Linear Regression, Perceptrons, Logistic Regression, and all that

CSE 253

Neural Networks Lecture 3

Overview

- In this lecture, if time, we will cover:
- Linear regression (not really, very short)
- The Perceptron
- Logistic Regression
- Multinomial Regression

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.

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:

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

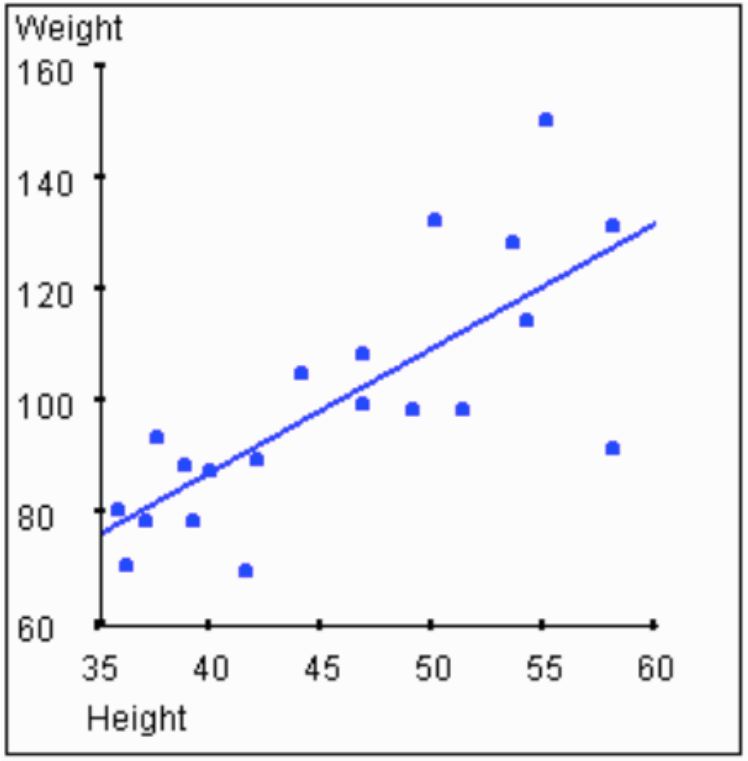
- Where "$\approx$" means "as close to as possible"

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:

Solving for w

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

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

$$\sum_{n=1}^{20}\left(w_0 + w_1 x_1^n - t^n\right) = 0$$

$$\sum_{n=1}^{20}\left(w_0 + w_1 x_1^n - t^n\right)x_1 = 0$$

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

$$w_0 = -\sum_{n=1}^{20}\left(w_1 x_1^n - t^n\right) / 20$$

$$\sum_{n=1}^{20}\left(-\sum_{n=1}^{20}\left(w_1 x_1^n - t^n\right) / 20 + w_1 x_1^n - t^n\right)x_1 = 0$$

$$\sum_{n=1}^{20}\left(-\sum_{n=1}^{20}\left(w_1 x_1^n - t^n\right) / 20 + w_1 x_1^n - t^n\right)x_1 = 0$$

...etc.

Solving it in matrix form:

Much simpler!

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

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

$$\vec{w} = \underbrace{(X^T X)^{-1} X^T}_{\text{The Pseudoinverse}}\ \vec{t}$$

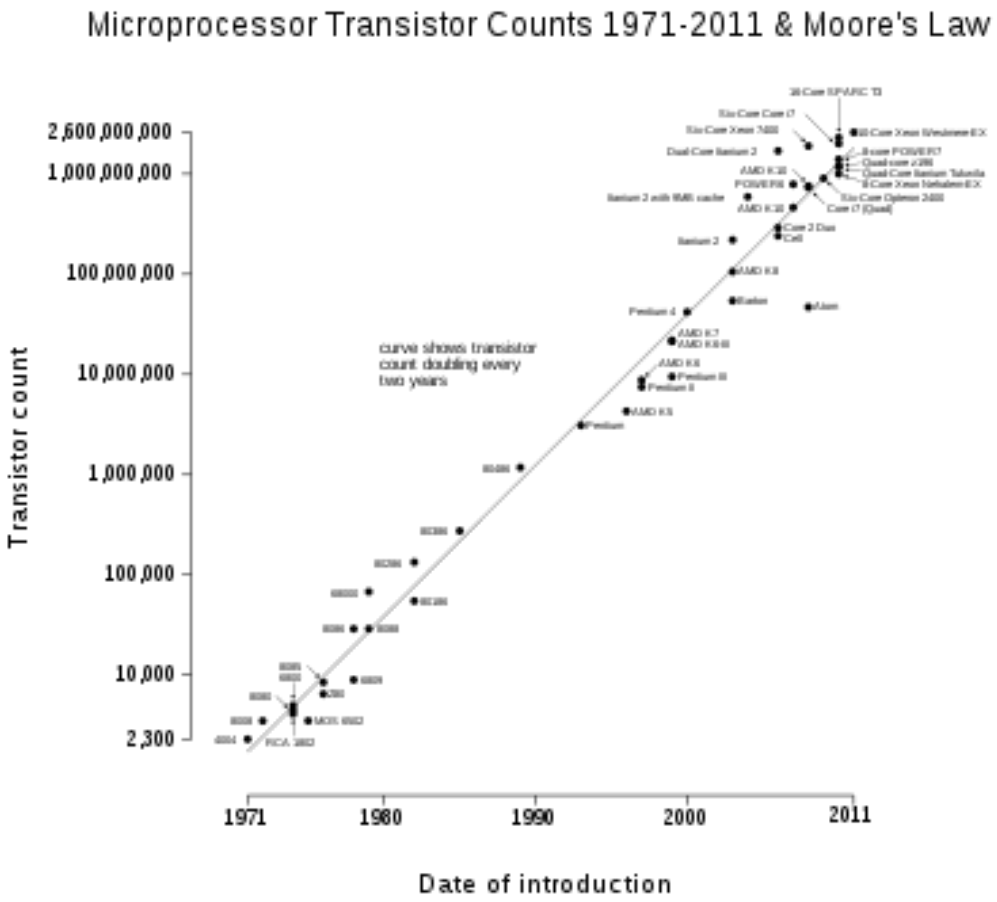
$$\vec{w} = X^\dagger\vec{t}$$

$X^\dagger$ is called the *pseudoinverse* of X

If X is square, then $X^\dagger X = I$

Note: Not all regression is Linear! Nonlinear regression

(by transforming a variable – here, using the log



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!

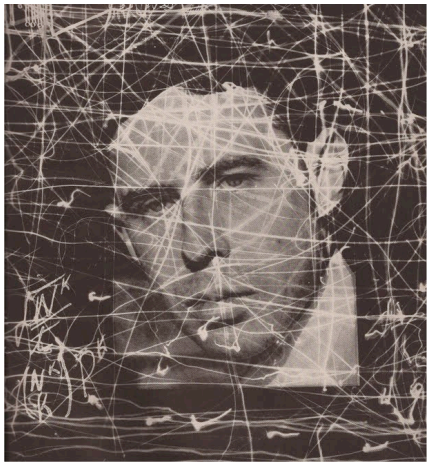
Overview

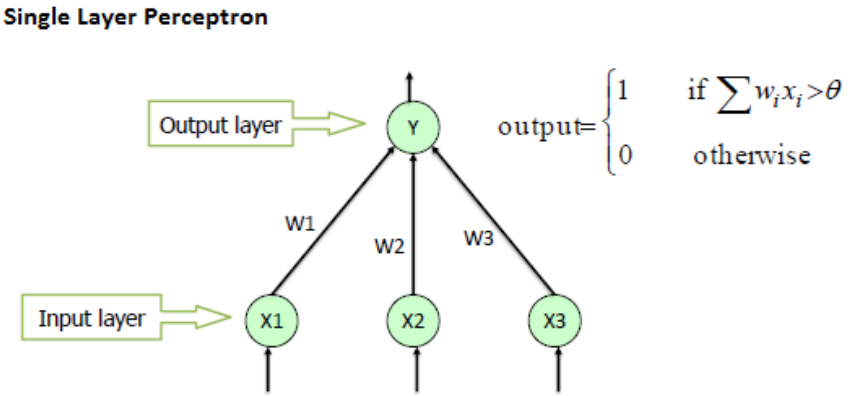
- In this lecture, if time, we will cover:
- Linear regression (not really, very short)
- **The Perceptron**
- Logistic Regression
- Multinomial Regression

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

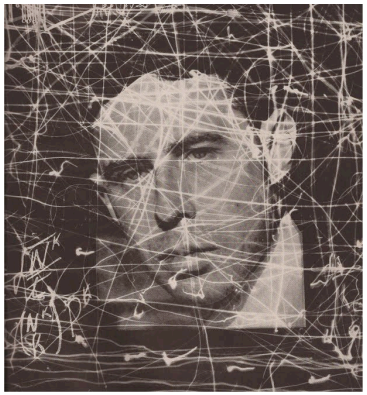


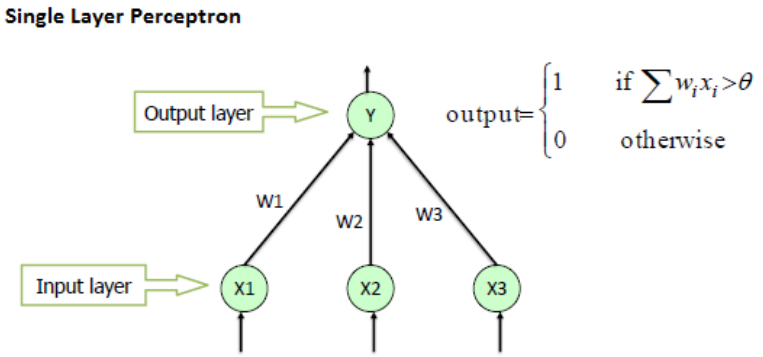


Frank Rosenblatt studied a simple version of a neural net called a ***perceptron***:

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

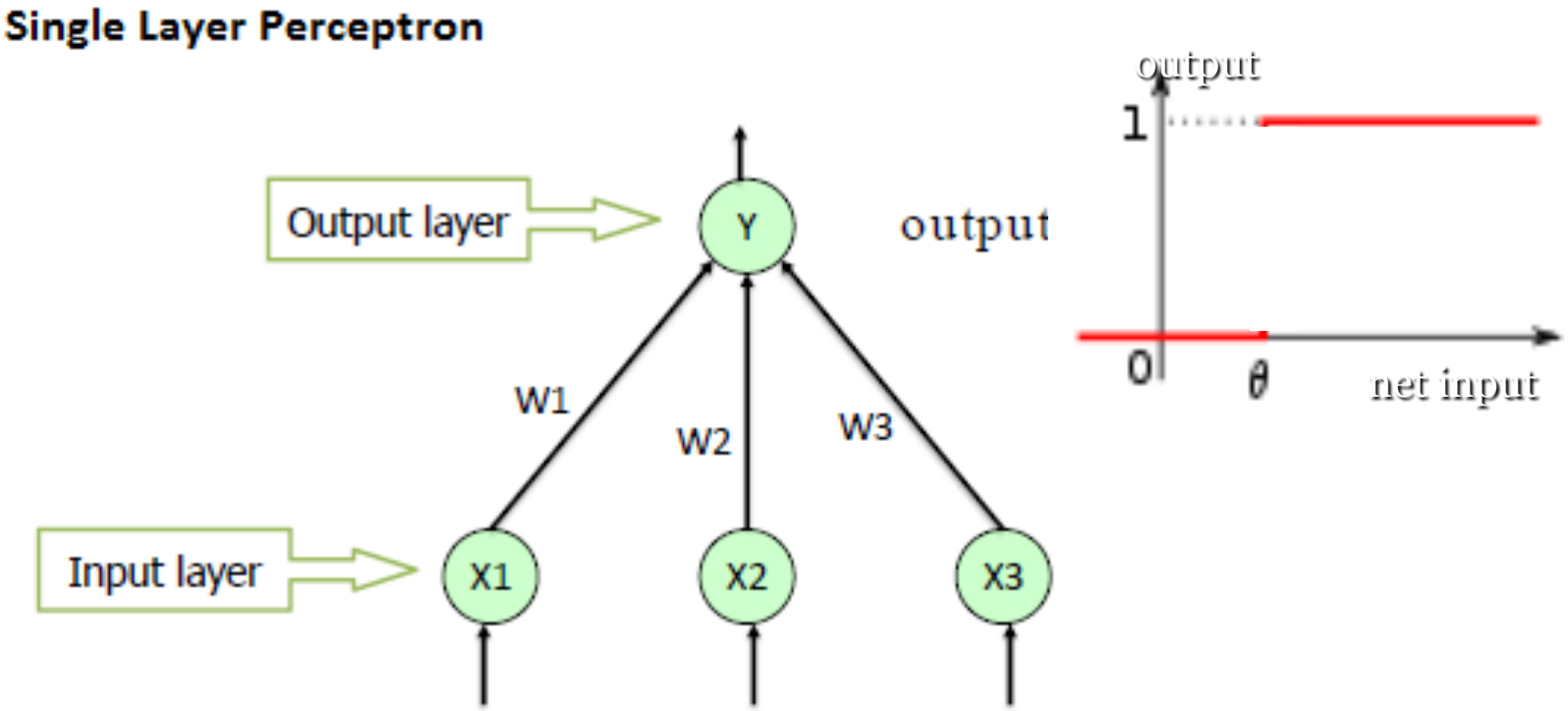
Perceptrons: A bit of history





- 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: DD

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

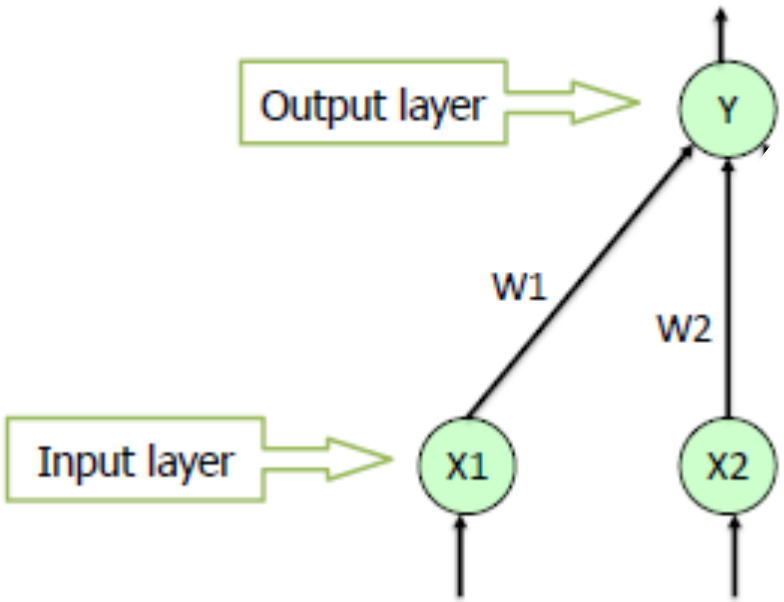
Clicker Question

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

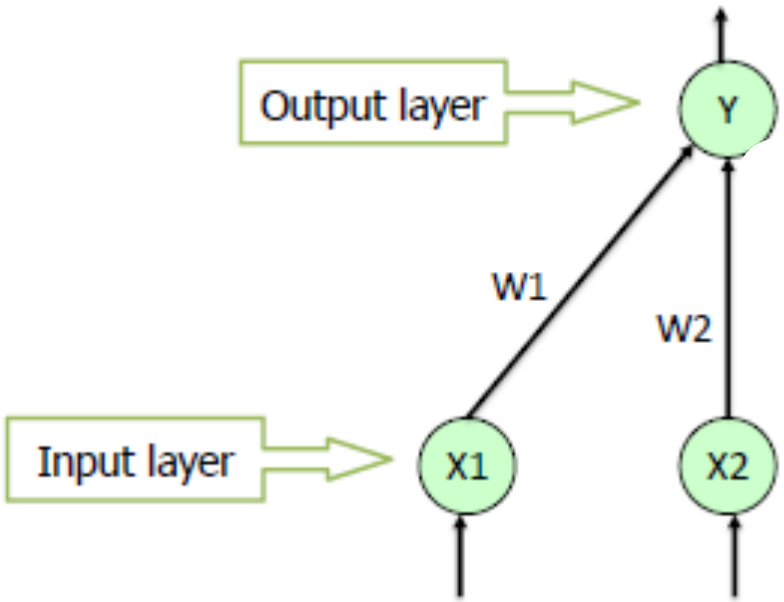
Assume:

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

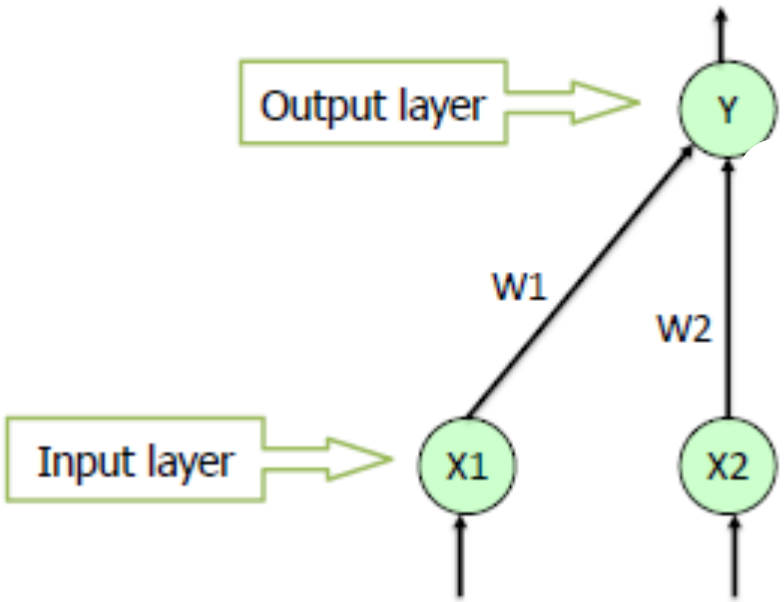
Assume:

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



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

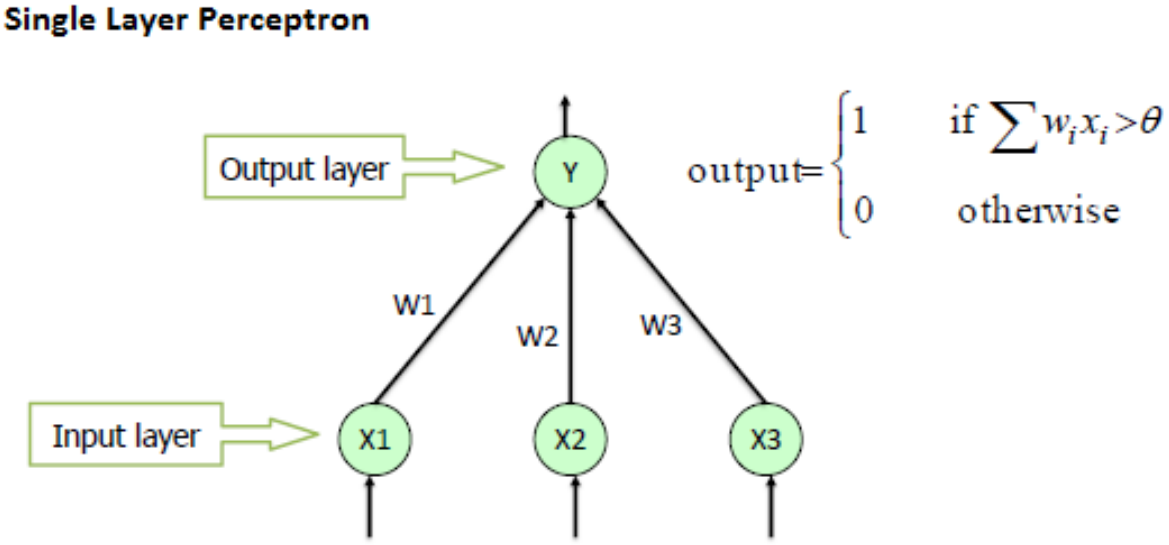
X1	X2	X1 XOR X2
0	0	0
0	1	1
1	0	1
1	1	0

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?

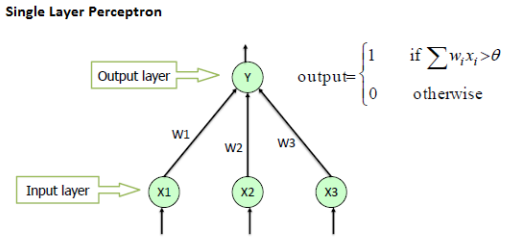
Perceptrons



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)

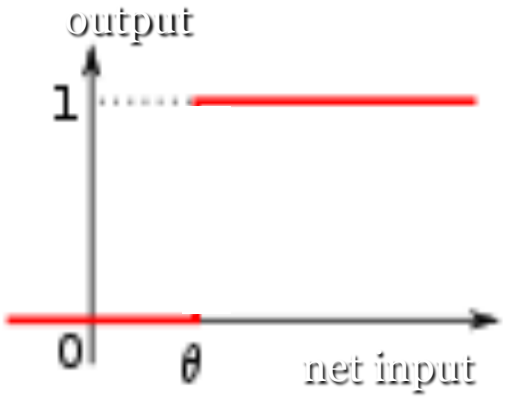
Perceptron Learning

- 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…)

Perceptron Learning Made Simple

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

$$y = \begin{cases} 1 & \text{if } \sum_i w_i x_i \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

- Learning rule:

  If y is 1 and should be 0, then *lower* weights to active inputs and *raise* θ

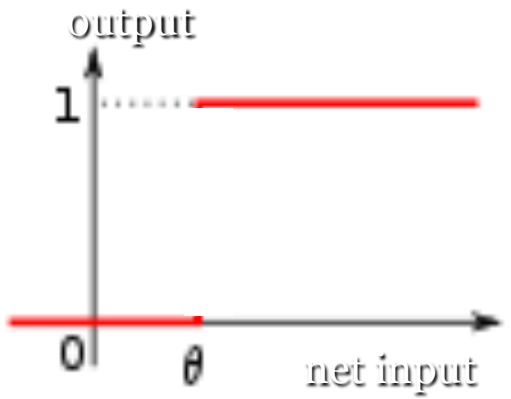
  If y is 0 and should be 1, then *raise* weights to active inputs and *lower* θ

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

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)

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 the *output of the network*:

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

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

- Learning rule (note it is even simpler with a bias!)

  If output is 1 and should be 0, then *lower* weights to active inputs and the bias w_0

  If output is 0 and should be 1, then *raise* weights to active inputs and the bias w_0

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

Perceptron Learning Made Simple for Computer Science

- Output activation rule:

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

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

- Learning rule:

$$w_i = w_i + \alpha*(\text{teacher - output})*x_i$$

(α is the *learning rate*)

- 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: δ = (teacher – output)
- *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)

  $w_i = w_i + (\text{teacher - output})*x_i$

  $w_i = w_i + \delta x_i$

  $(\delta = (\text{teacher-output}))$

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

  $w_i = w_i + (\text{teacher} - \text{output})x_i$

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

  $= w_i - 1$ *lower* weight

  What if x_i is inactive?

  $w_i = w_i + (\text{teacher} - \text{output})x_i$

  $w_i = w_i + (\text{teacher} - \text{output})0$

  $= w_i$ (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 *lower* the threshold

  $w_i = w_i + (\text{teacher} - \text{output})x_i$

  $w_i = w_i + (1 - 0)1$

  $= w_i + 1$ (*raise* weight)

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

- What about the bias? Recall $w_0 = -\theta$
- 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:

$$w_i = w_i + (\text{teacher} - \text{output})x_0$$

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

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

$w_0 = w_0 - 1$ *lower* w_0

Let's convince ourselves these are the same…

- What if we get it right??? Then teacher = output, and …
- Learning rule:

$$w_i = w_i + (\text{teacher} - \text{output})x_i$$

$$w_i = w_i + 0 * x_i$$

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

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?

- First, let's rewrite the activation rule:

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

$$output = \begin{cases} 1 & \text{if } \sum_i w_i x_i - \theta \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$output = \begin{cases} 1 & \text{if } \sum_i w_i x_i + w_0 \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

- w_o is also known as the *bias* and says when the neuron likes to be 1 in the absence of other input. REMEMBER: $bias = w_o = -\theta$

What can a perceptron compute?

- One more step:

$$output = \begin{cases} 1 & \text{if } \sum_i w_i x_i + w_0 \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

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

- Here we simply are rewriting the expression as a function:

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

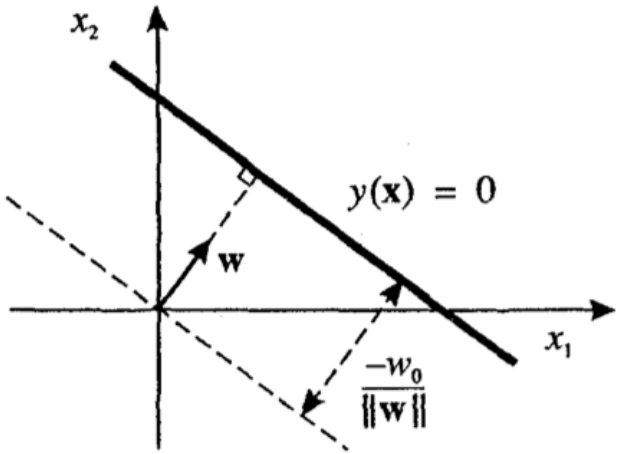
- Notation: BOLD **w** and BOLD **x** are vectors.

What can a perceptron compute?

- Now our activation rule is:
- We call $y(\mathbf{x})=0$ the *decision boundary* – where the output changes from 0 to 1.

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

- 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:



What can a perceptron compute?

- Now our activation rule is:
- Now, we call $y(\mathbf{x})=0$ the
- *decision boundary*
  - where the output changes from 0 to 1.
- Why is it a line in 2D?

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

$y(x) = 0$ ➔

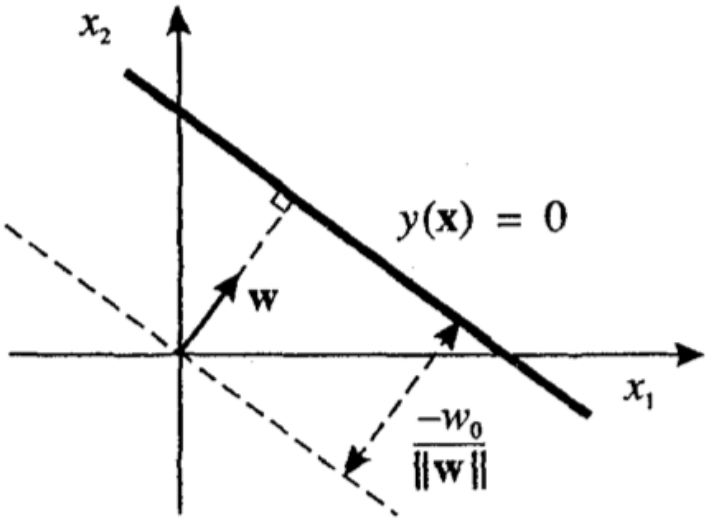
$w_1x_1 + w_2x_2 + w_0 = 0$ ➔

$w_2x_2 = -w_1x_1 - w_0$ ➔

$x_2 = -(w_1/w_2)\, x_1 - w_0/w_2$

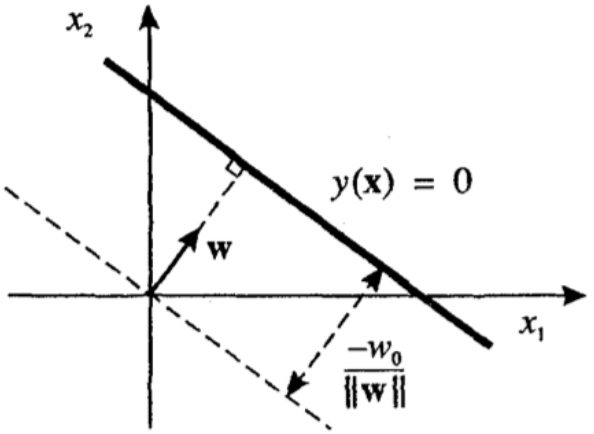
$x_2 = m\, x_1 + intercept$

(in slope-intercept form)



What can a perceptron compute?

- So for 2D, it 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

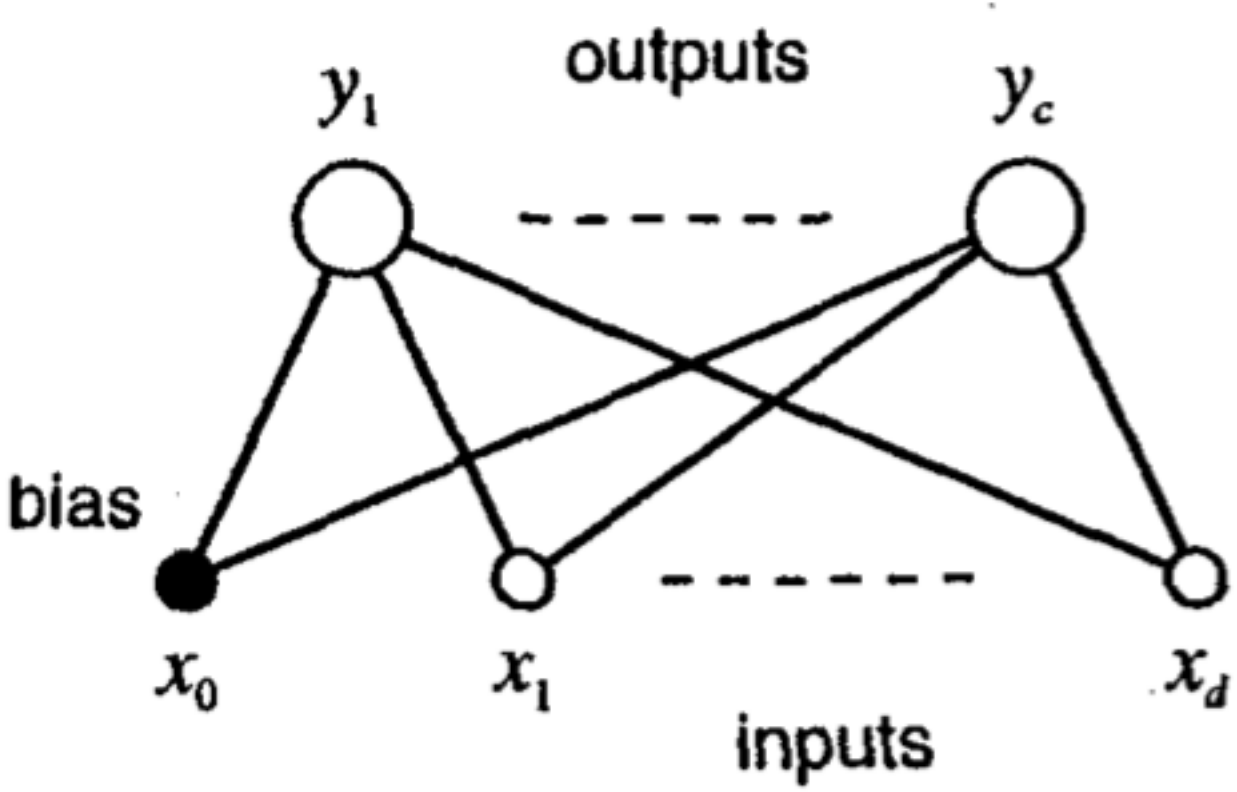


$$y(\mathbf{x}^A) = 0 = y(\mathbf{x}^B),$$
$$y(\mathbf{x}^A) - y(\mathbf{x}^B) = 0$$
$$\mathbf{w}^T\mathbf{x}^A + w_0 - \mathbf{w}^T\mathbf{x}^B - w_0 = 0$$
$$\mathbf{w}^T(\mathbf{x}^A - \mathbf{x}^B) = 0$$

And the distance l to the origin is given by:

$$l = \frac{\mathbf{w}^T\mathbf{x}}{\|\mathbf{w}\|} = \frac{-w_0}{\|\mathbf{w}\|}$$

Multiple Categories



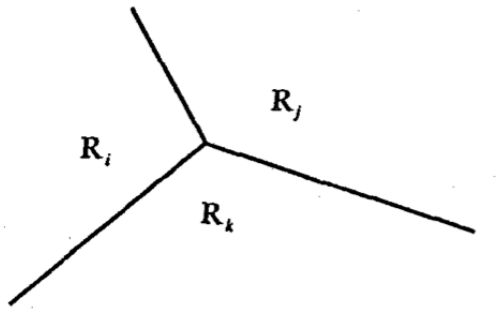
All we do is pick the largest output!
(using the net input to the neuron)

Multiple Categories

- Now we make decisions as follows:

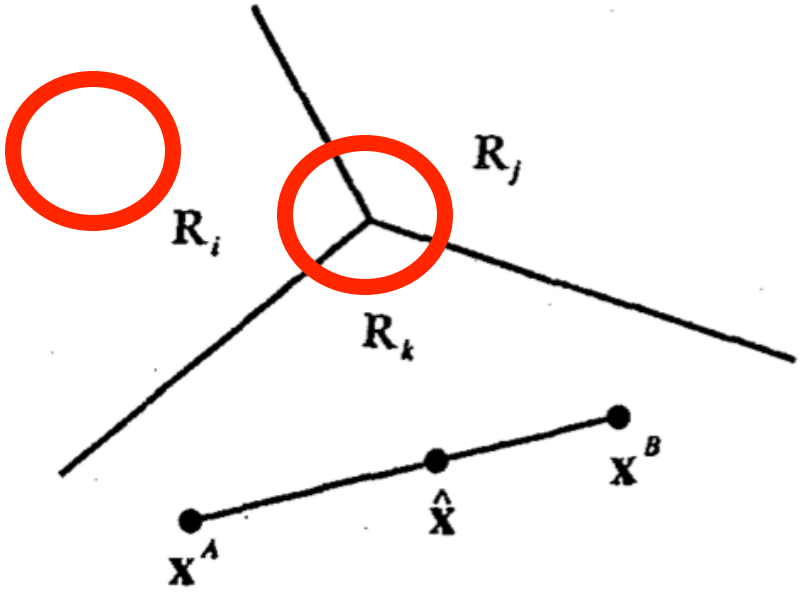
$\mathbf{x}$ is assigned to class C_k if $y_k(\mathbf{x}) \geq y_j(\mathbf{x}) \quad \forall j \neq k$

- 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*



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

- 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.