对抗样本攻击-实验报告

胡若凡 3200102312

1. 实验目的

- 理解对抗样本攻击的概念和原理，包括如何构造对抗样本、攻击目标模型等。
- 学习使用目前流行的对抗样本攻击算法，并掌握它们的优缺点。
- 探索对抗样本防御的方法PGD，了解常见的防御策略及其效果。
- 使用Python编程语言，掌握对抗样本攻击的实现过程，并通过实验对所学知识进行巩固和提高。
- 撰写实验报告，对实验过程和结果进行总结和分析，以便更好地理解和应用对抗样本攻击技术。

2. 数据集描述

本次使用的数据集为MNIST手写数字训练集，手写数字包括0-9十个数字，每个数字都有唯一的标签，可以用于训练和测试分类模型的性能。

MNIST数据集是一个经典的手写数字图片数据集，包含了60,000个训练样本和10,000个测试样本，每张图片的尺寸为28x28像素。它来自美国国家标准与技术研究所(NIST)的Special Database 3，Yann LeCun等人收集整理。近年来，它被广泛应用在计算机视觉领域的基础实验中，其简单和易用性使它成为机器学习的标准数据集之一。

3. 模型训练

3.1 模型定义

在实验中，首先我们需要了解这次cv实验的两个模型，分别是

Net类：使用了卷积神经网络(Convolutional Neural Network, CNN)算法。

在这个网络中，使用了两个卷积层，每个卷积层包含一个卷积核和一个ReLU激活函数。这些层网络自动从输入图像中提取特征，比如边缘、纹理、形状等。而在每个卷积层之间，执行最大池化操作来减少特征图的大小，并使用Dropout方法来防止过拟合。在最后一层全连接层上，使用log_softmax方法来将输出映射到10个不同的类别，以进行图像分类任务。

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 初始化卷积层1，输入通道数为3，输出通道数为32，卷积核大小为3x3，步长为1
        self.conv1 = nn.Conv2d(3, 32, 3, 1)
        # 初始化卷积层2，输入通道数为32，输出通道数为64，卷积核大小为3x3，步长为1
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        # 初始化dropout层1，随机舍弃25%的神经元
        self.dropout1 = nn.Dropout(0.25)
        # 初始化dropout层2，随机舍弃50%的神经元
        self.dropout2 = nn.Dropout(0.5)
        # 初始化全连接层1，输入尺寸为9216，输出尺寸为128
```

```
        self.fc1 = nn.Linear(9216, 128)
        # 初始化全连接层2，输入尺寸为128，输出尺寸为10
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        # 在输入图像上应用第一层卷积
        x = self.conv1(x)
        # 应用ReLU激活函数
        x = F.relu(x)
        # 在特征图上应用第二层卷积
        x = self.conv2(x)
        # 应用ReLU激活函数
        x = F.relu(x)
        # 对特征图进行最大池化，窗口大小为2x2
        x = F.max_pool2d(x, 2)
        # 应用第一个dropout层
        x = self.dropout1(x)
        # 对特征图展平成一维张量
        x = torch.flatten(x, 1)
        # 应用全连接层1
        x = self.fc1(x)
        # 应用ReLU激活函数
        x = F.relu(x)
        # 应用第二个dropout层
        x = self.dropout2(x)
        # 应用全连接层2
        x = self.fc2(x)
        # 应用log_softmax激活函数
        output = F.log_softmax(x, dim=1)
        return output
```

MobileNet类：使用了Depthwise Separable Convolution模型

MobileNet-V3采用了深度可分离卷积(Depthwise Separable Convolution)来减少模型参数数量和计算量。首先，对输入数据的每个通道单独应用空间卷积(kernel_size=3)，形成一个新的输出通道；然后，对所有输出通道应用1x1卷积核(kernel_size=1)进行线性组合。相较于传统卷积，该方法的计算量大大减少，同时还能提高模型的泛化能力。

在实现中，先引入了MobileNet-V3-Small预训练模型，并重新定义了最后一层全连接层以进行图像分类任务。具体地，我们使用了MobileNet-V3-Small的前n-2层作为"backbone"（除了最后两层fc层之外的所有层），并在其上添加一个自适应平均池化层和一个包含两个线性层和一个LogSoftmax层的序列模块，作为分类器。

```
class MobileNet(nn.Module):
    def __init__(self):
        super(MobileNet, self).__init__()
        net = models.mobilenet_v3_small(pretrained=False)
        #引入MobileNet-v3-small预训练模型并在其基础上进行微调。将pretrained参数设置为False表示不使用预训练权重,而是重新随机初始化权重
        self.trunk = nn.Sequential(*(list(net.children())[:-2]))
        #除了最后两层fc层之外的所有层,并以Sequential方式进行排列,即"backbone"
        self.avg_pool = nn.AdaptiveAvgPool2d(output_size=1)
        #定义一个自适应平均池化层,输出大小为1。该层的作用是将任意大小的输入张量转换为固定大小的输出张量。

        self.fc = nn.Sequential(
            nn.Linear(576, 10, bias=True),
```

```
            nn.LogSoftmax(dim=1)
        )
        #定义一个包含两个线性层和一个LogSoftmax层的序列模块。这是MobileNet模型的分类器

    def forward(self, x):
        #得到一个10维的概率向量。返回值为x
        x = self.trunk(x)
        x = self.avg_pool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)
        return x
```

3.2 训练流程

可以把训练流程定义为如下：

- 定义一些超参数，并打开 CUDA 计算。
- 设置训练数据集和测试数据集，并使用 `DataLoader` 对象进行封装。
- 创建一个模型，并将其移动到 GPU 上（如果可用）。
- 定义优化器和学习率调整方式，并进入训练循环。
  - 进入每轮训练，模型在训练数据上进行前向传播和反向传播，并更新模型参数。
  - 进行一次测试，计算模型在测试数据集上的准确率。
  - 更新学习率调整方式（例如：递减学习率）。
- 保存训练好的模型参数到文件中。

  并且，实验把模型存到了 **mnist_mobile.pt** 与 **mnist_cnn.pt** 之中

```
def main():
    # Training settings
    no_cuda = False
    seed = 1111
    batch_size = 128
    test_batch_size = 1000
    lr = 0.01
    save_model = True
    epochs = 2

    use_cuda = not no_cuda and torch.cuda.is_available()

    torch.manual_seed(seed)

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

    train_kwargs = {'batch_size': batch_size,'shuffle':True}
    test_kwargs = {'batch_size': test_batch_size,'shuffle':True}
    if use_cuda:
        cuda_kwargs = {'num_workers': 1,
                       'pin_memory': True,
                       'shuffle': False}
        train_kwargs.update(cuda_kwargs)
        test_kwargs.update(cuda_kwargs)

    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
```

```
        ])
    dataset1 = datasets.ImageFolder('mnist/training',transform=transform)
    dataset2 = datasets.ImageFolder('mnist/testing',transform=transform)
    train_loader = torch.utils.data.DataLoader(dataset1,**train_kwargs,
num_workers=8)
    test_loader = torch.utils.data.DataLoader(dataset2, **test_kwargs)

    model = MobileNet().to(device)

    optimizer = optim.SGD(model.parameters(), momentum=0.9, lr=lr)

    scheduler = StepLR(optimizer, step_size=3, gamma=0.1)
    for epoch in range(1, epochs + 1):
        train( model, device, train_loader, optimizer, epoch)
        test(model, device, test_loader)
        scheduler.step()

    if save_model:
        torch.save(model.state_dict(), "mnist_mobile.pt")


    return model
```

4. 对抗样本攻击

4.1 算法设计

在本次实验中，我们使用的对抗样本攻击为PGD，下面结合代码来进行源代码的分析：

代码的步骤为：

- 定义添加的扰动噪声delta，该delta即为最后输出时加在原图片上的扰动
- 开始迭代，迭代过程中，由于为 **白盒攻击** ，因此先用model的pred得到加了扰动后的图片与正确标签的CrossEntropyLoss，再根据loss进行调整。

  这里重要的是：由于python的loss函数自动是调整为向小的趋势，所以，我们希望与原分类差距越大。取个负号后，也就成了，**负数越大越小**，符合要求；而有目标攻击时，加上与target的loss，为正常的loss取法

```
def l_infinity_pgd(model, tensor, gt, epsilon=30./255, target=None, iter=100,
show=True):
    '''
    model:白盒攻击的模型
    tensor:输入的图片,攻击的对象
    gt:输入图片tensor的正确分类
    epsilon:噪声扰动的范围,限制图片改变不至于过大
    target:确定是否为有目标or无目标攻击，不为None时为希望攻击的目标
    iter:迭代次数
    '''
    # 定义一个 delta 张量，与输入相同的形状，用于保存扰动
    delta = torch.zeros_like(tensor, requires_grad=True)
    # 定义一个 SGD 优化器，用于更新扰动
    opt = optim.SGD([delta], lr=10)

    for t in range(iter):
```

```
        # 对输入加上扰动，并对加扰动后的输入进行前向传播得到输出
        pred = model(norm(tensor + delta))
        # 如果没有指定目标类别，则采用原始标签；否则采用目标标签
        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]))
        # 每 100 次迭代打印损失函数值
        if t % 100 == 0:
            print(t, loss.item())

        # 对扰动进行反向传播，计算梯度，并更新扰动
        opt.zero_grad()
        loss.backward()
        opt.step()
        # 将扰动裁剪到 [-epsilon, epsilon] 范围内
        delta.data.clamp_(-epsilon, epsilon)

    # 输出最终的预测结果和正确类别的概率
    print("True class probability:", nn.Softmax(dim=1)(pred))
    cnn_eval(norm(tensor+delta))

    # 如果 show 参数为 True，则显示不带扰动和加扰动后的图片
    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))

    # 返回加上扰动后的图片
    return tensor + delta
```

4.2 对抗样本构建

在明白了是如何添加扰动之后，实验对MNIST数据集随机添加了扰动，并设置了有目标攻击，把所有攻击后的图像，存在了原正确标签的文件夹中，例如pic-4添加扰动后，假设目标是1，那么在adv_ori_label中，就还是放在4的文件夹里

```
# 定义函数 create_adv_dataset()，用于生成对抗样本数据集
def create_adv_dataset():
    # 定义图像变换 transform，将 PIL.Image 或 numpy.ndarray 格式的图像转换为张量
    transform = transforms.Compose([transforms.ToTensor()])

    # 加载 MNIST 数据集，其中 'mnist/training' 为数据集路径
    dataset1 = datasets.ImageFolder('mnist/training', transform=transform)

    # 定义 DataLoader，用于加载数据。其中 batch_size 指定批量大小，num_workers 指定线程数量
    train_loader = torch.utils.data.DataLoader(dataset1, batch_size=1, shuffle=True, num_workers=8)

    # 定义变量 attack_target，表示攻击目标，初始值为 0
    attack_target = 0

    # 遍历每一个 batch
```

```
    for batch_idx, (data, target) in enumerate(train_loader):
        # 确定当前攻击目标
        attack_target = batch_idx // 100

        # 如果目标类别与攻击目标相同，跳过此次循环
        if target == attack_target:
            continue

        # 如果攻击目标无法实现
        if attack_target > 9:
            break

        # 实例化一个 Net() 类型的模型对象
        model = Net()

        # 加载预训练的模型参数，即 "mnist_cnn.pt" 文件
        model.load_state_dict(torch.load("mnist_cnn.pt"))

        # 将模型置为评估模式
        model.eval()

        # 调用 l_infinity_pgd() 函数，生成对抗样本数据
        adv_img = l_infinity_pgd(model, data, target, 35./255, attack_target,
50, False)

        # 定义图片保存路径
        # 一个使用原标签，一个使用攻击标签
        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))

        # 如果路径不存在，则创建此路径
        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)

        # 将对抗样本保存为 jpg 格式的图片文件，分别存储到 'mnist/adv_ori_label' 和
'mnist/adv_adv_label' 目录下
        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 dataset
```

4.3 实验结果

最后是demo代码的实验结果

```
test_transform=transforms.Compose([
        #transforms.GaussianBlur(3, sigma=(0.1, 1.0)),
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
        ])

dataset1 = datasets.ImageFolder('mnist/testing',transform=test_transform)
```

```
dataset2 = datasets.ImageFolder('mnist/adv_ori_label',transform=test_transform)
test_loader1 = torch.utils.data.DataLoader(dataset1,
shuffle=False,batch_size=100)
test_loader2 = torch.utils.data.DataLoader(dataset2,
shuffle=False,batch_size=100)

model=Net()
model.load_state_dict(torch.load("mnist_cnn.pt"))
model.eval()

test(model, 'cpu', test_loader1)
test(model, 'cpu', test_loader2)

model=MobileNet()
model.load_state_dict(torch.load("mnist_mobile.pt"))
model.eval()

test(model, 'cpu', test_loader1)
test(model, 'cpu', test_loader2)
```

若使用CNN作为白盒攻击的模型，结果为：

攻击后，CNN模型正确率下降到76%，而MobileNet模型正确率也有所下降，但幅度小了很多

```
Test set: Average loss: 0.0369, Accuracy: 9871/10000 (99%)


Test set: Average loss: 0.6173, Accuracy: 683/897 (76%)


Test set: Average loss: 0.1027, Accuracy: 9680/10000 (97%)


Test set: Average loss: 0.3932, Accuracy: 796/897 (89%)
```

若使用MobileNet作为白盒攻击的模型，结果为：

攻击后，CNN模型正确率下降到81%，而MobileNet模型正确率下降到74%

```
Test set: Average loss: 0.0369, Accuracy: 9871/10000 (99%)


Test set: Average loss: 0.5320, Accuracy: 1983/2453 (81%)


Test set: Average loss: 0.0696, Accuracy: 9788/10000 (98%)


Test set: Average loss: 0.7512, Accuracy: 1826/2453 (74%)
```

5. 优化与改进

5.1 攻击优化

首先，我们希望能够**增大迭代率**，并且设置一个**更小的学习率**，且使用能够**自动调节的Adam**

并且，为了防止原模型有较好的防御，我们还可以对delta再**加入随机性**，即再次设置一个微小的扰动，避免初始化的时候是从0开始的

```
def l_infinity_pgd(model, tensor, gt,epsilon=30./255, target=None,iter=100,
show=True):
    delta = torch.zeros_like(tensor, requires_grad=True)
    noise = torch.Tensor(np.random.uniform(low=-0.005, high=0.005,
size=delta_new.shape))
    delta += noise
    opt = optim.Adam([delta], lr=0.1)

    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 % 100 == 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
```

我们仅用cnn的模型做一个示例，再做了上面简单的调整后，我们发现，正确率再度下降，证明调整是有效的：

```
Test set: Average loss: 0.0369, Accuracy: 9871/10000 (99%)


Test set: Average loss: 0.9034, Accuracy: 623/910 (68%)


Test set: Average loss: 0.0696, Accuracy: 9788/10000 (98%)


Test set: Average loss: 0.7213, Accuracy: 683/910 (75%)
```

5.2 防御优化

对抗训练，数据增强、权重裁剪和Dropout正则化等方法，都可以提高MobileNet模型的鲁棒性，这里我重新训练并给出了使用的代码。 **FGSM**是仅仅对样本攻击一次的方法，这里由于是为了提高鲁棒性，不可以使用过多次迭代的图，并且幅度也需要较小，这是为了保证模型还是能学到正确的特征

```
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()

        # 使用数据增强，如旋转、平移、缩放、翻转等
        data = data + torch.randn_like(data) * 0.1

        # 新增对抗样本，以增强模型的鲁棒性
        perturbed_data = fgsm_attack(model, data, target, epsilon=0.1)
        output = model(perturbed_data)

        # 权重裁剪
        parameters_to_prune = ((model.conv1, 'weight'), (model.conv2, 'weight'),
(model.fc1, 'weight'), (model.fc2, 'weight'))
        prune.global_unstructured(
            parameters_to_prune,
            pruning_method=prune.L1Unstructured,
            amount=0.1,
            )

        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()))

# 对抗攻击函数
def fgsm_attack(model, data, target, epsilon):
    data_raw = data.clone().detach().requires_grad_(True)
    output = model(data_raw)
    loss = F.nll_loss(output, target)
    model.zero_grad()
    loss.backward()
    perturbed_data = data_raw + epsilon * torch.sign(data_raw.grad)
    perturbed_data = torch.clamp(perturbed_data, 0, 1)
    return perturbed_data
```

6. 实验心得

当面对强大的对抗攻击时，我们需要采取一系列防御措施来保障模型的鲁棒性和安全性。在这次的实验中，我们研究的主要是PGD攻击。PGD是一种迭代式的攻击方式，其基本思想是在给定一个初始样本的情况下，使用多次迭代生成对抗样本。PGD攻击相比于其他攻击方式来说比较强大，可以有效地迫使模型产生误判，并且可以克服部分防御措施。

而为了防御，我们也可以使用对抗训练，数据增强、权重裁剪和Dropout正则化等方法，去提高模型的鲁棒性，但是其中的参数需要精细调整，并且运气也至关重要，这终究是牺牲了一部分正确率去以求争取较为稳定的模型结果。

希望在之后的学习中，可以进一步了解到AI-security的更多知识。