03-06-2023

CS051A

INTRO TO COMPUTER SCIENCE WITH TOPICS IN AI

13: Perceptron learning and backpropagation

Alexandra Papoutsaki

she/her/hers

Lectures

Zilong Ye

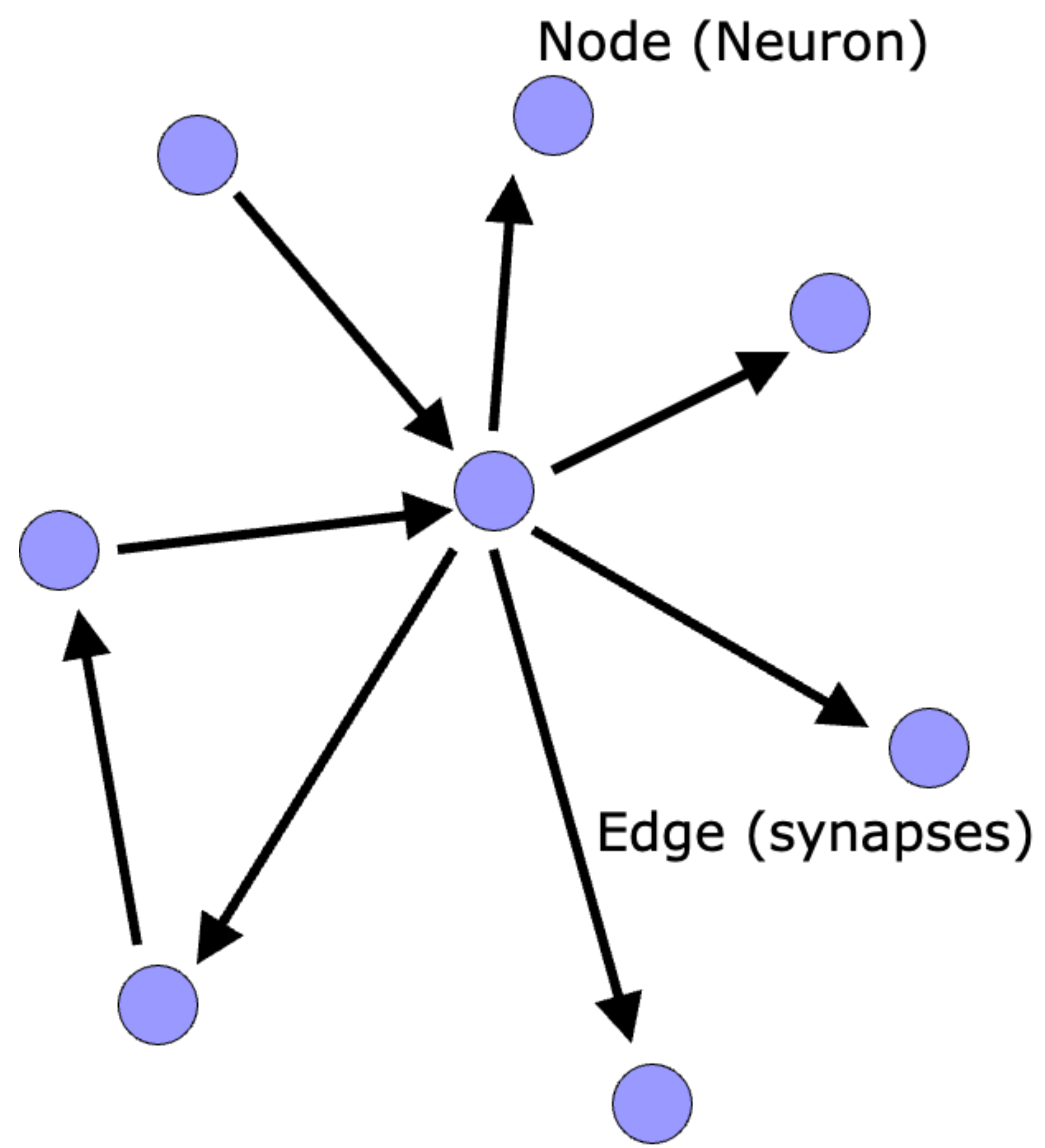
he/him/his

Labs

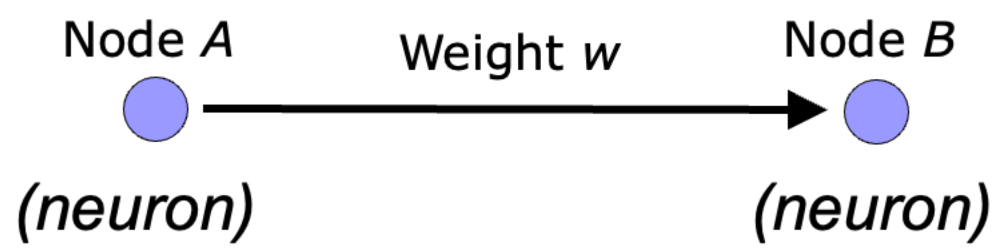
Lecture 13: Perceptron learning and back propagation

- Perceptron learning
- Back propagation

Artificial Neural Networks - Our approximation

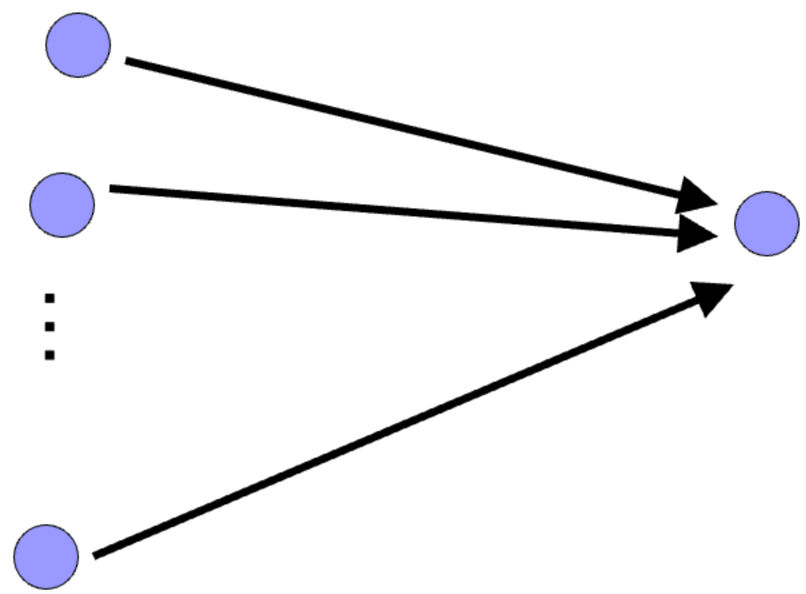


Strength of signal



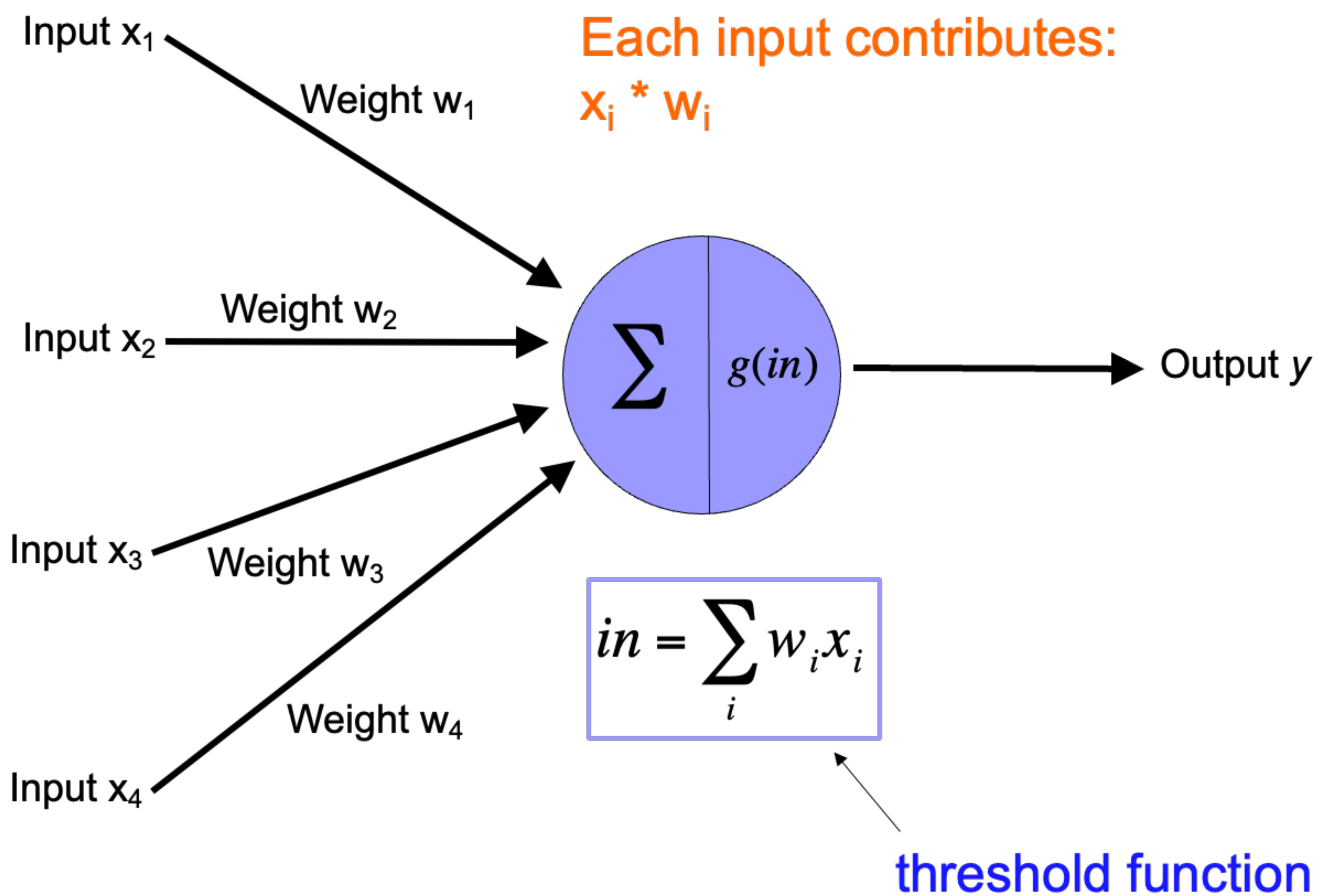
- w is the strength of signal sent between A and B.
- If A fires and w is positive, then A stimulates B.
- If A fires and w is negative, then A inhibits B.

Firing a neuron



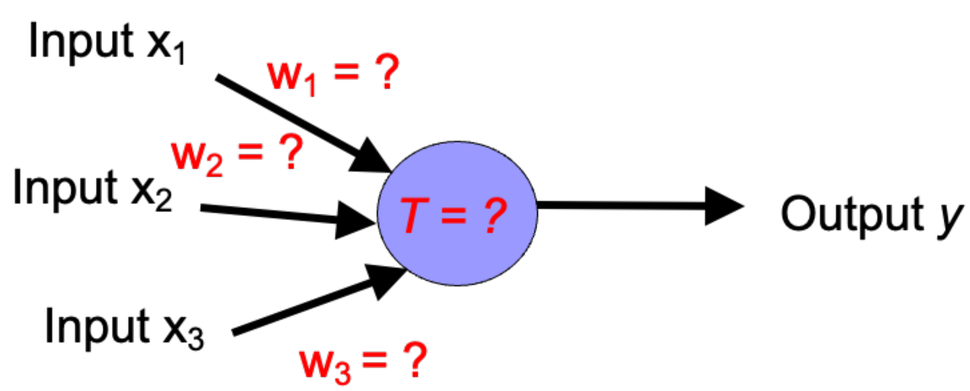
- A given neuron has many, many connecting, input neurons.
- If a neuron is stimulated enough, then it also fires.
- How much stimulation is required is determined by its threshold.

A single neuron/perceptron



Training neural networks

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

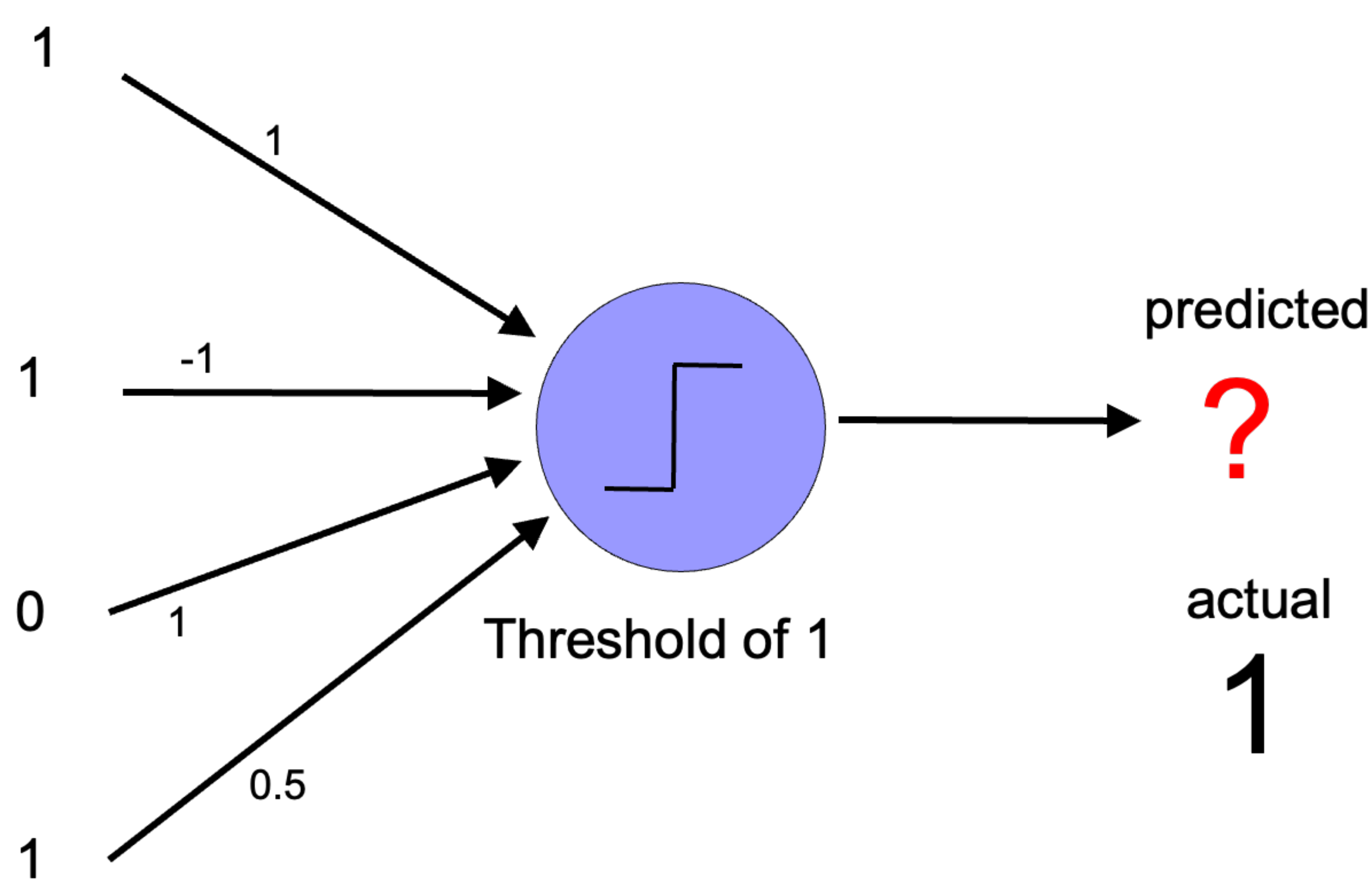


- start with some initial weights and thresholds
- show examples repeatedly to NN
- update weights/thresholds by comparing NN output to actual output

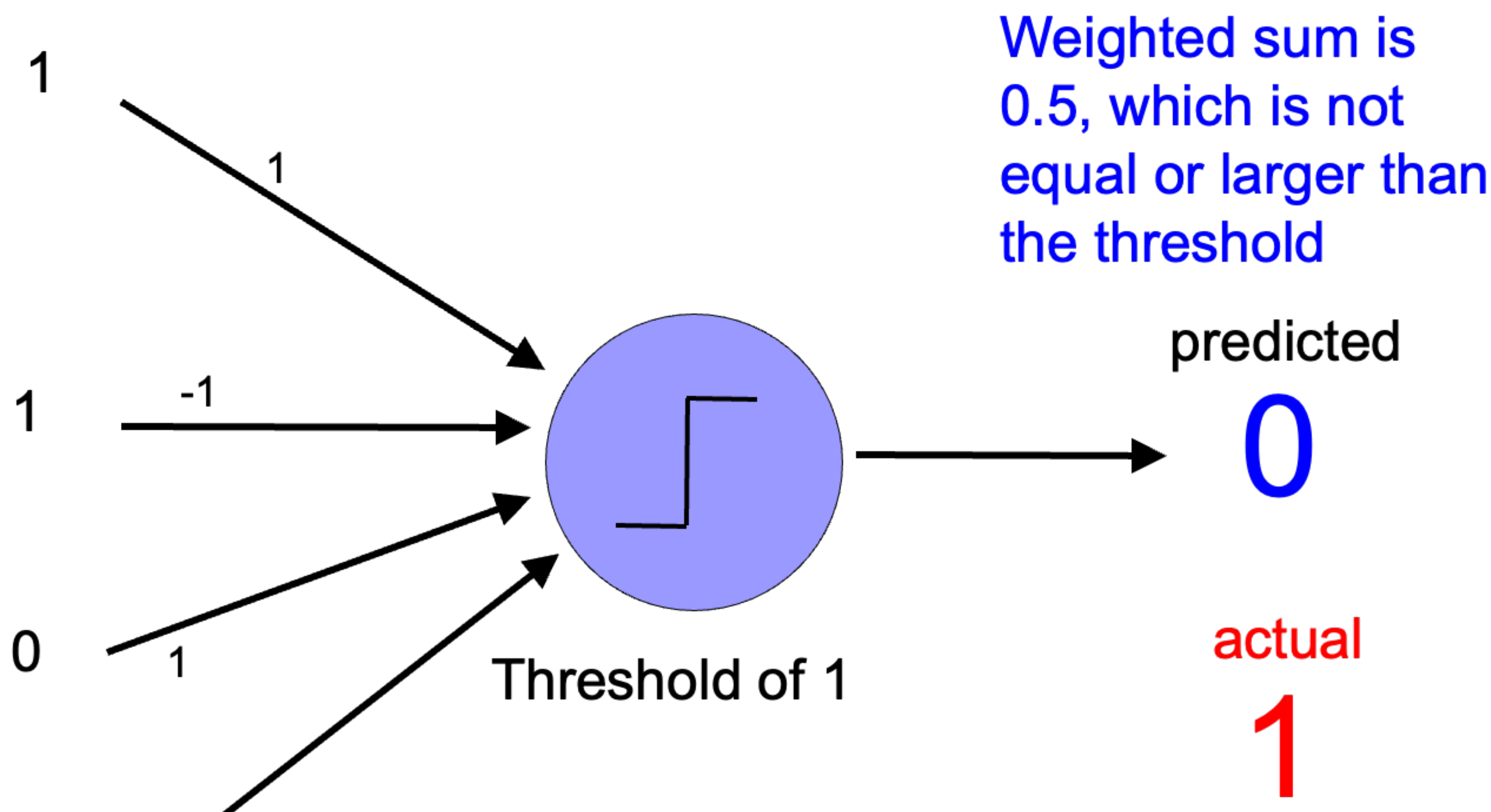
Perceptron learning algorithm

- 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

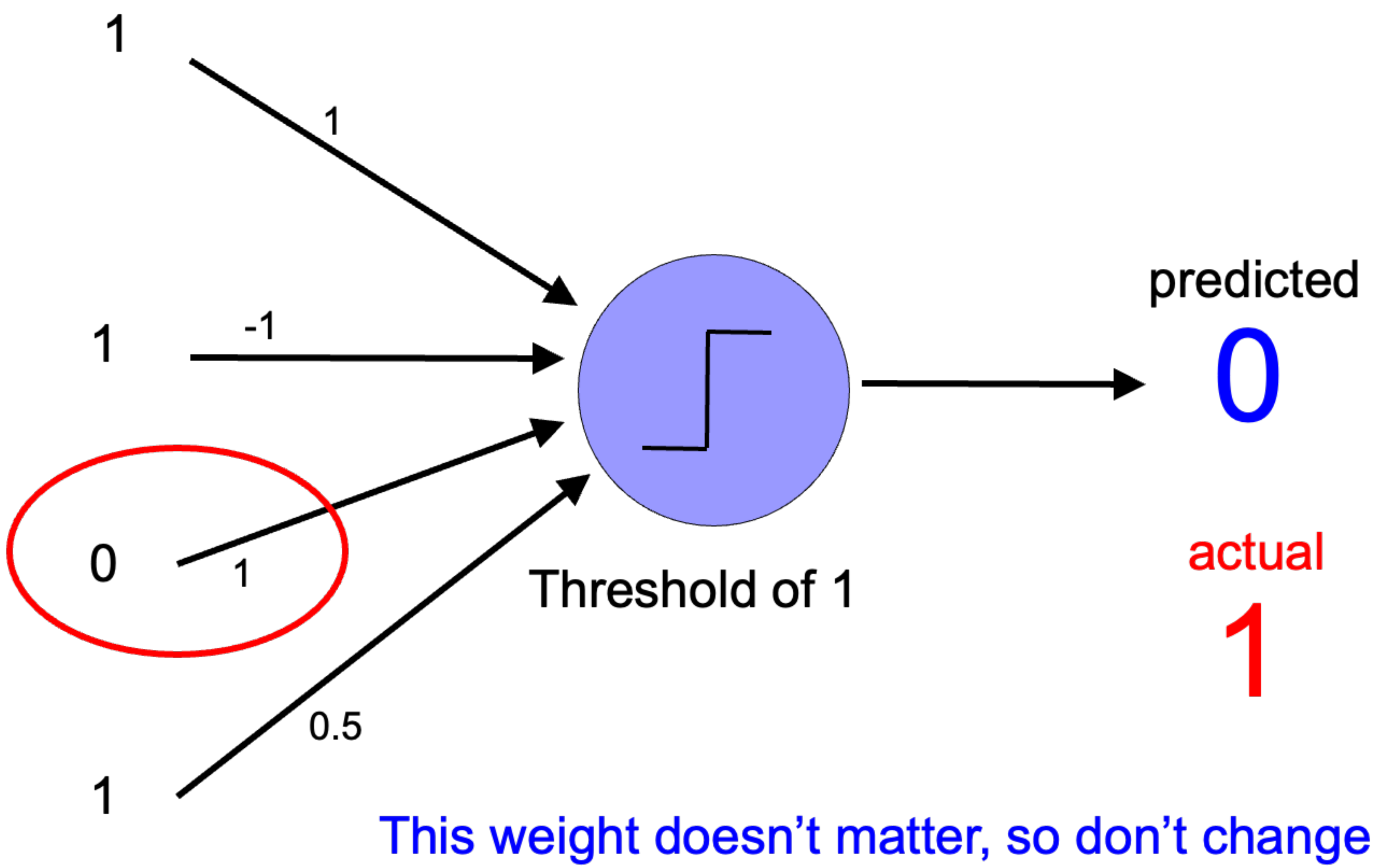


Perceptron learning

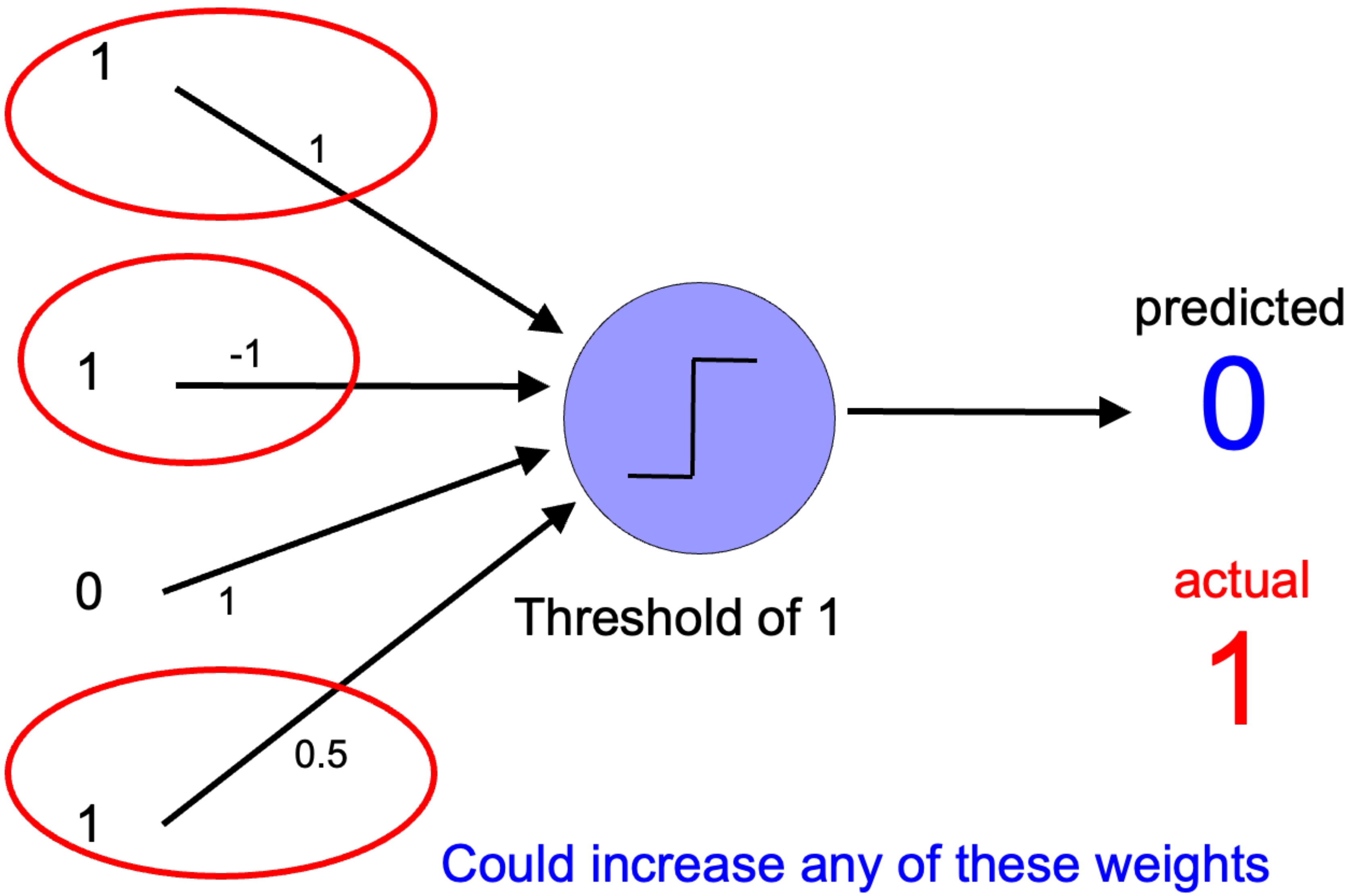


What could we adjust to make it right?

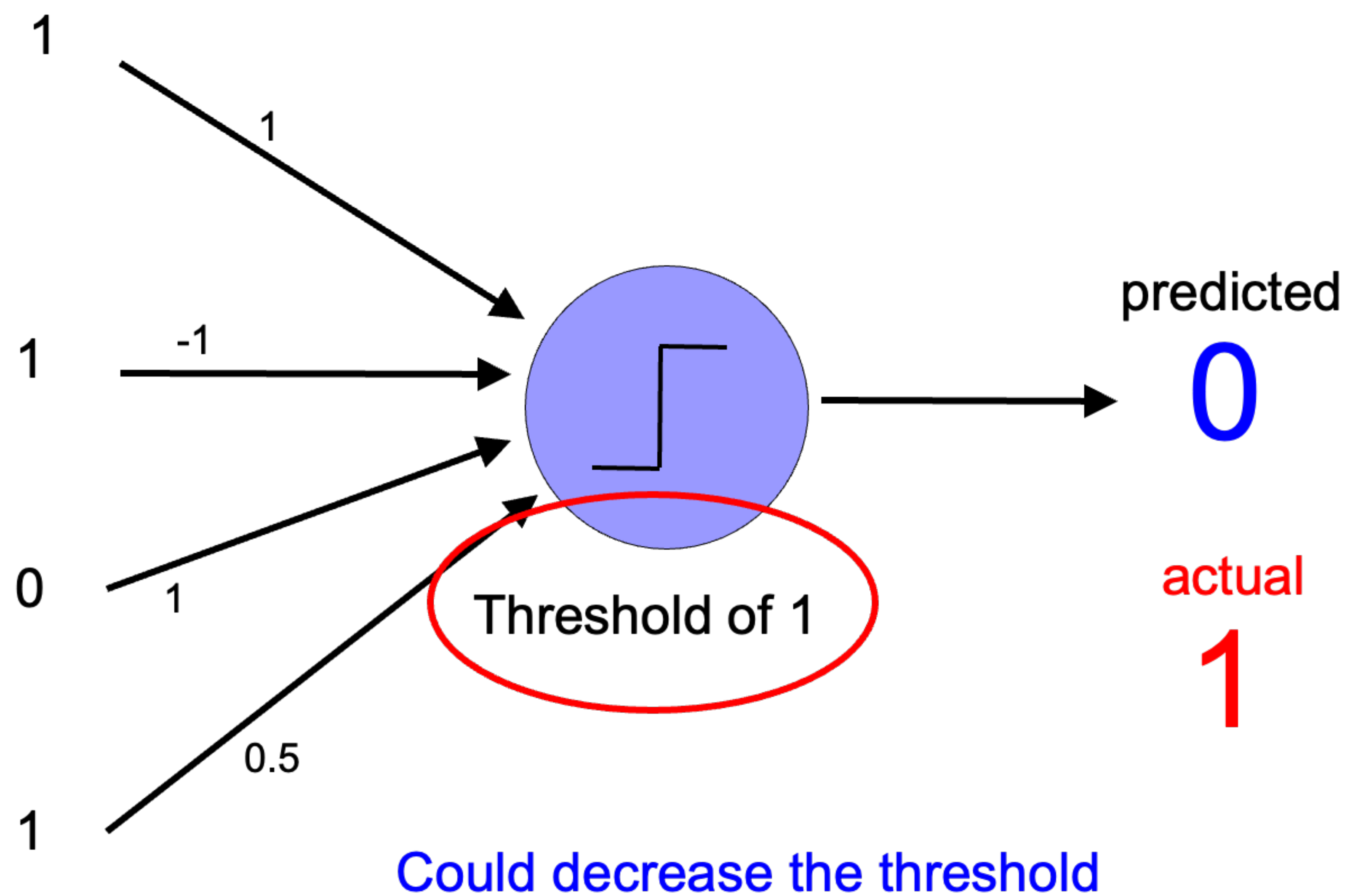
Perceptron learning



Perceptron learning



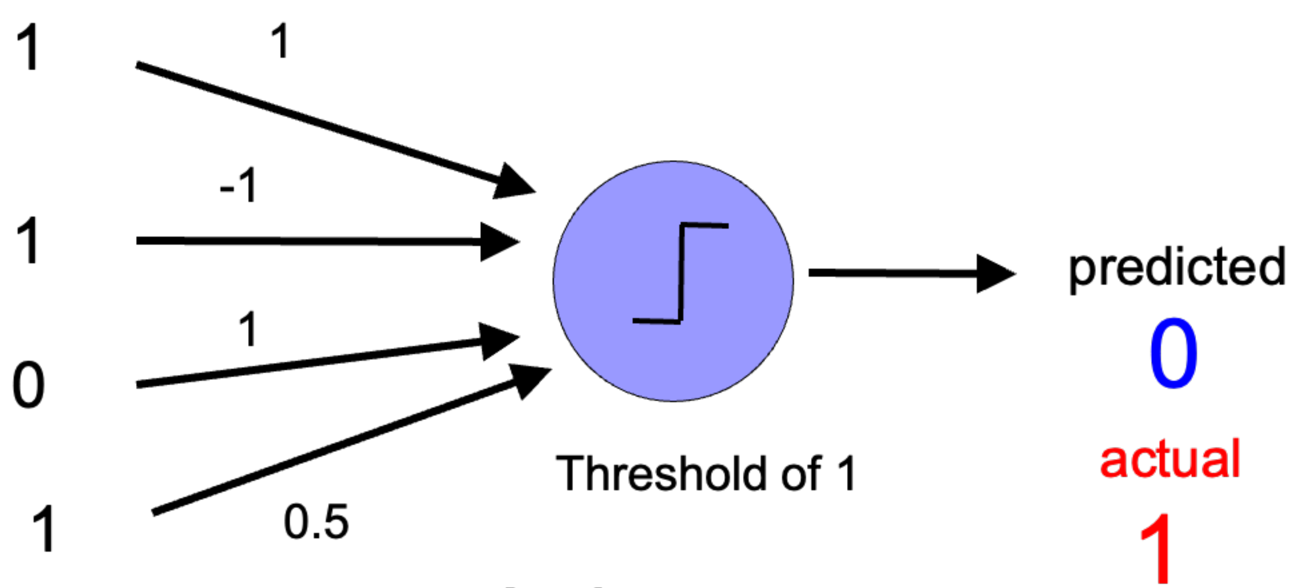
Perceptron learning



Perceptron update rule

- If wrong:
  - Update weights and threshold towards getting this example correct
  - $wi = wi + \Delta wi$
  - $\Delta wi = \lambda * (actual - predicted) * xi$

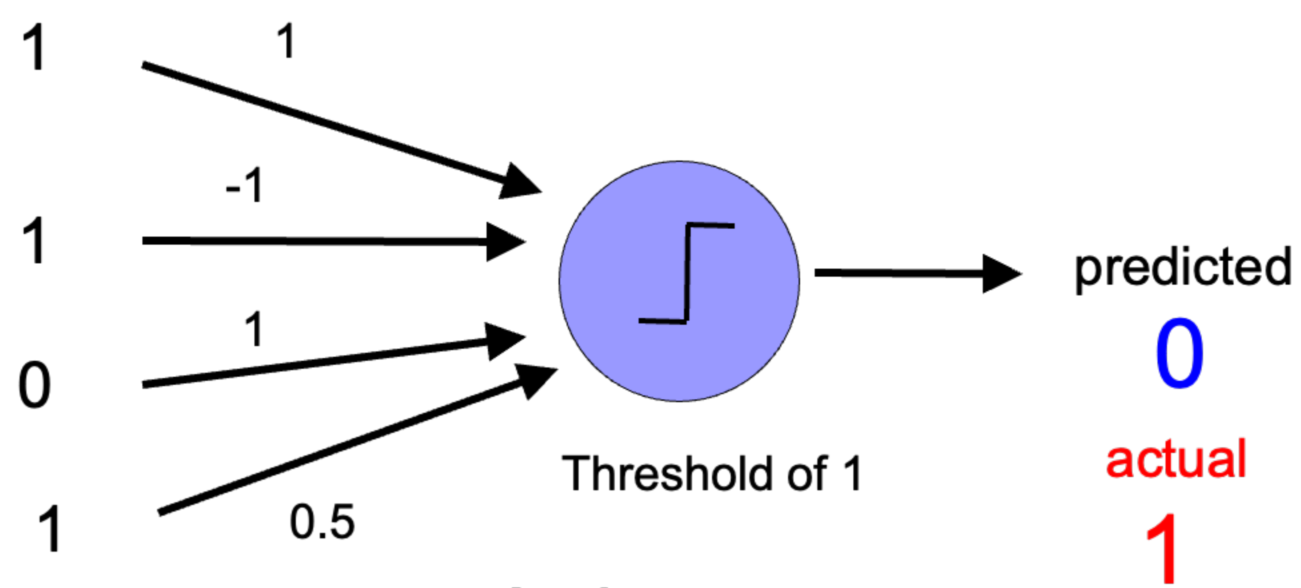
Perceptron learning



$$w_i = w_i + \Delta wi$$

$$\Delta w_i = \lambda * \underbrace{(\text{actual - predicted})}_{\text{What does this do in this case?}} * x_i$$

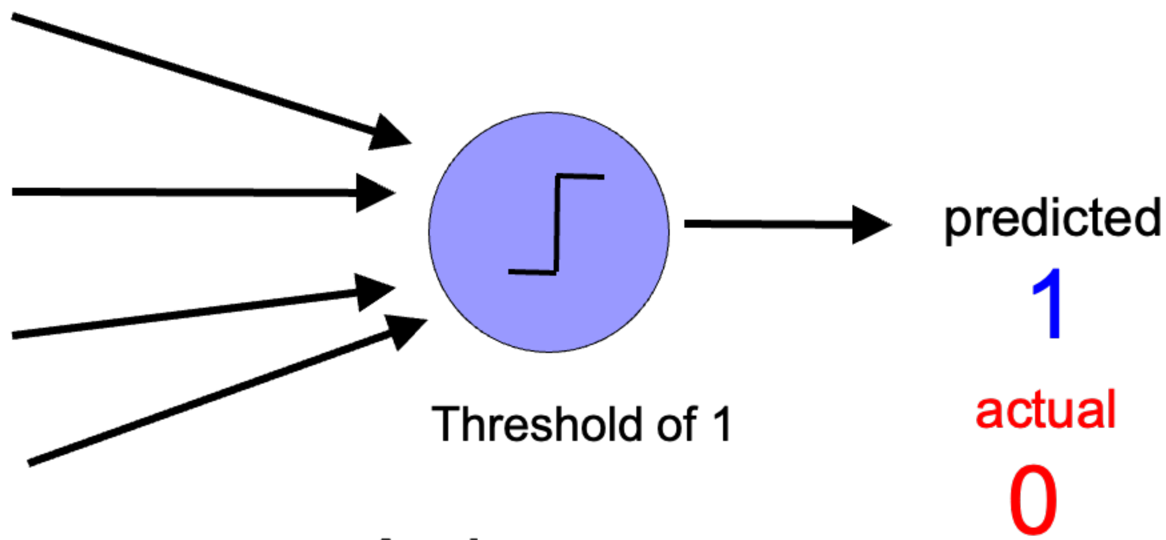
Perceptron learning



$$w_i = w_i + \Delta wi$$

$$\Delta w_i = \lambda * \underbrace{(\text{actual - predicted})}_{\text{causes us to increase the weights!}} * x_i$$

Perceptron learning

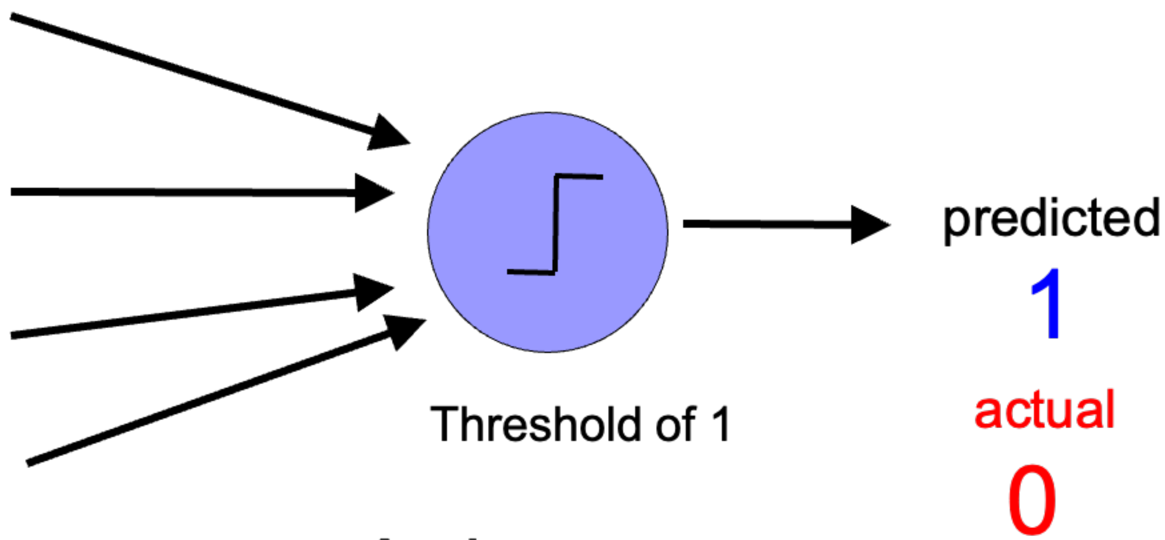


$$w_i = w_i + \Delta wi$$

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

What if predicted = 1 and actual = 0?

Perceptron learning

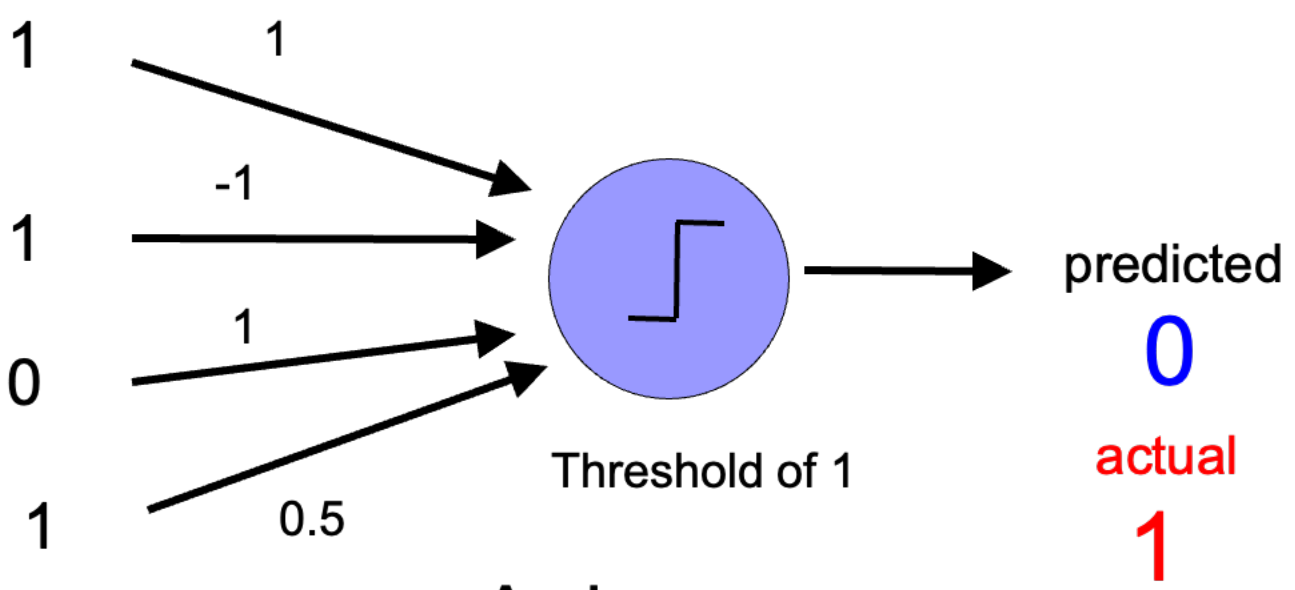


$$w_i = w_i + \Delta wi$$

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

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

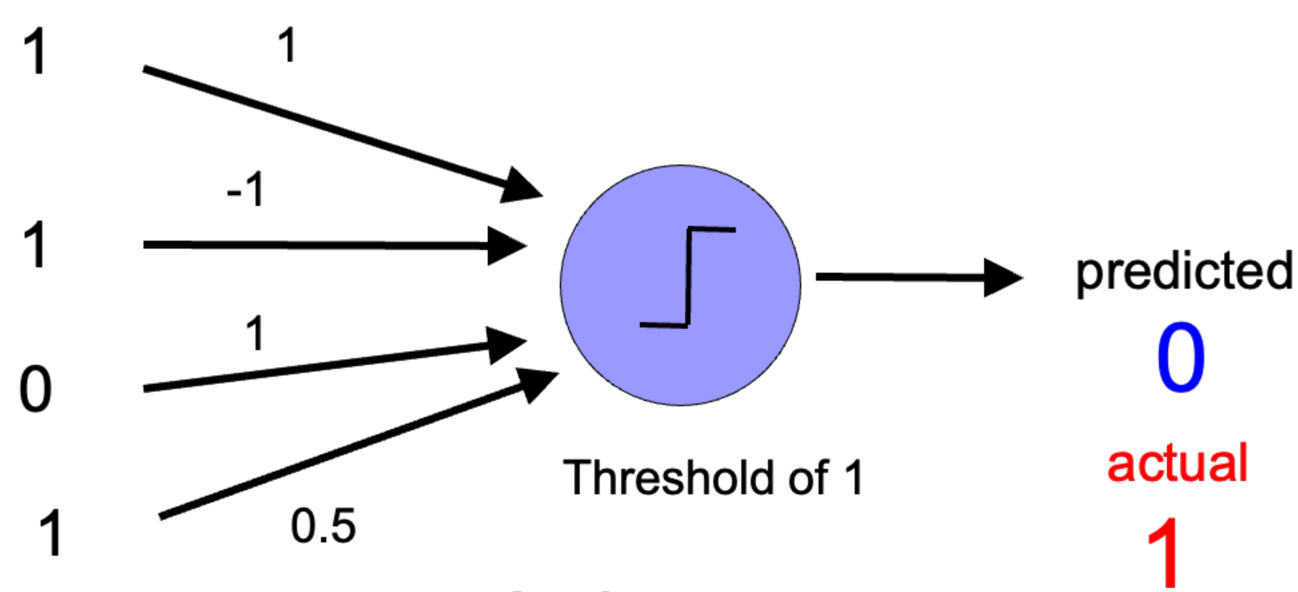
Perceptron learning



$$w_i = w_i + \Delta wi$$

$$\Delta w_i = \lambda * (\text{actual - predicted}) * \underbrace{x_i}_{\text{What does this do?}}$$

Perceptron learning

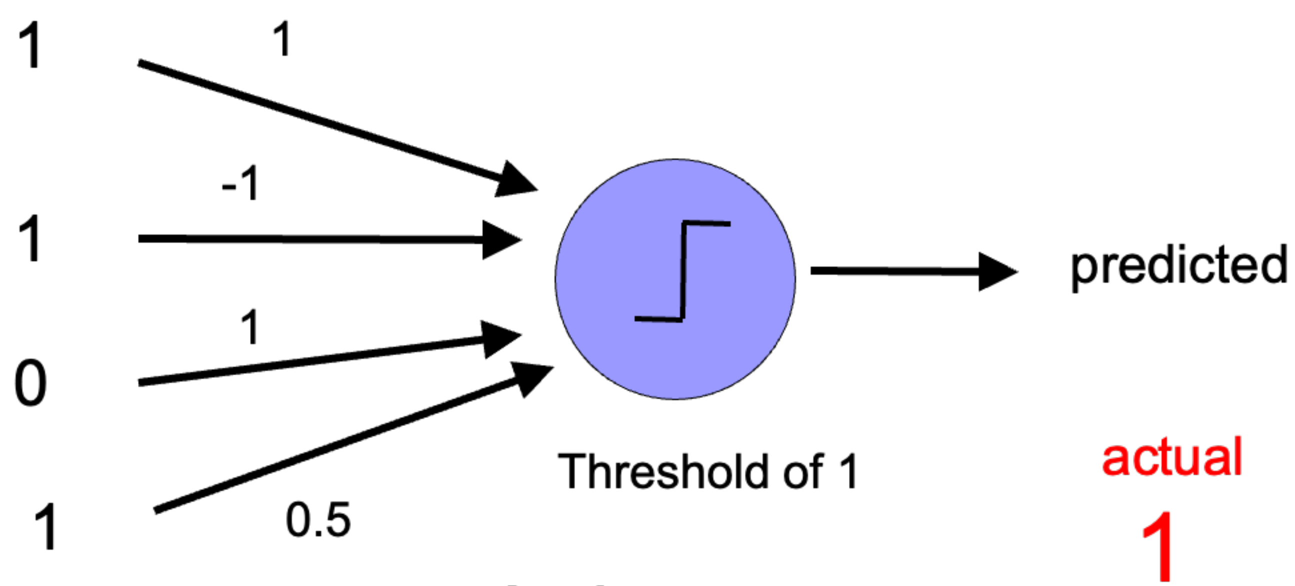


$$w_i = w_i + \Delta wi$$

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

Only adjust those weights that actually contributed!

Perceptron learning

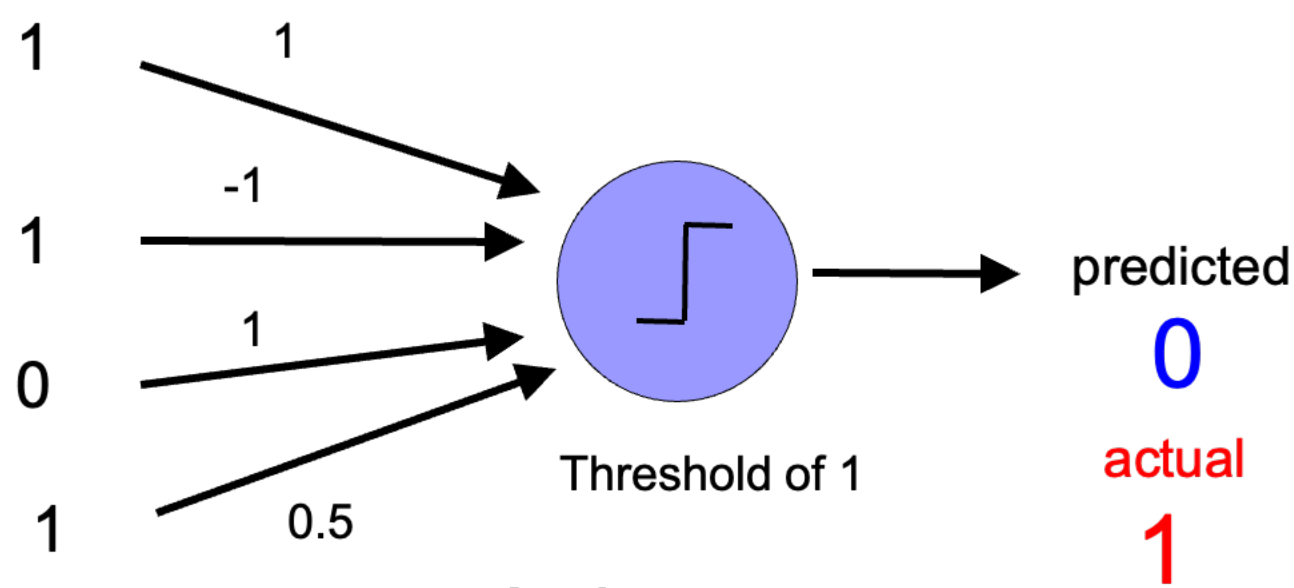


$$w_i = w_i + \Delta wi$$

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

What does this do?

Perceptron learning

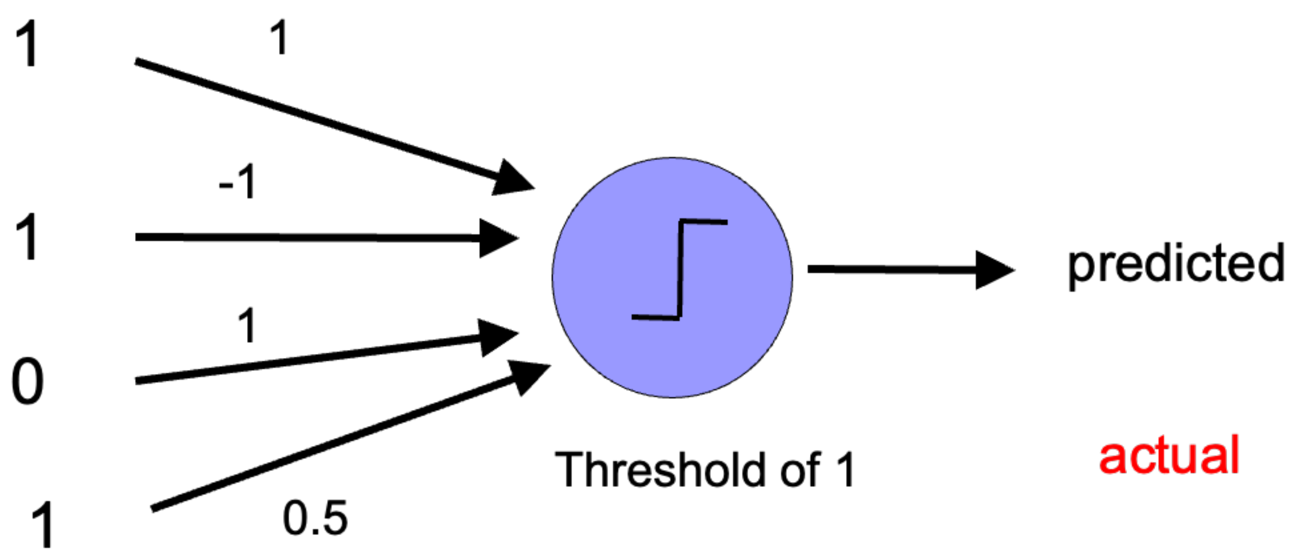


$$w_i = w_i + \Delta wi$$

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

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

Perceptron learning

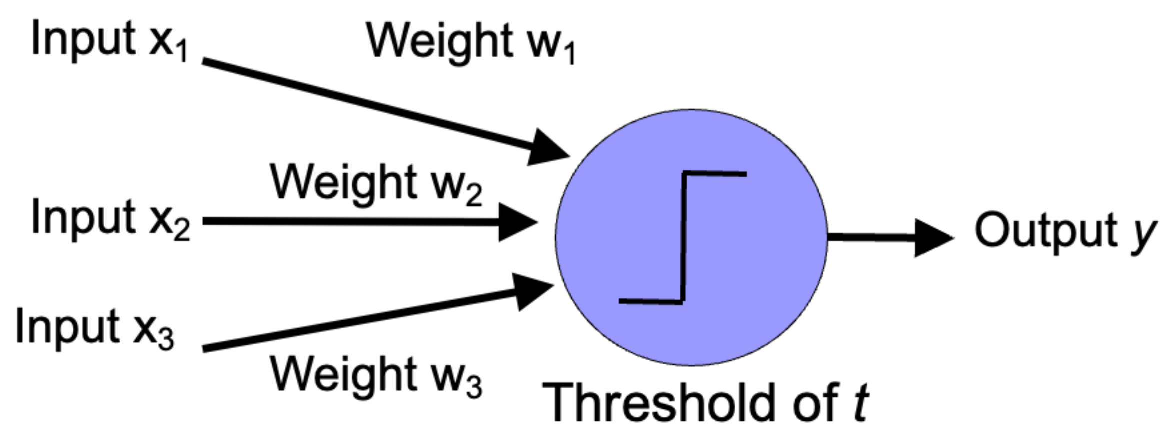


$$w_i = w_i + \Delta wi$$

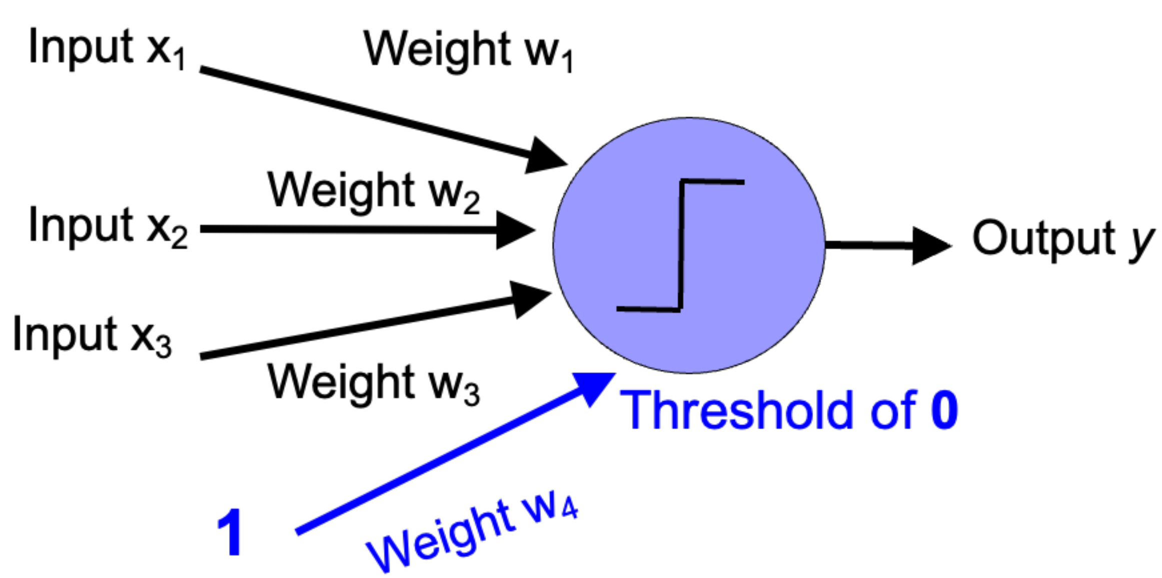
$$\Delta w_i = \lambda * (\text{actual} - \text{predicted}) * x_i$$

What about the threshold?

Perceptron learning

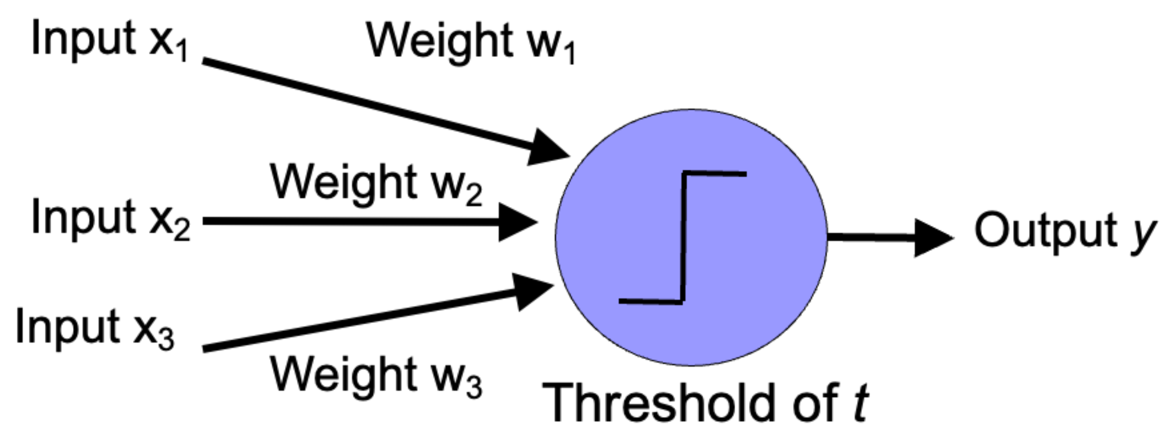


$$1 \text{ if } \sum_{i=1}^{3} w_i x_i \geq t$$

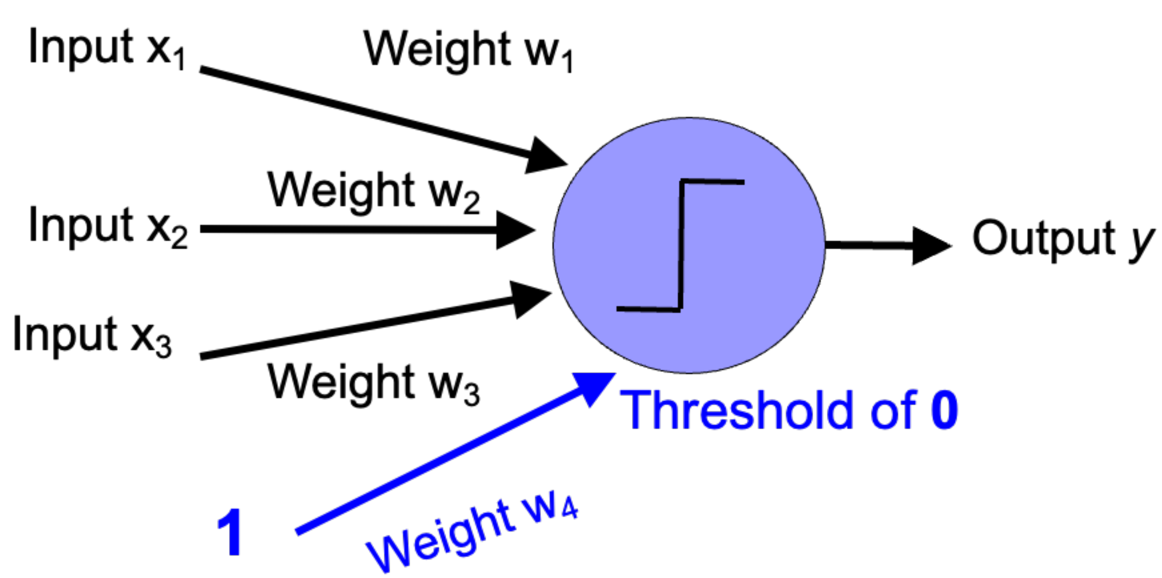


$$1 \text{ if } w_4 + \sum_{i=1}^{3} w_i x_i \geq 0$$

Perceptron learning



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



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

equivalent when $w_4 = -t$

Perceptron learning algorithm

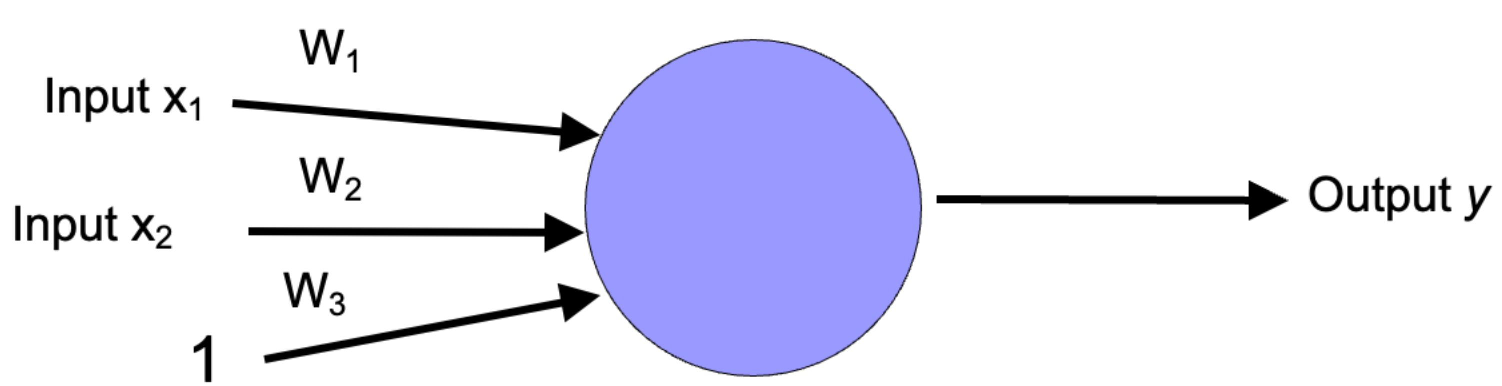
- Initialize weights of the model randomly
- Repeat until you get *all* examples right:
  - For each "training" example (in a random order):
    - Calculate current prediction on example
    - If *wrong*:
      - $wi = wi + \lambda * (actual - predicted) * xi$

Perceptron learning

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

$\lambda = 0.1$

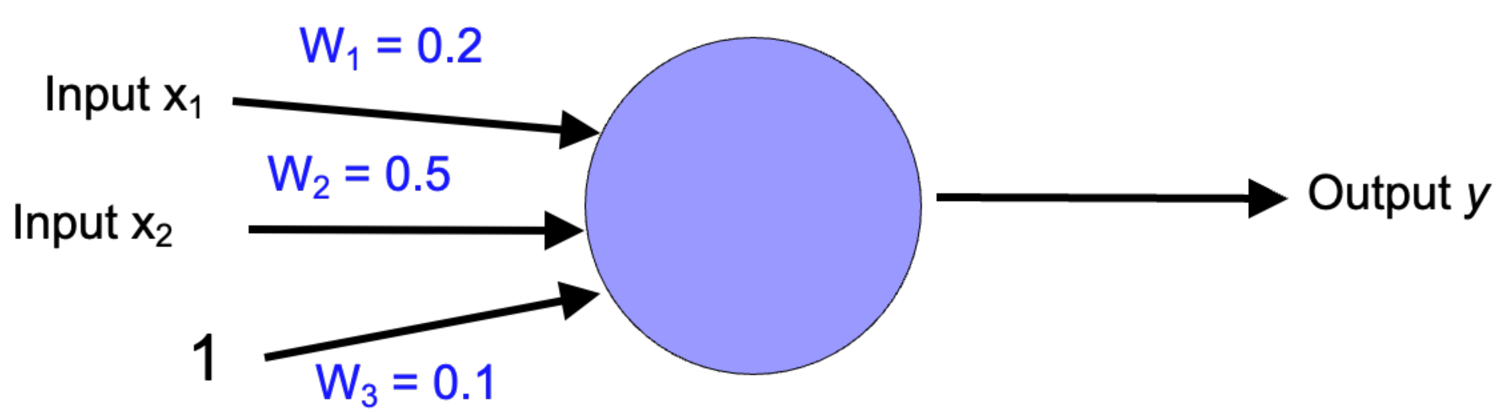
initialize with random weights



Perceptron learning

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

$\lambda = 0.1$



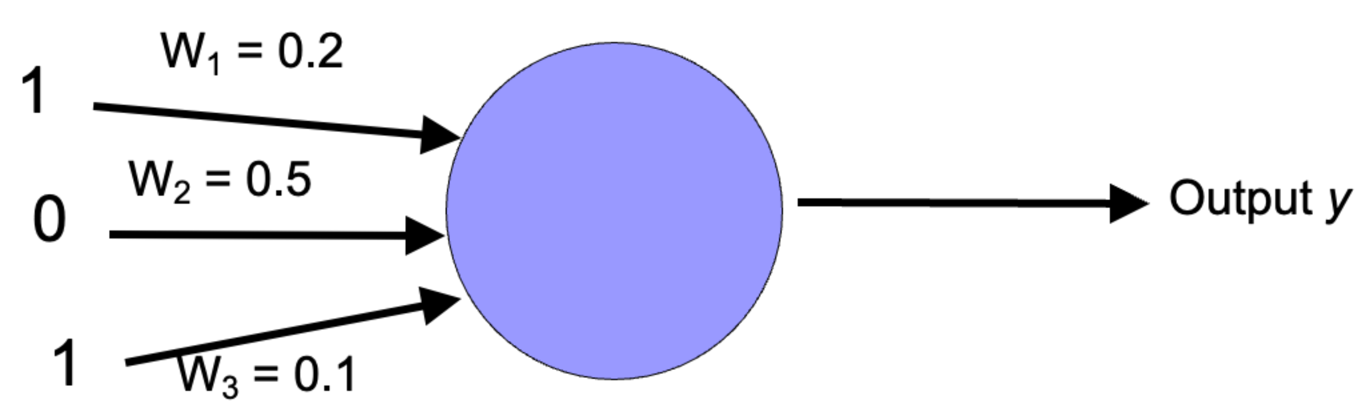
Perceptron learning

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:

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



Right or wrong?

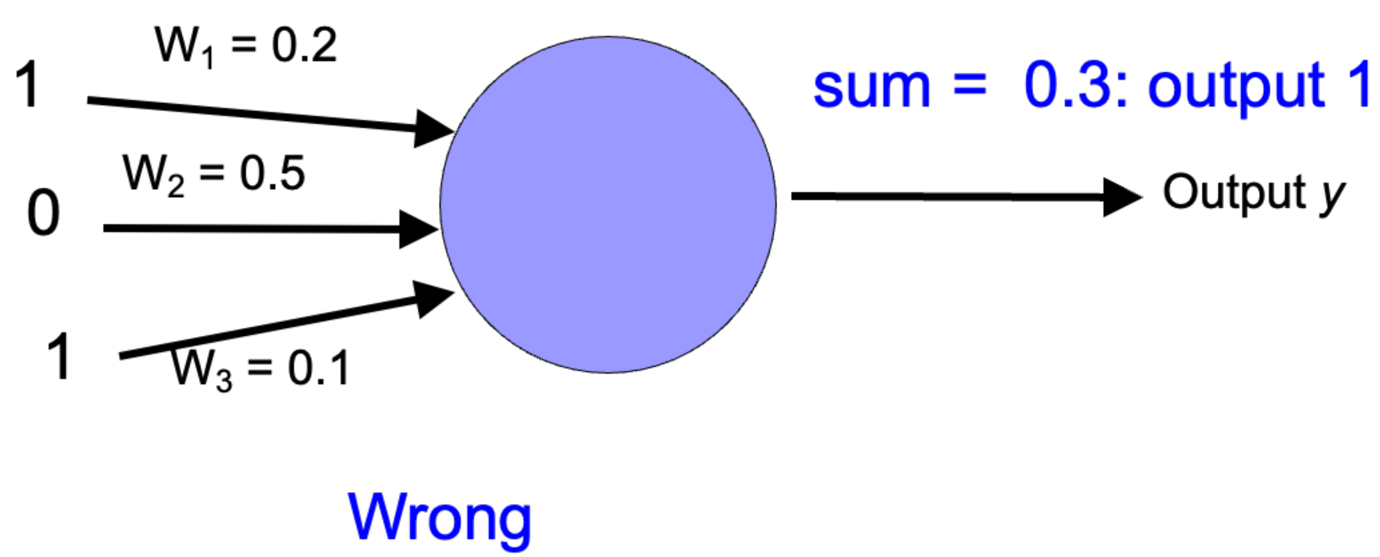
Perceptron learning

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:

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



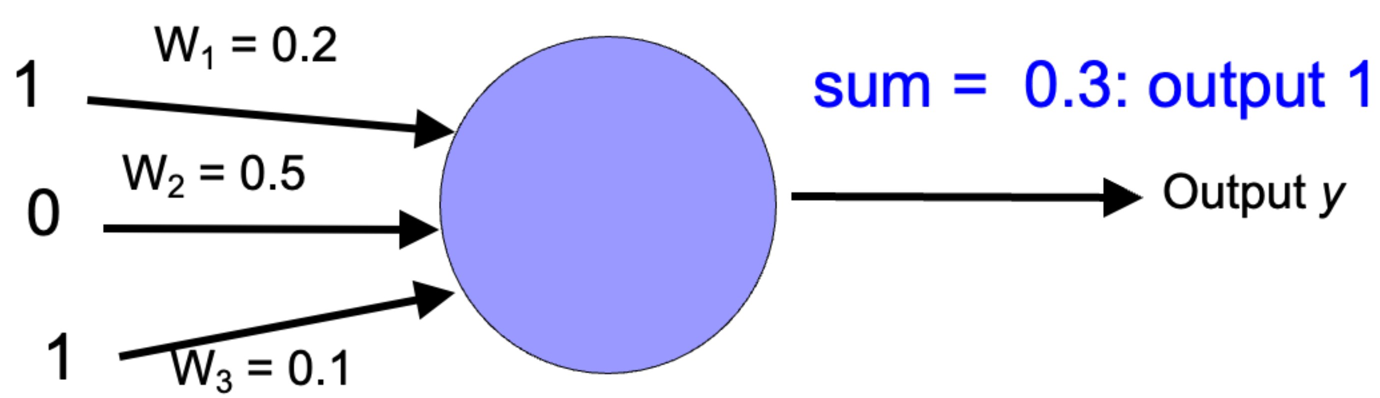
Perceptron learning

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:

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



new weights?

Perceptron learning

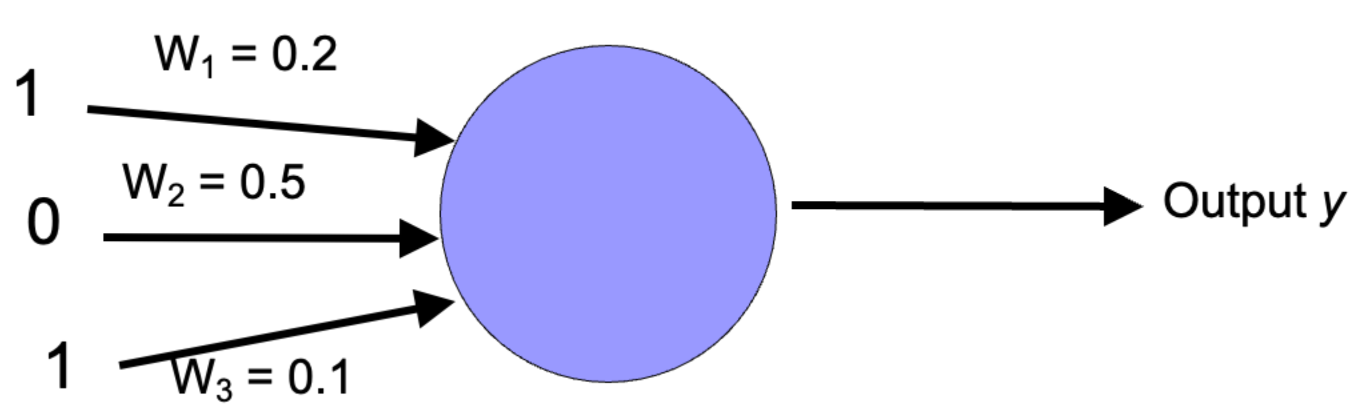
x_1	x_2	x_1 **and** x_2
0	0	0
0	1	0
1	0	0
1	1	1

$\lambda = 0.1$

if wrong:

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

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



Perceptron learning

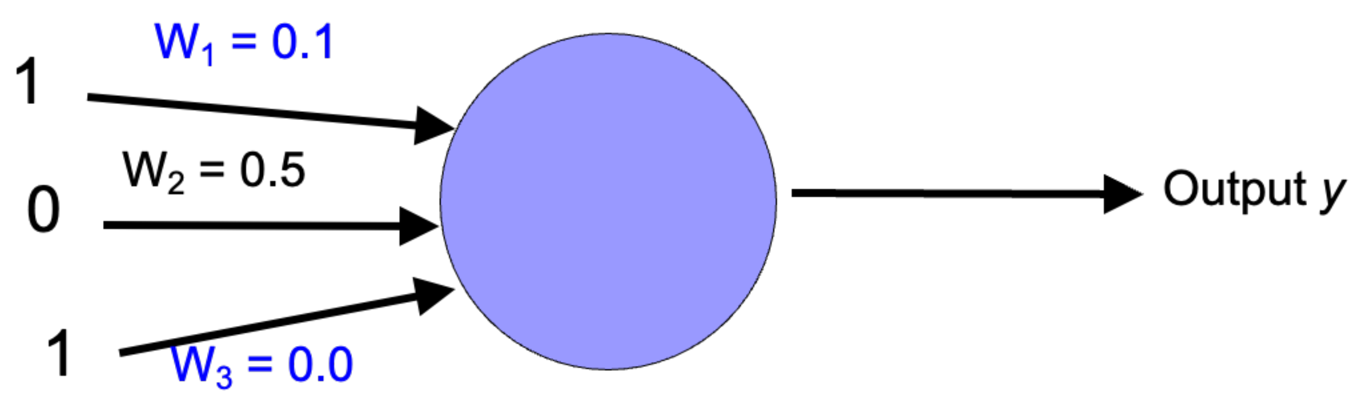
x_1	x_2	x_1 **and** x_2
0	0	0
0	1	0
1	0	0
1	1	1

$\lambda = 0.1$

if wrong:

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

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



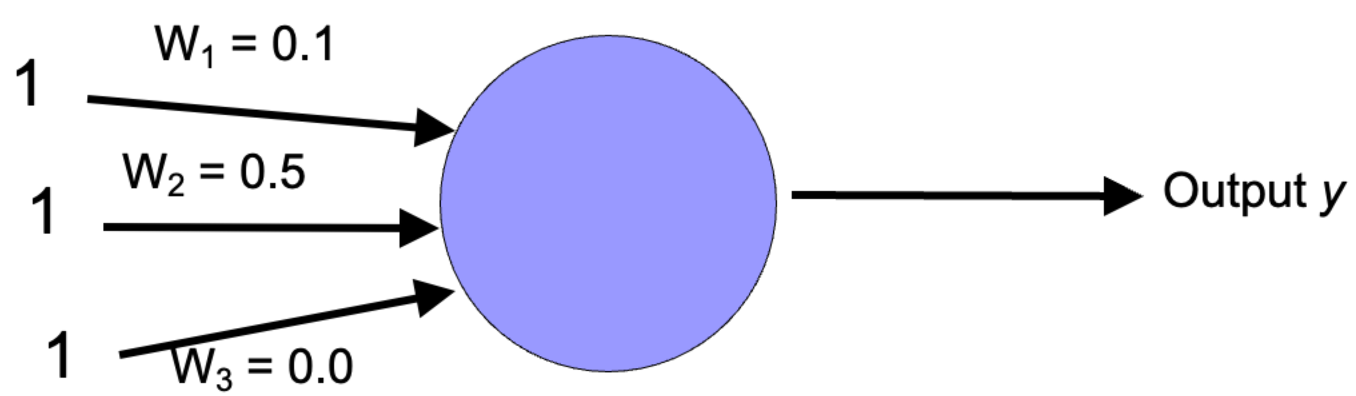
Perceptron learning

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?

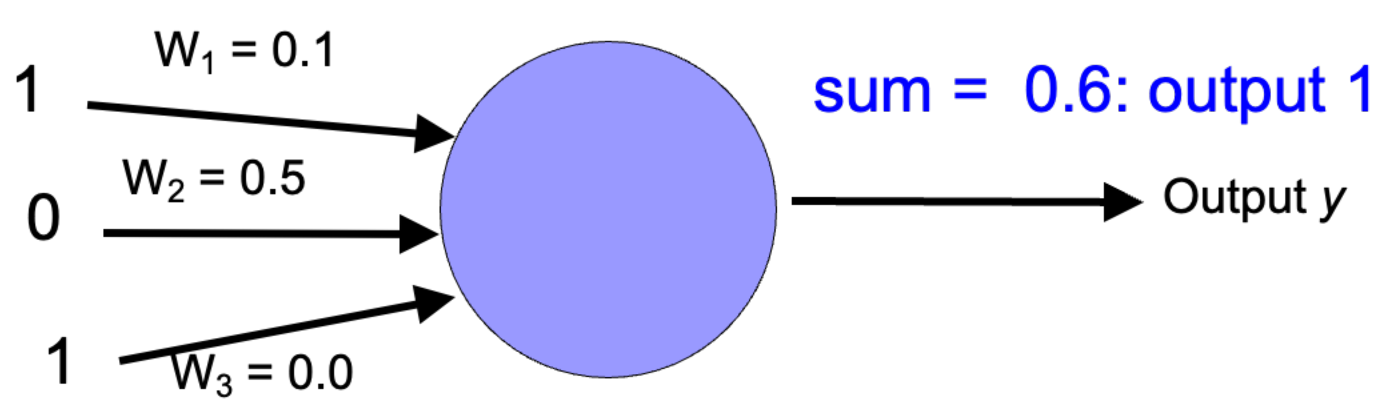
Perceptron learning

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:

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



Right. No update!

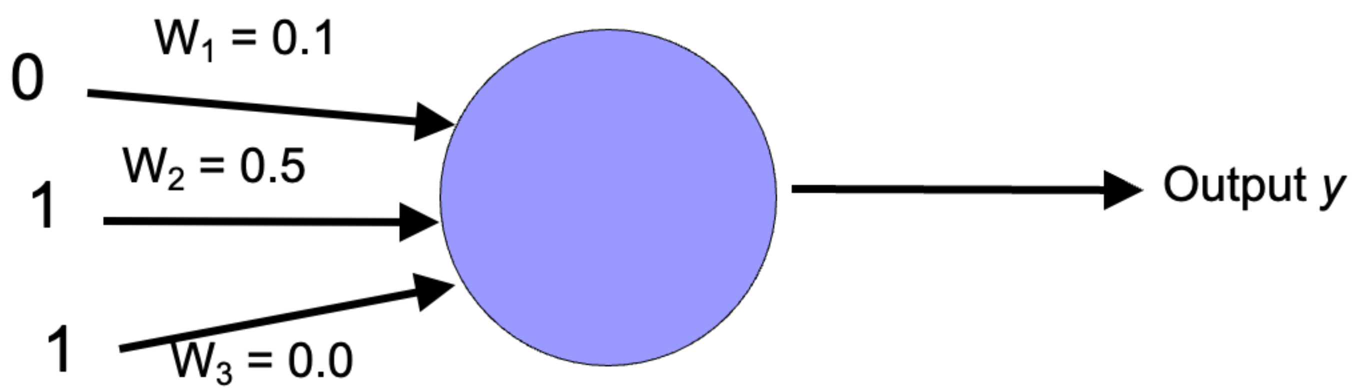
Perceptron learning

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:

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



Right or wrong?

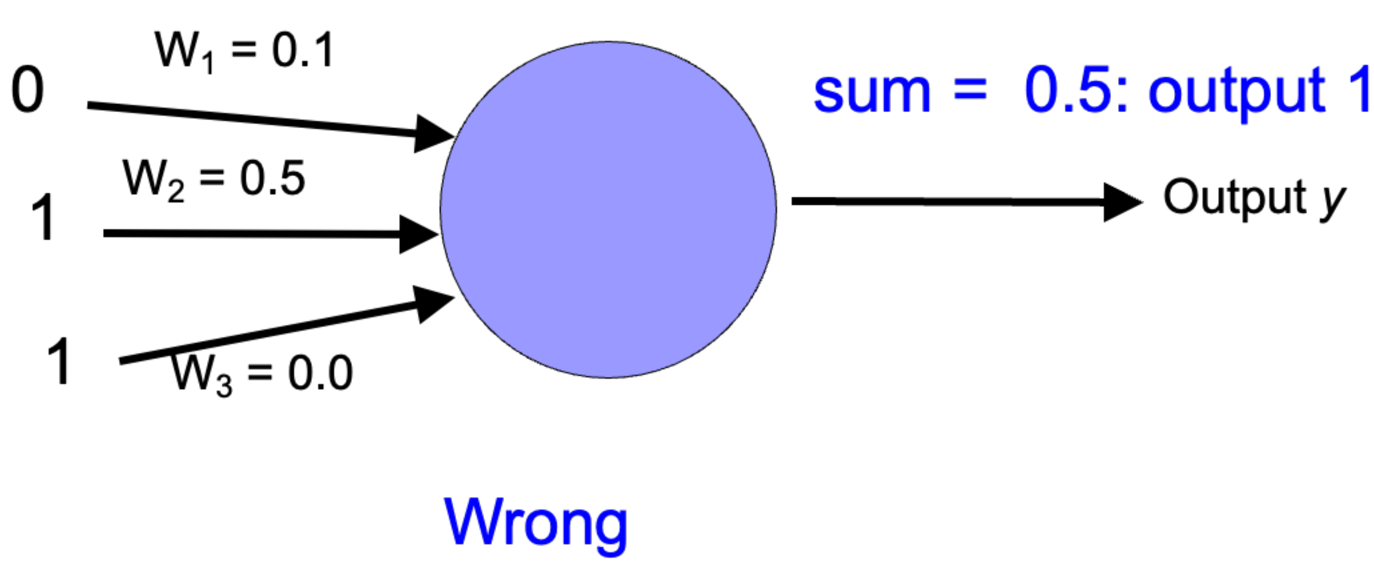
Perceptron learning

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:

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



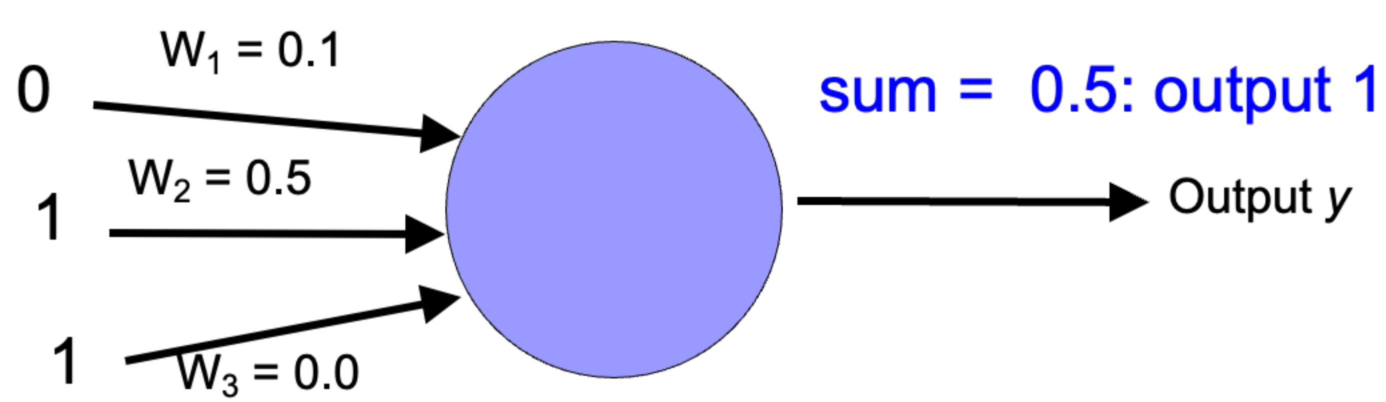
Perceptron learning

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:

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



new weights?

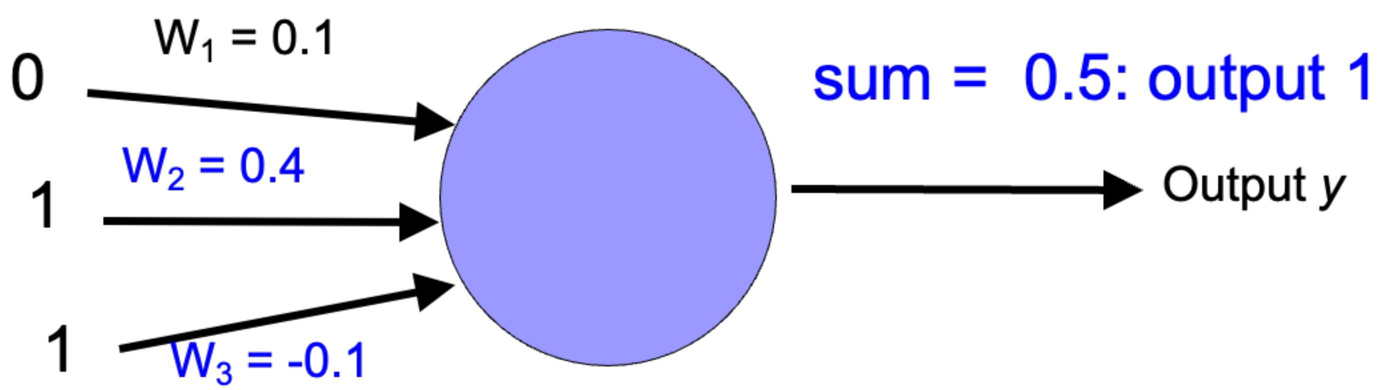
Perceptron learning

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:

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



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

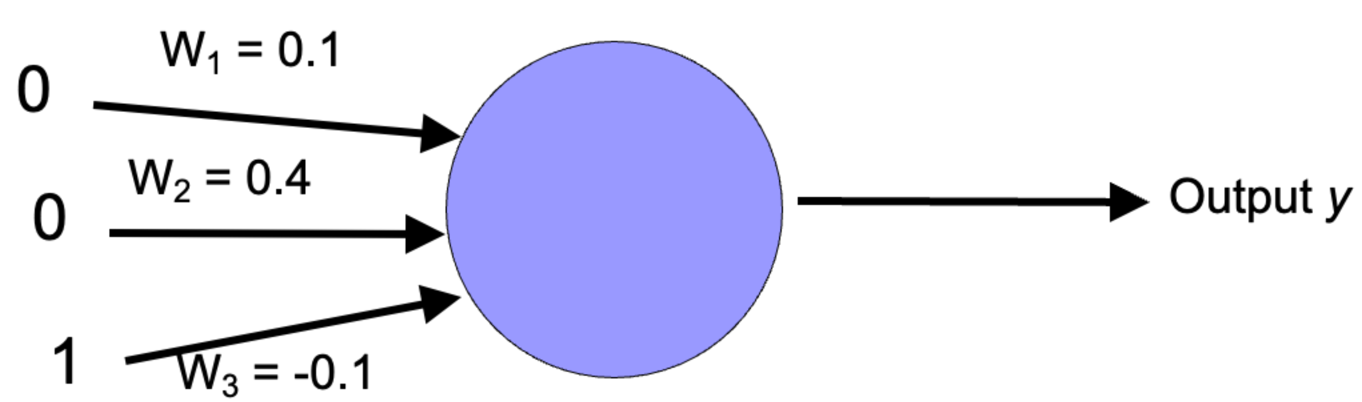
Perceptron learning

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?

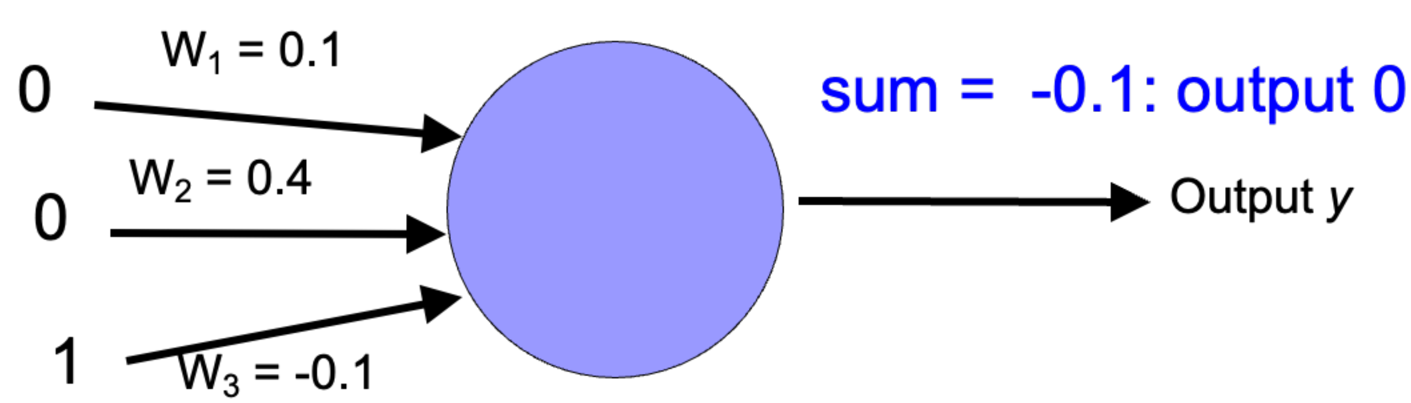
Perceptron learning

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:

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



Right. No update!

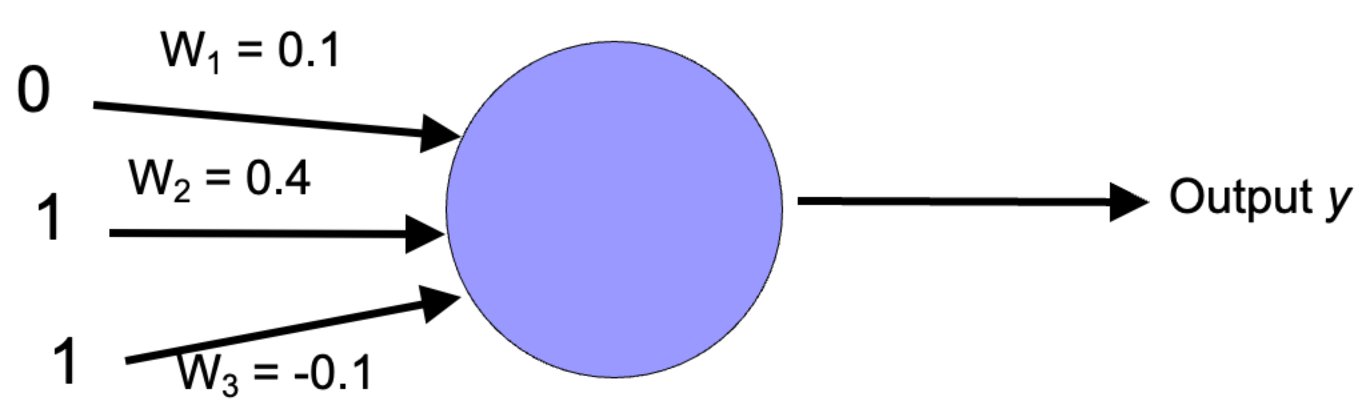
Perceptron learning

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?

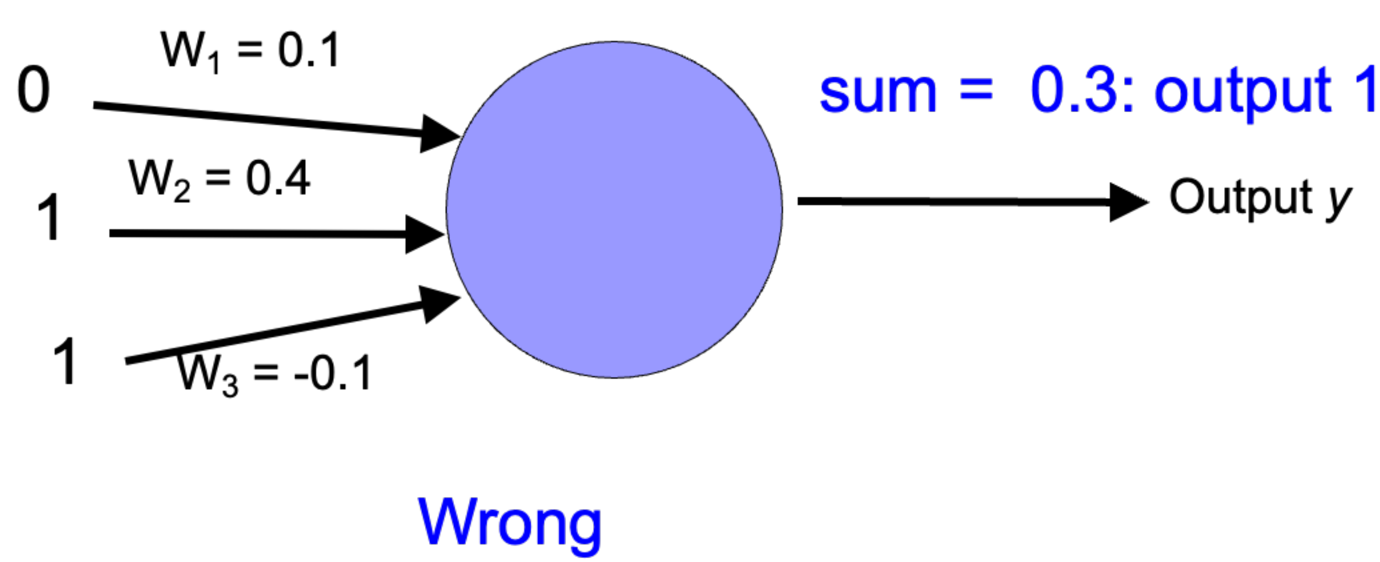
Perceptron learning

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:

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



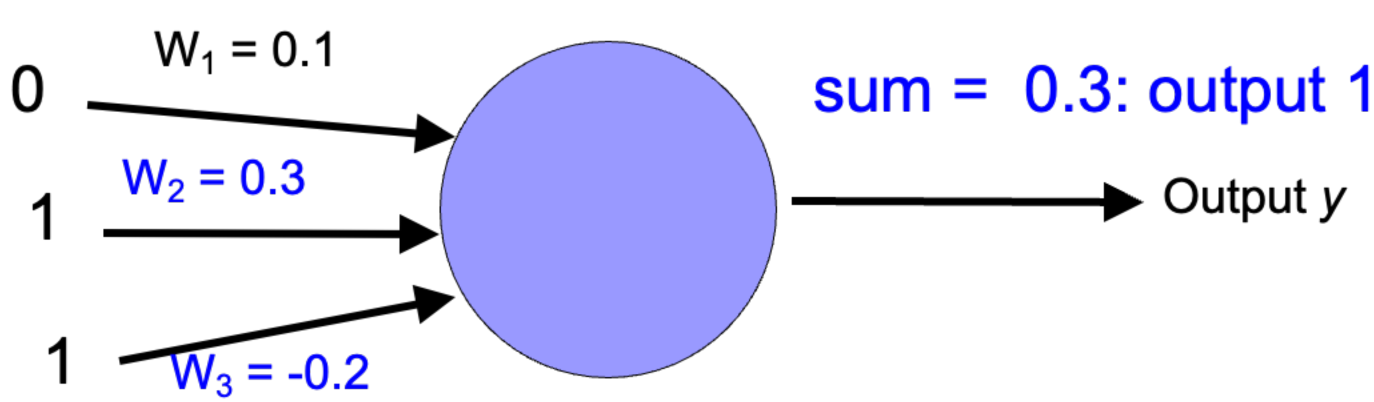
Perceptron learning

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

λ = 0.1

if wrong:

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



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

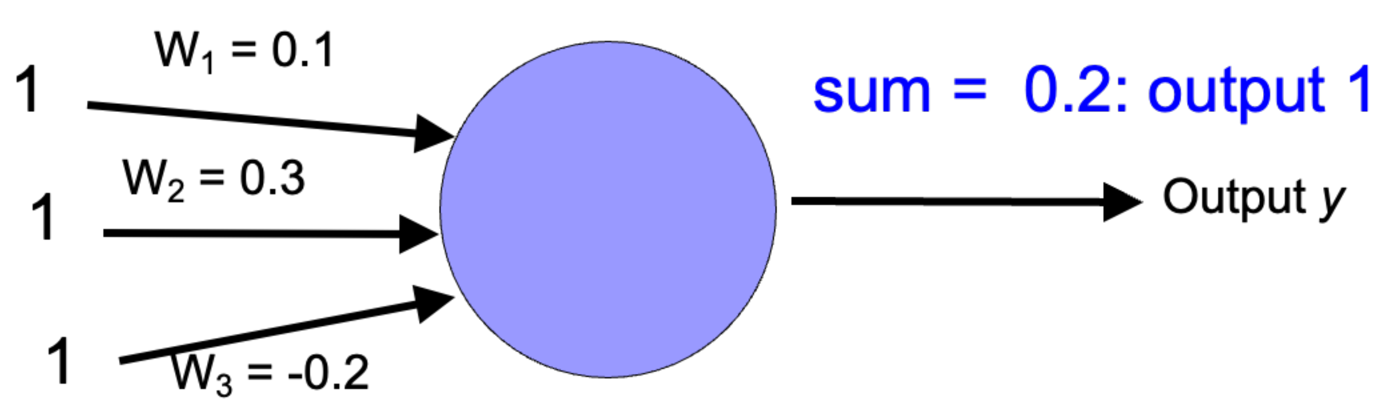
Perceptron learning

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:

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



Right. No update!

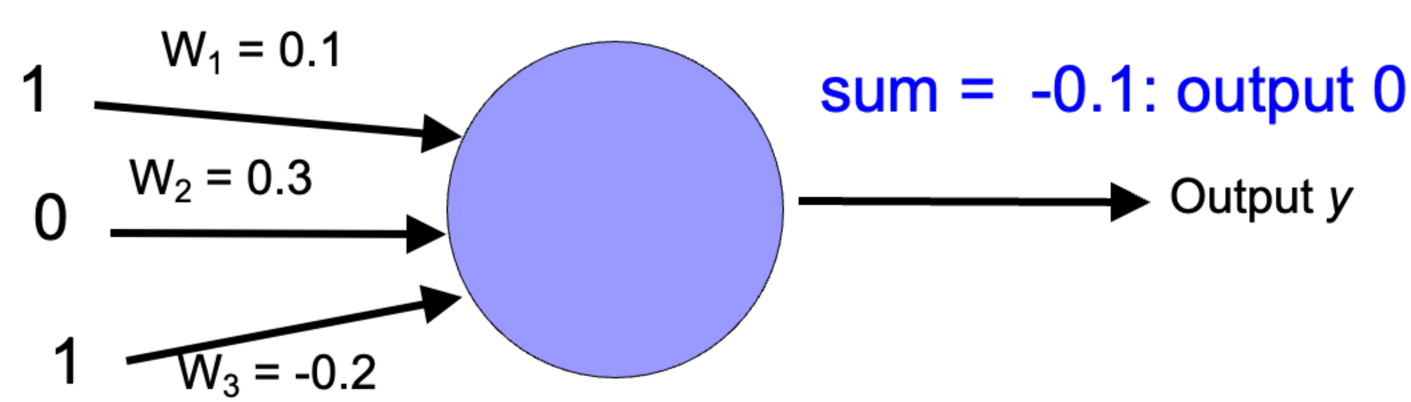
Perceptron learning

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:

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



Right. No update!

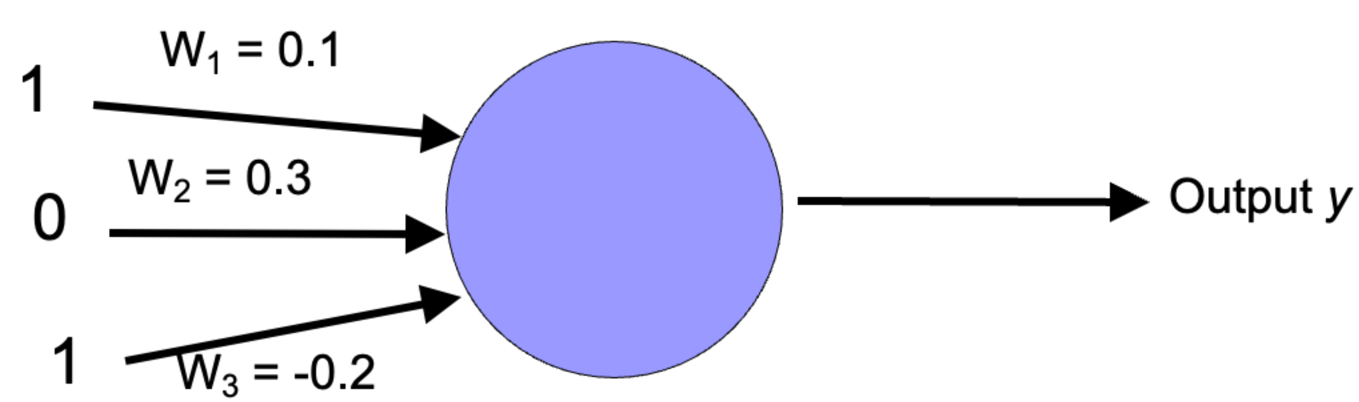
Perceptron learning

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:

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



Are they all right?

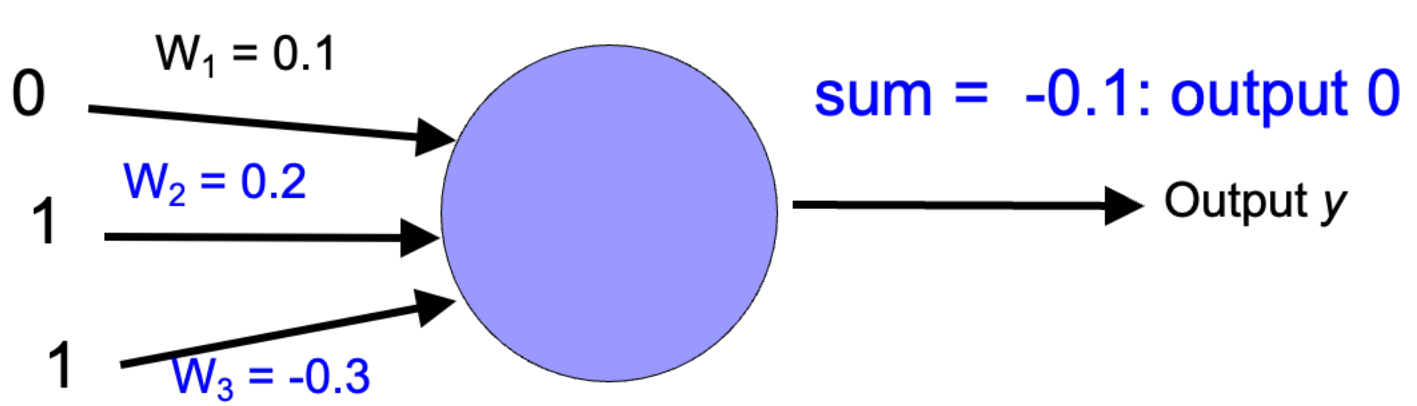
Perceptron learning

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:

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



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

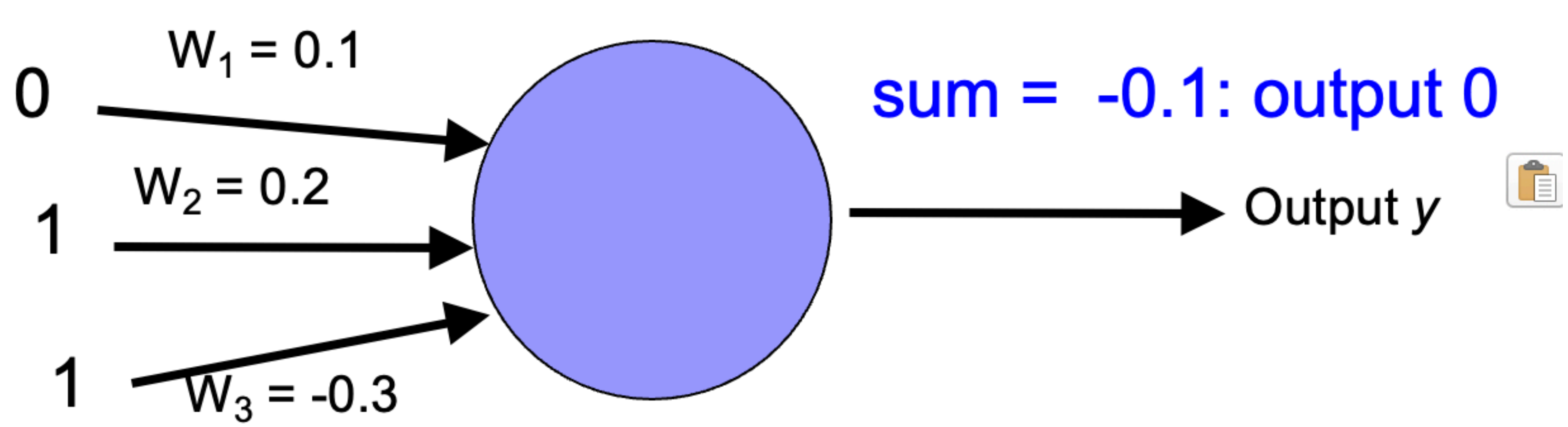
Perceptron learning

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:

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



Are they all right?

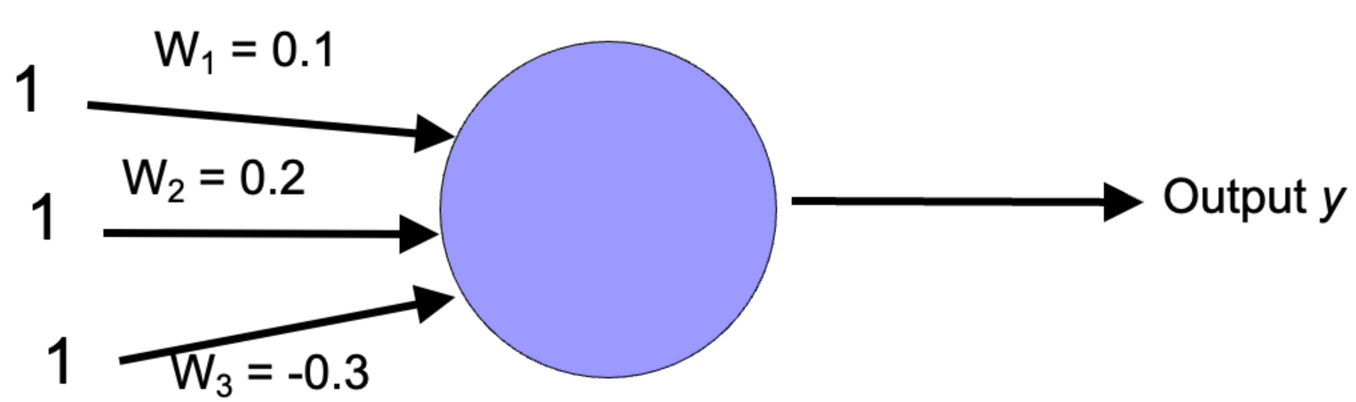
Perceptron learning

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:

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



We've learned AND!

Perceptron learning algorithm

- 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).

Lecture 13: Perceptron learning and back propagation

- Perceptron learning
- Back propagation

Linearly separable

- 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?

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

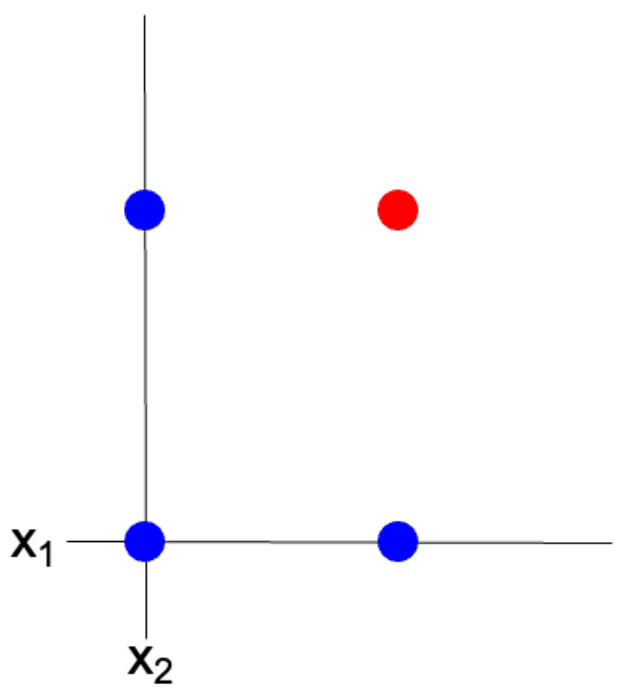
x_1	x_2	x_1 **xor** x_2
0	0	0
0	1	1
1	0	1
1	1	0

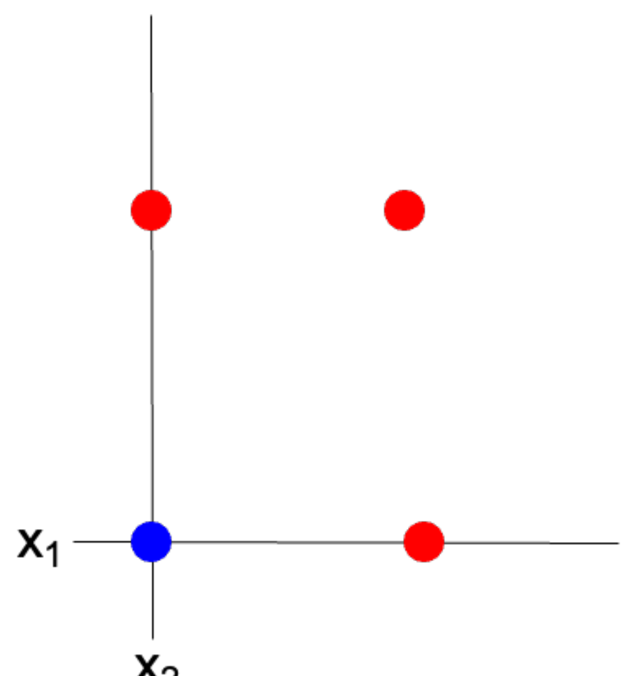
Which of these are linearly separable?

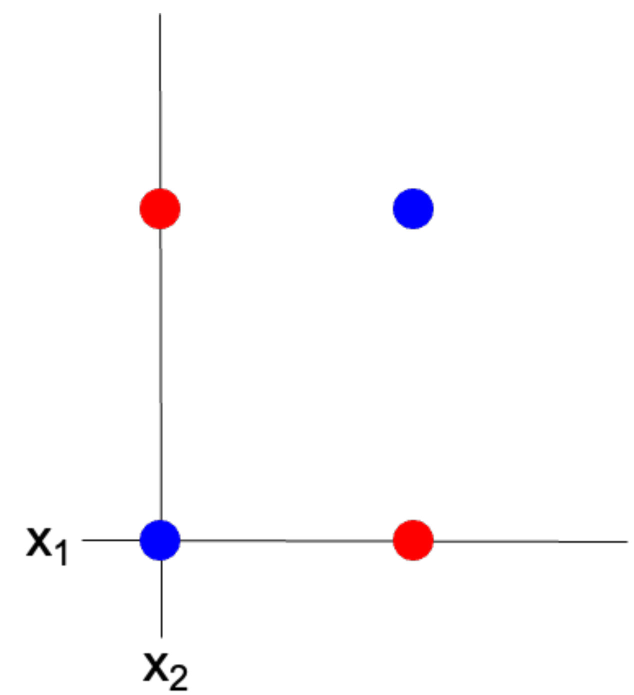
x_1	x_2	x_1 **and** x_2
0	0	0
0	1	0
1	0	0
1	1	1

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





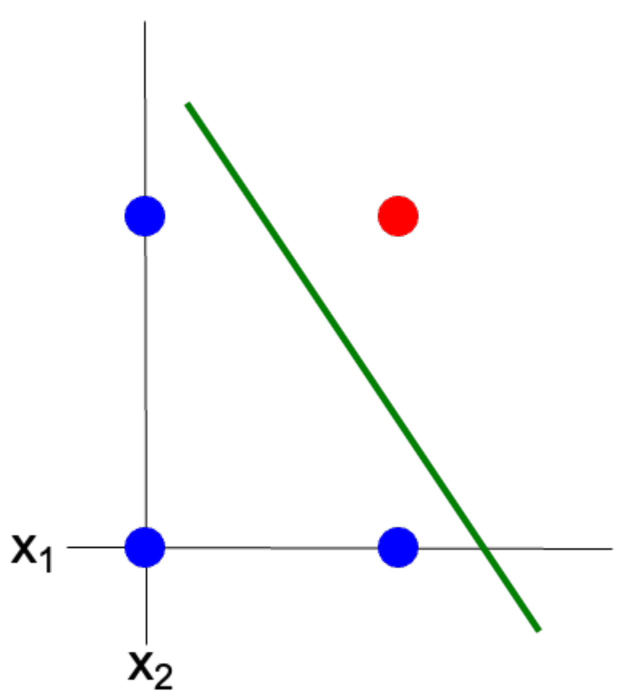


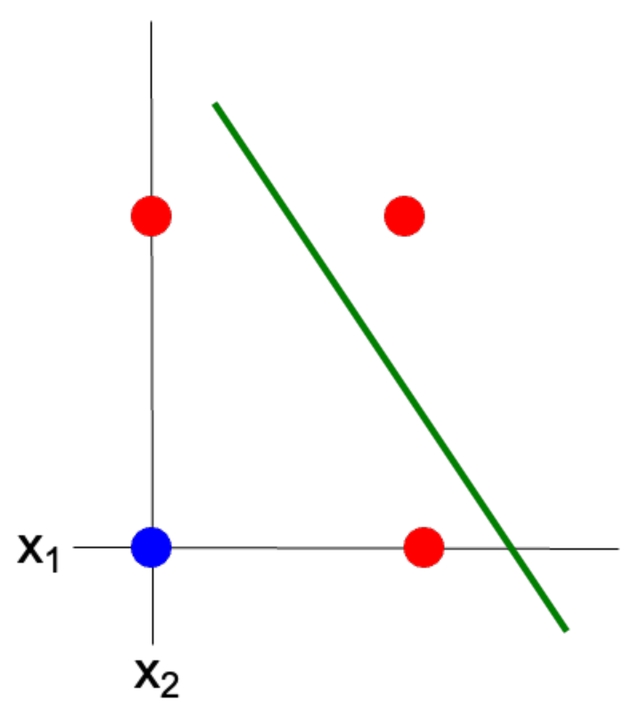
Which of these are linearly separable?

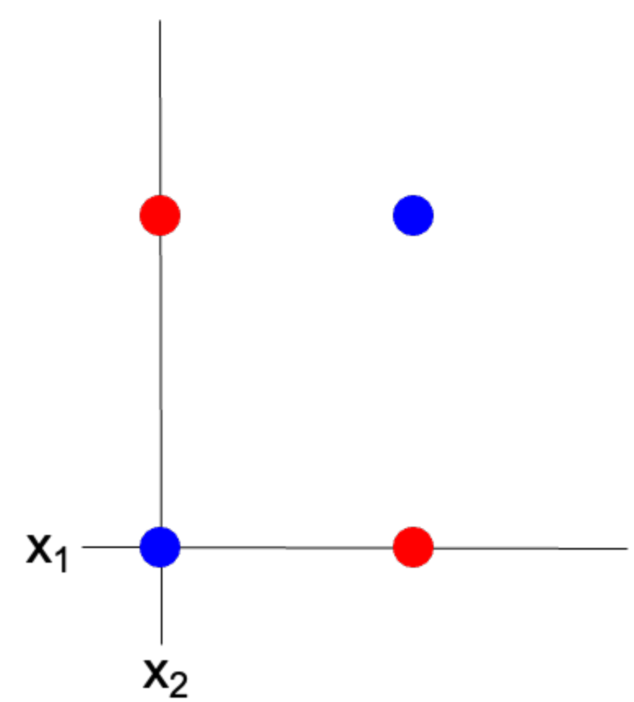
x_1	x_2	x_1 **and** x_2
0	0	0
0	1	0
1	0	0
1	1	1

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

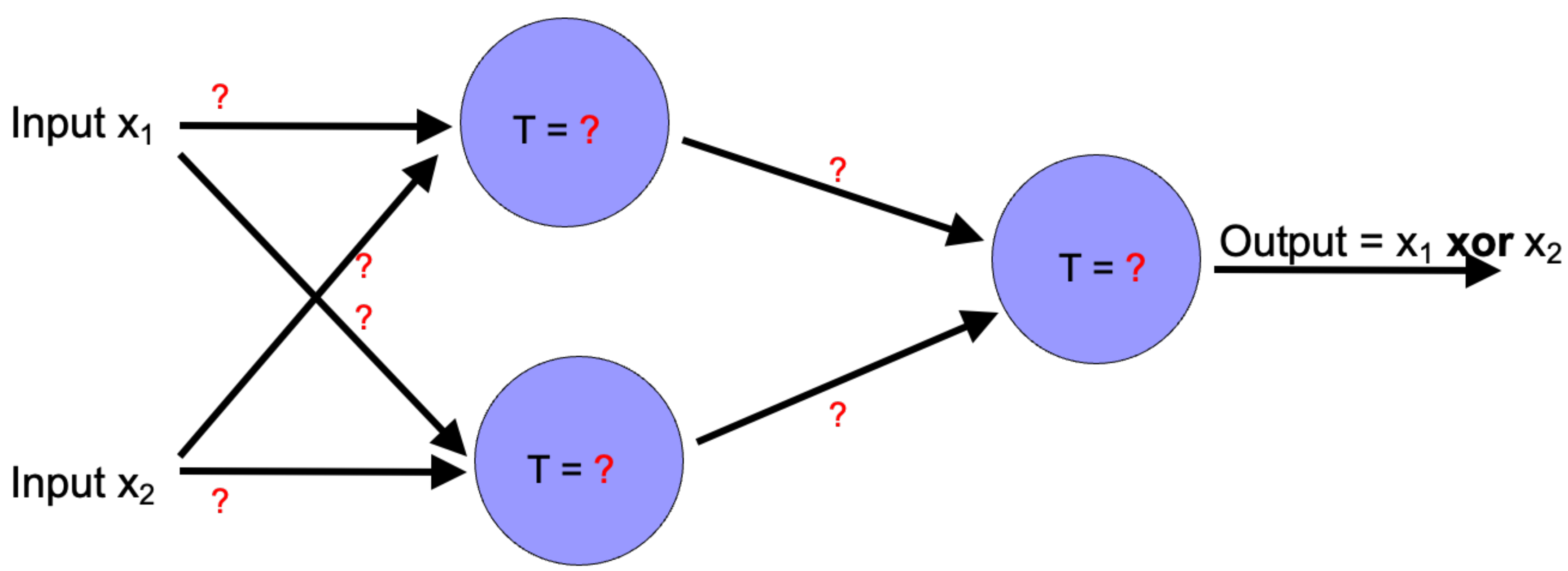






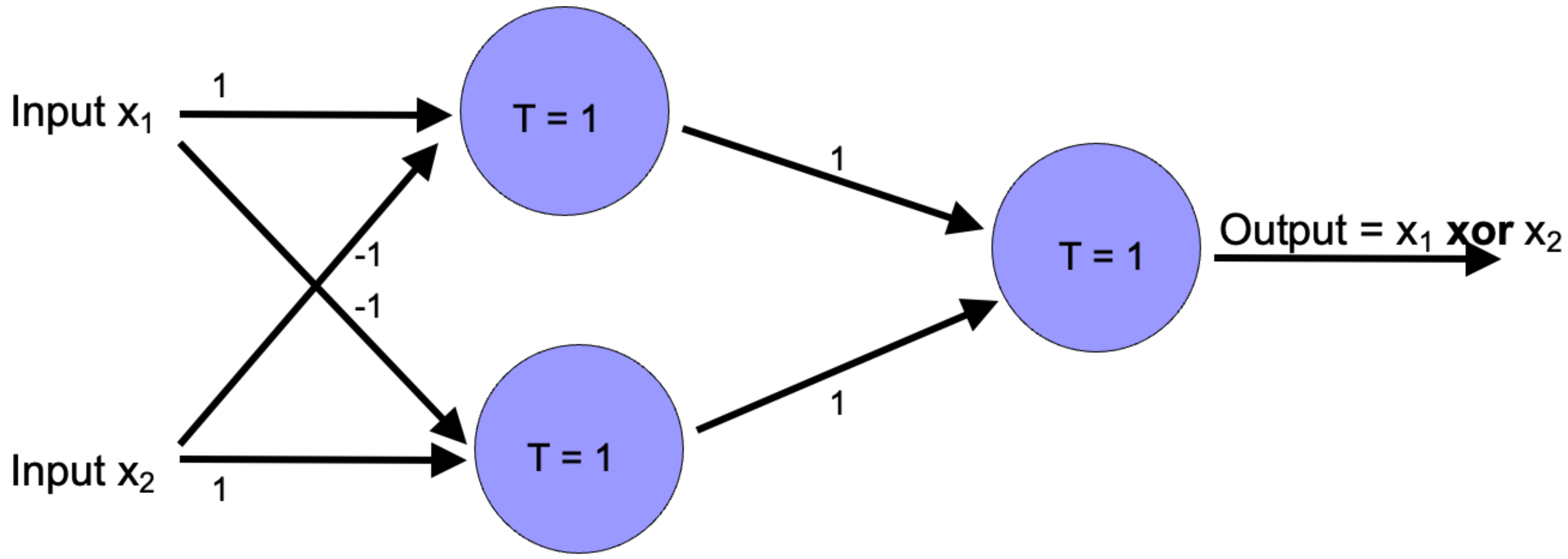
xor

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

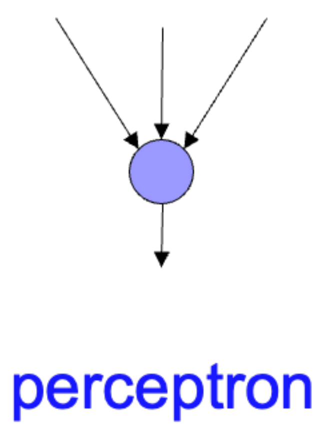


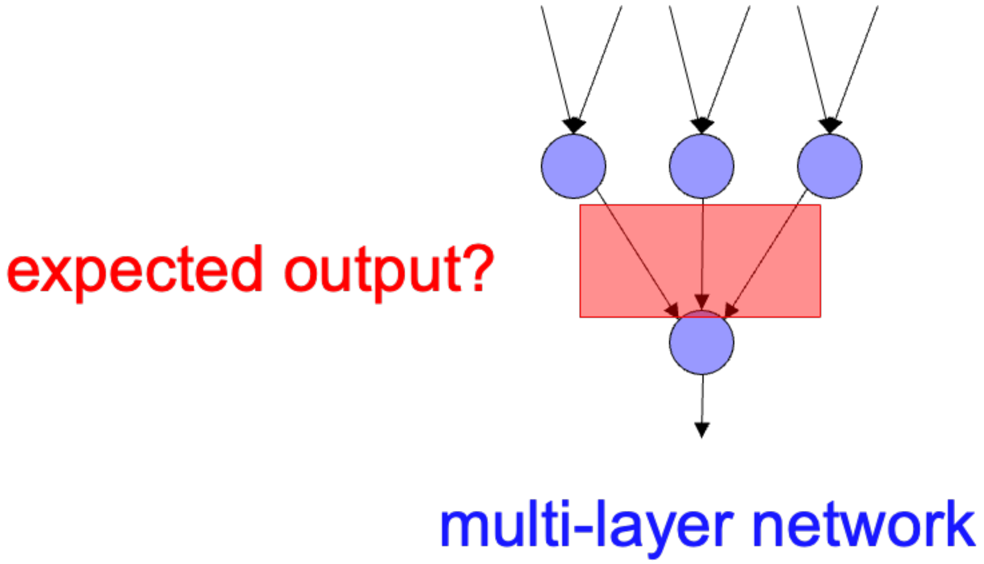
Learning in multilayer neural networks

- 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.

Challenge

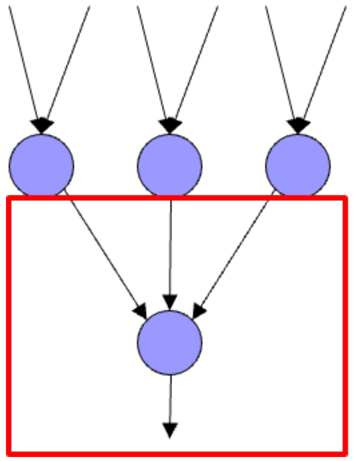
- for multilayer networks, we don't know what the expected output/error is for the internal nodes





Backpropagation

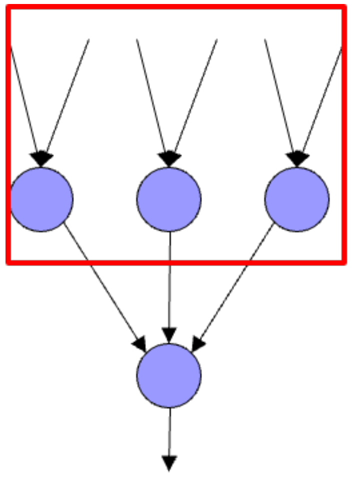
- Say we get it wrong, and we now want to update the weights



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



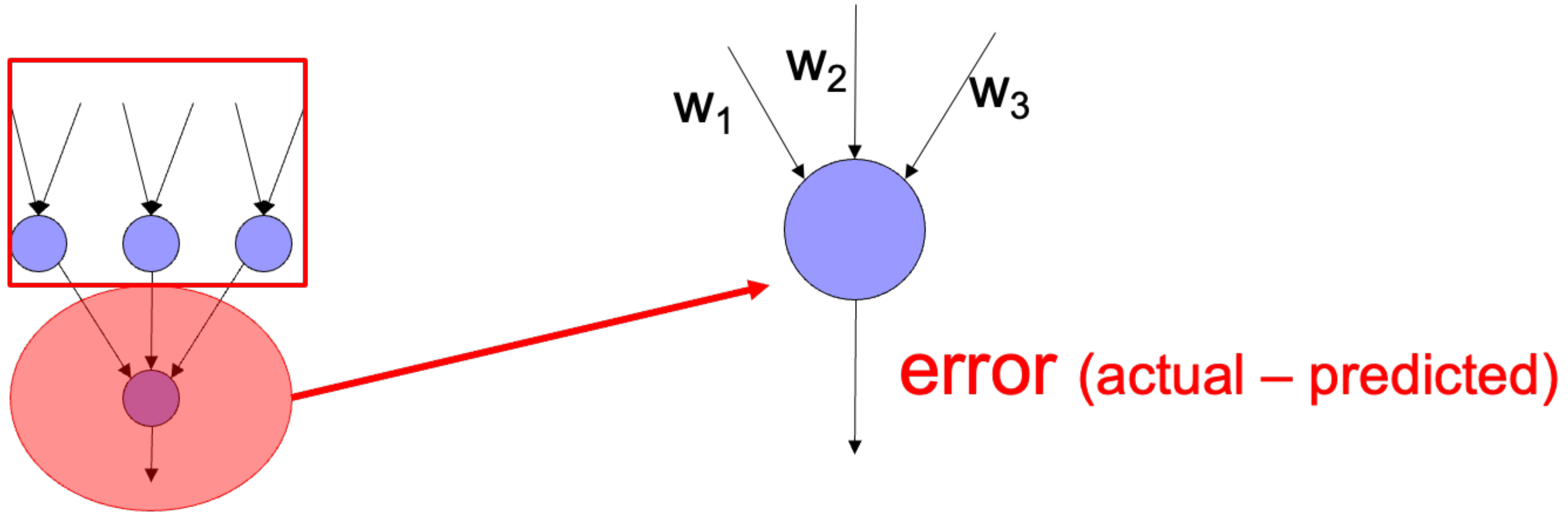
"back-propagate" the error (actual – predicted):

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

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

Backpropagation

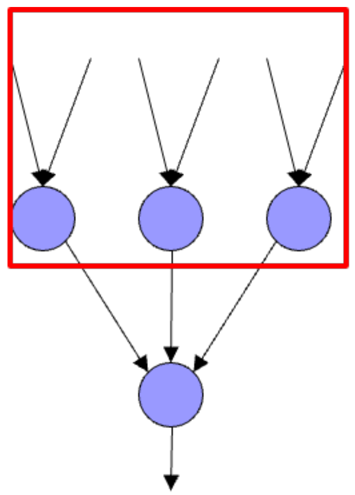
- Say we get it wrong, and we now want to update the weights



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.

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

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 data

Construct a new network:

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

constructor: constructs a new NN object

input nodes

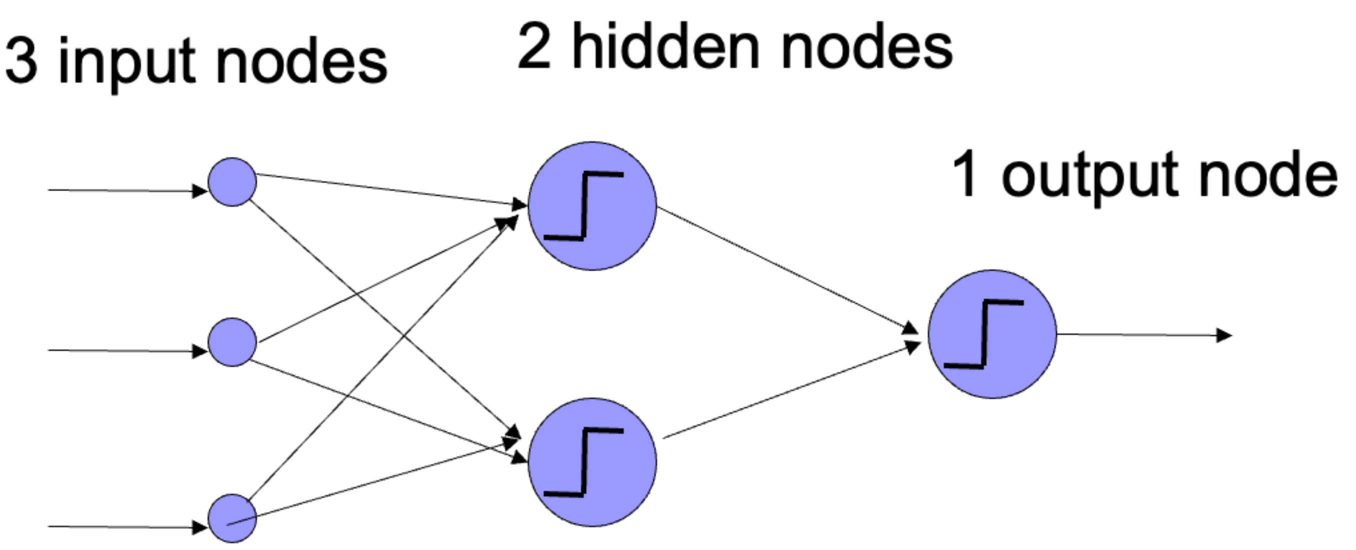
hidden nodes

output nodes

Training on data

Construct a new network:

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



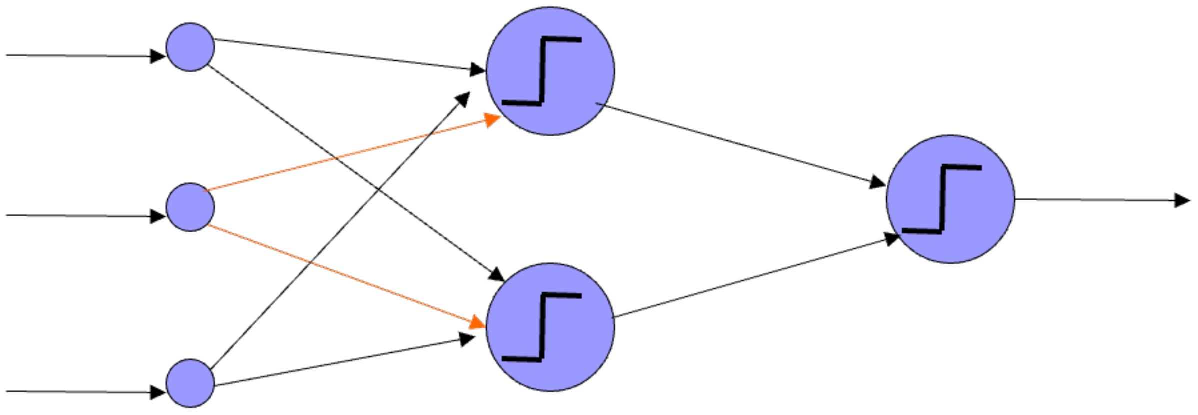
Training on data

```
>>> 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

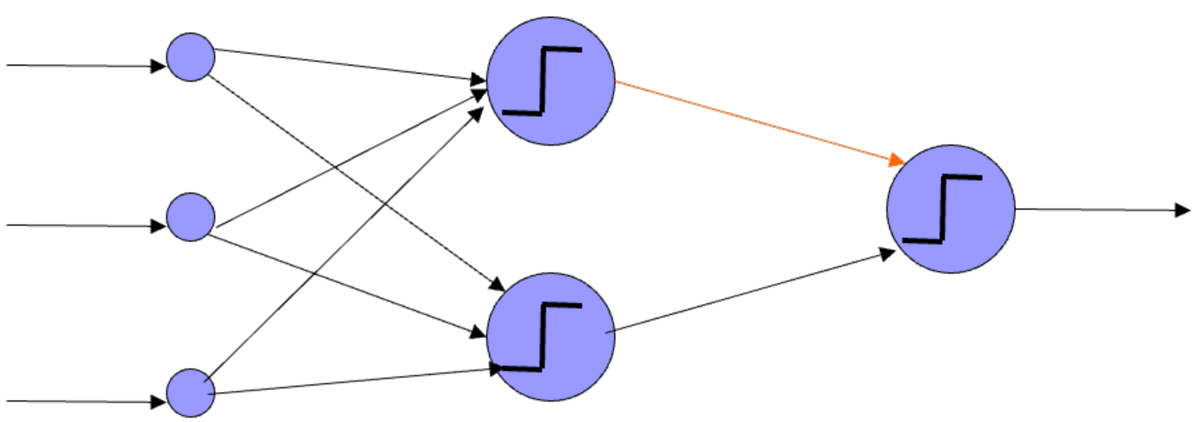
After training, can look at weights

```
>>> nn.train(table)
>>> nn.get_IH_weights()
[[ [w1a, w1b, w1c],
   [w2a, w2b, w2c] ],
   [b1, b2]]
```



After training, can look at 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

- optional_parameters.py
- 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)
```

Resources

- optional_parameters

Homework

- No homework for the week
- Sign up for group presentations