covers various pytorch basics; intended for interactive use. is a bit overly detailed... –matus

1 tensor operations

[1]: `import torch
# torch has its own PRNG seeds.
# setting it here so notebook is deterministic.
# warning: last i checked, torch and numpy and core python generators are all decoupled.
torch.manual_seed(0)

[1]: <torch._C.Generator at 0x7fb9980f7d30>

[2]: # create a non-inclusive range..
torch.arange(8)

[2]: tensor([0, 1, 2, 3, 4, 5, 6, 7])

[3]: # ..just like a regular python non-inclusive range
torch.tensor(range(8))

[3]: tensor([0, 1, 2, 3, 4, 5, 6, 7])

[4]: # a similar routine, subdividing an interval equally
torch.linspace(0, 1, 5)

[4]: tensor([0.0000, 0.2500, 0.5000, 0.7500, 1.0000])

[5]: # standard arithmetic operations act coordinate-wise
xs = torch.linspace(0, 1, 5)
print(xs ** 2) # coordinate-wise squaring
print(xs * xs) # coordinate-wise multiplication
print((xs ** 2 - xs * xs) < 1e-16) # compare the above,
print((2.7182818 * xs).log()) # coordinate-wise multiplication and ln

tensor([0.0000, 0.0625, 0.2500, 0.5625, 1.0000])
tensor([0.0000, 0.0625, 0.2500, 0.5625, 1.0000])
# pytorch tensors aren't just floating point

```python
print(torch.linspace(0,1,5).dtype,
      torch.arange(5).dtype,
      (torch.arange(5) / 10).dtype,
      (torch.arange(5) // 10).dtype,
      (torch.arange(5) == 0).dtype,
)
```

```
torch.float32 torch.int64 torch.float32 torch.int64 torch.bool
```

/home/matus/.local/lib/python3.9/site-packages/torch/_tensor.py:575:
UserWarning: floor_divide is deprecated, and will be removed in a future version of pytorch. It currently rounds toward 0 (like the 'trunc' function NOT 'floor'). This results in incorrect rounding for negative values. To keep the current behavior, use torch.div(a, b, rounding_mode='trunc'), or for actual floor division, use torch.div(a, b, rounding_mode='floor'). (Triggered internally at /pytorch/aten/src/ATen/native/BinaryOps.cpp:467.)
    return torch.floor_divide(self, other)

# arithmetic operations generally convert between types

```python
print(torch.arange(5) * torch.arange(5))
print(torch.arange(5) * (torch.arange(5) / 1))
print(torch.arange(5) + (torch.arange(5) / 1))
print(torch.arange(5) + (torch.arange(5) == 0))
```

```
tensor([  0,   1,   4,   9,  16])
tensor([  0.,  1.,  4.,  9., 16.])
tensor([ 0., 2., 4., 6., 8.])
tensor([1, 1, 2, 3, 4])
```

# Not all types support all operations

```python
try:
    # following does manual type conversion
    print(torch.arange(5).type(torch.float32).exp())
    print(torch.arange(5).exp())
except RuntimeError as E:
    print(f"Got exception: '{E}'")
```

```
tensor([ 1.0000, 2.7183, 7.3891, 20.0855, 54.5981])
tensor([ 1.0000, 2.7183, 7.3891, 20.0855, 54.5981])
```

# Here are some basic operations on matrix shapes

```python
ns = torch.arange(12)
print(ns)
print(ns.reshape(3,4))
# .view() is similar to .reshape() but reuses storage;
```

```
tensor([ 0,  1,  4,  7, 10, 13,  2,  5,  8, 11, 14, 17])
tensor([[ 0,  1,  4,  7],
        [10, 11, 14, 17]])
```
# we'll revisit it later.
print(ns.view(2,6))
tensor([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
tensor([[ 0, 1, 2, 3],
        [ 4, 5, 6, 7],
        [ 8, 9, 10, 11]])
tensor([[ 0, 1, 2, 3, 4, 5],
        [ 6, 7, 8, 9, 10, 11]])

[10]: # We can also reshape into 3 axes
print(ns.reshape(2,2,3))
# .reshape() and .view() also understand "-1" which means
# "choose the appropriate size so that this works out".
print(ns.reshape(-1,2,3).shape)
tensor([[ 0, 1, 2],
        [ 3, 4, 5],
        [ 6, 7, 8],
        [ 9, 10, 11]])
torch.Size([2, 2, 3])

[11]: # torch also has "0 axis" (or "0 order") tensors/"arrays".
# these are convenient because they still support .exp(), etc.
e = torch.tensor(2.7182818, dtype = torch.float32)
print(e, e.shape, e.log(), e.sin())
# method .item() extracts a python number
print(e, e.item())
# lol:
print(torch.tensor(2.7182818, dtype = torch.float64).item())
try:
    # .item() only works on single-element tensors
    print(torch.zeros(1,1,1,1).item())
    print(torch.zeros(2).item())
except ValueError as E:
    print(f"Got exception: '{E}'")
tensor(2.7183) torch.Size([]) tensor(1.) tensor(0.4108)
tensor(2.7183) 2.7182817459106445
2.7182818
0.0
Got exception: 'only one element tensors can be converted to Python scalars'

[12]: # back to larger tensors,
# _some_ (but not all) operations complain about size mismatch.
try:
    vs = torch.arange(6).reshape(2,3)
    print(vs + vs)
print(vs + vs.T)
except RuntimeError as E:
    print(f"Got exception: '{E}'")

tensor([[ 0,  2,  4],
        [ 6,  8, 10]])
Got exception: 'The size of tensor a (3) must match the size of tensor b (2) at non-singleton dimension 1'

[13]: # but some operations _do_ succeed with mismatched shapes!
    vs = torch.arange(4)
    print(vs.reshape(1, -1) + vs.reshape(-1, 1))
    print(vs.reshape(1, -1, 1) + vs.reshape(-1, 1, 1) + vs.reshape(1, 1, -1))

tensor([[ 0,  1,  2,  3],
        [ 1,  2,  3,  4],
        [ 2,  3,  4,  5],
        [ 3,  4,  5,  6]],
       tensor([[ 0,  1,  2,  3],
                [ 1,  2,  3,  4],
                [ 2,  3,  4,  5],
                [ 3,  4,  5,  6]],
               [[ 1,  2,  3,  4],
                [ 2,  3,  4,  5],
                [ 3,  4,  5,  6],
                [ 4,  5,  6,  7]],
               [[ 2,  3,  4,  5],
                [ 3,  4,  5,  6],
                [ 4,  5,  6,  7],
                [ 5,  6,  7,  8]],
               [[ 3,  4,  5,  6],
                [ 4,  5,  6,  7],
                [ 5,  6,  7,  8],
                [ 6,  7,  8,  9]])

[14]: # here's a simpler instance of the same behavior:
    torch.zeros(4, 4) + torch.arange(4).reshape(-1, 1)

[14]: tensor([[ 0.,  0.,  0.,  0.],
            [ 1.,  1.,  1.,  1.],
            [ 2.,  2.,  2.,  2.],
            [ 3.,  3.,  3.,  3.]]]

[15]: # This can be very convenient:
    # here we normalize the rows of a matrix:
X = torch.randn(3, 2)
print(X.norm(dim = 1))
# A few things going on here, I recommend trying this one
# yourself and studying each piece.
X /= X.norm(dim = 1, keepdim = True)
print(X.norm(dim = 1))
tensor([[1.5687, 2.2517, 1.7698]])
tensor([[1.0000, 1.0000, 1.0000]])

# slicing makes it easy to access submatrices/subtensors
ns = torch.arange(12).reshape(2,6)
print(ns)
print(ns[0, :]) # first row
print(ns[:, 0]) # first column
ms = ns.reshape(2,2,3)
print(ms)
print(ms[0,0,:])
print(ms[0, ...]) # dots mean "all remaining axes/dimensions"
print(ms[... , 0])
tensor([[[ 0, 1, 2, 3, 4, 5],
            [ 6, 7, 8, 9, 10, 11]]])
tensor([[ 0, 1, 2, 3, 4, 5]])
tensor([[ 0, 6]])
tensor([[[ 0, 1, 2],
            [ 3, 4, 5]],
        [[ 6, 7, 8],
            [ 9, 10, 11]]])
tensor([[ 0, 1, 2]])
tensor([[0, 1, 2],
        [3, 4, 5]])
tensor([[0, 3],
        [6, 9]])

tensor([[[ 0, 2, 4],
            [ 5]])
# many operations have in-place versions.
# superficially this is good for efficiency reasons.
# more importantly, pytorch does some internal book-keeping
# with autodifferentiation which is lost if you do not do
# in-place operations for variables you wish to compute
# gradients with respect to.
# (This will be clarified later.)
# For now, here are some example in-place operations
v = torch.randn(5,4)
print(v.norm())
v += v  # in-place arithmetic operations
v *= 2
print(v.norm())
v.clamp_(0, float('inf'))  # zero out negative values, in-place
print(v.norm())
tensor(3.7304)
tensor(14.9216)
tensor(11.3672)

# this step only matters if you have a gpu.
# this line of code is in my pytorch programs, it means
# "variable 'device' is first gpu if available, else cpu".
device = torch.device("cpu" if not torch.cuda.is_available() else "cuda:0")

# that didn't put anything on gpu; we manually move things there
ns = torch.arange(4)
ns2 = ns.to(device)
print(ns.device, ns2.device)

try:
    # python disallows operations mixing cpu and gpu;
    # this is good, since moving data between them is expensive.
    ns + ns2
print("no exception: no gpu in use")
except Exception as E:
    print(f"pytorch error: {E}")
cpu cpu
no exception: no gpu in use
2 matplotlib plotting

[20]: import matplotlib.pyplot as plt

[21]: 
# plt.plot() lets you display many curves.
# it has many parameters; in jupyter and ipython, you can execute
# "plt.plot?" to see some of them.
# note: gpu data must be moved to cpu before being passed to matplotlib
xs = torch.linspace(0, 2, 128)
plt.plot(xs, xs, marker = 's', markevery = 5,
         label = "identity")
plt.plot(xs, xs ** 2, marker = 'D', markevery = 7,
         label = 'squared')
plt.plot(xs, xs * xs, marker = '-', markevery = 9,
         label = 'squared again')
plt.plot(xs, (2 * xs).sin(), marker = 'd', markevery = 11,
         label = 'sin')
plt.legend()
X = torch.randn(100, 2)  # create some random data
u = torch.randn(2)  # sample a random "correct" linear predictor
# pick a norm for u that has easy visualization:
u **= X.norm(dim = 1).max() / u.norm()
y = X @ u  # label data according to the "planted" predictor
# scatterplot of data, y given by color:
plt.scatter(
    X[:, 0],
    X[:, 1],
    # color according to y:
    c = (y - y.min()) / (y.max() - y.min()),
    cmap = "copper",
)
# note that these plots are the weight vectors, not decision boundary
plt.plot([0, u[0]], [0, u[1]], lw = 10, label = "true solution")
ols = X.pinverse() @ y
plt.plot([0, ols[0]], [0, ols[1]], lw = 4, label = "ols")
plt.legend()

[22]: <matplotlib.legend.Legend at 0x7fb96e97c700>

# matplotlib has many features; here's a cute one to restyle a plot:
with plt.style.context("bmh"):
    plt.scatter(
        X[:, 0],

```python
X[:, i],
# color according to y:
c = (y - y.min()) / (y.max() - y.min()),
cmap = "copper",
)
```

[24]:  # more to come...