Open in Colab

Autograph: Graphs for complex code

In this ungraded lab, you'll go through some of the scenarios from the lesson `Creating graphs for complex code`.

Imports

In [1]:

```
try:
    # %tensorflow_version only exists in Colab.
    %tensorflow_version 2.x
except Exception:
    pass

import tensorflow as tf
```

As you saw in the lectures, seemingly simple functions can sometimes be difficult to write in graph mode. Fortunately, Autograph generates this complex graph code for us.

- Here is a function that does some multiplication and additon.

In [2]:

```
a = tf.Variable(1.0)
b = tf.Variable(2.0)

@tf.function
def f(x,y):
    a.assign(y * b)
    b.assign_add(x * a)
    return a + b

print(f(1.0, 2.0))

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

```
tf.Tensor(10.0, shape=(), dtype=float32)
def tf__f(x, y):
    with ag__.FunctionScope('f', 'fscope', ag__.ConversionOptions(recursive=True,
user_requested=True, optional_features=(), internal_convert_user_code=True)) as fscope:
        do_return = False
        retval_ = ag__.UndefinedReturnValue()
        ag__.converted_call(ag__.ld(a).assign, ((ag__.ld(y) * ag__.ld(b)),), None, fscope)
        ag__.converted_call(ag__.ld(b).assign_add, ((ag__.ld(x) * ag__.ld(a)),), None, fscope)
        try:
            do_return = True
            retval_ = (ag__.ld(a) + ag__.ld(b))
        except:
            do_return = False
            raise
        return fscope.ret(retval_, do_return)
```

- Here is a function that checks if the sign of a number is positive or not.

In [3]:

```
@tf.function
def sign(x):
    if x > 0:
        return 'Positive'
    else:
```

```
        return 'Negative or zero'

print("Sign = {}".format(sign(tf.constant(2))))
print("Sign = {}".format(sign(tf.constant(-2))))

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

```
Sign = b'Positive'
Sign = b'Negative or zero'
def tf__sign(x):
    with ag__.FunctionScope('sign', 'fscope', ag__.ConversionOptions(recursive=True,
user_requested=True, optional_features=(), internal_convert_user_code=True)) as fscope:
        do_return = False
        retval_ = ag__.UndefinedReturnValue()

        def get_state():
            return (do_return, retval_)

        def set_state(vars_):
            nonlocal do_return, retval_
            (do_return, retval_) = vars_

        def if_body():
            nonlocal do_return, retval_
            try:
                do_return = True
                retval_ = 'Positive'
            except:
                do_return = False
                raise

        def else_body():
            nonlocal do_return, retval_
            try:
                do_return = True
                retval_ = 'Negative or zero'
            except:
                do_return = False
                raise
        ag__.if_stmt((ag__.ld(x) > 0), if_body, else_body, get_state, set_state, ('do_return', 'ret
val_'), 2)
        return fscope.ret(retval_, do_return)
```

- Here is another function that includes a while loop.

In [4]:

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

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

```
def tf__f(x):
    with ag__.FunctionScope('f', 'fscope', ag__.ConversionOptions(recursive=True,
user_requested=True, optional_features=(), internal_convert_user_code=True)) as fscope:
        do_return = False
        retval_ = ag__.UndefinedReturnValue()

        def get_state():
            return (x,)

        def set_state(vars_):
            nonlocal x
            (x,) = vars_

        def loop_body():
            nonlocal x
            ag__.converted_call(ag__.ld(tf).print, (ag__.ld(x),), None, fscope)
```

```
            x = ag__.converted_call(ag__.ld(tf).tanh, (ag__.ld(x),), None, fscope)

        def loop_test():
            return (ag__.converted_call(ag__.ld(tf).reduce_sum, (ag__.ld(x),), None, fscope) > 1)
        ag__.while_stmt(loop_test, loop_body, get_state, set_state, ('x',), {})
        try:
            do_return = True
            retval_ = ag__.ld(x)
        except:
            do_return = False
            raise
        return fscope.ret(retval_, do_return)
```

- Here is a function that uses a for loop and an if statement.

In [5]:

```
@tf.function
def sum_even(items):
    s = 0
    for c in items:
        if c % 2 > 0:
            continue
        s += c
    return s

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

```
def tf__sum_even(items):
    with ag__.FunctionScope('sum_even', 'fscope', ag__.ConversionOptions(recursive=True,
user_requested=True, optional_features=(), internal_convert_user_code=True)) as fscope:
        do_return = False
        retval_ = ag__.UndefinedReturnValue()
        s = 0

        def get_state_2():
            return (s,)

        def set_state_2(vars_):
            nonlocal s
            (s,) = vars_

        def loop_body(itr):
            nonlocal s
            c = itr
            continue_ = False

            def get_state():
                return (continue_,)

            def set_state(vars_):
                nonlocal continue_
                (continue_,) = vars_

            def if_body():
                nonlocal continue_
                continue_ = True

            def else_body():
                nonlocal continue_
                pass
            ag__.if_stmt(((ag__.ld(c) % 2) > 0), if_body, else_body, get_state, set_state, ('contin
ue_',), 1)

            def get_state_1():
                return (s,)

            def set_state_1(vars_):
                nonlocal s
                (s,) = vars_

            def if_body_1():
                nonlocal s
```

```
                s = ag__.ld(s)
                s += c

            def else_body_1():
                nonlocal s
                pass
            ag__.if_stmt(ag__.not_(continue_), if_body_1, else_body_1, get_state_1, set_state_1, ('
s',), 1)
        continue_ = ag__.Undefined('continue_')
        c = ag__.Undefined('c')
        ag__.for_stmt(ag__.ld(items), None, loop_body, get_state_2, set_state_2, ('s',), {'iterate_
names': 'c'})
        try:
            do_return = True
            retval_ = ag__.ld(s)
        except:
            do_return = False
            raise
        return fscope.ret(retval_, do_return)
```

Print statements

Tracing also behaves differently in graph mode. First, here is a function (not decorated with `@tf.function` yet) that prints the value of the input parameter. `f(2)` is called in a for loop 5 times, and then `f(3)` is called.

In [6]:

```
def f(x):
    print("Traced with", x)

for i in range(5):
    f(2)

f(3)
```

```
Traced with 2
Traced with 2
Traced with 2
Traced with 2
Traced with 2
Traced with 3
```

If you were to decorate this function with `@tf.function` and run it, notice that the print statement only appears once for `f(2)` even though it is called in a loop.

In [7]:

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

for i in range(5):
    f(2)

f(3)
```

```
Traced with 2
Traced with 3
```

Now compare `print` to `tf.print`.

- `tf.print` is graph aware and will run as expected in loops.

Try running the same code where `tf.print()` is added in addition to the regular `print`.

- Note how `tf.print` behaves compared to `print` in graph mode.

In [8]:

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

for i in range(5):
    f(2)

f(3)
```

```
Traced with 2
Executed with 2
Executed with 2
Executed with 2
Executed with 2
Executed with 2
Traced with 3
Executed with 3
```

Avoid defining variables inside the function

This function (not decorated yet) defines a tensor `v` and adds the input `x` to it.

Here, it runs fine.

In [9]:

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

print(f(1))
```

```
<tf.Variable 'Variable:0' shape=() dtype=float32, numpy=2.0>
```

Now if you decorate the function with `@tf.function`.

The cell below will throw an error because `tf.Variable` is defined within the function. The graph mode function should only contain operations.

In [10]:

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

print(f(1))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-10-5729586b3383> in <module>
      5     return v
      6
----> 7 print(f(1))

/opt/conda/lib/python3.7/site-packages/tensorflow/python/eager/def_function.py in __call__(self, *
args, **kwds)
    778       else:
    779         compiler = "nonXla"
--> 780         result = self._call(*args, **kwds)
    781
    782       new_tracing_count = self._get_tracing_count()

/opt/conda/lib/python3.7/site-packages/tensorflow/python/eager/def_function.py in _call(self,
```

```
*args, **kwds)
    838         # Lifting succeeded, so variables are initialized and we can run the
    839         # stateless function.
--> 840         return self._stateless_fn(*args, **kwds)
    841     else:
    842       canon_args, canon_kwds = \

/opt/conda/lib/python3.7/site-packages/tensorflow/python/eager/function.py in __call__(self,
*args, **kwargs)
   2826     """Calls a graph function specialized to the inputs."""
   2827     with self._lock:
-> 2828       graph_function, args, kwargs = self._maybe_define_function(args, kwargs)
   2829     return graph_function._filtered_call(args, kwargs)  # pylint: disable=protected-access
   2830

/opt/conda/lib/python3.7/site-packages/tensorflow/python/eager/function.py in
_maybe_define_function(self, args, kwargs)
   3211
   3212       self._function_cache.missed.add(call_context_key)
-> 3213       graph_function = self._create_graph_function(args, kwargs)
   3214       self._function_cache.primary[cache_key] = graph_function
   3215       return graph_function, args, kwargs

/opt/conda/lib/python3.7/site-packages/tensorflow/python/eager/function.py in
_create_graph_function(self, args, kwargs, override_flat_arg_shapes)
   3073             arg_names=arg_names,
   3074             override_flat_arg_shapes=override_flat_arg_shapes,
-> 3075             capture_by_value=self._capture_by_value),
   3076         self._function_attributes,
   3077         function_spec=self.function_spec,

/opt/conda/lib/python3.7/site-packages/tensorflow/python/framework/func_graph.py in
func_graph_from_py_func(name, python_func, args, kwargs, signature, func_graph, autograph,
autograph_options, add_control_dependencies, arg_names, op_return_value, collections,
capture_by_value, override_flat_arg_shapes)
    984         _, original_func = tf_decorator.unwrap(python_func)
    985
--> 986       func_outputs = python_func(*func_args, **func_kwargs)
    987
    988       # invariant: `func_outputs` contains only Tensors, CompositeTensors,

/opt/conda/lib/python3.7/site-packages/tensorflow/python/eager/def_function.py in
wrapped_fn(*args, **kwds)
    598         # __wrapped__ allows AutoGraph to swap in a converted function. We give
    599         # the function a weak reference to itself to avoid a reference cycle.
--> 600         return weak_wrapped_fn().__wrapped__(*args, **kwds)
    601     weak_wrapped_fn = weakref.ref(wrapped_fn)
    602

/opt/conda/lib/python3.7/site-packages/tensorflow/python/framework/func_graph.py in wrapper(*args,
**kwargs)
    971           except Exception as e:  # pylint:disable=broad-except
    972             if hasattr(e, "ag_error_metadata"):
--> 973               raise e.ag_error_metadata.to_exception(e)
    974             else:
    975               raise

ValueError: in user code:

    <ipython-input-10-5729586b3383>:3 f  *
        v = tf.Variable(1.0)
    /opt/conda/lib/python3.7/site-packages/tensorflow/python/ops/variables.py:262 __call__  **
        return cls._variable_v2_call(*args, **kwargs)
    /opt/conda/lib/python3.7/site-packages/tensorflow/python/ops/variables.py:256
_variable_v2_call
        shape=shape)
    /opt/conda/lib/python3.7/site-packages/tensorflow/python/ops/variables.py:67 getter
        return captured_getter(captured_previous, **kwargs)
    /opt/conda/lib/python3.7/site-packages/tensorflow/python/eager/def_function.py:702
invalid_creator_scope
        "tf.function-decorated function tried to create "

    ValueError: tf.function-decorated function tried to create variables on non-first call.
```

To get around the error above, simply move `v = tf.Variable(1.0)` to the top of the cell before the `@tf.function`

decorator.

In [11]:

```
# define the variables outside of the decorated function
v = tf.Variable(1.0)

@tf.function
def f(x):
    return v.assign_add(x)

print(f(5))
```

```
tf.Tensor(6.0, shape=(), dtype=float32)
```

In [ ]:

```

```