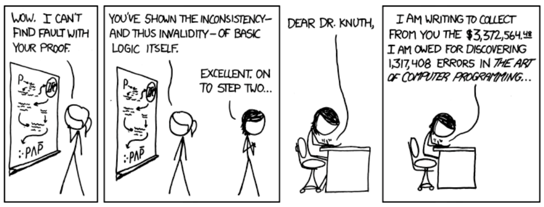


http://www.xkcd.com/816/

Neural Networks

David Kauchak
CS151
Fall 2010

Admin

- Pre-registration pizza
  - Tuesday 5:30-6:30pm
  - Edmunds lounge
- Assignment 5 due Wed. at midnight

Reviews

- Much improved from last time
- Some fun papers
- Technical correctness
  - most of you mentioned the experiments/results section
  - also comment on the correctness of the actual method description
- citation:
  - <authors>. <year>. <title>. <how_published>.
  - be consistent and keep it simple
  - look at the papers for examples
  - don't just copy it from citeseer!

What is this? How did you know?

293871947009

* √52.86301

/ 80.2341 = ?

What is the answer to this calculation?

293871947009

* √52.86301

/ 80.2341

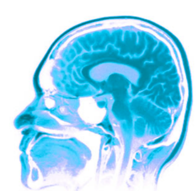
= **26630240520.936812470902167425359**

A computer can do this almost instantly!

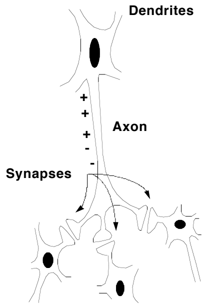
Neural Networks

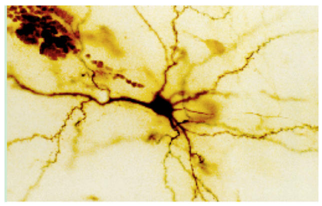
Neural Networks try to mimic the structure and function of our nervous system

People like biologically motivated approaches (like genetic algorithms)



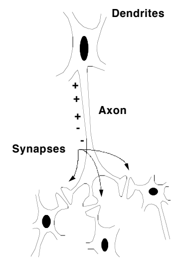
Our Nervous System





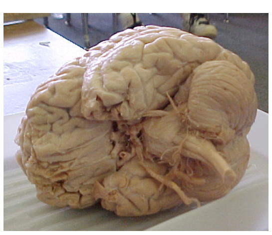
Neuron

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

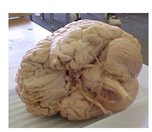
Our nervous system



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

How does this compare to a computer?

Man vs. Machine



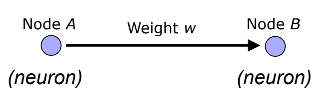
10^{11} neurons	10^{9} transistors
10^{11} neurons	10^{11} bits of ram
10^{14} synapses	10^{13} bits on disk
10^{-3} "cycle" time	10^{-9} cycle time

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
- 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.

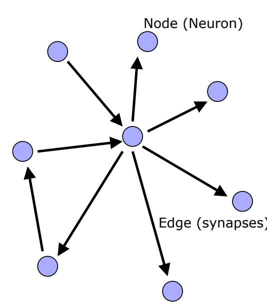
If *A* fires and *w* is **positive**, then *A* **stimulates** *B*.

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

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

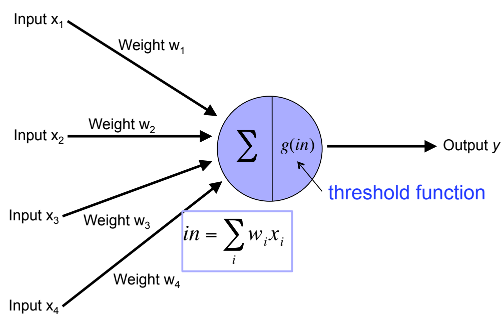
How much stimulation is required is determined by its **threshold**.

Neural Networks



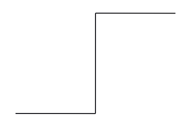
our approximation

A Single Neuron/Perceptron



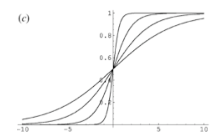
Possible threshold functions

- hard threshold
  - if *in* (the sum of weights) >= *threshold* 1, 0 otherwise

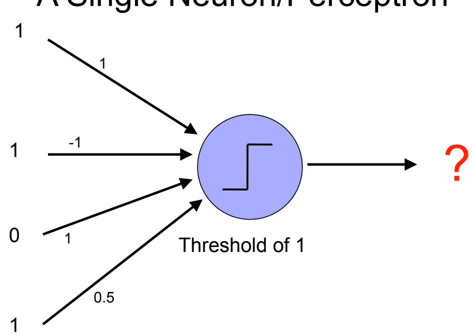


- Sigmoid

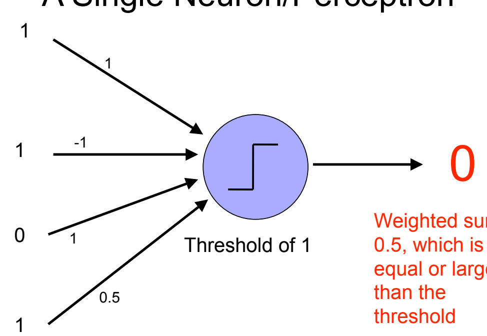
$$g(x) = \frac{1}{1+e^{-ax}}$$



A Single Neuron/Perceptron



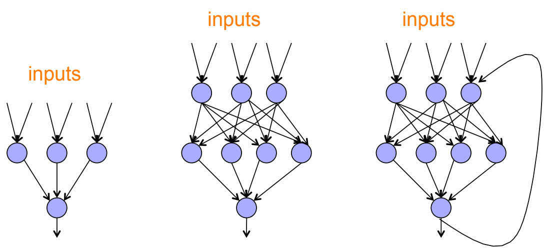
A Single Neuron/Perceptron



Weighted sum is 0.5, which is not equal or larger than the threshold

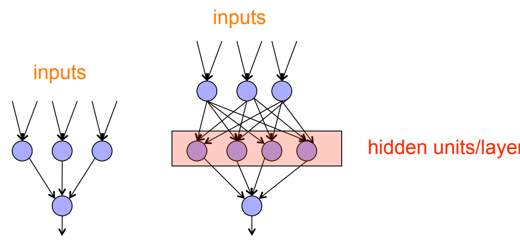
Neural networks

- Different kinds/characteristics of networks



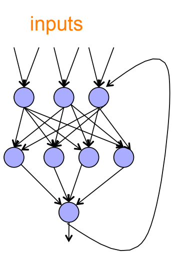
How are these different?

Neural networks



Feed forward networks (we'll mostly deal with these)

Neural networks



- Recurrent network
- Output is fed back to input
- Can support memory!
- How?

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

- First wave in neural networks in the 1960's
- Single neuron
- Trainable: 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

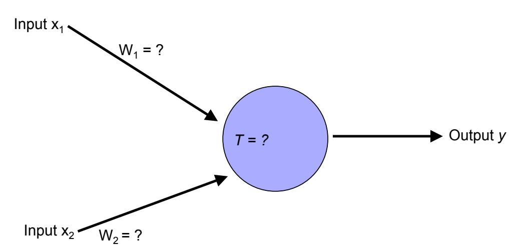
Examples - Logical operators

- **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

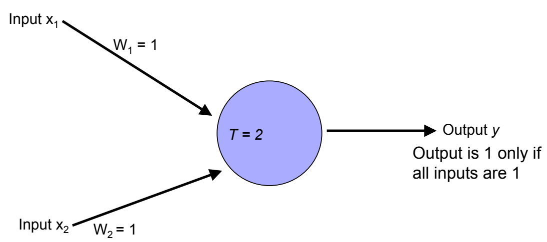
AND

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

AND

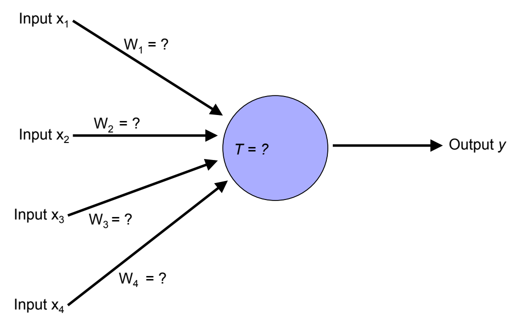


AND

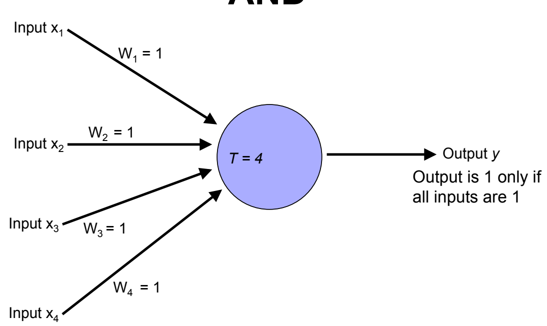


Inputs are either 0 or 1

AND



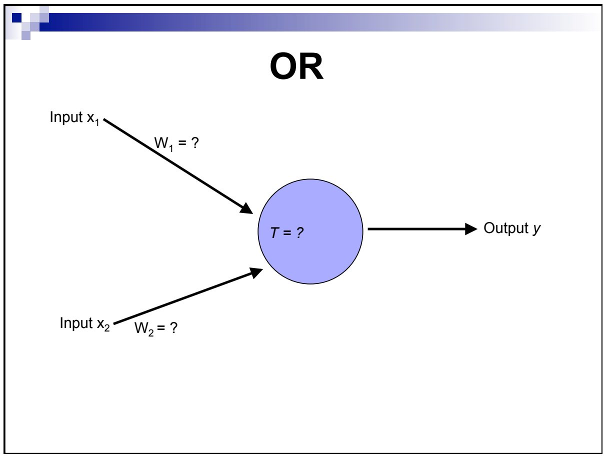
AND

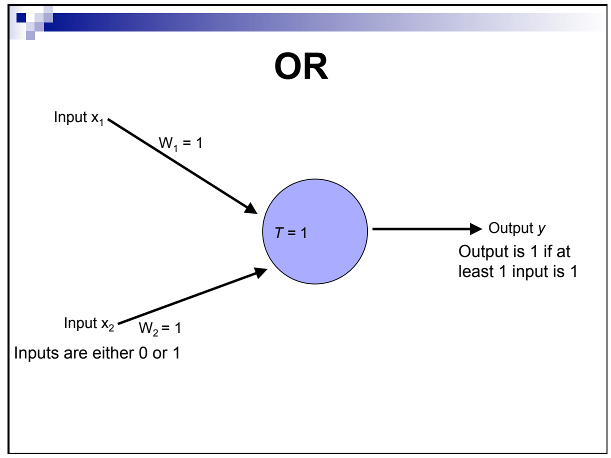


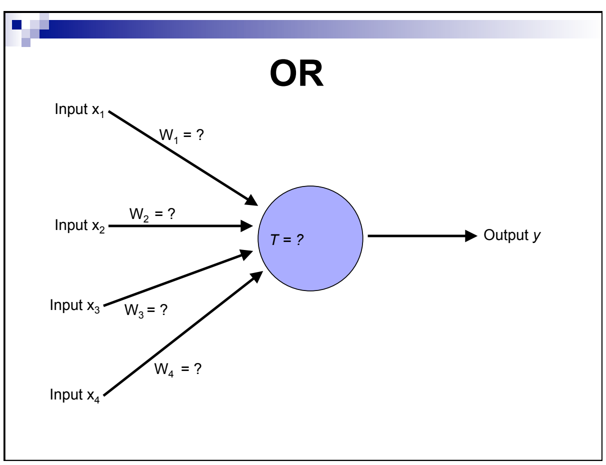
Inputs are either 0 or 1

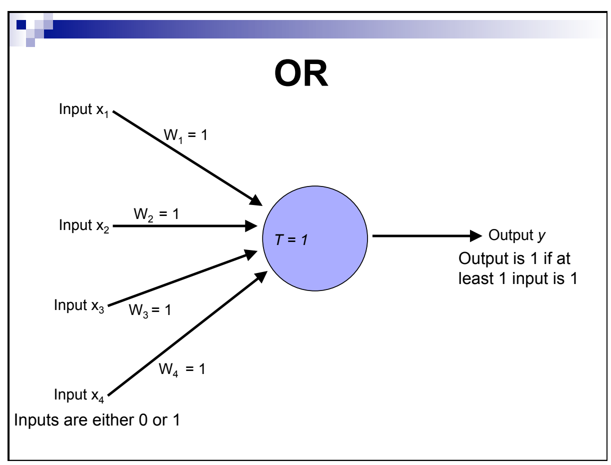
OR

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


OR
Input x1
W1 = ?
T = ?
Output y
Input x2
W2 = ?


OR
Input x1
W1 = 1
T = 1
Output y
Output is 1 if at least 1 input is 1
Input x2
W2 = 1
Inputs are either 0 or 1

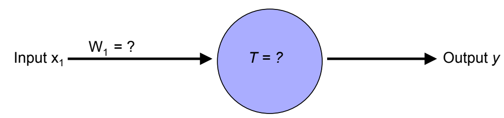

OR
Input x1
W1 = ?
Input x2
W2 = ?
T = ?
Output y
Input x3
W3 = ?
W4 = ?
Input x4


OR
Input x1
W1 = 1
Input x2
W2 = 1
T = 1
Output y
Output is 1 if at least 1 input is 1
Input x3
W3 = 1
W4 = 1
Input x4
Inputs are either 0 or 1

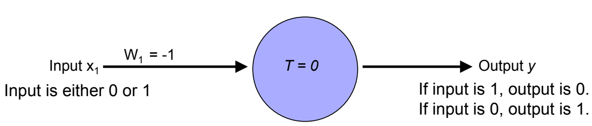
NOT

x_1	**not** x_1
0	1
1	0

NOT

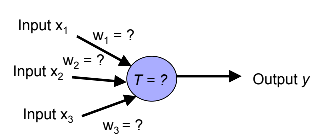


NOT

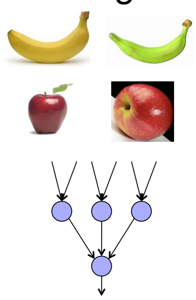


How about…

x_1	x_2	x_3	x_1 **and** x_2
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



Training neural nets

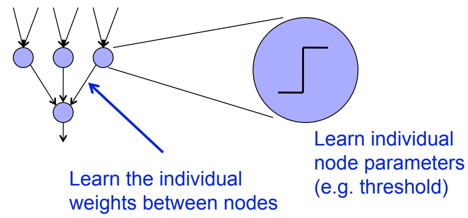


output: 1, -1

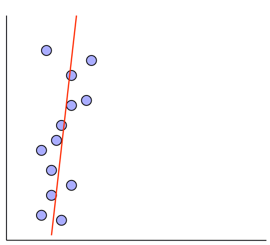
- We'd like to train neural networks
- We can learn to classify
- We can also learn a *regression function* from input to a real value

What are the parameters we can modify/learn for the NN?

NN parameters



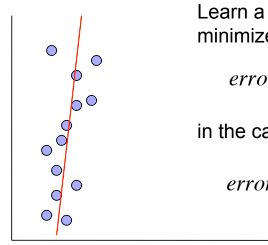
An aside: linear regression



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)

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

difference between actual and predicted

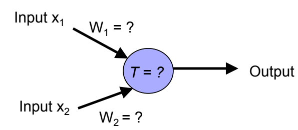
sum the error over all the examples

the value of the example in that dimension

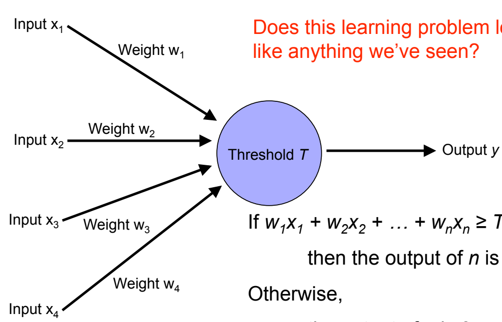
Back to training a perceptron

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



A Single Perceptron



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

If $w_1x_1 + w_2x_2 + \ldots + w_nx_n \geq T$,

then the output of n is 1.

Otherwise,

the output of n is 0.

Perceptron Training Rule

linear regression

- pick a random weight vector
- 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)

perceptron learning

- pick a random weight vector
- repeat until we correctly classify all the points:
  - pick an example
  - if we get it wrong:
    - modify the weights a small amount

Key difference: regression error vs. classification error

Perceptron Training Rule

linear regression

- pick a random weight vector
- 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)

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

perceptron learning

- pick a random weight vector
- repeat until we correctly classify all the points:
  - pick an example
  - if we get it wrong:
    - 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:

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

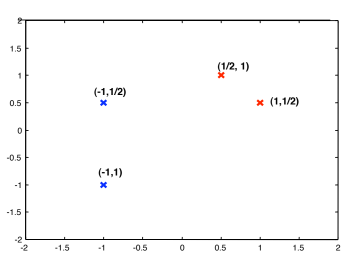
learning rate

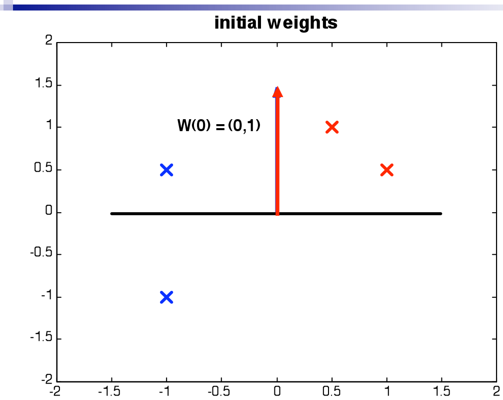
how much this feature played a role (e.g. active or not)

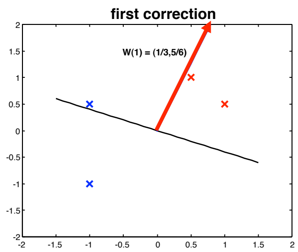
difference between actual and predicted

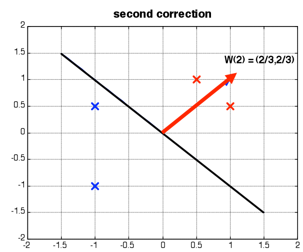
Example: a simple problem

4 points linearly separable









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 to exist:
  - Delta rule (Gradient Descent Approach)
  - Linear Programming

XOR

x_1	x_2	x_1 **xor** x_2
0	0	0
0	1	1
1	0	1
1	1	0

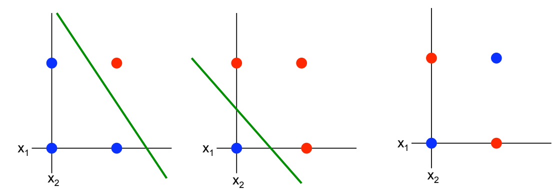
How would the perceptron do?

Linearly Separable

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

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

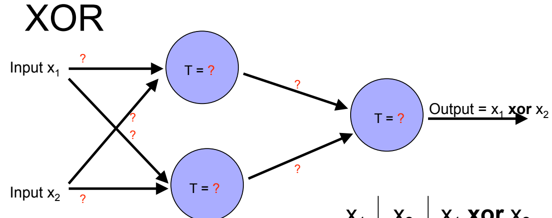
x_1	x_2	x_1 **xor** x_2
0	0	0
0	1	1
1	0	1
1	1	0



Perceptrons

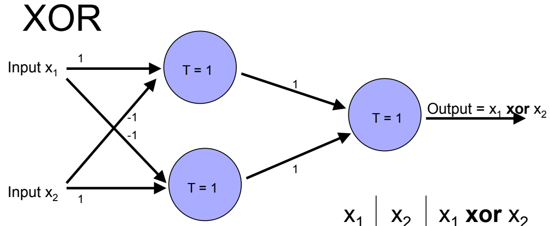
- 1969 book by Marvin Minsky and Seymour Papert
- The problem is that they can only work for classification problems that are linearly separable
- Insufficiently expressive
- "Important research problem" to investigate multilayer networks although they were pessimistic about their value

XOR



x_1	x_2	x_1 **xor** x_2
0	0	0
0	1	1
1	0	1
1	1	0

XOR



x_1	x_2	x_1 **xor** x_2
0	0	0
0	1	1
1	0	1
1	1	0

Logistic and other thresholds

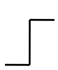
- Only update the weights when we get an example wrong

For each dimension *i* in the weight vector:

$$w_i = w_i - \alpha \underbrace{x_i\,(y_j - h(x))}_{\frac{d}{dw_i} error(w)}$$

Logistic and other thresholds

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



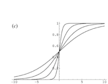
Any problem with using the threshold function?

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

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



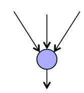
We'll use a sigmoid, which approximates a threshold but has a well defined derivative

Learning in Multilayer Networks

- Similar idea as perceptrons
- 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

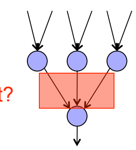
Learning in multilayer networks

- Key idea for perceptron learning: if the perceptron's output is different than the expected output, update the weights
- Challenge: for multilayer networks, we don't know what the expected output/error is for the internal nodes



perceptron

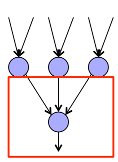
expected output?



multi-layer network

Backpropagation

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



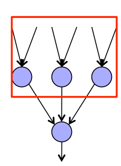
$$w_i = w_i - \alpha\, g'(w \cdot x_i)\,(y - h(x))\, x_i$$

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

the inputs to the output node

Backpropagation

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



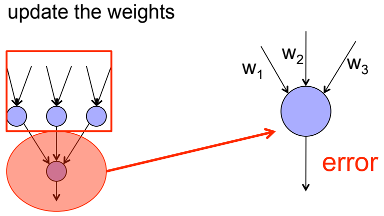
"back-propagate" the error:

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

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

Backpropagation

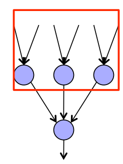
Say we get it wrong, and we now want to update the weights



error for node *i* is: w_i error

Backpropagation

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



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

$$w_i = w_i - \alpha\, g'(w \cdot a_i)\, error\; a_i$$

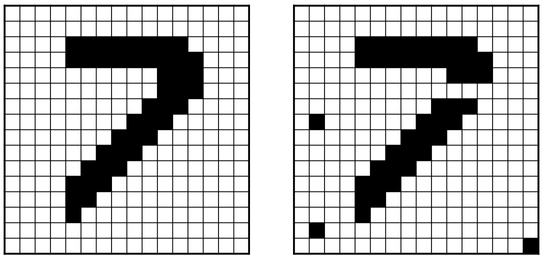
the nodes fraction of the error

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:
  - 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

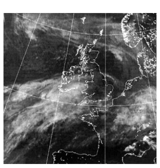
http://yann.lecun.com/exdb/mnist/

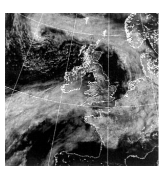
Cog sci people like NNs

- Expression/emotion recognition
  - Gary Cottrell et al

- Language learning

Interpreting Satellite Imagery for Automated Weather Forecasting





Summary

- Perceptrons, one layer networks, are insufficiently expressive
- Multi-layer networks are sufficiently expressive and can be trained by error back-propogation
- Many applications including speech, driving, hand written character recognition, fraud detection, driving, etc.