In [1]:

```
import tensorflow as tf
import numpy as np

import tensorflow as tf

print("Running TensorFlow version ",tf.__version__)
```

```
Running TensorFlow version  2.6.0
```

Reference (https://www.tensorflow.org/guide/intro_to_graphs)

Eager versus Graph computation

If your sole experience with TensorFlow is with versions 2 or greater, you probably take the notion of *eager execution* for granted

- each statement executes immediately, returning a result

But much more is occurring under the covers.

The "native" TensorFlow operates on *graphs*

- representations of statements/programs
- think of a graph as the analog of *compiled code*
- we can define a computation (e.g., graph) without requiring values for each of its inputs
    - think about it a a cell in your Jupyter notebook
        - the computation is well-defined
        - but can't be evaluated/executed until values are bound to each variable referenced in the cell
        - like the body of a function
            - defines a computation
            - but can't be evaluated to actual values are bound to the formal parameters

Before your statement is evaluated, it is turned into a graph ("compiled").

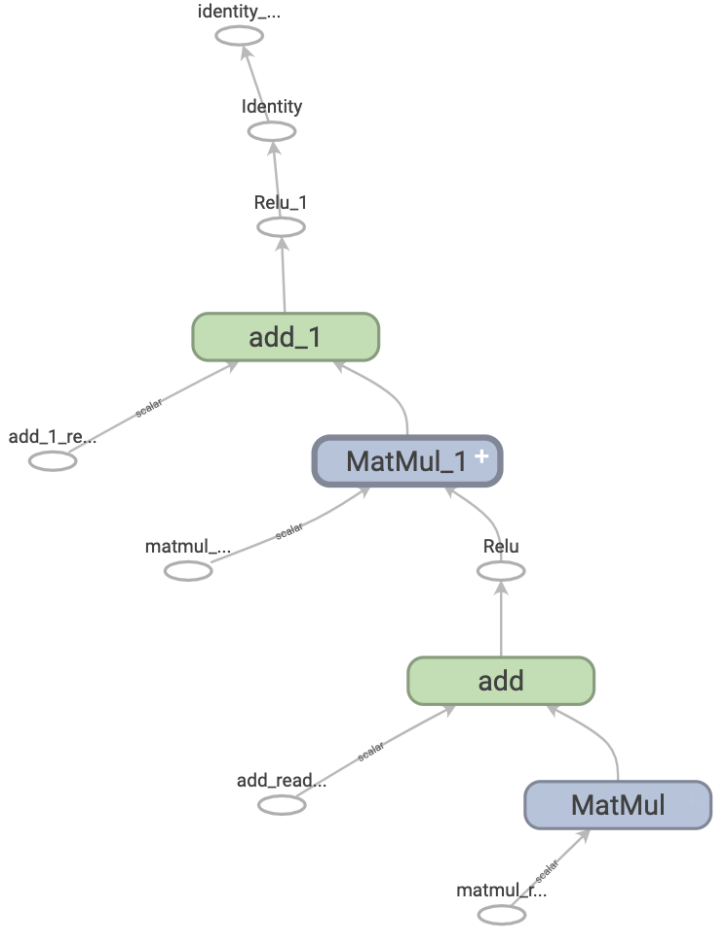
The nodes of the graph are

- Operations (e.g., addition, multiplication):
    - `tf.Operation` objects
- Variables/Constants/Placeholders: containers for values:
    - `Tensor` objects

Edges are directed into a node representing an operation

- from the outputs of other nodes
- upon which this node's operation depends

Here is an example of a graph from the reference



There are many advantages (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/intr to representing a program as a graph.

But the primary advantage (from our perspective) is that the graph form of program allows the automatic computation of analytical derivatives

- which are the basis for Gradient Descent, the optimization algorithm
    - compute gradients of parameterized computations with respect to its parameters
    - we can adjust parameter values in the negative direction of the gradient of a Loss
    - in order to solve the optimization problem: minimize Loss
    - which is the basis of training a Neural Network

Disadvantages

- The operations are *limited* to TensorFlow Operations
    - take Tensors as inputs
    - produce Tensors as outputs
    - can't have arbitrary Python operations
        - "side effects" (e.g., `print`) may not be included in the graph !
        - Modifying Python `list` or `dict` is a side-effect (https://colab.research.google.com/github/tensorflow/docs/blob/ !
- *Can* use Python control (`if-then-else`, `while`)
    - the Python `if` will be converted into `tf.cond`
    - the Python `for`, `while` looping constructs will be converted into `tf.whil`

This sometimes appears confusing

- many seemingly "Python" operators (e.g, "+") are *overloaded*
    - They perform a TensorFlow op (`tf.add`) when operating on `Tensor` arguments
    - Just like `'NumPy` overloads "+" to perform `numpy.add` when operating on NumPy arrays (`ndarray`)
- So code *looks like* Python but is really TensorFlow ops
- NumPy `ndarray` and `Tensor` are easily interchangeable and sometimes conversions happen "under the covers"

It feels like we *ought to be able* to use `.numpy()` on an `EagerTensor`, perform NumPy operations, and convert back to a `Tensor`

- Problem: works find in Python prototyping
- Fails when embedded in TensorFlow
    - arguments become `Tensor` rather than `EagerTensor` and don't respond to `.numpy()`

Bottom line

Be wary of using arbitrary Python libraries in the body of a function intended to execute in TensorFlow.

Why do we care ?

This may seem like TMI (Too Much Information)

- we ordinarily don't care about the "compiled form" of programs that we write

The reasons we care

- Evaluating a program via "eager computation" is not always identical to evaluating it via "graph computation"
- Your code is turned into a graph
    - The underlying library calls you make might throw a error
    - The error may only be understood by thinking about the graph
        - e.g., imagine a `map` operator
            - takes as arguments: a function definition, a Tensor
            - applies the operation to each element of the
            - the function you pass is turned into a graph
            - you discover that the function behaves differently than in eager mode

In [2]:

```
def add_one(x):
  print(f"x type: {type(x)}\n")

  result = x +1
  return result
```

In [3]:

```
val = tf.convert_to_tensor([0, 1, 2, 3], dtype=tf.float32)

print(val)
```

```
tf.Tensor([0. 1. 2. 3.], shape=(4,), dtype=float32)

2022-06-08 19:06:01.083232: I tensorflow/core/platform/cpu_feature_guard.cc:14
2] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical operati
ons:  SSE4.1 SSE4.2 AVX AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate co
mpiler flags.
2022-06-08 19:06:01.086445: I tensorflow/core/common_runtime/process_util.cc:1
46] Creating new thread pool with default inter op setting: 2. Tune using inte
r_op_parallelism_threads for best performance.
```

In [4]:
```
val_plus_1 = add_one( val )

print(f"Return type {type(val_plus_1)}\n")

val_plus_1
```

```
x type: <class 'tensorflow.python.framework.ops.EagerTensor'>

Return type <class 'tensorflow.python.framework.ops.EagerTensor'>
```

Out[4]:
```
<tf.Tensor: shape=(4,), dtype=float32, numpy=array([1., 2., 3., 4.], dtype=flo
at32)>
```

Some observations about calling `add_one` :

- We pass a `Tensor` (rather than a NumPy `ndarray`) as argument
  - simulating the usual case where the actual value is the result of a prior TensorFlow operation
- The type of the returned value is also a `Tensor`
  - that can be directly passed to the "next" step in the computation

Notice something odd ?

- The value of `val` in the calling cell is of type `Tensor`
- But the value of the argument (to which `val` is bound) is `EagerTensor`)
- And the return value is also of type `EagerTensor`

In "eager computation" (the default mode)

- to enable operations to be evaluated immediately
- `Tensors` (place-holders for values) are turned into `EagerTensors`
    - an `EagerTensor`'s value may be obtained via the `.numpy()` method

In [5]:
```
print(f"Value of result: {val_plus_1.numpy()}")
```

```
Value of result: [1. 2. 3. 4.]
```

The function body looks like plain Python.

- But the "+" operator is translated into `tf.add`, not NumPy addition for `ndarray`

Let's make this explicit

In [6]:

```
def add_one_v1(x):
  print(f"x type: {type(x)}\n")

  result = tf.add(x, 1)
  return result

add_one_v1(val)
```

```
x type: <class 'tensorflow.python.framework.ops.EagerTensor'>
```

Out[6]:

```
<tf.Tensor: shape=(4,), dtype=float32, numpy=array([1., 2., 3., 4.], dtype=flo
at32)>
```

In the first version of `add_one` ,

- it was easy to ignore the fact that the value of the formal parameter was a `Tensor` rather than an `ndarray`.
- this is because the "+" operator was overloaded (and turned into the type appropriate to its argument)
    - We can see this if we pass in an `ndarray` `

In [7]:
```
add_one( np.array( [0, 1, 2, 3]))
```

```
x type: <class 'numpy.ndarray'>
```

Out[7]:
```
array([1, 2, 3, 4])
```

Because we pass in an `ndarray` , the "+" operator gets turned in `np.add`

- resulting in a return type of `ndarray` rather than `Tensor`

In the second version, we expicitly invoked `tf.add` :

In [8]:
```
add_one_v1( [0, 1, 2])
```

```
x type: <class 'list'>
```

Out[8]:
```
<tf.Tensor: shape=(3,), dtype=int32, numpy=array([1, 2, 3], dtype=int32)>
```

We pass in a Python `list`

- which gets automatically converted into a `Tensor` when applying `tf.add`
- resulting in a return value of type `Tensor`

TensorFlow is quite forgiving in coercing objects to the correct type.

Differences between eager and graph execution

This level of comfort can get us into trouble.

Since in eager mode, the parameter is an `EagerTensor`, we can apply the `.numpy()` method to it in order to extract the values as an `ndarray`

In [9]:

```
def add_one_v2(x):
  print(f"x type: {type(x)}\n")

  result = x.numpy() + 1
  return result

add_one_v2(val)
```

```
x type: <class 'tensorflow.python.framework.ops.EagerTensor'>
```

Out[9]:

```
array([1., 2., 3., 4.], dtype=float32)
```

We turned the `EagerTensor` argument value into a NumPy `ndarray`.

The return type is now `ndarray`.

No harm:

- if we pass the `ndarray` result into another function that uses Tensor methods
- it will be converted into a `Tensor` (as with the `add_one_v1` example)

But there is a problem !

- if `add_one_v2` is evaluated via "graph execution"
- its parameter will be of type `Tensor`, not `EagerTensor`
- `Tensor` does not have a `.numpy()` method !

Your code which worked perfectly in "eager mode" will throw an error when evaluated via "graph computation".

When this occurs, it will be very confusing !

We illustrate this by applying a function to each element of a `Tensor`, using TensorFlow `map_fn`

In [10]:

```
def add_one_to_each(x):
  print(f"x type: {type(x)}\n")


  if tf.executing_eagerly():
    print("\tEager evalaution")
  else:
    print("\tGraph evaluation\n\n")

  result = x + 1
  return result
```

In [11]:
```
tf.map_fn(add_one_to_each, tf.constant([0, 1, 2]))
```

```
x type: <class 'tensorflow.python.framework.ops.EagerTensor'>

        Eager evalaution
x type: <class 'tensorflow.python.framework.ops.EagerTensor'>

        Eager evalaution
x type: <class 'tensorflow.python.framework.ops.EagerTensor'>

        Eager evalaution
```

Out[11]:
```
<tf.Tensor: shape=(3,), dtype=int32, numpy=array([1, 2, 3], dtype=int32)>
```

`add_one_to_each` gets called for each element of the `EagerTensor` argument

- resulting in a "print" statement being executed for each

But consider the version where we use `.numpy()` in the function body

In [12]:

```
def add_one_to_each_v1(x):
  print(f"x type: {type(x)}\n")

  if tf.executing_eagerly():
    print("\tEager evalaution")
  else:
    print("\tGraph evaluation\n\n")


  result = x.numpy() + 1
  return result
```

In [13]:
```
tf.map_fn(add_one_to_each_v1, tf.constant([0, 1, 2]))
```

```
x type: <class 'tensorflow.python.framework.ops.EagerTensor'>

        Eager evalaution
x type: <class 'tensorflow.python.framework.ops.EagerTensor'>

        Eager evalaution
x type: <class 'tensorflow.python.framework.ops.EagerTensor'>

        Eager evalaution
```

Out[13]:
```
<tf.Tensor: shape=(3,), dtype=int32, numpy=array([1, 2, 3], dtype=int32)>
```

Works so far.

But what if we turn `add_one_to_each` (and subsequent: `add_one_to_each_v1`) into it's graph version.

We can turn an ordinary function into a *TensorFlow Function* via the `tf.function` operator

- Note the capital "F" in Function.
    - "function" (lower case) will refer to Python
    - "Function" (upper case) will refer to `tf.Graph`

In [14]:
```
tf_add_one_to_each    = tf.function( add_one_to_each)
tf_add_one_to_each_v1 = tf.function( add_one_to_each_v1)
```

In [15]:
```
tf.map_fn( tf_add_one_to_each, tf.constant([0, 1, 2]))
```

```
x type: <class 'tensorflow.python.framework.ops.Tensor'>

        Graph evaluation



2022-06-08 19:06:02.613915: I tensorflow/compiler/mlir/mlir_graph_optimization
_pass.cc:185] None of the MLIR Optimization Passes are enabled (registered 2)
```

Out[15]:
```
<tf.Tensor: shape=(3,), dtype=int32, numpy=array([1, 2, 3], dtype=int32)>
```

No problem for the original function

- Even though the argument is now a `Tensor` rather than an `EagerTensor`
- We can see that it is executing via "Graph Evaluation"
- Behaves the same via "Graph evaluation" as in "Eager evaluation"

But what about `tf_add_one_to_each_v1`

- which needlessly (and incorrectly) uses the `.numpy()` operator

In [16]:
```
try:
  tf.map_fn( tf_add_one_to_each_v1, tf.constant([0, 1, 2]))
except Exception as e:
  print(f"map_fn fails: {e}")
```

```
x type: <class 'tensorflow.python.framework.ops.Tensor'>

        Graph evaluation


map_fn fails: in user code:

    /tmp/ipykernel_61095/903874480.py:10 add_one_to_each_v1  *
        result = x.numpy() + 1
    /home/kjp/anaconda3/envs/tf/lib/python3.9/site-packages/tensorflow/python/
framework/ops.py:401 __getattr__
        self.__getattribute__(name)

    AttributeError: 'Tensor' object has no attribute 'numpy'
```

From the above output, you can see that

- function (lower case) form of the functions executes in "Eager evaluation"
- the Function (`tf.Graph`) form of the functions executes in "Graph evaluation" mode

This may seem artificial.

But there are times when an underlying operator or library

- turns your function into a Function (`tf.graph`)
- without warning

Your experience is

- the code worked when run in "testing" mode
    - Eager evaluation in effect, by default
- the code fails when embedded in an operation that implicitly turns your Function into a function

This can be very confusing (and frustrating) !

The example serves to exemplify the need to be aware

- to limit the body of a function that gets (perhaps implicitly) turned int a TensorFlow Function (`tf.Graph`)
- to operations on `Tensor` types
- avoid "plain" Python (except for control) and NumPy

Tracing

The process of turning a function into a Function is called *tracing* (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/fun

There are some subtleties which may be confusing

- In an untyped language like Python the same function may be called with arguments of different type
    - e.g., `ndarray` , `Tensor`
    - The code generated by tracing a function sometimes depends on the type of the argument
    - different `tf.Graph` depending on type of argument passed
    - So sometimes a function can only be traced by passing in an argument
        - unlike our use of `tf.function` above

The conversion into a `tf.Graph` **does not preserve "Python side-effects"** (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/intr

- the Python "print" statement is a side-effect !
    - use `tf.print`
        - Changing a Python structure (`list`, `dict`) is a side effect too.

Here is an example from the reference

In [17]:

```
@tf.function
def get_MSE(y_true, y_pred):
  print("Calculating MSE!")
  sq_diff = tf.pow(y_true - y_pred, 2)
  return tf.reduce_mean(sq_diff)
```

In [18]:

```
y_true = tf.random.uniform([5], maxval=10, dtype=tf.int32)
y_pred = tf.random.uniform([5], maxval=10, dtype=tf.int32)
print(y_true)
print(y_pred)
```

```
tf.Tensor([3 3 6 2 9], shape=(5,), dtype=int32)
tf.Tensor([5 2 4 8 3], shape=(5,), dtype=int32)
```

In [19]:
```
get_MSE(y_true, y_pred)
```

```
Calculating MSE!
```

Out[19]:
```
<tf.Tensor: shape=(), dtype=int32, numpy=16>
```

Let's make sure we run in "Eager evaluation"

In [20]:
```
tf.config.run_functions_eagerly(True)
```

In [21]:

```
@tf.function
def get_MSE(y_true, y_pred):
  print("Calculating MSE!")
  sq_diff = tf.pow(y_true - y_pred, 2)
  return tf.reduce_mean(sq_diff)
```

In [22]:

```
error = get_MSE(y_true, y_pred)
error = get_MSE(y_true, y_pred)
error = get_MSE(y_true, y_pred)
```

```
Calculating MSE!
Calculating MSE!
Calculating MSE!
```

Now run it in "Graph evaluation"

In [23]:
```
# Don't forget to set it back when you are done.
tf.config.run_functions_eagerly(False)
```

In [24]:

```
@tf.function
def get_MSE(y_true, y_pred):
  print("Calculating MSE!")
  sq_diff = tf.pow(y_true - y_pred, 2)
  return tf.reduce_mean(sq_diff)
```

In [25]:

```
error = get_MSE(y_true, y_pred)
error = get_MSE(y_true, y_pred)
error = get_MSE(y_true, y_pred)

# Turn eager back on
tf.config.run_functions_eagerly(False)
```

```
Calculating MSE!
```

The "print" statement is executed only once in "Graph evaluation" !

- It is a Python side-effect
- The side effect is evaluated when the function is traced
    - The function is traced *only* on the first call
    - Unless the type of the argument changes (which necessitates a `tf.Graph` specific to that argument type)

Tracing is also somewhat odd to observe:

- Since the language is un-typed, it is possible to pass arguments of different type
    - The overloading of operators may depend on the type of the argument
    - There is one graph per type of argument encountered thus far
        - A new trace occurs upon seeing a new argument type or value

In [26]:

```
@tf.function
def f(x):
  print("Traced with", x)
  tf.print("Executed with", x)

f(1)
f(1)
f(2)
```

```
Traced with 1
Executed with 1
Executed with 1
Traced with 2
Executed with 2
```

In the above

- a new trace is triggered upon the first call with argument 1 and with argument 2

Advice

- Be aware of "Graph evaluation"
- If you write a Python function that operates on `Tensors`
    - Avoid turning your `Tensor` into an `ndarray`
        - if you must turn tensor `t` into an `ndarray`
            - use `np.array(t)`
            - rather than `t.numpy()`
            - they are equivalent, but only `EagerTensors` respond to the `.numpy()` methd
    - Prefer TensorFlow operators instead to Python libaries
        - TensorFlow operators are directly converted into `tf.Operation` nodes
        - The underlying Python library may do something invisibly that exposes the difference between Eager and Graph evaulation
- `tf.string` is [*not** a 1D array (https://www.tensorflow.org/guide/tensor#string_tensors)](https://www.tensorflow.org/guide/tensor#string_tensors)s is a Python string), it is 0 dimensional
    - i.e., it's length is not an array dimension
    - Can't index into it like an array

- Best not to "proto-type" something that will become a Function (`tf.Graph`) in

See also (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/fun

In [27]:
```
print("Done")
```

```
Done
```