Machine Learning: Part I

UMaine COS 470/570 – Introduction to AI

Spring 2019

Introduction

What are ANNs?

Artificial neural networks

- Systems of simple computing elements: *neurons*
- Each neuron accepts inputs from others, produces *activation*
- Neurons connected via *weights* that modulate activation
- Can be viewed as:
  - Pattern-learning (inductive) systems
  - Statistical programs
  - Dimension/feature-changing systems
  - Search programs (in weight space)

What can they do?

- Image classification and labeling
- Word recognition
- Natural language systems
- Machine translation systems
- General pattern recognition
- Superhuman-level performance on games, other RL tasks
- Pattern generators (images, music, . . .)

Inspiration: Natural pattern recognition

- Pattern recognition in natural world:
  - Chemoreceptors
  - Immune system
  - Biological neural networks
    - Animal/human vision system
    - Auditory system
    - Neocortex
    - Etc.

Neural systems

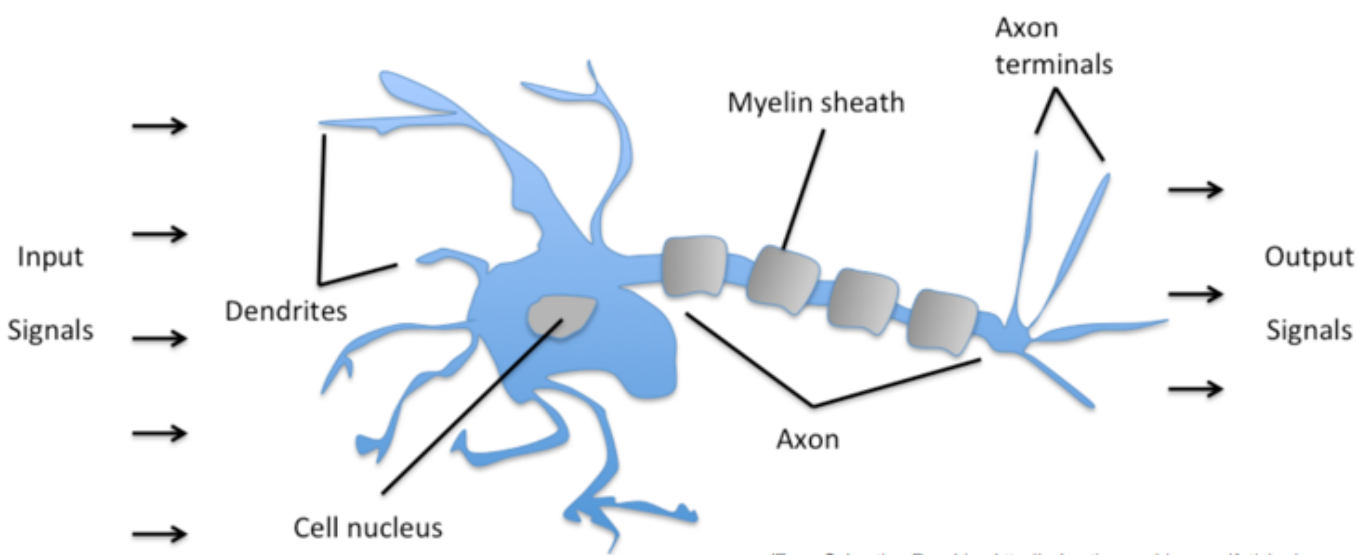
- Most flexible pattern recognizers:
- Biological computing elements: Neurons
- Neurons are *excitatory* cells
- Connections determine how activation spreads

Problem: Complexity

- Neurons are *very* complex

Problem: Complexity

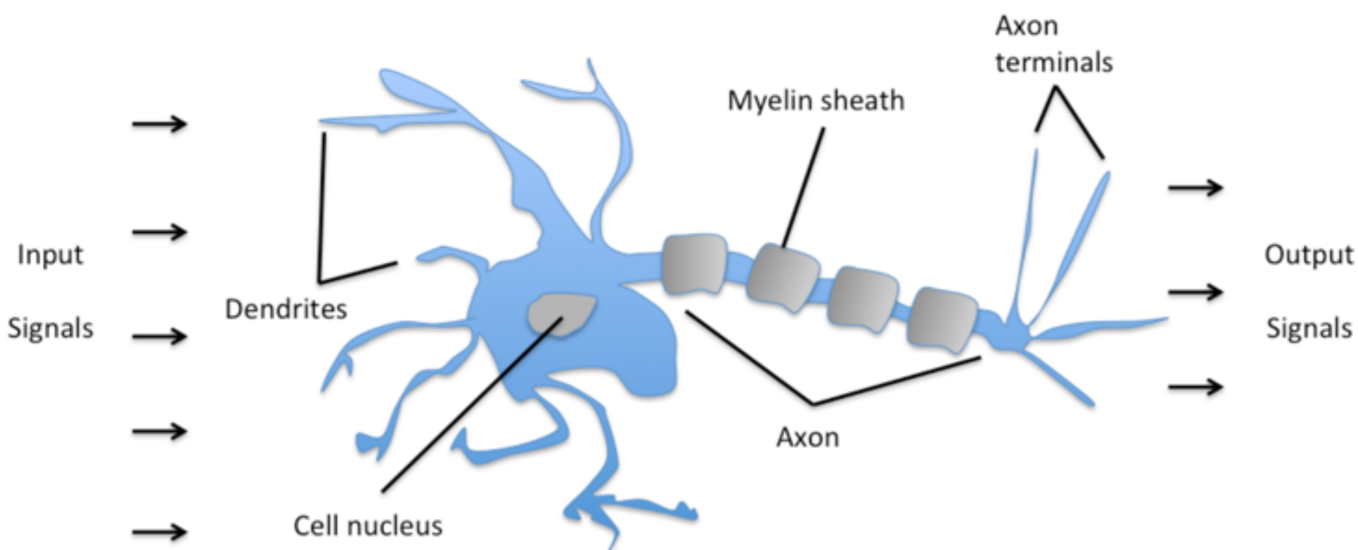
- Neurons are *very* complex



(From Sebastian Raschka, http://sebastianraschka.com/Articles/2015_singlelayer_neurons.html#artificial-neurons-and-the-mcculloch-pitts-model.)

Problem: Complexity

- Neurons are *very* complex

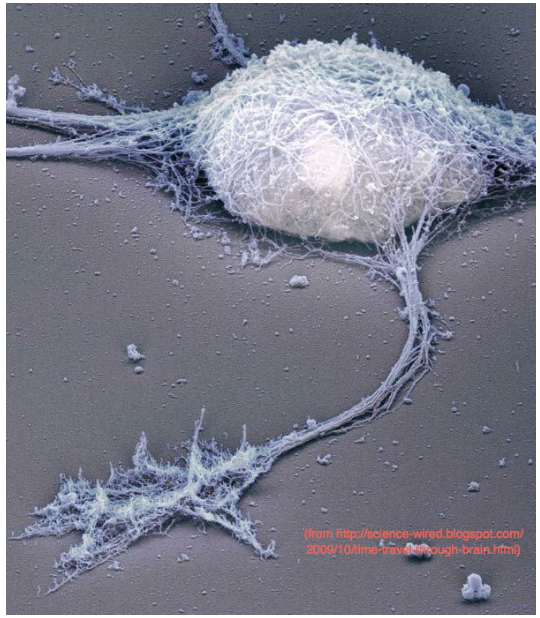


(From Sebastian Raschka, http://sebastianraschka.com/Articles/2015_singlelayer_neurons.html#artificial-neurons-and-the-mcculloch-pitts-model.)

- Synapses: change potential across cell membrane
- Neuron effectively sums excitations, inhibitions
- At some point: potential at threshold and neuron *fires*
- Excitatory pulse down axon, release neurotransmitter at synapses
- *Lots* more to it than this!

Problem: Complexity

- Neurons are *very* complex

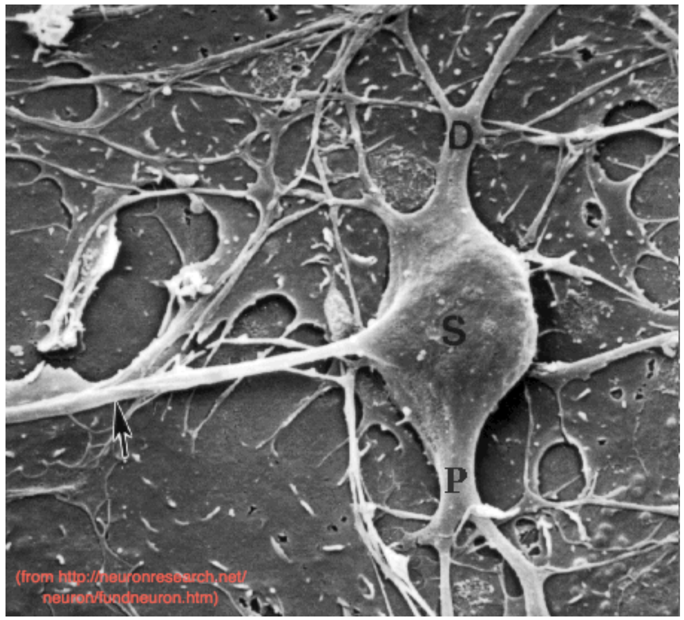


- Synapses: change potential across cell membrane
- Neuron effectively sums excitations, inhibitions
- At some point: potential at threshold and neuron *fires*
- Excitatory pulse down axon, release neurotransmitter at synapses
- *Lots* more to it than this!

Artificial Intelligence

Problem: Complexity

- Neurons are *very* complex



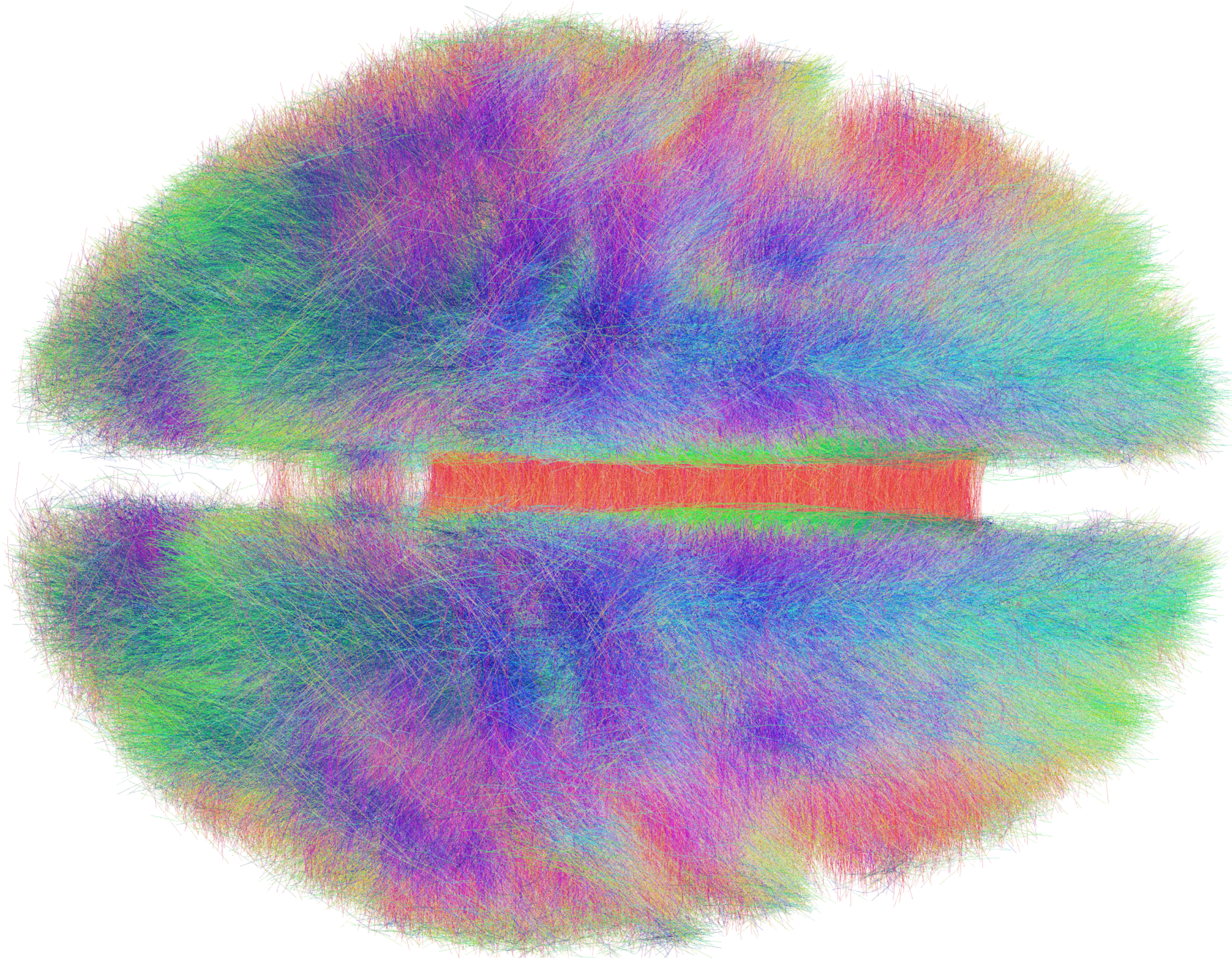
- Synapses: change potential across cell membrane
- Neuron effectively sums excitations, inhibitions
- At some point: potential at threshold and neuron *fires*
- Excitatory pulse down axon, release neurotransmitter at synapses
- *Lots* more to it than this!

Problem: Complexity

- Connectsome is incredibly complex

Problem: Complexity

- Connectsome is incredibly complex



Andreashorn [CC BY-SA 4.0 (https://creativecommons.org/licenses/by-sa/4.0)

Perceptrons

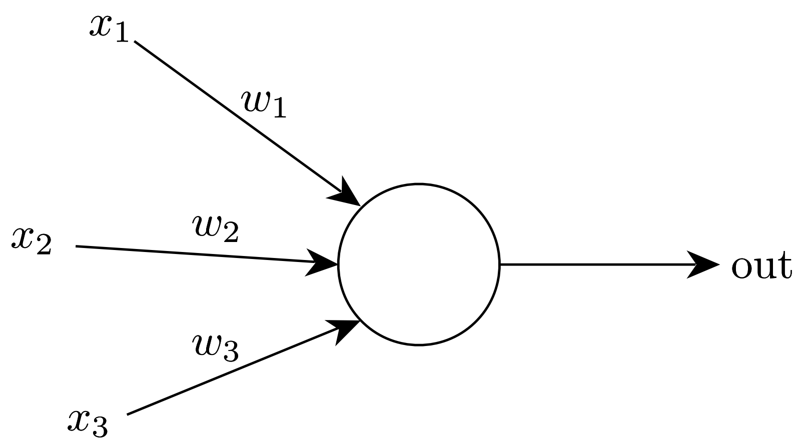
Perceptrons

First simple artificial neuron: Perceptron

- McCulloch & Pitts
- *Very* simple model of a neuron

First simple artificial neuron: Perceptron

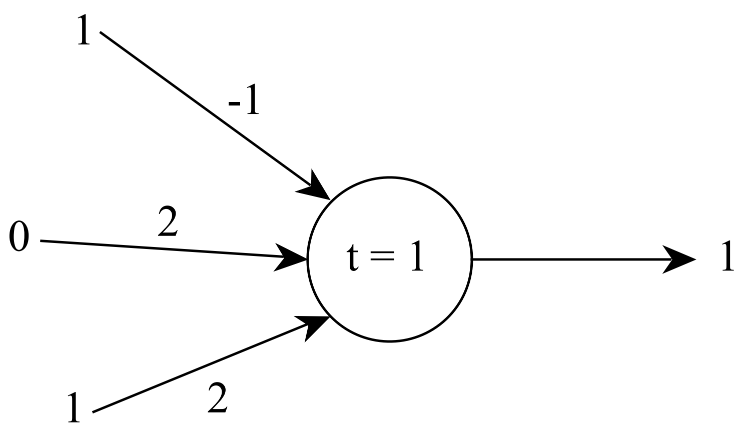
- McCulloch & Pitts
- *Very* simple model of a neuron



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

First simple artificial neuron: Perceptron

- McCulloch & Pitts
- *Very* simple model of a neuron



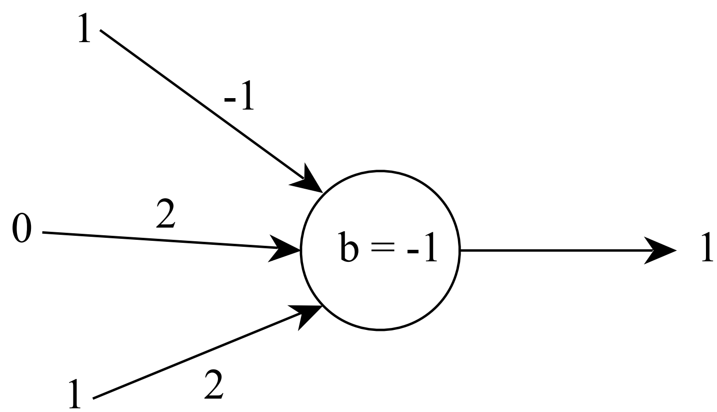
$$\boxed{\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{otherwise} \end{cases}}$$

First simple artificial neuron: Perceptron

- McCulloch & Pitts
- *Very* simple model of a neuron
- Usually change threshold to *bias* $(= -\text{threshold})$

First simple artificial neuron: Perceptron

- McCulloch & Pitts
- *Very* simple model of a neuron
- Usually change threshold to *bias* (= −threshold)



$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j + b \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

What can they do?

- "Weigh evidence" $\Rightarrow$ decision

What can they do?

- "Weigh evidence" $\Rightarrow$ decision
- E.g.:

What can they do?

- "Weigh evidence" $\Rightarrow$ decision
- E.g.:
  - output = "study"

What can they do?

- "Weigh evidence" $\Rightarrow$ decision
- E.g.:
  - output = "study"
  - $x_1 =$ test on Monday, $x_2 =$ confident of material, $x_3 =$ doing poorly in class

What can they do?

- "Weigh evidence" $\Rightarrow$ decision
- E.g.:
  - output = "study"
  - x_1 = test on Monday, x_2 = confident of material, x_3 = doing poorly in class
  - $w_1 = 1$, $w_2 = \text{-}1$, $w_3 = 2$

What can they do?

- "Weigh evidence" $\Rightarrow$ decision
- E.g.:
  - output = "study"
  - x_1 = test on Monday, x_2 = confident of material, x_3 = doing poorly in class
  - $w_1 = 1$, $w_2 = -1$, $w_3 = 2$
  - bias = 0

What can they do?

- "Weigh evidence" $\Rightarrow$ decision
- E.g.:
  - output = "study"
  - x_1 = test on Monday, x_2 = confident of material, x_3 = doing poorly in class
  - $w_1 = 1$, $w_2 = -1$, $w_3 = 2$
  - bias = 0
  - Test on Monday, confident, doing well in class $\Rightarrow$ output $= 0$

What can they do?

- "Weigh evidence" $\Rightarrow$ decision
- E.g.:
  - output = "study"
  - $x_1 =$ test on Monday, $x_2 =$ confident of material, $x_3 =$ doing poorly in class
  - $w_1 = 1$, $w_2 =$ -1, $w_3 = 2$
  - bias = 0
  - Test on Monday, confident, doing well in class $\Rightarrow$ output $= 0$
  - Test on Monday, not confident, doing well $\Rightarrow$ output $= 1$

What can they do?

- "Weigh evidence" $\Rightarrow$ decision
- E.g.:
  - output = "study"
  - x_1 = test on Monday, x_2 = confident of material, x_3 = doing poorly in class
  - $w_1 = 1$, $w_2 = -1$, $w_3 = 2$
  - bias = 0
  - Test on Monday, confident, doing well in class $\Rightarrow$ output $= 0$
  - Test on Monday, not confident, doing well $\Rightarrow$ output $= 1$
  - Test on Monday, confident, doing poorly: $1 + (-1) + 2 = 2 \Rightarrow$ output $= 1$

Learning the weights

- Rosenblatt's perceptron algorithm

Artificial Intelligence

Learning the weights

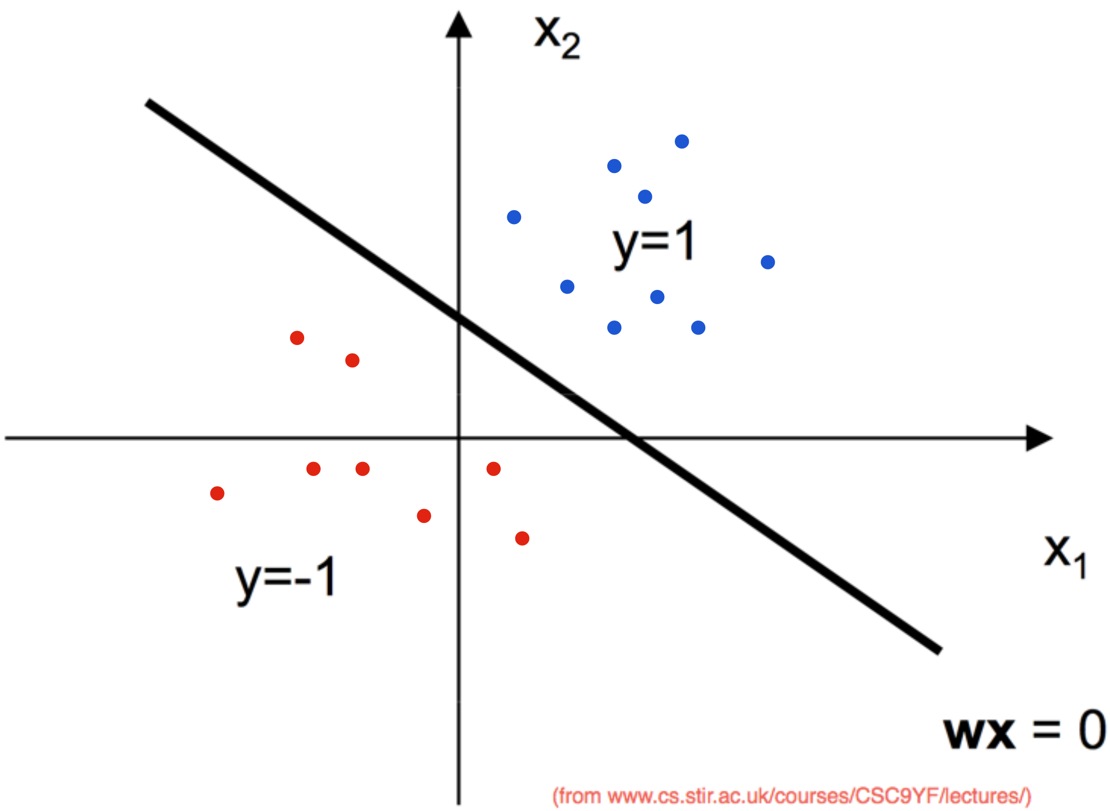
- Rosenblatt's perceptron algorithm
- Use *training examples*

Learning the weights

- Rosenblatt's perceptron algorithm
- Use *training examples*
- Modify weights such that output error is minimized

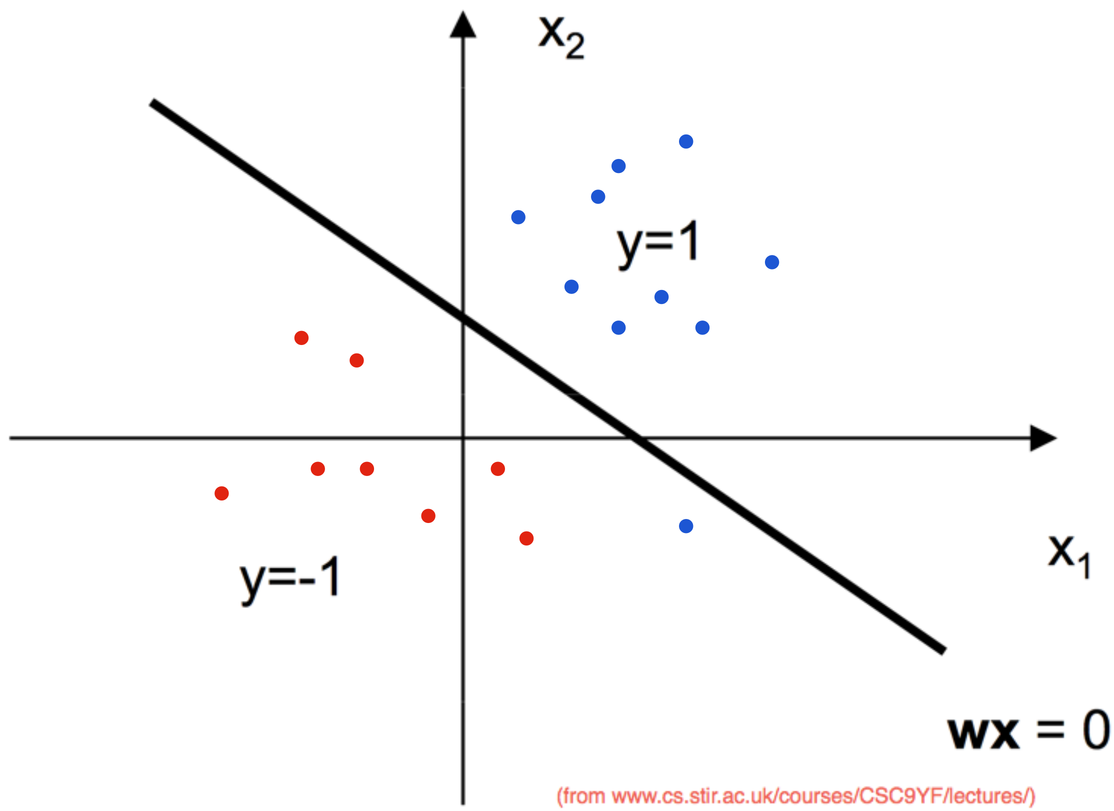
Learning the weights

- Rosenblatt's perceptron algorithm
- Use *training examples*
- Modify weights such that output error is minimized



Learning the weights

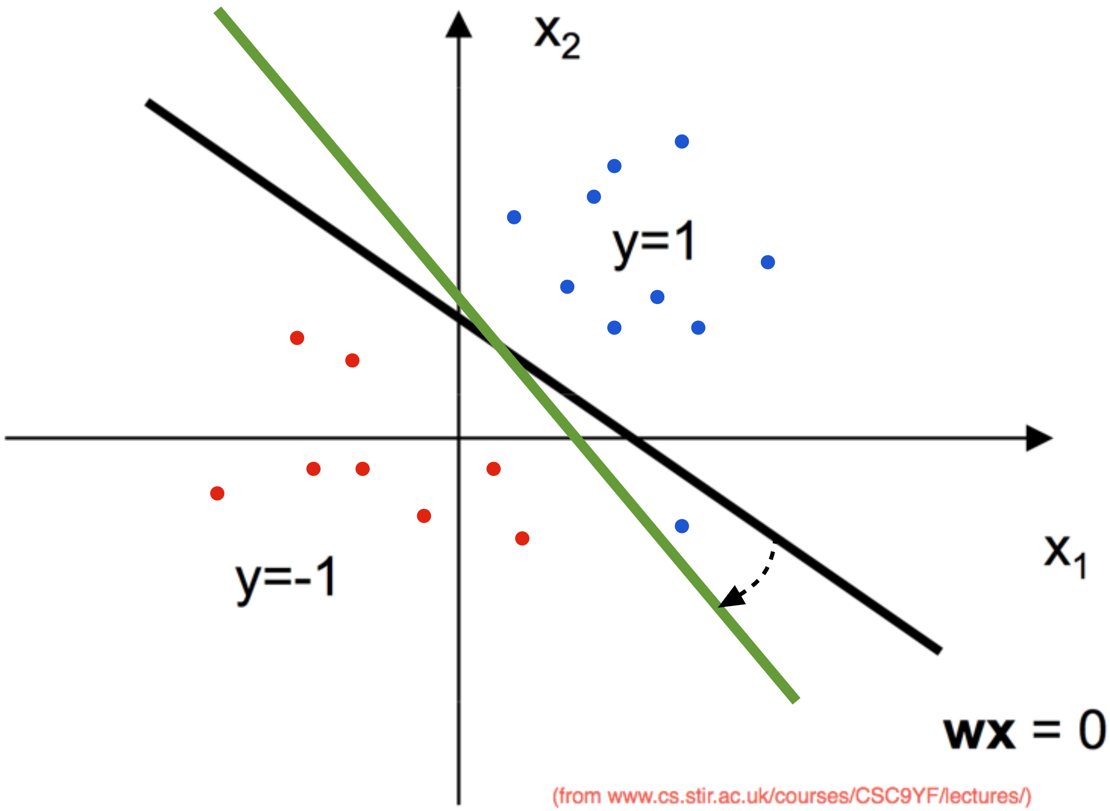
- Rosenblatt's perceptron algorithm
- Use *training examples*
- Modify weights such that output error is minimized



(from www.cs.stir.ac.uk/courses/CSC9YF/lectures/)

Learning the weights

- Rosenblatt's perceptron algorithm
- Use *training examples*
- Modify weights such that output error is minimized



(from www.cs.stir.ac.uk/courses/CSC9YF/lectures/)

Learning the weights

- Let:
  - y_k = desired output for example k
  - a_k = actual output for example k
- Error on example $k = y_k - a_k$
- Define an error function E_k for example k

$$E = \sum_k E_k = \frac{1}{2} \sum_k (y_k - a_k)^2$$

- Why?
  - Squaring make error always positive (parabola)
  - The 1/2 “makes the math easier” (as we’ll see)

Learning the weights

- Goal: minimize E by minimizing each E_k

Learning the weights

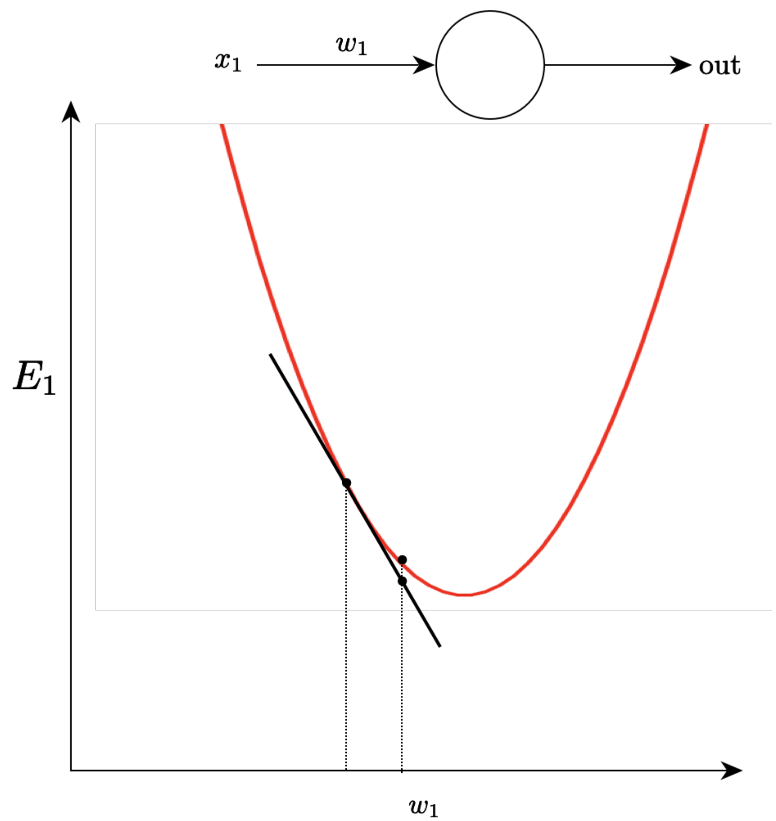
- Goal: minimize E by minimizing each E_k
- E_k is a function of the weights

Learning the weights

- Goal: minimize E by minimizing each E_k
- E_k is a function of the weights
- Use *gradient descent* instead

Learning the weights

- Goal: minimize E by minimizing each E_k
- E_k is a function of the weights
- Use *gradient descent* instead
- With one weight:

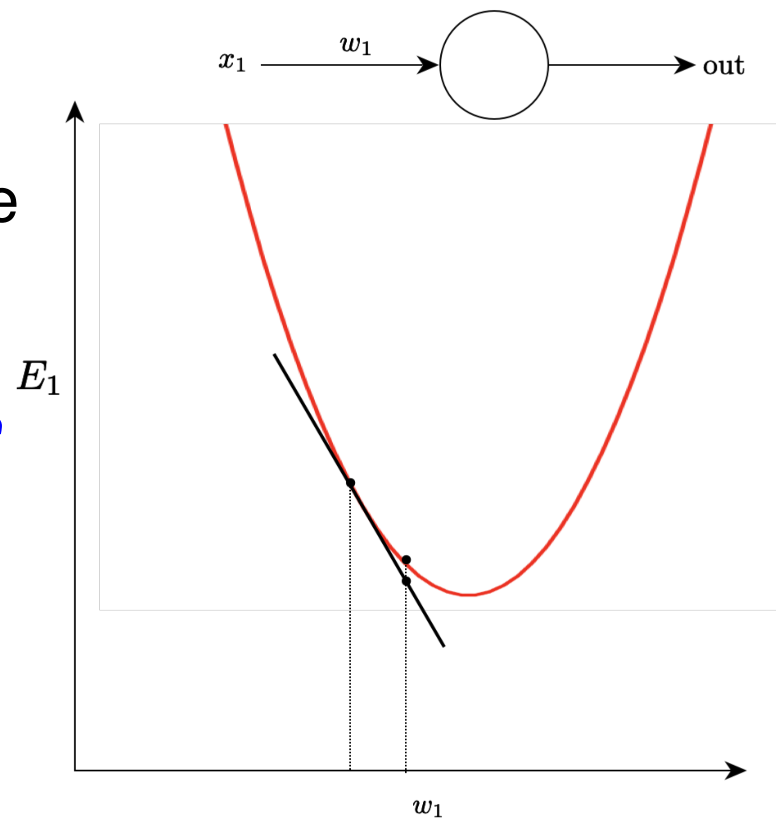


Learning the weights

- Goal: minimize E by minimizing each E_k
- E_k is a function of the weights
- Use *gradient descent* instead
- With one weight:
- Slope at point: $\dfrac{dE_k}{dx_i}$ tells which direction to move

$$w_1' = w_1 - \alpha \frac{dE_k}{dx_i}$$

where α is the *learning rate*

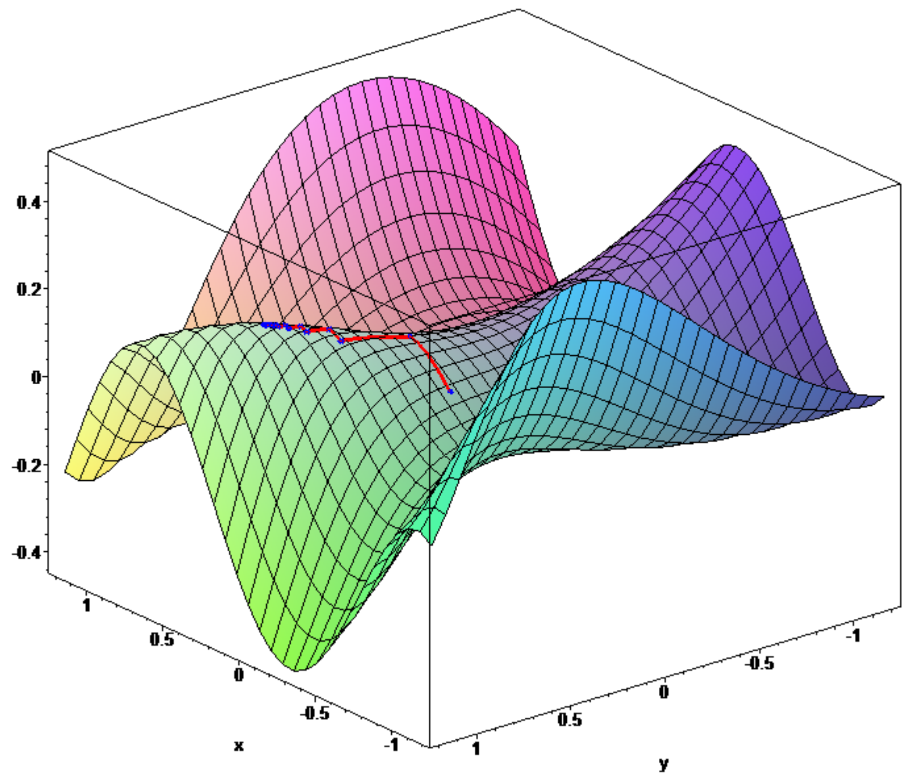


Learning the weights

- Suppose there are 2 weights, x and y

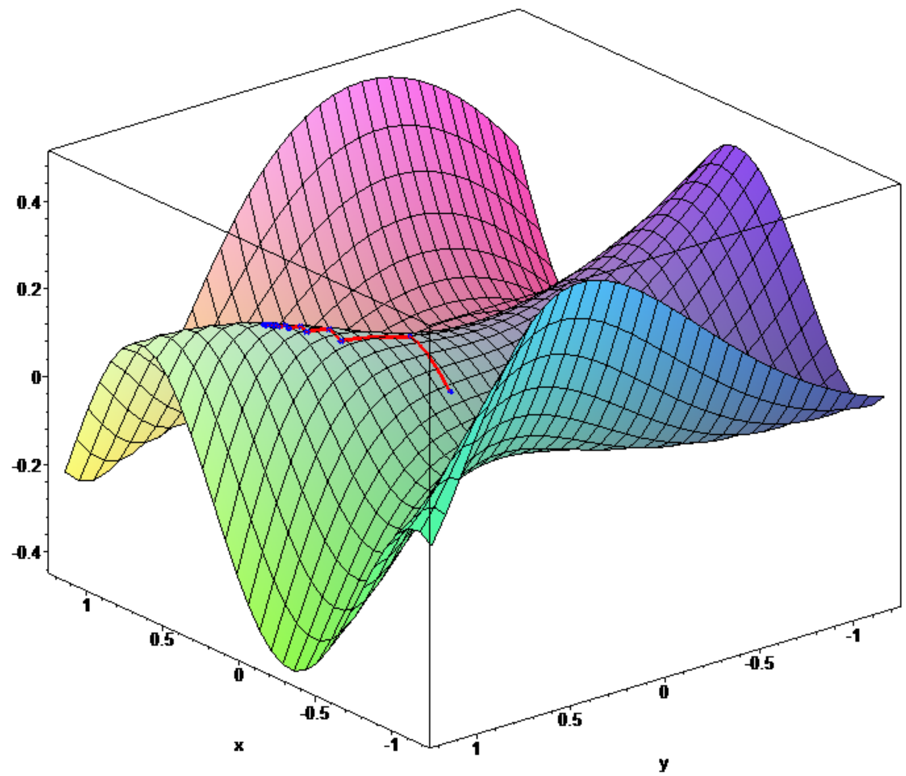
Learning the weights

- Suppose there are 2 weights, x and y



Learning the weights

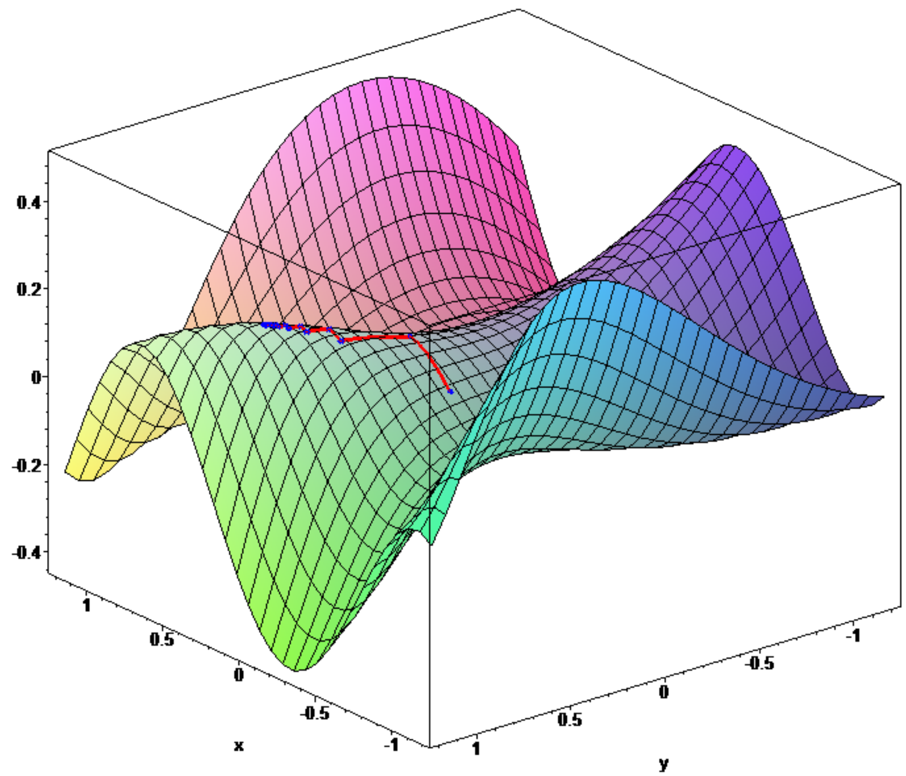
- Suppose there are 2 weights, x and y



- Now "slope" is really the *gradient* ∇ at (x, y)

Learning the weights

- Suppose there are 2 weights, x and y



- Now "slope" is really the *gradient* ∇ at (x, y)

$$\nabla(w_i) = \frac{\partial E_k}{\partial w_i} \text{ and } w_{i,t+1} = w_{i,t} - \alpha \frac{\partial E_k}{\partial w_{i,t}}$$

- Gradient descent: hill-climbing in multiple dimensions

Learning the weights

- What is $\dfrac{\partial E_k}{\partial w_i}$?

Learning the weights

- What is $\dfrac{\partial E_k}{\partial w_i}$?
- We know that the output for k^{th} example $a_k = \sum_i w_i x_i$

Learning the weights

- What is $\frac{\partial E_k}{\partial w_i}$?
- We know that the output for k^{th} example $a_k = \sum_i w_i x_i$
- *Chain rule*:

$$
\begin{aligned}
\frac{\partial E_k}{\partial w_i} &= \frac{\partial E_k}{\partial a_k} \frac{\partial a_k}{\partial w_i} \\
&= \frac{\partial \frac{1}{2}(y_k - a_k)^2}{\partial a_k} \frac{\partial (w_1 x_1 + w_2 x_2 + \cdots w_n x_n)}{\partial w_i} \\
&= -(y_k - a_k) x_i
\end{aligned}
$$

Learning the weights

- What is $\dfrac{\partial E_k}{\partial w_i}$?
- We know that the output for k^{th} example $a_k = \sum_i w_i x_i$
- *Chain rule*:

$$\begin{aligned}
\frac{\partial E_k}{\partial w_i} &= \frac{\partial E_k}{\partial a_k}\frac{\partial a_k}{\partial w_i} \\
&= \frac{\partial \frac{1}{2}(y_k - a_k)^2}{\partial a_k}\frac{\partial (w_1 x_1 + w_2 x_2 + \cdots w_n x_n)}{\partial w_i} \\
&= -(y_k - a_k)x_i
\end{aligned}$$

- Since $\Delta w_i = \alpha \dfrac{\partial E_k}{\partial w_i}$, then

$$w_{i,t+1} = w_{i,t} - \alpha(-(y_k - a_k)x_i) = w_{i,t} + \alpha(y_k - a_k)x_i$$

Learning the weights

- So for each example $< (x_1, x_2, \cdots, x_n), y >$
  - Compute output a
  - Adjust weights:

$$w_{i,t+1} = w_{i,t} + \alpha(y - a)x_i$$

for all weights weights w_i

Implementations

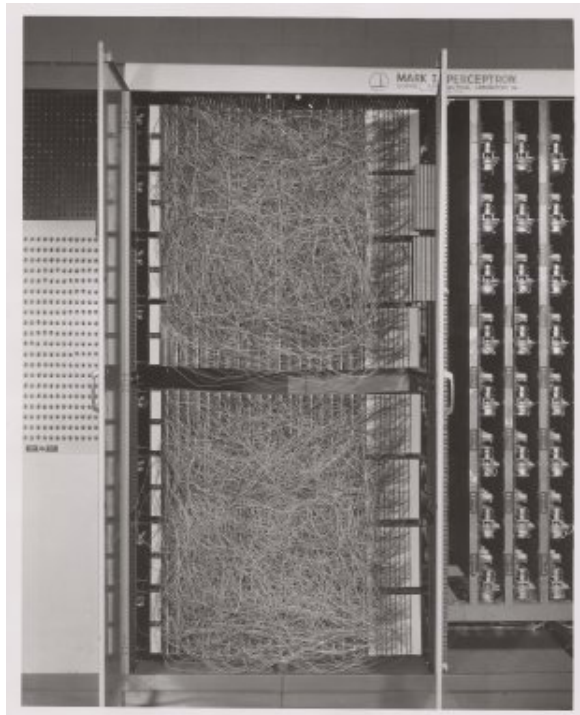
- Algorithm first in IBM 704 in late 50s

Implementations

- Algorithm first in IBM 704 in late 50s
- Then:

Implementations

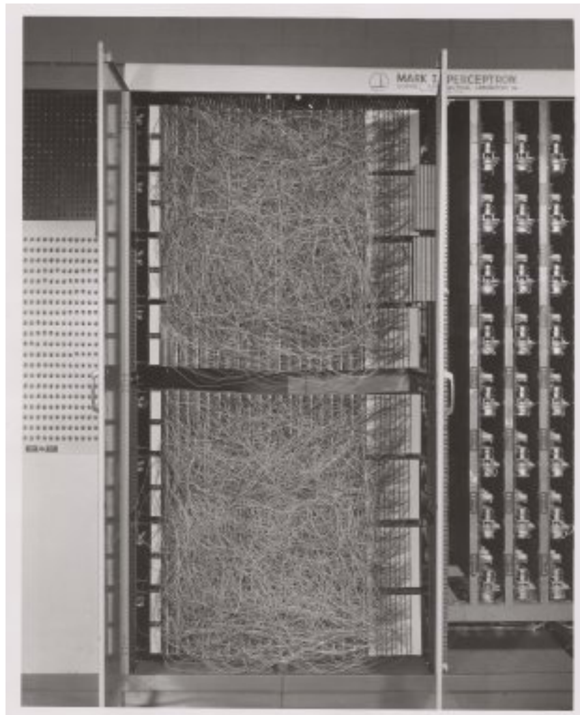
- Algorithm first in IBM 704 in late 50s
- Then:



  - Mark I Perceptron Machine (Wikipedia)

Implementations

- Algorithm first in IBM 704 in late 50s
- Then:



  - Mark I Perceptron Machine (Wikipedia)
  - Image recognition: 20×20 photocell array
  - Potentiometers: weights
  - Pots adjusted by motors from learning

Example

```
(defclass perceptron ()
  ((num-inputs :initarg :num-inputs :initform 3
               :accessor num-inputs)
   (inputs :initarg :inputs :initform nil :accessor inputs)
   (weights :initarg :weights :initform nil
            :accessor weights)
   (bias :initarg :bias :initform 0 :accessor bias)
   (output :initarg :output :initform nil :accessor output)
   (target :initarg :target :initform nil :accessor target)
   (alpha :initarg :alpha :initform 1.0 :accessor alpha)
   )
  )

(defmethod initialize-instance :after ((self perceptron)
                                       &rest l)
  (declare (ignore l))
  (with-slots (num-inputs weights) self
    (setq weights
      (loop for i from 1 to num-inputs
          collect (random 1.0)))))
```

Example

```lisp
(defmethod compute-output ((self perceptron))
  (with-slots (output bias inputs weights) self
    (setq output (if (> (+ bias
                           (apply #'+
                                  (mapcar #'* inputs
                                          weights)))
                        0.0)
                     1
                     0))))

(defmethod adjust-weights ((self perceptron))
  (with-slots (inputs weights target output alpha) self
    (compute-output self)
    (let ((delta (loop for weight in weights
     for input in inputs
                         collect (* alpha (- target output)
input))))
      (format t
      "~s -> ~s (desired = ~s), weights=~s, delta=~s~%"
              inputs output target weights delta)
      (setq weights (mapcar #'+ weights delta))
      (format t "     new weights=~s~%" weights))))
```

Example

```
(defmethod train ((self perceptron) examples)
  (with-slots (inputs target output weights) self
    (loop for count from 1 to (length examples)
          for example in examples
          do (setf inputs (car example)
                   target (cadr example))
             (compute-output self)
             (adjust-weights self)
             (compute-output self)
             )))
```

Example

```
(defvar *perceptron* nil)

(defun train-for-tt (&key new? examples (bias -6) (inputs 3))
  (when new?
    (setq *perceptron* (make-instance 'perceptron
 :bias bias :num-inputs inputs)))
  (train *perceptron* examples)
  ;; now check it:
  (loop for thing in examples
      do (setf (inputs *perceptron*) (car thing))
 (compute-output *perceptron*)
 (format t "~s => ~s~%" (car thing)
 (output *perceptron*))))
```

Example

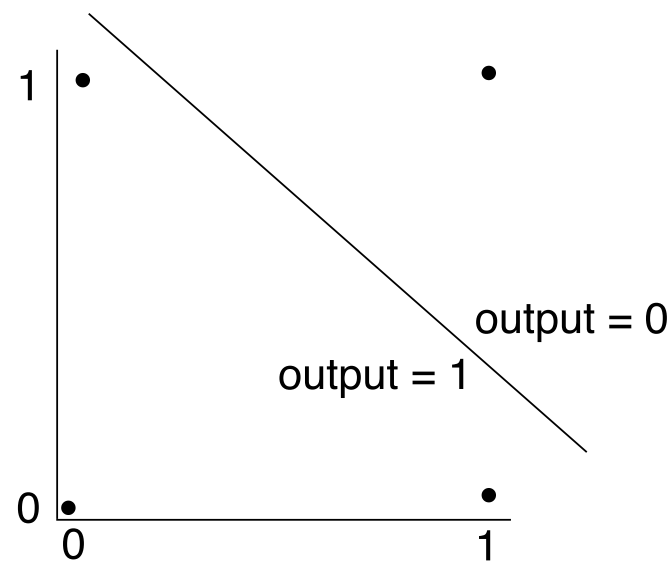
```
(defvar *and-tt* '(((0 0) 0)
    ((0 1) 0)
    ((1 0) 0)
    ((1 1) 1)))

(defvar *or-tt* '(((0 0) 0)
   ((0 1) 1)
   ((1 0) 1)
   ((1 1) 1)))

(defvar *xor-tt* '(((0 0) 0)
    ((0 1) 1)
    ((1 0) 1)
    ((1 1) 0)))
```

What can it do?

- *Linear classifier*:
  - Finds line/plane/hyperplane separating class 1 from class 2
    - 2 inputs $\Rightarrow$ line between sets
    - 3 inputs $\Rightarrow$ plane, etc.
  - Sets can be separated by hyperplane $\Rightarrow$ *linearly-separable*
- Training set linearly-separable, algorithm converges
- Example: can learn NAND function

What can it do?

- *Linear classifier*:
  - Finds line/plane/hyperplane separating class 1 from class 2
    - 2 inputs $\Rightarrow$ line between sets
    - 3 inputs $\Rightarrow$ plane, etc.
  - Sets can be separated by hyperplane $\Rightarrow$ *linearly-separable*
- Training set linearly-separable, algorithm converges
- Example: can learn NAND function



Problems

- May not be a unique solution

Problems

- May not be a unique solution
  - Thus may have suboptimal learning
  - *Support vector machine* (SM): "perceptron of optimal stability"

Problems

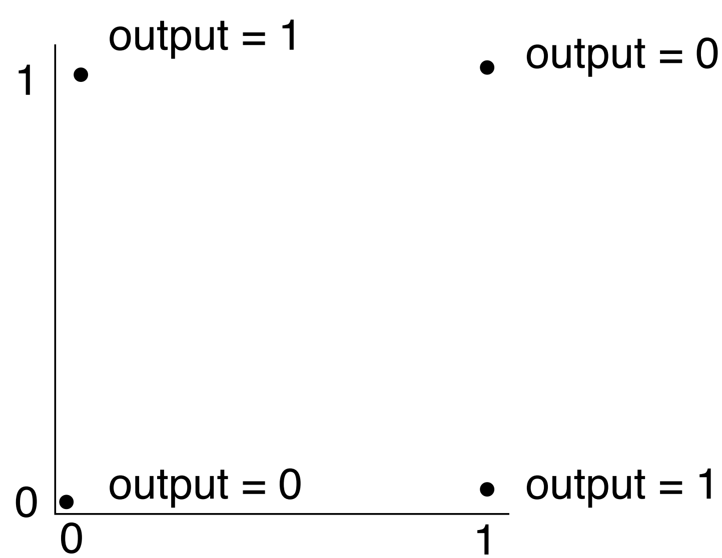
- May not be a unique solution
  - Thus may have suboptimal learning
  - *Support vector machine* (SM): “perceptron of optimal stability”
- Worse problem: can’t learn even simple non-linearly-separable function

Problems

- May not be a unique solution
  - Thus may have suboptimal learning
  - *Support vector machine* (SM): “perceptron of optimal stability”
- Worse problem: can’t learn even simple non-linearly-separable function
  - Minsky & Papert (1960): *Perceptrons* book
  - Perceptron can’t learn XOR function

Problems

- May not be a unique solution
  - Thus may have suboptimal learning
  - *Support vector machine* (SM): "perceptron of optimal stability"
- Worse problem: can't learn even simple non-linearly-separable function
  - Minsky & Papert (1960): *Perceptrons* book
  - Perceptron can't learn XOR function



Problems

- May not be a unique solution
  - Thus may have suboptimal learning
  - *Support vector machine* (SM): "perceptron of optimal stability"
- Worse problem: can't learn even simple non-linearly-separable function
  - Minsky & Papert (1960): *Perceptrons* book
  - Perceptron can't learn XOR function
  - Killed perceptron research for a while

Extending perceptrons

Perceptron networks

- Single perceptron: very limited

Perceptron networks

- Single perceptron: very limited
- Idea: hook a bunch together in a network

Perceptron networks

- Single perceptron: very limited
- Idea: hook a bunch together in a network
- What can a perceptron network do?

Perceptron networks

- Single perceptron: very limited
- Idea: hook a bunch together in a network
- What can a perceptron network do?
  - Based on what you know, what do *you* think?