IIT Kharagpur | IIT Madras | IIT Goa | IIT Palakkad

Applied Accelerated AI

Introduction to PyTorch

Dr. Satyajit Das
Assistant Professor
Data Science
Computer Science and Engineering
IIT Palakkad

National Supercomputing Mission

Centre for Development of Advanced Computing

What is PyTorch

- Open source machine learning library
- Developed by Facebook's AI Research lab
- It leverages the power of GPUs
- Automatic computation of gradients
- Makes it easier to test and develop new ideas.

Other libraries

PyTorch Tensor

- Similar to NumPy arrays
- They can also be used on a GPU
    - Faster computation
- Random matrix

```
import torch

x=torch.rand(2,3)
y=torch.rand(3,3)

print x
print y
```

PyTorch Tensor

- Similar to NumPy arrays
- They can also be used on a GPU
  - Faster computation
- All zeros
- Directly from data
- Size of a tensor

```
import torch

x = torch.zeros(5, 3)

x = torch.tensor([5.5, 3])

print x.size()
```

Operations

- Adding tensors
- Indexing

```
x = torch.randn(4, 4)

y = torch.randn(4, 4)

print(torch.add(x, y))

print(x[:, 1])
```

Operations

- Resizing
  - If you want to resize/reshape tensor

- `x = torch.randn(4, 4)`

- `y = x.view(16)`
- `z = x.view(-1, 8)`

- `print(x.size(), y.size(), z.size())`

- ***Output:***
- `torch.Size([4, 4])`
- `torch.Size([16])`
- `torch.Size([2, 8])`

```
x = torch.randn(1, 4, 32, 24)

y = x.view(8, 2, -1, 3, 8)

print(y.size())
```

Output shape?

Torch tensor vs NumPy array

- NumPy array
  - CPU
- Torch tensor
  - GPU

```
a = torch.ones(5)
tensor([1., 1., 1., 1., 1.])

b = a.numpy()

a = numpy.ones(5)
b = torch.from_numpy(a)
```

Matrix Multiplication in PyTorch

```
import torch

mat1=torch.randn(2,3)
mat2=torch.randn(3,3)
res=torch.mm(mat1,mat2)

print res.size()
```

Output: (2L, 3L)

Batch Matrix Multiplication in PyTorch

```
import torch

batch1=torch.randn(10,3,4)
batch2=torch.randn(10,4,5)
res=torch.bmm(batch1,batch2)

print res.size()

Output:  (10L, 3L, 5L)
```

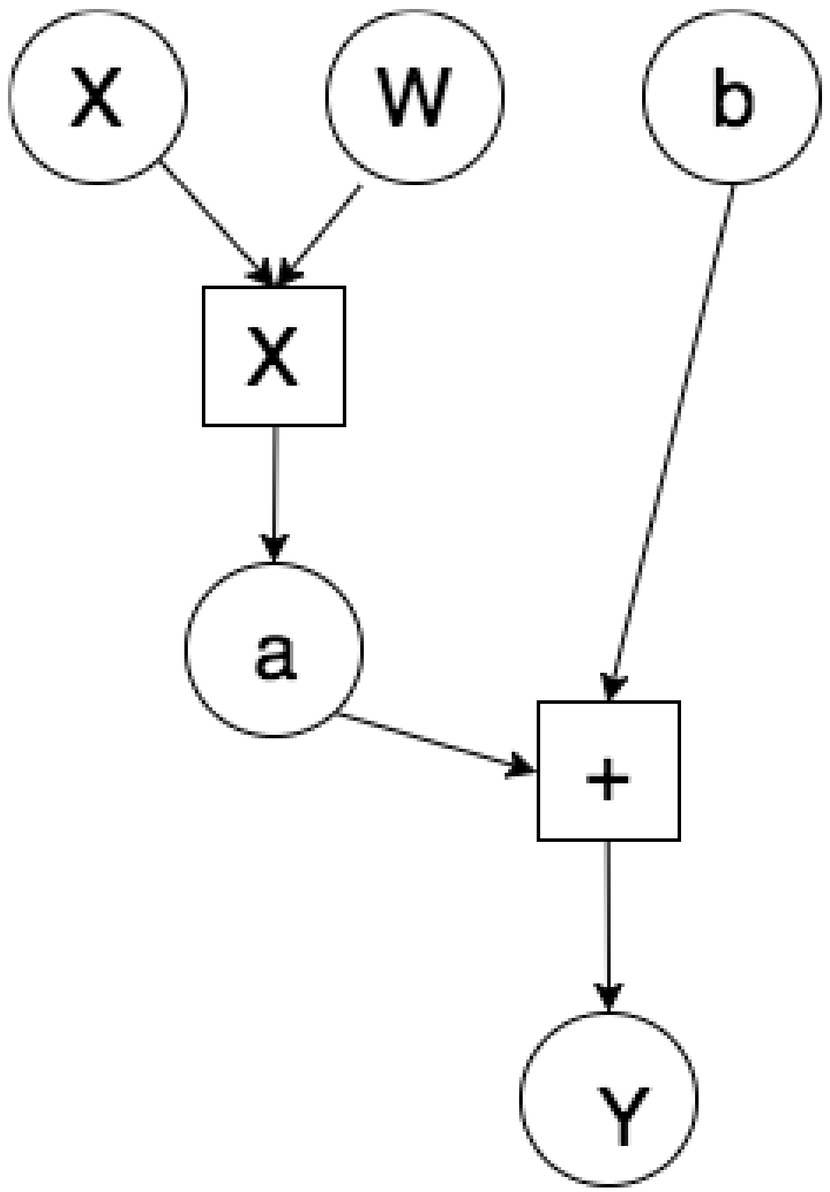
Many Tensor operations in PyTorch…

- torch.mm
- Matrix multiplication
- torch.bmm
- Batch matrix multiplication
- torch.cat
- Tensor Concatenation
- torch.sqeueeze/torch.unsqueeze
- Change Tensor dimensions
- …
- Check documentation at http://pytorch.org/docs/master/torch.html#tensors

Computational Graphs

```
import torch

x = torch.ones(2,2)
y = torch.ones(2,1)
w = torch.randn(2,1,requires_grad=True)
b = torch.randn(1,requires_grad=True)
```



Computational Graphs

```
p = torch.sigmoid(torch.mm(x, w) + b)   # prediction

loss = -y*torch.log(p)-(1-y)*torch.log(1-p)   # cross-entropy loss

cost = loss.mean()
# the cost to minimize
```

Automatic Gradient Computation

```
p = torch.sigmoid(torch.mm(x, w) + b)

loss = -y*torch.log(p)-(1-y)*torch.log(1-p)
cost = loss.mean()

cost.backward()

print w.grad
print b.grad
```

Training procedure

- Define the neural network
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss
- Propagate gradients back into the network's parameters
- Update the weights of the network

Training procedure

- Define the neural network
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss
- Propagate gradients back into the network’s parameters
- Update the weights of the network

Build Neural Networks using PyTorch

Neural networks can be constructed using the torch.nn package.

- Forward
  - An nn.Module contains layers, and
  - A method forward(input) that returns the output
  - You can use any of the Tensor operations in the forward function
- Backward
  - nn depends on autograd
  - You just have to define the forward function

Define a Network Class

```
import torch
import torch.nn as nn
```

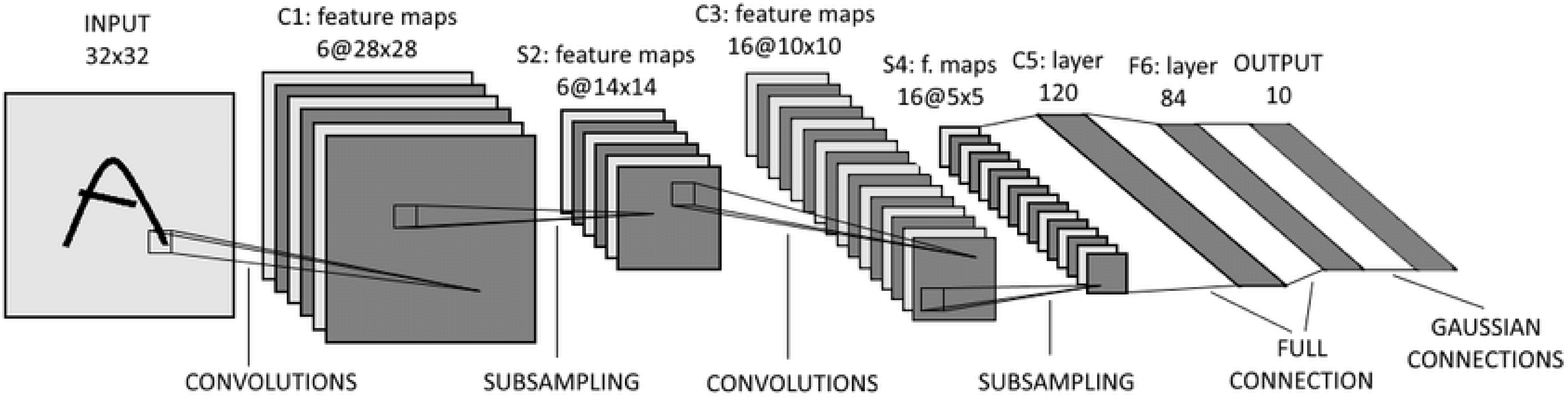
```
class Net(nn.Module):
```

```
    def_init__(self):
        super(Net, self).__init__()
        # create layers
```

```
    def forward(self, x):
        # define feed-forward function
```

You don’t need to define a backward function!

CNN for MNIST: A Full Example



https://medium.com/deep-learning-g/build-lenet-from-scratch-7bd0c67a151e

Define a CNN Network

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolutio
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        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

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
```

Define a CNN Network

```
def   init  (self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels,
 5x5 square convolution kernel

        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)

        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

Define a CNN Network

```
def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2)) # If the size is
        #square you can only specify a single number
        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 a CNN Network

```
def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s

        return num_features
```

Training procedure

- Define the neural network
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss
- Propagate gradients back into the network's parameters
- Update the weights of the network

Data

- For images
  - Pillow, OpenCV are useful
- For audio
  - Scipy and librosa
- For text
  - NLTK and SpaCy are useful

- Load data into memory as NumPy array
  - Then convert to tensor for GPU

Loading data - torchvision

- Torchvision
  - it’s extremely easy to load existing datasets

```
import torchvision
import torchvision.transforms as transforms
```

Loading data - torchvision

```
import torchvision
import torchvision.transforms as transforms

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

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)
```

Loading data - torchvision

```
import torchvision
import torchvision.transforms as transforms

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


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

Training procedure

- Define the neural network
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss
- Propagate gradients back into the network's parameters
- Update the weights of the network

Feed-forward

```
def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a
single number
        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
```

Training procedure

- Define the neural network
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss
- Propagate gradients back into the network's parameters
- Update the weights of the network

Loss function

- A loss function takes the (output, target) pair of inputs
- Computes a value that estimates how far away the output is from the target.
- There are several different loss functions under the nn package.
- A simple loss can be
  - nn.MSELoss
  - It computes the mean-squared error between the input and the target.

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)
```

Training procedure

- Define the neural network
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss
- Propagate gradients back into the network's parameters
- Update the weights of the network

Gradient computation

```
output = net(input)

loss = criterion(output, target)

loss.backward()
```

Training procedure

- Define the neural network
- Iterate over a dataset of inputs
- Process input through the network
- Compute the loss
- Propagate gradients back into the network’s parameters
- Update the weights of the network

Update parameters

```
import torch.optim as optim

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

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

- Total training samples = 80000 Batch-size = 50
- Each iteration takes 30 seconds
- How many hours for 3 epochs of training?

Full training

```
net = Net()

trainloader = torch.utils.data.DataLoader
                 (trainset, batch_size=4,
                  shuffle=True, num_workers=2)

criterion = nn.CrossEntropyLoss()

optimizer = optim.SGD(net.parameters(),
                lr=0.001, momentum=0.9)
```

Full training

```
for epoch in range(2):
# loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # training code for each batch

print('Finished Training')
```

Full training

```
for epoch in range(2):
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs;
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()
        ...
```

Full training

```
for epoch in range(2):
    for i, data in enumerate(trainloader, 0):
        ...
        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        ...
```

Full training

```
for epoch in range(2):
    for i, data in enumerate(trainloader, 0):
        ...
        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999: # every 2000 batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch+1,i+1,running_loss/2000))
            running_loss = 0.0
```

Full training

```
for epoch in range(2):
# loop over the dataset multiple times


        # training code for each batch
print('Finished Training')

PATH = './cifar_net.pth'
torch.save(net.state_dict(), PATH)
```

Testing

```
dataiter = iter(testloader)
images, labels = dataiter.next()

net = Net()
net.load_state_dict(torch.load(PATH))
outputs = net(images)

_, predicted = torch.max(outputs, 1)
```

Training on GPU

- Let's first define our device

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

net.to(device)

inputs, labels = data[0].to(device),
data[1].to(device)
```

References

- Sources for this lecture include materials from pytorch.org

Thank You