Introduction to PyTorch

CS236 Session, Fall 2019

Rui Shu

Slides almost entirely by Jiaming Song

Why PyTorch? (Instead of TF/Keras)

- Flatter learning curve*
- Easier to debug*
- Easier to customize (than Keras)
- Out-of-the-box multi-GPU / distributed support

*When compared to TF1. TF 2.0 hides the complexities, but you should still be mindful when compiling with tf.function.

What is PyTorch?

- **Numpy on GPU**
- Automatic Differentiation
- Deep Learning Framework

Tensors

Almost all PyTorch operations are performed on Tensors

- Batch of images: [N, C, H, W]
  - N images, C channels, H height, W width
- Sequence of words: [N, L, D]
  - N sequences, L length, D words

Similar to Numpy arrays, but allow GPU operations.

PyTorch vs. Numpy

```
numpy.ones([5, 3])                torch.ones(5, 3)

numpy.ones_like(ndarray)          torch.ones_like(tensor)

numpy.random.randn(5, 3)          torch.randn(5, 3)

numpy.empty([5, 3])               torch.empty(5, 3)

numpy.array([5., 3.])             torch.tensor([5., 3.])
```

PyTorch Tensor Operations

```
torch.ones(5, 3)

tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], dtype=torch.float64)
```

PyTorch Tensor Operations

```
torch.randn(5, 3)
```

```
tensor([[ 0.2349, -0.0427, -0.5053],
        [ 0.6455,  0.1199,  0.4239],
        [ 0.1279,  0.1105,  1.4637],
        [ 0.4259, -0.0763, -0.9671],
        [ 0.6856,  0.5047,  0.4250]])
```

PyTorch Tensor Operations

```
torch.ones_like(tensor)

Input: tensor([[ 0.2349, -0.0427, -0.5053],
               [ 0.6455,  0.1199,  0.4239]])

Output: tensor([[1., 1., 1.],
                [1., 1., 1.], dtype=torch.float64)
```

PyTorch Tensor Operations

```
torch.empty(5, 3)
```

```
tensor([[ 0.0000e+00,  2.5244e-29,  0.0000e+00],
        [ 2.5244e-29,  1.4569e-19,  2.7517e+12],
        [ 7.5338e+28,  3.0313e+32,  6.3828e+28],
        [ 1.4603e-19,  1.0899e+27,  6.8943e+34],
        [ 1.1835e+22,  7.0976e+22,  1.8515e+28]])
```

(The values are not initialized)

PyTorch Tensor Operations

```
torch.tensor([5., 3.])
tensor([ 5.,  3.,])  # defaults to torch.float32

torch.from_numpy(np.array([5., 3.]))
tensor([ 5.,  3.,], dtype=torch.float64) # because numpy defaults to 64bit

torch.tensor([5., 3.]).numpy()
array([5., 3.], dtype=float32)
```

PyTorch Math Operations

```
torch.tensor([5., 3.]) + torch.tensor([3., 5.])
tensor([ 8.,  8.,])
```

```
 -  z = torch.add(x, y)
 -  torch.add(x, y, out=z)
 -  y = y.add_(x)    # y += x
```

(Find out other operations in documentation!)

Indexing and Reshaping

```
torch.tensor([[5., 3.]])[0, :]
tensor([ 5.,  3.,])

torch.tensor([[5., 3.]]).view(-1)  # infer dimension size
torch.tensor([[5., 3.]]).view(2)
tensor([ 5.,  3.,])


torch.tensor([[5., 3.]]).size()
torch.Size([1, 2])
```

CUDA Tensor

Operation that “transfers” array in CPU to array in GPU (or vice versa).

```
if torch.cuda.is_available():
    device = torch.device("cuda")          # a CUDA device object
    x = torch.ones(2, device=device)       # directly create a tensor on GPU
    y = x.to(device)                       # or just use strings ``.to("cuda")``
    z = x + y
    print(z)                               # z is on GPU
    print(z.to("cpu", torch.double))       # to('cpu') moves array to CPU
```

What is PyTorch?

- Numpy on GPU
- **Automatic Differentiation**
- Deep Learning Framework

Automatic Differentiation

- autograd package in PyTorch
- Tensor has a `.requires_grad` attribute
- Set True: PyTorch track its operation, allowing for backprop
- Set False: PyTorch does not track its operation, allowing faster inference

Automatic Differentiation

```
x = torch.ones(2, 2, requires_grad=True)
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)

y = x + 2
print(y)
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)
```

Automatic Differentiation

```
z = y * y * 3
out = z.mean()

tensor(27., grad_fn=<MeanBackward1>)

out.backward()
print(x.grad)

tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

Automatic Differentiation

```
print(x.requires_grad)
print((x ** 2).requires_grad)

with torch.no_grad():
    print((x ** 2).requires_grad)
```

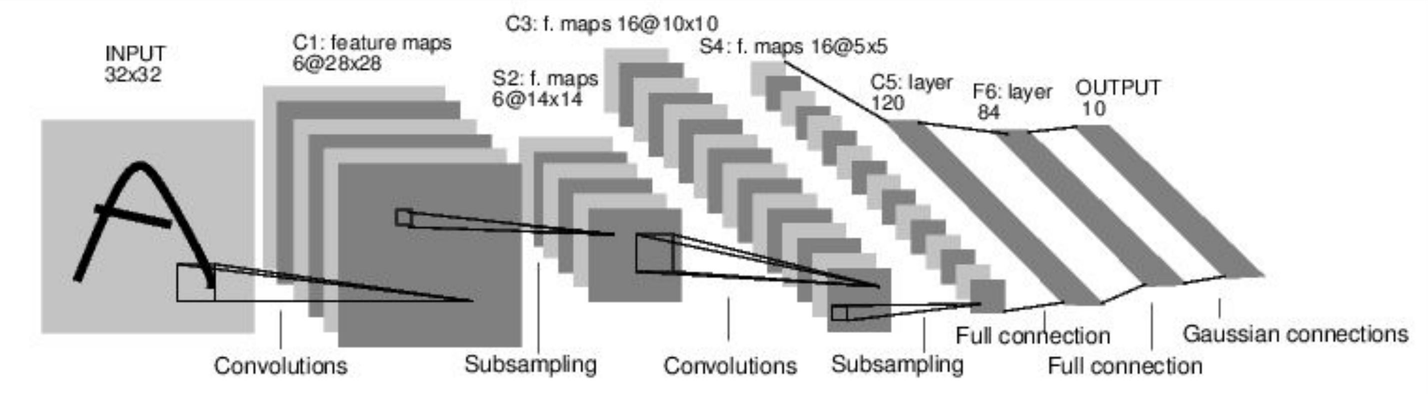
```
True
True
False
```

What is PyTorch?

- Numpy on GPU
- Automatic Differentiation
- **Deep Learning Framework**

Neural Networks

torch.nn package



Conv1 -> Pool -> Conv2 -> Pool -> FC1 -> FC2 -> FC3 -> Softmax

Training Neural Networks

1. Define the neural network that has some learnable parameters
2. Iterate over a dataset of inputs
3. Process input through the network
4. Compute the loss (how far is the output from being correct)
5. Propagate gradients back into the network’s parameters
6. Update the weights of the network

Define the Neural Network

```
import torch.nn as nn

class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

(Net has 2 Conv Layers, 3 FC Layers)

`nn.Module:` encapsulates parameters into a neural network module

- Loading
- Moving to GPU
- Exporting
- **`.forward()`** operation

Define Forward Operation

```python
import torch.nn.functional as F

def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Define Backward Operation (?)

No need -- PyTorch does automatic differentiation

```
params = list(net.parameters())
print(len(params))             # 10   (5 layers, each has weight and bias)
print(params[0].size())        # torch.Size([6, 1, 5, 5])

input = torch.randn(1, 1, 32, 32)
out = net(input)
print(out)
```

```
tensor([[ 0.1246, -0.0511,  0.0235,  0.1766, -0.0359, -0.0334,  0.1161,  0.0534,
          0.0282, -0.0202]], grad_fn=<ThAddmmBackward>)
```

Loss Function

```
output = net(input)
target = torch.randn(10)  # a dummy target, for example
target = target.view(1, -1)  # make it the same shape as output
criterion = nn.MSELoss()

loss = criterion(output, target)
print(loss)
```

```
tensor(1.3638, grad_fn=<MseLossBackward>)
```

Computational Graph

```
input -> conv2d -> relu -> maxpool2d -> conv2d -> relu -> maxpool2d
      -> view -> linear -> relu -> linear -> relu -> linear
      -> MSELoss
      -> loss
```

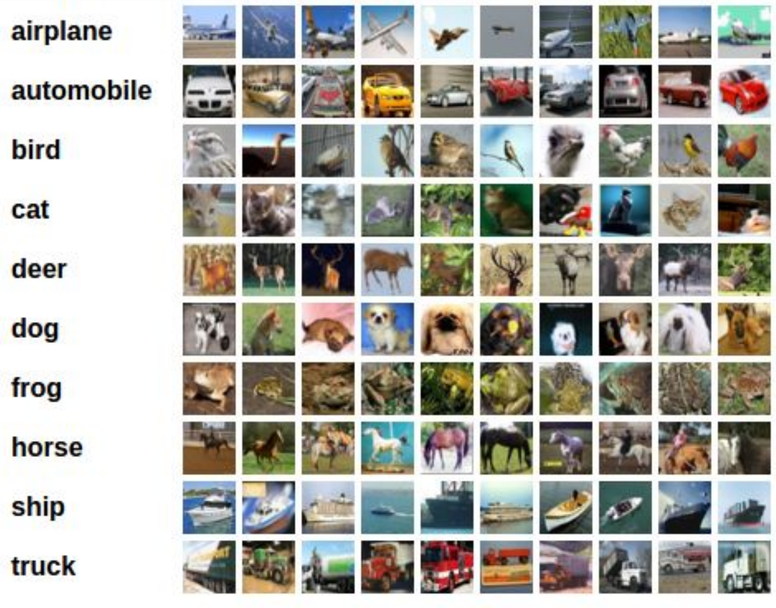
Optimization

```python
import torch.optim as optim

# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)   # compute the loss
loss.backward()
optimizer.step()   # SGD update
```

Datasets

CIFAR-10 Dataset

- 10 classes
- 32x32 images



Datasets Loader and Transform

```
import torchvision.transforms as transforms

transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
```

- Transform operator that normalizes the dataset.
- One could use transform operations for data augmentations!

Datasets Loader and Transform

```
import torchvision
trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)
```

- Create a training set and a DataLoader that iterates over it
- Similar to Python lists and list iterators!

Dataset Iterator

DataLoader can be used like a Python iterator!

```
images, labels = next(iter(trainloader))

for image, labels in trainloader:
      optimizer.zero_grad()
      output = net(image)
      loss = criterion(output, labels)
      loss.backward()
      optimizer.step()
```

Training on GPU

The network needs to be on GPU for it to be trained on GPU!

Fortunately, nn.Module encapsulates this for us with the `.to()` method

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
net.to(device)
```

Recap

1. Define the neural network that has some learnable parameters (nn.Module)
2. Iterate over a dataset of inputs (torchvision, torchvision.transform)
3. Process input through the network (torch.utils.data)
4. Compute the loss (torch.nn)
5. Propagate gradients back into the network’s parameters (.backward)
6. Update the weights of the network (torch.optim)

Recurrent Layers

https://pytorch.org/docs/stable/nn.html#rnn

torch.nn.LSTM

```
rnn = nn.LSTM(10, 17, 2)          # input dim: 10, hidden dim: 17, num stacks: 2
input = torch.randn(5, 3, 10)     # length: 5, batch: 3, dim: 10
h0 = torch.randn(2, 3, 17)        # stacks: 2, batch: 3, dim: 17
c0 = torch.randn(2, 3, 17)        # stacks: 2, batch: 3, dim: 17
output, (hn, cn) = rnn(input, (h0, c0))  # shapes: (5, 3, 17), ((2, 3, 17), (2, 3, 17))
```

Training on Multiple GPUs

Two types of Parallelism

- Data (which splits batch to multi-GPUs)
- Model (which splits model operation to multi-GPUs)

PyTorch data parallelism in one line: `net = nn.DataParallel(net)`

*Use with caution: Changes state_dict keys. Some edge cases only [recently fixed]

Additional Notes on `torch.nn.Module`

https://pytorch.org/docs/stable/_modules/torch/nn/modules/module.html

- Resetting weights
- Registering parameters and module lists
- Hooks

Colab

Additional Resources

- Neural Network Layers: https://pytorch.org/docs/stable/nn.html#
- Distributions: https://pytorch.org/docs/stable/distributions.html
- Checkpoint: https://pytorch.org/docs/stable/checkpoint.html*
- Initialization: https://pytorch.org/docs/stable/nn.html#torch-nn-init
- Distributed Training: https://pytorch.org/docs/stable/distributed.html

Documentation is your friend :)

*A little misleading: here, checkpoint doesn't mean saving your model parameters. It means not caching the intermediate values to save memory when training a very big model.

Try Out Other Libraries!

- Jax/Stax
- TF/Sonnet
- TF/Keras
- Build your own :D