实验 2 ：对抗网络训练及模型脆弱性分析

一、实验目的

掌握典型应用对抗网络训练实验方法，掌握对抗数据样本模型脆弱性分析方法。

二、实验原理

针对深度网络中可能存在的样本对抗性攻击方法，目前主流的防御方法从模型层面和数据层面对攻击行为进行防御。其中模型层面的防御方法通过在训练阶段修改原始深度网络结构、或添加外部网络作为附加网络为深度网络添加防御措施，使深度网络在面对对抗样本时能输出正常结果；另一类数据层面防御方法通过在训练阶段将对抗样本注入训练数据集训练网络、或在模型推理阶段处理对抗样本数据进行防御，使深度模型在训练过程中便具备对于对抗样本的鲁棒性，或直接使模型面对的输入样本更贴近未受攻击的样本输入。

三、实验内容

本次实验从数据层面的防御方法入手，采用在训练阶段将对抗样本直接注入训练数据集的形式添加防御，具体来说，使用投影梯度下降法（PGD, Project Gradient Descent）作为攻击方法，将受到 PGD 攻击的数据代替原始 MNIST 数据构建分类模型训练数据集，进行对

抗性训练，并分析对抗性训练方法的脆弱性。

四、实验步骤

本实验具体步骤如下所示，方便大家理解和使用。（**代码示例仅作参考，同学们有更好的思路可以自行编写代码**）

（1）参数定义

新建一个 python 脚本文件 argument.py，用于整理对抗性训练实验中的各类实验参数。

首先引用依赖包，并定义参数调用函数 parser()：

```
import argparse


def parser():
    parser = argparse.ArgumentParser(description='parser for AI security experiment1')
```

将模型训练的路径相关信息写入参数函数：

```
    # 模型基本参数
    # 选择模型运行模式（训练、验证、测试、结果可视化）
    parser.add_argument('--todo', choices=['train', 'valid', 'test', 'visualize'], default='train')
    # 选择模型训练加载数据集（默认为torchvision中的mnist数据集）
    parser.add_argument('--dataset', default='mnist', help='use what dataset')
    # 选择下载数据集存放位置
    parser.add_argument('--data_root', default='.', help='the directory to save the dataset')
    # 选择训练日志存放位置
    parser.add_argument('--log_root', default='log', help='the directory to save the logs')
    # 选择训练模型存放位置
    parser.add_argument('--model_root', default='checkpoint', help='the directory to save the models')
    # 选择测试/验证时加载模型存放位置
    parser.add_argument('--load_checkpoint', default='./model/default/model.pth')
    # 选择是否启用对抗性训练
    parser.add_argument('--adv_train', action='store_true')
```

将模型训练超参数写入参数函数包括训练 batchsize、训练轮次 epoch 以及模型学习率 learning rate：

```
    # 模型训练参数
    # 训练batch size（电脑训练缓慢尝试降低batch size）
    parser.add_argument('--batch_size', '-b', type=int, default=4, help='batch size')
    # 模型训练epoch
    parser.add_argument('--max_epoch', '-m_e', type=int, default=50, help='the maximum epoch of model training')
    # 模型训练学习率
    parser.add_argument('--learning_rate', '-lr', type=float, default=1e-4, help='learning rate')
```

将模型对抗性训练中使用 PGD 攻击方法的超参数写入参数函数，方便调整攻击强度：

```
# PGD对抗性训练相关参数
# PGD攻击参数ε
parser.add_argument('--epsilon', '-e', type=float, default=0.3, help='maximum perturbation of adversaries')
# PGD攻击步长α
parser.add_argument('--alpha', '-a', type=float, default=0.01, help='movement multiplier per iteration')
# PGD攻击迭代次数
parser.add_argument('--k', '-k', type=int, default=40, help='iteration when generating adversarial examples')
# 模型对抗性训练扰动类型
parser.add_argument('--perturbation_type', '-p', choices=['linf', 'l2'], default='linf',
                    help='the type of the perturbation (linf or l2)')
```

将模型训练过程需要输出日志的超参数写入参数函数，包括模型验证的频率、模型参数保存频率以及攻击图像保存图像，最后返回参数：

```
# 训练输出日志参数
# 模型验证频率
parser.add_argument('--n_eval_step', type=int, default=100, help='number of iteration per one evaluation')
# 模型保存checkpoint频率
parser.add_argument('--n_checkpoint_step', type=int, default=100, help='number of iteration to save checkpoint')
# 训练过程中结果保存频率
parser.add_argument('--n_store_image_step', type=int, default=100, help='number of iteration to save adversaries')

return parser.parse_args()
```

（2）对抗性训练实验基础模型搭建

新建一个 python 脚本文件 model.py，用于定义对抗性训练实验中的基础深度分类网络。

首先定义网络参数，实验使用 Lenet 网络结构，首先使用 kernel size=5，stride=1，padding=2 的卷积层对输入的 28×28×1 手写数字图像进行特征提取，并通过池化层，得到 14×14×32 的特征图像；然后

通过 kernel size=5，stride=1，padding=2 的卷积层对该特征图进行进一步特征提取，并通过池化层，得到 7×7×64 的深层特征图像；网络通过设计一个 flatten 层将三通道的深层特征图转换为单通道特征向量，维度为 7×7×64，该特征向量通过两个全连接层，将最终的输出尺寸与分类数目对齐，完成网络搭建，深度网络搭建代码如下：

```
import torch
import torch.nn as nn
```

```
class Model(nn.Module):
    """
        定义对抗性训练实验中基础分类网络结构
    """
    def __init__(self, i_c=1, n_c=10):
        super(Model, self).__init__()

        self.conv1 = nn.Conv2d(i_c, 32, 5, stride=1, padding=2, bias=True)
        self.pool1 = nn.MaxPool2d((2, 2), stride=(2, 2), padding=0)

        self.conv2 = nn.Conv2d(32, 64, 5, stride=1, padding=2, bias=True)
        self.pool2 = nn.MaxPool2d((2, 2), stride=(2, 2), padding=0)

        self.flatten = Expression(lambda tensor: tensor.view(tensor.shape[0], -1))
        self.fc1 = nn.Linear(7 * 7 * 64, 1024, bias=True)
        self.fc2 = nn.Linear(1024, n_c)
```

其中 flatten 函数的构建方法为：

```
class Expression(nn.Module):
    """
        定义模型中需要的flatten层
    """
    def __init__(self, func):
        super(Expression, self).__init__()
        self.func = func

    def forward(self, input):
        return self.func(input)
```

随后定义网络的前向计算过程，前向计算过程的输入为手写数

字图像对应的 tensor，输出为分类结果：

```
    def forward(self, x_i, _eval=False):

        if _eval:
            self.eval()
        else:
            self.train()

        x_o = self.conv1(x_i)
        x_o = torch.relu(x_o)
        x_o = self.pool1(x_o)

        x_o = self.conv2(x_o)
        x_o = torch.relu(x_o)
        x_o = self.pool2(x_o)

        x_o = self.flatten(x_o)

        x_o = torch.relu(self.fc1(x_o))

        self.train()

        return self.fc2(x_o)
```

最后执行一下测试，随机生成一个和手写数字图像尺寸相同的 tensor，作为网络的输入，输出的尺寸应该为[batch size, 10]。

```
if __name__ == '__main__':
    # 测试以MNIST数据集为输入时输出是否正常
    # input: (Batch, Channel, Image width, Image height)
    # output: (Batch, n_classification)
    input_image = torch.FloatTensor(4, 1, 28, 28)
    classification_model = Model()
    print(classification_model(input_image).size())
```

（3）网络训练工具性函数搭建

新建一个 python 脚本文件 utils.py，用于存放网络训练的工具性函数。脚本需要的依赖包包括 os、logging、numpy、torch：

```
import os
import logging
import numpy as np
import torch
```

1）数据类型转换函数

包括将 numpy 数组转换成 tensor 张量的 numpy2tensor 函数和将列表转换成 tensor 张量的 list2tensor 函数：

```
# transfer numpy to tensor
def numpy2tensor(array):
    tensor = torch.from_numpy(array)
    return tensor
```

```
# transfer list to tensor
def list2tensor(_list):
    array = np.array(_list)
    return numpy2tensor(array)
```

2）one-hot 编码函数

将模型的分类结果 id 列表转换为 one-hot 编码形式：

```
# define one hot coding
def one_hot(ids, n_class):
    """
        ids: (list, ndarray) shape:[batch_size]
        out_tensor:FloatTensor shape:[batch_size, depth]
    """
    assert len(ids.shape) == 1, 'the ids should be 1-D'
    out_tensor = torch.zeros(len(ids), n_class)
    out_tensor.scatter_(1, ids.cpu().unsqueeze(1), 1.)
    return out_tensor
```

3）模型加载/保存函数

包括将模型的参数进行保存的 save_model 函数和将保存的模型

进行加载的 load_model 函数：

```python
# load model checkpoints
def load_model(model, file_name):
    model.load_state_dict(
        torch.load(file_name, map_location=lambda storage, loc: storage))


# save model to checkpoint
def save_model(model, file_name):
    torch.save(model.state_dict(), file_name)
```

4）模型性能评估

将模型的分类结果与数据集中的标签值进行对比已得到模型分类准确率：

```python
# evaluate prediction and labels
def evaluate(_input, _target, method='mean'):
    correct = (_input == _target).astype(np.float32)
    if method == 'mean':
        return correct.mean()
    else:
        return correct.sum()
```

5）训练日志保存函数

```python
# save log to .txt
def create_logger(save_path='', file_type='', level='debug'):
    if level == 'debug':
        _level = logging.DEBUG
    elif level == 'info':
        _level = logging.INFO

    logger = logging.getLogger()
    logger.setLevel(_level)
    cs = logging.StreamHandler()
    cs.setLevel(_level)
    logger.addHandler(cs)

    if save_path != '':
        file_name = os.path.join(save_path, file_type + '_log.txt')
        fh = logging.FileHandler(file_name, mode='w')
        fh.setLevel(_level)
        logger.addHandler(fh)
    return logger
```

（4）PGD 攻击函数搭建

PGD 攻击函数的原理在前次实验中已介绍过，此处便不再赘述，PGD 攻击函数首先选择扰动类型，即攻击是沿着训练梯度的 l2 范数方向还是 l∞范数方向进行，PGD 攻击函数根据扰动类型为输入数据添加扰动，规范化数据后返回扰动图像。

首先定义单次扰动下 PGD 攻击函数：

```python
import torch
import torch.nn.functional as F


def project(x, original_x, epsilon, _type='linf'):

    if _type == 'linf':
        max_x = original_x + epsilon
        min_x = original_x - epsilon

        x = torch.max(torch.min(x, max_x), min_x)

    elif _type == 'l2':
        dist = (x - original_x)
        dist = dist.view(x.shape[0], -1)
        dist_norm = torch.norm(dist, dim=1, keepdim=True)

        mask = (dist_norm > epsilon).unsqueeze(2).unsqueeze(3)

        dist = dist / dist_norm
        dist *= epsilon
        dist = dist.view(x.shape)

        x = (original_x + dist) * mask.float() + x * (1 - mask.float())

    else:
        raise NotImplementedError
    return x
```

然后加入梯度计算及迭代循环，构建最终的 PGD 攻击函数，首

先对 PGD 攻击函数进行初始化：

```python
class PGD_attack():
    """
        Fast gradient sign un-targeted adversarial attack, minimizes the initial class activation
        with iterative grad sign updates
    """
    def __init__(self, model, epsilon, alpha, min_val, max_val, max_iters, _type='linf'):
        self.model = model
        # Maximum perturbation(ε)
        self.epsilon = epsilon
        # Movement multiplier per iteration(α)
        self.alpha = alpha
        # Minimum value of the pixels
        self.min_val = min_val
        # Maximum value of the pixels
        self.max_val = max_val
        # Maximum numbers of iteration to generated adversaries
        self.max_iters = max_iters
        # The perturbation of epsilon
        self._type = _type
```

然后定义扰动，计算模型训练的梯度后，通过设定好的迭代次数在输入数据中迭代添加 PGD 攻击，并将攻击后的输入图像返回：

```python
    def perturb(self, original_images, labels, reduction4loss='mean', random_start=False):
        # The adversaries created from random close points to the original data
        if random_start:
            rand_perturb = torch.FloatTensor(original_images.shape).uniform_(-self.epsilon, self.epsilon)
            x = original_images + rand_perturb
            x.clamp_(self.min_val, self.max_val)
        else:
            x = original_images.clone()

        x.requires_grad = True

        with torch.enable_grad():
            for _iter in range(self.max_iters):
                outputs = self.model(x, _eval=True)

                loss = F.cross_entropy(outputs, labels, reduction=reduction4loss)

                if reduction4loss == 'none':
                    grad_outputs = torch.ones(loss.shape)
                else:
                    grad_outputs = None

                grads = torch.autograd.grad(loss, x, grad_outputs=grad_outputs, only_inputs=True)[0]
                x.data += self.alpha * torch.sign(grads.data)
                x = project(x, original_images, self.epsilon, self._type)
                x.clamp_(self.min_val, self.max_val)

        return x
```

（5）模型训练代码搭建

新建一个 python 脚本文件 train.py，用于存放网络训练函数。脚本需要的依赖包如下，同时需要将前面构建的参数、攻击函数、工具函数和网络结构函数进行引用：

```
# AI security experiment 1 training code

# import dependency packages
import os
import torch
import torchvision
from torch.utils.data import DataLoader
import torch.nn.functional as F
from time import time

# import argument/model/attack/utils
from model import Model
from argument import parser
from utils import create_logger, numpy2tensor, evaluate, save_model
from attack import PGD_attack
```

1）定义训练过程

首先定义训练器，并定义初始化函数，将超参数、训练日志以及攻击方式作为初始化参数传入：

```
class Trainer():
    def __init__(self, args, logger, attack):
        self.args = args
        self.logger = logger
        self.attack = attack
```

定义训练函数，将深度网络模型、训练数据集、验证数据集和对抗性训练设置传入函数，并定义优化器 opt：

```
def train(self, model, tr_loader, va_loader=None, adv_train=False):
    # define args
    args = self.args
    logger = self.logger
    _iter = 0
    begin_time = time()

    # define optimizer
    opt = torch.optim.Adam(model.parameters(), args.learning_rate)
```

开始模型训练循环，每个训练 iteration，将数据集中的数据和标签传入。如果启用对抗性训练，则将输入数据进行扰动后再传入训练器，如果不启用对抗性训练，则直接将输入数据传入训练器。使用模型前向过程的输出和参考标签计算交叉熵损失函数，并利用 loss 值反向传播更新网络参数：

```
    # start model training
    for epoch in range(1, args.max_epoch + 1):
        for data, label in tr_loader:
            if adv_train:
                """
                    When training, the adversarial example is created from a random
                    close point to the original data point. If in evaluation mode,
                    just start from the original data point.
                """
                # use random perturbed data to replace original input data
                adv_data = self.attack.perturb(data, label, 'mean', True)
                output = model(adv_data, _eval=False)
            else:
                output = model(data, _eval=False)

            # calculate training loss(by cross entropy) and step optimizer
            loss = F.cross_entropy(output, label)
            opt.zero_grad()
            loss.backward()
            opt.step()
```

在模型训练轮次达到验证轮次时，模型输出测试结果。首先输出当前输入数据下，分类网络训练性能指标。如果启用对抗性训练，则以未受攻击的数据的模型分类结果作为基础网络性能，将受到攻击

的数据的模型分类结果作为对抗性训练网络性能；如果不启用对抗性训练，同样以未受攻击的数据的模型分类结果作为基础网络性能，将受到攻击的数据的模型分类结果作为对抗性训练网络性能：

```
# eval model every n_eval_step iterations
if _iter % args.n_eval_step == 0:
    if adv_train:
        # evaluate model performance when adversarial training able
        with torch.no_grad():
            stand_output = model(data, _eval=True)

        # calculate model classification accuracy by un-perturbed data
        pred = torch.max(stand_output, dim=1)[1]
        std_acc = evaluate(pred.cpu().numpy(), label.cpu().numpy()) * 100
        # calculate model classification accuracy by perturbed data
        pred = torch.max(output, dim=1)[1]
        adv_acc = evaluate(pred.cpu().numpy(), label.cpu().numpy()) * 100

    else:
        # evaluate model performance when adversarial training disable
        adv_data = self.attack.perturb(data, label, 'mean', False)
        with torch.no_grad():
            adv_output = model(adv_data, _eval=True)

        # calculate model classification accuracy by perturbed data
        pred = torch.max(adv_output, dim=1)[1]
        adv_acc = evaluate(pred.cpu().numpy(), label.cpu().numpy()) * 100
        # calculate model classification accuracy by un-perturbed data
        pred = torch.max(output, dim=1)[1]
        std_acc = evaluate(pred.cpu().numpy(), label.cpu().numpy()) * 100
```

将模型性能指标存入日志文件：

```
# print training info to console
logger.info('epoch: %d, iter: %d, spent %.2f s, tr_loss: %.3f' % (
    epoch, _iter, time() - begin_time, loss.item()))

# print evaluate info to console
logger.info('standard acc: %.3f %%, robustness acc: %.3f %%' % (
    std_acc, adv_acc))
```

然后加载验证数据集，对当前训练状态下的模型进行验证，得到验证准确率：

```
                # validation if necessary
                if va_loader is not None:
                    va_acc, va_adv_acc = self.test(model, va_loader, True)
                    va_acc, va_adv_acc = va_acc * 100.0, va_adv_acc * 100.0
                    logger.info('\n' + '=' * 30 + ' evaluation ' + '=' * 30)
                    logger.info('test acc: %.3f %%, test adv acc: %.3f %%' % (
                        va_acc, va_adv_acc))
                    logger.info('=' * 28 + ' end of evaluation ' + '=' * 28 + '\n')
```

重置时间计数器，并在设定好的图像保存轮次和模型保存轮次对当前的受 PGD 攻击前后对比图像数据、当前的模型参数进行保存。

```
            begin_time = time()

        # save original data and attacked data every n_store_image_step iterations
        if _iter % args.n_store_image_step == 0:
            torchvision.utils.save_image(torch.cat([data.cpu(), adv_data.cpu()], dim=0),
                                         os.path.join(args.log_folder, 'images_%d.jpg' % _iter),
                                         nrow=16)

        # save model checkpoint every n_checkpoint_step iterations
        if _iter % args.n_checkpoint_step == 0:
            file_name = os.path.join(args.model_folder, 'checkpoint_%d.pth' % _iter)
            save_model(model, file_name)

        _iter += 1
```

测试函数的原理相同，此处不再赘述：

```
def test(self, model, loader, adv_test=False):
    # initialize metrics
    num = 0
    total_acc = 0.0
    total_adv_acc = 0.0

    with torch.no_grad():
        for data, label in loader:
            # get classification model output
            output = model(data, _eval=True)

            # calculate model classification accuracy
            pred = torch.max(output, dim=1)[1]
            te_acc = evaluate(pred.cpu().numpy(), label.cpu().numpy(), 'sum')
            total_acc += te_acc
            num += output.shape[0]

            if adv_test:
                adv_data = self.attack.perturb(data, pred, 'mean', False)
                adv_output = model(adv_data, _eval=True)
                adv_pred = torch.max(adv_output, dim=1)[1]
                adv_acc = evaluate(adv_pred.cpu().numpy(), label.cpu().numpy(), 'sum')
                total_adv_acc += adv_acc
            else:
                total_adv_acc = -num

    return total_acc / num, total_adv_acc / num
```

2）定义主函数

首先从参数配置脚本中回复过程文件夹，并建立不存在的文件夹：

```
def main(args):
    # recovery parsers from args and make directions
    save_folder = args.dataset
    log_folder = os.path.join(args.log_root, save_folder)
    model_folder = os.path.join(args.model_root, save_folder)

    os.makedirs(log_folder, exist_ok=True)
    os.makedirs(model_folder, exist_ok=True)

    # add folders to args
    setattr(args, 'log_folder', log_folder)
    setattr(args, 'model_folder', model_folder)

    # save log to .txt
    logger = create_logger(log_folder, args.todo, 'info')
```

然后将深度网络模型、攻击函数、训练器进行实例化：

```
    # load base classification model
    model = Model(i_c=1, n_c=10)
    # load attack model
    attack = PGD_attack(model, args.epsilon, args.alpha, max_iters=args.k,
                        min_val=0, max_val=1, _type=args.perturbation_type)

    # initial model trainer
    trainer = Trainer(args, logger, attack)
```

从 torchvision 中加载 mnist 手写数字数据集，并采样部分图像构建训练 Dataloader，由于 mnist 数据集中图像数量较多，部分同学的电脑可能会出现训练过程缓慢的情况，代码示例中采样 400 张图像作为训练数据集，电脑计算条件较好的同学可以尝试适当加载更多的

图像进行训练：

```
if args.todo == 'train':
    # initialize training dataset(download if using MNIST for first time)
    tr_dataset = torchvision.datasets.MNIST(args.data_root, train=True, download=True,
                                            transform=torchvision.transforms.ToTensor())

    # MNIST数据集过大，此处采样400张图像作为训练集
    sample_index = [i for i in range(400)]
    image_train = []
    label_train = []
    for i in sample_index:
        image = tr_dataset[i][0]
        image_train.append(image)
        label = tr_dataset[i][1]
        label_train.append(label)
    sampled_train_data = [(image, label) for image, label in zip(image_train, label_train)]

    # initialize train dataloader by sampled dataset
    tr_loader = DataLoader(sampled_train_data, batch_size=args.batch_size, shuffle=True)
```

以同样的形式加载验证数据集，并构建验证 Dataloader，此处采样 100 张图像作为验证集：

```
    # initialize testing dataset(download if using MNIST for first time)
    te_dataset = torchvision.datasets.MNIST(args.data_root, train=False, download=True,
                                            transform=torchvision.transforms.ToTensor())

    # MNIST数据集过大，此处采样100张图像作为验证集
    sample_index = [i for i in range(100)]
    image_test = []
    label_test = []
    for i in sample_index:
        image = te_dataset[i][0]
        image_test.append(image)
        label = te_dataset[i][1]
        label_test.append(label)
    sampled_test_data = [(image, label) for image, label in zip(image_test, label_test)]

    # initialize valid dataloader by sampled dataset
    te_loader = DataLoader(sampled_test_data, batch_size=args.batch_size, shuffle=False, num_workers=4)
```

最后调用训练器的训练函数对深度网络进行训练：

```
    # start training
    trainer.train(model, tr_loader, te_loader, args.adv_train)
```

在实际执行中，首先实例化参数函数，然后调用主函数开始网络训练，如需启用对抗性训练，需要将超参数中的‘—adv_train’设置为 True。

（5）实验结果可视化

在完成各模型的训练后，为了直观对比各模型对 PGD 攻击的抵御程度，分别加载各项参数下的预训练模型，并将模型的分类结果在同一张图中进行可视化呈现，首先添加依赖包引用，加载数据并新建模型测试结果存放的文件夹：

```python
import os
import torch
import torchvision
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

from utils import load_model
from argument import parser
from attack import PGD_attack
from model import Model

if __name__ == '__main__':
    img_folder = './img'
    os.makedirs(img_folder, exist_ok=True)
    args = parser()
    te_dataset = torchvision.datasets.MNIST(args.data_root, train=False, download=True,
                                            transform=torchvision.transforms.ToTensor())
    # MNIST数据集过大，此处采样100张图像作为测试集
    sample_index = [i for i in range(100)]
    image_test = []
    label_test = []
    for i in sample_index:
        image = te_dataset[i][0]
        image_test.append(image)
        label = te_dataset[i][1]
        label_test.append(label)
    sampled_test_data = [(image, label) for image, label in zip(image_test, label_test)]

    # initialize valid dataloader by sampled dataset
    te_loader = DataLoader(sampled_test_data, batch_size=args.batch_size, shuffle=False, num_workers=4)
```

然后定义可视化脚本的部分参数：

```python
    for data, label in te_loader:
        break

    # types = ['Original', 'Standard', r'$l_{\infty}$-trained']
    types = ['Original', 'Standard_k10', 'Standard_k40']
    model_checkpoints = ['./checkpoint/mnist_standard_k10/checkpoint_4900.pth',
                         './checkpoint/mnist_standard_k40/checkpoint_4900.pth']
    adv_list = []
    pred_list = []

    max_epsilon = 0.8
    perturbation_type = 'linf'
```

推理获得不同模型下的网络分类结果：

```
with torch.no_grad():
    for checkpoint in model_checkpoints:
        model = Model(i_c=1, n_c=10)
        load_model(model, checkpoint)
        attack = PGD_attack(model,
                            max_epsilon,
                            args.alpha,
                            min_val=0,
                            max_val=1,
                            max_iters=args.k,
                            _type=perturbation_type)
        adv_data = attack.perturb(data, label, 'mean', False)

        output = model(adv_data, _eval=True)
        pred = torch.max(output, dim=1)[1]
        adv_list.append(adv_data.cpu().detach().numpy().squeeze())  # (N, 28, 28)
        pred_list.append(pred.cpu().numpy())
```

最后对可视化图像进行保存：

```
data = data.cpu().numpy().squeeze()  # (N, 28, 28)
data *= 255.0
label = label.cpu().numpy()
adv_list.insert(0, data)
pred_list.insert(0, label)

out_num = args.batch_size
fig, _axs = plt.subplots(nrows=len(adv_list), ncols=out_num)
axs = _axs

for j, _type in enumerate(types):
    axs[j, 0].set_ylabel(_type)
    for i in range(out_num):
        axs[j, i].set_xlabel('%d' % pred_list[j][i])
        axs[j, i].imshow(adv_list[j][i], cmap='gray')
        axs[j, i].get_xaxis().set_ticks([])
        axs[j, i].get_yaxis().set_ticks([])

plt.tight_layout()
plt.savefig(os.path.join(img_folder, 'mnist_large_%s.jpg' % (perturbation_type)))
```

五、验收要求

1. 分别在启用对抗性训练和不启用对抗性训练的条件下记录 LeNet 模型的分类性能，并分析说明对抗性训练的作用；

2. 在训练完成的对抗性训练网络中，使用与训练时不同强度的 PGD 攻击进行测试，观察对抗性训练得到的分类网络对不同强度攻击的鲁棒性，并分析原因；

3. 使用其他类型的攻击方法构建对抗性训练数据集，观察对抗性训练方法的泛化性，并尝试在训练和测试过程中使用不同类型的攻击方法，分析对抗性训练方法的缺点（选做）。