Pytorch Tutorial

Isabella Liu

Part1: Basic Concepts

What is Pytorch

- A more advanced Numpy with **GPU** support and other accelerations.
- An **automatic differentiation** library which is handy for Deep Learning.

Tensors

- Basic data structure in Pytorch
  - Similar to arrays, matrices, like np.ndarray
  - With more features (it can carry gradient)
- In Pytorch, all data are used as Tensors
  - Inputs
  - Outputs of the models
  - Parameters of the models
  - etc.

Tensors: Creation

Create Tensors:

- From list
- From numpy array
- Use some provided functions:
  - Rand
  - Ones
  - Zeros

```
data = [1,7,6]
tensor = torch.tensor(data)
print(tensor)
```

```
tensor([1, 7, 6])
```

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

```
tensor([1, 7, 6])
```

```
shape = (1,2)
rand_t = torch.rand(shape)
ones_t = torch.ones(shape)
zeros_t = torch.zeros(shape)

print(rand_t)
print(ones_t)
print(zeros_t)
```

```
tensor([[0.7883, 0.1005]])
tensor([[1., 1.]])
tensor([[0., 0.]])
```

Tensors: Conversion

- Host different types of data
  - Float
  - Double
  - Long
- Host data on different devices
  - CPU
  - GPU

```
a = torch.Tensor([1,7,6])
print(a.dtype)
print(a.double().dtype)
print(a.long().dtype)

b = torch.LongTensor([1,7,6])
print(b.dtype)

# Using GPU
print(a.to("cuda:0").device)
```

```
torch.float32
torch.float64
torch.int64
torch.int64
cuda:0
```

Tensors: Operation

Most of operations we used in numpy are supported:

- Slicing
- Concat
- Broadcasting
- Computation (e.g. Multiply)
- Tensor copying

- Convert from and to numpy

```
shape = (2,2)
a = torch.ones(shape)
b = torch.zeros(shape)
print("Slice:")
print(a[:, 1])
print("Concat:")
print(torch.cat([a,b]))
print("Broadcast:")
print(torch.ones((2,1)) * torch.ones(1, 2))
print("Compute:")
print(a * b)
```

```
Slice:
tensor([1., 1.])
Concat:
tensor([[1., 1.],
        [1., 1.],
        [0., 0.],
        [0., 0.]])
Broadcast:
tensor([[1., 1.],
        [1., 1.]])
Compute:
tensor([[0., 0.],
        [0., 0.]])
```

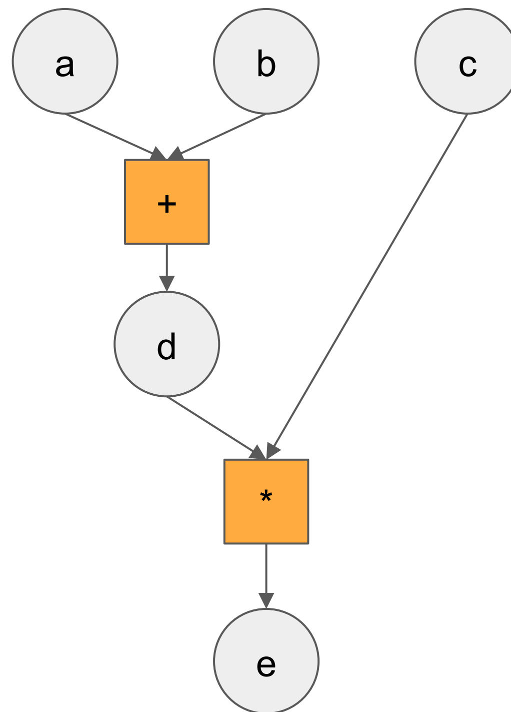
Autograd & Computation Graph

Computation sequence in Pytorch

-> Computation Graph

Autograd: Automatic gradient computation

- Pytorch handle the gradient flow automatically
- We only need to perform computation as usual

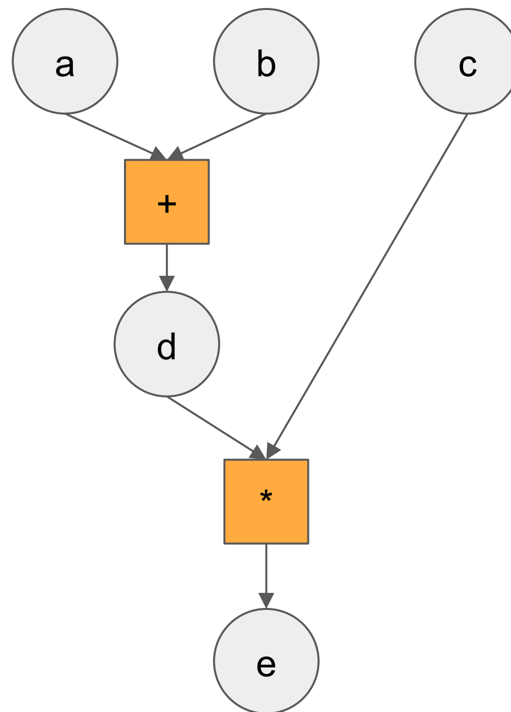


Autograd: Example

```
a = torch.tensor([1.], requires_grad=True)
b = torch.tensor([7.], requires_grad=True)
c = torch.tensor([6.], requires_grad=True)

d = a + b
e = c * d

e.backward()
print(a.grad)
print(b.grad)
print(c.grad)
```

```
tensor([6.])
tensor([6.])
tensor([8.])
```

$$e=(a+b)*c$$



Optim Module: Optimization

Optim module handles the optimization part during the learning.

Optimizer in Optim module:

- Update parameters according to optimization method: like SGD, Adam
- Common parameters:
  - Learning rate
  - Weight decay

Optim: Example

Use previous example to show how optim module works

```
a = torch.tensor([1.], requires_grad=True)
b = torch.tensor([7.], requires_grad=True)
c = torch.tensor([6.], requires_grad=True)

optim = torch.optim.SGD(
    [a, b],
    lr = 1e-3
)

d = a + b
e = c * d

optim.zero_grad()
e.backward()
optim.step()

print(a)
print(b)
print(c)
print(a.grad)
print(b.grad)
print(c.grad)
```

```
tensor([0.9940], requires_grad=True)
tensor([6.9940], requires_grad=True)
tensor([6.], requires_grad=True)
tensor([6.])
tensor([6.])
tensor([8.])
```

Useful Links

- Official Tutorial: https://pytorch.org/tutorials/
- Pytorch Youtube Series: https://www.youtube.com/playlist?list=PL_lsbAsL_o2CTlGHgMxNrKhzP97BaG9ZN

Part 2: Modules

NN Module: Neural Network

NN Module provide implementation of common layers:

- Linear, Conv2d, RNN, etc

Layers in NN module:

- Keep track of parameters
- Handle the computation

NN Module: Example

- Linear
  - Parameter: weight & bias
  - Computation: linear transformation

```
linear = torch.nn.Linear(5, 10)
input_t = torch.rand(2, 5)
output_t = linear(input_t)

s = torch.sum(output_t)
s.backward()

print("Output:")
print(output_t.shape)
print("Parameters:")
print("Weight:", linear.weight.shape)
print("Bias:", linear.bias.shape)

print("Weight Gradient:", linear.weight.grad)
print("Bias   Gradient:", linear.bias.grad)
```

```
Output:
torch.Size([2, 10])
Parameters:
Weight: torch.Size([10, 5])
Bias: torch.Size([10])
Weight Gradient: tensor([[0.8517, 1.0757, 1.5897, 1.4474,
1.2911],
        [0.8517, 1.0757, 1.5897, 1.4474, 1.2911],
        [0.8517, 1.0757, 1.5897, 1.4474, 1.2911],
        [0.8517, 1.0757, 1.5897, 1.4474, 1.2911],
        [0.8517, 1.0757, 1.5897, 1.4474, 1.2911],
        [0.8517, 1.0757, 1.5897, 1.4474, 1.2911],
        [0.8517, 1.0757, 1.5897, 1.4474, 1.2911],
        [0.8517, 1.0757, 1.5897, 1.4474, 1.2911],
        [0.8517, 1.0757, 1.5897, 1.4474, 1.2911],
        [0.8517, 1.0757, 1.5897, 1.4474, 1.2911]])
Bias   Gradient: tensor([2., 2., 2., 2., 2., 2., 2., 2.,
2., 2.])
```

NN Module: Conv2D

- Conv2D: Convolutional layer
  - Parameter: weight & bias
  - Computation: convolution between vectors
- Out = (W-K+2P)/S + 1
  - W: Input width
  - K: Kernel
  - P: Padding
  - S: Stride

```
conv = torch.nn.Conv2d(3, 32, 3, stride=1, padding=0)
input_t = torch.rand(10, 3, 32, 32)
output_t = conv(input_t)

print("Output:")
print(output_t.shape)
print("Parameters:")
print("Weights:",conv.weight.shape)
print("Bias:", conv.bias.shape)
```

```
Output:
torch.Size([10, 32, 30, 30])
Parameters:
Weights: torch.Size([32, 3, 3, 3])
Bias: torch.Size([32])
```

(32-3+0)/1 + 1 = 30

Animation example: https://github.com/vdumoulin/conv_arithmetic

NN Module: Conv2D

- Conv2D: Convolutional layer
  - Parameter: weight & bias
  - Computation: convolution between vectors
- Out = (W-K+2P)/S + 1
  - W: Input width
  - K: Kernel
  - P: Padding
  - S: Stride

```
conv = torch.nn.Conv2d(3, 32, 3, stride=1, padding=1)
input_t = torch.rand(10, 3, 32, 32)
output_t = conv(input_t)

print("Output:")
print(output_t.shape)
print("Parameters:")
print("Weights:",conv.weight.shape)
print("Bias:", conv.bias.shape)
```

```
Output:
torch.Size([10, 32, 32, 32])
Parameters:
Weights: torch.Size([32, 3, 3, 3])
Bias: torch.Size([32])
```

(32-3+2)/1 + 1 = 32

NN Module: Conv2D

- Conv2D: Convolutional layer
  - Parameter: weight & bias
  - Computation: convolution between vectors
- Out = (W-K+2P)/S + 1
  - W: Input width
  - K: Kernel
  - P: Padding
  - S: Stride

```
conv = torch.nn.Conv2d(3, 32, 3, stride=2)
input_t = torch.rand(10, 3, 32, 32)
output_t = conv(input_t)

print("Output:")
print(output_t.shape)
print("Parameters:")
print("Weights:",conv.weight.shape)
print("Bias:", conv.bias.shape)
```

```
Output:
torch.Size([10, 32, 15, 15])
Parameters:
Weights: torch.Size([32, 3, 3, 3])
Bias: torch.Size([32])
```

$(32-3+0)/2 + 1 = 15$

NN Module: Conv2D Example

- Conv2D: Convolutional layer
  - Parameter: weight & bias
  - Computation: convolution between vectors
- Out = (W-K+2P)/S + 1
  - W: Input width
  - K: Kernel
  - P: Padding
  - S: Stride

```
conv = torch.nn.Conv2d(3, 32, 3, stride=1, padding=1)
input_t = torch.rand(10, 3, 32, 32,)
output_t = conv(input_t)

s = torch.sum(output_t)
s.backward()

print("Output:")
print(output_t.shape)
print("Parameters:")
print("Weights:",conv.weight.shape)
print("Bias:", conv.bias.shape)

print("Weight gradient:", conv.weight.grad.shape)
print("Bias gradient: ", conv.bias.grad.shape)
```

```
Output:
torch.Size([10, 32, 32, 32])
Parameters:
Weights: torch.Size([32, 3, 3, 3])
Bias: torch.Size([32])
Weight gradient: torch.Size([32, 3, 3, 3])
Bias gradient:  torch.Size([32])
```

NN Module: Build model with Sequential

- Sequential: Combine multiple layers
- Create the model for your NN

```python
from torch import nn

model = nn.Sequential(nn.Conv2d(1,20,5),
            nn.ReLU(),
            nn.Conv2d(20,64,5),
            nn.ReLU()
          )
```

```python
model
```

```
Sequential(
  (0): Conv2d(1, 20, kernel_size=(5, 5), stride=(1, 1))
  (1): ReLU()
  (2): Conv2d(20, 64, kernel_size=(5, 5), stride=(1, 1))
  (3): ReLU()
)
```

NN Module: Build model with Sequential

- Sequential: Combine multiple layers
- Create the model for your NN

```
# Example of using Sequential with OrderedDict
from collections import OrderedDict

model = nn.Sequential(OrderedDict([
          ('conv1', nn.Conv2d(1,20,5)),
          ('relu1', nn.ReLU()),
          ('conv2', nn.Conv2d(20,64,5)),
          ('relu2', nn.ReLU())
        ]))
```

```
model
```

```
Sequential(
  (conv1): Conv2d(1, 20, kernel_size=(5, 5), stride=(1, 1))
  (relu1): ReLU()
  (conv2): Conv2d(20, 64, kernel_size=(5, 5), stride=(1, 1))
  (relu2): ReLU()
)
```

DataParallel

- Very easy to use multiple GPUs
- Run computations in parallel

```
# Put your model on a GPU
device = torch.device("cuda:0")
model.to(device)

# Copy your tensors to the GPU
mytensor = my_tensor.to(device)

# Run operations on Multiple GPUs parallely
model = nn.DataParallel(model)
```

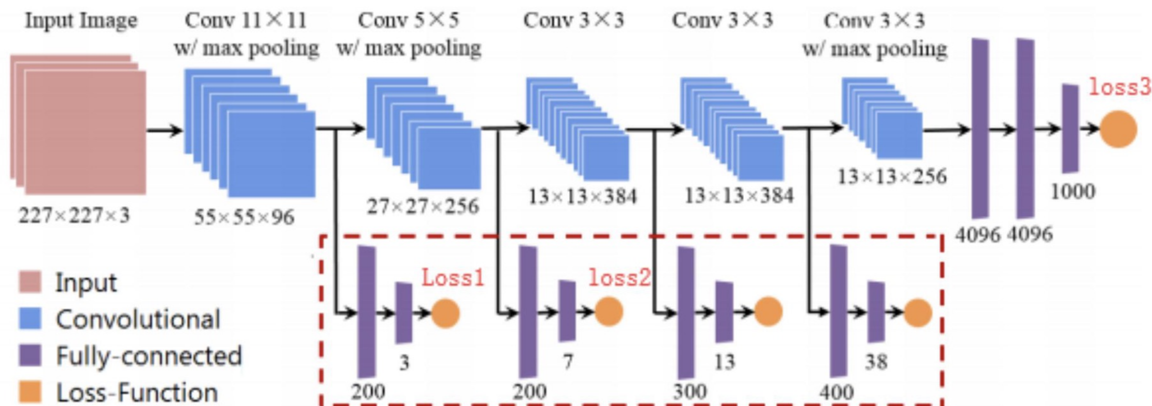
Dataset and DataLoader

- Iterable over a dataset
- Customizing data loading order
- automatic batching, etc.

```
from torch.utils.data import Dataset, DataLoader

class RandomDataset(Dataset):

    def __init__(self, size, length):
        self.len = length
        self.data = torch.randn(length, size)

    def __getitem__(self, index):
        return self.data[index]

    def __len__(self):
        return self.len

rand_loader = DataLoader(dataset=RandomDataset(5, 100),
                         batch_size=30, shuffle=True, num_workers=1)
```

Training with multiple Losses

```
#one
loss1.backward()
loss2.backward()
loss3.backward()
optimizer.step()
#two
loss1.backward()
optimizer.step()
loss2.backward()
optimizer.step()
loss3.backward()
optimizer.step()
#three
loss = loss1+loss2+loss3
loss.backward()
optimizer.step()
```



https://stackoverflow.com/questions/53994625/how-can-i-process-multi-loss-in-pytorch

Dynamic computation graphs

- Static computation graphs (TF):
  - Phase 1: Build the architecture/graph structure
  - Phase 2: Run data through it
- Dynamic computation graphs (Pytorch):
  - Dynamic graphs are more flexible
  - Build graph structure and perform computation
  - at the same time. Debug Friendly.
- Pytorch is slower, not used on edge devices