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!

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

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

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



##endof:  SimpleClassLecture0
```

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

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

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

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

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

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

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

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

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

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

##endof:  SimpleClassLecture1
```

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

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

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

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

In [ ]:
```
##  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
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
```

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

    def __init__(self):

        print("EXTEND!")

    ##endof:  __init__(self)

##endof:  ExtendedClassLecture0(SimpleClassLecture1)
```

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

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

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

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

    def __init__(self):

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

##endof:  ExtendedClassLecture(SimpleClassLecture)
```

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

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

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

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

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

    def __init__(self):

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

    ##endof:  __init__(self)

##endof:  ExtendedClassLecture(SimpleClassLecture)
```

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

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

From the class material

In [ ]:
```
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 [ ]:
```
sc = SimpleClass() # will throw an error
```

OUTPUT:

```
---------------------------------------------------------------------------
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 [ ]:
```
##  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")
```

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

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

##endof:  ExtendedClassNoSuper
```

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

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 [ ]:
```
#  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 [ ]:
```
s = ExtendedClass()
```

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

Operation

Lecture Version - with Dave's additions

In [1]:
```
class Operation():

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

        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
```

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

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

##endof:  Operation
```

Course Notes Version

In [2]:
```
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 [3]:
```
class Add(Operation):

    def __init__(self, x, y, do_show_steps=True):
        if do_show_steps:
            dashes = "-"*35
            print(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 [4]:
```
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 [5]:
```
class Multiply(Operation):

    def __init__(self, x, y, do_show_steps=True):
        if do_show_steps:
            dashes = "-"*35
            print(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 [6]:
```
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 [7]:
```
class MatMul(Operation):

    def __init__(self, x, y, do_show_steps=True):
        if do_show_steps:
            dashes = "-"*35
            print(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 [8]:
```
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 [9]:
```
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 [10]:
```
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 [11]:
```
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 [12]:
```
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 [13]:
```
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 [14]:
```
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 [15]:
```
g = Graph()
```

In [16]:
```
g.set_as_default()
```

In [17]:
```
A = Variable(10)
```

In [18]:
```
b = Variable(1)
```

In [19]:
```
# Jose says, "Will be filled out later"
x = Placeholder()
```

In [20]:
```
y = Multiply(A, x)
```

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

 Initializing a  Multiply  operation



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


 Now,   self.input_nodes =
     [<__main__.Variable object at 0x000001C66DF97080>, <__main__.Placeholder object at 0x000001C66DF97518>]


 Current node is:
     node = <__main__.Variable object at 0x000001C66DF97080>


 After assignment, node.output_nodes.append(self)
     node = <__main__.Variable object at 0x000001C66DF97080>

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


 Current node is:
     node = <__main__.Placeholder object at 0x000001C66DF97518>


 After assignment, node.output_nodes.append(self)
     node = <__main__.Placeholder object at 0x000001C66DF97518>

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


 Before appending  self  to  _default_graph,
     _default_graph =
               <__main__.Graph object at 0x000001C66DF972E8>


 After appending  self  to  _default_graph,
     _default_graph =
```

```
<__main__.Graph object at 0x000001C66DF972E8>
```

In [21]:
```
z = Add(y, b)
```

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

 Initializing an  Add  operation



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


 Now,   self.input_nodes = 
     [<__main__.Multiply object at 0x000001C66DF97160>, <__main__.Variable object at 0x000001C66DF97320>]


 Current node is:
     node = <__main__.Multiply object at 0x000001C66DF97160>


 After assignment, node.output_nodes.append(self)
     node = <__main__.Multiply object at 0x000001C66DF97160>

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


 Current node is:
     node = <__main__.Variable object at 0x000001C66DF97320>


 After assignment, node.output_nodes.append(self)
     node = <__main__.Variable object at 0x000001C66DF97320>

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


 Before appending  self  to  _default_graph,
     _default_graph = 
               <__main__.Graph object at 0x000001C66DF972E8>


 After appending  self  to  _default_graph,
     _default_graph = 
```

```
<__main__.Graph object at 0x000001C66DF972E8>
```

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 [22]:
```
g_CNV = GraphCNV()
```

In [23]:
```
g_CNV.set_as_default()
```

In [24]:
```
A_CNV = VariableCNV(10)
```

In [25]:
```
b_CNV = VariableCNV(1)
```

In [26]:
```
x_CNV = PlaceholderCNV()
```

In [27]:
```
y_CNV = multiplyCNV(A_CNV, x_CNV)
```

In [28]:
```
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`

Session

Lecture Version - with Dave's additions

In [ ]:

Course Notes Version

In [ ]:

Traversing Operation Nodes

In [ ]:

In [ ]:

Let's try it!

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Now, some matrix multiplication

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 [ ]:

Classification Example

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Defining the Perceptron

blah!

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

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

```
In [ ]:
```

That's all for now, folks!

In [ ]:

```

```