CMPE 232 Course Project

Project Title: Mimicking TensorFlow

Group Name: Placeholder

Students:
1. Hasan Kemik 116207076
2. Ali Çağan Keskin 116200079

Abstract

The main purpose of this project is to convert equations to computation graphs. First, we have to identify nodes from the given equation. Each node can be:

- Placeholder
- Variable
- Operation

We have to identify the operations and variables. Secodly, we have to connect all nodes with using the relations which will be given in equation. After these challenging steps we have to make forward propagation and backward propagation to execute graph.

Index

Methodology

Needed Libraries

```
import numpy as np
from functools import reduce #python 3
import operator
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline
```

In this project, we've used numpy for computing values, network and matplotlib for visualization of the graphs, and reduce and operator for matrix computations as helper functions.

Classes

First we create a node class for varibles and operations.

- With input attribute is the list of nodes which comes into the node
- With output attribute is the list of nodes which goes from the node
- With n_type attribute is the Type of Node. With this attribute, we can classify the Node's operation.

```
class node():
    def __init__(self,inputs,outputs,n_type):

        self.inputs = inputs
        self.outputs = outputs
        self.n_type = n_type
```

This **node()** class also contains some other condition statements and functions to identify operations and variables.

For n_type it can be Add, MatMul, Multiply, Sigmoid, Subtract,Division, Variable or Placeholder.

The **node()** class is acting like a parent class for all Variable, Placeholder and Operation classes.

After we choose the node type, we use some classes to make the computation.

For example if we see a “+” sign, we call the compute() function in **add()** class. Or if we see a “/” sign, when we call the compute() function in **node()** class it calls the **division()**’s compute() function.

In [2]:
```python
# class for addition
class add():
    def __init__(self,inputs):
        self.inputs = inputs

    def compute(self):
        return sum(self.inputs)
```

In [3]:
```python
# class for substraction
class subtract():
    def __init__(self,inputs):
        self.inputs = inputs

    def compute(self):
        return (self.inputs[1] - self.inputs[0])
```

In [4]:
```python
# class for division
class division():
    def __init__(self,inputs):
        self.inputs = inputs

    def compute(self):
        return (self.inputs[0] / self.inputs[1])
```

In [5]:
```python
# class for matrix multiplication
class matmul():
    def __init__(self,inputs):
        self.inputs = inputs

    def compute(self):
        if len(self.inputs[0]) > 1:
            x = self.inputs[0]
            for y in self.inputs[1:]:
                x = np.dot(x,y)
        return x
```

In [6]:
```python
# class for activation function
class sigmoid():
    def __init__(self,inputs):
        self.inputs = inputs

    def compute(self):
        return 1 / (1 + np.exp(-self.inputs[0]))
```

In [7]:
```python
# class for multiplication
class multiply():
    def __init__(self, inputs):
        self.inputs = inputs

    def compute(self):
        try:
            x = reduce(lambda x, y: x * y, self.inputs, 1)
        except:
            x = self.inputs[0]*self.inputs[1]
        return x
```

In [8]:
```python
# class that holds varible
class Variable():
    def __init__(self, initial_value):
        self.value = initial_value

    def compute(self):
        return self.value
```

In [9]:
```python
# class for placeholder
class Placeholder():
    def __init__(self,value = None):
        self.value = value

    def change_value(self,value):
        self.value = value

    def compute(self):
        return self.value
```

And for visualization, we create **draw_graph** and **draw_reverse_graph** classes. We are using Python's Networkx Library for this. The graph we used for this task is a Directed Unweighted Graph.

```
# the class which visualises the graph.
class draw_graph():
    def __init__(self,node_list,edge_list,color_map,label_dict):
        self.G = nx.DiGraph()
        self.color_map = color_map
        self.G.add_nodes_from(node_list)
        self.G.add_edges_from(edge_list)
        self.G = nx.relabel_nodes(self.G,label_dict)

    def draw(self):
        nx.draw_spring(self.G,node_color = self.color_map ,node_size = 500, with_labels = True)
```

```
the class which visualises the reversed graph.
lass draw_reverse_graph():
  def __init__(self,node_list,edge_list,color_map,label_dict):
      self.color_map = color_map
      self.G = nx.DiGraph()
      edge_list = reversedGraph().reverse(edge_list)
      self.G.add_nodes_from(node_list)
      self.G.add_edges_from(edge_list)
      self.G = nx.relabel_nodes(self.G,label_dict)
      #print(edge_list)

  def draw(self):
      nx.draw_circular(self.G,node_color = self.color_map ,node_size = 500, with_labels = True)
```

And for operation order we create a **DirectedDFS** class that will use *Depth First Search* which will give us the traversal of the graph.

```
# the class for Depth First Search
class DirectedDFS():
    def __init__(self, G, node_list):
        self.G = G
        self.visited = {node_name: False for node_name in node_list}
        self.traversal = []

    def dfs(self, sname):
        self.visited[sname] = True
        self.traversal.append(sname)
        for neighbor in sname.inputs:
            try:
                if (not self.visited[neighbor]):
                    self.dfs(neighbor)
            except:
                continue
    def return_traversal(self):
        x = self.traversal
        return x
```

Functions

Since that the equation will be given as a String, this is the function for understanding the characters we will be using *Djikstra's Two Stack Algorithm* to inspect the equation. This algorithm basically reads the string and understands if there's a number or an operation. It uses two stacks for numbers and operations. If the algorithm reads a closing parenthesis, it understands that there's an prior operation and executes it. After string is finished. It makes all the operations until the operation stacks' size is 0.

We define it in **math_operands_split(mat_op**) function. And here is a small example for understand how algorithm works.

In our case, we read the string twice first create the nodes, and then make the operations.

```
if len(mat_op) == 1 and type(mat_op[0]) == str:
    mat_op = list(mat_op[0])
    for char in mat_op:
        if char in operators:
            op_stack_cont.append(char)
        elif char not in operators and char != '(' and char != ')':
            numb_stack.append(char)
            numb_stack_cont.append(char)
    placeholder_nodes.append([node([],[],["Placeholder"]) for x in range (len(numb_stack)+1
    variable_nodes.append([node([x],[],["Variable"]) for x in numb_stack])

    numb_stack = []
    op_stack = []
    i = 0
    for char in mat_op:
        if char == ')' and len(op_stack) > 0:
            operation = op_stack.pop()
            val1 = numb_stack.pop()
            val2 = numb_stack.pop()
            if operation is '+':
                x = (node([val1,val2],[],["Operation","Add"]))
                operation_nodes.append(x)
                label_dict[x] = ("+"+ "/" +str(i))
                i = i + 1
                numb_stack.append(x)
            elif operation is '*':
```

After all, we create **forward_propogation(my_traversal)** function for Forward Propogation and create **backward_propogation(my_traversal)** function for Backward Propagation.

This is the Forward Propagation, which will compute the graph based on it's priority list which will be used as it's computing order. It takes the data's from the data dictionary user gave, and maps the inputs of the node to make the computation.

```
def forward_propogation(my_traversal):
    while len(my_traversal) > 0:
            my_node = my_traversal.pop()
            if my_node.return_type()[0] == "Placeholder":
                x = [data[number] for number in my_node.inputs]
                my_node.change_value(x)
                print(my_node.return_val())
            else:
                print(label_dict[my_node])
                x = [data[number] for number in my_node.inputs]
                print(x,"mapped values")
                my_node.change_inp(x)
                y = my_node.compute()
                print(y,"result")
                data[my_node] = y
```

The backward propagation is using a really simple algorithm which works like, add a delay to the signal, then take the difference from the value before delay, and divide it with the delay time. Which uses the basics of the derivative definition.

```
def backward_propogation(my_traversal):
    while len(my_traversal) > 0:
        my_node = my_traversal.pop()
        if my_node.return_type()[0] != 'Placeholder':
            if my_node.return_type()[1] != "Subtract":
                print(label_dict[my_node],"my node")
                x = []
                for inputs in my_node.inputs:
                    print(data[inputs],"inputs")
                    inputs_default = data[inputs]
                    changed_input = inputs_default + 0.001
                    x.append(changed_input)
                i = 0
                for number in my_node.inputs:
                    my_node.change_inp([data[number],x[i-1]])
                    y = my_node.compute()
                    the_diff = y -data[my_node]
                    print(round(the_diff/0.001),"The Derivative")
                    i = i + 1
```

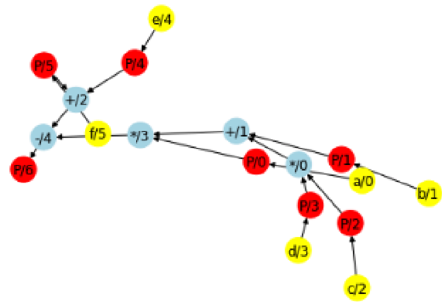
An example of the expected output from the equation is:

In [27]:
```
# [node_list, edge_list,color_map,label_dict,reverse_start_node] = math_operands_split("a*(b + (c*d)) - e + f")
# data = {'a':5.0,'b':3.0,'c':2.0,'d':4.0,'e':5.0,'f':2.0}
```

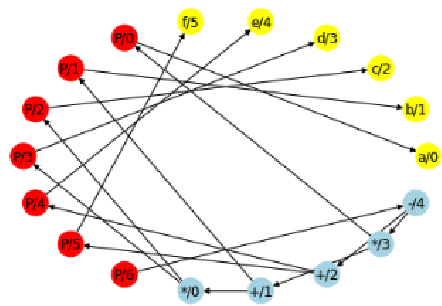
In [28]:
```
[node_list, edge_list,color_map,label_dict,reverse_start_node] = math_operands_split("a*(b + (c*d)) - (e + f)")
data = {'a':5.0,'b':3.0,'c':2.0,'d':4.0,'e':5.0,'f':2.0}
```

In [19]:
```
G1 = draw_graph(node_list,edge_list,color_map,label_dict)
G2 = draw_reverse_graph(node_list,edge_list,color_map,label_dict)
```

In [20]:
```
G1.draw()
```



In [21]:
```
G2.draw()
```



Forward Propogation

In [23]:
```
my_traversal = []
```

In [24]:
```
dfs = DirectedDFS(G2,node_list)
dfs.dfs(reverse_start_node)
none = [my_traversal.append(my_node) for my_node in dfs.return_traversal()]
```

In [25]:
```
forward_propogation(my_traversal)
```

```
*/0
[4.0, 2.0] mapped values
8.0 result
+/1
[8.0, 3.0] mapped values
11.0 result
*/3
[5.0, 11.0] mapped values
55.0 result
+/2
[2.0, 5.0] mapped values
7.0 result
-/4
[7.0, 55.0] mapped values
48.0 result
[48.0]
```

Backward Propogation

In [26]:
```
none = [my_traversal.append(my_node) for my_node in dfs.return_traversal()]
```

In [27]:
```
backward_propogation(my_traversal)
```

```
*/0 my node
4.0 inputs
2.0 inputs
4 The Derivative
2 The Derivative
+/1 my node
8.0 inputs
3.0 inputs
1 The Derivative
1 The Derivative
*/3 my node
5.0 inputs
11.0 inputs
5 The Derivative
11 The Derivative
+/2 my node
2.0 inputs
5.0 inputs
1 The Derivative
1 The Derivative
-/4 my node
7.0 inputs
55.0 inputs
1 The Derivative
-1 The Derivative
```