AI Competition Technical Report

A．战队信息

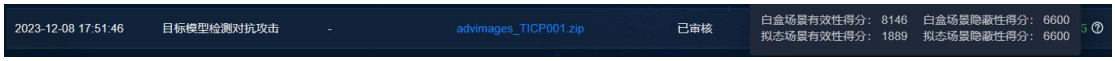
（1）战队名称:TICP001

（2）选择的赛题:目标模型检测对抗攻击

（3）最终得分情况

最终本赛题总得分：17925

其中白盒场景有效性得分： 8146 ； 白盒场景隐蔽性得分： 6600；拟态场景有效性得分： 1889 ； 拟态场景隐蔽性得分： 6600。



B．攻击实现流程

（1）攻击方法简介

对于本题的对抗补丁，因为没有限制 patch 的数量和形状我们采用了集中和分散相结合的方式来设计 patch，具体而言 patch 是基于 box 位置的均匀分布的横杠，初始化为灰色。对于 yolov3 模型，设定每个 box 的得分作为损失函数，通过符号梯度下降(svg)的方法实现对于图片的优化，达到最好的攻击效果。

如下图所示，图 1 是刚初始化即可攻击成功的对抗样本，图 2 是优化多轮后的对抗样本，可以看出像素点的颜色变化。

图 1

图 2

（2）攻击流程

下面从补丁位置确定、补丁形状选取、loss 损失函数选取、优化方法选择方面说明攻击流程。这几个部分一一确定的过程就是整个攻击实施的流程。

1. 补丁位置确定

这里我们本来是想基于 DPatch 的工作创建一个方形的 patch 并且就放在左上

角，但是后面参考了一些国内外的其他比赛和文献之后还是确定了 patch 的位置正好应该是图片中物体检测框的位置。在比赛中我们采取了在攻击脚本中调用模型的预测功能来实现对要攻击图片中物体的定位（这里其实有点傻，当时没注意到在 attacked_images_file.txt 直接给了框的坐标）。

有关的实现代码为：

```
    def detect_box(self, img):
        # 一些预处理，有点长不放出来了
        image_shape = [416, 416]
        img = img.to(device)
        img = torch.autograd.Variable(img)
        outputs = self.net(img)
        outputs = self.bbox_util.decode_box(outputs)
        #---------------------------------------------------------#
        #   将预测框进行堆叠，然后进行非极大抑制
        #---------------------------------------------------------#
        results = self.bbox_util.non_max_suppression(torch.cat(outputs,
1), self.num_classes, self.input_shape,
                image_shape, self.letterbox_image, conf_thres =
self.confidence, nms_thres = self.nms_iou)

        top_boxes  = results[0][:, :4]
        return top_boxes
```

在后面生成 patch 的时候调用这个接口即可。

2. 补丁形状选取

这一步我们其实经历了很多探索和挫折，这里先只放出最后的结果，原因在最后一部分再做解释。

我们选取的 patch 是在 box 中尽量均分的横线，每条横线的长度与 box 的宽度相同，每条横线的宽度为 1。同时满足这些横线占用的像素点总数尽量靠近 3000，但是需要注意的是，有些 box 的大小本身不足 3000，会出现完全遮挡 box 的情况，所以我们需要控制每两条横杠之间的距离大于 1，如下面的图 3 和图 4 所示。

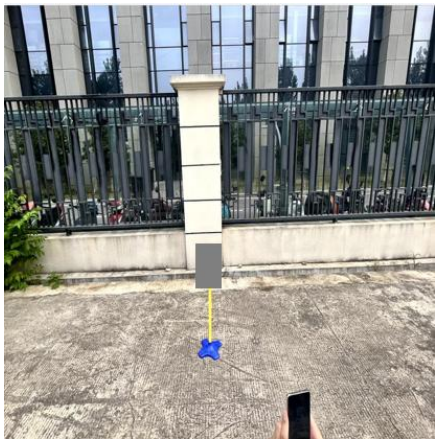


图 3 被完全遮挡的目标

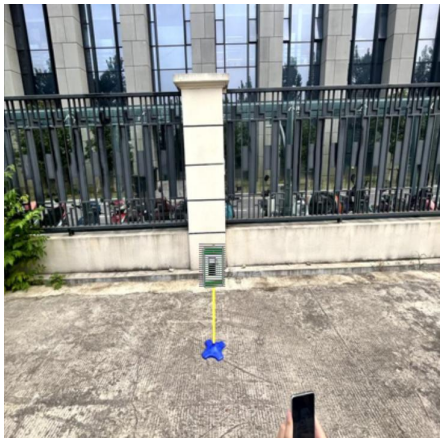


图 4 控制横杠距离就不会完全遮盖

综上，我们使用下面的公式确定横线的条数和没两条横线之间的距离：

$$Line\_num = \left\lfloor \frac{3000}{box\_w} \right\rfloor \text{.....(1)}$$

$$\text{int}erval\_h = \left\lfloor \frac{box\_h}{Line\_num} \right\rfloor = \left\lfloor \frac{box\_w \times box\_w}{3000} \right\rfloor \text{...(2)}$$

其中，Line_num 表示横杠的条数，interval_h 表示每两条横杠上端的距离。

使用上面的算式，从 y 坐标为 box 的最高处的坐标开始画横杠我们就可以确定出整个 patch 的样子，我们在代码中用 mask 表示这个形状 mask[i,j]=1 表示该处的像素点可以被修改，也就是 patch 所在的地方。相关的代码如下所示：

```
'''
创建全是横杠的 patch
'''
def create_plan_mask(yolo, net, img_path, box_scale, shape=(416, 416)):
    # 初始化一个空的遮罩
    mask = torch.zeros(*shape, 3)

    # 打开并转换图像
    img = Image.open(img_path).convert('RGB')
    # 调整图像大小为 416x416
    resize_small = transforms.Compose([
        transforms.Resize((416, 416)),
    ])
    img1 = resize_small(img)
    # 检测边框
    boxes = yolo.detect_box(img=img1)
    grids = boxes
    (y1, x1, y2, x2) =  boxes[0]

    # 确保坐标在图像范围内
    x1 = int(np.clip(x1, 0, 415))
    x2 = int(np.clip(x2, 0, 415))
    y1 = int(np.clip(y1, 0, 415))
    y2 = int(np.clip(y2, 0, 415))
    # 计算边框的高度和宽度
    box_h, box_w = y2 - y1, x2 - x1
    # 三千个像素点全部用掉,但是需要保证横纵的大小都大于 0
    pixel_num = 3000
    # 为了避免浮点误差需要向上取整
    interval_h = int(box_w * box_h / 3000) + 1
    if interval_h <= 1:
        interval_h = 2
    pixel_changed = 0
    # 在遮罩上添加像素点
```

```
    for i in range(int(box_h / interval_h)):
        for j in range(box_w):
            y = int(y1 + i * interval_h)
            x = int(x1 + j)
            if y < shape[0] and x < shape[1]:
                mask[y, x, :] = 1
                pixel_changed = pixel_changed + 1
                if(pixel_changed >= 3000):
                    break
        if(pixel_changed >= 3000):
            break
    # 将结果输出到 log.txt 文件
    print(pixel_changed)
    with open('log.txt', 'a') as log_file:
        log_file.write(f'File: {img_path}, Pixels Changed:
{pixel_changed}\n')
    return mask
```

3. loss 损失函数选取

我们采用直接从输出中获取每个类的得分作为对抗样本优化的损失函数，代码如下所示（其实我们觉得还可以再优化，后面的部分再说）：

```
def get_cls_scores(self, img:torch.tensor):
        img = _input_transform(img).to(device)
        output = self.net(img)

        scores = []
        for item in output:
            h, w = item.shape[-2], item.shape[-1]
            item = item.reshape(-1, 5+20,
h*w).permute(1,0,2).reshape(5+20, -1)
            scores += [item[4, :].sigmoid()]

        return scores
```

4. 优化方法选择

选择最简单的符号梯度下降方法对前面生成的 mask 中的每个像素进行反向传播和优化，有关代码如下所示：

```
def specific_attack(yolov3_helper, img_path, mask, save_image_dir):
    img = cv2.imread(img_path)
    img = torch.from_numpy(img).float()

    t, max_iterations = 0, 600
    eps = 1
    w = torch.zeros(img.shape).float() + 127
```

```
    w.requires_grad = True
    success_attack = False
    min_object_num = 1000
    min_img = img
    while t < max_iterations:
        t += 1
        patch_connecticity = torch.abs(get_delta(w) - img).sum(-1) == 0
        patch = get_delta(w)
        patch[patch_connecticity] += 1
        patch_img = img * (1 - mask) + patch * mask
        patch_img = patch_img.to(device)
        attack_loss, object_nums = yolov3_helper.attack_loss(patch_img)
        if min_object_num > object_nums:
            min_object_num = object_nums
            min_img = patch_img
        if object_nums == 0:
            success_attack = True
            break
        if t % 20 == 0:
            print("t: {}, attack_loss:{}, object_nums:{}".format(t,
attack_loss, object_nums))
        attack_loss.backward()
        w = w - eps * w.grad.sign()
        w = w.detach()
        w.requires_grad = True
    min_img = min_img.detach().cpu().numpy()
    with open('log_t.txt', 'a') as log_file:
        log_file.write(f'File: {img_path}, t: {t}\n')
    # 保存图像，攻击失败时在文件名中添加 "_fail"
    save_path = os.path.join(save_image_dir, img_path.split("/")[-1])
    if not success_attack:
        save_path = save_path.replace(".", "_fail.")
    cv2.imwrite(save_path, min_img)
    return success_attack
```

（3）攻击样例展示

这里展示一个 yolov3 中白盒攻击成功的样本，一个失败的样本（一共只有五个）如图 5、图 6、图 7 所示：

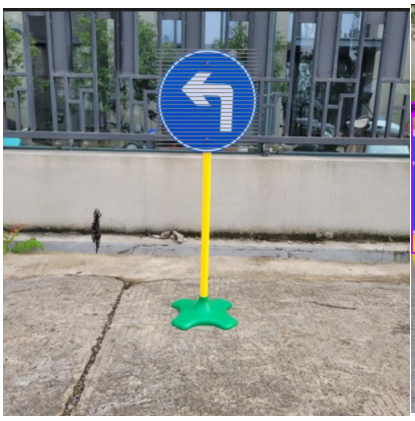


图 5 攻击成功的样本

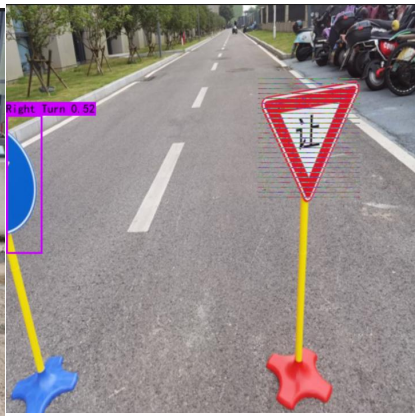


图 6 攻击失败的样本

（4）攻击成本评估

根据输出的日志来看，我们使用的攻击方法非常具有优势。下面的图 7 是我们的部分日志。

attack.py predict.py process.py log.txt log_t.txt

Patch-attack > log_t.txt

```
File: images/IMG_10547.png, t: 1
File: images/IMG_10003.png, t: 2
File: images/IMG_10549.png, t: 1
File: images/IMG_10005.png, t: 2
File: images/IMG_10557.png, t: 1
File: images/IMG_10007.png, t: 1
File: images/IMG_10559.png, t: 1
File: images/IMG_10017.png, t: 1
File: images/IMG_10558.png, t: 1
File: images/IMG_10066.png, t: 1
File: images/IMG_10560.png, t: 1
File: images/IMG_10069.png, t: 25
File: images/IMG_10562.png, t: 1
File: images/IMG_10073.png, t: 22
File: images/IMG_10564.png, t: 1
File: images/IMG_10074.png, t: 79
File: images/IMG_10698.png, t: 1
File: images/IMG_10078.png, t: 49
File: images/IMG_10662.png, t: 1
File: images/IMG_10079.png, t: 46
File: images/IMG_10699.png, t: 3
File: images/IMG_10080.png, t: 15
File: images/IMG_10728.png, t: 9
```

图 7 运行过程中产生的部分日志

从日志可以看出，有很多对抗样本都只优化了 1 轮就达到了使得模型失效的作用，证明我们的方法至少有两点优势：(1)优化快，硬件成本低，整个优化过程在 RTX 3090 的显卡中大概只需要半个小时；(2)泛用性好，可以用于其他模型的黑盒攻击。第一轮就完整攻击证明这个 patch 基本没有利用 yolo 白盒信息，而是从图片本身的特征上完成了攻击。这也是我们的方式在拟态模型中也能起到很好效果的原因。

C. 攻击分析

有关我们方法的优势，总结为以下几点：

1、在 yolo 白盒场景下攻击性能极好，能达到几乎 100%的攻击成功率
2、方法泛化性能良好，很多刚初始化的灰色 patch 就能完成攻击
3、方法添加的灰色 patch 特征不明显，隐蔽性好，相应的得分更高
4、优化速度高

其实本题还有很多可以优化的地方，例如：

- 优化算法，试试 PGD?
- loss 函数上做个 NMS?
- 或许某种程度的正余弦曲线比横线效果更好?有没有别的优化算法直接找到这样的线?

这些都是值得再考虑的问题。

参考文献

[1] https://tianchi.aliyun.com/forum/post/127470

附录

下面想随便扯扯是怎么诞生的创建这样 patch 的思路，也当做是对这道题的一个总结吧。

我们首先采取的方案其实来自于天池比赛中的有关优秀数据，如下图：

比赛的两种白盒模型都是 anchor-based 的模型，YOLO 会提出很多 anchors，模型会给每个 anchor 回归位置偏移以及类别的概率分布。 Faster RCNN 在 RPN 网络中会提出上千个 proposals，经过 RoI pooling 后，每个proposal 也会对应一个类别的概率分布。

损失函数

本次比赛的攻击目标是使得检测器不能检测出结果，而不仅仅是检测错误，因此对于攻击要求更高。我们的攻击方法主要是降低所有前景目标的概率值，因此我们只关注 YOLO 和 Faster RCNN 的最终结果的类别概率分布。我们定义 N 个 proposal 的概率分布为 \(p_1 .... p_N\)，攻击目标是使前景概率小于判定阈值 t（如果某个类别的概率值大于 t，则表示候选框对应这个类别的目标）。损失函数如下：

L(x , p, M) = \sum_{i=1}^{N} \sum_{c=1}^{C} max (0, f_i^{c}(x\cdot (1-M) + p\cdot M)-t)

这里，x 表示原始图片，p 表示 patch，与 x 的形状一致；M 表示 patch 的的位置掩码；f_i^{c}(\cdot) 表示第 i 个 proposal 的第 c 个 类别的概率值 (p_i=f_i(x))。

patch 的选择

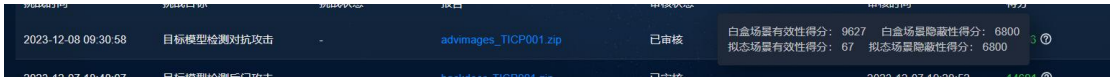
patch的选择对于攻击成功率影响很大。如果是正常的矩阵形状的 patch，可以以较高的成功率攻击 YOLO，但是对于 Faster RCNN 的攻击成功率很低。在 Faster RCNN中，存在 RPN 模块，因此特征图中的有些特征的感受域较小，patch 不足以影响到远距离位置的特征。

因此我们选择了比较发散的 patch 形状，在保证连通域个数和 patch 大小合理的前提下，预先定义 patch 的位置掩码。主要有两种形式， 网格形状的和星状的（如下图所示），网格形状的选取 1\times1, 2\times2, 3\times3的。我们选取每个图片最小的前十个目标方框，将 patch 放置在这些目标的方框内，。为了获得更好的白盒攻击效果，选取这些攻击本中白盒得分最高的样本。图示如下：

在比赛前我们准备的版本中，损失函数和优化算法基本与上面的博客一致，patch 的形状稍微做了一些改动，但是还是发散的星形，如下所示：

这个版本的答案交上去之后获得了 10000 出头的得分，但是有一个很自然的想法，这个 patch 能不能更加发散？于是就有了我们的第二版，将 3000 个像素点全部均匀分布在 box 当中，如下所示：

这一版交上去之后发现只得了 8000 多分，一看评分记录发现是拟态得分太低了，虽然在 yolo 白盒上隐蔽性和有效性都拉满了，但是拟态不忍直视，我一度以为是分数判错了.....



在得到没有错误的答复之后，我重新思考了一下：是不是拟态下的性能跟 patch 的集中程度有关？星星形状的 patch 在拟态场景下的得分更高，而散点几乎没分，是不是就说明了需要有一定的集中连续的 patch，来遮挡或者分割图片原本的特征，从而达到对于拟态模型上也有很高得分的效果？

但是太集中的 patch 在白盒上的攻击效果也相对来说没有那么好，那有没有一种方案，能使得这个 patch 即分散，又聚集？我当时想了好多种方法，包括：

- 画横线
- 画竖线
- 画四十五度斜线
- 画个星星，剩下的像素点均匀分布

  为了验证这些想法哪个最有效，甚至开了一堆炉子，一个一个验证

并且在本地部署了 FID 的算法，这些想法全都完成优化以后全部拿出来比较，最后选择了最为优秀的就是最终的横线方法。