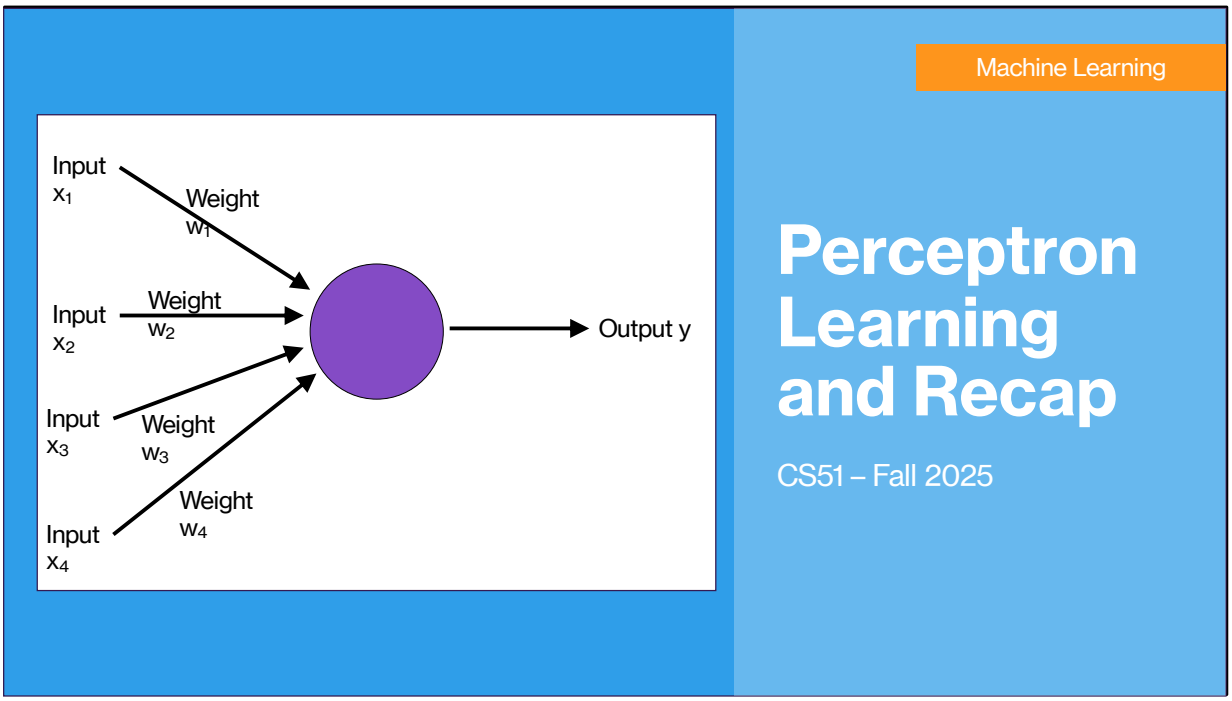
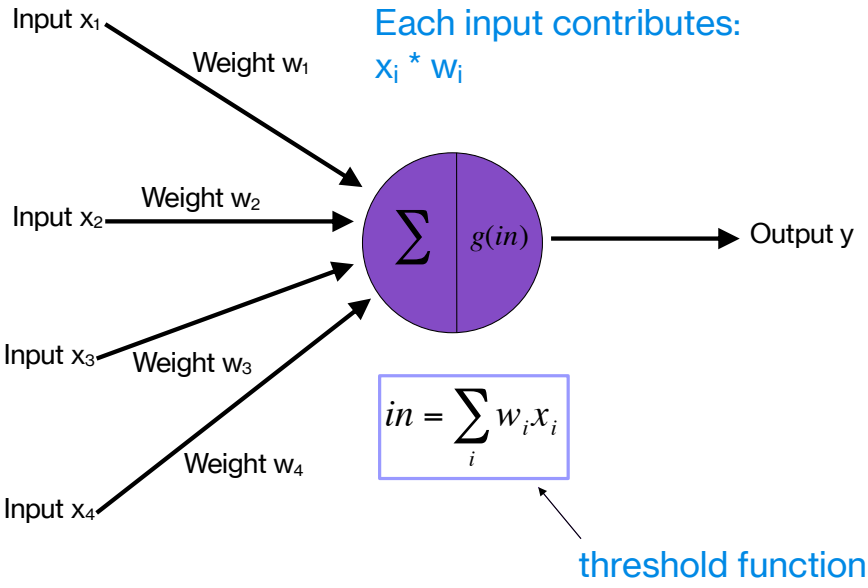


Today we will conclude our lecture on perceptron learning (and the entire course!)

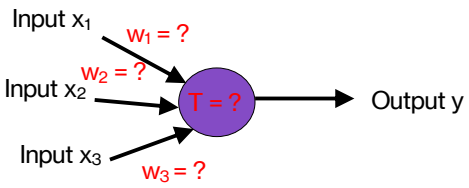
A single neuron/perceptron



As a reminder, last time, we talked about fundamental types of machine learning and discussed how artificial neural networks comprise of neurons that are connected by synapses. 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.

How about...

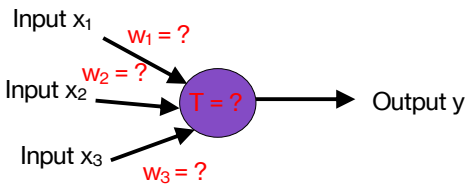
x_1	x_2	x_3	y
0	0	0	1
0	1	0	0
1	0	0	1
1	1	0	0
0	0	1	1
0	1	1	1
1	0	1	1
1	1	1	0



We saw how we can intuitively build find the right weights and threshold for different perceptrons to simulate the logic gates AND, OR, and NOT and wonder what if we are given a more complex truth table of an unknown Boolean function?

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

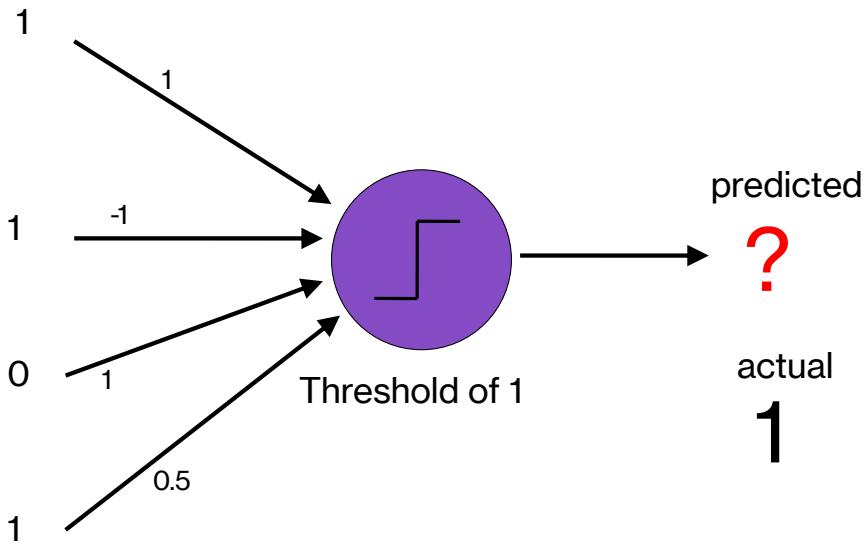
The basic idea with training neural networks is that we start with some initial weights and thresholds. We repeatedly show examples to the neural network and update the weights and threshold by comparing the neural network output to what we should have gotten it

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

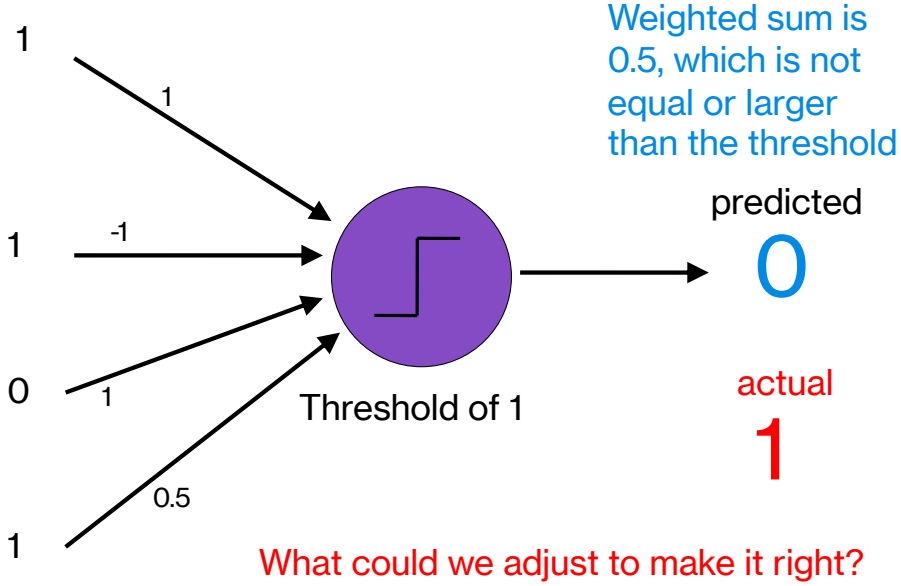
We repeat the following process until we get all examples right. For each training example, we calculated the current prediction from the perceptron. If it was off, we update the weights and threshold towards getting this example correct and repeat.

Perceptron learning algorithm



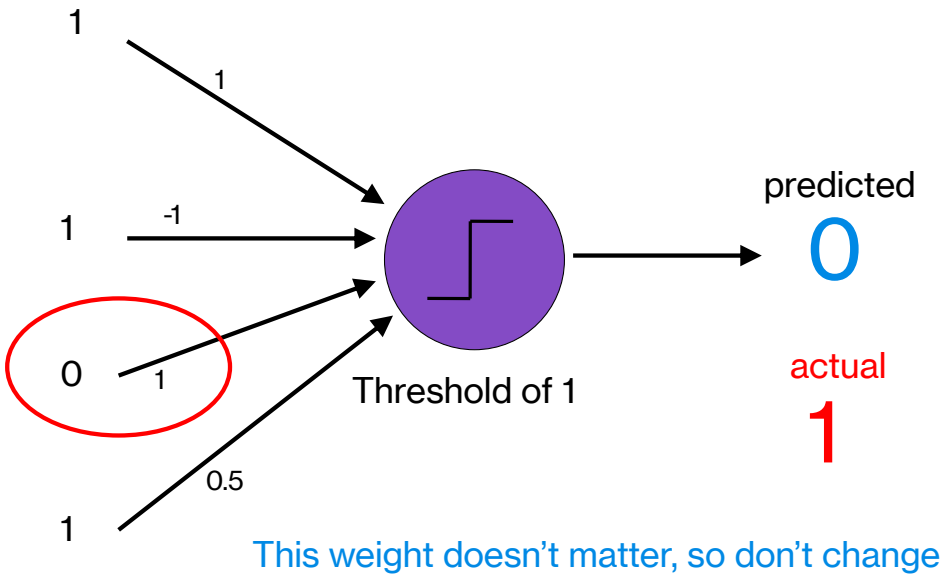
Let's say we have the following randomly selected weights (1, -1, 1, 0.5) and threshold (1). We know that for those inputs (1, 1, 0, 1), we should get 1. What do we get instead?

Perceptron learning algorithm

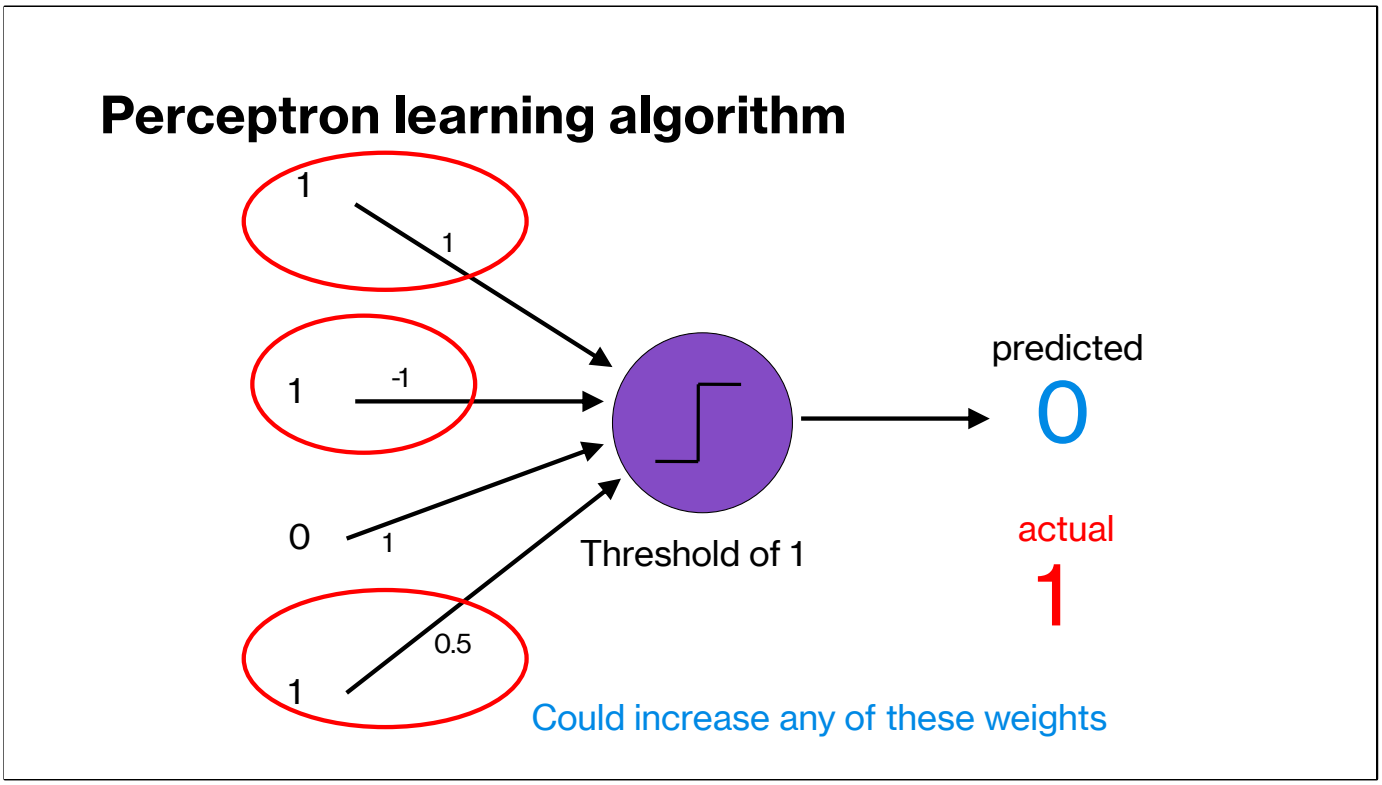


Our perceptron predicted 0 instead of 1.
What can we do to adjust it to make it right?

Perceptron learning algorithm

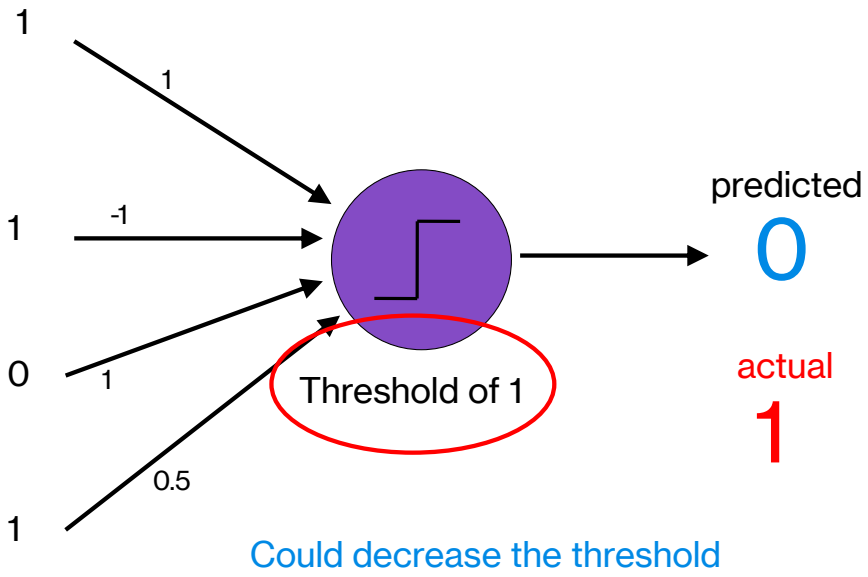


If we look at the inputs, x3=0 so that weight doesn't really matter so we won't change it



We could increase any of these weights to meet the threshold

Perceptron learning algorithm

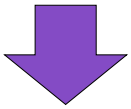


or we could decrease the threshold

Perceptron update rule

if wrong:

update weights and threshold towards getting this example correct



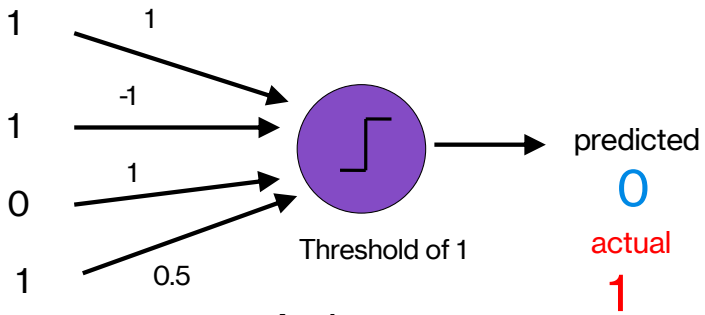
if wrong:

$$w_i = w_i + \Delta w_i$$

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

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



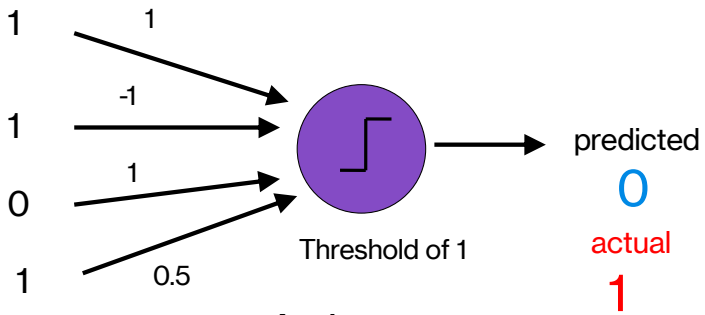
$w_i = w_i + \Delta wi$

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

What does this do in this case?

What does this look in practice?

Perceptron learning algorithm



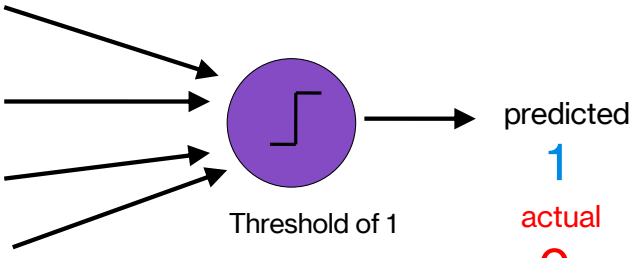
$w_i = w_i + \Delta wi$

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

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



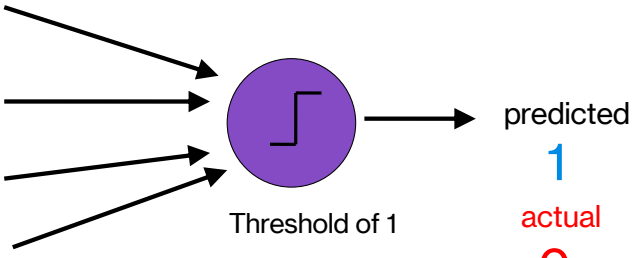
$w_i = w_i + \Delta wi$

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

What if predicted = 1 and actual = 0?

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

Perceptron learning algorithm



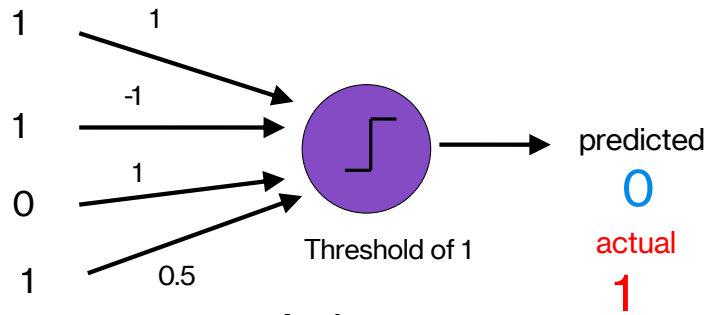
$w_i = w_i + \Delta wi$

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

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



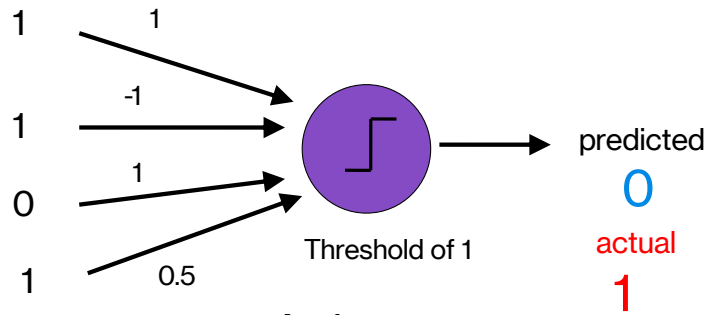
$w_i = w_i + \Delta wi$

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

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



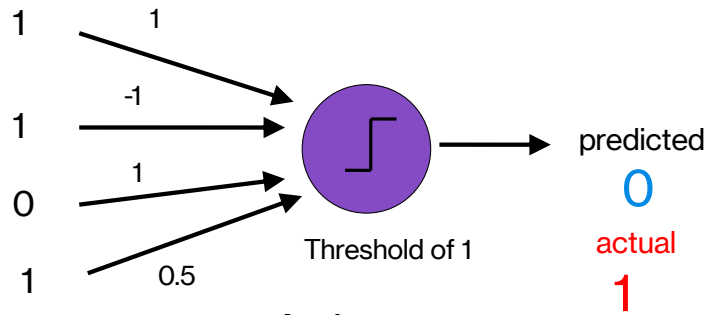
$w_i = w_i + \Delta wi$

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

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



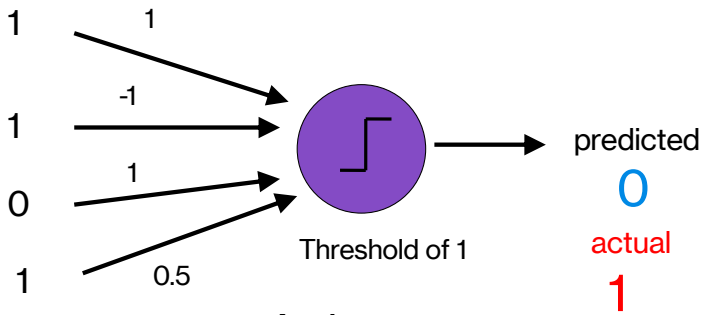
$w_i = w_i + \Delta wi$

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

What does this do?

The final unknown of calculating delta is this lambda value.

Perceptron learning algorithm



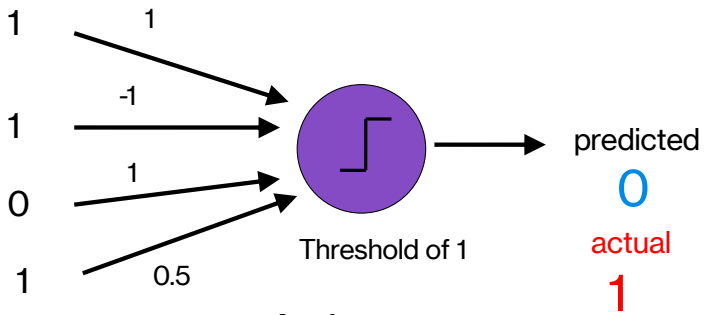
$w_i = w_i + \Delta wi$

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

"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

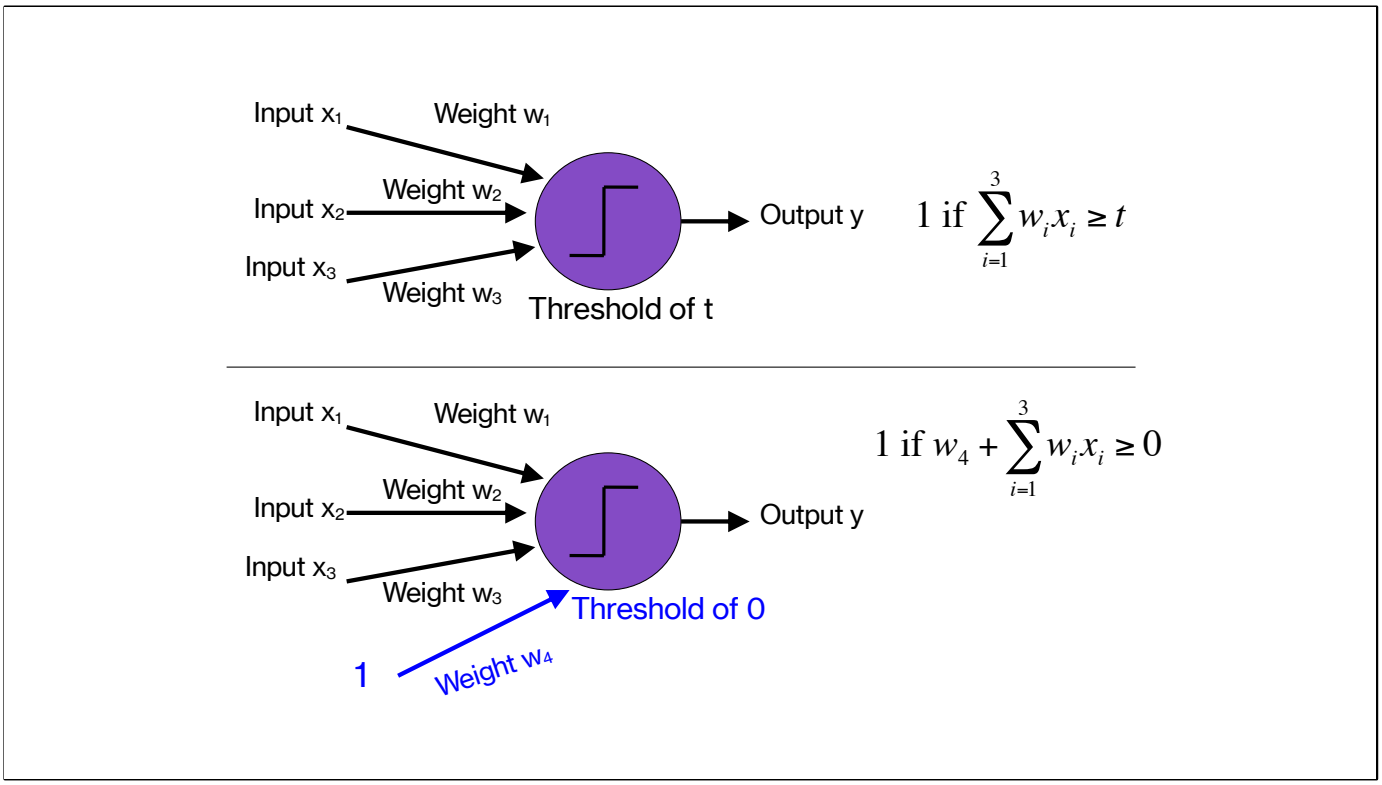


$w_i = w_i + \Delta wi$

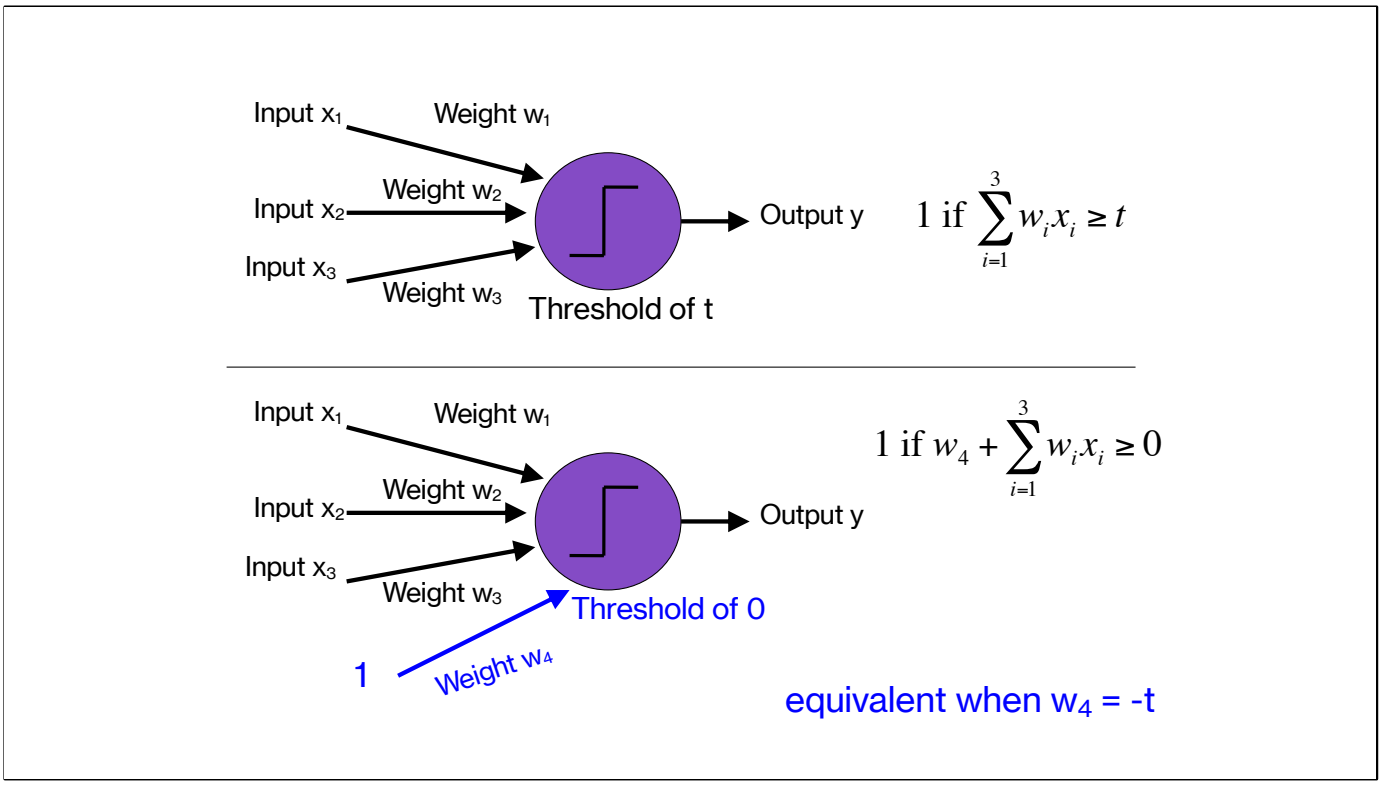
$\Delta w_i = \lambda * (\text{actual} - \text{predicted}) * x_i$

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:

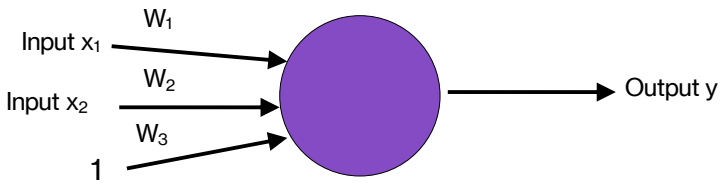
$$w_i = w_i + \lambda * (\text{actual} - \text{predicted}) * x_i$$

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

$\lambda = 0.1$

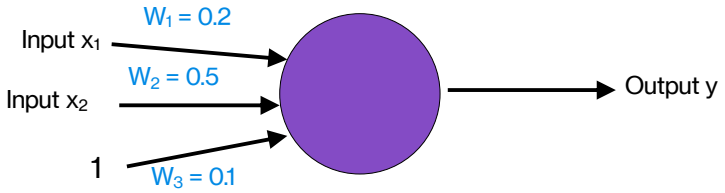
initialize with random weights



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

$\lambda = 0.1$



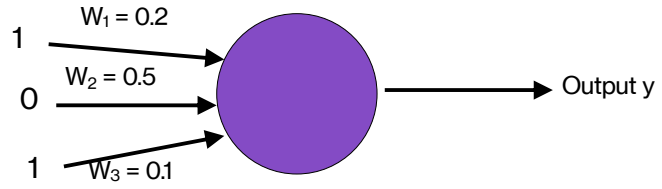
We will assign random weights to each input, say 0.2, 0.5 and 0.1

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?

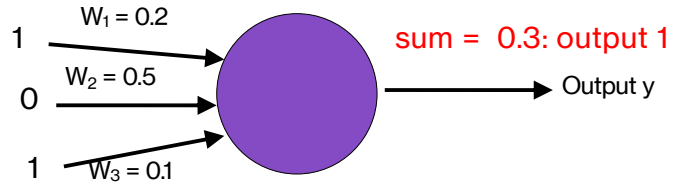
For the input 1, 0 we are expecting the output 0. Remember, our threshold is 0. Did we correctly predict 0?

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

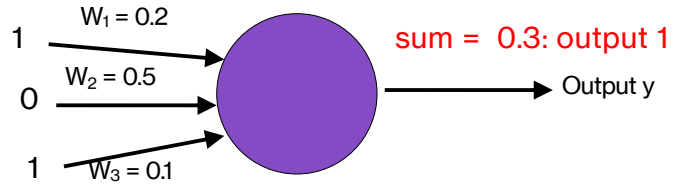
Unfortunately we are wrong. Our weighted sum was equal to 0.3 which is above the 0 threshold and we will predict 1 instead of 0.

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?

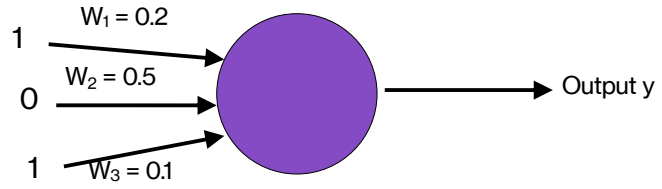
x_1	x_2	x_1 and x_2
0	0	0
0	1	0
1	0	0
1	1	1

$\lambda = 0.1$

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

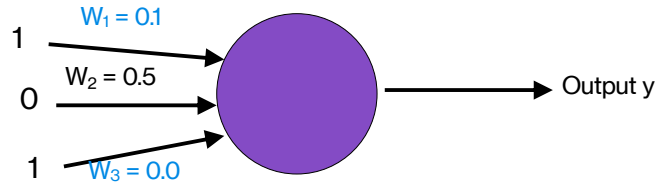
x_1	x_2	x_1 and x_2
0	0	0
0	1	0
1	0	0
1	1	1

$\lambda = 0.1$

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



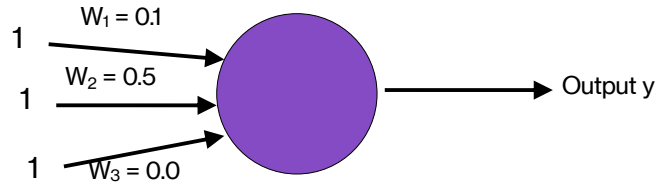
Which updates the weights for x1 and x3.

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?

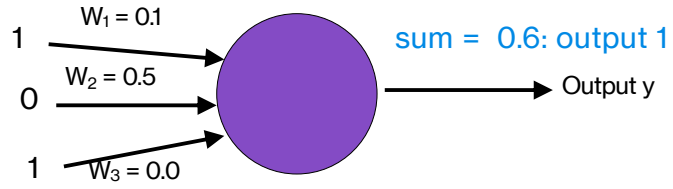
Given our new weights, let's test another row, for x1=1 and x2=1. What's our predicted output?

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!

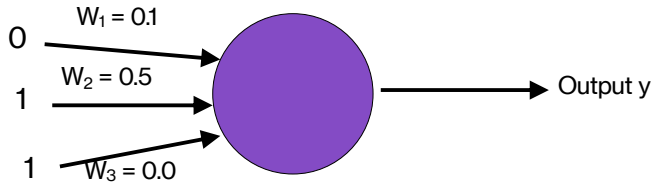
We correctly predicted 1.

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?

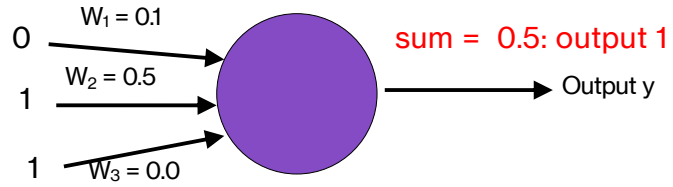
We are not done, we have to see if every single input is also leading to a correct prediction. What about 0 and 1 as inputs?

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

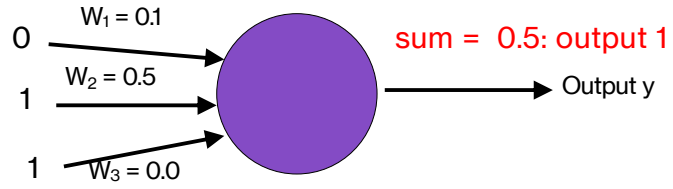
We are unfortunately wrong

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?

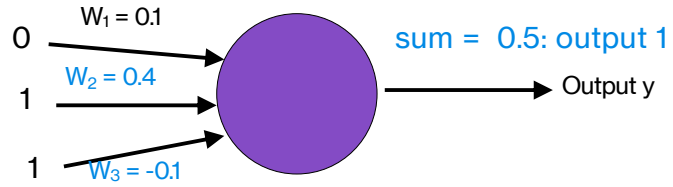
How do we adjust our weights?

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

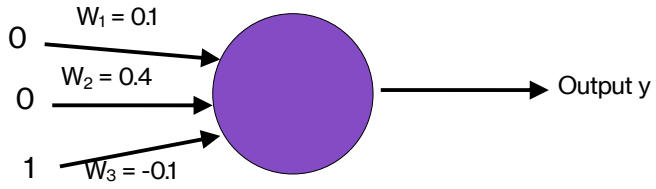
We are going again to decrease all non-zero inputs by 0.1

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?

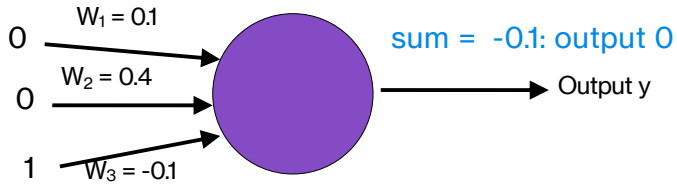
How about now with 0, 0 inputs?

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!

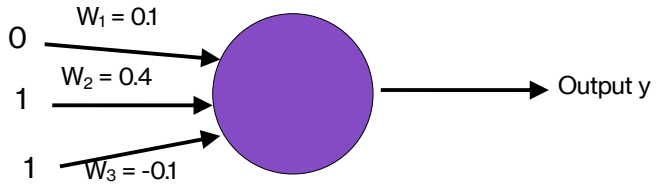
We are right!

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?

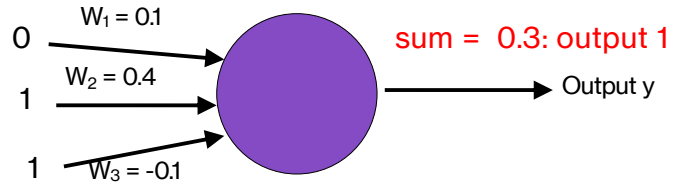
How about 0 and 1?

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

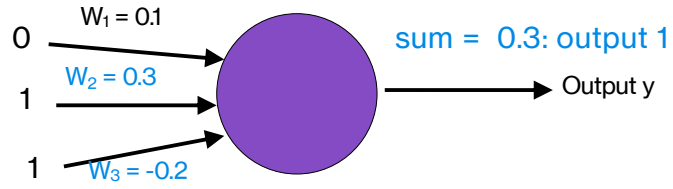
We unfortunately predicted 1 instead of 0. How do we adjust our weights?

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

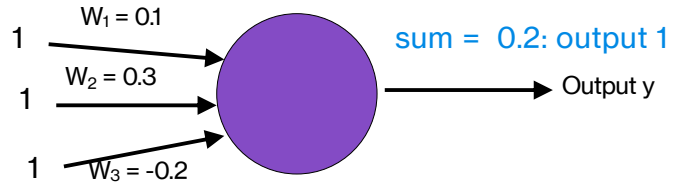
We are going to decrease all non-zero inputs by 0.1

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!

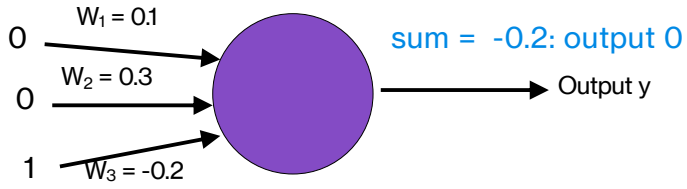
If we test for 1, 1 we see we are right.

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!

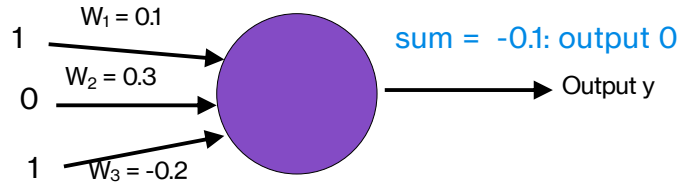
Same for 0, 0

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!

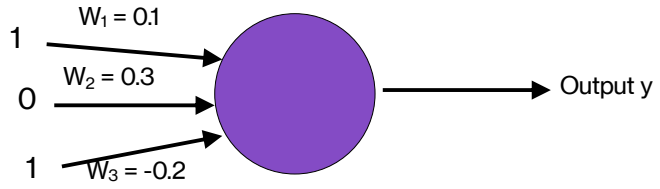
And 1, 0

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?

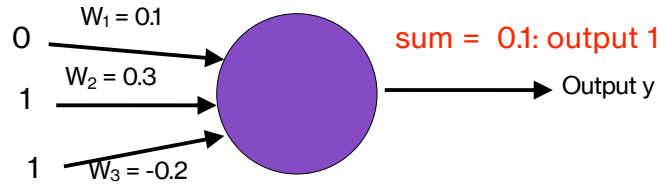
Are they all right?

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

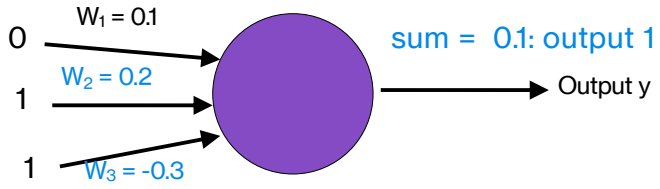
Oops, we are wrong for 0, 1.

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

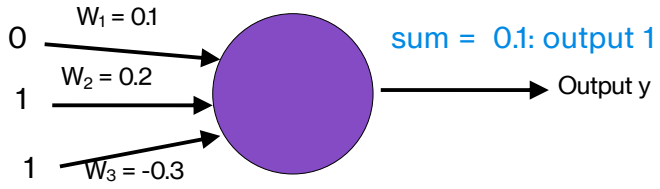
We will decrease all our non-zero inputs by 0.1

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?

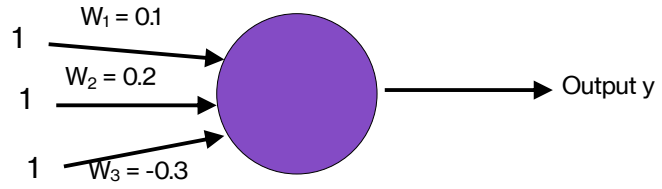
How about now, are they all right?

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

Final exam

- Friday, Dec. 12 at 2:00-5:00pm (in our classroom)
- Can bring **two** double-sided sheets of hand-written notes
- Cumulative for the entire semester
- Go over all slides and accompanying code/circuits/automata.
- Try to solve the practice time problems before you see the solution.
- Make a pass over all assignments
- If you still have questions, come to special office hours (see slack for announcements)
- Use the following guide as a (non-exhaustive) guide

History and Ethics

- Similar to checkpoint 1

Boolean Algebra

- NOT, AND, OR, XOR opetors
- De Morgan's laws
- DNFs and CNFs
- Minterms and maxterms
- Boolean Algebra in Python

Numeral Systems

- Decimal numbers
- Binary numbers (MSB and LSB)
- Hexadecimal numbers
- Equivalence across all three number systems

Binary Arithmetic and Representing Information

- Addition in decimal, binary, hexadecimal
- Overflow
- Unsigned, signed numbers(magnitude and two's complement representation)
- Equivalence across systems
- Sign extension
- Shifting: logical and arithmetic right, left shifting
- Bitwise operators

Combinational circuits

- Gates and truth tables for NOT, AND, OR, XOR, NAND, NOR
- minterm expansion to find DNF
- Karnaugh maps (see relevant assignment)
- Half- and full-adders
- Decoders
- 7-segment displays

Sequential circuits

- SR-latches
- Registers
- units of memory
- Calculating the size of memory
- Types of memory

Central Processing Unit

- How CPU coordinates with the different registers to read program instructions
- fetch-decode-execute cycle

Call stack

- Know how to visualize the call stack and how different stack frames are pushed and popped (including for recursion later on)

Introduction to Proofs

- Atomic and compound propositions
- Logical connectives including if then and if and only if
- Precedence
- Tautologies
- Satisfiability
- Logical equivalence

Loop invariants

- Know how to state pre- and post-conditions
- How to write a loop invariant
- How to prove correctness of an iterative algorithm:
  - Initialization
  - Maintenance
  - Postcondition
  - Termination

Proof by induction

- Revisit summation notation
- Know how to apply template for proof by induction

Proof by strong induction

- Understand the difference between weak and strong induction
- Floor, ceiling, modulo, prime, composite, divisible
- Apply the template for strong induction

Recursion

- Practice with writing recursive functions on integers, lists, strings

Turtle graphics

- Know basic commands and how they manipulate the turtle

Sequences and dictionaries

- Strings, lists, tuples, ranges
- Indexing
- Slicing
- Iteration
- Membership
- Immutable and mutable
- Methods vs functions
- Aliasing
- Call stack
- Dictionaries

Files and errors

- How to open a file to read it line by line or read all of its lines
- How to write in a file
- Structure of dealing with exceptions

Sorting Algorithms

- Bubble-,selection-, insertion sort
- Know how to run them and their basic properties

Asymptotic Running Analysis

- Big O notation
- Common classes of runtime complexities and what they mean in practice when we double the input size of the problem we try to solve
- Ordering classes of runtime complexities from best to worst
- Understanding how to count comparisons in different sorting algorithms to identify the big O for worst, average, and best case performance

Higher order functions

- Understand that functions are objects and can be passed as parameters, be returned from functions, and created on the fly
- Look into higher_order_functions.py and practice calling its different functions
- Understand how the built-in map and filter work
- Anonymous functions using lambda
- sorted function

DFAs

- Understand that DFAs are finite automata that accept or reject words built out of a specific alphabet and which belong to a specific language
- You need a starting state but can have more than 1 accept state
- For every state, you need to be explicit about what transitions need to happen for each character of the alphabet
- Regular expressions using concatenation, Kleene star, and union

NFAs

- Similar to DFAs but for a state we could have no transition for a character (empty set), one or even more
- There is a way to translate an NFA to DFA
- Regular languages can be described by a DFA/NFA

Turing machines

- More powerful than DFA and NFAs
- Be comfortable with transitions

Machine learning

- Supervised (classification, regression, ranking), unsupervised (clustering), reinforcement learning
- Biological idea behind artificial neural networks
- Neuron/perceptron as a unit that receives inputs with certain weights and gets activated/excited/stimulated if the weighted sum is above threshold

Perceptron training

- Add one extra 1-input, assign random weights to the n+1 inputs, threshold is set at 0.
- Fix learning rate between 0 and 1
- For each training point, if the prediction was wrong, adjust the non-zero input weights by delta.