Computation graphs

Intuitive explanation

When you first look at TensorFlow code, it looks like your familiar imperative program:

- familiar operators
    - assignment, addition, multiplication
    - may overload operators =, +, *

Here is some familiar Python

In [4]:

```
a = 0
b = 1
c = a + b

print(c)
```

```
1
```

and the very similar looking TensorFlow

In [5]:

```
a = tf.Variable(0)
b = tf.Variable(1)
c= tf.add(a,b)

print(c)
```

```
tf.Tensor(1, shape=(), dtype=int32)
```

In fact we could have written

```
c = tf.add(a,b)
```

as

```
c = a + b
```

in order to make Tensorflow look more like Python.

It is **not** the same.

Tensorflow distinguishes between

- program declaration/definition
- program evaluation/execution

Program declaration

Although the code *looks* just like ordinary Python, the statements are *defining* a computation, not demanding that the statements be executed immediately.

Program evaluation

You must take *explicit* steps to execute the program, after passing in initial values

- The statements in TensorFlow are not executed immediately (as in an imperative program)
    - they are defining a future computation (the "computation graph")
    - think of it a defining the *body* of a function

- In order to evaluate (i.e., "call") the function ('computation graphs")
    - You must create a "session" in TensorFlow
    - All code must be run within a session
    - The code is evaluated by explicitly asking for something to be "evaluated" or "run"
        - When evaluating/running: you must pass in actual values for the formal parameters (function arguments/place holders) We've swept some subtle but important details under the rug.

Consider the imperative Python program

Raw Tensorflow Notebook from github (https://colab.research.google.com/github/kenperry-public/ML_Fall_2019/blob/master/Raw_TensorFlow.ipynb)

DNN Tensorflow example Notebook from github (https://colab.research.google.com/github/kenperry-public/ML_Fall_2019/blob/master/DNN_TensorFlow_example.ipynb)

Think of the above Tensorflow code as defining a function (we'll call it foo)

In [18]:

```
def foo():
    a = 2
    b = 1
    c= a + b

    return c
    # print(c)
```

The above was *program definition.

As for evaluation of the program: you must explicitly request it:

In [19]:
```
foo()
```

Out[19]:
```
3
```

We can make the distinction between definition and evaluation even more sharply by changing one local variable to a parameter

In [20]:

```
def foo1(a):
    b = 1
    c= a + b

    print(c)
```

In [21]:
```
foo1(2)
```

```
3
```

That is: the definition of `foo1` is partial.

There is an unbound variable `a` that must be provided at evaluation time.

This simple analogy is only partially accurate.

In spite of the fact that `a` is an unbound variable, I can assert that

- the value of `c` is the sum of the values of `a` and `b`, regardless of what those values are

So it is not entirely unreasonable (if you had never seen Python before) to have expected that

```
print(c)
```

would have returned the string `a + b`

That is: you can view the program definition as specifying a *manipulation of symbols* (algebra)

- rather than a manipulation of *values*
    - which has a prerequisite of binding values to every symbol
- the manipulation of symbols holds even though I may not yet know the value of any symbol

TO DO: Modify for TF 2

So the Tensorflow program above will print

```
Tensor("Add_1:0", shape=(), dtype=int32)
```

which is Tensorflow's way as saying that `c` involves addition

- further inspection would show that the two addends are `a` and `b`

Tensorflow program definition is specifying a *symbolic computation*

- I can *describe* the symbol manipulation that each statement is specifying
- This description can be made even before we have bound a value to any symbol

Let's take this one step further: suppose a was the product of two other symbols:

In [22]:
```
d = tf.Variable(10)
e = tf.Variable(10)
```

In [23]:

```
a = tf.math.multiply(d, e)
b = tf.Variable(1)
c= tf.add(a,b)

print(c)
```

```
tf.Tensor(101, shape=(), dtype=int32)
```

Then `print(c)` will still give the same result

- but the inspection of a
  - will reveal that Tensorflow will apply a multiplication
  - with multiplicands d and e

In [24]:
```
print(a)
```

```
tf.Tensor(100, shape=(), dtype=int32)
```

Tensorflow is literally building a graph of information flow

To actually do something, you have to "evaluate" part of the graph

In [11]:

```
init = tf.global_variables_initializer()

with tf.Session() as sess:
  init.run()

  c_value = sess.run(c)
  print("c value:", c_value)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-11-cbd4f701088f> in <module>
----> 1 init = tf.global_variables_initializer()
      2
      3 with tf.Session() as sess:
      4   init.run()
      5

AttributeError: module 'tensorflow' has no attribute 'global_variables_initial
izer'
```

In the imperative program (Python), each line is evaluated immediately after it is executed.

In the declarative program (Tensorflow), it is not evaluated:

- it just creates a dependence between outputs (c) and inputs (a and b).
- When you evaluate c, it recursively evaluates all the things that c depends on.

Hence, you are declaring a graph that is evaluated later.

Computation graph: a node is an expression, not a value

Imagine that a variable has two attributes

- `c.value`: the current "value" of the variable
- `c.expr` - the expression that computes `c`

When we write

```
c = a + b
```

in our familiar imperative programming languages, this really denotes the imperative

```
c.value = a.value + b.value
```

That is, the string `c = a + b` is a *command* to modify the value of `c`.

In a declarative program, the string `c = a + b` defines a *function* that computes `c` from two inputs `a, b`

```
c.expr = lambda a,b: plus(a,b)
```

Thus, it's possible to write the string `c = a + b` even before `a, b` have been initialized because `a, b` are just formal parameters to the function `c.expr`.

In order to *evaluate* `c.expr` (i.e., compute the concrete value `c.value`) we must first evaluate

```
a.expr, b.expr
```

Note that the declarative program distinguishes between *declaring/defining* an expression and *evaluating* it.

More formally, the `eval` operator (which derives a value from a function) applied to `c` results in

```
eval(c.expr) = plus( eval(a.expr),
eval(b.expr) )
```

These in turn might be expressions that depend on other expressions, e.g.,

```
a.expr = lambda d, e: mult(d,e)
```

So the evaluation of the top-level expression `c.expr` involves recursively evaluating all expressions on which `c.expr` depends. Eventually the recursion unwinds to a base case in which the expression involves no further computation

```
d = lambda: d.value
```

As we traverse the code of the declarative program, we are defining more and more functions, and dependencies between functions (i.e., some functions consume the results of other functions as arguments).

This collection of functions is called a *computation tree*. A computation tree is just a collection of functions and dependencies. A node `c` in the tree has *no concrete* value until we request it to be *evaluated*, which involves

- binding concrete values to all leaf nodes of the sub-tree defining `c.expr`
- recursively evaluating the nodes on which `c` depends.

Eager execution

Many people find declarative programming confusing (and perhaps pointless).

As you will see, there is a point (and a very big one. Hint: do you like to write derivatives ?)

TF supports "eager execution" which makes TF look like an imperative language. This is optional in TF v1, and standard in TF v2.

So, when reading other people's code, it's important to observe whether eager execution has been enabled.

TF v2 is not yet standard so most code you will currently see is declarative.

You may stumble at first, but it is very powerful.

Introducing Eager execution (https://developers.googleblog.com/2017/10/eager-execution-imperative-define-by.html?source=post_page----------)

- Because you are not building a graph, the training loop is different
    - more Pythonic
    - no need to
        - instantiate session
        - `eval` or `run` the training step

optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001) for (x, y) in tfe.Iterator(dataset): grads = tfe.implicit_gradients(loss_function)(model, x, y) optimizer.apply_gradients(grads)

In [ ]:
```
print("Done")
```