BppAttack 复现

目标模型：PreActResNet18
数据集:CIFAR10

代码理解

BppAttack 代码的实现是在 WaNet 代码的基础上进行修改的，部分代码逻辑核心与其保持一致，所以接下来重点解释不一致的地方即核心代码部分，即 bppattack.py。

（1）初始化模型 get_model()

```
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)
```

根据传来的参数 opt（这是 config.py 文件里的，包含一大堆相关的变量）选择数据集和网络模型架构。

```
# 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)
```

初始化优化器。使用随机梯度下降算法。（SGD）
初始化学习率调度器。给予优化器 optimizer 的数值来调整学习率，milestones 表示里程点，达到该点后学习率乘以 lambda 数值。

（2）反标准化 back_to_np()函数

```
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]
```

根据数据集选择期望和方差，用于反标准化的计算。

```
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
```

将标准化后的图像转化回原始图像的像素范围值。

先将输入的图片数据进行克隆，防止改变了原始图像数据。

后面即是进行反标准化操作：因为 mnist 数据集里都是单通道灰度图像（就是黑白的），所以只有一个 channel；而其他数据集都是彩色图像（RGB），所以有三个 channel。根据对应的期望和方差算出来原始数据。返回值要乘以 255，以回到 0-255 的范围。

（3）反标准化 back_to_np_4d()函数

这个函数以上述（2）函数基本上一模一样，只有下述差别。

```
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]
```

因为这个用来处理 4d 图像的，即（batch_size, channels, height, width），比（2）的 3d 图像多一个 batch_size。其他一样。

（4）标准化 np_4d_to_tensor()函数

这个函数也与上面两个差不多，只是过程是反着的。如下：

```
if opt.dataset == "mnist":
    inputs_clone[:,:,:,:] = (inputs_clone[:,:,:,:] - expected_values[0]).div(variance[0])
else:
    for channel in range(3):
        inputs_clone[:,channel,:,:] = (inputs_clone[:,channel,:,:] - expected_values[channel]).div(variance[channel])
return inputs_clone
```

期望和方差都是一样，只是计算过程是完全相反的。div（）函数就是除以括号里面的值，将原本 0-255 范围的数值标准化到 0-1 间。

（5）四舍五入处理 rnd1()函数

```
def rnd1(x, decimals, out):
    return np.round_(x, decimals, out)
```

将 x 进行四舍五入，decimals 是要保留的小数点位数，out 是结果保存数组。

（6）图像抖动 floydDitherspeed()函数

```
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
```

该算法的计算过程如上。本算法代表着图像抖动过程，是本论文提出的核心之一。其中包括量化操作（根据图片的长和宽），即 new 这一行，也是核心。“先量化、后抖动”，这是本论文提出的处理方法。

（7）对抗训练 train()函数

```
rate_bd = opt.injection_rate
total_loss_ce = 0
total_sample = 0

total_clean = 0
total_bd = 0
total_cross = 0
total_clean_correct = 0
total_bd_correct = 0
total_cross_correct = 0
criterion_CE = torch.nn.CrossEntropyLoss()
criterion_BCE = torch.nn.BCELoss()

denormalizer = Denormalizer(opt)
transforms = PostTensorTransform(opt).to(opt.device)
total_time = 0

avg_acc_cross = 0
```

一大堆在生成训练数据和训练中要用到的变量值，用于统计数据量等。

下面就是迭代批次学习，属于重点代码。

```
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().cpu().numpy(),float(opt.squeeze_num))
    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)
```

生成后门数据。在后门攻击样本和**负样本（扰动攻击样本，这里就是论文中提出的对比对抗训练的思想）**都不为 0 时，进行输入数据的处理，包括反标准化（本论文提出的图像量化）、是否进行图像抖动操作，最后对后门攻击样本又进行标准化，将原始数据交付后面的操作进行处理。

根据攻击方式确定后门攻击目标数，后生成负样本。先进行反标准化，在原始像素上进行 random.sample（）随机扰动，并用 clamp（）函数确保像素值在 0-255 间，最后再次标准化，交付后面处理。

在最后两行代码，就是将处理后的图像和标签合并，形成完整的训练数据。

```
elif (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().cpu().numpy(),float(opt.squeeze_num))
    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)

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

elif (num_bd==0 and num_neg==0):
    total_inputs = inputs
    total_targets = targets
```

这两段是根据后门攻击样本和负样本之间的数量（是否存在）进行完整训练数据的生成，处理思想和上面是一样的。

```
total_inputs = transforms(total_inputs)
start = time()
total_preds = model(total_inputs)
total_time += time() - start
loss_ce = criterion_CE(total_preds, total_targets)
loss = loss_ce
loss.backward()
optimizer.step()
total_sample += bs
total_loss_ce += loss_ce.detach()
total_clean += bs - num_bd - num_neg
total_bd += num_bd
total_cross += num_neg
total_clean_correct += torch.sum(
    torch.argmax(total_preds[(num_bd + num_neg) :], dim=1) == total_targets[(num_bd + num_neg) :]
)
```

定义训练时涉及到的 loss 值等变量，在后面会利用其进行计算。

```
if num_bd:
    total_bd_correct += torch.sum(torch.argmax(total_preds[:num_bd], dim=1) == targets_bd)
    avg_acc_bd = total_bd_correct * 100.0 / total_bd
else:
    avg_acc_bd = 0

if num_neg:
    total_cross_correct += torch.sum(
        torch.argmax(total_preds[num_bd : (num_bd + num_neg)], dim=1)
        == total_targets[num_bd : (num_bd + num_neg)]
    )
    avg_acc_cross = total_cross_correct * 100.0 / total_cross
else:
    avg_acc_cross = 0

avg_acc_clean = total_clean_correct * 100.0 / total_clean
avg_loss_ce = total_loss_ce / total_sample
```

定义后门攻击样本、负样本的成功率。有就通过公式算，没有就是 0。
接下来也定义了干净样本的识别准确率和 loss 值的计算公式。

```
if not batch_idx % 50:
    if not os.path.exists(opt.temps):
        os.makedirs(opt.temps)

    path = os.path.join(opt.temps, "backdoor_image.png")
    path_cross = os.path.join(opt.temps, "negative_image.png")
    if num_bd>0:
        torchvision.utils.save_image(inputs_bd, path, normalize=True)
    if num_neg>0:
        torchvision.utils.save_image(inputs_negative, path_cross, normalize=True)

    if (num_bd>0 and num_neg==0):
        print(
        batch_idx,
        len(train_dl),
        "CE Loss: {:.4f} | Clean Acc: {:.4f} | Bd Acc: {:.4f}".format(
            avg_loss_ce, avg_acc_clean, avg_acc_bd,
        ))
    elif (num_bd>0 and num_neg>0):
        print(
        batch_idx,
        len(train_dl),
        "CE Loss: {:.4f} | Clean Acc: {:.4f} | Bd Acc: {:.4f} | Cross Acc: {:.4f}".format(
            avg_loss_ce, avg_acc_clean, avg_acc_bd, avg_acc_cross
        ))
    else:
        print(
        batch_idx,
        len(train_dl),
        "CE Loss: {:.4f} | Clean Acc: {:.4f}".format(avg_loss_ce, avg_acc_clean))
```

每 50 个批次执行，定期保存训练中的样本图像，并输出模型的训练状态。在后门攻击样本和负样本数的各种情况下，都要保存，并输出。

```
# Image for tensorboard
if batch_idx == len(train_dl) - 2:
    if num_bd>0:
        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)

        print(torch.round(back_to_np(inputs_bd[0],opt)))
        print(back_to_np(inputs[0],opt))
        print(torch.round(back_to_np(inputs_bd[0],opt))-back_to_np(inputs[0],opt))
        print("done")

        path = os.path.join(opt.temps, "batch_img.png")
        torchvision.utils.save_image(batch_img, path, normalize=True)
```

保存一个特定批次的图像，并将其用于 TensorBoard 可视化或其他用途。具体来说，这段代码的功能是展示后门攻击过程中的图像变化，包括原始输入图像、后门图像、带有残差的图像，以及残差本身。

```
# for tensorboard
if not epoch % 1:
    tf_writer.add_scalars(
        "Clean Accuracy", {"Clean": avg_acc_clean, "Bd": avg_acc_bd, "Cross": avg_acc_cross}, epoch
    )
    if num_bd>0:
        tf_writer.add_image("Images", grid, global_step=epoch)

scheduler.step()
```

在训练过程中，将一些关键指标和图像信息记录到 TensorBoard 以便后续分析和可视化

（8）评估 eval（）函数

```
total_sample = 0
total_clean_correct = 0
total_bd_correct = 0
total_cross_correct = 0
total_ae_loss = 0

criterion_BCE = torch.nn.BCELoss()
```

定义各种评估时用的变量，如 loss 等。

下面就是迭代批次评估，属于重点代码。

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

inputs_bd = back_to_np_4d(inputs,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().cpu().numpy(),

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

if batch_idx ==0:
    print("backdoor target",targets_bd)
    print("clean target",targets)

preds_bd = model(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
```

估模型在干净样本和后门样本上的性能。具体来说，它计算了模型在这两种样本上的正确预测数量，并输出了这些指标。
该过程大致思想流程其实是和训练过程差不多的，两个过程算的值在本质上差别不大。

```
# Evaluate cross
if opt.neg_rate:

    inputs_negative = back_to_np_4d(inputs,opt) + torch.cat(random.sample(residual_list_test,inputs.shape[0]),dim=0)
    inputs_negative = np_4d_to_tensor(inputs_negative,opt)

    preds_cross = model(inputs_negative)
    total_cross_correct += torch.sum(torch.argmax(preds_cross, 1) == targets)

    acc_cross = total_cross_correct * 100.0 / total_sample

    info_string = (
        "Clean Acc: {:.4f} - Best: {:.4f} | Bd Acc: {:.4f} - Best: {:.4f} | Cross: {:.4f}".format(
            acc_clean, best_clean_acc, acc_bd, best_bd_acc, acc_cross, best_cross_acc
        )
    )
else:
    info_string = "Clean Acc: {:.4f} - Best: {:.4f} | Bd Acc: {:.4f} - Best: {:.4f}".format(
        acc_clean, best_clean_acc, acc_bd, best_bd_acc
    )
```

是用来评估模型在交叉扰动样本（即负样本）上的性能，即在原始输入上添加了额外的扰动后的预测准确率，再将干净样本、后门攻击样本、负样本的准确率输出。

```
# tensorboard
if not epoch % 1:
    tf_writer.add_scalars("Test Accuracy", {"Clean": acc_clean, "Bd": acc_bd}, epoch)

# Save checkpoint
if acc_clean > best_clean_acc or (acc_clean > best_clean_acc - 0.1 and acc_bd > best_bd_acc):
    print(" Saving...")
    best_clean_acc = acc_clean
    best_bd_acc = acc_bd
    if opt.neg_rate:
        best_cross_acc = acc_cross
    else:
        best_cross_acc = torch.tensor([0])
    state_dict = {
        "model": model.state_dict(),
        "scheduler": scheduler.state_dict(),
        "optimizer": optimizer.state_dict(),
        "best_clean_acc": best_clean_acc,
        "best_bd_acc": best_bd_acc,
        "best_neg_acc": best_cross_acc,
        "epoch_current": epoch,
    }
    torch.save(state_dict, opt.ckpt_path)
    with open(os.path.join(opt.ckpt_folder, "results.txt"), "w+") as f:
        results_dict = {
            "clean_acc": best_clean_acc.item(),
            "bd_acc": best_bd_acc.item(),
            "cross_acc": best_cross_acc.item(),
        }
        json.dump(results_dict, f, indent=2)

return best_clean_acc, best_bd_acc, best_cross_acc
```

这里是进行可视化和保存训练检查点（checkpoints）的代码，非重点。

（9）main()函数

前部分依然是选择数据集、加载数据集、继续训练、从头训练，这个之前已经说明了，就不再说了。

```
for j in range(n):
    for batch_idx, (inputs, targets) in enumerate(train_dl):
        print(batch_idx)
        temp_negetive = back_to_np_4d(inputs,opt)

        temp_negetive_modified = back_to_np_4d(inputs,opt)
        if opt.dithering:
            for i in range(temp_negetive_modified.shape[0]):
                temp_negetive_modified[i,:,:,:] = torch.round(torch.from_numpy(floydDitherspeed(temp_negetive_modified[i].detach().cpu
        else:
            temp_negetive_modified = torch.round(temp_negetive_modified/255.0*(opt.squeeze_num-1))/(opt.squeeze_num-1)*255

        residual = temp_negetive_modified - temp_negetive
        for i in range(residual.shape[0]):
            residual_list_train.append(residual[i].unsqueeze(0).cuda())
            count = count + 1
```

生成扰动并将其添加到 residual_list_train 列表中。这些扰动是在干净图像上应用一定的处理（例如，量化和 抖动）后得到的,用于生成训练数据。if 判断表示已经进行量化就进行抖动操作，否则先进行量化操作。该量化操作是对图片的反标准化后的张量值。

后面还有一个用于生成测试数据的代码，一样的，就不说明了。

最后就是输出评估过程的各种数据，也不进行说明了。

复现过程：

1.安装环境

```
# root @ d7d149c7484f in ~/jxd/BppAttack on git:main o [6:56:56]
$ pip install -r requirements.txt
```

注意：

```
absl-py==1.1.0
cachetools==5.2.0
certifi @ file:///opt/conda/conda-bld/certifi_1655968806487/work/certifi
charset-normalizer==2.0.12
google-auth==2.8.0
```

红框内的文件应该是作者的本地文件，在开源项目里面没有找到，所以我将其替换成最新版的 certifi。

2.训练、评估过程

运行下述命令

```
python -u bppattack.py --dataset cifar10 --attack_mode all2one --squeeze_num 32
```

训练过程共一千轮，在每一次的训练后，都会进行一次评估。

训练

```
Epoch 1000:
 Train:
cifar10
0 391 CE Loss: 0.0001 | Clean Acc: 100.0000 | Bd Acc: 96.0000 | Cross Acc: 100.0000
50 391 CE Loss: 0.0001 | Clean Acc: 99.9246 | Bd Acc: 99.1373 | Cross Acc: 98.9020
100 391 CE Loss: 0.0001 | Clean Acc: 99.9619 | Bd Acc: 98.9307 | Cross Acc: 98.8515
150 391 CE Loss: 0.0001 | Clean Acc: 99.9576 | Bd Acc: 98.8609 | Cross Acc: 98.8344
200 391 CE Loss: 0.0001 | Clean Acc: 99.9490 | Bd Acc: 98.7662 | Cross Acc: 98.8259
250 391 CE Loss: 0.0001 | Clean Acc: 99.9540 | Bd Acc: 98.7410 | Cross Acc: 98.9004
300 391 CE Loss: 0.0001 | Clean Acc: 99.9489 | Bd Acc: 98.7110 | Cross Acc: 98.9634
350 391 CE Loss: 0.0001 | Clean Acc: 99.9452 | Bd Acc: 98.7464 | Cross Acc: 98.9858
torch.Size([3, 32, 32])
```

训练过程会输出 loss 以及干净样本、后门攻击样本、负样本准确率。此外，还有在训练过程中的一系列 size 和张量。

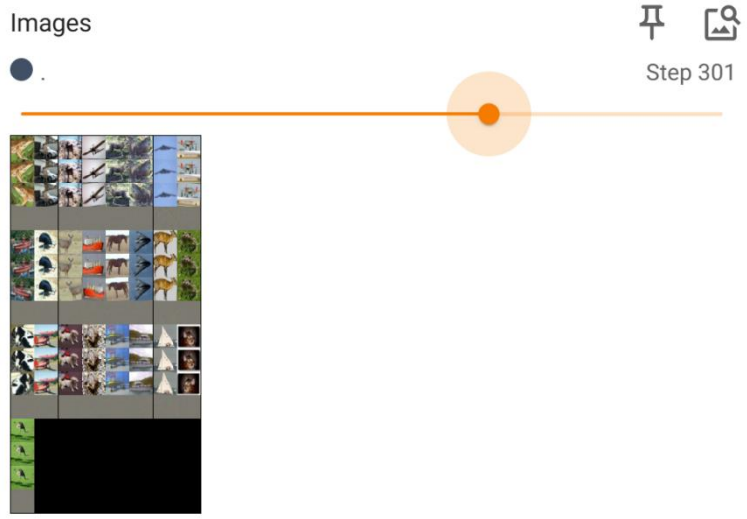
评估

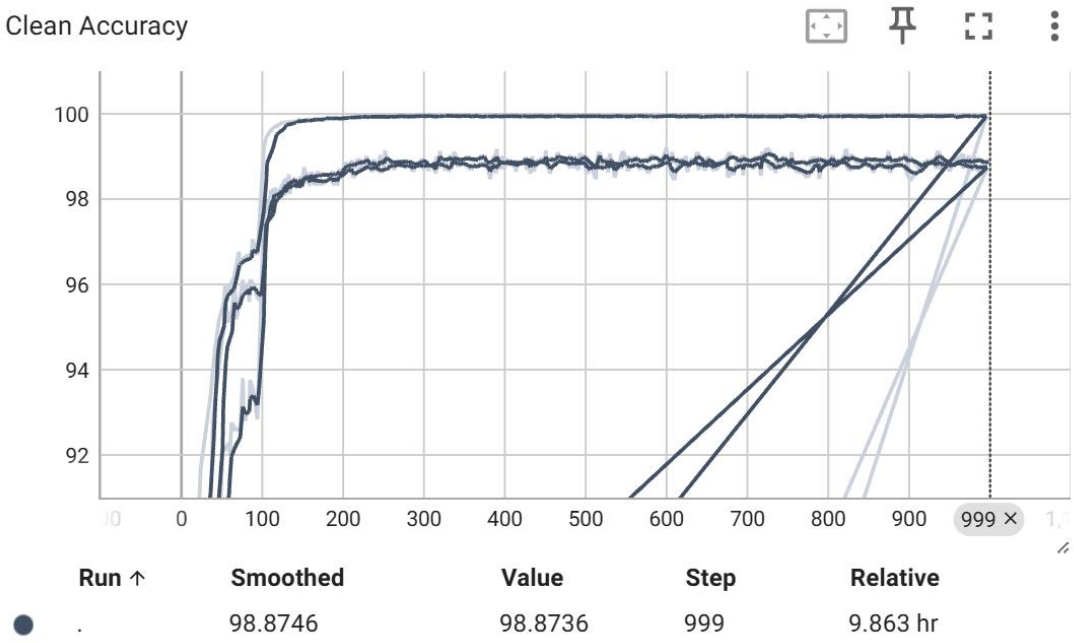
```
 Eval:
backdoor target tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0], device='cuda:0')
clean target tensor([4, 5, 5, 5, 0, 0, 2, 1, 6, 1, 8, 2, 6, 0, 3, 8, 1, 4, 9, 9, 7, 2, 0, 1,
        5, 1, 3, 5, 0, 4, 1, 1, 2, 6, 2, 4, 7, 4, 7, 7, 0, 6, 3, 8, 7, 6, 0, 7,
        2, 3, 0, 4, 7, 7, 6, 5, 0, 0, 9, 5, 8, 1, 2, 7, 1, 9, 5, 7, 0, 8, 0, 7,
        9, 5, 3, 7, 2, 9, 1, 3, 7, 8, 0, 2, 2, 0, 4, 1, 6, 5, 1, 2, 4, 9, 6, 9,
        9, 8, 4, 7, 3, 3, 5, 9, 0, 3, 1, 0, 7, 5, 3, 0, 9, 3, 9, 9, 3, 8, 4, 7,
        0, 8, 4, 7, 8, 1, 8, 5], device='cuda:0')
78 79 Clean Acc: 94.6700 - Best: 94.7000 | Bd Acc: 99.9700 - Best: 99.9700 | Cross: 94.3700
```

评估过程会输出本次过程和历史最优的准确率。

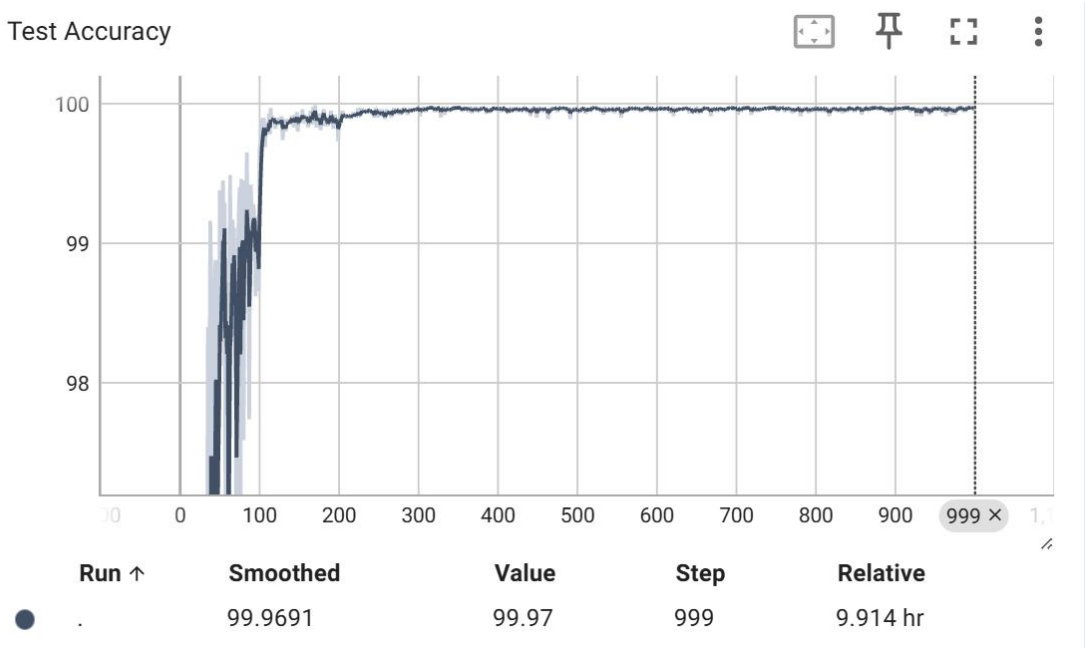
Backdoor target tensor：全为 0，因为本次任务是一个 all2one 任务，所以目标 class 就是用一个张量，这里选择 0。

相关图：
攻击时的相关图片





训练过程中，干净样本检测准确率，横轴是训练轮数。



训练过程中，测试样本检测准确率，横轴是训练轮数。

{
"clean_acc": 94.69999694824219,
"bd_acc": 99.97000122070312,
"cross_acc": 94.41999816894531
}

最终准确率。bd_acc 是后门攻击样本攻击成功率；cross_acc 为负样本。

复现难点：

唯一难点就是指定 torch 版本不适用于服务器，多次尝试更换 torch 版本后，就 OK 了。