Computation Graph

Group Name:

3 Authors Including:

Name	**Sur name**	**ID**
Erol	YIGITALP	
Baris	AREM	
Berker	ARSLAN	117200033

Project Link

Link **->** **Github**

Istanbul Bilgi University

Istanbul, Turkey

Abstract

This paper concludes basic implementation of computational graph for better understanding deep learning synapsis and forward, backward propagations.

Introduction

Past 7 years deep learning getting popular with unique implementations and getting one of the main structures of our daily lives. For a Data Scientist Student we suppose to use deep learning algorithms in our analysis models for improve our works efficiency but first of all we suppose to understand the basic idea behind deep learning operations like Tensor nodes. In this project our goal is putting clarify on specific question; 'How a Tensor Graph Works behind the scene?'

This paper is organized as follows: Section 1 describes which data structures selected for which operations. Section 2 describes Operation and Placeholder node's properties. Section 3 describes how to solve first problem; 'How to convert an equation string to process able type? '. Section 4 describes how can use process equation in a graph with Operation and Placeholder nodes. Section 5 describes How to terminate Forward Propagation on ordered graph list. Section 6 describes How to terminate Backward Propagation on reversed graph list. Section 7 shows the general diagram about how this algorithms works together. Section 8 shows the example outputs of a given 2 equations.

Section 1: Data Structures and Libraries

This section describes reasons of used structures and libraries:

- Class declarations for combining String converter methods with Graph generate methods.
- For Loops for iterate over given objects.
- Array structures for store Topologic orders and graph variables.
- Tuples for store derivative values with respect to connected inputs and creating data frame of initialized graph.
- Pandas Library's Data Frame method for debug option.
- Numpy Library's calculation methods for Matrix calculations.
- If-else blocks for avoid unnecessary calculations
- Networkx library for initializing graphs and getting Topologic orders
- Mat plot library for draw output figures.

Section 2 Operations and Placeholders

This section describes properties of Operation and Placeholder nodes

Placeholder Node:

- Name Variable which holds first input for drawing initial graph
- Value variable which holds updated input name
- Weight variable which holds derivate respect to itself for connected operation
- Neighbor variable which holds the connected node Id

Operation nodes:

- X,Y variables which holds two input nodes Id information
- Weight variable which holds derivate respect to itself for connected

- Derivative variable which is a tuple filled with previous two nodes id-weight pairs
- Name Variable which holds string type of symbolic calculation between input X and input Y
- Value variable which holds calculation result
- Derivative Method which finds differential between f and f(X+1) and divide result with 1
- Neighbor variable which holds connected nodes Id's

Section 3: String Process

This section describes string type equation to process able type algorithm which named 'BasicC'. BasicC algorithm takes 1 string input which refers to an equation. First it splits unnecessary space's, and ')' characters and add space after each character for identify operation strings like '*' for fit it to process () method. In process method first it split's each '('for help to determine order of the function after that it puts every character on a two dimensional list which rows were number of functions and columns were the inside operations of each function. After Process () is done algorithm calls Framize () method for fit processed equation to operation and place holder nodes. Time complexity of this operation is n*log (n).

Example Output -> Figure1-Figure4

BasicC (string):

```
1. string.split('',')')
2. string.add(' ')
3. process(string) → for fit string to
   Framize method
4. Framize(string) → for fit string to node
   objects
5. draw(Graph) → Display Graph
6. Neighbors(Graph) → Fill neighbors
   variables
```

Process (string):

```
1. For I in space: →spited string
    a. For z in I :
        i. If not z = ' ' : Final.
           append(helper(z))
```

Helper (string):

```
1. Return Change.String{Operand→Name}
   : like '*' to 'Multi'
2. end
```

Framize (string, Boolean):

```
1. if length (string)=>3:
    a. addOperation(string(0,2,1))
    b. String. subtract(0,1,2)
    c. Framize(false)
2. if length (string)=2:
    a. combineOp(string(0,1),lastOp)
    b. String. subtract(0,1)
    c. Framize(false)
3. if length (string)=1:
    a. CombineFull(lastOp,Memory,string)
    b. String. subtract(0)
    c. Framize(true)
```

Section 4: Graph Initializer

This section describes Graph initializer methods which used in above String process algorithm.

Main purpose of this class is design placeholders and operation nodes respect to given inputs and connect them in two different arrays which one of them represents visual characters for initial graph drawing and node names with connected nodes for Session class.

Time Complexity of this Algorithm is constant time.

AddOperation (X, Y, o):

```
1. X = placeholder(X) → initialize Input
2. Y = placeholder(Y) → initialize Input
3. Operation = eval(o)(X,Y)→ format string
   to function with given inputs ex. : x*y
   while o = 'Multi'
4. Connections (X, Y, Operation)
   →establish network
```

CombineOperation(X, y, o):

```
1. X = Placeholder(X) → initialize Input
2. Y = operations(y) → y'th operation
3. Operation = eval(o)(X,Y)→ format string
   to function with given inputs ex. :
   a*(x*y) while o = 'Multi'
4. Connections (X, Y, Operation)
   →establish network
```

CombineFull(x, y, o):

```
1. X = operations(x) → x'th operation
2. Y = operations(y) → y'th operation
3. Operation = eval(o)(X,Y)→ format string
   to function with given inputs ex. :
   (a*b)/(x*y) while o = 'Div.'
4. Connections (X, Y, Operation)
   →establish network
```

Connections(X, Y, Z):

```
1. Visual.append(X.name,Z.name,0)
2. Visual.append(Y.name,Z.name,0)
3. Proccesable.append(X,Z,0)
4. Processable.append(Y,Z,0)
```

Section 5: Forward Propagation

This section describes Forward Prop. Method for Session class which we calculate given equation with given variables. This algorithm runs in linear Time complexity N.

Ex. Figure2-Figure6

ForwardSession (Procurable, Variables):

```
1. addVariables(Variables)→update node
   values
2. T = TopologicOrder(Procurable)→get
   Topologic order for calculations
3. For I in T:
     a. If type I == Operation:
          i. I.calc()
4. Graph = (i.val,i.neighbors.val,i.val for i in
   T)
5. Draw(graph)
6. BackwardSession(T)
```

Section 6: Backward Propagation

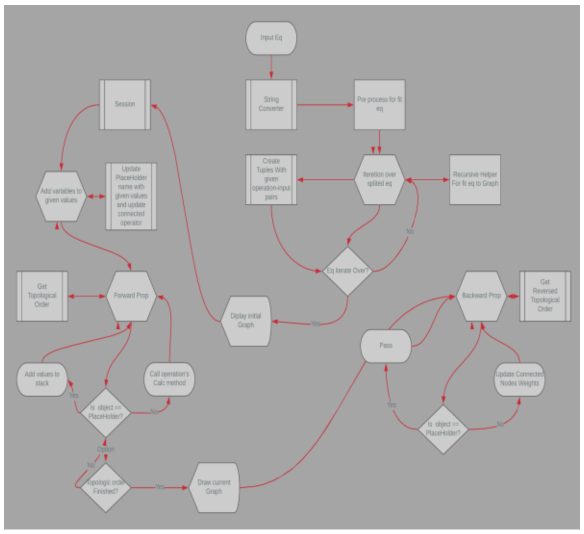
This section describes Backward Prop. Method for session class which calculates derivatives from reversed Topological Order. This algorithm runs in Linear Time complexity N.

Ex. Figure3-Figure5

BackwardSession (T):

```
1. T = reversed(T) → for get Reversed Topological order
2. For I in T:
    a. If type = Operation:
        i. I.derivative()
        ii. I.x.weight = I.der(I.x)
        iii. I.y.weight = I.der(I.y)
3. Graph = (i.val,i.neighbors.val,i.weight for i in T) → initialize reverse graph with new weights
4. Draw(Graph)
```

Section 7 General Diagram


Input Eq
Session
String Converter
Pre process for fit eq
Add variables to given values
Update PlaceHolder name with given values and update connected operator
Create Tuples With given operation-input pairs
Iteration over splited eq
Recursive Helper For fit eq to Graph
No
Eq iterate Over?
Get Topological Order
Forward Prop
Diplay initial Graph
Yes
Backward Prop
Get Reversed Topological Order
Pass
Add values to stack
Call operation's Calc method
Yes
Is object == PlaceHolder?
No
Update Connected Nodes Weights
Yes
Is object == PlaceHolder?
No
Option
No
Topologic order Finished?
Yes
Draw current Graph

Section 8 Example Figures

```
Equation = 'a*b*(c*v+(x/y+z))'
Variables = ['a',4],['b',5],['x',8],['y',9],['z',10],['c',7],['v',6]
```

Figure 1:

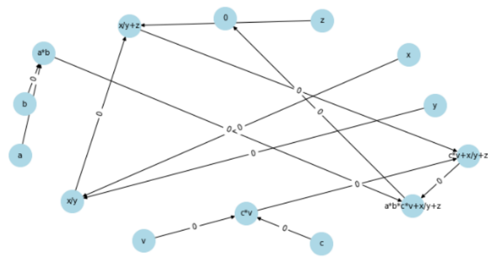


Figure 2:

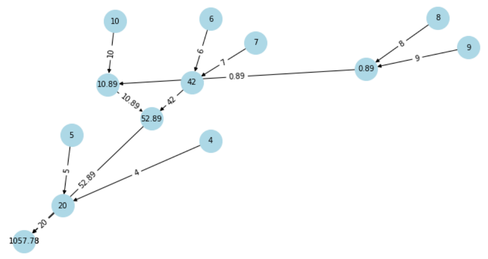
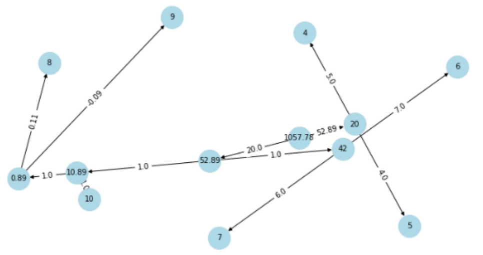


Figure 3:



```
Equation = 'aXb+(zXc)'
Variables = ['a',[0,1,2]],[
'b',[2,2,2]],['c',[2,3,2]],['z',[4
,2,2]]
```

Figure 4:

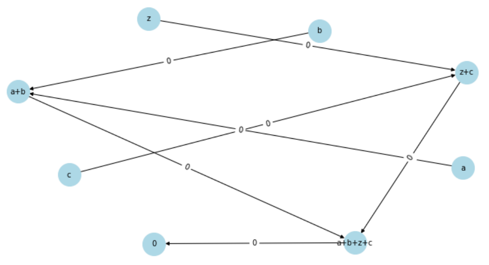


Figure 5:

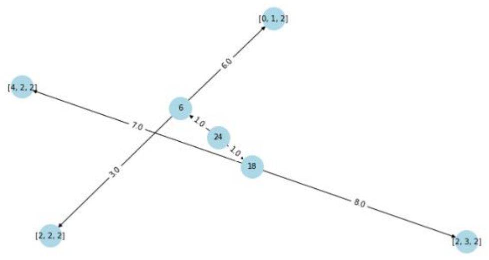
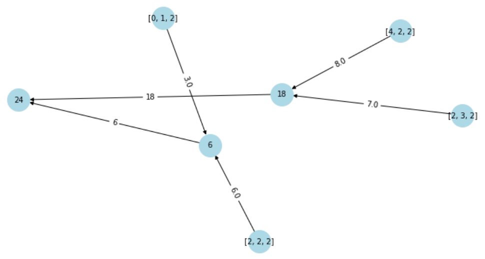


Figure 6:



References

1. Deep Learning From Scratch I: Computational Graphs
2. Understand TensorFlow by mimicking its API from scratch