

# 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 0x20d42cf24a8>
```

```
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):

        self.do_show_steps = do_show_steps

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

        self.inputs = [x_var, y_var]

        if self.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 self.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):

        self.do_show_steps = do_show_steps

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

        self.inputs = [x_var, y_var]

        if self.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 self.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):

        self.do_show_steps = do_show_steps

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

        self.inputs = [x_var, y_var]

        if self.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 self.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 0x0000020D42D0C320>, <\_\_main\_\_.Placeholder object at 0x0000020D42D13080>]

Current node is:  
node = <\_\_main\_\_.Variable object at 0x0000020D42D0C320>

After assignment, node.output\_nodes.append(self)  
node = <\_\_main\_\_.Variable object at 0x0000020D42D0C320>

-----

Current node is:  
node = <\_\_main\_\_.Placeholder object at 0x0000020D42D13080>

After assignment, node.output\_nodes.append(self)  
node = <\_\_main\_\_.Placeholder object at 0x0000020D42D13080>

-----

Before appending self to \_default\_graph,  
\_default\_graph =  
    <\_\_main\_\_.Graph object at 0x0000020D42D138D0>

After appending self to \_default\_graph,  
\_default\_graph =

```
<__main__.Graph object at 0x0000020D42D138D0>
```

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

-----  
Initializing an Add operation

-----  
In \_\_init\_\_

Now, self.input\_nodes =  
[<\_\_main\_\_.Multiply object at 0x0000020D42D1B048>, <\_\_main\_\_.Variable object at 0x0000020D42D0CCF8>]

Current node is:  
node = <\_\_main\_\_.Multiply object at 0x0000020D42D1B048>

After assignment, node.output\_nodes.append(self)  
node = <\_\_main\_\_.Multiply object at 0x0000020D42D1B048>

-----  
Current node is:  
node = <\_\_main\_\_.Variable object at 0x0000020D42D0CCF8>

After assignment, node.output\_nodes.append(self)  
node = <\_\_main\_\_.Variable object at 0x0000020D42D0CCF8>

-----  
Before appending self to \_default\_graph,  
\_default\_graph =  
    <\_\_main\_\_.Graph object at 0x0000020D42D138D0>

After appending self to \_default\_graph,  
\_default\_graph =

<\_\_main\_\_.Graph object at 0x0000020D42D138D0>

## 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 comment 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 0x20d42d138d0>
```

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

```
Out[66]: <__main__.GraphCNV at 0x20d42d1b0b8>
```

```
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)

        nodes_postorder.append(node)

    ##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])) #"<probably not yet ready>")
                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)) #"<probably not yet ready>")
                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>
## ## DWB, my first attempt here was off.
## elif type(node) == Operation:
elif isinstance(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_nodes")
        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 isinstance(node, Operation)
    else:
        print()
        print("Session: SOMETHING IS WRONG, AND THINGS WILL PROBABLY BREAK")
        print()
    ##endof: if/elif/else <type(node)>

    # Get things numpy-y
    if type(node.output) == list:
        node.output = np.array(node.output)
    ##endof: if type(node.output) == list
##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()
        op_out = operation.output
        print("operation.output = " + str(op_out))

```

```

print()
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
##endof:  if do_show_steps

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=global_do_show_steps_bool)
```



!!! Running the Session !!!

-----

Calling recurse(operation)

with operation =  
<\_\_main\_\_.Add object at 0x0000020D42D1B1D0>

-----

Inside recurse(node)

node = <\_\_main\_\_.Add object at 0x0000020D42D1B1D0>

node, <\_\_main\_\_.Add object at 0x0000020D42D1B1D0>  
is an Operation

Current input\_node =  
<\_\_main\_\_.Multiply object at 0x0000020D42D1B048>

-----

Inside recurse(node)

node = <\_\_main\_\_.Multiply object at 0x0000020D42D1B048>

node, <\_\_main\_\_.Multiply object at 0x0000020D42D1B048>  
is an Operation

Current input\_node =  
<\_\_main\_\_.Variable object at 0x0000020D42D0C320>

-----

Inside recurse(node)

```
node = <__main__.Variable object at 0x0000020D42D0C320>
```

```
Current input_node =
<__main__.Placeholder object at 0x0000020D42D13080>
```

```
-----
```

```
Inside recurse(node)
```

```
node = <__main__.Placeholder object at 0x0000020D42D13080>
```

```
Current input_node =
<__main__.Variable object at 0x0000020D42D0CCF8>
```

```
-----
```

```
Inside recurse(node)
```

```
node = <__main__.Variable object at 0x0000020D42D0CCF8>
```

Exited the recursion

```
We now have nodes_postorder =
[<__main__.Variable object at 0x0000020D42D0C320>, <__main__.Placeholder object at 0x0000020D42D13080>, <__ma
in__.Multiply object at 0x0000020D42D1B048>, <__main__.Variable object at 0x0000020D42D0CCF8>, <__main__.Add
object at 0x0000020D42D1B1D0>]
```

After running

```
nodes_postorder = traverse_postorder(operation)
we have
nodes_postorder =
[<__main__.Variable object at 0x0000020D42D0C320>, <__main__.Placeholder object at 0x0000020D42D13080>, <__ma
in__.Multiply object at 0x0000020D42D1B048>, <__main__.Variable object at 0x0000020D42D0CCF8>, <__main__.Add
```

object at 0x0000020D42D1B1D0>]

We have a Variable and will  
assign node.value to node.output  
( which which means the value,  
node.value = 10  
will be assigned.

Checking, node.output = 10

We have a Placeholder and will  
assign feed\_dict[node] to node.output  
( which which means the value,  
feed\_dict[node] = 10  
will be assigned.

Checking, node.output = 10

We have an Operation and will  
compute the output of the operation  
based on each input\_node's output,  
for each node's input\_nodes  
We will assign the result of the  
computation to node.output

Some pertinent parts:

[<\_\_main\_\_.Variable object at 0x0000020D42D0C320>, <\_\_main\_\_.Placeholder object at 0x0000020D42D13080>]  
I'm not going to mess around finding  
the output of each input\_node here,  
since it will become the node.inputs

We will now assign the value of  
node.output  
We will use  
node.inputs = [10, 10]

Now, computing the Multiply operation

```
Now, self.inputs =
    [10, 10]
```

```
We will return
    result_of_multiply = 100
```

-----

```
Inspecting, node.output = 100
```

```
We have a Variable and will
assign node.value to node.output
( which which means the value,
node.value = 1
will be assigned.
```

```
Checking, node.output = 1
```

```
We have an Operation and will
compute the output of the operation
based on each input_node's output,
for each node's input_nodes
We will assign the result of the
computation to node.output
```

Some pertinent parts:

```
[<__main__.Multiply object at 0x0000020D42D1B048>, <__main__.Variable object at 0x0000020D42D0CCF8>]
I'm not going to mess around finding
the output of each input_node here,
since it will become the node.inputs
```

```
We will now assign the value of
node.output
We will use
node.inputs = [100, 1]
```

Now, computing the Add operation

```
Now, self.inputs =
    [100, 1]
```

```
We will return
    result_of_add = 101
```

```
-----
```

```
Inspecting, node.output = 101
```

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

```
operation = <__main__.Add object at 0x0000020D42D1B1D0>
```

```
operation.output = 101
```

```
Looking at nodes_postorder =
[<__main__.Variable object at 0x0000020D42D0C320>, <__main__.Placeholder object at 0x0000020D42D13080>, <__main__.Multiply object at 0x0000020D42D1B048>, <__main__.Variable object at 0x0000020D42D0CCF8>, <__main__.Add object at 0x0000020D42D1B1D0>]
```

```
Looking at nodes_postorder[0], which I hope is an Operation
    nodes_postorder[0] = <__main__.Variable object at 0x0000020D42D0C320>
```

If we got an Operation, let's print its output

```
We will return the output of the operation,
=====
output_to_return = operation.output = 101
=====
```

!!! Finished Running the Session !!!

### 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 [73]: 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 [74]: sess_CNV = SessionCNV()
```

```
In [75]: result_CNV = sess_CNV.run(operation=z_CNV,  
                                     feed_dict={x_CNV:10})
```

```
In [76]: result_CNV
```

```
Out[76]: 101
```

That worked. We also fixed the bug in my version. On we go! Time for Q&R as I continue forward.

## Now, some matrix multiplication

### Lecture Version of matrix multiplication



```
In [77]: g_mat = Graph()

g_mat.set_as_default()

A_mat = Variable([[10, 20],[30, 40]])
b_mat = Variable([1, 2])

x_mat = Placeholder()

y_mat = MatMul(A_mat, x_mat)

z_mat = Add(y_mat, b_mat)
```

-----

Initializing a MatMul operation

-----

In `__init__`

Now, `self.input_nodes =`  
`[<__main__.Variable object at 0x0000020D43077208>, <__main__.Placeholder object at 0x0000020D43077470>]`

Current node is:  
`node = <__main__.Variable object at 0x0000020D43077208>`

After assignment, `node.output_nodes.append(self)`  
`node = <__main__.Variable object at 0x0000020D43077208>`

-----

Current node is:  
`node = <__main__.Placeholder object at 0x0000020D43077470>`

After assignment, `node.output_nodes.append(self)`  
`node = <__main__.Placeholder object at 0x0000020D43077470>`

-----

Before appending `self` to `_default_graph`,  
`_default_graph =`  
`<__main__.Graph object at 0x0000020D42D96978>`

After appending `self` to `_default_graph`,  
`_default_graph =`  
`<__main__.Graph object at 0x0000020D42D96978>`

-----  
Initializing an Add operation

-----  
In \_\_init\_\_

Now, self.input\_nodes =  
[<\_\_main\_\_.MatMul object at 0x0000020D430775C0>, <\_\_main\_\_.Variable object at 0x0000020D42D96A58>]

Current node is:  
node = <\_\_main\_\_.MatMul object at 0x0000020D430775C0>

After assignment, node.output\_nodes.append(self)  
node = <\_\_main\_\_.MatMul object at 0x0000020D430775C0>

-----  
Current node is:  
node = <\_\_main\_\_.Variable object at 0x0000020D42D96A58>

After assignment, node.output\_nodes.append(self)  
node = <\_\_main\_\_.Variable object at 0x0000020D42D96A58>

-----  
Before appending self to \_default\_graph,  
\_default\_graph =  
    <\_\_main\_\_.Graph object at 0x0000020D42D96978>

After appending self to \_default\_graph,  
\_default\_graph =

<\_\_main\_\_.Graph object at 0x0000020D42D96978>

In [78]: sess\_mat = Session()

```
In [79]: sess_mat.run(operation=z_mat, feed_dict={x_mat:10})
```

!!! Running the Session !!!

-----

Calling recurse(operation)

with operation =  
<\_\_main\_\_.Add object at 0x0000020D43077240>

-----

Inside recurse(node)

node = <\_\_main\_\_.Add object at 0x0000020D43077240>

node, <\_\_main\_\_.Add object at 0x0000020D43077240>  
is an Operation

Current input\_node =  
<\_\_main\_\_.MatMul object at 0x0000020D430775C0>

-----

Inside recurse(node)

node = <\_\_main\_\_.MatMul object at 0x0000020D430775C0>

node, <\_\_main\_\_.MatMul object at 0x0000020D430775C0>  
is an Operation

Current input\_node =  
<\_\_main\_\_.Variable object at 0x0000020D43077208>

-----

Inside recurse(node)

```
node = <__main__.Variable object at 0x0000020D43077208>
```

```
Current input_node =
<__main__.Placeholder object at 0x0000020D43077470>
```

```
-----
```

```
Inside recurse(node)
```

```
node = <__main__.Placeholder object at 0x0000020D43077470>
```

```
Current input_node =
<__main__.Variable object at 0x0000020D42D96A58>
```

```
-----
```

```
Inside recurse(node)
```

```
node = <__main__.Variable object at 0x0000020D42D96A58>
```

Exited the recursion

```
We now have nodes_postorder =
[<__main__.Variable object at 0x0000020D43077208>, <__main__.Placeholder object at 0x0000020D43077470>, <__main__.MatMul object at 0x0000020D430775C0>, <__main__.Variable object at 0x0000020D42D96A58>, <__main__.Add object at 0x0000020D43077240>]
```

After running

```
nodes_postorder = traverse_postorder(operation)
we have
nodes_postorder =
[<__main__.Variable object at 0x0000020D43077208>, <__main__.Placeholder object at 0x0000020D43077470>, <__main__.MatMul object at 0x0000020D430775C0>, <__main__.Variable object at 0x0000020D42D96A58>, <__main__.Add object at 0x0000020D43077240>]
```

ject at 0x0000020D43077240>]

We have a Variable and will  
 assign node.value to node.output  
 ( which which means the value,  
 node.value = [[10, 20], [30, 40]]  
 will be assigned.

Checking, node.output = [[10, 20], [30, 40]]

We have a Placeholder and will  
 assign feed\_dict[node] to node.output  
 ( which which means the value,  
 feed\_dict[node] = 10  
 will be assigned.

Checking, node.output = 10

We have an Operation and will  
 compute the output of the operation  
 based on each input\_node's output,  
 for each node's input\_nodes  
 We will assign the result of the  
 computation to node.output

Some pertinent parts:

[<\_\_main\_\_.Variable object at 0x0000020D43077208>, <\_\_main\_\_.Placeholder object at 0x0000020D43077470>]  
 I'm not going to mess around finding  
 the output of each input\_node here,  
 since it will become the node.inputs

We will now assign the value of  
 node.output  
 We will use  
 node.inputs = [array([[10, 20],  
                   [30, 40]]), 10]



Now, computing the MatMul operation

```
Now, self.inputs =
      [array([[10, 20],
              [30, 40]]), 10]
```

```
We will return
      result_of_matmul = [[100 200]
                          [300 400]]
```

```
-----
Inspecting, node.output = [[100 200]
                           [300 400]]
```

```
We have a Variable and will
assign node.value to node.output
( which which means the value,
node.value = [1, 2]
will be assigned.
```

```
Checking, node.output = [1, 2]
```

```
We have an Operation and will
compute the output of the operation
based on each input_node's output,
for each node's input_nodes
We will assign the result of the
computation to node.output
```

Some pertinent parts:

```
[<__main__.MatMul object at 0x0000020D430775C0>, <__main__.Variable object at 0x0000020D42D96A58>]
I'm not going to mess around finding
the output of each input_node here,
since it will become the node.inputs
```

We will now assign the value of  
 node.output  
 We will use  
 node.inputs = [array([[100, 200],  
                   [300, 400]]), array([1, 2])]

Now, computing the Add operation

Now, self.inputs =  
       [array([[100, 200],  
               [300, 400]]), array([1, 2])]

We will return  
       result\_of\_add = [[101 202]  
                       [301 402]]

-----  
 Inspecting, node.output = [[101 202]  
                           [301 402]]

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

operation = <\_\_main\_\_.Add object at 0x0000020D43077240>

operation.output = [[101 202]  
                   [301 402]]

Looking at nodes\_postorder =  
 [<\_\_main\_\_.Variable object at 0x0000020D43077208>, <\_\_main\_\_.Placeholder object at 0x0000020D43077470>, <\_\_main\_\_.MatMul object at 0x0000020D430775C0>, <\_\_main\_\_.Variable object at 0x0000020D42D96A58>, <\_\_main\_\_.Add object at 0x0000020D43077240>]

Looking at nodes\_postorder[0], which I hope is an Operation  
       nodes\_postorder[0] = <\_\_main\_\_.Variable object at 0x0000020D43077208>

If we got an Operation, let's print its output

We will return the output of the operation,

```
=====
output_to_return = operation.output = [[101 202]
[301 402]]
=====
```

!!! Finished Running the Session !!!

```
Out[79]: array([[101, 202],
               [301, 402]])
```

### Course Notes Version of matrix multiplication

```
In [80]: g_mat_CNV = GraphCNV()

g_mat_CNV.set_as_default()

A_mat_CNV = VariableCNV([[10, 20],[30, 40]])
b_mat_CNV = VariableCNV([1, 1])

x_mat_CNV = PlaceholderCNV()

y_mat_CNV = matmulCNV(A_mat_CNV, x_mat_CNV)

z_mat_CNV = addCNV(y_mat_CNV, b_mat_CNV)
```

```
In [81]: sess_mat_CNV = SessionCNV()
```

```
In [82]: result_mat_CNV = sess_mat_CNV.run(operation=z_mat_CNV,
                                             feed_dict={x_mat_CNV:10})
```

```
In [83]: result_mat_CNV
```

```
Out[83]: array([[101, 201],  
               [301, 401]])
```

## Classification

## Activation Function

```
In [84]: import matplotlib.pyplot as plt  
%matplotlib inline
```

### Sigmoid as an Operation

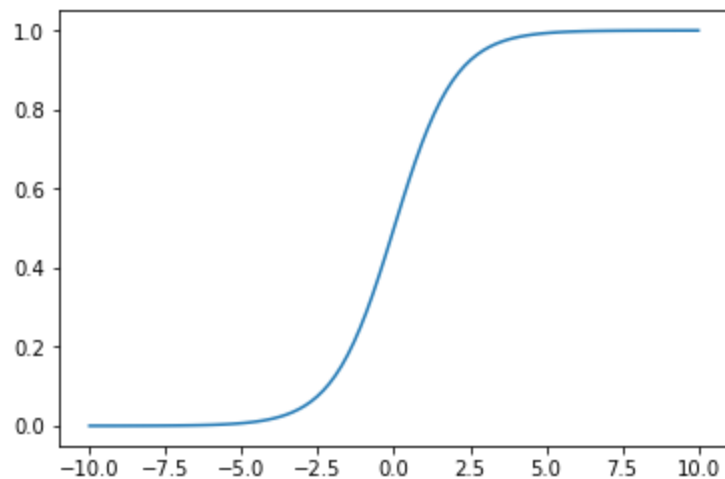
#### Lecture Version - with Dave's additions

```
In [85]: def Sigmoid(z):  
         return 1 / (1 + np.exp(-z))  
##endof: Sigmoid(z)
```

```
In [86]: sample_z = np.linspace(-10, 10, 100)  
sample_a = Sigmoid(sample_z)
```

```
In [87]: plt.plot(sample_z, sample_a)
```

```
Out[87]: [<matplotlib.lines.Line2D at 0x20d42d96ac8>]
```



```
In [88]: class Sigmoid(Operation):  
  
    def __init__(self, z):  
        super().__init__([z]) # need z inside square brackets  
        ##endof: __init__(self, z)  
  
    def compute(self, z_val):  
        return 1 / (1 + np.exp(-z_val))  
        ##endof: compute(self, z)  
  
    ##endof: Sigmoid(Operation)
```

**Course Notes Version**

```
In [89]: class sigmoidCNV(OperationCNV):  
    def __init__(self, z):  
        super().__init__([z])  
    ##endof: __init__(self, z)  
    def compute(self, z_val):  
        return 1 / (1 + np.exp(-z_val))  
    ##endof: compute(self, z_val)  
    ##endof: sigmoidCNV(Operation)
```

## Classification Example

```
In [90]: from sklearn.datasets import make_blobs
```

## Lecture Version

```
In [91]: data = make_blobs(n_samples=50, n_features=2,  
                           centers=2, random_state=75)  
    # centers : how many blobs
```

```
In [92]: data # first array is features, second array is labels
```

```
Out[92]: (array([[ 7.3402781 ,  9.36149154],
 [ 9.13332743,  8.74906102],
 [ 1.99243535, -8.85885722],
 [ 7.38443759,  7.72520389],
 [ 7.97613887,  8.80878209],
 [ 7.76974352,  9.50899462],
 [ 8.3186688 , 10.1026025 ],
 [ 8.79588546,  7.28046702],
 [ 9.81270381,  9.46968531],
 [ 1.57961049, -8.17089971],
 [ 0.06441546, -9.04982817],
 [ 7.2075117 ,  7.04533624],
 [ 9.10704928,  9.0272212 ],
 [ 1.82921897, -9.86956281],
 [ 7.85036314,  7.986659  ],
 [ 3.04605603, -7.50486114],
 [ 1.85582689, -6.74473432],
 [ 2.88603902, -8.85261704],
 [ -1.20046211, -9.55928542],
 [ 2.00890845, -9.78471782],
 [ 7.68945113,  9.01706723],
 [ 6.42356167,  8.33356412],
 [ 8.15467319,  7.87489634],
 [ 1.92000795, -7.50953708],
 [ 1.90073973, -7.24386675],
 [ 7.7605855 ,  7.05124418],
 [ 6.90561582,  9.23493842],
 [ 0.65582768, -9.5920878 ],
 [ 1.41804346, -8.10517372],
 [ 9.65371965,  9.35409538],
 [ 1.23053506, -7.98873571],
 [ 1.96322881, -9.50169117],
 [ 6.11644251,  9.26709393],
 [ 7.70630321, 10.78862346],
 [ 0.79580385, -9.00301023],
 [ 3.13114921, -8.6849493 ],
 [ 1.3970852 , -7.25918415],
 [ 7.27808709,  7.15201886],
 [ 1.06965742, -8.1648251 ],
 [ 6.37298915,  9.77705761],
 [ 7.24898455,  8.85834104],
 [ 2.09335725, -7.66278316],
 [ 1.05865542, -8.43841416],
```



```

[ 6.43807502,  7.85483418],
[ 6.94948313,  8.75248232],
[-0.07326715, -11.69999644],
[ 0.61463602, -9.51908883],
[ 1.31977821, -7.2710667 ],
[ 2.72532584, -7.51956557],
[ 8.20949206, 11.90419283]],
array([1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1,
       1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1,
       1, 0, 0, 0, 0, 1]))

```

```

In [93]: features = data[0]
        labels = data[1]

```

```

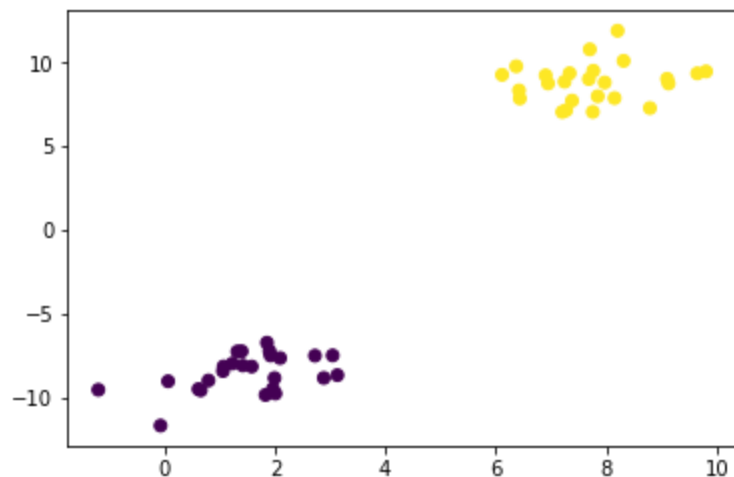
In [94]: plt.scatter(features[:, 0], features[:, 1], c=labels)

```

```

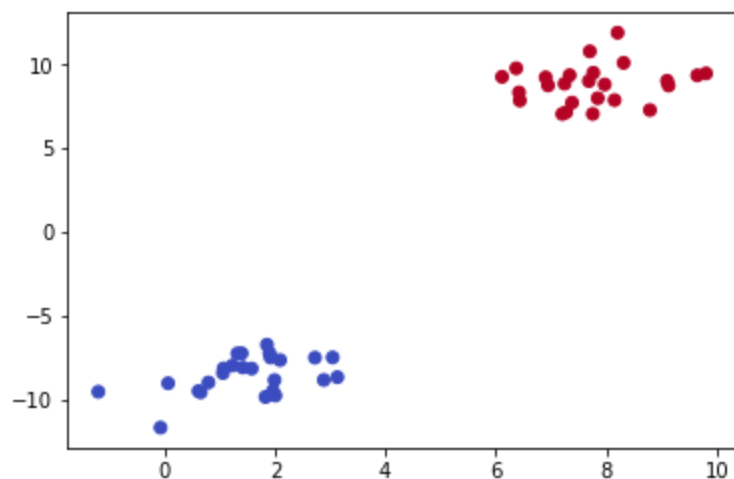
Out[94]: <matplotlib.collections.PathCollection at 0x20d45e6c860>

```



```
In [95]: plt.scatter(features[:, 0], features[:, 1],  
                    c=labels, cmap='coolwarm')
```

```
Out[95]: <matplotlib.collections.PathCollection at 0x20d45ef1c18>
```



## Course Notes Verison

```
In [96]: # Maybe Later  
#  
# # ## Quick later, only need date once. I'm on Windows  
# #!powershell -c (Get-Date -UFormat "%s_%Y%m%dT%H%M%S%Z00") -replace '[.][0-9]*_', '_'
```

Result was: 1704284456\_20240103T122056-0700

```
In [97]: data_CNV = make_blobs(n_samples=50, n_features=2, centers=2, random_state=75)
```

In [98]: data\_CNV

```
Out[98]: (array([[ 7.3402781 ,  9.36149154],
 [ 9.13332743,  8.74906102],
 [ 1.99243535, -8.85885722],
 [ 7.38443759,  7.72520389],
 [ 7.97613887,  8.80878209],
 [ 7.76974352,  9.50899462],
 [ 8.3186688 , 10.1026025 ],
 [ 8.79588546,  7.28046702],
 [ 9.81270381,  9.46968531],
 [ 1.57961049, -8.17089971],
 [ 0.06441546, -9.04982817],
 [ 7.2075117 ,  7.04533624],
 [ 9.10704928,  9.0272212 ],
 [ 1.82921897, -9.86956281],
 [ 7.85036314,  7.986659  ],
 [ 3.04605603, -7.50486114],
 [ 1.85582689, -6.74473432],
 [ 2.88603902, -8.85261704],
 [ -1.20046211, -9.55928542],
 [ 2.00890845, -9.78471782],
 [ 7.68945113,  9.01706723],
 [ 6.42356167,  8.33356412],
 [ 8.15467319,  7.87489634],
 [ 1.92000795, -7.50953708],
 [ 1.90073973, -7.24386675],
 [ 7.7605855 ,  7.05124418],
 [ 6.90561582,  9.23493842],
 [ 0.65582768, -9.5920878 ],
 [ 1.41804346, -8.10517372],
 [ 9.65371965,  9.35409538],
 [ 1.23053506, -7.98873571],
 [ 1.96322881, -9.50169117],
 [ 6.11644251,  9.26709393],
 [ 7.70630321, 10.78862346],
 [ 0.79580385, -9.00301023],
 [ 3.13114921, -8.6849493 ],
 [ 1.3970852 , -7.25918415],
 [ 7.27808709,  7.15201886],
 [ 1.06965742, -8.1648251 ],
 [ 6.37298915,  9.77705761],
 [ 7.24898455,  8.85834104],
 [ 2.09335725, -7.66278316],
 [ 1.05865542, -8.43841416],
```

```

[ 6.43807502,  7.85483418],
[ 6.94948313,  8.75248232],
[ -0.07326715, -11.69999644],
[ 0.61463602, -9.51908883],
[ 1.31977821, -7.2710667 ],
[ 2.72532584, -7.51956557],
[ 8.20949206, 11.90419283]]),
array([1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1,
       1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1,
       1, 0, 0, 0, 0, 1]))

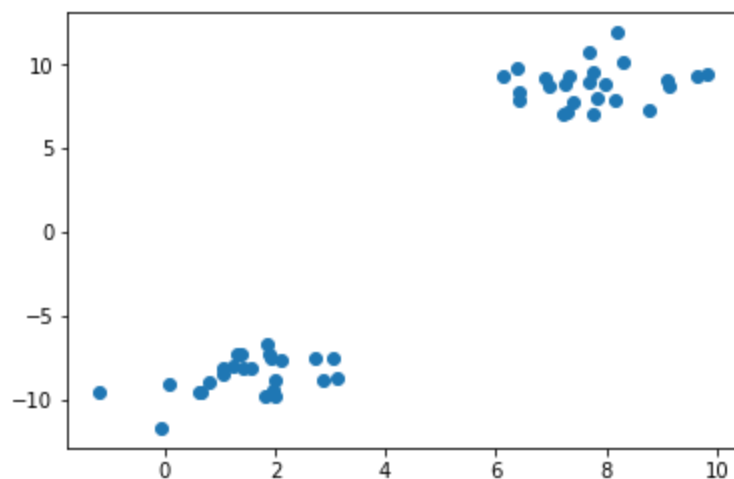
```

```

In [99]: features_CNV = data_CNV[0]
plt.scatter(features_CNV[:, 0], features_CNV[:, 1])
# everything in the zeroth column (feature_x)
# then everything in the first column (feature_y)

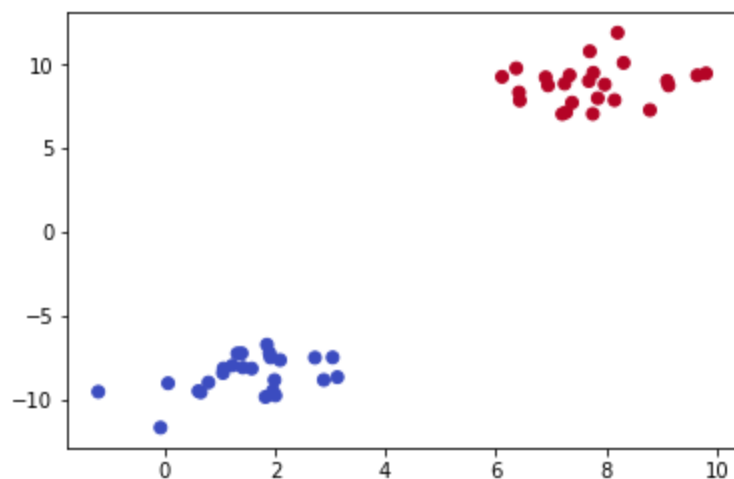
```

Out[99]: <matplotlib.collections.PathCollection at 0x20d45f5a240>



```
In [100]: labels_CNV = data_CNV[1]
plt.scatter(features_CNV[:, 0], features_CNV[:, 1], c=labels_CNV, cmap='coolwarm')
```

```
Out[100]: <matplotlib.collections.PathCollection at 0x20d45fc6d68>
```

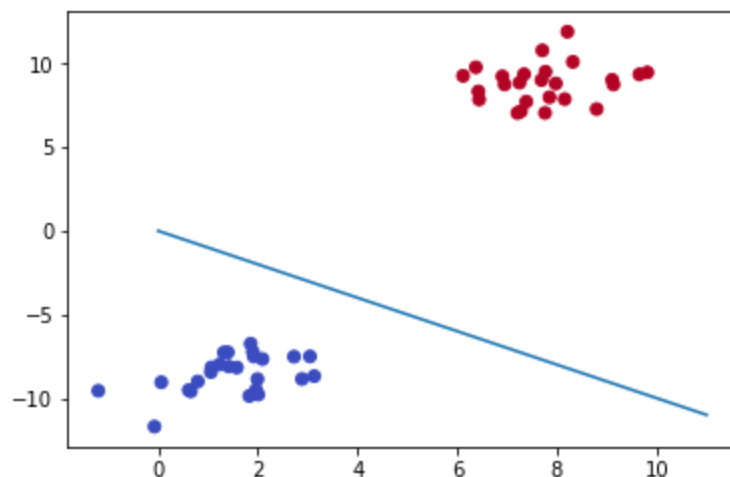


## Try to get a line separating the two blobs

### Lecture Version

```
In [101]: x_line_pre = np.linspace(0, 11, 10)
y_line_pre = -x_line_pre
plt.scatter(features[:, 0], features[:, 1],
            c=labels, cmap='coolwarm')
plt.plot(x_line_pre, y_line_pre)
```

Out[101]: [<matplotlib.lines.Line2D at 0x20d45f95438>]



```
In [102]: # It looks as if it would be better to raise it
y_line = -x_line + 5
plt.scatter(features[:, 0], features[:, 1],
            c=labels, cmap='coolwarm')
plt.plot(x_line, y_line)
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-102-582f03bfae0f> in <module>()
      1 # It looks as if it would be better to raise it
----> 2 y_line = -x_line + 5
      3 plt.scatter(features[:, 0], features[:, 1],
      4             c=labels, cmap='coolwarm')
      5 plt.plot(x_line, y_line)
```

NameError: name 'x\_line' is not defined

Anything above that line belongs to the red class. Anything below the line belongs to the blue class.

Formalized as  $y = mx + b$

Could be restated as  $f_2 = mf_1 + b$  with  $m = -1$  and  $b = 5$

We want a placeholder ready, into which we can put the features.

Now, we're going to do a matrix representation to create a perceptron model and do our classification.

<https://stackoverflow.com/a/59594891/6505499> (<https://stackoverflow.com/a/59594891/6505499>)

<https://web.archive.org/web/20231231183005/https://stackoverflow.com/questions/59098171/how-to-insert-python-variable-into-latex-matrix-in-jupyter-notebook-markdown-cel> (<https://web.archive.org/web/20231231183005/https://stackoverflow.com/questions/59098171/how-to-insert-python-variable-into-latex-matrix-in-jupyter-notebook-markdown-cel>)

<https://www.physicsread.com/latex-column-vector-and-column-matrix/> (<https://www.physicsread.com/latex-column-vector-and-column-matrix/>)

<https://web.archive.org/web/20231231183214/https://www.physicsread.com/latex-column-vector-and-column-matrix/>  
(<https://web.archive.org/web/20231231183214/https://www.physicsread.com/latex-column-vector-and-column-matrix/>)

<https://tex.stackexchange.com/a/367971/188930> (<https://tex.stackexchange.com/a/367971/188930>)

<https://web.archive.org/web/20231231183433/https://tex.stackexchange.com/questions/367960/creating-a-matrix-with-column-or-row-vectors-as-arguments> (<https://web.archive.org/web/20231231183433/https://tex.stackexchange.com/questions/367960/creating-a-matrix-with-column-or-row-vectors-as-arguments>)

## Course Notes Version

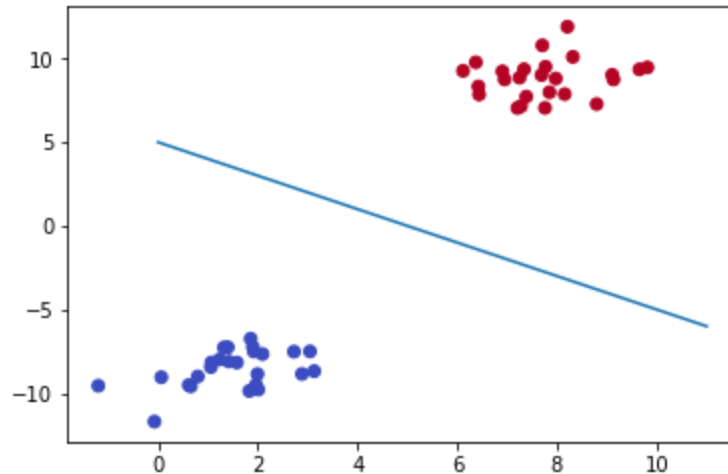
```
In [103]: # Not now - Q&R
          # # Okay, maybe now. Only need date once => comment out
          # !powershell -c (Get-Date -UFormat "%s_%Y%m%dT%H%M%S%Z00") -replace '[.][0-9]*_', '_'
```

Output: 1704285338\_20240103T123538-0700



```
In [104]: # > DRAW A LINE THAT SEPARATES CLASSES
x_line_CNV = np.linspace(0, 11, 10)
y_line_CNV = -x_line_CNV + 5
plt.scatter(features_CNV[:, 0], features_CNV[:, 1], c=labels_CNV, cmap='coolwarm')
plt.plot(x_line_CNV, y_line_CNV)
```

Out[104]: [<matplotlib.lines.Line2D at 0x20d45f758d0>]



## Defining the Perceptron

$$\begin{aligned}
 y &= mx + b \\
 y &= -x + 5 \\
 f_2 &= m f_1 + b, \quad m = 1 \\
 f_2 &= -f_1 + 5 \\
 f_1 + f_2 - 5 &= 0
 \end{aligned}$$

Jose uses  $f_{whatever}$  for a feaature

## Convert to a Matrix Representation of Features

**Dave/Lecture Version**

## TL/DR

$$\begin{bmatrix} 1 & 1 \end{bmatrix} \cdot \overleftrightarrow{f} - 5 = 0$$

## Verbosity Turned Up

For our matrix, continuing the *Defining the Perceptron* cell, we're basically looking at

$$\text{FeatMatrix} \begin{bmatrix} 1, & 1 \end{bmatrix} - 5 = 0$$

or, more nicely than Jose's

$$(1,1) * \mathbf{f} - 5 = 0$$

we can use

$$\begin{bmatrix} 1 & 1 \end{bmatrix} \cdot \text{textbf{f}} - 5 = 0$$

which gives us

$$\begin{bmatrix} 1 & 1 \end{bmatrix} \cdot \mathbf{f} - 5 = 0$$

or, even better,

$$\begin{bmatrix} 1 & 1 \end{bmatrix} \cdot \overleftrightarrow{f} - 5 = 0$$

where

## Matrix Notes

The matrix,  $\overleftrightarrow{f}$  is defined as

$$\overline{\overline{f}} \ ; \ ; \ \overrightarrow{\Delta} = \ ; \ ; \ \begin{bmatrix} f_1 & f_2 \end{bmatrix}$$

$$\overline{f} \triangleq \begin{bmatrix} f_1 \\ f_2 \end{bmatrix}$$

My favorite matrix notation is

$\overset{\rightharpoonup}{f}$

though I think a nice second is the one I just thought to try:

$\overset{=}{f}$

and third place going to the standard textbook form,

$\textbf{f}$

Course Notes Version

(after *Defining the Perceptron*)

Convert to a Matrix Representation of features

$$w^T x + b = 0$$

As Jose had it

$$\text{Big}(1, 1) f - 5 = 0$$

$$(1, 1) f - 5 = 0$$

My preference is

$$\begin{bmatrix} 1 & 1 \end{bmatrix} \cdot \overrightarrow{f} - 5 = 0$$

$$\begin{bmatrix} 1 & 1 \end{bmatrix} \cdot \overrightarrow{f} - 5 = 0$$

Then if the result is  $> 0$  it's label  $= 1$ , if it is less than 0, it is label  $= 0$

## Example Point

### Lecture version

Picking one that will quite obviously be in the red-dot side. We should get a positive number, since it's over the line. Jose went with (8, 10)

```
In [105]: np.array([1, 1]).dot(np.array([[8], [10]])) - 5
```

```
Out[105]: array([13])
```

It's very positive, so it's in the red class.

Now for one in the blue points, which will give us a negative number. Jose went with  $(2, -10)$ .

```
In [106]: np.array([1, 1]).dot(np.array([[2],[ -10]]))
```

```
Out[106]: array([-8])
```

That's negative, so it's in the blue class (the class under the line).

### Course Notes Version, with Dave making things look nicer

Let's say we have the point,  $f_1=2, f_2=2$  (*I think Jose meant to put  $f_1 = 8, f_2 = 10$* ), *otherwise states as*  $(8, 10)$  \$. Then we have

<Jose has>

```
$$
\begin{pmatrix}
1 , 1
\end{pmatrix}
\begin{pmatrix}
8 \\
10
\end{pmatrix} + 5 = $$
```

$$(1, 1) \begin{pmatrix} 8 \\ 10 \end{pmatrix} + 5 =$$

Let's give him a parentheses matrix ( `pmatrix` ) instead of our previously used (square) bracket matrix ( `bmatrix` ), and see if we can make things look nicer (as well as completing the RHS of the equation).

```
$$ \begin{pmatrix} 1 & 1 \end{pmatrix} \begin{pmatrix} 8 \\ 10 \end{pmatrix} + 5 = 0 $$
```

$$\begin{pmatrix} 1 & 1 \end{pmatrix} \begin{pmatrix} 8 \\ 10 \end{pmatrix} + 5 = 0$$

```
In [107]: np.array([[1, 1]]).dot(np.array([[8], [10]])) - 5
```

```
Out[107]: array([[13]])
```

or if we have  $(4, -10)$

```
In [108]: np.array([[1, 1]]).dot(np.array([[4], [-10]])) - 5
```

```
Out[108]: array([[ -11]])
```

# DWB here

Or, let's say we have  $(2, 2)$ , as Jose seemed to write the first time.

```
In [109]: np.array([[1, 1]]).dot(np.array([[2], [2]])) - 5
```

```
Out[109]: array([[ -1]])
```

It's in the  $\theta$  class, below the line, but the algorithm isn't as sure.

## Using an Example Session Graph

### Lecture Version

```
In [110]: g_perc = Graph() # I use "perc" for "perceptron"
```

```
In [111]: g_perc.set_as_default()
```

```
In [112]: x_perc = Placeholder()
```

```
In [113]: w_perc = Variable([1, 1]) # Weights
```

```
In [114]: b_perc = Variable(-5)
```



```
In [115]: # Due to these problems, I was going to put in the course notes version,
          #+ but it's working.

          ## Seems my version has a problem with combining the operations
          ##+ Nope, it was a non-list in the Sigmoid.
          ##+ I got the same decimal value as Jose with this one
          z_perc = Add(MatMul(w_perc, x_perc), b_perc) # Combining two operations!

          # ## This one worked, but gave me 1.0 rather than a -.999and-so-on
          # y_perc = MatMul(x_perc, x_perc)
          # z_perc = Add(y_perc, b_perc)
```

-----

Initializing a MatMul operation

-----

In `__init__`

Now, `self.input_nodes =`  
`[<__main__.Variable object at 0x0000020D46124710>, <__main__.Placeholder object at 0x0000020D46162860>]`

Current node is:  
`node = <__main__.Variable object at 0x0000020D46124710>`

After assignment, `node.output_nodes.append(self)`  
`node = <__main__.Variable object at 0x0000020D46124710>`

-----

Current node is:  
`node = <__main__.Placeholder object at 0x0000020D46162860>`

After assignment, `node.output_nodes.append(self)`  
`node = <__main__.Placeholder object at 0x0000020D46162860>`

-----

Before appending `self` to `_default_graph`,  
`_default_graph =`  
`<__main__.Graph object at 0x0000020D46162048>`

After appending `self` to `_default_graph`,  
`_default_graph =`  
`<__main__.Graph object at 0x0000020D46162048>`

-----

Initializing an Add operation

-----

In \_\_init\_\_

Now, self.input\_nodes =  
[<\_\_main\_\_.MatMul object at 0x0000020D46118F28>, <\_\_main\_\_.Variable object at 0x0000020D46118A90>]

Current node is:  
node = <\_\_main\_\_.MatMul object at 0x0000020D46118F28>

After assignment, node.output\_nodes.append(self)  
node = <\_\_main\_\_.MatMul object at 0x0000020D46118F28>

-----

Current node is:  
node = <\_\_main\_\_.Variable object at 0x0000020D46118A90>

After assignment, node.output\_nodes.append(self)  
node = <\_\_main\_\_.Variable object at 0x0000020D46118A90>

-----

Before appending self to \_default\_graph,  
\_default\_graph =  
    <\_\_main\_\_.Graph object at 0x0000020D46162048>

After appending self to \_default\_graph,  
\_default\_graph =

<\_\_main\_\_.Graph object at 0x0000020D46162048>

```
In [116]: a_perc = Sigmoid(z_perc)
```

-----  
In \_\_init\_\_

Now, self.input\_nodes =  
[<\_\_main\_\_.Add object at 0x0000020D46118BA8>]

Current node is:  
node = <\_\_main\_\_.Add object at 0x0000020D46118BA8>

After assignment, node.output\_nodes.append(self)  
node = <\_\_main\_\_.Add object at 0x0000020D46118BA8>

-----  
Before appending self to \_default\_graph,  
\_default\_graph =  
    <\_\_main\_\_.Graph object at 0x0000020D46162048>

After appending self to \_default\_graph,  
\_default\_graph =  
    <\_\_main\_\_.Graph object at 0x0000020D46162048>

**First session, with the obvious red (above the line, positive).**

```
In [117]: sess_perc = Session()
```

```
In [118]: sess_perc.run(operation=a_perc, feed_dict={x_perc:[8, 10]})
```

!!! Running the Session !!!

-----

Calling recurse(operation)

with operation =  
<\_\_main\_\_.Sigmoid object at 0x0000020D46118438>

-----

Inside recurse(node)

node = <\_\_main\_\_.Sigmoid object at 0x0000020D46118438>

node, <\_\_main\_\_.Sigmoid object at 0x0000020D46118438>  
is an Operation

Current input\_node =  
<\_\_main\_\_.Add object at 0x0000020D46118BA8>

-----

Inside recurse(node)

node = <\_\_main\_\_.Add object at 0x0000020D46118BA8>

node, <\_\_main\_\_.Add object at 0x0000020D46118BA8>  
is an Operation

Current input\_node =  
<\_\_main\_\_.MatMul object at 0x0000020D46118F28>

-----

Inside recurse(node)

```
node = <__main__.MatMul object at 0x0000020D46118F28>
```

```
node, <__main__.MatMul object at 0x0000020D46118F28>  
is an Operation
```

```
Current input_node =  
<__main__.Variable object at 0x0000020D46124710>
```

```
-----
```

```
Inside recurse(node)
```

```
node = <__main__.Variable object at 0x0000020D46124710>
```

```
Current input_node =  
<__main__.Placeholder object at 0x0000020D46162860>
```

```
-----
```

```
Inside recurse(node)
```

```
node = <__main__.Placeholder object at 0x0000020D46162860>
```

```
Current input_node =  
<__main__.Variable object at 0x0000020D46118A90>
```

```
-----
```

```
Inside recurse(node)
```

```
node = <__main__.Variable object at 0x0000020D46118A90>
```

```
Exited the recursion
```

We now have `nodes_postorder =`  
`[<__main__.Variable object at 0x0000020D46124710>, <__main__.Placeholder object at 0x0000020D46162860>, <__main__.MatMul object at 0x0000020D46118F28>, <__main__.Variable object at 0x0000020D46118A90>, <__main__.Add object at 0x0000020D46118BA8>, <__main__.Sigmoid object at 0x0000020D46118438>]`

After running

```
nodes_postorder = traverse_postorder(operation)
```

we have

```
nodes_postorder =
```

```
[<__main__.Variable object at 0x0000020D46124710>, <__main__.Placeholder object at 0x0000020D46162860>, <__main__.MatMul object at 0x0000020D46118F28>, <__main__.Variable object at 0x0000020D46118A90>, <__main__.Add object at 0x0000020D46118BA8>, <__main__.Sigmoid object at 0x0000020D46118438>]
```

We have a `Variable` and will  
 assign `node.value` to `node.output`  
 ( which which means the value,  
`node.value = [1, 1]`  
 will be assigned.

Checking, `node.output = [1, 1]`

We have a `Placeholder` and will  
 assign `feed_dict[node]` to `node.output`  
 ( which which means the value,  
`feed_dict[node] = [8, 10]`  
 will be assigned.

Checking, `node.output = [8, 10]`

We have an `Operation` and will  
 compute the output of the operation  
 based on each `input_node`'s output,  
 for each `node`'s `input_nodes`  
 We will assign the result of the  
 computation to `node.output`



Some pertinent parts:

```
[<__main__.Variable object at 0x0000020D46124710>, <__main__.Placeholder object at 0x0000020D46162860>]
```

I'm not going to mess around finding  
the output of each input\_node here,  
since it will become the node.inputs

We will now assign the value of  
node.output

We will use

```
node.inputs = [array([1, 1]), array([ 8, 10])]
```

Now, computing the MatMul operation

```
Now, self.inputs =  
[array([1, 1]), array([ 8, 10])]
```

We will return

```
result_of_matmul = 18
```

-----

Inspecting, node.output = 18

We have a Variable and will  
assign node.value to node.output  
( which which means the value,  
node.value = -5  
will be assigned.

Checking, node.output = -5

We have an Operation and will  
compute the output of the operation  
based on each input\_node's output,  
for each node's input\_nodes  
We will assign the result of the

computation to `node.output`

Some pertinent parts:

```
[<__main__.MatMul object at 0x0000020D46118F28>, <__main__.Variable object at 0x0000020D46118A90>]
```

I'm not going to mess around finding  
the output of each `input_node` here,  
since it will become the `node.inputs`

We will now assign the value of  
`node.output`

We will use

```
node.inputs = [18, -5]
```

Now, computing the `Add` operation

```
Now, self.inputs =  
[18, -5]
```

```
We will return  
result_of_add = 13
```

-----

Inspecting, `node.output = 13`

We have an `Operation` and will  
compute the output of the operation  
based on each `input_node`'s output,  
for each `node`'s `input_nodes`  
We will assign the result of the  
computation to `node.output`

Some pertinent parts:

```
[<__main__.Add object at 0x0000020D46118BA8>]
```

I'm not going to mess around finding  
the output of each `input_node` here,  
since it will become the `node.inputs`

We will now assign the value of  
 node.output  
 We will use  
 node.inputs = [13]

Inspecting, node.output = 0.999997739675702

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

operation = <\_\_main\_\_.Sigmoid object at 0x0000020D46118438>

operation.output = 0.999997739675702

Looking at nodes\_postorder =  
 [<\_\_main\_\_.Variable object at 0x0000020D46124710>, <\_\_main\_\_.Placeholder object at 0x0000020D46162860>, <\_\_main\_\_.MatMul object at 0x0000020D46118F28>, <\_\_main\_\_.Variable object at 0x0000020D46118A90>, <\_\_main\_\_.Add object at 0x0000020D46118BA8>, <\_\_main\_\_.Sigmoid object at 0x0000020D46118438>]

Looking at nodes\_postorder[0], which I hope is an Operation  
 nodes\_postorder[0] = <\_\_main\_\_.Variable object at 0x0000020D46124710>

If we got an Operation, let's print its output

We will return the output of the operation,  
 =====  
 output\_to\_return = operation.output = 0.999997739675702  
 =====

!!! Finished Running the Session !!!

Out[118]: 0.999997739675702

**Second session, with the obvious blue (below the line, negative).**

```
In [119]: sess_perc_2 = Session()
```

```
In [120]: sess_perc_2.run(operation=a_perc, feed_dict={x_perc:[2, -10]})
```

!!! Running the Session !!!

-----

Calling recurse(operation)

with operation =  
<\_\_main\_\_.Sigmoid object at 0x0000020D46118438>

-----

Inside recurse(node)

node = <\_\_main\_\_.Sigmoid object at 0x0000020D46118438>

node, <\_\_main\_\_.Sigmoid object at 0x0000020D46118438>  
is an Operation

Current input\_node =  
<\_\_main\_\_.Add object at 0x0000020D46118BA8>

-----

Inside recurse(node)

node = <\_\_main\_\_.Add object at 0x0000020D46118BA8>

node, <\_\_main\_\_.Add object at 0x0000020D46118BA8>  
is an Operation

Current input\_node =  
<\_\_main\_\_.MatMul object at 0x0000020D46118F28>

-----

Inside recurse(node)

```
node = <__main__.MatMul object at 0x0000020D46118F28>
```

```
node, <__main__.MatMul object at 0x0000020D46118F28>  
is an Operation
```

```
Current input_node =  
<__main__.Variable object at 0x0000020D46124710>
```

```
-----
```

```
Inside recurse(node)
```

```
node = <__main__.Variable object at 0x0000020D46124710>
```

```
Current input_node =  
<__main__.Placeholder object at 0x0000020D46162860>
```

```
-----
```

```
Inside recurse(node)
```

```
node = <__main__.Placeholder object at 0x0000020D46162860>
```

```
Current input_node =  
<__main__.Variable object at 0x0000020D46118A90>
```

```
-----
```

```
Inside recurse(node)
```

```
node = <__main__.Variable object at 0x0000020D46118A90>
```

```
Exited the recursion
```

We now have `nodes_postorder =`  
`[<__main__.Variable object at 0x0000020D46124710>, <__main__.Placeholder object at 0x0000020D46162860>, <__main__.MatMul object at 0x0000020D46118F28>, <__main__.Variable object at 0x0000020D46118A90>, <__main__.Add object at 0x0000020D46118BA8>, <__main__.Sigmoid object at 0x0000020D46118438>]`

After running

```
nodes_postorder = traverse_postorder(operation)
```

we have

```
nodes_postorder =
```

```
[<__main__.Variable object at 0x0000020D46124710>, <__main__.Placeholder object at 0x0000020D46162860>, <__main__.MatMul object at 0x0000020D46118F28>, <__main__.Variable object at 0x0000020D46118A90>, <__main__.Add object at 0x0000020D46118BA8>, <__main__.Sigmoid object at 0x0000020D46118438>]
```

We have a `Variable` and will  
 assign `node.value` to `node.output`  
 ( which which means the value,  
`node.value = [1, 1]`  
 will be assigned.

Checking, `node.output = [1, 1]`

We have a `Placeholder` and will  
 assign `feed_dict[node]` to `node.output`  
 ( which which means the value,  
`feed_dict[node] = [2, -10]`  
 will be assigned.

Checking, `node.output = [2, -10]`

We have an `Operation` and will  
 compute the output of the operation  
 based on each `input_node`'s output,  
 for each `node`'s `input_nodes`  
 We will assign the result of the  
 computation to `node.output`



Some pertinent parts:

```
[<__main__.Variable object at 0x0000020D46124710>, <__main__.Placeholder object at 0x0000020D46162860>]
```

I'm not going to mess around finding  
the output of each input\_node here,  
since it will become the node.inputs

We will now assign the value of

node.output

We will use

```
node.inputs = [array([1, 1]), array([ 2, -10])]
```

Now, computing the MatMul operation

```
Now, self.inputs =  
[array([1, 1]), array([ 2, -10])]
```

We will return

```
result_of_matmul = -8
```

-----

Inspecting, node.output = -8

We have a Variable and will  
assign node.value to node.output  
( which which means the value,  
node.value = -5  
will be assigned.

Checking, node.output = -5

We have an Operation and will  
compute the output of the operation  
based on each input\_node's output,  
for each node's input\_nodes  
We will assign the result of the

computation to `node.output`

Some pertinent parts:

```
[<__main__.MatMul object at 0x0000020D46118F28>, <__main__.Variable object at 0x0000020D46118A90>]
```

I'm not going to mess around finding  
the output of each `input_node` here,  
since it will become the `node.inputs`

We will now assign the value of  
`node.output`

We will use

```
node.inputs = [-8, -5]
```

Now, computing the `Add` operation

```
Now, self.inputs =  
[-8, -5]
```

We will return

```
result_of_add = -13
```

-----

Inspecting, `node.output = -13`

We have an `Operation` and will  
compute the output of the operation  
based on each `input_node`'s output,  
for each `node`'s `input_nodes`  
We will assign the result of the  
computation to `node.output`

Some pertinent parts:

```
[<__main__.Add object at 0x0000020D46118BA8>]
```

I'm not going to mess around finding  
the output of each `input_node` here,  
since it will become the `node.inputs`

We will now assign the value of  
 node.output  
 We will use  
 node.inputs = [-13]

Inspecting, node.output = 2.2603242979035746e-06

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

operation = <\_\_main\_\_.Sigmoid object at 0x0000020D46118438>

operation.output = 2.2603242979035746e-06

Looking at nodes\_postorder =  
 [<\_\_main\_\_.Variable object at 0x0000020D46124710>, <\_\_main\_\_.Placeholder object at 0x0000020D46162860>, <\_\_main\_\_.MatMul object at 0x0000020D46118F28>, <\_\_main\_\_.Variable object at 0x0000020D46118A90>, <\_\_main\_\_.Add object at 0x0000020D46118BA8>, <\_\_main\_\_.Sigmoid object at 0x0000020D46118438>]

Looking at nodes\_postorder[0], which I hope is an Operation  
 nodes\_postorder[0] = <\_\_main\_\_.Variable object at 0x0000020D46124710>

If we got an Operation, let's print its output

We will return the output of the operation,  
 =====  
 output\_to\_return = operation.output = 2.2603242979035746e-06  
 =====

!!! Finished Running the Session !!!

Out[120]: 2.2603242979035746e-06

## Course Notes Verison

```
In [121]: # Maybe later  
# Doing it like the other ones
```

```
In [122]: g_perc_CNV = GraphCNV()
```

```
In [123]: g_perc_CNV.set_as_default()
```

```
In [124]: x_perc_CNV = PlaceholderCNV()
```

```
In [125]: w_perc_CNV = VariableCNV([1, 1])
```

```
In [126]: b_perc_CNV = VariableCNV(-5)
```

```
In [127]: z_perc_CNV = addCNV(matmulCNV(w_perc_CNV, x_perc_CNV), b_perc_CNV)
```

```
In [128]: a_perc_CNV = sigmoidCNV(z_perc_CNV)
```

```
In [129]: sess_perc_CNV = SessionCNV()
```

```
In [130]: sess_perc_CNV.run(operation=a_perc_CNV, feed_dict={x_perc_CNV:[8, 10]})
```

Out[130]: 0.999997739675702

```
In [131]: sess_perc_CNV.run(operation=a_perc_CNV, feed_dict={x_perc_CNV:[0, -10]})
```

Out[131]: 3.059022269256247e-07

*That's all for now, folks!*