Machine Intelligence (UE21CS352A)

Dr. Uma D
Email: umaprabha@pes.edu
Teaching Assistants:
mohannitin28@gmail.com
pracheth.thamankar@gmail.com

PyTorch Tutorial

PyTorch is an optimized **tensor library** for deep learning using GPUs and CPUs. developed by the team at Facebook and open sourced on GitHub in 2017. It provides a flexible and efficient platform for building and training machine learning and deep learning models.

Prerequisites

- Make sure Python version is 3.8 or greater.
- It is recommended, **but not required**, that your system has an NVIDIA GPU in order to harness the full power of PyTorch's CUDA support. To use CUDA follow Nvidia's Installation Guide for Windows and Linux.
- All tasks performed today can be done using CPU alone. PyTorch defaults to storing tensors on cpu.

Installation

To get started with PyTorch, you need to have it installed on your system.

Windows
To install without CUDA
```
pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cpu
```
To install with CUDA
```
pip3 install torch torchvision torchaudio
```

Mac
CUDA is not available on MacOS, therefore default package is installed
```
pip3 install torch torchvision torchaudio
```

Linux

To install without CUDA

```
pip3 install torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cpu
```

To install with CUDA

```
pip3 install torch torchvision torchaudio
```

For further details, you can also use the start locally guide on PyTorch's website.

Tensors

Tensors are a specialized data structure that are very similar to arrays and matrices. In PyTorch, we use tensors to encode the inputs and outputs of a model, as well as the model's parameters. Tensors are similar to NumPy's ndarrays, except that tensors can run on GPUs. To use Tensors and other PyTorch functionality first import torch.

```
import torch
```

Tensor Creation

Directly from data

```
data = [[1, 2], [3, 4]]
x_data = torch.tensor(data)
print(x_data)
```

From Numpy array

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
print(x_np)
```

From other tensors

```
x_ones = torch.ones_like(x_data) # retains the properties of x_data
print(f"Ones Tensor: \n {x_ones} \n")

x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the datatype of x_data
print(f"Random Tensor: \n {x_rand} \n")
```

Populating tensors with random/constant values

```
shape = (2, 3,)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)
torch.manual_seed(1789)
rand_tensor = torch.rand(shape)
empty_tensor = torch.empty(shape) # memory is allocated, tensor populated with garbage-values
print(f"Random Tensor: \n {rand_tensor} \n")
print(f"Ones Tensor: \n {ones_tensor} \n")
print(f"Zeros Tensor: \n {zeros_tensor}")
print(f"Empty Tensor: \n {empty_tensor}")
```

Using seed allows you to reproduce same random tensors across runs. This is especially useful in research settings - where you'll want some assurance of the reproducibility of your results.

Tensor Attributes

Tensor attributes describe their shape, datatype, and the device on which they are stored.

1. Shape - Extent of each dimension of a tensor

```
x = torch.empty(2, 2, 3)
print(x.shape)
print(x)
```

2. Datatype - An object that represents the data type of a `torch.Tensor`. PyTorch has twelve different data types.

```
a = torch.ones((2, 3), dtype=torch.int16)
print(a.dtype)
```

3. Device - Object representing the device on which a `torch.Tensor` is or will be allocated

```
tensor = torch.rand(3,2)
print(f"Device tensor is stored on: {tensor.device}")
# We can move our tensor to the GPU if available
```

Operations on Tensors

PyTorch defines over 100 tensor operations, including arithmetic, linear algebra, matrix manipulation (transposing, indexing, slicing), sampling and more.

Note :
Each of these operations can be run on the GPU (at typically higher speeds than on a CPU). By default, tensors are created on the CPU. We need to explicitly move tensors to the GPU using `.to` method (after checking for GPU availability).

```
# We move our tensor to the GPU if available
if torch.cuda.is_available():
    tensor = tensor.to("cuda")
```

Numpy-like indexing and slicing

```
tensor = torch.ones(4, 4)
print(f"First row: {tensor[0]}")
print(f"First column: {tensor[:, 0]}")
print(f"Last column: {tensor[:, -1]}")
tensor[:,1] = 0
print(tensor)
```

Arithmetic operations

```
ones = torch.zeros(2, 2) + 1
twos = torch.ones(2, 2) * 2
threes = (torch.ones(2, 2) * 7 - 1) / 2
fours = twos ** 2
sqrt2s = twos ** 0.5

print(ones)
print(twos)
print(threes)
print(fours)
print(sqrt2s)
```

Similar operations can be performed between two tensors:

```
powers2 = [twos ** torch.tensor([[1, 2], [3, 4]])
print(powers2)

fives= ones + fours
print(fives)

dozens = threes * fours
print(dozens)
```

Note : Please make sure to check the shapes of the tensors before performing these operations, otherwise they'll result in errors.

Tensor Broadcasting

The exception to the same-shapes rule is *tensor broadcasting*.
Broadcasting is a way to perform an operation between tensors that have similarities in their shapes.

```
rand = torch.rand(2,4)
doubled  = rand * (torch.ones(1,4)*2)

print(rand)
print(doubled)
```

In the above example , the 1x4 tensor is multiplied by *both rows* of the 2x4 tensor.
The rules for broadcasting are:

1. Each tensor must have at least one dimension - no empty tensors.
2. Comparing the dimension sizes of the two tensors, _going from last to first:
    1. Each dimension must be equal, *or*
    2. One of the dimensions must be of size 1, *or*
    3. The dimension does not exist in one of the tensors.

Tensors of identical shape, of course, are trivially “broadcastable”.
Here are some examples of situations that honor the above rules and allow broadcasting:

```
a = torch.ones(4, 3, 2)
b = a * torch.rand(   3, 2) # 3rd & 2nd dims identical to a, dim 1 absent
print(b)
c = a * torch.rand(   3, 1) # 3rd dim = 1, 2nd dim identical to a
print(c)
d = a * torch.rand(   1, 2) # 3rd dim identical to a, 2nd dim = 1
print(d)
```

Common Functions

```
a = torch.rand(2, 4) * 2 - 1
print(torch.abs(a))
print(torch.ceil(a))
print(torch.floor(a))
print(torch.clamp(a, -0.5, 0.5))
```

Trigonometric functions and their inverses

```
angles = torch.tensor([0, math.pi / 4, math.pi / 2, 3 * math.pi / 4])
sines = torch.sin(angles)
inverses = torch.asin(sines)
print('\nSine and arcsine:')
print(angles)
print(sines)
print(inverses)
```

Bitwise operations

```
b = torch.tensor([1, 5, 11])
c = torch.tensor([2, 7, 10])

print(torch.bitwise_xor(b, c))
```

Comparisons

```
print('\nBroadcasted, element-wise equality comparison:')

d = torch.tensor([[1., 2.], [3., 4.]])
e = torch.ones(1, 2)  # many comparison ops support broadcasting!

print(torch.eq(d, e)) # returns a tensor of type bool
```

Reductions

```
d = torch.tensor([[1., 2.], [3., 4.]])
print(torch.max(d))        # returns a single-element tensor

print(torch.max(d).item()) # extracts the value from the returned tensor

print(torch.mean(d))       # average

print(torch.std(d))        # standard deviation

print(torch.prod(d))       # product of all numbers

print(torch.unique(torch.tensor([1, 2, 1, 2, 1, 2]))) # filter unique elements
```

Vector and linear algebra operations

```
v1 = torch.tensor([1., 0., 0.])         # x unit vector
v2 = torch.tensor([0., 1., 0.])         # y unit vector
```

```
m1 = torch.rand(2, 2)                   # random matrix
m2 = torch.tensor([[3., 0.], [0., 3.]]) # three times identity matrix

print(torch.cross(v2, v1)) # returns cross product of vectors v1 and v2
print(m1)

m3 = torch.matmul(m1, m2) # same as m3 = m1@m2

print(m3)                 # 3 times m1
print(torch.svd(m3))      # singular value decomposition
```

Altering Tensors in Place

Most binary operations on tensors will return a third, new tensor.When we say `c = a * b` (where `a` and `b` are tensors), the new tensor `c` will occupy a region of memory distinct from the other tensors.

There are times, though, that you may wish to alter a tensor in place. For this, most of the math functions have a version with an appended underscore (`_`) that will alter a tensor in place.

```
a = torch.tensor([0, math.pi / 4, math.pi / 2, 3 * math.pi / 4])
print('a:')
print(a)

print(torch.sin(a))   # this operation creates a new tensor in memory
print(a)              # a has not changed

b = torch.tensor([0, math.pi / 4, math.pi / 2, 3 * math.pi / 4])
print('\nb:')
print(b)
print(torch.sin_(b))  # note the underscore
print(b)              # b has changed
```

For arithmetic operations, there are functions that behave similarly:

```
a = torch.ones(2, 2)
b = torch.rand(2, 2)

print('Before:')
print(a)
print(b)

print('\nAfter adding:')

print(a.add_(b))
```

```
print(a)
print(b)

print('\nAfter multiplying')

print(b.mul_(b))
print(b)
```

There is another option for placing the result of a computation in an existing, allocated tensor. Many of the methods and functions we've seen so far - including creation methods! - have an `out` argument that lets you specify a tensor to receive the output. If the `out` tensor is the correct shape and `dtype`, this can happen without a new memory allocation:

```
a = torch.rand(2, 2)
b = torch.rand(2, 2)
c = torch.zeros(2, 2)
old_id = id(c)
print(c)

d = torch.matmul(a, b, out=c)
print(c)                # contents of c have changed

assert c is d           # test c & d are same object, not just containing equal values
assert id(c) == old_id  # make sure that our new c is the same object as the old one

torch.rand(2, 2, out=c) # works for creation too!
print(c)                # c has changed again

assert id(c) == old_id  # still the same object!
```

Manipulating Tensor shape

```
a = torch.tensor([1,2,3,4])
torch.reshape(a, (2, 2))
print(a)

b = torch.rand(56,56) # Consider 56x56 image
c = b.unsqueeze(0) # unsqueeze(i) adds dimension of length 1 at index i of shape of
tensor
print(c) # c is now a batch of 1 image of shape 28x28

d = torch.rand(1, 20)
print(d.shape)

e = d.squeeze(0) # squeeze(i) removes a dimension if shape[i] is 1
print(e.shape)
```

```
x,y,z = torch.rand(2,3),torch.rand(2,3),torch.rand(2,3)
cat_tensor = torch.cat((x,y,z),dim = 0) # added as rows
print(cat_tensor)
```

Note: The way to understand the “axis” or "dim" of a torch function is that it collapses the specified axis.

Copying Tensors

Assigning a tensor to a variable makes the variable a *label* of the tensor, and does not copy it. For example:

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

a[0][1] = 561  # we change a...

print(b)       # ...and b is also altered
```

But what if you want a separate copy of the data to work on? The `clone()` method is there for you:

```
a = torch.ones(2, 2)
b = a.clone()

assert b is not a      # different objects in memory...
print(torch.eq(a, b))  # ...but still with the same contents!

a[0][1] = 561          # a changes...

print(b)               # ...but b is still all ones
```

Moving to GPU

Note: First, we should check whether a GPU is available, with the `is_available()` method.

```
if torch.cuda.is_available():
    print('We have a GPU!')
else:
    print('Sorry, CPU only.')
```

There are multiple ways to get your data onto your target device. You may do it at creation time:

```
if torch.cuda.is_available():
    my_device = torch.device('cuda')
else:
    my_device = torch.device('cpu')
```

```
x = torch.rand(2, 2, device=my_device)

print(x)
```

If you have an existing tensor living on one device, you can move it to another with the `to()` method.

```
y = torch.rand(2, 2)
y = y.to(my_device)
```

Numpy Bridge

If you have existing ML or scientific code with data stored in NumPy ndarrays, you may wish to express that same data as PyTorch tensors, whether to take advantage of PyTorch's GPU acceleration, or its efficient abstractions for building ML models. It's easy to switch between ndarrays and PyTorch tensors:

```
import numpy as np

numpy_array = np.ones((2, 3))
print(numpy_array)

pytorch_tensor = torch.from_numpy(numpy_array) # converting ndarray to tensor
print(pytorch_tensor)

pytorch_rand = torch.rand(2, 3)
print(pytorch_rand)

numpy_rand = pytorch_rand.numpy() #converting tensor to ndarray
print(numpy_rand)
```

It is important to know that these converted objects are using *the same underlying memory* as their source objects, meaning that changes to one are reflected in the other:

```
numpy_array[1, 1] = 23
print(pytorch_tensor)

pytorch_rand[1, 1] = 17
print(numpy_rand)
```

Questions

1. Obtain a tensor containing only zeroes from the given tensor

```
pattern = torch.tensor([
                [1,1,1,1],
                [1,0,0,1],
                [1,0,0,1],
                [1,1,1,1]
])
```

2. Create a Numpy array of shape (1,3,3) using PyTorch
3. Create two random (2,2,2) tensors and find the max, min, mean, std of their product (matrix multiplication)
4. Convert a 16x16 tensor into 1x256 tensor
5. Given two tensors x and Y, find the coefficients that best model the linear relationship Y = ax + b (Linear Regression)

$$b = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(Y_i - \bar{Y})}{\sum_{i-1}^{n}(x_i - \bar{x})^2}$$

$$a = \bar{Y} - b\bar{x}$$