ДОДАТОК А

Вихідний код для імітаційного моделювання

```
from __future__ import print_function
import argparse
import torch
import torch.utils.data
from torch import nn, optim
import torchvision
from torch.nn import functional as F
from torchvision import datasets, transforms
from torchvision.utils import save_image
from glob import glob
import os
from PIL import Image
from torch.utils.data import Dataset, DataLoader
import cv2

parser = argparse.ArgumentParser(description='VAE MNIST Example')
parser.add_argument('--batch-size', type=int, default=32, metavar='N',
                    help='input batch size for training (default: 128)')
parser.add_argument('--epochs', type=int, default=10, metavar='N',
                    help='number of epochs to train (default: 10)')
parser.add_argument('--no-cuda', action='store_true', default=False,
                    help='enables CUDA training')
parser.add_argument('--seed', type=int, default=1, metavar='S',
                    help='random seed (default: 1)')
parser.add_argument('--log-interval', type=int, default=10, metavar='N',
                    help='how many batches to wait before logging training status')
parser.add_argument('--model-version', type=int, default=1001, metavar='N',
                    help='model version')
args = parser.parse_args()
args.cuda = not args.no_cuda and torch.cuda.is_available()

torch.manual_seed(args.seed)

device = torch.device("cuda" if args.cuda else "cpu")

kwargs = {'num_workers': 1, 'pin_memory': True} if args.cuda else {}


class Data(Dataset):
    def __init__(self, root_path: str, transform=None):
        super().__init__()
        self.transform = transform
        self.items = glob(os.path.join(root_path, "*.jpg"))

    def __len__(self):
        return len(self.items)

    def __getitem__(self, idx):
        # img = Image.open(self.items[idx])
        img = cv2.imread(self.items[idx], cv2.IMREAD_GRAYSCALE)
        if self.transform:
            img = self.transform(img)
        return img


normilizer = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                  std=[0.229, 0.224, 0.225])


def get_data(path_to_folder: str):
    # train_dataset = torchvision.datasets.ImageFolder(root=path_to_folder,
    #                                     transform=transforms.Compose([transforms.ToTensor(), normilizer])
    #                                     )
```

```
    train_dataset = torchvision.datasets.ImageFolder(root=path_to_folder,
                                        transform=transforms.Compose([  # transforms.Grayscale(),
                                            # transforms.RandomSizedCrop(max((224, 224))),
                                            # transforms.RandomCrop(100, padding=4),
                                            transforms.RandomRotation(10),
                                            transforms.RandomHorizontalFlip(),
                                            transforms.ToTensor()])
                                        )
    train_loader = DataLoader(
        train_dataset,
        batch_size=args.batch_size,
        num_workers=0,
        shuffle=True
    )

    return train_loader


train_loader = get_data(path_to_folder='../dataset_for_classifier/train')
validation_loader = get_data(path_to_folder='../dataset_for_classifier/val')
test_loader = get_data(path_to_folder='../dataset_for_classifier/val')


# 224 * 224 * 3

class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()

        # self.bnorm1 = nn.BatchNorm2d(64)

        self.im_size = 224 * 224 * 3
        self.conv1 = nn.Conv2d(3, 32, 3)  # 32
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(32, 64, 3)  # 64
        self.dropout = nn.Dropout(0.2)
        self.fc1 = nn.Linear(64 * 54 * 54, 400)  # 900
        self.bnorm1 = nn.BatchNorm1d(num_features=400)
        self.fc21 = nn.Linear(400, 64)  # 64
        self.fc22 = nn.Linear(400, 64)

        self.fc3 = nn.Linear(64, 900)  # 900
        self.bnorm2 = nn.BatchNorm1d(num_features=900)
        self.fc4 = nn.Linear(900, self.im_size)

    # self.fc5 = nn.Linear(900, self.im_size)

    # self.im_size = 224 * 224 * 3
    # self.conv1 = nn.Conv2d(3, 4, 3) # 32
    # self.pool = nn.MaxPool2d(2, 2)
    # self.conv2 = nn.Conv2d(4, 8, 3) # 64
    # self.dropout = nn.Dropout(0.2)
    # self.fc1 = nn.Linear(8 * 54 * 54, 64)  # 900
    # self.fc21 = nn.Linear(64, 16) # 64
    # self.fc22 = nn.Linear(64, 16)
    # self.fc3 = nn.Linear(16, 128) # 900
    # self.fc4 = nn.Linear(128, self.im_size)

    # self.fc2 = nn.Linear(20, 400)
    # self.fc1 = nn.Linear(self.im_size, 400)
    # self.fc21 = nn.Linear(400, 20)
    # self.fc22 = nn.Linear(400, 20)
    # self.fc3 = nn.Linear(20, 400)
```

```
    # self.fc4 = nn.Linear(400, self.im_size)

    def encode(self, x):
        x = self.conv1(x)
        x = self.pool(F.leaky_relu(x))
        # x = self.dropout(x)  # drop=v3 if not drop=v2
        x = self.pool(F.leaky_relu(self.conv2(x)))
        x = self.dropout(x)
        x = x.view(-1, 64 * 54 * 54)
        h1 = F.leaky_relu(self.fc1(x))
        h1 = self.bnorm1(h1)
        # F.leaky_relu()
        return F.leaky_relu(self.fc21(h1)), F.leaky_relu(self.fc22(h1))

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5 * logvar)
        eps = torch.randn_like(std)
        return mu + eps * std

    def decode(self, z):
        h3 = F.leaky_relu(self.fc3(z))
        h3 = self.bnorm2(h3)
        h4 = F.leaky_relu(self.fc4(h3))

        return torch.sigmoid(h4)  # F.sigmoid(h4) # torch.sigmoid(h4)

        # return torch.sigmoid(self.fc5(h4))
        # h3 = torch.sigmoid(self.fc2(z))
        # return h3

    def forward(self, x):
        # mu, logvar = self.encode(x.view(-1, self.im_size))
        mu, logvar = self.encode(x)
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar


version_model = args.model_version
log_file_name = f'logs/log_{version_model}.txt'
losses_file_name = f'losses/losses_{version_model}.txt'
model_folder_names = f'vae_models_{version_model}'
result_images_folder_name = f'results_{version_model}'

if not 'logs' in os.listdir('.'):
    os.mkdir('logs')

if not 'losses' in os.listdir('.'):
    os.mkdir('losses')

if not model_folder_names in os.listdir('.'):
    os.mkdir(model_folder_names)

if not result_images_folder_name in os.listdir('.'):
    os.mkdir(result_images_folder_name)


model = VAE().to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)
# optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.8)
# scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', verbose=True)
scheduler = torch.optim.lr_scheduler.CyclicLR(optimizer,
                                              cycle_momentum=False,
                                              step_size_up=2500,  # 500
```

```
                                base_lr=0.00001,
                                max_lr=0.01)


# Reconstruction + KL divergence losses summed over all elements and batch
def loss_function(recon_x, x, mu, logvar):
    BCE = F.binary_cross_entropy(recon_x, x.view(-1, 224 * 224 * 3), reduction='sum')
    # BCE = F.binary_cross_entropy(recon_x, x.view(-1, 224 * 224), reduction='sum')
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())

    return BCE + KLD


def train(epoch):
    model.train()
    train_loss = 0
    for batch_idx, (data, _) in enumerate(train_loader):
        if batch_idx==1:
            break
        data = data.to(device)
        optimizer.zero_grad()
        recon_batch, mu, logvar = model(data)
        loss = loss_function(recon_batch, data, mu, logvar)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()
        scheduler.step()

        if batch_idx % args.log_interval == 0:
            with open(log_file_name, "a") as f:
                f.write('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                    epoch, batch_idx * len(data), len(train_loader.dataset),
                        100. * batch_idx / len(train_loader),
                        loss.item() / len(data)) + '\n')
                f.write(f'lr: {scheduler.get_lr()}\n')

            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                    100. * batch_idx / len(train_loader),
                    loss.item() / len(data)))
            print(f'lr: {scheduler.get_lr()}')

    with open(log_file_name, "a") as f:
        f.write('====> Epoch: {} Average loss: {:.4f}'.format(
            epoch, train_loss / len(train_loader.dataset)) + '\n')

    print('====> Epoch: {} Average loss: {:.4f}'.format(
        epoch, train_loss / len(train_loader.dataset)))

    # print(optim.state_dict())
    return train_loss / len(train_loader.dataset)


def val(epoch):
    model.eval()
    val_loss = 0
    with torch.no_grad():
        for i, (data, _) in enumerate(validation_loader):
            data = data.to(device)
            recon_batch, mu, logvar = model(data)
            val_loss += loss_function(recon_batch, data, mu, logvar).item()

    val_loss /= len(test_loader.dataset)
```

```
    with open(log_file_name, "a") as f:
        f.write('====> Val loss: {:.4f}'.format(val_loss) + '\n')

    print('====> Val loss: {:.4f}'.format(val_loss))

    # scheduler.step(val_loss)
    return val_loss


def test(epoch):
    model.eval()
    test_loss = 0
    with torch.no_grad():
        for i, (data, _) in enumerate(test_loader):
            data = data.to(device)
            recon_batch, mu, logvar = model(data)
            test_loss += loss_function(recon_batch, data, mu, logvar).item()
            if i == 0:
                n = min(data.size(0), 8)
                # comparison = torch.cat([data[:n], recon_batch.view(args.batch_size, 1, 224, 224)[:n]])
                comparison = torch.cat([data[:n], recon_batch.view(args.batch_size, 3, 224, 224)[:n]])
                save_image(comparison.cpu(), f'{result_images_folder_name}/reconstruction_{epoch}.png', nrow=n)

    test_loss /= len(test_loader.dataset)
    with open(log_file_name, "a") as f:
        f.write('====> Test set loss: {:.4f}'.format(test_loss) + '\n')

    print('====> Test set loss: {:.4f}'.format(test_loss))
    return test_loss


if __name__ == "__main__":
    for epoch in range(1, args.epochs + 1):
        train_loss_e = train(epoch)
        val_loss_e = val(epoch)
        test_loss_e = test(epoch)

        with open(losses_file_name, "a") as f:
            # f.write(f'Train: {train_loss_e} Val: {val_loss_e} \n')
            f.write(f'Train: {train_loss_e} Val: {val_loss_e} Test: {test_loss_e} \n')

        with torch.no_grad():
            sample = torch.randn(64, 64).to(device)
            sample = model.decode(sample).cpu()
            # save_image(sample.view(64, 3, 224, 224), 'results/sample_' + str(epoch) + '.png')
            save_image(sample.view(64, 3, 224, 224), f'{result_images_folder_name}/sample_{epoch}.png')

        torch.save(model.state_dict(), f'{model_folder_names}/model_chekpoint_{epoch}')

import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import torchvision
import torch
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, utils
from torchvision import models
from glob import glob
from PIL import Image
from scipy import stats

from tqdm import tqdm
```

```
from typing import List, Dict, Tuple
from shutil import copyfile

import torch
import torch.nn as nn
import torch.nn.functional as F


from sklearn import metrics

path_to_synt = '../processed_for_VGG16/synt'
path_to_real = '../processed_for_VGG16/real'


class Identity(torch.nn.Module):
    def __init__(self):
        super(Identity, self).__init__()

    def forward(self, x):
        return x


class Data(Dataset):
    def __init__(self, root_path: str, transform=None):
        super().__init__()
        self.transform = transform
        self.items = glob(os.path.join(root_path, "*.jpg"))

    def __len__(self):
        return len(self.items)

    def __getitem__(self, idx):
        img = Image.open(self.items[idx])
        if self.transform:
            img = self.transform(img)
        return img


normilizer = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                  std=[0.229, 0.224, 0.225])
real_dataset = Data(root_path=path_to_real, transform=transforms.Compose([transforms.ToTensor(),
normilizer]))
synt_dataset = Data(root_path=path_to_synt, transform=transforms.Compose([transforms.ToTensor(),
normilizer]))

batch_size = 32
real_loader = DataLoader(
    real_dataset,
    batch_size=batch_size,
    num_workers=10,
    shuffle=False
)

synt_loader = DataLoader(
    synt_dataset,
    batch_size=batch_size,
    num_workers=10,
    shuffle=False
)

alexnet = models.alexnet(pretrained=True)
alexnet.classifier = alexnet.classifier[:5]
```

```
mobilenetv2 = models.mobilenet_v2(pretrained=True)
mobilenetv2 = mobilenetv2.features[:18]

model_vgg13_bn = models.vgg13_bn(pretrained=True)
model_vgg13_bn.classifier = model_vgg13_bn.classifier[:1]

model_vgg16_bn = models.vgg16_bn(pretrained=True)
model_vgg16_bn.classifier = model_vgg16_bn.classifier[:4]

mnasnet = models.mnasnet1_0(pretrained=True)
mnasnet = mnasnet.layers[:14]

mnasnet0_5 = models.mnasnet0_5(pretrained=True)
mnasnet0_5 = mnasnet0_5.layers[:14]

resnet152 = models.resnet152(pretrained=True)
resnet152.fc = Identity()

resnext101 = models.resnext101_32x8d(pretrained=True)
resnext101.fc = Identity()

dnn_models = {'alexnet': alexnet,
              'mobilenet_v2': mobilenetv2,
              'model_vgg13_bn': model_vgg13_bn,
              'vgg16_bn': model_vgg16_bn,
              'mnasnet': mnasnet,
              'mnasnet0_5': mnasnet0_5,
              'resnet152': resnet152,
              'resnext101': resnext101
              }


# TODO:
# delete if and change path

path_to_features = '../features'
def process(model_name: str,
            model: torch.nn.modules.container.Sequential,
            real_or_synt: str,
            current_data_loader: DataLoader):

    path_to = f'{path_to_features}/{real_or_synt}/{model_name}'

    if not os.path.isdir(path_to_features):
        os.mkdir(path_to_features)

    if not os.path.isdir(f'{path_to_features}/{real_or_synt}'):
        os.mkdir(f'{path_to_features}/{real_or_synt}')

    if not os.path.isdir(f'{path_to_features}/{real_or_synt}/{model_name}'):
        os.mkdir(f'{path_to_features}/{real_or_synt}/{model_name}')

    for idx, real_img in enumerate(current_data_loader):
        out = model(real_img)
        features = out.view([batch_size, -1]).detach().numpy()
        path_to_batch = f'{path_to}/batch_{idx + 1}.npy'
        np.save(path_to_batch, features)
        # if idx == 2:
        #     break


for model_name, model in dnn_models.items():
```

```
print(model_name)
process(model_name=model_name, model=model, real_or_synt='real', current_data_loader=real_loader)
process(model_name=model_name, model=model, real_or_synt='synt', current_data_loader=synt_loader)
```

ДОДАТОК Б

Відомість атестаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	79с.
		.
	Інші документи	
2.	Презентаційні матеріали	плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Норцова А.В			Розробка методу генерації масивів даних для навчання глибинних нейронних мереж	Шифр групи	Код напр./спец.
Перевірив		Бодянський Є.В.				СШІм-18-2	122
Н.контр.		.				ХНУРЕ кафедра ШІ	
Затв.		Філатов В.О.					