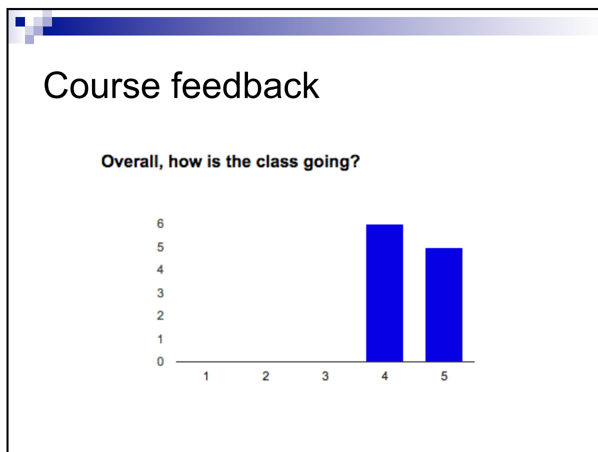
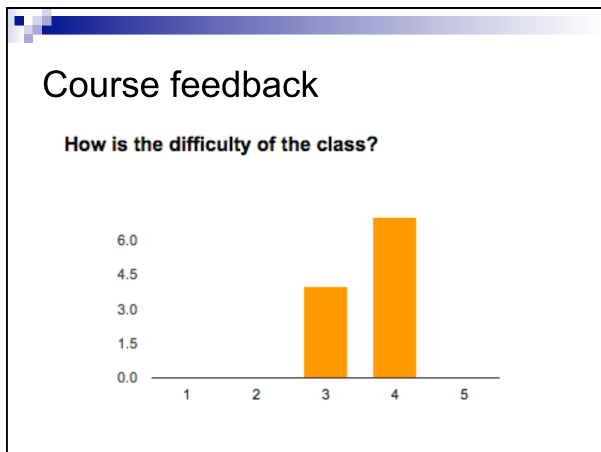
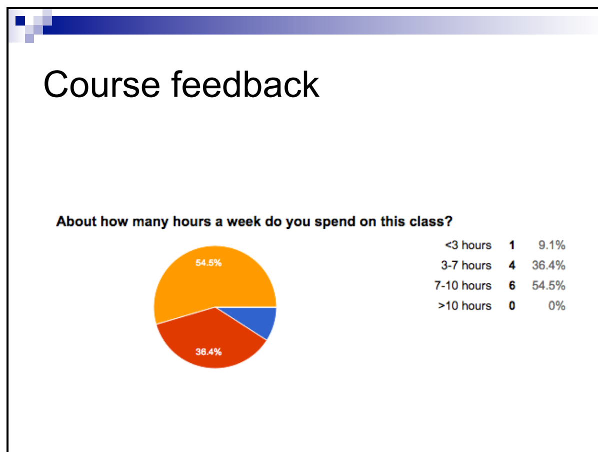
Neural Networks 2

David Kauchak
CS30
Spring 2016







Course feedback

I like thinking in a way that is opposite from what I normally do in humanities courses. I enjoy the creativity of getting to the end goal by means that may be the same or different from the other people in the class.

Course feedback

we create something that is somewhat useful! somewhat understand how things / programs on our computers work

Course feedback

I think I would benefit more if you could break down the assignments. Rather than having one large weekly assignment, having it due by parts (like assignment 5 is) can help me stay on track and keep it more manageable.

Course feedback

Somehow increase the use of Piazza. The mentor sessions don't work well with my schedule, and I wish I could see what people are asking about.

Course feedback

It seems like it would be helpful for the mentors to have the solutions before lab because often they aren't sure what the correct answer is and spend a significant amount of time trying to figure it out themselves.

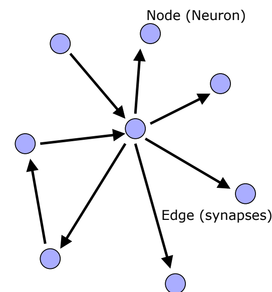
Course feedback

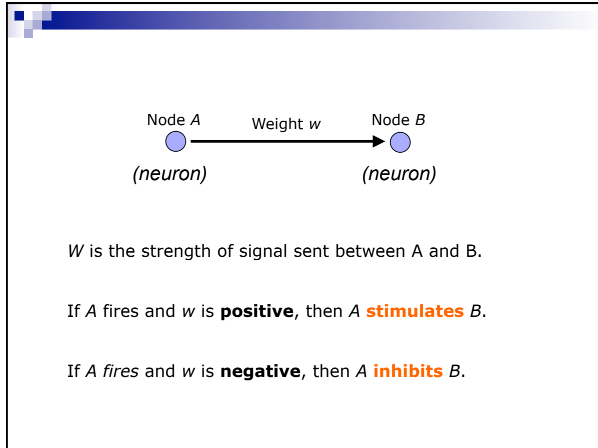
More info of how coding is done in the real world and it's applications and purposes...however I'm guessing that that will be covered in the weeks to come.

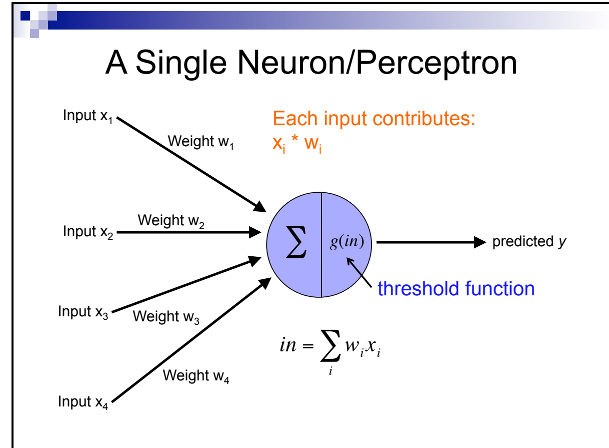
Optional parameters

- Look at optional_parameters.py

Artificial Neural Networks







Training neural networks

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

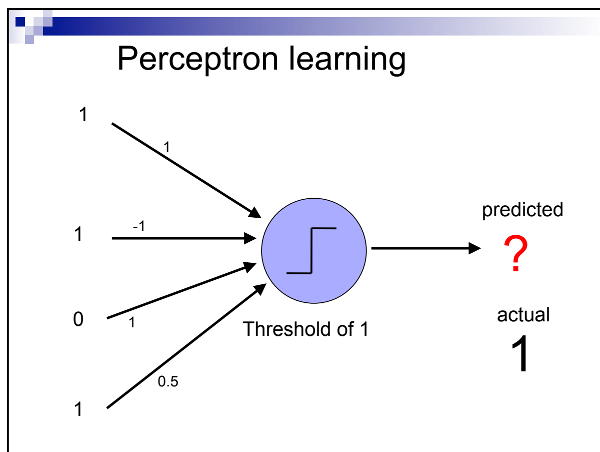
Input x_1 — w_1 = ?
Input x_2 — w_2 = ?
Input x_3 — w_3 = ?
T = ?
predicted *y*

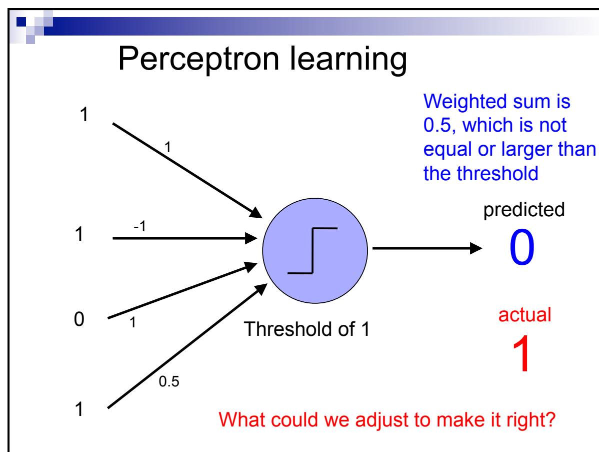
1. start with some initial weights and thresholds
2. show examples repeatedly to NN
3. update weights/thresholds by comparing NN predicted to actual predicted

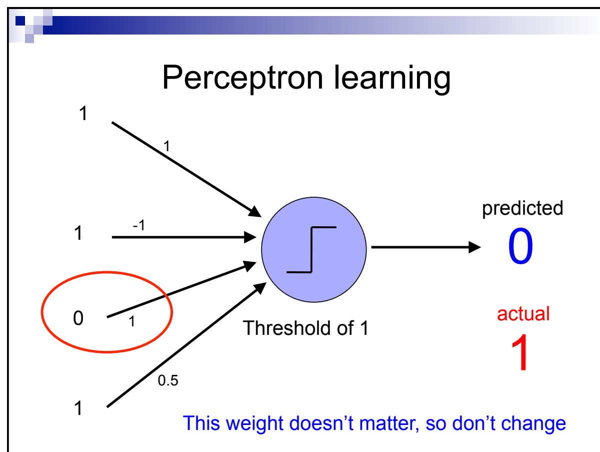
Perceptron learning algorithm

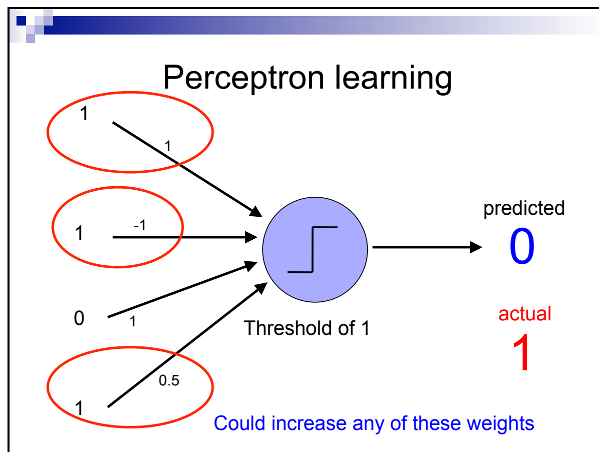
repeat until you get all examples right:

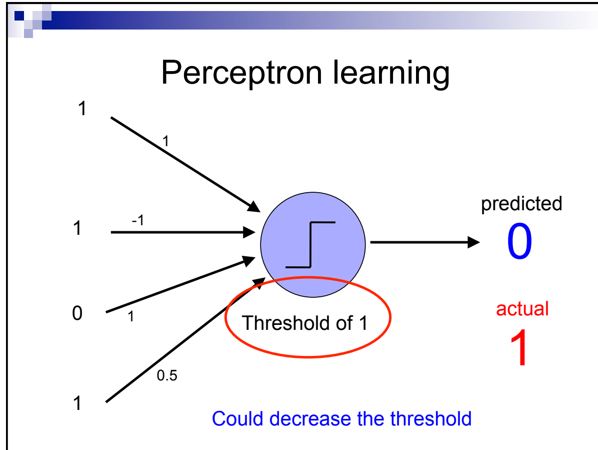
- for each "training" example:
  - calculate current prediction on example
  - if *wrong*:
    - update weights and threshold towards getting this example correct

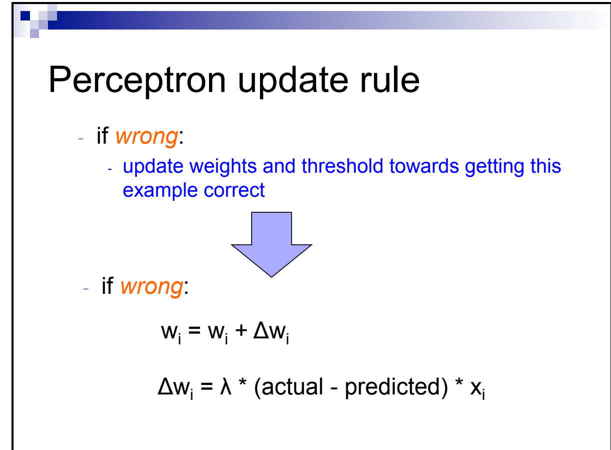

Perceptron learning
1
1
1
-1
0
1
1
0.5
Threshold of 1
predicted
?
actual
1

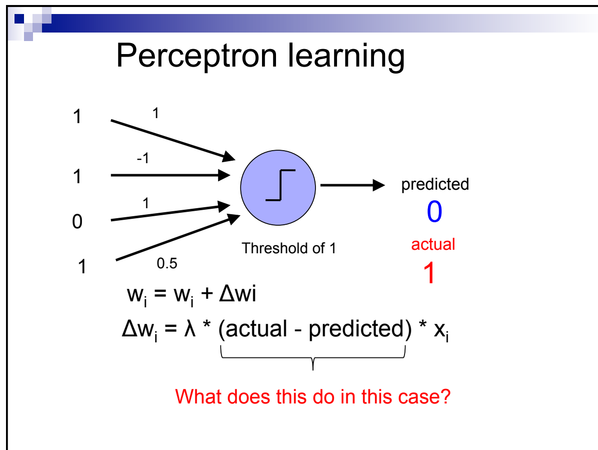

Perceptron learning
Weighted sum is 0.5, which is not equal or larger than the threshold
1
1
1
-1
0
1
1
0.5
Threshold of 1
predicted
0
actual
1
What could we adjust to make it right?

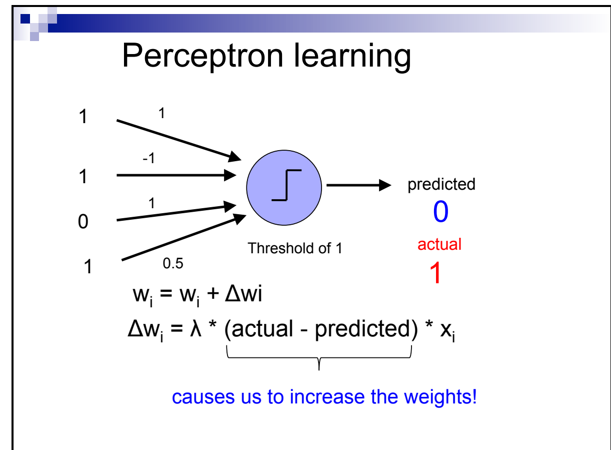

Perceptron learning
1
1
1
-1
0
1
1
0.5
Threshold of 1
predicted
0
actual
1
This weight doesn't matter, so don't change


Perceptron learning
1
1
1
-1
0
1
1
0.5
Threshold of 1
predicted
0
actual
1
Could increase any of these weights


Perceptron learning
1
1
1
-1
0
1
1
0.5
predicted
0
actual
1
Threshold of 1
Could decrease the threshold


Perceptron update rule
if wrong:
update weights and threshold towards getting this example correct
if wrong:
$w_i = w_i + \Delta w_i$
$\Delta w_i = \lambda * (\text{actual} - \text{predicted}) * x_i$


Perceptron learning
1
1
1
-1
0
1
1
0.5
predicted
0
actual
1
Threshold of 1
$w_i = w_i + \Delta wi$
$\Delta w_i = \lambda * (\text{actual} - \text{predicted}) * x_i$
What does this do in this case?


Perceptron learning
1
1
1
-1
0
1
1
0.5
predicted
0
actual
1
Threshold of 1
$w_i = w_i + \Delta wi$
$\Delta w_i = \lambda * (\text{actual} - \text{predicted}) * x_i$
causes us to increase the weights!

Perceptron learning

Threshold of 1

predicted
1
actual
0

$w_i = w_i + \Delta wi$

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

What if predicted = 1 and actual = 0?

Perceptron learning

Threshold of 1

predicted
1
actual
0

$w_i = w_i + \Delta wi$

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

We're over the threshold, so want to decrease weights:
actual - predicted = -1

Perceptron learning

Inputs: 1, 1, 0, 1
Weights: 1, -1, 1, 0.5

Threshold of 1

predicted
0
actual
1

$w_i = w_i + \Delta wi$

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

What does this do?

Perceptron learning

Inputs: 1, 1, 0, 1
Weights: 1, -1, 1, 0.5

Threshold of 1

predicted
0
actual
1

$w_i = w_i + \Delta wi$

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

Only adjust those weights that actually contributed!

Perceptron learning

$w_i = w_i + \Delta wi$

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

What does this do?

Perceptron learning

$w_i = w_i + \Delta wi$

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

"learning rate": value between 0 and 1 (e.g 0.1) adjusts how abrupt the changes are to the model

Perceptron learning

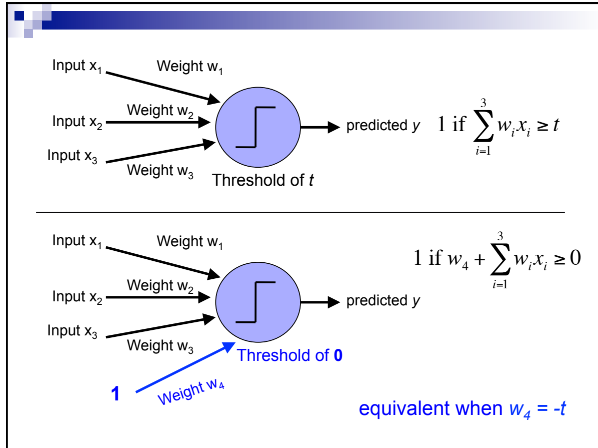
$w_i = w_i + \Delta wi$

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

What about the threshold?

$$1 \text{ if } \sum_{i=1}^{3} w_i x_i \geq t$$

$$1 \text{ if } w_4 + \sum_{i=1}^{3} w_i x_i \geq 0$$



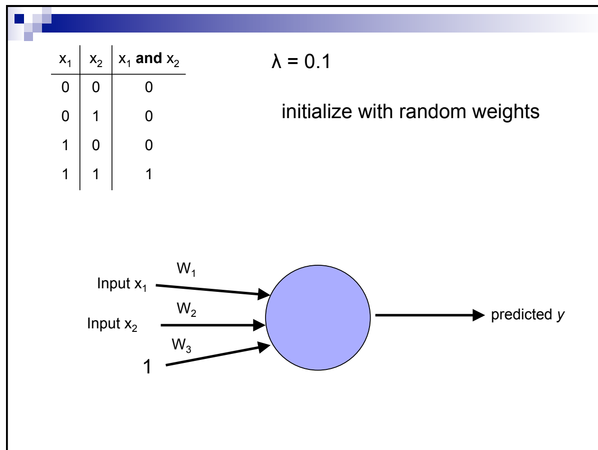
Perceptron learning algorithm

initialize weights of the model randomly

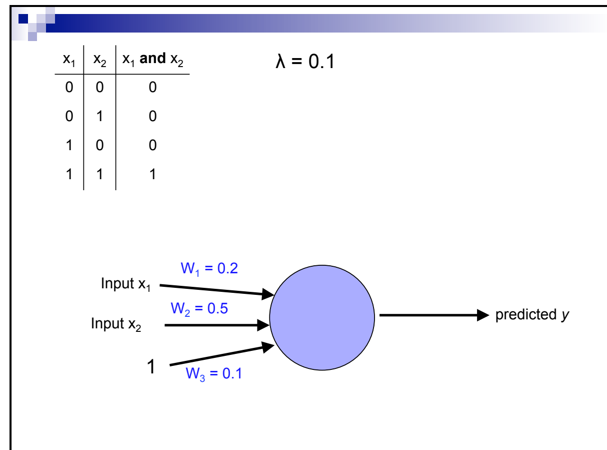
repeat until you get all examples right:

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

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



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

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

$\lambda = 0.1$

if wrong:

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

1 $W_1 = 0.2$

0 $W_2 = 0.5$

1 $W_3 = 0.1$

predicted *y*

Right or wrong?

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

$\lambda = 0.1$

if wrong:

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

1 $W_1 = 0.2$

0 $W_2 = 0.5$

1 $W_3 = 0.1$

sum = 0.3: predicted 1

predicted *y*

Wrong

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

$\lambda = 0.1$

if wrong:

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

1 $W_1 = 0.2$

0 $W_2 = 0.5$

1 $W_3 = 0.1$

sum = 0.3: predicted 1

predicted *y*

new weights?

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

$\lambda = 0.1$

if wrong:

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

1 $W_1 = 0.1$

0 $W_2 = 0.5$

1 $W_3 = 0.0$

sum = 0.3: predicted 1

predicted *y*

decrease (0-1=-1) all non-zero x_i by 0.1

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

$\lambda = 0.1$

if wrong:

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

Right or wrong?

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

$\lambda = 0.1$

if wrong:

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

Right. No update!

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

$\lambda = 0.1$

if wrong:

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

Right or wrong?

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

$\lambda = 0.1$

if wrong:

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

Wrong

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

$\lambda = 0.1$

if wrong:

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

Inputs: 0, 1, 1; $W_1 = 0.1$, $W_2 = 0.5$, $W_3 = 0.0$

sum = 0.5: predicted 1

predicted *y*

new weights?

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

$\lambda = 0.1$

if wrong:

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

Inputs: 0, 1, 1; $W_1 = 0.1$, $W_2 = 0.4$, $W_3 = -0.1$

sum = 0.5: predicted 1

predicted *y*

decrease (0-1=-1) all non-zero x_i by 0.1

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

$\lambda = 0.1$

if wrong:

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

Inputs: 0, 0, 1; $W_1 = 0.1$, $W_2 = 0.4$, $W_3 = -0.1$

predicted *y*

Right or wrong?

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

$\lambda = 0.1$

if wrong:

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

Inputs: 0, 0, 1; $W_1 = 0.1$, $W_2 = 0.4$, $W_3 = -0.1$

sum = -0.1: predicted 0

predicted *y*

Right. No update!

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

$\lambda = 0.1$

if wrong:

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

0 — $W_1 = 0.1$
1 — $W_2 = 0.4$
1 — $W_3 = -0.1$

predicted y

Right or wrong?

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

$\lambda = 0.1$

if wrong:

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

0 — $W_1 = 0.1$
1 — $W_2 = 0.4$
1 — $W_3 = -0.1$

sum = 0.3: predicted 1

predicted y

Wrong

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

$\lambda = 0.1$

if wrong:

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

0 — $W_1 = 0.1$
1 — $W_2 = 0.3$
1 — $W_3 = -0.2$

sum = 0.3: predicted 1

predicted y

decrease (0-1=-1) all non-zero x_i by 0.1

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

$\lambda = 0.1$

if wrong:

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

1 — $W_1 = 0.1$
1 — $W_2 = 0.3$
1 — $W_3 = -0.2$

sum = 0.2: predicted 1

predicted y

Right. No update!

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

$\lambda = 0.1$

if wrong:

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

Inputs: 0, 0, 1; $W_1 = 0.1$, $W_2 = 0.3$, $W_3 = -0.2$

sum = -0.2: predicted 0

predicted *y*

Right. No update!

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

$\lambda = 0.1$

if wrong:

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

Inputs: 1, 0, 1; $W_1 = 0.1$, $W_2 = 0.3$, $W_3 = -0.2$

sum = -0.1: predicted 0

predicted *y*

Right. No update!

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

$\lambda = 0.1$

if wrong:

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

Inputs: 1, 0, 1; $W_1 = 0.1$, $W_2 = 0.3$, $W_3 = -0.2$

predicted *y*

Are they all right?

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

$\lambda = 0.1$

if wrong:

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

Inputs: 0, 1, 1; $W_1 = 0.1$, $W_2 = 0.3$, $W_3 = -0.2$

sum = 0.1: predicted 1

predicted *y*

Wrong

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

$\lambda = 0.1$

if wrong:

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

Inputs: 0, 1, 1

$W_1 = 0.1$

$W_2 = 0.2$

$W_3 = -0.3$

sum = 0.1: predicted 1

predicted *y*

decrease (0-1=-1) all non-zero x_i by 0.1

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

$\lambda = 0.1$

if wrong:

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

Inputs: 1, 0, 1

$W_1 = 0.1$

$W_2 = 0.2$

$W_3 = -0.3$

sum = 0.1: predicted 1

predicted *y*

Are they all right?

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

$\lambda = 0.1$

if wrong:

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

Inputs: 1, 1, 1

$W_1 = 0.1$

$W_2 = 0.2$

$W_3 = -0.3$

predicted *y*

We've learned AND!

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

x_1 x_2 x_1 x_2 x_1 x_2

XOR

Input x_1 ? ? T = ? ?

Input x_2 ? ? T = ? ?

T = ? predicted = x_1 **xor** x_2

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

XOR

Input x_1 1 -1 T = 1 1

Input x_2 1 -1 T = 1 1

T = 1 predicted = x_1 **xor** x_2

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

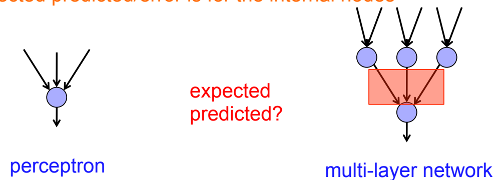
If the network computes an predicted that matches the desired, nothing is done

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

Learning in multilayer networks

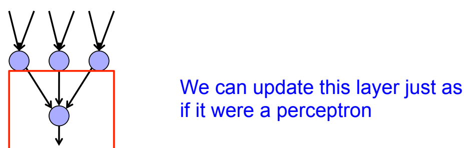
Key idea for perceptron learning: if the perceptron's predicted is different than the expected predicted, update the weights

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



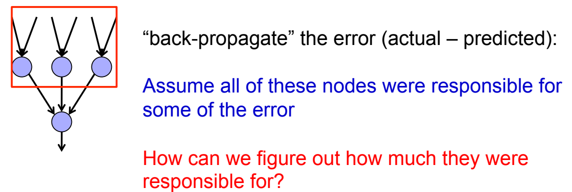
Backpropagation

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



Backpropagation

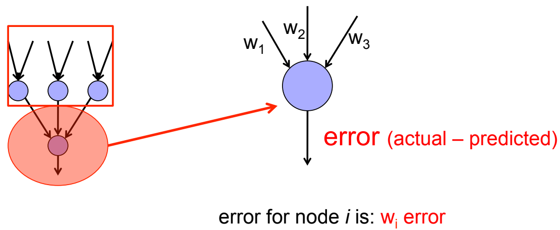
Say we get it wrong, and we now want to update the weights



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

Backpropagation

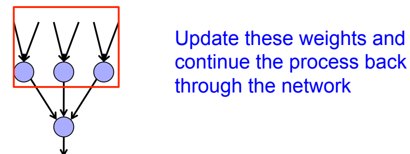
Say we get it wrong, and we now want to update the weights



error for node *i* is: w_i error

Backpropagation

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



Backpropagation

calculate the error at the predicted 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

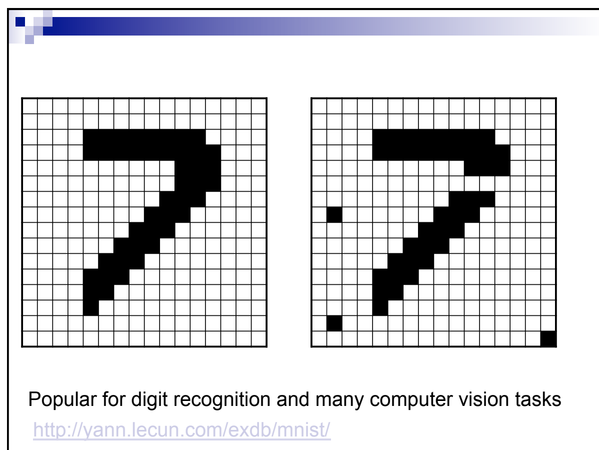
That said, many neuroscientists don't think the brain does backpropagation of errors

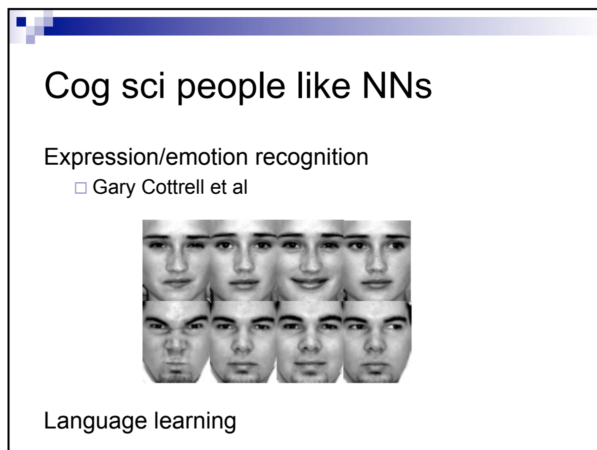
Neural network regression

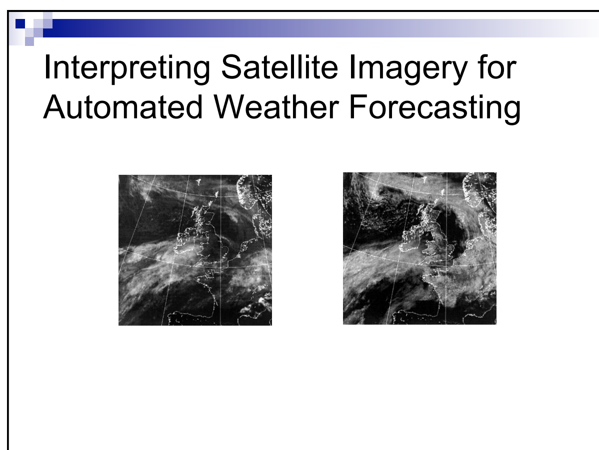
Given enough hidden nodes, you can learn *any* function with a neural network

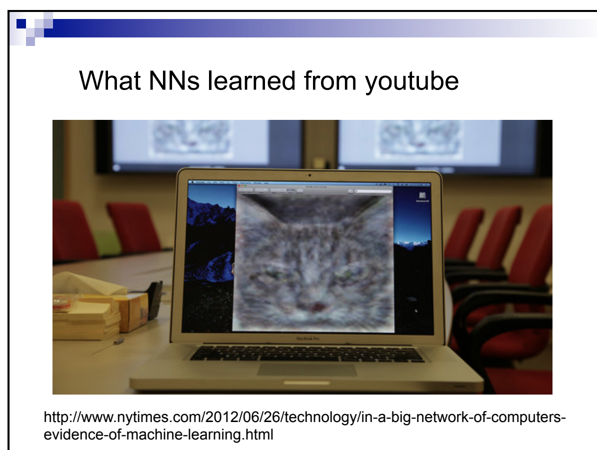
Challenges:

- 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

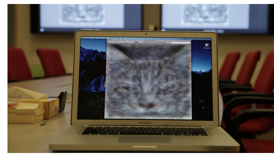

What NNs learned from youtube
http://www.nytimes.com/2012/06/26/technology/in-a-big-network-of-computers-evidence-of-machine-learning.html

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

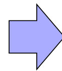
Multi-layer networks are sufficiently expressive and can be trained by error back-propogation

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 — predicted list

example = tuple

Training on the data

Construct a new network:

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


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
```

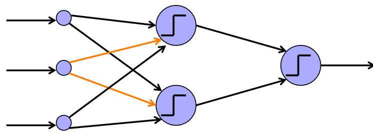
by default trains 1000 iteration and prints out error values every 100 iterations

After training, can look at the weights

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

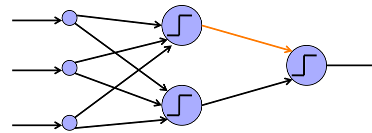
After training, can look at the weights

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



After training, can look at the weights

```
>>> nn.get_HO_weights()
[[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

train_data

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

test_data

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

```
>>> nn.train(train_data)
>>> nn.test(test_data)
```

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