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 raw 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

- In Tensorflow version 1: You must take *explicit* steps to execute the program, after passing in initial values
- In Tensorflow version 2: no explicit step required (with default *eager execution*)

The statements in TensorFlow

- Are *defining a future computation* (the "computation graph")
- Think of it a defining the *body* of a function

This definition is called a *computation graph*

By way of analogy:

- Your unexecuted Jupyter notebook defines a computation
- In order to evaluate the definition (i.e, "call the function")
    - You must connect the notebook to a run-time (the "kernel")
    - The run-time contains the program "state": the value of variables defined in previously executed cells

Even without a kernel, a cell

- Is like a function: defines a computation
- Whose "parameters" are the values on which the cell's statements depend

We can still speak about the cell as *defining* a function, even without executing it.

A running kernel "passes" parameters to this function when you execute the cell.

In Tensorflow

- A statement defines part of a computation graph (like a cell in a notebook)
- A *session* must be created to embody the execution state (like the kernel in a notebook)
- All statements must be evaluated in the same session in order for your sequence of cells to execute like an imperative program

In Tensorflow 1: *you* explicitly created the session (i.e., started the notebook kernel) In Tensorflow 2: a session is automatically created for you

- all statements are evaluated immediately in this session
- This is called *eager execution*

Eager execution was *optional* in Tensorflow 1, and is the *default* in Tensorflow 2.

We've swept some subtle but important details under the rug.

Consider the statement

```
c = a + b
```

Whether this appears in the cell of a notebook or as a statement in Tensorflow

- This statement implicitly defines a function (we'll call it `foo`)

In [6]:

```
def foo(a,b):
    c= a + b

    return c
```

This function is well-defined even before we execute it.

To turn the definition into a value

- The kernel/session needs to call the function, passing in values for the paramters

```
foo(a=0, b=1)
```

Although this seems abstract, the real point is that you can view the statement

- As specifying the *manipulation of symbols (algebra)*
- Rather than evaluating values
- And, most importantly:
- Because this is just algebra: we can easily obtain **analytic derivatives**
    - Of the statement
    - With respect to its parameters

Derivatives are the fuel that powers Gradient Descent, the engine of Deep Learning training.

That is why viewing your statements/notebook cells as symbolic manipulation is important.

In Tensorflow, your code is defining a computation graph *along with* the derivatives.

- Note that many statements depend on (their functions are parameterized by) the weights
- So having the derivative of each statement with respect to the weights
- Enables having the derivative of the entire program with respect to the weights

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

The following is a more Computer Science oriented version of the intuitive explanation.

Feel free to gloss over it if the intuitive explanation suffices for you.

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.

In [7]:
```
print("Done")
```

```
Done
```