COMP9444
Neural Networks and Deep Learning

2c. PyTorch

Typical Structure of a PyTorch Progam

```
# create neural network
net = MyNetwork().to(device)  # CPU or GPU

# prepare to load the training and test data
train_loader = torch.utils.data.DataLoader(...)
test_loader  = torch.utils.data.DataLoader(...)

# choose between SGD, Adam or other optimizer
optimizer = torch.optim.SGD(net.parameters,...)

for epoch in range(1, epochs): # training loop
   train(args, net, device, train_loader, optimizer)
   # periodically evaluate network on test data
   if epoch % 10 == 0:
       test( args, net, device, test_loader)
```

Defining a Network Structure

```
class MyNetwork(torch.nn.Module):

   def __init__(self):
      super(MyNetwork, self).__init__()
      # define structure of the network here

   def forward(self, input):
      # apply network and return output
```


Defining a Custom Model

This module computes a function of the form $(x,y) \mapsto Ax\log(y) + By^2$

```
import torch.nn as nn

class MyModel(nn.Module):

  def __init__(self):
      super(MyModel, self).__init__()
      self.A = nn.Parameter(torch.randn((1),requires_grad=True))
      self.B = nn.Parameter(torch.randn((1),requires_grad=True))

  def forward(self, input):
      output = self.A * input[:,0] * torch.log(input[:,1]) \
             + self.B * input[:,1] * input[:,1]
      return output
```

Building a Net from Individual Components

```
class MyModel(torch.nn.Module):

    def __init__(self):
        super(MyModel, self).__init__()
        self.in_to_hid = torch.nn.Linear(2,2)
        self.hid_to_out = torch.nn.Linear(2,1)

    def forward(self, input):
        hid_sum = self.in_to_hid(input)
        hidden = torch.tanh(hid_sum)
        out_sum = self.hid_to_out(hidden)
        output = torch.sigmoid(out_sum)
        return output
```

Defining a Sequential Network

```
class MyModel(torch.nn.Module):

    def __init__(self, num_input, num_hid, num_out):
        super(MyModel, self).__init__()
        self.main = nn.Sequential(
            nn.Linear(num_input, num_hid),
            nn.Tanh(),
            nn.Linear(num_hid, num_out),
            nn.Sigmoid()
        )
    def forward(self, input):
        output = self.main(input)
        return output
```

Sequential Components

Network layers:

- `nn.Linear()`
- `nn.Conv2d()` (Week 4)

Intermediate Operators:

- `nn.Dropout()`
- `nn.BatchNorm()` (Week 4)

Activation Functions:

- `nn.Sigmoid()`
- `nn.Tanh()`
- `nn.ReLU()` (Week 3)

Declaring Data Explicitly

```
import torch.utils.data

# input and target values for the XOR task
input = torch.Tensor([[0,0],[0,1],[1,0],[1,1]])
target = torch.Tensor([[0],[1],[1],[0]])

xdata = torch.utils.data.TensorDataset(input,target)
train_loader = torch.utils.data.DataLoader(xdata,batch_size=4)
```

Loading Data from a `.csv` File

```
import pandas as pd

df = pd.read_csv("sonar.all-data.csv")
df = df.replace('R',0)
df = df.replace('M',1)
data = torch.tensor(df.values,dtype=torch.float32)
num_input = data.shape[1] - 1
input = data[:,0:num_input]
target = data[:,num_input:num_input+1]
dataset = torch.utils.data.TensorDataset(input,target)
```

Custom Datasets

```
from data import ImageFolder
    # load images from a specified directory
    dataset = ImageFolder(folder, transform)

import torchvision.datasets as dsets
    # download popular image datasets remotely
    mnistset = dsets.MNIST(...)
    cifarset = dsets.CIFAR10(...)
    celebset = dsets.CelebA(...)
```


Choosing an Optimizer

```
# SGD stands for \Stochastic Gradient Descent"
optimizer = torch.optim.SGD( net.parameters(),
    lr=0.01, momentum=0.9,
    weight_decay=0.0001)

# Adam = Adaptive Moment Estimation (good for deep networks)
optimizer = torch.optim.Adam(net.parameters(),eps=0.000001,
    lr=0.01, betas=(0.5,0.999),
    weight_decay=0.0001)
```


Training

```
def train(args, net, device, train_loader, optimizer):

   for batch_idx, (data,target) in enumerate(train_loader):
       optimizer.zero_grad()      # zero the gradients
       output = net(data)         # apply network
       loss = ...                 # compute loss function
       loss.backward()            # compute gradients
       optimizer.step()           # update weights
```

Loss Functions

```
loss = torch.sum((output-target)*(output-target))
loss = F.nll_loss(output,target)                   # (Week 3)
loss = F.binary_cross_entropy(output,target) # (Week 3)
loss = F.softmax(output,dim=1)                    # (Week 3)
loss = F.log_softmax(output,dim=1)                # (Week 3)
```

Testing

```
def test(args, net, device, test_loader):

    with torch.no_grad():  # suppress updating of gradients
        net.eval()   # toggle batch norm, dropout
        for data, target in test_loader:
            output = model(data)
            test_loss = ...
            print(test_loss)
        net.train()  # toggle batch norm, dropout back again
```

Computational Graphs

PyTorch automatically builds a computational graph, enabling it to backpropagate derivatives.

Every parameter includes `.data` and `.grad` components, for example:

```
A.data
A.grad
```

`optimizer.zero_grad()` sets all `.grad` components to zero.

`loss.backward()` updates the `.grad` component of all Parameters by backpropagating gradients through the computational graph.

`optimizer.step()` updates the `.data` components.

Controlling the Computational Graph

If we need to stop the gradients from being backpropagated through a certain variable (or expression) A, we can exclude it from the computational graph by using:

```
A.detach()
```

By default, `loss.backward()` discards the computational graph after computing the gradients.

If needed, we can force it to keep the computational graph by calling it this way:

```
loss.backward(retain_graph=True)
```

