# Manual Neural Network

This is really cool, and it's something I've wanted to do. I've got this and several other ways to do a similar thing. This one gets done first. It's going to mimic the TensorFlow API. When I get back to TensorFlow, I should have a better understanding.

From Jose

> In this notebook we will manually build out a neural network that mimics the TensorFlow API. This will greatly help your understanding when working with the real TensorFlow!

In [1]:
```
##  It can be useful to see errors to know how to fix them.
##+ However, it messes with the "compute all cells" type of
##+ stuff. Here, you can decide whether to see the errors or
##+ not. (In some places, I've put the error text in
##+ markdown cells.)

do_show_errors = False
```

## Some Info About super() and Object Oriented Programming in General

In [2]:
```
class SimpleClassLecture0():

    def __init__(self):
        print("hello")
    ##endof:  __init__(self)



##endof:  SimpleClassLecture0
```

In [3]:
```
s = "world"
```

In [4]:
```
type(s)
```

Out[4]:
```
str
```

In [5]:
```
# s.<then press [Tab]>
# Gives a list of methods
```

In [6]:
```
x0 = SimpleClassLecture0
```

In [7]:
```
x0 # what we get without the parentheses - __init__ doesn't get called
```

Out[7]:
```
__main__.SimpleClassLecture0
```

In [8]:
```
x0 = SimpleClassLecture0()
```

```
hello
```

In [9]:
```
x0 # Instance of SimpleClassLecture and where it exists in memory
```

Out[9]:
```
<__main__.SimpleClassLecture0 at 0x2c8dba824a8>
```

In [10]:
```
class SimpleClassLecture1():

    def __init__(self):
        print("hello")
    ##endof:  __init__(self)

    def yell(self):
        print("YELLING")
    ##endof:  yell(self)

##endof:  SimpleClassLecture1
```

In [11]:
```
x1 = SimpleClassLecture1()
```

```
hello
```

In [12]:
```
#  I'm going to type 'x1.' then hit [Tab].
#+ it will autocomplete 'x1.yell', after
#+ which I'll add the parenthesis
x1.yell()
```

```
YELLING
```

In [13]:
```
# Now, I'll just type it all out.
x1.yell()
```

```
YELLING
```

In [14]:
```
##  adding in this illustration. These first calls will work fine.
sc = SimpleClassLecture1()
print("--- some separation ---")
sc.yell()
```

```
hello
--- some separation ---
YELLING
```

In [15]:
```
##  continuing with the illustration. This is called
##+ as if it were the lecture notes. It will throw
##+ an error/exception/whatever-you-want-to-call-it
if do_show_errors:
    sc_oops = SimpleClassLecture1("Basket Weaving 101")
    print("--- some separation ---") # won't execute b/c error before
    sc_oops.yell()                   # won't execute b/c error before
##endof:  if do_show_errors
```

OUTPUT (error) should be

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-87-6c1cabf9d07d> in <module>()
      2 ##+ as if it were the lecture notes. It will throw
      3 ##+ an error/exception/whatever-you-want-to-call-it
----> 4 sc_oops = SimpleClassLecture1("Basket Weaving 101")
      5 print("--- some separation ---") # won't execute b/c error before
      6 sc_oops.yell()                  # won't execute b/c error before

TypeError: __init__() takes 1 positional argument but 2 were given
```

**Remember the code:**

```
class SimpleClassLecture1():

    def __init__(self):
        print("hello")
    ##endof:  __init__(self)

    def yell(self):
        print("YELLING")
    ##endof:  yell(self)

##endof:  SimpleClassLecture1
```

In [16]:
```
class ExtendedClassLecture0(SimpleClassLecture1):

    def __init__(self):

        print("EXTEND!")

    ##endof:  __init__(self)

##endof:  ExtendedClassLecture0(SimpleClassLecture1)
```

In [17]:
```
y0 = ExtendedClassLecture0()
#  Remember, there's no 'super' call for '__init__'
```

```
EXTEND!
```

In [18]:
```
#  No 'super' with '__init__', but other things work
y0.yell()
```

```
YELLING
```

Now, let's use the `super` keyword.

In [19]:
```
class ExtendedClassLecture1(SimpleClassLecture1):

    def __init__(self):

        super().__init__()
        print("EXTEND!")
    ##endof:  __init__(self)

##endof:  ExtendedClassLecture(SimpleClassLecture)
```

In [20]:
```
y1 = ExtendedClassLecture1()
```

```
hello
EXTEND!
```

In [21]:
```
y1.yell()
```

```
YELLING
```

Here, we're going to add an argument to the `SimpleClass __init__` (i.e. its constructor). Since this is the final state in which Jose leaves it, I'm going to use `SimpleClassLecture` instead of continuing with `SimpleClassLecture2`. I'll do similarly with the extended class - using `ExtendedClassLecture` instead of staying with the pattern and using `ExtendedClassLecture2`.

In [22]:
```
class SimpleClassLecture():

    def __init__(self, name):
        print("hello " + name) #  Jose put the space here, which
                               #+ I consider the correct place.
            #+ a minute or so after 1701113954_2023-11-27T123914-0700
    ##endof:  __init__(self)

    def yell(self):
        print("YELLING")
    ##endof:  yell(self)

##endof:  SimpleClassLecture1
```

In [23]:
```
x = SimpleClassLecture("Dave")
```

```
hello Dave
```

In [24]:
```
x.yell()
```

```
YELLING
```

In [25]:
```
class ExtendedClassLecture(SimpleClassLecture):

    def __init__(self):

        super().__init__("Davidushka!")
        print("EXTEND!")

    ##endof:  __init__(self)

##endof:  ExtendedClassLecture(SimpleClassLecture)
```

In [26]:
```
y = ExtendedClassLecture()
```

```
hello Davidushka!
EXTEND!
```

In [27]:
```
y.yell()
```

```
YELLING
```

---

**From the class material**

In [28]:
```
class SimpleClass():

    def __init__(self, str_input):
        #  DWB: I'm not fixing his lack of space after "SIMPLE".
        #+      1701111285_2023-11-27T115445-0700
        print("SIMPLE" + str_input)
    ##endof:  __init__(self, str_input)

##endof:  SimpleClass
```

I'll do the same two illustrations.

In [29]:
```
if do_show_errors:
    sc = SimpleClass() # will throw an error
##endof:  if do_show_errors
```

OUTPUT (which is an error) should be:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-29-1a19d7d610fd> in <module>()
----> 1 sc = SimpleClass() # will throw an error

TypeError: __init__() missing 1 required positional argument: 'str_input'
```

In [30]:
```
##  This one should work fine, though the lack of a space between
##+ "SIMPLE" and "Basket Weaving 101" - i.e.
##+ "SIMPLEBasket Weaving 101", grates on my nerves a bit. Q&R
sc = SimpleClass("Basket Weaving 101")
```

```
SIMPLEBasket Weaving 101
```

Remember the code (defined in the lecture notes)

```
class SimpleClass():

    def __init__(self, str_input):
        #  DWB: I'm not fixing his lack of space after "SIMPLE".
        #+      1701111285_2023-11-27T115445-0700
        print("SIMPLE" + str_input)
    ##endof:  __init__(self, str_input)

##endof:  SimpleClass
```

In [31]:
```
class ExtendedClassNoSuper(SimpleClass):

    def __init__(self):
        print('EXTENDED')
    ## endof:  __init__(self)

##endof:  ExtendedClassNoSuper
```

In [32]:
```
s = ExtendedClassNoSuper()
```

```
EXTENDED
```

With the output, remember that we *overwrote* the `__init__(self)` method.

What I'll call `ExtendedClass` is building upon the `ExtendedClassNoSuper` code. I could have added `Super` at the end ( `ExtendedClassSuper` ), or I could have done as the lecture notes did and call both `ExtendedClass` , with one replacing the other. Anyway, `ExtendedClass` will use `super` .

In [33]:
```
#  remember to use 'class' instead of 'def'
#+ (Oops, DWB 1701111919_2023-11-27T120519-0700)

class ExtendedClass(SimpleClass):

    def __init__(self):

        super().__init__(" My String") # Jose puts the space in the string here.
        print('EXTENDED')

    ##endof:  def __init__(self)

##endof:  ExtendedClass
```

In [34]:
```
s = ExtendedClass()
```

```
SIMPLE My String
EXTENDED
```

## We've finished learning some OOP stuff - now for the Manual NN

I've put in a bunch of stuff which should give some general idea of what's going on (though there will be a lot of memory addresses rather than useful info). You can turn this on or off in the next cell.

In [35]:
```
global_do_show_steps_bool = True
```

# Operation

## Lecture Version - with Dave's additions

In [36]:
```
class Operation():

    def __init__(self, input_nodes=[],
                 do_show_steps=global_do_show_steps_bool):

        if do_show_steps:
            dashes = "-"*50
            print("\n\n" + dashes)
            print("In __init__")
            print()
        ##endof:  if do_show_steps

        self.input_nodes = input_nodes

        if do_show_steps:
            print("\n Now,   self.input_nodes = ")
            print("     " + str(self.input_nodes))
            print()
        ##endof:  if do_show_steps

        self.output_nodes = []

        for node in input_nodes:
            if do_show_steps:
                print("\n Current node is:")
                print("     node = " + str(node))
                print()
```

```
            ##endof:  if do_show_steps

            node.output_nodes.append(self)

            if do_show_steps:
                print("\n After assignment, node.output_nodes.append(self)")
                print("     node = " + str(node))
                print()
                print(dashes)
                print()
            ##endof:  if do_show_steps

        ##endof:  for node in input_nodes

        if do_show_steps:
            print("\n Before appending  self  to  _default_graph,")
            print("     _default_graph = ")
            print("                " + str(_default_graph))
            print()
        ##endof:  if do_show_steps

        _default_graph.operations.append(self) #  Came back to add this
                                               #+ after we had created
                                               #+ the graph class

        if do_show_steps:
            print("\n After appending  self  to  _default_graph,")
            print("     _default_graph = ")
            print("                " + str(_default_graph))
            print()
        ##endof:  if do_show_steps

    ##endof:  __init__(self, input_nodes=[]):

    def compute(self):
        pass
    ##endof:  compute(self)

##endof:  Operation
```

## Course Notes Version

In [37]:
```
class OperationCNV():
    '''
    An Operation is a node in a "Graph". TensorFlow will also use this concept of a Graph.

    This Operation class will be inherited by other classes that actually compute the specific
    operation, such as adding or matrix multiplication.
    '''

    def __init__(self, input_nodes=[]):
        '''
        Initialize an Operation
        '''

        self.input_nodes = input_nodes #  The list of input nodes coming in to the node
        self.output_nodes = []          #  List of nodes that will consume the output
                                       #+ of this node

        #  For every node in the input, we append this operation (self) to the list of
        #+ to the list of the input nodes' consumers (i.e. this operation becomes an
        #+ output node)
        for node in input_nodes:
            node.output_nodes.append(self)
        ##endof:  for node in input_nodes

        #  There will be a global default graph (TensorFlow works this way)
        #+ We will append this particular operation (to the global default graph)
        #
        #  Append this operation to the list of operations in the currently-active
        #+ default graph
        _default_graph.operations.append(self)

    ##endof:  __init_(self, input_nodes=[])

    def compute(self):
        '''
        This is a placeholder function. It will be overwritten by the actual specific operation
        that inherits from this class
        '''
```

```
        pass

    ##endof:  compute(self)

##endof:  class OperationCNV()
```

# Example Operations

## Addition

### Lecture Version - with Dave's additions

In [38]:
```
class Add(Operation):

    def __init__(self, x, y,
                 do_show_steps=global_do_show_steps_bool):

        if do_show_steps:
            dashes = "-"*35
            print("\n" + dashes)
            print("\n Initializing an  Add  operation")
            print()
        ##endof:  if do_show_steps
        super().__init__([x, y])

    ##endof:  __init__(self, x, y)

    def compute(self, x_var, y_var):

        if do_show_steps:
            print("\n Now, computing the  Add  operation ")
            print()
        ##endof:  if do_show_steps

        self.inputs = [x_var, y_var]

        if do_show_steps:
            print("\n Now,   self.inputs = ")
            print("     " + str(self.inputs))
            print()
        ##endof:  if do_show_steps

        result_of_add = x_var + y_var

        if do_show_steps:
            print("\n We will return")
            print("     result_of_add = " + str(result_of_add))
            dashes = "-"*35
            print(dashes)
            print()
        ##endof:  if do_show_steps

        return result_of_add
```

```
    ##endof:  compute(self, x_var, y_var):

##endof:  class Add(Operation)
```

### Course Notes Version

In [39]:
```
class addCNV(OperationCNV):

    def __init__(self, x, y):

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

    ##endof:  __init__(self, x, y)

    def compute(self, x_var, y_var):

        self.inputs = [x_var, y_var]
        return x_var + y_var

    ##endof:  compute(self, x_var, y_var)

##endof:  addCNV(OperationCNV)
```

## Multiplication

### Lecture Version - with Dave's additions

In [40]:
```
class Multiply(Operation):

    def __init__(self, x, y,
                 do_show_steps=global_do_show_steps_bool):

        if do_show_steps:
            dashes = "-"*35
            print("\n" + dashes)
            print("\n Initializing a  Multiply  operation")
            print()
        ##endof:  if do_show_steps
        super().__init__([x, y])
    ##endof:  __init__(self, x, y)

    def compute(self, x_var, y_var):

        if do_show_steps:
            print("\n Now, computing the  Multiply  operation ")
            print()
        ##endof:  if do_show_steps

        self.inputs = [x_var, y_var]

        if do_show_steps:
            print("\n Now,   self.inputs = ")
            print("     " + str(self.inputs))
            print()
        ##endof:  if do_show_steps

        result_of_multiply = x_var * y_var

        if do_show_steps:
            print("\n We will return")
            print("     result_of_multiply = " + str(result_of_multiply))
            dashes = "-"*35
            print(dashes)
            print()
        ##endof:  if do_show_steps

        return result_of_multiply

    ##endof:  compute(self, x_var, y_var):
```

```
##endof:  class Multiply(Operation)
```

### Course Notes Version

In [41]:
```
class multiplyCNV(OperationCNV):

    def __init__(self, a, b):

        super().__init__([a, b])

    ##endof:  __init__(self, a, b)

    def compute(self, a_var, b_var):

        self.inputs = [a_var, b_var]
        return a_var * b_var

    ##endof:  compute(self, a_var, b_var)

##endof:  multiplyCNV(OperationCNV)
```

## Matrix Multiplication

### Lecture Version - with Dave's additions

In [42]:
```
class MatMul(Operation):

    def __init__(self, x, y,
                 do_show_steps=global_do_show_steps_bool):
        if do_show_steps:
            dashes = "-"*35
            print("\n" + dashes)
            print("\n Initializing a  MatMul  operation")
            print()
        ##endof:  if do_show_steps
        super().__init__([x, y])
    ##endof:  __init__(self, x, y)

    def compute(self, x_var, y_var):

        if do_show_steps:
            print("\n Now, computing the  MatMul  operation ")
            print()
        ##endof:  if do_show_steps

        self.inputs = [x_var, y_var]

        if do_show_steps:
            print("\n Now,   self.inputs = ")
            print("     " + str(self.inputs))
            print()
        ##endof:  if do_show_steps

        #  We're assuming we have numpy arrays (matrices), so we can
        #+ use the  var.dot()  operation
        result_of_matmul = x_var.dot(y_var)

        if do_show_steps:
            print("\n We will return")
            print("     result_of_matmul = " + str(result_of_matmul))
            dashes = "-"*35
            print(dashes)
            print()
        ##endof:  if do_show_steps

        return result_of_matmul
```

```
    ##endof:  compute(self, x_var, y_var):

##endof:  class MatMul(Operation)
```

### Course Notes Version

In [43]:
```
class matmulCNV(OperationCNV):

    def __init__(self, a, b):

        super().__init__([a, b])

    ##endof:  __init__(self, a, b)

    def compute(self, a_mat, b_mat):

        self.inputs = [a_mat, b_mat]
        return a_mat.dot(b_mat)

    ##endof:  compute(self, a_mat, b_mat)

##endof:  matmulCNV(OperationCNV)
```

# Placeholders

### Lecture Version - with (maybe) Dave's additions

In [44]:
```
class Placeholder():

    def __init__(self):

        self.output_nodes = []

        _default_graph.placeholders.append(self) #  this is added in during
                                                 #+ the course of the lecture.
                                                 #+ whereas those before
                                                 #+ weren't


    ##endof:  __init__(self)

##endof:  Placeholder()
```

**Course Notes Version**

In [45]:
```
class PlaceholderCNV():
    '''
    A placeholder is a node that needs to be provided a value for
    computing the output in the graph.
    '''

    def __init__(self):

        self.output_nodes = []

        _default_graph.placeholders.append(self) #  this is added in during
                                                 #+ the course of the lecture.
                                                 #+ whereas those before
                                                 #+ weren't


    ##endof:  __init__(self)

##endof:  PlaceholderCNV()
```

# Variables

### Lecture Version - with (maybe) Dave's additions

In [46]:
```
class Variable():

    def __init__(self, initial_value=None):

        self.value = initial_value
        self.output_nodes = []

        _default_graph.variables.append(self)

    ##endof:  __init__(self, initial_value=None)

##endof:  class Variable
```

### Course Notes Version

In [47]:
```
class VariableCNV():
    '''
    This variable is a changeable parameter of the Graph.
    (Jose said we can think of it as a weight.)
    '''

    def __init__(self, initial_value=None):

        self.value = initial_value
        self.output_nodes = []

        _default_graph.variables.append(self)

    ##endof:  __init__(self, initial_value=None)

##endof:  VariableCNV()
```

# Graph

### Lecture Version - with (maybe) Dave's additions

In [48]:
```
class Graph():

    def __init__(self):

        self.operations = []
        self.placeholders = []
        self.variables = []

    ##endof:  __init__(self)

    def set_as_default(self):

        global _default_graph
        _default_graph = self

    ##endof:  set_as_default(self)

##endof:  Graph()
```

### Course Notes Version

In [49]:
```
class GraphCNV():
    '''
    No docstring in the course notes
    '''

    def __init__(self):

        self.operations = []
        self.placeholders = []
        self.variables = []

    ##endof:  def __init__(self)

    def set_as_default(self):
        '''
        Sets this Graph instance as the Global Default Graph
        '''

        global _default_graph
        _default_graph = self

    ##endof:  set_as_default(self)

##endof:  GraphCNV()
```

# A Basic Graph

$$z = Ax + b$$

With $A = 10$ and $b = 1$

$$z = 10x + 1$$

Just need a placeholder for $x$ and then, once $x$ is filled in, we can solve it!

In [50]:
```
g = Graph()
```

In [51]:
```
g.set_as_default()
```

In [52]:
```
A = Variable(10)
```

In [53]:
```
b = Variable(1)
```

In [54]:
```
# Jose comments, "Will be filled out later"
x = Placeholder()
```

In [55]:
```
y = Multiply(A, x)
```

```
----------------------------------

 Initializing a  Multiply  operation



--------------------------------------------------
In __init__


 Now,   self.input_nodes =
     [<__main__.Variable object at 0x000002C8DBA97FD0>, <__main__.Placeholder object at 0x000002C8DBAA8198>]


 Current node is:
     node = <__main__.Variable object at 0x000002C8DBA97FD0>


 After assignment, node.output_nodes.append(self)
     node = <__main__.Variable object at 0x000002C8DBA97FD0>

--------------------------------------------------


 Current node is:
     node = <__main__.Placeholder object at 0x000002C8DBAA8198>


 After assignment, node.output_nodes.append(self)
     node = <__main__.Placeholder object at 0x000002C8DBAA8198>

--------------------------------------------------


 Before appending  self  to  _default_graph,
     _default_graph =
               <__main__.Graph object at 0x000002C8DBAA8320>


 After appending  self  to  _default_graph,
     _default_graph =
```

```
<__main__.Graph object at 0x000002C8DBAA8320>
```

In [56]:
```
z = Add(y, b)
```

```
----------------------------------

 Initializing an  Add  operation



------------------------------------------------
In __init__


 Now,   self.input_nodes =
     [<__main__.Multiply object at 0x000002C8DBAAD160>, <__main__.Variable object at 0x000002C8DBAA8F60>]


 Current node is:
     node = <__main__.Multiply object at 0x000002C8DBAAD160>


 After assignment, node.output_nodes.append(self)
     node = <__main__.Multiply object at 0x000002C8DBAAD160>

------------------------------------------------


 Current node is:
     node = <__main__.Variable object at 0x000002C8DBAA8F60>


 After assignment, node.output_nodes.append(self)
     node = <__main__.Variable object at 0x000002C8DBAA8F60>

------------------------------------------------


 Before appending  self  to  _default_graph,
     _default_graph =
               <__main__.Graph object at 0x000002C8DBAA8320>


 After appending  self  to  _default_graph,
     _default_graph =
```

```
<__main__.Graph object at 0x000002C8DBAA8320>
```

### A Comment or 2

Now, we just need to actually compute the $z$. We need to add in 1) a traverse-post-order function, which allows a post order traversal of nodes, which is necessary to make sure the computation is done in the correct order; 2) a Session class, which actually executes this graph.

### The Basic Graph with the Course Notes Version

In [57]:
```
g_CNV = GraphCNV()
```

In [58]:
```
g_CNV.set_as_default()
```

In [59]:
```
A_CNV = VariableCNV(10)
```

In [60]:
```
b_CNV = VariableCNV(1)
```

In [61]:
```
x_CNV = PlaceholderCNV()
```

In [62]:
```
y_CNV = multiplyCNV(A_CNV, x_CNV)
```

In [63]:
```
z_CNV = addCNV(y_CNV, b_CNV)
```

**We got here, and everything computes**, both for my lecture version and the course notes version. When I go through the next lecture, I'll commment out the Course Notes Version. I might come back and do the Course Notes Version. The problem now isn't the same variable names (though I added `'_CNV'` to all of them) - it's the `Graph.set_as_default` function.

DWB

```
1701317785_2023-11-29T211625-0700
```

Actually, I think both would be fine, but I'm not going to spend the extra time doing both.

```
1701394493_2023-11-30T183453-0700
```

# Session

In [64]:
```
import numpy as np
```

Check on graphs, due to error.

In [65]:
```
g
```

Out[65]:
```
<__main__.Graph at 0x2c8dbaa8320>
```

In [66]:
```
g_CNV
```

Out[66]:
```
<__main__.GraphCNV at 0x2c8dbaad1d0>
```

In [67]:
```
g == g_CNV
```

Out[67]:
```
False
```

# Traversing Operation Nodes

### Lecture Version of Classes - with Dave's additions - AND of Running the Session

In [68]:
```
def traverse_postorder(operation,
                       do_show_steps=global_do_show_steps_bool):

    '''
    PostOrder Traversal of Nodes. Basically makes sure computations are
    done in the correct order ( A*x first , then A*x + b ). Feel free
    to copy and paste this code. (DWB 1701792896_2023-12-05T091456-0700,
    nope, typing it out). It is not super important for understanding
    the basic fundamentals of deep learning.
    '''

    nodes_postorder = []
    def recurse(node):
        if do_show_steps:
            dashes = "-"*40
            print("\n" + dashes)
            print("\n Inside  recurse(node)")
            print()
            print("     node = " + str(node))
            print()
        ##endof:  if do_show_steps
        if isinstance(node, Operation):
            if do_show_steps:
                print("\n node, " + str(node))
                print(" is an  Operation")
            ##endof:  if do_show_steps
            for input_node in node.input_nodes:
                if do_show_steps:
                    print("\n Current  input_node = ")
                    print(str(input_node))
                ##endof:  if do_show_steps
                recurse(input_node)
            ##endof:  for input_node in node.input_nodes
        ##endof:  if isinstance(node, Operation)
    ##endof:  recurse(node)

    if do_show_steps:
        dashes = "-"*43
        print("\n\n" + dashes)
        print("\n Calling  recurse(operation)")
        print("\n     with operation = ")
        print(str(operation))
```

```
        print()
    ##endof:  if do_show_steps
    recurse(operation)

    if do_show_steps:
        print("\n\n")
        dashes = "-"*43
        print("\n\n")
        print("Exited the recursion")
        print("\n")
        print(" We now have  nodes_postorder = ")
        print(str(nodes_postorder))
        print()
    ##endof:  if do_show_steps
    return nodes_postorder

##endof traverse_postorder(operation)
```

In [69]:
```
class Session():

    ##  use  operation  and  feed_dict  as these are the names used by
    ##+ TensorFlow.  feed_dict matches placeholders to input values.
    ##+ Later on, we'll feed our network batches of data through that
    ##+ dictionary.
    def run(self, operation, feed_dict={},
            do_show_steps=global_do_show_steps_bool):

        if do_show_steps:
            print("\n\n   !!! Running the Session !!!\n")
        ##endof:  if do_show_steps

        nodes_postorder = traverse_postorder(operation)

        if do_show_steps:
            print("\n After running")
            print("   nodes_postorder = traverse_postorder(operation)")
            print(" we have")
            print(" nodes_postorder = ")
            print(str(nodes_postorder))
            print()
        ##endof:  if do_show_steps

        for node in nodes_postorder:
            if type(node) == Placeholder:
                if do_show_steps:
                    print("\n We have a  Placeholder  and will")
                    print(" assign  feed_dict[node]  to  node.output")
                    print(" ( which which means the value,")
                    print(" feed_dict[node] = " + str(feed_dict[node]))
                    print(" will be assigned.")
                    print()
                ##endof:  if do_show_steps
                node.output = feed_dict[node]
                if do_show_steps:
                    print("\n Checking, node.output = " + str(node.output))
                    print()
                ##endof:  if do_show_steps
            ##endof:  if type(node) == Placeholder
            elif type(node) == Variable:
                if do_show_steps:
```

```
            print("\n We have a  Variable  and will")
            print(" assign  node.value  to  node.output")
            print(" ( which which means the value,")
            print(" node.value = " + str(node.value))
            print(" will be assigned.")
            print()
        ##endof:  if do_show_steps
        node.output = node.value
        if do_show_steps:
            print("\n Checking, node.output = " + str(node.output))
            print()
        ##endof:  if do_show_steps
    ##endof:  elif type(node) == Variable
    # # DWB commenting out the  else  and its assumption
    # else:
    #     # <s>OPERATION</s>
    elif type(node) == Operation:
        if do_show_steps:
            print("\n We have an  Operation  and will")
            print(" compute the output of the operation")
            print(" based on each input_node's output,")
            print(" for each node's input_node-s")
            print(" node.value = " + str(node.value))
            print(" We will assign the result of the")
            print(" computation to  node.output")
            print()
            print(" Some pertinent parts:")
            print(str(node.input_nodes))
            print(" I'm not going to mess around finding")
            print(" the  output  of each  input_node  here,")
            print(" since it will become the  node_inputs")
            print()
        ##endof:  if do_show_steps
        node.inputs = \
          [input_node.output for input_node in node.input_nodes]

        #  For the next command,
        #+   node.output = node.compute((node_inputs))
        #+ asterisk is basically a sort of args asterisk.
        #+ Allows us to combine inputs
        #+ without knowing how many we might have. (Note: each of
        #+ the operations we've made only has two inputs, but it's
        #+ nice to have it generalized, as I'm sure Tensorflow has
```

```
            #+ it generalized. -DWB 1701796074_2023-12-05T100754-0700)

            if do_show_steps:
                print("\n We will now assign the value of")
                print(" node.output")
                print(" We will use")
                #  next line might need
                #+   for nd_inp in *node_inputs: print(nd_inp)
                #+ instead of  str(*node_inputs)
                #+ Nope, seems we're okay
                print(" *node_inputs = " + str(*node_inputs))
                print()
            ##endof:  if do_show_steps

            node.output = node.compute(*node_inputs)

            if do_show_steps:
                print("\n Inspecting, node.output = " + str(node.output))
                print()
            ##endof:  if do_show_steps

        ##endof:  elif type(node) == Operation
        else:
            print()
            print("Session: SOMETHING IS WRONG, AND THINGS WILL PROBABLY BREAK")
            print()
        ##endof:  if/elif/else <type(node)>
    ##endof:  for node in nodes_postorder

    if do_show_steps:
        print("\n\n Looking at a few things, where we are getting")
        print(" errors, as shown in a cell below.")
        print()
        print("operation = " + str(operation))
        print()
        try:
            # <get operation.output>
            print(" We will try to get  operation.output  and print it.")
            op_out = operation.output
            print("operation.output = " + op_out)
            print(" That was a success.")
        except Exception as e1:
            print(" No dice with  operation.output  due to exception, 'e1'")
```

```
            print(str(e1))
            print(" That was a failure.")
        finally:
            print()
            print(" End of trying to get  operation.output")
            print()
        ##endof:  try/except/finally <operation.output>
        try:
            # <get nodes_postorder[0]>
            print("\n Looking at  nodes_postorder = ")
            print(str(nodes_postorder))
            print()
            print(" Looking at nodes_postorder[0], which I hope is an Operation")
            print("     nodes_postorder[0] = " + str(nodes_postorder[0]))
            print()
            print("\n If we got an Operation, let's print its  output")
            print()
            if type(nodes_postorder[0]) == Operation:
                print("\n It is an operation, and")
                print("     nodes_postorder[0].output = " + \
                      str(nodes_postorder[0].output))
                print()
            ##endof:  if/else type(nodes_postorder[0]) == Operation
        except Exception as e2:
            print(" No dice with  nodes_postorder[0]  due to exception, 'e2'")
            print(str(e2))
            print(" That was a failure.")
        finally:
            print()
            print(" End of trying to get  nodes_postorder[0]")
            print()
        ##endof:  try/except/finally <nodes_postorder[0]>

    ##endof:  if do_show_steps

    # ## ORIGINAL CODE, WHICH GAVE AN ERROR
    #output_to_return = operation.output

    # ## NEW CODE TRY 1, WHICH ALSO GAVE AN ERROR
    # output_to_return = nodes_postorder.output

    # Returning to original for the here-is-the-problem commit
    print("\n\n #@#@# I EXPECT IT TO FAIL HERE #@#@#")
```

```
        print()
        print(" It's going to try   output_to_return = operation.output")
        print(" I'll save a commit after whatever happens, then I'll use the")
        print(" Course Notes Version of everything to see if it works. That")
        print(" will be the next commit.")
        print(" I'm thinking that adding a  self.output  to the Operation-s,")
        print(" e.g.  self.output = var_x + var_y  for  Addition")
        print(" Otherwise, I'll have to look at changing the")
        print("   elif type(node) == Operation:")
        print(" back to")
        print("   else:")
        print()
        print(" Or, as I think I see now, I will need to add a")
        print("   nodes_postorder.append(node)  to the  traverse_postorder  method.")
        print()
        print(" DWB 1701799914_2023-12-05T111154-0700")
        print("\n #@#@# EXPECTING FAILURE IMMINENTLY #@#@#")

        output_to_return = operation.output

        if do_show_steps:
            print("\n\n We will return the output of the operation,")
            equals_str = "="*60
            print("  " + equals_str)
            print("output_to_return = operation.output = " + \
                  str(output_to_return))
            print("  " + equals_str)
            print()
            print("\n\n   !!! Finished Running the Session !!!\n")
            print()
        ##endof:  if do_show_steps

        return output_to_return

    ##endof:  run(self, operation, feed_dict={}, do_show_steps=True)
##endof:  Session()
```

In [70]:
```
sess = Session()
```

In [71]:
```
result = sess.run(operation=z, feed_dict={x:10}, do_show_steps=True)
```

```
   !!! Running the Session !!!



----------------------------------------

 Calling  recurse(operation)

     with operation =
<__main__.Add object at 0x000002C8DBAAD0F0>


----------------------------------------

 Inside  recurse(node)

     node = <__main__.Add object at 0x000002C8DBAAD0F0>


 node, <__main__.Add object at 0x000002C8DBAAD0F0>
 is an  Operation

 Current  input_node =
<__main__.Multiply object at 0x000002C8DBAAD160>

----------------------------------------

 Inside  recurse(node)

     node = <__main__.Multiply object at 0x000002C8DBAAD160>


 node, <__main__.Multiply object at 0x000002C8DBAAD160>
 is an  Operation

 Current  input_node =
<__main__.Variable object at 0x000002C8DBA97FD0>

----------------------------------------

 Inside  recurse(node)
```

```
     node = <__main__.Variable object at 0x000002C8DBA97FD0>


 Current  input_node =
<__main__.Placeholder object at 0x000002C8DBAA8198>

----------------------------------------

 Inside  recurse(node)

     node = <__main__.Placeholder object at 0x000002C8DBAA8198>


 Current  input_node =
<__main__.Variable object at 0x000002C8DBAA8F60>

----------------------------------------

 Inside  recurse(node)

     node = <__main__.Variable object at 0x000002C8DBAA8F60>







Exited the recursion


 We now have  nodes_postorder =
[]


 After running
   nodes_postorder = traverse_postorder(operation)
 we have
 nodes_postorder =
[]
```

```
 Looking at a few things, where we are getting
 errors, as shown in a cell below.

operation = <__main__.Add object at 0x000002C8DBAAD0F0>

 We will try to get  operation.output  and print it.
 No dice with  operation.output  due to exception, 'e1'
'Add' object has no attribute 'output'
 That was a failure.

 End of trying to get  operation.output


 Looking at  nodes_postorder =
[]

 Looking at nodes_postorder[0], which I hope is an Operation
 No dice with  nodes_postorder[0]  due to exception, 'e2'
list index out of range
 That was a failure.

 End of trying to get  nodes_postorder[0]



 #@#@# I EXPECT IT TO FAIL HERE #@#@#

 It's going to try   output_to_return = operation.output
 I'll save a commit after whatever happens, then I'll use the
 Course Notes Version of everything to see if it works. That
 will be the next commit.
 I'm thinking that adding a  self.output  to the Operation-s,
 e.g.  self.output = var_x + var_y  for  Addition
 Otherwise, I'll have to look at changing the
   elif type(node) == Operation:
 back to
   else:

 Or, as I think I see now, I will need to add a
   nodes_postorder.append(node)  to the  traverse_postorder  method.

 DWB 1701799914_2023-12-05T111154-0700
```

```
 #@#@# EXPECTING FAILURE IMMINENTLY #@#@#

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-71-180f5a3ae826> in <module>()
----> 1 result = sess.run(operation=z, feed_dict={x:10}, do_show_steps=True)

<ipython-input-69-de65ae5e9d16> in run(self, operation, feed_dict, do_show_steps)
    186         print("\n #@#@# EXPECTING FAILURE IMMINENTLY #@#@#")
    187
--> 188         output_to_return = operation.output
    189
    190         if do_show_steps:

AttributeError: 'Add' object has no attribute 'output'
```

With the lecture's code and my additions (I could have copied some of the lecture's code wrong, and my additions could have messed things up, but I don't think so), I got the error,

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-87-180f5a3ae826> in <module>()
----> 1 result = sess.run(operation=z, feed_dict={x:10}, do_show_steps=True)

<ipython-input-85-7b200a5c2a69> in run(self, operation, feed_dict, do_show_steps)
    169         print("\n #@#@# EXPECTING FAILURE IMMINENTLY #@#@#")
    170
--> 171         output_to_return = operation.output
    172
    173         if do_show_steps:

AttributeError: 'Add' object has no attribute 'output'
```

Looks like I was wrong. It seems to me that I missed a

`nodes_postorder.append(node)`

in the

`traverse_postorder`

method. We'll try the CNV (Course Notes Version), first.

DWB 1701802137_2023-12-05T114857-0700

**Course Notes Version of Classes AND of Running the Session**

In [72]:
```
def traverse_postorder_CNV(operation):
    """
    PostOrder Traversal of Nodes. Basically makes sure computations are done in
    the correct order (Ax first , then Ax + b). Feel free to copy and paste this code.
    It is not super important for understanding the basic fundamentals of deep learning.
    """

    nodes_postorder = []
    def recurse(node):
        if isinstance(node, OperationCNV):
            for input_node in node.input_nodes:
                recurse(input_node)
        nodes_postorder.append(node)

    recurse(operation)
    return nodes_postorder

##endof traverse_postorder_CNV(operation)

##+ I just copy/pasted it and added the '_CNV'
##+ or "CNV" where necessary
##+ DWB 1701800683_2023-12-05T112443-0700
```

In [76]:
```
class SessionCNV():

    def run(self, operation, feed_dict = {}):
        """
          operation: The operation to compute
          feed_dict: Dictionary mapping placeholders to input values (the data)
        """

        # Puts nodes in correct order
        nodes_postorder = traverse_postorder_CNV(operation)

        for node in nodes_postorder:

            if type(node) == PlaceholderCNV:

                node.output = feed_dict[node]

            elif type(node) == VariableCNV:

                node.output = node.value

            else: # Operation

                node.inputs = [input_node.output for input_node in node.input_nodes]


                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


##endof:  class SessionCNV()

##+ I just copy/pasted it and added the '_CNV'
##+ or "CNV" where necessary
##+ DWB 1701800683_2023-12-05T112443-0700
```

In [77]:
```
sess_CNV = SessionCNV()
```

In [79]:
```
result_CNV = sess_CNV.run(operation=z_CNV,
                          feed_dict={x_CNV:10})
```

In [81]:
```
result_CNV
```

Out[81]: 101

That worked. I've put in notes as to what my problem was (missing an `append` operation in the postorder traversal). I'll save this one as going back over my tracks and keeping a good trail. I probably shouldn't do that (Q&R), but I will use Q&R as I continue forward.

## Let's try it!

### Lecture Version of Let's Try It

In [ ]:
```

```

### Course Notes Version of Let's Try It

In [ ]:
```

```

*Hooray!*

## Now, some matrix multiplication

### Lecture Version of matrix multiplication

In [ ]:

### Course Notes Version of matrix multiplication

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

# Activation Function

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

### Sigmoid as an Operation

### Lecture Version - with Dave's additions

In [ ]:

### Course Notes Version

In [ ]:
```
class Sigmoid(Operation):

    def __init__(self, z):

    ##endof:  __init__(self, z)

    def compute(self, z_val):

    ##endof:  compute(self, z_val)

##endof:  Sigmoid (Operation)
```

# Classification Example

## Lecture Version

In [ ]:

In [ ]:

## Course Notes Verison

In [ ]:

In [ ]:

In [ ]:

In [ ]:

# Defining the Perceptron

# Defining the Perceptron

$$y = mx + b$$
$$y = -x + 5$$
$$f_1 = m\, f_2 + b\,,\ m = 1$$
$$f_1 = -f_2 + 5$$
$$f_1 + f_2 - 5 = 0$$

Jose uses $f_{whatever}$ for a feauture

## Convert to a Matrix Representation of Features

blah! Strong Bad. blah!

## Example Point

and blah! again.

In [ ]:

something else

In [ ]:

## Using an Example Session Graph

### Lecture Version

In [ ]:

In [ ]:

### Course Notes Verison

In [ ]:

In [ ]:

In [ ]:

In [ ]:

*That's all for now, folks!*