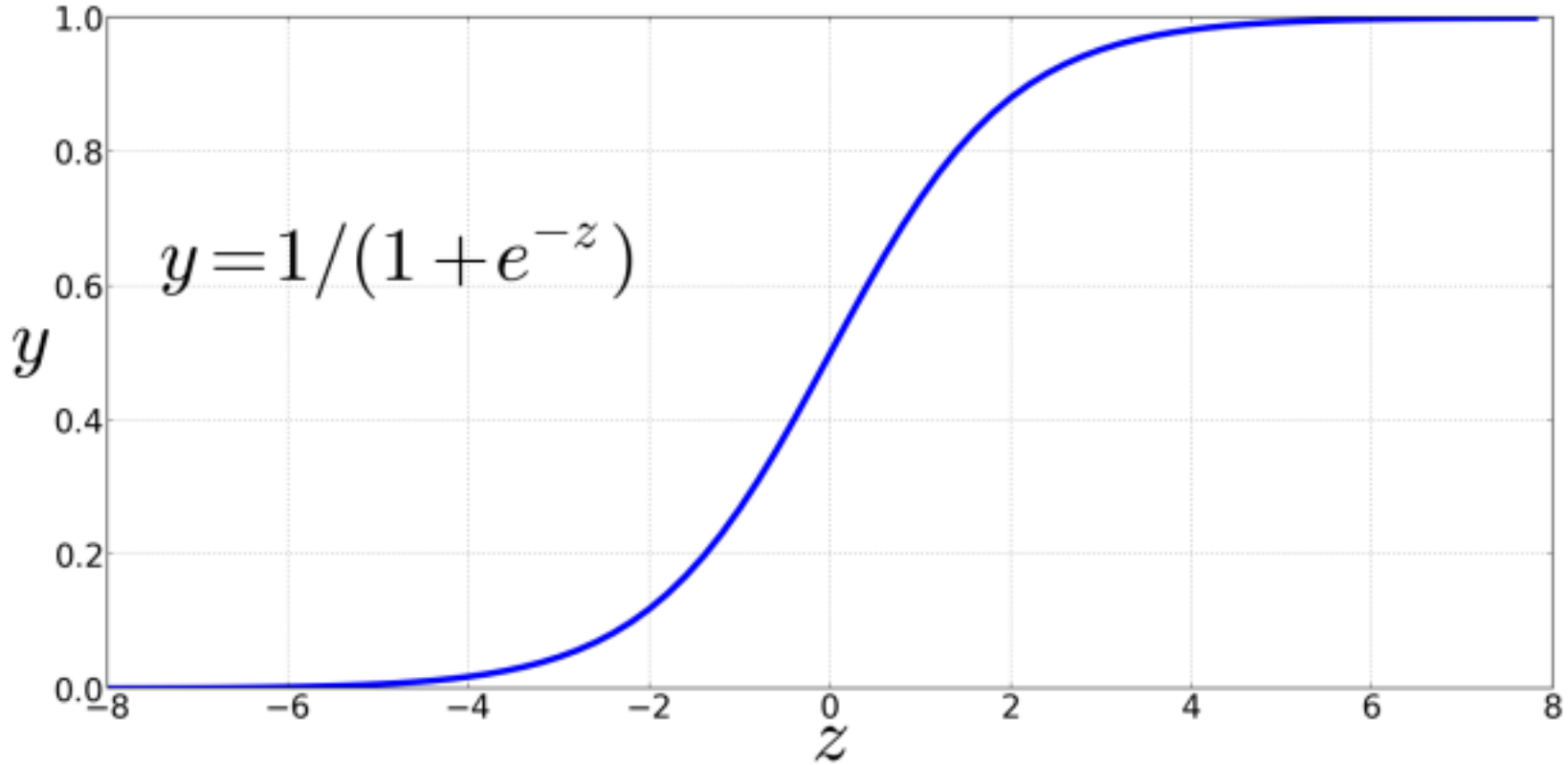
Logistic regression solves this task by learning, from a training set, a vector of **weights** and a **bias term**.

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

We can also write this as a dot product:

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

Sigmoid function



Probabilities

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

Decision boundary

Now we have an algorithm that given an instance *x* computes the probability *P*(*y* = 1 | *x*). How do we make a decision?

$$\hat{y} = \begin{cases} 1 \; if \; P(y = 1|x) > 0.5 \\ 0 \; otherwise \end{cases}$$

For a test instance *x*, we say **yes** if the probability *P*(*y* = 1 | *x*) is more than .5, and **no** otherwise. We call .5 the decision boundary

Extracting Features

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

Var	Definition	Value
x_1	Count of positive lexicon words	
x_2	Count of negative lexicon words	
x_3	Does no appear? (binary feature)	
x_4	Number of 1st and 2nd person pronouns	
x_5	Does ! appear? (binary feature)	
x_6	Log of the word count for the document	

Extracting Features

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

Var	Definition	Value
x_1	Count of positive lexicon words	3
x_2	Count of negative lexicon words	
x_3	Does no appear? (binary feature)	
x_4	Number of 1st and 2nd person pronouns	
x_5	Does ! appear? (binary feature)	
x_6	Log of the word count for the document	

Extracting Features

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

Var	Definition	Value
x_1	Count of positive lexicon words	3
x_2	Count of negative lexicon words	2
x_3	Does no appear? (binary feature)	
x_4	Number of 1st and 2nd person pronouns	
x_5	Does ! appear? (binary feature)	
x_6	Log of the word count for the document	

Extracting Features

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

Var	Definition	Value
x_1	Count of positive lexicon words	3
x_2	Count of negative lexicon words	2
x_3	Does no appear? (binary feature)	1
x_4	Number of 1st and 2nd person pronouns	
x_5	Does ! appear? (binary feature)	
x_6	Log of the word count for the document	

Extracting Features

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

Var	Definition	Value
x_1	Count of positive lexicon words	3
x_2	Count of negative lexicon words	2
x_3	Does no appear? (binary feature)	1
x_4	Number of 1st and 2nd person pronouns	3
x_5	Does ! appear? (binary feature)	
x_6	Log of the word count for the document	

Extracting Features

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

Var	Definition	Value
x_1	Count of positive lexicon words	3
x_2	Count of negative lexicon words	2
x_3	Does no appear? (binary feature)	1
x_4	Number of 1st and 2nd person pronouns	3
x_5	Does ! appear? (binary feature)	0
x_6	Log of the word count for the document	

Extracting Features

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

Word count = 64, ln(64) = 4.15

Var	Definition	Value
x_1	Count of positive lexicon words	3
x_2	Count of negative lexicon words	2
x_3	Does no appear? (binary feature)	1
x_4	Number of 1st and 2nd person pronouns	3
x_5	Does ! appear? (binary feature)	0
x_6	Log of the word count for the document	4.15

Var	Definition	Value	Weight	Product
x_1	Count of positive lexicon words	3	2.5	
x_2	Count of negative lexicon words	2	-5.0	
x_3	Does no appear? (binary feature)	1	-1.2	
x_4	Num 1st and 2nd person pronouns	3	0.5	
x_5	Does ! appear? (binary feature)	0	2.0	
x_6	Log of the word count for the doc	4.15	0.7	
b	bias	1	0.1	

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

Computing Z

Var	Definition	Value	Weight	Product
x_1	Count of positive lexicon words	3	2.5	7.5
x_2	Count of negative lexicon words	2	-5.0	-10
x_3	Does no appear? (binary feature)	1	-1.2	-1.2
x_4	Num 1st and 2nd person pronouns	3	0.5	1.5
x_5	Does ! appear? (binary feature)	0	2.0	0
x_6	Log of the word count for the doc	4.1	0.7	2.905
b	bias	1	0.1	0.1

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

z=0.805

Sigmoid(Z)

Var	Definition	Value	Weight	Product
x_1	Count of positive lexicon words	3	2.5	7.5
x_2	Count of negative lexicon words	2	-5.0	-10
x_3	Does no appear? (binary feature)	1	-1.2	-1.2
x_4	Num 1st and 2nd person pronouns	3	0.5	1.5
x_5	Does ! appear? (binary feature)	0	2.0	0
x_6	Log of the doc		7	2.905
b	bias		1	0.1

$$\sigma(0.805)=0.69$$

Learning in logistic regression

How do we get the weights of the model? We learn the parameters (weights + bias) via learning. This requires 2 components:

1. An objective function or **loss function** that tells us *distance* between the system output and the gold output. We will use **cross-entropy loss**.
2. An algorithm for optimizing the objective function. We will use stochastic gradient descent to **minimize** the **loss function.**

Loss functions

We need to determine for some observation *x* how close the classifier output ($\hat{y}$= σ ($w \cdot x + b$)) is to the correct output (*y*, which is 0 or 1).

$$L(\hat{y}, y) = \text{how much } \hat{y} \text{ differs from the true } y$$

One example is mean squared error

$$L_{MSE}(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

Loss functions for probabilistic classification

We use a loss function that prefers the correct class labels of the training example to be more likely.

Conditional maximum likelihood estimation: Choose parameters *w, b* that maximize the (log) probabilities of the true labels in the training data.

The resulting loss function is the negative log likelihood loss, more commonly called the **cross entropy loss.**

Loss functions for probabilistic classification

For one observation x, let's **maximize** the probability of the correct label p(y|x).

$$p(y|x) = \hat{y}^{y}(1-\hat{y})^{1-y}$$

If y = 1, then $\mathrm{p(y|x)} = \hat{y}$.

If y = 0, then $\mathrm{p(y|x)} = 1 - \hat{y}$.

Loss functions for probabilistic classification

Change to logs (still maximizing)

$$\log p(y|x) = \log[\hat{y}^y(1-\hat{y})^{1-y}]$$
$$= y \log \hat{y} + (1-y)\log(1-\hat{y})$$

This tells us what log likelihood should be maximized. But for loss functions, we want to minimize things, so we'll flip the sign.

Cross-entropy loss

The result is cross-entropy loss:

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

Finally, plug in the definition for $\hat{y} = \sigma(w \cdot x) + b$

$$L_{CE}(\hat{y}, y) = -[y \log \sigma(w \cdot x + b) + (1-y) \log(1 - \sigma(w \cdot x + b))]$$

Cross-entropy loss

Why does minimizing this negative log probability do what we want? We want the **loss** to be **smaller** if the model's estimate is **close to correct**, and we want the **loss** to be **bigger if it is confused.**

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

P(sentiment=1 | It's hokey…) = 0.69. Let's say y=1.

$$L_{CE}(\hat{y}, y) = -[y \log \sigma(w \cdot x + b) + (1 - y) \log(1 - \sigma(w \cdot x + b))]$$

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

$$= -\log(0.69) = \mathbf{0.37}$$

Cross-entropy loss

Why does minimizing this negative log probability do what we want? We want the **loss** to be **smaller** if the model's estimate is **close to correct**, and we want the **loss** to be **bigger if it is confused.**

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

P(sentiment=1 | It's hokey…) = 0.69. Let's **pretend** y=0.

$$L_{CE}(\hat{y}, y) = -[y \log \sigma(w \cdot x + b) + (1 - y) \log(1 - \sigma(w \cdot x + b))]$$

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

$$= -\log(0.31) = \mathbf{1.17}$$

Cross-entropy loss

Why does minimizing this negative log probability do what we want? We want the **loss** to be **smaller** if the model's estimate is **close to correct**, and we want the **loss** to be **bigger if it is confused.**

It's hokey. There are virtually no surprises , and the writing is second-rate . So why was it so enjoyable? For one thing , the cast is great . Another nice touch is the music . I was overcome with the urge to get off the couch and start dancing . It sucked me in , and it'll do the same to you .

If our prediction is **correct**, then our CE loss is **lower**

$$= -\log(0.69) = \mathbf{0.37}$$

If our prediction is **incorrect**, then our CE loss is **higher**

$$-\log(0.31) = \mathbf{1.17}$$

Loss on all training examples

$$\log p(training\ labels) = \log \prod_{i=1}^{m} p(y^{(i)}|x^{(i)})$$

$$= \sum_{i=1}^{m} \log p(y^{(i)}|x^{(i)})$$

$$= -\sum_{i=1}^{m} \mathrm{L_{CE}}(\hat{y}^{(i)}|y^{(i)})$$

Finding good parameters

We use gradient descent to find good settings for our weights and bias by minimizing the loss function.

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^{m} L_{CE}(y^{(i)}, x^{(i)}; \theta)$$

Gradient descent is a method that finds a minimum of a function by figuring out in which direction (in the space of the parameters θ) the function's slope is rising the most steeply, and moving in the opposite direction.

Gradient descent

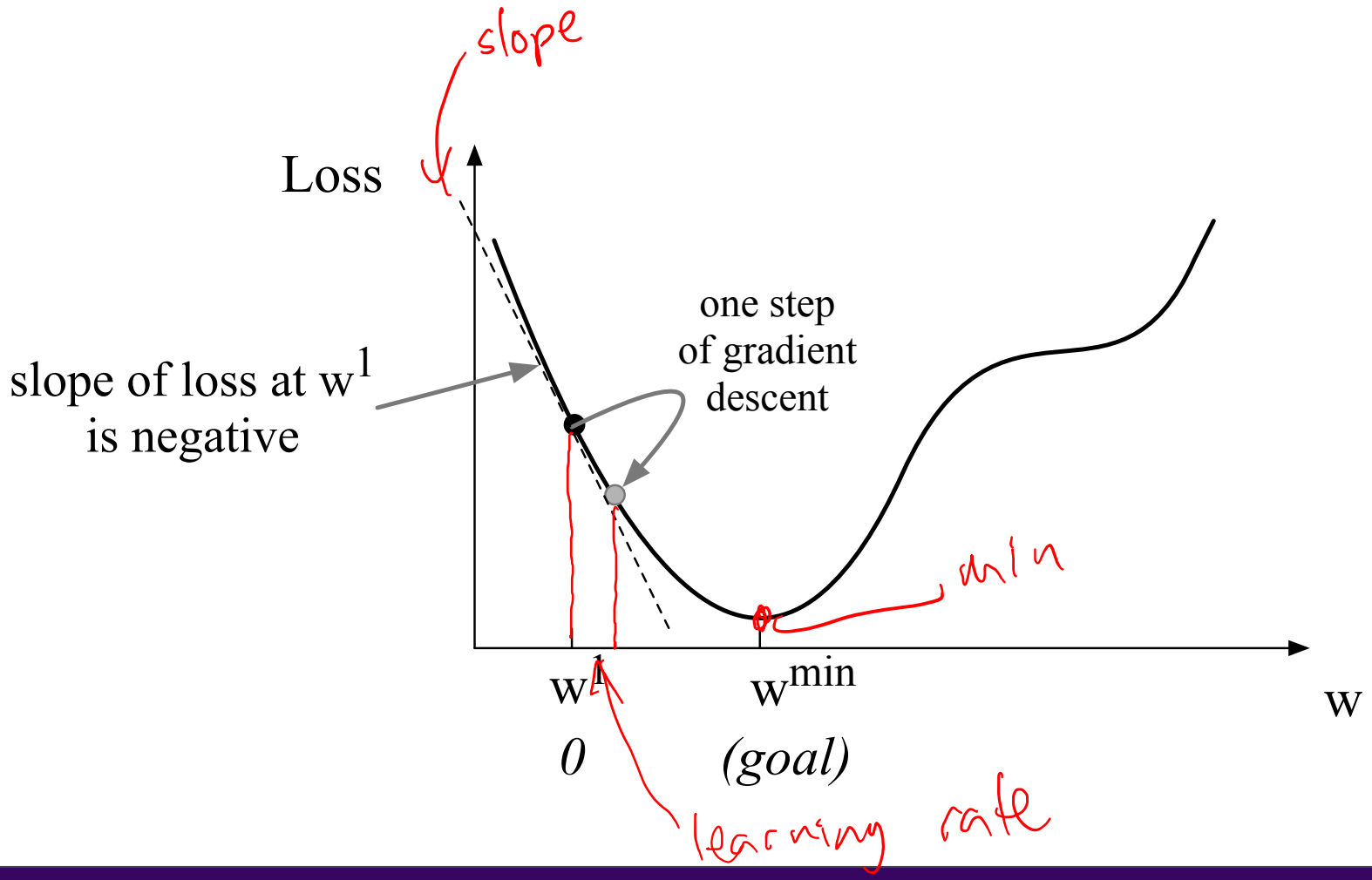
Global v. Local Minimums

For logistic regression, this loss function is conveniently **convex**.

A convex function has just **one minimum**, so there are no local minima to get stuck in.

So gradient descent starting from any point is guaranteed to find the minimum.

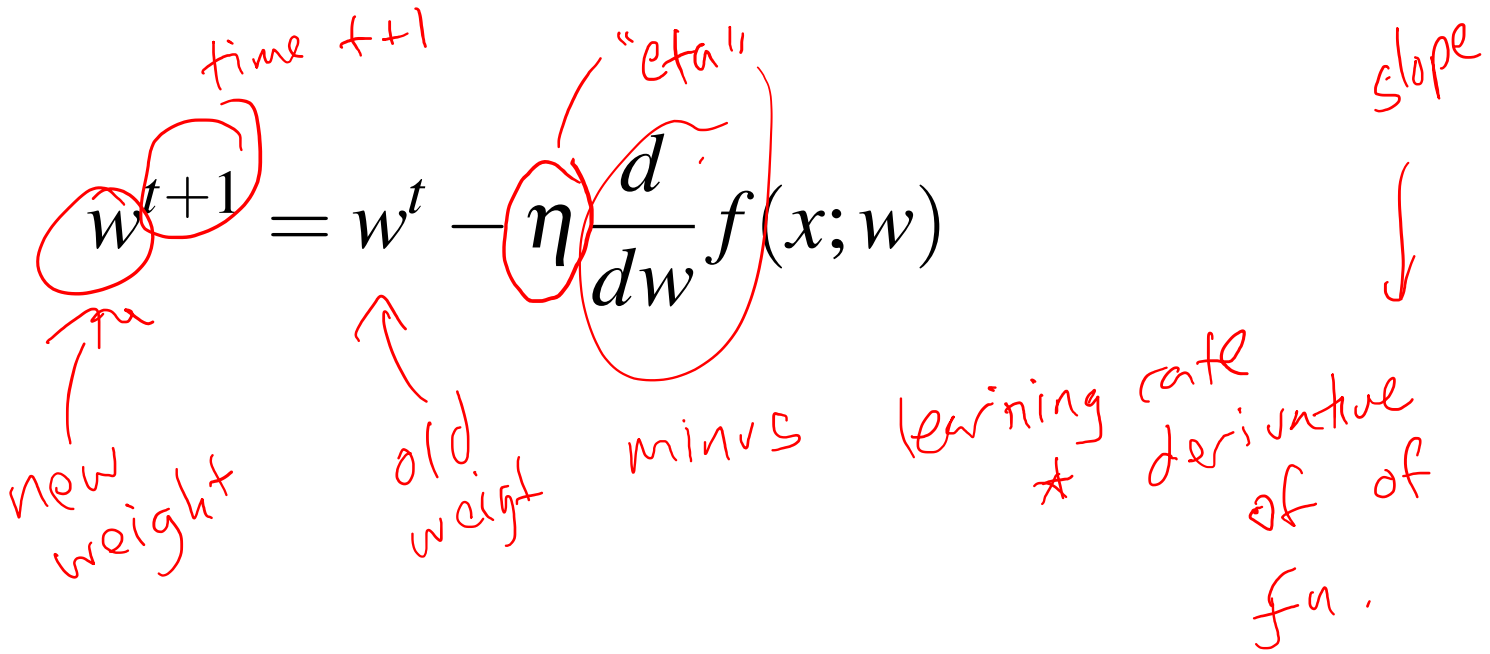
Iteratively find minimum



How much should we update the parameter by?

The magnitude of the amount to move in gradient descent is the value of the slope weighted by a learning rate η.

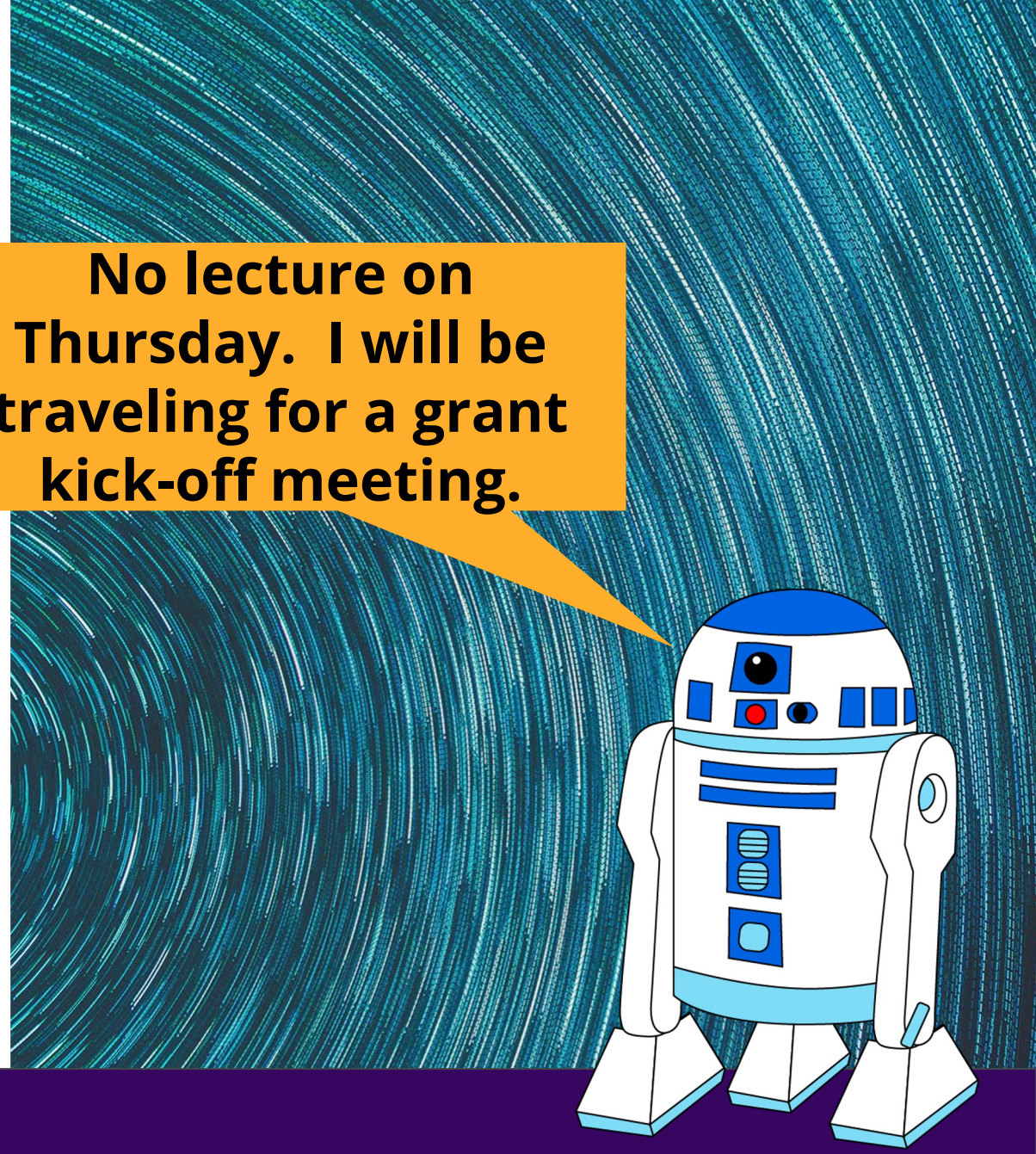
A higher/faster learning rate means that we should move *w* more on each step.



CIS 4210/5210:
ARTIFICIAL INTELLIGENCE

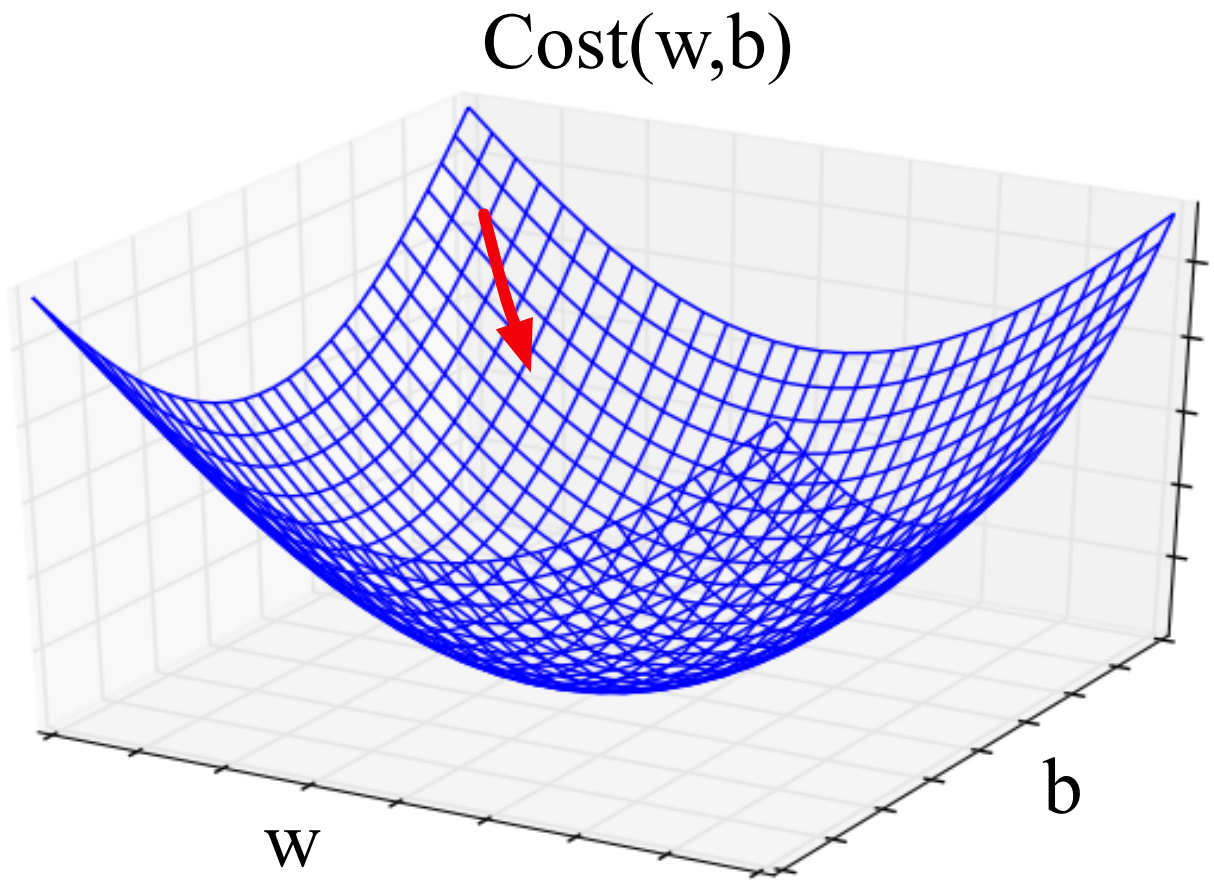
Stochastic Gradient Descent

Jurafsky and Martin Chapter 5



Gradient descent

Many dimensions



Updating each dimension w_i

$$\nabla_\theta L(f(x;\theta),y)) = \begin{bmatrix} \frac{\partial}{\partial w_1} L(f(x;\theta),y) \\ \frac{\partial}{\partial w_2} L(f(x;\theta),y) \\ \vdots \\ \frac{\partial}{\partial w_n} L(f(x;\theta),y) \end{bmatrix}$$

y

our model's prediction for input x given parameters θ

The final equation for updating θ based on the gradient is

$$\theta_{t+1} = \theta_t - \eta \nabla L(f(x;\theta),y)$$

Learning rate

The Gradient

To update θ, we need a definition for the gradient $\nabla L(f(x;\theta), y)$.

For logistic regression, the cross-entropy loss function is:

$$L_{\text{CE}}(\hat{y}, y) = -[y \log \sigma(\mathbf{w}\cdot\mathbf{x}+b) + (1-y)\log(1-\sigma(\mathbf{w}\cdot\mathbf{x}+b))]$$

The derivative of this function for one observation vector x for a single weight w_j is

$$\begin{aligned}\frac{\partial L_{\text{CE}}(\hat{y}, y)}{\partial \mathbf{w}_j} &= [\sigma(\mathbf{w}\cdot\mathbf{x}+b) - y]\mathbf{x}_j \\ &= (\hat{y}-y)\mathbf{x}_j\end{aligned}$$

The gradient is a very intuitive value: the difference between the true y and our estimate for x, multiplied by the corresponding input value x_j .

Average Loss

$$Cost(\hat{y}, y) = \frac{1}{m}\sum_{i=1}^{m} L_{CE}(\hat{y}^{(i)}, y^{(i)})$$

$$= -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \sigma\left(\boldsymbol{w} \cdot \boldsymbol{x}^{(i)} + b\right) + \left(1 - y^{(i)}\right)\log\left(1 - \sigma\left(\boldsymbol{w} \cdot \boldsymbol{x}^{(i)} + b\right)\right)$$

This is what we want to minimize!!

Batch Training

The loss for a batch of data or an entire dataset is just the average loss over the *m* examples

$$Cost(\hat{y}, y) = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)} \log \sigma\left(w \cdot x^{(i)} + b\right) + \left(1 - y^{(i)}\right) \log\left(1 - \sigma\left(w \cdot x^{(i)} + b\right)\right)$$

The gradient for multiple data points is the sum of the individual gradients:

$$\frac{\partial Cost(\hat{y}, y)}{\partial w_j} = \sum_{i=1}^{m} [\sigma\left(w \cdot x^{(i)} + b\right) - y^{(i)}] x_j^{(i)}$$

Stochastic gradient descent algorithm

function STOCHASTIC GRADIENT DESCENT($L()$, $f()$, x, y) **returns** θ

where: L is the loss function
f is a function parameterized by θ
x is the set of training inputs $x^{(1)}, x^{(2)}, \ldots, x^{(n)}$
y is the set of training outputs (labels) $y^{(1)}, y^{(2)}, \ldots, y^{(n)}$

$\theta \leftarrow 0$
repeat T times
  For each training tuple $(x^{(i)}, y^{(i)})$ (in random order)
  Compute $\hat{y}^{(i)} = f(x^{(i)}; \theta)$ # What is our estimated output $\hat{y}$?
  Compute the loss $L(\hat{y}^{(i)}, y^{(i)})$ # How far off is $\hat{y}^{(i)}$ from the true output $y^{(i)}$?
  $g \leftarrow \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$ # How should we move θ to maximize loss ?
  $\theta \leftarrow \theta - \eta\, g$ # go the other way instead
return θ

Worked example

Let's walk though a single step of the gradient descent algorithm. We'll use a simple sentiment classifier with just 2 features, and 1 training instance where the correct value is $y = 1$ (this is a positive review).

$x_1 = 3$ (count of positive lexicon words)

$x_2 = 2$ (count of negative lexicon words)

The initial weights and bias in θ^0 are all set to 0, and the initial learning rate η is 0.1:

$$w_1 = w_2 = b = 0$$

$$\eta = 0.1$$

The single update step requires that we compute the gradient, multiplied by the learning rate:

$$\theta^{t+1} = \theta^t - \eta \nabla_\theta L(f(x^{(i)}; \theta), y^{(i)})$$

Worked example

The derivative of this function for **a single training example** *x* for a single weight w_j is

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

The gradient vector has 3 dimensions, for w_1, w_2, and b.
For our input, $x_1 = 3$ and $x_2 = 2$

$x_2 = 2$

$$\nabla_{w,b} = \begin{bmatrix} \frac{\partial L_{CE}(w,b)}{\partial w_1} \\ \frac{\partial L_{CE}(w,b)}{\partial w_2} \\ \frac{\partial L_{CE}(w,b)}{\partial b} \end{bmatrix} = \begin{bmatrix} (\sigma(w \cdot x + b) - y)x_1 \\ (\sigma(w \cdot x + b) - y)x_2 \\ \sigma(w \cdot x + b) - y \end{bmatrix} = \begin{bmatrix} (\sigma(0) - 1)x_1 \\ (\sigma(0) - 1)x_2 \\ \sigma(0) - 1 \end{bmatrix} = \begin{bmatrix} -0.5x_1 \\ -0.5x_2 \\ -0.5 \end{bmatrix} = \begin{bmatrix} -1.5 \\ -1.0 \\ -0.5 \end{bmatrix}$$

Worked example

Now that we have a gradient $\nabla_{w,b}$, we compute the new parameter vector θ^1 by moving θ^0 in the opposite direction from the gradient:

$$\theta^1 = \begin{bmatrix} w_1 \\ w_2 \\ b \end{bmatrix} - \eta \begin{bmatrix} -1.5 \\ -1.0 \\ -0.5 \end{bmatrix} = \begin{bmatrix} .15 \\ .1 \\ .05 \end{bmatrix}$$

So after one step of gradient descent, the weights have shifted to be:

$w_1 = 0.15$, $w_2 = 0.1$, and $b = .05$

Mini-batch training

Stochastic gradient descent chooses a **single random example** at a time and updates its weights on that example. As a result, the updates can fluctuate.

An alternate is **batch training,** which computes the gradient over the **entire dataset**. This gives a much better estimate of which direction to move the weights but takes a long time to compute.

A commonly used compromise is **mini-batch training**, where we train on a small batch. The batch size can be 512 or 1024, often selected based on computational resources, so that all examples in the mini-batch can be processed in parallel. The loss is then accumulated.

Regularization

Overfitting is a problem with many machine learning models. Overfitting results in poor generalization and poor performance on unseen test set.

In logistic regression, if a feature only occurs in one class then it will get a **high weight**. Sometimes we are just modelling noisy factors that just accidentally correlate with the class.

Regularization is a way to penalize large weights. A regularization term is added to the loss function.

Lasso regression uses L1 regularization
Ridge regression uses L2 regularization

Multinomial logistic regression

Instead of binary classification, we often want more than two classes. For sentiment classification we might extend the class labels to be **positive, negative**, and **neutral**.

We want to know the probability of y for each class $c \in C$, $p(y = c|x)$.

To get a proper probability, we will use a **generalization of the sigmoid function** called the **softmax function.**

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

Softmax

The softmax function takes in an input vector $z = [z_1, z_2, \ldots, z_k]$ and outputs a vector of values normalized into probabilities.

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

For example, for this input:

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

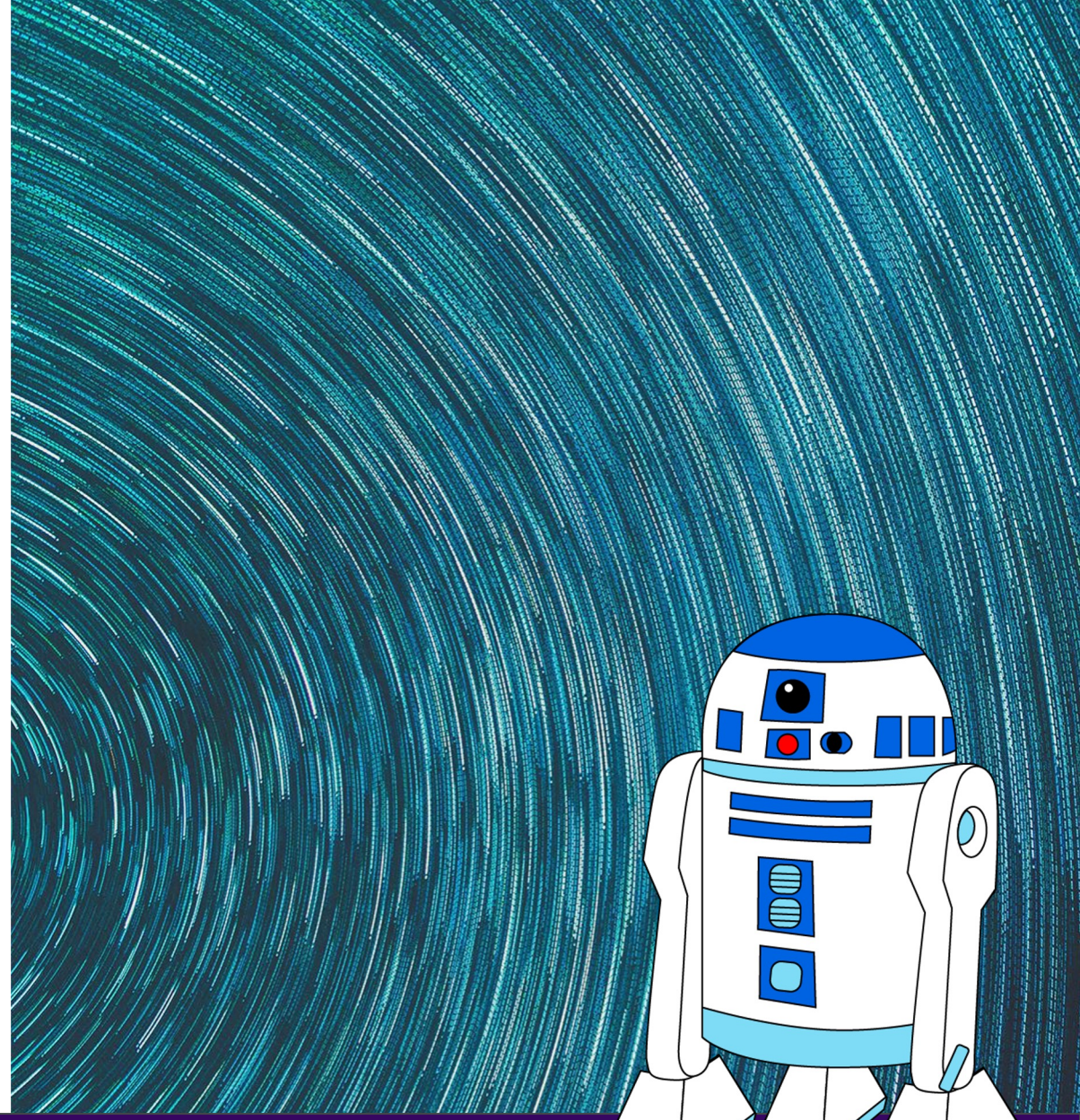
Softmax will output:

$$[0.056, 0.090, 0.007, 0.099, 0.74, 0.010]$$

CIS 4210/5210:
ARTIFICIAL INTELLIGENCE

Neural Networks

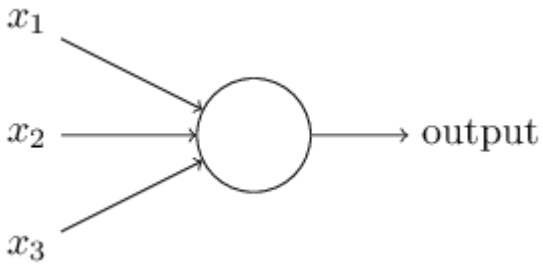
Jurafsky and Martin Chapter 7

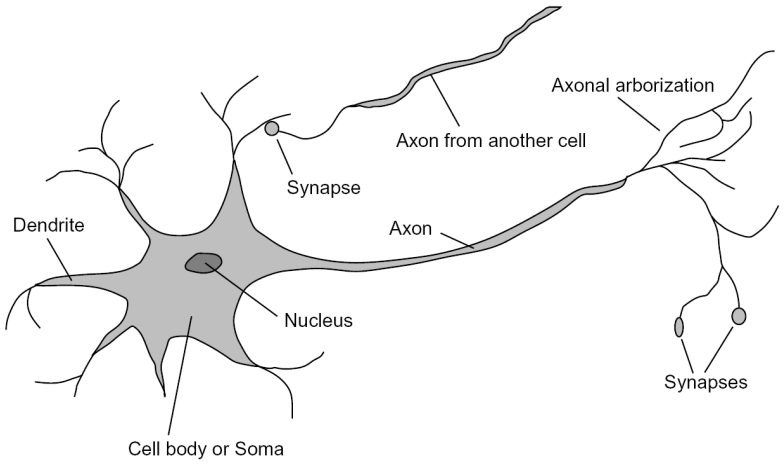


Penn Engineering
University of Pennsylvania

Review: Perceptron

Perceptrons were developed in the 1950s and 1960s loosely inspired by the neuron.





Eléctronic 'Brain' Teaches Itself

The Navy last week demonstrated the embryo of an electronic computer named the Perceptron which, when completed in about a year, is expected to be the first non-living mechanism able to "perceive, recognize and identify its surroundings without human training or control." Navy officers demonstrating a preliminary form of the device in Washington said they hesitated to call it a machine because it is so much like a "human being without life."

Dr. Frank Rosenblatt, research psychologist at the Cornell Aeronautical Laboratory, Inc., Buffalo, N. Y., designer of the Perceptron, conducted the demonstration. The machine, he said, would be the first electronic device to think as the human brain. Like humans, Perceptron will make mistakes at first, "but it will grow wiser as it gains experience," he said.

The first Perceptron, to cost about $100,000, will have about 1,000 electronic "association cells" receiving electrical impulses from an eyelike scanning device with 400 photocells. The human brain has ten billion

recognize the difference between right and left, almost the way a child learns.

When fully developed, the Perceptron will be designed to remember images and information it has perceived itself, whereas ordinary computers remember only what is fed into them on punch cards or magnetic tape.

Later Perceptrons, Dr. Rosenblatt said, will be able to recognize people and call out their names. Printed pages, longhand letters and even speech commands are within its reach. Only one more step of development, a difficult step, he said, is needed for the device to hear speech in one language and instantly translate it to speech or writing in another language.

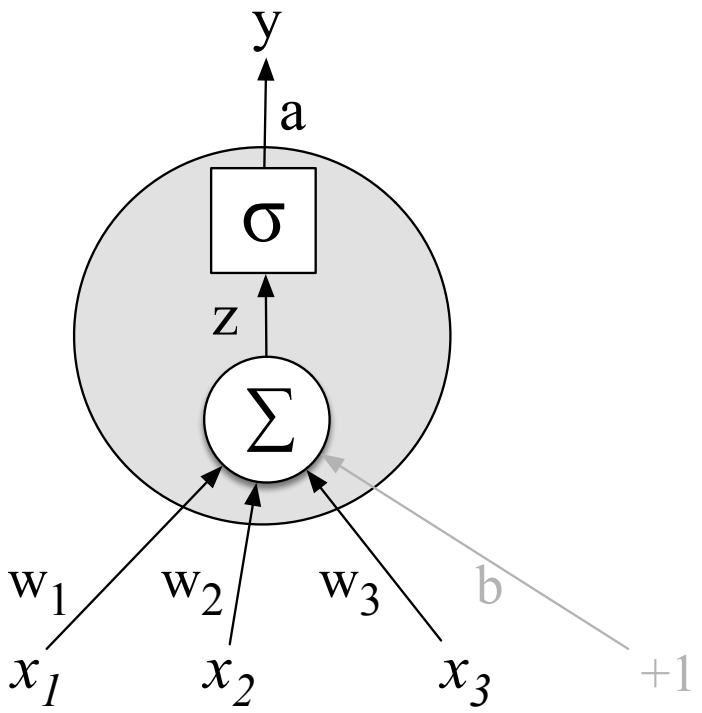
Self-Reproduction

In principle, Dr. Rosenblatt said, it would be possible to build Perceptrons that could reproduce themselves on an assembly line and which would be "conscious" of their existence.

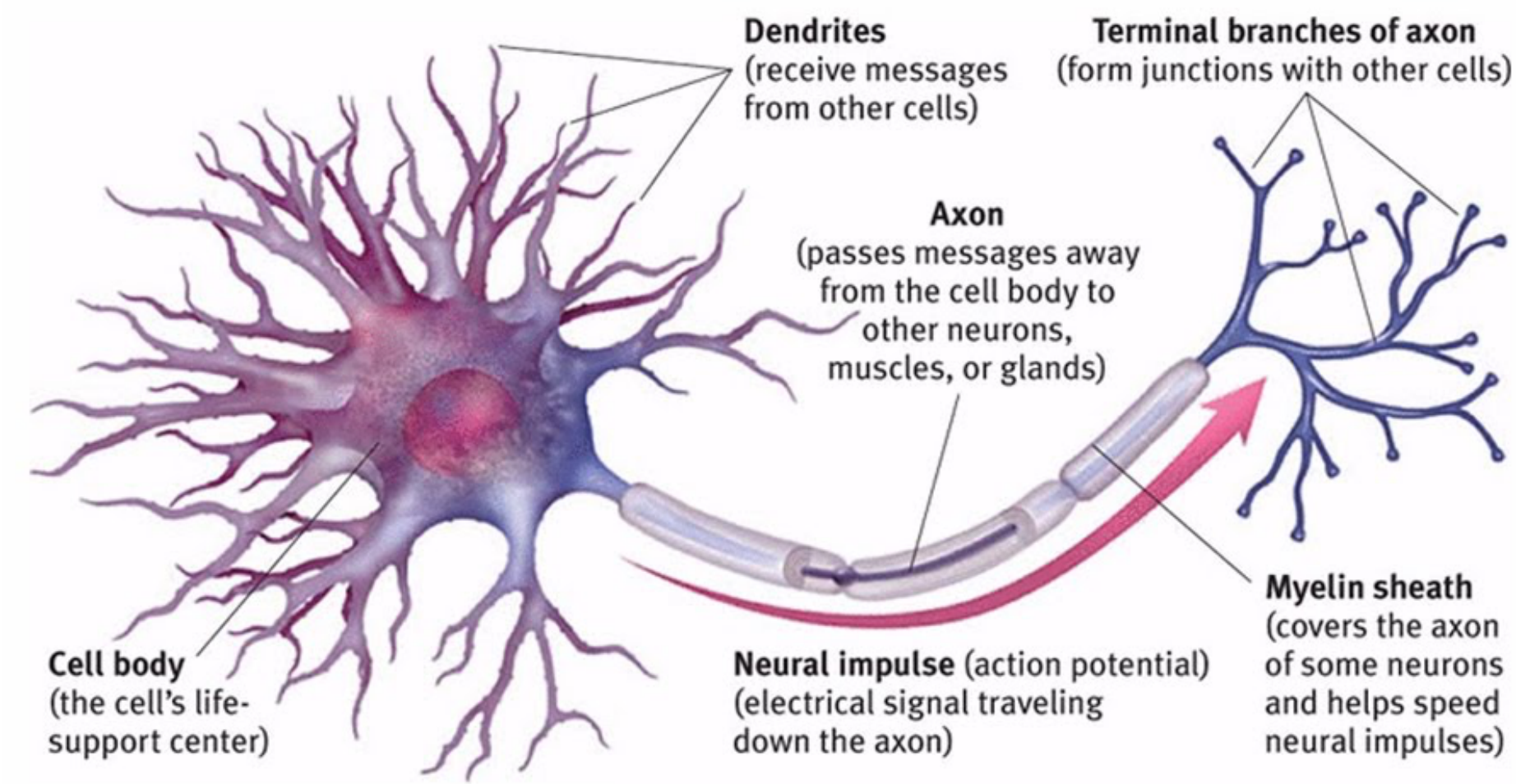
Perceptron, it was pointed out, needs no "priming." It is not nec-

Neural Networks

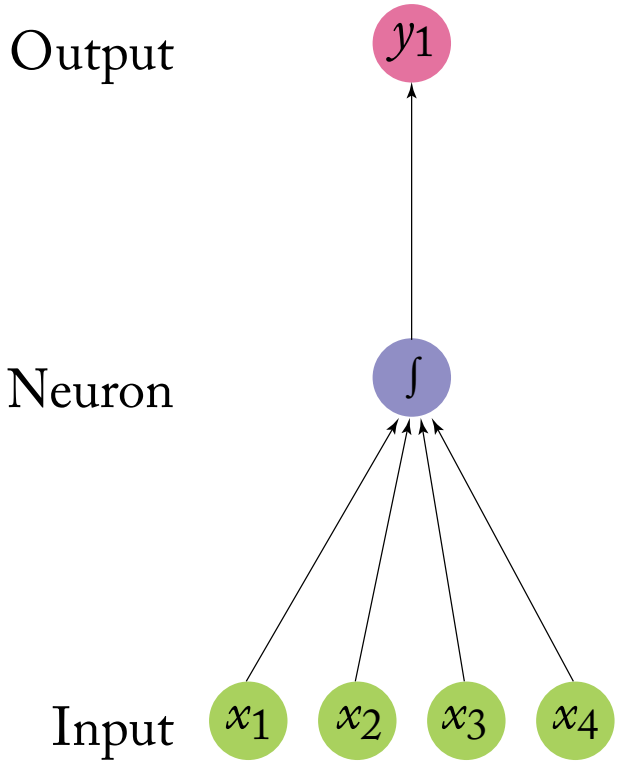
The building block of a neural network is a single computational unit. A unit takes a set of real valued numbers as input, performs some computation.

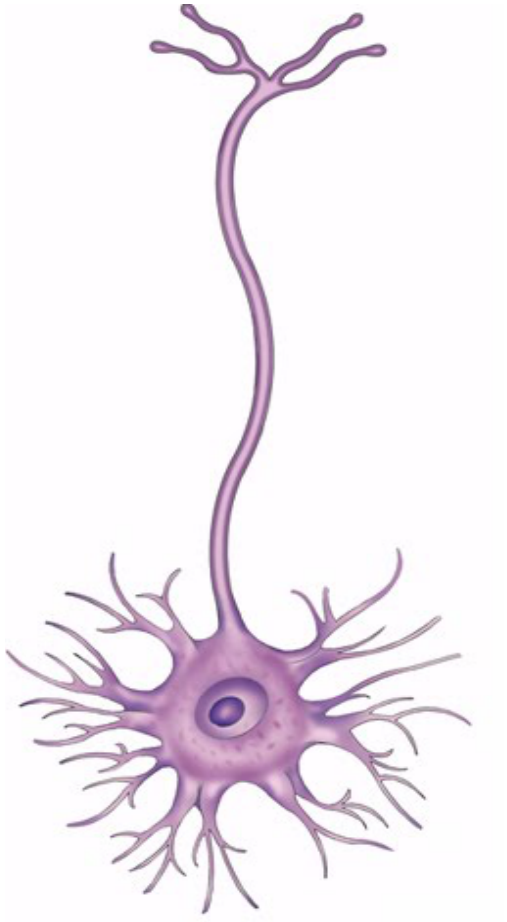


Neural Networks: A brain-inspired metaphor

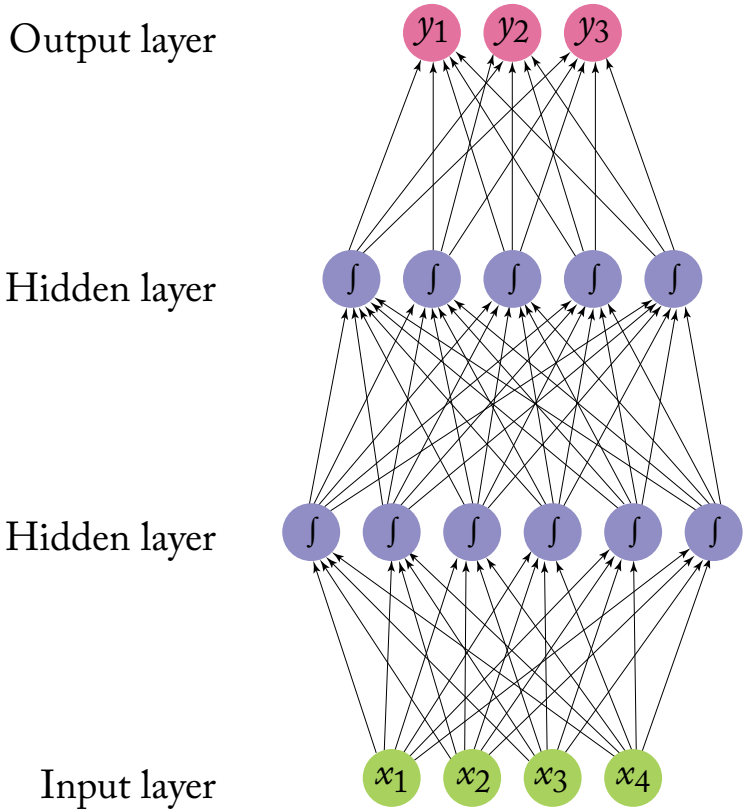


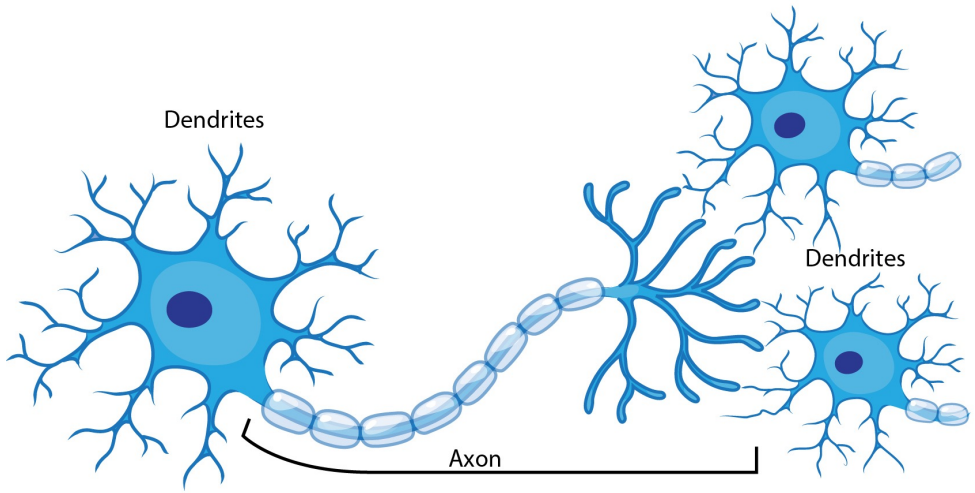
A single neuron





Neural networks





Perceptron -> Logistic Regression

Like the Perceptron, logistic regression uses a vector of **weights** and a **bias term**.

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

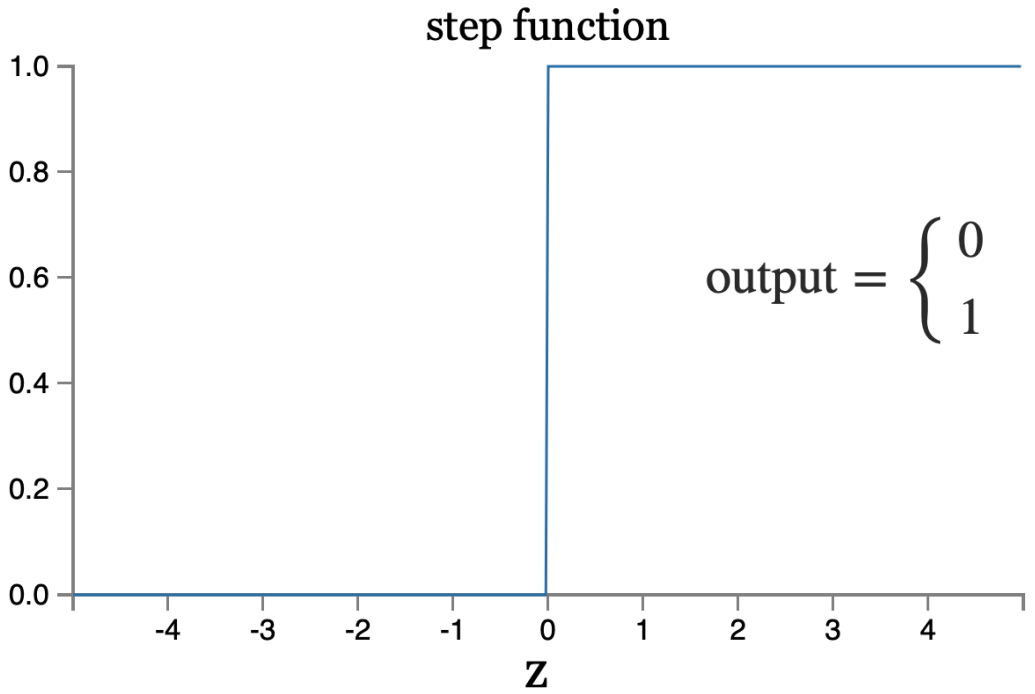
This can also be written as a dot product:

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

Instead of outputting z directly, logistic regression transforms it with the sigmoid function σ(z).

Perceptron

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

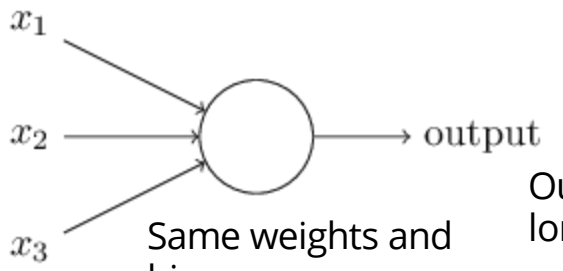


$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Sigmoid neurons

Problem: a small change in the weights or bias of any single perceptron in the network can causes the output to completely flip from 0 to 1.

Solution: sigmoid neuron



Inputs: any real-valued number

Same weights and bias as perceptron

Output is no longer just 1 or 0.

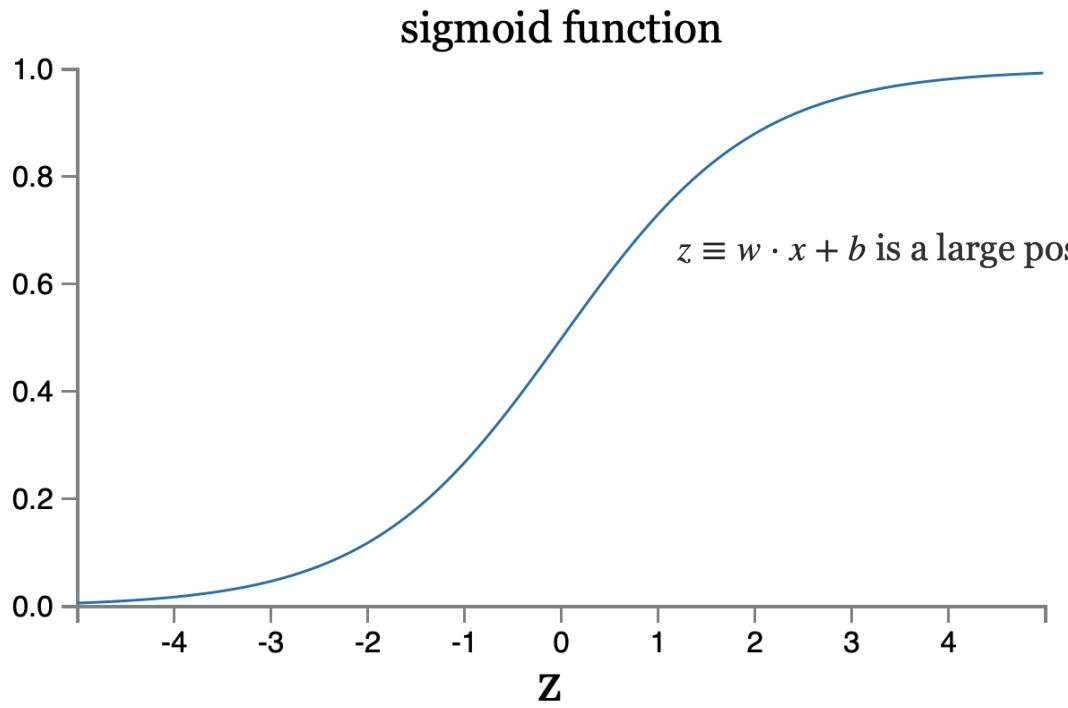
Perceptron output $= \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$

Sigmoid neuron output = $\sigma(w \cdot x + b)$

Sigmoid function: $\sigma(z) \equiv \frac{1}{1 + e^{-z}}$

Sigmoid neuron

$z \equiv w \cdot x + b$



$z \equiv w \cdot x + b$ is a large positive number. Then $e^{-z} \approx 0$

$z = w \cdot x + b$ is very negative. Then $e^{-z} \to \infty$, and $\sigma(z) \approx 0$

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

Smoothness is crucial

Smoothness of σ means that small changes in the weights w_j and in the bias b will produce a small change the output from the neuron

$$\Delta \text{output} \approx \sum_j \frac{\partial \, \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \, \text{output}}{\partial b} \Delta b$$

Δoutput is a *linear function* of the changes Δwj and Δb

This makes it easy to choose small changes in the weights and biases to achieve any desired small change in the output

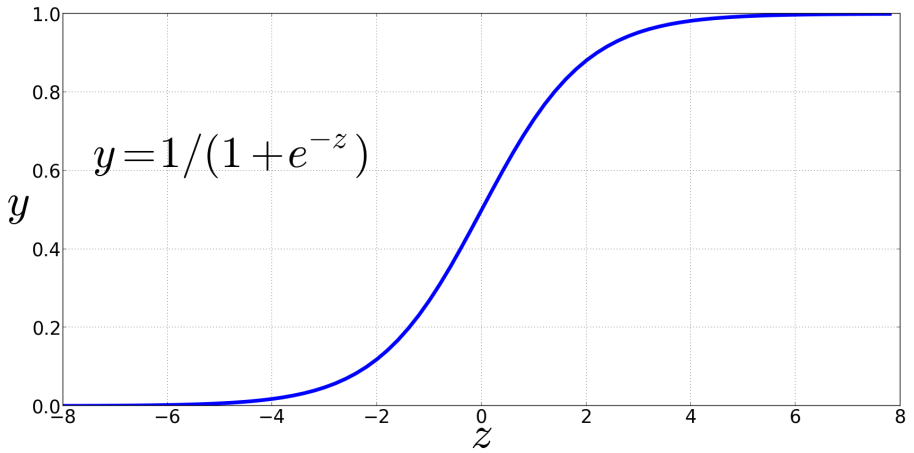
Activation Functions

Instead of directly outputting $z = w \cdot x + b$, which is a linear function of x, neuron units apply a non-linear function f to z.

The output of this function is called the **activation value** for the unit, represented by the variable **a**. The output of a neural network is called **y,** so if the activation of a node is the final output of a network then

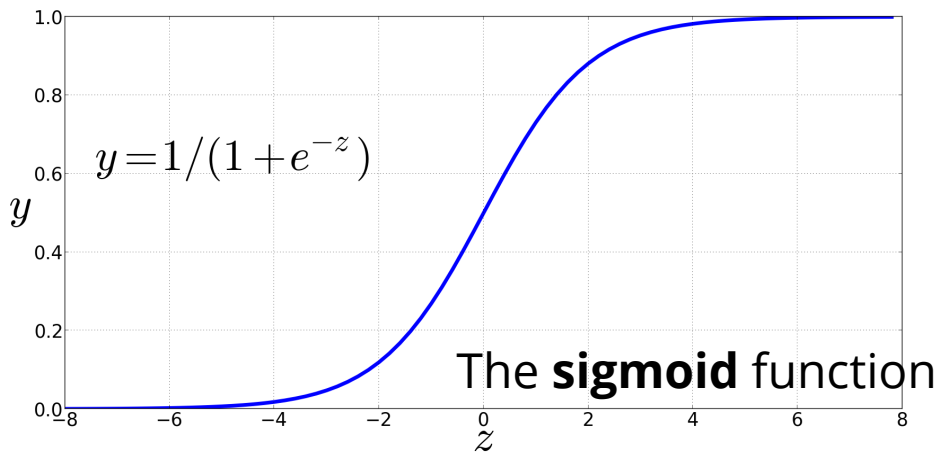
$$y = a = f(z)$$

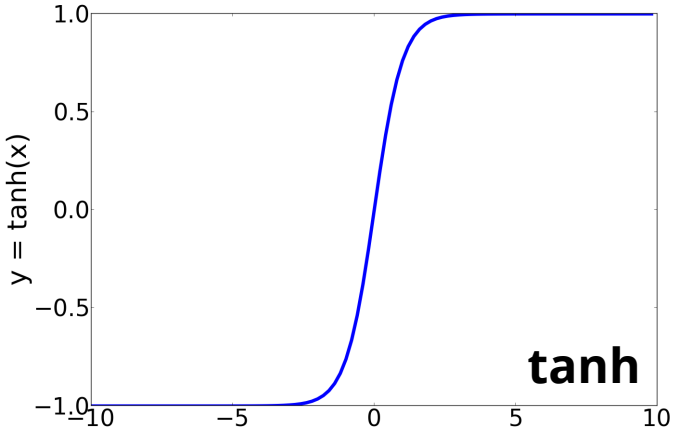
There are 3 commonly used non-linear functions used for *f:*
The **sigmoid** function
The **tanh** function
The **rectified linear unit ReLU**

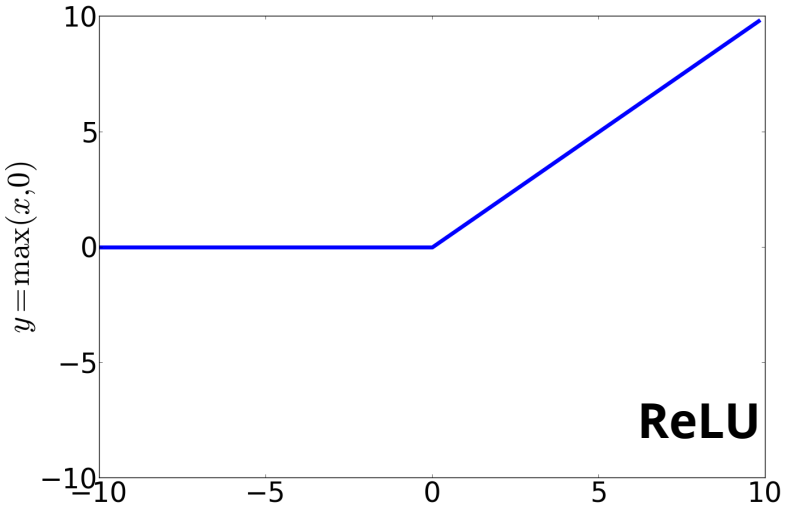


The **sigmoid** function

Activation Functions





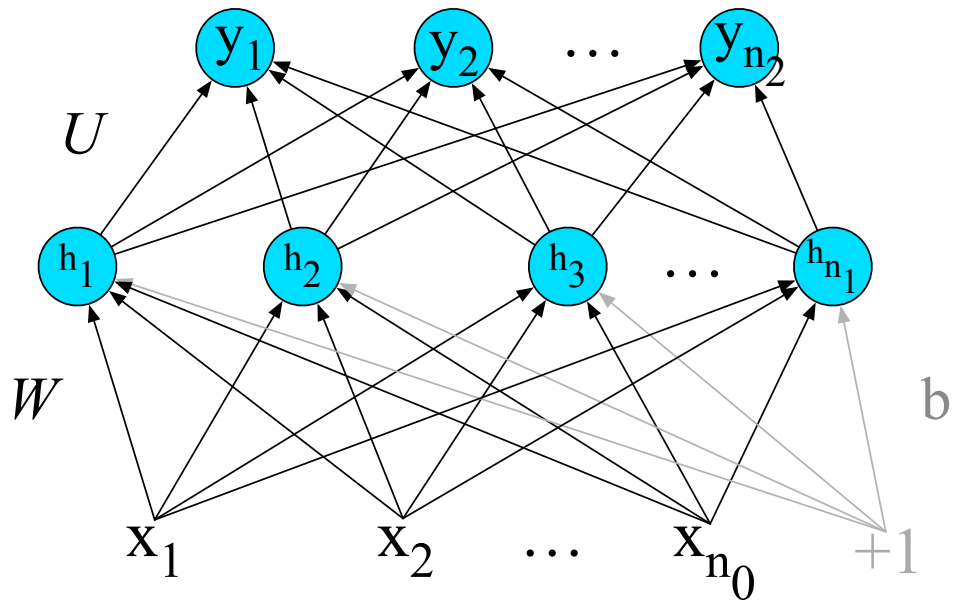


Feed-Forward Neural Network

The simplest kind of NN is the **Feed-Forward Neural Network**

Multilayer network, all units are usually **fully-connected**, and **no cycles**.

The outputs from each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers.



Layer 2 (output layer)

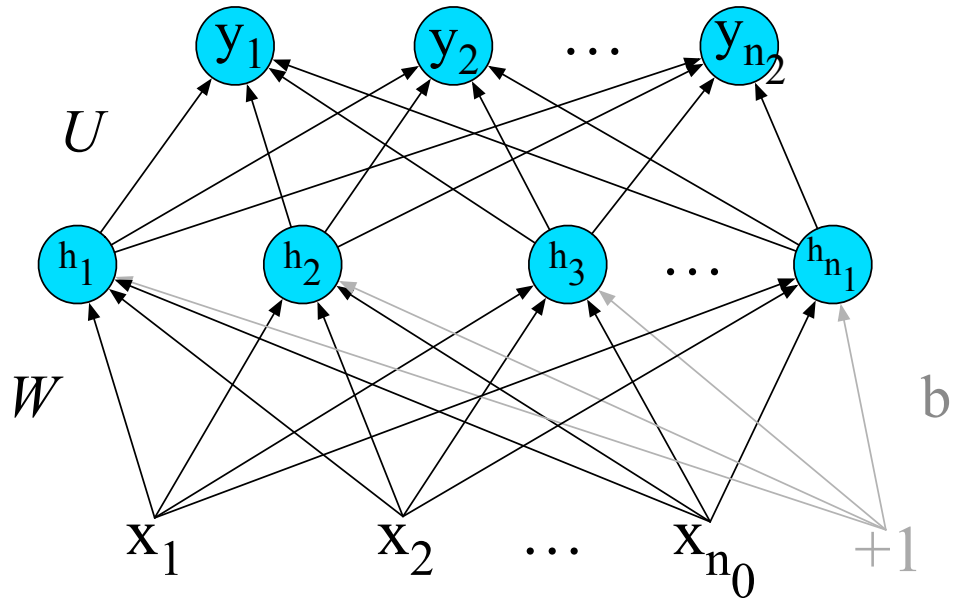
Layer 1 (hidden layer)

Layer 0 (input layer)

Equations for a feedforward network

A single hidden unit has parameters ***w*** (the weight vector) and ***b*** (the bias scalar).

We represent the parameters for the **entire hidden layer** by combining the weight vector $\mathbf{w}_i$ and bias $\boldsymbol{b}_i$ for each unit ***i*** into a single weight matrix ***W*** and a single bias vector **b** for the whole layer.



Equations for a feedforward network

The advantage of using a single matrix ***W*** for the weights of the entire layer is the hidden layer computation can be done efficiently with simple matrix operations.

The computation has three steps:

1. multiplying the weight matrix by the input vector *x*,
2. adding the bias vector *b*, and
3. applying the activation function *g* (such as Sigmoid)

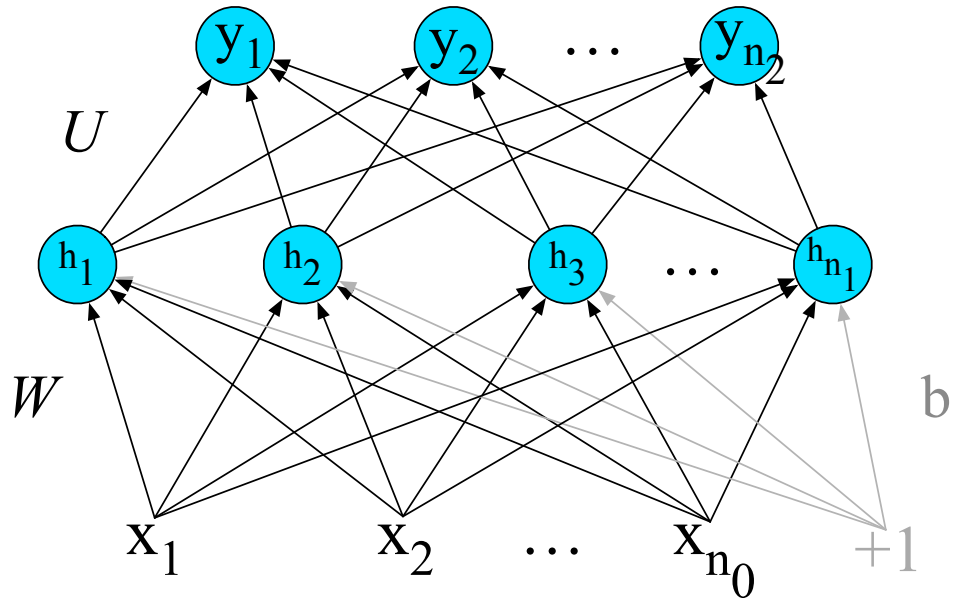
The output of the hidden layer, the vector *h*, is thus the following, using the sigmoid function σ:

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

Equations for a feedforward network

Like the hidden layer, the output layer has a weight matrix ***U***.
Its weight matrix is multiplied by its input vector (***h***) to produce the intermediate output ***z***.

$$z = Uh$$



Equations for a feedforward network

Here are the final equations for a feedforward network with a single hidden layer, which takes an input vector x, outputs a probability distribution y, and is parameterized by weight matrices W and U and a bias vector b:

$h = \sigma(Wx+b)$
$z = Uh$
$y = \text{softmax}(z)$

Like with logistic regression, softmax normalizes the output and turns it into a probability distribution.

Training Neural Nets

Like logistic regression, we want to learn the best parameters for the neural net to make its predictions $\hat{y}$ as close to possible as the gold standard labels in our training data *y*.

What do we need?

A loss function – cross-entropy loss

An optimization algorithm – gradient descent

A way of computing the gradient of the loss function – error propagation

Cross-Entropy Loss

If the neural network is a binary classifier with a sigmoid at the final layer, the loss function is exactly the same as we saw in logistic regression:

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

Cross-Entropy Loss

If the neural network is a binary classifier with a sigmoid at the final layer, the loss function is exactly the same as we saw in logistic regression:

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

For multinomial classification

$$L_{CE}(\hat{y}, y) = -\sum_{i=1}^{C} y_i \log \hat{y}_i$$

If there is only one correct answer, where the truth is $y_i=1$, then this simplifies to be

$$L_{CE}(\hat{y}, y) = -\log \hat{y}_i$$

Plugging into softmax:

$$L_{CE}(\hat{y}, y) = -\log \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

Computing the gradient

Logistic regression can be thought of as a network with just one weight layer and a sigmoid output. In that case the gradient is:

$$\frac{\partial LCE(\hat{y}, y)}{\partial wj} = (\hat{y} - y)\, xj$$

$$= (\sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b) - y)x_j$$

But these derivatives **only give correct updates for the last weight layer**!

For deeper networks, computing the gradients requires looking back through all the earlier layers in the network, even though the loss is only computed with respect to the output of the network.

Solution: **error backpropagation algorithm**

Computation Graphs

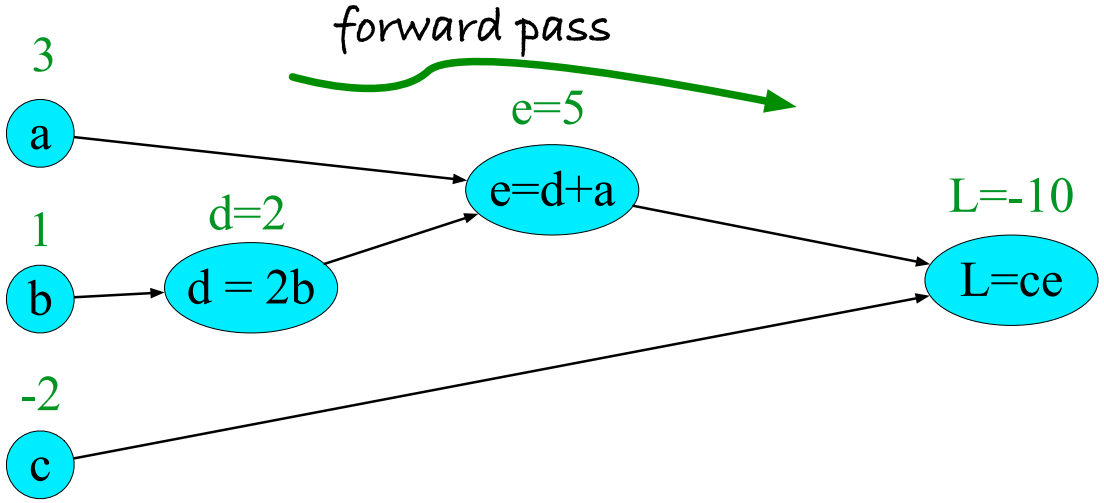
Although backpropagation was invented for neural nets, it is related to general procedure called **backward differentiation**, which depends on the notion of **computation graphs**.

A computation graph represents the process of computing a mathematical expression. The computation is broken down into separate operations. Each operation is a node in a graph.

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

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

Forward pass



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

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

inputs $a = 3$, $b = 1$, $c = -2$,

Backward differentiation

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

How do we compute the derivative of our output function L with respect to the input variables a, b, and c?

Backwards differentiation uses the **chain rule** from calculus.

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

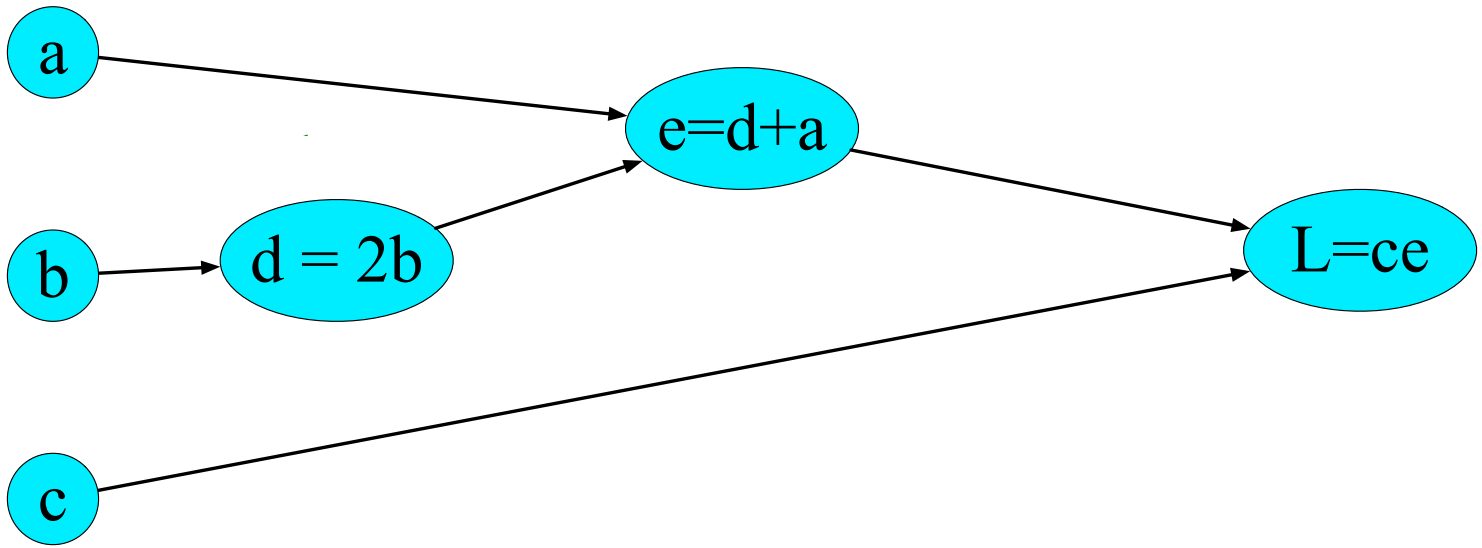
Chain rule

For a composite function $f(x) = u(v(x))$, the derivative of $f(x)$ is:

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

Similarly for, $f(x) = u(v(w(x)))$, the derivative of $f(x)$ is:

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

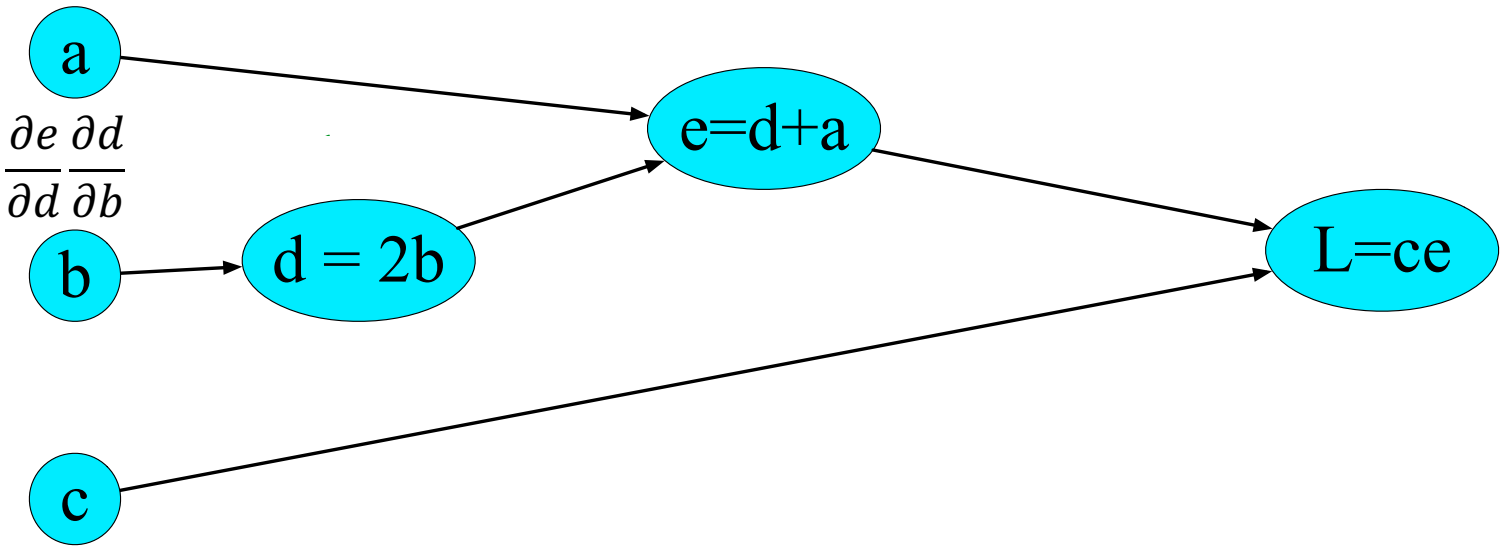


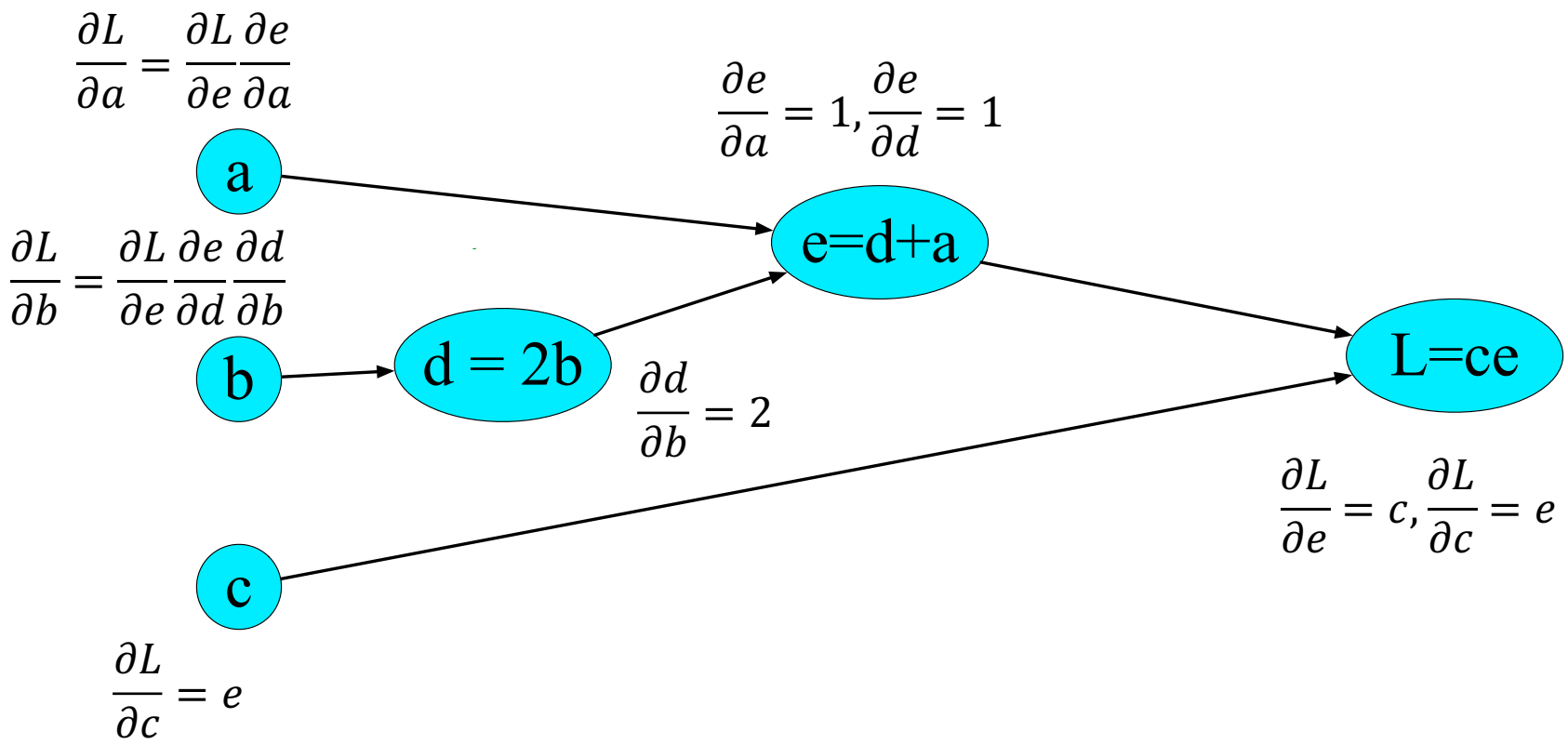
$$\frac{\partial L}{\partial c} = e$$

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

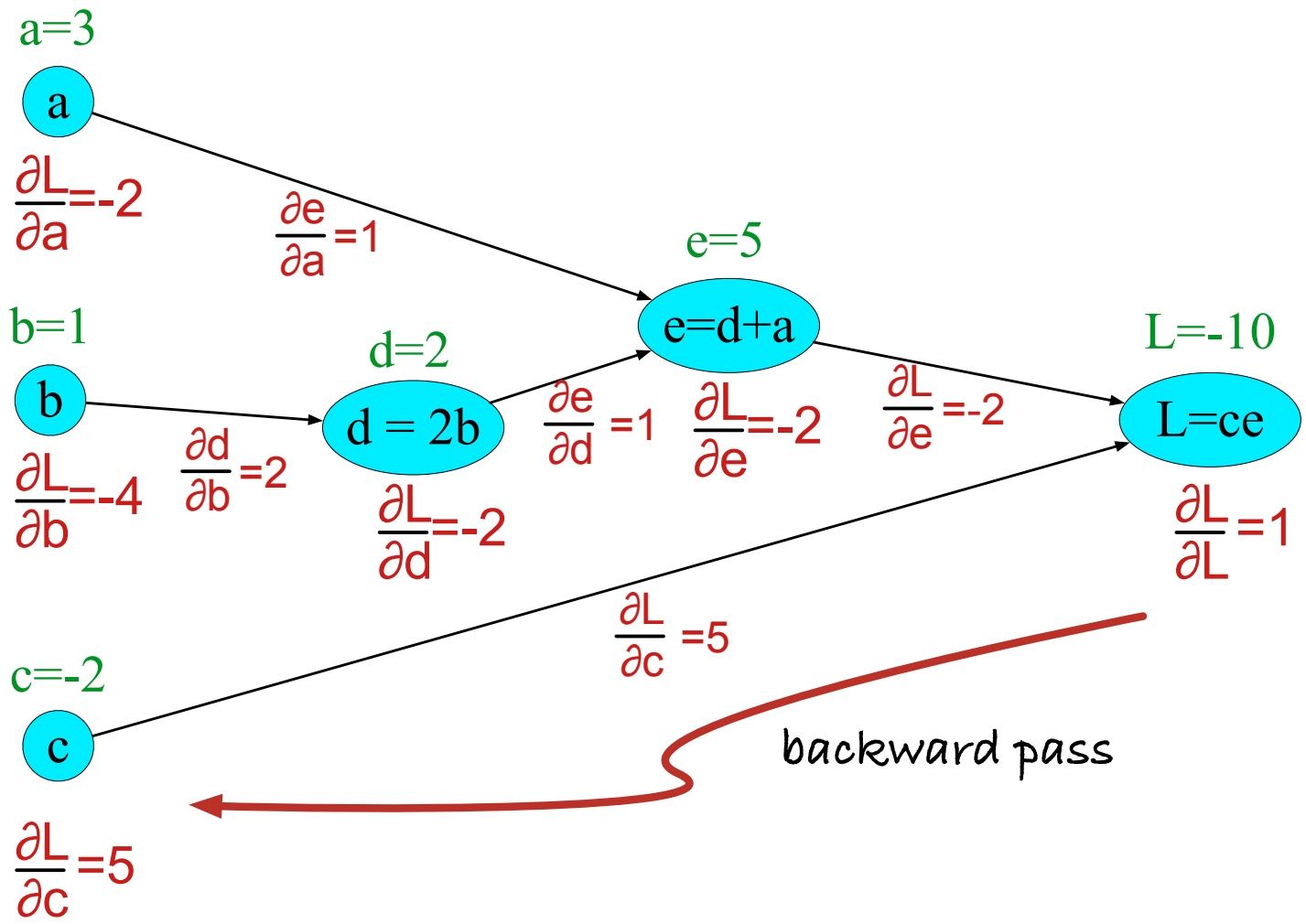
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

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

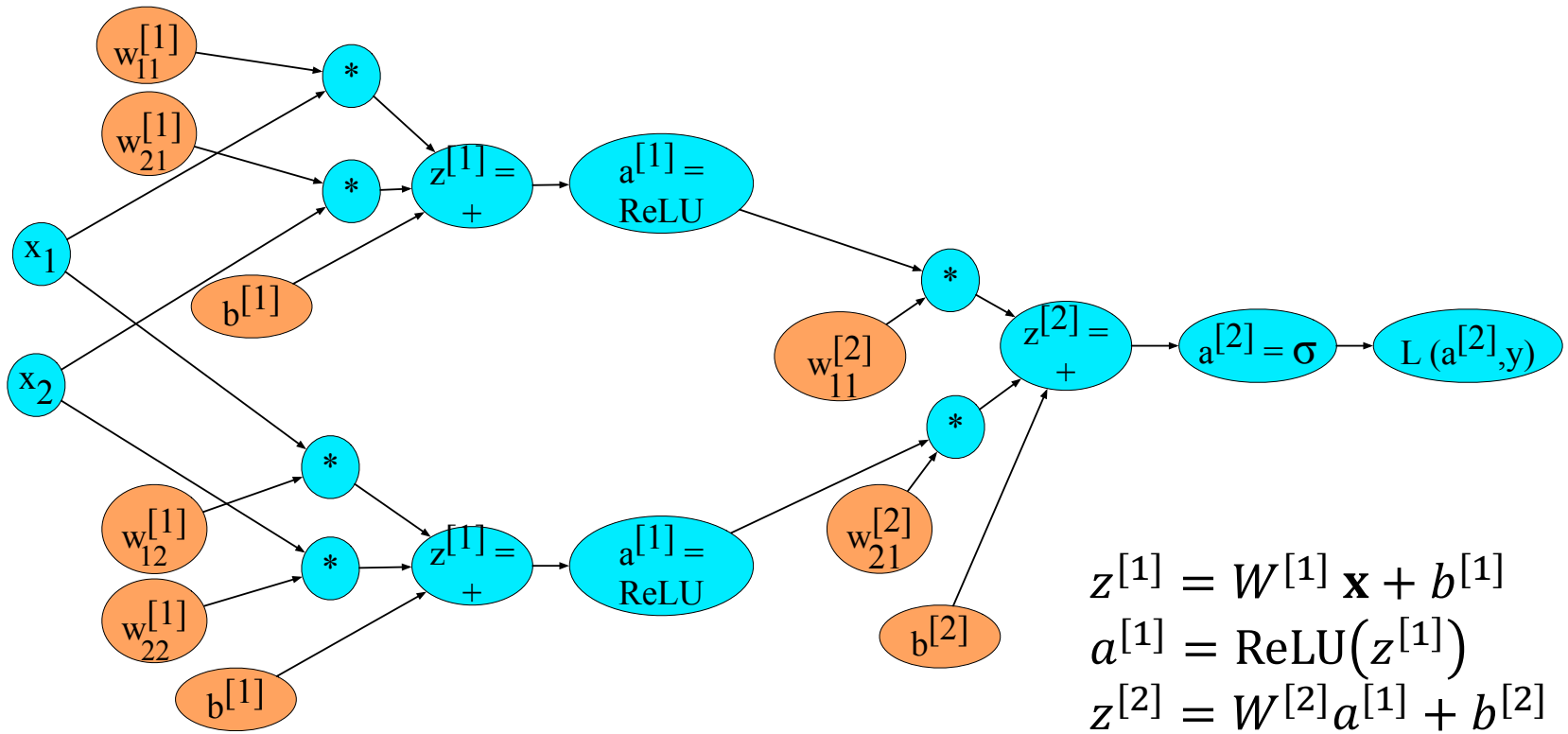



$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$
$\frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$
a
e=d+a
$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$
b
d = 2b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

Backward pass



Computation Graph for a NN



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

Summary

Like with logistic regression, we learn the best parameters for the neural net to make its predictions $\hat{y}$ as close to possible as the gold standard labels in our training data *y*.

We use:

A cross entropy loss function

Gradient descent to perform optimization

Error back propagation to compute the gradient of the loss function

Since Neural Networks combine many perceptron-like neural units, we have many more parameters in the models (weights and biases). This allows us to train systems to perform even more sophisticated tasks.

Next time: Neural Language Models