Better performance with tf.function

Run in Google Colab (https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/guide/fu

In TensorFlow 2, eager execution (/guide/basics) is turned on by default. The user interface is intuitive and flexible (running one-off operations is much easier and faster), but this can come at the expense of performance and deployability.

You can use `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function) to make graphs out of your programs. It is a transformation tool that creates Python-independent dataflow graphs out of your Python code. This will help you create performant and portable models, and it is required to use `SavedModel`.

This guide will help you conceptualize how `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function) works under the hood, so you can use it effectively.

The main takeaways and recommendations are:

- Debug in eager mode, then decorate with `@tf.function` (https://www.tensorflow.org/api_docs/python/tf/function).
- Don't rely on Python side effects like object mutation or list appends.
- `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function) works best with TensorFlow ops; NumPy and Python calls are converted to constants.

Setup

```
import tensorflow as tf
```

```
2023-09-07 01:23:38.963463: E tensorflow/compiler/xla/stream_executor/cuda/cu
2023-09-07 01:23:38.963507: E tensorflow/compiler/xla/stream_executor/cuda/cu
```

```
2023-09-07 01:23:38.963542: E tensorflow/compiler/xla/stream_executor/cuda/cu
```

Define a helper function to demonstrate the kinds of errors you might encounter:

```
import traceback
import contextlib

# Some helper code to demonstrate the kinds of errors you might encounter.
@contextlib.contextmanager
def assert_raises(error_class):
  try:
    yield
  except error_class as e:
    print('Caught expected exception \n  {}:'.format(error_class))
    traceback.print_exc(limit=2)
  except Exception as e:
    raise e
  else:
    raise Exception('Expected {} to be raised but no error was raised!'.forma
        error_class))
```

Basics

Usage

A `Function` you define (for example by applying the `@tf.function` (https://www.tensorflow.org/api_docs/python/tf/function) decorator) is just like a core TensorFlow operation: You can execute it eagerly; you can compute gradients; and so on.

```
@tf.function  # The decorator converts `add` into a `Function`.
def add(a, b):
  return a + b

add(tf.ones([2, 2]), tf.ones([2, 2]))  #  [[2., 2.], [2., 2.]]
```

```
<tf.Tensor: shape=(2, 2), dtype=float32, numpy=
array([[2., 2.],
```

```
       [2., 2.]], dtype=float32)>
```

```
v = tf.Variable(1.0)
with tf.GradientTape() as tape:
  result = add(v, 1.0)
tape.gradient(result, v)
```

```
<tf.Tensor: shape=(), dtype=float32, numpy=1.0>
```

You can use `Functions` inside other `Functions`.

```
@tf.function
def dense_layer(x, w, b):
  return add(tf.matmul(x, w), b)

dense_layer(tf.ones([3, 2]), tf.ones([2, 2]), tf.ones([2]))
```

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

`Functions` can be faster than eager code, especially for graphs with many small ops. But for graphs with a few expensive ops (like convolutions), you may not see much speedup.

```
import timeit
conv_layer = tf.keras.layers.Conv2D(100, 3)

@tf.function
def conv_fn(image):
  return conv_layer(image)

image = tf.zeros([1, 200, 200, 100])
# Warm up
conv_layer(image); conv_fn(image)
print("Eager conv:", timeit.timeit(lambda: conv_layer(image), number=10))
print("Function conv:", timeit.timeit(lambda: conv_fn(image), number=10))
```

```
print("Note how there's not much difference in performance for convolutions")
```

```
Eager conv: 0.005479190999949424
Function conv: 0.005729689000190774
Note how there's not much difference in performance for convolutions
```

Tracing

This section exposes how `Function` works under the hood, including implementation details *which may change in the future*. However, once you understand why and when tracing happens, it's much easier to use `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function) effectively!

What is "tracing"?

A `Function` runs your program in a TensorFlow Graph (https://www.tensorflow.org/guide/intro_to_graphs#what_are_graphs). However, a `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) cannot represent all the things that you'd write in an eager TensorFlow program. For instance, Python supports polymorphism, but `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) requires its inputs to have a specified data type and dimension. Or you may perform side tasks like reading command-line arguments, raising an error, or working with a more complex Python object; none of these things can run in a `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph).

`Function` bridges this gap by separating your code in two stages:

1) In the first stage, referred to as "**tracing**", `Function` creates a new `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph). Python code runs normally, but all TensorFlow operations (like adding two Tensors) are *deferred*: they are captured by the `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) and not run.

2) In the second stage, a `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) which contains everything that was deferred in the first stage is run. This stage is much faster than the tracing stage.

Depending on its inputs, `Function` will not always run the first stage when it is called. See "Rules of tracing" (#rules_of_tracing) below to get a better sense of how it makes that determination. Skipping the first stage and only executing the second stage is what gives you TensorFlow's high performance.

When `Function` does decide to trace, the tracing stage is immediately followed by the second stage, so calling the `Function` both creates and runs the `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph). Later you will see how you can run only the tracing stage with `get_concrete_function` (#obtaining_concrete_functions).

When you pass arguments of different types into a `Function`, both stages are run:

```
@tf.function
def double(a):
  print("Tracing with", a)
  return a + a

print(double(tf.constant(1)))
print()
print(double(tf.constant(1.1)))
print()
print(double(tf.constant("a")))
print()
```

```
Tracing with Tensor("a:0", shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)

Tracing with Tensor("a:0", shape=(), dtype=float32)
tf.Tensor(2.2, shape=(), dtype=float32)

Tracing with Tensor("a:0", shape=(), dtype=string)
tf.Tensor(b'aa', shape=(), dtype=string)
```

Note that if you repeatedly call a `Function` with the same argument type, TensorFlow will skip the tracing stage and reuse a previously traced graph, as the generated graph would be identical.

```
# This doesn't print 'Tracing with ...'
print(double(tf.constant("b")))
```

```
tf.Tensor(b'bb', shape=(), dtype=string)
```

You can use `pretty_printed_concrete_signatures()` to see all of the available traces:

```
print(double.pretty_printed_concrete_signatures())
```

```
double(a)
  Args:
    a: int32 Tensor, shape=()
  Returns:
    int32 Tensor, shape=()

double(a)
  Args:
    a: float32 Tensor, shape=()
  Returns:
    float32 Tensor, shape=()

double(a)
  Args:
```

So far, you've seen that `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function) creates a cached, dynamic dispatch layer over TensorFlow's graph tracing logic. To be more specific about the terminology:

- A `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) is the raw, language-agnostic, portable representation of a TensorFlow computation.
- A `ConcreteFunction` wraps a `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph).
- A `Function` manages a cache of `ConcreteFunction`s and picks the right one for your inputs.
- `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function) wraps a Python function, returning a `Function` object.
- **Tracing** creates a `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) and wraps it in a `ConcreteFunction`, also known as a **trace.**

Rules of tracing

When called, a `Function` matches the call arguments to existing `ConcreteFunction`s using `tf.types.experimental.TraceType` (https://www.tensorflow.org/api_docs/python/tf/types/experimental/TraceType) of each argument. If a matching `ConcreteFunction` is found, the call is dispatched to it. If no match is found, a new `ConcreteFunction` is traced.

If multiple matches are found, the most specific signature is chosen. Matching is done by subtyping (https://en.wikipedia.org/wiki/Subtyping), much like normal function calls in C++ or Java, for instance. For example, `TensorShape([1, 2])` is a subtype of `TensorShape([None, None])` and so a call to the tf.function with `TensorShape([1, 2])` can be dispatched to the `ConcreteFunction` produced with `TensorShape([None, None])` but if a `ConcreteFunction` with `TensorShape([1, None])` also exists then it will be prioritized since it is more specific.

The `TraceType` is determined from input arguments as follows:

- For `Tensor`, the type is parameterized by the `Tensor`'s `dtype` and `shape`; ranked shapes are a subtype of unranked shapes; fixed dimensions are a subtype of unknown dimensions
- For `Variable`, the type is similar to `Tensor`, but also includes a unique resource ID of the variable, necessary to correctly wire control dependencies
- For Python primitive values, the type corresponds to the **value** itself. For example, the `TraceType` of the value `3` is `LiteralTraceType<3>`, not `int`.
- For Python ordered containers such as `list` and `tuple`, etc., the type is parameterized by the types of their elements; for example, the type of `[1, 2]` is `ListTraceType<LiteralTraceType<1>, LiteralTraceType<2>>` and the type for `[2, 1]` is `ListTraceType<LiteralTraceType<2>, LiteralTraceType<1>>` which is different.
- For Python mappings such as `dict`, the type is also a mapping from the same keys but to the types of values instead the actual values. For example, the type of `{1: 2, 3: 4}`, is `MappingTraceType<<KeyValue<1, LiteralTraceType<2>>>, <KeyValue<3, LiteralTraceType<4>>>>`. However, unlike ordered containers, `{1: 2, 3: 4}` and `{3: 4, 1: 2}` have equivalent types.
- For Python objects which implement the `__tf_tracing_type__` method, the type is whatever that method returns
- For any other Python objects, the type is a generic `TraceType`, its matching precedure is:
    - First it checks if the object is the same object used in the previous trace (using python `id()` or `is`). Note that this will still match if the object has changed, so if you use python objects as `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function) arguments it's best to use *immutable* ones.

- Next it checks if the object is equal to the object used in the previous trace (using python ==).

Note that this procedure only keeps a weakref (https://docs.python.org/3/library/weakref.html) to the object and hence only works as long as the object is in scope/not deleted.)

Note: `TraceType` is based on the `Function` input parameters so changes to global and free variables (https://docs.python.org/3/reference/executionmodel.html#binding-of-names) alone will not create a new trace. See this section (#depending_on_python_global_and_free_variables) for recommended practices when dealing with Python global and free variables.

Controlling retracing

Retracing, which is when your `Function` creates more than one trace, helps ensure that TensorFlow generates correct graphs for each set of inputs. However, tracing is an expensive operation! If your `Function` retraces a new graph for every call, you'll find that your code executes more slowly than if you didn't use `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function).

To control the tracing behavior, you can use the following techniques:

Pass a fixed `input_signature` to `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function)

```
@tf.function(input_signature=(tf.TensorSpec(shape=[None], dtype=tf.int32),))
def next_collatz(x):
  print("Tracing with", x)
  return tf.where(x % 2 == 0, x // 2, 3 * x + 1)

print(next_collatz(tf.constant([1, 2])))
# You specified a 1-D tensor in the input signature, so this should fail.
with assert_raises(TypeError):
  next_collatz(tf.constant([[1, 2], [3, 4]]))

# You specified an int32 dtype in the input signature, so this should fail.
with assert_raises(TypeError):
  next_collatz(tf.constant([1.0, 2.0]))
```

```
Tracing with Tensor("x:0", shape=(None,), dtype=int32)
tf.Tensor([4 1], shape=(2,), dtype=int32)
Caught expected exception
  <class 'TypeError'>:
Caught expected exception
  <class 'TypeError'>:
Traceback (most recent call last):
  File "/tmpfs/tmp/ipykernel_8877/3551158538.py", line 8, in assert_raises
    yield
  File "/tmpfs/tmp/ipykernel_8877/3657259638.py", line 9, in <module>
    next_collatz(tf.constant([[1, 2], [3, 4]]))
TypeError: Binding inputs to tf.function failed due to `Can not cast TensorS
array([[1, 2],
```

Use unknown dimensions for flexibility

Since TensorFlow matches tensors based on their shape, using a **`None`** dimension as a wildcard will allow **`Function`**s to reuse traces for variably-sized input. Variably-sized input can occur if you have sequences of different length, or images of different sizes for each batch. You can check out the Transformer (https://www.tensorflow.org/text/tutorials/transformer) and Deep Dream (https://www.tensorflow.org/tutorials/generative/deepdream) tutorials for examples.

```
@tf.function(input_signature=(tf.TensorSpec(shape=[None], dtype=tf.int32),))
def g(x):
  print('Tracing with', x)
  return x

# No retrace!
print(g(tf.constant([1, 2, 3])))
print(g(tf.constant([1, 2, 3, 4, 5])))
```

```
Tracing with Tensor("x:0", shape=(None,), dtype=int32)
tf.Tensor([1 2 3], shape=(3,), dtype=int32)
tf.Tensor([1 2 3 4 5], shape=(5,), dtype=int32)
```

Pass tensors instead of python literals

Often, Python arguments are used to control hyperparameters and graph constructions - for example, `num_layers=10` or `training=True` or `nonlinearity='relu'`. So, if the Python argument changes, it makes sense that you'd have to retrace the graph.

However, it's possible that a Python argument is not being used to control graph construction. In these cases, a change in the Python value can trigger needless retracing. Take, for example, this training loop, which AutoGraph will dynamically unroll. Despite the multiple traces, the generated graph is actually identical, so retracing is unnecessary.

```
def train_one_step():
  pass

@tf.function
def train(num_steps):
  print("Tracing with num_steps = ", num_steps)
  tf.print("Executing with num_steps = ", num_steps)
  for _ in tf.range(num_steps):
    train_one_step()

print("Retracing occurs for different Python arguments.")
train(num_steps=10)
train(num_steps=20)

print()
print("Traces are reused for Tensor arguments.")
train(num_steps=tf.constant(10))
train(num_steps=tf.constant(20))
```

```
Retracing occurs for different Python arguments.
Tracing with num_steps =  10
Executing with num_steps =  10
Tracing with num_steps =  20
Executing with num_steps =  20

Traces are reused for Tensor arguments.
Tracing with num_steps =  Tensor("num_steps:0", shape=(), dtype=int32)
Executing with num_steps =  10
Executing with num_steps =  20
```

If you need to force retracing, create a new `Function`. Separate `Function` objects are guaranteed not to share traces.

```
def f():
  print('Tracing!')
  tf.print('Executing')

tf.function(f)()
tf.function(f)()
```

```
Tracing!
Executing
Tracing!
Executing
```

Use the tracing protocol

Where possible, you should prefer converting the Python type into a `tf.experimental.ExtensionType` (https://www.tensorflow.org/api_docs/python/tf/experimental/ExtensionType) instead. Moreover, the `TraceType` of an `ExtensionType` is the `tf.TypeSpec` (https://www.tensorflow.org/api_docs/python/tf/TypeSpec) associated with it. Therefore, if needed, you can simply override the default `tf.TypeSpec` (https://www.tensorflow.org/api_docs/python/tf/TypeSpec) to take control of an `ExtensionType`'s `Tracing Protocol`. Refer to the *Customizing the ExtensionType's TypeSpec* section in the Extension types (/guide/extension_type) guide for details.

Otherwise, for direct control over when `Function` should retrace in regards to a particular Python type, you can implement the `Tracing Protocol` for it yourself.

```
@tf.function
def get_mixed_flavor(fruit_a, fruit_b):
  return fruit_a.flavor + fruit_b.flavor

class Fruit:
  flavor = tf.constant([0, 0])

class Apple(Fruit):
  flavor = tf.constant([1, 2])

class Mango(Fruit):
  flavor = tf.constant([3, 4])
```

```
# As described in the above rules, a generic TraceType for `Apple` and `Mango
# is generated (and a corresponding ConcreteFunction is traced) but it fails
# match the second function call since the first pair of Apple() and Mango()
# have gone out out of scope by then and deleted.
get_mixed_flavor(Apple(), Mango()) # Traces a new concrete function
get_mixed_flavor(Apple(), Mango()) # Traces a new concrete function again

# However, each subclass of the `Fruit` class has a fixed flavor, and you
# can reuse an existing traced concrete function if it was the same
# subclass. Avoiding such unnecessary tracing of concrete functions
# can have significant performance benefits.

class FruitTraceType(tf.types.experimental.TraceType):
  def __init__(self, fruit):
    self.fruit_type = type(fruit)
    self.fruit_value = fruit

  def is_subtype_of(self, other):
      # True if self subtypes `other` and `other`'s type matches FruitTraceTy
      return (type(other) is FruitTraceType and
              self.fruit_type is other.fruit_type)

  def most_specific_common_supertype(self, others):
      # `self` is the specific common supertype if all input types match it.
      return self if all(self == other for other in others) else None

  def placeholder_value(self, placeholder_context=None):
      # Use the fruit itself instead of the type for correct tracing.
      return self.fruit_value

  def __eq__(self, other):
    return type(other) is FruitTraceType and self.fruit_type == other.fruit_t

  def __hash__(self):
    return hash(self.fruit_type)

class FruitWithTraceType:

  def __tf_tracing_type__(self, context):
    return FruitTraceType(self)

class AppleWithTraceType(FruitWithTraceType):
  flavor = tf.constant([1, 2])

class MangoWithTraceType(FruitWithTraceType):
  flavor = tf.constant([3, 4])

# Now if you try calling it again:
```

```
get_mixed_flavor(AppleWithTraceType(), MangoWithTraceType()) # Traces a new c
get_mixed_flavor(AppleWithTraceType(), MangoWithTraceType()) # Re-uses the tr
```

```
<tf.Tensor: shape=(2,), dtype=int32, numpy=array([4, 6], dtype=int32)>
```

Obtaining concrete functions

Every time a function is traced, a new concrete function is created. You can directly obtain a concrete function, by using `get_concrete_function`.

```
print("Obtaining concrete trace")
double_strings = double.get_concrete_function(tf.constant("a"))
print("Executing traced function")
print(double_strings(tf.constant("a")))
print(double_strings(a=tf.constant("b")))
```

```
Obtaining concrete trace
Executing traced function
tf.Tensor(b'aa', shape=(), dtype=string)
tf.Tensor(b'bb', shape=(), dtype=string)
```

```
# You can also call get_concrete_function on an InputSpec
double_strings_from_inputspec = double.get_concrete_function(tf.TensorSpec(sh
print(double_strings_from_inputspec(tf.constant("c")))
```

```
tf.Tensor(b'cc', shape=(), dtype=string)
```

Printing a `ConcreteFunction` displays a summary of its input arguments (with types) and its output type.

```
print(double_strings)
```

```
ConcreteFunction double(a)
  Args:
    a: string Tensor, shape=()
  Returns:
    string Tensor, shape=()
```

You can also directly retrieve a concrete function's signature.

```
print(double_strings.structured_input_signature)
print(double_strings.structured_outputs)
```

```
((TensorSpec(shape=(), dtype=tf.string, name='a'),), {})
Tensor("Identity:0", shape=(), dtype=string)
```

Using a concrete trace with incompatible types will throw an error

```
with assert_raises(tf.errors.InvalidArgumentError):
  double_strings(tf.constant(1))
```

```
Caught expected exception
  <class 'tensorflow.python.framework.errors_impl.InvalidArgumentError'>:
Traceback (most recent call last):
  File "/tmpfs/src/tf_docs_env/lib/python3.9/site-packages/tensorflow/python/
    bound_arguments = function_type.bind_with_defaults(
  File "/tmpfs/src/tf_docs_env/lib/python3.9/site-packages/tensorflow/core/fu
    with_default_args[arg_name] = constraint._cast(  # pylint: disable=protec
TypeError: Can not cast TensorSpec(shape=(), dtype=tf.int32, name=None) to Te

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "/tmpfs/src/tf_docs_env/lib/python3.9/site-packages/tensorflow/python/
    return self._call_with_structured_signature(args, kwargs)
```

You may notice that Python arguments are given special treatment in a concrete function's input signature. Prior to TensorFlow 2.3, Python arguments were simply removed from the concrete function's signature. Starting with TensorFlow 2.3, Python arguments remain in the signature, but are constrained to take the value set during tracing.

```
@tf.function
def pow(a, b):
  return a ** b

square = pow.get_concrete_function(a=tf.TensorSpec(None, tf.float32), b=2)
print(square)
```

```
ConcreteFunction pow(a, b=2)
  Args:
    a: float32 Tensor, shape=<unknown>
  Returns:
    float32 Tensor, shape=<unknown>
```

```
assert square(tf.constant(10.0)) == 100

with assert_raises(TypeError):
  square(tf.constant(10.0), b=3)
```

```
Caught expected exception
  <class 'TypeError'>:
Traceback (most recent call last):
  File "/tmpfs/src/tf_docs_env/lib/python3.9/site-packages/tensorflow/python/
    bound_arguments = function_type.bind_with_defaults(
  File "/tmpfs/src/tf_docs_env/lib/python3.9/site-packages/tensorflow/core/fu
    with_default_args[arg_name] = constraint._cast(  # pylint: disable=protec
ValueError: Can not cast 3 to Literal(value=2)

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "/tmpfs/src/tf_docs_env/lib/python3.9/site-packages/tensorflow/python/
    return self._call_with_structured_signature(args, kwargs)
```

Obtaining graphs

Each concrete function is a callable wrapper around a **`tf.Graph`** (https://www.tensorflow.org/api_docs/python/tf/Graph). Although retrieving the actual **`tf.Graph`** (https://www.tensorflow.org/api_docs/python/tf/Graph) object is not something you'll normally need to do, you can obtain it easily from any concrete function.

```
graph = double_strings.graph
for node in graph.as_graph_def().node:
  print(f'{node.input} -> {node.name}')
```

```
[] -> a
['a', 'a'] -> add
['add'] -> Identity
```

Debugging

In general, debugging code is easier in eager mode than inside `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function). You should ensure that your code executes error-free in eager mode before decorating with `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function). To assist in the debugging process, you can call `tf.config.run_functions_eagerly(True)` (https://www.tensorflow.org/api_docs/python/tf/config/run_functions_eagerly) to globally disable and reenable `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function).

When tracking down issues that only appear within `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function), here are some tips:

- Plain old Python `print` calls only execute during tracing, helping you track down when your function gets (re)traced.
- `tf.print` (https://www.tensorflow.org/api_docs/python/tf/print) calls will execute every time, and can help you track down intermediate values during execution.
- `tf.debugging.enable_check_numerics` (https://www.tensorflow.org/api_docs/python/tf/debugging/enable_check_numerics) is an easy way to track down where NaNs and Inf are created.
- `pdb` (the Python debugger (https://docs.python.org/3/library/pdb.html)) can help you understand what's going on during tracing. (Caveat: `pdb` will drop you into AutoGraph-transformed source code.)

AutoGraph transformations

AutoGraph is a library that is on by default in `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function), and transforms a subset of Python eager code into graph-compatible TensorFlow ops. This includes control flow like `if`, `for`, `while`.

TensorFlow ops like `tf.cond` (https://www.tensorflow.org/api_docs/python/tf/cond) and `tf.while_loop` (https://www.tensorflow.org/api_docs/python/tf/while_loop) continue to work, but control flow is often easier to write and understand when written in Python.

```
# A simple loop

@tf.function
def f(x):
  while tf.reduce_sum(x) > 1:
    tf.print(x)
    x = tf.tanh(x)
  return x

f(tf.random.uniform([5]))
```

```
[0.421437502 0.579795122 0.521502137 0.594003677 0.983272552]
[0.398140728 0.522516489 0.478858501 0.532768965 0.754479051]
[0.378356963 0.479639918 0.445328951 0.487494886 0.637813449]
[0.361279756 0.445955217 0.418051839 0.452225834 0.563408911]
[0.346340775 0.418568552 0.395287931 0.423726916 0.510502338]
[0.333126366 0.39572382 0.375909954 0.400065511 0.470336497]
[0.32132715 0.376284122 0.359150231 0.380005 0.438471138]
[0.310706407 0.35947606 0.344465315 0.362711817 0.412376374]
[0.301079571 0.344752431 0.331458 0.3476004 0.390488565]
[0.292300254 0.331713527 0.319830239 0.334245741 0.371781319]
[0.284250587 0.320059627 0.309353411 0.322330564 0.355548829]
[0.276834488 0.309560835 0.299848735 0.311612695 0.341287315]
[0.269972801 0.300037533 0.291174173 0.301903516 0.328626156]
[0.263599515 0.291346937 0.283215135 0.293053627 0.317285746]
```

If you're curious you can inspect the code autograph generates.

```
print(tf.autograph.to_code(f.python_function))
```

```
        def loop_body():
            nonlocal x
            ag__.converted_call(ag__.ld(tf).print, (ag__.ld(x),), None, fscop
            x = ag__.converted_call(ag__.ld(tf).tanh, (ag__.ld(x),), None, fs

        def loop_test():
            return ag__.converted_call(ag__.ld(tf).reduce_sum, (ag__.ld(x),)
        ag__.while_stmt(loop_test, loop_body, get_state, set_state, ('x',),
        try:
            do_return = True
            retval_ = ag__.ld(x)
        except:
```

Conditionals

AutoGraph will convert some `if <condition>` statements into the equivalent `tf.cond` calls. This substitution is made if `<condition>` is a Tensor. Otherwise, the `if` statement is executed as a Python conditional.

A Python conditional executes during tracing, so exactly one branch of the conditional will be added to the graph. Without AutoGraph, this traced graph would be unable to take the alternate branch if there is data-dependent control flow.

`tf.cond` (https://www.tensorflow.org/api_docs/python/tf/cond) traces and adds both branches of the conditional to the graph, dynamically selecting a branch at execution time. Tracing can have unintended side effects; check out AutoGraph tracing effects (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/autograph/g3doc/reference/control_flow.md#effects-of-the-tracing-process) for more information.

```
@tf.function
def fizzbuzz(n):
  for i in tf.range(1, n + 1):
    print('Tracing for loop')
    if i % 15 == 0:
      print('Tracing fizzbuzz branch')
      tf.print('fizzbuzz')
    elif i % 3 == 0:
      print('Tracing fizz branch')
      tf.print('fizz')
    elif i % 5 == 0:
      print('Tracing buzz branch')
      tf.print('buzz')
    else:
      print('Tracing default branch')
      tf.print(i)
```

```
fizzbuzz(tf.constant(5))
fizzbuzz(tf.constant(20))
```

```
Tracing for loop
Tracing fizzbuzz branch
Tracing fizz branch
Tracing buzz branch
Tracing default branch
1
2
fizz
4
buzz
1
2
fizz
4
```

See the reference documentation (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/autograph/g3doc/reference/control_flow.md#if-statements) for additional restrictions on AutoGraph-converted if statements.

Loops

AutoGraph will convert some `for` and `while` statements into the equivalent TensorFlow looping ops, like `tf.while_loop` (https://www.tensorflow.org/api_docs/python/tf/while_loop). If not converted, the `for` or `while` loop is executed as a Python loop.

This substitution is made in the following situations:

- `for x in y`: if `y` is a Tensor, convert to `tf.while_loop` (https://www.tensorflow.org/api_docs/python/tf/while_loop). In the special case where `y` is a `tf.data.Dataset` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset), a combination of `tf.data.Dataset` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) ops are generated.
- `while <condition>`: if `<condition>` is a Tensor, convert to `tf.while_loop`.

A Python loop executes during tracing, adding additional ops to the `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) for every iteration of the loop.

A TensorFlow loop traces the body of the loop, and dynamically selects how many iterations to run at execution time. The loop body only appears once in the generated `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph).

See the reference documentation (https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/autograph/g3doc/reference/control_flow.md#while-statements) for additional restrictions on AutoGraph-converted `for` and `while` statements.

Looping over Python data

A common pitfall is to loop over Python/NumPy data within a `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function). This loop will execute during the tracing process, adding a copy of your model to the `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) for each iteration of the loop.

If you want to wrap the entire training loop in `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function), the safest way to do this is to wrap your data as a `tf.data.Dataset` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset) so that AutoGraph will dynamically unroll the training loop.

```
def measure_graph_size(f, *args):
  g = f.get_concrete_function(*args).graph
  print("{}({}) contains {} nodes in its graph".format(
      f.__name__, ', '.join(map(str, args)), len(g.as_graph_def().node)))

@tf.function
def train(dataset):
  loss = tf.constant(0)
  for x, y in dataset:
    loss += tf.abs(y - x) # Some dummy computation.
  return loss

small_data = [(1, 1)] * 3
big_data = [(1, 1)] * 10
measure_graph_size(train, small_data)
measure_graph_size(train, big_data)

measure_graph_size(train, tf.data.Dataset.from_generator(
    lambda: small_data, (tf.int32, tf.int32)))
measure_graph_size(train, tf.data.Dataset.from_generator(
    lambda: big_data, (tf.int32, tf.int32)))
```

```
train([(1, 1), (1, 1), (1, 1)]) contains 11 nodes in its graph
train([(1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1), (1, 1)
train(<_FlatMapDataset element_spec=(TensorSpec(shape=<unknown>, dtype=tf.int
train(<_FlatMapDataset element_spec=(TensorSpec(shape=<unknown>, dtype=tf.int
```

When wrapping Python/NumPy data in a Dataset, be mindful of `tf.data.Dataset.from_generator` (https://www.tensorflow.org/api_docs/python/tf/data/Dataset#from_generator) versus `tf.data.Dataset.from_tensor_slices`. The former will keep the data in Python and fetch it via `tf.py_function` (https://www.tensorflow.org/api_docs/python/tf/py_function) which can have performance implications, whereas the latter will bundle a copy of the data as one large `tf.constant()` (https://www.tensorflow.org/api_docs/python/tf/constant) node in the graph, which can have memory implications.

Reading data from files via `TFRecordDataset`, `CsvDataset`, etc. is the most effective way to consume data, as then TensorFlow itself can manage the asynchronous loading and prefetching of data, without having to involve Python. To learn more, see the `tf.data`: Build TensorFlow input pipelines (/guide/data) guide.

Accumulating values in a loop

A common pattern is to accumulate intermediate values from a loop. Normally, this is accomplished by appending to a Python list or adding entries to a Python dictionary. However, as these are Python side effects, they will not work as expected in a dynamically unrolled loop. Use `tf.TensorArray` (https://www.tensorflow.org/api_docs/python/tf/TensorArray) to accumulate results from a dynamically unrolled loop.

```
batch_size = 2
seq_len = 3
feature_size = 4

def rnn_step(inp, state):
  return inp + state

@tf.function
def dynamic_rnn(rnn_step, input_data, initial_state):
  # [batch, time, features] -> [time, batch, features]
  input_data = tf.transpose(input_data, [1, 0, 2])
  max_seq_len = input_data.shape[0]

  states = tf.TensorArray(tf.float32, size=max_seq_len)
```

```
  state = initial_state
  for i in tf.range(max_seq_len):
    state = rnn_step(input_data[i], state)
    states = states.write(i, state)
  return tf.transpose(states.stack(), [1, 0, 2])

dynamic_rnn(rnn_step,
            tf.random.uniform([batch_size, seq_len, feature_size]),
            tf.zeros([batch_size, feature_size]))
```

```
<tf.Tensor: shape=(2, 3, 4), dtype=float32, numpy=
array([[[0.06165314, 0.06311452, 0.08294392, 0.34895766],
        [0.5234206 , 0.44568503, 0.9094547 , 1.2709353 ],
        [1.0457213 , 0.5137875 , 1.356309  , 1.2856586 ]],

       [[0.53330934, 0.9121362 , 0.4042704 , 0.52004325],
        [0.5769839 , 1.3394864 , 1.1908944 , 1.3369724 ],
        [1.4918177 , 1.9452012 , 1.8275222 , 1.413373  ]]], dtype=float32)>
```

Limitations

TensorFlow `Function` has a few limitations by design that you should be aware of when converting a Python function to a `Function`.

Executing Python side effects

Side effects, like printing, appending to lists, and mutating globals, can behave unexpectedly inside a `Function`, sometimes executing twice or not all. They only happen the first time you call a `Function` with a set of inputs. Afterwards, the traced `tf.Graph` (https://www.tensorflow.org/api_docs/python/tf/Graph) is reexecuted, without executing the Python code.

The general rule of thumb is to avoid relying on Python side effects in your logic and only use them to debug your traces. Otherwise, TensorFlow APIs like `tf.data` (https://www.tensorflow.org/api_docs/python/tf/data), `tf.print` (https://www.tensorflow.org/api_docs/python/tf/print), `tf.summary` (https://www.tensorflow.org/api_docs/python/tf/summary), `tf.Variable.assign` (https://www.tensorflow.org/api_docs/python/tf/Variable#assign), and `tf.TensorArray`

(https://www.tensorflow.org/api_docs/python/tf/TensorArray) are the best way to ensure your code will be executed by the TensorFlow runtime with each call.

```
@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
```

If you would like to execute Python code during each invocation of a `Function`, `tf.py_function` (https://www.tensorflow.org/api_docs/python/tf/py_function) is an exit hatch. The drawback of `tf.py_function` (https://www.tensorflow.org/api_docs/python/tf/py_function) is that it's not portable or particularly performant, cannot be saved with SavedModel, and does not work well in distributed (multi-GPU, TPU) setups. Also, since `tf.py_function` (https://www.tensorflow.org/api_docs/python/tf/py_function) has to be wired into the graph, it casts all inputs/outputs to tensors.

Changing Python global and free variables

Changing Python global and free variables (https://docs.python.org/3/reference/executionmodel.html#binding-of-names) counts as a Python side effect, so it only happens during tracing.

```
external_list = []

@tf.function
def side_effect(x):
  print('Python side effect')
  external_list.append(x)

side_effect(1)
```

```
side_effect(1)
side_effect(1)
# The list append only happened once!
assert len(external_list) == 1
```

```
Python side effect
```

Sometimes unexpected behaviors are very hard to notice. In the example below, the `counter` is intended to safeguard the increment of a variable. However because it is a python integer and not a TensorFlow object, it's value is captured during the first trace. When the `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function) is used, the `assign_add` will be recorded unconditionally in the underlying graph. Therefore `v` will increase by 1, every time the `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function) is called. This issue is common among users that try to migrate their Grpah-mode Tensorflow code to Tensorflow 2 using `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function) decorators, when python side-effects (the `counter` in the example) are used to determine what ops to run (`assign_add` in the example). Usually, users realize this only after seeing suspicious numerical results, or significantly lower performance than expected (e.g. if the guarded operation is very costly).

```
class Model(tf.Module):
  def __init__(self):
    self.v = tf.Variable(0)
    self.counter = 0

  @tf.function
  def __call__(self):
    if self.counter == 0:
      # A python side-effect
      self.counter += 1
      self.v.assign_add(1)

    return self.v

m = Model()
for n in range(3):
  print(m().numpy()) # prints 1, 2, 3
```

```
1
2
3
```

A workaround to achieve the expected behavior is using `tf.init_scope` (https://www.tensorflow.org/api_docs/python/tf/init_scope) to lift the operations outside of the function graph. This ensures that the variable increment is only done once during tracing time. It should be noted `init_scope` has other side effects including cleared control flow and gradient tape. Sometimes the usage of `init_scope` can become too complex to manage realistically.

```
class Model(tf.Module):
  def __init__(self):
    self.v = tf.Variable(0)
    self.counter = 0

  @tf.function
  def __call__(self):
    if self.counter == 0:
      # Lifts ops out of function-building graphs
      with tf.init_scope():
        self.counter += 1
        self.v.assign_add(1)

    return self.v

m = Model()
for n in range(3):
  print(m().numpy()) # prints 1, 1, 1
```

```
1
1
1
```

In summary, as a rule of thumb, you should avoid mutating python objects such as integers or containers like lists that live outside the `Function`. Instead, use arguments and TF objects. For example, the section "Accumulating values in a loop" (#accumulating_values_in_a_loop) has one example of how list-like operations can be implemented.

You can, in some cases, capture and manipulate state if it is a `tf.Variable` (https://www.tensorflow.org/guide/variable). This is how the weights of Keras models are updated with repeated calls to the same `ConcreteFunction`.

Using Python iterators and generators

Many Python features, such as generators and iterators, rely on the Python runtime to keep track of state. In general, while these constructs work as expected in eager mode, they are examples of Python side effects and therefore only happen during tracing.

```
@tf.function
def buggy_consume_next(iterator):
  tf.print("Value:", next(iterator))

iterator = iter([1, 2, 3])
buggy_consume_next(iterator)
# This reuses the first value from the iterator, rather than consuming the ne
buggy_consume_next(iterator)
buggy_consume_next(iterator)
```

```
Value: 1
Value: 1
Value: 1
```

Just like how TensorFlow has a specialized `tf.TensorArray` (https://www.tensorflow.org/api_docs/python/tf/TensorArray) for list constructs, it has a specialized `tf.data.Iterator` (https://www.tensorflow.org/api_docs/python/tf/data/Iterator) for iteration constructs. See the section on AutoGraph transformations (#autograph_transformations) for an overview. Also, the `tf.data` (https://www.tensorflow.org/guide/data) API can help implement generator patterns:

```
@tf.function
def good_consume_next(iterator):
  # This is ok, iterator is a tf.data.Iterator
  tf.print("Value:", next(iterator))

ds = tf.data.Dataset.from_tensor_slices([1, 2, 3])
iterator = iter(ds)
good_consume_next(iterator)
good_consume_next(iterator)
```

```
good_consume_next(iterator)
```

```
Value: 1
Value: 2
Value: 3
```

All outputs of a tf.function must be return values

With the exception of **`tf.Variable`** (https://www.tensorflow.org/api_docs/python/tf/Variable)s, a tf.function must return all its outputs. Attempting to directly access any tensors from a function without going through return values causes "leaks".

For example, the function below "leaks" the tensor `a` through the Python global `x`:

```
x = None

@tf.function
def leaky_function(a):
  global x
  x = a + 1  # Bad - leaks local tensor
  return a + 2

correct_a = leaky_function(tf.constant(1))

print(correct_a.numpy())  # Good - value obtained from function's returns
try:
  x.numpy()  # Bad - tensor leaked from inside the function, cannot be used h
except AttributeError as expected:
  print(expected)
```

```
3
'SymbolicTensor' object has no attribute 'numpy'
```

This is true even if the leaked value is also returned:

```
@tf.function
def leaky_function(a):
  global x
```

```
  x = a + 1  # Bad - leaks local tensor
  return x  # Good - uses local tensor

correct_a = leaky_function(tf.constant(1))

print(correct_a.numpy())  # Good - value obtained from function's returns
try:
  x.numpy()  # Bad - tensor leaked from inside the function, cannot be used h
except AttributeError as expected:
  print(expected)

@tf.function
def captures_leaked_tensor(b):
  b += x  # Bad - `x` is leaked from `leaky_function`
  return b

with assert_raises(TypeError):
  captures_leaked_tensor(tf.constant(2))
```

```
2
'SymbolicTensor' object has no attribute 'numpy'
Caught expected exception
  <class 'TypeError'>:
Traceback (most recent call last):
  File "/tmpfs/tmp/ipykernel_8877/3551158538.py", line 8, in assert_raises
    yield
  File "/tmpfs/tmp/ipykernel_8877/566849597.py", line 21, in <module>
    captures_leaked_tensor(tf.constant(2))
TypeError: <tf.Tensor 'add:0' shape=() dtype=int32> is out of scope and cann
Please see https://www.tensorflow.org/guide/function#all_outputs_of_a_tffunc

<tf.Tensor 'add:0' shape=() dtype=int32> was defined here:
  File "/usr/lib/python3.9/runpy.py", line 197, in _run_module_as_main
```

Usually, leaks such as these occur when you use Python statements or data structures. In addition to leaking inaccessible tensors, such statements are also likely wrong because they count as Python side effects, and are not guaranteed to execute at every function call.

Common ways to leak local tensors also include mutating an external Python collection, or an object:

```
class MyClass:

  def __init__(self):
```

```
    self.field = None

external_list = []
external_object = MyClass()

def leaky_function():
  a = tf.constant(1)
  external_list.append(a)  # Bad - leaks tensor
  external_object.field = a  # Bad - leaks tensor
```

Recursive tf.functions are not supported

Recursive `Function`s are not supported and could cause infinite loops. For example,

```
@tf.function
def recursive_fn(n):
  if n > 0:
    return recursive_fn(n - 1)
  else:
    return 1

with assert_raises(Exception):
  recursive_fn(tf.constant(5))  # Bad - maximum recursion error.
```

```
Caught expected exception
  <class 'Exception'>:
Traceback (most recent call last):
  File "/tmpfs/tmp/ipykernel_8877/3551158538.py", line 8, in assert_raises
    yield
  File "/tmpfs/tmp/ipykernel_8877/2233998312.py", line 9, in <module>
    recursive_fn(tf.constant(5))  # Bad - maximum recursion error.
tensorflow.python.autograph.impl.api.StagingError: in user code:

    File "/tmpfs/tmp/ipykernel_8877/2233998312.py", line 4, in recursive_fn
        return recursive_fn(n - 1)
    File "/tmpfs/tmp/ipykernel_8877/2233998312.py", line 4, in recursive_fn
        return recursive_fn(n - 1)
    File "/tmpfs/tmp/ipykernel_8877/2233998312.py", line 4, in recursive_fn
```

Even if a recursive `Function` seems to work, the python function will be traced multiple times and could have performance implication. For example,

```
@tf.function
def recursive_fn(n):
  if n > 0:
    print('tracing')
    return recursive_fn(n - 1)
  else:
    return 1

recursive_fn(5)  # Warning - multiple tracings
```

```
tracing
tracing
tracing
tracing
tracing
<tf.Tensor: shape=(), dtype=int32, numpy=1>
```

Known Issues

If your `Function` is not evaluating correctly, the error may be explained by these known issues which are planned to be fixed in the future.

Depending on Python global and free variables

`Function` creates a new `ConcreteFunction` when called with a new value of a Python argument. However, it does not do that for the Python closure, globals, or nonlocals of that `Function`. If their value changes in between calls to the `Function`, the `Function` will still use the values they had when it was traced. This is different from how regular Python functions work.

For that reason, you should follow a functional programming style that uses arguments instead of closing over outer names.

```
@tf.function
def buggy_add():
  return 1 + foo

@tf.function
```

```
def recommended_add(foo):
  return 1 + foo

foo = 1
print("Buggy:", buggy_add())
print("Correct:", recommended_add(foo))
```

```
Buggy: tf.Tensor(2, shape=(), dtype=int32)
Correct: tf.Tensor(2, shape=(), dtype=int32)
```

```
print("Updating the value of `foo` to 100!")
foo = 100
print("Buggy:", buggy_add())  # Did not change!
print("Correct:", recommended_add(foo))
```

```
Updating the value of `foo` to 100!
Buggy: tf.Tensor(2, shape=(), dtype=int32)
Correct: tf.Tensor(101, shape=(), dtype=int32)
```

Another way to update a global value, is to make it a **`tf.Variable`** (https://www.tensorflow.org/api_docs/python/tf/Variable) and use the **`Variable.assign`** (https://www.tensorflow.org/api_docs/python/tf/Variable#assign) method instead.

```
@tf.function
def variable_add():
  return 1 + foo

foo = tf.Variable(1)
print("Variable:", variable_add())
```

```
Variable: tf.Tensor(2, shape=(), dtype=int32)
```

```
print("Updating the value of `foo` to 100!")
foo.assign(100)
```

```
print("Variable:", variable_add())
```

```
Updating the value of `foo` to 100!
Variable: tf.Tensor(101, shape=(), dtype=int32)
```

Depending on Python objects

Passing custom Python objects as arguments to `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function) is supported but has certain limitations.

For maximum feature coverage, consider transforming the objects into Extension types (/guide/extension_type) before passing them to `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function). You can also use Python primitives and `tf.nest` (https://www.tensorflow.org/api_docs/python/tf/nest)-compatible structures.

However, as covered in the rules of tracing (#rules_of_tracing), when a custom `TraceType` is not provided by the custom Python class, `tf.function` (https://www.tensorflow.org/api_docs/python/tf/function) is forced to use instance-based equality which means it will **not create a new trace** when you pass the **same object with modified attributes**.

```
class SimpleModel(tf.Module):
  def __init__(self):
    # These values are *not* tf.Variables.
    self.bias = 0.
    self.weight = 2.

@tf.function
def evaluate(model, x):
  return model.weight * x + model.bias

simple_model = SimpleModel()
x = tf.constant(10.)
print(evaluate(simple_model, x))
```

```
tf.Tensor(20.0, shape=(), dtype=float32)
```

```
print("Adding bias!")
simple_model.bias += 5.0
print(evaluate(simple_model, x))  # Didn't change :(
```

```
Adding bias!
tf.Tensor(20.0, shape=(), dtype=float32)
```

Using the same **`Function`** to evaluate the modified instance of the model will be buggy since it still has the same instance-based TraceType (#rules_of_tracing) as the original model.

For that reason, you're recommended to write your **`Function`** to avoid depending on mutable object attributes or implement the Tracing Protocol (#use_the_tracing_protocol) for the objects to inform **`Function`** about such attributes.

If that is not possible, one workaround is to make new **`Function`**s each time you modify your object to force retracing:

```
def evaluate(model, x):
  return model.weight * x + model.bias

new_model = SimpleModel()
evaluate_no_bias = tf.function(evaluate).get_concrete_function(new_model, x)
# Don't pass in `new_model`, `Function` already captured its state during tra
print(evaluate_no_bias(x))
```

```
tf.Tensor(20.0, shape=(), dtype=float32)
```

```
print("Adding bias!")
new_model.bias += 5.0
# Create new Function and ConcreteFunction since you modified new_model.
evaluate_with_bias = tf.function(evaluate).get_concrete_function(new_model, x
print(evaluate_with_bias(x)) # Don't pass in `new_model`.
```

```
Adding bias!
```

```
tf.Tensor(25.0, shape=(), dtype=float32)
```

As retracing can be expensive (https://www.tensorflow.org/guide/intro_to_graphs#tracing_and_performance), you can use **`tf.Variable`** (https://www.tensorflow.org/api_docs/python/tf/Variable)s as object attributes, which can be mutated (but not changed, careful!) for a similar effect without needing a retrace.

```
class BetterModel:

  def __init__(self):
    self.bias = tf.Variable(0.)
    self.weight = tf.Variable(2.)

@tf.function
def evaluate(model, x):
  return model.weight * x + model.bias

better_model = BetterModel()
print(evaluate(better_model, x))
```

```
tf.Tensor(20.0, shape=(), dtype=float32)
```

```
print("Adding bias!")
better_model.bias.assign_add(5.0)  # Note: instead of better_model.bias += 5
print(evaluate(better_model, x))  # This works!
```

```
Adding bias!
tf.Tensor(25.0, shape=(), dtype=float32)
```

Creating tf.Variables

`Function` only supports singleton **`tf.Variable`** (https://www.tensorflow.org/api_docs/python/tf/Variable)s created once on the first call, and reused across subsequent function calls. The code snippet below would create a new

`tf.Variable` (https://www.tensorflow.org/api_docs/python/tf/Variable) in every function call, which results in a `ValueError` exception.

Example:

```
@tf.function
def f(x):
  v = tf.Variable(1.0)
  return v

with assert_raises(ValueError):
  f(1.0)
```

```
Caught expected exception
  <class 'ValueError'>:
Traceback (most recent call last):
  File "/tmpfs/tmp/ipykernel_8877/3551158538.py", line 8, in assert_raises
    yield
  File "/tmpfs/tmp/ipykernel_8877/3018268426.py", line 7, in <module>
    f(1.0)
ValueError: in user code:

    File "/tmpfs/tmp/ipykernel_8877/3018268426.py", line 3, in f  *
        v = tf.Variable(1.0)

    ValueError: tf.function only supports singleton tf.Variables created on
```

A common pattern used to work around this limitation is to start with a Python None value, then conditionally create the `tf.Variable` (https://www.tensorflow.org/api_docs/python/tf/Variable) if the value is None:

```
class Count(tf.Module):
  def __init__(self):
    self.count = None

  @tf.function
  def __call__(self):
    if self.count is None:
      self.count = tf.Variable(0)
    return self.count.assign_add(1)

c = Count()
print(c())
```

```
print(c())
```

```
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)
```

Using with multiple Keras optimizers

You may encounter `ValueError: tf.function only supports singleton tf.Variables created on the first call.` when using more than one Keras optimizer with a `tf.function`. This error occurs because optimizers internally create `tf.Variables` when they apply gradients for the first time.

```
opt1 = tf.keras.optimizers.Adam(learning_rate = 1e-2)
opt2 = tf.keras.optimizers.Adam(learning_rate = 1e-3)

@tf.function
def train_step(w, x, y, optimizer):
   with tf.GradientTape() as tape:
       L = tf.reduce_sum(tf.square(w*x - y))
   gradients = tape.gradient(L, [w])
   optimizer.apply_gradients(zip(gradients, [w]))

w = tf.Variable(2.)
x = tf.constant([-1.])
y = tf.constant([2.])

train_step(w, x, y, opt1)
print("Calling `train_step` with different optimizer...")
with assert_raises(ValueError):
  train_step(w, x, y, opt2)
```

```
Calling `train_step` with different optimizer...
Caught expected exception
  <class 'ValueError'>:
Traceback (most recent call last):
  File "/tmpfs/tmp/ipykernel_8877/3551158538.py", line 8, in assert_raises
    yield
  File "/tmpfs/tmp/ipykernel_8877/950644149.py", line 18, in <module>
    train_step(w, x, y, opt2)
ValueError: in user code:
```

```
    File "/tmpfs/tmp/ipykernel_8877/950644149.py", line 9, in train_step  *
        optimizer.apply_gradients(zip(gradients, [w]))
    File "/tmpfs/src/tf_docs_env/lib/python3.9/site-packages/keras/src/optim
```

If you need to change the optimizer during training, a workaround is to create a new `Function` for each optimizer, calling the `ConcreteFunction` (#obtaining_concrete_functions) directly.

```
opt1 = tf.keras.optimizers.Adam(learning_rate = 1e-2)
opt2 = tf.keras.optimizers.Adam(learning_rate = 1e-3)

# Not a tf.function.
def train_step(w, x, y, optimizer):
   with tf.GradientTape() as tape:
       L = tf.reduce_sum(tf.square(w*x - y))
   gradients = tape.gradient(L, [w])
   optimizer.apply_gradients(zip(gradients, [w]))

w = tf.Variable(2.)
x = tf.constant([-1.])
y = tf.constant([2.])

# Make a new Function and ConcreteFunction for each optimizer.
train_step_1 = tf.function(train_step)
train_step_2 = tf.function(train_step)
for i in range(10):
  if i % 2 == 0:
    train_step_1(w, x, y, opt1)
  else:
    train_step_2(w, x, y, opt2)
```

Using with multiple Keras models

You may also encounter `ValueError: tf.function only supports singleton tf.Variables created on the first call.` when passing different model instances to the same `Function`.

This error occurs because Keras models (which do not have their input shape defined (https://www.tensorflow.org/guide/keras/custom_layers_and_models#best_practice_deferring_weight_creation_until_the_shape_of_the_inputs_is_known)) and Keras layers create `tf.Variables` when they are first called. You may be attempting to initialize those variables inside a `Function`, which has already been called. To avoid this

error, try calling `model.build(input_shape)` to initialize all the weights before training the model.

Further reading

To learn about how to export and load a `Function`, see the SavedModel guide (https://www.tensorflow.org/guide/saved_model). To learn more about graph optimizations that are performed after tracing, see the Grappler guide (https://www.tensorflow.org/guide/graph_optimization). To learn how to optimize your data pipeline and profile your model, see the Profiler guide (https://www.tensorflow.org/guide/profiler).