Introduction to

Thomas Viehmann, MathInf GmbH

3rd Advanced Course on Data Science and Machine Learning (https://acdl2020.icas.xyz/)

In [1]:
```
import torch
```

Hello

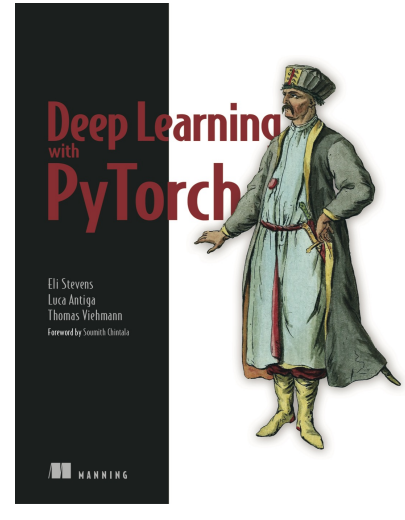
I'm Thomas Viehmann.

I founded MathInf GmbH, a boutique research and development consultancy & PyTorch training company

I'm a PyTorch core developer, and with ~140 patches one of the most prolific independent contributors to PyTorch

With Eli Stevens and Luca Antiga, I wrote Deep Learning with PyTorch (https://pytorch.org/deep-learning-with-pytorch), currently available for free download from PyTorch.org.

Background: PhD in pen and paper Mathematics, been an Actuary & Consultant for 10 years, Debian Developer emeritus



Plan

Conundrum: Introduction to PyTorch @ Advanced Course on Data Science and Machine Learning

We take a tour of PyTorch, nominally from 0, but we put best practices and pitfalls into the spotlight.

- Tensors
- Functions
- Autograd
- Models
- Optimizers / Data
- Excursion: Efficiency and Benchmarking
- Beyond Python / JIT
- All of this is in Jupyter: https://github.com/t-vi/acdl2020/ (https://github.com/t-vi/acdl2020/) has the notebook

Your favourite topic?

PyTorch is ... Tensors

At the heart of PyTorch is the Tensor data type. It is a multidimensional array.

When we speak of the dimension of tensors, it is the number of axes (or, in PyTorch, dimensions) - so dimension is in the sense of arrays, not the mathematical one.

Tensors can be 0d (i.e. scalars - single numbers), 1d (vectors), 2d (matrices), 3d...

In [3]:
```
t = torch.tensor([[0.7071, 1.4142, 1.6180],
                  [0.8346, 2.7183, 3.1416]])
t
```

Out[3]:
```
tensor([[0.7071, 1.4142, 1.6180],
        [0.8346, 2.7183, 3.1416]])
```

Tensors are arrays

Tensors have a shape (the size of each dimension):

In [4]:
```
t.dim(), t.shape
```

Out[4]:
```
(2, torch.Size([2, 3]))
```

Indexing - also with assignments - works as expected. Use .item() to get numbers from tensors.

In [5]:
```
t[1, 2]
```

Out[5]:
```
tensor(3.1416)
```

In [6]:
```
t[1, 2] = 3.141592653589793
t, t[1, 2].item()
```

Out[6]:
```
(tensor([[0.7071, 1.4142, 1.6180],
         [0.8346, 2.7183, 3.1416]]),
 3.1415927410125732)
```

"Advanced indexing" with masks also works

In [7]:
```
t[t < 1] = 0
t
```

Out[7]:
```
tensor([[0.0000, 1.4142, 1.6180],
        [0.0000, 2.7183, 3.1416]])
```

so does slicing

In [8]:
```
t[:, 1:], t[:, ::2], t[1, 2, None]
```

Out[8]:
```
(tensor([[1.4142, 1.6180],
         [2.7183, 3.1416]]),
 tensor([[0.0000, 1.6180],
         [0.0000, 3.1416]]),
 tensor([3.1416]))
```

Tensors always hold elements (numbers) of a given scalar type, the "data type" or dtype. For PyTorch (with its GPU focus), `float32` is perhaps the most common dtype. Apropos GPU, they do have a device.

In [9]:
```
t.dtype, t.device
```

Out[9]:
```
(torch.float32, device(type='cpu'))
```

We can move between devices and convert between dtypes.

In [10]:
```
t.to(dtype=torch.float64, device="cuda")  # if we have a GPU
```

Out[10]:
```
tensor([[0.0000, 1.4142, 1.6180],
        [0.0000, 2.7183, 3.1416]], device='cuda:0', dtype=torch.float64)
```

Tensors - a scenic view of memory blobs

Internally, Tensors are stored as a blob of memory

In [11]:
```
t.storage()
```

Out[11]:
```
 0.0
 1.414199948310852
 1.6180000305175781
 0.0
 2.7183001041412354
 3.1415927410125732
[torch.FloatStorage of size 6]
```

In [12]:
```
t.stride(), t.shape, t.storage_offset(), t.data_ptr()
```

Out[12]:
```
((3, 1), torch.Size([2, 3]), 0, 78994688)
```

In [13]:
```
t
```

Out[13]:
```
tensor([[0.0000, 1.4142, 1.6180],
        [0.0000, 2.7183, 3.1416]])
```

Indexing and Slicing (and taking Diagonals, too) do *not* copy, but create views:

In [14]:
```
v = t[:, 2:]
```

In [15]:
```
v.stride(), v.shape, v.storage_offset(), v.data_ptr(), v.storage().data_ptr()
```

Out[15]:
```
((3, 1), torch.Size([2, 1]), 2, 78994696, 78994688)
```

In [16]:
```
v, v.storage()
```

Out[16]:
```
(tensor([[1.6180],
         [3.1416]]),
  0.0
  1.414199948310852
  1.6180000305175781
  0.0
  2.7183001041412354
  3.1415927410125732
 [torch.FloatStorage of size 6])
```

(`data_ptr` already subsumes the storage offset...)

Generalized tensors are tensors, too.

- These were strided tensors
- There are other Tensor types: Sparse, on TPU,...
- Quantized is a bit special, but similar in spirit.

In [17]:
```
t.layout
```

Out[17]:
```
torch.strided
```

In [18]:
```
s = torch.sparse_coo_tensor(torch.arange(10)[None], torch.randn(10))
s, s.layout
```

Out[18]:
```
(tensor(indices=tensor([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]),
        values=tensor([-1.9326, -0.6894, -0.6399,  0.6636, -0.0204, -0.6179,
                       -0.7636,  1.5748, -1.0660,  1.0288]),
        size=(10,), nnz=10, layout=torch.sparse_coo),
 torch.sparse_coo)
```

Where to get Tensors?

PyTorch has ~90 factory functions:

- torch.tensor, torch.sparse_coo_tensor
- torch.zeros, ones, randn, ...
- torch.zeros_like, ones_like, randn_like, ...

Conversion to and from Numpy works without copying (there also is `torch.utils.dlpack` for a similar effect with GPU tensors)

In [19]:
```
a = t.numpy()
t2 = torch.from_numpy(a)
t2.data_ptr() == t.data_ptr()
```

Out[19]:
```
True
```

Careful: Numpy uses a dtype of float64 by default, PyTorch float32 (as you typically want that for GPUs)

Sometimes you see FloatTensor, .data, ... in old code or copied from old code. Don't do that.

There also are `torch.utils.dlpack.to_dlpack / from_dlpack` for interchange with GPU tensors.

Saving and loading tensors

Saving and loading tensors and dicts, lists of them (and other types) can be done though `torch.save` and `torch.load`. Optionally, you can specify where the tensors should be loaded to (the default is to put GPU tensors back on the GPU).

In [20]:
```
torch.save(t, 'my-tensor.pt')
torch.load('my-tensor.pt', map_location="cpu")
```

Out[20]:
```
tensor([[0.0000, 1.4142, 1.6180],
        [0.0000, 2.7183, 3.1416]])
```

Style advice: Do use `.pt` as an extension, not `.pth` or anything else (`pth` collides with Python "path" files).

PyTorch is ... operations on tensors

PyTorch mostly follows numpy in naming. Most things are available as methods (`t.tanh()`) and functions (`torch.tanh(t)`). More neural-network specific things are in `torch.nn.functional` .

I'm not going to list them, but generally we follow NumPy naming, the documentation (https://pytorch.org/docs/) has all the details.

In [21]:
```
t.tanh()
```

Out[21]:
```
tensor([[0.0000, 0.8884, 0.9243],
        [0.0000, 0.9913, 0.9963]])
```

Inplace operations are signaled by a trailing underscore `_` .

In [22]:
```
z = torch.randn(5, 5)
z.tanh_()
z
```

Out[22]:
```
tensor([[-0.1275,  0.4194,  0.7845,  0.6627,  0.3954],
        [ 0.6835, -0.7321,  0.9535, -0.7996,  0.0952],
        [-0.7956, -0.3266, -0.9758,  0.9814, -0.0405],
        [-0.4738,  0.9680,  0.0144, -0.5584, -0.2566],
        [ 0.0568,  0.9590,  0.5114, -0.2172,  0.7584]])
```

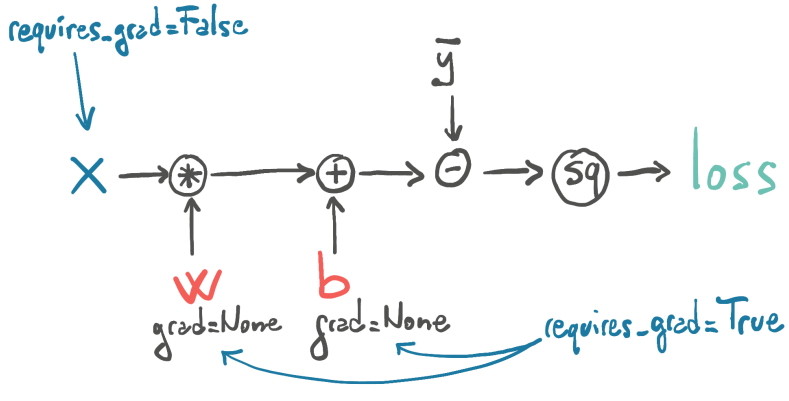
The `torch.nn.functional` namespace (sometimes mapped to `F.`) contains functions that are more neural-network related.

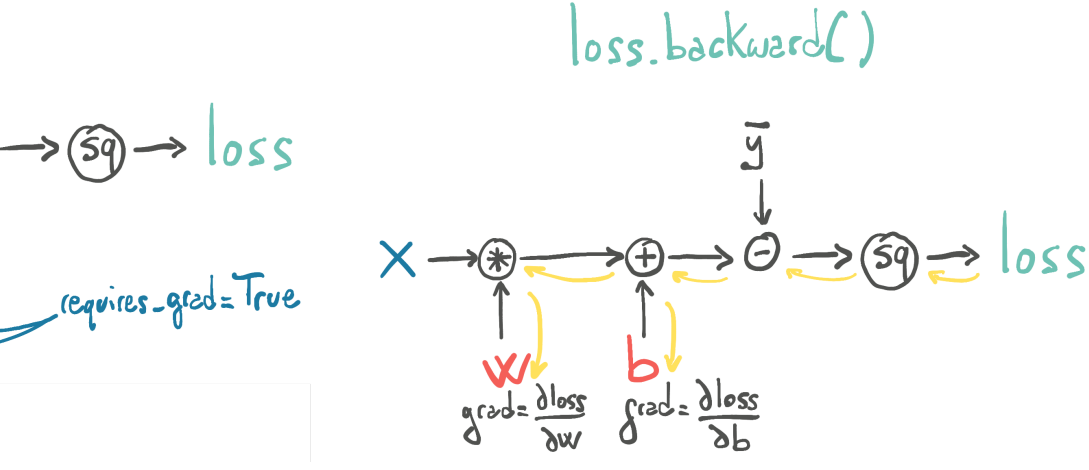
In [23]:
```
w = torch.randn(10, 5)
i = torch.randn(2, 5)
torch.nn.functional.linear(i, w)
```

Out[23]:
```
tensor([[-0.3343,  2.0150,  0.3340,  1.2196,  5.7402, -1.3553,  3.2065, -3.189
4,
         -2.1207,  0.4356],
        [ 0.9437,  0.3812, -0.2400,  0.7666,  1.9859, -1.8378,  0.5127,  0.111
9,
          1.2590, -0.3914]])
```

PyTorch is ... autograd

PyTorch has a backpropagation-based autograd mechanism. It computes the gradient of *scalar* functions (and, by proxy of Jacobian-Vector products) at the point where is was evaluated. (Image from Deep Learning with PyTorch)

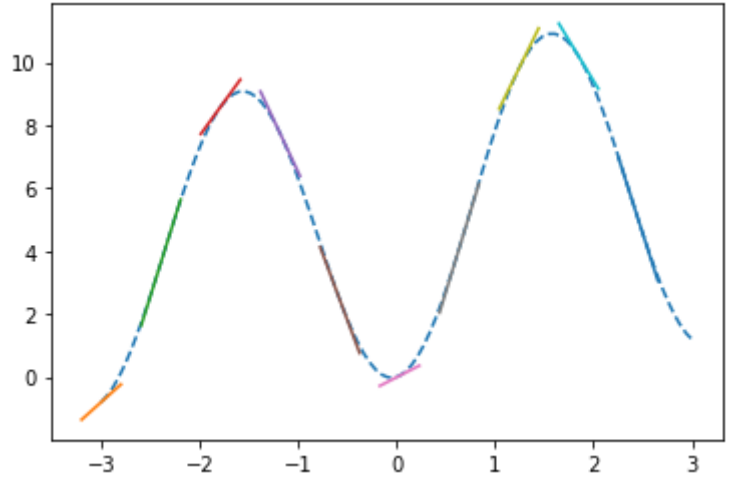




Autograd in action

In [24]:
```
x = torch.linspace(-3, 3, requires_grad=True)
f = 10 * torch.sin(x)**2 + torch.tanh(x)
gr, = torch.autograd.grad(f.sum(), [x]) # gr always returns a list of gradients
```

In [25]:
```
pyplot.plot(x.detach().numpy(), f.detach().numpy(), '--')
delta = 0.2
for xi, fi, gi in zip(x[::10].tolist(), f[::10].tolist(), gr[::10].tolist()):
    pyplot.plot([xi-delta, xi+delta], [fi-delta*gi, fi+delta*gi]) # tangents at
every 10th point
```



A mickey mouse example of optimizing a model (actually 100 times, starting from random initializations).

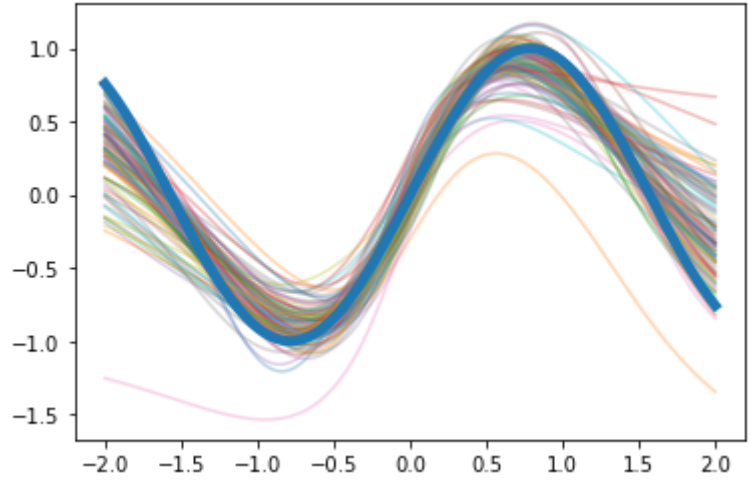
- We have 100 mini batches à 10 items.
- Each is, for a (normally) random x the value $f(x) = \sin(2x)$.
- This we approximate by a 2-layer network with 10 hidden units. Our loss is the squared euclidean distance.
- By trying various starting values, we represent the uncertainty coming from the random initializaiton of the weights (of course, we would expect that to go away with more iterations).

In [26]:
```
final_preds = []
for no in range(100):   # 100 experiments
    w1 = torch.randn(1, 10, requires_grad=True)   # initialise random paramters
    b1 = torch.randn(10, requires_grad=True)
    w2 = torch.randn(10, 1, requires_grad=True)
    b2 = torch.randn(1, requires_grad=True)
    params = [w1, b1, w2, b2]

    for i in range(100):  # fitting 100 minibatches
        x = torch.randn(10, 1)   # input value
        target = torch.sin(2 * x)   # target value (usually from dataset)

        pred = torch.tanh(x @ w1 + b1) @ w2 + b2   # our model

        loss = ((pred - target) ** 2).sum()

        grads = torch.autograd.grad(loss, params)

        for p, g in zip(params, grads):
            with torch.no_grad():
                p -= 1e-2 * g

    with torch.no_grad(): # no gradients here...
        x0 = torch.linspace(-2, 2)
        final_preds.append(torch.tanh(x0[:, None] @ w1 + b1) @ w2 + b2)

final_preds = torch.cat(final_preds, dim=1)
final_preds.shape
```

Out[26]:
```
torch.Size([100, 100])
```

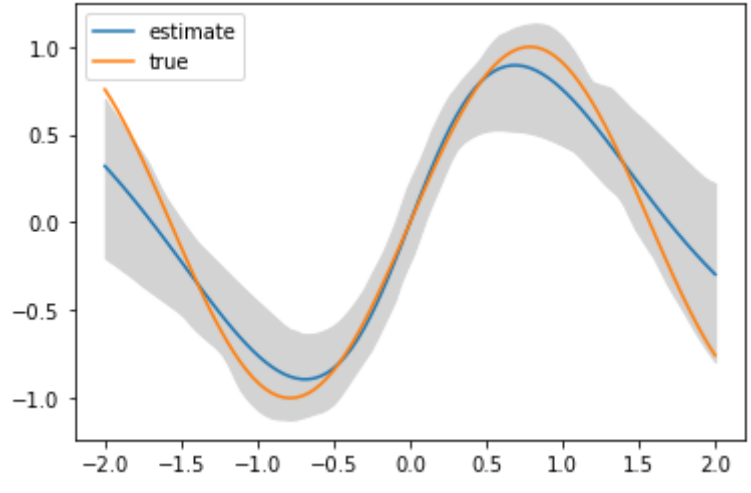
In [27]:
```
x0 = torch.linspace(-2, 2)
pyplot.plot(x0, final_preds, alpha=0.3)
pyplot.plot(x0, torch.sin(2 * x0), linewidth=5)
```

Out[27]:
```
[<matplotlib.lines.Line2D at 0x7f65d4385fd0>]
```



In [28]:
```
pyplot.plot(x0, final_preds.mean(1), label='estimate')
pyplot.plot(x0, torch.sin(2 * x0), label='true')
pyplot.fill_between(x0, numpy.percentile(final_preds, 2.5, axis=1), numpy.perce
ntile(final_preds, 97.5, axis=1),
                    color='lightgray')
pyplot.legend()
```

Out[28]:
```
<matplotlib.legend.Legend at 0x7f65d41d1dc0>
```



Leaves, intermediates, inplace and ...

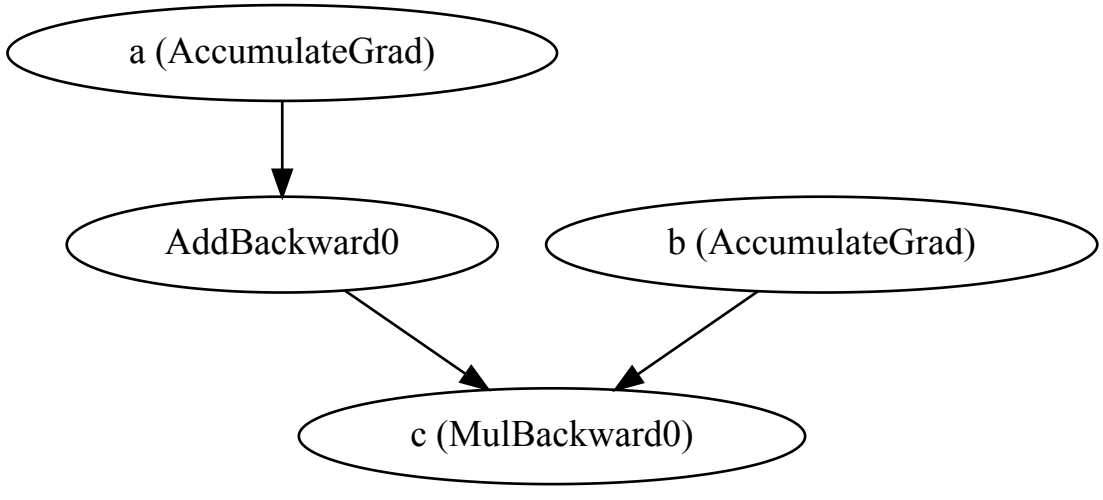
There roughly are these types of tensors from an autograd perspective:

- "Plain" tensors, not requiring gradients (in particular all integral tensors).
- Leaves are the initial tensors requiring grad, from the factory functions or by taking a non-grad-requiring tensor and saying `t.requires_grad_()`.
- Computed tensors where an input requires gradients will, in general, require gradient.
- "views", tensors sharing memory, are special.

Let's look at this in some detail.

In [30]:
```
a = torch.randn(5, requires_grad=True)
b = torch.randn(5, requires_grad=True)
c = (a + 1) * b
ptgraph(c)
```

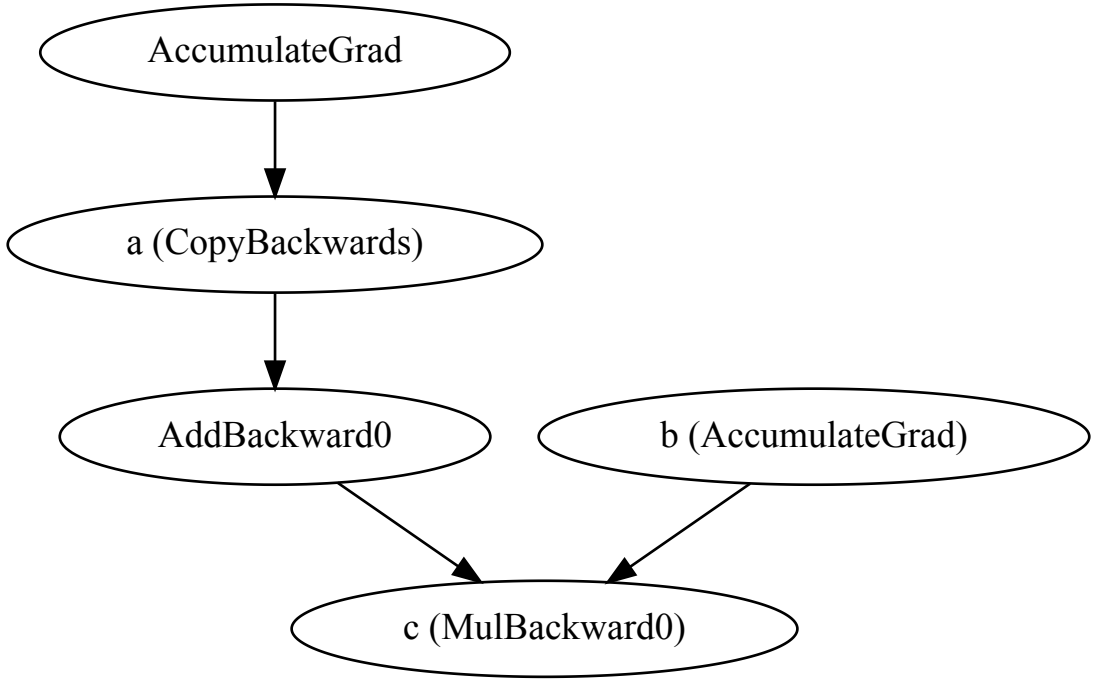
Out[30]:



A common mistake is to accidentally throw away the true leaf. Note the difference between `a` and `b`.

In [31]:
```
a = torch.randn(5, requires_grad=True).cuda()
b = torch.randn(5, requires_grad=True, device='cuda')
c = (a + 1) * b
ptgraph(c)
```

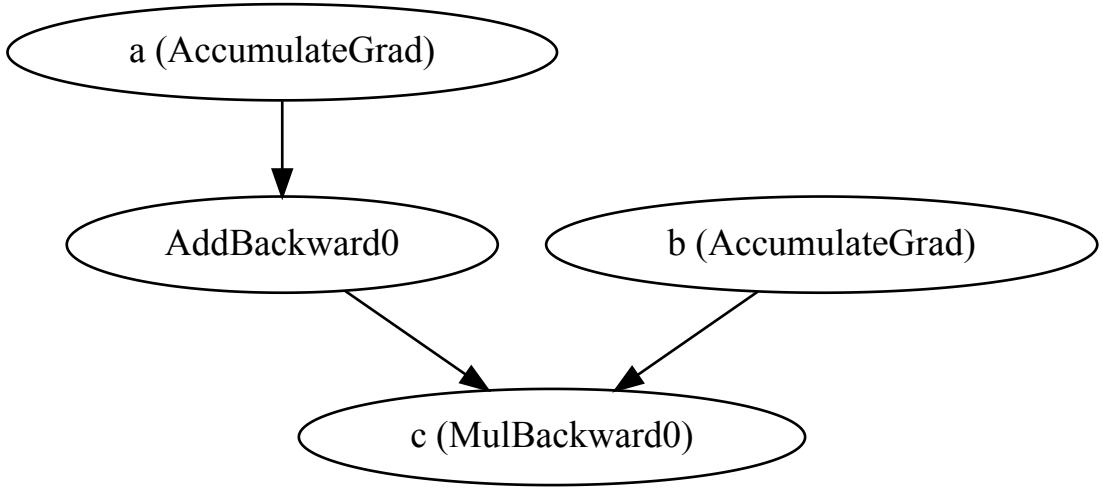
Out[31]:



This would be a repaired version (but, if you can, working directly on the GPU is more efficient).

In [32]:
```
a = torch.randn(5).cuda().requires_grad_()
b = torch.randn(5, requires_grad=True, device='cuda')
c = (a + 1) * b
ptgraph(c)
```

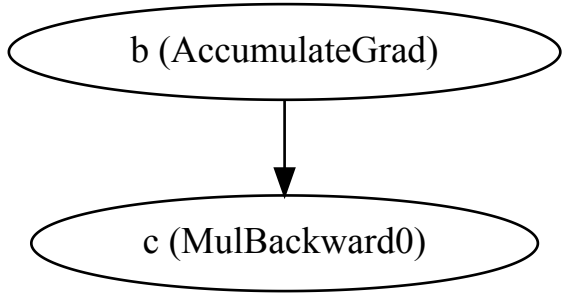
Out[32]:



You can cut things from autograd with `detach()` (and `detach_()`)

In [33]:
```
a = torch.randn(5, requires_grad=True).cuda()
b = torch.randn(5, requires_grad=True, device='cuda')
c = (a + 1).detach() * b
ptgraph(c)
```

Out[33]:



Inplace with autograd can be tricky

In [34]:
```
a = torch.randn(5, requires_grad=True)
b1 = a + 1
b2 = b1 + 1

b1 += 1

torch.autograd.grad(b2.sum(), a) # works
```

Out[34]:
```
(tensor([1., 1., 1., 1., 1.]),)
```

In [35]:
```
a = torch.randn(5, requires_grad=True)
b1 = a + 1
b2 = b1 ** 2
b1 += 1

torch.autograd.grad(b2.sum(), a) # this would need the original b1
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-35-950f59f473fd> in <module>
      4 b1 += 1
      5
----> 6 torch.autograd.grad(b2.sum(), a) # this would need the original b1

/usr/local/lib/python3.8/dist-packages/torch/autograd/__init__.py in grad(outp
uts, inputs, grad_outputs, retain_graph, create_graph, only_inputs, allow_unus
ed)
    188         retain_graph = create_graph
    189
--> 190     return Variable._execution_engine.run_backward(
    191         outputs, grad_outputs, retain_graph, create_graph,
    192         inputs, allow_unused)

RuntimeError: one of the variables needed for gradient computation has been mo
dified by an inplace operation: [torch.FloatTensor [5]], which is output 0 of
 AddBackward0, is at version 1; expected version 0 instead. Hint: enable anoma
ly detection to find the operation that failed to compute its gradient, with t
orch.autograd.set_detect_anomaly(True).
```

We did not get a TraceBack in Jupyter before PyTorch 1.7 (PR #41116), but now we get too much (the crucial `b2 = b1 ** 2` is there, but off-screen)...

In [36]:
```
with torch.autograd.detect_anomaly():
    a = torch.randn(5, requires_grad=True)
    b1 = a + 1
    b2 = b1 ** 2
    b1 += 1

    torch.autograd.grad(b2.sum(), a) # this would need the original b1
```

```
<ipython-input-36-fb1eaba42148>:1: UserWarning: Anomaly Detection has been ena
bled. This mode will increase the runtime and should only be enabled for debug
ging.
  with torch.autograd.detect_anomaly():
/usr/local/lib/python3.8/dist-packages/torch/autograd/__init__.py:190: UserWar
ning: Error detected in PowBackward0. Traceback of forward call that caused th
e error:
  File "/usr/lib/python3.8/runpy.py", line 194, in _run_module_as_main
    return _run_code(code, main_globals, None,
  File "/usr/lib/python3.8/runpy.py", line 87, in _run_code
    exec(code, run_globals)
  File "/usr/lib/python3/dist-packages/ipykernel_launcher.py", line 16, in <mo
dule>
    app.launch_new_instance()
  File "/usr/lib/python3/dist-packages/traitlets/config/application.py", line
664, in launch_instance
    app.start()
  File "/usr/lib/python3/dist-packages/ipykernel/kernelapp.py", line 597, in s
tart
    self.io_loop.start()
  File "/usr/lib/python3/dist-packages/tornado/platform/asyncio.py", line 149,
in start
    self.asyncio_loop.run_forever()
  File "/usr/lib/python3.8/asyncio/base_events.py", line 570, in run_forever
    self._run_once()
  File "/usr/lib/python3.8/asyncio/base_events.py", line 1859, in _run_once
    handle._run()
  File "/usr/lib/python3.8/asyncio/events.py", line 81, in _run
    self._context.run(self._callback, *self._args)
```

```
  File "/usr/lib/python3/dist-packages/tornado/ioloop.py", line 690, in <lambd
a>
    lambda f: self._run_callback(functools.partial(callback, future))
  File "/usr/lib/python3/dist-packages/tornado/ioloop.py", line 743, in _run_c
allback
    ret = callback()
  File "/usr/lib/python3/dist-packages/tornado/gen.py", line 787, in inner
    self.run()
  File "/usr/lib/python3/dist-packages/tornado/gen.py", line 748, in run
    yielded = self.gen.send(value)
  File "/usr/lib/python3/dist-packages/ipykernel/kernelbase.py", line 381, in
dispatch_queue
    yield self.process_one()
  File "/usr/lib/python3/dist-packages/tornado/gen.py", line 225, in wrapper
    runner = Runner(result, future, yielded)
  File "/usr/lib/python3/dist-packages/tornado/gen.py", line 714, in __init__
    self.run()
  File "/usr/lib/python3/dist-packages/tornado/gen.py", line 748, in run
    yielded = self.gen.send(value)
  File "/usr/lib/python3/dist-packages/ipykernel/kernelbase.py", line 365, in
process_one
    yield gen.maybe_future(dispatch(*args))
  File "/usr/lib/python3/dist-packages/tornado/gen.py", line 209, in wrapper
    yielded = next(result)
  File "/usr/lib/python3/dist-packages/ipykernel/kernelbase.py", line 268, in
dispatch_shell
    yield gen.maybe_future(handler(stream, idents, msg))
  File "/usr/lib/python3/dist-packages/tornado/gen.py", line 209, in wrapper
    yielded = next(result)
  File "/usr/lib/python3/dist-packages/ipykernel/kernelbase.py", line 543, in
execute_request
    self.do_execute(
  File "/usr/lib/python3/dist-packages/tornado/gen.py", line 209, in wrapper
    yielded = next(result)
  File "/usr/lib/python3/dist-packages/ipykernel/ipkernel.py", line 300, in do
_execute
    res = shell.run_cell(code, store_history=store_history, silent=silent)
  File "/usr/lib/python3/dist-packages/ipykernel/zmqshell.py", line 536, in ru
```

```
n_cell
    return super(ZMQInteractiveShell, self).run_cell(*args, **kwargs)
  File "/usr/lib/python3/dist-packages/IPython/core/interactiveshell.py", line
2857, in run_cell
    result = self._run_cell(
  File "/usr/lib/python3/dist-packages/IPython/core/interactiveshell.py", line
2886, in _run_cell
    return runner(coro)
  File "/usr/lib/python3/dist-packages/IPython/core/async_helpers.py", line 6
8, in _pseudo_sync_runner
    coro.send(None)
  File "/usr/lib/python3/dist-packages/IPython/core/interactiveshell.py", line
3062, in run_cell_async
    has_raised = await self.run_ast_nodes(code_ast.body, cell_name,
  File "/usr/lib/python3/dist-packages/IPython/core/interactiveshell.py", line
3254, in run_ast_nodes
    if (await self.run_code(code, result,  async_=asy)):
  File "/usr/lib/python3/dist-packages/IPython/core/interactiveshell.py", line
3331, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-36-fb1eaba42148>", line 4, in <module>
    b2 = b1 ** 2
 (Triggered internally at  ../torch/csrc/autograd/python_anomaly_mode.cpp:58.)
  return Variable._execution_engine.run_backward(

---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-36-fb1eaba42148> in <module>
      5     b1 += 1
      6
----> 7     torch.autograd.grad(b2.sum(), a) # this would need the original b1

/usr/local/lib/python3.8/dist-packages/torch/autograd/__init__.py in grad(outp
uts, inputs, grad_outputs, retain_graph, create_graph, only_inputs, allow_unus
ed)
    188         retain_graph = create_graph
    189
--> 190     return Variable._execution_engine.run_backward(
```

There are some opportunities where inplace is actually very useful. One of my faviourite tricks is to copy things into a preallocated tensor.

In [37]:
```
a = torch.randn(5, requires_grad=True)
b = torch.randn(5, requires_grad=True)
c = torch.zeros(4, 7)  # padded torch.stack([a,b])
c[1, 1:-1] = a
c[2, 1:-1] = b
c
```

Out[37]:
```
tensor([[ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0539,  0.2435, -0.3037, -0.2411, -0.0319,  0.0000],
        [ 0.0000,  0.7659,  1.2740,  0.2249, -1.1687, -1.7693,  0.0000],
        [ 0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000,  0.0000]],
       grad_fn=<CopySlices>)
```

In [38]:
```
gr_a, gr_b = torch.autograd.grad(c.sum(), [a, b])
assert (gr_a -1).abs().max().item() < 1e-6 and (gr_b -1).abs().max().item() < 1
e-6  # float and "==" don't mix well
```

Inplace Summary

There are two main situations where inplace operations cannot be used:

- When operating on leaf tensors, the leaf tensor would be moved into the graph, which would be bad.
- When the operation before the inplace wants to have its result to compute the backward. Whether this is the case is not easy to tell from the outside, unfortunately.

As a corollary, you should avoid using inplace on the inputs of your re-usable module lest in the future use could be in one of the two situations.

Later: You might also be less eager to use inplace when planning to use the JIT, as it will fuse pointwise non-inplace operations like ReLU if there are several in a row.

For validation, inference or other things where the *end user* decides she does not want gradients, there is the `torch.no_grad` context manager.

We'll use it below to evaluate models after training.

- If you need to use `torch.no_grad()` somewhere where it isn't because you're evaluating something that's written for training, you should ask yourself if you're doing it wrong.
- Using item and detach for things to keep around longer than the next backward is generally a good idea (e.g. when you record loss history, statistics, ...), but be careful to not ruin your graph. (Targeted detach is good in nn.Module subclass code, with torch.no_grad() should be needed very rarely.)

It is not true that tensors created in `torch.no_grad` never require grad (as often views are special):

In [39]:
```
a = torch.randn(5, 5, requires_grad=True)

with torch.no_grad():
    b = a * 2
    c = a[:2, :2]

b, c
```

Out[39]:
```
(tensor([[ 4.5442,  2.7897, -2.8073,  1.9178,  2.6865],
         [-0.4001, -1.4518,  2.4382,  4.3330,  2.4355],
         [-1.9214, -0.5341,  2.4665, -3.8517,  3.4939],
         [ 0.3584,  1.1731,  1.1691, -3.1120, -0.5954],
         [ 0.7143, -0.2832, -1.8675, -3.3314, -0.5892]]),
 tensor([[ 2.2721,  1.3948],
         [-0.2001, -0.7259]], requires_grad=True))
```

Second derivatives

You cannot get Hessians, but you can get Hessian vector products or really derivatives of any scalar function of the gradient.

In [40]:
```
a = torch.arange(5., requires_grad=True)
b = a ** 4
gr, = torch.autograd.grad(b.sum(), [a], create_graph=True)
hvp, = torch.autograd.grad(gr.sum(), [a])  # what is the vector in HVP?
a, hvp # 4 * 3 * x
```

Out[40]:
```
(tensor([0., 1., 2., 3., 4.], requires_grad=True),
 tensor([  0.,  12.,  48., 108., 192.]))
```

backward as a convenience method

Above we used `torch.autograd.grad`. More commonly, you find `loss.backward()`. This is a convenience method for computing the gradient of all leaves `t` and storing the gradient in `t.grad`. In particular, PyTorch's built-in optimizers use these.

You can ask for a gradient of an intermediate to be stored by `t.retain_grad()`.

My advice: Only use backward to get gradients to then use with the optimizer. Don't use backward with `create_graph`.

A fun trick

Sometimes, we want the backward to not quite be like the forward. Round obviously has gradient 0, but wouldn't it be sometimes be convenient if...

In [41]:
```
a = torch.randn(5, requires_grad=True)
b = a.round()
gr, = torch.autograd.grad(b.sum(), [a])
gr
```

Out[41]:
```
tensor([0., 0., 0., 0., 0.])
```

...we can actually do that (not this construction, but the same rounding trick is the gist of "quantization aware training", other things like "gumble softmax" also work like this):

In [42]:
```
def differentiable_round(a, ndigits=None):
    if ndigits is None:
        b = a.round()
    else:
        scale = 10 ** ndigits
        b = (a * scale).round() / scale
    return a + (b - a).detach()  # PyTorch's version of "cleverly add 0"
c = differentiable_round(a)
gr2, = torch.autograd.grad(c.sum(), [a])
(c == b).all().item(), gr2
```

Out[42]:
```
(True, tensor([1., 1., 1., 1., 1.]))
```

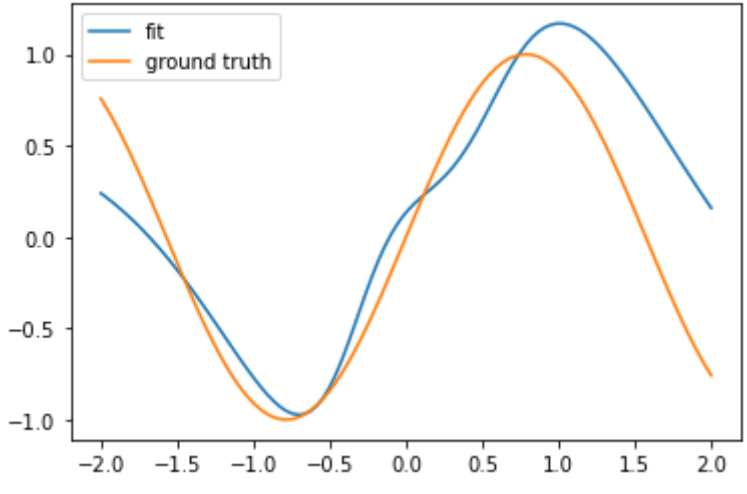
Fake quantization training the hard way

In [43]:
```
w1 = torch.randn(1, 10, requires_grad=True)
b1 = torch.randn(10, requires_grad=True)
w2 = torch.randn(10, 1, requires_grad=True)
b2 = torch.randn(1, requires_grad=True)
params = [w1, b1, w2, b2]


for i in range(200):
    x = torch.randn(10, 1)
    target = torch.sin(2 * x)

    xr = differentiable_round(x)
    w1r = differentiable_round(w1, 0.5)
    b1r = differentiable_round(b1, 0.5)
    w2r = differentiable_round(w2, 0.5)
    b2r = differentiable_round(b2, 0.5)
    pred = torch.tanh(xr @ w1r + b1r) @ w2r + b2r

    loss = ((pred - target) ** 2).sum()

    grads = torch.autograd.grad(loss, params)

    for p, g in zip(params, grads):
        with torch.no_grad():
            p -= 1e-2 * g
```

Did it work?

In [44]:
```
with torch.no_grad(): # no gradients here...
    w1r = differentiable_round(w1, 0.5)
    b1r = differentiable_round(b1, 0.5)
    w2r = differentiable_round(w2, 0.5)
    b2r = differentiable_round(b2, 0.5)
    x0 = torch.linspace(-2, 2)
    final_preds = torch.tanh(x0[:, None] @ w1r + b1r) @ w2r + b2r
    gt = torch.sin(2*x0)

pyplot.plot(x0, final_preds, label='fit')
pyplot.plot(x0, gt, label='ground truth')
pyplot.legend()
```

Out[44]:
```
<matplotlib.legend.Legend at 0x7f65d403e0d0>
```



Own Autograd functions

Let's consider $f : \mathbb{R}^n \to \mathbb{R}^m$ implicitly given by $F(x, f(x)) = 0$ for some function $F : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$.

For nice (smooth, nondegenerate) functions, the implicit function theorem (https://en.wikipedia.org/wiki/Implicit_function_theorem) gives us:

Given x, y such that $F(x, y) = 0$ there is a neighborhood $U \ni x$ and a function f such that $f(x) = y$ and $F(x, f(x)) = 0$. And if F is nice enough, we can also compute the derivative of f at x, namely $\frac{df}{dx}(x) = -(\frac{dF}{dy}(x, y))^{-1} \frac{dF}{dx}(x, y)$.

Given F in PyTorch, can we define f and backpropagate through the computation?

$\Rightarrow$ Need to extend autograd.

In [45]:

```
class Implicit(torch.autograd.Function):  # derive autograd.Function
    @staticmethod
    def forward(ctx, x, y0, F, max_iter=100):  # Context ctx as special first p
arameter to static method forward
        with torch.enable_grad():
            y = y0.clone().detach().requires_grad_()
            xv = x.detach()
            opt = torch.optim.LBFGS([y], max_iter=max_iter)  # numerical search
for y s.t. F(x, y) = 0
            def reevaluate():
                opt.zero_grad()
                z = F(xv,y)**2
                z.backward()
                return z
            opt.step(reevaluate)
        ctx._the_function = F                 # save non-tensors
        ctx.save_for_backward(x, y)           # safely save tensors
        return y
    @staticmethod
    def backward(ctx, output_grad):          # ouput_grad = dl / dy
        x, y = ctx.saved_tensors             # get the saved tensors
        F = ctx._the_function                # other arguments
        with torch.enable_grad():
            xv = x.detach().requires_grad_()
            y = y.detach().requires_grad_()
            z = F(xv,y)
            z.backward()
        return -xv.grad/y.grad*output_grad, None, None, None  # return dl / dx
 and None for all other inputs
```

Let's try it on a simple case:

In [46]:
```
def circle(x,y):
    return x**2+y**2-1  # 0-set = unit circle

x = torch.tensor([0.5], dtype=torch.double, requires_grad=True)
y0 = torch.tensor([0.5], dtype=torch.double) # starting value
y= Implicit.apply(x, y0, circle)    # this is how to use our new function
print (y, (1-0.5**2)**0.5)
gr, = torch.autograd.grad(y, x)
gr_exp, = torch.autograd.grad((1-x**2)**0.5, [x])
gr, gr_exp
```

```
tensor([0.8660], dtype=torch.float64, grad_fn=<ImplicitBackward>) 0.8660254037
844386
```

Out[46]:
```
 (tensor([-0.5774], dtype=torch.float64),
  tensor([-0.5774], dtype=torch.float64))
```

We could now backpropagate through it, too.

PyTorch is ... a Neural Network (and general Modelling) Library

`torch.nn.Module` and subclasses play the central role as Neural Network building blocks. Modules are for bundling neural network *state*, i.e. Parameters, other Buffers, Submodules, with the rules to compute outputs from inputs.

- Parameters: wrap in torch.nn.Parameter + assign
- Buffers (Tensors noch trained via SGD, e.g. statistics): register_buffer
- Submodules: assign

Here is the equivalent of our previous model:

In [47]:
```
class MyModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.w1 = torch.nn.Parameter(torch.randn(1, 10))
        self.b1 = torch.nn.Parameter(torch.randn(10))
        self.w2 = torch.nn.Parameter(torch.randn(10, 1))
        self.b2 = torch.nn.Parameter(torch.randn(1))

    def forward(self, x):
        return torch.tanh(x @ self.w1 + self.b1) @ self.w2 + self.b2

model = MyModel()
model.to(device="cuda")
model.w1
```

Out[47]:
```
Parameter containing:
tensor([[ 1.1177,  1.2803, -3.9744, -2.1031, -1.7723, -0.8169,  0.2313,  1.334
9,
         -0.5564, -0.0529]], device='cuda:0', requires_grad=True)
```

A rich library of pre-made modules under `torch.nn`.

- `torch.nn.Linear`, `Conv2d`, `MaxPool2D`, `Tanh`, ... Also loss functions:
- `CrossEntropyLoss`, `MSELoss`, `CTCLoss`...

Careful: Naming `CrossEntropyLoss` vs. `NLLLoss` etc. is particular to PyTorch. Check the documentation for PyTorch's expectation of whether input have `softmax` / `log_softmax` / something else applied.

We can make our model again with these:

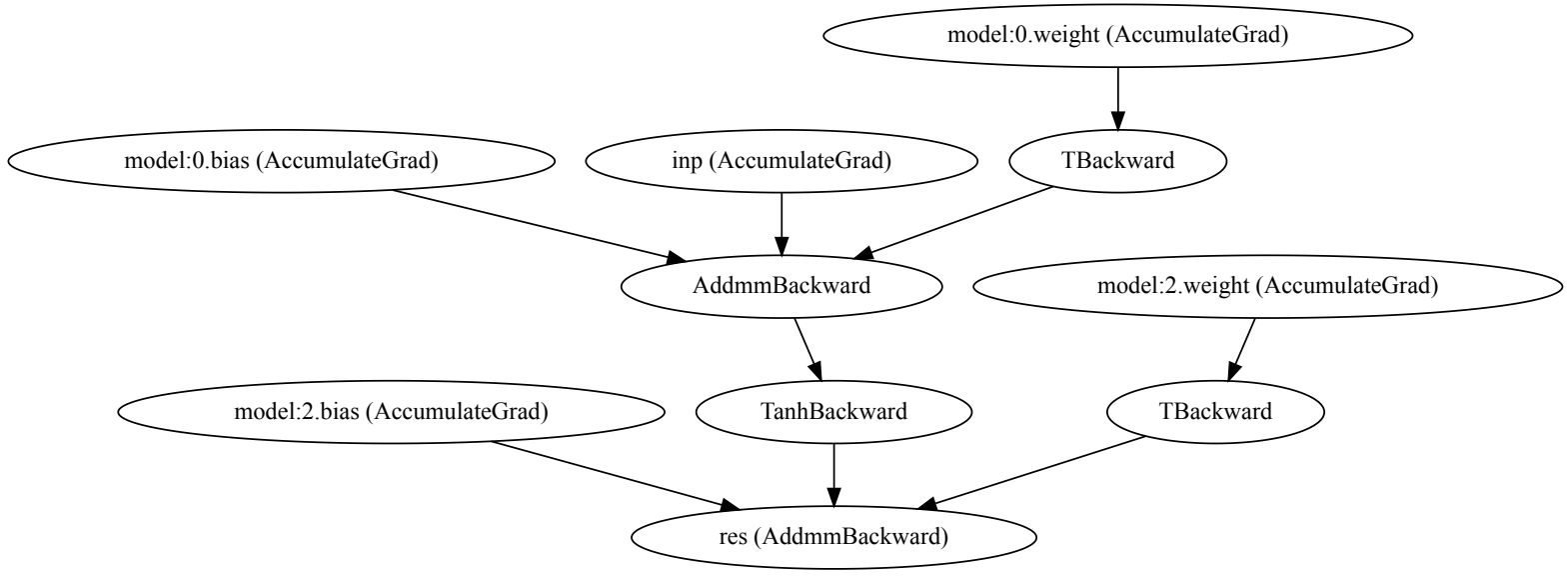
In [48]:
```
class MyModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.l1 = torch.nn.Linear(1, 10)
        self.l2 = torch.nn.Linear(10, 1)

    def forward(self, x):
        x = torch.tanh(self.l1(x))
        return self.l2(x)

model = MyModel()
model
```

Out[48]:
```
MyModel(
  (l1): Linear(in_features=1, out_features=10, bias=True)
  (l2): Linear(in_features=10, out_features=1, bias=True)
)
```

There also is a `Sequential` class for particularly simple "chains" of modules (with only one input/output).

In [49]:
```
model = torch.nn.Sequential(
            torch.nn.Linear(1, 10),
            torch.nn.Tanh(),
            torch.nn.Linear(10, 1))

inp = torch.randn(5, 1, requires_grad=True)
res = model(inp)
ptgraph(res)
```

Out[49]:



Task: Can you subclass sequential to get a model class?

Loading and Saving models

While you can save and load models using `torch.save` and `torch.load`, it is generally not recommended.

Instead, save the state dict and upon loading instantiate the model and load the state dict. This is much better for forward compatibility.

In [50]:
```
torch.save(model.state_dict(), 'saved_model.pt')
sd = torch.load('saved_model.pt')
model.load_state_dict(sd)
```

Out[50]:
```
<All keys matched successfully>
```

Some thoughts on functional vs. Module

- If you write for re-use, the functional / Module split of PyTorch has turned out to be a good idea.
- Use functional for stuff without state (unless you have a quick and dirty Sequential).
- Never re-use modules (define one `torch.nn.ReLU` and use it 5 times). It's a trap!

  When doing analysis or quantization (when `ReLU` becomes stateful due to quantization params), this will break.

PyTorch has ... optimizers

In `torch.optim`.

- Most common probably: `torch.optim.SGD`, `torch.optim.Adam`, but also `torch.optim.LBFGS` etc.
- Schedules for the learning are also available.

Pattern:

- `loss = ...`
- `opt.zero_grad()`
- `loss.backward()`
- `opt.step()`

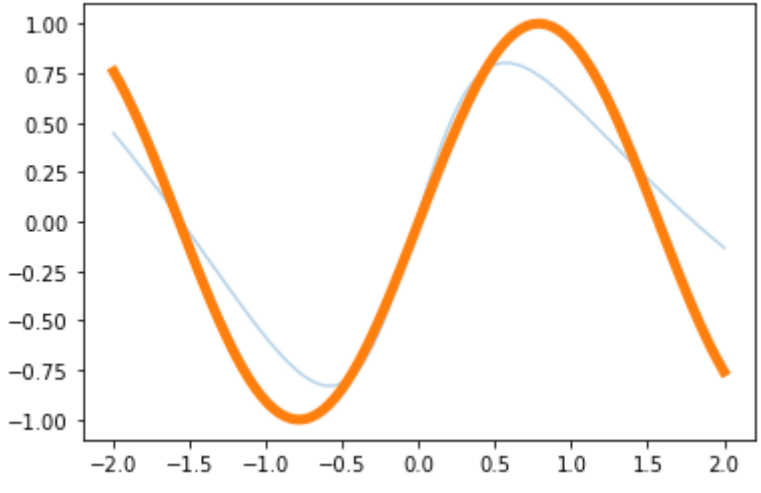
Optimizers also have a `state_dict` for saving / restoring.

Let's see that in action with our toy model:

In [51]:
```
model = torch.nn.Sequential(
            torch.nn.Linear(1, 10),
            torch.nn.Tanh(),
            torch.nn.Linear(10, 1))

opt = torch.optim.SGD(model.parameters(), lr=1e-2)

for i in range(200):
    x = torch.randn(10, 1)    # input  from dataset
    target = torch.sin(2 * x) # target from dataset

    pred = model(x)
    loss = torch.nn.functional.mse_loss(pred, target, reduction='sum')

    opt.zero_grad()
    loss.backward()
    opt.step()
```

In [52]:

```
with torch.no_grad():
    x0 = torch.linspace(-2, 2)
    pyplot.plot(x0, model(x0[:, None]).detach(), alpha=0.3)
    pyplot.plot(x0, torch.sin(2 * x0), linewidth=5)
```



PyTorch has ... utilities for data handling

PyTorch splits data handling in two parts:

- Problem specific (DataSet)
- Generic, e.g. Sampling, Batching (DataLoader)

User's responsibility: Define `torch.utils.data.DataSet` subclass with

- a `__len__` method (how many data points)
- a `__getitem__` method to get the `i`th element, returning tuples of numbers and tensors

`DataLoader` is more or less universal. In several parallel processes, get items from dataset and assemble them into batches (for numbers, tensors, make tensors, for other things, lists).

Let's try that:

In [53]:
```
import torch.utils.data
class DS(torch.utils.data.Dataset):
    def __init__(self, shape):
        self.shape = shape
    def __len__(self):
        return 100
    def __getitem__(self, i):
        return (i % 10 + torch.randn(self.shape), i % 10) # observation and lab
el

ds = DS((5,))
dl = torch.utils.data.DataLoader(ds, batch_size=8, shuffle=True)

for inp, label in dl:
    print(inp, label)
    break
```

```
tensor([[ 9.5571,  7.1095,  7.1393,  7.4666,  6.7035],
        [ 5.3171,  9.0150,  6.7444,  7.5128,  5.4807],
        [ 9.7531,  7.9709,  8.2818, 10.1165, 10.7010],
        [-1.1085,  1.3127,  0.5249,  1.1068, -0.7207],
        [ 1.0284,  0.7190,  3.0181,  3.5250,  3.3695],
        [ 3.6932,  0.9453,  1.3540,  0.8289,  0.6140],
        [ 5.3443,  3.6160,  5.4480,  5.8480,  4.3890],
        [ 1.3029,  1.7199,  0.7438,  0.8375,  2.5249]]) tensor([7, 7, 9, 0, 2,
1, 5, 1])
```

What happened here?

The DataLoader provides an iterator and for each iteration

- requested elements from the dataset,
- took the 1d 5-tensors and `stack`ed them to a 2d `batchsize x 5` tensor,
- took the integers and put them into a 1d `batchsize` tensor.
- gives the tuple of input and label to the for loop.

Good to know: If `batchsize` does not divide `len(ds)` , the last batch will be shorter. This can be troublesome (e.g. batch norm running with a batch size of 1), so you have the `drop_last` optional command to just skip this minibatch.

Performance of the DataLoader

Using `Dataset / DataLoader` is simple, but keep in mind that it can greatly impact performance.

How do you detect this? If, during training, your GPU is not constantly fully used (shown by `*smi` tools), you have a CPU bottleneck somewhere, and the data loading is one thing to check.

- If more cores help, there is the `num_workers` argument to the dataloader.
- Get a SSD.
- If you can do preprocessing beforehand (no augmentation), try that. (But careful, I/O is often more expensive than processing.)
- If you can do preprocessing on the GPU instead (augmentation in particular), do that.
- Sometimes it helps to put tensors into *GPU pinned memory*, with the `pin_memory` argument for the DataLoader.

Excursion: GPU, efficiency, measurement

One of my informal mottos is *It's not optimization until you measure*. I'll only discuss the most basic measurement here, PyTorch offers a capable profiling facility, too.

When you think about code being slow, it's important to figure out what is slow and why. To my mind, a lot of measurement can be done with very basic tools, e.g. IPython's `%timeit` magic.

As GPU computation is and shold be ansynchronous, avoid unneeded synchronization points. Synchronization happens when the CPU waits for the GPU (to get the results).

- Synchronizations can happen because the program needs to know something (e.g. sizes of tensors depending on the input). Often, these are unavoidable.
- Typical sources of spurious synchronizations are too frequent `.to(device="cpu")`, `.item()`, `.to_list()`, `print`.

If we want to time GPU kernels, we want to be sure to synchronize before taking the start and end times. Typically, we also want to have some "warm-up", i.e. run the measured function before timing.

Let's take the uniformity loss from Wang and Isola: Understanding Contrastive Representation Learning through Alignment and Uniformity on the Hypersphere (https://arxiv.org/abs/2005.10242) (a great paper!).

The Uniformity loss is defined as a function of the pairwise distances over a largish set of vectors.

In [54]:
```
def lunif(x, t=2): # copied from the paper
    sq_pdist = torch.pdist(x, p=2).pow(2)
    return sq_pdist.mul(-t).exp().mean().log()

x = torch.randn(1024, 128, device="cuda")
x /= x.norm(p=2, dim=1, keepdim=True).requires_grad_()

lunif(x)
```

Out[54]:
```
tensor(-3.9367, device='cuda:0', grad_fn=<LogBackward>)
```

One would think that the specialised `pdist` function is the right tool for the job. But is it? Let's time it.

In [55]:
```
def totime(fn):
    l = fn(x)
    g, = torch.autograd.grad(l, x)
    torch.cuda.synchronize()

totime(lunif) # warmup
%timeit totime(lunif)
```

```
19.4 ms ± 103 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Let's use $|x-y|^2 = |x|^2 + |y|^2 - 2 < x, y >$ and compare.

In [56]:
```
def lunif2(x, t=2):
    t=2
    xnorm = torch.norm(x, p=2, dim=1).pow(2)
    sq_pdist = xnorm[None] + xnorm[:, None] - 2 * torch.mm(x, x.t())
    exp = sq_pdist.mul(-t).exp().tril(diagonal=-1)
    N = x.size(0)
    res = exp.sum().mul(2/(N*N-N)).log()
    return res

print((lunif2(x.to(torch.double)) - lunif(x.to(torch.double))).item())

totime(lunif2)
%timeit totime(lunif2)
```

```
9.313225746154785e-10
2.26 ms ± 16.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Even though we have stark inefficiencies (like taking tril and taking a copy to do so), this is almost an order of magnitude faster!

Largely due to backward of `pdist` implementation.

But Python is slow...

Uniformity loss: "what formulas you use" is the real bottleneck (unless you optimize pdist).

The "what" typically should be the first optimization target.

But when we fix the task ("what"), how can we optimize?

Conventional wisdom: **Python is slow**

- certainly, Python isn't fast (`for` loop vs C++ `for` loop)
- but, if the GPU is saturated $\Rightarrow$ Python isn't the bottleneck

How PyTorch programs spend their time

At a very high level, you can divide time spent into these parts:

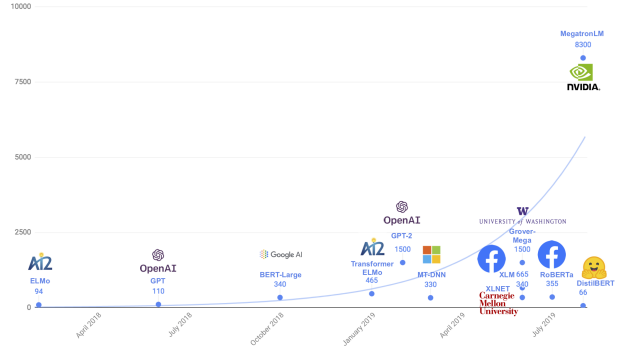
- Python program flow,
- Data "administrative overhead" (creating `Tensor` data structures, autograd `Node`s etc.),
- Data aquisition (I/O),
- Computation roughly as
    - fixed overhead (kernel launches etc.),
    - reading / writing memory,
    - "real computation".

Thomas' rule of thumb: As long as your operands are reasonably large (say 100s of elements, not single elements), Python and Data "administrative overhead" probably isn't your main problem.

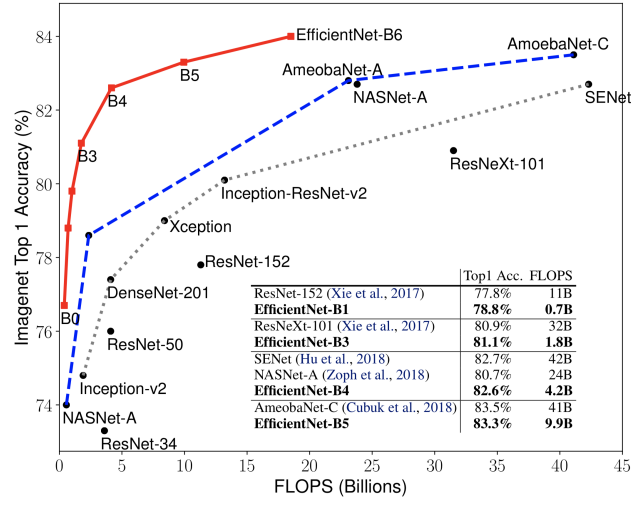
Optimization potentials

remember that modelling does have a huge lever regarding efficiency

Millions of parameters vs. timeline for the popular transformer models



Source: V. Sanh (Huggingface), DistilBERT



Source: M. Tan, Q. Le: EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks

The PyTorch JIT

So if we have our model in PyTorch. Can we export it?

This is where the PyTorch Just-In-Time compiler comes into play. It provides a graphical representation called *TorchScript* (used somewhat synonymously for the language and the representation).

There are two main ways to get a TorchScript graph:

- scripting and
- tracing.

We'll look at these in some detail.

Scripting

Scripting compiles (mostly) a subset of Python. It takes the Python source code and transforms it. "Here is what the function should do", just like normal programming.

In [57]:
```
@torch.jit.script
def fn(x):
    return x * 2
```

In [58]:
```
fn, fn.graph
```

Out[58]:
```
(<torch.jit.ScriptFunction at 0x7f65d43b2220>,
 graph(%x.1 : Tensor):
   %2 : int = prim::Constant[value=2]() # <ipython-input-57-6dcc6c3c1c8e>:3:15
   %3 : Tensor = aten::mul(%x.1, %2) # <ipython-input-57-6dcc6c3c1c8e>:3:11
   return (%3))
```

Tracing

Tracing runs the code and observers the calls into PyTorch with some sample input. "Watch me, now you know how to do the same."

In [59]:
```
def fn(x):
    return x * 2
fn = torch.jit.trace(fn, [torch.randn(5)])

print(fn.graph, fn.code)
```

```
graph(%x : Float(5:1)):
  %1 : Long() = prim::Constant[value={2}]() # <ipython-input-59-7d43d0af2e80>:
2:0
  %2 : Float(5:1) = aten::mul(%x, %1) # <ipython-input-59-7d43d0af2e80>:2:0
  return (%2)
 def fn(x: Tensor) -> Tensor:
  return torch.mul(x, CONSTANTS.c0)
```

N.B.: The specialization for the Tensor shape isn't relevant here and will be erased e.g. during saving of the model.

TorchScript

One important difference between TorchScript and Python is that in TorchScript everything is typed. Important types are

- `bool`, `int`, `long`, `double` for numbers (int = 32 bit integer, long = 64 bit integer)
- `Tensor` for tensors (of arbitrary shape, dtype, ...)
- `List[T]` a list with elements fo type T (one of the above)
- Tuples are of fixed size with arbitrary but fixed element type, so e.g. `Tuple(Tensor, int)`.
- `Optional[T]` for things that can be `None`

`None` always is of type `Optional[T]` for some specific `T` (except in the rarest circumstances).

PyTorch will mostly infer the intermediate and return types, but you need to annotate any non-Tensor inputs.

Tracing vs. Scripting

Scripting will process all code but may not understand all. This means it captures all constructs (like control flow) it understands, but it will fail if it doesn't understand something.

Tracing doesn't see anything not calling into PyTorch and will happily ignore that (e.g. control flow). This is also the reason why it will loudly complain if you have non-tensor inputs.

In [60]:
```
def fn(x):
    for i in range(x.dim()):
        x = x * x
    return x

script_fn = torch.jit.script(fn)
trace_fn = torch.jit.trace(fn, [torch.randn(5, 5)])
```

In [61]:
```
print(script_fn.code)
```

```
def fn(x: Tensor) -> Tensor:
  x0 = x
  for i in range(torch.dim(x)):
    x0 = torch.mul(x0, x0)
  return x0
```

In [62]:
```
print(trace_fn.code)
```

```
def fn(x: Tensor) -> Tensor:
  x0 = torch.mul(x, x)
  return torch.mul(x0, x0)
```

Tracing and Scripting Modules

But our models often are not functions. What now?

With tracing, we can work just like with functions. We get a `ScriptModule` subclass that behaves much like a `Module` with parameters, state dict etc.

In [63]:
```
traced_model = torch.jit.trace(model, [torch.randn(8, 1)])
type(traced_model), traced_model
```

Out[63]:
```
(torch.jit._trace.TopLevelTracedModule,
 Sequential(
   original_name=Sequential
   (0): Linear(original_name=Linear)
   (1): Tanh(original_name=Tanh)
   (2): Linear(original_name=Linear)
 ))
```

Saving is a bit different, here we include the model on purpose:

In [64]:
```
traced_model.save('./traced_model.pt')
loaded_model = torch.jit.load('./traced_model.pt')

loaded_model(torch.randn(8,1))
```

Out[64]:
```
tensor([[-0.8112],
        [ 0.3232],
        [ 0.7830],
        [-0.8254],
        [ 0.7857],
        [-0.3138],
        [-0.7612],
        [ 0.5701]], grad_fn=<DifferentiableGraphBackward>)
```

Scripting Modules

Scripting modules is ... a bit tricky. We don't script the class in its entirety but instead take an instance (in particular past `__init__`) and process its data members and methods (the latters work like script functions).

In [65]:
```
scripted_model = torch.jit.script(model)
print(scripted_model.code)
```

```
def forward(self,
    input: Tensor) -> Tensor:
  _0 = getattr(self, "0")
  _1 = getattr(self, "1")
  _2 = getattr(self, "2")
  input0 = (_0).forward(input, )
  input1 = (_1).forward(input0, )
  return (_2).forward(input1, )
```

We can also look at the graph including submodules, but it gets unwieldy rather fast:

In [66]:
```
scripted_model.forward.inlined_graph
```

Out[66]:
```
graph(%self : __torch__.torch.nn.modules.container.___torch_mangle_13.Sequenti
al,
      %input.1 : Tensor):
  %2 : __torch__.torch.nn.modules.linear.___torch_mangle_10.Linear = prim::Get
Attr[name="0"](%self)
  %3 : __torch__.torch.nn.modules.activation.___torch_mangle_11.Tanh = prim::G
etAttr[name="1"](%self)
  %4 : __torch__.torch.nn.modules.linear.___torch_mangle_12.Linear = prim::Get
Attr[name="2"](%self)
  %8 : int = prim::Constant[value=1]()
  %9 : int = prim::Constant[value=2]() # /usr/local/lib/python3.8/dist-package
s/torch/nn/functional.py:1672:22
  %10 : Tensor = prim::GetAttr[name="weight"](%2)
  %11 : Tensor = prim::GetAttr[name="bias"](%2)
  %12 : int = aten::dim(%input.1) # /usr/local/lib/python3.8/dist-packages/tor
ch/nn/functional.py:1672:7
  %13 : bool = aten::eq(%12, %9) # /usr/local/lib/python3.8/dist-packages/torc
h/nn/functional.py:1672:7
  %input.3 : Tensor = prim::If(%13) # /usr/local/lib/python3.8/dist-packages/t
orch/nn/functional.py:1672:4
    block0():
      %15 : Tensor = aten::t(%10) # /usr/local/lib/python3.8/dist-packages/tor
ch/nn/functional.py:1674:39
      %ret.2 : Tensor = aten::addmm(%11, %input.1, %15, %8, %8) # /usr/local/l
ib/python3.8/dist-packages/torch/nn/functional.py:1674:14
      -> (%ret.2)
    block1():
      %17 : Tensor = aten::t(%10) # /usr/local/lib/python3.8/dist-packages/tor
ch/nn/functional.py:1676:30
      %output.2 : Tensor = aten::matmul(%input.1, %17) # /usr/local/lib/python
3.8/dist-packages/torch/nn/functional.py:1676:17
      %output.4 : Tensor = aten::add_(%output.2, %11, %8) # /usr/local/lib/pyt
hon3.8/dist-packages/torch/nn/functional.py:1678:12
      -> (%output.4)
  %input.5 : Tensor = aten::tanh(%input.3) # /usr/local/lib/python3.8/dist-pac
```

What can you do with scripted modules?

- Run them as is, bypassing Python.
    - not as much speedup as often is expected (maybe 5%-10% for some models I tested),
    - but - sometimes crucially - it avoids the dreaded Python Global Interpreter Lock (GIL), so it is useful e.g. for multithreaded things like serving PyTorch models.
- Export and run in C++ / Mobile / ..., export to other frameworks like TVM (https://tvm.ai/).
- Apply holistic optimizations (this is what a submodule, the JIT fuser does).

Holistic Optimizations - JIT fusers

So currently the fuser is a hotspot of development, look at the competition. We'll use `fuser0` here.

In [67]:
```
help(torch.jit.fuser)
```

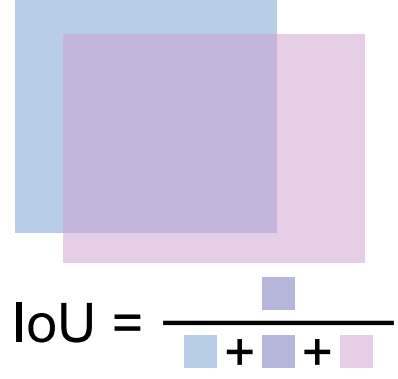
```
Help on function fuser in module torch.jit:

fuser(name)
    A context manager that facilitates switching between
    backend fusers.

    Valid names:
    * ``fuser0`` - enables only legacy fuser
    * ``fuser1`` - enables only NNC
    * ``fuser2`` - enables only nvFuser
```

Example: Intersection over Union loss

...for detection models.

Many elementwise operations. Usually PyTorch executes them one by one, reading input from memory and writing back outputs.

By fusing this into a single GPU kernel, we can save all that.



In [68]:
```
def ratio_iou(x1, y1, w1, h1, x2, y2, w2, h2):
    xi = torch.max(x1, x2)                                  # Intersection (yi si
milarly)
    yi = torch.max(y1, y2)                                  # Intersection (yi si
milarly)
    wi = torch.clamp(torch.min(x1+w1, x2+w2) - xi, min=0)
    hi = torch.clamp(torch.min(y1+h1, y2+h2) - yi, min=0)
    area_i = wi * hi                                        # Area Intersection
    area_u = w1 * h1 + w2 * h2 - wi * hi                    # Area Union
    return area_i / torch.clamp(area_u, min=1e-5)

ratio_iou_scripted = torch.jit.script(ratio_iou)
```

This gives us a good speedup:

In [69]:
```
x1, y1, w1, h1, x2, y2, w2, h2 = torch.randn(8, 100, 1000, device='cuda').exp()

def take_time(fn):
    _ = fn(x1, y1, w1, h1, x2, y2, w2, h2)
    torch.cuda.synchronize()

take_time(ratio_iou) # warmup
%timeit take_time(ratio_iou)


take_time(ratio_iou_scripted)
%timeit take_time(ratio_iou_scripted)
```

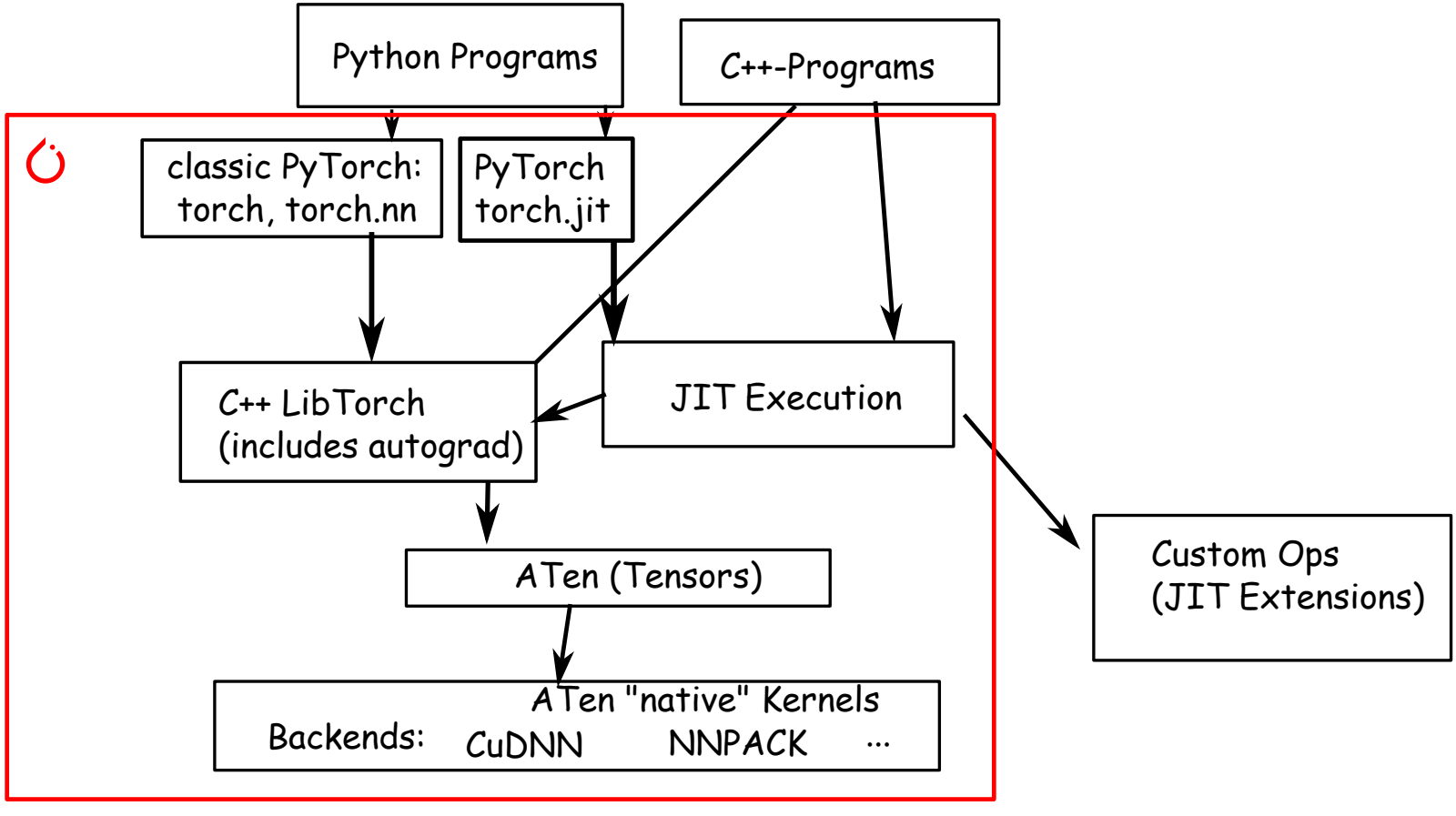
```
238 µs ± 998 ns per loop (mean ± std. dev. of 7 runs, 1000 loops each)
40.3 µs ± 113 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

We can see in the graph specialised for the inputs which operations are fused:

In [70]:
```
ratio_iou_scripted.graph_for(x1, y1, w1, h1, x2, y2, w2, h2)
```

Out[70]:
```
graph(%x1.1 : Float(*, *),
      %y1.1 : Float(*, *),
      %w1.1 : Float(*, *),
      %h1.1 : Float(*, *),
      %x2.1 : Float(*, *),
      %y2.1 : Float(*, *),
      %w2.1 : Float(*, *),
      %h2.1 : Float(*, *)):
  %32 : Float(*, *) = prim::FusionGroup_0(%w2.1, %h2.1, %w1.1, %h1.1, %y2.1, %
y1.1, %x2.1, %x1.1)
  return (%32)
with prim::FusionGroup_0 = graph(%14 : Float(*, *),
      %15 : Float(*, *),
      %17 : Float(*, *),
      %18 : Float(*, *),
      %34 : Float(*, *),
      %37 : Float(*, *),
      %51 : Float(*, *),
      %54 : Float(*, *)):
  %4 : float = prim::Constant[value=1.0000000000000001e-05]() # <ipython-input
-68-5fb00890f03c>:8:44
  %42 : None = prim::Constant()
  %41 : int = prim::Constant[value=0]() # <ipython-input-68-5fb00890f03c>:4:55
  %55 : int = prim::Constant[value=1]()
  %xi.1 : Float(*, *) = aten::max(%54, %51) # <ipython-input-68-5fb00890f03c>:
2:9
  %yi.1 : Float(*, *) = aten::max(%37, %34) # <ipython-input-68-5fb00890f03c>:
3:9
  %56 : Float(*, *) = aten::add(%54, %17, %55) # <ipython-input-68-5fb00890f03
c>:4:31
  %53 : Float(*, *) = aten::add(%51, %14, %55) # <ipython-input-68-5fb00890f03
c>:4:38
  %50 : Float(*, *) = aten::min(%56, %53) # <ipython-input-68-5fb00890f03c>:4:
21
  %47 : Float(*, *) = aten::sub(%50, %xi.1, %55) # <ipython-input-68-5fb00890f
```

PyTorch is ... Pythonic, but not Python only



When we looked at *PyTorch is functions*, we actually conflated PyTorch the interface and PyTorch the backend.

The JIT was another way of accessing the backend.

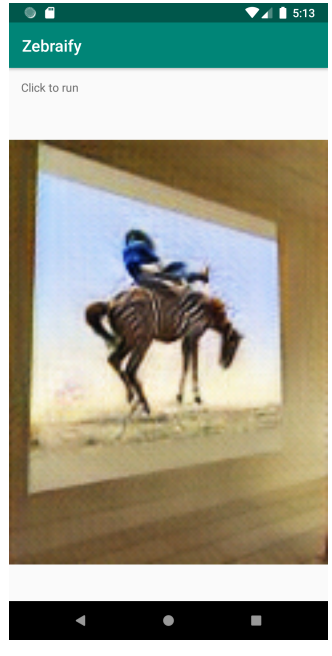
Looking closer, we have `libtorch` (the C++-implemented autograd-enabled backend) calling down into `ATen` for tensor operations and there the dispatching to ATen's own kernels or libraries.

Beyond Python

We can also run our JITed models from C++ in a few lines of code or on mobile.

The picture shows a CycleGAN running on PyTorch Mobile (from Chapter 15 of our book, just like the PyTorch sketch above).

We also cover C++ there, but not now.



PyTorch is ... much more

We skipped `torch.distributions`, `distributed`, automatic mixed precision and more PyTorch has to offer. Check out the tutorials on PyTorch.org.

Forums

Forums at discuss.pytorch.org (https://discuss.pytorch.org/) probably are the best source of casual help, with a number of core developers also keeping an eye for the advanced questions. (I'm @tom there.)

Wider Ecosystem

Libraries for many classes of models building on PyTorch (not complete)

- Immediate "Satellites" like TorchVision, TorchAudio, TorchText, TorchServing
- Extending capabilities: PyTorch-NLP, Graph NNs, Gaussian Processes, Pyro (Probabilistic)
- Training helpers

Thank you!

Your questions and comments

Graves style Handwriting in PyTorch (https://github.com/t-vi/pytorch-tvmisc/blob/master/misc/graves_handwriting_generation.ipynb)

Contact: Thomas Viehmann
tv@mathinf.eu (mailto:tv@mathinf.eu)

In [ ]: