Double-click (or enter) to edit

COSE474-2024F: Deep Learning

2.1. Data Manipulation

2.1.1. Getting Started

```
import torch
```

```
x = torch.arange(12, dtype=torch.float32)
x
```

```
tensor([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.])
```

```
x.numel()
```

```
12
```

```
x.shape
```

```
torch.Size([12])
```

```
X = x.reshape(3,4)
X
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

```
torch.zeros((2, 3, 4))
```

```
tensor([[[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]],

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]])
```

```
torch.ones((2, 3, 4))
```

```
tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]],

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
```

```
torch.randn(3, 4)
```

```
tensor([[ 0.9408, -0.7046, -0.9199,  0.8976],
        [-0.0096,  0.0765,  0.7844, -1.0034],
        [-0.9448,  2.1987, -0.5243,  0.1473]])
```

```
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4],
        [4, 3, 2, 1]])
```

Double-click (or enter) to edit

2.1.2. Indexing and Slicing

```
X[-1], X[1:3]
```

```
(tensor([ 8.,  9., 10., 11.]),
 tensor([[ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.]]))
```

```
X[1, 2] = 17
X
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5., 17.,  7.],
        [ 8.,  9., 10., 11.]])
```

```
X[:2, :] = 12
X
```

```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8.,  9., 10., 11.]])
```

2.1.3 Operations

```
torch.exp(x)
```

```
tensor([162754.7969, 162754.7969, 162754.7969, 162754.7969, 162754.7969,
        162754.7969, 162754.7969, 162754.7969,   2980.9580,   8103.0840,
         22026.4648,  59874.1406])
```

Likewise, we denote *binary* scalar operators, which map pairs of real numbers to a (single) real number via the signature $f : \mathbb{R}, \mathbb{R} \to \mathbb{R}$. Given any two vectors $\mathbf{u}$ and $\mathbf{v}$ *of the same shape*, and a binary operator f, we can produce a vector $\mathbf{c} = F(\mathbf{u}, \mathbf{v})$ by setting $c_i \leftarrow f(u_i, v_i)$ for all i, where c_i, u_i, and v_i are the i^{th} elements of vectors $\mathbf{c}$, $\mathbf{u}$, and $\mathbf{v}$. Here, we produced the vector-valued $F : \mathbb{R}^d, \mathbb{R}^d \to \mathbb{R}^d$ by *lifting* the scalar function to an elementwise vector operation. The common standard arithmetic operators for addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and exponentiation (`**`) have all been *lifted* to elementwise operations for identically-shaped tensors of arbitrary shape.

```
x = torch.tensor([1.0, 2, 4, 8])
y = torch.tensor([2, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y
```

```
(tensor([ 3.,  4.,  6., 10.]),
 tensor([-1.,  0.,  2.,  6.]),
 tensor([ 2.,  4.,  8., 16.]),
 tensor([0.5000, 1.0000, 2.0000, 4.0000]),
 tensor([ 1.,  4., 16., 64.]))
```

```
X = torch.arange(12, dtype=torch.float32).reshape((3,4))
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
torch.cat((X, Y), dim=0), torch.cat((X, Y), dim=1)
```

```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [ 2.,  1.,  4.,  3.],
         [ 1.,  2.,  3.,  4.],
         [ 4.,  3.,  2.,  1.]]),
 tensor([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
         [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
         [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]]))
```

```
X == Y
```

```
tensor([[False,  True, False,  True],
        [False, False, False, False],
        [False, False, False, False]])
```

```
X.sum()
```

```
tensor(66.)
```

2.1.4 Broadcasting

```
a = torch.arange(3).reshape((3, 1))
b = torch.arange(2).reshape((1, 2))
a, b
```

```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

Since `a` and `b` are 3×1 and 1×2 matrices, respectively, their shapes do not match up. Broadcasting produces a larger 3×2 matrix by replicating matrix `a` along the columns and matrix `b` along the rows before adding them elementwise.

```
a + b
```

```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

2.1.5 Saving Memory

Running operations can cause new memory to be allocated to host results. For example, if we write Y = X + Y, we dereference the tensor that Y used to point to and instead point Y at the newly allocated memory. We can demonstrate this issue with Python's id() function, which gives us the exact address of the referenced object in memory. Note that after we run Y = Y + X, id(Y) points to a different location. That is because Python first evaluates Y + X, allocating new memory for the result and then points Y to this new location in memory.

```
before = id(Y)
Y = Y + X
id(Y) == before
```

```
False
```

This might be undesirable for two reasons.

- First, we do not want to run around allocating memory unnecessarily all the time. In machine learning, we often have hundreds of megabytes of parameters and update all of them multiple times per second. Whenever possible, we want to perform these updates in place.
- Second, we might point at the same parameters from multiple variables. If we do not update in place, we must be careful to update all of these references, lest we spring a memory leak or inadvertently refer to stale parameters.

Fortunately, performing in-place operations is easy. We can assign the result of an operation to a previously allocated array Y by using slice notation: Y[:] = . To illustrate this concept, we overwrite the values of tensor Z, after initializing it, using zeros_like, to have the same shape as Y.

```
Z = torch.zeros_like(Y)
print('id(Z):', id(Z))
Z[:] = X + Y
print('id(Z):', id(Z))
```

```
id(Z): 134456531408656
id(Z): 134456531408656
```

If the value of X is not reused in subsequent computations, we can also use X[:] = X + Y or X += Y to reduce the memory overhead of the operation.

```
before = id(X)
X += Y
id(X) == before
```

```
True
```

2.1.6. Conversion to Other Python Objects

```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

```
(numpy.ndarray, torch.Tensor)
```

```
a = torch.tensor([3.5])
a, a.item(), float(a), int(a)
```

```
(tensor([3.5000]), 3.5, 3.5, 3)
```

2.1.7. Summary

The tensor class is the main interface for storing and manipulating data in deep learning libraries. Tensors provide a variety of functionalities including construction routines; indexing and slicing; basic mathematics operations; broadcasting; memory-efficient assignment; and conversion to and from other Python objects.

2.1.8. Exercises

Run the code in this section. Change the conditional statement X == Y to X < Y or X > Y, and then see what kind of tensor you can get.

Replace the two tensors that operate by element in the broadcasting mechanism with other shapes, e.g., 3-dimensional tensors. Is the result the same as expected?

```
X = torch.arange(12, dtype=torch.float32).reshape((3, 4))
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])

result_lt = X < Y
result_gt = X > Y

print("X < Y:\n", result_lt)
print("X > Y:\n", result_gt)
```

```
X < Y:
 tensor([[ True, False,  True, False],
        [False, False, False, False],
        [False, False, False, False]])
X > Y:
 tensor([[False, False, False, False],
        [ True,  True,  True,  True],
        [ True,  True,  True,  True]])
```

```
a = torch.arange(6).reshape((2, 3, 1))
b = torch.arange(2).reshape((1, 1, 2))

result = a + b

print("Tensor a:\n", a)
print("Tensor b:\n", b)
print("Result of a + b:\n", result)
```

```
Tensor a:
 tensor([[[0],
         [1],
         [2]],

        [[3],
         [4],
         [5]]])
Tensor b:
 tensor([[[0, 1]]])
Result of a + b:
 tensor([[[0, 1],
         [1, 2],
         [2, 3]],

        [[3, 4],
```

```
                [4, 5],
                [5, 6]]])
```

Start coding or generate with AI.