CMPE 232

Basics of Computational Graphs

İbrahim Ethem Karalı

Oğuzhan Bayram

Tolga Aslım

Burak Demirel

Aleyna Buse Güzel

What Is Computational Graph

- A **computational graph** is a directed **graph** where the nodes correspond to operations or variables. Variables can feed their value into operations, and operations can feed their output into other operations. This way, every node in the **graph** defines a function of the variables.

Implemented Classes

- represents a graph node that performs a computation.
- an `operation` is a node in a `graph` that takes zero or more objects as input, and produces zero or more objects as output.

```
class Operation:
    """Represents a graph node that performs a computation.

    An `Operation` is a node in a `Graph` that takes zero or
    more objects as input, and produces zero or more objects
    as output.
    """

    def __init__(self, input_nodes=[]):
        """Construct Operation
        """
        self.input_nodes = input_nodes

        # Initialize list of consumers (i.e. nodes that receive this operation's output as input)
        self.consumers = []

        # Append this operation to the list of consumers of all input nodes
        for input_node in input_nodes:
            input_node.consumers.append(self)

        # Append this operation to the list of operations in the currently active default graph
        _default_graph.operations.append(self)

    def compute(self):
        """Computes the output of this operation.
        "" Must be implemented by the particular operation.
        """
        pass
```

- Add Class is a sub class of Operation class it does the adding operation.

```
class add(Operation):
    """Returns x + y element-wise.
    """

    def __init__(self, x, y):
        """Construct add

        Args:
          x: First summand node
          y: Second summand node
        """
        self.string = '(' + x.string + '+' + y.string + ')'

        super().__init__([x, y])

    def compute(self, x_value, y_value):
        """Compute the output of the add operation

        Args:
          x_value: First summand value
          y_value: Second summand value
        """
        return x_value + y_value
```

- Multiply Class is a sub class of Operation class it does the multiplication.

```
class multiply(Operation):
    """Returns x * y element-wise.
    """

    def __init__(self, x, y):
        """Construct multiply

        Args:
          x: First multiplicand node
          y: Second multiplicand node
        """
        self.string = '(' + x.string + '*' + y.string + ')'

        super().__init__([x, y])

    def compute(self, x_value, y_value):
        """Compute the output of the multiply operation

        Args:
          x_value: First multiplicand value
          y_value: Second multiplicand value
        """
        return x_value * y_value
```

- Placeholder class is a class that can be created without a value ant this can

```
class placeholder:
    """Represents a placeholder node that has to be
provided with a value
       when computing the output of a computational graph
    """

    def __init__(self, p):
        """Construct placeholder
        """
        self.consumers = []
        self.string = p
        #print(self.string)

        # Append this placeholder to the list of
placeholders in the currently active default graph
        _default_graph.placeholders.append(self)
```

- Multiply Class is a sub class of Operation class it does the multiplication.

```
class Variable:
    """
    Represents a variable (i.e. an intrinsic,
    changeable parameter of a computational graph).
    """

    def __init__(self, initial_value):
        """Construct Variable

        Args:
          initial_value: The initial value of this variable
        """
        self.string = str(initial_value)
        self.value = initial_value
        self.consumers = []
        #print(self.string)

        # Append this variable to the list of variables in the
currently active default graph

        _default_graph.variables.append(self)
```

- Graph is a class that bundles all the operations, placeholders and variables together. When creating a new graph, we can call its `as_default` method to set the `_default_graph` to this graph. This way, we can create operations, placeholders and variables without having to pass in a reference to the graph every time.

```
class Graph:
    """Represents a computational graph
    """

    def __init__(self):
        """Construct Graph"""
        self.operations = []
        self.placeholders = []
        self.variables = []

    def as_default(self):
        global _default_graph
        _default_graph = self
```

- Session class encapsulates an execution of an operation. We would like to be able to create a session instance and call a `run` method on this instance, passing the operation that we want to compute and a dictionary containing values for the placeholders.

```
import numpy as np


class Session:
    """Represents a particular execution of a computational graph.
    """

    def run(self, operation, feed_dict={}):
        """Computes the output of an operation

        Args:
          operation: The operation whose output we'd like to compute.
          feed_dict: A dictionary that maps placeholders to values for this session
        """

        # Perform a post-order traversal of the graph to bring the nodes into the right order
        nodes_postorder = traverse_postorder(operation)

        # Iterate all nodes to determine their value
        for node in nodes_postorder:

            if type(node) == placeholder:
                # Set the node value to the placeholder value from feed_dict
                node.output = feed_dict[node]
            elif type(node) == Variable:
                # Set the node value to the variable's value attribute
                node.output = node.value
            else:  # Operation
                # Get the input values for this operation from node_values
                node.inputs = [input_node.output for input_node in node.input_nodes]

                # Compute the output of this operation
                node.output = node.compute(*node.inputs)

            # Convert lists to numpy arrays
            if type(node.output) == list:
                node.output = np.array(node.output)

        # Return the requested node value
        return operation.output
```

- In order to compute the function represented by an operation, we need to apply the computations in the right order. For example, we cannot compute ‘z’ before we have computed ‘y’ as an intermediate result. Therefore, we have to make sure that the operations are carried out in the right order, such that the values of every node that is an input to an operation ‘o’ has been computed before ‘o’ is computed to do that we need a helper function ;

```
def traverse_postorder(operation):
    """Performs a post-order traversal, returning a list of nodes
    in the order in which they have to be computed

    Args:
       operation: The operation to start traversal at
    """

    nodes_postorder = []

    def recurse(node):
        if isinstance(node, Operation):
            for input_node in node.input_nodes:
                recurse(input_node)
        nodes_postorder.append(node)

    recurse(operation)
    return nodes_postorder
```

How to run manually?

```
# Create a new graph
Graph().as_default()

# Create placeholder
#
a = placeholder('a')
b = placeholder('b')
c = placeholder('c')
d = placeholder('d')

# Create hidden node y
y = multiply(b, c)

# Create output node z
z = add(y, a)

J = multiply(d, z)

session = Session()
output = session.run(J, {
    a: 5 , b: 1, c: 2 ,d: 3
})
print("J = d (a + bc)")
print("if a=5, b=1, c=2 ,d=3;")

print("Result =", output)
```

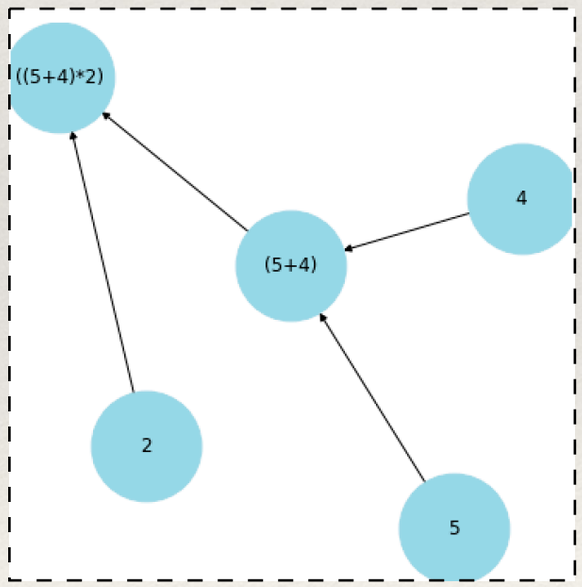
Output:

J = d (a + bc)
if a=5, b=1,
c=2 ,d=3;
Result = 21

```python
class node():
    def __init__(self, name):
        self.name = name
        self.neighbors = [] # list of nodes (not just names)

    def neighbors_name(self):
        """
        info about neighbors names
        """
        return [node_s.name for node_s in self.neighbors]

class digraph():
    def __init__(self, elist):
        """
            self.nodes is a dictionary
                key   : node name
                value : node class
        """
        self.elist = elist
        self.node_names = list(set([s for s,t in elist] + [t for s,t in elist]))
        self.nodes = {s:node(s) for s in self.node_names}

        self.create_graph()

    def add_edge(self, s,t):
        """directed Edge"""
        self.nodes[s].neighbors.append(self.nodes[t])

    def create_graph(self):
        for s,t in self.elist:
             self.add_edge(s,t)

    def info(self):
        return {s:node_s.neighbors_name() for s,node_s in self.nodes.items()}

    def draw(self, color = 'lightblue'):
        """
            Usage of networkx for visualisation
        """
        G = nx.DiGraph()
        G.add_edges_from(self.elist)
        plt.figure(figsize=(10,10))
        nx.draw(G, node_size=5000, node_color=color, with_labels=True)
```

We used digraph and node classes from our lecture book 'Algorithms' to draw computational graph as directed graph. We will create elist(which represent edge-list) in our next class called "program".

How it looks if equation is (2*(4+5)):



- Martix multipication and Sigmoid implementation.

```python
class matmul(Operation):
    """Multiplies matrix a by matrix b, producing a * b.
    """

    def __init__(self, a, b):
        """Construct matmul

        Args:
          a: First matrix
          b: Second matrix
        """
        self.string = ''
        super().__init__([a, b])

    def compute(self, a_value, b_value):
        """Compute the output of the matmul operation

        Args:
          a_value: First matrix value
          b_value: Second matrix value
        """
        return a_value.dot(b_value)
```

```python
class sigmoid(Operation):
    """Returns the sigmoid of x element-wise.
    """

    def __init__(self, a):
        """Construct sigmoid

        Args:
          a: Input node
        """
        self.string = ''
        super().__init__([a])

    def compute(self, a_value):
        """Compute the output of the sigmoid operation

        Args:
          a_value: Input value
        """
        return 1 / (1 + np.exp(-a_value))
```

Manual run;

```python
Graph().as_default()

x = placeholder('a')
w = Variable([1, 1])
b = Variable(0)
p = sigmoid(add(matmul(w, x), b))

session = Session()
print(session.run(p, {
    x: [3, 2]
}))
```

Output: `0.9933071490757153`

How to do all of these automatically

- Create a class called '*Program*' that takes a string as *func* and a dictionary as *feed_dict*.
- *func:* It contains the function as string.(e.g. "(a*(b+c))")
- *feed_dict:* It contains name of the placeholder as key and the given value of the placeholder as value. (e.g. "{a: 5, b: 2, c: 5}")

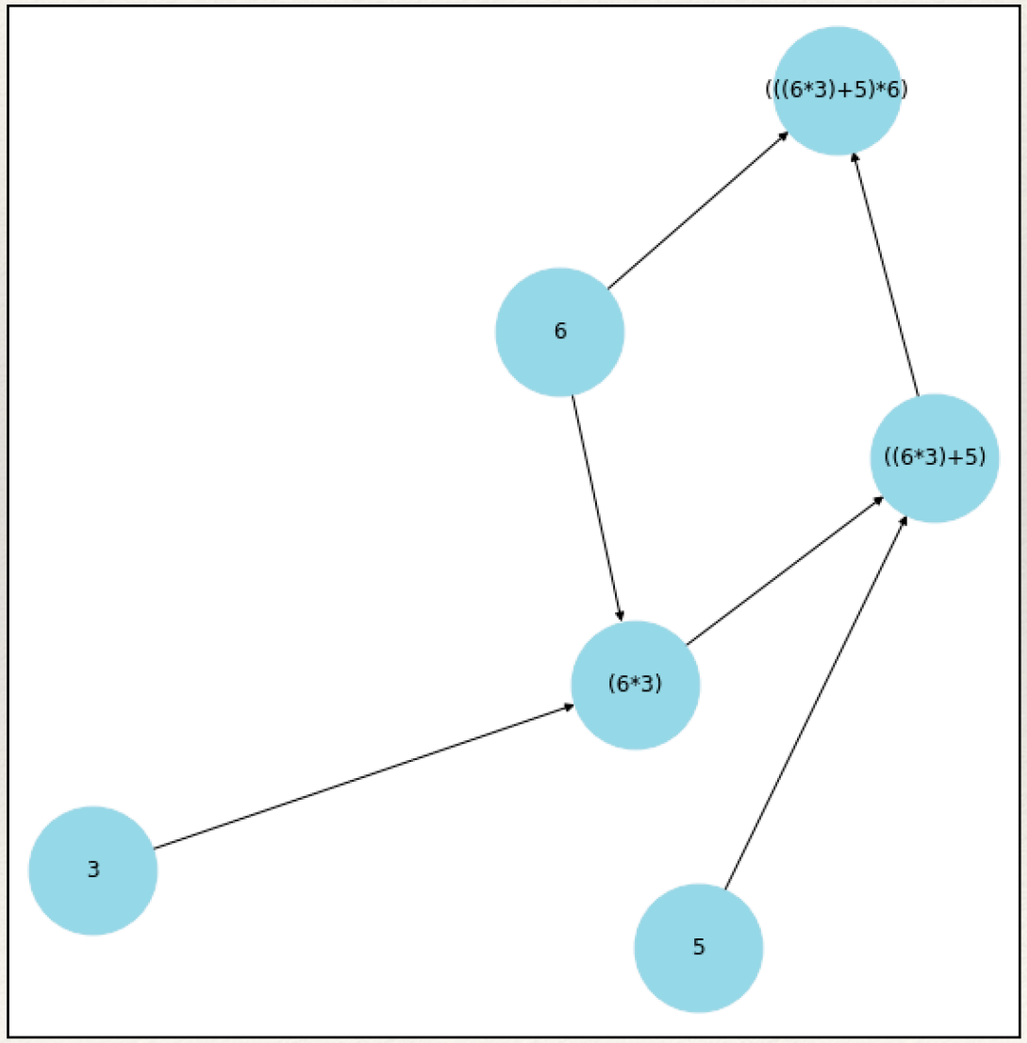
```
def program(func, var_dic):
    chars = list(func)
    ops = []  # list that have operations of given function
    vals = []  # list that have values of given function
    elist = []  # edge-list that stores edges between nodes
    textLength = chars.__len__()
    for index in range(textLength):

        if chars[index] == '(' or chars[index] == ' ':
            continue
        elif chars[index].__eq__('+'):
            ops.append('+')
        elif chars[index].__eq__('*'):
            ops.append('*')
        elif chars[index] == ')':  # if current char is ')' then pop the last added operation and two variable
            op = ops.pop()
            first = vals.pop()
            second = vals.pop()
            if op.__eq__('+'):
                vals.append(add(first, second))
                goes_to = vals.__getitem__(vals.__len__() - 1)
                elist.append((first.string, goes_to.string))
                elist.append((second.string, goes_to.string))
            elif op.__eq__('*'):
                vals.append(multiply(first, second))
                goes_to = vals.__getitem__(vals.__len__() - 1)
                elist.append((first.string, goes_to.string))
                elist.append((second.string, goes_to.string))
        elif chars[index].isalpha():  # if the char is a letter then it is placeholder
            vals.append(Variable(var_dic[chars[index]]))  # add placeholder as Variable with a constant value
        else:  # If the given value is a constant
            vals.append(Variable(int(chars[index])))

    session = Session()
    output = session.run(vals.pop(), var_dic)
    print('F =', func)
    if var_dic != {}:  # if var_dic have item print them
        print('if', var_dic)
    print("F =", output)

    G = digraph(elist)
    G.draw()
    G.info()
```

Run program with a function and values

```
program('(d*(a+(b*d)))', {'a': 5, 'b': 3, 'c': 2, 'd': 6})
```

Output:

F = (d*(a+(b*2)))
if {'a': 5, 'b': 3, 'c': 2, 'd': 6}
F = 66



- Thanks for looking into our project.
- Also, We wrote a project report to give more information about out team, project and development process.