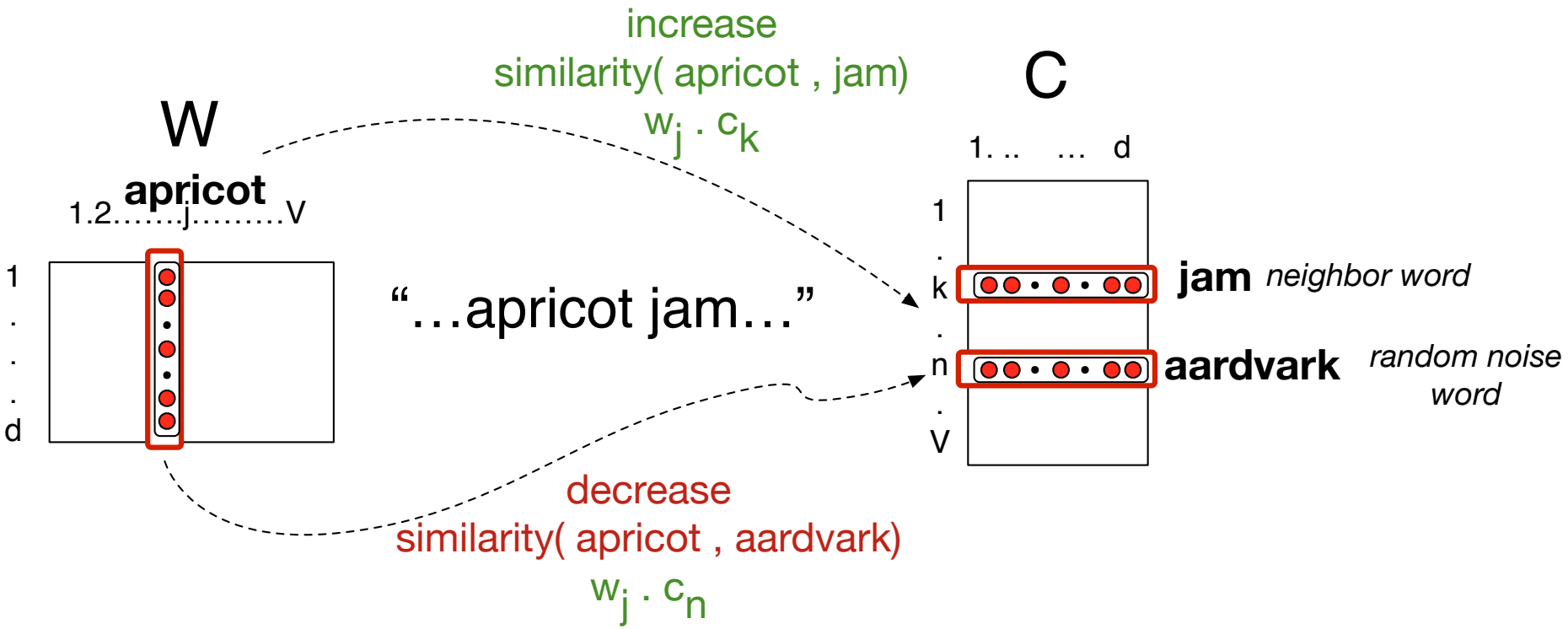
Logistic Regression

JURAFSKY AND MARTIN CHAPTER 5


increase
similarity(apricot , jam)
$w_j \cdot c_k$
W
apricot
1.2.....j.....V
1
.
.
.
d
"…apricot jam…"
C
1. .. ... d
1
.
k
.
n
.
V
jam
neighbor word
aardvark
random noise word
decrease
similarity(apricot , aardvark)
$w_j \cdot c_n$

Recap: How to learn word2vec (skip-gram) embeddings

Start with V random 300-dimensional vectors as initial embeddings

Use **logistic regression** to

- Take a corpus and take pairs of words that co-occur as positive examples
- Take pairs of words that don't co-occur as negative examples
- Train the classifier to distinguish these by slowly adjusting all the embeddings to improve the classifier performance
- Throw away the classifier code and keep the embeddings.

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 $\boldsymbol{P(y = 1|x)}$.

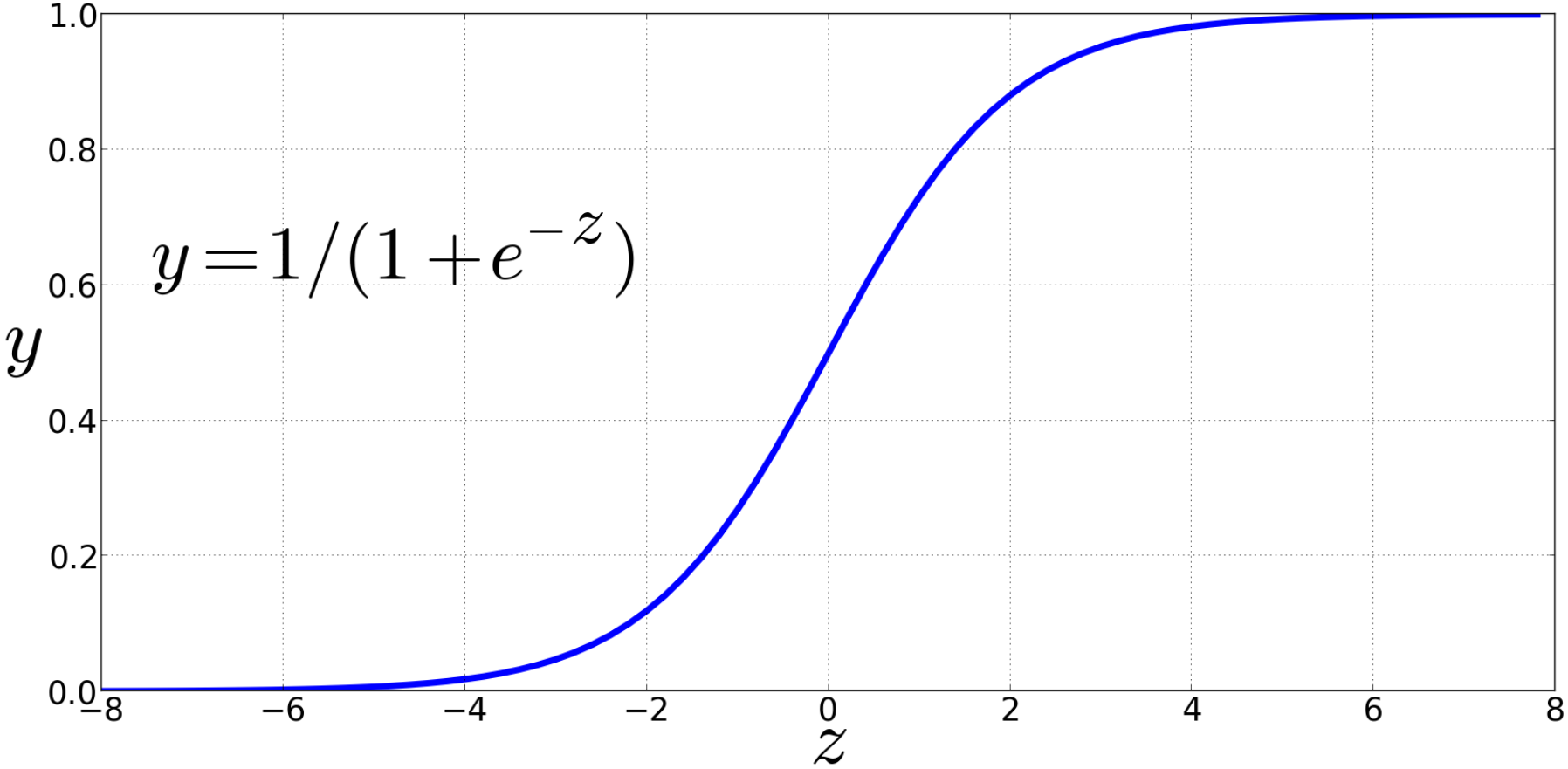
Logistic regression solves this task by learning, from a training set, a vector of **weights** and a **bias term**.

$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b$$

We can also write this as a dot product:

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

Sigmoid function



Probabilities

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

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?

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

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 1^{st} 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 1^{st} 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 1^{st} 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 1^{st} 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 1^{st} 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 .

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	

$$z = \left(\sum_{i=1}^{n} w_i x_i \right) + b$$

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	4.15	0.7	2.905
b	bias	1	0.1	.1

$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b$$

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		0.7	2.905
b	bias		0.1	.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 $\mathbf{p(y|x)} = \hat{y}$.

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

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

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

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

$$
\begin{aligned}
\log p(\text{training labels}) &= \log \prod_{i=1}^{m} p(y^{(i)} | x^{(i)}) \\
&= \sum_{i=1}^{m} \log p(y^{(i)} | x^{(i)}) \\
&= -\sum_{i=1}^{m} L_{CE}(\hat{y}^{(i)}, y^{(i)})
\end{aligned}
$$

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

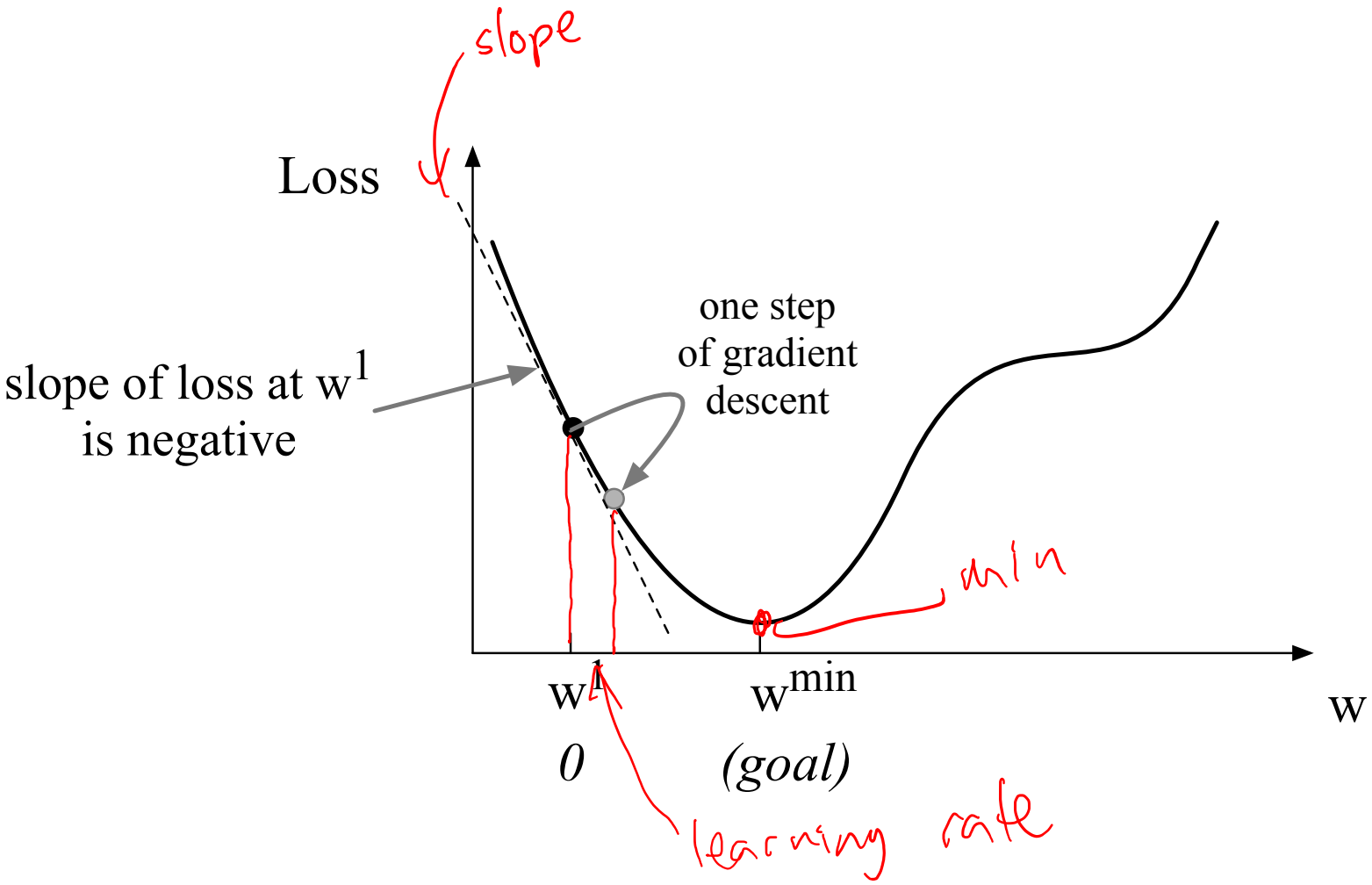
Global v. Local Minimums

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.

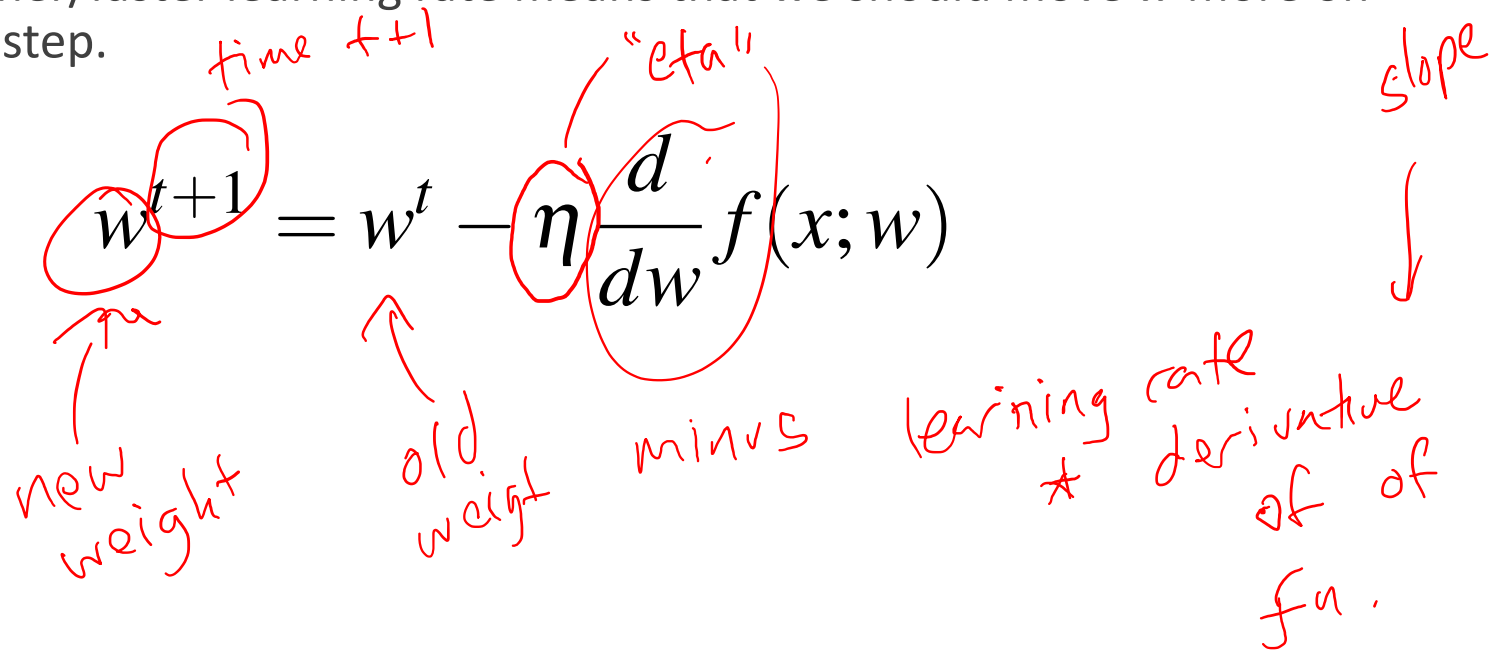
Iteratively find minimum



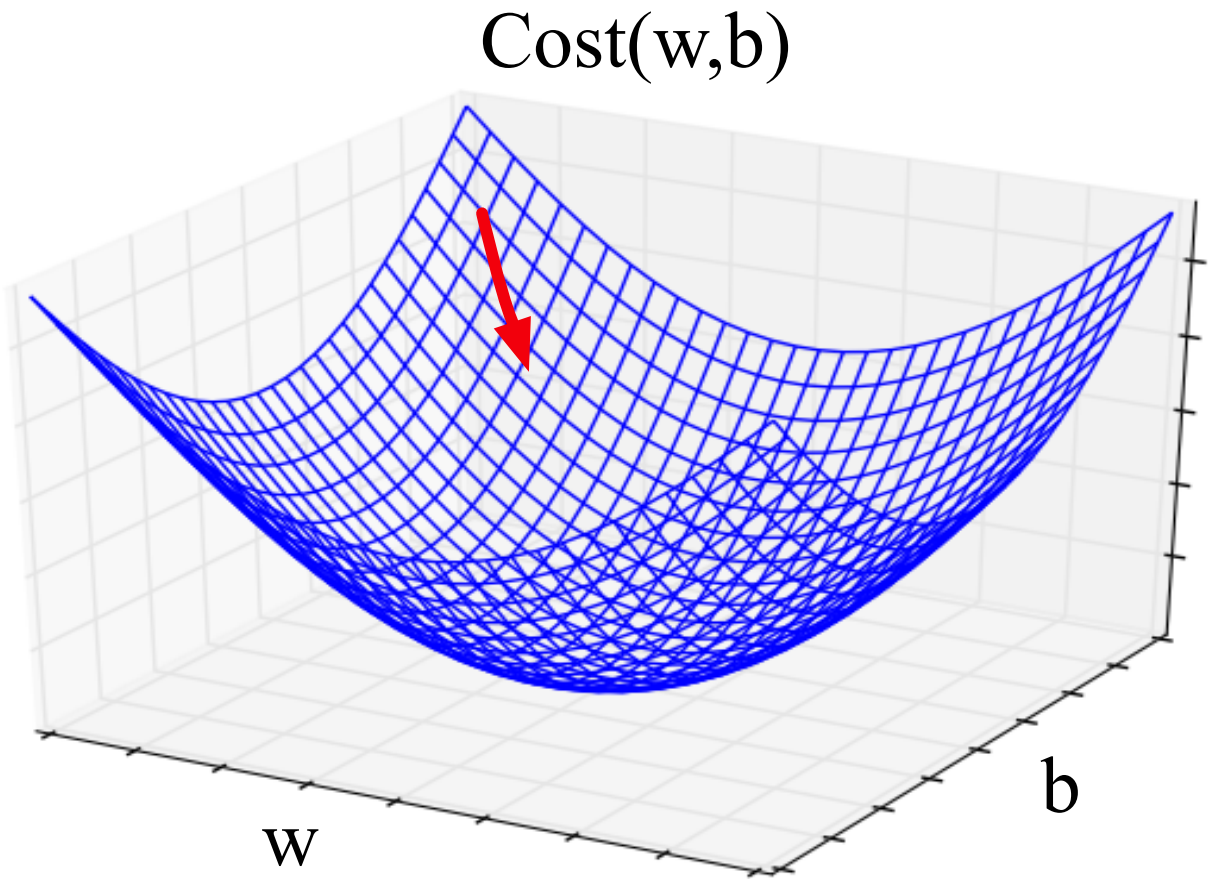
How much should we update the parameter by?

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.



Many dimensions



Updating each dimension w_i

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

$\hat{y}$

our model's prediction for input x given parameters θ

The final equation for updating θ based on the gradient is

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

η: 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:

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

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

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

truth

our model's prediction

* value of feature j

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

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

This is what we want to minimize!!

The Gradient

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

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

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

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

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 $x^{(1)}, x^{(2)}, ..., x^{(n)}$
y is the set of training outputs (labels) $y^{(1)}, y^{(2)}, ..., y^{(n)}$

$\theta \leftarrow 0$
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).

$$x_1 = 3 \quad \text{(count of positive lexicon words)}$$

$$x_2 = 2 \quad \text{(count of positive negative words)}$$

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 one observation vector *x* for a single weight w_j is

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

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

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

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:

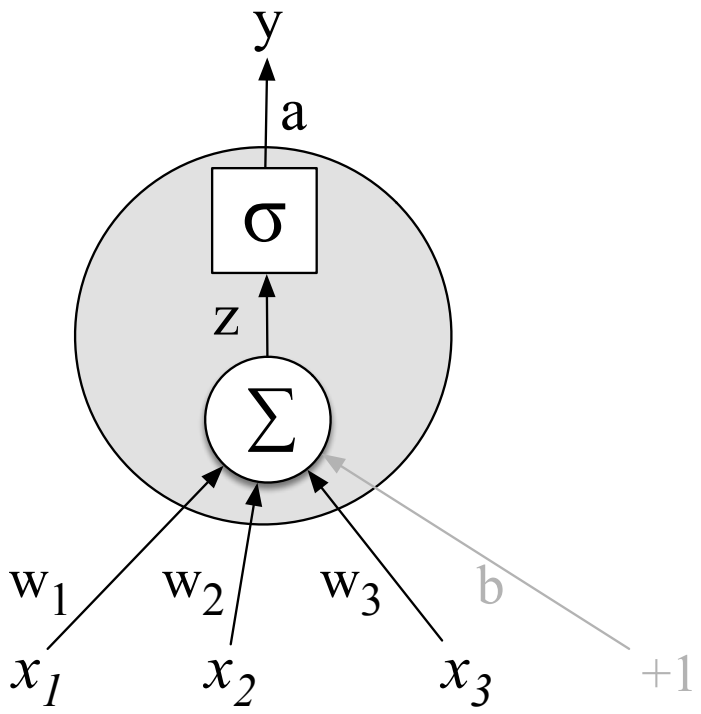
$$\theta^1 = \begin{bmatrix} w_1 \\ w_2 \\ b \end{bmatrix} - \eta \begin{bmatrix} -1.5 \\ -1.0 \\ -0.5 \end{bmatrix} = \begin{bmatrix} .15 \\ .1 \\ .05 \end{bmatrix}$$

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

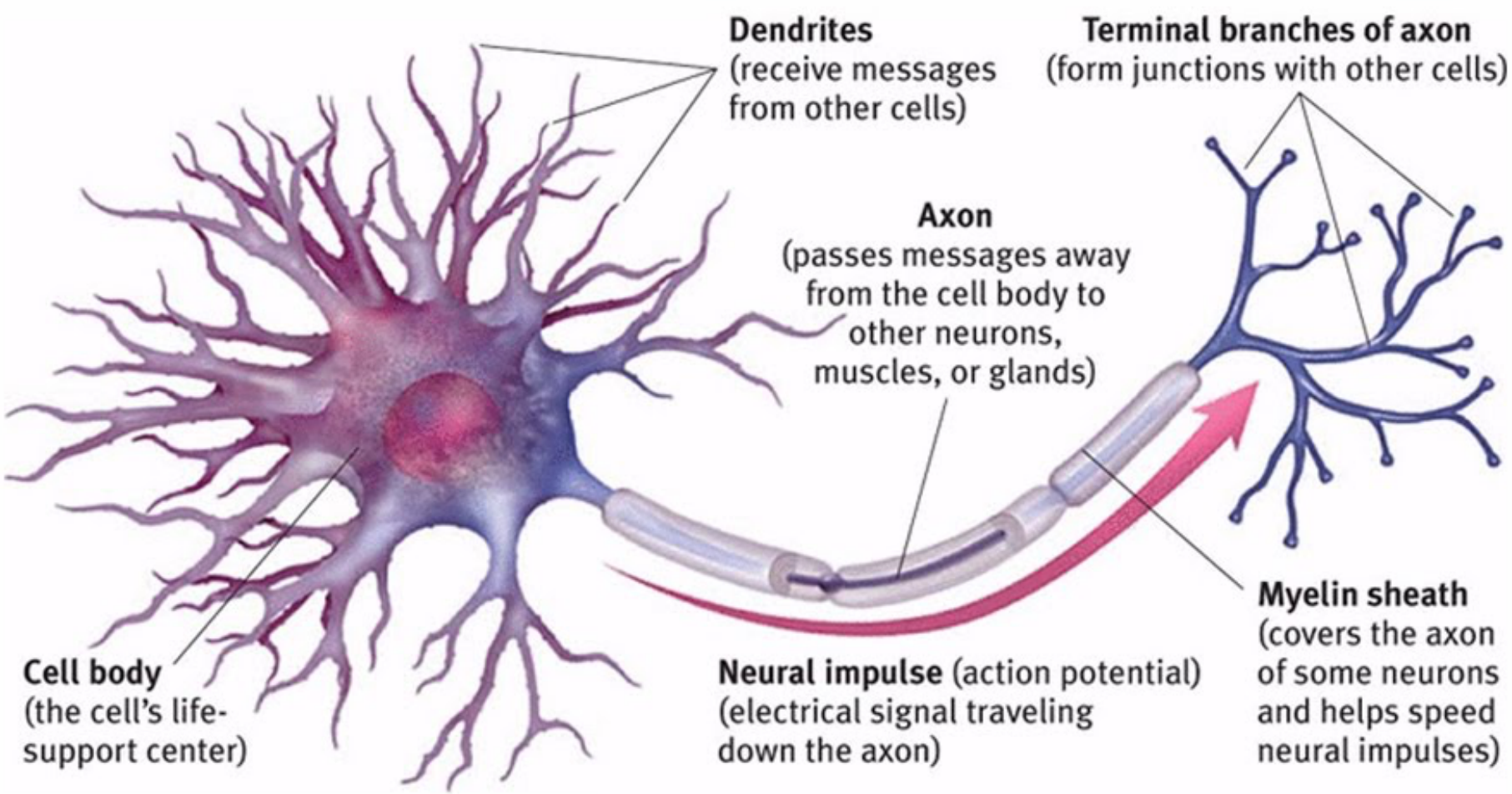
$w_1 = 0.15$, $w_2 = 0.1$, and $b = .05$

Neural Networks

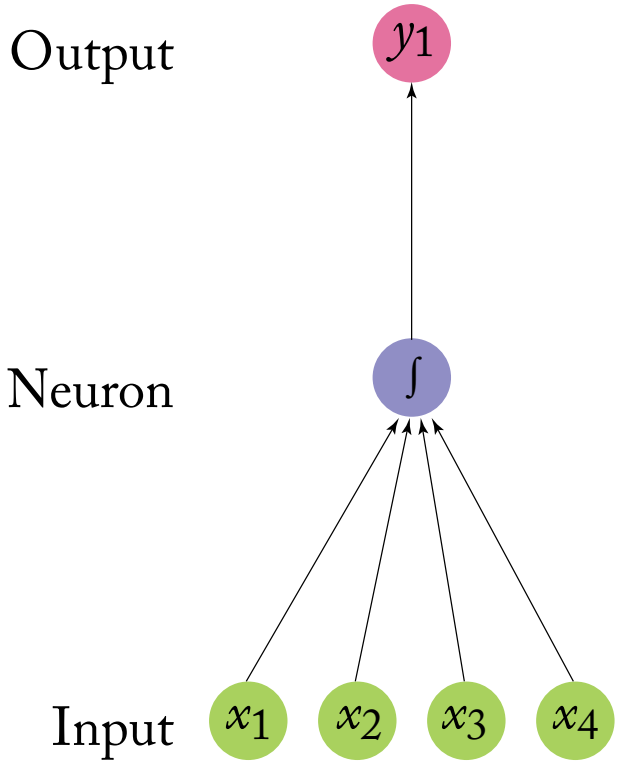
The building block of a neural network is a single computational unit. A unit takes a set of real valued numbers as input, performs some computation.

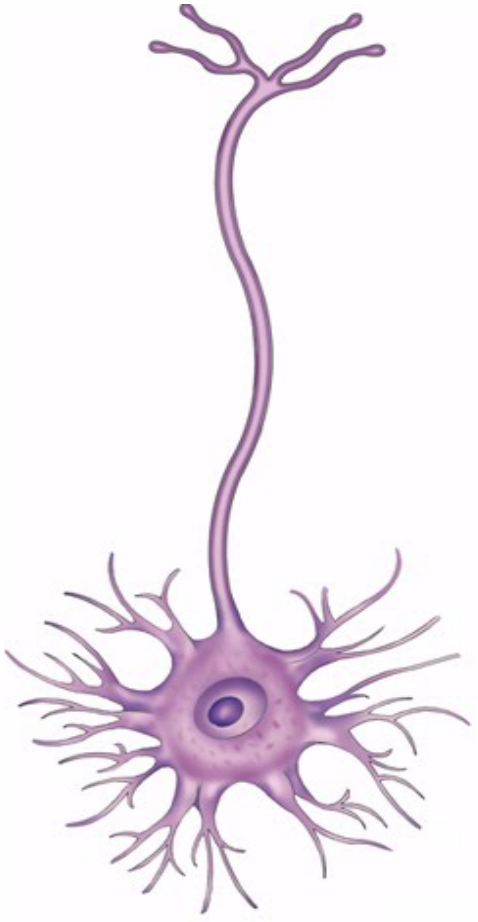


Neural Networks: A brain-inspired metaphor

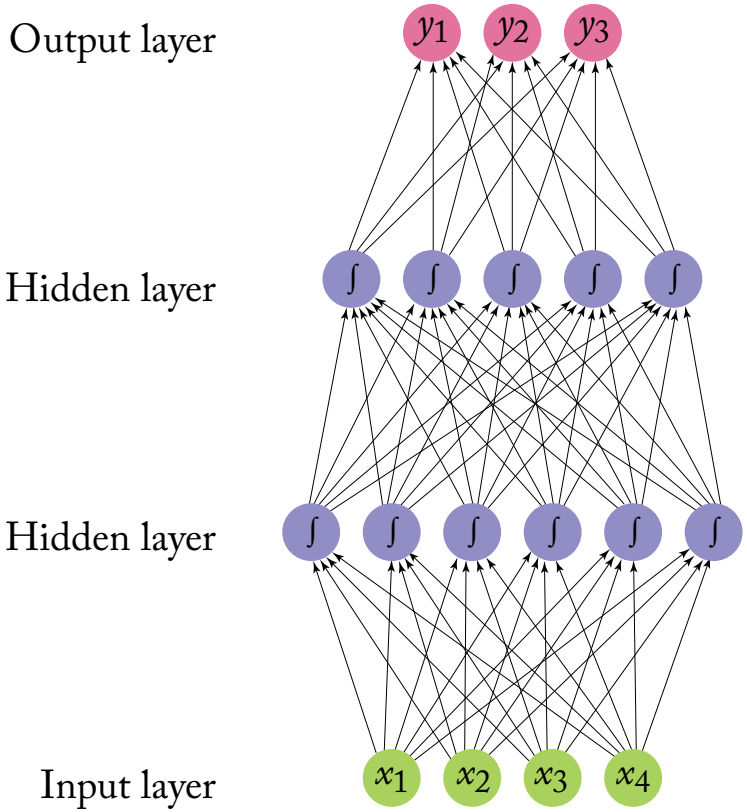


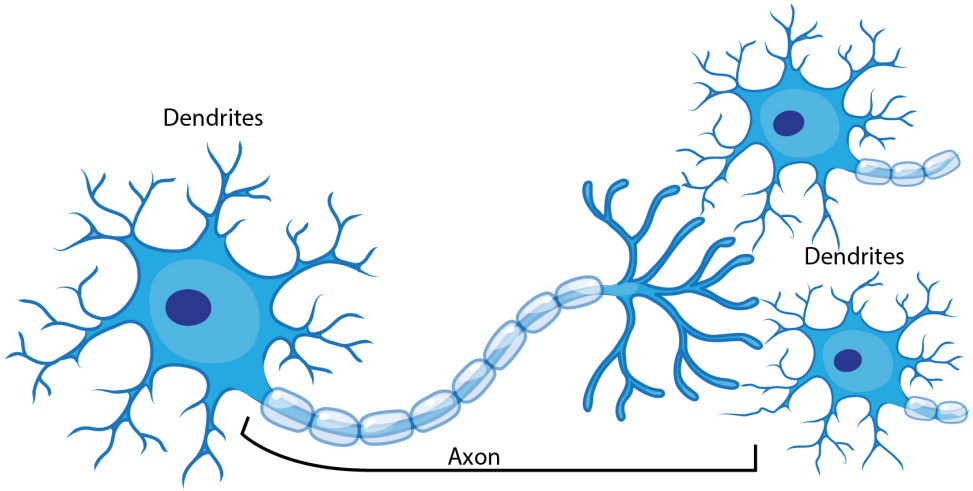
A single neuron





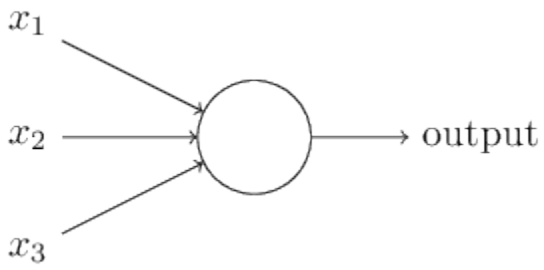
Neural networks

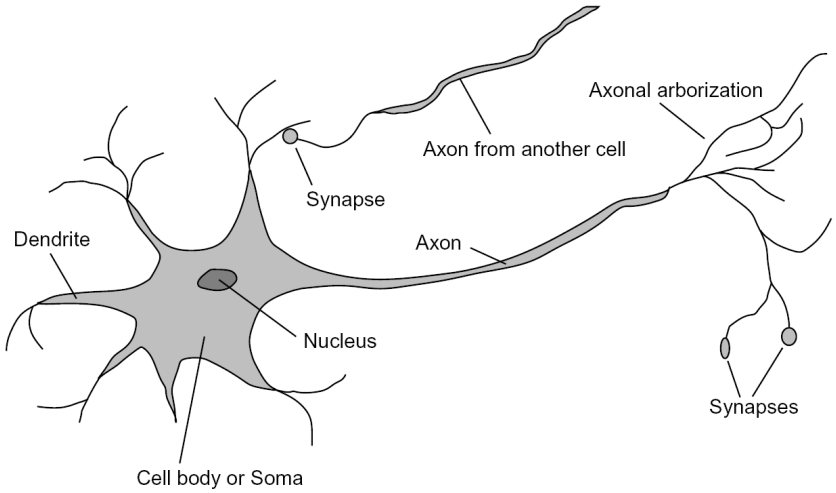




Review: Perceptron

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

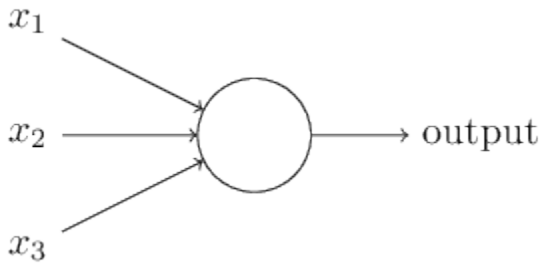




Review: Perceptron

Perceptron has inputs, $x_1, x_2, .. x_N$, and weights $w_1, w_2, .. w_N$

The perceptron outputs 0 or 1, based on the weighted sum is less than or greater than a threshold value

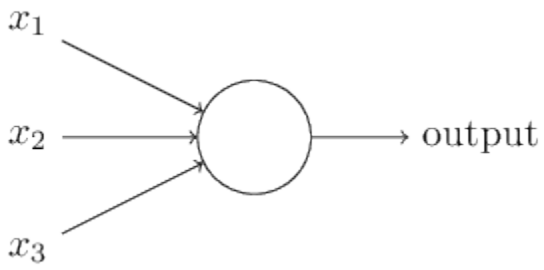


$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

Perceptrons for decision making

We can think about the perceptron as a device that makes decisions by weighing up evidence.

Example: Suppose there's a cheese festival in your town. You like cheese.

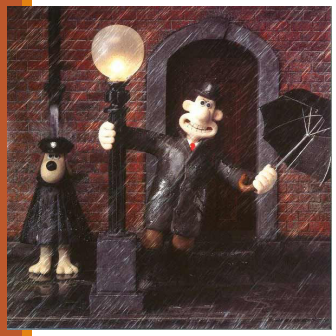


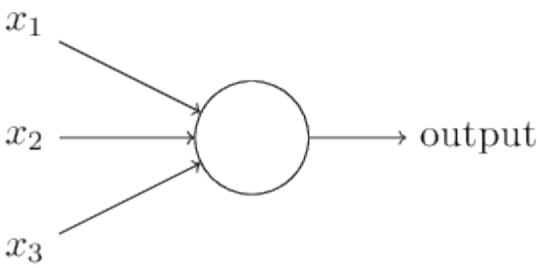
Perceptrons for decision making

You might use 3 factors to decide whether to go.

1. Is the weather good?
2. Can your loyal companion come with you?
3. Is the festival near public transit?

These can be the binary input values to a perceptron



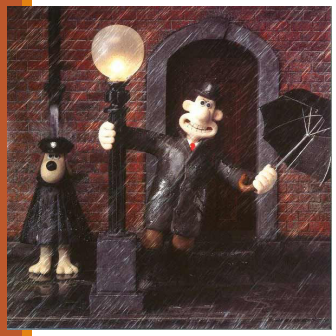


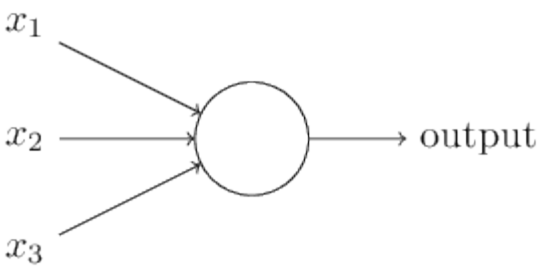
Perceptrons for decision making

By varying weights and the threshold we get different models of decision making

Example 1: $w_1 = 6$ $w_2 = 2$ $w_3 = 2$, threshold = 5

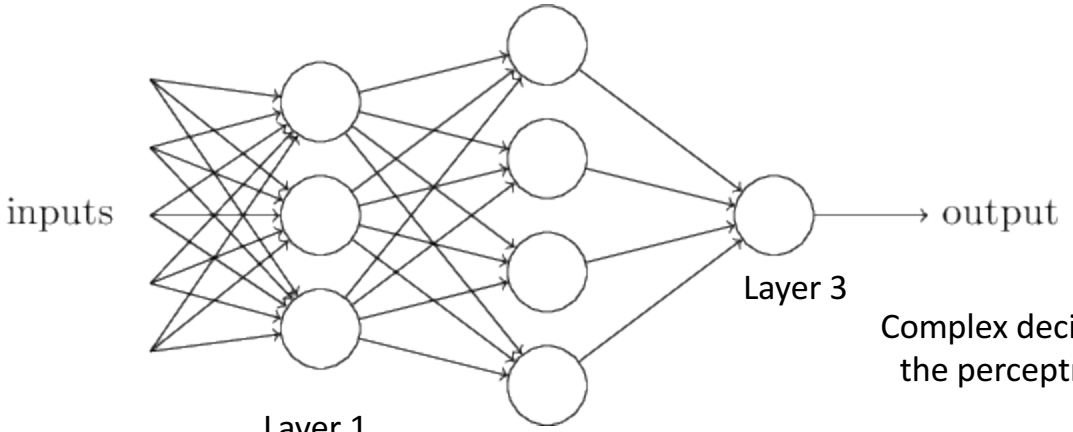
Example 2: $w_1 = 6$ $w_2 = 2$ $w_3 = 2$, threshold = 3





Perceptrons for decision making

- A complex network of perceptrons could make quite subtle decisions:



1st layer makes 3 simple decisions by weighing the input evidence

2nd layer makes a decision by weighing up the results from the 1st layer of decision-making

Complex decisions can be made by the perceptron in the third layer

Weights, bias and dot products

- Two notational changes simplify the way that perceptrons are described.
- The first change is to replace the weighted sum as a dot product

$$w \cdot x \equiv \sum_j w_j x_j$$

- The second change is to move the threshold to the other side of the inequality, and to replace it by a *bias*, b≡−threshold

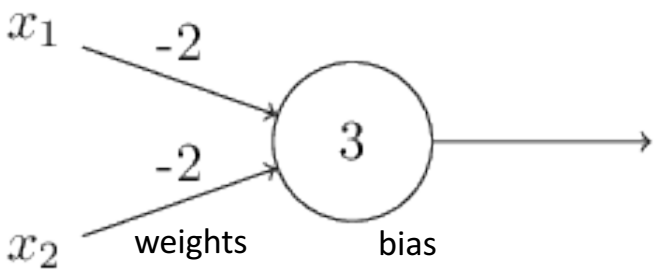
$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

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

Decision making OR logical functions

Perceptrons can be used is to compute logical functions like AND, OR and NAND

Example:



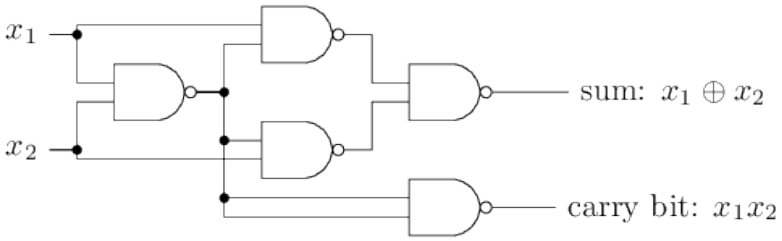
Input	Weighted sum	Output
00	-2*0 + -2*0 +3 = 3	1
10 or 01	-2*1 + -2*0 +3 = 1	1
11	-2*1 + -2*1 +3 = -1	0

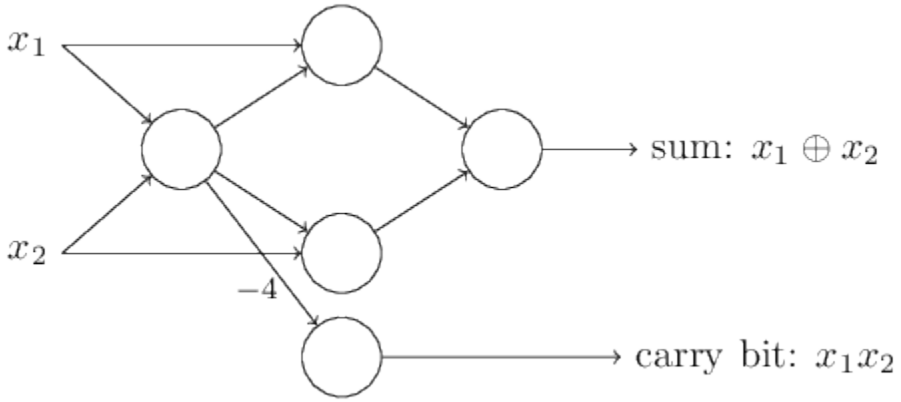
Logical functions

Networks of perceptrons to compute *any* logical function

We can build any computation up out of NAND gates.

For example, a circuit which adds two bits x_1 and x_2





All unlabeled weights are -2, all biases =3.

Power of Perceptrons

Networks of Perceptrons are universal for computation, like NAND gates

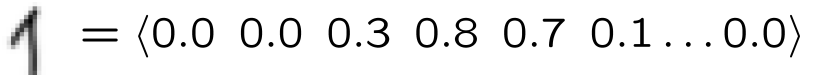
Perceptrons can be as powerful as any other computing device!

We can devise *learning algorithms* to automatically tune the weights and biases of a network of artificial neurons

Instead of laying out a circuit of NAND and other gates, neural networks can simply learn to solve problems

Learning to classify digits

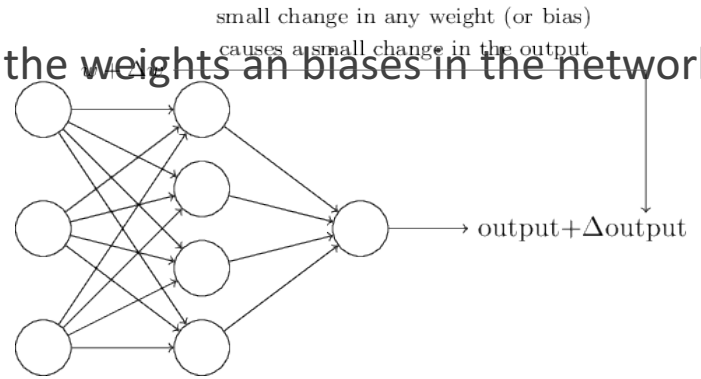
Input: image encoded input as a vector of intensities:

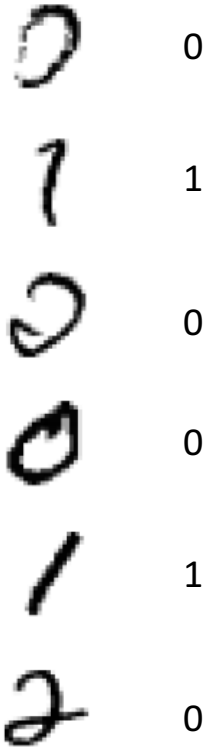


Output: is this a one or not?

Goal: set the parameters of the network to correctly classify the digits

Learning = changing the weights an biases in the network.





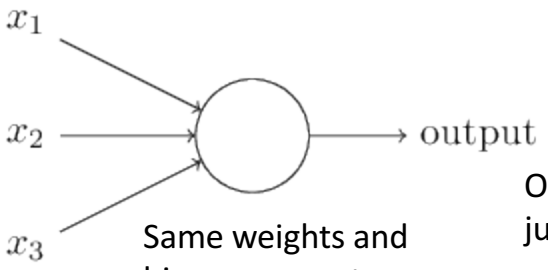
Sigmoid neurons

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

Perceptron

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



Same weights and bias as perceptron

Output is no longer just 1 or 0.

Inputs: any real-valued number

Sigmoid neuron

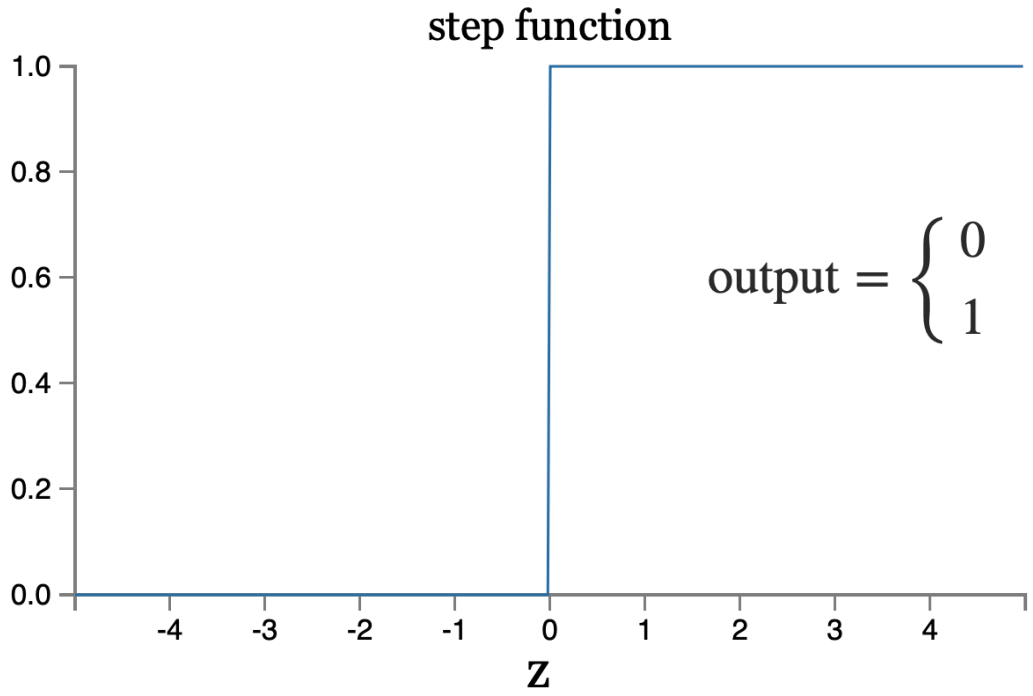
output = $\sigma(w \cdot x + b)$

Sigmoid function:

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

Perceptron

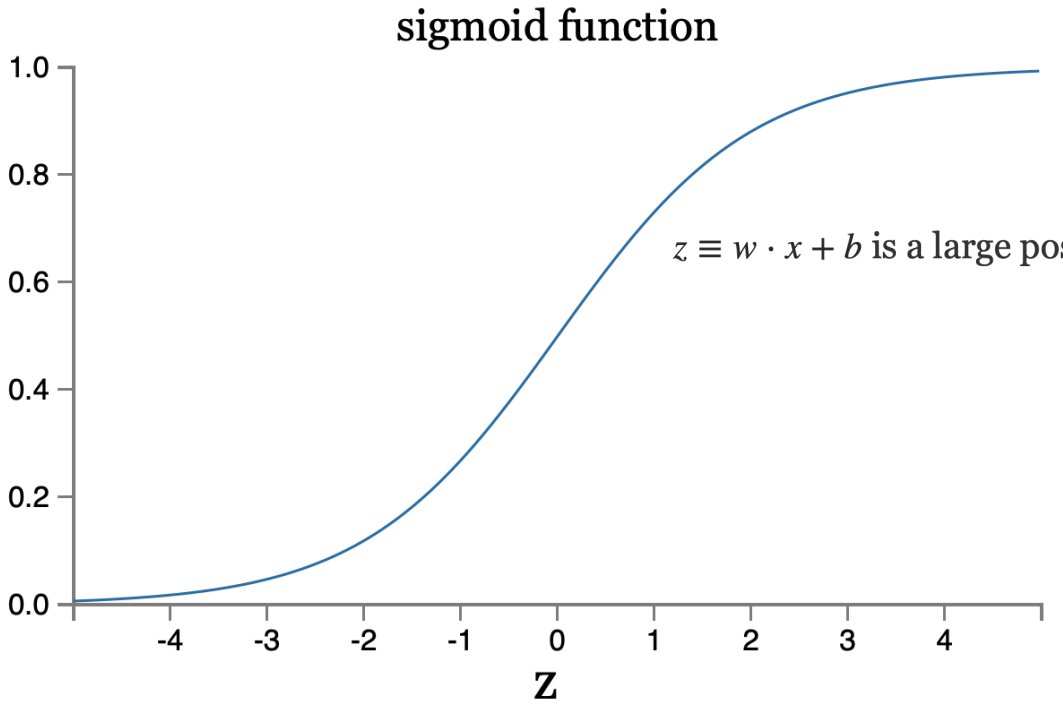
$z \equiv w \cdot x + b$



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

Sigmoid 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 Δw_j and Δb

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

Next time: Neural Nets