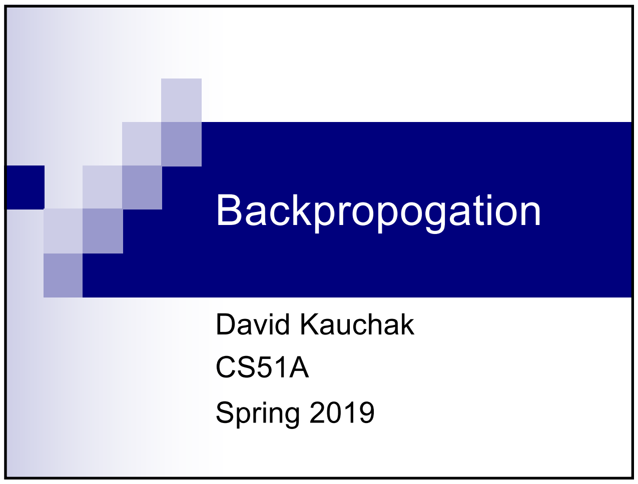
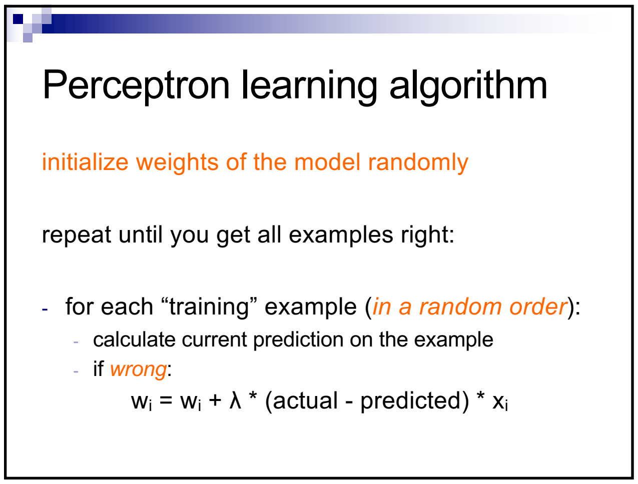


Backpropogation

David Kauchak
CS51A
Spring 2019



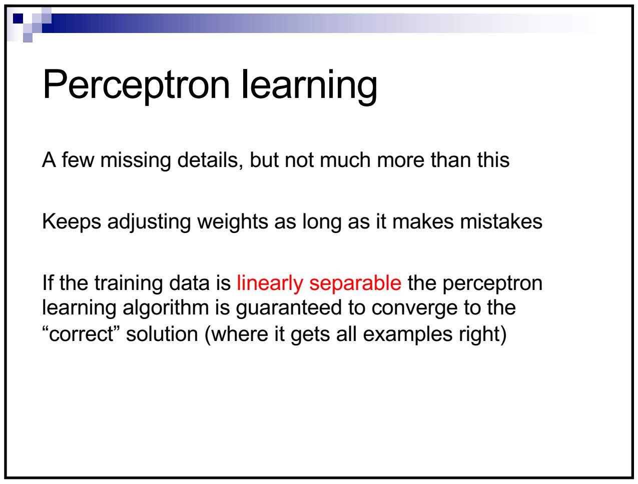
Perceptron learning algorithm

initialize weights of the model randomly

repeat until you get all examples right:

- for each "training" example (*in a random order*):
  - calculate current prediction on the example
  - if *wrong*:

    $w_i = w_i + \lambda * (\text{actual - predicted}) * x_i$

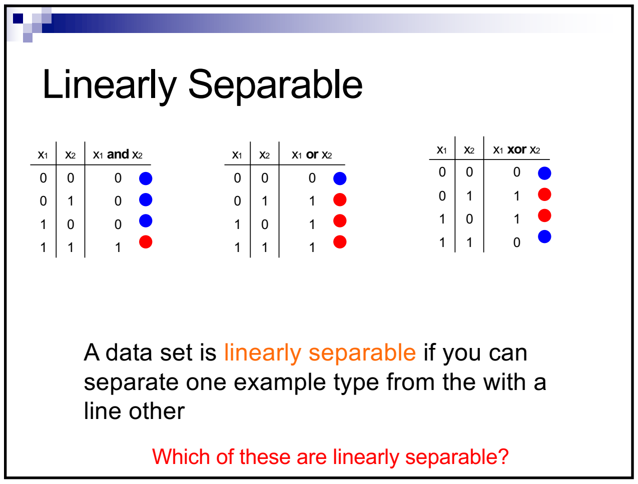


Perceptron learning

A few missing details, but not much more than this

Keeps adjusting weights as long as it makes mistakes

If the training data is linearly separable the perceptron learning algorithm is guaranteed to converge to the "correct" solution (where it gets all examples right)



Linearly Separable

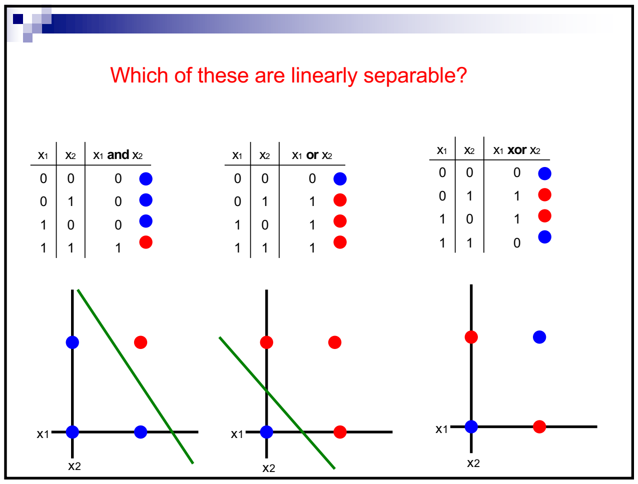
x_1	x_2	x_1 and x_2	
0	0	0	🔵
0	1	0	🔵
1	0	0	🔵
1	1	1	🔴

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	🔵

A data set is linearly separable if you can separate one example type from the with a line other

Which of these are linearly separable?

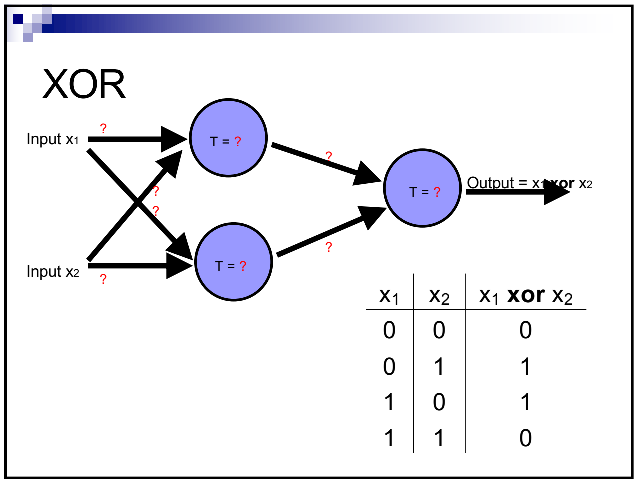


Which of these are linearly separable?

x_1	x_2	x_1 **and** x_2	
0	0	0	🔵
0	1	0	🔵
1	0	0	🔵
1	1	1	🔴

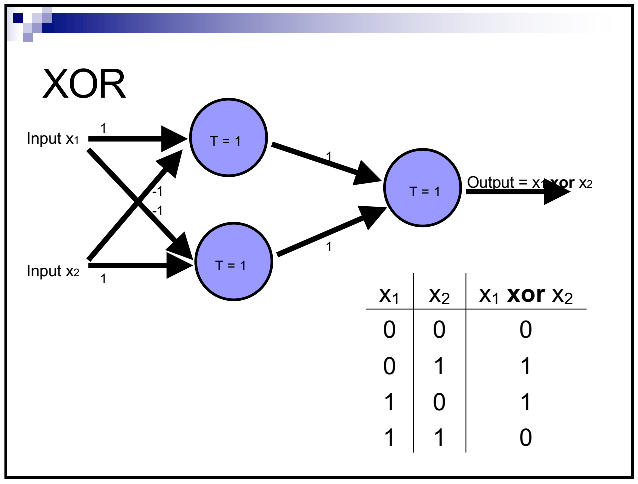
x_1	x_2	x_1 **or** x_2	
0	0	0	🔵
0	1	1	🔴
1	0	1	🔴
1	1	1	🔴

x_1	x_2	x_1 **xor** x_2	
0	0	0	🔵
0	1	1	🔴
1	0	1	🔴
1	1	0	🔵



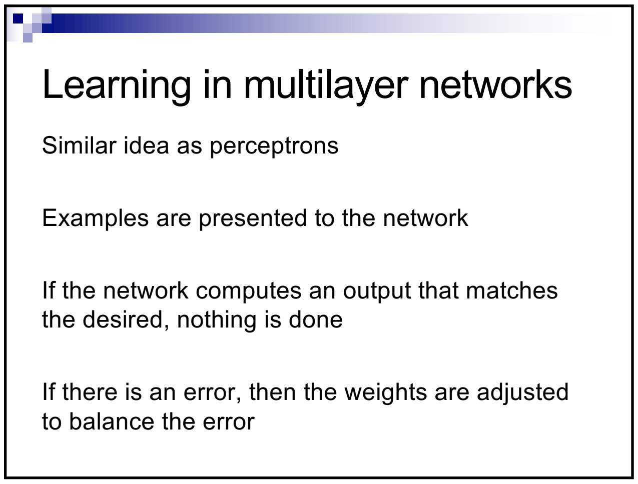
XOR

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



Learning in multilayer networks

Similar idea as perceptrons

Examples are presented to the network

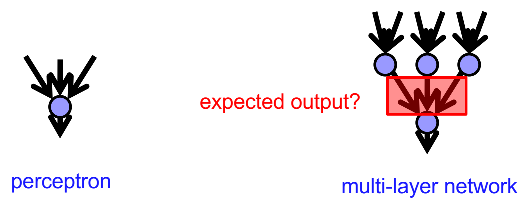
If the network computes an output that matches the desired, nothing is done

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

Learning in multilayer networks

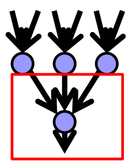
Key idea for perceptron learning: if the perceptron's output is different than the expected output, update the weights

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



Backpropagation

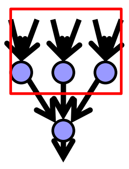
Say we get it wrong, and we now want to update the weights



We can update this layer just as if it were a perceptron

Backpropagation

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



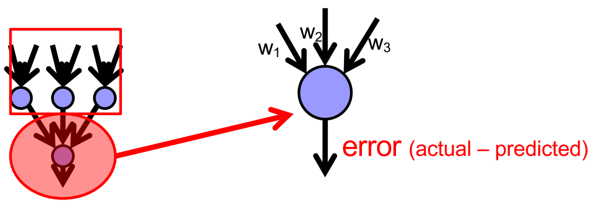
"back-propagate" the error (actual – predicted):

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

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

Backpropagation

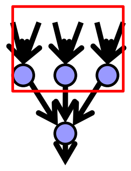
Say we get it wrong, and we now want to update the weights



error for node *i* is: w_i error

Backpropagation

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



Update these weights and continue the process back through the network

Backpropagation

calculate the error at the output layer

backpropagate the error up the network

Update the weights based on these errors

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

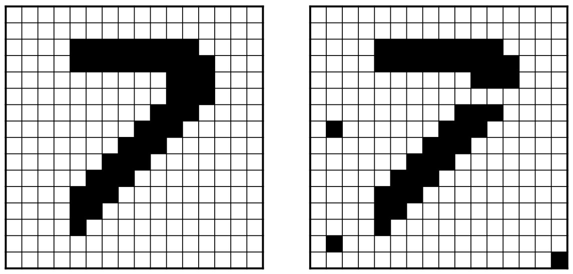
That said, many neuroscientists don't think the brain does backpropagation of errors

Neural network regression

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

Challenges:

- overfitting – learning only the training data and not learning to generalize
- picking a network structure
- can require a lot of tweaking of parameters, preprocessing, etc.



Popular for digit recognition and many computer vision tasks

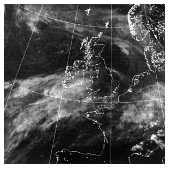
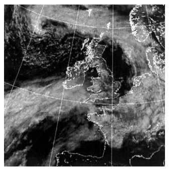
http://yann.lecun.com/exdb/mnist/

Cog sci people like NNs

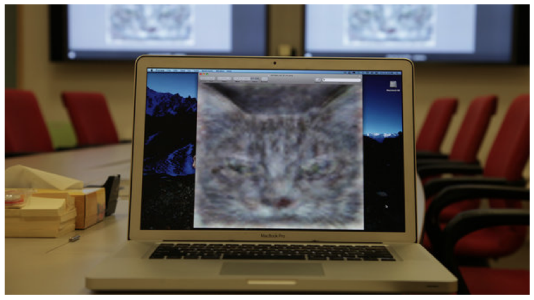
Expression/emotion recognition
- Gary Cottrell et al

Language learning

Interpreting Satellite Imagery for Automated Weather Forecasting

What NNs learned from youtube



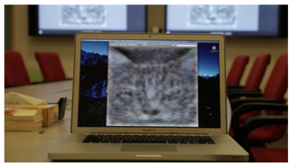
http://www.nytimes.com/2012/06/26/technology/in-a-big-network-of-computers-evidence-of-machine-learning.html

What NNs learned from youtube

trained on 10M snapshots from youtube videos

NN with 1 billion connections

16,000 processors



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.

Our python NN module

Data:

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

```
table = \
[ ([0.0, 0.0, 0.0], [1.0]),
  ([0.0, 1.0, 0.0], [0.0]),
  ([1.0, 0.0, 0.0], [1.0]),
  ([1.0, 1.0, 0.0], [0.0]),
  ([0.0, 0.0, 1.0], [1.0]),
  ([0.0, 1.0, 1.0], [1.0]),
  ([1.0, 0.0, 1.0], [1.0]),
  ([1.0, 1.0, 1.0], [0.0]) ]
```

Data format

list of examples

```
table = \
[ ([0.0, 0.0, 0.0], [1.0]),
  ([0.0, 1.0, 0.0], [0.0]),
  ([1.0, 0.0, 0.0], [1.0]),
  ([1.0, 1.0, 0.0], [0.0]),
  ([0.0, 0.0, 1.0], [1.0]),
  ([0.0, 1.0, 1.0], [1.0]),
  ([1.0, 0.0, 1.0], [1.0]),
  ([1.0, 1.0, 1.0], [0.0]) ]
```

([0.0, 0.0, 0.0], [1.0])

input list output list

example = tuple

Training on the data

Construct a new network:

```
>>> nn = NeuralNet(3, 2, 1)
```

constructor: constructs a new NN object

input nodes

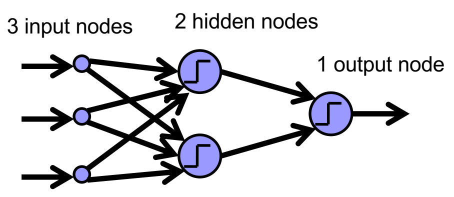
hidden nodes

output nodes

Training on the data

Construct a new network:

```
>>> nn = NeuralNet(3, 2, 1)
```



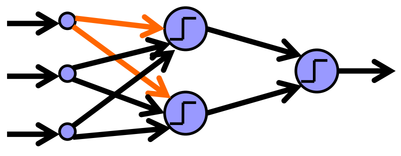
Training on the data

```
>>> nn.train(table)
error 0.195200
error 0.062292
error 0.031077
error 0.019437
error 0.013728
error 0.010437
error 0.008332
error 0.006885
error 0.005837
error 0.005047
```

by default trains 1000 iteration and prints out error values every 100 iterations

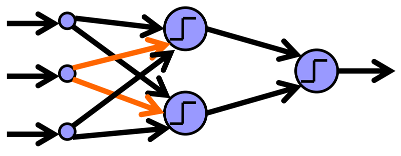
After training, can look at the weights

```
>>> nn.train(table)
>>> nn.getIHWeights()
[[-3.3435628797862624, -0.272324373735495],
 [-4.846203738642956, -4.601230952566068],
 [3.4233831101145973, 0.573534695637572],
 [2.9388429644152128, 1.8509761272713543]]
```



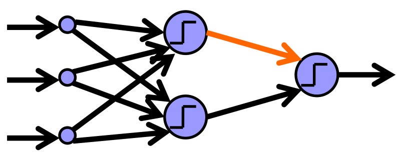
After training, can look at the weights

```
>>> nn.train(table)
>>> nn.getIHWeights()
[[-3.3435628797862624, -0.272324373735495],
 [-4.846203738642956, -4.601230952566068],
 [3.4233831101145973, 0.573534695637572],
 [2.9388429644152128, 1.8509761272713543]]
```

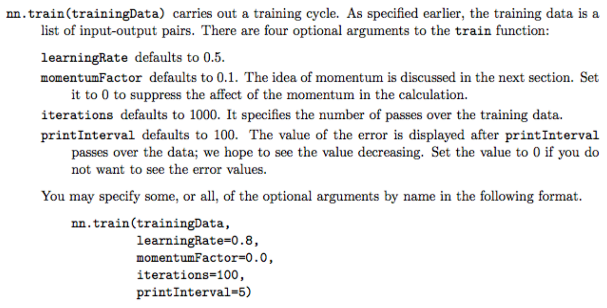


After training, can look at the weights

```
>>> nn.getHOWeights()
[[8.116192424400454],
 [5.358094903107918],
 [-4.373829543609533]]
```



Many parameters to play with



`nn.train(trainingData)` carries out a training cycle. As specified earlier, the training data is a list of input-output pairs. There are four optional arguments to the `train` function:

`learningRate` defaults to 0.5.

`momentumFactor` defaults to 0.1. The idea of momentum is discussed in the next section. Set it to 0 to suppress the affect of the momentum in the calculation.

`iterations` defaults to 1000. It specifies the number of passes over the training data.

`printInterval` defaults to 100. The value of the error is displayed after `printInterval` passes over the data; we hope to see the value decreasing. Set the value to 0 if you do not want to see the error values.

You may specify some, or all, of the optional arguments by name in the following format.

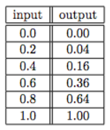
```
nn.train(trainingData,
         learningRate=0.8,
         momentumFactor=0.0,
         iterations=100,
         printInterval=5)
```

Calling with optional parameters

```
>>> nn.train(table, iterations = 5, printInterval = 1)
error 0.005033
error 0.005026
error 0.005019
error 0.005012
error 0.005005
```

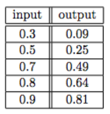
Train vs. test

TrainData



input	output
0.0	0.00
0.2	0.04
0.4	0.16
0.6	0.36
0.8	0.64
1.0	1.00

TestData



input	output
0.3	0.09
0.5	0.25
0.7	0.49
0.8	0.64
0.9	0.81

```
>>> nn.train(trainData)
>>> nn.test(testData)
```

http://www.sciencebytes.org/2011/05/03/bl
ueprint-for-the-brain/