Data Structures and Algorithms

Lesson 5. Stacks and Queues

Alexandra Ciobotaru

17 mar 2022

Syllabus

- **Lesson 1. Introduction: Algorithms, Data Structures and Cognitive Science.**
- **Lesson 2. Python Data Types and Structures.**
- **Lesson 3. Principles of Algorithm Design.**
- **Lesson 4. Lists and Pointer Structures.**
- **Lesson 5. Stacks and Queues.**
- Lesson 6. Trees.
- Lesson 7. Hashing and Symbol Tables.
- Lesson 8. Graphs and Other Algorithms.
- Lesson 9. Searching.
- Lesson 10. Sorting. Mid-term evaluation quiz (30%)
- Lesson 11. Selection Algorithms.
- Lesson 12. String Algorithms and Techniques.
- Lesson 13. Design Techniques and Strategies.
- Lesson 14. Implementations, Applications and Tools.
- Lab (20%) + Project (50%).

Intro

- In this chapter we will create special list implementations and understand the concepts of stacks and queues.
- We will cover the following:
  - Implementing stacks using various methods
  - Real-life applications of stacks
  - Implementing queues using various methods
  - Real-life applications of queues
- If we have time: the perceptron

Stacks

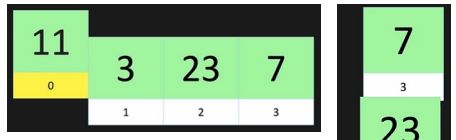
- An analogy for a stack is a stack of tennis balls
- Adding a tennis ball on top = ***push*** an item onto the stack
  - After this, we can push another item onto the stack
- But what makes a stack, a stack, is that you can make use only of the last item pushed onto the stack
- If one want to get to the second tennis ball in stack, he/she needs to ***pop*** out the last element first, in order to get to it.
- The only element we can get to in a stack is the last one – this property of stacks is called **LIFO = Last In First Out**

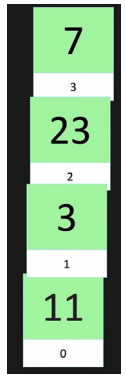
Example of using stacks

- Your browser – whenever you navigate to a new website, your browser keeps your browsing session in a stack, so when you click the back button, it takes you to the previous website (it pops out the last website in stack)
- Undo/Redo stacks in Excel or Word
- Example of stacks in real-life:
  - A stack of trays at the canteen (the waiter adds clean trays on top, and each client takes a clean tray from the top)

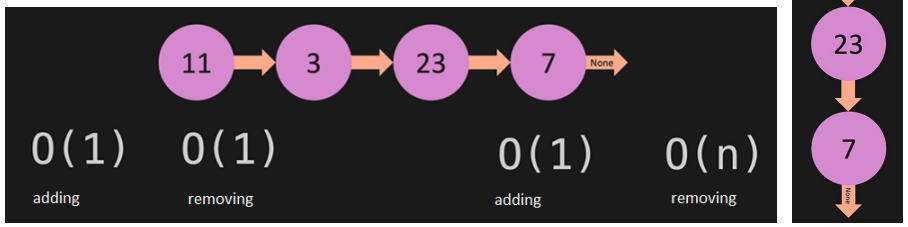
Implementing Stacks

- By using lists – we will use an end of the list for adding and removing
  - Using the left end : O(n) (we have to reindex the list if we remove the first element in list, as well if we add another element at the beginning of the list)
  - Using the right end: O(1)
    - Append instead of push



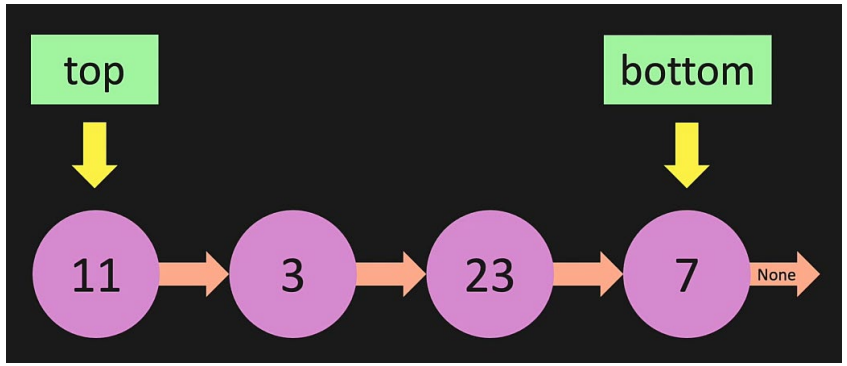


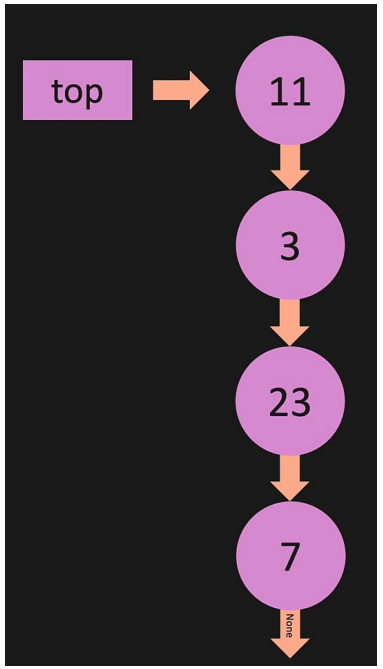
- By using Linked Lists – prepend to push



Stacks with Linked Lists

- Head and tail become top and bottom,





- but we don’t need the bottom

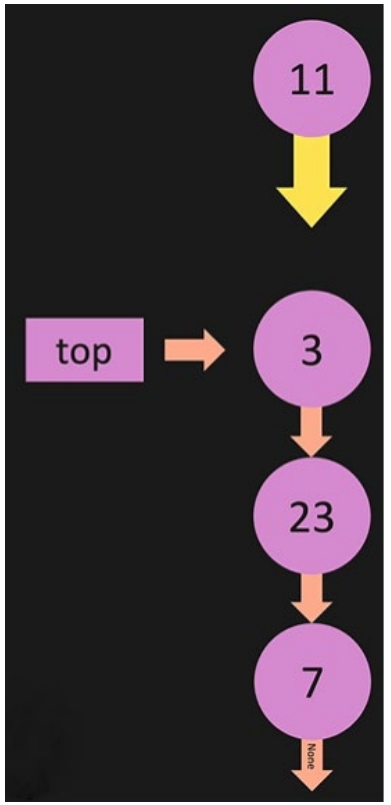
Creating the Constructor

```
class Node:
    # to create nodes (the same as in LL)
    def __init__(self, value):
        self.value = value
        self.next = None
class Stack:
    def __init__(self, value):
        #create new Node
        new_node = Node(value)
        self.top = new_node # we don't need to keep track of the bottom
        self.height = 1 # length becomes height
    def print_stack(self): #same as in LL
        temp = self.top
        # we stop running the loop when temp is None
        while temp is not None:
            print(temp.value)
            temp = temp.next
my_stack = Stack(4)
my_stack.print_stack()
```

Push

- Pushing a new node onto the stack
  - We are going to put a new node on to the stack and top point to that new node
  - We will be borrowing from the prepend method from LL
  - We need to code the situation when we don't have anything in the stack
  - In the case when we have something in the stack, the new appended node first is going to point to what the top is pointing, then top will be pointing to the added element

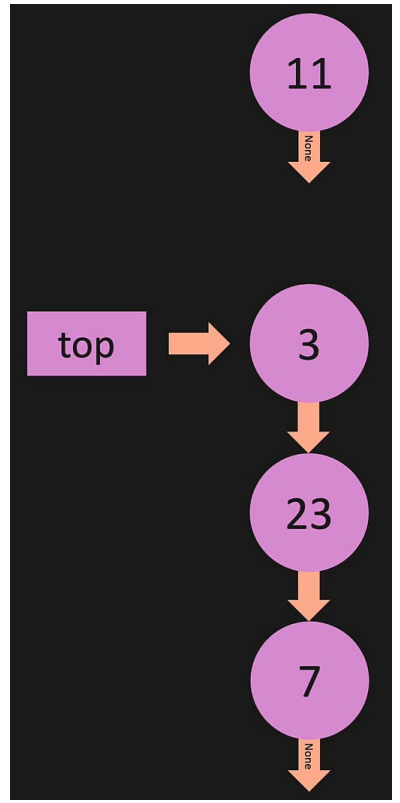
```
def push(self, value):
        new_node = Node(value)
        if self.height == 0:
            self.top = new_node
        else:
            new_node.next = self.top # new node is going
to point to the same node that top is pointing to
            self.top = new_node # now top is pointing to
the added element
        self.height += 1
My_stack.push(1)
My_stack.print_stack()
```



Pop

- We remove the top node, and then the top pointer will point to the next node down
- Next we return the node popped out from the stack
- We also code for the situation where we don’t have anything in stack

```
def pop(self):
        if self.height == 0:
            return None
        else:
            # a temporary variable will point to the top node
            temp = self.top
            # we move top down to the next node (top points to the next
node)
            self.top = self.top.next
            # we remove the node from the stack (the next node after
temp is None)
            temp.next = None
        self.height -= 1
        return temp
my_stack.pop()
my_stack.print_stack()
```



Differences between stacks and LL

STACK	LINKED LIST
An abstract data type that serves as a collection of elements with two principal operations which are push and pop	A linear collection of data elements whose order is not given by their location in memory
Push, pop and peek are the main operations performed on a stack	Insert, delete and traversing are the main operations performed on a linked list
Can read the topmost element	It is required to traverse through each element to access a specific element
Works according to the FIFO mechanism	Elements connect to each other by references
Simpler than linked list	More complex than stack

Visit www.PEDIAA.com

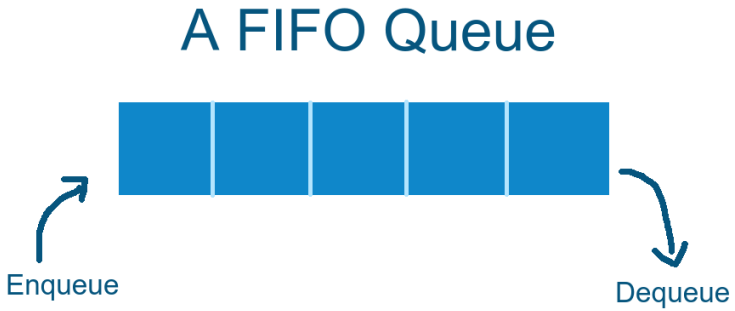
LL applications:

- Image viewer with previous and next buttons
- Songs in a music player (previous and next buttons)

https://pediaa.com/what-is-the-difference-between-stack-and-linked-list/

Queues

- We are all familiar with queues – when we get in line, there are other people that can get in line behind us, but if you were there first, you will be the first person out of the line
- This property of queues is called **FIFO – First In First Out**

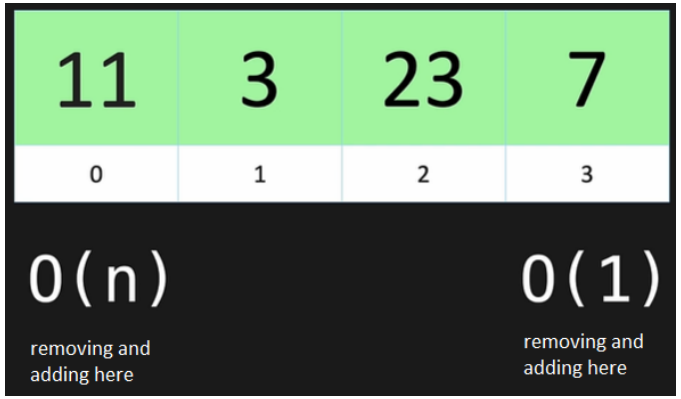


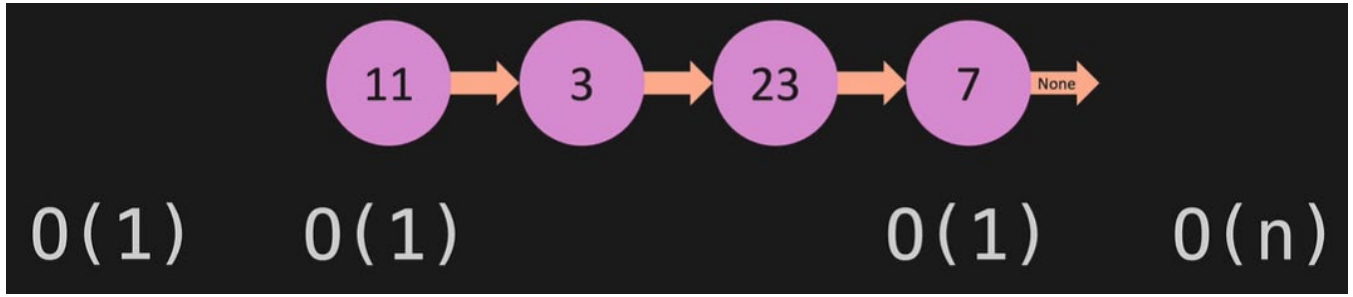
Source: https://www.codingem.com/what-is-a-fifo-queue/

- When we add elements to a queue we use the term ***enqueue***, and when we remove elements from the queue we use the term ***dequeue***.

Implementing Queues

- By using lists – adding elements at one end and removing at the other end (either O(1) or O(n))
  - (with **append**()), and dequeuing at the tail (with **pop**()).
- By using linked lists
  - O(1) at the head (first) for either removing or adding
  - O(1) at the tail (last) for adding, but O(n) for removing
  - Thus, enqueuing will be done at the head





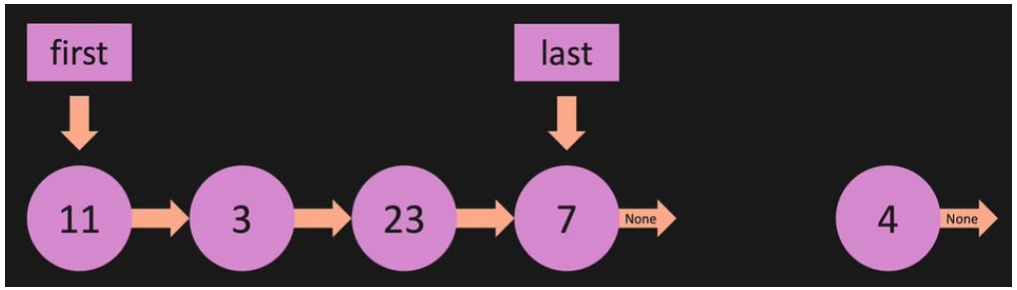
Creating the Constructor

```
class Node:
    # to create nodes (the same as in stacks)
    def __init__(self, value):
        self.value = value
        self.next = None

class Queue:
    def __init__(self, value):
        #create new Node
        new_node = Node(value)
        self.first = new_node
        self.last = new_node
        self.length = 1 # length again, not height

    def print_queue(self):
        temp = self.first
        # we stop running the loop when temp is None
        while temp is not None:
            print(temp.value)
            temp = temp.next
```

Enqueue



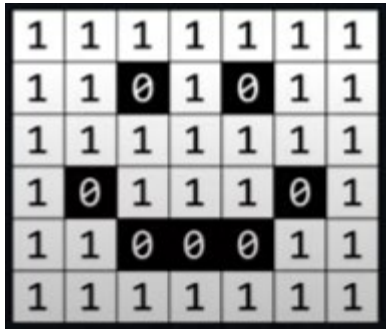

```
class Queue:
….
     def enqueue(self, value):
                # we use the value to create a node
                # if there are no elements in queue, first and last point to the same new
    node
            new_node = Node(value)
            if self.first is None:
                self.first = new_node
                self.last = new_node
            else:
    # the next value in last pointer is going to be the new_node (new node is going to point
    to the same node that first is pointing to)
                self.last.next = new_node
    # we move the last pointer to point to the new node
                   self.last = new_node
             self.length += 1
my_queue = Queue(1)
my_queue.enqueue(2)
my_queue.print_queue()
```

Dequeue

```
 def dequeue(self):
        if self.length == 0:
            return None
            # a temporary variable will point to the first node
        temp = self.first
        if self.length == 1:
            self.first = None
            self.last = None
        else:
            # first will point to the next node in queue
            self.first = self.first.next
            # remove the node
            temp.next = None
        self.length -= 1
        return temp  # change to temp.value to get the dequeued value
my_queue = Queue(1)
my_queue.enqueue(2)
my_queue.dequeue()
my_queue.print_queue()
```

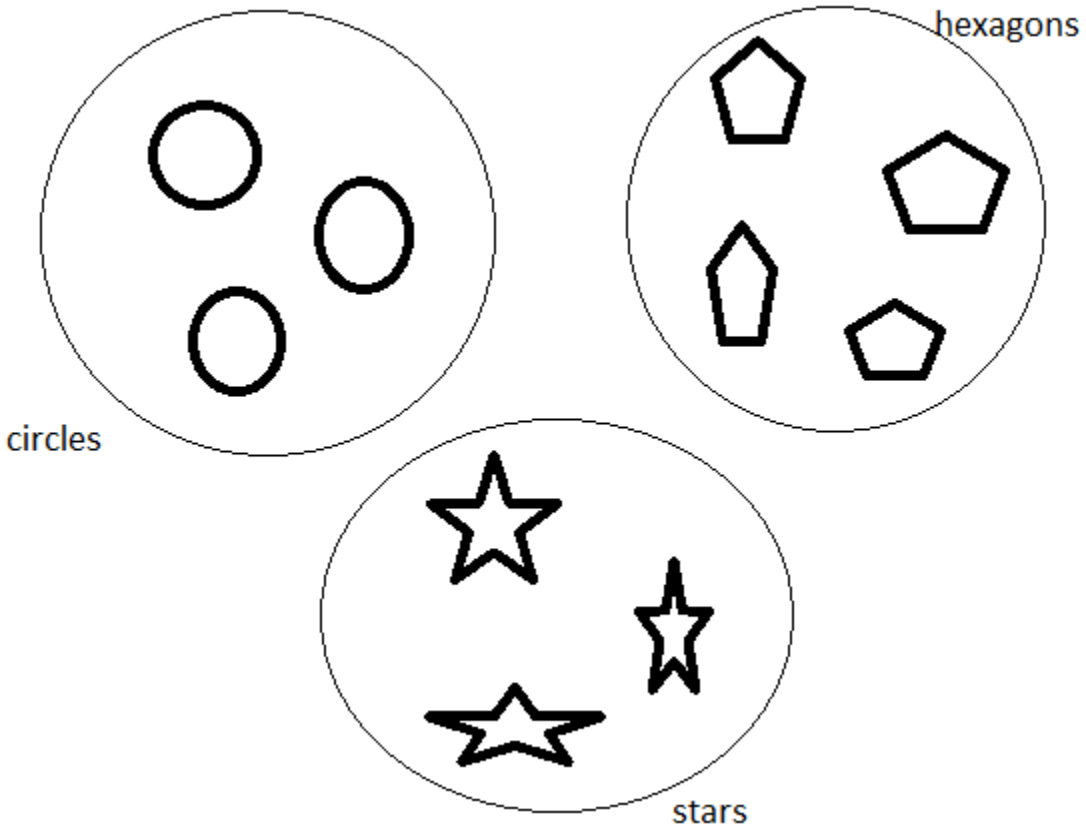
A Machine Learning Algorithm (extra)

- The Neural Network (https://www.youtube.com/watch?v=i1AqHG4k8mE)
  - it was designed to mimic the process of human thought
  - We use it to solve problems that traditional computer programs find very hard to solve, but humans are very good at it:
    - Face recognition
    - Object detection
    - Image classification (Captcha)
- Remember that a computer sees an image as a collection of pixels, where each pixel has a numeric value representing it's colour intensity – thus a picture is seen by the computer as a very long list of numeric values



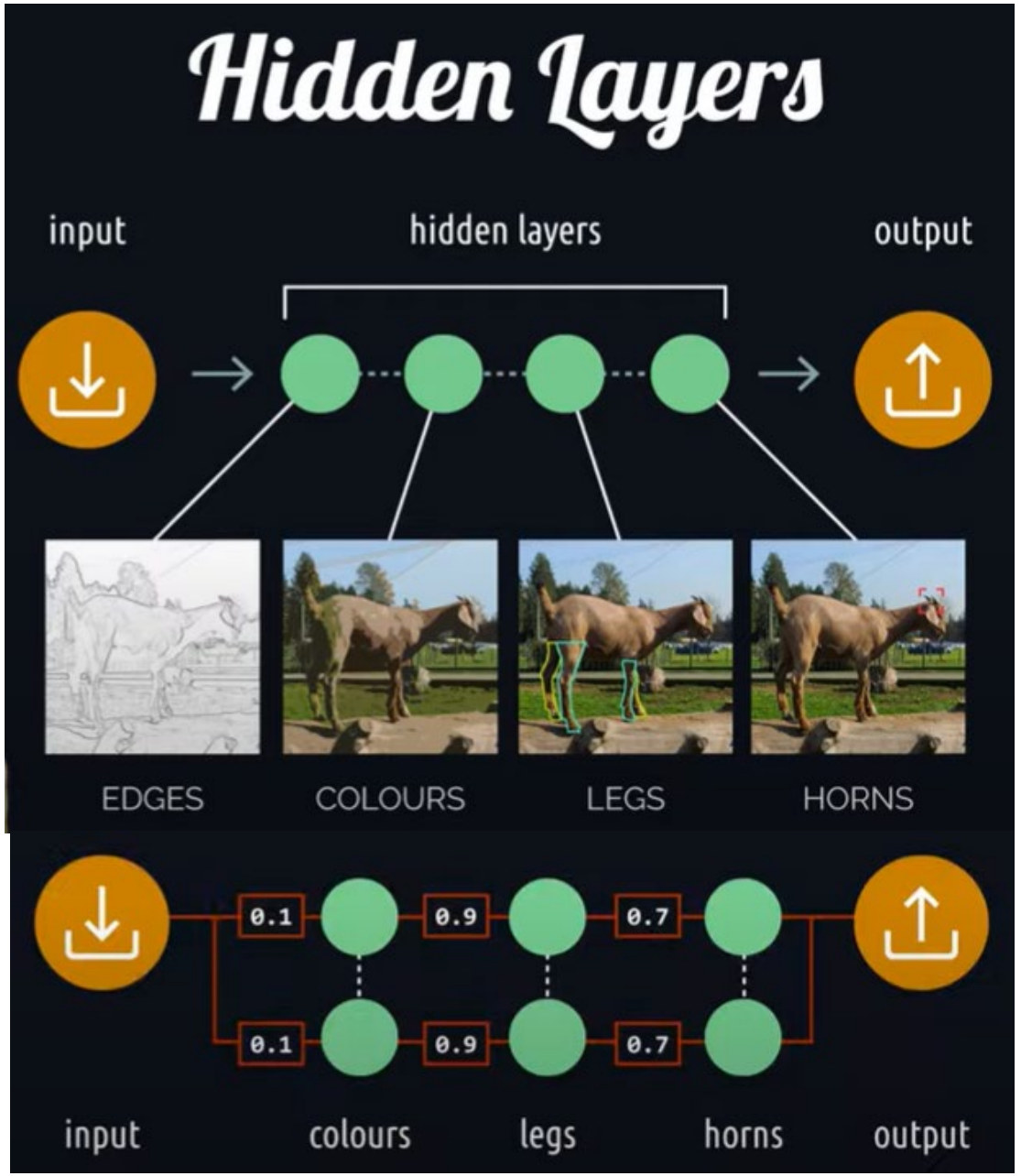
Classes and labels

- Classes: circles, stars, hexagons
- In each class we have probes
- Each probe is labelled with the class name
- We feed the model the labelled data, and the model learns the “features” of the data, in order to recognize in the future, based on these features, to what class a new unseen probe, belongs (this is called prediction)
- In order to classify a new probe, the model does statistical operations, that happen in the hidden layers



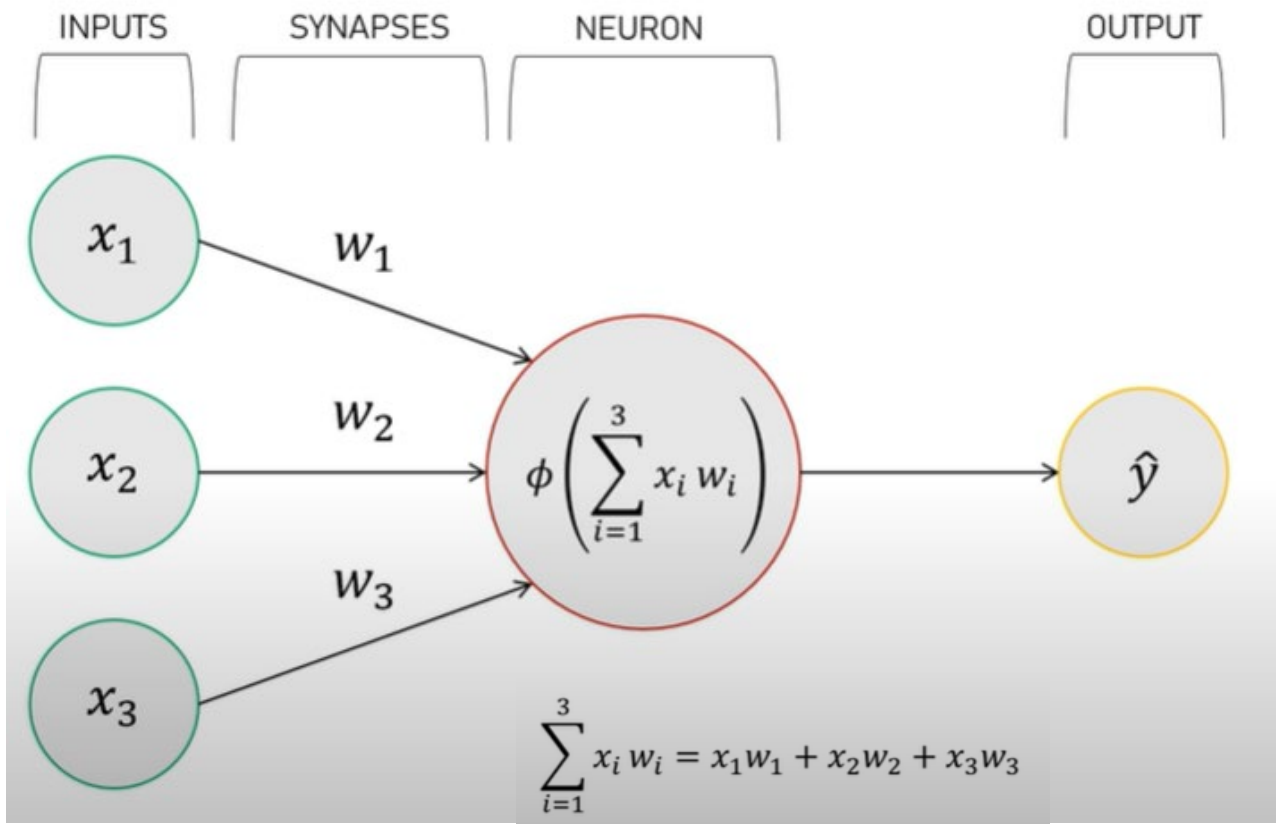
Hidden layers

- Each hidden layers is responsible with detecting a feature
- All hidden layers combined are powerful
- The links between the nodes in the hidden layers are called weights, and these weights influence how much impact each node has on the input
- The idea is to adjust weights and other parameters until the neural network is optimal



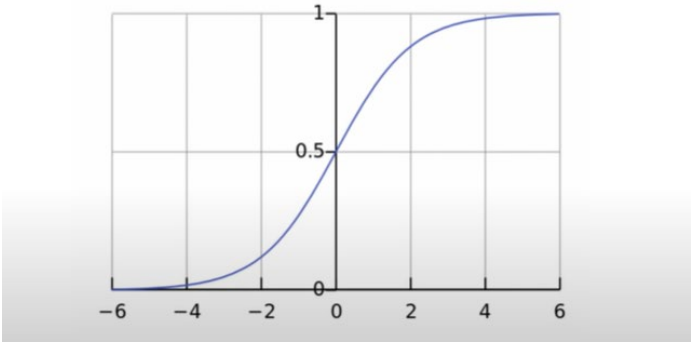
The perceptron

- The perceptron is a neural network with no hidden layers



- The neuron does a weighted sum of the inputs
- The weighted sum goes through a normalization function, because we want the output to be between 0 and 1 (a probability)
- Example: sigmoid function

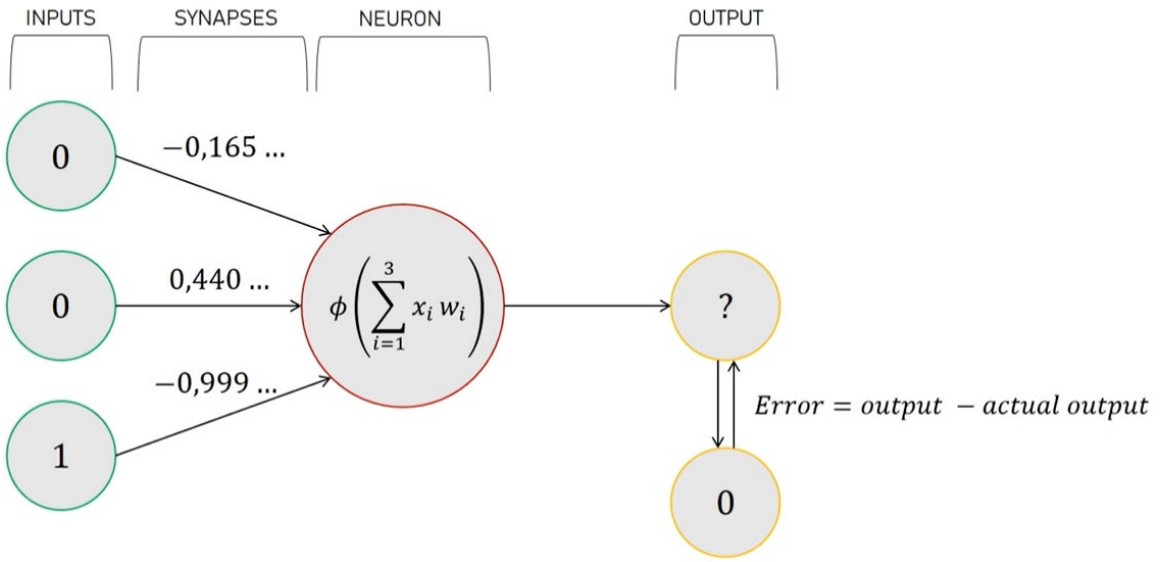
$$\phi(x) = \frac{1}{1+e^{-x}} \longrightarrow \phi(x) = \frac{1}{1+e^{-\sum_{i=1}^{3} x_i w_i}}$$



Source: https://youtu.be/kft1AJ9WVDk

Training process

1. Take the inputs from the training example, pass them to their formula to get the neuron's output (**Forward Propagation**)
2. Compute the error – the difference between the output we got and the actual output (the optimal output)
3. Adjust the weights accordingly (Back Propagation)
4. Repeat process in order to minimize this error



Assigning weights

```
import numpy as np
# sigmoid function to normalize inputs
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
# sigmoid derivatives to adjust synaptic weights
def sigmoid_derivative(x):
    return x * (1 - x)
# input dataset
training_inputs = np.array([[0,0,1],
                            [1,1,1],
                            [1,0,1],
                            [0,1,1]])
# outputs (the labels)
training_outputs = np.array([[0,1,1,0]]).T
# seed random numbers to make calculation (evalue of 1 means that the randomization is made in a specific, repeatable way)
np.random.seed(1)
# initialize weights randomly with mean 0 to create weight matrix, synaptic weights
synaptic_weights = 2 * np.random.random((3,1)) - 1
print('Random starting synaptic weights: ')
print(synaptic_weights)
```

Start training

```
# Iterate 10,000 times
for iteration in range(10000):
    # Define input layer
    input_layer = training_inputs
    # Normalize the product of the input layer with the synaptic weights
    outputs = sigmoid(np.dot(input_layer, synaptic_weights))
    # how much did we miss?
    error = training_outputs - outputs
    # multiply how much we missed by the
    # slope of the sigmoid at the values in outputs
    adjustments = error * sigmoid_derivative(outputs)
    # update weights
    synaptic_weights += np.dot(input_layer.T, adjustments)
print('Synaptic weights after training: ')
print(synaptic_weights)
print("Output After Training:")
print(outputs)
```

Thank you!