

Neural Networks

David Kauchak
CS311
Spring 2013

Admin

- Assignment 5
 - due Friday at 6pm
- Schedule for the next two weeks



What is this? How did you know?

$$293871947009$$
$$* \sqrt{52.86301}$$
$$/ 80.2341 = ?$$

What is the answer to this calculation?


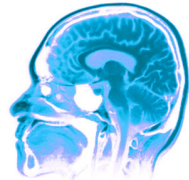
293871947009
 * $\sqrt{52.86301}$
 / 80.2341
= 26630240520.936812470902167425359

A computer can do this almost instantly!

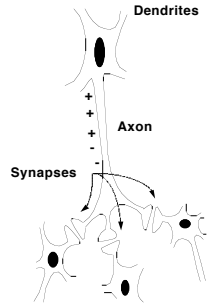
Neural Networks

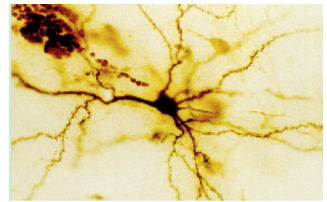
Neural Networks try to mimic the structure and function of our nervous system

People like biologically motivated approaches (e.g. genetic algorithms)

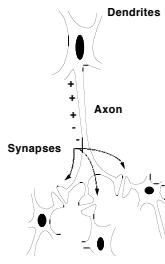
Our Nervous System





Neuron

Our nervous system: *the computer science view*

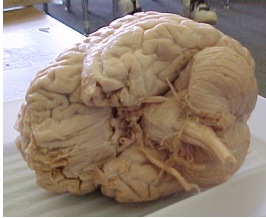


the human brain is a large collection of interconnected neurons

a **NEURON** is a brain cell

- collect, process, and disseminate electrical signals
- Neurons are connected via synapses
- They **FIRE** depending on the conditions of the neighboring neurons

Our nervous system



The human brain

- contains $\sim 10^{11}$ (100 billion) neurons
- each neuron is connected to $\sim 10^4$ (10,000) other neurons
 - What is this in CS language?
- Neurons can fire as fast as 10^{-3} seconds

How does the brain compare to a computer?

Man vs. Machine



10^{11} neurons
 10^{11} neurons
 10^{14} synapses
 10^{-3} "cycle" time

10^9 transistors
 10^{11} bits of ram
 10^{13} bits on disk
 10^{-9} cycle time

Brains are still pretty fast



Who is this?

Brains are still pretty fast



If you were me, you'd be able to identify this person in 10^{-1} s

Brains are still pretty fast

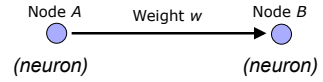


Given a neuron firing time of 10^{-3} s, **how many neurons in sequence could fire in this time?**

- A few hundred

What are possible explanations?

- either neurons are performing some very complicated computations
- brain is taking advantage of the massive parallelization



w is the strength of signal sent between A and B.

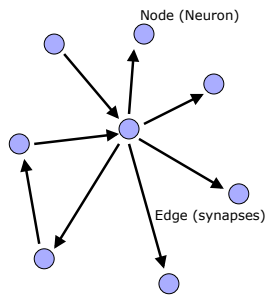
If A fires and w is **positive**, then A **stimulates** B.

If A fires and w is **negative**, then A **inhibits** B.

If a node is stimulated enough, then it also fires.

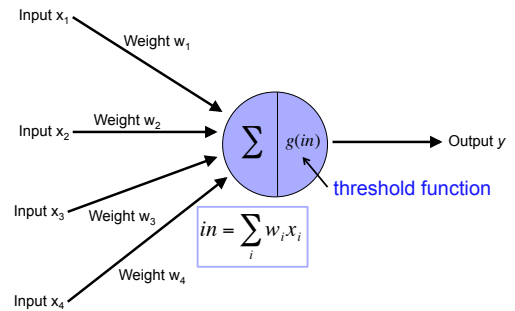
How much stimulation is required is determined by its **threshold**.

Neural Networks



our approximation to the brain

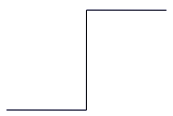
A Single Neuron/Perceptron



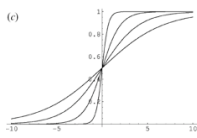
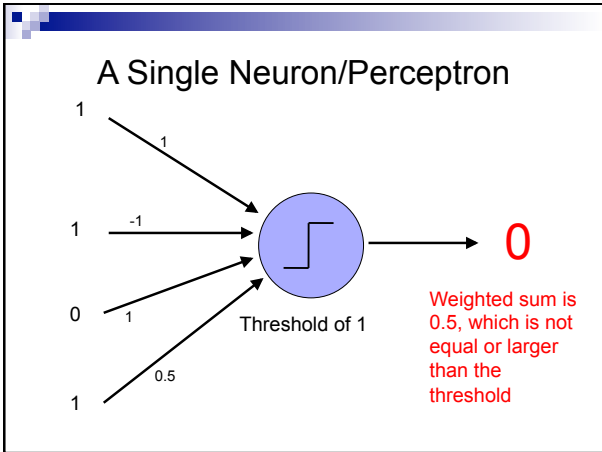
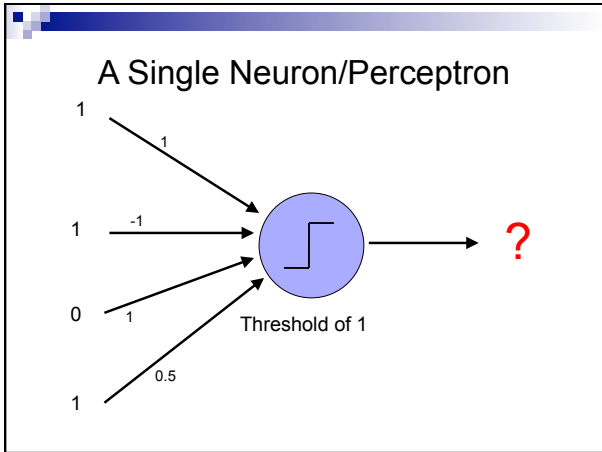
Possible threshold functions

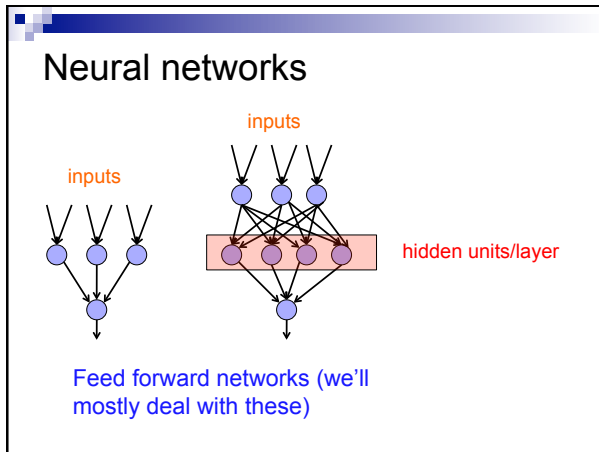
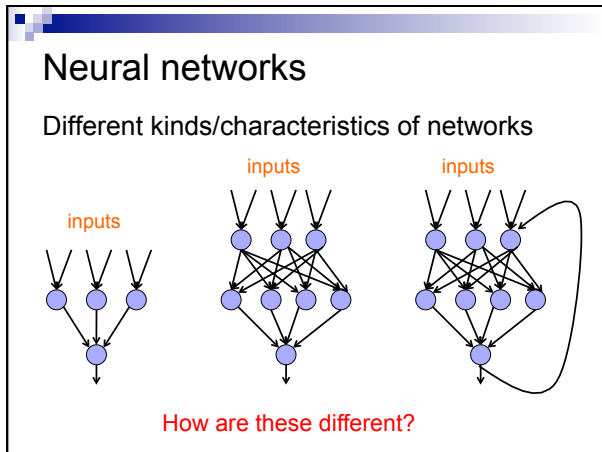
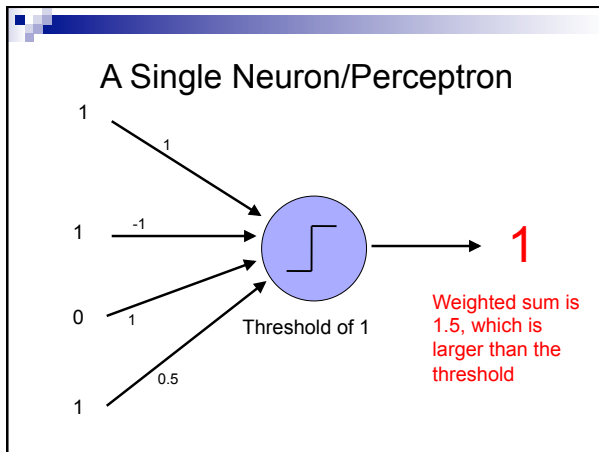
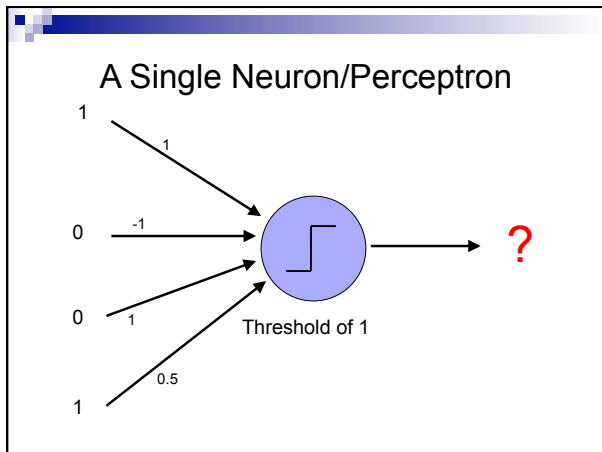
Possible threshold functions

hard threshold:
if in (the sum of weights) \geq threshold 1, 0 otherwise

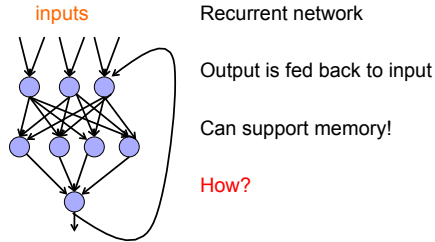


Sigmoid

$$g(x) = \frac{1}{1 + e^{-ax}}$$





Neural networks



History of Neural Networks

McCulloch and Pitts (1943) – introduced model of artificial neurons and suggested they could learn

Hebb (1949) – Simple updating rule for learning

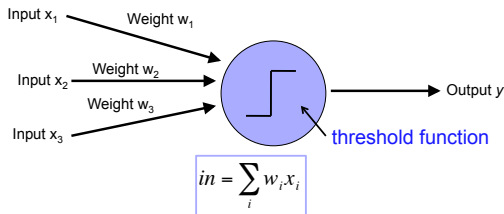
Rosenblatt (1962) - the *perceptron* model

Minsky and Papert (1969) – wrote *Perceptrons*

Bryson and Ho (1969, but largely ignored until 1980s) – invented back-propagation learning for multilayer networks

Perceptron: single neuron

First wave in neural networks in the 1960's



What defines the perceptron? How could we “train” it?

Perceptron

Defined by its threshold and input weights can be modified

If the neuron doesn't give the desired output, then it has made a mistake.

Input weights and threshold can be changed according to a learning algorithm when it makes a mistake

Examples - Logical operators

AND – if all inputs are 1, return 1, otherwise return 0

OR – if at least one input is 1, return 1, otherwise return 0

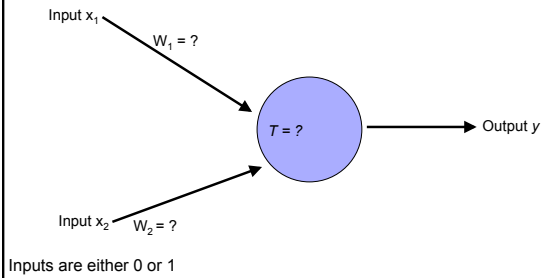
NOT – return the opposite of the input

XOR – if exactly one input is 1, then return 1, otherwise return 0

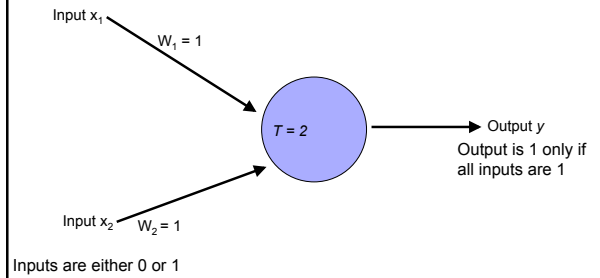
AND

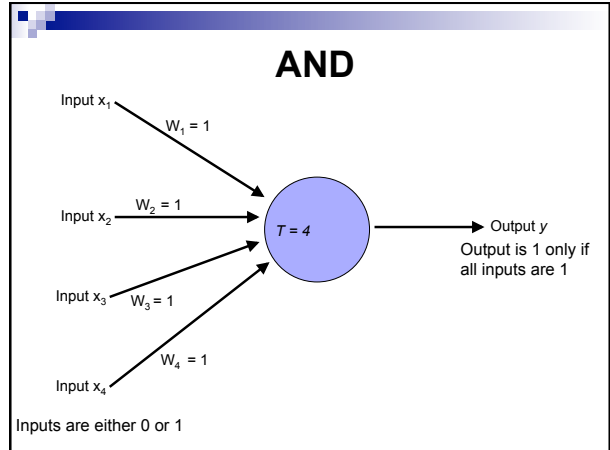
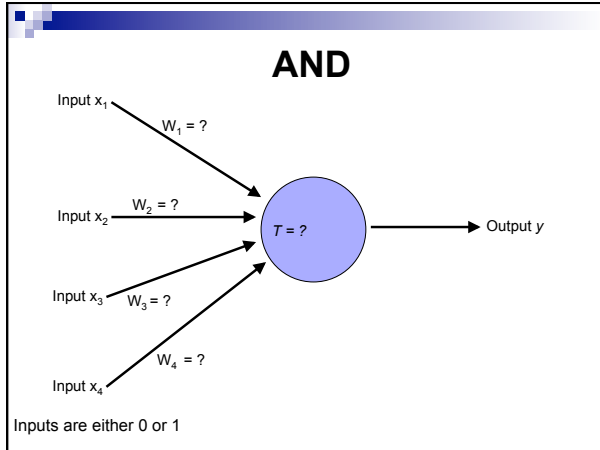
x_1	x_2	x_1 and x_2
0	0	0
0	1	0
1	0	0
1	1	1

AND



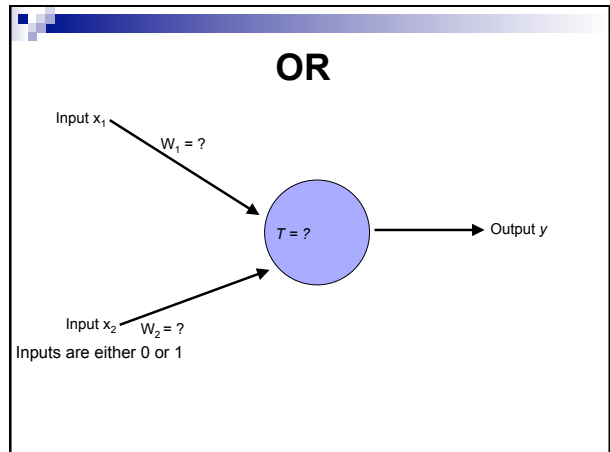
AND

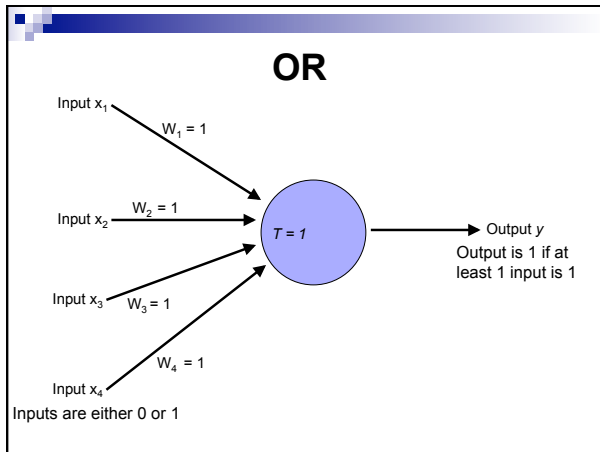
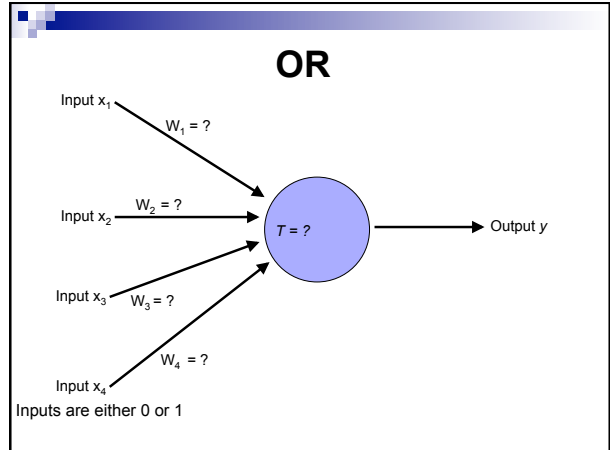
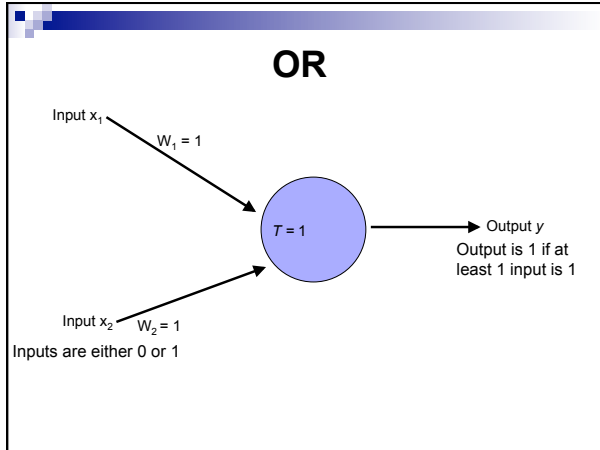




OR

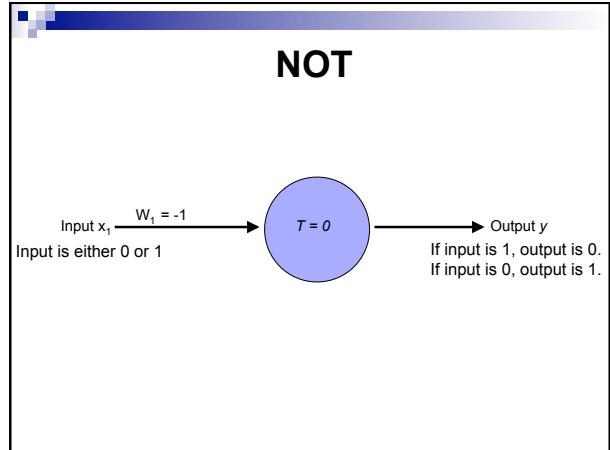
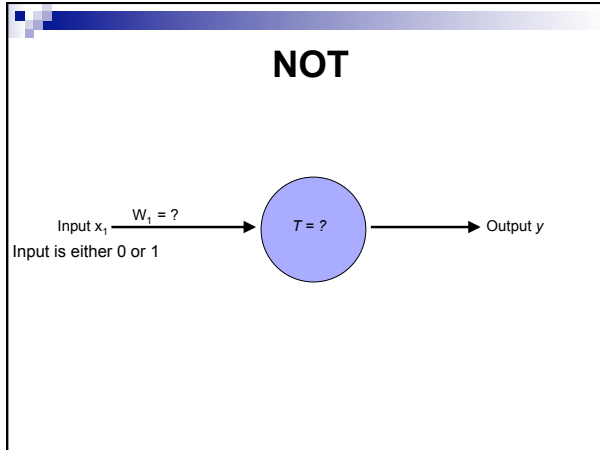
x_1	x_2	x_1 or x_2
0	0	0
0	1	1
1	0	1
1	1	1





NOT

x_1	not x_1
0	1
1	0



How about...

x_1	x_2	x_3	y
0	0	0	1
0	1	0	0
1	0	0	1
1	1	0	0
0	0	1	1
0	1	1	1
1	0	1	1
1	1	1	0

Input x_1 $W_1 = ?$
 Input x_2 $W_2 = ?$
 Input x_3 $W_3 = ?$ → $T = ?$ → Output y

Training neural nets

We'd like to train neural networks
 We can learn to classify
 We can also learn a *regression function* from input to a real value

output: 1, -1
 What can we adjust/learn for NNs in general?

NN parameters

Learn the individual weights between nodes

Learn individual node parameters (e.g. threshold)

An aside: linear regression

Given some points, find the **line** that best fits/explains the data

How can we find this line?

An aside: linear regression

Learn a line h that minimizes an error function:

$$error(h) = \sum_{i=1}^n (y_i - h(x_i))^2$$

in the case of a 2d line:

$$error(h) = \sum_{i=1}^n (y_i - \underbrace{(w_1 x_i + w_0)}_{\text{function for a line}})^2$$

Linear regression

We'd like to *minimize* the error

Find w_1 and w_0 such that the error is minimized

$$error(h) = \sum_{i=1}^n (y_i - (w_1 x_i + w_0))^2$$

How can we do this?

Linear regression

minimize: $error(h) = \sum_{i=1}^n (y_i - (w_1 x_i + w_0))^2$

Partial derivatives give us the slope in that dimension

Option 1

- When slope is 0, it's a min or a max
- This approach gets hard if we want to do non-linear regression

Option 2: gradient descent

- move in the appropriate direction (but not necessarily down to 0)
- we can view the problem as a search for w_i that minimizes the loss

Gradient descent

If the loss function is convex, what does this mean for our minimum?

- In three dimensions, think about a curved piece of paper
- Or, think of it like skiing in a big bowl

Approach:

- pick a starting point (w)
- repeat until loss doesn't decrease in all dimensions:
 - pick a dimension
 - move a small amount in that dimension towards decreasing loss (using the derivative)

Gradient descent

pick a starting point (w)

repeat until loss doesn't decrease in all dimensions:

- pick a dimension
- move a small amount in that dimension towards decreasing loss (using the derivative)

$$w_i = w_i - \alpha \frac{d}{dw_i} error(w)$$

learning rate (how much we want to move in the error direction)

Linear gradient descent

pick a starting point (w)

repeat until loss doesn't decrease in all dimensions:

- pick a dimension
- move a small amount in that dimension towards decreasing loss (using the derivative)

$$w_i = w_i - \alpha \sum_{j=1}^n x_{j,i} (y_j - h(x))$$

sum the error over all the examples the value of the example in that dimension difference between actual and predicted

intuitively, why does this work?

Back to training a perceptron

x_1	x_2	x_1 and x_2
0	0	0
0	1	0
1	0	0
1	1	1

We want to train a perceptron to learn a function given training data

Input x_1 $w_1 = ?$
 Input x_2 $w_2 = ?$
 Threshold $T = ?$
 Output y

A Single Perceptron

Input x_1 Weight w_1
 Input x_2 Weight w_2
 Input x_3 Weight w_3
 Input x_4 Weight w_4
 Threshold T
 Output y

Does this learning problem look like anything we've seen?

If $w_1x_1 + w_2x_2 + \dots + w_nx_n \geq T$,
 then the output of n is 1.
 Otherwise,
 the output of n is 0.

Perceptron Training Rule

<p>linear regression</p> <ul style="list-style-type: none"> - pick a random weight vector - repeat until loss doesn't decrease in all dimensions: <ul style="list-style-type: none"> - pick a dimension - move a small amount in that dimension towards decreasing loss (using the derivative) 	<p>perceptron learning</p> <ul style="list-style-type: none"> - pick a random weight vector - repeat until we correctly classify all the points: <ul style="list-style-type: none"> - pick an example - if we get it wrong: <ul style="list-style-type: none"> - modify the weights a small amount
--	--

Key difference: regression error vs. classification error

Perceptron Training Rule

<p>linear regression</p> <ul style="list-style-type: none"> - pick a random weight vector - repeat until loss doesn't decrease in all dimensions: <ul style="list-style-type: none"> - pick a dimension - move a small amount in that dimension towards decreasing loss (using the derivative) $w_i = w_i - \alpha \sum_{j=1}^n x_{j,i}(y_j - h(x))$	<p>perceptron learning</p> <ul style="list-style-type: none"> - pick a random weight vector - repeat until we correctly classify all the points: <ul style="list-style-type: none"> - pick an example - if we get it wrong: <ul style="list-style-type: none"> - modify the weights a small amount $w_i = w_i - \alpha x_i (y_j - h(x))$
--	--

Modifying the weights

Only update the weights when we get an example wrong

For each dimension i in the weight vector:

$$w_i = w_i - \alpha x_i (y_j - h(x))$$

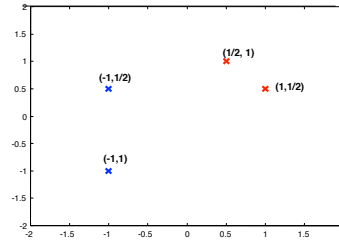
learning rate

how much this feature played a role (e.g. active or not)

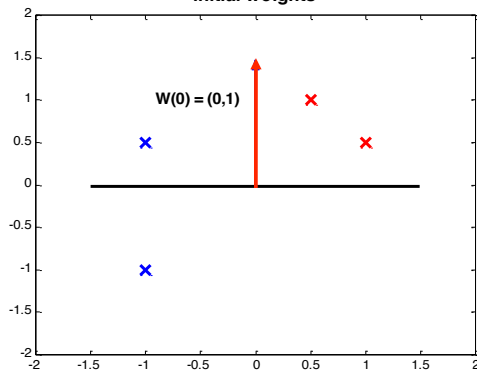
difference between actual and predicted

Example: a simple problem

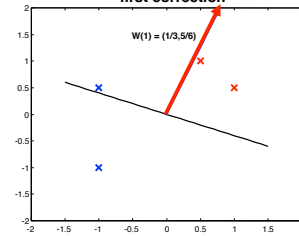
4 points linearly separable

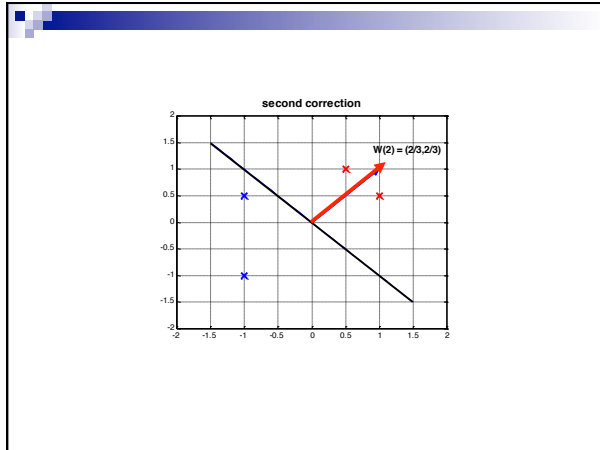


initial weights



first correction





Perceptron learning

How does this compare to say the linear SVM?

Perceptron learning

Only works when data is linearly separable

Voted perceptron helps get a better linear separator

Has remained popular as an approach for learning weights in high dimensional space

Other approaches for training perceptrons do exist:

- Delta rule (Gradient Descent Approach)
- Linear Programming

XOR

x_1	x_2	$x_1 \text{ XOR } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

How would the perceptron do?

Linearly Separable

x_1	x_2	$x_1 \text{ and } x_2$
0	0	0 (blue)
0	1	0 (blue)
1	0	0 (blue)
1	1	1 (red)

x_1	x_2	$x_1 \text{ or } x_2$
0	0	0 (blue)
0	1	1 (red)
1	0	1 (red)
1	1	1 (red)

x_1	x_2	$x_1 \text{ xor } x_2$
0	0	0 (blue)
0	1	1 (red)
1	0	1 (red)
1	1	0 (blue)

Perceptrons

1969 book by Marvin Minsky and Seymour Papert

The problem is that they can only work for classification problems that are linearly separable

Insufficiently expressive

"Important research problem" to investigate multilayer networks although they were pessimistic about their value

XOR

x_1	x_2	$x_1 \text{ xor } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

XOR

x_1	x_2	$x_1 \text{ xor } x_2$
0	0	0
0	1	1
1	0	1
1	1	0

Learning in multilayer networks

Similar idea to perceptron learning

Examples are presented to the network

If the network computes an output that matches the desired, nothing is done

If there is an error, then the weights are adjusted to balance the error

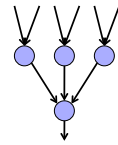
Learning in multilayer networks

Key idea for perceptron learning: if the perceptron's output is different than the expected output, update the weights

Any problem with this for general NNs?

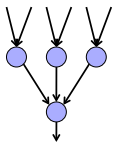


perceptron



multi-layer network

Learning in multilayer networks



multi-layer network

$$w_i = w_i - \alpha \frac{d}{dw_i} error(w)$$

What does this derivative look like?

Logistic and other thresholds

$$w_i = w_i - \alpha \frac{d}{dw_i} error(w)$$



the slope at that point

$$w_i = w_i - \alpha x_i g'(w \cdot \vec{x})(y - h(x))$$

Depends on the threshold function!

Logistic and other thresholds


$$w_i = w_i - \alpha \frac{d}{dw_i} error(w)$$

↓

the slope at that point

$$w_i = w_i - \alpha x_i g'(w \cdot \bar{x})(y - h(x))$$

Any problem with using the threshold function?



Logistic and other thresholds

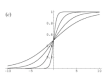
$$w_i = w_i - \alpha \frac{d}{dw_i} error(w)$$

↓

Now have a term for the slope at that point

$$w_i = w_i - \alpha x_i g'(w \cdot \bar{x})(y - h(x))$$


We'll use a sigmoid, which approximates a threshold but has a well defined derivative



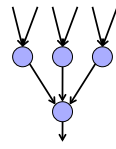
Learning in multilayer networks

Key idea for perceptron learning: if the perceptron's output is different than the expected output, update the weights

Any other problem with this for general NNs?



perceptron




multi-layer network

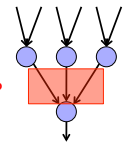
Learning in multilayer networks

Key idea for perceptron learning: if the perceptron's output is different than the expected output, update the weights

Challenge: for multilayer networks, we don't know what the expected output/error is for the internal nodes



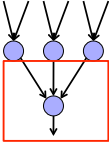
perceptron



multi-layer network

Backpropagation

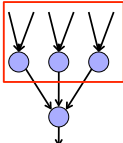
Say we get it wrong, and we now want to update the weights



We can calculate the actual error here

Backpropagation

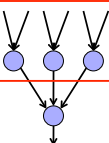
Say we get it wrong, and we now want to update the weights



We'd like to propagate the error back

Backpropagation

Say we get it wrong, and we now want to update the weights



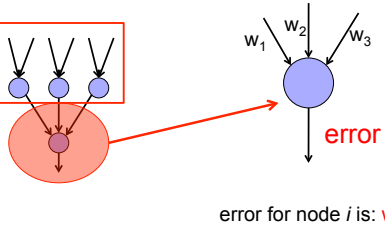
"back-propagate" the error:

Assume all of these nodes were responsible for some of the error

How can we figure out how much they were responsible for?

Backpropagation

Say we get it wrong, and we now want to update the weights



error for node i is: w_i error

Backpropagation

Say we get it wrong, and we now want to update the weights

error for node i is: w_i error

how much do we update the weights by?

Backpropagation

Say we get it wrong, and we now want to update the weights

$$w_i = w_i - \alpha g'(w \cdot x) (y - h(x)) x_i$$

↓

$$w_i = w_i - \alpha g'(w \cdot a_i) \text{error } a_i$$

the nodes fraction of the error

Backpropagation

calculate the error at the output layer

backpropagate the error up the network

- if a node has multiple output nodes, sum the error of these nodes

Update the weights based on these errors

Can be shown that this is the appropriate thing to do based on our assumptions

That said, many neuroscientists don't think the brain does backpropagation of errors

Neural network regression

Given enough hidden nodes, you can learn *any* function with a neural network

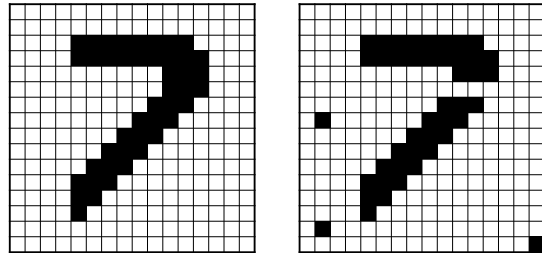
Challenges:

Neural network regression

Given enough hidden nodes, you can learn *any* function with a neural network

Challenges:

- overfitting
- picking a network structure (like picking our Bayes net structure)
- can require a lot of tweaking of parameters, preprocessing, etc.



Popular for digit recognition and many computer vision tasks
<http://yann.lecun.com/exdb/mnist/>

Cog sci people like NNs

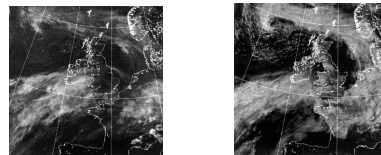
Expression/emotion recognition

- Gary Cottrell et al



Language learning

Interpreting Satellite Imagery for Automated Weather Forecasting



Summary

Perceptrons (one layer networks)

- convenient and fast to train
- linear separator
- work well in high-dimensions
- insufficiently expressive in general

Multi-layer networks are sufficiently expressive and can be trained by error back-propagation

Many applications including speech, driving, hand written character recognition, fraud detection, driving, etc.