import os

import torch

from torch import nn

from torchvision.transforms import ToTensor

from torch.utils.data import DataLoader

from torch.utils.data import Dataset

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import MinMaxScaler

from sklearn.preprocessing import LabelEncoder

#Pands导入数据

df = pd.read\_csv('/root/dataset/kddcup.data\_10\_percent\_corrected')

df.columns = [

    'duration',

    'protocol\_type',

    'service',

    'flag',

    'src\_bytes',

    'dst\_bytes',

    'land',

    'wrong\_fragment',

    'urgent',

    'hot',

    'num\_failed\_logins',

    'logged\_in',

    'num\_compromised',

    'root\_shell',

    'su\_attempted',

    'num\_root',

    'num\_file\_creations',

    'num\_shells',

    'num\_access\_files',

    'num\_outbound\_cmds',

    'is\_host\_login',

    'is\_guest\_login',

    'count',

    'srv\_count',

    'serror\_rate',

    'srv\_serror\_rate',

    'rerror\_rate',

    'srv\_rerror\_rate',

    'same\_srv\_rate',

    'diff\_srv\_rate',

    'srv\_diff\_host\_rate',

    'dst\_host\_count',

    'dst\_host\_srv\_count',

    'dst\_host\_same\_srv\_rate',

    'dst\_host\_diff\_srv\_rate',

    'dst\_host\_same\_src\_port\_rate',

    'dst\_host\_srv\_diff\_host\_rate',

    'dst\_host\_serror\_rate',

    'dst\_host\_srv\_serror\_rate',

    'dst\_host\_rerror\_rate',

    'dst\_host\_srv\_rerror\_rate',

    'label'

]

Df

#one-hot编码

# 数值列

number\_col = df.select\_dtypes(include=['number']).columns

# 分类变量

cat\_col = df.columns.difference(number\_col)

cat\_col = cat\_col.drop('label')

# 将分类变量筛选出来

df\_cat = df[cat\_col].copy()

# one-hot编码

one\_hot\_data = pd.get\_dummies(df\_cat, columns=cat\_col)

# 将原数据的分类变量去掉

one\_hot\_df = pd.concat([df, one\_hot\_data],axis=1)

one\_hot\_df.drop(columns=cat\_col, inplace=True)

one\_hot\_df

#归一化

minmax\_scale = MinMaxScaler(feature\_range=(0, 1))

def normalization(df,col):

    for i in col:

        arr = df[i]

        arr = np.array(arr)

        df[i] = minmax\_scale.fit\_transform(arr.reshape(len(arr),1))

    return df

normalized\_df = normalization(one\_hot\_df.copy(), number\_col)

normalized\_df

#标签编码

# 为不同的类别进行编码

labels = pd.DataFrame(df.label)

label\_encoder = LabelEncoder()

enc\_label = labels.apply(label\_encoder.fit\_transform)

normalized\_df.label = enc\_label

label\_encoder.classes\_

data = normalized\_df

#定义训练参数

X = data.drop(columns=['label'])

y = data['label']

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.20, random\_state=50)

class LoadData(Dataset):

    def \_\_init\_\_(self, X, y):

        self.X = X

        self.y = y

    def \_\_len\_\_(self):

        return len(self.X)

    def \_\_getitem\_\_(self, index):

        X = torch.tensor(self.X.iloc[index])

        y = torch.tensor(self.y.iloc[index])

        return X, y

train\_data = LoadData(X\_train, y\_train)

test\_data = LoadData(X\_test, y\_test)

X\_dimension = len(X\_train.columns)

y\_dimension = len(y\_train.value\_counts())

print(f"X的维度：{X\_dimension}")

print(f"y的维度：{y\_dimension}")

batch\_size = 128

train\_dataloader = DataLoader(train\_data, batch\_size=batch\_size)

test\_dataloader = DataLoader(test\_data, batch\_size=batch\_size)

device = 'cuda:0' if torch.cuda.is\_available() else 'cpu'

def train(model, optimizer, loss\_fn, epochs):

    losses = []

    iter = 0

    for epoch in range(epochs):

        print(f"epoch {epoch+1}\n-----------------")

        for i, (X, y) in enumerate(train\_dataloader):

            X, y = X.to(device).to(torch.float32), y.to(device).to(torch.float32)

            X = X.reshape(X.shape[0], 1, X\_dimension)

            y\_pred = model(X)

            loss = loss\_fn(y\_pred, y.long())

            optimizer.zero\_grad()

            loss.backward()

            optimizer.step()

            if i % 100 == 0:

                print(f"loss: {loss.item()}\t[{(i+1)\*len(X)}/{len(train\_data)}]")

                iter += 1

                losses.append(loss.item())

    return losses, iter

def test(model):

    positive = 0

    negative = 0

    with torch.no\_grad():

        iter = 0

        loss\_sum = 0

        for X, y in test\_dataloader:

            X, y = X.to(device).to(torch.float32), y.to(device).to(torch.float32)

            X = X.reshape(X.shape[0], 1, X\_dimension)

            y\_pred = model(X)

            loss = loss\_fn(y\_pred, y.long())

            loss\_sum += loss.item()

            iter += 1

            for item in zip(y\_pred, y):

                if torch.argmax(item[0]) == item[1]:

                    positive += 1

                else:

                    negative += 1

    acc = positive / (positive + negative)

    avg\_loss = loss\_sum / iter

    print("Accuracy:", acc)

    print("Average Loss:", avg\_loss)

def loss\_value\_plot(losses, iter):

    plt.figure()

    plt.plot([i for i in range(1, iter+1)], losses)

    plt.xlabel('Iterations (×100)')

    plt.ylabel('Loss Value')

#CNN模型

class CNN(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.backbone = nn.Sequential(

            nn.Conv1d(1, 3, kernel\_size=2),

            nn.MaxPool1d(2, 2),

            nn.Conv1d(3, 8, kernel\_size=2),

            nn.MaxPool1d(2, 2),

            nn.Conv1d(8, 16, kernel\_size=2)

        )

        self.flatten = nn.Flatten()

        self.fc = nn.Sequential(

            nn.Linear(432, 64),

            nn.ReLU(),

            nn.Linear(64, 64),

            nn.ReLU(),

            nn.Linear(64, y\_dimension)

        )

    def forward(self, X):

        X = self.backbone(X)

        X = self.flatten(X)

        logits = self.fc(X)

        return logits

CNN\_model = CNN()

CNN\_model.to(device=device)

epochs = 2

lr = 1e-3

momentum = 0.9

optimizer = torch.optim.SGD(CNN\_model.parameters(), lr=lr, momentum=momentum)

loss\_fn = nn.CrossEntropyLoss()

if os.path.exists('CNN\_model.pth'):

    CNN\_model.load\_state\_dict(torch.load('CNN\_model.pth'))

else:

    losses, iter = train(CNN\_model, optimizer, loss\_fn, epochs)

    torch.save(CNN\_model.state\_dict(), 'CNN\_model.pth')

    loss\_value\_plot(losses, iter)

    plt.savefig('CNN\_loss.png')

test(CNN\_model)

#DNN模型

class DNN(nn.Module):

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.flatten = nn.Flatten()

        self.network = nn.Sequential(

            nn.Linear(X\_dimension, 64),

            nn.ReLU(),

            nn.Linear(64, 64),

            nn.ReLU(),

            nn.Linear(64, y\_dimension)

        )

    def forward(self, X):

        X = self.flatten(X)

        logits = self.network(X)

        return logits

DNN\_model = DNN()

DNN\_model.to(device=device)

# 超参数

epochs = 5

lr = 1e-3

momentum = 0.9

optimizer = torch.optim.SGD(DNN\_model.parameters(), lr=lr, momentum=momentum)

loss\_fn = nn.CrossEntropyLoss()

if os.path.exists('DNN\_model.pth'):

    DNN\_model.load\_state\_dict(torch.load('DNN\_model.pth'))

else:

    losses, iter = train(DNN\_model, optimizer, loss\_fn, epochs)

    torch.save(DNN\_model.state\_dict(), 'DNN\_model.pth')

    loss\_value\_plot(losses, iter)

    plt.savefig('DNN\_loss.png')

test(DNN\_model)