PyTorch Fundamentals - Matrices

Matrices

Matrices Brief Introduction

- ✓ Basic definition: rectangular array of numbers.
- ✓ Tensors (PyTorch)
- ✓ Ndarrays (NumPy)

2 x 2 Matrix (R x C)

1	1
1	1

2 x 3 Matrix

1	1	1
1	1	1

Creating Matrices

Create list

```
# Creating a 2x2 array
arr = [[1, 2], [3, 4]]
print(arr)
```

```
[[1, 2], [3, 4]]
```

Create numpy array via list

```
import numpy as np
```

```
# Convert to NumPy
np.array(arr)
```

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

Convert numpy array to PyTorch tensor

```
import torch
```

```
# Convert to PyTorch Tensor
torch.Tensor(arr)
```

```
1  2
3  4
[torch.FloatTensor of size 2x2]
```

Create Matrices with Default Values

Create 2x2 numpy array of 1's

```
np.ones((2, 2))
```

```
array([[ 1.,  1.],
       [ 1.,  1.]])
```

Create 2x2 torch tensor of 1's

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

```
 1  1
 1  1
[torch.FloatTensor of size 2x2]
```

Create 2x2 numpy array of random numbers

```
np.random.rand(2, 2)
```

```
array([[ 0.68270631,  0.87721678],
       [ 0.07420986,  0.79669375]])
```

Create 2x2 PyTorch tensor of random numbers

```
torch.rand(2, 2)
```

```
0.3900  0.8268
0.3888  0.5914
[torch.FloatTensor of size 2x2]
```

Seeds for Reproducibility

Why do we need seeds?

We need seeds to enable reproduction of experimental results. This becomes critical later on where you can easily let people reproduce your code's output exactly as you've produced.

Create seed to enable fixed numbers for random number generation

```
# Seed
np.random.seed(0)
np.random.rand(2, 2)
```

```
array([[ 0.5488135 ,  0.71518937],
       [ 0.60276338,  0.54488318]])
```

Repeat random array generation to check

If you do not set the seed, you would not get the same set of numbers like here.

```
# Seed
np.random.seed(0)
np.random.rand(2, 2)
```

```
array([[ 0.5488135 ,  0.71518937],
       [ 0.60276338,  0.54488318]])
```

Create a numpy array without seed

Notice how you get different numbers compared to the first 2 tries?

```
# No seed
np.random.rand(2, 2)
```

```
array([[ 0.56804456,  0.92559664],
       [ 0.07103606,  0.0871293 ]])
```

Repeat numpy array generation without seed

You get the point now, you get a totally different set of numbers.

```
# No seed
np.random.rand(2, 2)
```

```
array([[ 0.0202184 ,  0.83261985],
       [ 0.77815675,  0.87001215]])
```

Create a PyTorch tensor with a fixed seed

```
# Torch Seed
torch.manual_seed(0)
torch.rand(2, 2)
```

Repeat creating a PyTorch fixed seed tensor

```
# Torch Seed
torch.manual_seed(0)
torch.rand(2, 2)
```

```
0.5488  0.5928
0.7152  0.8443
[torch.FloatTensor of size 2x2]
```

Creating a PyTorch tensor without seed

Like with a numpy array of random numbers without seed, you will not get the same results as above.

```
# Torch No Seed
torch.rand(2, 2)
```

```
0.6028  0.8579
0.5449  0.8473
[torch.FloatTensor of size 2x2]
```

Repeat creating a PyTorch tensor without seed

Notice how these are different numbers again?

```
# Torch No Seed
torch.rand(2, 2)
```

```
0.4237  0.6236
0.6459  0.3844
[torch.FloatTensor of size 2x2]
```

Seed for GPU is different for now...

Fix a seed for GPU tensors

When you conduct deep learning experiments, typically you want to use GPUs to accelerate your computations and fixing seed for tensors on GPUs is different from CPUs as we have done above.

```
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(0)
```

NumPy and Torch Bridge

NumPy to Torch

Create a numpy array of 1's

```
# Numpy array
np_array = np.ones((2, 2))
```

```
print(np_array)
```

```
[[ 1.  1.]
[ 1.  1.]]
```

Get the type of class for the numpy array

```
print(type(np_array))
```

```
<class 'numpy.ndarray'>
```

Convert numpy array to PyTorch tensor

```
# Convert to Torch Tensor
torch_tensor = torch.from_numpy(np_array)
```

```
print(torch_tensor)
```

```
 1  1
 1  1
[torch.DoubleTensor of size 2x2]
```

Get type of class for PyTorch tensor

Notice how it shows it's a torch DoubleTensor? There're actually tensor types and it depends on the numpy data type.

```
print(type(torch_tensor))
```

```
<class 'torch.DoubleTensor'>
```

Create PyTorch tensor from a different numpy datatype

You will get an error running this code because PyTorch tensor don't support all datatype.

```
# Data types matter: intentional error
np_array_new = np.ones((2, 2), dtype=np.int8)
torch.from_numpy(np_array_new)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)

<ipython-input-57-b8b085f9b39d> in <module>()
      1 # Data types matter
      2 np_array_new = np.ones((2, 2), dtype=np.int8)
----> 3 torch.from_numpy(np_array_new)


RuntimeError: can't convert a given np.ndarray to a tensor - it has an invalid type. The only
supported types are: double, float, int64, int32, and uint8.
```

What conversion support does Numpy to PyTorch tensor bridge gives?

- `double`
- `float`
- `int64`, `int32`, `uint8`

Create PyTorch long tensor

See how a int64 numpy array gives you a PyTorch long tensor?

```
# Data types matter
np_array_new = np.ones((2, 2), dtype=np.int64)
torch.from_numpy(np_array_new)
```

```
1  1
1  1
[torch.LongTensor of size 2x2]
```

Create PyTorch int tensor

```
# Data types matter
np_array_new = np.ones((2, 2), dtype=np.int32)
torch.from_numpy(np_array_new)
```

```
1  1
1  1
[torch.IntTensor of size 2x2]
```

Create PyTorch byte tensor

```
# Data types matter
np_array_new = np.ones((2, 2), dtype=np.uint8)
torch.from_numpy(np_array_new)
```

```
1  1
1  1
[torch.ByteTensor of size 2x2]
```

Create PyTorch Double Tensor

```
# Data types matter
np_array_new = np.ones((2, 2), dtype=np.float64)
torch.from_numpy(np_array_new)
```

Alternatively you can do this too via `np.double`

```
# Data types matter
np_array_new = np.ones((2, 2), dtype=np.double)
torch.from_numpy(np_array_new)
```

```
1  1
1  1
[torch.DoubleTensor of size 2x2]
```

Create PyTorch Float Tensor

```
# Data types matter
np_array_new = np.ones((2, 2), dtype=np.float32)
torch.from_numpy(np_array_new)
```

```
1  1
1  1
[torch.FloatTensor of size 2x2]
```

Tensor Type Bug Guide

These things don't matter much now. But later when you see error messages that require these particular tensor types, refer to this guide!

NumPy Array Type	Torch Tensor Type
int64	LongTensor
int32	IntegerTensor
uint8	ByteTensor
float64	DoubleTensor
float32	FloatTensor
double	DoubleTensor

Torch to NumPy

Create PyTorch tensor of 1's

You would realize this defaults to a float tensor by default if you do this.

```
torch_tensor = torch.ones(2, 2)
```

```
type(torch_tensor)
```

```
torch.FloatTensor
```

Convert tensor to numpy

It's as simple as this.

```
torch_to_numpy = torch_tensor.numpy()
```

```
type(torch_to_numpy)
```

```
# Wowza, we did it.
numpy.ndarray
```

Tensors on CPU vs GPU

Move tensor to CPU and back

This by default creates a tensor on CPU. You do not need to do anything.

```
# CPU
tensor_cpu = torch.ones(2, 2)
```

If you would like to send a tensor to your GPU, you just need to do a simple `.cuda()`

```
# CPU to GPU
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
tensor_cpu.to(device)
```

And if you want to move that tensor on the GPU back to the CPU, just do the following.

```
# GPU to CPU
tensor_cpu.cpu()
```

Tensor Operations

Resizing Tensor

Creating a 2x2 tensor

```
a = torch.ones(2, 2)
print(a)
```

```
1  1
1  1
[torch.FloatTensor of size 2x2]
```

Getting size of tensor

```
print(a.size())
```

```
torch.Size([2, 2])
```

Resize tensor to 4x1

```
a.view(4)
```

```
1
1
1
1
[torch.FloatTensor of size 4]
```

Get size of resized tensor

```
a.view(4).size()
```

```
torch.Size([4])
```

Element-wise Addition

Creating first 2x2 tensor

```
a = torch.ones(2, 2)
print(a)
```

```
1  1
1  1
[torch.FloatTensor of size 2x2]
```

Creating second 2x2 tensor

```
b = torch.ones(2, 2)
print(b)
```

```
1  1
1  1
[torch.FloatTensor of size 2x2]
```

Element-wise addition of 2 tensors

```
# Element-wise addition
c = a + b
print(c)
```

```
 2  2
 2  2
[torch.FloatTensor of size 2x2]
```

Alternative element-wise addition of 2 tensors

```
# Element-wise addition
c = torch.add(a, b)
print(c)
```

```
 2  2
 2  2
[torch.FloatTensor of size 2x2]
```

In-place element-wise addition

This would replace the c tensor values with the new addition.

```
# In-place addition
print('Old c tensor')
print(c)

c.add_(a)

print('-'*60)
print('New c tensor')
print(c)
```

```
Old c tensor

 2  2
 2  2
[torch.FloatTensor of size 2x2]

------------------------------------------------------------
```

```
New c tensor

 3  3
 3  3
[torch.FloatTensor of size 2x2]
```

Element-wise Subtraction

Check values of tensor a and b'

Take note that you've created tensor a and b of sizes 2x2 filled with 1's each above.

```
print(a)
print(b)
```

```
 1  1
 1  1
[torch.FloatTensor of size 2x2]


 1  1
 1  1
[torch.FloatTensor of size 2x2]
```

Element-wise subtraction: method 1

```
a - b
```

```
0  0
0  0
[torch.FloatTensor of size 2x2]
```

Element-wise subtraction: method 2

```
# Not in-place
print(a.sub(b))
print(a)
```

```
0  0
0  0
[torch.FloatTensor of size 2x2]


1  1
1  1
[torch.FloatTensor of size 2x2]
```

Element-wise subtraction: method 3

This will replace a with the final result filled with 2's

```
# Inplace
print(a.sub_(b))
print(a)
```

```
0  0
0  0
[torch.FloatTensor of size 2x2]


0  0
0  0
[torch.FloatTensor of size 2x2]
```

Element-Wise Multiplication

Create tensor a and b of sizes 2x2 filled with 1's and 0's

```
a = torch.ones(2, 2)
print(a)
b = torch.zeros(2, 2)
print(b)
```

```
1  1
1  1
[torch.FloatTensor of size 2x2]


0  0
0  0
[torch.FloatTensor of size 2x2]
```

Element-wise multiplication: method 1

```
a * b
```

```
0  0
0  0
[torch.FloatTensor of size 2x2]
```

Element-wise multiplication: method 2

```
# Not in-place
print(torch.mul(a, b))
print(a)
```

```
0  0
0  0
[torch.FloatTensor of size 2x2]

1  1
1  1
[torch.FloatTensor of size 2x2]
```

Element-wise multiplication: method 3

```
# In-place
print(a.mul_(b))
print(a)
```

```
0  0
0  0
[torch.FloatTensor of size 2x2]

0  0
0  0
[torch.FloatTensor of size 2x2]
```

Element-Wise Division

Create tensor a and b of sizes 2x2 filled with 1's and 0's

```
a = torch.ones(2, 2)
print(a)
b = torch.zeros(2, 2)
print(b)
```

```
1  1
1  1
[torch.FloatTensor of size 2x2]


0  0
0  0
[torch.FloatTensor of size 2x2]
```

Element-wise division: method 1

```
b / a
```

```
0  0
0  0
[torch.FloatTensor of size 2x2]
```

Element-wise division: method 2

```
torch.div(b, a)
```

```
0  0
0  0
[torch.FloatTensor of size 2x2]
```

Element-wise division: method 3

```
# Inplace
b.div_(a)
```

```
0  0
0  0
[torch.FloatTensor of size 2x2]
```

Tensor Mean

$$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 = 55$$

$$mean = 55/10 = 5.5$$

Create tensor of size 10 filled from 1 to 10

```
a = torch.Tensor([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
a.size()
```

```
torch.Size([10])
```

Get tensor mean

Here we get 5.5 as we've calculated manually above.

```
a.mean(dim=0)
```

```
5.5000
[torch.FloatTensor of size 1]
```

Get tensor mean on second dimension

Here we get an error because the tensor is of size 10 and not 10x1 so there's no second dimension to calculate.

```
a.mean(dim=1)
```

```
RuntimeError                              Traceback (most recent call last)

<ipython-input-7-81aec0cf1c00> in <module>()
----> 1 a.mean(dim=1)


RuntimeError: dimension out of range (expected to be in range of [-1, 0], but got 1)
```

Create a 2x10 Tensor, of 1-10 digits each

```
a = torch.Tensor([[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]])
```

```
a.size()
```

```
torch.Size([2, 10])
```

Get tensor mean on second dimension

Here we won't get an error like previously because we've a tensor of size 2x10

```
a.mean(dim=1)
```

```
 5.5000
 5.5000
[torch.FloatTensor of size 2x1]
```

Tensor Standard Deviation

Get standard deviation of tensor

```
a = torch.Tensor([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
a.std(dim=0)
```

```
 3.0277
[torch.FloatTensor of size 1]
```

Summary

We've learnt to...

Success

- ✓ Create Matrices
- ✓ Create Matrices with Default Initialization Values
    - ✓ Zeros

    - [x] Ones
- [x] Initialize Seeds for Reproducibility on GPU and CPU
- [x] Convert Matrices: NumPy to Torch and Torch to NumPy
- [x] Move Tensors: CPU to GPU and GPU to CPU
- [x] Run Important Tensor Operations
    - [x] Element-wise addition, subtraction, multiplication and division
    - [x] Resize
    - [x] Calculate mean
    - [x] Calculate standard deviation

Comments