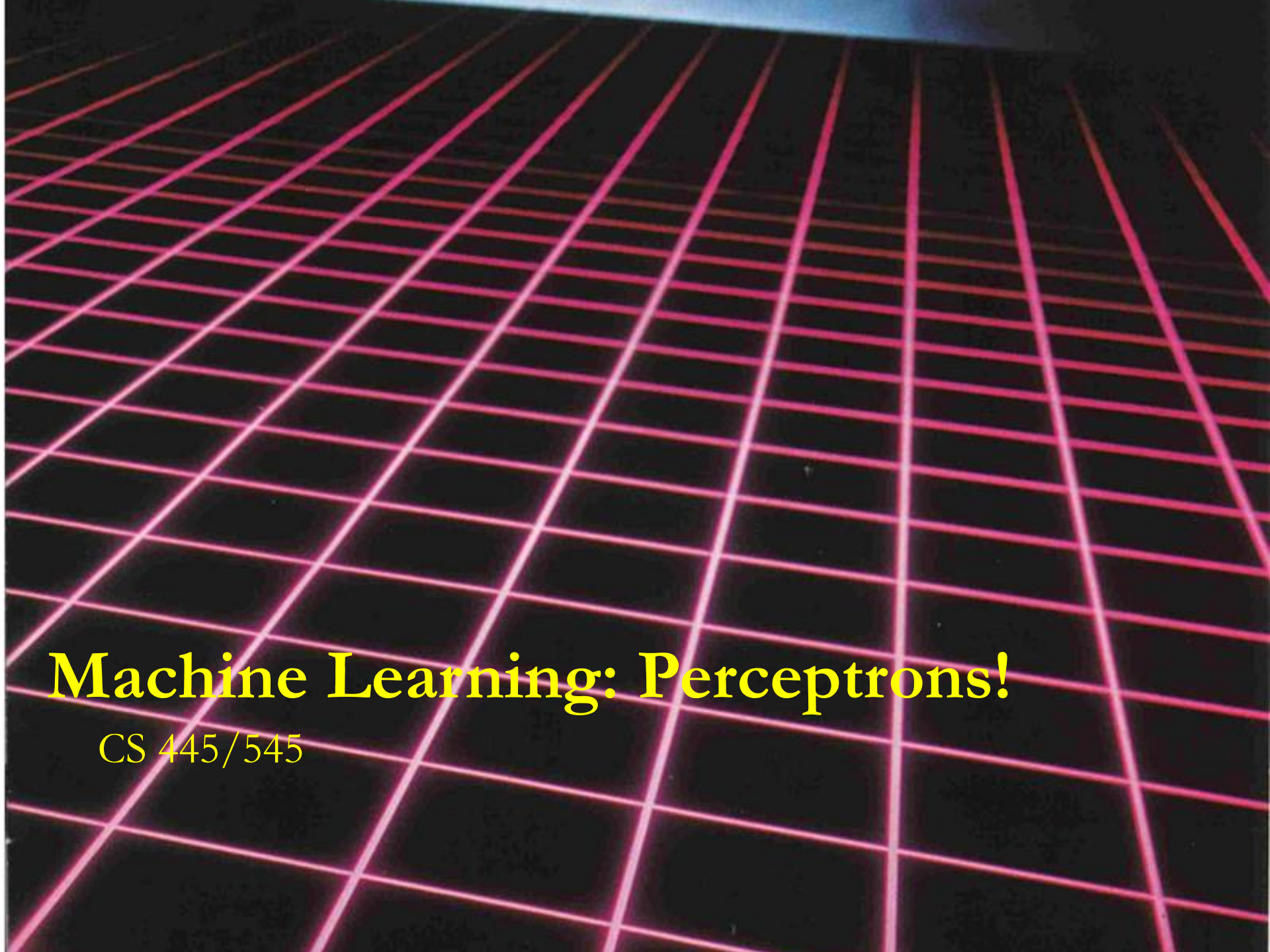


Machine Learning: Perceptrons!

CS 445/545

500 Service Unavailable Error - Windows Internet Explorer

http://www.amazon.com/Disciplined-Entrepreneurship-Steps-Successful-Startup/dp/1118692284/ref=sr_1_1?ie=UTF8&q

File Edit View Favorites Tools Help

Favorites | Editorial Links | News | Search Engines | Best of the Web | Internet Start | Suggested Sites | Web Interface Log In

500 Service Unavailable Error

amazon.com

Oops!

We're very sorry, but we're having trouble doing what you just asked us to do. Please give us another chance--click the Back button on your browser and try your request again. Or start from the beginning on our homepage.

500 Service Unavailable Error - Windows Internet Explorer

http://www.amazon.com/Disciplined-Entrepreneurship-Steps-Successful-Startup/dp/1118692284/ref=sr_1_1?ie=UTF8&q

File Edit View Favorites Tools Help

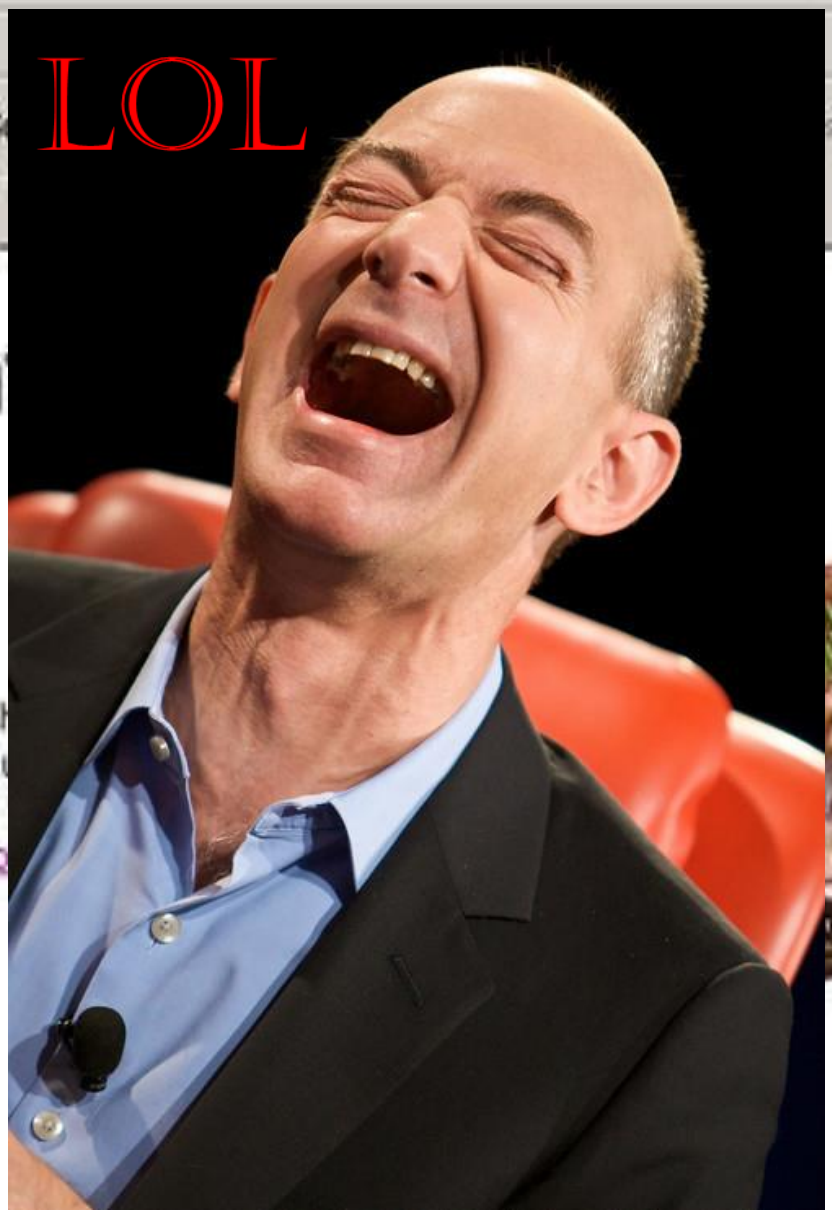
Favorites Editorial Links News Web Interface Log In

500 Service Unavailable Error

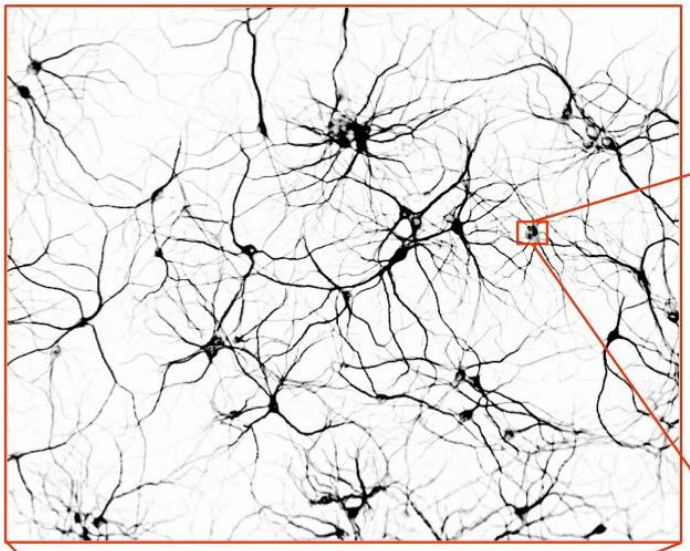
amazon.com

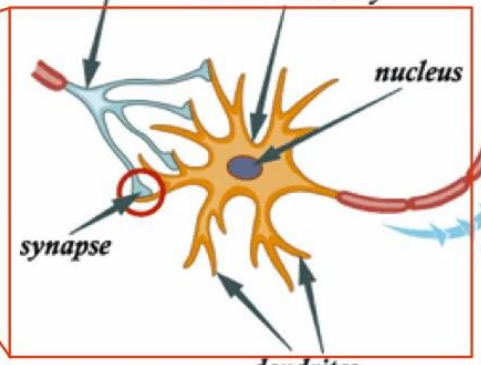
Oops!

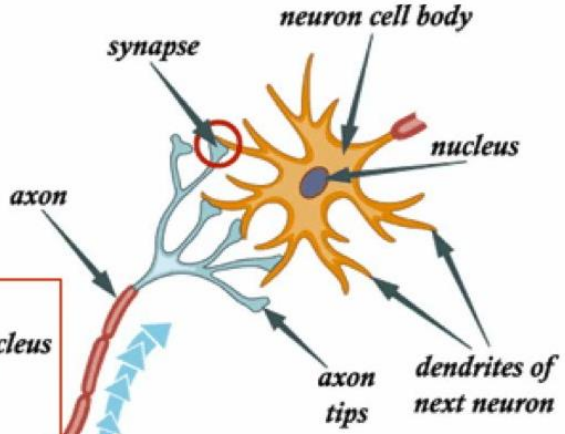
We're very sorry, but we're h
asked us to do. Please give u
button on your browser and
from the beginning on our ho



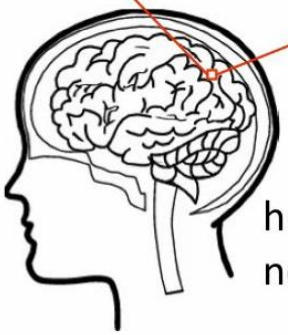
Neurons & the Brain



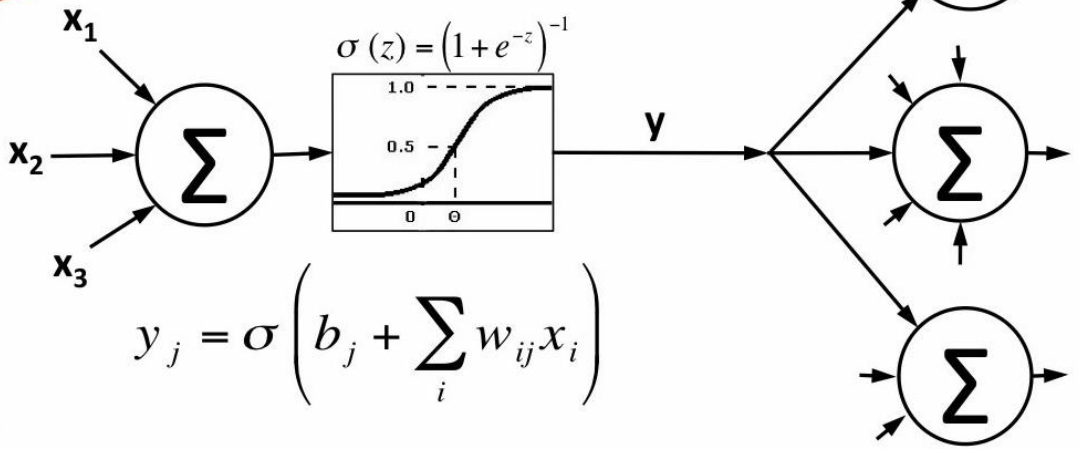




electrical signal

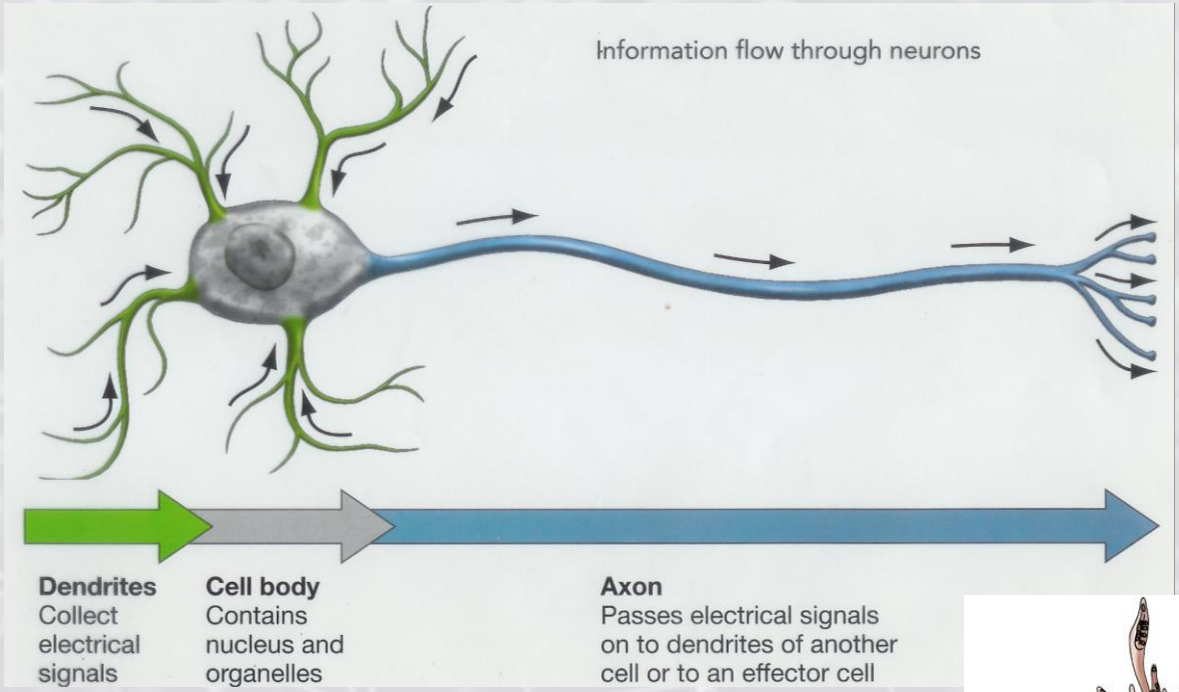


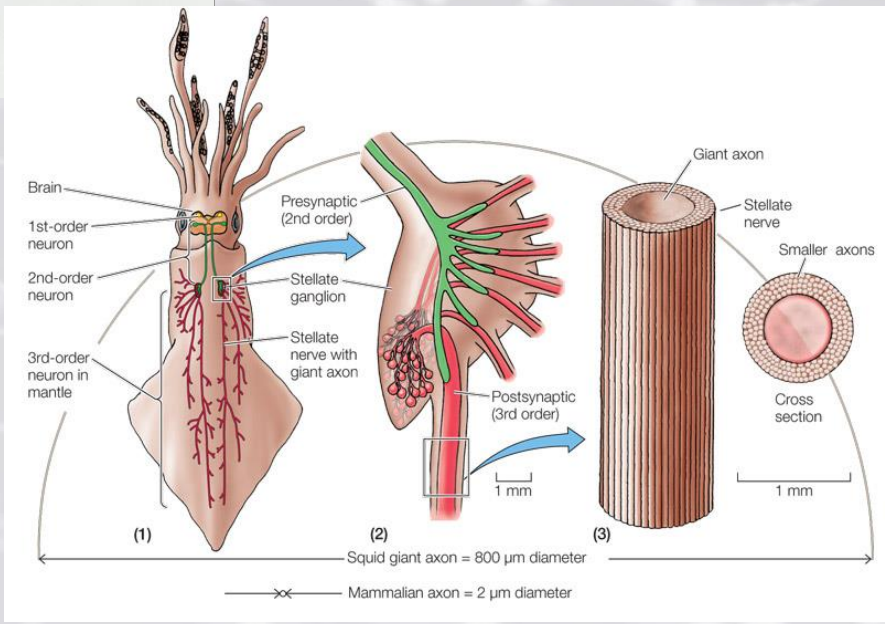
humans don't need features



$$y_j = \sigma\left(b_j + \sum_i w_{ij} x_i\right)$$

Neurons & the Brain



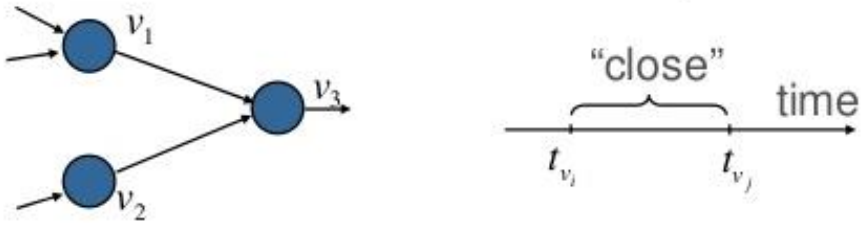


Hebb's Postulate

"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."

- In other words: if two neurons fire "close in time" then strength of synaptic connection between them increases.

$$\Delta w_{ij}(t) = \eta v_i v_j g(t_{v_i}, t_{v_j})$$



- Weights reflect correlation between firing events.

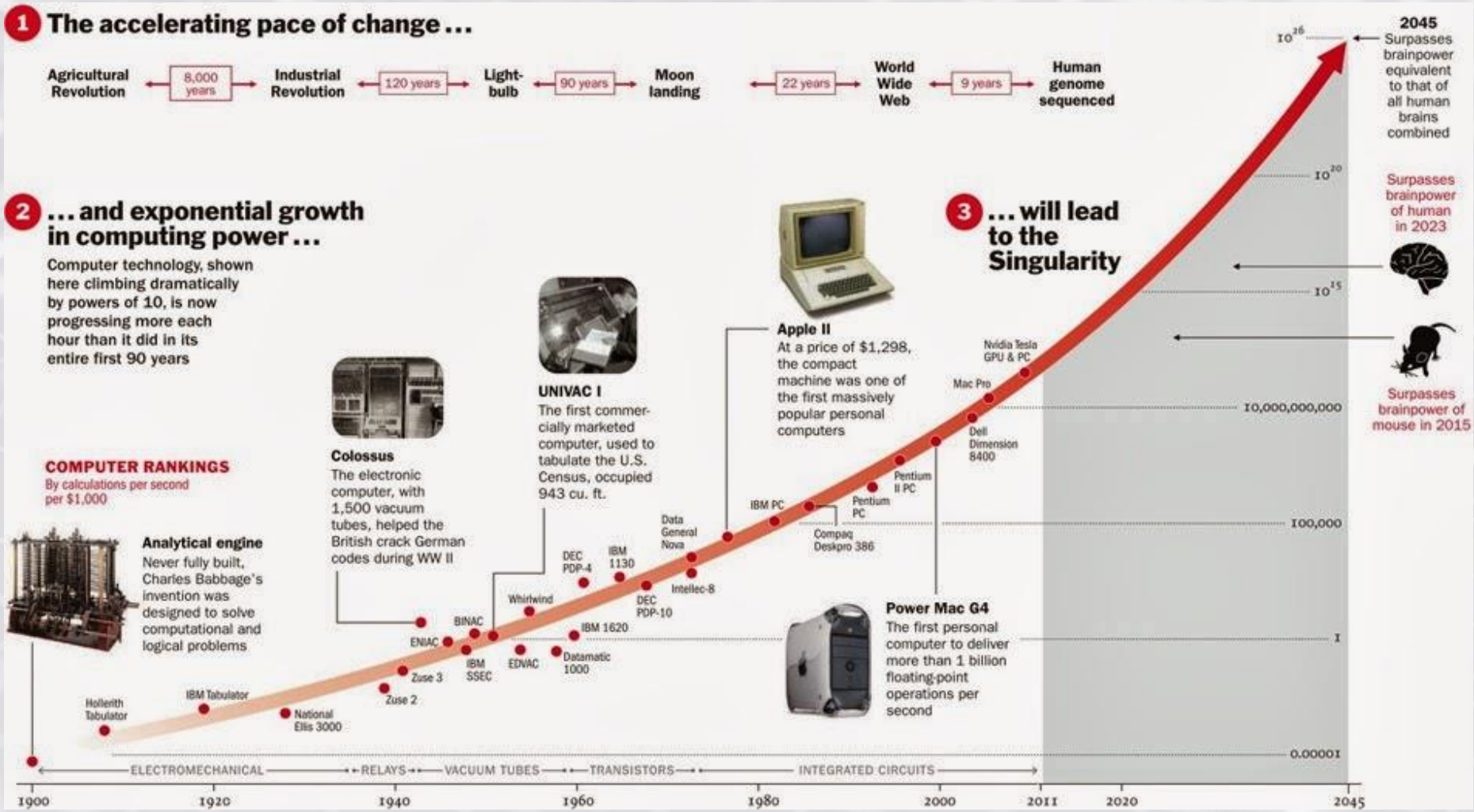
Neurons & the Brain

- Human brain contains ~10^{11} neurons
- Each individiaul neuron connects to ~10^{4} neuron
- ~10^{14} total synapses!

	Brain	Computer
Number of Processing Units	$\approx 10^{11}$	$\approx 10^{9}$
Type of Processing Units	Neurons	Transistors
Form of Calculation	Massively Parallel	Generally Serial
Data Storage	Associative	Address-based
Response Time	$\approx 10^{-3}$s	$\approx 10^{-9}$s
Processing Speed	Very Variable	Fixed
Potential Processing Speed	$\approx 10^{13}$ FLOPS [14]	$\approx 10^{18}$ FLOPS
Real Processing Speed	$\approx 10^{12}$ FLOPS	$\approx 10^{10}$ FLOPS
Resilience	Very High	Almost None
Power Consumption per Day	20W	300W [15]

Neurons & the Brain

– Is the singularity near?



Neurons & the Brain

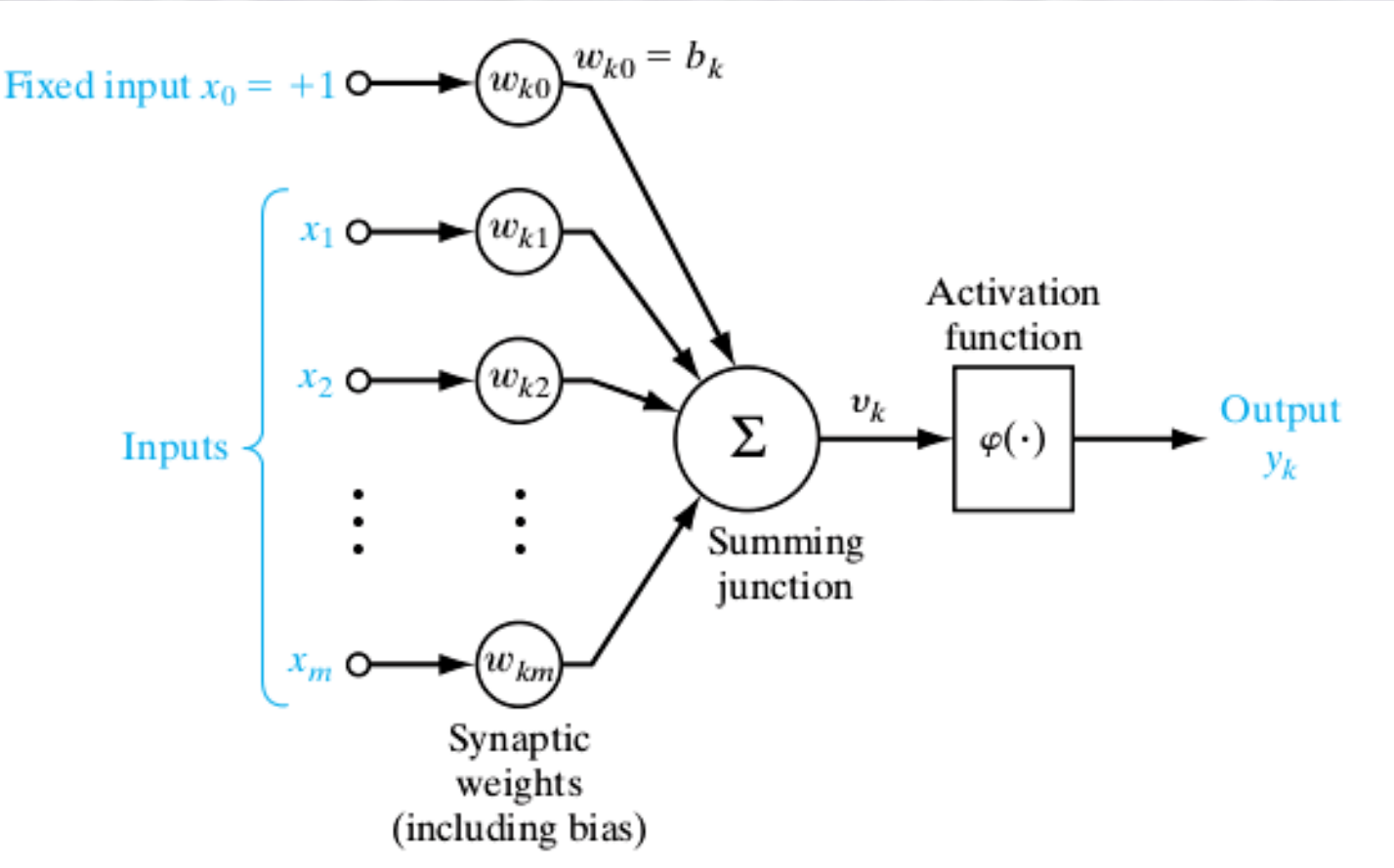
TYPE I CIVILIZATION harnesses all the resources of a planet. Carl Sagan estimated that Earth rates about 0.7 on the scale.

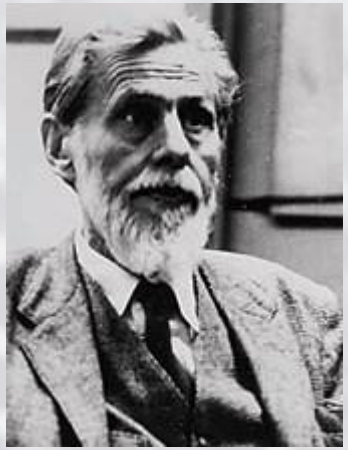
TYPE II CIVILIZATION harnesses all the radiation of a star. Humans might reach Type II in a few thousand years.

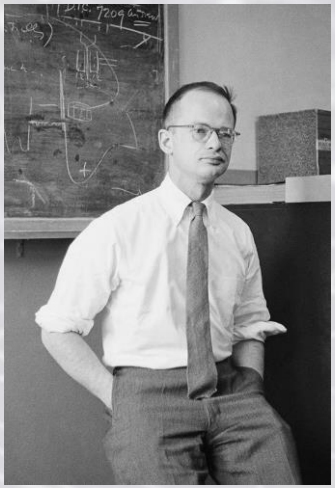
TYPE III CIVILIZATION harnesses all the resources of a galaxy. Humans might reach Type III in a few hundred thousand to a million years.

The Kardashev Scale

McCulloch & Pitts Neuron Model (1943)







(3) Components:

(1) Set of weighted inputs $\{w_i\}$ that correspond to synapses

(2) An "adder" that sums the input signals (equivalent to membrane of the cell that collects the electrical charge)

(3) An activation function (initially a threshold function) that decides whether the neuron fires ("spikes") for the current inputs.

McCulloch & Pitts Neuron Model (1943)

– Limitations & Deviations of the M-P Neuron Model:

(*) Summing is linear.

(*) No explicit model of "spike trains" (sequence of pulses that encodes information in biological neuron).

(*) Threshold value is usually fixed.

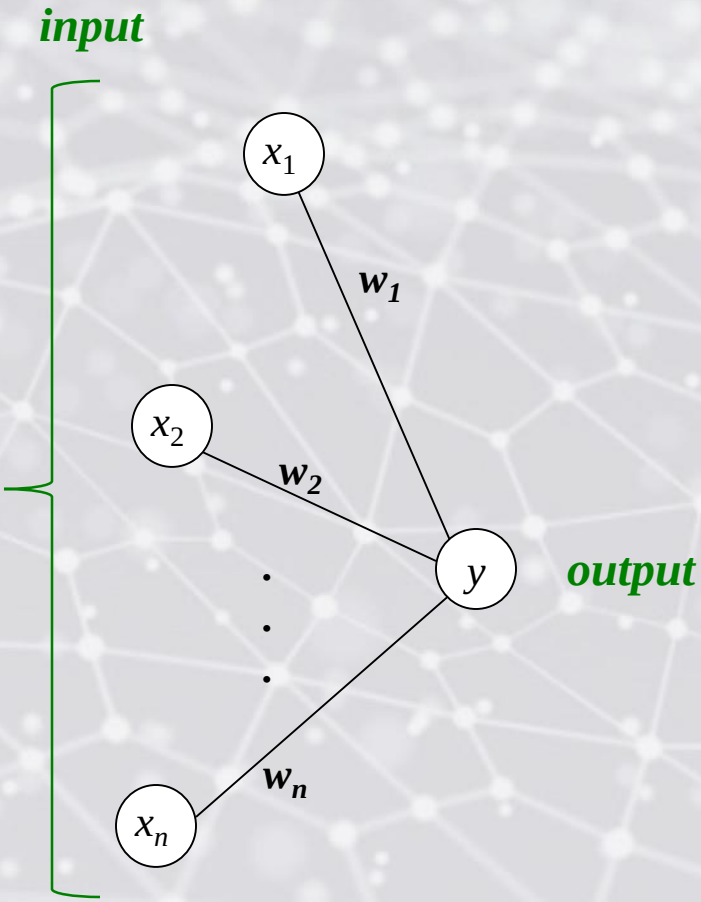
(*) Sequential updating implicit (biological neurons usually update themselves asynchronously)

(*) Weights can be positive (excitatory) or negative (inhibitory); biological neurons do not change in this way.

(*) Real neurons can have synapses that link back to themselves (e.g. feedback loop) – see RNNs (recurrent neural networks).

(*) Other biological aspects ignored: chemical concentrations, refractory periods, etc.

Perceptrons as simplified “neurons”

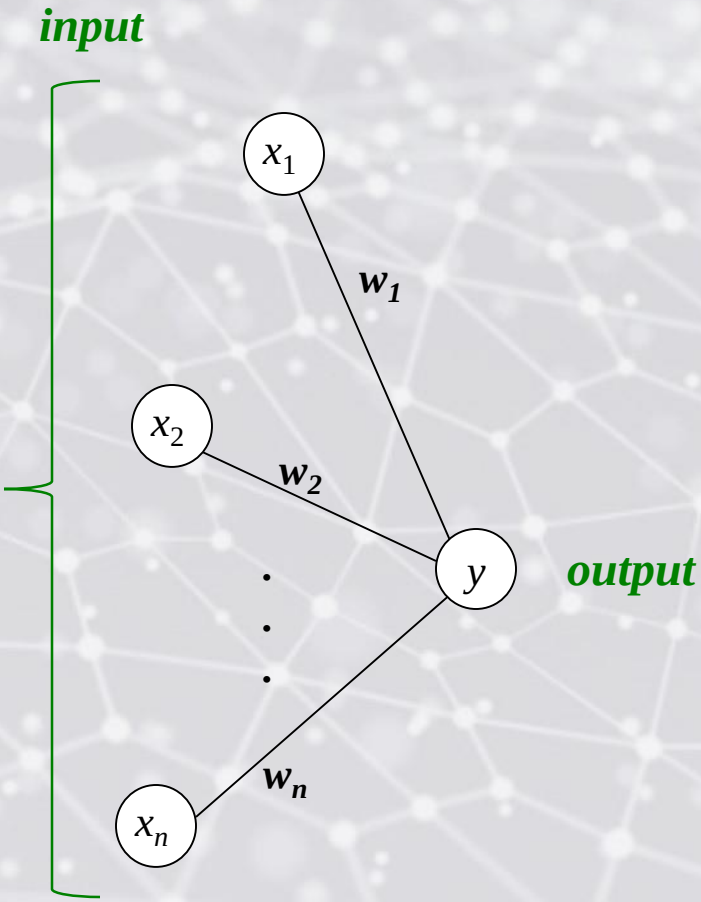


Input is $(x_1, x_2, \ldots x_n)$

Weights are $(w_1, w_2, \ldots w_n)$

Output y is 1 (“the neuron fires”) if the sum of the inputs times the weights is greater or equal to the threshold:

Perceptrons as simplified "neurons"



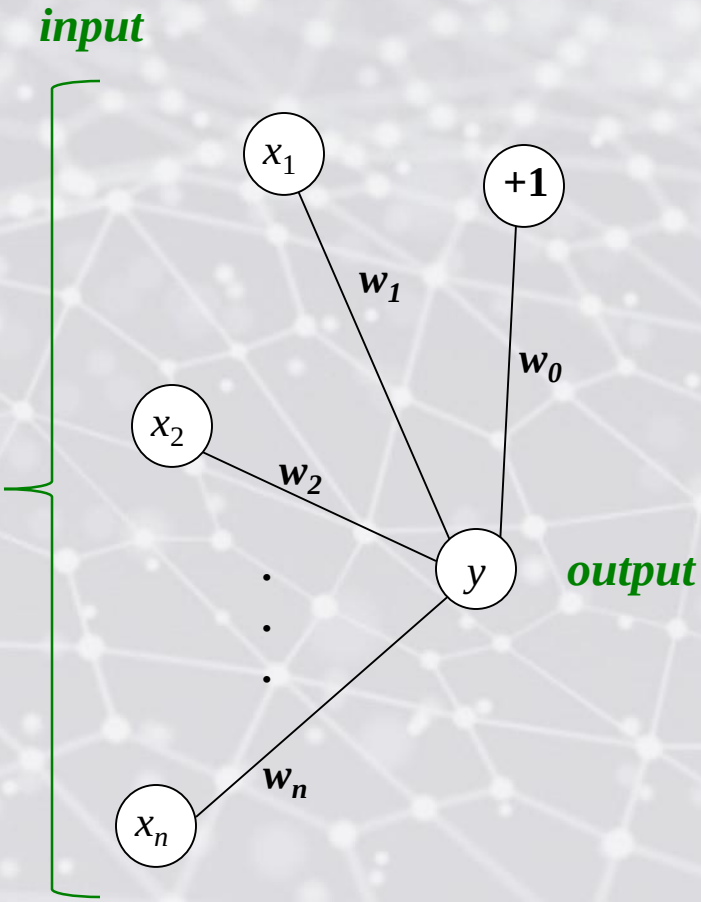
Input is $(x_1, x_2, \ldots x_n)$

Weights are $(w_1, w_2, \ldots w_n)$

Output y is 1 ("the neuron fires") if the sum of the inputs times the weights is greater or equal to the threshold:

$$If\ w_1x_1 + w_2x_2 + \ldots + w_nx_n > threshold$$
$$then\ y = 1,\ else\ y = 0$$

Perceptrons as simplified "neurons"



Input is $(x_1, x_2, \ldots x_n)$

Weights are $(w_1, w_2, \ldots w_n)$

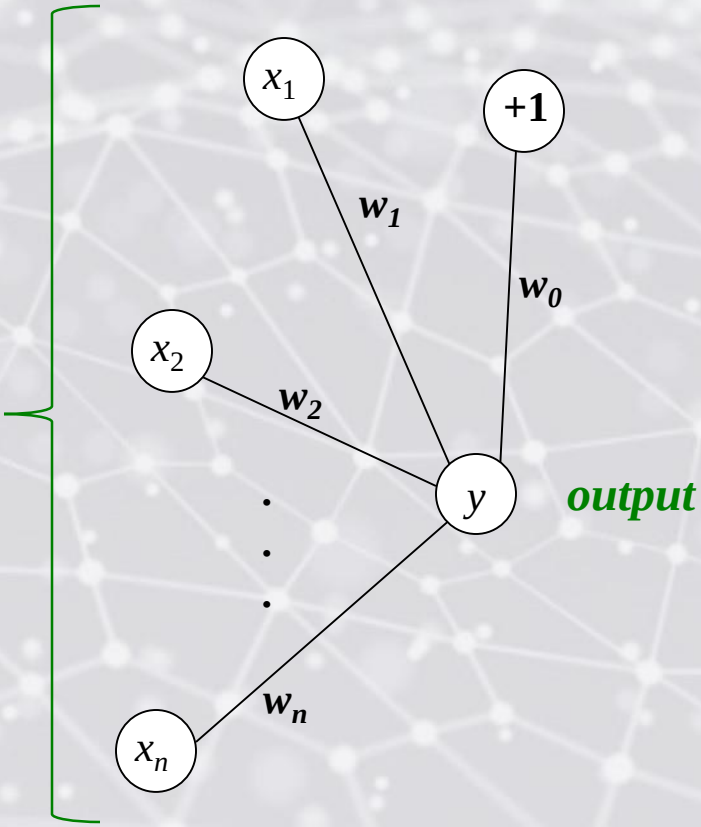
Output y is 1 ("the neuron fires") if the sum of the inputs times the weights is greater or equal to the threshold:

$$If\ w_1x_1 + w_2x_2 + \ldots + w_nx_n > threshold$$
$$then\ y = 1,\ else\ y = 0$$

Perceptrons as simplified "neurons"

w_0 is called the "**bias**"

$-w_0$ is called the "**threshold**"



Input is $(x_1, x_2, \ldots x_n)$

Weights are $(w_1, w_2, \ldots w_n)$

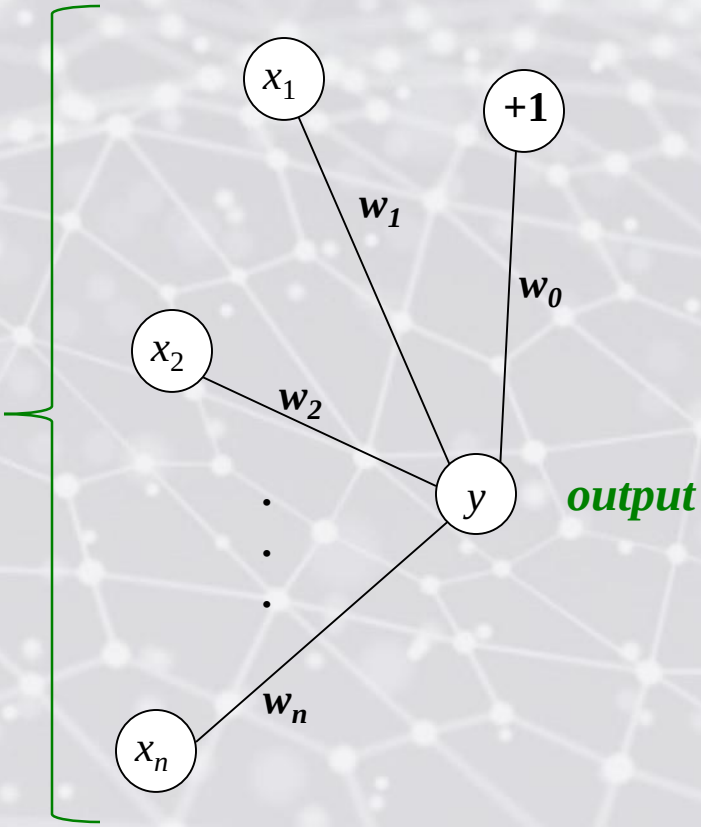
Output y is 1 ("the neuron fires") if the sum of the inputs times the weights is greater or equal to the threshold:

$$If\ w_1x_1 + w_2x_2 + \ldots + w_nx_n > threshold$$
$$then\ y = 1,\ else\ y = 0$$

Perceptrons as simplified "neurons"

w_0 is called the "**bias**"

$-w_0$ is called the "**threshold**"



Input is $(x_1, x_2, \ldots x_n)$

Weights are $(w_1, w_2, \ldots w_n)$

Output y is 1 ("the neuron fires") if the sum of the inputs times the weights is greater or equal to the threshold:

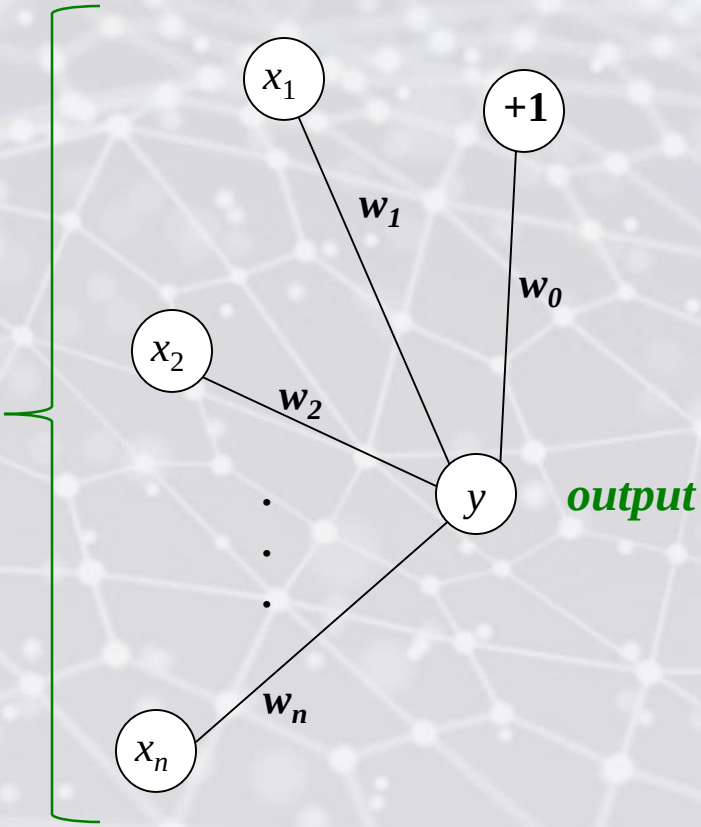
$$If\ w_1x_1 + w_2x_2 + \ldots + w_nx_n > threshold$$
$$then\ y = 1,\ else\ y = 0$$

$$If\ w_1x_1 + w_2x_2 + \ldots + w_nx_n > -\ w_0$$
$$then\ y = 1,\ else\ y = 0$$

Perceptrons as simplified "neurons"

w_0 is called the "**bias**"

$-w_0$ is called the "**threshold**"



Input is $(x_1, x_2, \ldots x_n)$

Weights are $(w_1, w_2, \ldots w_n)$

Output y is 1 ("the neuron fires") if the sum of the inputs times the weights is greater or equal to the threshold:

$$If\ w_1x_1 + w_2x_2 + \ldots + w_nx_n > threshold$$
$$then\ y = 1,\ else\ y = 0$$

$$If\ w_1x_1 + w_2x_2 + \ldots + w_nx_n > -\ w_0$$
$$then\ y = 1,\ else\ y = 0$$

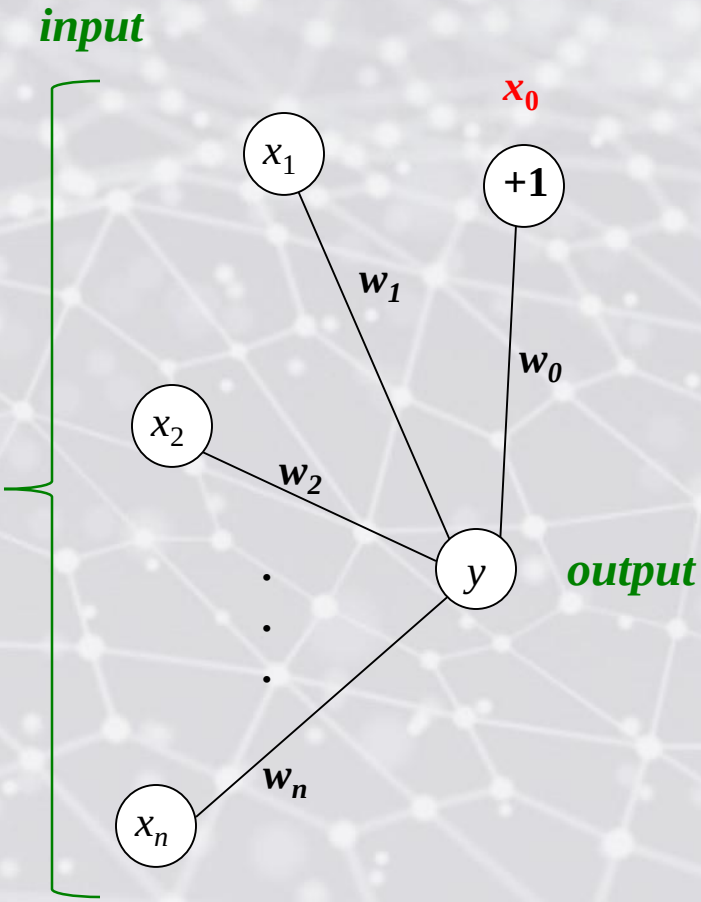
$$If\ w_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n > 0$$
$$then\ y = 1,\ else\ y = 0$$

Q: Why do we introduce a bias term?

Perceptrons as simplified "neurons"

w_0 is called the "**bias**"

$-w_0$ is called the "**threshold**"



Input is $(x_1, x_2, \ldots x_n)$

Weights are $(w_1, w_2, \ldots w_n)$

Output y is 1 ("the neuron fires") if the sum of the inputs times the weights is greater or equal to the threshold:

$$If\ w_1x_1 + w_2x_2 + \ldots + w_nx_n > threshold$$
$$then\ y = 1,\ else\ y = 0$$

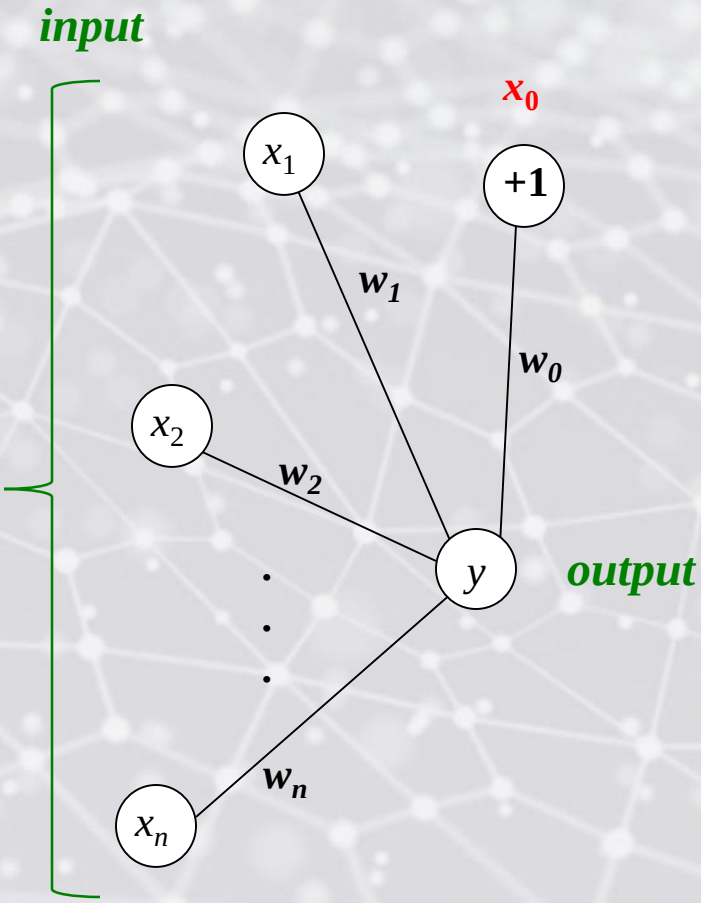
$$If\ w_1x_1 + w_2x_2 + \ldots + w_nx_n > -\ w_0$$
$$then\ y = 1,\ else\ y = 0$$

$$If\ w_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n > 0$$
$$then\ y = 1,\ else\ y = 0$$

Perceptrons as simplified "neurons"

w_0 is called the "**bias**"

$-w_0$ is called the "**threshold**"



Input is $(x_1, x_2, \ldots x_n)$

Weights are $(w_1, w_2, \ldots w_n)$

Output y is 1 ("the neuron fires") if the sum of the inputs times the weights is greater or equal to the threshold:

$$If\ w_1x_1 + w_2x_2 + \ldots + w_nx_n > threshold$$
$$then\ y = 1,\ else\ y = 0$$

$$If\ w_1x_1 + w_2x_2 + \ldots + w_nx_n > -\ w_0$$
$$then\ y = 1,\ else\ y = 0$$

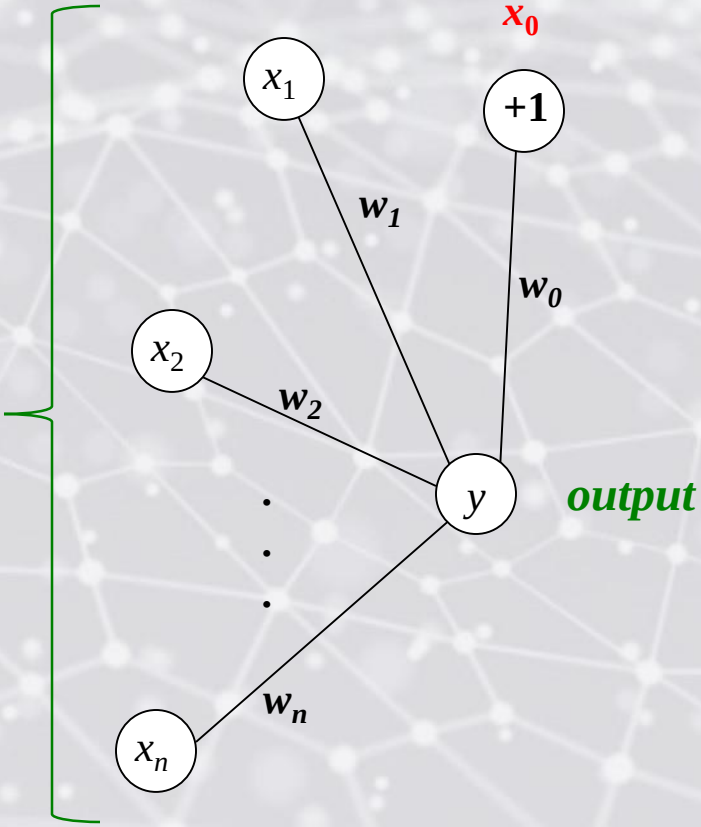
$$If\ w_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n > 0$$
$$then\ y = 1,\ else\ y = 0$$

$$If\ w_0x_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n > 0$$
$$then\ y = 1,\ else\ y = 0$$

Perceptrons as simplified "neurons"

w_0 is called the "**bias**"

$-w_0$ is called the "**threshold**"



Input is $(x_1, x_2, \ldots x_n)$

Weights are $(w_1, w_2, \ldots w_n)$

Output y is 1 ("the neuron fires") if the sum of the inputs times the weights is greater or equal to the threshold:

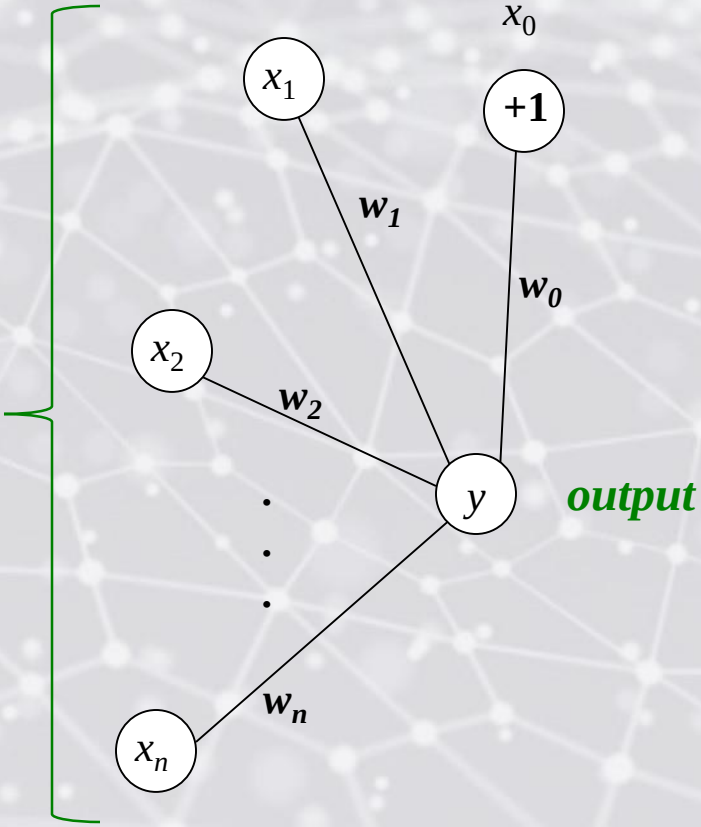
$$output = y(\mathbf{x}) = a(w_0x_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n)$$

where $a(z) = \begin{cases} 0 & if\ z \le 0 \\ 1 & if\ z > 0 \end{cases}$

Perceptrons as simplified "neurons"

w_0 is called the "**bias**"

$-w_0$ is called the "**threshold**"



Input is $(x_1, x_2, \ldots x_n)$

Weights are $(w_1, w_2, \ldots w_n)$

Output y is 1 ("the neuron fires") if the sum of the inputs times the weights is greater or equal to the threshold:

$$output = y(\mathbf{x}) = a(w_0x_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n)$$

where $a(z) = \begin{cases} 0 & if\ z \le 0 \\ 1 & if\ z > 0 \end{cases}$

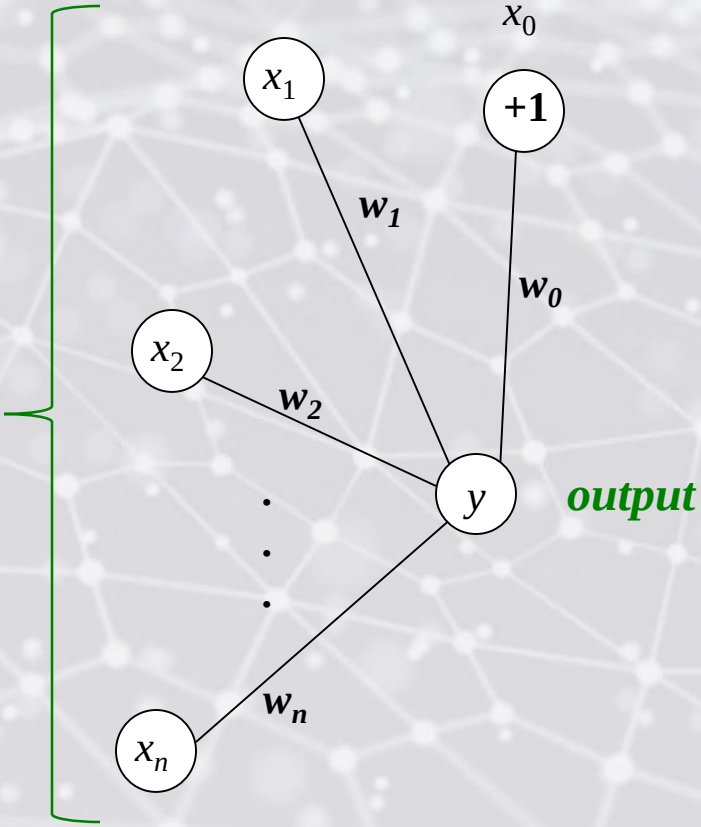
Let $\mathbf{x} = (x_0, x_1, x_2, \ldots x_n)$

$\mathbf{w} = (w_0, w_1, w_2, \ldots w_n)$

Perceptrons as simplified "neurons"

w_0 is called the "**bias**"

$-w_0$ is called the "**threshold**"



Input is $(x_1, x_2, \ldots x_n)$

Weights are $(w_1, w_2, \ldots w_n)$

Output y is +1 ("the neuron fires") if the sum of the inputs times the weights is greater or equal to the threshold:

$$output = y(\mathbf{x}) = a(w_0 x_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n)$$

where $a(z) = \begin{cases} 0 & if\ z \le 0 \\ 1 & if\ z > 0 \end{cases}$

Let $\mathbf{x} = (x_0, x_1, x_2, \ldots x_n)$

$\mathbf{w} = (w_0, w_1, w_2, \ldots w_n)$

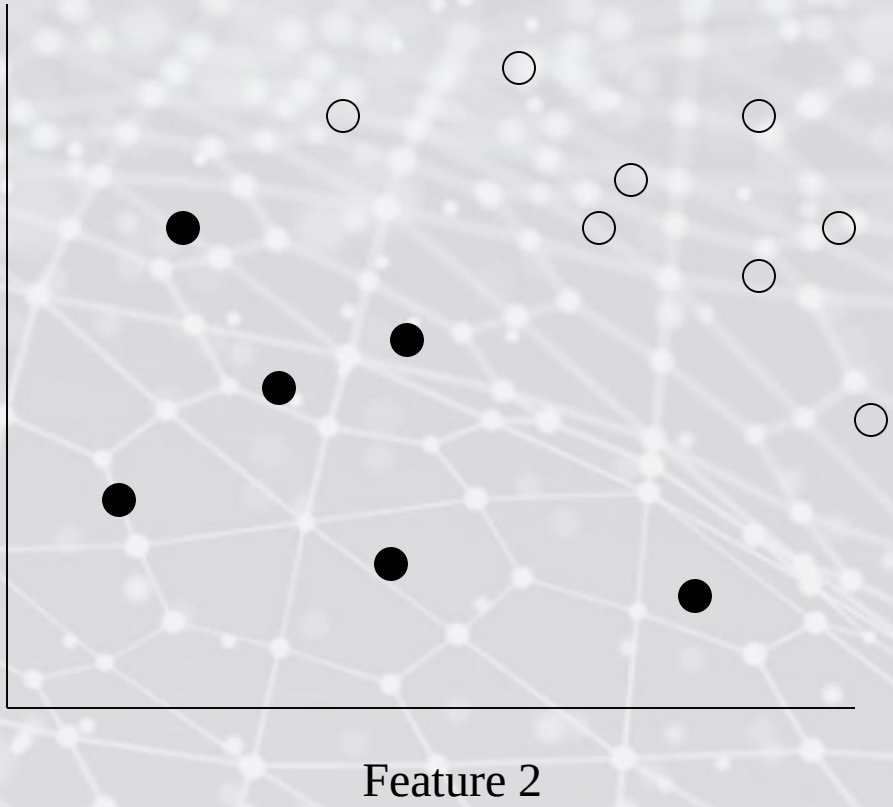
Then:

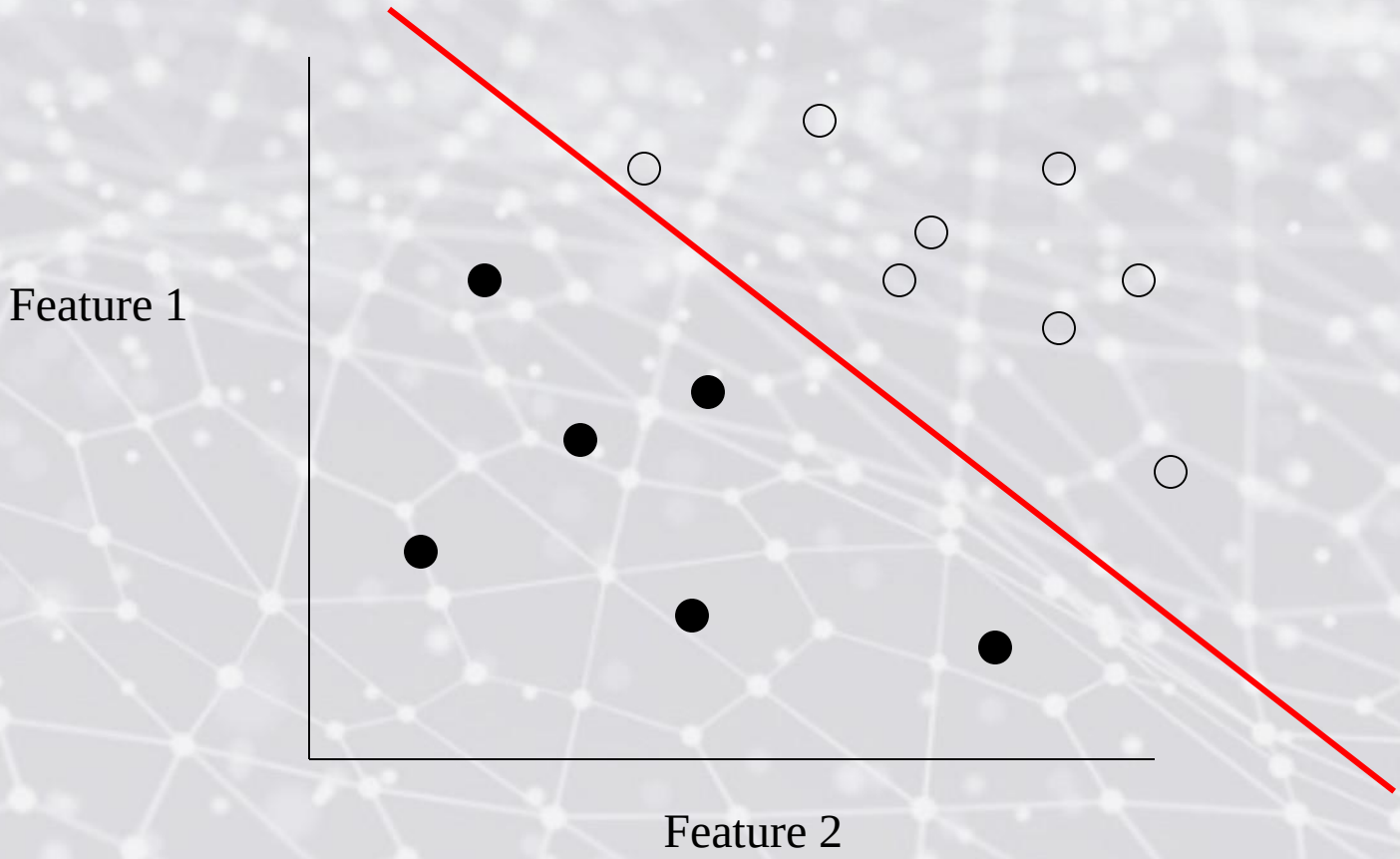
$$y = a(\mathbf{w} \times \mathbf{x})$$

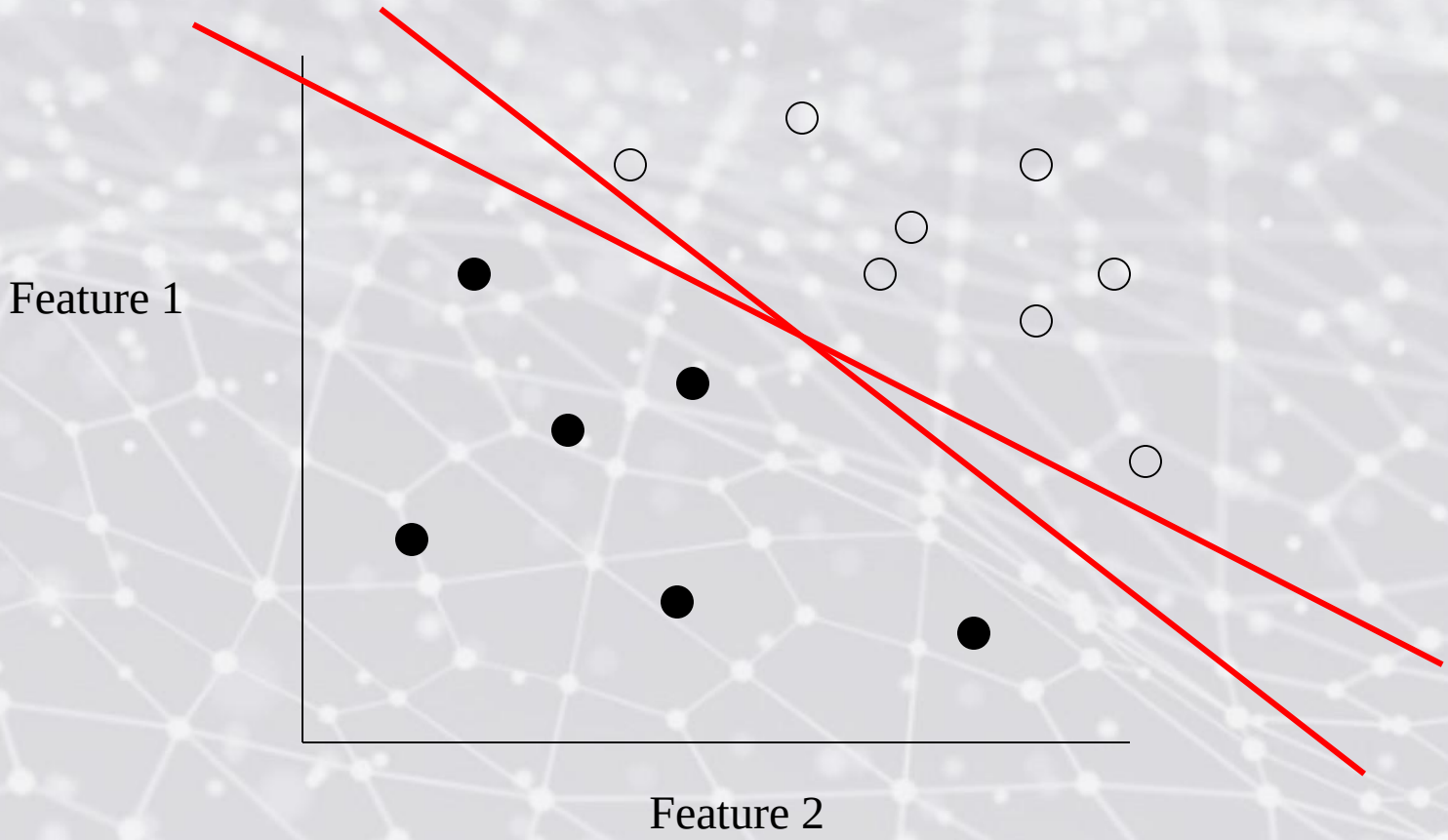
Decision Surfaces

- Assume data can be separated into two classes, positive and negative, by a linear *decision surface*.

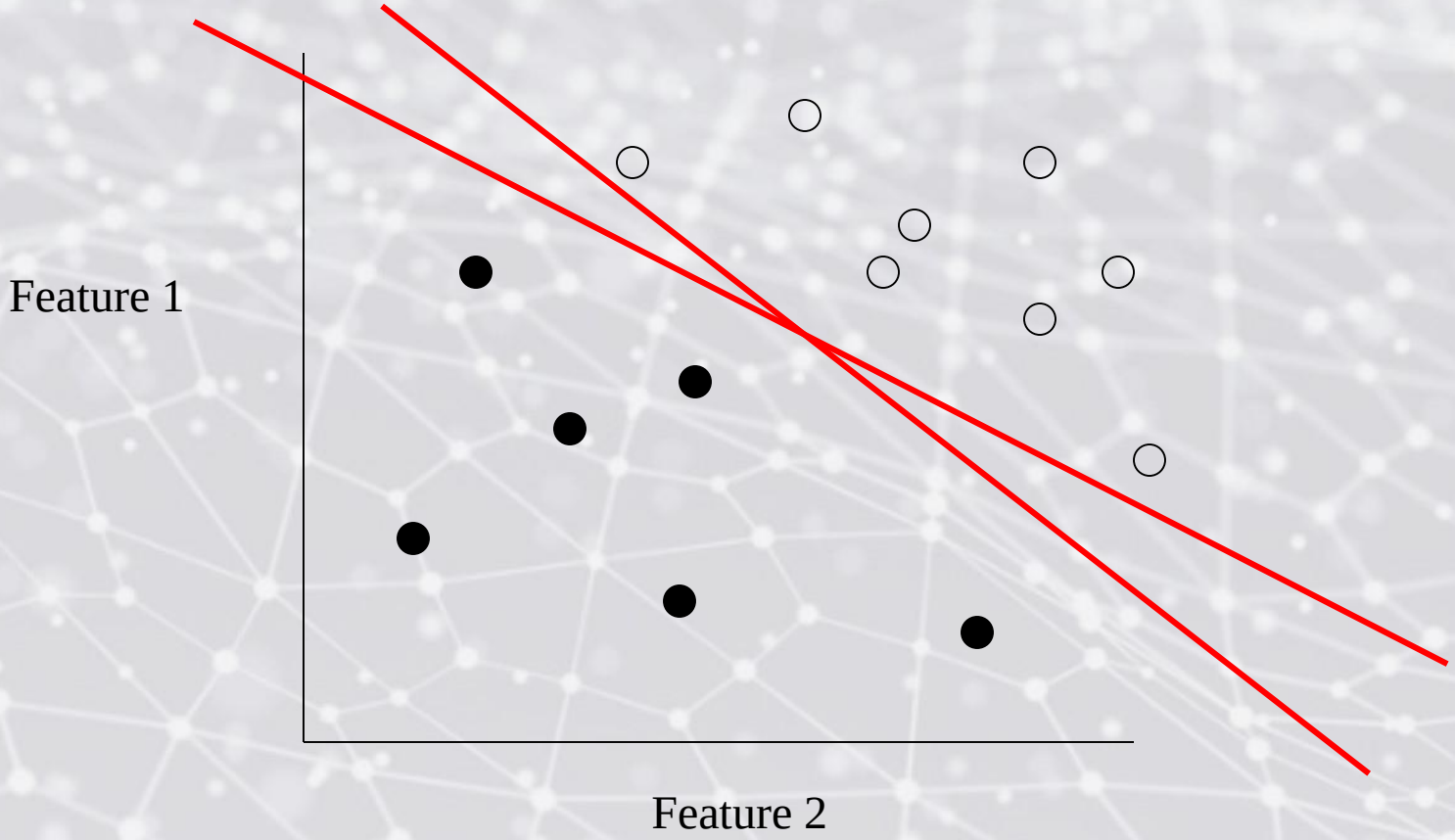
- Assuming data is *n*-dimensional, a perception represents a ($n-1$)-dimensional hyperplane that separates the data into two classes, *positive* (1) and *negative* (*0*).


Feature 1
Feature 2


Feature 1
Feature 2

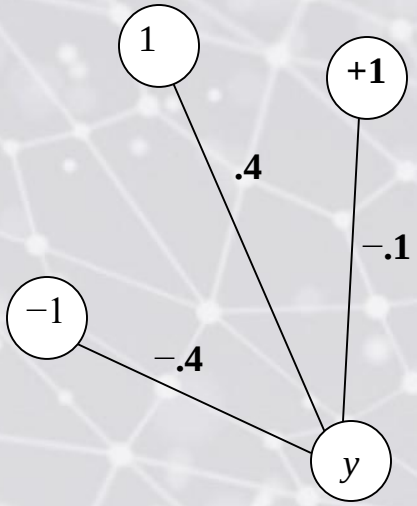

Feature 1
Feature 2

Example where line won’t work?



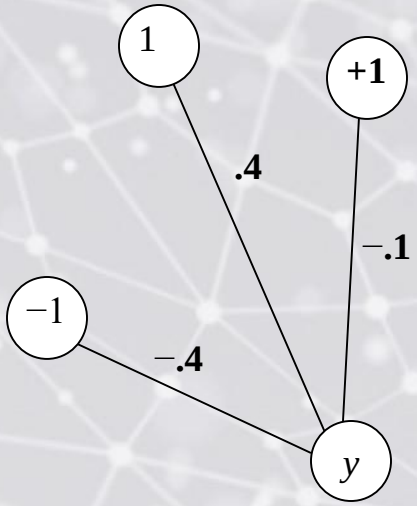
Example

- What is the predicted class *y*?



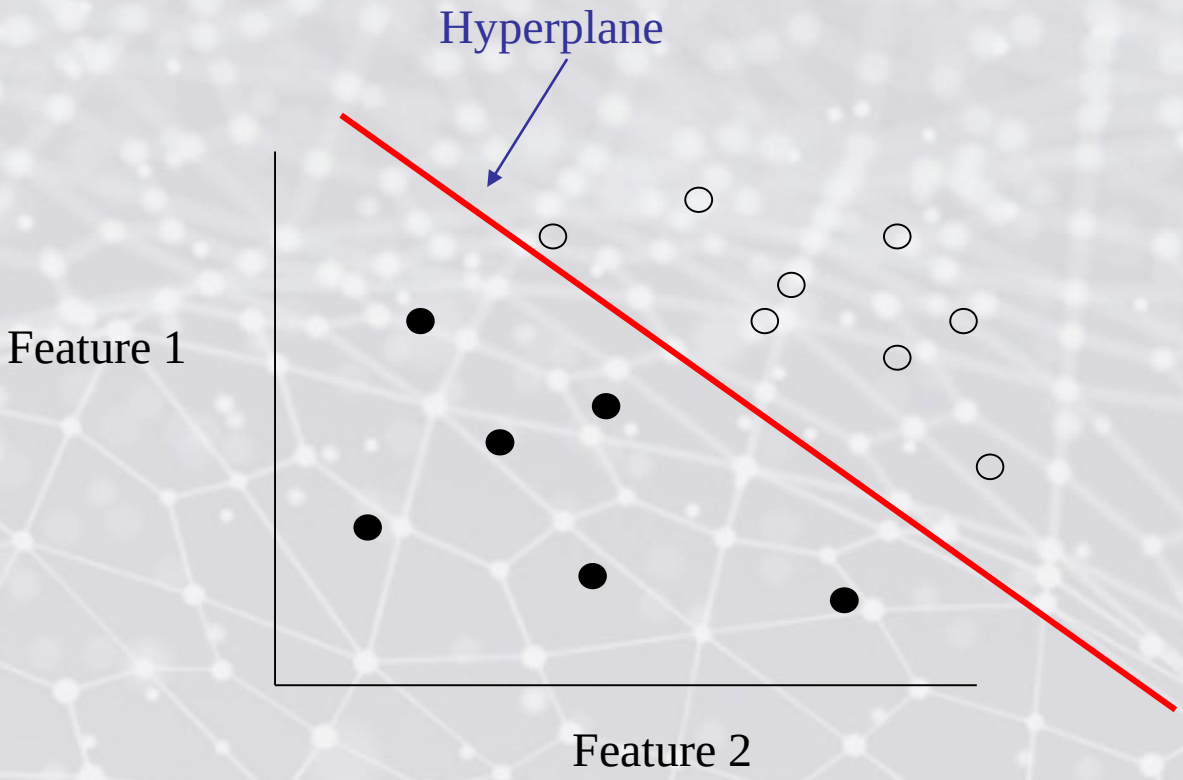
Example

- What is the predicted class *y*?



$$y = a\left(\vec{w}\cdot\vec{x}\right) = a((-.1)(1)+.4(1)+(-.4)(-1)) = a(0.7) = 1$$

Geometry of the perceptron

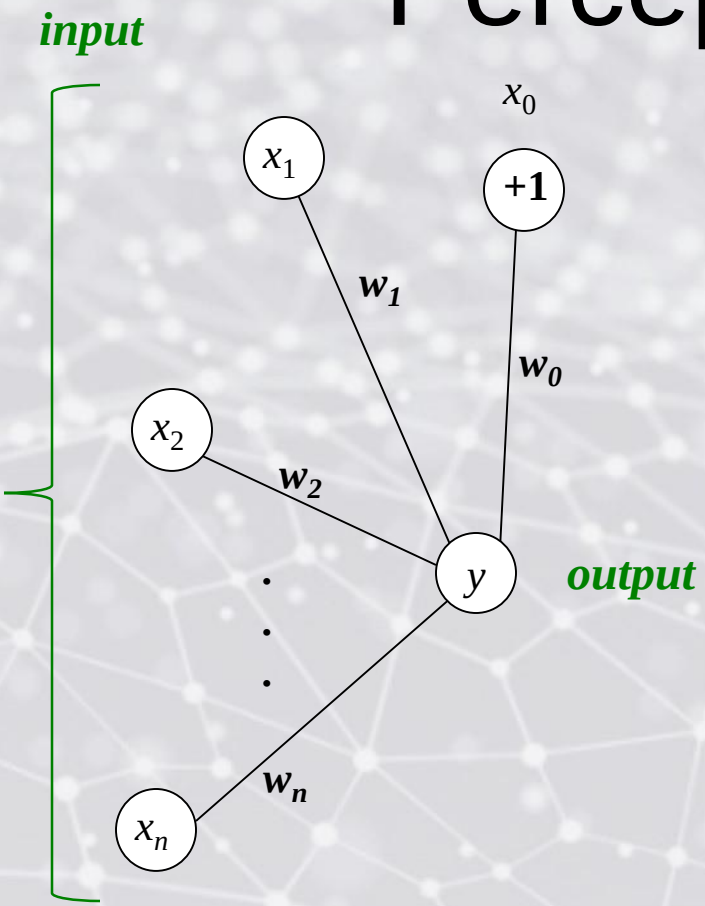


In 2D:

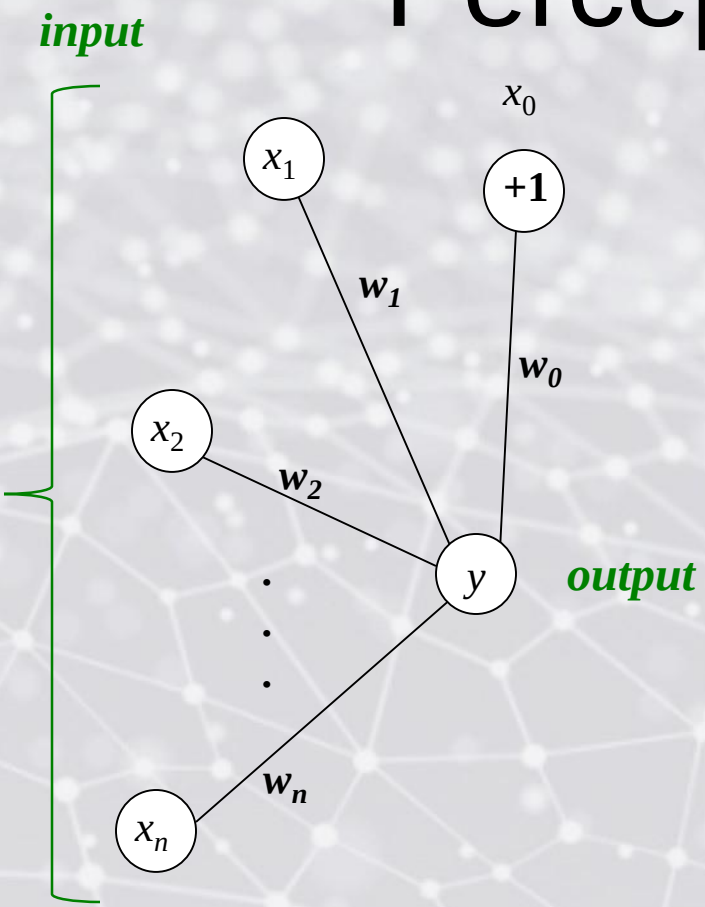
$$w_1 x_1 + w_2 x_2 + w_0 = 0$$

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{w_0}{w_2}$$

Perceptron Learning

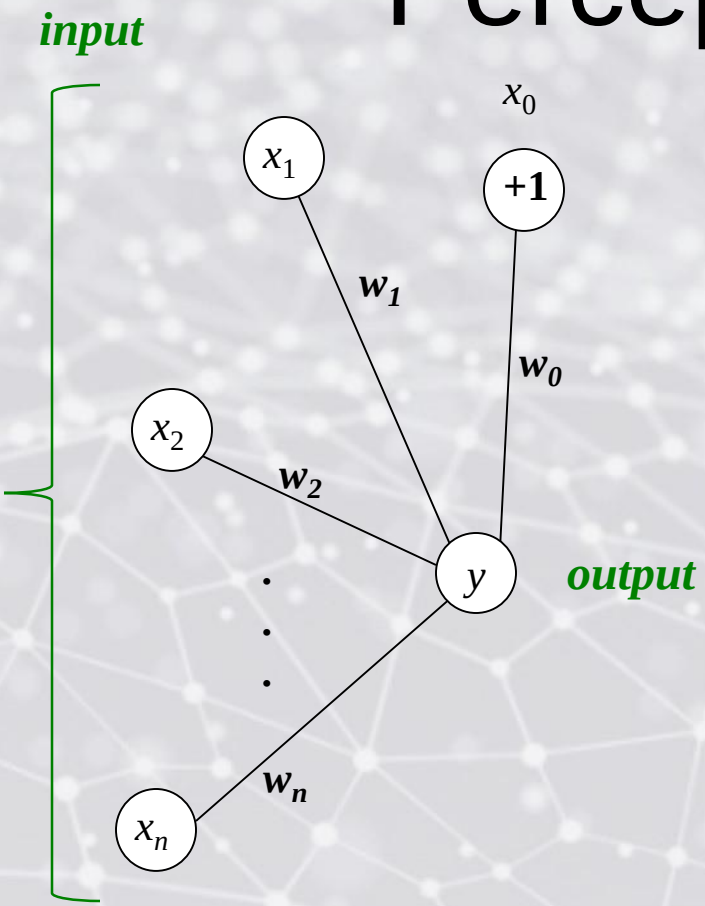


Perceptron Learning



Input instance: $\mathbf{x}^k = (x_1, x_2, \ldots x_n)$,
with target class $t^k \in \{0,1\}$

Perceptron Learning



Goal is to use the training data to learn a set of weights that will:

(1) correctly classify the training data

(2) generalize to unseen data

Input instance: $\mathbf{x}^k = (x_1, x_2, \ldots x_n)$,
with target class $t^k \hat{I} \{0,1\}$

Perceptron Learning

Learning is often framed as an optimization problem:

- Find **w** that minimizes average "loss":

$$J(\mathbf{w}) = \frac{1}{M}\sum_{k=1}^{M} L(\mathbf{w}, \mathbf{x}^k, t^k)$$

where M is number of training examples and L is a "loss" function.

One part of the "art" of ML is to define a good loss function.

- Here, define the loss function as follows:

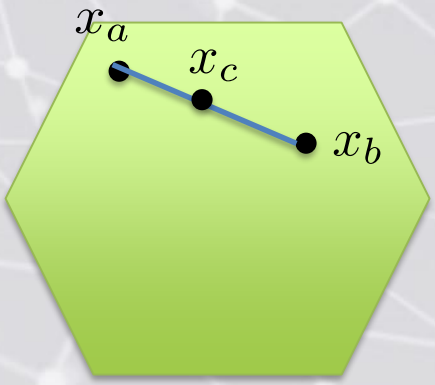
Let $y = a(\mathbf{w} \times \mathbf{x})$

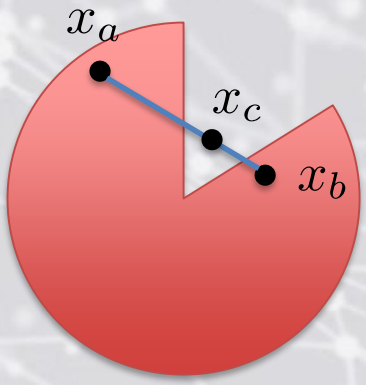
$$L(\mathbf{w}, \mathbf{x}^k, t^k) = \frac{1}{2}\left(t^k - y\right)^2 \quad \text{"squared loss"}$$

How to solve this minimization problem? **Gradient descent.**

Aside: Convex Regions

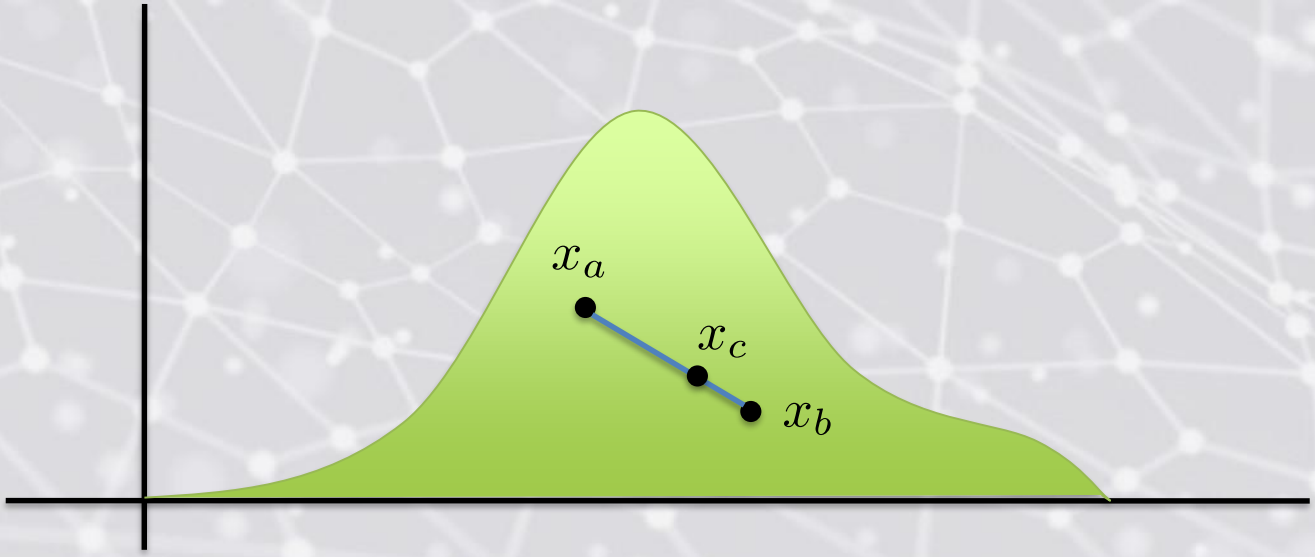
- Convex: for any pair of points x_a and x_b within a region, every point x_c on a line between x_a and x_b is in the region





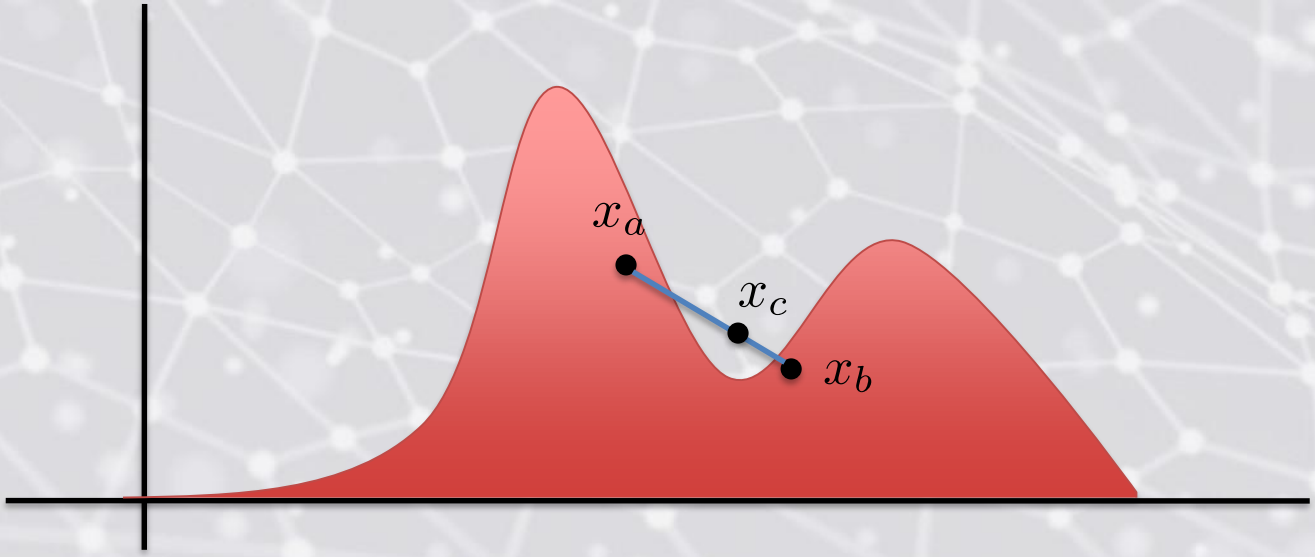
Aside: Convex Functions

- Convex: for any pair of points x_a and x_b within a region, every point x_c on a line between x_a and x_b is in the region



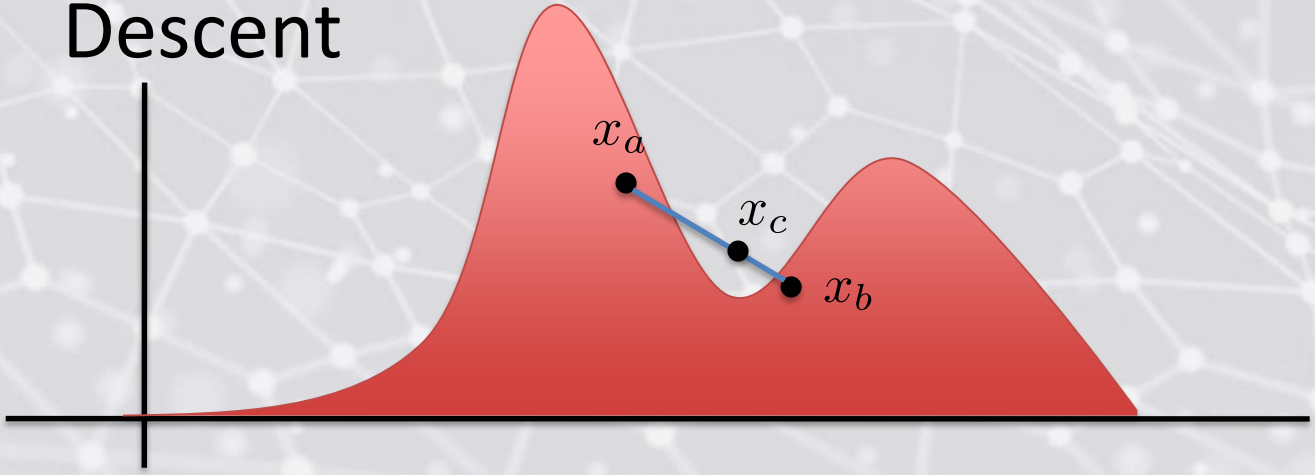
Aside: Convex Functions

- Convex: for any pair of points x_a and x_b within a region, every point x_c on a line between x_a and x_b is in the region



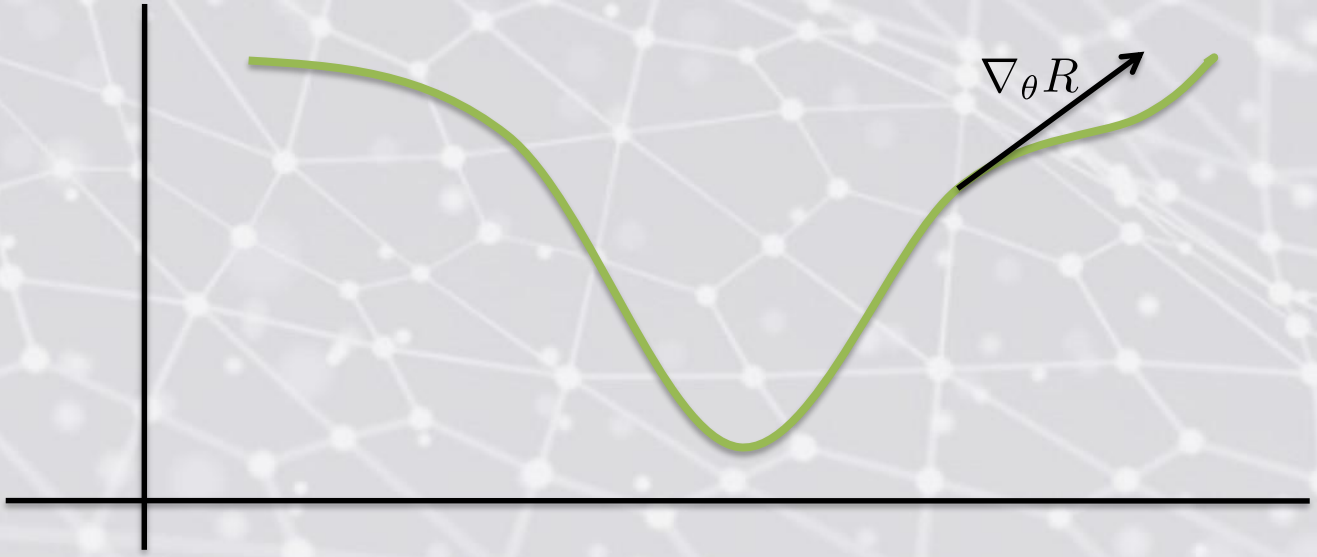
Aside: Convex Functions

- Convex functions have a single maximum and minimum!
- How does this help us?
- (nearly) Guaranteed optimality of Gradient Descent



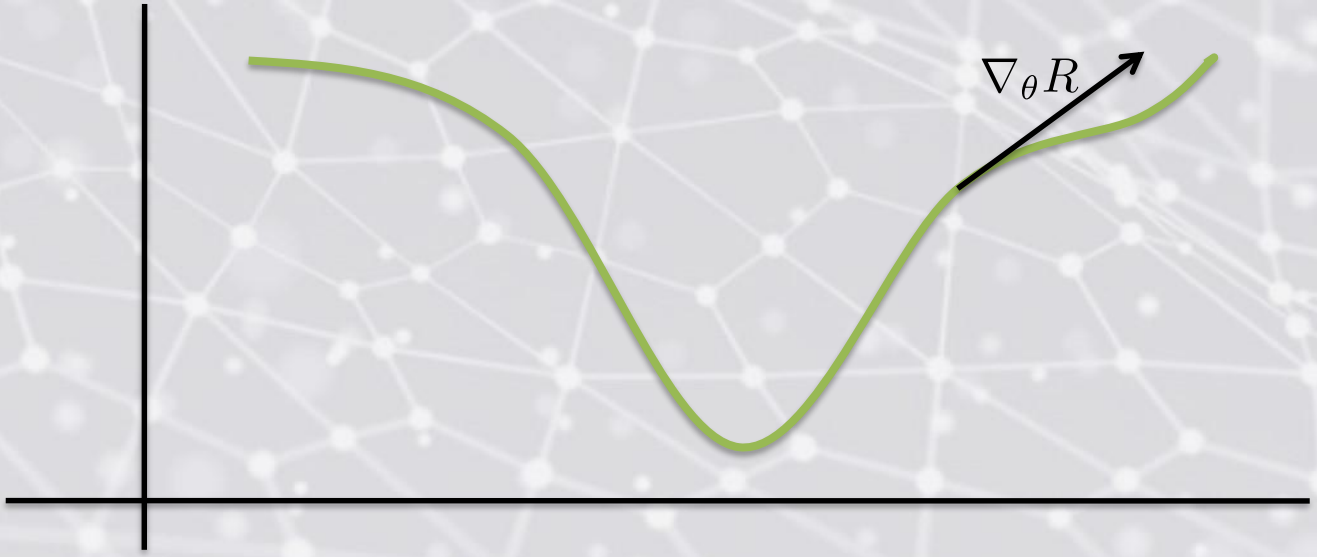
Gradient Descent

- The Gradient is defined (though we can't solve directly) $\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$
- Points in the direction of fastest increase



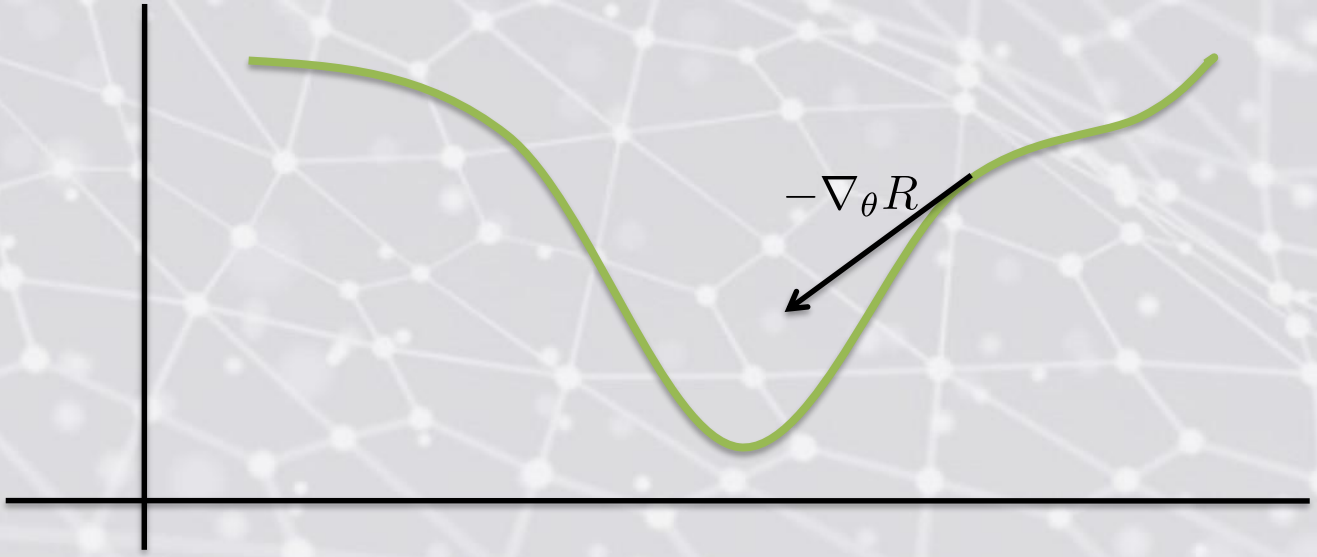
Gradient Descent

- Gradient points in the direction of fastest increase $\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$
- To minimize R, move in the opposite direction



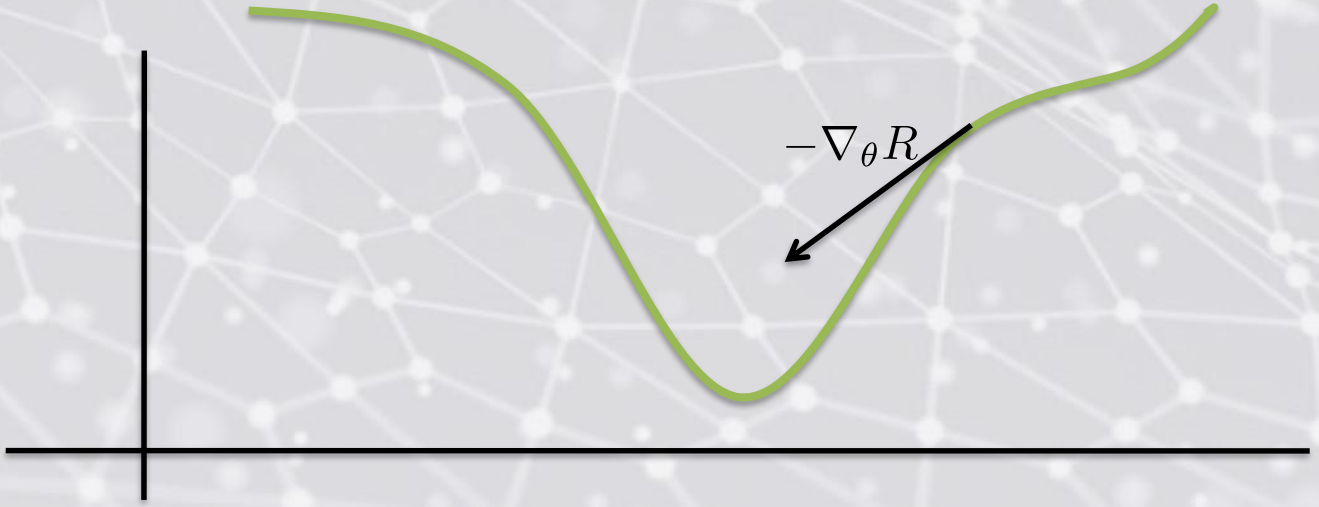
Gradient Descent

- Gradient points in the direction of fastest increase $$\nabla_\theta R = \frac{1}{2N}\sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i = 0$$
- To minimize R, move in the opposite direction



Gradient Descent

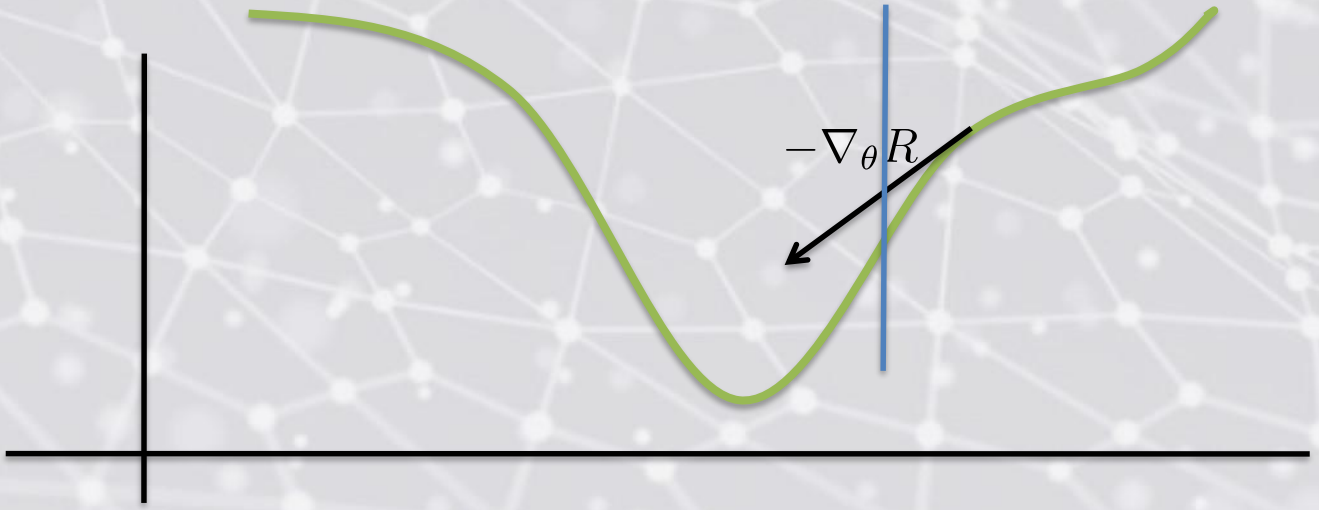
- Initialize Randomly $\theta_0 = random$
- Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- (nearly) guaranteed to converge to the minimum



$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$$

Gradient Descent

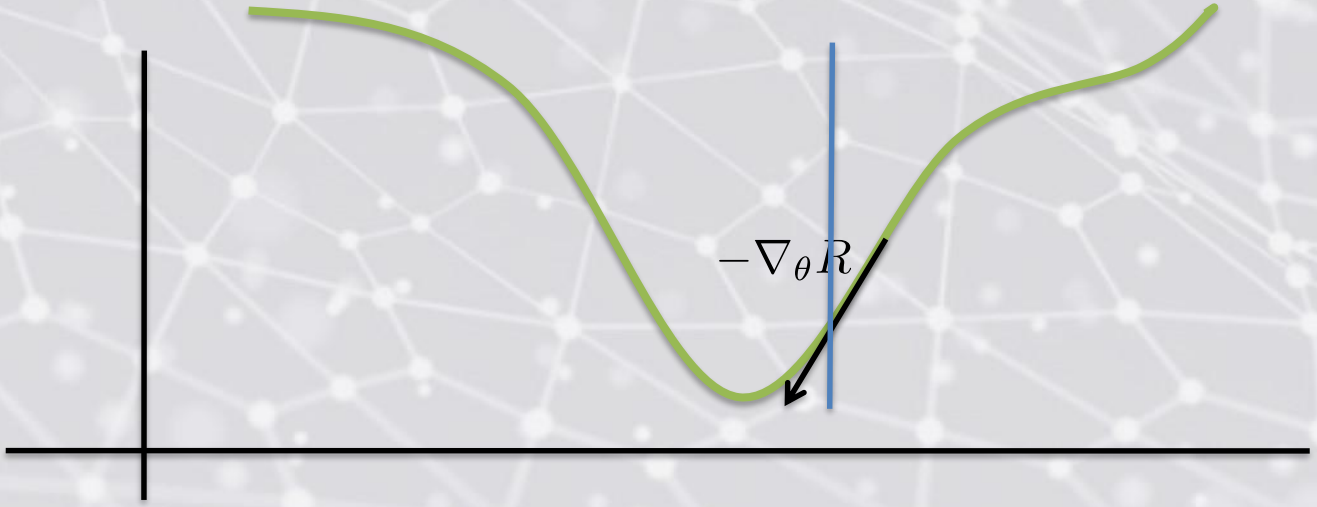
- Initialize Randomly $\theta_0 = random$
- Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- (nearly) guaranteed to converge to the minimum



$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$$

Gradient Descent

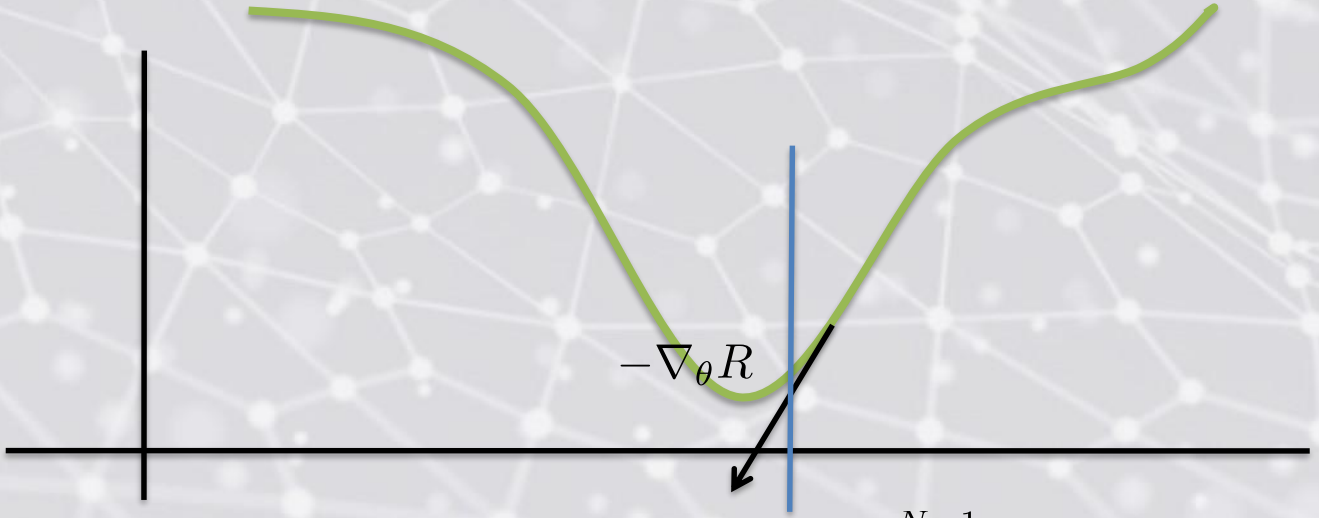
- Initialize Randomly $\theta_0 = random$
- Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- (nearly) guaranteed to converge to the minimum



$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$$

Gradient Descent

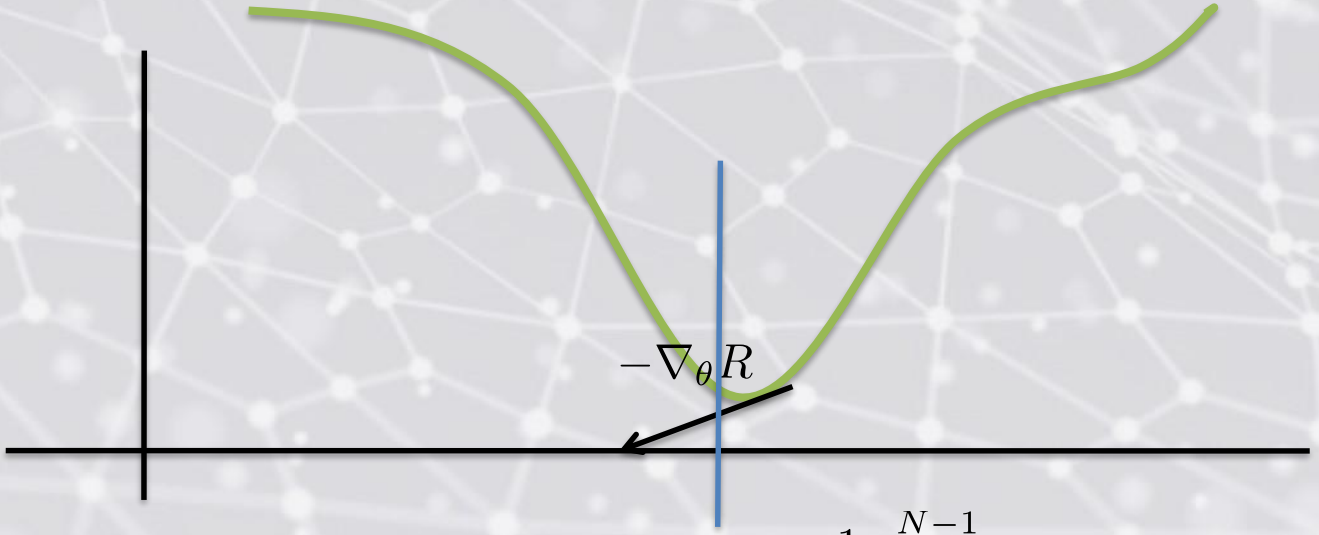
- Initialize Randomly $\theta_0 = random$
- Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- (nearly) guaranteed to converge to the minimum



$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$$

Gradient Descent

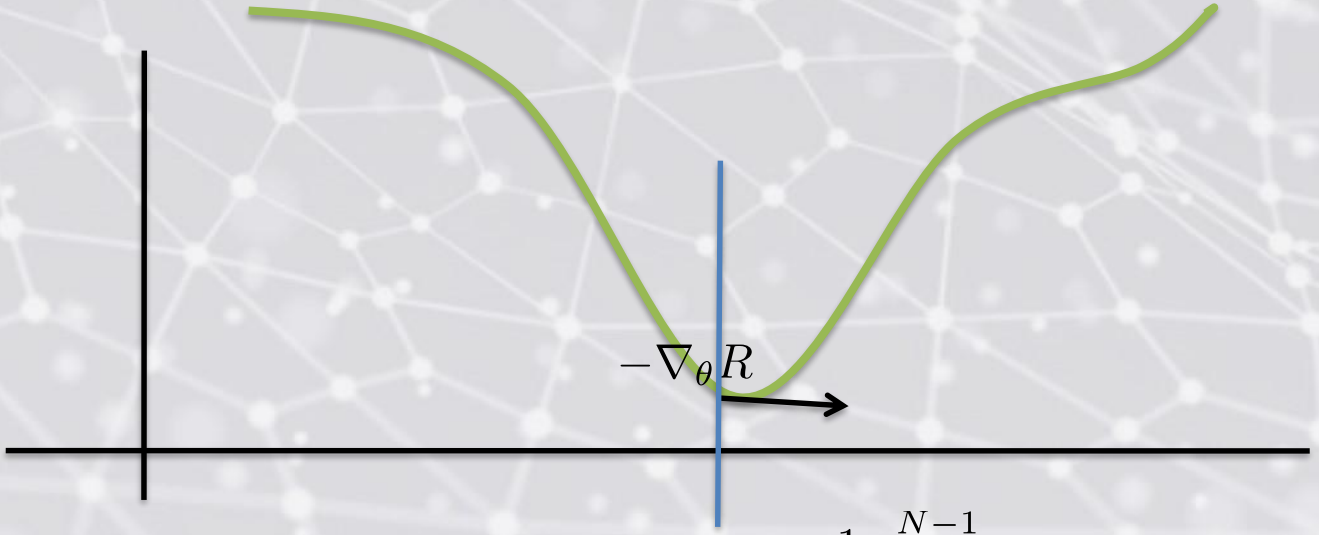
- Initialize Randomly $\theta_0 = random$
- Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- (nearly) guaranteed to converge to the minimum



$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$$

Gradient Descent

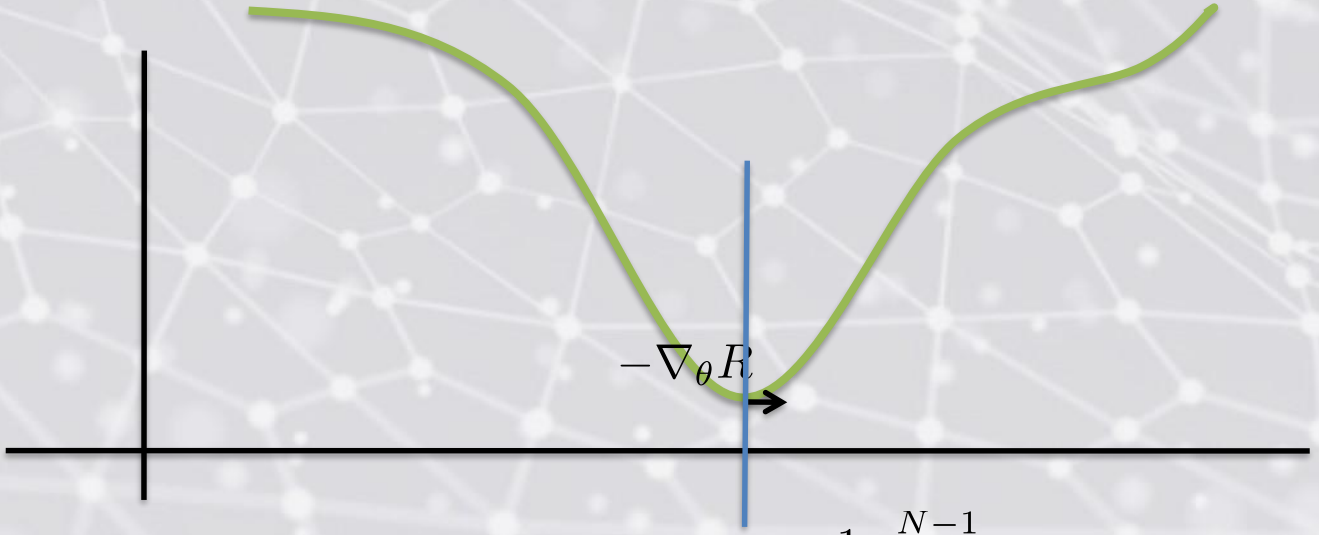
- Initialize Randomly $\theta_0 = random$
- Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- (nearly) guaranteed to converge to the minimum



$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$$

Gradient Descent

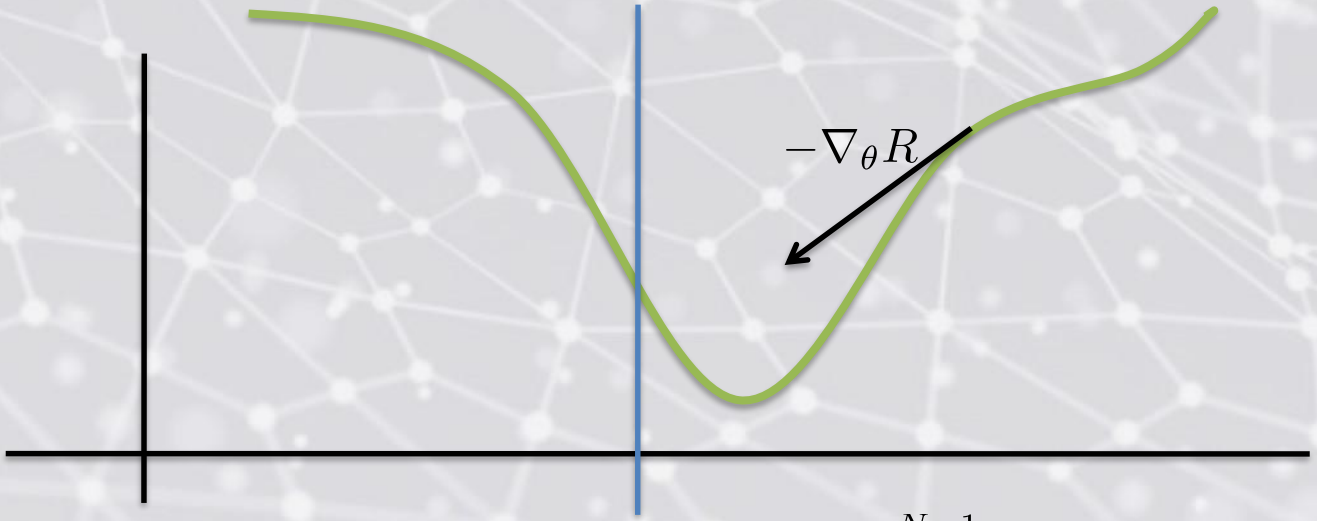
- Initialize Randomly $\theta_0 = random$
- Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- (nearly) guaranteed to converge to the minimum



$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$$

Gradient Descent

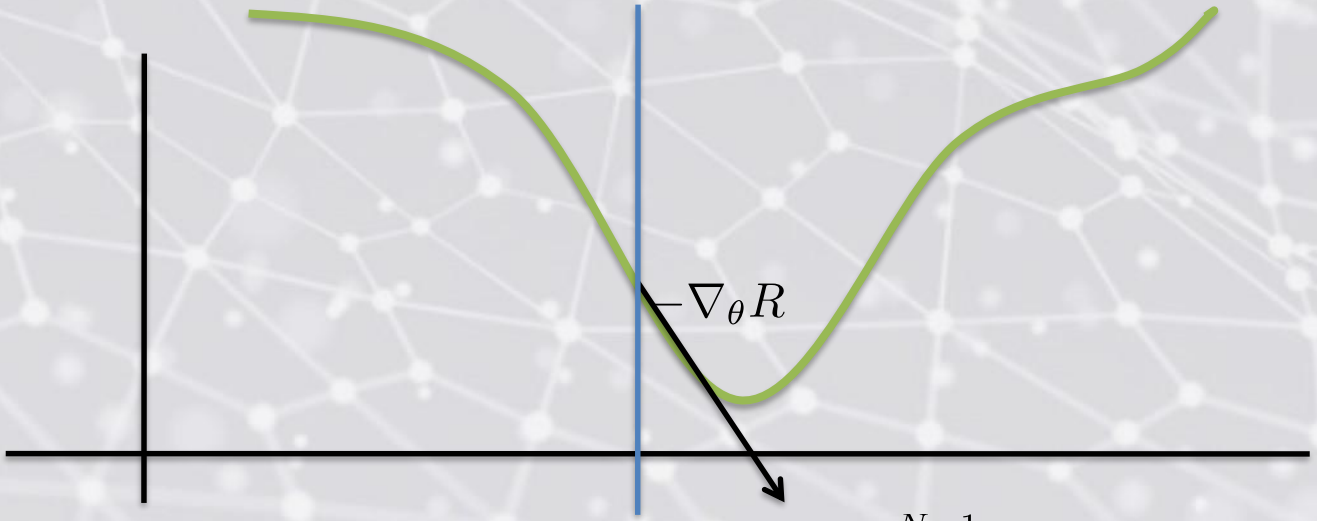
- Initialize Randomly $\theta_0 = random$
- Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- Can oscillate if η is too large



$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$$

Gradient Descent

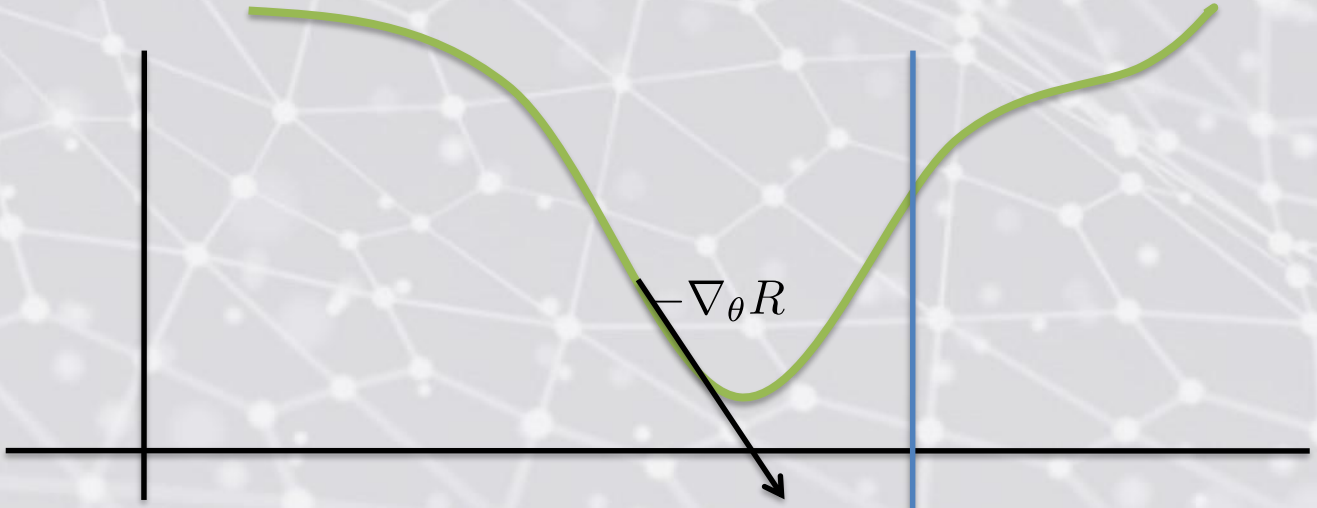
- Initialize Randomly $\theta_0 = random$
- Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- Can oscillate if η is too large



$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$$

Gradient Descent

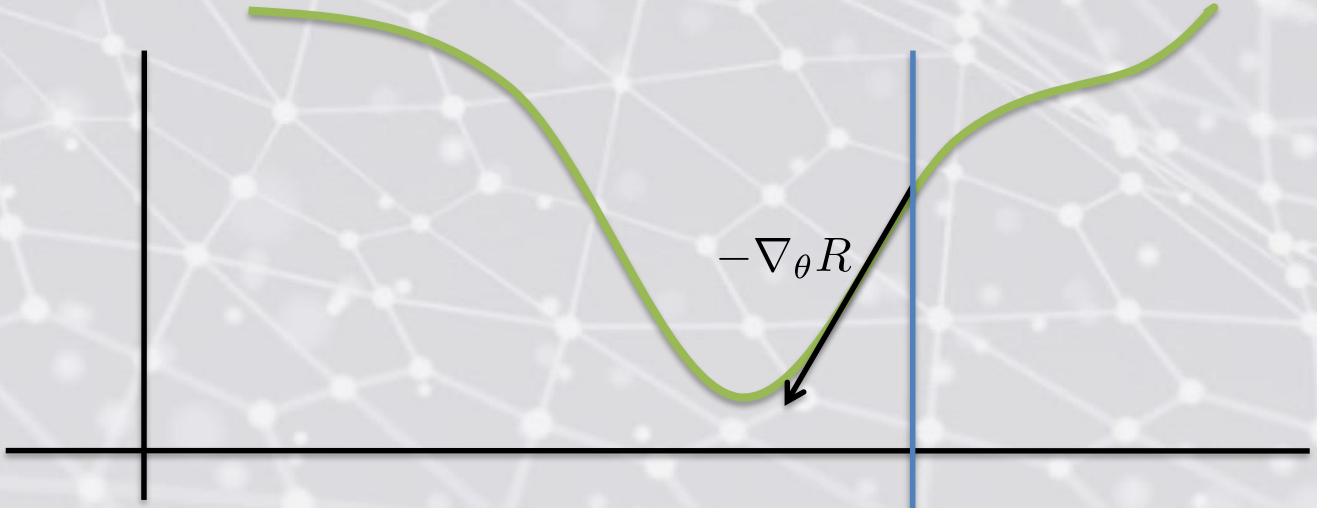
- Initialize Randomly $\theta_0 = random$
- Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- Can oscillate if η is too large



$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$$

Gradient Descent

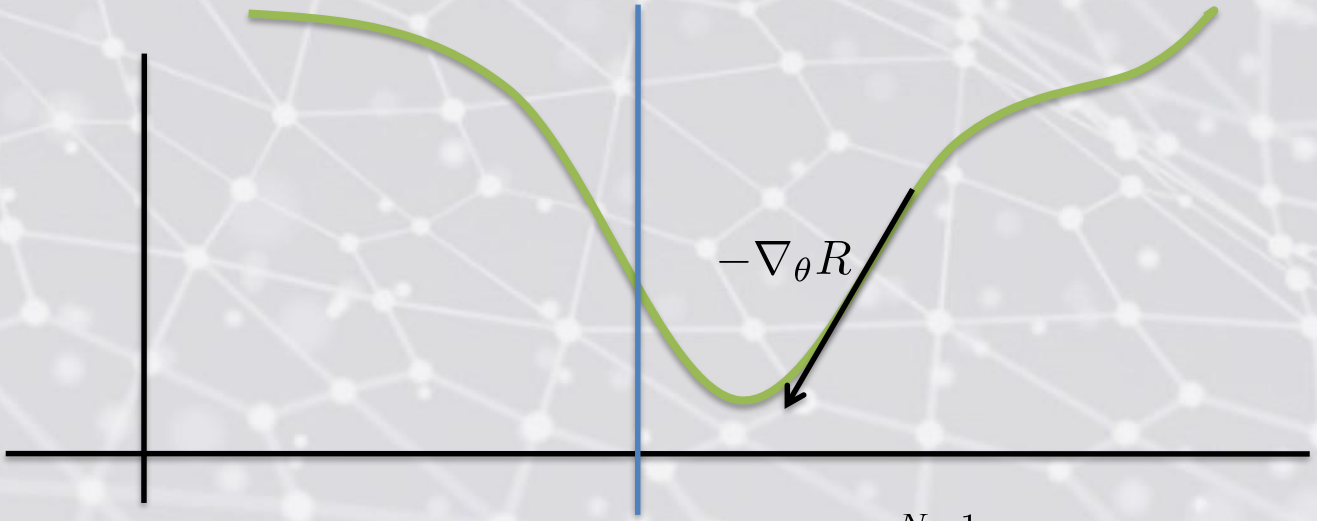
- Initialize Randomly $\theta_0 = random$
- Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- Can oscillate if η is too large



$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$$

Gradient Descent

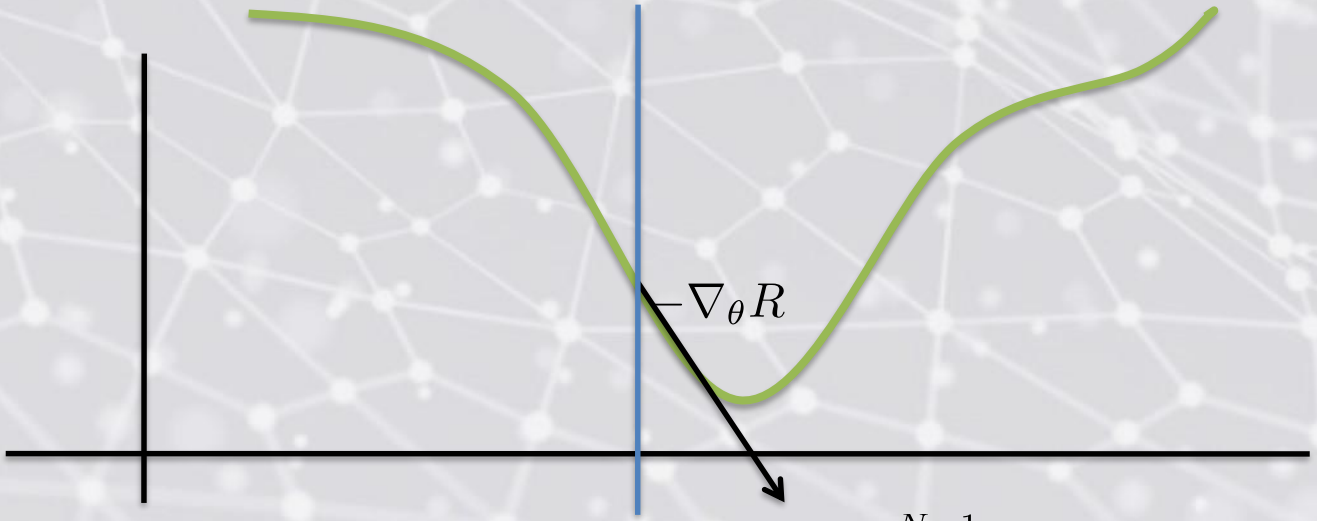
- Initialize Randomly $\theta_0 = random$
- Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- Can oscillate if η is too large



$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$$

Gradient Descent

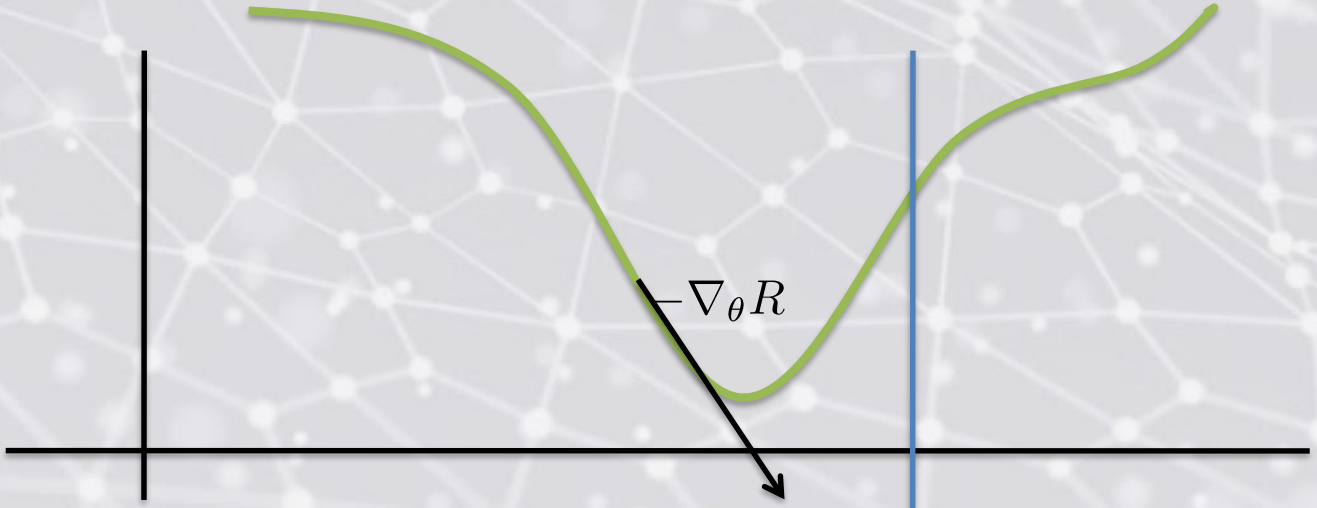
- Initialize Randomly $\theta_0 = random$
- Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- Can oscillate if η is too large



$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$$

Gradient Descent

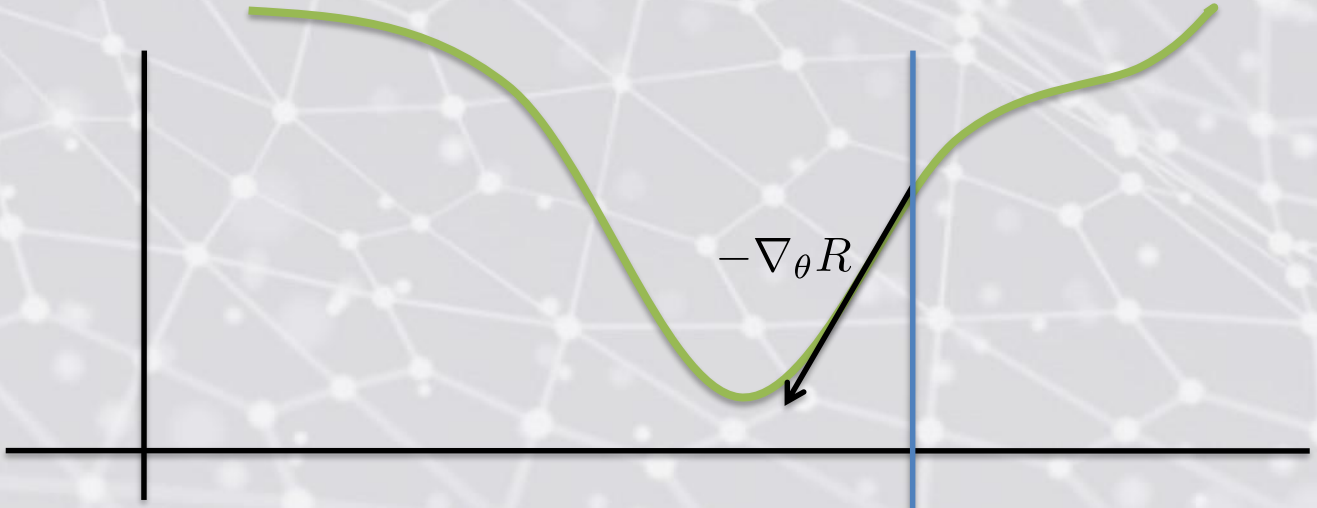
- Initialize Randomly $\theta_0 = random$
- Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- Can oscillate if η is too large



$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$$

Gradient Descent

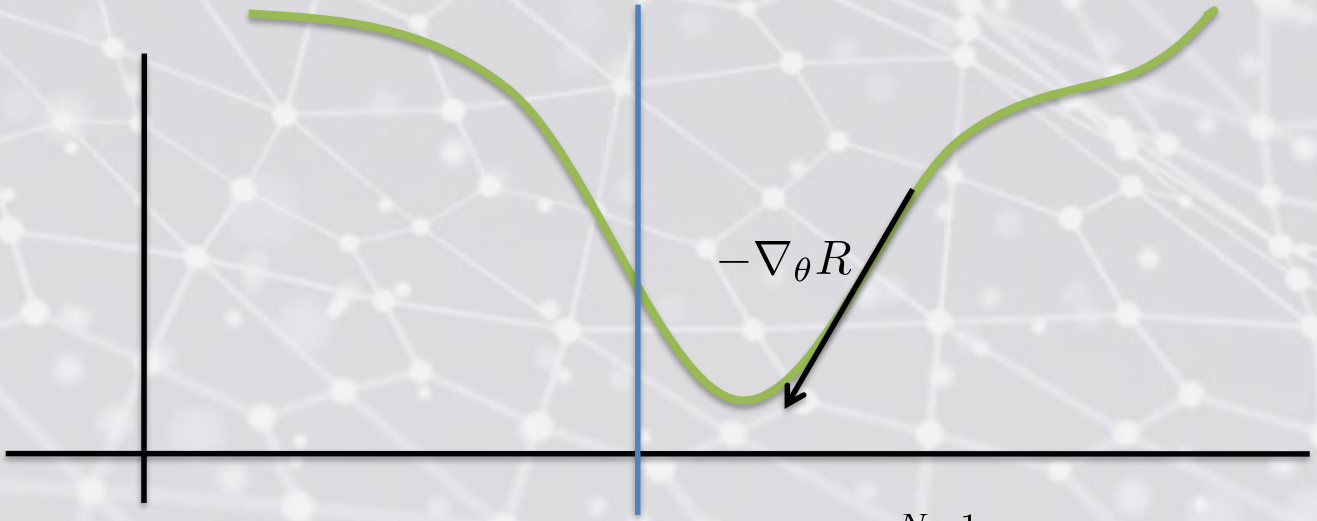
- Initialize Randomly $\theta_0 = random$
- Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- Can oscillate if η is too large



$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$$

Gradient Descent

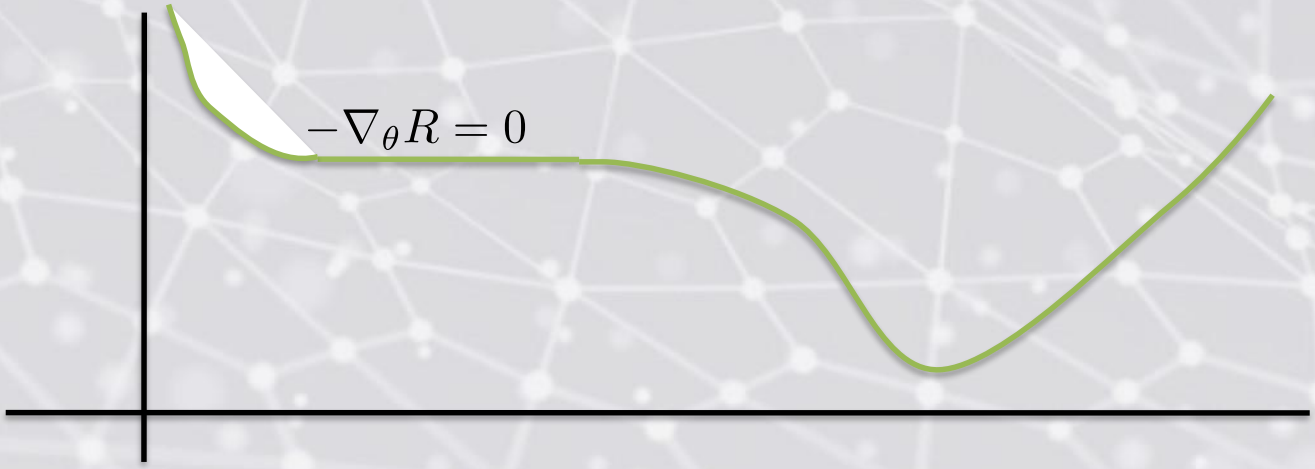
- Initialize Randomly $\theta_0 = random$
- Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- Can oscillate if η is too large



$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$$

Gradient Descent

- Initialize Randomly $\theta_0 = random$
- Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- Can stall if $-\nabla_\theta R$ is ever 0 not at the minimum



$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1) g'(\theta^T x_i) x_i = 0$$

Perceptron Learning

- Find **w** that minimizes average "loss":

$$J(\mathbf{w}) = \frac{1}{M}\sum_{k=1}^{M} L(\mathbf{w}, \mathbf{x}^k, t^k)$$

where M is number of training examples and L is a "loss" function.

- Here, define the loss function as follows:

$$\text{Let } y = a(\mathbf{w} \times \mathbf{x})$$

$$L(\mathbf{w}, \mathbf{x}^k, t^k) = \frac{1}{2}\left(t^k - y\right)^2 \quad \text{"squared loss"}$$

How to solve this minimization problem? **Gradient descent.**

Gradient descent

- To find direction of steepest descent, take the derivative of $J(\mathbf{w})$ with respect to $\mathbf{w}$.

- A vector derivative is called a "gradient": $\nabla J(\mathbf{w})$

$$\nabla J(\mathbf{w}) = \left[\frac{\partial J}{\partial w_0}, \frac{\partial J}{\partial w_1}, \ldots, \frac{\partial J}{\partial w_n} \right]$$

- Here is how we change each weight:

For $i = 0$ to n:

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = -\eta \frac{\partial J}{\partial w_i}$$

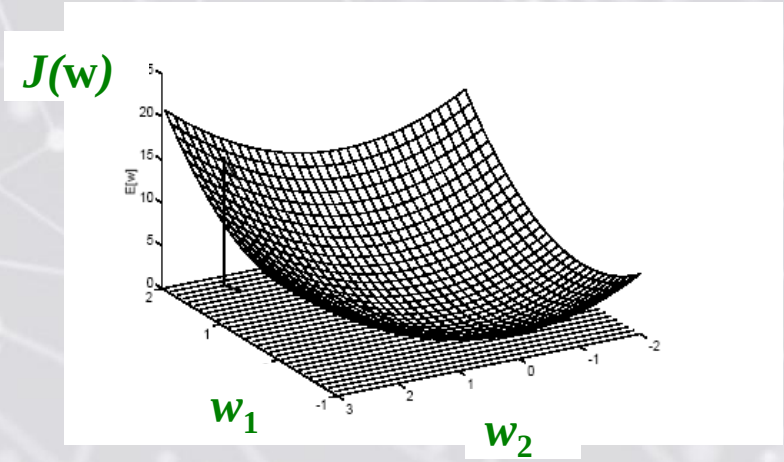
“True” (or “batch”) gradient descent

- One *epoch* = one iteration through the training data.

- After each epoch, compute average loss over the training set:

$$J(\mathbf{w}) = \frac{1}{M}\sum_{k=1}^{M} L(\mathbf{w}, \mathbf{x}^k, t^k) = \frac{1}{M}\sum_{k=1}^{M} \frac{1}{2}\left(t^k - y\right)^2$$

- Change the weights to move in direction of steepest descent in the average-loss surface:



From T. M. Mitchell, *Machine Learning*

- Problem with ***true gradient descent***:

  Training process is slow.

  Training process can land in local optimum.

- Common approach to this: use ***stochastic gradient descent***:
  - Instead of doing weight update after all training examples have been processed, do weight update after each training example has been processed (i.e., perceptron output has been calculated).
  - Stochastic gradient descent approximates true gradient descent increasingly well as $\eta \rightarrow 1/\infty$.

Derivation of perceptron learning rule (stochastic gradient descent)

We defined $J = \frac{1}{2}\left(t - y\right)^2$

Here, use

$J = \frac{1}{2}\left(t - (\mathbf{w} \cdot \mathbf{x})\right)^2$

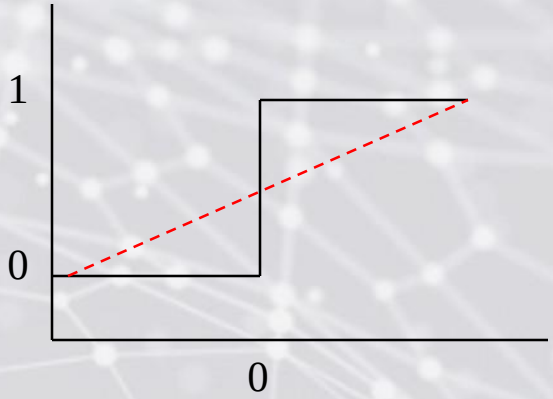
$= \frac{1}{2}\left(t - (w_1 x_1 + w_2 x_2 + \cdots + w_n x_n)\right)^2$

Then,

$\frac{\partial J}{\partial w_i} = -\left(t - (\mathbf{w} \cdot \mathbf{x})\right) x_i^k$

But we'll use

$\frac{\partial J}{\partial w_i} = -\left(t - y\right) x_i^k$



$y = a\left(\mathbf{w} \cdot \mathbf{x}^k\right)$

$y = \mathbf{w} \cdot \mathbf{x}^k$

$$\Delta w_i = -\eta \frac{\partial J}{\partial w_i} = \eta (t^k - y^k) x_i^k$$

This is called the "perceptron learning rule"

Perceptron Learning Algorithm

Start with small random weights, $\mathbf{w} = (w_0, w_1, w_2, \ldots, w_n)$, where $w_i \in [-.05, .05]$

Repeat until accuracy on the training data stops increasing or for a maximum number of *epochs* (iterations through training set):

For $k = 1$ to M (total number in training set):

1. Select next training example $(\mathbf{x}^k, t^k)$.
2. Run the perceptron with input $\mathbf{x}^k$ and weights $\mathbf{w}$ to obtain y.
3. If $y \neq t^k$, update weights:

   for $i = 0, \ldots, n$: ; Note that bias weight w_0 is changed just like all other weights!

   $w_i \leftarrow w_i + \Delta w_i$

   where

   $\Delta w_i = \eta (t^k - y^k) x_i^k \quad (or\ equivalently: \Delta w_i = -\eta (y^k - t^k) x_i^k)$

1. Go to 1.

Example: logical OR

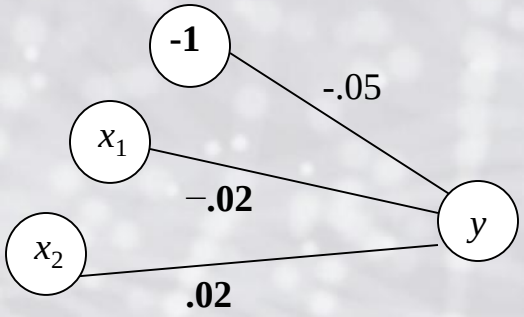
Training set:

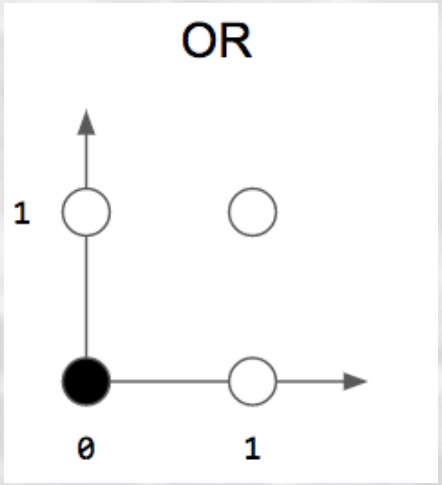
((0, 0), 0)

(0, 1), 1)

(1, 0), 1)

(1,1),1))





Initial weights:

$\{w_0, w_1, w_2\} = \{-.05, -0.2, 0.2\}$

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = -\eta\,(y^k - t^k)x_i^k$$

Apply perceptron learning rule for one epoch with $\eta = 0.25$

(*) Please note that the text uses “-1” for the extra input weight that corresponds with the bias (w_0). It is slightly more conventional to use “+1” so please note we will also use this convention from time to time. (Either value generates a model of comparable expressive power)

Example: logical OR

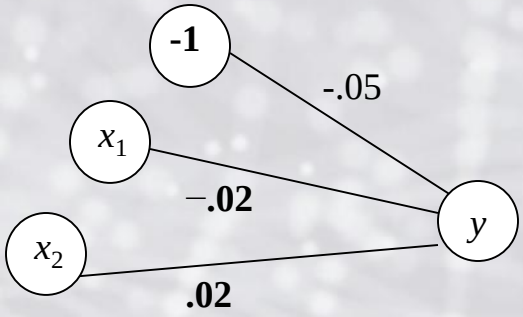
Training set:

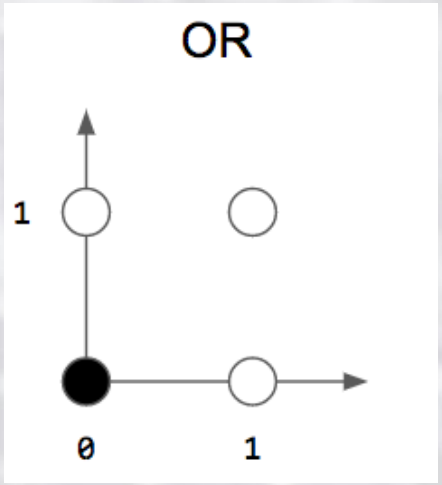
((0, 0), 0)

(0, 1), 1)

(1, 0), 1)

(1,1),1))





Initial weights:

$\{w_0, w_1, w_2\} = \{-.05, -0.2, 0.2\}$

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = -\eta\,(y^k - t^k)\,x_i^k$$

$$Input: (0,0)$$

$$w_0 : -0.05 - 0.25 \times (1-0) \times -1 = 0.2$$

$$w_1 : -0.02 - 0.25 \times (1-0) \times 0 = -0.02$$

$$w_2 : 0.02 - 0.25 \times (1-0) \times 0 = 0.02$$

$$Input: (0,1)$$

$$w_0 : 0.2 - 0.25 \times (0-1) \times -1 = -.05$$

$$w_1 : -0.02 - 0.25 \times (0-1) \times 0 = -0.02$$

$$w_2 : 0.02 - 0.25 \times (0-1) \times 1 = 0.27$$

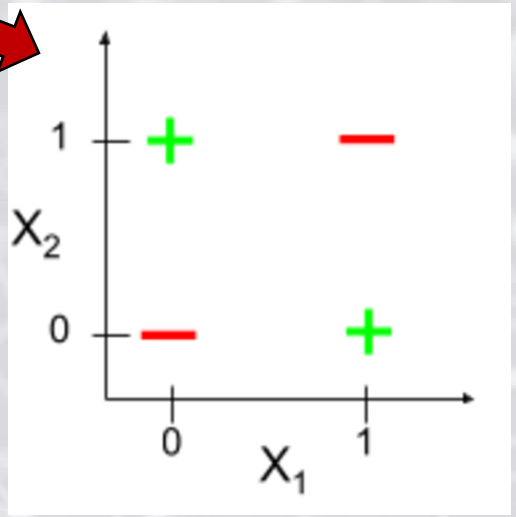
No weight updates are needed for inputs: (1,0) and (1,1); why?

Perceptron Learning

Does the Perceptron Learning Algorithm (PLA) always work?

Q: Will the PLA yield a solution to the logical XOR problem?

Why/why not?



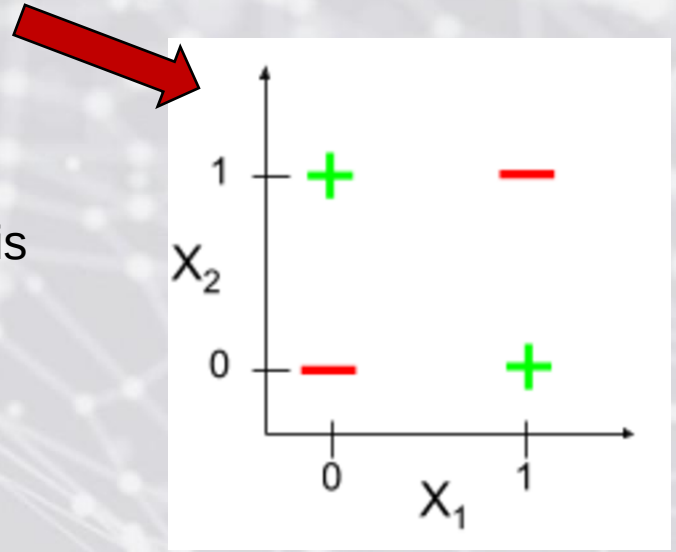
Perceptron Learning

Does the Perceptron Learning Algorithm (PLA) always work?

Q: Will the PLA yield a solution to the logical XOR problem?

Why/why not?

The PLA will not yield a solution, since the data is not linearly separable.



Perceptron Learning

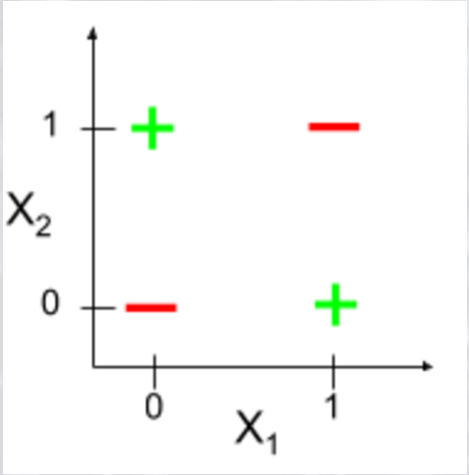
Does the Perceptron Learning Algorithm (PLA) always work?

Q: Will the PLA yield a solution to the logical XOR problem?

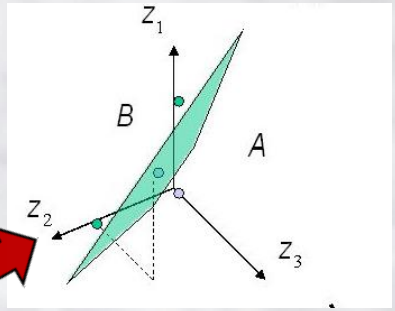
Why/why not?

The PLA will not yield a solution, since the data is not linearly separable.

Q: Does this mean the classification problem is hopeless?



Perceptron Learning



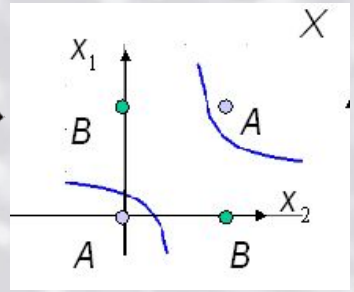
Q: Does this mean the classification problem is hopeless?

No! There are at least (3) remedies:

(1) Project the XOR problem into a higher dimensional space (e.g. 3-space) where it is linearly separable!

(2) Use a kernel/polynomial decision boundary in 2-space.

(3) Use a multi-layer perceptron (i.e. a neural network).



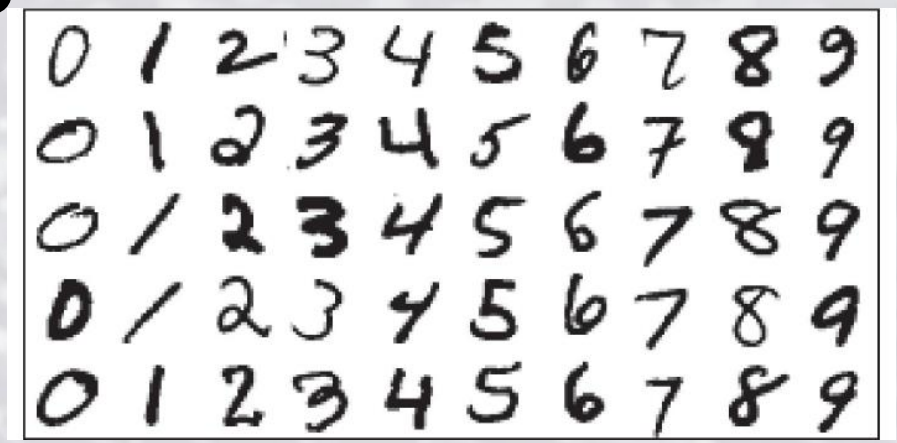
(*) Minsky and Papert (1969) published an influential (viz. notorious) text, “Perceptrons”, that identified the learning capabilities and limitations of Perceptrons.

(*) The major effect of this text, unfortunately, was to set back NN research for two decades (see: “AI Winter”)

(*) What brought it back? LeCun, et al. 1990s MNIST results, etc.

Recognizing Handwritten Digits

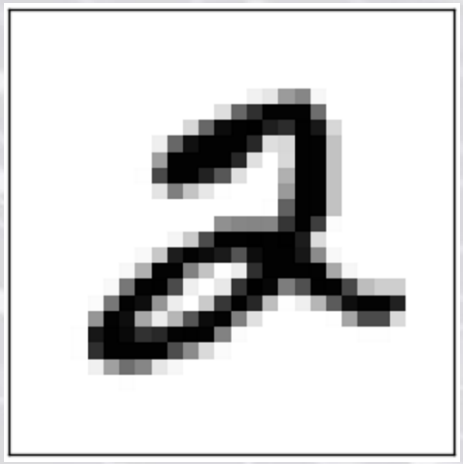
- MNIST dataset
  - 60,000 training examples
  - 10,000 test examples



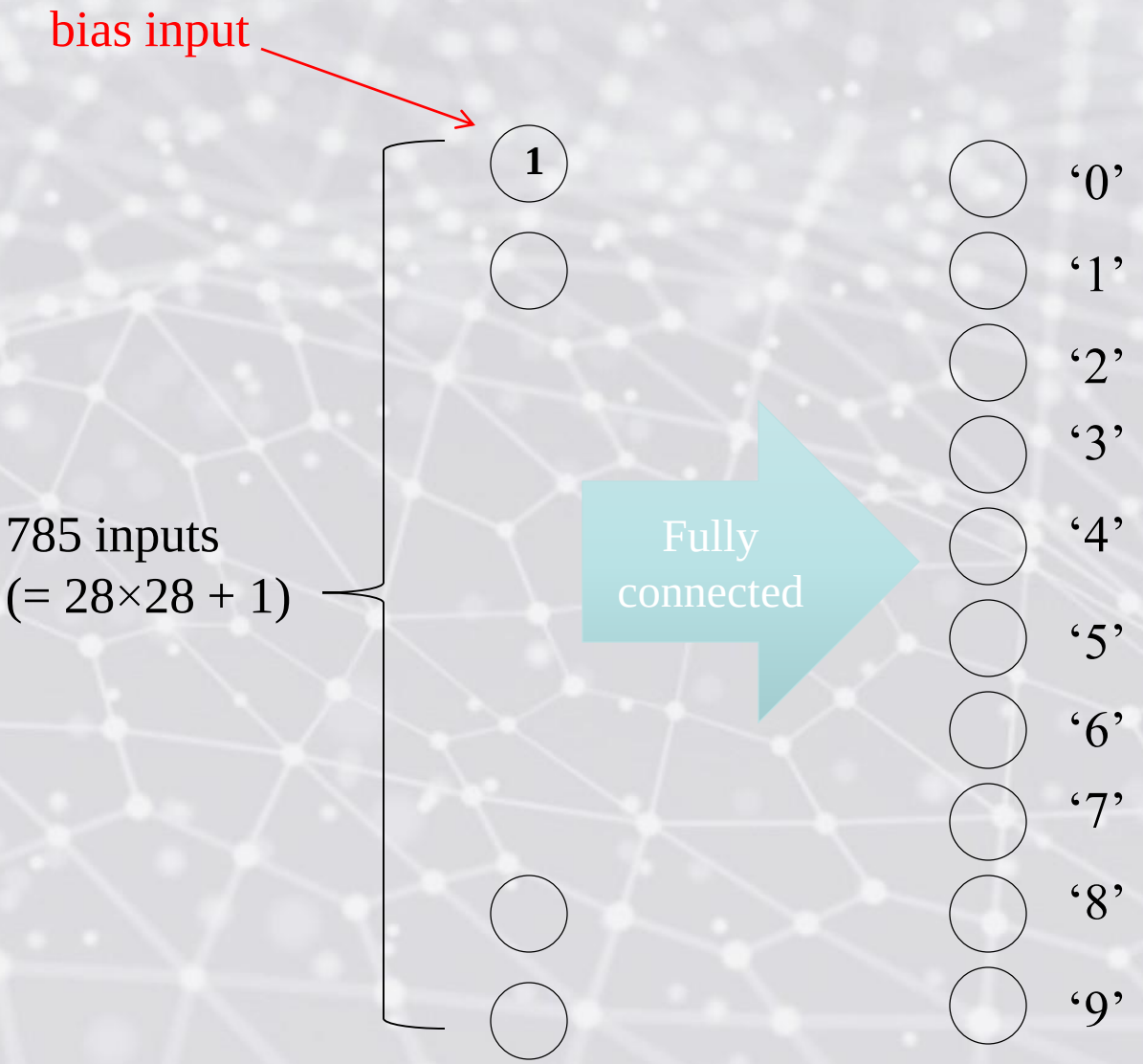
Each example is a 28×28-pixel image, where each pixel is a grayscale value in [0,255].

See csv files.

First value in each row is the target class.



Perceptron architecture for handwritten digits classification

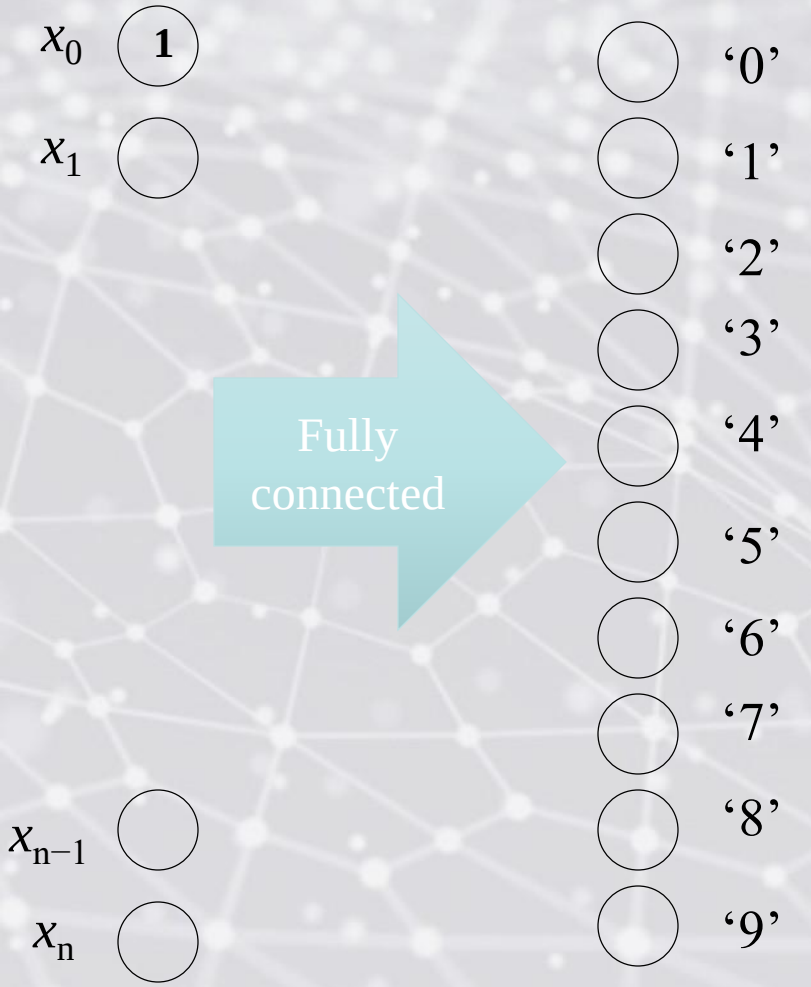


Preprocessing
(To keep weights from growing very large)

Scale each feature to a fraction between 0 and 1:

$$x_i' = \frac{x_i}{255}$$

Processing an input



At each output, compute:

$$\sum_{i=0}^{n} w_i x_i$$

The output with highest value is the prediction.

If correct, do nothing.
If not correct, adjust weights according to perceptron learning rule.

See assignment for details.

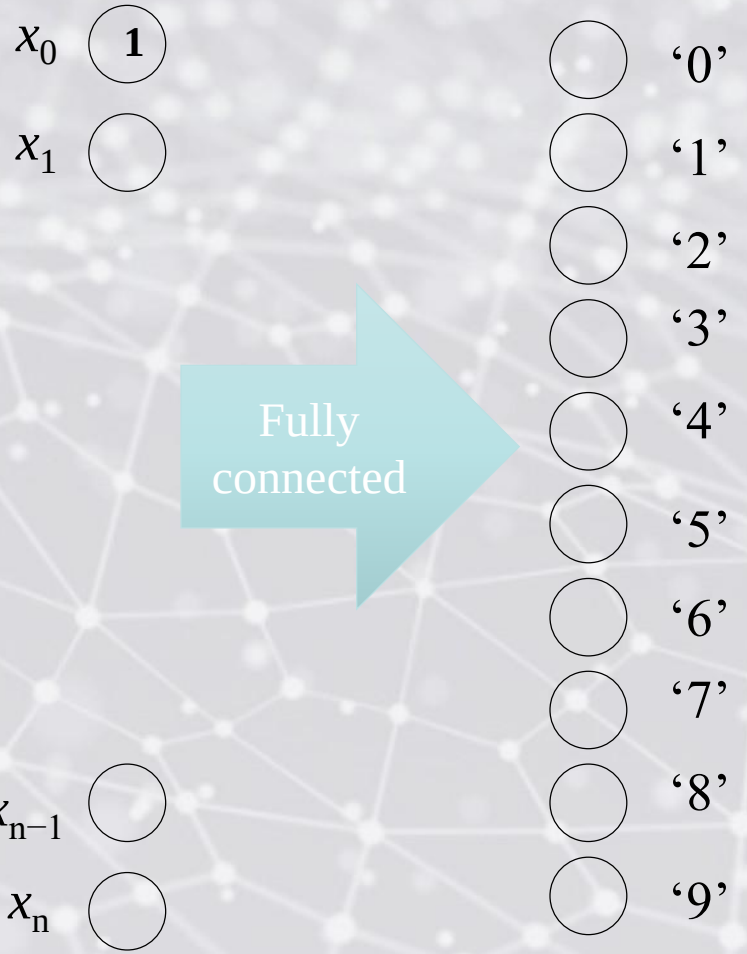
Example

For each training example k, input $\mathbf{x}^k$ to the perceptron.

Suppose $\mathbf{x}^k$ is a ‘2’.

$t^k = 1$ for the ‘2’ output.

$t^k = 0$ for all other outputs.



For each output, calculate

$$\sum_{i=0}^{735} w_i x_i$$

Set $y = 1$ for outputs that are greater than 0. Set $y = 0$ otherwise.

Then for all weights coming into each output:

$$w_i \leftarrow w_i + \Delta w_i$$

where

$$\Delta w_i = \eta\,(t^k - y^k)x_i^k$$

Homework 1

- Implement perceptron (785 inputs, 10 outputs) and perceptron learning algorithm in any programming language you would like.

- Download MNIST data from website

- Preprocess MNIST data: Scale each feature to a fraction between 0 and 1.

Homework 1 Experiments

Train perceptrons with three different learning rates:

η = 0.001, 0.01, and 0.1

For each learning rate:

1. Choose small random weights $w_i \in [-.05, .05]$

2. Repeat cycling through the training data until the accuracy on the training data has essentially stopped improving (i.e., the difference between training accuracy from one epoch to the next is less than some small number, like 0.01.)

After the initialization, and after each epoch (one cycle through training data), compute accuracy on training and test set (for plot), without changing weights.

Homework 1: Presenting results

For each experiment, plot training and test accuracy over epochs:

For each experiment, give confusion matrix:

Predicted class →

Actual class ↓

	0	1	2	3	4	5	6	7	8	9
0										
1										
2										
3										
4										
5										
6										
7										
8										
9										

In each cell (i,j). put number of test examples that were predicted (classified) as class i, and whose actual class is class j.

Homework 1 FAQ

Q: Can I use code for perceptrons from another source?

A: No, you need to write your own code.

Q: How long will it take to train the perceptrons?

A: Depends on your computer and your code, but probably an hour or more.

Q: What accuracy should I expect on the test set?

A: It depends on initial weights, but probably over 80%.

Q: Should I wait until the last minute?

A: No!!! Start as soon as possible.

Perceptron Convergence Theorem

(*) Rosenblatt (1962) proved that:

Given a linearly separable dataset, the Perceptron will converge to a solution that separates the classes, and that it will do it after a finite number of iterations.

Pf. Let: γ := the distance between separating hyperplane and Closest data point; let w^* be a unit weight vector that separates the data (known to exist by assumption of linear separability).

WLOG, assume $\|x\| \leq 1$ for all input data.

Recall to check "similarity" of two vectors (in particular, "perfect alignment" connotes parallel).

Perceptron Convergence Theorem

Recall: to check "similarity" of two vectors we take their dot product (in particular, "perfect alignment" connotes parallel); when two vectors are parallel, their inner product is maximal.

If we therefore show that at each weight update, $w^* \cdot w$ Increases, then we have nearly show that the algorithm will converge.

However, we do need to check that the length of w does not increase too much as well, since $\vec{w}^* \cdot \vec{w} = \|\vec{w}^*\|\|\vec{w}\|\cos\theta$.

In summary, we need (2) checks:

(1) $\vec{w}^* \cdot \vec{w}$

(2) $\|\vec{w}\|$

Perceptron Convergence Theorem

Suppose that at the *t*th iteration of the algorithm, the network sees a particular x that should output y, and that it gets this input wrong, so:

$$y\vec{w}^{(t-1)} \cdot \vec{x} < 0$$

(*) This means that the weights need to be updated.

(*) The weight update will be: $\vec{w}^{(t)} = \vec{w}^{(t-1)} + y\vec{x}$ (where η=1, WLOG)

Consider: $\vec{w}^* \cdot \vec{w}^{(t)} = \vec{w}^* \cdot \left(\vec{w}^{(t-1)} + y\vec{x} \right)$

Perceptron Convergence Theorem

Suppose that at the *t*th iteration of the algorithm, the network sees a particular x that should output y, and that it gets this input wrong, so:

$$y\vec{w}^{(t-1)} \cdot \vec{x} < 0$$

(*) This means that the weights need to be updated.

(*) The weight update will be: $\vec{w}^{(t)} = \vec{w}^{(t-1)} + y\vec{x}$ (where η=1, WLOG)

Consider: $\vec{w}^* \cdot \vec{w}^{(t)} = \vec{w}^* \cdot \left(\vec{w}^{(t-1)} + y\vec{x} \right)$

$= \vec{w}^* \cdot \vec{w}^{(t-1)} + y\vec{w}^* \cdot \vec{x}$  Why?

Perceptron Convergence Theorem

Suppose that at the *t*th iteration of the algorithm, the network sees a particular x that should output y, and that it gets this input wrong, so:

$$y\vec{w}^{(t-1)} \cdot \vec{x} < 0$$

(*) This means that the weights need to be updated.

(*) The weight update will be: $\vec{w}^{(t)} = \vec{w}^{(t-1)} + y\vec{x}$ (where η=1, WLOG)

Consider: $\vec{w}^* \cdot \vec{w}^{(t)} = \vec{w}^* \cdot \left(\vec{w}^{(t-1)} + y\vec{x} \right)$

$$= \vec{w}^* \cdot \vec{w}^{(t-1)} + y\vec{w}^* \cdot \vec{x}$$

$$\geq \vec{w}^* \cdot \vec{w}^{(t-1)} + \gamma$$

Why?

Perceptron Convergence Theorem

$$\vec{w}^* \cdot \vec{w}^{(t)} = \vec{w}^* \cdot \left(\vec{w}^{(t-1)} + y\vec{x} \right)$$

$$= \vec{w}^* \cdot \vec{w}^{(t-1)} + y\vec{w}^* \cdot \vec{x}$$

$$\geq \vec{w}^* \cdot \vec{w}^{(t-1)} + \gamma$$

This means that at each update of the weights, this inner product increases by at least γ, and so after t updates of the weights:

$$\vec{w}^* \cdot \vec{w}^{(t)} \geq t\gamma$$

(*) We can use this to put a bound on the length of $\|w(t)\|$ using the Cauchy-Schwartz inequality, which tells us:

$$\vec{w}^* \cdot \vec{w}^{(t)} \leq \left\|\vec{w}^*\right\| \left\|\vec{w}^{(t)}\right\| \text{ and so, } \left\|\vec{w}^{(t)}\right\| \geq t\gamma$$

Perceptron Convergence Theorem

$$\left\|\vec{w}^{(t)}\right\| \geq t\gamma$$

The length of the weight vector after *t* steps is:

$$\left\|\vec{w}^{(t)}\right\|^2 = \left\|\vec{w}^{(t-1)} + y\vec{x}\right\|^2$$

$$= \left\|\vec{w}^{(t-1)}\right\|^2 + y^2\left\|\vec{x}\right\|^2 + 2y\vec{w}^{(t-1)} \cdot \vec{x}$$

Why?

$$\leq \left\|\vec{w}^{(t-1)}\right\|^2 + 1$$

(*) The last inequality holds: since $y^2=1$, $\|\vec{x}\| \leq 1$, and the network made an error so $\vec{w}^{(t-1)}$ and $\vec{x}$ yield a negative value (when we assume y=1, WLOG).

(*) The inequality above shows that: $\left\|\vec{w}^{(t)}\right\|^2 \leq t$

(*) If we put these two inequalities together, we get:

$$t\gamma \leq \left\|\vec{w}^{(t-1)}\right\| \leq \sqrt{t}$$

Perceptron Convergence Theorem

$$t\gamma \leq \left\|\vec{w}^{(t-1)}\right\| \leq \sqrt{t}$$

(*) So $t \leq 1/\gamma^2$

(*) Thus after this many updates the algorithm must have converged. QED

(*) We showed that the quantity: $\vec{w}^* \cdot \vec{w} = \|\vec{w}^*\|\|\vec{w}\|\cos\theta$ increases monotonically and it is upper-bounded.

Summary: We demonstrated that if the data are linearly separable, then the algorithm will converge and that the time this takes is a function of the distance between the separating hyperplane and the nearest data point (this distance is called the margin).

If the data are not linearly separable, then the algorithm is not guaranteed to converge and may oscillate infinitely.

Neuron inspires Regression

- Edges multiply the signal (x_i) by some weight (θ_i).
- Nodes sum inputs
- Equivalent to Linear Regression

$$f(x,\theta) = \sum_{d=1}^{D} \theta_d x^d + \theta_0$$