2b: PyTorch

PyTorch

The following code fragments illustrate the typical structure of a PyTorch program, with further details and various options for each component.

Typical Structure of a PyTorch Program

```
# create neural network according to model specification
net = MyModel().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,...)

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

Defining a model

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

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

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

Defining a Custom Model

This code defines a module for computing 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()

Intermediate Operators:

- nn.Dropout()
- nn.BatchNorm()

Activation Functions:

- nn.Tanh()
- nn.Sigmoid()
- nn.ReLU()

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() # update gradients
    optimizer.step() # update weights
```

Loss Functions

```
import torch.nn.functional as F
loss = torch.sum((output-target)*(output-target))
loss = F.nll_loss(output,target)
loss = F.binary_cross_entropy(output,target)
loss = F.softmax(output,dim=1)
loss = F.log_softmax(output,dim=1)
```

Testing

```
def test(args, model, device, test_loader):
with torch.no_grad(): # suppress updating of gradients
    net.eval() # toggle batch norm, dropout
    test_loss = 0
    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)
```

Optional Video on PyTorch

Exercise: Running PyTorch

The following program solves the simplest possible machine learning task:

solve $f(x) = Ax$ such that $f(1) = 1$

```
import torch
import torch.utils.data
import numpy as np

lr = 1.9 # learning rate
mom = 0.0 # momentum

class MyModel(torch.nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        self.A = torch.nn.Parameter(torch.zeros((1), requires_grad=True))
    def forward(self, input):
        output  = self.A * input
        return(output)

device = 'cpu'

input  = torch.Tensor([[1]])
target = torch.Tensor([[1]])

slope_dataset = torch.utils.data.TensorDataset(input,target)
train_loader  = torch.utils.data.DataLoader(slope_dataset,batch_size=1)

# create neural network according to model specification
net = MyModel().to(device) # CPU or GPU

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

epochs = 1000

for epoch in range(1, epochs):
    for batch_id, (data,target) in enumerate(train_loader):
        optimizer.zero_grad() # zero the gradients
        output = net(data)    # apply network
        loss = 0.5*torch.mean((output-target)*(output-target))
        if type(net.A.grad) == type(None):
            print('Ep%3d: zero_grad(): A.grad=  None  A.data=%7.4f loss=%7.4f' \
                    % (epoch, net.A.data, loss))
        else:
            print('Ep%3d: zero_grad(): A.grad=%7.4f A.data=%7.4f loss=%7.4f' \
                    % (epoch, net.A.grad, net.A.data, loss))
        loss.backward()       # compute gradients
        optimizer.step()      # update weights
```

```
        print('            step(): A.grad=%7.4f A.data=%7.4f' \
                     % (net.A.grad, net.A.data))
        if loss < 0.000000001 or np.isnan(loss.data):
            exit(0)
```

Question 1 *Submitted Sep 7th 2022 at 11:16:16 pm*

Run the code above and look at the output.

Change the learning rate `lr` to each of the following values by editing line 5 in the above code.

```
0.01, 0.1, 0.5, 1.0, 1.5, 1.9, 2.0, 2.1
```

Try running the code and describe what happens for each value of `lr`, in terms of the success and speed of the algorithm.

at `lr=0.01`, `A.data` converges to 1 at epoch 988

at `lr=0.1` , `A.data` converges to 1 at epoch 97

at `lr=0.5`, `A.data` converges to 1 at epoch 16

at `lr=1.0`, `A.data` converges to 1 at epoch 2

at `lr=1.5`, `A.data` converges to 1 at epoch 16. It looks like it overshoots substantially then oscillates around 1 for a few epochs before settling.

at `lr=1.9`, `A.data` converges to 1 at epoch 97. similar pattern

at `lr=2.0`, `A.data` does not converge to 1. It consistently over (2) and under (0) shoots

at `lr=2.1`, `A.data` does not converge to 1. It gets pretty wild. It explodes to infinity then changes to Nan

Question 2 *Submitted Sep 7th 2022 at 11:29:41 pm*

Now keep the learning rate at `1.9`, but try each of the following values for momentum by changing the value of `mom` on line 6.

```
0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9
```

For which value of momentum is the task solved in the fewest epochs?

What happens when the momentum is `1.0`? What happens when it is `1.1`?

at `mom=0.1`, it converges to 1 at epoch 25 at `mom=0.2`, it converges to 1 at epoch 14

at `mom=0.3`, it converges to 1 at epoch 13

at `mom=0.4`, it converges to 1 at epoch 24

at `mom=0.5`, it converges to 1 at epoch 30

at `mom=0.6`, it converges to 1 at epoch 37

at `mom=0.7`, it converges to 1 at epoch 59

at `mom=0.8`, it converges to 1 at epoch 92

at `mom=0.9`, it converges at epoch 190

at `mom=1.0` it does not converge

at `mom=1.1` it explodes

Exercise: XOR with PyTorch

This program trains a two-layer neural network on the famous XOR task.

```
import torch
import torch.utils.data
import torch.nn.functional as F

lr = 0.1
mom = 0.0
init = 1.0

class MyModel(torch.nn.Module):
    def __init__(self):
        super(MyModel, self).__init__()
        # define structure of the network here
        self.in_hid  = torch.nn.Linear(2,2)
        self.hid_out = torch.nn.Linear(2,1)
    def forward(self, input):
        # apply network and return output
        hid_sum = self.in_hid(input)
        hidden  = torch.tanh(hid_sum)
        out_sum = self.hid_out(hidden)
        output  = torch.sigmoid(out_sum)
        return(output)

device = 'cpu'

input  = torch.Tensor([[0,0],[0,1],[1,0],[1,1]])
target = torch.Tensor([[0],[1],[1],[0]])

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

# create neural network according to model specification
net = MyModel().to(device) # CPU or GPU

# initialize weight values
net.in_hid.weight.data.normal_(0,init)
net.hid_out.weight.data.normal_(0,init)

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

epochs = 10000

for epoch in range(1, epochs):
    #train(net, device, train_loader, optimizer)
    for batch_id, (data,target) in enumerate(train_loader):
        optimizer.zero_grad() # zero the gradients
```

```
        output = net(data)    # apply network
        loss = F.binary_cross_entropy(output,target)
        loss.backward()       # compute gradients
        optimizer.step()      # update weights
        if epoch % 100 == 0:
            print('ep%3d: loss = %7.4f' % (epoch, loss.item()))
        if loss < 0.01:
            print("Global Mininum")
            exit(0)
print("Local Minimum")
```

Question 1 *Submitted Sep 7th 2022 at 11:38:22 pm*

Run the code ten times. For how many runs does it reach the Global Minimum? For how many runs does it reach a Local Minimum?

Solution

It should reach the Global Minimum in approximately half of runs, and gets stuck in a Local Minimum for the other half.

Question 2 *Submitted Sep 7th 2022 at 11:39:26 pm*

Keeping the learning rate fixed at `0.1`, adjust the values of momentum (`mom`) on line 6 and initial weight size (`init`) on line 7 to see if you can find values for which the code converges relatively quickly to the Global Minimum on virtually every run.

Solution

With `mom = 0.9` and `init = 0.01` it should successfully reach the Global Minimum in 99% of runs.

Coding Exercise: Basic PyTorch Operations

Objective

The **Tensor** is a fundamental structure in PyTorch which is very similar to an array or matrix. Tensors are used to encode the inputs and outputs of a model, as well as the model's parameters. In this exercise, you will learn how to implement basic tensor operations.

Instructions

Before starting the exercise, please go through the tutorial about tensors from the PyTorch website.

https://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html#sphx-glr-beginner-blitz-tensor-tutorial-py

For some of the exercises, the `torch.Tensor` documentation should be very helpful.

https://pytorch.org/docs/stable/tensors.html