《人工智能安全》实验报告

Lab 2：PGD 对抗样本生成

姓名： 周炜

学号： 32010103790

专业： 计算机科学与技术

邮箱： 3210103790@zju.edu.cn

实验总览 复现了最经典的**PGD(Project Gradient Descent)** 对抗样本生成算法，运行此代码并探究了它的实现效果

实验原理

> 参考论文：
>
> 1. Towards Deep Learning Models Resistant to Adversarial Attacks
> 2. Towards Evaluating the Robustness of Neural Networks

快速梯度符号法FGSM

FastGradient-SignMethod 是梯度上升的单步 Fast 优化方法 优化方向沿着训练模型梯度下降的反方向

深度网络容易收到对抗性扰动的主要原因是它们的线性性质，因此高维空间中的线性行为足以引起对抗样本

在输入图像中加上计算得到的梯度方向，这样修改后的图像经过分类网络时的损失值就比修改前的图像经过分类网络时的损失值要大。换句话说，模型预测对的概率变小了

对于一个线性分类器：

$$f(x) = w^T x + b$$

对抗扰动随着权重维度的增加而增加
• 如果 w有 n个维度 权重向量中每个元素的平均大小为 m 且 $|\eta|_\infty \leq \epsilon$则产生的偏移最大可达到 ϵnm
• 虽然ϵ值很小 但是当 w的维度足够大时 可以使激活值提高 ϵnm 就有可能使分类器分类错误

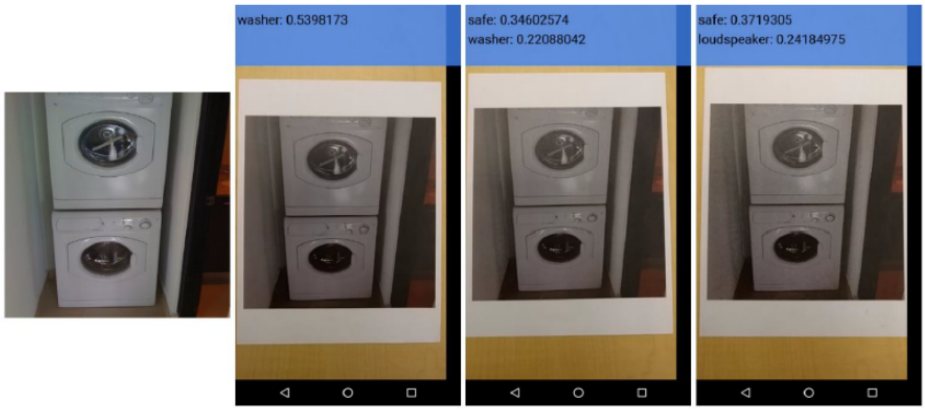
对抗目标：将代价函数变化限制在ϵ内 使得正确标签与模型预测结果的损失最大化

$$\max_{\delta} l(f(x+\delta), y) \qquad subject \quad to: \quad ||\delta||_\infty \leq \varepsilon$$

现在，将损失函数在点(x, y)附近进行线性近似$l(f(x+\delta), y) \approx l(f(x), y) + \nabla_x l(f(x), y)\delta$

损失函数是关于δ的线性函数，且δ受到l_∞的约束，在x的每个分量上独立地添加最大化地扰动（梯度上升），就能得到最优解$\delta^* = \varepsilon sign\{\nabla_x l(f(x), y\}$

FGSM攻击举例



(a) Image from dataset (b) Clean image (c) Adv. image, $\epsilon = 4$ (d) Adv. image, $\epsilon = 8$

- 图为对于洗衣机的图片的分类
- 在添加扰动后，模型将洗衣机错误地分类为保险柜

投影梯度下降法PGD

由于神经网络是非线性的，用FGSM效果一般比较差; PGD算法是FGSM算法的优化版本, 也属于白盒攻击的范畴

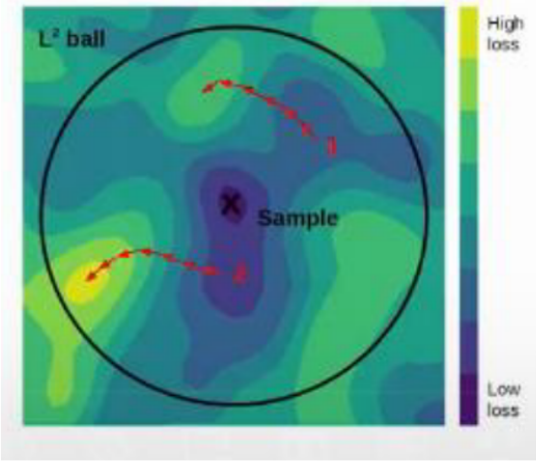
PGD采用小步多走的策略进行对抗。具体来说，就是一次次地进行前后向传播，一次次地根据grad计算扰动，最终最大化与原分类的差别（该差别由损失函数度量），如果是有目标攻击，则在最大化与原分类的差别同时，最小化与目标分类的差别，实现对抗样本生成的效果

$$\underset{(x,y)\sim\mathcal{D}}{} \left[\max_{\delta\in\mathcal{S}} L(\theta, x+\delta, y)\right]$$

(x, y)：样本空间的样本　　L：损失函数
D：样本空间的分布　　δ：扰动
θ：模型参数　　S：扰动的最大范围

$$x^{t+1} = \Pi_{x+\mathcal{S}}\left(x^t + \alpha\,\mathrm{sgn}(\nabla_x L(\theta, x, y))\right).$$

(x, y)：样本空间的样本　　L：损失函数，其中L内的x指的是x^t
$\prod$：投影操作　　α：每次移动的步长
θ：模型参数　　S：扰动的最大范围



在扰动范围的球体内，最佳对抗样本出现的点不一定在球面上，也可能出现在球体的内部。如上图中右侧的1 和 2 两条路径 PGD 算法可以沿着1 和 2 的路径找到使损失函数最大的对抗样本点。但是 FGSM 算法只能找到球面上的点，但显然这不是最好的对抗样本点

一般认为PGD 算法比 FGSM 算法更有效且更难防御

本实验中，我们实现的是 l_∞**范数** 度量距离的PGD算法生成对抗样本

代码分析 & 我的调整

整个工程可以分为三部分，第一部分为训练图像分类模型(**手写数字识别**），第二部分为生成对抗数据集，最后一部分为检测对抗效果

模型训练

首先使用PyTorch 训练两种不同的模型：一种为`Net`（一个自定义的卷积神经网络），另一种为`MobileNet`，基于 MobileNetV3 架构

自定义神经网络 Net

`Net`定义了一个自定义神经网络，其中包含两个卷积层（`conv1`和`conv2`）、丢弃层（`dropout1`和`dropout2`）以及完全连接层（`fc1`和`fc2`）。该`forward`方法定义了网络的前向传递，其中输入`x`通过层和激活以产生输出

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # Define the layers of the network
        self.conv1 = nn.Conv2d(3, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout(0.25)
        self.dropout2 = nn.Dropout(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)
    def forward(self, x):
        # Define the forward pass of the network
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output
```

基于MobileNetV3架构的神经网络 MobileNet

`MobileNet`使用预训练的 MobileNetV3 模型`torchvision.models`并删除最后两层（`nn.Sequential(*(list(net.children())[:-2]))`）来提取特征。然后，它应用平均池化（`self.avg_pool`），展平特征，并通过具有 log-softmax 激活的全连接层传递它们以获得最终输出

```
class MobileNet(nn.Module):
    def __init__(self):
        super(MobileNet, self).__init__()
        net = models.mobilenet_v3_small(pretrained=False)
        self.trunk = nn.Sequential(*(list(net.children())[:-2]))
        self.avg_pool = nn.AdaptiveAvgPool2d(output_size=1)
        self.fc = nn.Sequential(
            nn.Linear(576, 10, bias=True),
            nn.LogSoftmax(dim=1)
        )
    def forward(self, x):
```

```
        x = self.trunk(x)
        x = self.avg_pool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)
        return x
```

train and test

`train`函数执行给定模型的训练过程。它迭代训练数据批次，将数据和目标张量移动到指定设备，计算模型的输出，使用负对数似然损失()计算损失，执行反向传播，并使用优化器更新模型的参数`F.nll_loss`

`test`函数在测试数据集上评估训练模型。它将模型设置为评估模式(`model.eval()`)，迭代测试数据批次，计算模型的输出，使用负对数似然损失(`F.nll_loss`)计算测试损失，并计算正确预测样本的数量以计算准确性

```
def train(model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % 100 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))

def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item()  #
sum up batch loss
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the
max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= len(test_loader.dataset)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{}
({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

main

`main`定义了各种训练设置，比如，是否使用CUDA进行GPU加速（`use_cuda`）、随机种子（`seed`）、训练和测试的批量大小（`batch_size`和`test_batch_size`）、学习率（`lr`）、是否保存训练好的模型（`save_model`），以及要训练的时期数（`epochs`）

学习率调度器（`StepLR`）被设置为以指定的时间间隔调整学习率。

训练循环开始，其中模型被训练指定的时期数。在每个时期，`train`调用函数在训练数据集上训练模型，然后调用函数`test`在测试数据集上评估模型。学习率调度器也更新了。

训练循环完成后，如果`save_model`设置为`True`，则训练模型的参数将保存到文件中

`m1`即最后训练好的模型

训练过程可视化

我增加了一部分代码来体现训练过程，主要修改如下：

1. 导入python的绘图库`matplotlib.pyplot`，并且定义一个绘图的函数

```
import matplotlib.pyplot as plt
def plot_training_results(train_losses, test_losses, test_accuracy):
    epochs = len(train_losses)
    plt.style.use('seaborn')
    plt.figure(figsize=(12, 4))
    plt.subplot(1, 2, 1)
    plt.plot(range(1, epochs + 1), train_losses, label='Train Loss')
    plt.plot(range(1, epochs + 1), test_losses[:epochs], label='Test Loss')  #
Use test_losses[:epochs]
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.title('Training and Test Loss')
    plt.legend()
    plt.subplot(1, 2, 2)
    plt.plot(range(1, epochs + 1), test_accuracy, label='Test Accuracy')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.title('Test Accuracy')
    plt.tight_layout()
    plt.show()
```

2. 修改`train`函数和`test`函数，使得每次迭代时的参数都能被记录

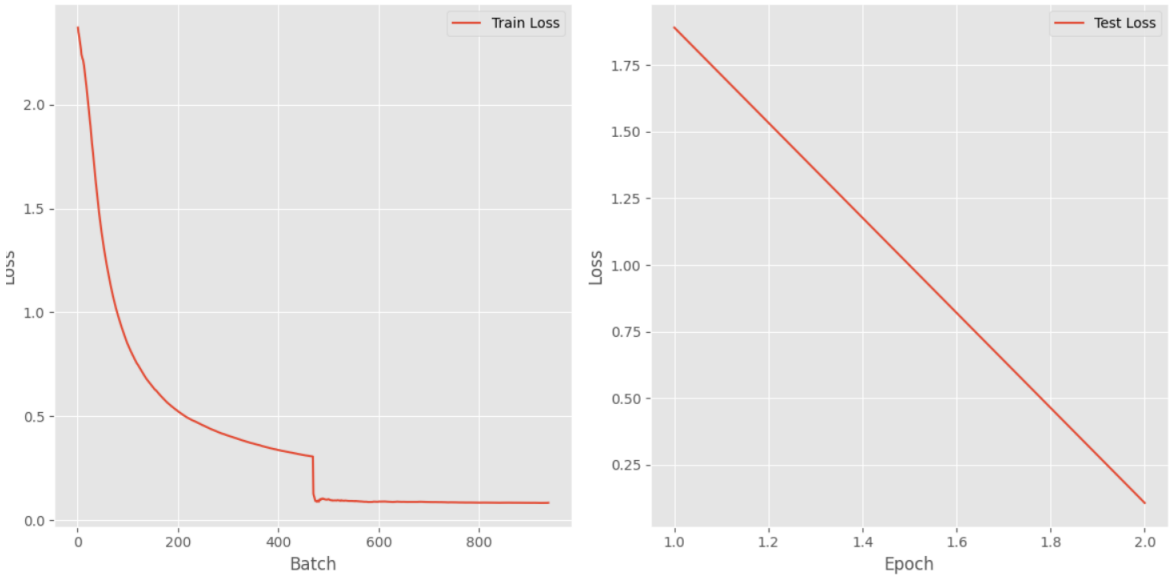
```
def train(model, device, train_loader, optimizer, epoch, train_losses):
    model.train()
    train_loss = 0
    batch_count = 0
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
```

```
        train_loss += loss.item()  # sum up batch loss
        batch_count += 1
        average_loss_so_far = train_loss / batch_count
        train_losses.append(average_loss_so_far)
    print(f'Train Epoch: {epoch}
[{len(train_loader.dataset)}/{len(train_loader.dataset)} (100%)]\tLoss:
{average_loss_so_far:.6f}')

def test(model, device, test_loader, test_losses, test_accuracy):
    model.eval()
    test_loss = 0
    correct = 0
    total_samples = len(test_loader.dataset)
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            loss = F.nll_loss(output, target, reduction='sum')
            test_loss += loss.item()  # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the
max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= total_samples  # calculate average loss over the dataset
    test_losses.append(test_loss)
    accuracy = correct / total_samples
    test_accuracy.append(accuracy)
    print(f'\nTest set: Average loss: {test_loss:.4f}, Accuracy:
{correct}/{total_samples} ({100. * accuracy:.2f}%)\n')
```

3. 在`main`函数中增加画图部分

```
def main():
    # ... keep not changed

    train_losses = []  # Store training losses
    test_losses = []  # Store test losses
    test_accuracy = []  # Store test accuracy
    scheduler = StepLR(optimizer, step_size=3, gamma=0.1)
    #  Modify the loop body function to record each training process
    for epoch in range(1, epochs + 1):
        train(model, device, train_loader, optimizer, epoch, train_losses)
        test(model, device, test_loader, test_losses, test_accuracy)
        scheduler.step()
    if save_model:
        torch.save(model.state_dict(), "mnist_mobile.pt")
    plot_training_results(train_losses, test_losses, test_accuracy, epochs)
 #plotting the results

    return model
```

经过2个`Epoch`后测试数据集上的准确率为: 9665/10000 (96.65%)



检验训练效果

打开一个图像（提供的代码为`4.jpg`，将其转换为张量，对其进行归一化，然后将其传递给预训练的CNN 模型（`Net`或`MobileNet`）以预测图像的类

```
from PIL import Image
from torchvision import transforms
import matplotlib.pyplot as plt

four_img = Image.open("mnist/pic/4.jpg")
four_img = four_img.convert('RGB')
...
cnn_eval(norm(four_tensor))
```

这里使用的模型并不是我们现在训练的模型，而是使用了提前训练好的CNN网络模型`mnist_cnn.pt`进行预测

```
def cnn_eval(tensor):
    model=Net()
    model.load_state_dict(torch.load("mnist_cnn.pt"))
    model.eval()
    print(torch.argmax(model(tensor)))
def mobile_eval():
    model=MobileNet()
    model.load_state_dict(torch.load("mnist_mobile.pt"))
    model.eval()
    print(torch.argmax(model(tensor)))
cnn_eval(norm(four_tensor))
```

输出为：`tensor(4)`

说明识别为数字4，并且人眼一看就是4，这说明识别成功

通过PGD算法生成对抗样本

对抗样本实例

使用l_∞范数对提供的模型执行PGD攻击, 通过在一定约束内优化 delta 张量并相应地扰动输入图像来生成对抗图像。然后返回并可选择显示对抗性图像, 具体来说，关键步骤有：

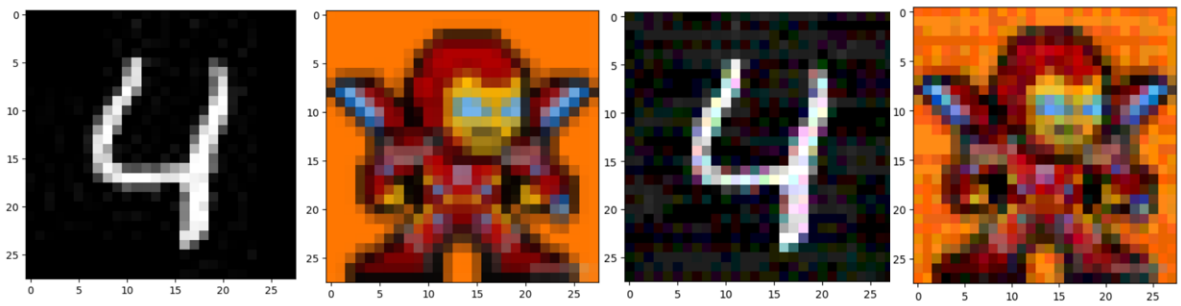
1. `l_infinity_pgd` 函数的输入接口：
    - `model`：要攻击的模型
    - `tensor`：表示图像的tensor类型
    - `gt`: 图像的真实类标签
    - `epsilon`：l_∞ 的扰动极限。默认值为 30/255
    - `target`：目标攻击的目标类标签。默认为无
    - `iter`：投影梯度下降 (PGD) 攻击的迭代次数。默认值为 50
    - `show`：指示是否显示扰动和对抗图像。默认为真
2. 运行`iter`迭代循环以执行投影梯度下降攻击
    - 该模型用于`tensor + delta`对归一化后的扰动图像 ( ) 进行预测
    - 如果`target`为 None，则损失计算为预测的 logits 和真实标签之间的**负交叉熵损失**
    - 如果`target`指定，则将附加项添加到损失中，这鼓励预测成为目标类标签
    - 每 10 次迭代打印一次损失
    - 优化器归零 (`opt.zero_grad()`)，损失反向传播 (`loss.backward()`)，然后执行优化器步骤 (`opt.step()`)
    - 增量张量被限制在 [-epsilon, epsilon] 范围内以强制执行扰动限制
3. 返回对抗图像 (`tensor + delta`)

```
import torch.optim as optim

def l_infinity_pgd(model, tensor, gt,epsilon=30./255, target=None,iter=50,
show=True):
    delta = torch.zeros_like(tensor, requires_grad=True)
    opt = optim.SGD([delta], lr=10)

    for t in range(iter):
        pred = model(norm(tensor + delta))
        if target is None:
            loss = -nn.CrossEntropyLoss()(pred, torch.LongTensor([gt]))
        else:
            loss = loss = -nn.CrossEntropyLoss()(pred, torch.LongTensor([gt])) +
nn.CrossEntropyLoss()(pred, torch.LongTensor([target]))
        if t % 10 == 0:
            print(t, loss.item())
        opt.zero_grad()
        loss.backward()
        opt.step()
        delta.data.clamp_(-epsilon, epsilon)
    print("True class probability:", nn.Softmax(dim=1)(pred))
    cnn_eval(norm(tensor+delta))
    if show:
        f,ax = plt.subplots(1,2, figsize=(10,5))
        ax[0].imshow((delta)[0].detach().numpy().transpose(1,2,0))
        ax[1].imshow((tensor + delta)[0].detach().numpy().transpose(1,2,0))
    return tensor + delta
```

```
x= l_infinity_pgd(m1,four_tensor,4)
```

（所给的代码中还有用另一张图片来生成对抗样例的，为了不再赘述，代码没有写在报告中）

处理后的结果为：



原图像　　PGD算法处理后的对抗样例

可以发现，通过制作这样的对抗性示例，可以欺骗模型做出错误的预测，而**人类观察者可能察觉不到这些变化**

上图中的数字4，再次进行检测，该样本被识别为tensor(8)，**识别错误**

大规模生成mnist对抗样本

对`mnist/training`目录中的图像执行有针对性的对抗性攻击来生成对抗性数据集。使用该函数生成对抗性示例`l_infinity_pgd`，生成的图像根据其原始和对抗性类标签保存在单独的目录中

```
# create dataset
import os
from torchvision.utils import save_image
def create_adv_dataset():
    transform=transforms.Compose([
        transforms.ToTensor()
        ])
    dataset1 = datasets.ImageFolder('mnist/training',transform=transform)
    train_loader = torch.utils.data.DataLoader(dataset1,batch_size=1,
shuffle=True, num_workers=8)
    attack_target = 0
    for batch_idx, (data, target) in enumerate(train_loader):
        attack_target = batch_idx//100
        if target== attack_target:
            continue
        if attack_target>9:
            break
        model=Net()
        model.load_state_dict(torch.load("mnist_cnn.pt"))
        model.eval()
        adv_img =
l_infinity_pgd(model,data,target,35./255,attack_target,50,False)
        image_dir_1 = os.path.join('mnist/adv_ori_label',str(target.item()))
        image_dir_2 = os.path.join('mnist/adv_adv_label',str(attack_target))
        if not os.path.exists(image_dir_1):
            os.makedirs(image_dir_1)
        if not os.path.exists(image_dir_2):
            os.makedirs(image_dir_2)
        save_image(adv_img, os.path.join(image_dir_1,str(batch_idx)+'.jpg'))
```

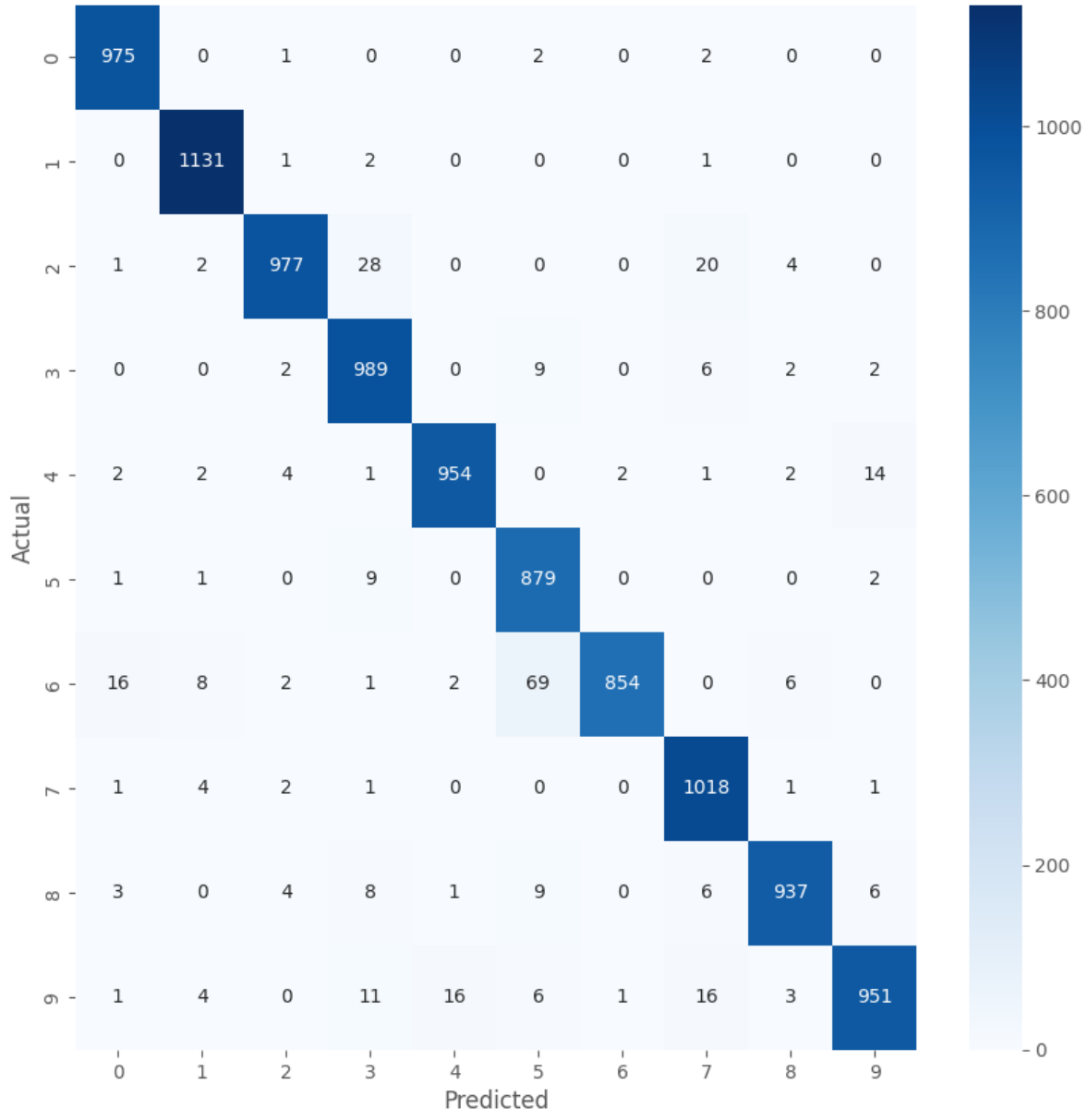
```
        save_image(adv_img, os.path.join(image_dir_2,str(batch_idx)+'.jpg'))
create_adv_dataset()
```

结果测试

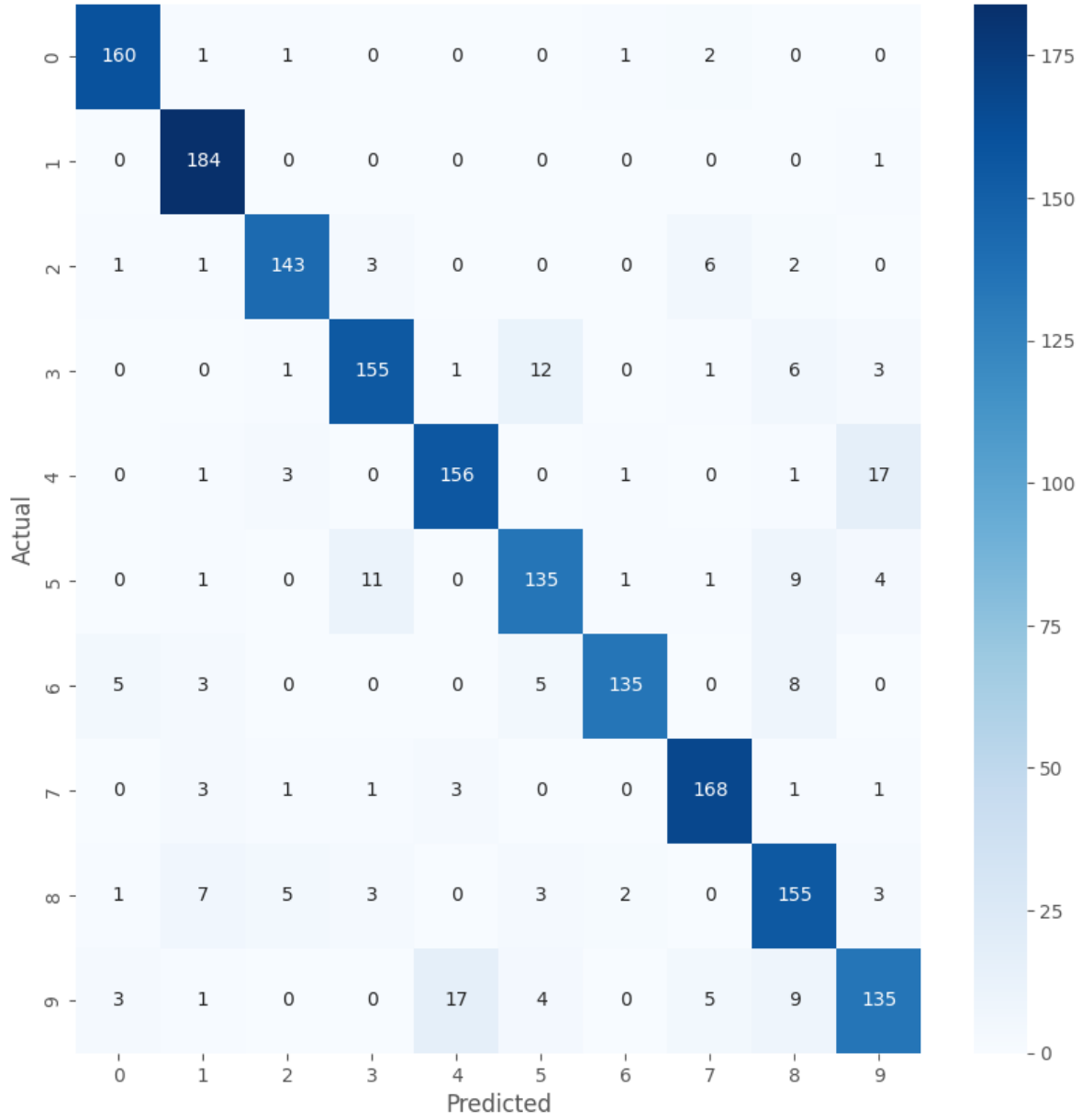
应用经过训练的`Net`模型`MobileNet`来评估它们在测试数据集和对抗数据集上的性能

绘制`MobileNet`的预测矩阵（`Net`类似，但是我没有绘制）：

`MobileNet`模型对原始测试数据的预测（`test_loader1`）



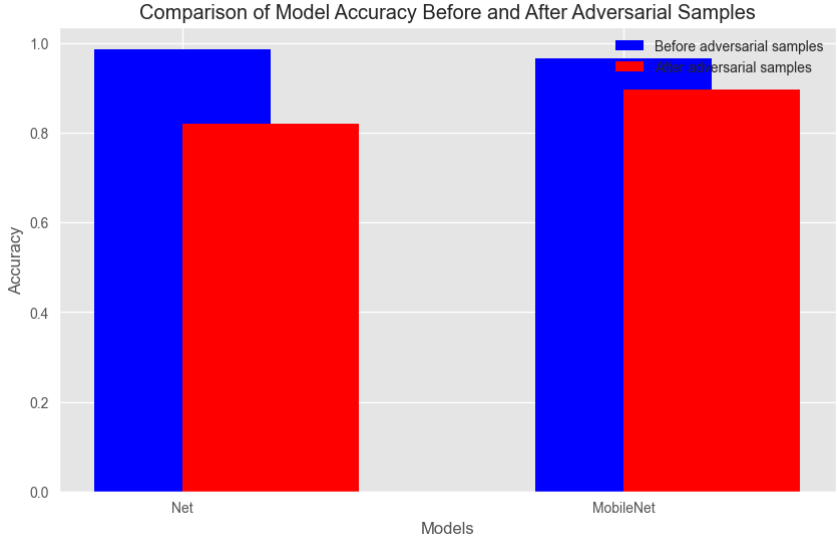
`MobileNet`模型对具有对抗性示例（`test_loader2`）的测试数据的预测



我们可以发现，对于对抗样本，`MobileNet`的识别率下降

另外，两个模型的识别率均有所下降，证明PGD算法生成的样本有一定的泛化能力

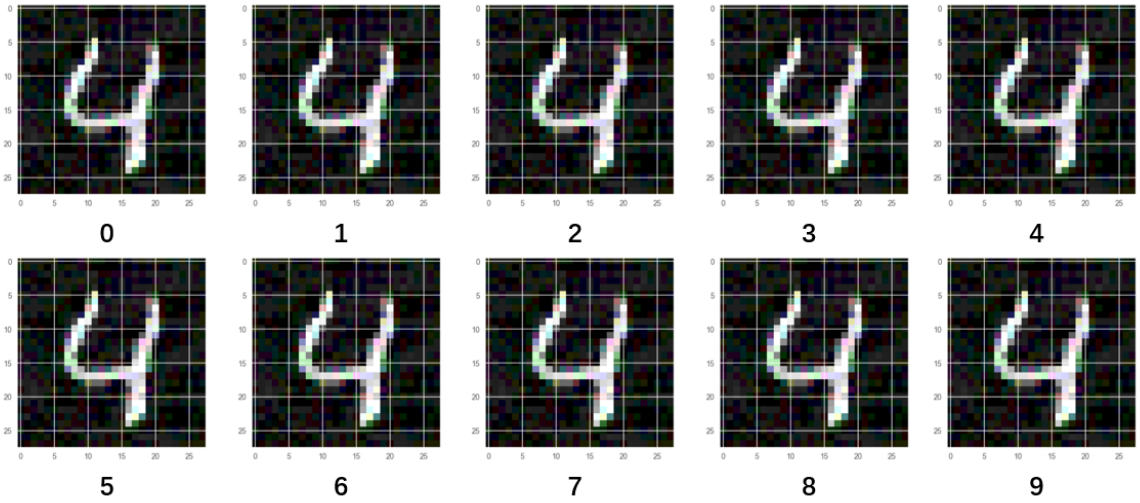
由于我们是针对mnist_cnn模型进行攻击的，故该模型识别成功率下降更大



再用如下代码，进行测试

```
for t in range(0,10):
    x = l_infinity_pgd(ma,four_tensor,4,target=t,iter=50,show=False)
    mobile_eval(norm(x))
```

从输出可以发现，人眼看出为4的图片，依次被模型判为了0到9，说明对抗样本攻击效果显著



踩坑记录&思考

代码错误1

在攻击部分，`+ nn.CrossEntropyLoss()(pred, torch.LongTensor([target]))`由于换行，因此实际上并没有加上，因此不管如何，loss函数始终固定

需要把`+ nn.CrossEntropyLoss()(pred, torch.LongTensor([target]))`和上一行合并

```
def l_infinity_pgd(model, tensor, gt,epsilon=30./255, target=None,iter=50,
show=True):
    delta = torch.zeros_like(tensor, requires_grad=True)
    opt = optim.SGD([delta], lr=10)

    for t in range(iter):
        pred = model(norm(tensor + delta))
        if target is None:
            loss = -nn.CrossEntropyLoss()(pred, torch.LongTensor([gt]))
        else:
            loss = loss = -nn.CrossEntropyLoss()(pred, torch.LongTensor([gt]))
            + nn.CrossEntropyLoss()(pred, torch.LongTensor([target]))
        if t % 10 == 0:
            print(t, loss.item())
```

思考1

这里只展示了l_∞的情况，但是实际上，使用其他范数也可以（其实，在数学上，许多范数的效果是等价的），都可以生成效果不错的对抗样本

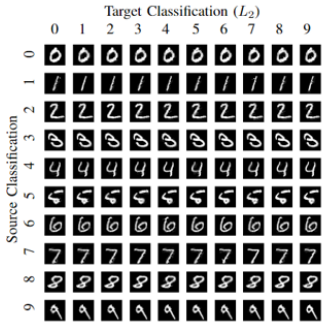


Fig. 3. Our L_2 adversary applied to the MNIST dataset performing a targeted attack for every source/target pair. Each digit is the first image in the dataset with that label.

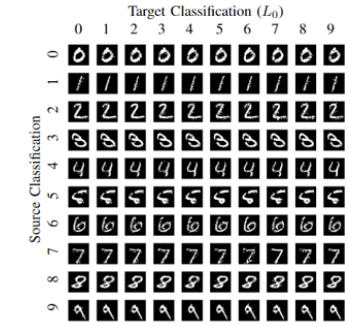


Fig. 4. Our L_0 adversary applied to the MNIST dataset performing a targeted attack for every source/target pair. Each digit is the first image in the dataset with that label.

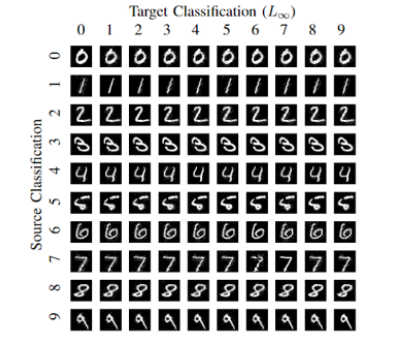


Fig. 5. Our L_∞ adversary applied to the MNIST dataset performing a targeted attack for every source/target pair. Each digit is the first image in the dataset with that label.

思考2

固定得带次数实际上是不必要的，当达到效果的时候就可以停止了

```
    for t in range(iter):
        pred = model(norm(tensor + delta))
        if target is None:
            loss = -nn.CrossEntropyLoss()(pred, torch.LongTensor([gt]))
        else:
            if torch.argmax(pred).item() == target:  #增加
                print("Attack succeed!")
                break
            loss = loss = -nn.CrossEntropyLoss()(pred, torch.LongTensor([gt])) +
nn.CrossEntropyLoss()(pred, torch.LongTensor([target]))
```