TensorFlow Eager Mode

Background and History

Training a Machine Learning model requires tuning the model's many (typically, hundreds to millions) numerical parameters in order to optimize the model's quality.

An almost-obvious approach to numerical optimization is gradient descent (in various forms and refinements). This, then, requires computing gradients efficiently in very high-dimensional spaces.

If we are given some almost-everywhere differentiable real-valued high-dimensional function, such as $f : \mathbb{R}^{1000} \to \mathbb{R}$, a naive way to compute gradients is to estimate slopes by taking a small step in every direction. For this example, this would require at least 1001 evaluations of f for one single gradient, and moreover, we only could expect a gradient accurate to half numerical precision, since the available numerical precision has to be split between accurately representing the function-value and accurately representing the step-size ϵ.

While the computational cost for computing gradients in high dimensions in this way is prohibitive, other methods often are feasible. In particular, there usually is some underlying formula for such high-dimensional problems, which can be differentiated analytically. In the 70s, people realized that it is not only usually possible to hand-implement gradient-functions, but also that careful coding and re-using intermediate quantities typically allows one to obtain a good gradient with a computational cost of only about 4-10 times the cost of computing the original function, even for very high-dimensional functions. Ultimately, Bert Speelpenning realized (and explained in his 1980 PhD dissertation, which also nicely summarizes earlier work on program differentiation) that this can always be done systematically in the following sense:

If we are given a piece of computer code that implements an almost-everywhere-differentiable real-valued function $f : \mathbb{R}^D \to \mathbb{R}$, and we can compute f in such a way that we store all intermediate quantities used in the computation, plus one extra floatin gpoint value per intermediate quantity, then there is an algorithm that takes the code for f as its input and produces the code for a function $g : \mathbb{R}^D \to \mathbb{R}^D$, which computes the gradient of f at a given point, with no more than 8x the computational effort for an evaluation of f, and as accurate as it can reasonably be given limited machine precision.

Here, the upper bound '8x' actually somewhat depends on the machine model (i.e., relative cost of instructions, and cache behaviour). On modern hardware, one typically observes a factor of no more than 5. The general consensus is that the break-even point for Speelpenning's method to beat naive gradient computation in terms of speed is at about 5-6 parameters.

Speelpenning described an algorithm for transforming FORTRAN programs that has become known as 'reverse-mode automatic differentiation (AD)' (we will soon see where that name comes from).

As high-dimensional numerical optimization is obviously economically very relevant across many disciplines, including for example Engineering, there has been considerable pressure on Numerical Optimization to develop practical AD-tools. The www.autodiff.org web page provides a partial list. Basically, these tools provide reverse-mode AD via one of two strategies, which approximately correspond to 'normal TensorFlow' and 'eager mode TensorFlow', but before we look into this, let us review the idea behind reverse-mode AD by means of an example. Why is this relevant, apart from giving us some insight into why Eager Mode was introduced in TensorFlow? Even as TensorFlow mostly is able to do gradient-computations for us, we often find ourselves in situations where we can not get gradients this way, such as:

- For low-level TensorFlow extension Ops which we implement ourselves.
- For problems where using TensorFlow is not appropriate (e.g.: low-level embedded applications).

Let us pick a very simple function that is actually too-low-dimensional for reverse-mode AD to make sense, but otherwise shows all the relevant ideas, except perhaps for branching decisions (which do turn out to not be a problem at all). A nice candidate here would be the function that computes the surface area of a box with side lengths A, B, C, $f_3(A, B, C) = 2 \cdot (AB + BC + CA)$, but unfortunately, this function is just a little bit too simple for our purposes. Its four-dimensional cousin that computes the (3-dimensional) hyper-surface of a 4-dimensional box, works well for explaining the method. An easy way to derive this function is to see how the volume of a 4-dimensional box changes if we add a layer of thickness ϵ on its surface, relative to ϵ. As this will make every side longer by 2ϵ, we have:

$$f_4(A,B,C,D) = \frac{d}{d\epsilon}_{|\epsilon=0}(A+2\epsilon)\cdot(B+2\epsilon)\cdot(C+2\epsilon)\cdot(D+2\epsilon)$$

This gives us the almost-obvious generalization of the 3-d formula to 4-d:

$$f_4(A,B,C,D) = 2(ABC + ACD + ABD + BCD)$$

Let us look at this Python implementation, which has been written with re-use of intermediate values in mind, but does not over-write or otherwise lose any intermediate quantity. All the 'temporary intermediate quantities' are named `t_SOMETHING`, and the function takes a single vector as its input:

```
def f4(side_lengths):
  t_a, t_b, t_c, t_d = side_lengths
  t_ab = t_a * t_b
  t_cd = t_c * t_d
  t_abc = t_ab * t_c
  t_abd = t_ab * t_d
  t_acd = t_a * t_cd
  t_bcd = t_b * t_cd
  t_s = t_abc + t_abd + t_acd + t_bcd
  ret = 2.0 * t_s
  return ret
```

The corresponding gradient-function will need storage for one additional floating point value per each intermediate quantity. Let us introduce these right at the top, initialized to zero. For reasons that will become clear very soon, we want to name the quantity associated with `t_SOMETHING` as `s_SOMETHING`. Also, the first step in computing the gradient is computing the original function's value (which, however, we will not return). So, our partially-complete gradient-function looks as follows:

```
def grad_f4_incomplete0(side_lengths):
  (s_a, s_b, s_c, s_d, s_ab, s_cd,
   s_abc, s_abd, s_acd, s_bcd, s_s) = [0.0] * 11
  t_a, t_b, t_c, t_d = side_lengths
  t_ab = t_a * t_b
  t_cd = t_c * t_d
  t_abc = t_ab * t_c
  t_abd = t_ab * t_d
  t_acd = t_a * t_cd
  t_bcd = t_b * t_cd
  t_s = t_abc + t_abd + t_acd + t_bcd
  ret = 2.0 * t_s
  # return ret  # Do not return this.
  # TODO: Add code to compute the gradient
  raise RuntimeError('TODO: Complete me!')
  # return gradient
```

The basic idea is now as follows: For some intermediate quantity, let us say `t_cd`, we want to answer the question: If, right after the computation of `t_cd`, we added a statement that changes the value of this quantity by some ϵ, how much would this change the final result, relative to ϵ, and ignoring terms of higher than linear order in ϵ. So, we can imagine having this function:

```
def f4_debug(side_lengths, eps_cd=0.0):
  t_a, t_b, t_c, t_d = side_lengths
  t_ab = t_a * t_b
  t_cd = t_c * t_d
  t_cd += eps_cd  # <===
  t_abc = t_ab * t_c
  t_abd = t_ab * t_d
  t_acd = t_a * t_cd
  t_bcd = t_b * t_cd
  t_s = t_abc + t_abd + t_acd + t_bcd
  ret = 2.0 * t_s
  return ret

BOX1_4D = (2.0, 3.0, 4.0, 5.0)
EPS = 1e-7

print("The sensitivity on intermediate quantity 't_cd'\n"
      "for a box with sides %r is (approx): %g" % (
      BOX1_4D,
      (f4_debug(BOX1_4D, eps_cd=EPS) -
       f4_debug(BOX1_4D, eps_cd=-EPS))/(2 * EPS)))

# This prints:
#
# The sensitivity on intermediate quantity 't_cd'
# for a box with sides (2.0, 3.0, 4.0, 5.0) is (approx): 10
```

We want each of the `s_SOMETHING` that we just introduced and initialized to 0 to ultimately hold the sensitivity of the final (returned) result on the intermediate quantity `t_SOMETHING`. For some intermediate quantities, answering that question is easier than for others. It so turns out that it is easiest for `t_s`: The result is twice this value, so if we change this by adding ϵ, the result will obviously change by 2ϵ, so the sensitivity `s_s` is 2. Let us add this in step 1.

```
def grad_f4_incomplete1(side_lengths):
  (s_a, s_b, s_c, s_d, s_ab, s_cd,
   s_abc, s_abd, s_acd, s_bcd, s_s) = [0.0] * 11
  t_a, t_b, t_c, t_d = side_lengths
  t_ab = t_a * t_b
  t_cd = t_c * t_d
  t_abc = t_ab * t_c
  t_abd = t_ab * t_d
  t_acd = t_a * t_cd
  t_bcd = t_b * t_cd
  t_s = t_abc + t_abd + t_acd + t_bcd
  ret = 2.0 * t_s
  # return ret  # Do not return this.
  # Fill in sensitivities.
  s_s += 2.0
  # TODO: Fill in more sensitivities.
  # TODO: Add code to compute the gradient
  raise RuntimeError('TODO: Complete me!')
  # return gradient
```

We have actually written `s_s += 2.0` rather than `s_s = 2` here, since this will turn out to be the more natural form if we think about the procedure that we are developing in terms of a code transformation. Imagine we had this equivalent code instead:

```
ret1 = t_s
ret = ret1 + t_s
return ret
```

This would in the first step transform to:

```
ret1 = t_s
ret = ret1 + t_s
# return ret
# Analyzing the line: "ret = ret1 + t_s"
```

```
s_ret1 += 1.0
s_ts += 1.0
```

This tells us the sensitivity of the result on `t_s` *due to use of this quantity in the computation* `ret = ret1 + t_s` is 1.0. However, `t_s` is used more than once now, and the other use "ret1 = t_s" will also compute to the sensitivity on `t_s`: If we changed `t_s` by ϵ, this would change `ret1` by ϵ, and (since we already know the sensitivity of the result on `ret1` to be 1) this in turn would change the result by ϵ. So, in a 2nd step, we get:

```
ret1 = t_s
ret = ret1 + t_s
# return ret
# Analyzing the line: "ret = ret1 + t_s".
s_ret1 += 1.0
s_ts += 1.0
# Analyzing the line: "ret1 = t_s"
s_ts += s_ret1
```

…and again, we finally end up with `s_ts = 2.0`.

We note that we are here processing the original computations in reverse. This is why it is called "reverse-mode" Automatic Differentiation.

Using this principle, we can work out more sensitivities for our gradient:

```
def grad_f4_incomplete2(side_lengths):
  (s_a, s_b, s_c, s_d, s_ab, s_cd,
   s_abc, s_abd, s_acd, s_bcd, s_s) = [0.0] * 11
  t_a, t_b, t_c, t_d = side_lengths
  t_ab = t_a * t_b
  t_cd = t_c * t_d
  t_abc = t_ab * t_c
  t_abd = t_ab * t_d
  t_acd = t_a * t_cd
  t_bcd = t_b * t_cd
  t_s = t_abc + t_abd + t_acd + t_bcd
  ret = 2.0 * t_s
  # return ret  # Do not return this.
  # Fill in sensitivities.
  # Analyzing the line: "ret = 2.0 * t_s"
  s_s += 2.0
  # Analyzing the line: "t_s = t_abc + t_abd + t_acd + t_bcd"
  s_abc += s_s
  s_abd += s_s
  s_acd += s_s
  s_bcd += s_s
  # Analyzing the line: "t_bcd = t_b * t_cd"
  # ***TODO***
  # TODO: Fill in more sensitivities.
  # TODO: Add code to compute the gradient
  raise RuntimeError('TODO: Complete me!')
  # return gradient
```

What shall we put in the place marked `# ***TODO***`? We know the sensitivity of the result on `t_bcd`. This is `s_bcd`. There will be no further contributions to add to `s_bcd`, since we are going through the code in reverse, and this line is the one that introduces `t_bcd`, so all its uses will be downstream. Now, if we changed `t_b` by ϵ, this would change `t_bcd` by $\mathtt{t}_{\mathtt{cd}} \cdot \epsilon$, and hence the end result by $\mathtt{t}_{\mathtt{cd}} \cdot \mathtt{s}_{\mathtt{bcd}} \cdot \epsilon$, so we get `s_b += t_cd * s_bcd`. Reasoning likewise for the other factor, we also get `s_cd += t_b * s_bcd`:

```
def grad_f4_incomplete3(side_lengths):
  (s_a, s_b, s_c, s_d, s_ab, s_cd,
   s_abc, s_abd, s_acd, s_bcd, s_s) = [0.0] * 11
  t_a, t_b, t_c, t_d = side_lengths
  t_ab = t_a * t_b
  t_cd = t_c * t_d
  t_abc = t_ab * t_c
  t_abd = t_ab * t_d
  t_acd = t_a * t_cd
  t_bcd = t_b * t_cd
  t_s = t_abc + t_abd + t_acd + t_bcd
  ret = 2.0 * t_s
  # return ret  # Do not return this.
  # Fill in sensitivities.
  # Analyzing the line: "ret = 2.0 * t_s"
  s_s += 2.0
  # Analyzing the line: "t_s = t_abc + t_abd + t_acd + t_bcd"
  s_abc += s_s
  s_abd += s_s
  s_acd += s_s
  s_bcd += s_s
  # Analyzing the line: "t_bcd = t_b * t_cd"
  s_b += t_cd * s_bcd
  s_cd += t_b * s_bcd
  # TODO: Fill in more sensitivities.
  # TODO: Add code to compute the gradient
  raise RuntimeError('TODO: Complete me!')
  # return gradient
```

Now that we have seen the general principle, we can finish all the other sensitivities by continuing to go backwards through the code.

```
def grad_f4_incomplete4(side_lengths):
  (s_a, s_b, s_c, s_d, s_ab, s_cd,
   s_abc, s_abd, s_acd, s_bcd, s_s) = [0.0] * 11
  t_a, t_b, t_c, t_d = side_lengths
  t_ab = t_a * t_b
  t_cd = t_c * t_d
  t_abc = t_ab * t_c
  t_abd = t_ab * t_d
  t_acd = t_a * t_cd
  t_bcd = t_b * t_cd
  t_s = t_abc + t_abd + t_acd + t_bcd
  ret = 2.0 * t_s
  # return ret  # Do not return this.
  # Fill in sensitivities.
  # Analyzing the line: "ret = 2.0 * t_s"
  s_s += 2.0
  # Analyzing the line: "t_s = t_abc + t_abd + t_acd + t_bcd"
  s_abc += s_s
  s_abd += s_s
  s_acd += s_s
  s_bcd += s_s
  # Analyzing the line: "t_bcd = t_b * t_cd"
  s_b += t_cd * s_bcd
  s_cd += t_b * s_bcd
  # Analyzing the line: "t_acd = t_a * t_cd"
  s_a += t_cd * s_acd
  s_cd += t_a * s_acd
  # Analyzing the line: "t_abd = t_ab * t_d"
  s_ab += t_d * s_abd
  s_d += t_ab * s_abd
  # Analyzing the line: "t_abc = t_ab * t_c"
  s_ab += t_c * s_abc
  s_c += t_ab * s_abc
  # Analyzing the line: "t_cd = t_c * t_d"
  s_c += t_d * s_cd
  s_d += t_c * s_cd
  # Analyzing the line: "t_ab = t_a * t_b"
  s_a += t_b * s_ab
  s_b += t_a * s_ab
  # TODO: Add code to compute the gradient
  raise RuntimeError('TODO: Complete me!')
  # return gradient
```

Now, we notice that we have the sensitivities `s_a`, `s_b`, `s_c`, `s_d`. What are these? They tell us the rate-of-change for the final result relative to the rate-of-change of the input parameter `a`, `b`, `c`, `d`. But these sensitivities-on-input-parameters are then nothing else but the gradient we are looking for. So we can complete our gradient-function as:

```
def grad_f4(side_lengths):
  (s_a, s_b, s_c, s_d, s_ab, s_cd,
   s_abc, s_abd, s_acd, s_bcd, s_s) = [0.0] * 11
  t_a, t_b, t_c, t_d = side_lengths
  t_ab = t_a * t_b
  t_cd = t_c * t_d
  t_abc = t_ab * t_c
  t_abd = t_ab * t_d
  t_acd = t_a * t_cd
  t_bcd = t_b * t_cd
  t_s = t_abc + t_abd + t_acd + t_bcd
  ret = 2.0 * t_s
  # return ret  # Do not return this.
  # Fill in sensitivities.
  # Analyzing the line: "ret = 2.0 * t_s"
  s_s += 2.0
  # Analyzing the line: "t_s = t_abc + t_abd + t_acd + t_bcd"
  s_abc += s_s
  s_abd += s_s
  s_acd += s_s
  s_bcd += s_s
  # Analyzing the line: "t_bcd = t_b * t_cd"
  s_b += t_cd * s_bcd
  s_cd += t_b * s_bcd
  # Analyzing the line: "t_acd = t_a * t_cd"
  s_a += t_cd * s_acd
  s_cd += t_a * s_acd
  # Analyzing the line: "t_abd = t_ab * t_d"
  s_ab += t_d * s_abd
  s_d += t_ab * s_abd
  # Analyzing the line: "t_abc = t_ab * t_c"
  s_ab += t_c * s_abc
  s_c += t_ab * s_abc
  # Analyzing the line: "t_cd = t_c * t_d"
  s_c += t_d * s_cd
  s_d += t_c * s_cd
  # Analyzing the line: "t_ab = t_a * t_b"
  s_a += t_b * s_ab
  s_b += t_a * s_ab
  gradient = [s_a, s_b, s_c, s_d]
  # May want to return a numpy.array here instead.
  return gradient

# Indeed:
# >>> grad_f4([1.0, 2.0, 3.0, 4.0])
# [52.0, 38.0, 28.0, 22.0]
# >>> f4([1.001, 2.0, 3.0, 4.0]) - f4([1.0, 2.0, 3.0, 4.0])
# 0.0519999999999925  # == 0.001 * 52.0 (approx.)
```

Neither divisions, nor special functions, inner functions, or control structure such as loops or branches pose much of a problem if we remember where we went, and also all the intermediate quantities.

Considering this approach, there are basically three different ways to implement this as an automatic system:

1. Parse the source code into a syntax-tree, and then transform this tree, producing code for the gradient.
2. Introduce some restricted domain-specific language (DSL) for computations that allows to do the above dependency-analysis without having to build some sort of code-transformer that understands the entire language.
3. Wrap up all numerical variables so that all the arithmetic operations on them get recorded on a 'tape'. For the reverse-step, play the tape in reverse.

"Normal" TensorFlow uses approach 2, for different reasons, but most importantly because we want the actual computation to be fast and resource-efficient. So, transforming Python-code is not attractive, as we would much prefer our numerics not to be done with Python on a single CPU core. In particular, we want to have the option to put most of the computation onto GPU hardware, and so the restricted computations-DSL should keep GPU limitations in mind. Approach 1 is, in its purest form, implemented by R6RS-AD (for Scheme).

"Eager" TensorFlow uses approach 3. This method is by no means new and is also what packages such as ADOL-C(C/C++), Adept(C/C++), or autograd(Python) use. The advantage of using a tape is conceptual simplicity, but this comes with some efficiency cost. In particular, since all computations get recorded, the tape needs more space (and bookkeeping) than 'remembering one floating point value for every intermediate quantity'. How does such a 'tape' look like? We can roughly imagine it as containing all the arithmetic computations that we have gone through when evaluating a function in a form like this:

```
value8 = value1 + value3
value9 = value8 - value2
value10 = 2 * value8
value11 = -value9
value12 = value10 + value11
value13 = sin(value12)
(...)
```

Quoting from Speelpenning's dissertation on previous work done by Joos at ETHZ in 1976 (section 2.2) that essentially introduced this idea:

"The basic idea is again quite intuitive: for any given value of x, the program goes through a definite (...) sequence of assignment statements. That sequence of assignment statements, which might have been obtained from an execution trace, defines a straight-line program. For the particular value of x, the straight-line program would produce the same value for y as the original program. Moreover, we may reasonably expect that both programs produce the same value for y in some very small neighborhood of the point x. If this turns out to be true, we may differentiate the straight-line program. We know how to differentiate a straight program from Warner's work, and we know that his method does not radically change the structure of the original program; rather it is a mild expansion of it. (...)"

Let us have a look at the most basic example how to get an automatic gradient in Python via the `autograd` package. This is good to know, as TF-Eager is partially inspired by autograd, and also since it can be useful to have some basic automated differentiation available in Python without having to install TensorFlow. Finally, there are some rare situations where one would like to extend TensorFlow in a way that requires writing one's own gradient-functions, and `autograd` sometimes can help with that (at least during the development phase, for example for gradient-debugging).

```
# Autograd is available via: 'pip install autograd'.

import autograd

def f4(side_lengths):
  t_a, t_b, t_c, t_d = side_lengths
  t_ab = t_a * t_b
  t_cd = t_c * t_d
  t_abc = t_ab * t_c
  t_abd = t_ab * t_d
  t_acd = t_a * t_cd
  t_bcd = t_b * t_cd
  t_s = t_abc + t_abd + t_acd + t_bcd
  ret = 2.0 * t_s
  return ret

grad_f4 = autograd.grad(f4)

# >>> grad_f4([1.0, 2.0, 3.0, 4.0])
# [array(52.), array(38.), array(28.), array(22.)]
#
# Strangely, this uses numpy rank-0 arrays/tensors to represent numbers,
# but these just behave like float in most ways (and can be converted
# to float).
```

In general, using a 'tape' that records all arithmetic operations may in practice not be as bad as it sounds, since we can often record SIMD-like linear algebra operations as collective operations, rather than as many many individual number-plus-number additions and number-times-number multiplications. So, we can imagine some entries on our tape to be vector-arithmetic like operations.

Using TensorFlow Eager Mode

Basically, TF-Eager tensors can mostly be used like numpy-arrays, and we can always extract a numpy array from a TF-Eager tensor. The main new thing they offer on top of numpy-arrays is automatic computations of gradients via 'tapes', and also the ability to move parts of a calculation to specialized hardware, such as a GPU. The code below shows basic use of TF-Eager.

```
from __future__ import print_function

import numpy
import tensorflow as tf

tfe = tf.contrib.eager  # Abbreviation.


tf.enable_eager_execution()

print('TF Eager Execution is:',
      'Enabled' if tf.executing_eagerly() else 'Disabled')


def f4(abcd):
    """Hyperbox-hypersurface-area."""
    return 2.0 * sum(abcd[i] * abcd[j] * abcd[k]
                     for (i, j, k) in [(0, 1, 2), (0, 1, 3),
                                       (0, 2, 3), (1, 2, 3)])


# Thanks to Python-overloading, our implementation of `f4`
# does not even actually care what sort of sequence we feed it.
print(f4([1.0, 2.0, 3.0, 4.0]), f4(numpy.array([2.0, 3.0, 4.0, 5.0])))

# This would not work with eager mode enabled:
# xs = tf.Variable([3.0, 4.0, 5.0, 6.0])
#
# We need a tf-eager variable:
xs = tfe.Variable([3.0, 4.0, 5.0, 6.0])

# tf-eager tensors 'know their numerical value'.
# When printing them (at the prompt, or via `print`), we see:
# >>> xs
# <tf.Variable 'Variable:0' shape=(4,) dtype=float32,
#  numpy=array([3., 4., 5., 6.], dtype=float32)>
#
# We can get at the underlying numpy-array with the .numpy() method:
# >>> xs.numpy()
# array([3., 4., 5., 6.], dtype=float32)

# Our 'f4' method also accepts tf-eager tensors:
# >>> print(f4(xs))
# tf.Tensor(684.0, shape=(), dtype=float32)

# A rank-0 tensor has 1 entry.
# >>> print(f4(xs).numpy())
# 684.0

# In most ways, tf-eager tensors behave equivalent to numpy arrays.
# >>> print(2*xs)
# tf.Tensor([ 6.  8. 10. 12.], shape=(4,), dtype=float32)

# What is the difference between tf-eager tensors and numpy arrays?
# We can ask TensorFlow to give us a gradient, but we need to explicitly
# ask for a tape.

# Let us use the point for which we know the gradient well:
p0 = tfe.Variable([1.0, 2.0, 3.0, 4.0])

# This `with` context-construction is interesting:
# The tape, which we bind to the variable 'tape', records a computation-tape
# for all the TensorFlow computations that happen during the execution
# of its body.
with tf.GradientTape() as tape:
    hypersurface = f4(p0)

# After the body finished, we then keep referring to this `tape`-variable.
# This is very different from common uses of `with` in Python, such as:
#
# with open(filename, 'w') as h:
#   h.write(data)
```

```
#
# After this block, we no longer expect 'h' to have some valid content.
# We may even have thought that there might be an implicit `del h`
# after the block (there is not). We only use `h` inside the body.
# The `with tf.GradientTape()` actually does use this scope-extrusion,
# and we want to avoid it in the future.

# The tape's .gradient() method gives us the list of gradient-coefficients
# (=="sensitivities") for all the tensors listed in the 2nd argument
# to tape.gradient():
gradients_at_p0 = tape.gradient(hypersurface, [p0])

print('Hypersurface:', hypersurface.numpy(),
      'Gradient:', gradients_at_p0[0].numpy())

# For a single tensor with respect to which we want to compute
# the gradient, we can use:
#
# gradient_at_p0 = tape.gradient(hypersurface, p0)
#
# However, trying to execute the line above right here will fail,
# as every recorded tape can only be used once(!)

# Of course, we can do more complex computations.
# Let us compute the hypersurface area of a box plus the area
# of a box twice-as-large in every dimension:

# This shows a variant for writing the with-context that perhaps
# makes it clearer what is used inside and what is used outside
# of the with-context:
tape2 = tf.GradientTape()
with tape2:
    hypersurface1 = f4(p0)
    hypersurface2 = f4(2 * p0)
    total_hypersurface = hypersurface1 + hypersurface2

# Showing the single-tensor-as-2nd-arg form of tape.gradient here.
print('Total Hypersurface:', total_hypersurface.numpy(),
      'Gradient:', tape2.gradient(total_hypersurface, p0).numpy())

# We can use TF-Eager tensors pretty much like NumPy arrays with an extra layer
# of wrapping around them (so TF can get gradients via tapes). In particular,
# we can freely create them and have them collected, just like NumPy arrays.
# The practical Notebooks will show this.
```