Hw5_coding

May 23, 2023

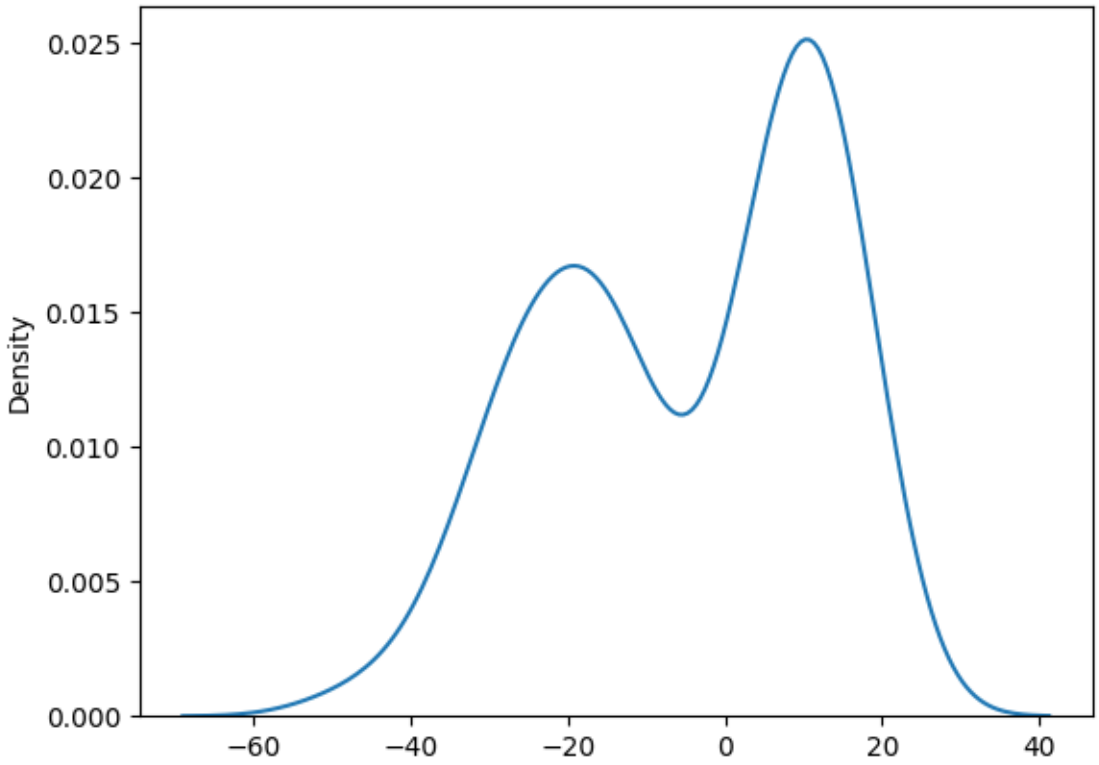
1 1.KNN estimator with a kernel funtion (20 pts)

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
np.random.seed(2023)
```

1.1 1.1Generate data points

[ ]:
```
x_trian = np.concatenate([np.random.normal(-20, 10, 20), np.random.normal(10,␣
↪5, 10)])
x_trian =x_trian.reshape((-1,1))
x_test = np.linspace(np.min(x_trian), np.max(x_trian), 100)
x_test = x_test.reshape((-1,1))
```

[ ]:
```
#  sampling  many dpoins for approximating the true distribution
# this is a multimodal density
sns.kdeplot(np.concatenate([np.random.normal(-20, 10, 100), np.random.
↪normal(10, 5, 100)]))
plt.show()
```



1.2 1.2 Complete the code and draw the pictures

Completing KNN estimator wtih a kernel funtion. The kernel funtion is a Gaussian kernel, defined by

$$K(u) = \frac{1}{\sqrt{2\pi}} exp(-\frac{u^2}{\tau}),$$

where τ is the length-scale.

Task: You should complete the code for the KNN estimator with a Gaussian kernel function and plot the estimated density with three sets of parameters, as shown below - k=3,$\tau = 2$ - k=3,$\tau = 0.2$ - k=5,$\tau = 2$

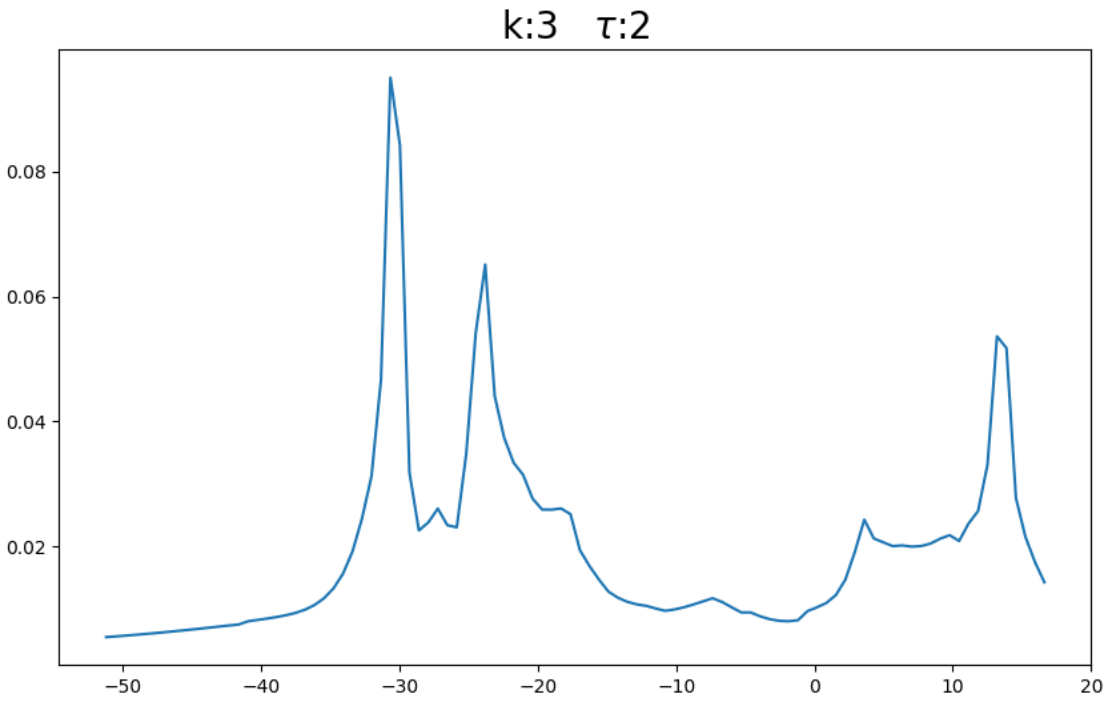
Hint: We have already generated a set of test points and saved them in the variable x_test. Your task is to plot the estimated density function $\hat{p}(x)$ based on these points. The code framework has been provided, and you can either write it from scratch or fill in the missing parts in the framework.

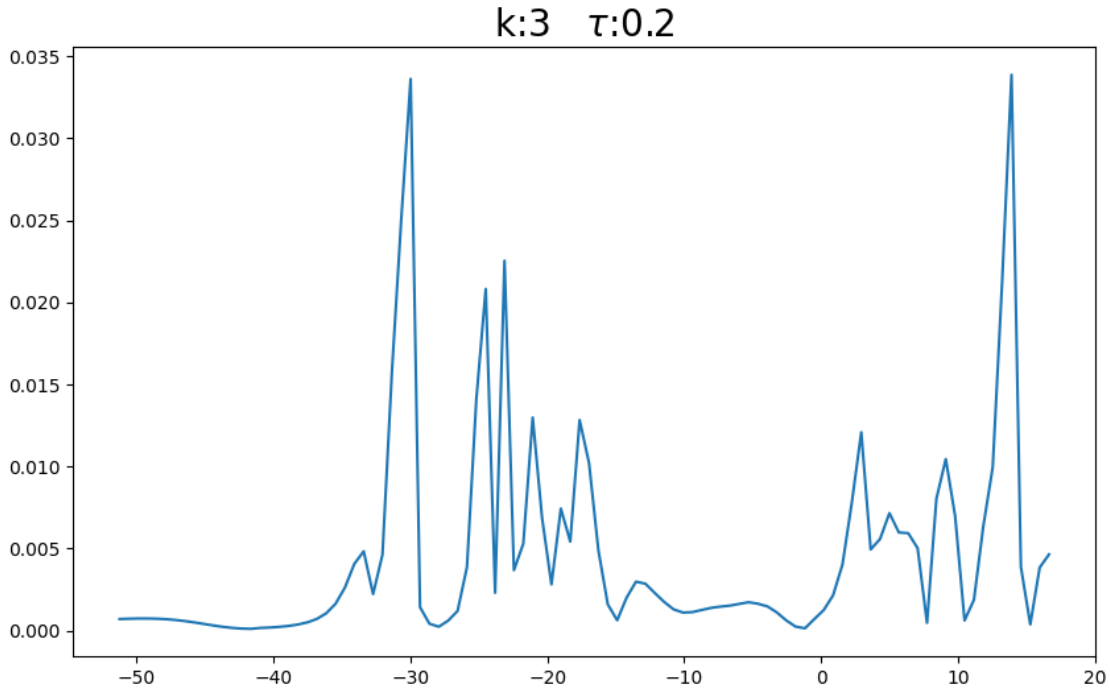
[ ]:
```
##########################
#Define your KNN estimator
##########################
class KNNKernelEstimator:
    def __init__(self, k, tau):
        self.k = k
        self.tau = tau
```

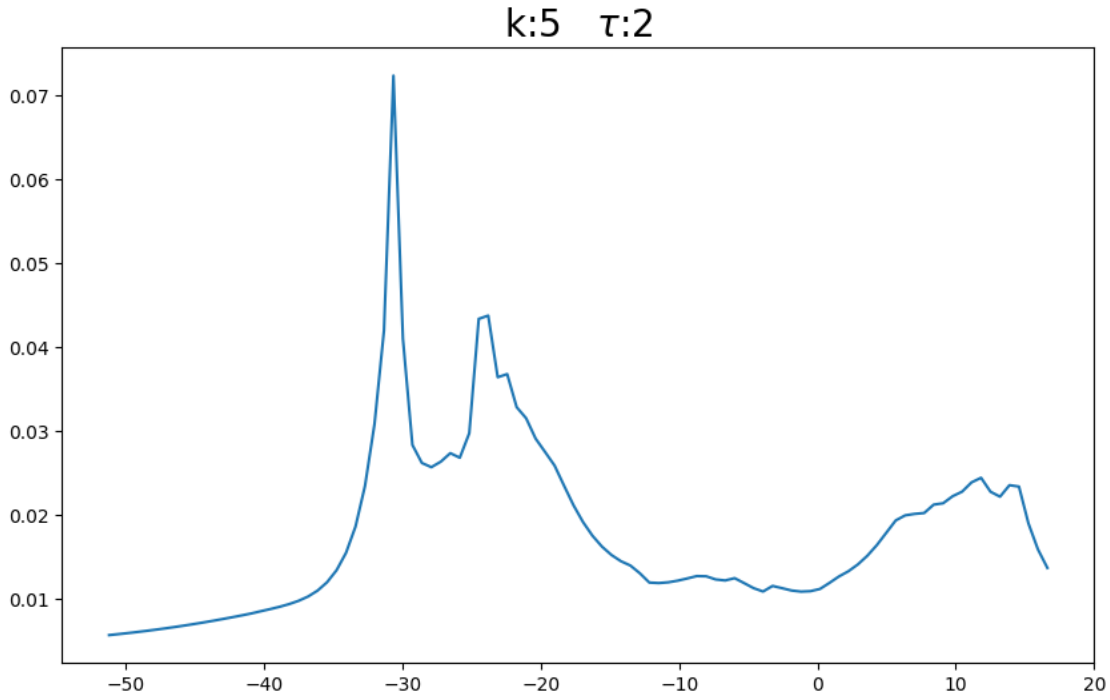
```
    def gaussian_kernel(self, u):
        return (1 / np.sqrt(2 * np.pi)) * np.exp(-1 * (u ** 2) / self.tau)

    def fit(self, x_train):
        self.x_train = x_train

    def predict(self, x_test):
        # all the density estimates for the test points
        density_estimates = []
        for x_ts in x_test:
            kernel_sum = 0
            distances = []
            for x in self.x_train:
                distances.append(np.abs(x_ts - x))
            distances = sorted(distances)
            k_nearest_distances = distances[self.k - 1]
            for x in self.x_train:
                kernel_sum += self.gaussian_kernel(np.abs(x_ts - x) /␣
↪k_nearest_distances)
            estimate = kernel_sum / (self.x_train.shape[0] *␣
↪k_nearest_distances)
            density_estimates.append(estimate)
        return np.array(density_estimates)
```

[ ]:
```
k=3
tau=2
###################
#Fill in the blanks by your code or using following template
kke = KNNKernelEstimator(k,tau)
kke.fit(x_trian)
predict_x = kke.predict(x_test)
###################

plt.figure(figsize=(10,6))
plt.plot(x_test,predict_x)
plt.title("k:"+str(k)+r"   $\tau$:"+str(tau),fontsize=20)
plt.show()

k=3
tau=0.2
###################
#Fill in the blanks by your code
kke = KNNKernelEstimator(k,tau)
kke.fit(x_trian)
predict_x = kke.predict(x_test)
###################
```

```
plt.figure(figsize=(10,6))
plt.plot(x_test,predict_x)
plt.title("k:"+str(k)+r"    $\tau$:"+str(tau),fontsize=20)
plt.show()

k=5
tau=2
###################
#Fill in the blanks by your code
kke = KNNKernelEstimator(k,tau)
kke.fit(x_trian)
predict_x = kke.predict(x_test)
###################
plt.figure(figsize=(10,6))
plt.plot(x_test,predict_x)
plt.title("k:"+str(k)+r"    $\tau$:"+str(tau),fontsize=20)
plt.show()
```




k:3 τ:0.2
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
−50
−40
−30
−20
−10
0
10
20


k:5 τ:2
0.07
0.06
0.05
0.04
0.03
0.02
0.01
−50
−40
−30
−20
−10
0
10
20

2 2.Deep Learning for classification (30 pts)

For this task, you are required to design and train a deep neural network to perform a classification task on a provided dataset. The dataset can be found at the following link: http://pan.shanghaitech.edu.cn/cloudservice/outerLink/decode?c3Vnb24xNjgyNzcwODk4OTU5c3Vnb24=

The dataset consists of a training set and test set. The training set should be used to train your model, and the test set should be used to evaluate the performance of your model.

Your goal is to achieve at least 70% accuracy on the test set using your trained model. One point is deducted for every point the accuracy decreases such as 66.2% will lose 4 points.

If your computer does not have the necessary resources to train a deep neural network, you may use the computing resources of a school computing cluster, Kaggle, or Google Colab.

Hint:The use of pre-training models is prohibited (direct zero points) and custom neural networks are encouraged.

[ ]:
```
import os
import numpy as np
# !pip install opencv-python  # the command for installing opencv,i.e, cv2
import cv2
import torch
import torch.nn as nn
import torchvision.transforms as transforms
import pandas as pd
from torch.utils.data import DataLoader, Dataset
import time
import warnings
import  torch.nn.functional as F

warnings.filterwarnings("ignore")
torch.manual_seed(3407)
```

```
c:\Users\1\.conda\envs\ml\lib\site-packages\tqdm\auto.py:21: TqdmWarning:
IProgress not found. Please update jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

[ ]:
```
<torch._C.Generator at 0x1e783a77670>
```

[ ]:
```
# load the data set
def readfile(path, label):
    image_dir = sorted(os.listdir(path))
    x = np.zeros((len(image_dir), 128, 128, 3), dtype=np.uint8)
    y = np.zeros((len(image_dir)), dtype=np.uint8)
    for i, file in enumerate(image_dir):
        img = cv2.imread(os.path.join(path, file))
        x[i, :, :] = cv2.resize(img,(128, 128))
```

```
        if label:
            y[i] = int(file.split("_")[0])
    if label:
        return x, y
    else:
        return x
```

[ ]:
```
workspace_dir = './food-11'
print("Reading data")
train_x, train_y = readfile(os.path.join(workspace_dir, "training"), True)
print("Size of training data = {}".format(len(train_x)))
test_x, test_y = readfile(os.path.join(workspace_dir, "testing"), True)
print("Size of test data = {}".format(len(test_x)))
```

```
Reading data
Size of training data = 9866
Size of test data = 3430
```

[ ]:
```
train_transform = transforms.Compose([
    ######################
    #Fill in the blanks by your code
    ##########################
    transforms.ToPILImage(),
    transforms.RandomResizedCrop(128),
    transforms.RandomHorizontalFlip(),
    transforms.RandomRotation(10),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
test_transform = transforms.Compose([
    ######################
    #Fill in the blanks by your code
    ##########################
    transforms.ToPILImage(),
    transforms.Resize(128),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
class ImgDataset(Dataset):
    def __init__(self, x, y=None, transform=None):
        self.x = x
        # label is required to be a LongTensor
        self.y = y
        if y is not None:
            self.y = torch.LongTensor(y)
        self.transform = transform
    def __len__(self):
```

```
        return len(self.x)
    def __getitem__(self, index):
        X = self.x[index]
        if self.transform is not None:
            X = self.transform(X)
        if self.y is not None:
            Y = self.y[index]
            return X, Y
        else:
            return X
```

[ ]:
```
batch_size = 128
train_set = ImgDataset(train_x, train_y, train_transform)
train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)

test_set = ImgDataset(test_x, test_y, test_transform)
test_loader = DataLoader(test_set, batch_size=batch_size, shuffle=False)
```

3 2.2 Construct deep learning model

[ ]:
```
##################################
#Define your nerual network
##################################
class Classifier(nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()
        ########################
        #   Use the pytorch API to build the neural network
        ########################
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3,␣
 ↪stride=1, padding=1)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3,␣
 ↪stride=1, padding=1)
        self.conv3 = nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3,␣
 ↪stride=1, padding=1)
        self.conv4 = nn.Conv2d(in_channels=128, out_channels=256,␣
 ↪kernel_size=3, stride=1, padding=1)
        # relu layer
        self.relu = nn.ReLU()
        # max pooling layer
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
        # fully connected layer
        self.fc1 = nn.Linear(256*8*8, 512)
        self.fc2 = nn.Linear(512, 11)
        # dropout layer
        self.dropout = nn.Dropout(p=0.5)
```

```
    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.conv2(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.conv3(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = self.conv4(x)
        x = self.relu(x)
        x = self.maxpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        x = self.relu(x)
        x = self.fc2(x)
        return x
```

3.1 2.3 trianing your model

There are some tips which I hope can help you complete the task: - If the loss function goes down too slowly, you can make the step size larger. This trick is better combined with an adaptive learning rate regulator - If the model is overfitting, you can add a dropout layer in your model. - Data augmentation is also a good way to increase model generalization. We recommend that you do this.

[ ]:
```
import matplotlib.pyplot as plt

model = Classifier().cuda()
loss = nn.CrossEntropyLoss() # use cross entropy loss function
optimizer = torch.optim.Adam(model.parameters(), lr=0.001) # use adam optimizer
num_epoch = 100

scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=75, gamma=0.1)
loss_list = []

for epoch in range(num_epoch):
    epoch_start_time = time.time()
    train_acc = 0.0
    train_loss = 0.0

    model.train()
    for i, data in enumerate(train_loader):
        inputs, labels = data
        inputs, labels = inputs.cuda(), labels.cuda()
```

```
        optimizer.zero_grad()
        outputs = model(inputs)
        batch_loss = loss(outputs, labels)
        batch_loss.backward()
        optimizer.step()

        train_acc += torch.sum(torch.argmax(outputs.cpu().data, dim=1) ==␣
↪labels.cpu()).item()
        train_loss += batch_loss.item()
    scheduler.step()

    # print the leraning rate
    for param_group in optimizer.param_groups:
        print("Learning rate:", param_group['lr'])

        #  print
    print('[%03d/%03d] %2.2f sec(s) Train Acc: %3.6f Loss: %3.6f' % \
      (epoch + 1, num_epoch, time.time()-epoch_start_time, \
      train_acc/train_set.__len__(), train_loss/train_set.__len__()))
    loss_list.append(train_loss/train_set.__len__())
```

```
Learning rate: 0.001
[001/100] 9.47 sec(s) Train Acc: 0.195925 Loss: 0.017491
Learning rate: 0.001
[002/100] 9.52 sec(s) Train Acc: 0.291506 Loss: 0.015653
Learning rate: 0.001
[003/100] 9.51 sec(s) Train Acc: 0.332860 Loss: 0.014925
Learning rate: 0.001
[004/100] 9.37 sec(s) Train Acc: 0.360531 Loss: 0.014396
Learning rate: 0.001
[005/100] 9.46 sec(s) Train Acc: 0.395500 Loss: 0.013622
Learning rate: 0.001
[006/100] 9.45 sec(s) Train Acc: 0.418001 Loss: 0.013236
Learning rate: 0.001
[007/100] 9.51 sec(s) Train Acc: 0.438982 Loss: 0.012842
Learning rate: 0.001
[008/100] 9.51 sec(s) Train Acc: 0.462700 Loss: 0.012382
Learning rate: 0.001
[009/100] 9.37 sec(s) Train Acc: 0.479019 Loss: 0.012032
Learning rate: 0.001
[010/100] 9.45 sec(s) Train Acc: 0.475674 Loss: 0.012042
Learning rate: 0.001
[011/100] 9.45 sec(s) Train Acc: 0.493412 Loss: 0.011545
Learning rate: 0.001
[012/100] 9.50 sec(s) Train Acc: 0.515711 Loss: 0.011114
Learning rate: 0.001
```

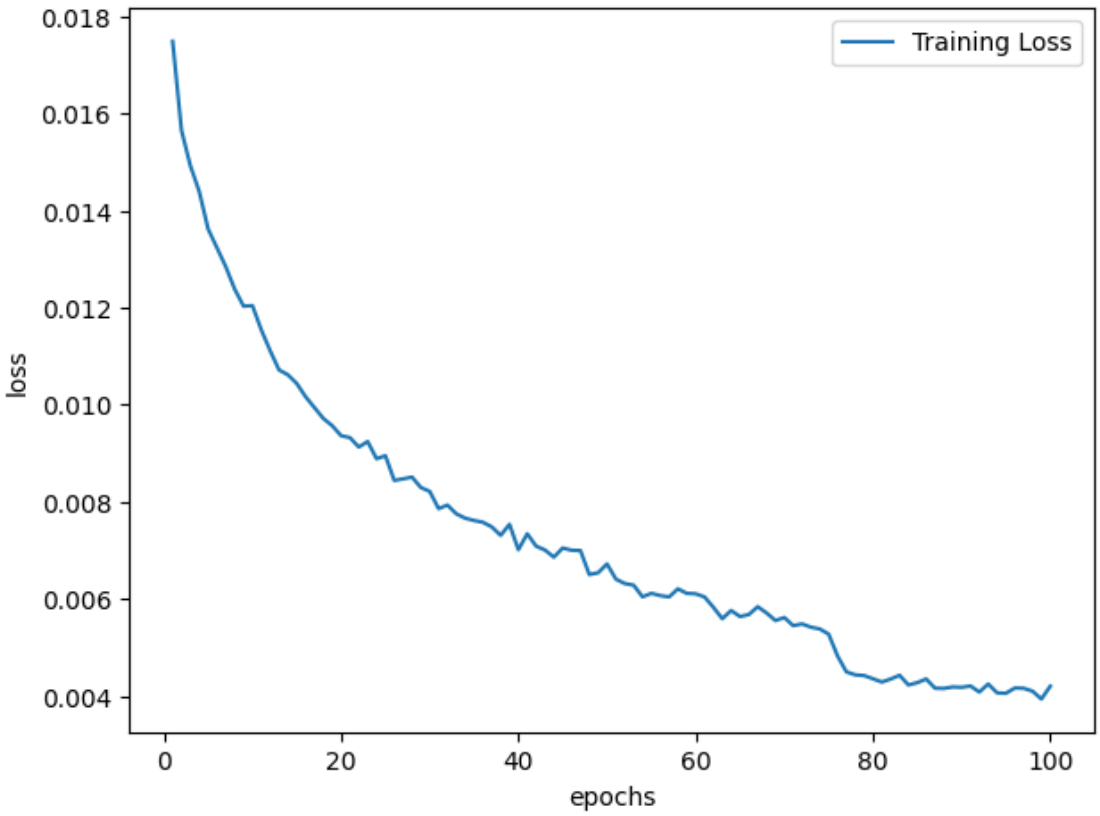
```
[013/100] 9.45 sec(s) Train Acc: 0.531624 Loss: 0.010718
Learning rate: 0.001
[014/100] 9.48 sec(s) Train Acc: 0.534056 Loss: 0.010618
Learning rate: 0.001
[015/100] 9.43 sec(s) Train Acc: 0.545408 Loss: 0.010438
Learning rate: 0.001
[016/100] 9.38 sec(s) Train Acc: 0.557267 Loss: 0.010165
Learning rate: 0.001
[017/100] 9.38 sec(s) Train Acc: 0.567809 Loss: 0.009940
Learning rate: 0.001
[018/100] 9.52 sec(s) Train Acc: 0.579668 Loss: 0.009718
Learning rate: 0.001
[019/100] 9.52 sec(s) Train Acc: 0.590107 Loss: 0.009568
Learning rate: 0.001
[020/100] 9.51 sec(s) Train Acc: 0.593047 Loss: 0.009363
Learning rate: 0.001
[021/100] 9.53 sec(s) Train Acc: 0.594871 Loss: 0.009322
Learning rate: 0.001
[022/100] 9.46 sec(s) Train Acc: 0.604906 Loss: 0.009128
Learning rate: 0.001
[023/100] 9.75 sec(s) Train Acc: 0.606325 Loss: 0.009242
Learning rate: 0.001
[024/100] 9.84 sec(s) Train Acc: 0.628522 Loss: 0.008887
Learning rate: 0.001
[025/100] 9.63 sec(s) Train Acc: 0.615548 Loss: 0.008950
Learning rate: 0.001
[026/100] 9.64 sec(s) Train Acc: 0.634806 Loss: 0.008440
Learning rate: 0.001
[027/100] 9.47 sec(s) Train Acc: 0.636023 Loss: 0.008473
Learning rate: 0.001
[028/100] 9.44 sec(s) Train Acc: 0.636732 Loss: 0.008509
Learning rate: 0.001
[029/100] 9.40 sec(s) Train Acc: 0.642003 Loss: 0.008294
Learning rate: 0.001
[030/100] 9.72 sec(s) Train Acc: 0.648490 Loss: 0.008214
Learning rate: 0.001
[031/100] 9.79 sec(s) Train Acc: 0.665518 Loss: 0.007860
Learning rate: 0.001
[032/100] 10.16 sec(s) Train Acc: 0.657511 Loss: 0.007931
Learning rate: 0.001
[033/100] 10.23 sec(s) Train Acc: 0.670282 Loss: 0.007752
Learning rate: 0.001
[034/100] 9.82 sec(s) Train Acc: 0.672714 Loss: 0.007662
Learning rate: 0.001
[035/100] 9.86 sec(s) Train Acc: 0.674133 Loss: 0.007615
Learning rate: 0.001
[036/100] 9.84 sec(s) Train Acc: 0.675046 Loss: 0.007578
Learning rate: 0.001
```

```
[037/100] 9.61 sec(s) Train Acc: 0.680114 Loss: 0.007483
Learning rate: 0.001
[038/100] 9.73 sec(s) Train Acc: 0.693797 Loss: 0.007308
Learning rate: 0.001
[039/100] 9.51 sec(s) Train Acc: 0.676161 Loss: 0.007536
Learning rate: 0.001
[040/100] 9.77 sec(s) Train Acc: 0.706061 Loss: 0.007013
Learning rate: 0.001
[041/100] 9.83 sec(s) Train Acc: 0.686905 Loss: 0.007341
Learning rate: 0.001
[042/100] 9.87 sec(s) Train Acc: 0.702311 Loss: 0.007086
Learning rate: 0.001
[043/100] 9.57 sec(s) Train Acc: 0.705048 Loss: 0.007006
Learning rate: 0.001
[044/100] 9.91 sec(s) Train Acc: 0.707683 Loss: 0.006860
Learning rate: 0.001
[045/100] 9.22 sec(s) Train Acc: 0.705656 Loss: 0.007047
Learning rate: 0.001
[046/100] 9.32 sec(s) Train Acc: 0.708291 Loss: 0.007001
Learning rate: 0.001
[047/100] 9.55 sec(s) Train Acc: 0.696128 Loss: 0.006995
Learning rate: 0.001
[048/100] 9.63 sec(s) Train Acc: 0.723698 Loss: 0.006501
Learning rate: 0.001
[049/100] 9.55 sec(s) Train Acc: 0.726840 Loss: 0.006537
Learning rate: 0.001
[050/100] 9.45 sec(s) Train Acc: 0.711028 Loss: 0.006720
Learning rate: 0.001
[051/100] 9.59 sec(s) Train Acc: 0.729171 Loss: 0.006406
Learning rate: 0.001
[052/100] 9.68 sec(s) Train Acc: 0.730387 Loss: 0.006315
Learning rate: 0.001
[053/100] 10.02 sec(s) Train Acc: 0.731299 Loss: 0.006282
Learning rate: 0.001
[054/100] 9.83 sec(s) Train Acc: 0.743665 Loss: 0.006041
Learning rate: 0.001
[055/100] 9.59 sec(s) Train Acc: 0.735759 Loss: 0.006113
Learning rate: 0.001
[056/100] 9.58 sec(s) Train Acc: 0.744273 Loss: 0.006067
Learning rate: 0.001
[057/100] 9.67 sec(s) Train Acc: 0.744172 Loss: 0.006039
Learning rate: 0.001
[058/100] 9.35 sec(s) Train Acc: 0.737482 Loss: 0.006205
Learning rate: 0.001
[059/100] 9.32 sec(s) Train Acc: 0.742043 Loss: 0.006113
Learning rate: 0.001
[060/100] 9.50 sec(s) Train Acc: 0.746402 Loss: 0.006103
Learning rate: 0.001
```

```
[061/100] 9.41 sec(s) Train Acc: 0.747618 Loss: 0.006039
Learning rate: 0.001
[062/100] 9.38 sec(s) Train Acc: 0.751470 Loss: 0.005826
Learning rate: 0.001
[063/100] 9.40 sec(s) Train Acc: 0.764950 Loss: 0.005590
Learning rate: 0.001
[064/100] 9.39 sec(s) Train Acc: 0.756842 Loss: 0.005759
Learning rate: 0.001
[065/100] 9.44 sec(s) Train Acc: 0.762214 Loss: 0.005635
Learning rate: 0.001
[066/100] 9.33 sec(s) Train Acc: 0.757855 Loss: 0.005676
Learning rate: 0.001
[067/100] 9.46 sec(s) Train Acc: 0.754612 Loss: 0.005838
Learning rate: 0.001
[068/100] 9.43 sec(s) Train Acc: 0.756740 Loss: 0.005708
Learning rate: 0.001
[069/100] 9.45 sec(s) Train Acc: 0.766369 Loss: 0.005553
Learning rate: 0.001
[070/100] 9.28 sec(s) Train Acc: 0.761403 Loss: 0.005614
Learning rate: 0.001
[071/100] 9.31 sec(s) Train Acc: 0.769816 Loss: 0.005448
Learning rate: 0.001
[072/100] 9.59 sec(s) Train Acc: 0.770424 Loss: 0.005483
Learning rate: 0.001
[073/100] 9.45 sec(s) Train Acc: 0.775188 Loss: 0.005416
Learning rate: 0.001
[074/100] 9.43 sec(s) Train Acc: 0.768903 Loss: 0.005378
Learning rate: 0.0001
[075/100] 9.39 sec(s) Train Acc: 0.772654 Loss: 0.005272
Learning rate: 0.0001
[076/100] 9.38 sec(s) Train Acc: 0.795459 Loss: 0.004824
Learning rate: 0.0001
[077/100] 9.43 sec(s) Train Acc: 0.813400 Loss: 0.004495
Learning rate: 0.0001
[078/100] 9.44 sec(s) Train Acc: 0.817758 Loss: 0.004433
Learning rate: 0.0001
[079/100] 9.35 sec(s) Train Acc: 0.817454 Loss: 0.004419
Learning rate: 0.0001
[080/100] 9.34 sec(s) Train Acc: 0.817859 Loss: 0.004353
Learning rate: 0.0001
[081/100] 9.34 sec(s) Train Acc: 0.818873 Loss: 0.004285
Learning rate: 0.0001
[082/100] 9.49 sec(s) Train Acc: 0.819177 Loss: 0.004347
Learning rate: 0.0001
[083/100] 9.43 sec(s) Train Acc: 0.820900 Loss: 0.004424
Learning rate: 0.0001
[084/100] 9.46 sec(s) Train Acc: 0.825563 Loss: 0.004224
Learning rate: 0.0001
```

```
[085/100] 9.78 sec(s) Train Acc: 0.825765 Loss: 0.004272
Learning rate: 0.0001
[086/100] 9.64 sec(s) Train Acc: 0.820596 Loss: 0.004348
Learning rate: 0.0001
[087/100] 9.47 sec(s) Train Acc: 0.826880 Loss: 0.004158
Learning rate: 0.0001
[088/100] 9.61 sec(s) Train Acc: 0.829718 Loss: 0.004154
Learning rate: 0.0001
[089/100] 9.44 sec(s) Train Acc: 0.822725 Loss: 0.004184
Learning rate: 0.0001
[090/100] 9.40 sec(s) Train Acc: 0.828705 Loss: 0.004176
Learning rate: 0.0001
[091/100] 9.53 sec(s) Train Acc: 0.828705 Loss: 0.004204
Learning rate: 0.0001
[092/100] 9.45 sec(s) Train Acc: 0.829617 Loss: 0.004079
Learning rate: 0.0001
[093/100] 9.38 sec(s) Train Acc: 0.824448 Loss: 0.004245
Learning rate: 0.0001
[094/100] 9.42 sec(s) Train Acc: 0.829921 Loss: 0.004062
Learning rate: 0.0001
[095/100] 9.47 sec(s) Train Acc: 0.830124 Loss: 0.004054
Learning rate: 0.0001
[096/100] 9.43 sec(s) Train Acc: 0.826982 Loss: 0.004162
Learning rate: 0.0001
[097/100] 9.53 sec(s) Train Acc: 0.829211 Loss: 0.004157
Learning rate: 0.0001
[098/100] 9.44 sec(s) Train Acc: 0.834482 Loss: 0.004097
Learning rate: 0.0001
[099/100] 9.39 sec(s) Train Acc: 0.835800 Loss: 0.003935
Learning rate: 0.0001
[100/100] 9.36 sec(s) Train Acc: 0.826779 Loss: 0.004199
```

[ ]:
```python
# visualize the loss as the network trained
plt.plot(range(1,len(loss_list)+1), loss_list, label='Training Loss')
plt.xlabel('epochs')
plt.ylabel('loss')
plt.legend()
plt.tight_layout()
plt.show()
```



4 Test your model

use your trained model to test the test set and print the accuracy.

[ ]:
```
model.eval()
test_acc = 0
with torch.no_grad():
    for i, data in enumerate(test_loader):
        inputs, labels = data
        inputs = inputs.cuda()
        labels = labels.cuda()
        test_pred = model(inputs)
        test_acc +=np.sum(np.argmax(test_pred.cpu().data.numpy(), axis=1) ==␣
 ↪data[1].numpy())

test_acc = test_acc/test_set.__len__()
print("Test Acc: "+str(test_acc))
```

```
Test Acc: 0.7055393586005831
```