



Backpropogation

Dave Kauchak

Alexandra Papoutsaki

Zilong Ye

CSCI 051A

Spring 2022

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*:

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



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 other with a line.

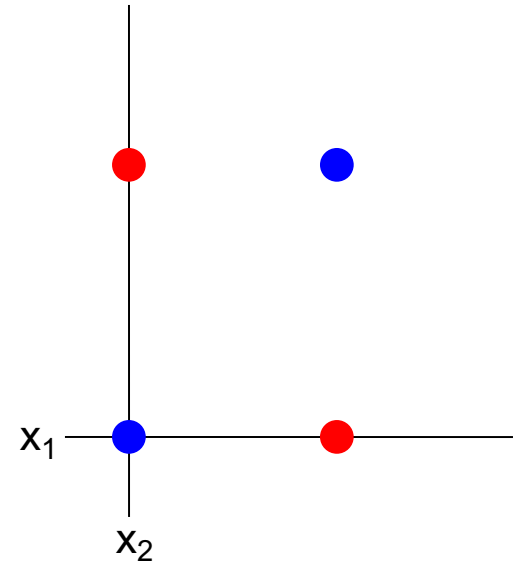
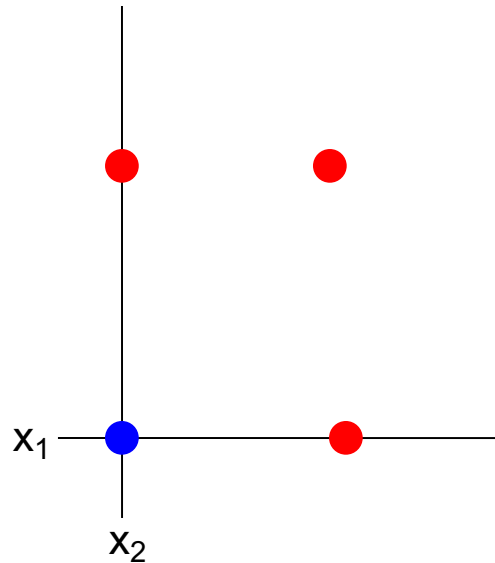
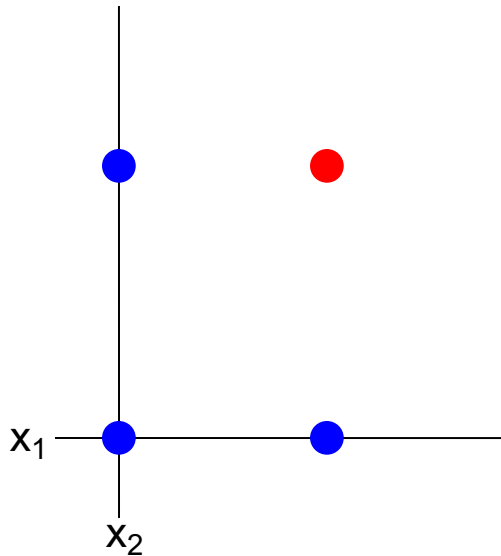
Which of these are linearly separable?

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

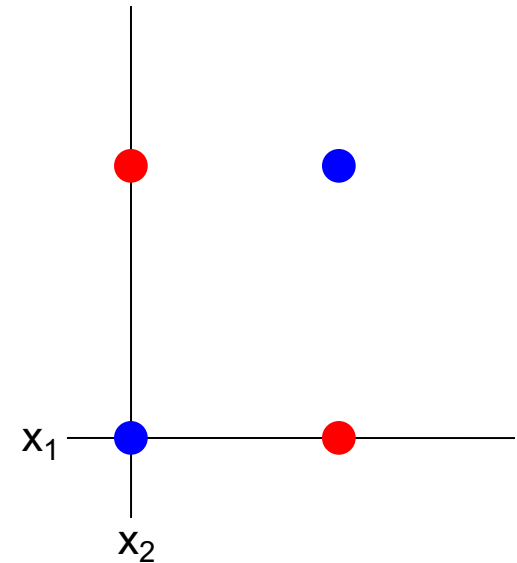
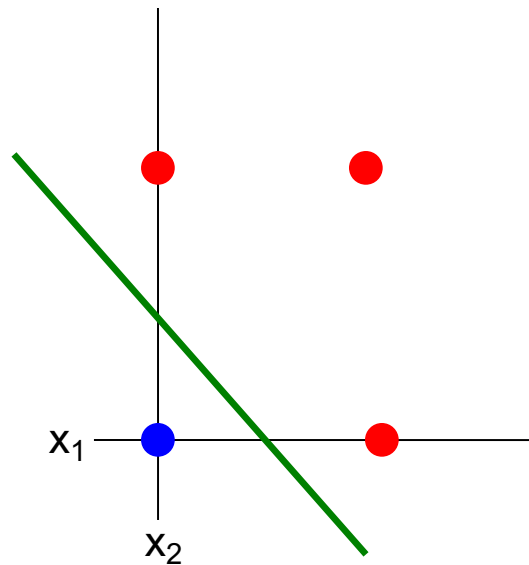
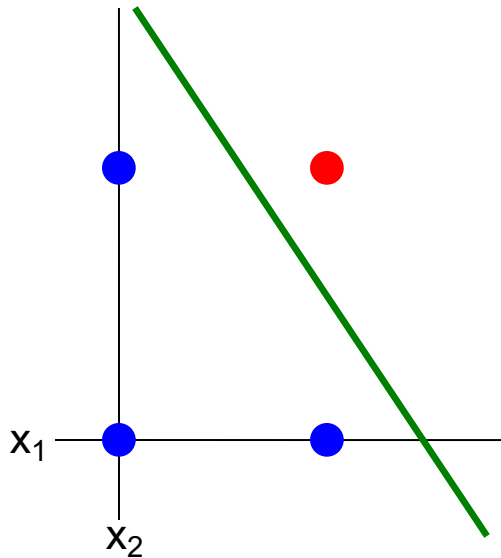


Which of these are linearly separable?

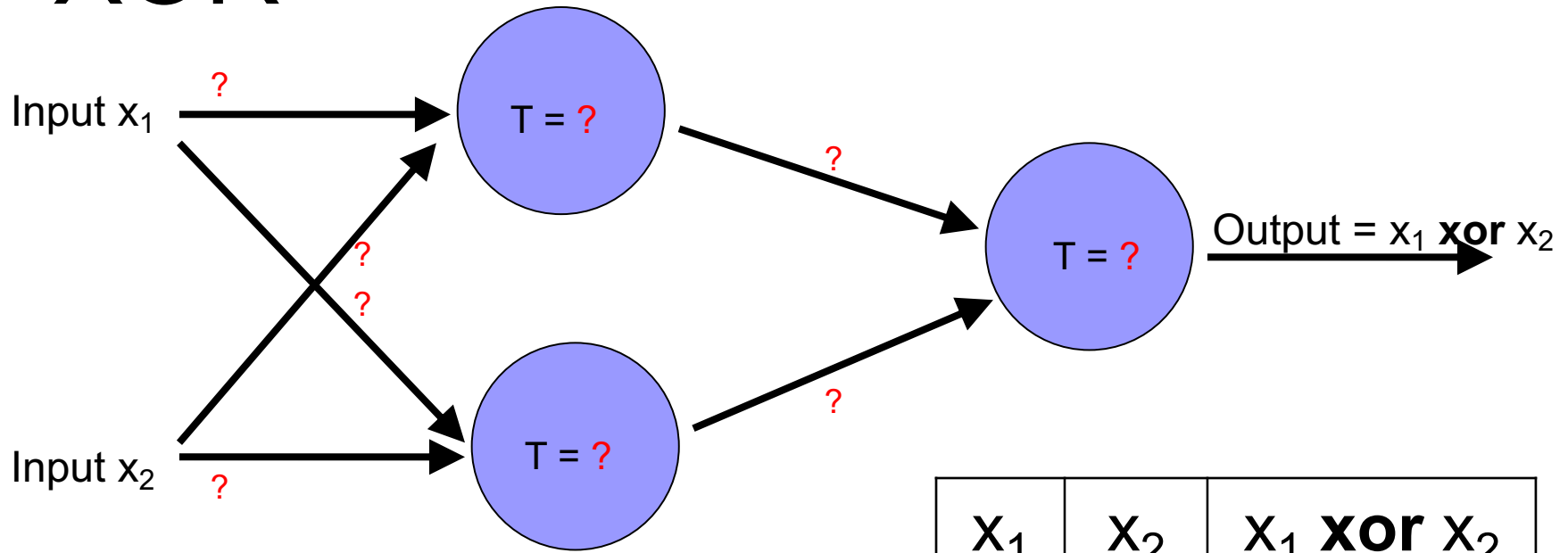
x_1	x_2	x_1 and x_2
0	0	0
0	1	0
1	0	0
1	1	1

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

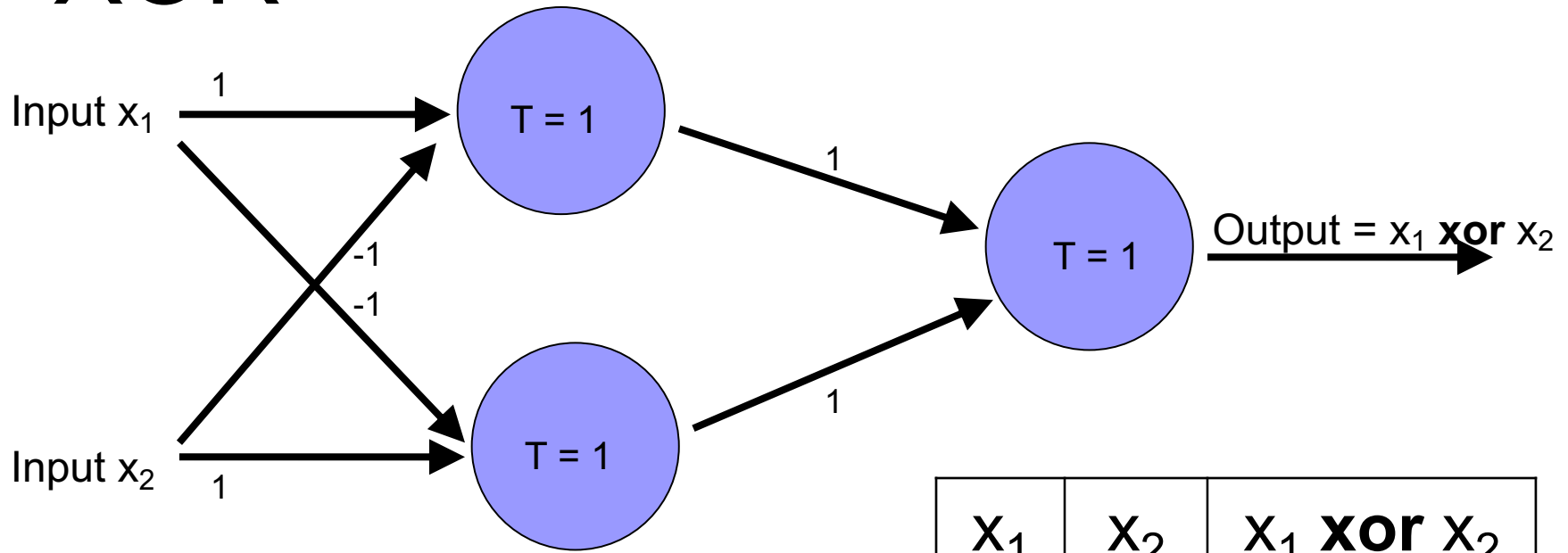


XOR



x_1	x_2	$x_1 \text{ xor } x_2$
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 as perceptrons

Examples are presented to the network

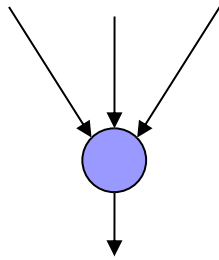
If the network computes an output that matches the desired, nothing is done

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

Learning in multilayer networks

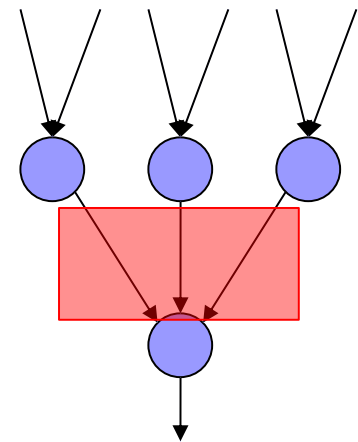
Key idea for perceptron learning: if the perceptron's output is different than the expected output, update the weights

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



perceptron

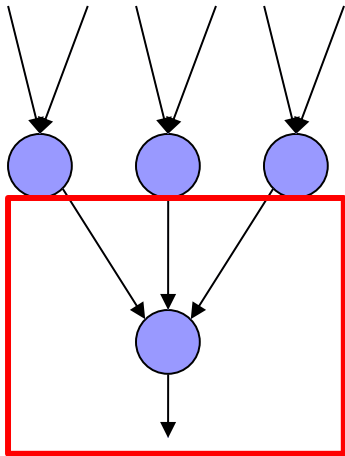
expected output?



multi-layer network

Backpropagation

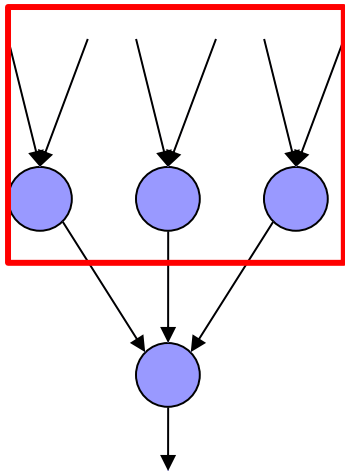
Say we get it wrong, and we now want to update the weights



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



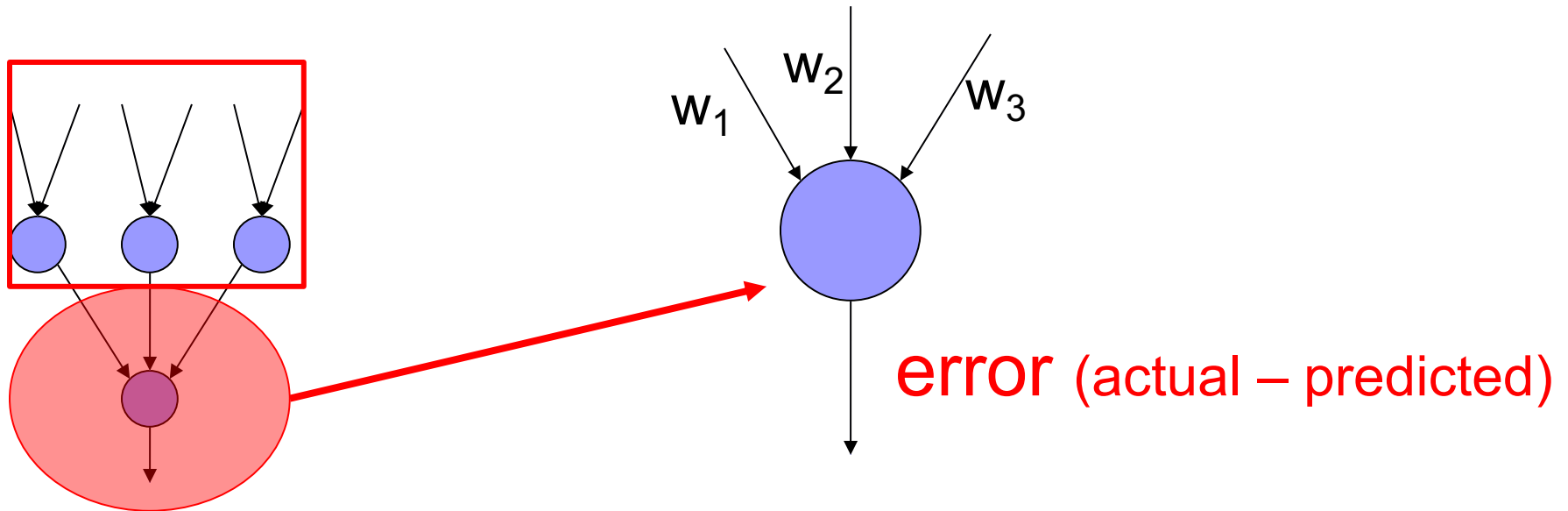
“back-propagate” the error (actual – predicted):

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

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

Backpropagation

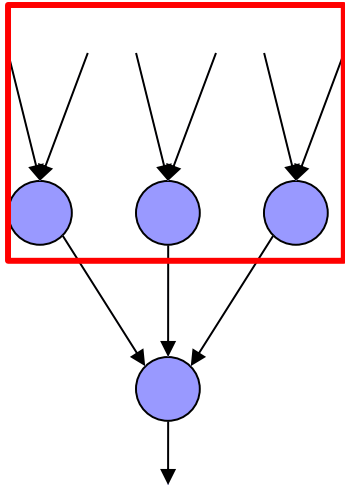
Say we get it wrong, and we now want to update the weights



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.

Handwritten digits (MNIST)





Summary

Perceptrons, one-layer networks, are insufficiently expressive

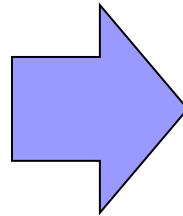
Multi-layer networks are sufficiently expressive and can be trained by error back-propagation

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

Our python NN module

Data:

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



```
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

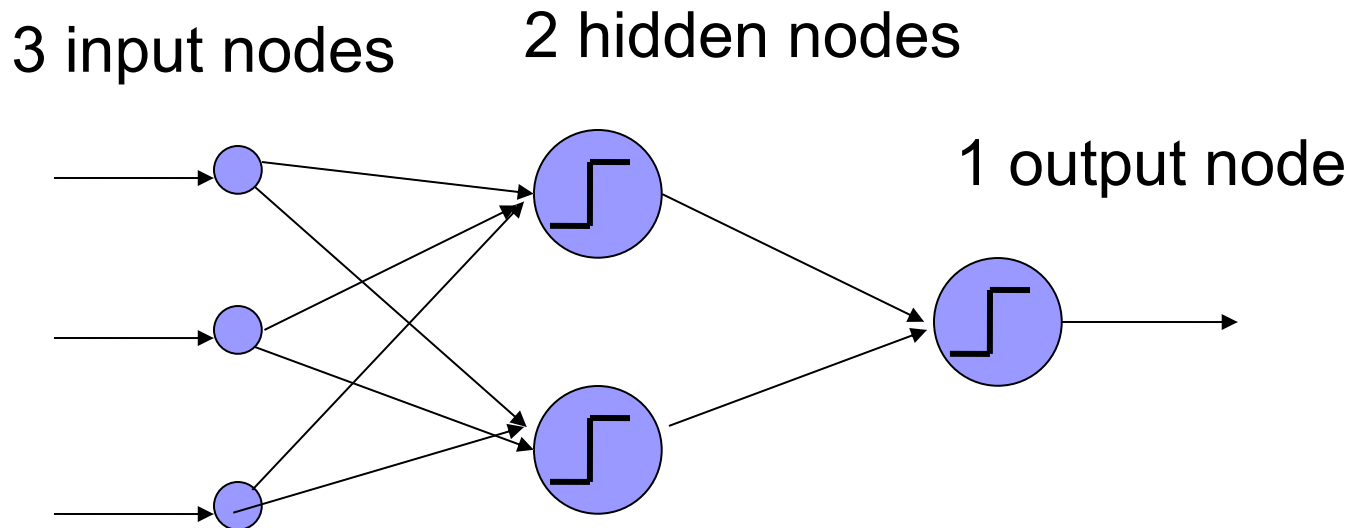
hidden nodes

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
```

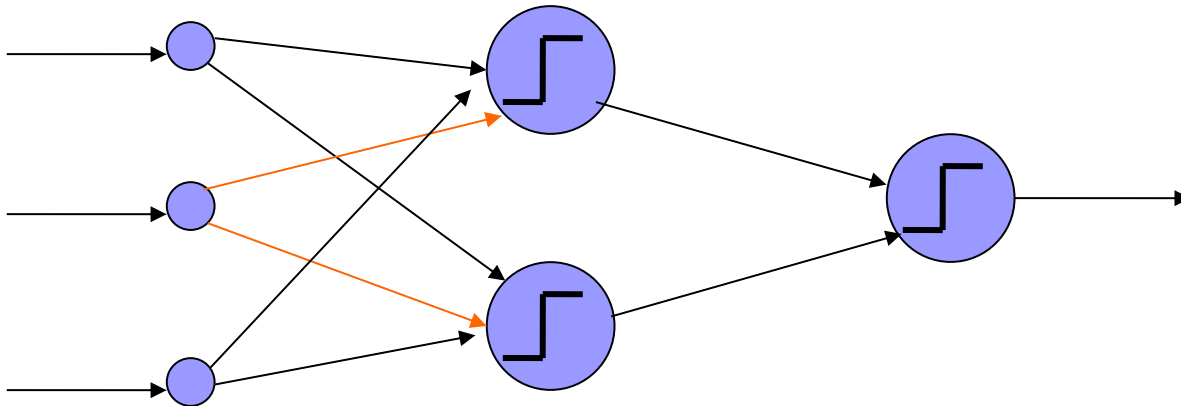
by default, trains 1000 iterations and prints out error values every 100 iterations

After training, can look at the weights

```
>>> nn.train(table)
```

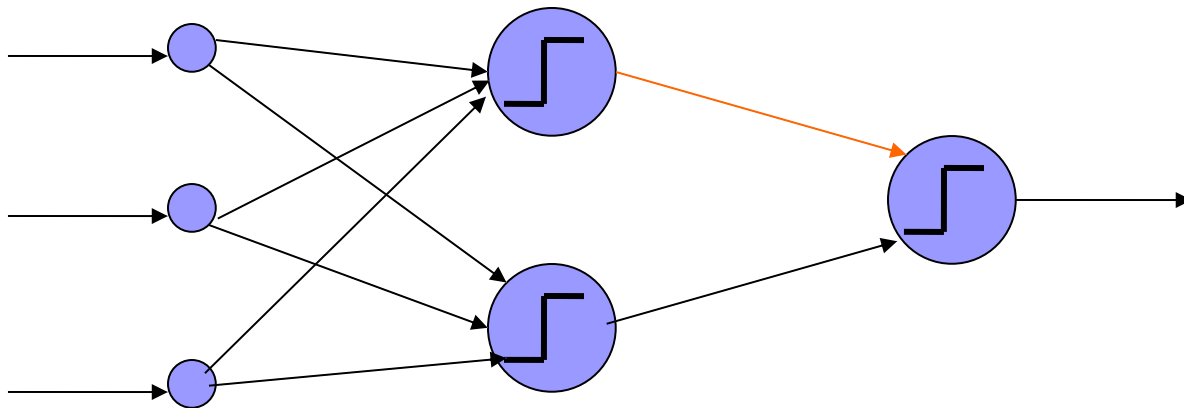
```
>>> nn.get_IH_weights()
```

```
[[ [w1a, w1b, w1c],  
   [w2a, w2b, w2c] ],  
 [b1, b2]]
```



After training, can look at the weights

```
>>> nn.get_HO_weights()  
[[ [w1a, w1b] ],  
 [b1]]
```



Many parameters to play with

```
def train(data,  
          learning_rate=0.01,  
          iterations=1000, print_interval=100)
```

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

`learning_rate` defaults to 0.01

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

`print_interval` defaults to 100. The value of the error is displayed after `print_interval` 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.

Calling with optional parameters

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



Optional parameters

See:

https://cs.pomona.edu/classes/cs51a/examples/optional_parameters.txt



Optional parameters

Check out the constructor (`__init__` function) of `NeuralNet` for another interesting optional parameter: `activation` function!

It may be worth experimenting with different activation functions to see what happens to accuracy and run time...

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)
```