Linear Regression and Perceptrons

CSE 253

Deep Learning

Overview

- In this lecture, if time, we will cover:
- Linear regression
- Gradient descent for linear regression
- The Perceptron

Last time

- We talked about the four kinds of learning studied in machine learning:
- Unsupervised (learn a model of the data)
- Supervised (learn a *mapping* from input to a target)
- Reinforcement (learn from your mistakes)
- Imitation learning (not usually covered in textbooks…)

Last time

- Supervised learning (learn a *mapping* from input to a target) comes in two forms:
  - Classification (learn a mapping to a category label)
  - Regression (learn a real-valued function)
- This is the textbook story, but in fact, it is hard to categorize all supervised learning into just these two – for example, transforming English sentences to French ones.

Overview

- In this lecture, if time, we will cover:
- **Linear regression**
- Gradient descent for linear regression
- The Perceptron

Linear regression

- An example of supervised learning
- There is a set of inputs and associated outputs
- We assume there is an underlying line that has been corrupted by noise.
- (go to board)
- And it is a neural network!

Linear regression

- Given:

  A set of inputs: (the *design matrix*)

$$\begin{pmatrix} x_1^1, x_2^1, \cdots, x_d^1 \\ x_1^2, x_2^2, \cdots, x_d^2 \\ \cdots \\ x_1^N, x_2^N, \cdots, x_d^N \end{pmatrix}$$

  and a set of *targets*:

$$\begin{pmatrix} t^1 \\ t^2 \\ \cdots \\ t^N \end{pmatrix}$$

  Find a set of coefficients (weights) that linearly combine the inputs to get as close to the targets as possible.

Linear regression

- First, to make things a bit simpler, we tack on a column of ones:

$$\begin{pmatrix} 1, x_1^1, x_2^1, \cdots, x_d^1 \\ 1, x_1^2, x_2^2, \cdots, x_d^2 \\ \cdots \\ 1, x_1^N, x_2^N, \cdots, x_d^N \end{pmatrix} \qquad \begin{pmatrix} t^1 \\ t^2 \\ \cdots \\ t^N \end{pmatrix}$$

Linear regression

- The goal:

$$\begin{pmatrix} 1, x_1^1, x_2^1, \cdots, x_d^1 \\ 1, x_1^2, x_2^2, \cdots, x_d^2 \\ \cdots \\ 1, x_1^N, x_2^N, \cdots, x_d^N \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ \cdots \\ w_d \end{pmatrix} \approx \begin{pmatrix} t^1 \\ t^2 \\ \cdots \\ t^N \end{pmatrix}$$

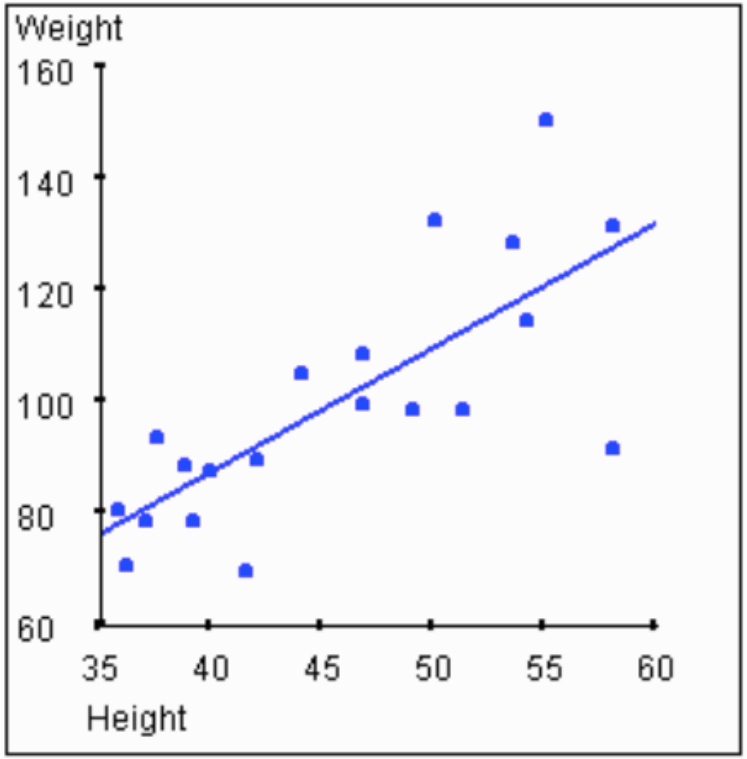
- Where "$\approx$" means "as close to as possible"

Linear regression

- The goal:
- "Close" means the w that minimizes the Sum Squared Error (SSE)

$$SSE = \frac{1}{2}\sum_{n=1}^{N}\left(\sum_{j=0}^{d} w_j x_j^n - t^n\right)^2$$

Linear regression example: Height vs. Weight



$$X = \begin{pmatrix} 1, 36 \\ 1, 42 \\ \dots \\ 1, 56 \end{pmatrix} \quad t = \begin{pmatrix} 70 \\ 92 \\ \dots \\ 149 \end{pmatrix}$$

Linear regression example: Height vs. Weight

$$X = \begin{pmatrix} 1,36 \\ 1,42 \\ \dots \\ 1,56 \end{pmatrix} \quad t = \begin{pmatrix} 70 \\ 92 \\ \dots \\ 149 \end{pmatrix} \quad \begin{aligned} w_0 + w_1 36 &= 70 \\ w_0 + w_1 42 &= 92 \\ &\dots \\ w_0 + w_1 56 &= 149 \end{aligned}$$

Solving this corresponds to finding w_0 and w_1 – we have 20 equations in two unknowns.

Linear regression example: Height vs. Weight

$$w_0 + w_1 36 = 70$$

$$w_0 + w_1 42 = 92$$

$$\dots$$

$$w_0 + w_1 56 = 149$$

$$SSE = \frac{1}{2}\sum_{n=1}^{20}\left(w_0 + w_1 x_1^n - t^n\right)^2$$

Taking the derivative of the *SSE* with respect to w and setting it equal to 0:

(removed because it's too complicated!)

Better to use *matrix form*!

- Assuming the design matrix is:

$$X = \begin{pmatrix} 1,36 \\ 1,42 \\ \dots \\ 1,56 \end{pmatrix}$$

- And the target vector is:

$$t = \begin{pmatrix} 70 \\ 92 \\ \dots \\ 149 \end{pmatrix}$$

- And the weight vector is:

$$\vec{w} = \begin{pmatrix} w_0 \\ w_1 \end{pmatrix}$$

- Then we can write the SSE as:

$$SSE = \frac{1}{2} \| X\vec{w} - \vec{t} \|^2$$

- In fact, this is completely general.
- Now, since this is a quadratic, it has a single minimum, which we can find by taking the derivative and setting it to 0:

$$\frac{\partial SSE}{\partial \vec{w}} = \frac{\partial \frac{1}{2} \| X\vec{w} - \vec{t} \|^2}{\partial \vec{w}} = 0$$

Moving right along...

$$\frac{\partial SSE}{\partial \vec{w}} = \frac{\partial \frac{1}{2} \| X\vec{w} - \vec{t} \|^2}{\partial \vec{w}} = 0$$

$$\frac{\partial \frac{1}{2} \| X\vec{w} - \vec{t} \|^2}{\partial \vec{w}} = (X\vec{w} - \vec{t}) \frac{\partial (X\vec{w} - \vec{t})}{\partial \vec{w}} = 0$$

$$(X^T X)\vec{w} - X^T \vec{t} = 0$$

$$(X^T X)\vec{w} = X^T \vec{t}$$

$$\vec{w} = (X^T X)^{-1} X^T \vec{t}$$

$(X^T X)^{-1} X^T = X^\dagger$ is called the *pseudoinverse* of X

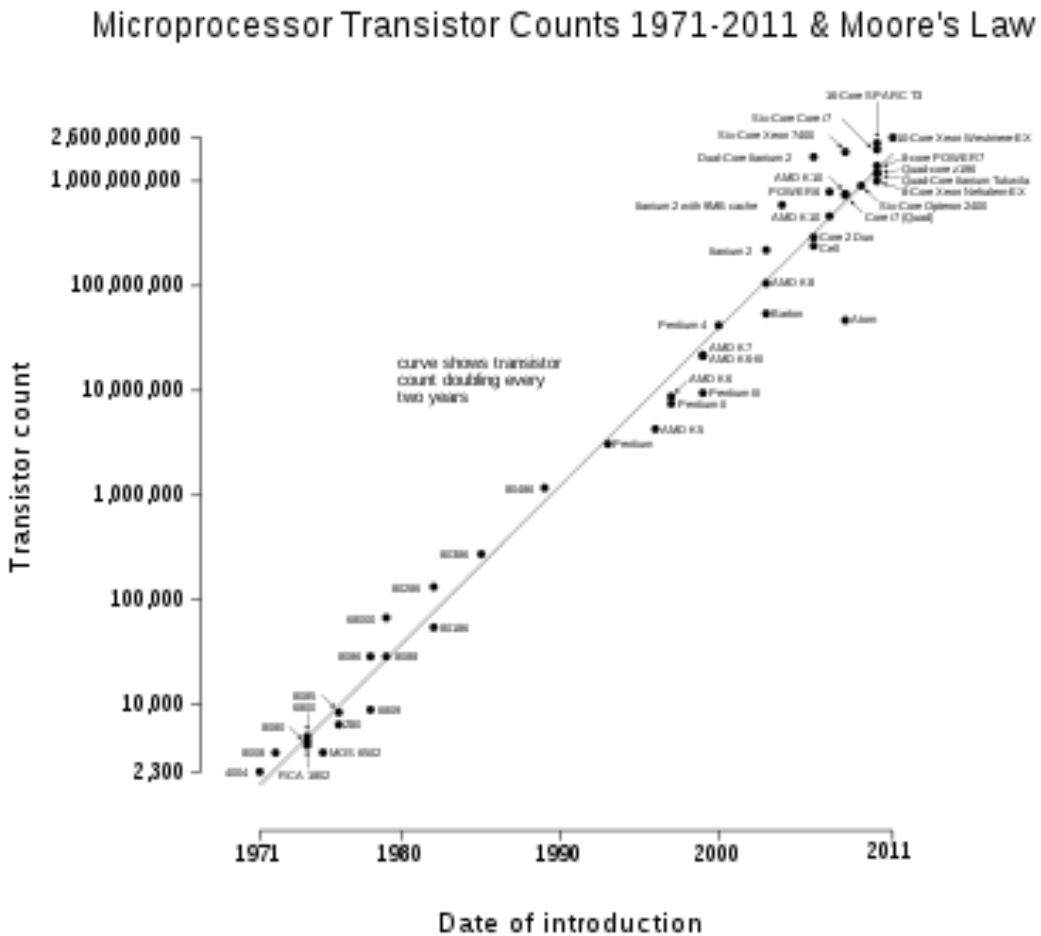
So, $\vec{w} = X^\dagger \vec{t}$ ... a closed form formula.

This won't happen again! ;-)

Note that: $(X^T X)^{-1} X^T X = I$

Note: Not all regression is Linear!
Nonlinear regression

(by transforming a variable – here, using the log, we can make some problems linear again)



Note: Not all regression is Linear! Nonlinear regression

- Or by expanding the inputs by assuming some underlying functional form (e.g., a polynomial)
- We try to find coefficients of the polynomial that fit the data (a.k.a. *polynomial curve fitting*)
- Or, by using a neural network to fit the data…

Linear Regression Summary

- In regression problems, we are trying to fit a continuous function
- Linear regression is fitting a line
- It can be solved by inverting a matrix
- Some problems might be better solved by gradient descent in the SSE – when the design matrix is too large
- Not all regression is linear!

Overview

- In this lecture, if time, we will cover:
- Linear regression
- **Gradient descent for linear regression (on board…)**
- The Perceptron

Gradient descent for linear regression

In case you missed it on the board!

Recall our model is: $y(x) = g(a)$, where g is the activation function.

For linear regression, g is the identity function: $g(a) = a$.

$a^n = w^{\mathrm{T}} x^n = \sum_{i=0}^{d} w_i x_i^n$ is the weighted sum of the inputs for pattern n.

Now, we need an Objective Function, something to minimize.

We will use Mean Squared Error, which tries to minimize

the distance between the targets and the outputs: $MSE = \frac{1}{N} \sum_{n=1}^{N} (t^n - y^n)^2$

Gradient Descent

- Recall the goal of gradient descent is to reduce the Objective function by adjusting the parameters to go downhill in the objective function (A.K.A., Loss, or Error)
- To do this, we move the parameters in the direction of the negative slope:

$$w_i = w_i - \alpha \frac{\partial MSE}{\partial w_i}$$

Gradient Descent

- So, we need to figure out this expression:

$$\frac{\partial MSE}{\partial w_i} = \frac{\partial \frac{1}{2N}\sum_{n=1}^{N}(t^n - y^n)^2}{\partial w_i} = \frac{1}{2N}\sum_{n=1}^{N}\frac{\partial (t^n - y^n)^2}{\partial w_i}$$

(the derivative of the sum is the sum of the derivatives)

Gradient Descent

- Moving right along…

$$\frac{1}{N}\sum_{n=1}^{N}\frac{\frac{1}{2}\partial(t^n - y^n)^2}{\partial w_i} = \frac{1}{N}\sum_{n=1}^{N}(t^n - y^n)\left(\frac{\partial t^n}{\partial w_i} - \frac{\partial y^n}{\partial w_i}\right)$$

$$= \frac{1}{N}\sum_{n=1}^{N}(t^n - y^n)\left(-\frac{\partial y^n}{\partial w_i}\right) = \frac{1}{N}\sum_{n=1}^{N}(t^n - y^n)\left(-\frac{\partial a^n}{\partial w_i}\right)$$

$$= \frac{1}{N}\sum_{n=1}^{N}(t^n - y^n)\left(-\frac{\partial\sum_{j=0}^{d} w_j x_j^n}{\partial w_i}\right) = -\frac{1}{N}\sum_{n=1}^{N}(t^n - y^n)x_i^n$$

Gradient Descent

Plugging this result into the gradient descent schema, we get the delta rule (batch version)

$$w_i = w_i - \alpha \frac{\partial MSE}{\partial w_i}$$

$$= w_i - -\alpha \frac{1}{N} \sum_{n=1}^{N} (t^n - y^n) x_i^n$$

$$= w_i + \alpha \frac{1}{N} \sum_{n=1}^{N} (t^n - y^n) x_i^n = w_i + \alpha \frac{1}{N} \sum_{n=1}^{N} \delta^n x_i^n$$

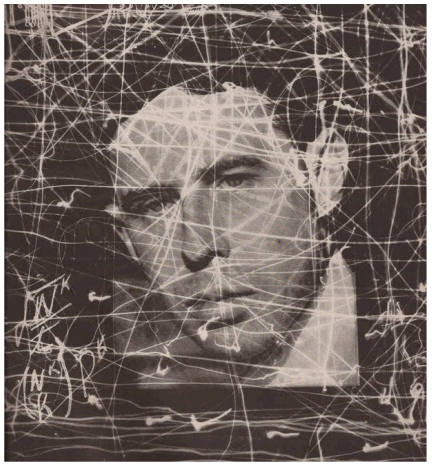
Overview

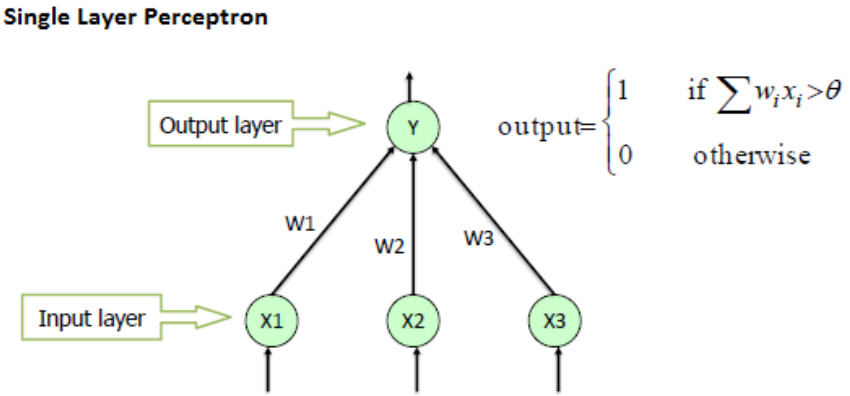
- In this lecture, if time, we will cover:
- Linear regression
- Gradient descent for linear regression
- **The Perceptron**

Perceptrons

- The first trainable, linear classifier
- We start with a bit of history
- Consider what they can represent
- Give an intuitive version of the learning rule
- Give the computer science version of the learning rule

Perceptrons: A bit of history

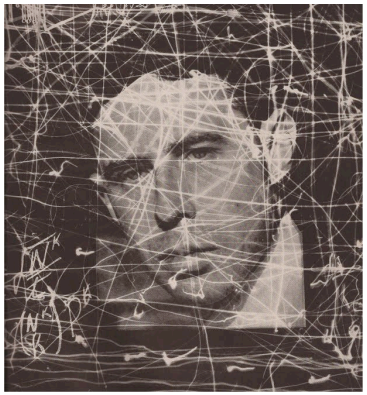


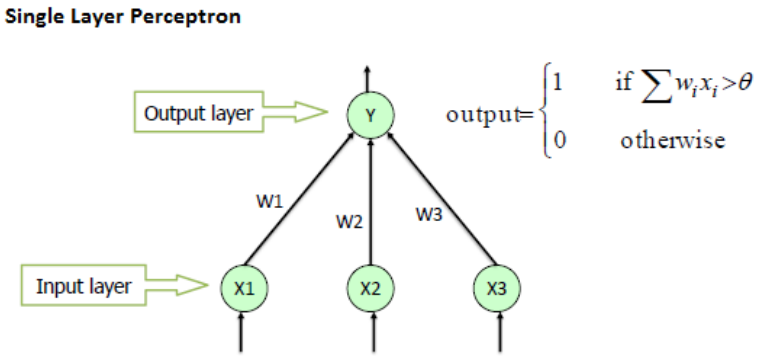


Frank Rosenblatt studied a simple version of a neural net called a ***perceptron***:

- A single layer of processing
- Binary output
- Can compute simple things like (some) boolean functions (OR, AND, etc.)

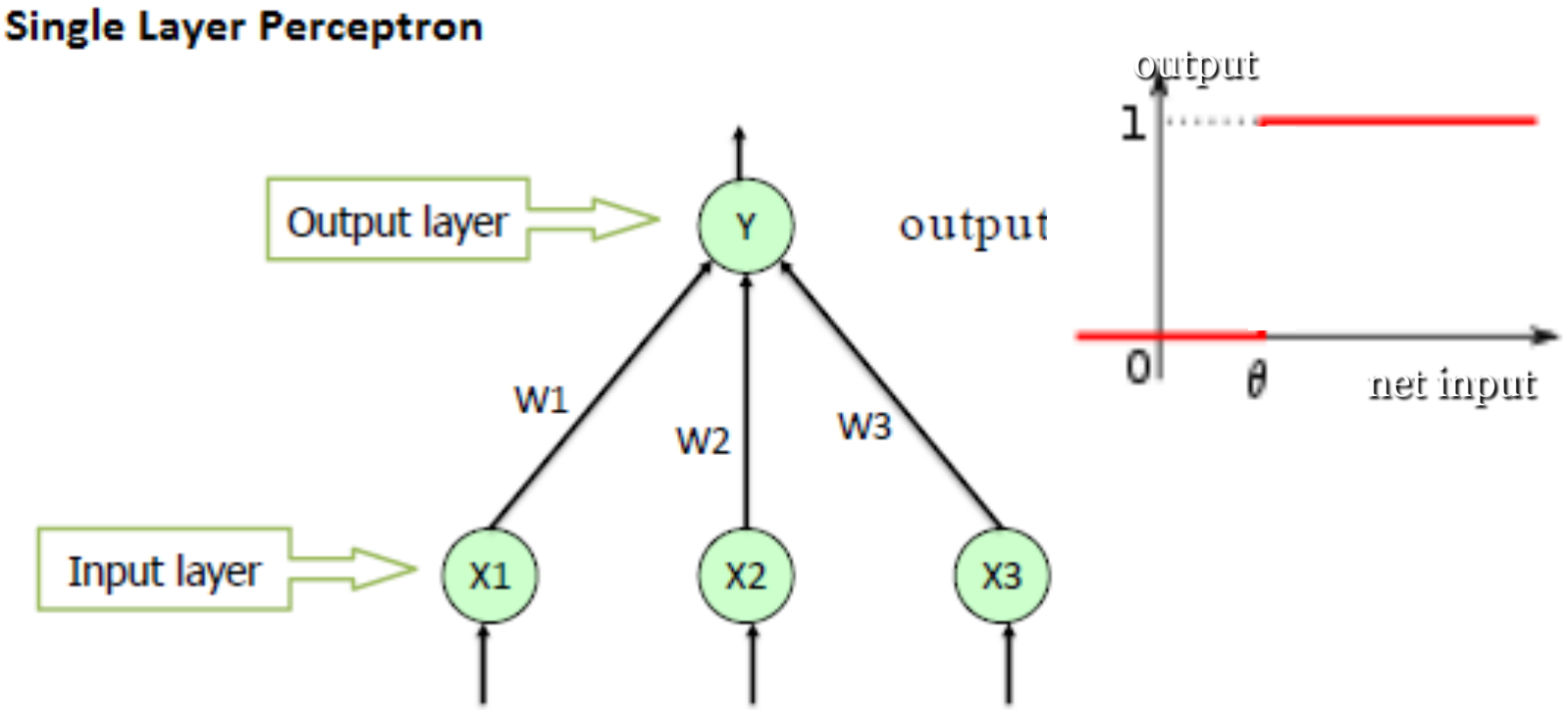
Perceptrons: A bit of history





- Computes the weighted sum of its inputs (the *net input*, compares it to a threshold, and "fires" if the *net* is greater than or equal than the threshold.
- (Note in the picture it's ">", but we will use ">=" to make sure you're awake!)

The Perceptron Activation Rule



This is called a *binary threshold unit*

Clicker Question
FREQUENCY: AA

The input to a model neuron that we have discussed can be written as (where w and x are the weight vector and input vector, respectively)

a. xw
b. $x+ w$
c. $x^T w$
d. $w^T x$
e. Either c or d

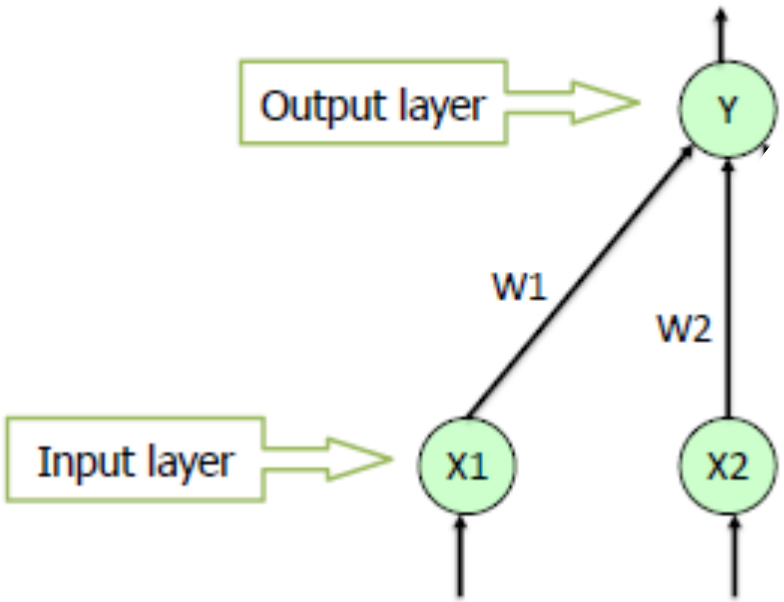
Clicker Question

The input to a model neuron that we have discussed can be written as (where w and x are the weight vector and input vector, respectively)

a. xw
b. $x+ w$
c. $x^T w$
d. $w^T x$
e. Either c or d

Quiz

Single Layer Perceptron



$$\text{output} = \begin{cases} 1 & \text{if } \sum w_i x_i > \theta \\ 0 & \text{otherwise} \end{cases}$$

X1	X2	X1 OR X2
0	0	0
0	1	1
1	0	1
1	1	1

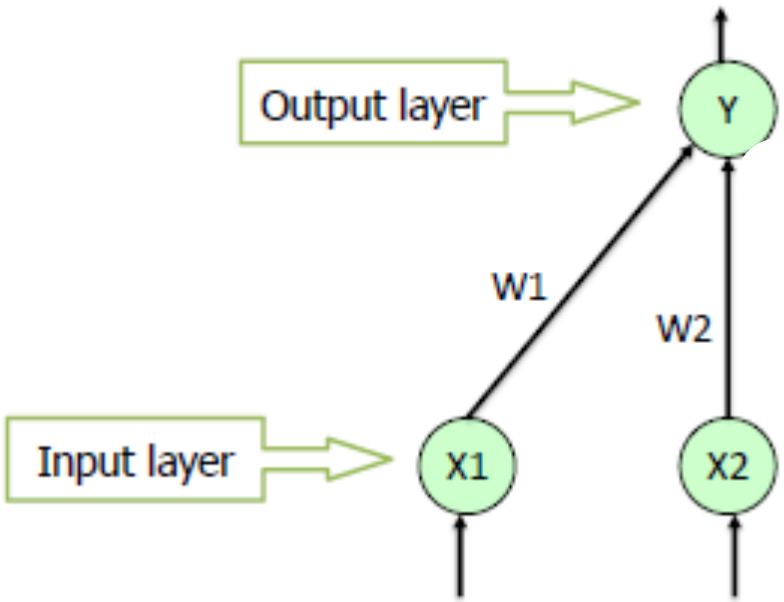
Assume:

FALSE == 0, TRUE==1, so if X1 is false, it is 0.

Can you come up with a set of weights and a threshold so that a two-input perceptron computes OR?

Quiz

Single Layer Perceptron



$$\text{output} = \begin{cases} 1 & \text{if } \sum w_i x_i > \theta \\ 0 & \text{otherwise} \end{cases}$$

X1	X2	X1 AND X2
0	0	0
0	1	0
1	0	0
1	1	1

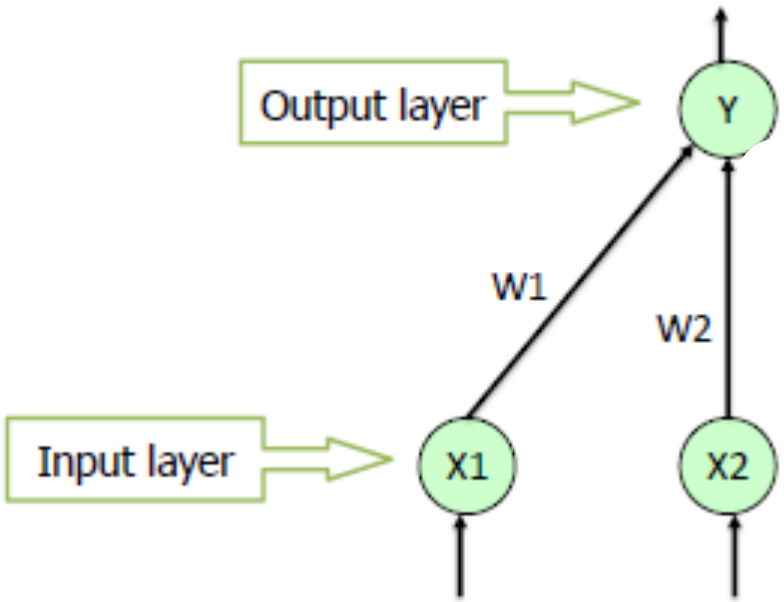
Assume:

FALSE == 0, TRUE==1

Can you come up with a set of weights and a threshold so that a two-input perceptron computes AND?

Quiz

Single Layer Perceptron



$$\text{output} = \begin{cases} 1 & \text{if } \sum w_i x_i > \theta \\ 0 & \text{otherwise} \end{cases}$$

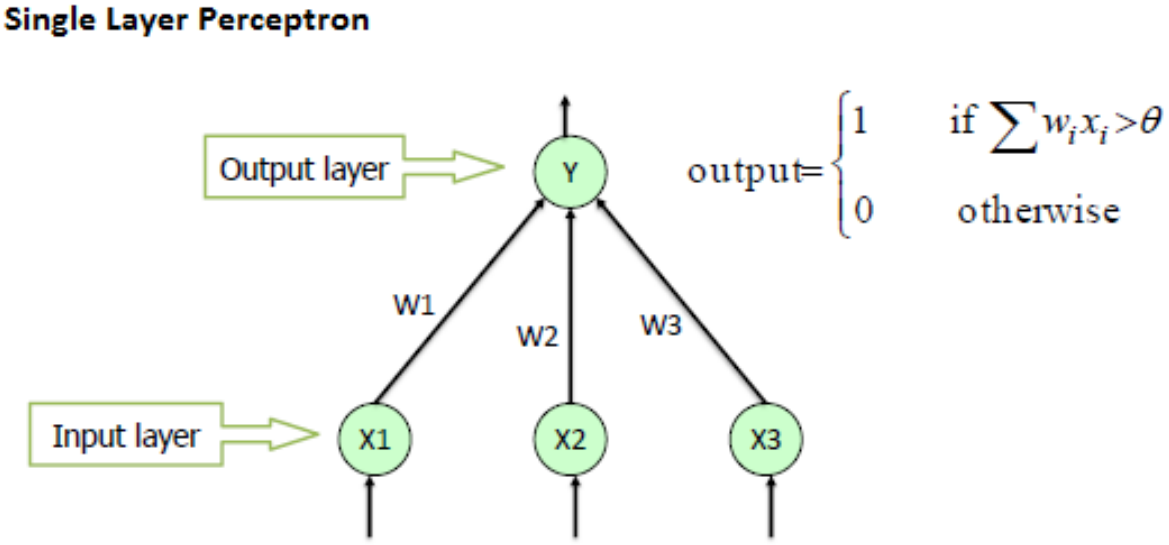
X1	X2	X1 XOR X2
0	0	0
0	1	1
1	0	1
1	1	0

Assume:

FALSE == 0, TRUE==1

Can you come up with a set of weights and a threshold so that a two-input perceptron computes XOR?

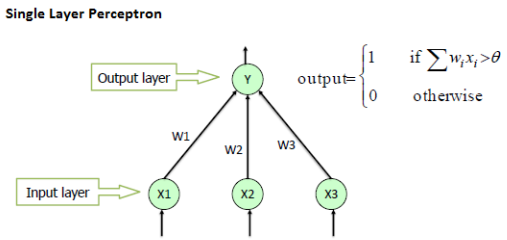
Perceptrons



The goal was to make a neurally-inspired machine that could categorize inputs – and *learn* to do this from examples

Learning: A bit of history

- Rosenblatt (1962) discovered a learning rule for perceptrons called the *perceptron convergence procedure*



- Guaranteed to learn anything computable (by a two-layer perceptron)

- Unfortunately, not everything was computable (Minsky & Papert, 1969)

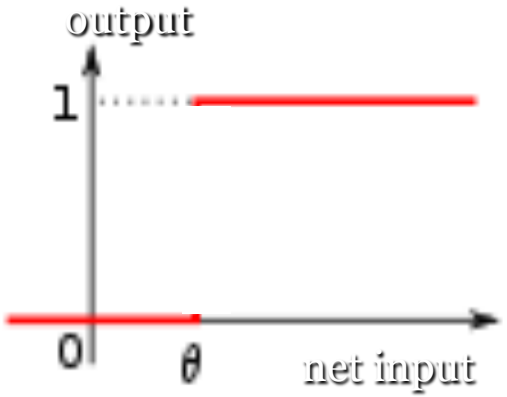
Perceptron Learning

- It is ***supervised learning***:
  - There is a set of input patterns (called the *design matrix*)
  - and a set of desired outputs (the *targets* or *teaching signal*)
- The network is presented with the inputs, and FOOMP, it computes the output, and the output is compared to the target.
- If they don't match, it changes the weights and threshold so it will get closer to producing the target next time.

Perceptron Learning

First, get a training set - let's choose OR
Four "patterns":

INPUT	TARGET
0 0	0
0 1	1
1 0	1
1 1	1



This is the design matrix (you don't need to know that, but it makes you sound smarter at conferences…)

First let's transform the activation rule to a more modern version

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{d} w_i x_i \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

First let's transform the activation rule to a more modern version

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{d} w_i x_i \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{d} w_i x_i - \theta \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

First let's transform the activation rule to a more modern version

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{d} w_i x_i \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{d} w_i x_i - \theta \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{d} w_i x_i + w_0 \geq 0 \\ 0 & \text{otherwise} \end{cases} \qquad w_0 = -\theta$$

First let's transform the activation rule to a more modern version

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{d} w_i x_i \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{d} w_i x_i - \theta \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$y = \begin{cases} 1 & \text{if } \sum_{i=1}^{d} w_i x_i + w_0 \geq 0 \\ 0 & \text{otherwise} \end{cases} \qquad w_0 = -\theta$$

$$y = \begin{cases} 1 & \text{if } \sum_{i=0}^{d} w_i x_i \geq 0 \\ 0 & \text{otherwise} \end{cases} \qquad x_0 \triangleq 1$$

Perceptron Learning Made Simple

- Output activation rule:
  - First, compute the *output of the network*:

$$y = \begin{cases} 1 & \text{if } \sum_{i=0}^{d} w_i x_i \geq 0 \\ 0 & \text{otherwise} \end{cases} \qquad x_0 \triangleq 1$$

- Learning rule:

  If output is 1 and should be 0, then *lower* weights from active inputs

  If output is 0 and should be 1, then *raise* weights from active inputs

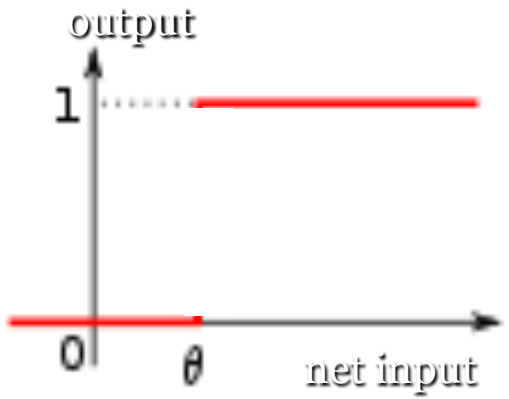
  ("active input" means $x_i = 1$, not 0)

  NOTE: For w_0 the input is *always* active.

Perceptron Learning

– First, get a training set - let's choose OR
– Four "patterns":

INPUT	TARGET
0 0	0
0 1	1
1 0	1
1 1	1



Now, randomly present these to the network, apply the learning rule, and continue until it doesn't make any mistakes.

STOP HERE FOR DEMO
(on board)

Ok, no board. I'll figure something else out…

Characteristics of perceptron learning

- Supervised learning: Gave it a set of input-output examples for it to model the function (a *teaching signal*)
- *Error correction learning*: only corrected it when it is wrong (never praised! ;-))
- Random presentation of patterns.
- Slow! Learning on some patterns ruins learning on others.

Perceptron Learning Made Simple for Computer Science

- Output activation rule:
  - First, compute *y*, the output of the network:

$$y = \begin{cases} 1 & \text{if } \sum_{i=0}^{d} w_i x_i \geq 0 \\ 0 & \text{otherwise} \end{cases} \qquad x_0 \triangleq 1$$

$$y(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x} + w_0 = \sum_i w_i x_i + w_0$$

- Learning rule
- If output is 1 and should be 0, then *lower* weights from active inputs
- If output is 0 and should be 1, then *raise* weights from active inputs
- ("active input" means $x_i = 1$, not 0)

Perceptron Learning Made Simple for Computer Science

- Output activation rule:

$$y = \begin{cases} 1 & \text{if } \sum_{i=0}^{d} w_i x_i \geq 0 \\ 0 & \text{otherwise} \end{cases} \qquad \text{if } x_0 \triangleq 1$$

$$output = \begin{cases} 1 & \text{if } y(x) \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

- Learning rule:

$w_i = w_i + \alpha*(t\text{-}y)*x_i$ (α is the *learning rate*)

Or $w_i = w_i + \alpha*\delta*x_i \qquad (\delta = t\text{-}y)$

This is known as the *delta rule* because learning is based on the *delta* (difference) between what you did and what you should have done: $\delta = (t\text{-}y)$

- *N.B.* In what follows, to make things simple, I will leave out α by assuming it is 1.

Are these the same rule?

- Learning rule:

  If output is 1 and should be 0, then *lower* weights to active inputs and the bias

  If output is 0 and should be 1, then *raise* weights to active inputs and the bias

- Learning rule: (remember I am assuming α=1, so leaving it out)

  $w_i = w_i + (t\text{-}y)*x_i$

  $w_i = w_i + \delta x_i$

Let's convince ourselves these are the same...

- Learning rule:

  If output is 1 and should be 0, then *lower* weights to active inputs and the bias

  $w_i = w_i + (t\text{-}y)x_i$

  $w_i = w_i + (0 - 1)1$ ("x_i is an active input" *means* $x_i = 1$)

  $= w_i - 1$ *lower* weight

  What if x_i is inactive?

  $w_i = w_i + (t\text{-}y)x_i$

  $w_i = w_i + (t\text{-}y)0$

  $= w_i$ (no change)

Let's convince ourselves these are the same...

- Learning rule:

  If output is 0 and should be 1, then *raise* weights to active inputs and the bias

  $w_i = w_i + (t\text{-}y)x_i$

  $w_i = w_i + (1 - 0)1$ ("x_i is an active input" *means* $x_i = 1$)

  $= w_i + 1$ *raise* weight

  What if x_i is inactive?

  $w_i = w_i + (t\text{-}y)x_i$

  $w_i = w_i + (t\text{-}y)0$

  $= w_i$ (no change)

Let's convince ourselves these are the same...

- What about the bias?
- We just treat w_0 as a weight from a unit that is always a constant 1 (i.e., x_0=1)

  If output is 1 and should be 0, then *lower* weights to active inputs *and lower the bias*
- Learning rule:

$$w_i = w_i + (t\text{-}y)x_0$$

I.e.: $w_0 = w_0 + (0 - 1)(1)$

$w_0 = w_0 + (-1)(1)$

$w_0 = w_0 - 1$ *lower* w_0 (other case is obvious)

Let's convince ourselves these are the same...

- What if we get it right??? Then $t=y$, and …
- Learning rule:

$$w_i = w_i + (t\text{-}y)x_i$$

$$w_i = w_i + 0*x_i$$

$$= w_i \qquad \text{(no change)}$$

And... x_i does *not* need to be binary!

In that case, it changes the weight in proportion to the size of the error δ and x_i .

.

The Guarantee

- Anything a perceptron can compute, it can *learn* to compute
- Trained on a function it can learn to compute, the perceptron convergence procedure (i.e., for our purposes, the delta rule) *will always converge.*
- Ok, so what *can* a perceptron compute?

What can a perceptron compute?

- One more step:

$$y = \begin{cases} 1 & \text{if } y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

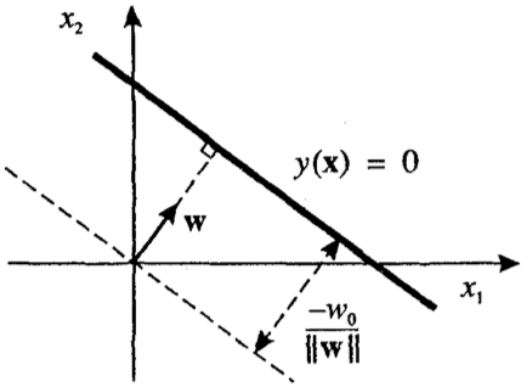
- Here we simply are rewriting the expression as a function.
- Notation: BOLD **w** and BOLD **x** are vectors.

What can a perceptron compute?

- Ok, so, now our activation rule is:

$$y = \begin{cases} 1 & \text{if } y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

- Now, we call $y(\mathbf{x})=0$ the *decision boundary* – where the output changes from 0 to 1.
- This now has a simple geometrical interpretation: $y(\mathbf{x})=0$ is a d-1 dimensional hyperplane in a d-dimensional input space



- So for 2D, it is a line:

Here we are abusing notation:

$$\mathbf{w} = (w_1, w_2, w_3, \ldots, w_d)$$

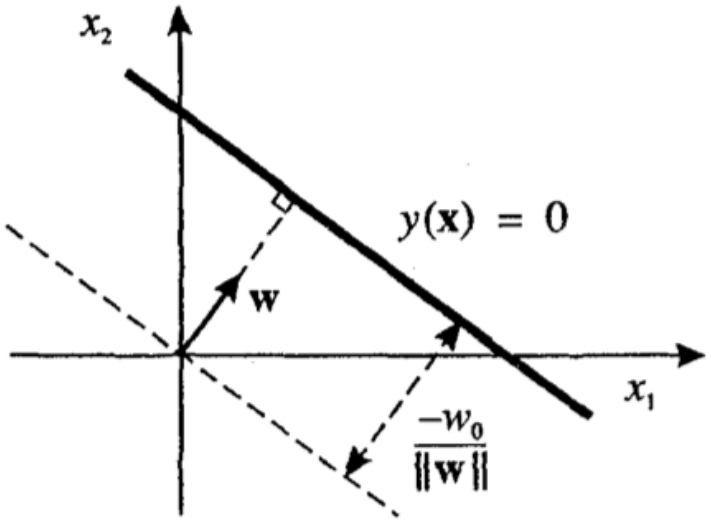
What can a perceptron compute?

- Why is it a line in 2D?

  $y(x) = 0$ ➔

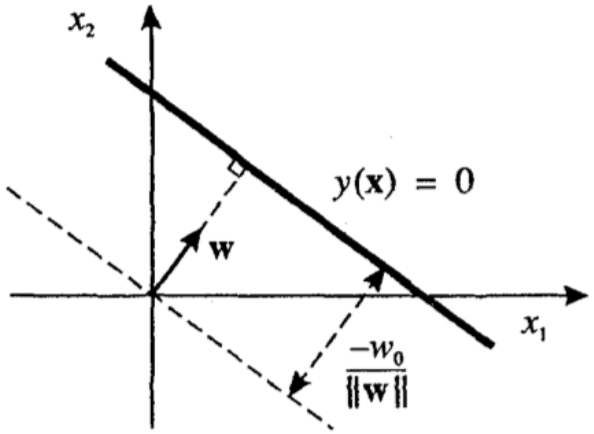
  $w_1x_1 + w_2x_2 + w_0 = 0$ ➔ $x_2 = -(w_1/w_2)x_1 - w_0$

  (slope-intercept form)



What can a perceptron compute?

- So for 2D, the decision boundary is a line
- Why is it perpendicular to **w**?
- Take 2 points on the line $y(\mathbf{x})=0$, call them $\mathbf{x}^A$ and $\mathbf{x}^B$. Then:



$$y(\mathbf{x}^A) = 0 = y(\mathbf{x}^B)$$

$$y(\mathbf{x}^A) - y(\mathbf{x}^B) = 0$$

$$\mathbf{w}^T\mathbf{x}^A + w_0 - \mathbf{w}^T\mathbf{x}^B - w_0 = 0$$

$$\mathbf{w}^T(\mathbf{x}^A - \mathbf{x}^B) = 0 \quad \text{Q.E.D.}$$

And the distance l to the origin is given by:

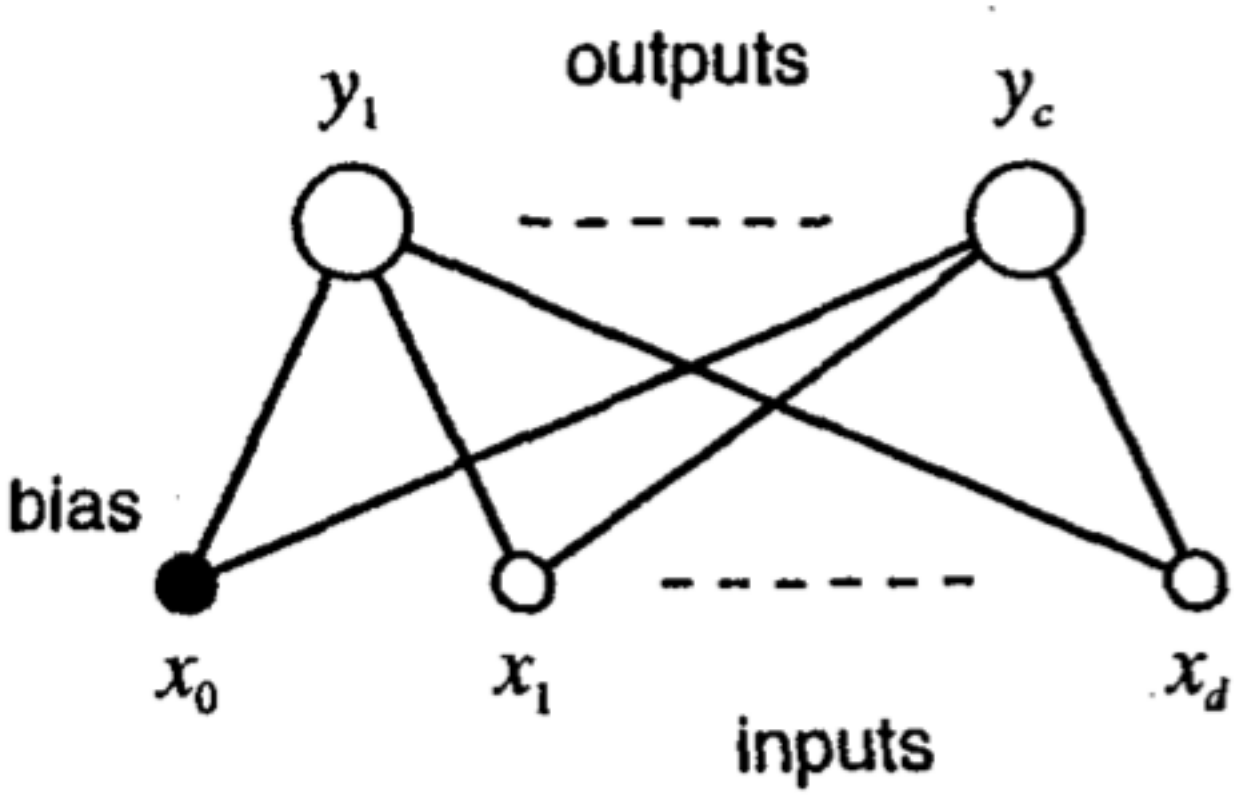
$$l = \frac{w^T x}{\|\mathbf{w}\|} = \frac{-w_0}{\|\mathbf{w}\|}, \text{ where } \mathbf{w} = (w_1, w_2, \ldots, w_d)$$

and we have used $y(x) = 0$

What can a perceptron compute?

- So, we can think of the perceptron as a *linear discriminant*.
- A 2-class discriminant function $y(\mathbf{x})$ is one where we decide that $\mathbf{x}$ is in Category C_1 if $y(\mathbf{x}) \geq 0$ else $\mathbf{x}$ is in C_2
- Note the subtle difference here – instead of the output being 1 or 0, we make a *decision* that $\mathbf{x}$ is in C_1 or C_2
- This makes generalization to > 2 categories simpler…

Multiple Categories



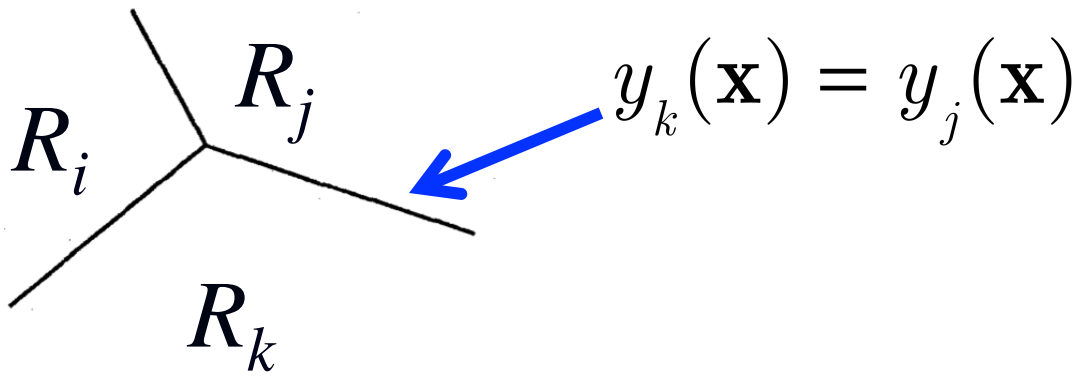
All we do is pick the largest output!
(using the net input to the neuron)

Multiple Categories

- Now we make decisions as follows:

$$\mathbf{x} \text{ is assigned to class } C_k \text{ if } k = \underset{j}{\operatorname{argmax}}\, y_j(\mathbf{x})$$

- Now, the decision boundary between C_i and C_j is where $y_i(\mathbf{x}) = y_j(\mathbf{x})$
- We call the part of input space that is class C_i *Region* R_i:



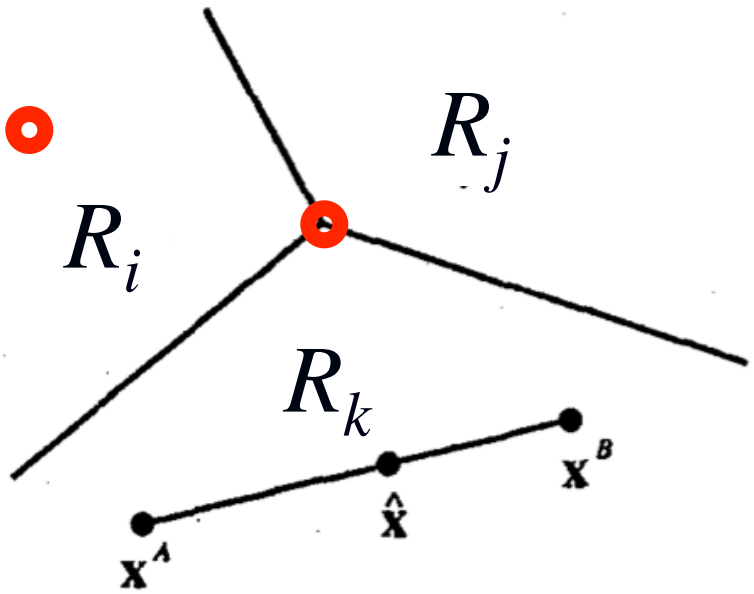
Multiple Categories

- The decision boundary between C_i and C_j is where

$$y_i(\mathbf{x}) = y_j(\mathbf{x})$$

- What can we say about this point?
- What about this point?
- In fact, every region is *convex*:

If x^A and x^B are in a region, so is any point between them.



You can prove that $\hat{x}$ is in Region R_k, given that $\mathbf{x}^A$ and $\mathbf{x}^B$ are…

Hint:

$$\hat{x} = \alpha x^A + (1-\alpha)x^B$$

Summary

- A *perceptron* is a single-layer neural network
- The output is 0 or 1 – it can be thought of as a *classifier*: if the output is 1, the input is in category 1, else it is not.
- The type of categories it can discriminate are *linearly separable categories*.
- The weight vector determines the orientation of the decision boundary, and the bias (w_o) controls where the boundary is along the weight vector.

Summary

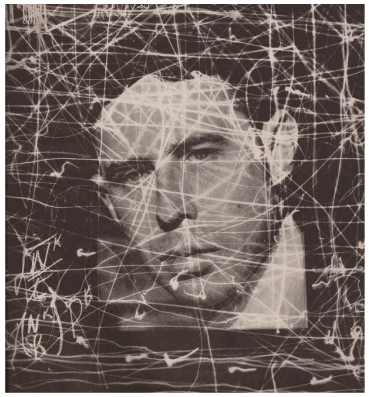
- A *perceptron* is a single-layer neural network
- The output is 0 or 1 – it can be thought of as a *linear classifier*: if the output is 1, the input is in category 1, else it is not.
- The type of categories it can discriminate are *linearly separable categories*.
- The weight vector determines the orientation of the decision boundary, and the bias (w_o) controls where the boundary is along the weight vector.

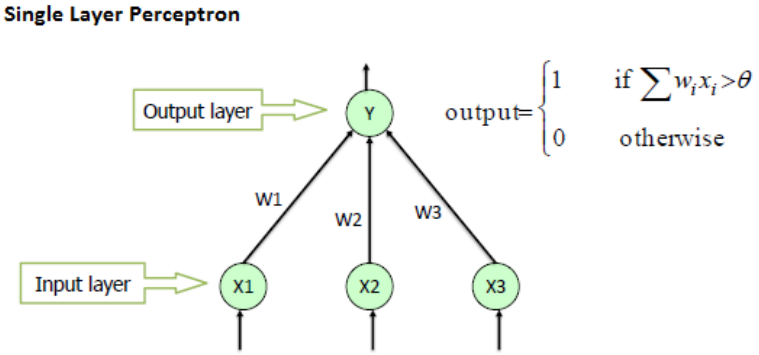
Summary

- Anything a perceptron can compute, it can *learn* to compute, using the delta rule.
- Thinking of the perceptron as a linear discriminant makes the generalization to multiple categories simpler.
- You can train one perceptron for each category.
- Then, you can decide *which* category an input belongs to based on the weighted sum of the inputs:

  Pick the category with the strongest evidence.

Clicker Question





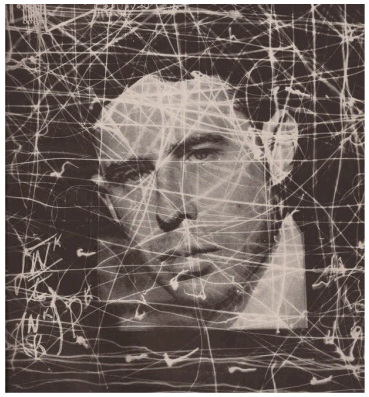
- The guy on the left is:

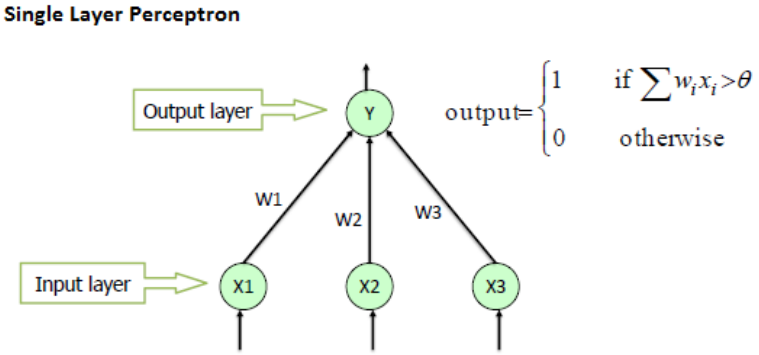
a) Geoff Hinton

b) Frank Rosenblatt

c) Yann LeCun

Clicker Question





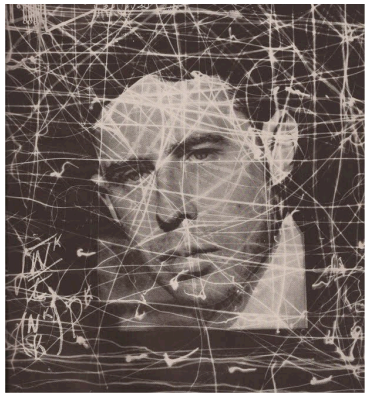
- The guy on the left is:

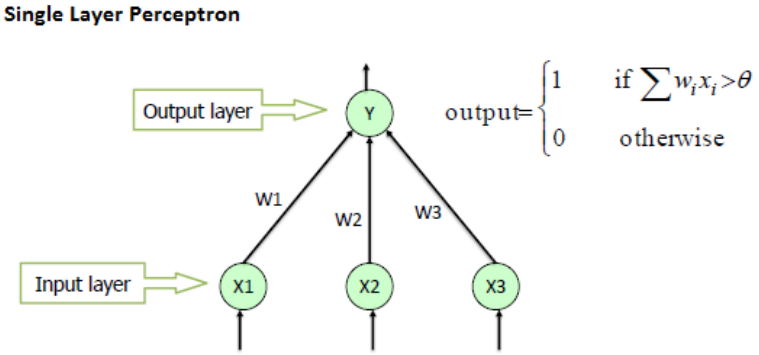
a) Geoff Hinton

b) Frank Rosenblatt

c) Yann LeCun

Clicker Question

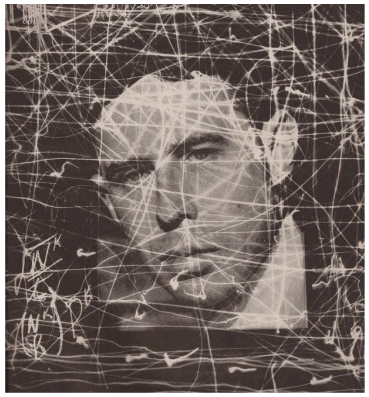


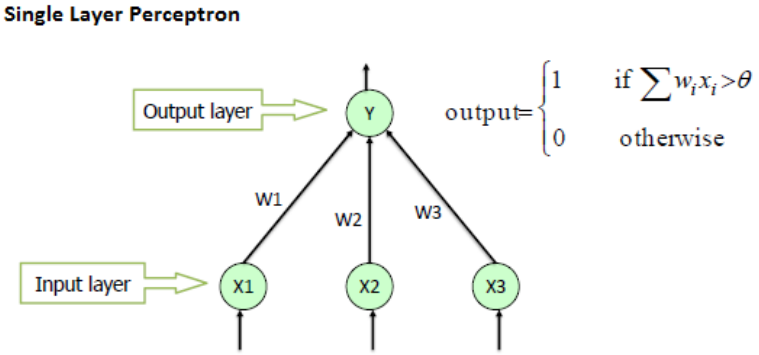


- The guy on the right is:

a) A multi-layer perceptron
b) A single-layer perceptron
c) A binary threshold unit
d) Frank Rosenblatt
e) b&c

Clicker Question





- The guy on the right is:

a) A multi-layer perceptron
b) A single-layer perceptron
c) A binary threshold unit
d) Frank Rosenblatt
e) b&c (sorry, this is a bit of a trick question!)