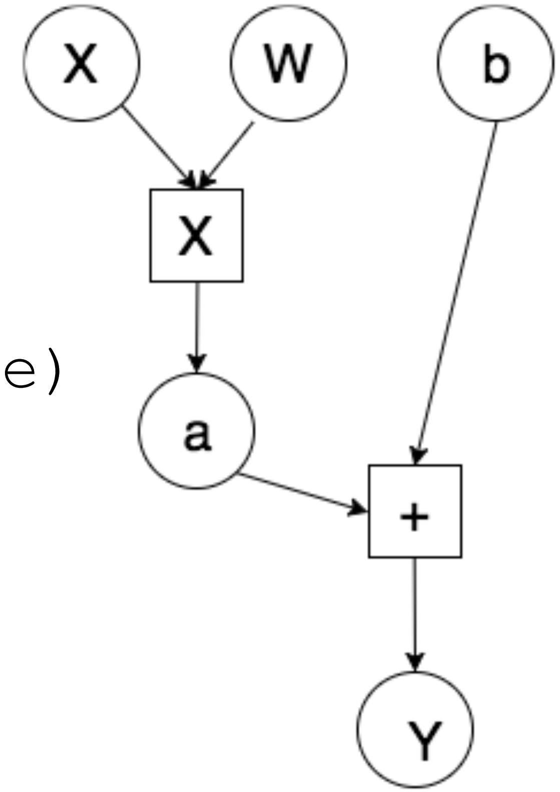
PyTorch Tutorial - II

Lecture 10

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)
```



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

Define a CNN Network

```python
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
```

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

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()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()    # Does the update
```

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
    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # 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)
```

Questions?

Sources for this lecture include materials from pytorch.org