对抗训练大作业报告

韦锡宇 200013085

1. Introduction

深度神经网络在近年来由于其用途的泛化性，被用于包括图像分类在内的诸多领域。但是广泛的应用也为其网络的鲁棒性(robust)带来了挑战，一个具有比较好鲁棒性的网络能够在输入有噪声的干扰下仍能给出置信度高的结果。

为了应对多种情况的输入，包括来自恶意客户端(malicious client)的输入，我们希望在对抗样本(adversarial sample)上测试模型的表现，并用其作为评价指标。这种实验性或者目的性的攻击可以分为两种类型：

- 白盒攻击(white -box attack)：生成对抗样本时可以拿到模型以及内部参数，根据数据的梯度等内部信息开展攻击
- 黑盒攻击(black-box attack)：生成样本时不知道模型的内部结构与参数，仅可以调用模型获取对于给定输入的输出结果。

这次的对抗训练作业主要采取的是白盒攻击的形式

2. Methods

这个部分主要内容是作业的几个防御方法的实现过程，包括PGD，模型量化，动态阈值函数以及正交和谱范正则惩罚项的实现。

a. PGD

PGD是脱胎于FGSM的一种迭代攻击方法，其相比于基本迭代方法（Basic Iterative Method)，加入了随机启动项，使得对抗样本更真实。其中FGSM的数学表示可写作$\Delta(x, x^{'}) = \varepsilon \cdot sign(\nabla_x f(x), y)$

容易分析这其实就是使用原样本和标签的loss产生的梯度，在原图中叠加自定义强度的梯度，作为扰动后样本 $x_{m+1} = x_m + Clip(\varepsilon \cdot sign(\nabla_{x_m} f(x_m), y))$，迭代方法的多次迭代可以找到更好的扰动方向，让对抗样本的扰动表现更好。

以下给出PGD的单步实现代码流程，为了简化流程省略了代码细节。

```
    def single_step(adv_x)

        preds = self.model(adv_x)#获取x_m的预测向量
        loss = loss_func(preds, labels)#计算损失
        loss.backward()#拿到x_m的梯度

        pertubation = (torch.tensor(self.iter_eps)
*torch.sign(grad))#用指定的iter_eps乘以梯度
        adv_x_new= adv_x + pertubation#得到新的x_(m+1)

        pertubation = torch.clip(adv_x_new,
self.clip_min, self.clip_max) - x
        pertubation=torch.clip(pertubation,-
self.eps,self.eps)

        return pertubation#夹紧扰动并回传
```

利用PGD我们可以实现模型的对抗训练,在这里只是简单的将对抗样本当作数据增广进行训练。

```
def adversarial_train(model,inputs,loss):

    output=model(inputs)
    loss+=loss_func(output,labels)#获取原始样本的loss

    adv_inputs=PGD_Attack(inputs)#拿到对抗样本
    output=model(adv_inputs)#增广训练
    loss+=loss_func(output,labels)#拿到对抗样本的loss

    return loss#回传loss
```

B. Quant

模型量化是模型压缩的一种常用做法。其主要做法是将参数的精度量化到指定的精度，例如$8bit$。假设有一个权重矩阵W，它本来是浮点类型的表示，我们可以采用$Scale \cdot W_{integer}$的方式来近似表达矩阵，其中$Scale$定义为矩阵中绝对值的最大值除以精度系数，考虑$8bit$量化，$Scale$即为$\frac{max(W)}{2^{8-1}-1}$，在这里我们可以把精度作为超参数进行设置。

以下给出Quent的实现代码，实际上，为了实现Quent我们可以包装一个关于nn.Conv2d 和nn. Linear的类，在forward中实现Quent过程，同样为了方便理解省去了一些细节，情景是$8bit$

```
def Quent(x):
        x_new=torch.abs(x)
        max_v=torch.max(x_new)
        scale=max_v/torch.tensor(127)#算出当前矩阵的scale值

        output=x/scale
        output=torch.round(output)
        output=torch.clip(output,-128,127)#获得分解后的
Matrix_integer

        return output,scale#返回分解后矩阵和scale值
```

值得注意的是，在Quent过程中因为涉及对权重的改变，所以必须**谨慎地在用整数进行运算之后把权重换回浮点数**，才能让优化器真正更新到结点权重

```
def forward(self, x):

        x_new,scale1=Quant(x)
        x.data=x_new*scale1#得到关于x的量化向量

        self.orignal_weight=self.weight.data
        weight_new,scale2=self.quant2(self.weight)
        self.weight.data=weight_new*scale2#得到权重的量化向
量

        output=F.linear(x, self.weight, self.bias)
        self.weight.data=self.orignal_weight#记得把权重替换
回原结点

        return output#返回量化计算的值
```

由此封装好的Linear和Conv就可以直接在网络结构中使用实现量化。

关于权重替换的细节，其实可以有两种选择，可以loss.backward()前替换，也可以在loss.backward()后替换，两者分别对应的是用量化值计算梯度还是原始值计算梯度，但是经过实验，实际上两者的性能差别并不大，模型实际在量化中得到的应该是**经过量化后的向前输出值**，梯度的计算的误差则不是重要方面。

c. Activation

要实现动态阈值的激活函数，首先我们应该能够确定关于这个激活函数的输入的大致区间，再进行分段的激活，如果一开始直接指定了确定的区间，根据我在模型上的实验，即使把学习率调大依然很难得到收敛，因为这样的激活函数在一开始很可能阈值区间和输入数据存在很大的差别，导致收敛失败。所以我们考虑在训练过程中，先采用基本的Relu包装函数，统计输入数据的均值和方差，在给定的参数T个$Epoch$后再启用阈值激活。这里的方法参考了PWLU(Piecewise Linear Unit)的设计思想。[straight-through estimator]

具体来说，我们可以统计经过动态阈值函数的参数范围。

```
mean = x.mean([0,1,2,-1])
var = x.var([0,1,2,-1])#求出输入数据的均值和方差
self.running_mean = (self.momentum * self.running_mean) +
(1.0-self.momentum) * mean
self.running_var = (self.momentum * self.running_var) +
(1.0-self.momentum)*(x.shape[0]/(x.shape[0]-1)*var)
```

然后根据统计的$mean$和var我们可以得到关于输入数据的统计上界和统计下界，这里基于数据是正态分布的假设

$$B_l = mean - 3 * var \qquad B_r = mean + 3 * var$$

有了统计下界，我们就可以先对数据切分出N个区间，在每个区间内考虑激活

$$\begin{cases}(x-B_L)*K_L+Y_P^0 & x<B_L\\ (x-B_R)*K_R+Y_P^N & x\geq B_R\\ (x-B_{idx})*K_{idx}+Y_P^{idx} & B_L\leq x<B_R\end{cases}$$

$$idx=\lfloor\frac{x-B_L}{d}\rfloor \tag{2}$$

$$B_{idx}=B_L+idx*d \tag{3}$$

$$K_{idx}=\frac{Y_P^{idx+1}-Y_P^{idx}}{d} \tag{4}$$

公式推导

显然这样的激活设计使得我们计算idx和激活后的值时，计算过程是可导连续的。d表示左右边界均分成N份产生的长度，这样的话我们的激活区间实际上是均匀的，但是每个区间的激活初值Y又是可学习的，为网络带来了灵活性。

```
def forward(self, x, mode):

        if mode==1:
            d=(self.Br-self.Bl)/self.N#均匀的划分值，可以看作x轴区间左端点
            DATAind = torch.clamp(torch.floor((x-self.Bl.item())/d),0,self.N-1).to(self.device)#将输入进行区间划分
            Bdata = self.Bl+DATAind*d#拿到每个数据对应的左端点

            maskBl = x<self.Bl#左边溢出的Mask
            maskBr = x⩾self.Br#右溢出的Mask
            maskOther = ~(maskBl+maskBr)#区间内的Mask

            Ydata = self.Yidx[DATAind.type(torch.int64)]
            Kdata = (self.Yidx[(DATAind).type(torch.int64)+1]-self.Yidx[DATAind.type(torch.int64)])/d#数据的斜率
            return  maskBl*((x-self.Bl)*self.Kl+self.Yidx[0]) + maskBr*((x-self.Br)*self.Kr + self.Yidx[-1]) + maskOther*((x-Bdata)*Kdata + Ydata)#计算最后的激活值
```

这就是最后激活函数的forward流程

D. Regularization

正则化主要要实现两种正则，正交正则和谱范正则。

首先是正交正则，其目的是为惩罚网络权重的非正交性，由于正交矩阵的奇异值为定值，鼓励正交化可以让扰动的干扰减少，这是我们希望得到的。

```
            for layer in self.modules():

                if type(layer) == Linear:

                    w = layer._parameters[ 'weight' ]
                    m = w @ w.T
                    loss=loss+torch.norm(m -
torch.eye(m.shape[ 0 ])

                elif type(layer) == Conv:

                    w = layer._parameters[ 'weight' ]
                    N, C, H, W = w.shape
                    w = w.view(N * C, H, W)
                    m = torch.bmm(w, w.permute(0, 2, 1))
                    loss=loss+ torch.norm(m -
torch.eye(H)

                #分别对卷积和线性层权重计算正交惩罚

            return beta*loss
```

接下来是谱范正则化，谱范正则化给出了一个更紧的上界，我们希望得到矩阵的最大奇异值，然后限制最大奇异值的大小来限制扰动的影响，当然，矩阵的奇异值分解是计算代价十分高昂的，我们可以通过迭代法来逼近得到最大奇异值。

$$v = \frac{W^T u}{||W^T u||_2} \qquad u = \frac{W^T v}{||W^T v||_2}$$

根据原论文，这样的迭代在保存v的情况下只需要进行一次

```
                if type(layer) == Linear or
type(layer)==Linear_re:
```

```
                    iteration = 1
                    w = layer._parameters[ 'weight' ]
                    v = layer.v#保存v值

                    u = torch.empty((w.shape[ 0 ], 1))

                    nn.init.uniform_(u,-0.1,0.1)#u的值可以
随机赋值

                    for _ in range(iteration):
                        u =
torch.nn.functional.normalize(torch.mm(w.detach(), v),
dim=0)
                        v =
torch.nn.functional.normalize(torch.mm(w.detach().T, u),
dim=0)
                    layer.v = v#更新v值

                    value=torch.squeeze((u.T @ w @
v).view(-1))
                    loss=loss+value
```

值得注意的是，**卷积层的权重矩阵略有不同，我们可以把(C_{out}, C_{in}, H, W)的矩阵reshape成$(C_{out}, C_{in} * H * W)$进行迭代操作。**

E. FEATURE squeezing

本次作业附加题我选择了复现feature squeezing的三种样本特征压缩方法

- Squeezing Color Bits 考虑CIFAR-10数据集，其图片都是采用每通道$8bit$真彩表示，但是这么多的位数用来展现色彩很可能是多余的，如果我们能把色彩空间压缩到$4bit$或者更低，对抗样本的扰动可能会对模型影响减少
- Local Smoothing 采用一个中值滤波核在整个图片上进行滤波，在padding上采用反射padding，这样也可以减少样本扰动的影响
- Non-local Smoothing 一种指定搜索空间的压缩特征算法，考虑每个patch和周围patch的关联性进行压缩特征

首先是Squeezing Color Bits的实现

```
def bit_squeeze(inputs,out_bits=4):

    matrix_interger=torch.round(inputs*(2**out_bits-
1))
    new_inputs=matrix_interger/(2**out_bits-1)#乘以对应
的压缩指数再重新置回0-1

    return new_inputs
```

然后是Local Smoothing 采用了Scipy的包

```
def median_filter(inputs,kernel_size=2):

    inputs=inputs.detach().cpu().numpy()

    inputs=ndimage.filters.median_filter(inputs, size=
(1,1,kernel_size,kernel_size), mode='reflect')#注意 在四维
空间卷积

    return torch.from_numpy(inputs)
```

最后是Non_local Smoothing，注意在smoothing前必须进行[0-1]的标准化处理

```
def
non_local_filter(inputs,search_window=11,patch_size=3,str
ength=4):

    inputs=inputs.detach().cpu().numpy()
    inputs=np.transpose(inputs,(0,2,3,1))
    inputs=np.round(inputs*255)
    inputs=inputs.astype(np.uint8)

    for i in range(inputs.shape[0]):
        inputs[i]=
cv2.fastNlMeansDenoisingColored(inputs[i],None,search_win
dow,search_window,patch_size,strength)

    inputs=inputs/255
    inputs=inputs.astype(np.float32)
    return torch.from_numpy(np.transpose(inputs,
(0,3,1,2)))
```

3. EXPERIMENT

这个部分将展示实验获得的数据，并且对数据做一定的分析

以下是**作业基础要求**下的消融实验。

8-Bit 量化	激活函数	对抗训练	正交正则	谱范正则	Accuracy	PGD Accuracy	Attack Success Rate
×	×	×	×	×	**71.3%**	**27.4%**	**66.4%**
√	×	×	×	×	72.3%	31.1%	62.5%
×	√	×	×	×	72.1%	30.1%	64.4%
√	√	×	×	×	71.0%	32.0%	61.0%
×	×	×	√	×	71.1%	26.6%	67.3%
×	×	×	×	√	71.1%	27.8%	65.9%
√	×	×	×	√	70.9%	30.9%	61.8%
√	√	×	×	√	**72.3%**	**33.9%**	**59.5%**
×	×	√	×	×	**71.0%**	**55.0%**	**26.0%**
√	√	√	×	×	**71.0%**	**60.0%**	**21.0%**

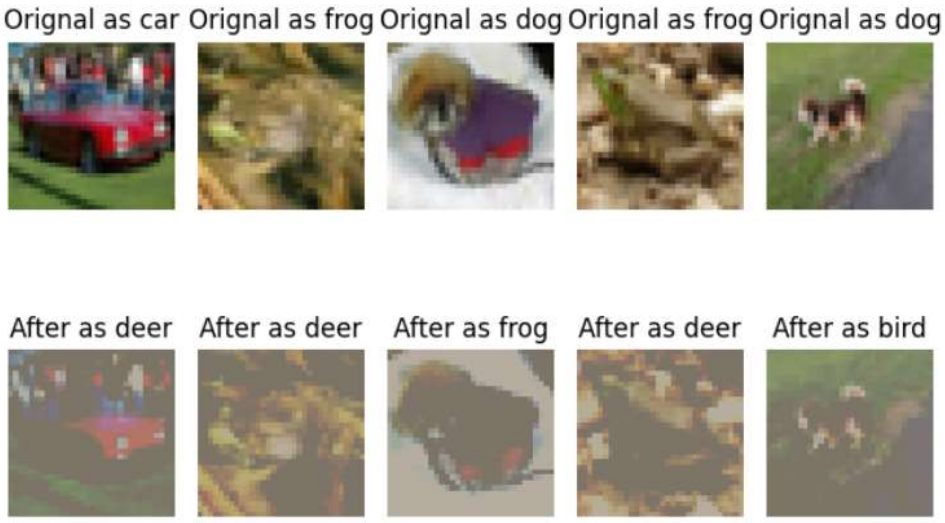
> 标红表示为基准数据
>
> 蓝色表示对抗数据
>
> 加粗表示不使用对抗训练的最佳数据

从表格中可以看出，在实验条件下，不使用对抗样本下最佳的组合条件为***量化&激活函数&谱范正则***。由于本身模型的**训练权重个数不多**，单独的谱范正则获得的提升并不明显，正交正则甚至出现了下降的情况，但是在后面引入了激活函数后（行8），可以发现由于可训练权重的增多，计算的复杂性增大，谱范正则出现了明显的提升效果。

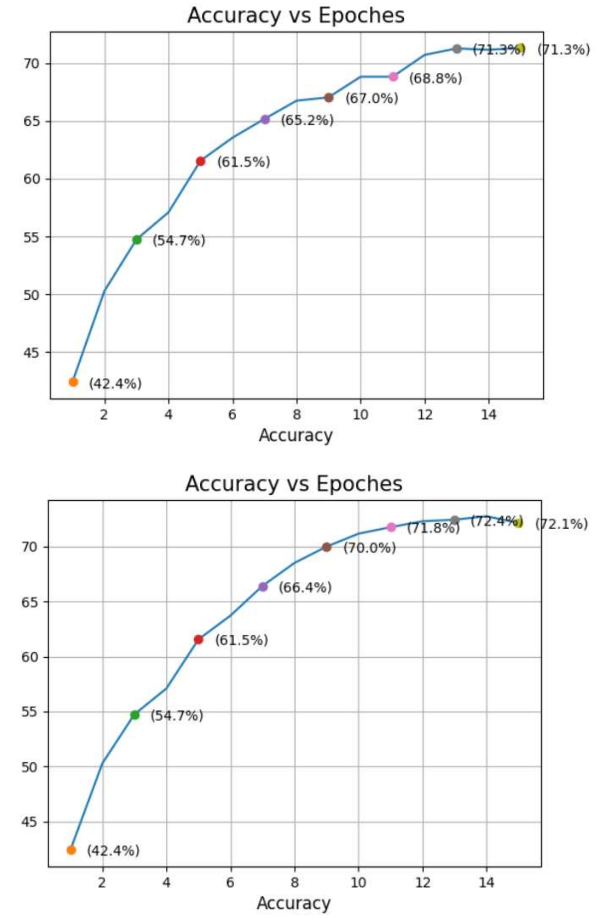
关于模型压缩的两种离散化方法，量化和激活函数，都各自取得了不错的成绩，单独量化和单独激活都有将近3%-4%的提升（行1 2），而两者组合更是达到了5%的提升。

对于对抗训练来说，因为使用了攻击的样本当作训练输入，所以获得的提升是非常可观的，对抗训练搭配上量化以及激活函数提升也非常明显，相比于原来有5%的提升。

观察下面的攻击前后展示(经过了反normalize过程)，可以看到，PGD攻击在这里主要针对于色差，并没有肉眼可见的改变形状



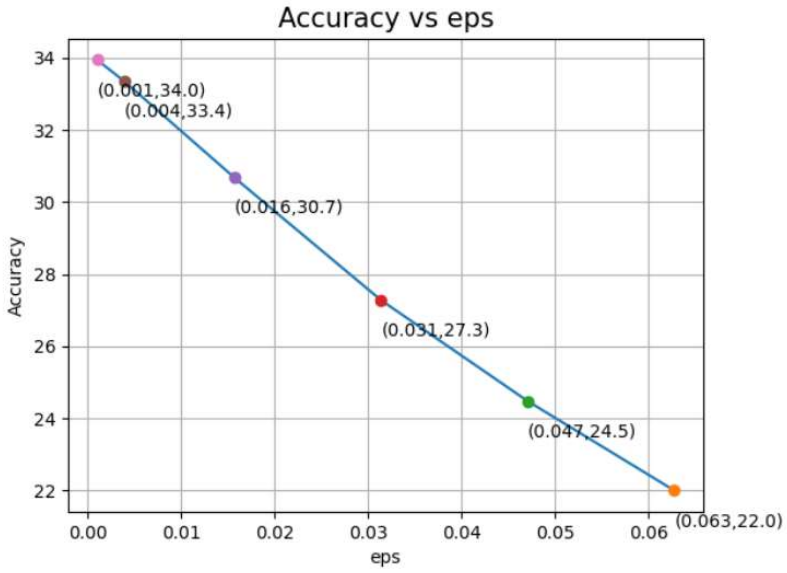
攻击图片展示



训练曲线 左边为基准模型 右边为Activation模型

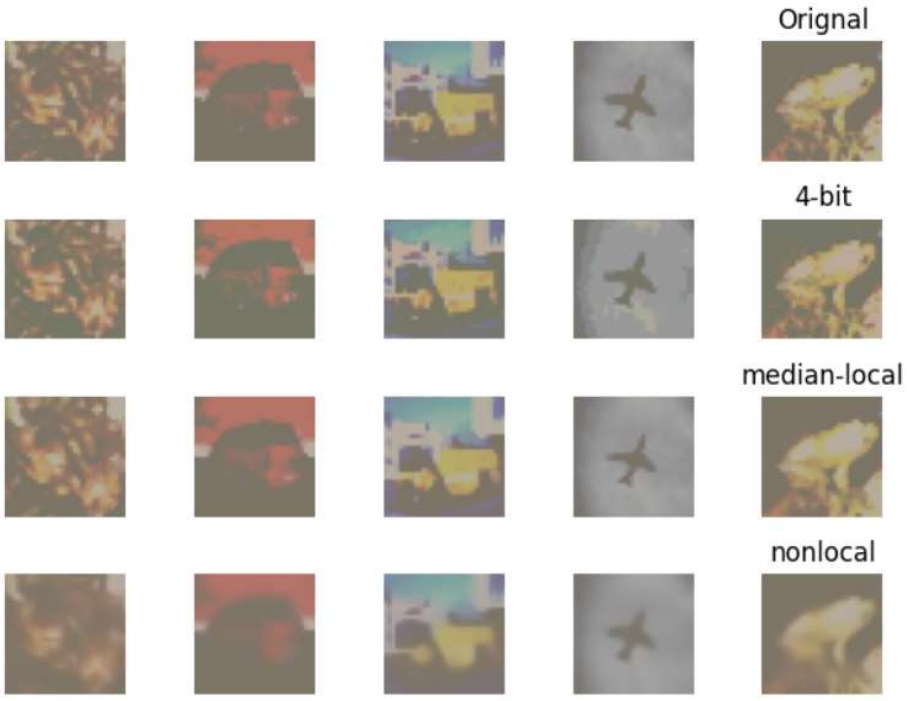
右边训练曲线之所以在5个$Epoch$后斜率变高，是因为在5个$Epoch$后引入了动态阈值激活，使得其表现更好。

随后我改变PGD的最大扰动参数eps，以基准模型为基础进行了对比实验，可以看到，随着eps的增大，扰动值的变大，模型的在PGD攻击下的准确率逐渐降低，这是符合预期的



Accuracy-Eps

接下来是关于附加任务的实验，首先我们通过可视化观察一下三种处理方式带来图片的改变



Squeeze 可视化

可以看到，其中$4-bit$的色彩压缩带来了一些色彩的变化，但是图片依旧是具有强辨识度的，而经过$median-local$处理之后，图片变得有些模糊，并且把发生突兀变化的像素进行了模糊。而$non-local$的处理则是得到了最平滑的图片。接下来我们可以观察一下它们分别在对抗防御的效果

Squeeze 类别	Test Accuracy	PGD Accuracy	Attack Success Rate
基准模型	**71.3%**	**27.4%**	**66.4%**
4-bit Color	**71.3%**	27.4%	66.4%
1-bit Color	66.9%	28.4%	66.2%
Local Smoothing	67.3%	**29.6%**	**64.1%**
Non-Local Smoothing	68.5%	26.1%	69.7%

> 标红表示为基准数据
>
> 加粗表示最佳数据

可以看到，**4-bit Color**的压缩，对于原模型几乎没有造成任何影响，**1-bit Color**的压缩让模型丢失了精度但是提升了些许的对抗精度。并且和论文中报告得一样，**Local Smoothing**对于CIFAR10的效果是最佳的，提升了一个点左右的对抗精度。**Non-Local Smoothing**的平滑处理丢失的精度比中值滤波要小，但是在对抗精度上不尽如人意，但是后面可以验证，这是由于我们的数据集为了加速收敛使用了

Normalize这对于平滑处理来说是致命的，**Normalize**一定程度上减少了平滑处理后的平滑性。

为了对比的公平性，我们加入了一组**未经Normalize**的训练对比。

Squeeze 类别	Test Accuracy	PGD Accuracy	Attack Success Rate
基准模型	**65.2%**	**1.9%**	**97.1%**
4-bit Color	**64.6%**	3.8%	94.1%
Local Smoothing	63.0%	19.9%	69.5%
Non-Local Smoothing	63.2%	**22.1%**	**66.2%**

从数据可以发现，当不加入**Normalize**时，因为不用经过反**Normalize**的处理，三种方法均有提升，其中Smoothing的两个方法提都提升可观。**Test Accuracy**的降低是由于未经**Normalize**的数据收敛速度会比原来的慢，而**PGD Accuracy**的下降，则是由于**Normalize**以后，矩阵值被相应的放大了些许，所以对扰动的敏感性会下降。

总结来说基于样本去噪附加方法的对抗防御在实验中表现的不如前面的从训练角度入手的防御手段行之有效，而且不太能够处理**Normalize**的数据，灵活性也要下降不少，对抗防御最行之有效的办法当然是采用对抗样本加入训练，当然，在对方攻击手段未知的情况下，进行模型量化，阈值激活以及正则约束，也都是很好的做法。

https://github.com/lemon-prog123/against_job代码已经开源在我的github