**# Main**

import numpy as np # to handle matrix and data operation

import pandas as pd # to read csv and handle dataframe

import os # operation system library

import torch # to load pytorch library

import torch.nn as nn # to load pytorch library

from network import LeNet5 # load network from network.py

import torch.utils.data # to load data processor

from torch.autograd import Variable # pytorch data type

from torchvision import datasets, transforms # to load torch data processor

from data import get\_train\_set, get\_test\_set

# import training and testing sets

import matplotlib.pyplot as plt

import argparse

**#============================ parse the command line =============================================**

parser = argparse.ArgumentParser()

parser.add\_argument("-Batch\_size", type=int, default=1000, help="Batch number")

parser.add\_argument("-Epoch", type=int, default=100, help="Number of epoches")

parser.add\_argument("-lr", type=float, default=0.01, help="Learning rate")

parser.add\_argument("-Optimizer", type=str, default="SGD", help="Optimization approach (SGD, Adam)")

parser.add\_argument("-Init", type=str, default="Kaiming", help="Initialization approach (Kaiming, Xavier, Random)")

parser.add\_argument("-pretrained", type=int, default=0, help="use pre-trained model or not")

parser.add\_argument("-pretrained\_model", type=str, help="pre-trained model name")

opt = parser.parse\_args()

**#============================ load training and testing data to torch data loader==================**

BATCH\_SIZE = opt.Batch\_size

train = get\_train\_set()

test = get\_test\_set()

train\_loader = torch.utils.data.DataLoader(dataset=train, batch\_size = BATCH\_SIZE, shuffle = True, num\_workers=4)

test\_loader = torch.utils.data.DataLoader(dataset=test, batch\_size = BATCH\_SIZE, shuffle = True, num\_workers=4)

**#================================= define useful functions =========================================**

**# define training process**

def train(epoch):

# set model to training mode(with gradient calculation)

model.train()

epoch\_loss = 0

# load training data

for batch\_idx, (data, target) in enumerate(train\_loader):

data, target = data.cuda(), target.cuda() # load data to GPU

#Variables in Pytorch are differenciable.

data, target = Variable(data), Variable(target)

#This will zero out the gradients for this batch.

optimizer.zero\_grad()

# input data to the network and ouput prediction

output = model(data)

# Calculate the loss The negative log likelihood loss. It is useful to train a classification problem with C classes.

loss = criterion(output, target)

#dloss/dx for every Variable

loss.backward()

#to do a one-step update on our parameter.

optimizer.step()

epoch\_loss += loss.item()

#Print out the loss periodically.

avg\_loss = epoch\_loss / len(train\_loader)

print("===> Epoch {} Complete: Training loss: {:.4f}".format(epoch, avg\_loss))

return avg\_loss

def train\_accuracy(epoch):

# set model to evaluation mode (without gradient calculation)

model.eval()

correct = 0

# load training data

for batch\_idx, (data, target) in enumerate(train\_loader):

data, target = data.cuda(), target.cuda()

data, target = Variable(data), Variable(target)

with torch.no\_grad():

output = model(data)

# get the index of the max log-probability

pred = output.argmax(dim=1, keepdim=True)

# count the number of correct prediction

correct += pred.eq(target.view\_as(pred)).sum().item()

avg\_accuracy = correct / len(train\_loader.dataset)

print("===> Epoch {} Complete: Training accuracy: {:.4f}%".format(epoch, 100. \* avg\_accuracy))

return avg\_accuracy

def test(epoch):

model.eval()

correct = 0

# load testing data

for batch\_idx, (data, target) in enumerate(test\_loader):

data, target = data.cuda(), target.cuda()

data, target = Variable(data), Variable(target)

with torch.no\_grad():

output = model(data)

# get the index of the max log-probability

pred = output.argmax(dim=1, keepdim=True)

correct += pred.eq(target.view\_as(pred)).sum().item()

avg\_accuracy = correct / len(test\_loader.dataset)

print("===> Epoch {} Complete: Testing accuracy: {:.4f}%".format(epoch, 100. \* avg\_accuracy))

return avg\_accuracy

def checkpoint(epoch):

# define the name of learned model

model\_out\_path = "model/Training\_epoch\_{}.pth".format(epoch)

# save learned model

torch.save(model.state\_dict(), model\_out\_path)

print("Checkpoint saved to {}".format(model\_out\_path))

**#======================define the complete process, including training and testing processes==============**

model = LeNet5(opt.Init) # define the network

model.cuda() # load the network to GPU

criterion = nn.CrossEntropyLoss() # define loss function

criterion = criterion.cuda() #load the loss function to GPU

**# Whether use pre-trained model to continue training**

if opt.pretrained == 1:

model\_name = 'model/' + opt.pretrained\_model

if os.path.exists(model\_name):

# load pre-trained parameter to the model

model.load\_state\_dict(torch.load(model\_name, map\_location=lambda storage, loc: storage))

print('Pre-trained model is loaded.')

**# define the network optimizer**

if opt.Optimizer == 'SGD':

optimizer = torch.optim.SGD(model.parameters(), lr=opt.lr, momentum=0.5)

elif opt.Optimizer =='Adam':

optimizer = torch.optim.Adam(model.parameters(), lr=opt.lr, betas=(0.9, 0.999), eps=1e-8)

**# define the loss and accuracy matrix to store the intermediate results**

f1 = open('train\_loss\_record.txt','a+')

f2 = open('train\_accuracy\_record.txt','a+')

f3 = open('test\_accuracy\_record.txt','a+')

for epoch in range(opt.Epoch):

f1.write('%f\t' % train(epoch)) # use training data to train the network

f2.write('%f\t' % train\_accuracy(epoch)) # use training data to test the network

f3.write('%f\t' % test(epoch)) # use test data to test the network

# Every 10 epoch it save a learned model

if (epoch+1) % 10 == 0:

checkpoint(epoch+1)