# 代码复现报告

网安　林镇俊　2022302181287
https://github.com/lkbj0717/WR

---

## Deepfool

### 基本思路

对于每个模型每张图片要让当前模型"更改"原来的决定，重新作出一个错误判断，所需要加入的最小扰动，对一个模型来说，可以算出在对应数据集下的"（加权）平均最小扰动"（加权量取决于图形的复杂性），模型的"平均最小扰动"越大，说明鲁棒性越强，对应原文公式 $\Delta(\boldsymbol{x};\hat{k}) := \min_{\boldsymbol{r}} \|\boldsymbol{r}\|_2 \text{ subject to } \hat{k}(\boldsymbol{x}+\boldsymbol{r}) \neq \hat{k}(\boldsymbol{x})$。

为了降低计算复杂度，采用一阶泰勒展开"线性化"。

### 代码分析

#### 函数参数

解释：image是待攻击的初始图像，net是预训练网络模型（ResNet18），num_class为分类结果种类数（实际上是限制考虑的类别数，减少计算量），overshoot是为了放大扰动，以便跨越分类界限，让模型更改分类结果，max_iter为最大迭代次数（deepfool需要多次迭代），可以避免模型陷入无限循环。

```
def deepfool(image, net, num_classes=10, overshoot=0.02, max_iter=50):
```

#### 获取初始分类标签

```
    f_image = net.forward(Variable(image[None, :, :, :],
requires_grad=True)).data.cpu().numpy().flatten()
    I = (np.array(f_image)).flatten().argsort()[::-1]
    I = I[0:num_classes]
    label = I[0]
```

通过网络模型的前向传播来获取输入图像的分类结果，对应原文公式
$\hat{k}(\boldsymbol{x}) = \arg\max_k f_k(\boldsymbol{x})$

用`argsort()`函数对`logit`值（分类置信度）进行排序，选择前`num_classes`个最大的值，并将最大值对应的类别作为图像的初始分类标签。

**初始化了扰动计算的相关变量**

```
    input_shape = image.cpu().numpy().shape
    pert_image = copy.deepcopy(image)
    w = np.zeros(input_shape)
    r_tot = np.zeros(input_shape)
```

`w` 是用于存储每次迭代计算出的分类梯度差，用于生成最小的扰动向量，而 `r_tot` 是累积的扰动向量，初始值为零，随着迭代不断更新，最终成为足以使图像分类结果发生变化的对抗性扰动。

**核心代码**

本部分通过计算输入图像在不同类别之间的梯度差，来确定最小的扰动。

```
    while k_i == label and loop_i < max_iter:

        pert = np.inf
        fs[0, I[0]].backward(retain_graph=True)
        grad_orig = x.grad.data.cpu().numpy().copy()

        for k in range(1, num_classes):
            if x.grad is not None:
                x.grad.zero_()

            fs[0, I[k]].backward(retain_graph=True)
            cur_grad = x.grad.data.cpu().numpy().copy()

            # set new w_k and new f_k
            w_k = cur_grad - grad_orig
            f_k = (fs[0, I[k]] - fs[0, I[0]]).data.cpu().numpy()

            pert_k = abs(f_k)/np.linalg.norm(w_k.flatten())

            # determine which w_k to use
            if pert_k < pert:
                pert = pert_k
                w = w_k
```

进入循环，直到当前预测标签 `k_i` 与真实标签 `label` 不一致或者达到最大迭代次数 `max_iter`。

`fs[0, I[0]].backward()` 是为了计算初始分类的梯度 `grad_orig`，对于每一个非初始分类的类别（`I[k]`），分别计算它们对应的梯度，并以 `cur_grad - grad_orig` 计算类别之间的梯度差 `w_k`，`f_k` 表示当前类别与类别 `k` 之间的logit差值，表示图像从当前分类变为类别 `k` 所需的“推力”，即改变的程度。

`pert_k`表示从当前类别切换到类别 `k` 所需的最小扰动量。

对应原文这两条公式：

$$
\begin{aligned}
\boldsymbol{r}_*\left(\boldsymbol{x}_0\right) &:= \arg\min \|\boldsymbol{r}\|_2 \\
&\quad \text{subject to } \operatorname{sign}(f\left(\boldsymbol{x}_0+\boldsymbol{r}\right)) \neq \operatorname{sign}(f\left(\boldsymbol{x}_0\right)) \\
&= -\frac{f\left(\boldsymbol{x}_0\right)}{\|\boldsymbol{w}\|_2^2}\boldsymbol{w}
\end{aligned}
$$

$$
\mathbf{r}_*(\mathbf{x_0}) = \frac{\left|f_{\hat{l}\,(\mathbf{x}_0)}(\mathbf{x}_0) - f_{\hat{k}(\mathbf{x}_0)}(\mathbf{x}_0)\right|}{\|\mathbf{w}_{\hat{l}\,(\mathbf{x}_0)} - \mathbf{w}_{\hat{k}(\mathbf{x}_0)}\|_2^2}(\mathbf{w}_{\hat{l}\,(\mathbf{x}_0)} - \mathbf{w}_{\hat{k}(\mathbf{x}_0)})
$$

`if` 是比较每个类别最小扰动，选择扰动最小那个类别对应的梯度作为方向。

**更新图像与分类标签**

```
        r_i =  (pert+1e-4) * w / np.linalg.norm(w)
        r_tot = np.float32(r_tot + r_i)

        if is_cuda:
            pert_image = image +
(1+overshoot)*torch.from_numpy(r_tot).cuda()
        else:
            pert_image = image + (1+overshoot)*torch.from_numpy(r_tot)

        x = Variable(pert_image, requires_grad=True)
        fs = net.forward(x)
        k_i = np.argmax(fs.data.cpu().numpy().flatten())
```

计算单次迭代的扰动`r_i`，并累加到总扰动`r_tot`中，然后将原始图像`image`加入被`overshoot`放大过的累积扰动，得到新的被扰动图像`pert_image`，对被扰动的图像再次前向传播，计算新的logits值，用`np.argmax()`获取新的分类标签`k_i`，倘若分类结果改变，算法停止。

以上两部分对应原文算法2，如下：

**Algorithm 2** DeepFool: multi-class case

**input:** Image $\boldsymbol{x}$, classifier $f$.
**output:** Perturbation $\hat{\boldsymbol{r}}$.

Initialize $\boldsymbol{x}_0 \leftarrow \boldsymbol{x}$, $i \leftarrow 0$.
**while** $\hat{k}(\boldsymbol{x}_i) = \hat{k}(\boldsymbol{x}_0)$ **do**
  **for** $k \neq \hat{k}(\boldsymbol{x}_0)$ **do**
    $\boldsymbol{w}'_k \leftarrow \nabla f_k(\boldsymbol{x}_i) - \nabla f_{\hat{k}(\boldsymbol{x}_0)}(\boldsymbol{x}_i)$
    $f'_k \leftarrow f_k(\boldsymbol{x}_i) - f_{\hat{k}(\boldsymbol{x}_0)}(\boldsymbol{x}_i)$
  **end for**
  $\hat{l} \leftarrow \arg\min_{k \neq \hat{k}(\boldsymbol{x}_0)} \frac{|f'_k|}{\|\boldsymbol{w}'_k\|_2}$
  $\boldsymbol{r}_i \leftarrow \frac{|f'_{\hat{l}}|}{\|\boldsymbol{w}'_{\hat{l}}\|_2^2} \boldsymbol{w}'_{\hat{l}}$
  $\boldsymbol{x}_{i+1} \leftarrow \boldsymbol{x}_i + \boldsymbol{r}_i$
  $i \leftarrow i + 1$
**end while**
**return** $\hat{\boldsymbol{r}} = \sum_i \boldsymbol{r}_i$

## test_Deepfool.py

原文使用的是`ImageNet`训练集（`pytorch`预训练的`ResNet18`），而要求用的是`cifar10`，所以这里使用了https://github.com/ZOMIN28/ResNet18_Cifar10_95.46 的预训练模型。

对其架构进行了一些调整以适应 CIFAR-10 数据集。

```
# set device
device = 'cuda' if torch.cuda.is_available() else 'cpu'
n_class = 10
batch_size = 100
model = ResNet18() # 得到预训练模型
model.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3,
stride=1, padding=1, bias=False)
model.fc = torch.nn.Linear(512, n_class) # 将最后的全连接层修改
# 载入权重
model.load_state_dict(torch.load('checkpoint/resnet18_cifar10.pt'))
model = model.to(device)
model.eval()

def de_normalize(tensor, mean, std):
    mean = torch.tensor(mean).reshape(1, 3, 1, 1)
    std = torch.tensor(std).reshape(1, 3, 1, 1)
    tensor = tensor * std + mean
    return tensor

# 反标准化并转换为图像
def tensor_to_pil(tensor):
    tensor = de_normalize(tensor, mean, std)
```

```
    tensor = torch.clamp(tensor, 0, 1).numpy()  # 限制值在 [0, 1] 之间
    return tensor.squeeze(0) # 去掉批量维度

im_orig = Image.open('check.jpg')

mean = [ 0.485, 0.456, 0.406 ]
std = [ 0.229, 0.224, 0.225 ]


# Remove the mean
im = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.485, 0.456, 0.406],std=[0.229,
0.224, 0.225]),
    ])(im_orig)

r, loop_i, label_orig, label_pert, pert_image = deepfool(im, model)

#labels = open(os.path.join('synset_words.txt'),
'r').read().split('\n')

labels = ['plane', 'car', 'bird', 'cat','deer', 'dog', 'frog',
'horse', 'ship', 'truck']

str_label_orig = labels[np.int32(label_orig)].split(',')[0]
str_label_pert = labels[np.int32(label_pert)].split(',')[0]

print("Original label = ", str_label_orig)
print("Perturbed label = ", str_label_pert)
```

## 攻击结果

初始正确分类的标签为船“ship”，错误分类成了飞机“plane”，攻击成功


## 复现难点

因为原文给出了`git`仓库源代码，基本上不需要做修改，直接就能够运行，对于`deepfool`代码，主要的难点其实是把原文的思想和代码对应结合起来，对应着看看懂了就没有太困难的地方。复现过程中遇到的最主要的问题是原文使用的训练集和我们要求的不一样，这可能需要我们自己重新训练模型，考虑到时间成本，此处直接使用了别人预训练好的模型。

此外，由于源代码并不会输出攻击成功的示例，所以需要自己额外写一段代码，从测试集中抽取一张图片，获取其原始标签，同时获得这张图表被扰动后的图像和对应标签，并打印输出，虽然没有类似的经验，但通过网络资料等方式，可以完成这段代码，得到攻击成功示例。

# WaNet

## 基本思路

因为人类不擅长识别细微的图像扭曲，所以本文提出了一种采用图片扭曲的方式来生成触发器的方法，该方法能同时瞒过机器和人类的“两双眼睛”。

## 代码分析

### train

**模型选择**

```
def get_model(opt):
    netC = None
    optimizerC = None
    schedulerC = None

    if opt.dataset == "cifar10" or opt.dataset == "gtsrb":
        netC =
PreActResNet18(num_classes=opt.num_classes).to(opt.device)
    if opt.dataset == "celeba":
        netC = ResNet18().to(opt.device)
    if opt.dataset == "mnist":
        netC = NetC_MNIST().to(opt.device)
```

模型选择在输入指令的时候选`cifar10`即可

**后门数据比例**

```
        num_bd = int(bs * rate_bd)
        num_cross = int(num_bd * opt.cross_ratio)
```

在训练过程中，不是每张图都被用来生成后门数据，而是**只有一部分数据被选作后门数据**，`num_bd`表示选出的这些后门样本的数量，由总样本数乘以后门比例`rate_bd`得到。

**生成后门网格变换参数**

```
        grid_temps = (identity_grid + opt.s * noise_grid /
opt.input_height) * opt.grid_rescale
        grid_temps = torch.clamp(grid_temps, -1, 1)
```

`identity_grid`表示一个初始的单位网格，用于表示图像的原始像素位置，不做任何变换，而`noise_grid`：是一个随机扰动网格，用于给图像添加随机的形变。`opt.s`是扰动强度的缩放因子，控制扭曲的程度。

`torch.clamp(grid_temps, -1, 1)`是为了将网格变换的坐标值限制在[-1, 1]之间，保证网格变换后像素仍在图像的有效范围内。

最终生成的`grid_temps`就是这个扭曲后的网格。

此部分对应于论文公式中的注入函数，注入函数接收原图像`x`作为输入，通过扭曲形变，输出带有"触发器"的新图像，具体公式为 $\mathcal{B}(\boldsymbol{x}) = \mathcal{W}(\boldsymbol{x}, \boldsymbol{M})$ ，`M`作为形变场，定义了图像每个像素的移动方向和距离，相当于对图像的每个点进行了一个小的偏移。

```
        ins = torch.rand(num_cross, opt.input_height,
opt.input_height, 2).to(opt.device) * 2 - 1
        grid_temps2 = grid_temps.repeat(num_cross, 1, 1, 1) + ins /
opt.input_height
        grid_temps2 = torch.clamp(grid_temps2, -1, 1)

        inputs_bd = F.grid_sample(inputs[:num_bd],
grid_temps.repeat(num_bd, 1, 1, 1), align_corners=True)
```

为了让攻击更加复杂和隐蔽，除了固定的网格变换外，代码还生成了额外的随机扰动 `ins`，并将其加到原有的网格变换 `grid_temps` 上，生成新的 `grid_temps2`，专门用于**交叉数据**

最后用 `F.grid_sample()` 将生成的扭曲网格 `grid_temps` 应用到输入图像上。

生成的交叉数据集如下：

```
         inputs_cross = F.grid_sample(inputs[num_bd : (num_bd +
num_cross)], grid_temps2, align_corners=True)
```

**拼接**

```
total_inputs = transforms(total_inputs)
total_targets = torch.cat([targets_bd, targets[num_bd:]], dim=0)
```

**前向传播、损失计算与反向传播**

```
        start = time()
        total_preds = netC(total_inputs)
        total_time += time() - start
        loss_ce = criterion_CE(total_preds, total_targets)
        loss = loss_ce
        loss.backward()
        optimizerC.step()
```

输出 `total_preds` 是前向传播，而 `criterion_CE(total_preds, total_targets)` 是用**交叉熵损失函数**计算模型预测值与真实标签之间的损失，`loss.backward()` 执行反向传播。

**计算准确率**

```
        total_clean += bs - num_bd - num_cross
```

```
        total_bd += num_bd
        total_cross += num_cross
        total_clean_correct += torch.sum(
           torch.argmax(total_preds[(num_bd + num_cross) :], dim=1) ==
total_targets[(num_bd + num_cross) :]
        )
        total_bd_correct +=
torch.sum(torch.argmax(total_preds[:num_bd], dim=1) == targets_bd)
        if num_cross:
            total_cross_correct += torch.sum(
                torch.argmax(total_preds[num_bd : (num_bd +
num_cross)], dim=1)
                == total_targets[num_bd : (num_bd + num_cross)]
            )
            avg_acc_cross = total_cross_correct * 100.0 / total_cross

        avg_acc_clean = total_clean_correct * 100.0 / total_clean
        avg_acc_bd = total_bd_correct * 100.0 / total_bd
        avg_loss_ce = total_loss_ce / total_sample
```

对应文章所述**“在后门数据上，输出目标错误分类，但在干净数据上，模型应保持较高的分类准确率”**。

#### 可视化

```
        # Image for tensorboard
        if batch_idx == len(train_dl) - 2:
            residual = inputs_bd - inputs[:num_bd]
            batch_img = torch.cat([inputs[:num_bd], inputs_bd,
total_inputs[:num_bd], residual], dim=2)
            batch_img = denormalizer(batch_img)
            batch_img = F.upsample(batch_img, scale_factor=(4, 4))
            grid = torchvision.utils.make_grid(batch_img,
normalize=True)
```

将处理后的图像结果上传到`TensorBoard`以供进一步可视化分析。

## eval

#### 干净数据评估

```
            # Evaluate Clean
            preds_clean = netC(inputs)
            total_clean_correct += torch.sum(torch.argmax(preds_clean,
1) == targets)
```

`torch.sum(torch.argmax(preds_clean, 1) == targets)`是通过比较预测标签和真实标签，计算正确分类的样本数，并累加到`total_clean_correct`

对应原文在干净数据上保持高准确率

**评估后门数据**

```
            # Evaluate Backdoor
            grid_temps = (identity_grid + opt.s * noise_grid / opt.input_height) * opt.grid_rescale
            grid_temps = torch.clamp(grid_temps, -1, 1)

            ins = torch.rand(bs, opt.input_height, opt.input_height, 2).to(opt.device) * 2 - 1
            grid_temps2 = grid_temps.repeat(bs, 1, 1, 1) + ins / opt.input_height
            grid_temps2 = torch.clamp(grid_temps2, -1, 1)

            inputs_bd = F.grid_sample(inputs, grid_temps.repeat(bs, 1, 1, 1), align_corners=True)
            if opt.attack_mode == "all2one":
                targets_bd = torch.ones_like(targets) * opt.target_label
            if opt.attack_mode == "all2all":
                targets_bd = torch.remainder(targets + 1, opt.num_classes)
            preds_bd = netC(inputs_bd)
            total_bd_correct += torch.sum(torch.argmax(preds_bd, 1) == targets_bd)

            acc_clean = total_clean_correct * 100.0 / total_sample
            acc_bd = total_bd_correct * 100.0 / total_sample
```

`total_bd_correct`统计了后门数据被成功攻击的样本数

对应原文后门攻击要有效。

**评估交叉数据**

```
# Evaluate cross
if opt.cross_ratio:
    inputs_cross = F.grid_sample(inputs, grid_temps2, align_corners=True)
    preds_cross = netC(inputs_cross)
    total_cross_correct += torch.sum(torch.argmax(preds_cross, 1) == targets)
```

与论文中提到的"增强攻击隐蔽性"对应，通过随机扰动使后门数据更加难以被检测到。

**显示准确率与评估结果**

# main

```
# Prepare grid
ins = torch.rand(1, 2, opt.k, opt.k) * 2 - 1
ins = ins / torch.mean(torch.abs(ins))
noise_grid = (
    F.upsample(ins, size=opt.input_height, mode="bicubic",
align_corners=True)
    .permute(0, 2, 3, 1)
    .to(opt.device)
)
array1d = torch.linspace(-1, 1, steps=opt.input_height)
x, y = torch.meshgrid(array1d, array1d)
identity_grid = torch.stack((y, x), 2)[None, ...].to(opt.device)
```

对应原文：本文选择了覆盖整个图像的大小为 k×k 的均匀网格上的目标点，它们的反向扭曲场记作$P \in \mathbb{R}^{k\times k\times 2}$。参数s定义了P（扭曲场）的强度，具体公式如下
$\boldsymbol{P} = \psi(rand_{[-1,1]}(k,k,2)) \times s$ ，其中

$\mathrm{rand}_{[-1,1]}(k,k,2)$：是一个随机张量，生成的值在[-1,1]的范围内，大小为$k \times k\times$ 2,对应图像中每个控制点的随机偏移。
ψ：归一化函数，用以确保随机张量的元素通过其平均绝对值进行归一化。

size(A)：即张量 A 的大小，其对应的元素个数，
$\sum_{a_i\in A}|a_i|$:张量 $A$ 中所有元素的绝对值之和。

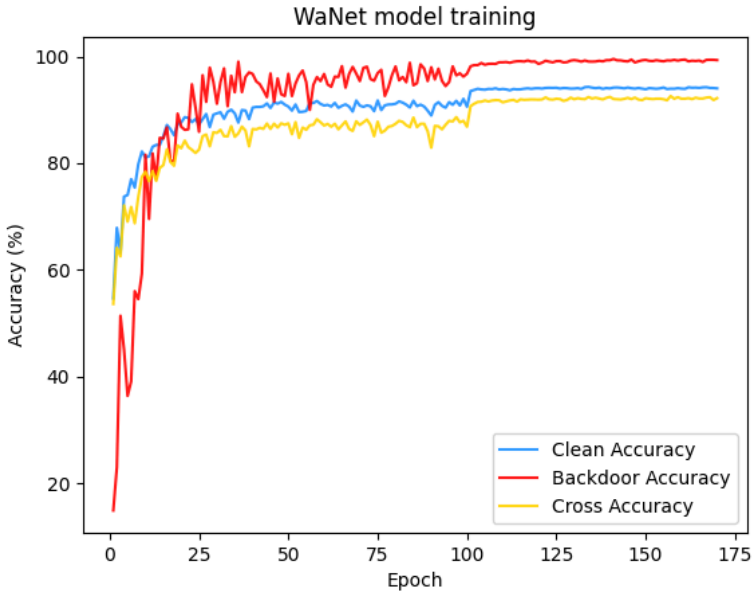
归一化函数的具体公式 $\psi(\boldsymbol{A}) = \frac{\boldsymbol{A}}{\frac{1}{size(\boldsymbol{A})}\sum_{a_i\in\boldsymbol{A}}|a_i|}$

**其余已在上述子函数分析**

# 攻击结果

**训练曲线**

一开始不会用`tensorboard`，于是写了一个`tmp.py`函数去画图如下：



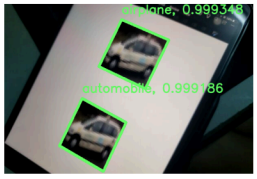

后来在bpp复现的时候学会了`tensorboard`，所以下一个复现部分会展示相应曲线，本次就使用自己写的画图函数。

原论文攻击结果的准确率如下，在93%~99%附近，我们的结果与原文结果大致相同。

| Dataset | Clean | Attack | Noise |
|---|---|---|---|
| MNIST | 99.52 | 99.86 | 98.20 |
| CIFAR-10 | 94.15 | 99.55 | 93.55 |
| GTSRB | 98.97 | 98.78 | 98.01 |
| CelebA | 78.99 | 99.33 | 76.74 |

(a) Network performance

(b) Sample backdoor images

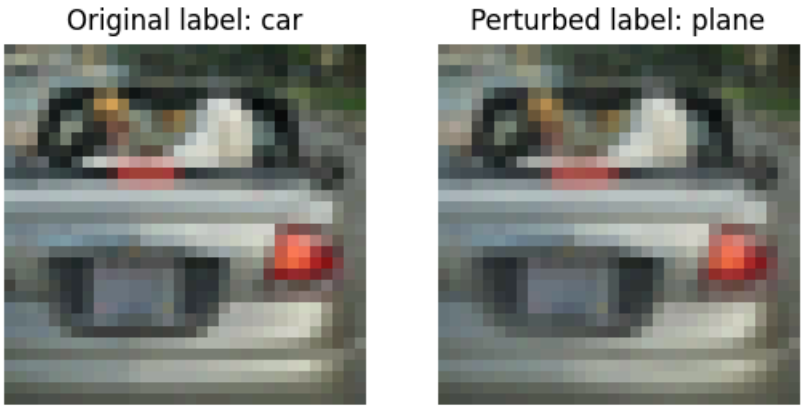


(c) Physical attack test

Figure 5: Attack experiments. In (b), we provide the clean (top) and backdoor (bottom) images.

**攻击成功**

初始准确标签为汽车car，攻击后模型分类标签为飞机plane




## 复现难点

本文的代码相对理论部分而言比较容易理解，每一步操作也很清晰，每段代码前面会有关键性的注释，这让我个人阅读代码相对轻松一些，反而是文章中比较晦涩难懂的数学公式对我造成了一定的困扰。

另外，在复现过程中遇到的一个问题是，由于我们要求“后门攻击要给出训练曲线图”，而源代码中的`tensorboard`我当时还不了解，所以并没有用好这个工具，反而自己去花时间修改了原来的代码，`dump`出了每一轮的训练成果，最后再将`dump`结果输入给画图函数得到训练曲线，这导致了我训练了2次这个模型，耗费了大量的时间，所以在`bpp`复现的时候我吸取经验教训，了解了`tensorboard`后，实验顺利很多。

# BppAttack

## 基本思路

根据文献，因为人类检查员对色彩深度的微小变化不敏感，本文提出了一种利用人类视觉系统的新攻击`BppAttack`。在该方法中，利用了图像量化和抖动，同时还提出了一种基于对比学习和对抗训练的方法来增强投毒过程，以注入触发器。

## 代码分析

### 模型选择与初始化

```
def get_model(opt):
    model = None
    optimizer = None
    scheduler = None

    if opt.dataset == "cifar10" or opt.dataset == "gtsrb":
        model =
PreActResNet18(num_classes=opt.num_classes).to(opt.device)
    if opt.dataset == "celeba":
        model = ResNet18().to(opt.device)
    if opt.dataset == "mnist":
        model = NetC_MNIST().to(opt.device)

    if opt.set_arch:
        if opt.set_arch == "densenet121":
            model = DenseNet121().to(opt.device)
        elif opt.set_arch == "mobilnetv2":
            model = MobileNetV2().to(opt.device)
        elif opt.set_arch == "resnext29":
            ResNeXt29_2x64d().to(opt.device)
        elif opt.set_arch == "senet18":
            SENet18().to(opt.device)

    # Optimizer
    optimizer = torch.optim.SGD(model.parameters(), opt.lr,
momentum=0.9, weight_decay=5e-4)

    # Scheduler
    scheduler = torch.optim.lr_scheduler.MultiStepLR(optimizer,
opt.scheduler_milestones, opt.scheduler_lambda)

    return model, optimizer, scheduler
```

## 图像量化

将标准化后的图像数据还原为原始图像数据，便于后续的处理和可视化

```
def back_to_np(inputs,opt):

    if opt.dataset == "cifar10":
        expected_values = [0.4914, 0.4822, 0.4465]
        variance = [0.247, 0.243, 0.261]
    elif opt.dataset == "mnist":
        expected_values = [0.5]
        variance = [0.5]
    elif opt.dataset in ["gtsrb","celeba"]:
        expected_values = [0,0,0]
        variance = [1,1,1]
    inputs_clone = inputs.clone()
```

```
    print(inputs_clone.shape)
    if opt.dataset == "mnist":
        inputs_clone[:,:,:] = inputs_clone[:,:,:] * variance[0] +
expected_values[0]
    else:
        for channel in range(3):
            inputs_clone[channel,:,:] = inputs_clone[channel,:,:] *
variance[channel] + expected_values[channel]
    return inputs_clone*255

def back_to_np_4d(inputs,opt):
    if opt.dataset == "cifar10":
        expected_values = [0.4914, 0.4822, 0.4465]
        variance = [0.247, 0.243, 0.261]
    elif opt.dataset == "mnist":
        expected_values = [0.5]
        variance = [0.5]
    elif opt.dataset in ["gtsrb","celeba"]:
        expected_values = [0,0,0]
        variance = [1,1,1]
    inputs_clone = inputs.clone()

    if opt.dataset == "mnist":
        inputs_clone[:,:,:,:] = inputs_clone[:,:,:,:] * variance[0] +
expected_values[0]
    else:
        for channel in range(3):
            inputs_clone[:,channel,:,:] = inputs_clone[:,channel,:,:]
* variance[channel] + expected_values[channel]

    return inputs_clone*255
```

用 `inputs.clone()` 创建输入张量的克隆，以避免对原始数据的修改，对于`cifar10`，遍历每个通道，分别进行逆归一化处理，最后函数返回经过逆归一化处理后的张量，乘以255，将值范围从 `[0, 1]` 转换到 `[0, 255]`，适合于图像显示。

对应原文：

通过减少每个颜色通道的位数，将图像的原始颜色调色板（每个像素的每个通道的m位）压缩到较小的颜色调色板（d位），压缩函数$T$定义如下：

$$T(\boldsymbol{x}) = \frac{round\left(\frac{x}{2^m-1}*(2^d-1)\right)}{2^d-1} * (2^m-1)$$

公式中，round函数用于四舍五入，$\frac{\mathrm{x}}{2^m-1}$表示颜色值归一化,乘以$\left(2^d-1\right)$后实现了颜色量化。

## 图像抖动

```
def floydDitherspeed(image,squeeze_num):
    channel, h, w = image.shape
    for y in range(h):
        for x in range(w):
            old = image[:,y, x]
            temp=np.empty_like(old).astype(np.float64)
            new = rnd1(old/255.0*(squeeze_num-1),0,temp)/(squeeze_num-
1)*255
            error = old - new
            image[:,y, x] = new
            if x + 1 < w:
                image[:,y, x + 1] += error * 0.4375
            if (y + 1 < h) and (x + 1 < w):
                image[:,y + 1, x + 1] += error * 0.0625
            if y + 1 < h:
                image[:,y + 1, x] += error * 0.3125
            if (x - 1 >= 0) and (y + 1 < h):
                image[:,y + 1, x - 1] += error * 0.1875
    return image
```

双重循环遍历每个像素。

用 Floyd-Steinberg 抖动算法的权重，对应原文：用Floyd–Steinberg dithering量化每个像素并将误差分布到周围像素，从而保持图像的整体视觉效果，同时引入难以察觉的扰动，具体算法如下。

循环遍历图像像素，对于每个像素，计算其量化误差 error，然后将此误差扩散到相邻像素，通过权重系数$[a_1, a_2, a_3, a_4]$来调整扩散的影响。

**Algorithm 1** Quantization with Floyd-Steinberg Dithering

**Input:** Image $I$, Diffusion Distribution $[a_1, a_2, a_3, a_4]$
**Output:** Quantized Image

**function** PROCESS($I$)
  **for** x from right to left **do**
    **for** y from top to bottom **do**
      $error = quantize(I[x][y]) - I[x][y]$
      $I[x][y] = I[x][y] + error$
      $I[x+1][y] = I[x][y] + error * a_1$
      $I[x+1][y+1] = I[x][y] + error * a_2$
      $I[x][y+1] = I[x][y] + error * a_3$
      $I[x-1][y+1] = I[x][y] + error * a_4$

## 后门攻击

### 创建后门数据

```
        num_bd = int(bs * rate_bd)
        num_neg = int(bs * opt.neg_rate)
```

**后门数据处理**

```
# Create backdoor data
num_bd = int(bs * rate_bd)
num_neg = int(bs * opt.neg_rate)

if num_bd!=0 and num_neg!=0:
    inputs_bd = back_to_np_4d(inputs[:num_bd],opt)
    if opt.dithering:
        for i in range(inputs_bd.shape[0]):
            inputs_bd[i,:,:,:] =
torch.round(torch.from_numpy(floydDitherspeed(inputs_bd[i].detach().cp
u().numpy(),float(opt.squeeze_num))).cuda())
    else:
        inputs_bd = torch.round(inputs_bd/255.0*(squeeze_num-
1))/(squeeze_num-1)*255

    inputs_bd = np_4d_to_tensor(inputs_bd,opt)

    if opt.attack_mode == "all2one":
        targets_bd = torch.ones_like(targets[:num_bd]) *
opt.target_label
    if opt.attack_mode == "all2all":
        targets_bd = torch.remainder(targets[:num_bd] + 1,
opt.num_classes)

    inputs_negative = back_to_np_4d(inputs[num_bd : (num_bd +
num_neg)],opt) +
torch.cat(random.sample(residual_list_train,num_neg),dim=0)
    inputs_negative=torch.clamp(inputs_negative,0,255)
    inputs_negative = np_4d_to_tensor(inputs_negative,opt)

    total_inputs = torch.cat([inputs_bd, inputs_negative,
inputs[(num_bd + num_neg) :]], dim=0)
    total_targets = torch.cat([targets_bd, targets[num_bd:]], dim=0)
```

调用 `back_to_np_4d` 函数对后门样本进行逆归一化处理，若启用抖动（`opt.dithering`），则用 Floyd-Steinberg 抖动算法，通过 `np_4d_to_tensor` 将处理后的数据转换回张量，然后根据攻击模式设置后门目标标签（`targets_bd`）。

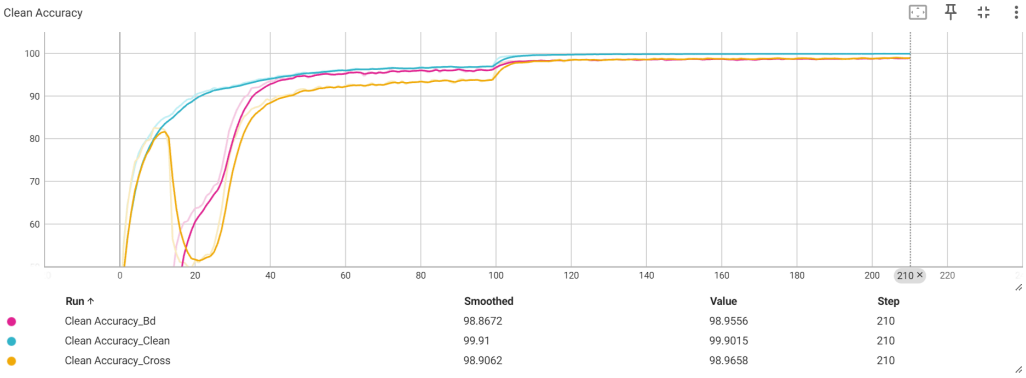
对于负样本，先进行逆归一化，而后与从 `residual_list_train` 中随机采样的样本进行拼接，用 `torch.clamp` 限制像素值在0到255之间，最后转换为张量。

对应原文内容主要是上面的**图像量化**和**图像抖动**部分，上面已提到，不赘述。

# 攻击结果

## 训练曲线

有了上次`WaNet`的经验，此次的`Bpp`攻击使用了`tensorboard`来可视化训练过程，具体如下：




由图可知，准确率最终都达到了99%左右，而论文的数据如下，在`cifar10`和`all2one`的情况下基本与论文数据一致。

| Dataset | Non-attack | WaNet | | BppAttack | |
|---|---|---|---|---|---|
| | BA | BA | ASR | BA | ASR |
| MNIST | 99.67% | 99.52% | 99.86% | 99.36% | 99.79% |
| CIFAR-10 | 94.88% | 94.15% | 99.55% | 94.54% | 99.91% |
| GTSRB | 99.31% | 98.97% | 98.78% | 99.25% | 99.96% |
| CelebA | 79.14% | 78.99% | 99.33% | 79.06% | 99.99% |

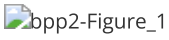
Table 2. Effectiveness on all-to-one attacks.

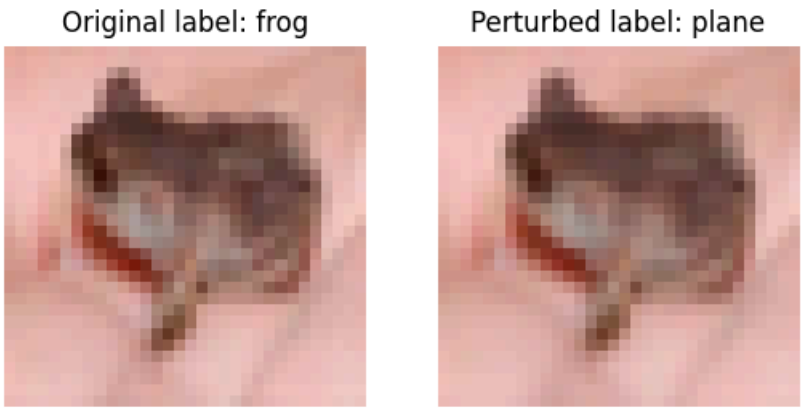
| Dataset | Non-attack | WaNet | | BppAttack | |
|---|---|---|---|---|---|
| | BA | BA | ASR | BA | ASR |
| MNIST | 99.67% | 99.44% | 95.90% | 99.25% | 98.46% |
| CIFAR-10 | 94.88% | 94.43% | 93.36% | 94.73% | 94.32% |
| GTSRB | 99.52% | 99.39% | 98.31% | 99.46% | 99.29% |
| CelebA | 79.14% | 78.73% | 78.58% | 78.84% | 78.72% |

Table 3. Effectiveness on all-to-all attacks.

## 攻击成功

示例如下：训练结果为，对所有后门图像，模型均识别为plane。






## 复现难点

因为有了做`WaNet`的经验，所以训练曲线的提取本身不再困难，本文也是基于`WaNet`修改而来的，所以代码理解上相对容易，此外，相比于前两篇，本文的代码量更少，理解起来并不困难，所以本篇的复现比较顺利。