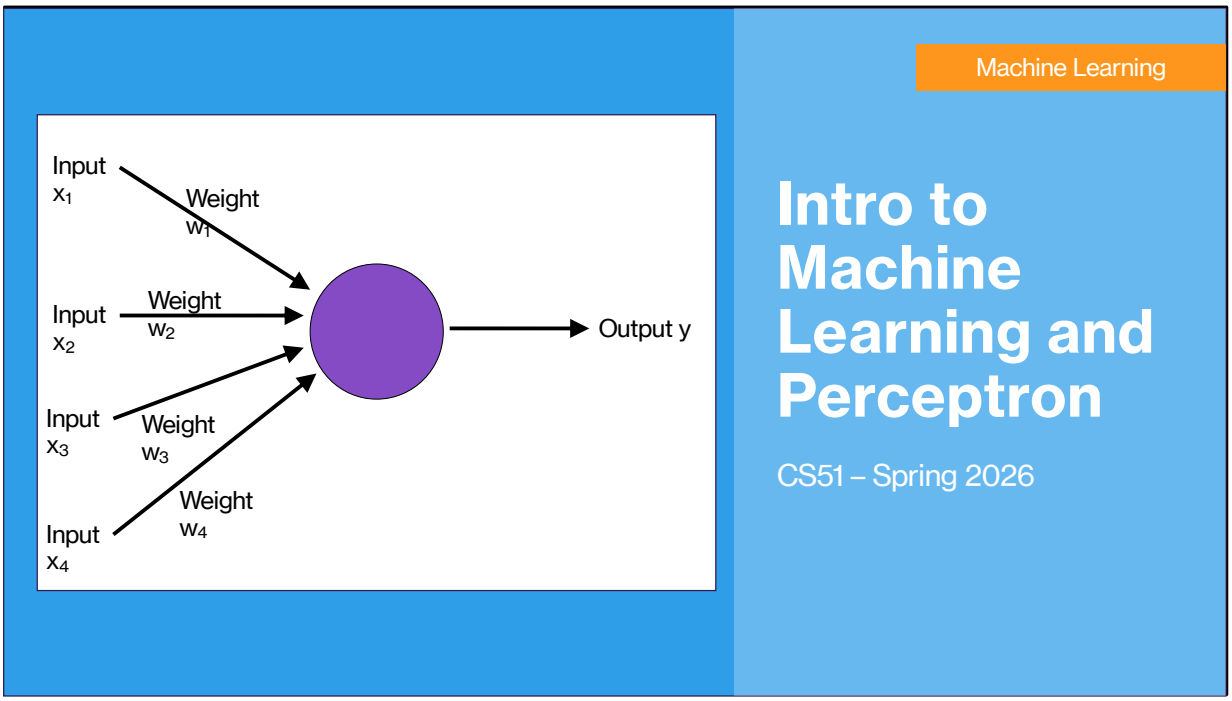
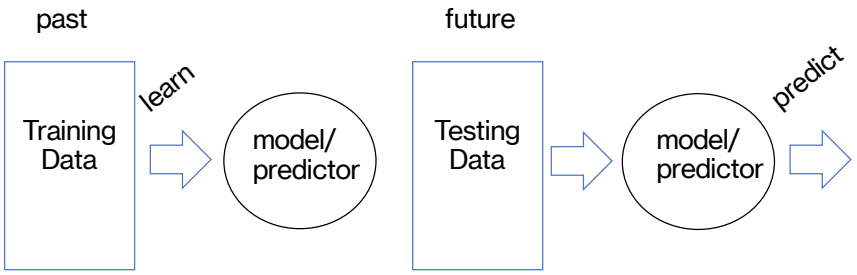


Welcome to the last unit of the class where we will learn about machine learning, a subfield of computer science that is extremely active and behind all the AI developments that are in the news.

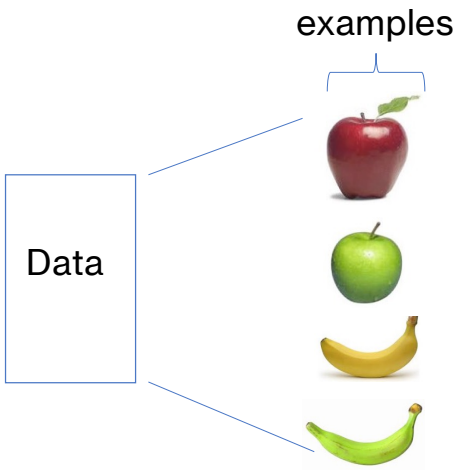
Machine Learning



2

Machine learning as a field studies algorithms that learn from past data and experiences in order to predict the future. The data the algorithms learn from are called training data. We build a model or predictor that learns the training data with the goal that in the future, when we are given new data we can use the prior knowledge. The new data are called testing data. They are passed on the model/predictor we have trained on prior data and make predictions.

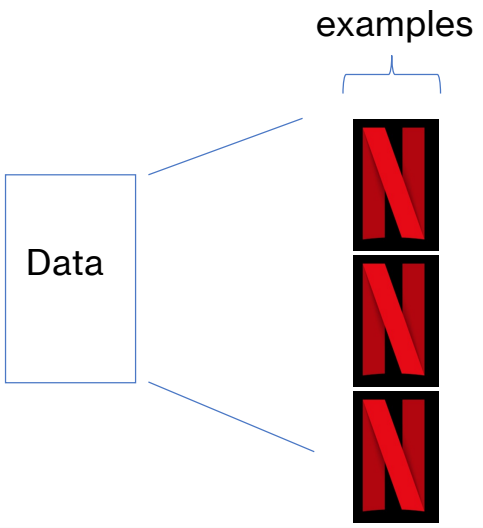
Data



3

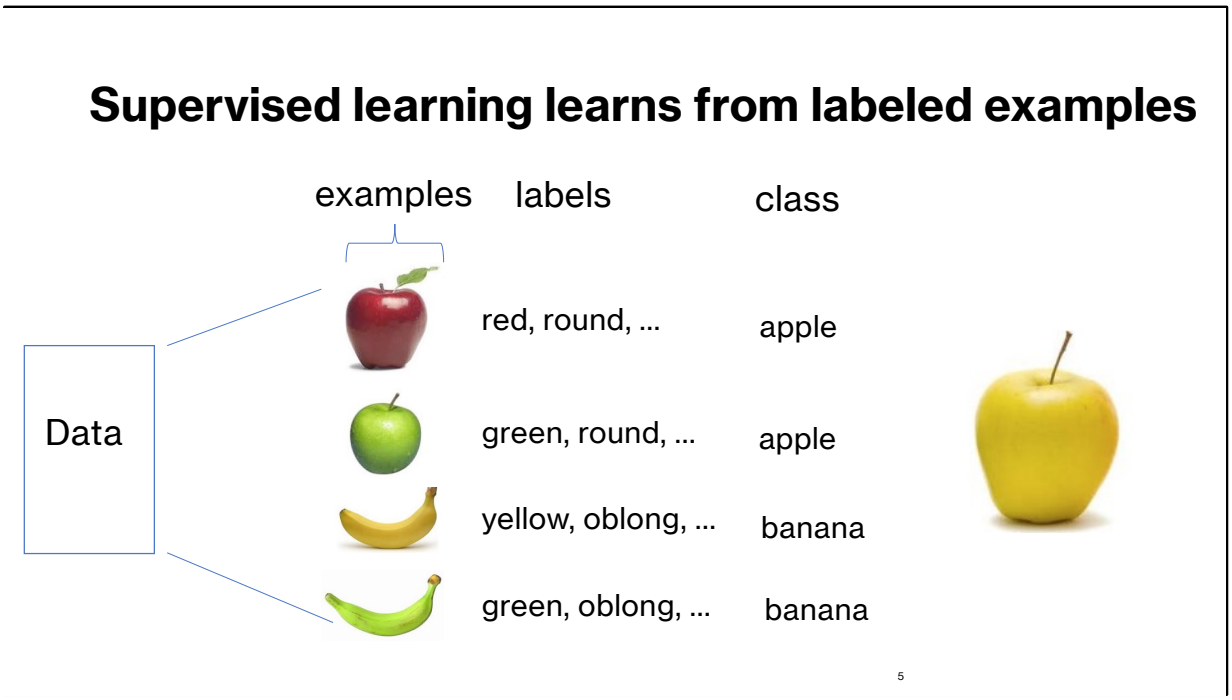
For example, we might have a dataset with picture, let's say of fruit, and we build a model that learns to differentiate the pictures of different fruit. If in the future we are given a new picture, we can recognize whether it is a fruit and maybe even predict what kind of fruit it is.

Data



4

Or maybe we have a model that learns from past movies we have liked watching on Netflix and makes recommendations about movies we are likely to enjoy in the future. These are just two examples of machine learning algorithms but there are SO many applications out there.



The are different flavors of machine learning algorithms. The most common one is supervised learning where the algorithm learns from labeled examples. Let's take our fruit picture dataset. For each picture, we would have different labels that characterize the picture. E.g., the color and shape. For each example, we also have done somehow the hard work of assigning it to a class, let's say the class here represents the type of fruit, apple or banana. We train our model on this dataset which learns a relationship between labels and classes. When in the future we get a picture of a fruit we have never seen before, we ask the model to predict what kind of fruit it is based on its own set of labels, e.g., yellow, round. This example of supervised learning is called classification because we classify examples with a finite number of labels into a finite and already agreed upon number of classes. As you can imagine, it can be quite costly to build datasets that are perfectly labeled and properly classified in order to build good classifiers.

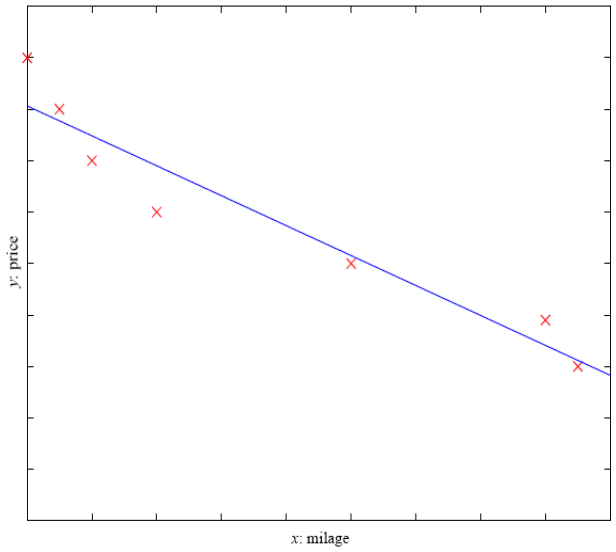
Classification

- Predicts a discrete category (e.g., yes/no, or a discrete number of classes)
- Face recognition
- Character recognition
- Spam detection
- Medical diagnosis
- Biometrics

Classification has a lot of applications. Face recognition is an increasingly popular one where individuals are recognized in video frames or pictures. As we have seen during the first week of class, this application is rife with racial biases. To the more innocuous side, character recognition refers to the process of recognizing non-digital text, e.g., a scanned copy of a book, or signs when walking in a new city. Spam detection is an application that we rarely think of when our email filters work well but can have catastrophic financial consequences if not successful. Medical diagnosis is another rising area of applying machine learning. Finally, biometrics where information like fingerprints, iris, etc, are used for security purposes is another example of classification.

Regression

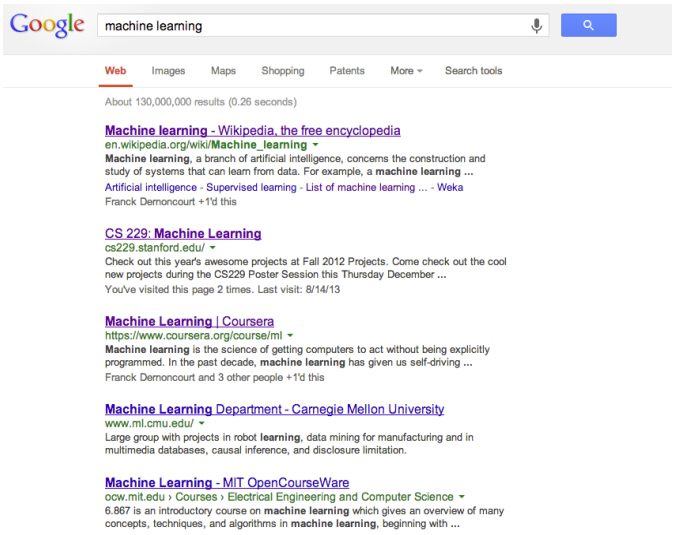
- Predicts a continuous numerical value
- Economics/Finance
- Epidemiology
- Car/plane navigation
- Temporal trends



Another type of supervised learning is regression where the outcomes are real-valued. For example, the training data could have labels like a car's mileage and the model could learn the price. When trying to buy a car, we can look into a car's mileage and see what price is reasonable. Regression examples can be found in economics and finance, e.g when predicting the value of stocks, in epidemiology, when predict outbreaks, in navigation, e.g., with autonomous cars, in temporal trends such as weather forecast and so on.

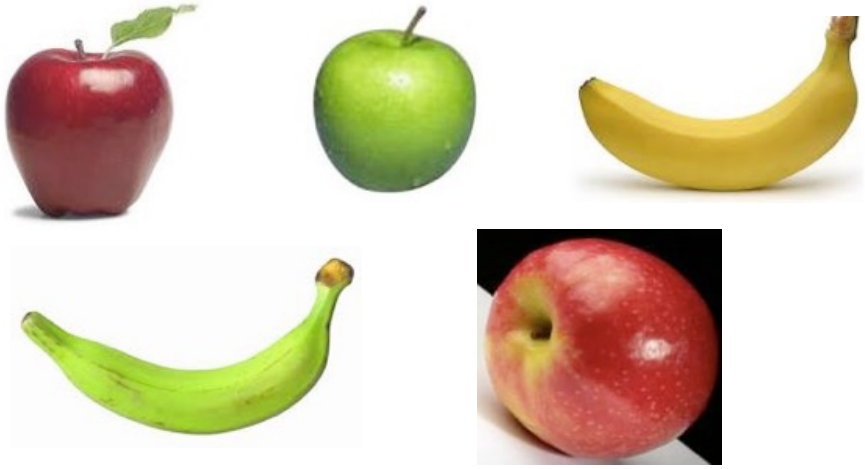
Ranking

- Predicts a ranking of items
- Web search
- Movie/music recommendation
- Flight search



Finally, ranking algorithms are also part of supervised learning where the predicted outcome is a ranking of items, e.g., of web pages based on their relevance to a query, movie or music recommendations that are most likely to align with a user's taste, or flight search recommendations that rank flight options.

Unsupervised learning



- Provided with data but not labels

Other than supervised learning, there is unsupervised learning where we have a dataset to learn from but although we are provided with data, we are not given any labels. Think of it as being a given a bunch of pictures of fruit where you don't know which pictures are apples and which bananas. There are a lot of examples of unsupervised learning algorithms, e.g., clustering algorithms which build groups based on similarities of features or ways to segment customers etc

Reinforcement learning

left, right, straight, left, left, left, straight	GOOD
left, straight, straight, left, right, straight, straight	BAD
left, right, straight, left, left, left, straight	18.5
left, straight, straight, left, right, straight, straight	-3

Given a sequence of examples/states and a reward after completing that sequence, learn to predict the action to take in for an individual example/state

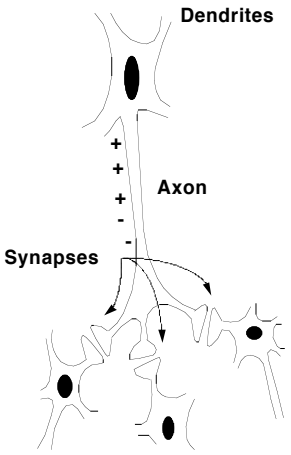
Finally, with reinforcement learning, we are given a sequence of examples/states and some utility function which represents a reward loss. We learn which examples lead to good outcomes and try to maximize those. Reinforcement learning is behind a lot of robotics research.

Neural networks

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

Lately, much of the research activity in machine learning is driven by neural networks that are behind deep learning. Today, we will explore the foundations of artificial networks which draw the inspiration from biology, something that researchers have loved trying in their search for building artificial intelligence.

Our nervous system according to CS



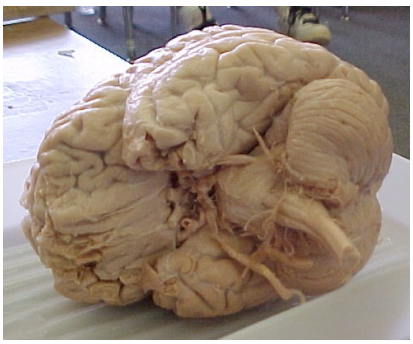
Neuron

- the human brain is a large collection of interconnected neurons
- a neuron is a brain cell
  - they collect, process, and disseminate electrical signals
  - hey are connected via synapses
  - they fire depending on the conditions of the neighboring neurons

I am going to fudge a bit the details but neurons are the basic working unit of the brain, and are designed to transmit transmit information to other nerve cells,

muscle, or gland cells. Most neurons have a cell body, an axon, and dendrites. The axon extends from the cell body and often gives rise to many smaller branches before ending at nerve terminals. Dendrites extend from the neuron cell body and receive messages from other neurons. Synapses are the contact points where one neuron communicates with another. The dendrites are covered with synapses formed by the ends of axons from other neurons. Dendrites bring information to the cell body and axons take information away from the cell body. Computer scientists don't really care about these details. In our view, the human brain is a large collection of interconnected neurons which are connected via synapses and process electrical signals. A neuron may fire depending on the conditions of the neighboring neurons.

Our brains in numbers



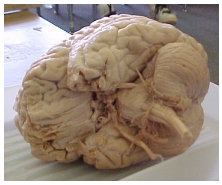
The human brain

- contains ~10^{11} (100 billion) neurons
- each neuron is connected to ~10^{4} (10,000) other neurons
- Neurons can fire as fast as 10^{-3} seconds

How does this compare to a computer?

Here are some cool numbers about our nervous system. The human brain contains 100 billion neurons and each neuron is connected to about 10,000 other neurons. Nuerons can fire as fast as 10^-3 seconds. How does a brain compare to a computer?

Human Vs. Machine



10^{11} neurons
10^{11} neurons
10^{14} synapses
10^{-3} “cycle” time

10^{10} transistors
10^{11} bits of RAM
10^{13} bits on disk
10^{-9} cycle time

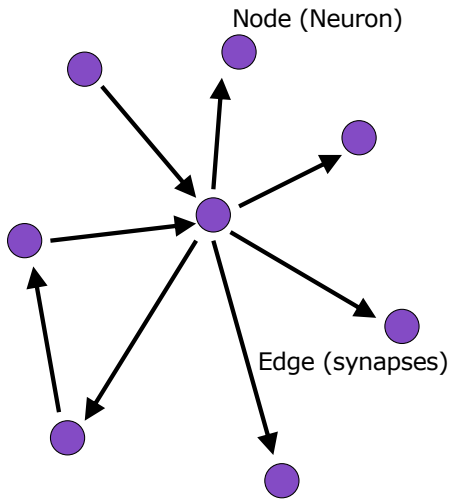
You can see that there are parallels between neurons and physical components like transistors and memory. Our brains are slower than computers.

Brains are still pretty fast

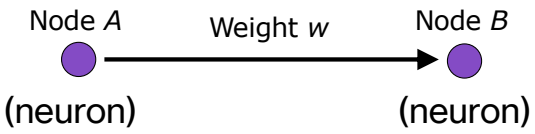
Who is this?

Still, brains are quite fast. As I flashed this picture, how quickly did you think it's Lebron James? In less than second.

Artificial neural networks



We use our basic biological understanding of the brain to approximate artificial networks where nodes are neurons and edges are synapses.

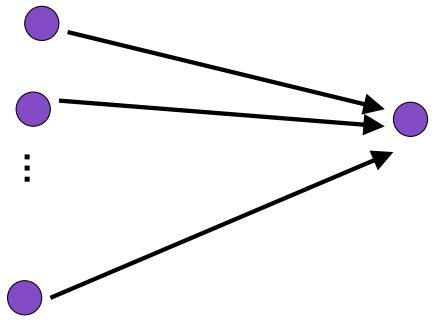


W is the strength of signal sent between A and B.

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

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

If we have two neurons, A and B, connected by an edge with a weight w that represent the signal strength from A to B, we say that if A fires and w is positiven then A stimulates or excites B. If A fires and w is negative, then A inhibits B.

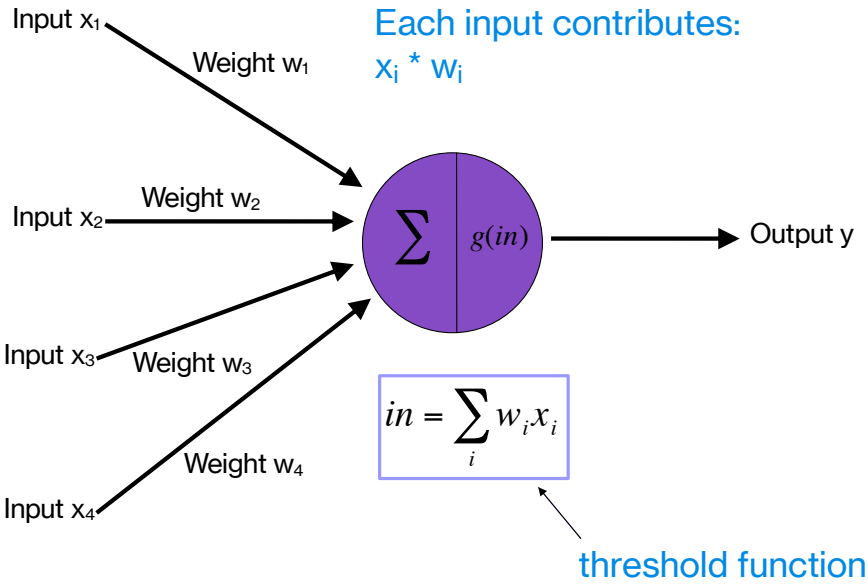


A given neuron has many, many connecting, input neurons

If a neuron is stimulated enough, then it also fires

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

A single neuron/perceptron

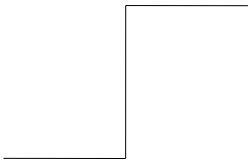


In machine learning, the perceptron is a single neuron supervised learning algorithm that works on a number of inputs with certain weights, sums these weighted inputs and if it is above a threshold it fires.

Possible thresholds

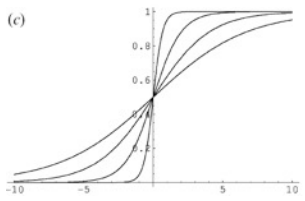
hard threshold

$$g(x) = \begin{cases} 1 & if \ \text{x} \geq threshold \\ 0 & otherwise \end{cases}$$



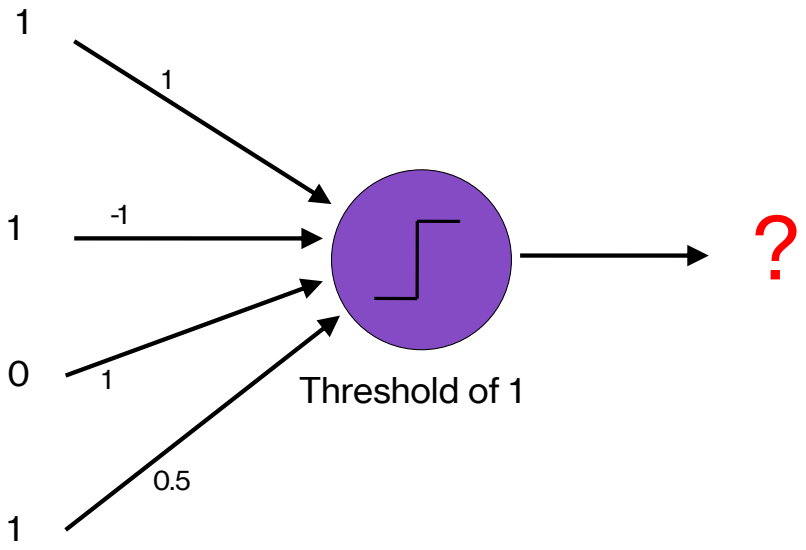
sigmoid

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



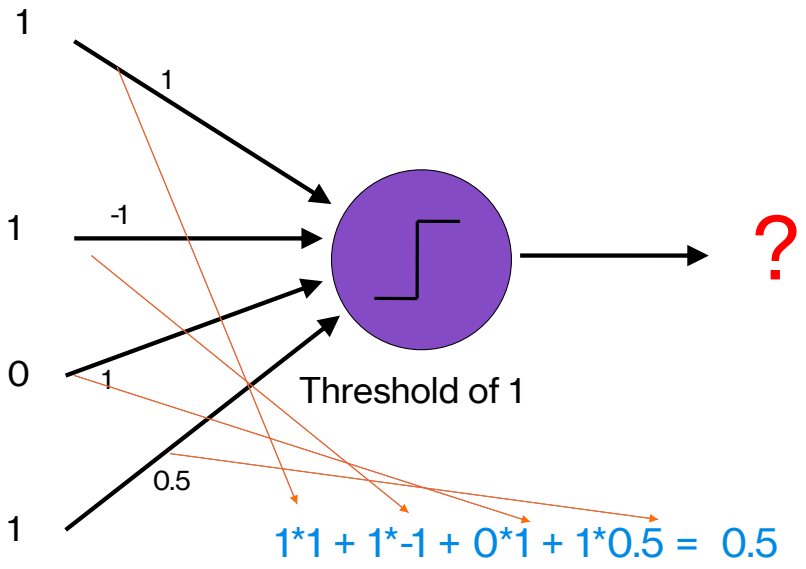
There are many ways to define thresholds, e.g., we could have a hard threshold where the output is either 1 or 0 based on whether we meet the threshold or we could be following something more complicated like a sigmoid curve. We will assume the former.

A single neuron/perceptron



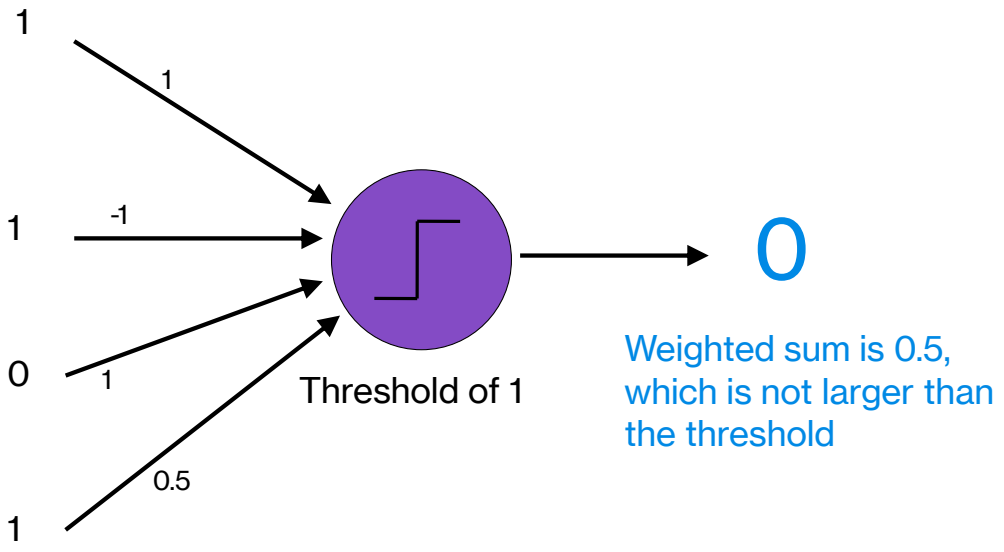
Let's say we have a neuron with four inputs and their associated weights shown on top of the edges. If the threshold is 1, will this neuron fire?

A single neuron/perceptron



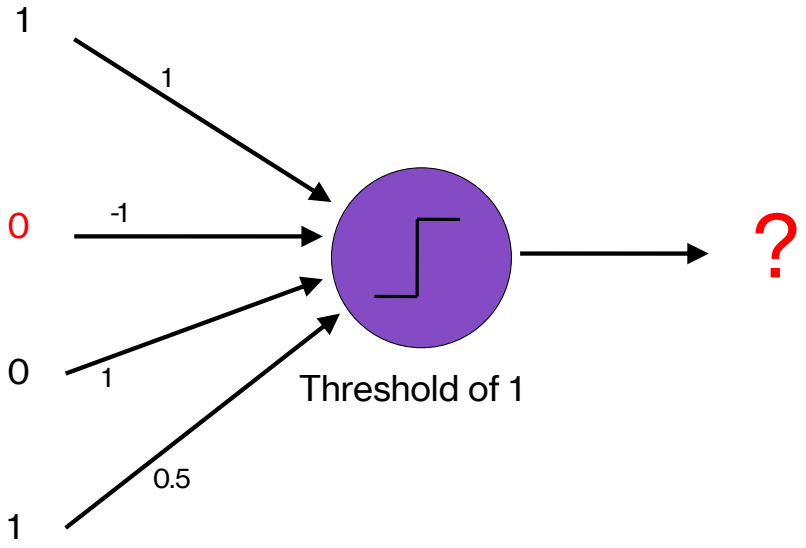
We would have to see if the weighted sum of the inputs is above or equal to 1.

A single neuron/perceptron



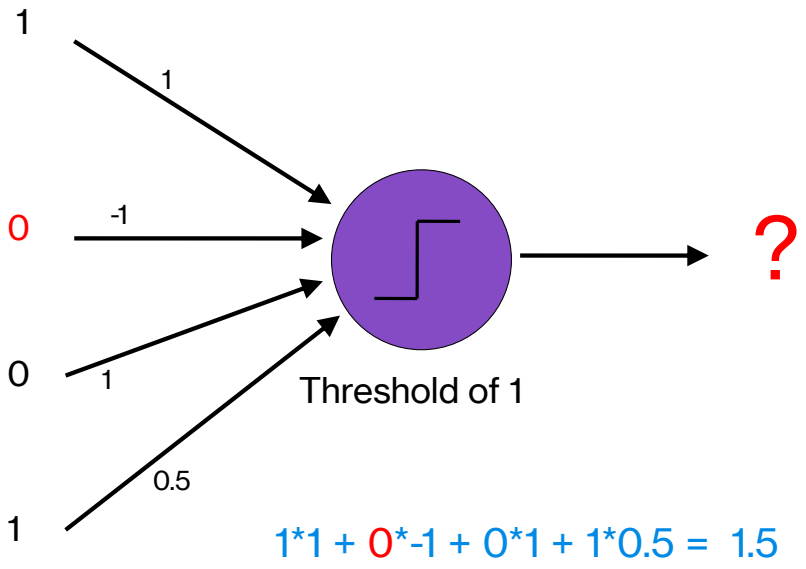
Since it's not, the neuron doesn't fire

A single neuron/perceptron



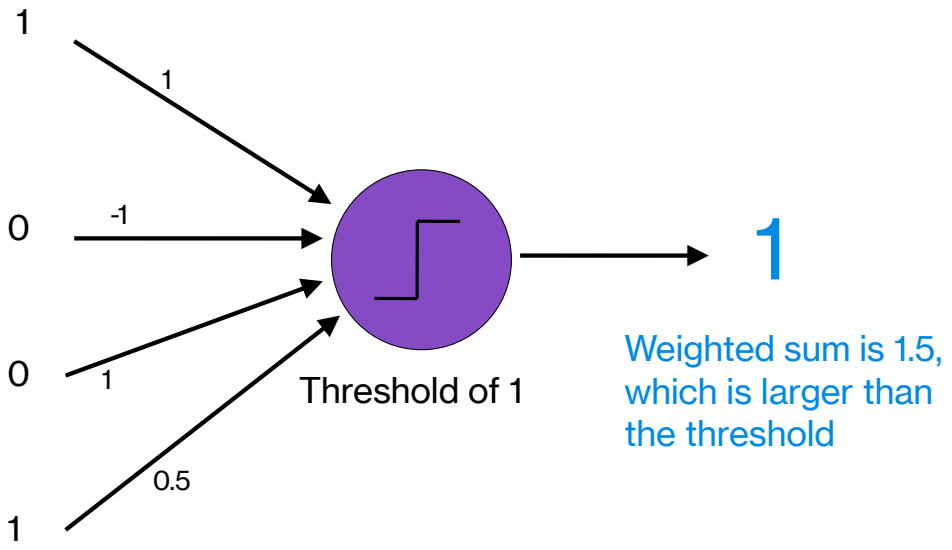
What if we change the second input to 0?

A single neuron/perceptron



This small change brought the sum to 1.5

A single neuron/perceptron



which meets the threshold and the neuron fires

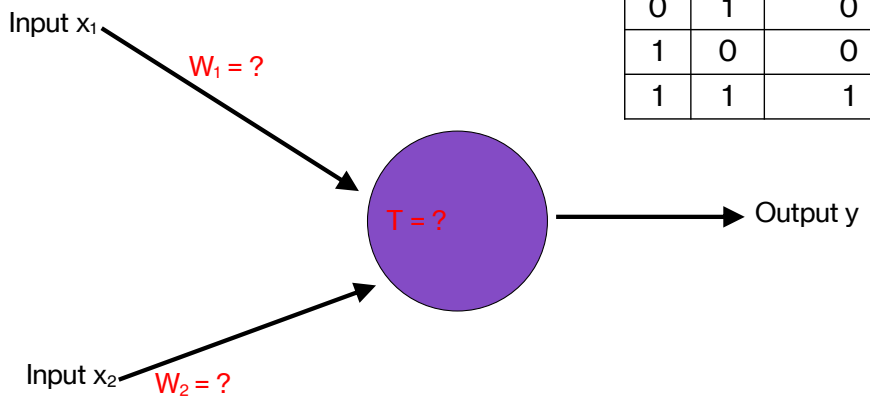
AND logical operator

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

If you don't remember, here is the truth table for the AND operator.

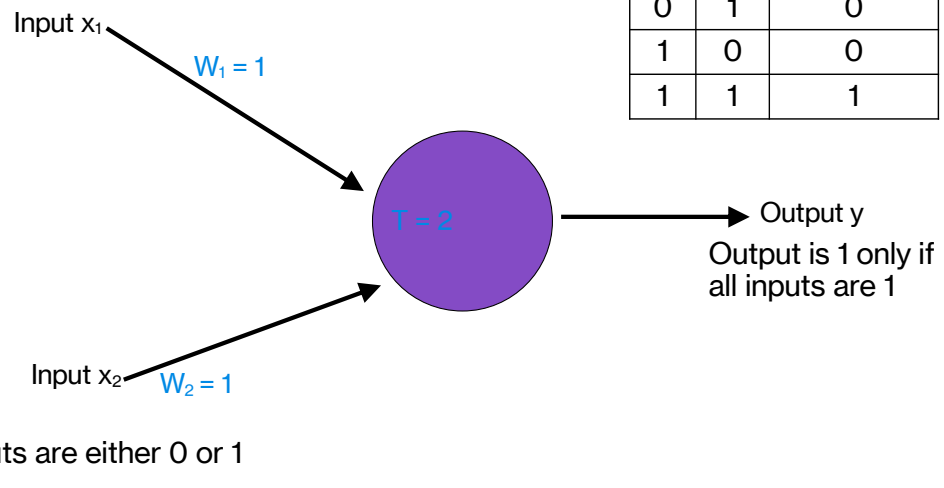
AND logical operator

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



If we assume that all inputs are either 1 or 0 (on or off) and this neuron is activated when it is 1 or on, what weights do we need to assign to these two edges and what theshold do we need to put to this neuro to represent the AND gate?

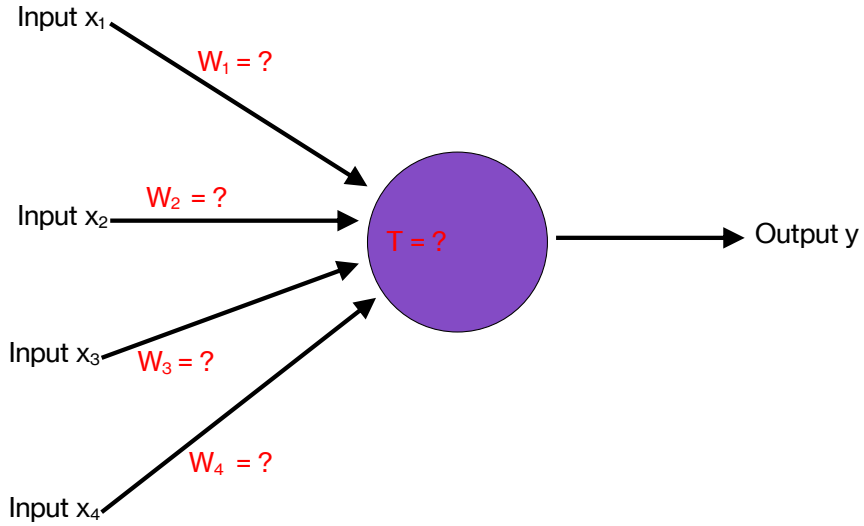
AND logical operator



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

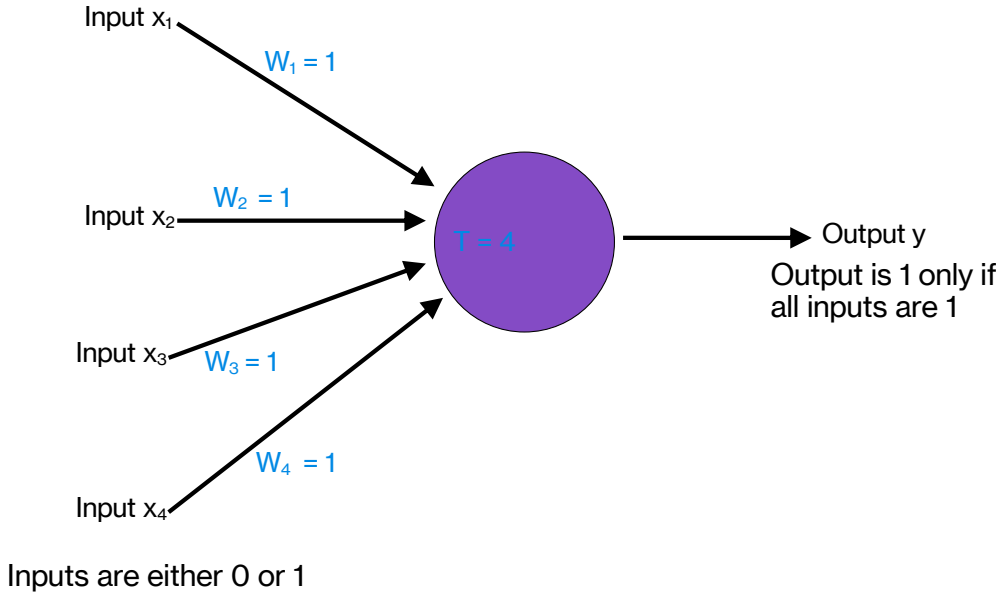
If they both have weights of 1 and the threshold is 2 then we represent the AND gate

AND logical operator



What if we extend this AND gate to four inputs. What weights and threshold would we need?

AND logical operator



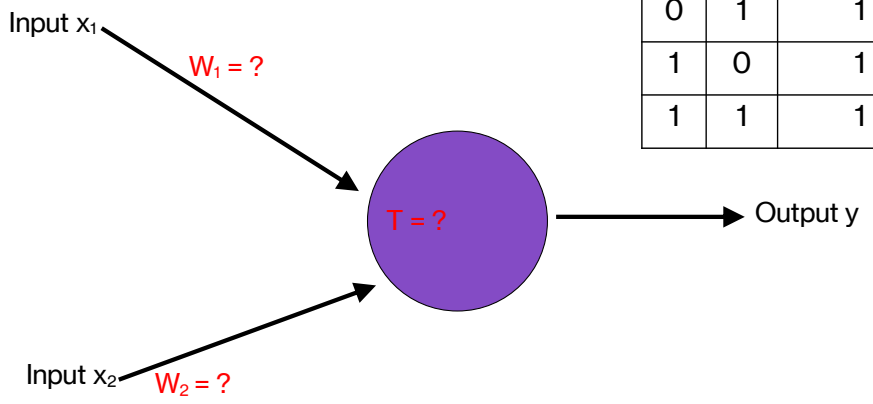
If they all have a weight of 1 the threshold would need to be 4.

OR logical operator

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

Let's try to learn the OR logical operator

OR logical operator

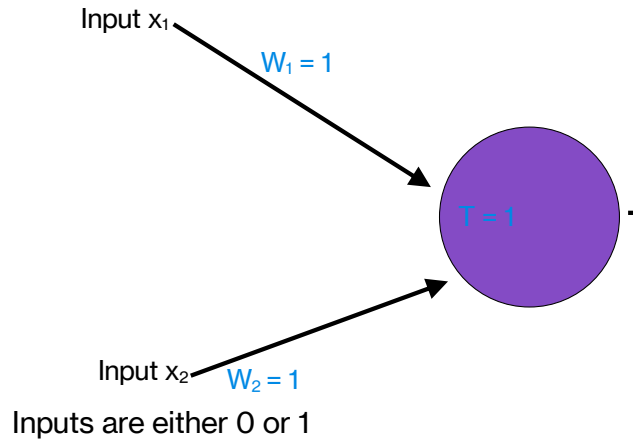


x_1	x_2	x_1 or x_2
0	0	0
0	1	1
1	0	1
1	1	1

What about for an OR gate?

What weights and thresholds would serve to create an OR gate?

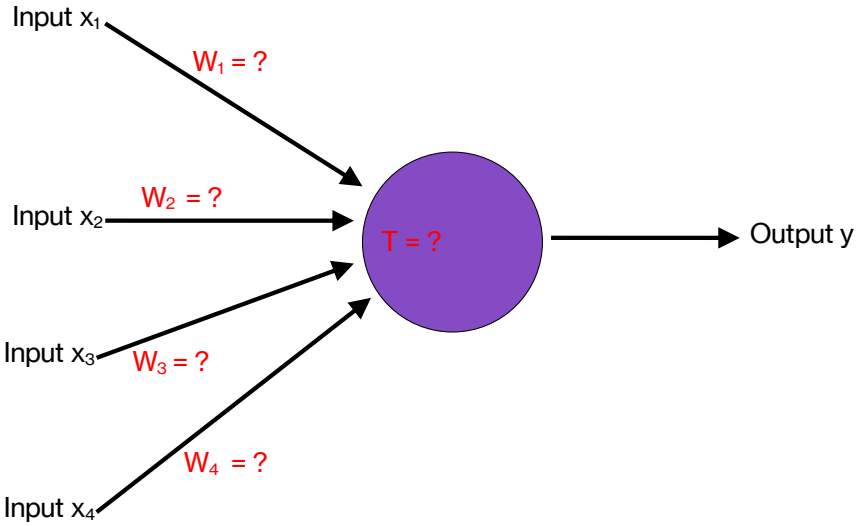
OR logical operator



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

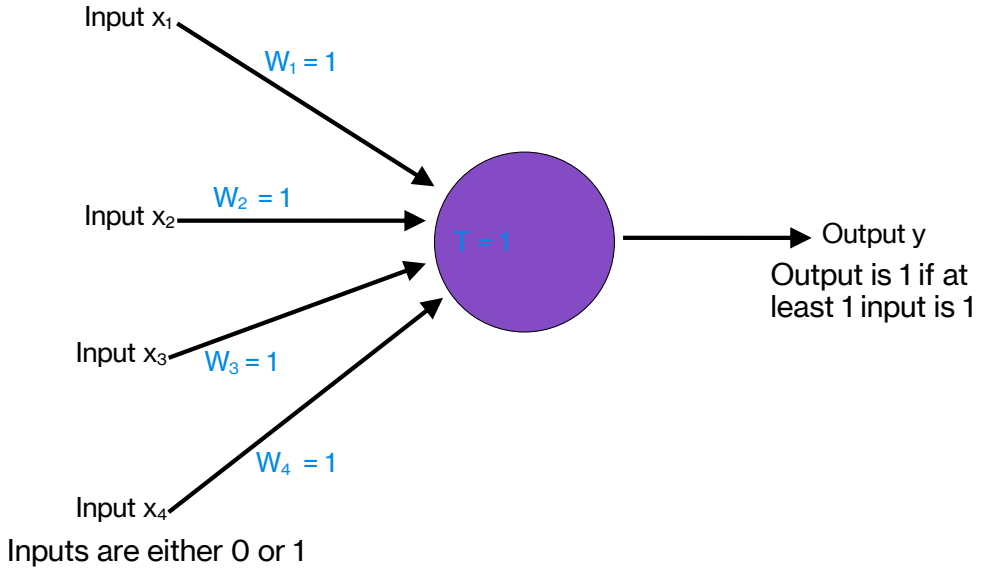
For weights of 1, the threshold would need to be 1.

OR logical operator



Similarly, we can expand it to take more inputs

OR logical operator



Note that the weights are 1 but the threshold remains 1

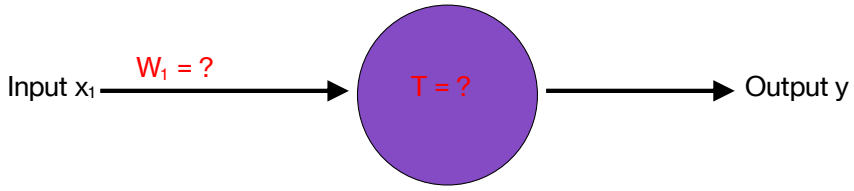
NOT logical operator

x_1	not x_1
0	1
1	0

Time to learn the NOT gate

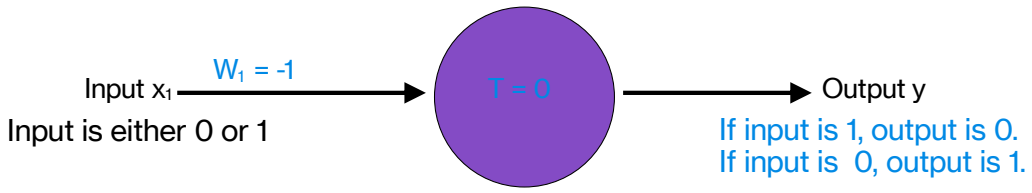
NOT logical operator

x_1	not x_1
0	1
1	0



- Now it gets a little bit trickier, what weight and threshold might work to create a NOT gate?

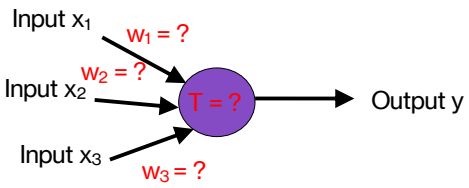
NOT logical operator



Did you get this?

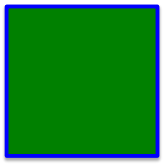
How about...

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



What if we are given a more complex truth table of an unknown Boolean function?

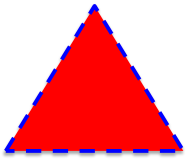
Positive or negative?



NEGATIVE

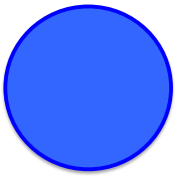
Let's put a pause here and play a game

Positive or negative?



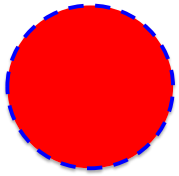
NEGATIVE

Positive or negative?



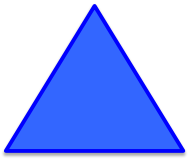
POSITIVE

Positive or negative?



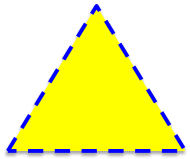
NEGATIVE

Positive or negative?



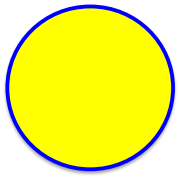
POSITIVE

Positive or negative?



POSITIVE

Positive or negative?



NEGATIVE

Positive or negative?

POSITIVE

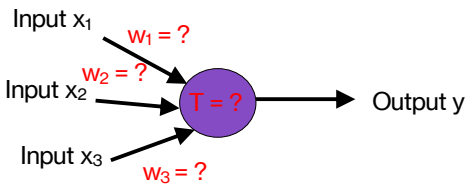
A method to the madness...

- blue = positive
- yellow triangles = positive
- all others negative

How did you figure it out?

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



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

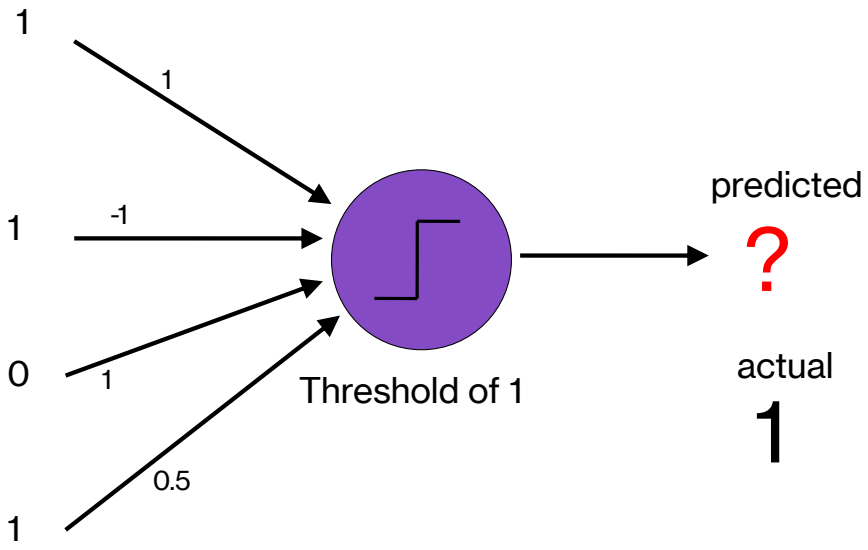
Next time we will see how to train neural networks. The basic idea is that we start with some initial weights and thresholds. We repeatedly show examples to the neural network and update the weights and threshold by comparing the neural network output to what we should have gotten it

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

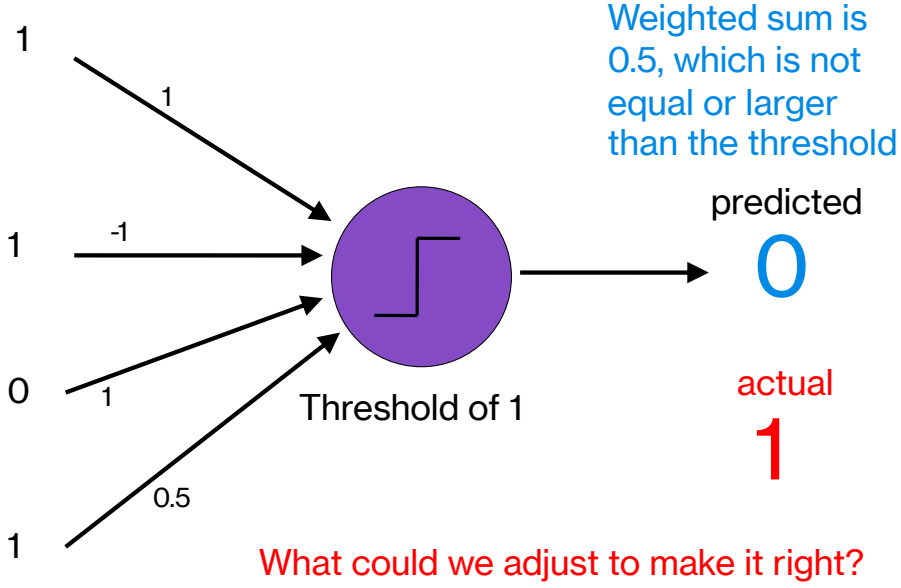
We repeat the following process until we get all examples right. For each training example, we calculated the current prediction from the perceptron. If it was off, we update the weights and threshold towards getting this example correct and repeat.

Perceptron learning algorithm



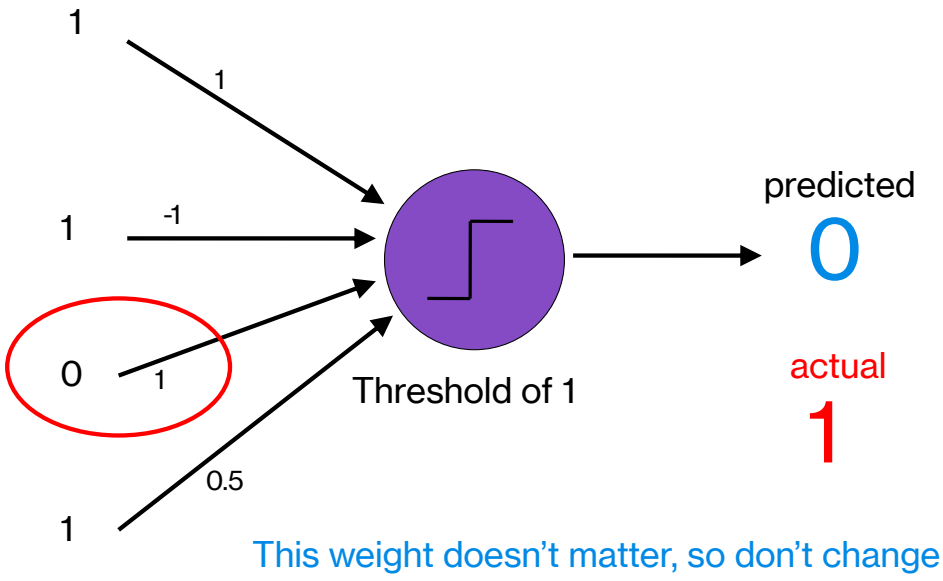
Let's say we have the following randomly selected weights (1, -1, 1, 0.5) and threshold (1). We know that for those inputs (1, 1, 0, 1), we should get 1. What do we get instead?

Perceptron learning algorithm

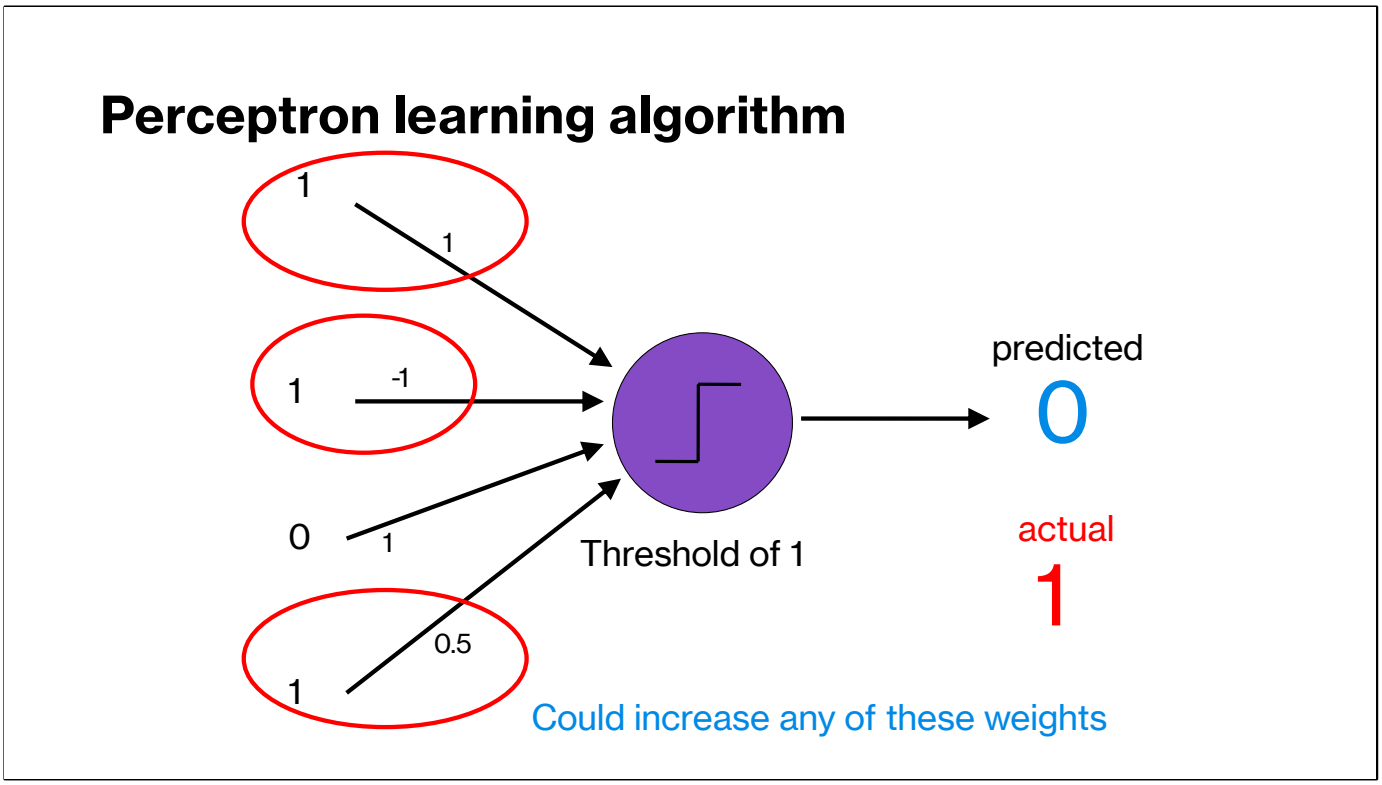


Our perceptron predicted 0 instead of 1.
What can we do to adjust it to make it right?

Perceptron learning algorithm

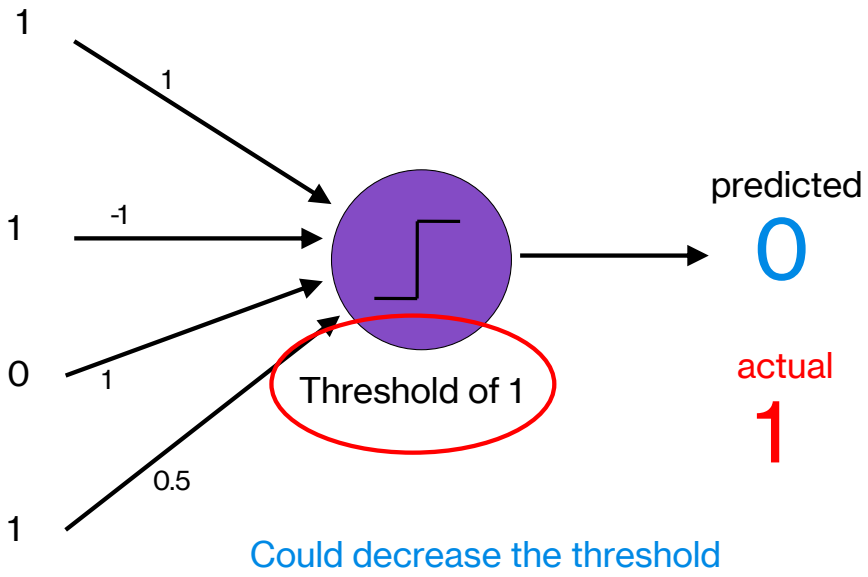


If we look at the inputs, x3=0 so that weight doesn't really matter so we won't change it



We could increase any of these weights to meet the threshold

Perceptron learning algorithm

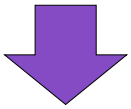


or we could decrease the threshold

Perceptron update rule

if wrong:

update weights and threshold towards getting this example correct



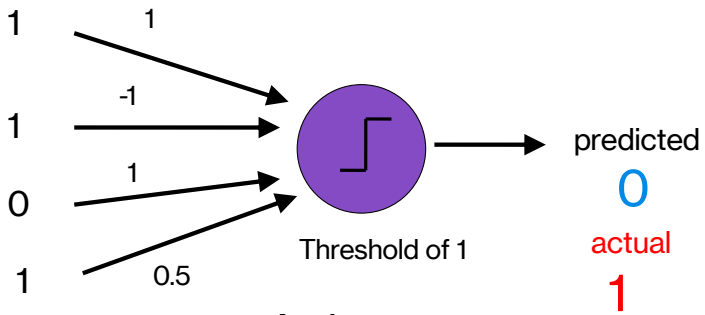
if wrong:

$$w_i = w_i + \Delta w_i$$

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

The rule for updating is that we will add a delta w to each of the weights, where that delta will be a lambda parameter times the difference of actual minus predicted value for each input.

Perceptron learning algorithm



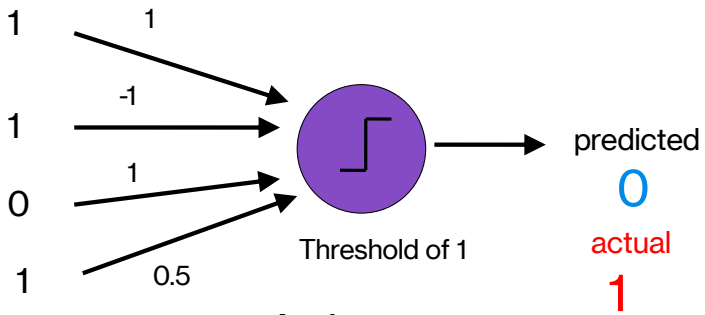
$w_i = w_i + \Delta wi$

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

What does this do in this case?

What does this look in practice?

Perceptron learning algorithm



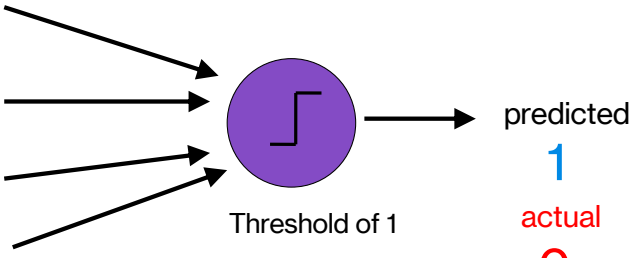
$w_i = w_i + \Delta wi$

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

causes us to increase the weights!

Since the actual value was 1 but we predicted 0, it will result to 1 and it causes us to increase the weights since delta will be positive.

Perceptron learning algorithm



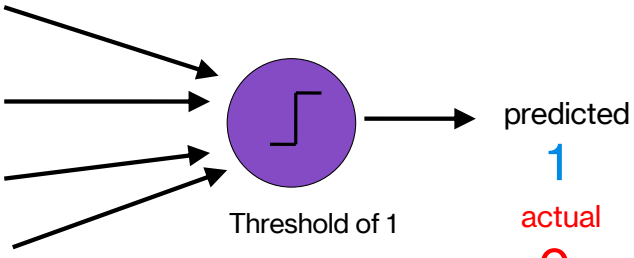
$w_i = w_i + \Delta wi$

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

What if predicted = 1 and actual = 0?

If the predicted and actual values were flipped, we would have the oppositive effect.

Perceptron learning algorithm



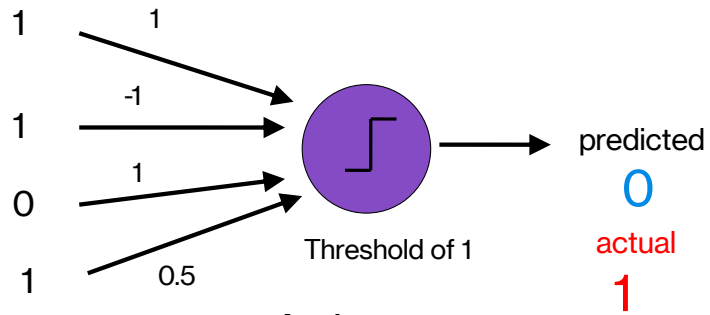
$w_i = w_i + \Delta wi$

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

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

With the difference being negative, we want to decrease the weights to not exceed the threshold

Perceptron learning algorithm



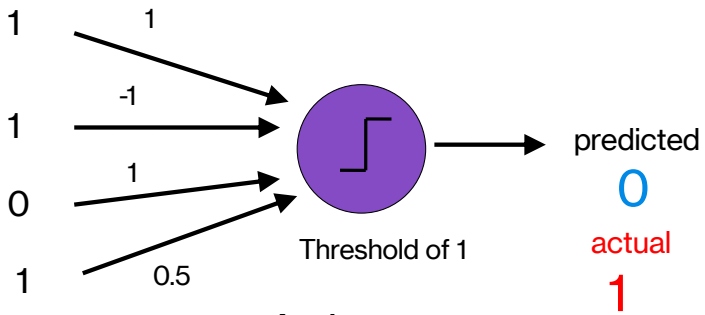
$w_i = w_i + \Delta wi$

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

What does this do?

If you look at our formula for delta, there is a multiplication with xi. what does it do?

Perceptron learning algorithm



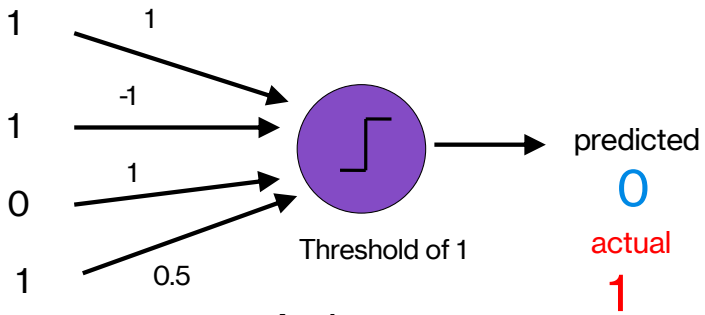
$w_i = w_i + \Delta wi$

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

Only adjust those weights that actually contributed!

If an xi==0, it will not adjust it since delta will be also 0 as a product. That means that our algorithm will only adjust those weights that actually contributed to the weighted sum.

Perceptron learning algorithm



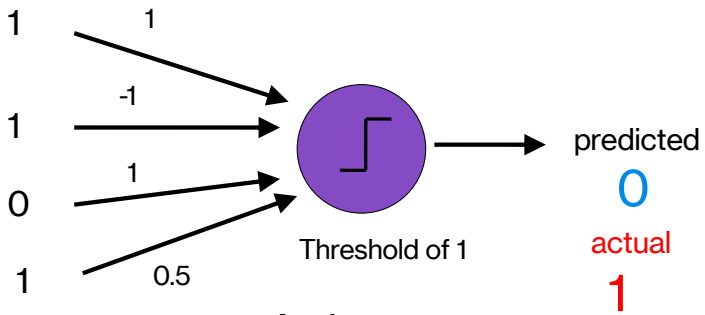
$w_i = w_i + \Delta wi$

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

What does this do?

The final unknown of calculating delta is this lambda value.

Perceptron learning algorithm



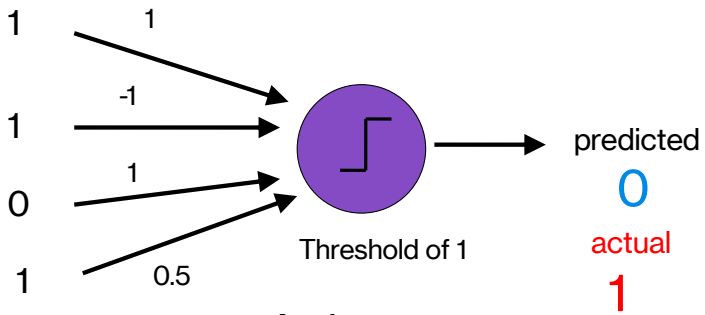
$w_i = w_i + \Delta wi$

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

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

This is known as the learning rate and it is a value between 0 and 1 which adjusts how aggressively we make changes to the model.

Perceptron learning algorithm

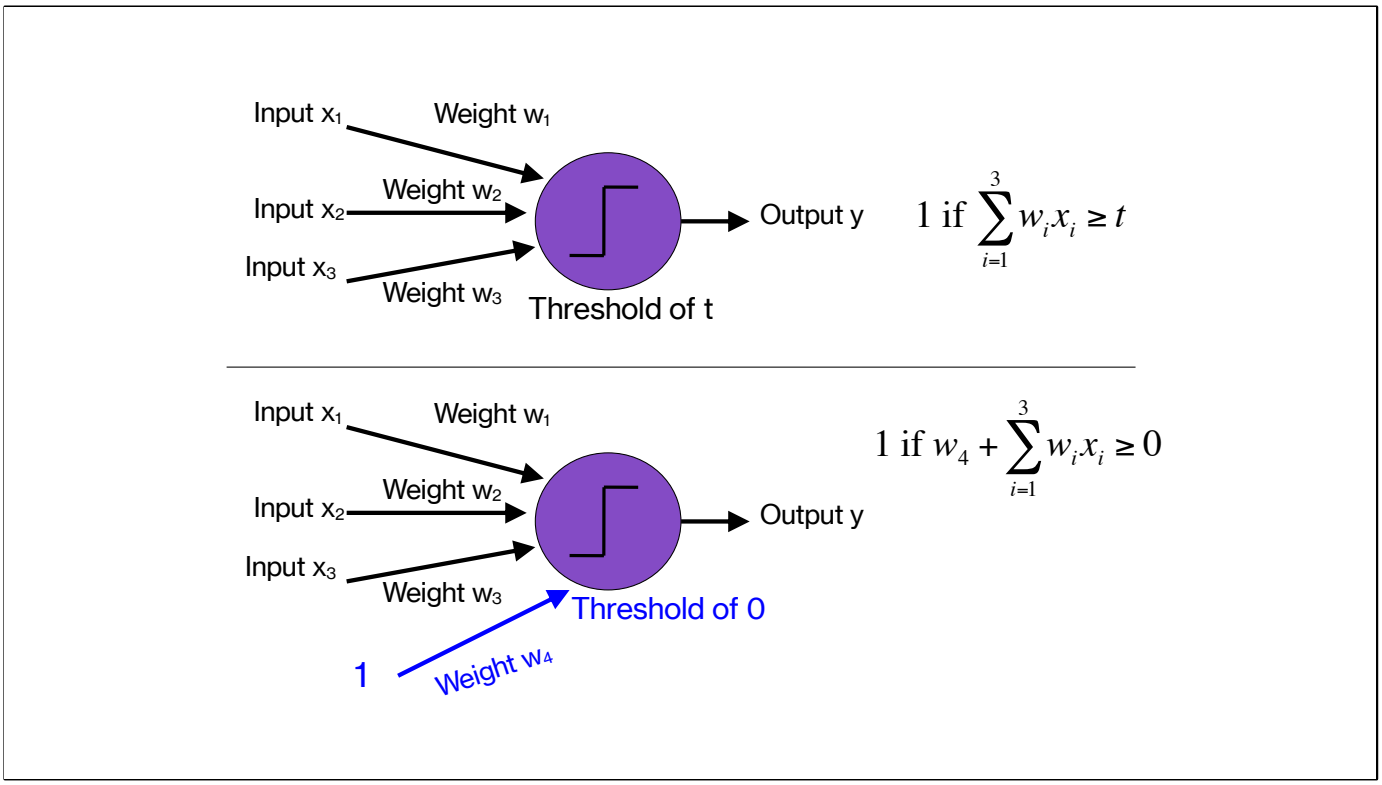


$w_i = w_i + \Delta wi$

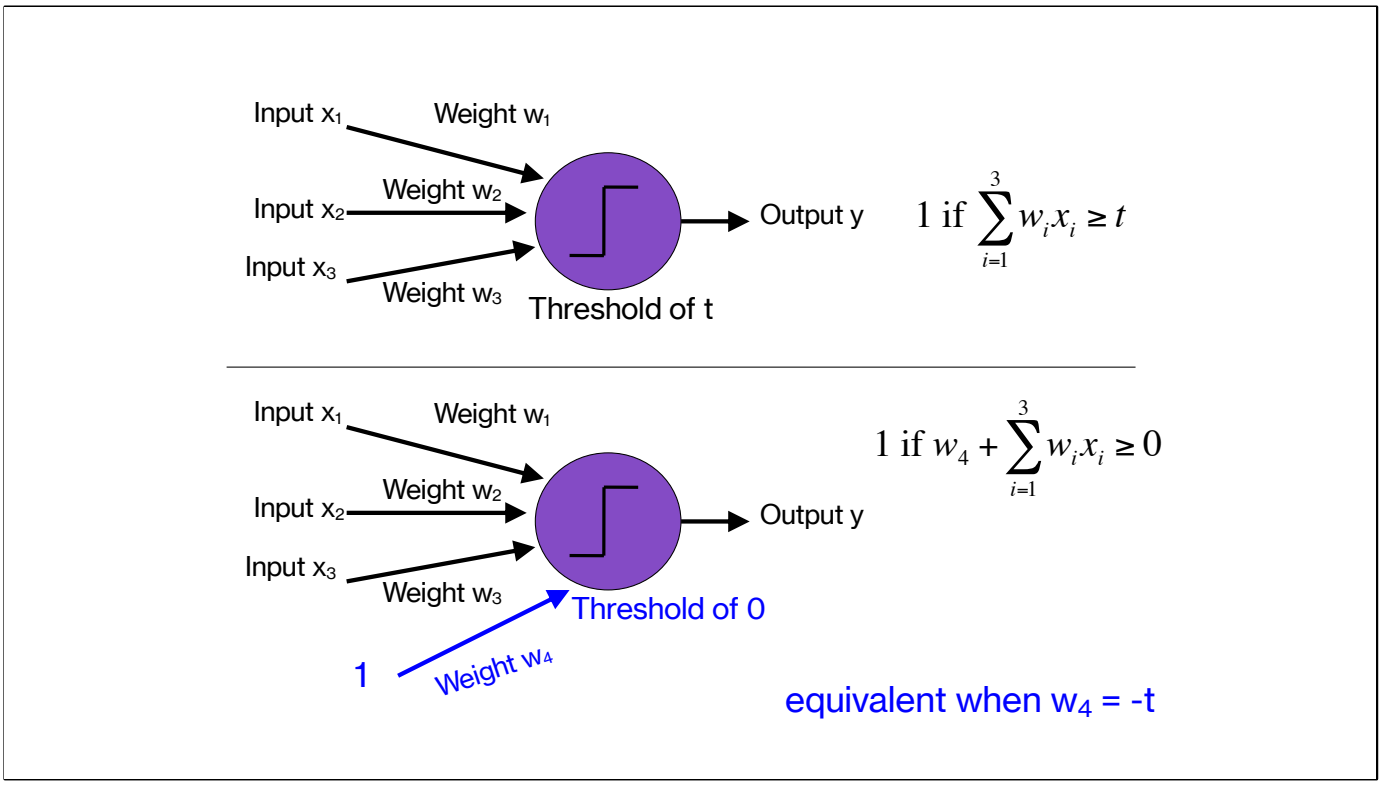
$\Delta w_i = \lambda * (\text{actual} - \text{predicted}) * x_i$

What about the threshold?

What if we want to change the threshold?



We will make two small changes to our perceptron. We will ass a new input x4=1 and we will change to threshold to 0.



This is equivalent to w4 being equal to -t, our threshold.

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:

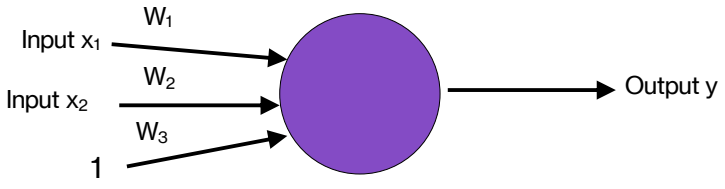
$$w_i = w_i + \lambda * (\text{actual} - \text{predicted}) * x_i$$

Putting it all together, our perceptron learning algorithm asks us to initialize weights of the model randomly and repeat until we get all the examples right by adjusting the weights.

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

$\lambda = 0.1$

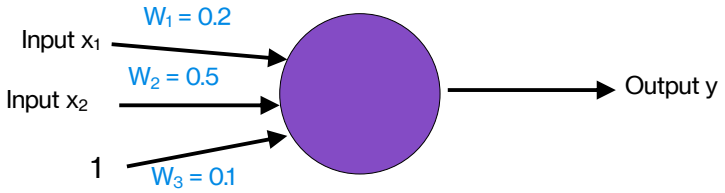
initialize with random weights



Let's try learning AND. We will assume our learning rate lambda is equal to 0.1 and we will add a third input x3=1.

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

$\lambda = 0.1$



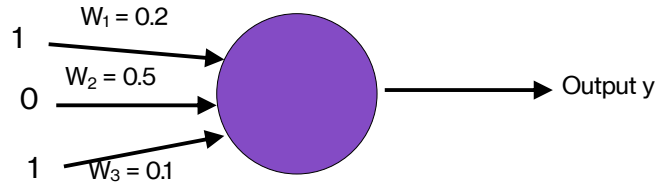
We will assign random weights to each input, say 0.2, 0.5 and 0.1

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:

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



Right or wrong?

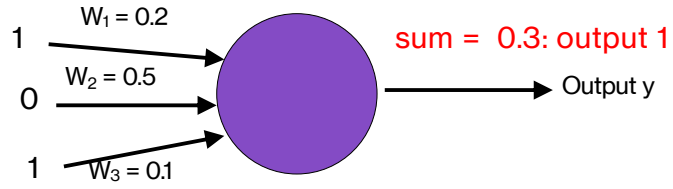
For the input 1, 0 we are expecting the output 0. Remember, our threshold is 0. Did we correctly predict 0?

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:

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



Wrong

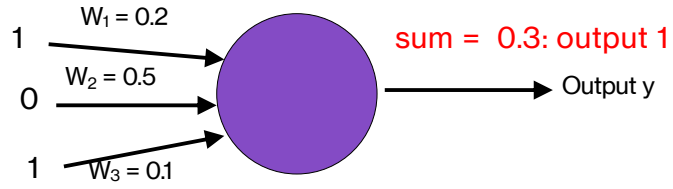
Unfortunately we are wrong. Our weighted sum was equal to 0.3 which is above the 0 threshold and we will predict 1 instead of 0.

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?

How do we adjust our weights?

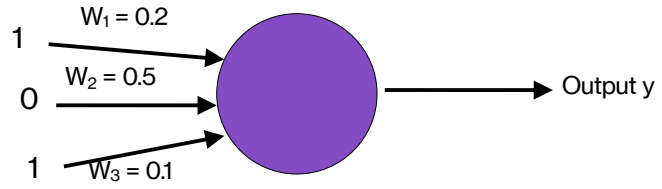
x_1	x_2	x_1 and x_2
0	0	0
0	1	0
1	0	0
1	1	1

$\lambda = 0.1$

if wrong:

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

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



Looking at our delta, we will need to decrease all non-zero xi by 0.1

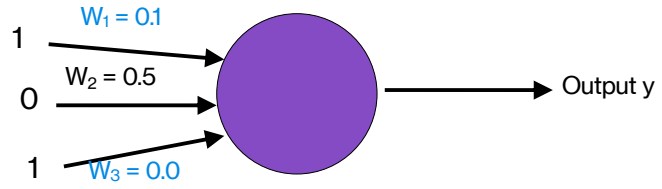
x_1	x_2	x_1 and x_2
0	0	0
0	1	0
1	0	0
1	1	1

$\lambda = 0.1$

if wrong:

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

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



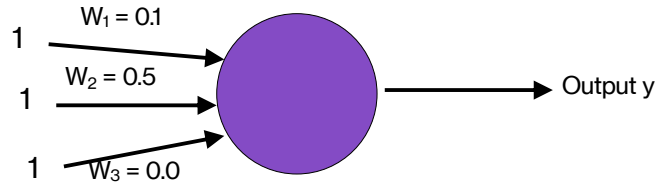
Which updates the weights for x1 and x3.

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?

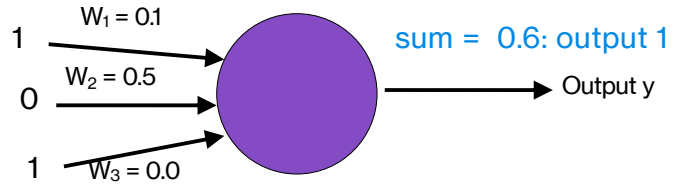
Given our new weights, let's test another row, for x1=1 and x2=1. What's our predicted output?

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!

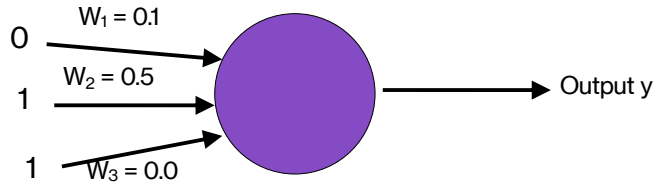
We correctly predicted 1.

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:

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



Right or wrong?

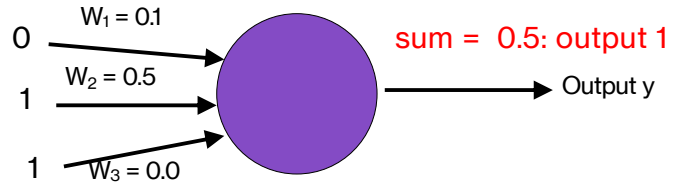
We are not done, we have to see if every single input is also leading to a correct prediction. What about 0 and 1 as inputs?

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:

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



Wrong

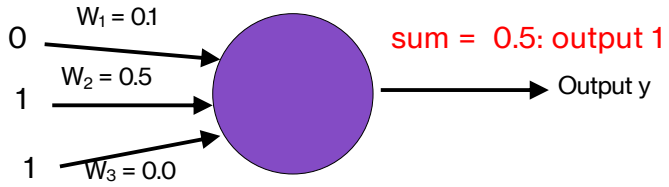
We are unfortunately wrong

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:

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



new weights?

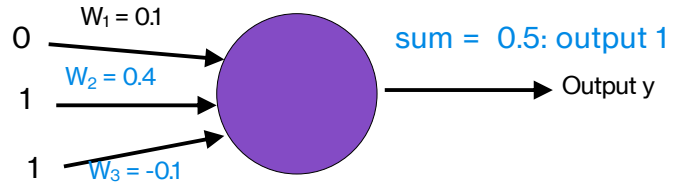
How do we adjust our weights?

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:

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



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

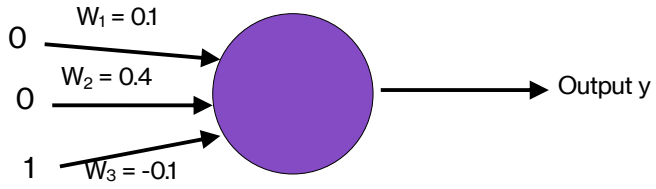
We are going again to decrease all non-zero inputs by 0.1

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:

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



Right or wrong?

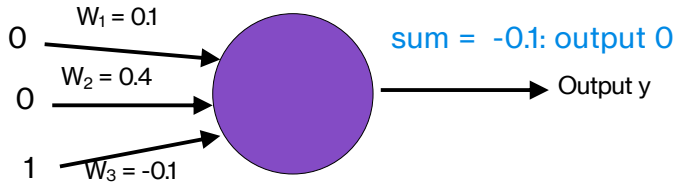
How about now with 0, 0 inputs?

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!

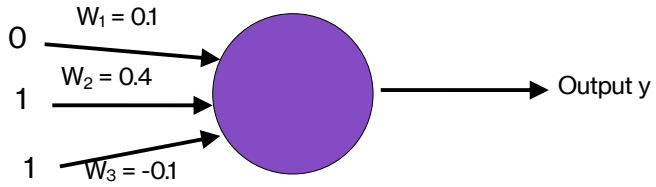
We are right!

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:

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



Right or wrong?

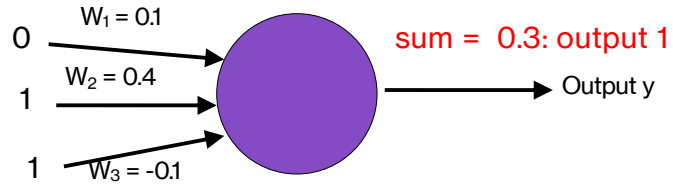
How about 0 and 1?

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:

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



Wrong

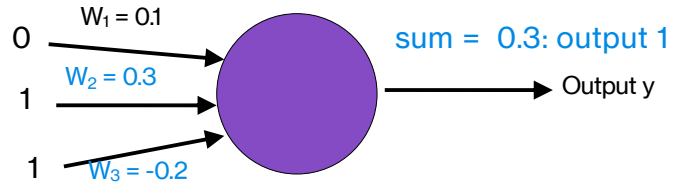
We unfortunately predicted 1 instead of 0. How do we adjust our weights?

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:

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



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

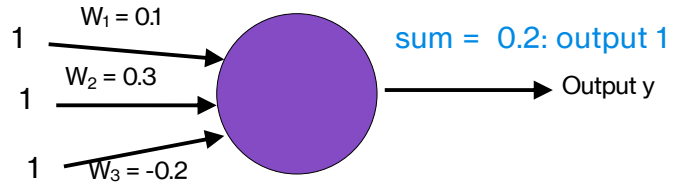
We are going to decrease all non-zero inputs by 0.1

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:

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



Right. No update!

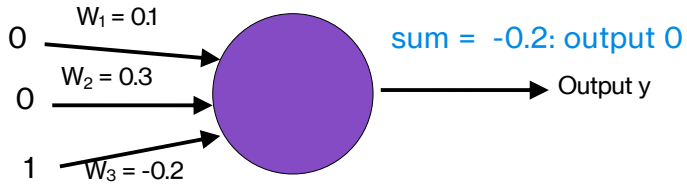
If we test for 1, 1 we see we are right.

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:

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



Right. No update!

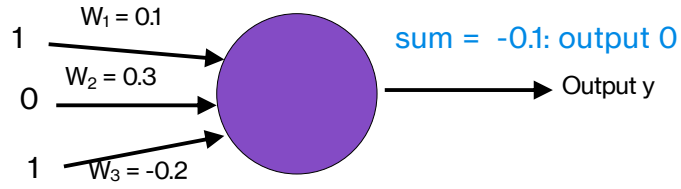
Same for 0, 0

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!

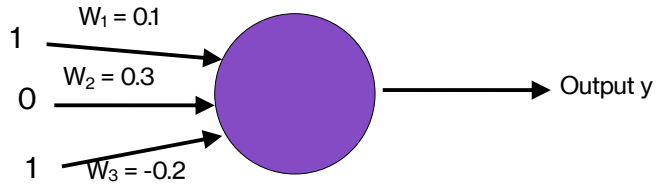
And 1, 0

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:

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



Are they all right?

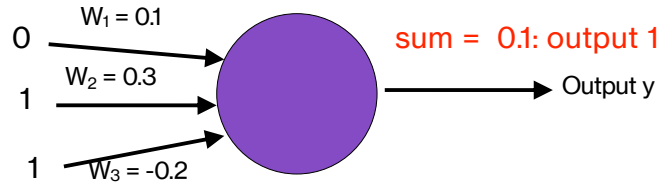
Are they all right?

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:

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



Wrong

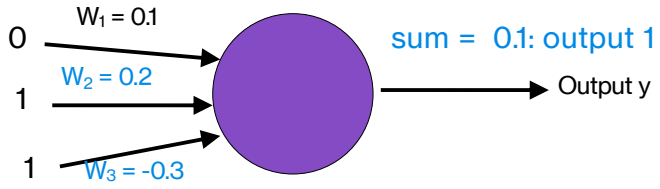
Oops, we are wrong for 0, 1.

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:

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



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

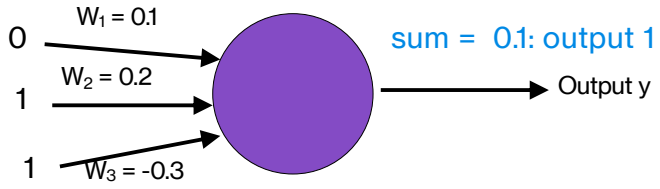
We will decrease all our non-zero inputs by 0.1

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:

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



Are they all right?

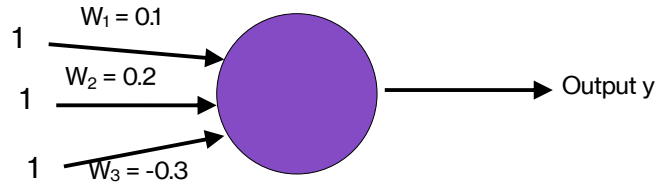
How about now, are they all right?

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:

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



We've learned AND!

Yes, we have 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)

Overall, there are a few missing details but this is the key idea behind training a simple neuron: we keep adjusting weights as long we make 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. Two sets of points in a two-dimensional space are linearly separable if they can be completely separated by a single line