武汉大学国家网络安全学院

信 息 隐 藏 项 目 结 题 报 告

项目名称　生成式 AI 模型的水印生成与检测

指导教师　任延珍

小组成员　陈聪睿、李杰、王天翔、王梓宇

目录

一、项目介绍

本项目旨在通过对生成式 AI 模型在训练过程中添加水印来判别图像是由何种模型所生成。预期可以达到水印肉眼无法识别和模型之间的水印唯一性。小组分工如下：

陈聪睿同学负责水印微调模型的编写、训练与测试，并完善实验报告；

李杰同学负责鲁棒水印的生成；

王天翔同学负责考察水印对故意篡改的抵抗力；

王梓宇同学负责实现利用 Stable Diffusion 模型在现实中的实践框架，并完成实验报告。

二、相关背景

2023 年 10 月 11 日，全国信息安全标准化技术委员会官网发布《生成式人工智能服务安全基本要求》（征求意见稿），面向社会公开征求意见。征求意见稿首次提出生成式 AI 服务提供者需遵循的安全基本要求，涉及语料安全、模型安全、安全措施、安全评估等方面。在模型安全要求方面，征求意见稿从基础模型使用、生成内容安全、服务透明度、内容生成准确性、内容生成可靠性五大方面做出了严格要求。比如：提供者如使用基础模型进行研发，不应使用未经主管部门备案的基础模型；应在网站首页等显著位置向社会公开第三方基础模型使用情况等信息；生成内容所包含的数据及表述应符合科学常识或主流认知、不含错误内容等。

同时，生成式 AI 模型被盗用、AI 生成信息（图像、音频）被直接商用的现象层出不穷。误报（FP）成为实践中异常检测系统面临的最具挑战性的问题。为了防止发行者自训练的模型遭到盗窃，以及 AI 生成的信息被商用，需要对这些生成信息进行检测。

三、项目内容

（一）总体流程设计

在我们的实验中，我们采用了一种两阶段的方法来实现数字水印的嵌入和提取。首先，我们对水印模型进行了预训练，其中使用了载体信息和一串隐藏信息来训练一个水印注入模型。这个模型的目标是生成带有隐式水印的生成信息。为了增强水印的鲁棒性，我们借用了 HiDDeN 模型，该模型联合优化了水印编码器和提取网络的参数，将信息嵌入到图像中，同时对训练过程中应用的变换具有鲁棒性。

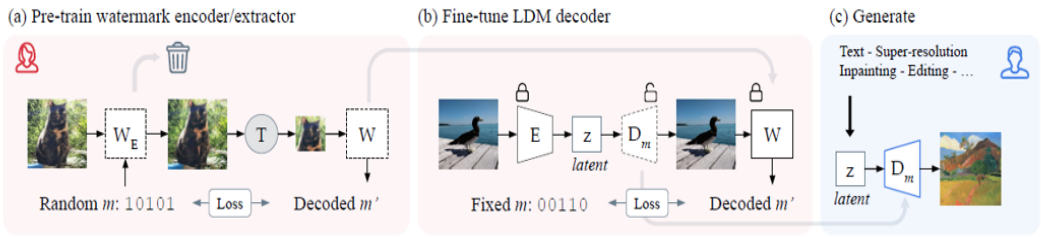


图 1： 总体流程

在水印鲁棒性的分析中， 我们使用了 HiDDeN，这是深度水印文献中的经典方法。它联合优化了水印编码器 WE 和提取网络 W 的参数，将 k 位信息嵌入到图像中，对训练过程中应用的变换具有鲁棒性。

经典的数据隐藏方法通常使用试探法来决定对每个像素修改多少。例如，一些算法处理一些选定像素的最低有效位；其他方法改变频域中的中频成分。这些试探法在它们被设计的领域中是有效的，但是它们基本上是静态的。相比之下，HiDDeN 可以很容易地适应新的需求，因为我们直接针对感兴趣的目标进行优化。对于水印，人们可以简单地重新训练模型以获得对新型噪声的鲁棒性，而不是发明新的算法。端到端学习在隐写术中也是有利的，在隐写术中，具有不同类别的嵌入函数(相同的架构，用不同的随机初始化训练，产生非常不同的嵌入策略)可以阻碍对手检测隐藏消息的能力。

（二）水印预训练模型

首先我们对水印预训练模型进行训练，我们参考了 HiDDeN，一个在深度水印文献中的经典方法。一次的训练过程如下：

```python
def train_one_epoch(encoder_decoder: models.EncoderDecoder, loader, optimizer, scheduler, epoch, params):
    """
    One epoch of training.
    """
    if params.scheduler is not None:
        scheduler.step(epoch)
    encoder_decoder.train()
    header = 'Train - Epoch: [{}/{}]'.format(epoch, params.epochs)
    metric_logger = utils.MetricLogger(delimiter="  ")

    for it, (imgs, _) in enumerate(metric_logger.log_every(loader, 10, header)):
        imgs = imgs.to(device, non_blocking=True)  # b c h w

        msgs_ori = torch.rand((imgs.shape[0], params.num_bits)) > 0.5  # b k
        msgs = 2 * msgs_ori.type(torch.float).to(device) - 1  # b k

        fts, (imgs_w, imgs_aug) = encoder_decoder(imgs, msgs)

        loss_w = message_loss(fts, msgs, m=params.loss_margin, loss_type=params.loss_w_type)  # b k -> 1
        loss_i = image_loss(imgs_w, imgs, loss_type=params.loss_i_type)  # b c h w -> 1

        loss = params.lambda_w*loss_w + params.lambda_i*loss_i

        # gradient step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

图 2： 预训练水印模型的训练过程

为了更好的利用我们的网络，我们对 HiDDeN 水印模型进行了修改，让其更适合我们的之后的工作。

HiDDeN 编码器 WE 构造如图，图像经过卷积层压缩后形成中间向量，之后将水印信息（msg）嵌入到中间向量后，再重新利用反卷积层将图片还原，得到嵌入好的水印图像。（其中的卷积层和反卷积层共用了 ConvBNRelu，其是一个经典的 Conv2D、BatchNorm2d 结构，最后使用 GELU 作为激活函数，详情请看源代码）

```python
class HiddenEncoder(nn.Module):
    """
    Inserts a watermark into an image.
    """
    def __init__(self, num_blocks, num_bits, channels, last_tanh=True):
        super(HiddenEncoder, self).__init__()
        layers = [ConvBNRelu(3, channels)]
        for _ in range(num_blocks-1):
            layer = ConvBNRelu(channels, channels)
            layers.append(layer)
        self.conv_bns = nn.Sequential(*layers)
        self.after_concat_layer = ConvBNRelu(channels + 3 + num_bits, channels)
        self.final_layer = nn.Conv2d(channels, 3, kernel_size=1)
        self.last_tanh = last_tanh
        self.tanh = nn.Tanh()
    def forward(self, imgs, msgs):
        msgs = msgs.unsqueeze(-1).unsqueeze(-1)# b l 1 1
        msgs = msgs.expand(-1,-1, imgs.size(-2), imgs.size(-1))# b l h w
        encoded_image = self.conv_bns(imgs)# b c h w
        concat = torch.cat([msgs, encoded_image, imgs], dim=1)# b l+c+3 h w
        im_w = self.after_concat_layer(concat)
        im_w = self.final_layer(im_w)
        if self.last_tanh:
            im_w = self.tanh(im_w)
        return im_w
```

图 3： WE 模型定义

之后我们再将水印图片进行提取，我们的 HiDDeN 解码器 W 模型如下图，将水印图像输入后，我们可以经过卷积层与平均池化后，直接用线性层输出我们的水印信息：

```python
class HiddenDecoder(nn.Module):
    """
    Decoder module. Receives a watermarked image and extracts the watermark.
    The input image may have various kinds of noise applied to it,
    such as Crop, JpegCompression, and so on. See Noise layers for more.
    """
    def __init__(self, num_blocks, num_bits, channels):
        super(HiddenDecoder, self).__init__()
        layers = [ConvBNRelu(3, channels)]
        for _ in range(num_blocks - 1):
            layers.append(ConvBNRelu(channels, channels))
        layers.append(ConvBNRelu(channels, num_bits))
        layers.append(nn.AdaptiveAvgPool2d(output_size=(1, 1)))
        self.layers = nn.Sequential(*layers)
        self.linear = nn.Linear(num_bits, num_bits)

    def forward(self, img_w):
        x = self.layers(img_w)# b d 1 1
        x = x.squeeze(-1).squeeze(-1)# b d
        x = self.linear(x)# b d
        return x
```

图 4： W 模型定义

为了加强我们的水印模型鲁棒性，上述 W 模型收到的水印图片信息是经过加噪处理过的。我们使用 data_augmentation 库，将水印图片进行数据增强，其默认使用的方法如下：

```
if params.data_augmentation == 'combined':
    data_aug = data_augmentation.HiddenAug(params.img_size,
                                           params.p_crop,
                                           params.p_blur,
                                           params.p_jpeg,
                                           params.p_rot,
                                           params.p_color_jitter,
                                           params.p_res).to(device)
```

图 5： 数据增强

我们每次编码解码器训练方法如下：

```
        # encoder
        deltas_w = self.encoder(imgs, msgs) # b c h w

        # scaling channels: more weight to blue channel
        if self.scale_channels:
            aa = 1/4.6 # such that aas has mean 1
            aas = torch.tensor([aa*(1/0.299), aa*(1/0.587), aa*(1/0.114)]).to(imgs.device)
            deltas_w = deltas_w * aas[None, :, None, None]

        # add heatmaps
        if self.attenuation is not None:
            heatmaps = self.attenuation.heatmaps(imgs) # b 1 h w
            deltas_w = deltas_w * heatmaps # # b c h w * b 1 h w -> b c h w
        imgs_w = self.scaling_i * imgs + self.scaling_w * deltas_w # b c h w

        # data augmentation
        if eval_mode:
            imgs_aug = eval_aug(imgs_w)
            fts = self.decoder(imgs_aug) # b c h w -> b d
        else:
            imgs_aug = self.augmentation(imgs_w)
            fts = self.decoder(imgs_aug) # b c h w -> b d

        fts = fts.view(-1, self.num_bits, self.redundancy) # b k*r -> b k r
        fts = torch.sum(fts, dim=-1) # b k r -> b k

        return fts, (imgs_w, imgs_aug)
```

图 6： 编码解码器训练

至此，我们实现了水印编码器的预训练，可以使用如下的代码进行测评：

```
# encode
img_w = encoder_with_jnd(img_pt, msg)
clip_img = torch.clamp(UNNORMALIZE_IMAGENET(img_w), 0, 1)
clip_img = torch.round(255 * clip_img)/255
clip_img = transforms.ToPILImage()(clip_img.squeeze(0).cpu())
```

图 7： 嵌入消息

```
# decode
ft = decoder(default_transform(clip_img).unsqueeze(0).to(device))
decoded_msg = ft > 0 # b k -> b k
accs = (~torch.logical_xor(decoded_msg, msg_ori)) # b k -> b k
print(f"Message: {msg2str(msg_ori.squeeze(0).cpu().numpy())}")
print(f"Decoded: {msg2str(decoded_msg.squeeze(0).cpu().numpy())}")
print(f"Bit Accuracy: {accs.sum().item() / params.num_bits}")
```

图 8： 提取消息

（三）水印模型微调

利用上述方法得到的预训练水印模型，我们将进行水印模型的微调。

我们设置了一个固定的序列号，以确保生成的载体信息能够提取该序列号。同时，加载了实验所需的 LDM 模型。

```
# 创建输出文件夹
if not os.path.exists(params.output_dir):
    os.makedirs(params.output_dir)
imgs_dir = os.path.join(params.output_dir, 'imgs')
params.imgs_dir = imgs_dir
if not os.path.exists(imgs_dir):
    os.makedirs(imgs_dir, exist_ok=True)

# 加载LDM auto-encoder models
print(f'调用LDM模型。 config:{params.ldm_config} weights:{params.ldm_ckpt}...')
config = OmegaConf.load(f"{params.ldm_config}")
ldm_ae: LatentDiffusion = libs_model.load_model_from_config(config, params.ldm_ckpt)
ldm_ae: AutoencoderKL = ldm_ae.first_stage_model
ldm_ae.eval()
ldm_ae.to(device)
```

图 5： 加载 LDM 模型

加载了训练好的水印注入模型后，我们会进行白化处理，通过多训练一个线性层来降低输入数据的相关性。这一步的目的是去除输入数据的冗余信息，尤其是在图像训练数据中，相邻像素之间存在强烈的相关性。其方法就是在水印提取模型后新增一个线性层来实现白化。

```
    with torch.no_grad():
        transform = transforms.Compose([...])
        loader = utils.get_dataloader(params.train_dir, transform, batch_size=16, collate_fn=None)
        ys = []
        for i, x in enumerate(loader):...
        ys = torch.cat(ys, dim=0)
        nbit = ys.shape[1]
        # 训练一个线性层，然后将训练好的线性层放到原模型的最后，达到白化效果
        mean = ys.mean(dim=0, keepdim=True)  # NxD -> 1xD
        ys_centered = ys - mean  # NxD
        cov = ys_centered.T @ ys_centered
        e, v = torch.linalg.eigh(cov)
        L = torch.diag(1.0 / torch.pow(e, exponent=0.5))
        weight = torch.mm(L, v.T)
        bias = -torch.mm(mean, weight.T).squeeze(0)
        linear = nn.Linear(nbit, nbit, bias=True)
        linear.weight.data = np.sqrt(nbit) * weight
        linear.bias.data = np.sqrt(nbit) * bias
        msg_decoder = nn.Sequential(msg_decoder, linear.to(device))
        torchscript_m = torch.jit.script(msg_decoder)
        params.msg_decoder_path = params.msg_decoder_path.replace(".pth", "_whit.pth")
        print(f'Creating torchscript at {params.msg_decoder_path}...')
        torch.jit.save(torchscript_m, params.msg_decoder_path)
```

图 6： 白化处理

在微调训练阶段，我们采取了和水印与训练相似的思路，加载了训练集并创建了所需的损失函数。通过提取图像编码器的中间层 vector，我们使用原始 decoder 和微调过的 decoder 对中间层 vector 进行解码，并提取水印。计算两个 decoder 的损失后，通过梯度下降进行微调训练，记录相关信息。

```
    for ii, imgs in enumerate(metric_logger.log_every(data_loader, params.log_freq, header)):
        imgs = imgs.to(device)
        keys = key.repeat(imgs.shape[0], 1)

        utils.adjust_learning_rate(optimizer, ii, params.steps, params.warmup_steps, base_lr)
        # 图像编码器提取中间层vector
        imgs_z = ldm_ae.encode(imgs)  # b c h w -> b z h/f w/f
        imgs_z = imgs_z.mode()

        # 分别用原始的decoder和微调过的decoder对中间层vector解码
        imgs_d0 = ldm_ae.decode(imgs_z)  # b z h/f w/f -> b c h w
        imgs_w = ldm_decoder.decode(imgs_z)  # b z h/f w/f -> b c h w

        # 提取水印
        decoded = msg_decoder(vqgan_to_imnet(imgs_w))  # b c h w -> b k

        # 计算原始的decoder和微调过的decoder各自的loss
        lossw = loss_w(decoded, keys)#提取的信息和key计算loss
        lossi = loss_i(imgs_w, imgs_d0)#两个decoder解码后的图像计算loss
        loss = params.lambda_w * lossw + params.lambda_i * lossi

        # 梯度下降
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

图 7： 模型微调

最后，我们进行了微调测试，使用测试集来评估我们方法的性能。我们尝试了各种攻击方法，以测试水印的鲁棒性，包括高斯模糊、逐像素丢失、裁剪和 JPEG 压缩。我们展示了我们的方法相对于现有的基于深度学习的隐写术的优越性，并且指出我们的方法能够产生鲁棒的盲水印。即使在存在图像失真的情况下，网络仍能够学习重构编码图像中的隐藏信息。

```python
def val(data_loader: Iterable, ldm_ae: AutoencoderKL, ldm_decoder: AutoencoderKL, msg_decoder: nn.Module,
        vqgan_to_imnet: nn.Module, key: torch.Tensor, params: argparse.Namespace):
    header = 'Eval'
    metric_logger = utils.MetricLogger(delimiter="  ")
    ldm_decoder.decoder.eval()
    for ii, imgs in enumerate(metric_logger.log_every(data_loader, params.log_freq, header)):

        imgs = imgs.to(device)

        imgs_z = ldm_ae.encode(imgs)  # b c h w -> b z h/f w/f
        imgs_z = imgs_z.mode()

        imgs_d0 = ldm_ae.decode(imgs_z)  # b z h/f w/f -> b c h w
        imgs_w = ldm_decoder.decode(imgs_z)  # b z h/f w/f -> b c h w

        keys = key.repeat(imgs.shape[0], 1)

        log_stats = {
            "iteration": ii,
            "psnr": libs_img.psnr(imgs_w, imgs_d0).mean().item(),
            # "psnr_ori": libs_img.psnr(imgs_w, imgs).mean().item(),
        }
```

图 8： 加载测试

```python
    'none': lambda x: x,
    'crop_01': lambda x: libs_img.center_crop(x, 0.1),
    'crop_05': lambda x: libs_img.center_crop(x, 0.5),
    'rot_25': lambda x: libs_img.rotate(x, 25),
    'rot_90': lambda x: libs_img.rotate(x, 90),
    'resize_03': lambda x: libs_img.resize(x, 0.3),
    'resize_07': lambda x: libs_img.resize(x, 0.7),
    'brightness_1p5': lambda x: libs_img.adjust_brightness(x, 1.5),
    'brightness_2': lambda x: libs_img.adjust_brightness(x, 2),
    'jpeg_80': lambda x: libs_img.jpeg_compress(x, 80),
    'jpeg_50': lambda x: libs_img.jpeg_compress(x, 50),
}
for name, attack in attacks.items():
    imgs_aug = attack(vqgan_to_imnet(imgs_w))
    decoded = msg_decoder(imgs_aug)  # b c h w -> b k
    diff = (~torch.logical_xor(decoded > 0, keys > 0))  # b k -> b k
    bit_accs = torch.sum(diff, dim=-1) / diff.shape[-1]  # b k -> b
    word_accs = (bit_accs == 1)  # b
    log_stats[f'bit_acc_{name}'] = torch.mean(bit_accs).item()
    log_stats[f'word_acc_{name}'] = torch.mean(word_accs.type(torch.float)).item()
for name, loss in log_stats.items():
    metric_logger.update(**{name: loss})
```

图 9： 测试类型设置

我们评估了一系列图像处理操作对生成的图像的鲁棒性，包括峰值信噪比、无攻击准确率、中心放大（0.1 倍和 0.5 倍）、翻转（25° 和 90° ）、缩放（0.3 倍和 0.7 倍）、调整明暗度（1.5 倍和 2 倍）、以及 JPEG 压缩（80 和 50）。这些操作涵盖了典型的几何学和光度学编辑，包括强裁剪、亮度偏移，以及裁剪、亮度偏移和 JPEG 压缩的组合。我们综合考虑了这些信息以评估图像处理算法的整体性能。

我们的评估旨在深入了解这些图像处理操作对算法鲁棒性的影响。峰值信噪比的变化考察了图像清晰度和噪声水平，无攻击准确率则反映了算法在没有外部

攻击的情况下的性能。中心放大、翻转和缩放等几何变换能够测试算法对空间变化的适应能力，而调整明暗度则模拟了不同光照条件下的处理情况。

同时，我们关注 JPEG 压缩对图像质量的影响，通过不同质量参数的压缩来模拟实际应用中可能遇到的不同压缩程度。这种综合的评估方法有助于全面了解算法在面对多样化的图像处理操作时的稳健性，为其在实际应用中的性能提供更全面的参考。

四、项目成果

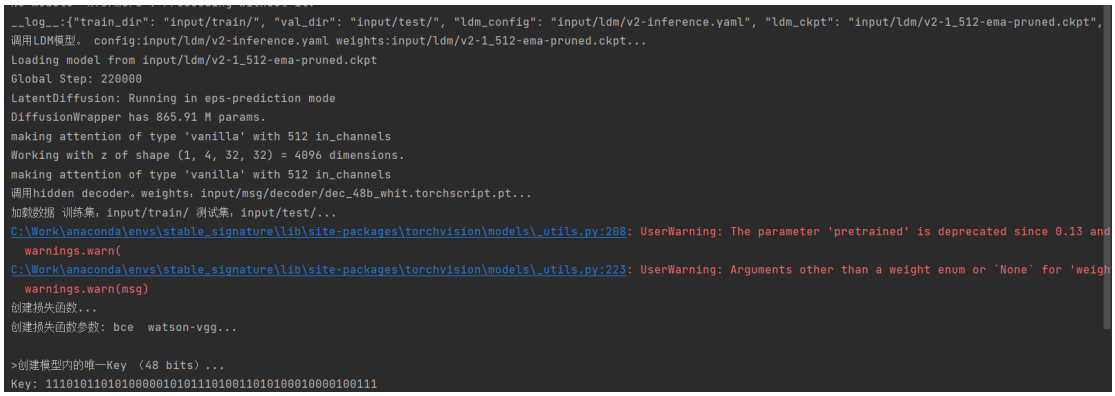


图 10 加载过程

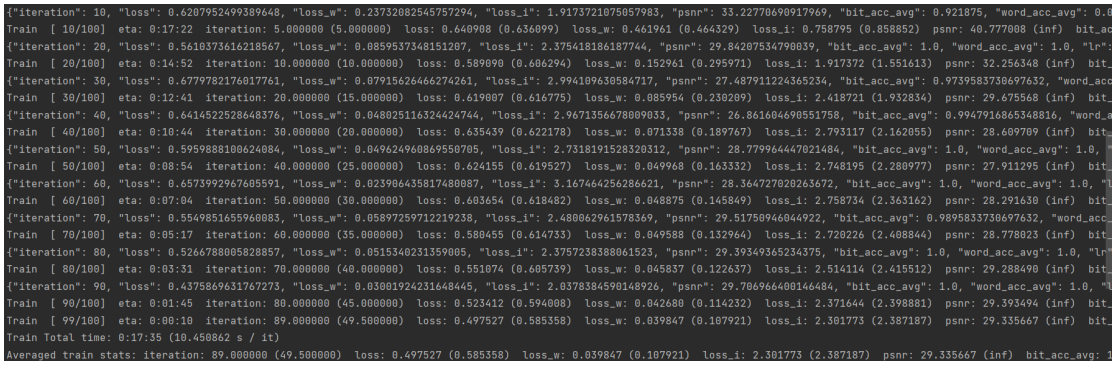


图 11 训练过程

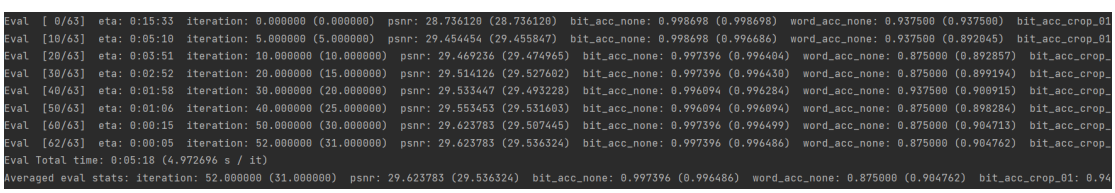
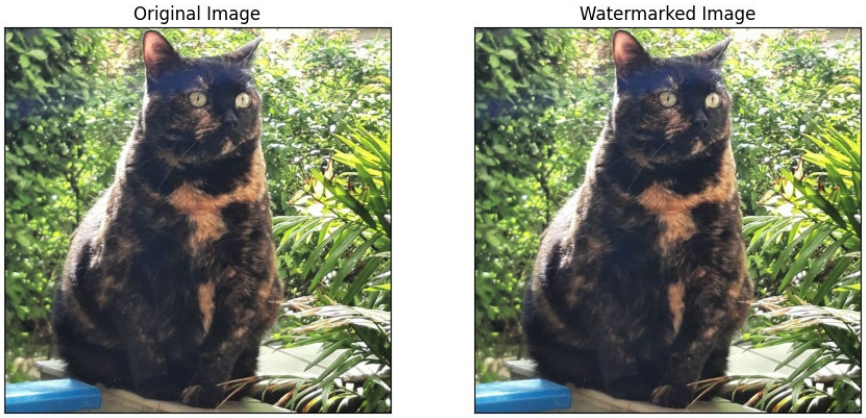


图 12 测试过程

通过以上过程，我们得到的结果如下：

```
PSNR: 35.11827402749864
Message: 00100100110100101111111011001011000000010110100
Decoded: 00100100110100101111111011001011000000010110100
Bit Accuracy: 1.0
```



可以看出，我们提取到的消息与原消息一致，且嵌入前后肉眼无法识别差异。

同时，我们使用测试集进行了测试，并尝试使用各种攻击方法对水印鲁棒性进行测试，可以看到得到了较好的结果：

测试集（比特数为单位）	准确率
峰值信噪比	29.53626278
无攻击准确率	0.997065161
中心放大 0.1 倍	0.938016719
中心放大 0.5 倍	0.992848894
翻转 25°	0.674954552
翻转 90°	0.458726039
缩放 0.3 倍	0.733010932

缩放 0.7 倍	0.984871052
调整明暗度 1.5 倍	0.988198593
调整明暗度 2 倍	0.967840627
jpeg 压缩 80	0.906684049
jpeg 压缩 50	0.850487784

图 13 训练结果

五、总结及展望

通过对 Latent Diffusion 模型的解码器进行快速微调，我们可以将水印嵌入它们生成的所有图像中。这并不改变扩散过程，使其与大多数基于 LDM 的生成模型兼容。这些水印是稳健的，人眼看不到，可以用来检测生成的图像并识别生成的用户，性能非常高。

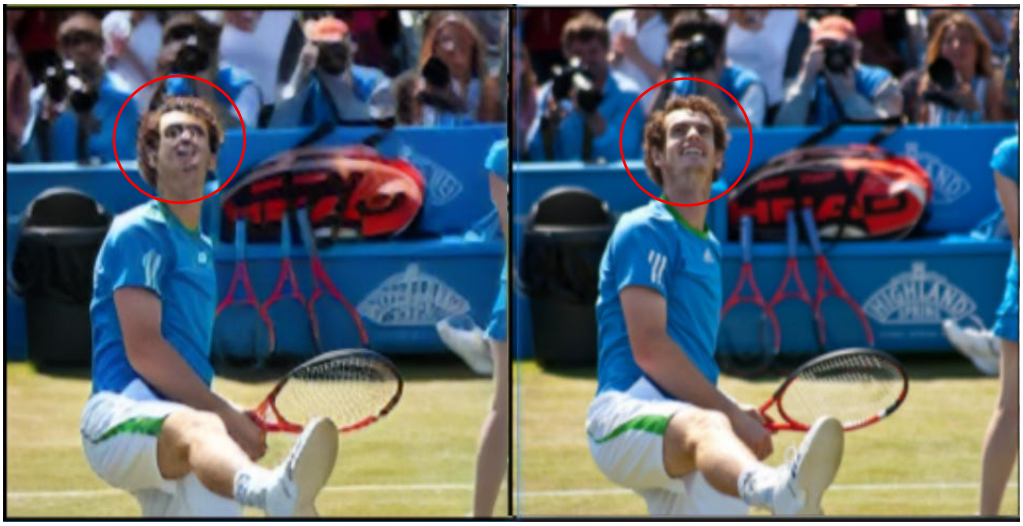
应用场景：

图像生成模型的公开发布已经产生了重要的社会影响。这些模型存在着被滥用的风险。然而，这项工作使研究人员和从业人员有可能分发他们的模型，同时可以追溯图像到生成它们的模型。因此，它可以被看作是对图像生成所带来的社会风险的一种缓解。

一般来说，水印可以提高内容的可追溯性。这种可追溯性可被用于政府识别生成式 AI 服务提供者使用的模型是否是主管部门备案的基础模型等。

此外，我们在进行训练的过程中，还遇到了以下的问题：

首先，我们的模型在处理人脸时表现不佳，容易被肉眼识别。具体效果如下：



其次，在利用 Stable Diffusion 模型实现生成式 AI 模型水印时由于显存不足，我们始终无法得到满意的结果。

```
  File "E:\college\大三上\信息隐藏\stablediffusion\ldm\modules\attention.py", line 272, in _forward
    x = self.attn1(self.norm1(x), context=context if self.disable_self_attn else None) + x
  File "C:\Work\anaconda\envs\stable_signature\lib\site-packages\torch\nn\modules\module.py", line 1130, in _call_impl
    return forward_call(*input, **kwargs)
  File "E:\college\大三上\信息隐藏\stablediffusion\ldm\modules\attention.py", line 177, in forward
    sim = einsum('b i d, b j d -> b i j', q, k) * self.scale
RuntimeError: CUDA out of memory. Tried to allocate 9.49 GiB (GPU 0; 11.99 GiB total capacity; 16.62 GiB already allocated; 0 bytes free; 25.17 GiB reserved in total by PyTorch)
```

参考文献

[1] TRAMÈR F, ZHANG F, JUELS A, et al. Stealing machine learningmodels via predictionapis[C]// Proceedings of the 25th{USENIX}Security Symposium ({USENIX}Security 16. 2016:601G618.

[2] XIE Chen-qi, ZHANG Bao-wen, YI Ping. Survey on Artificial Intelligence ModelWatermarking[J]. Computer Science, 2021, 48(7): 9-16.

[3]UCHIDA Y,NAGAI Y,SAKAZAWA S, et al. Embedding watermarks intodeep neural networks[C] // Proceedings of the 2017 ACMon International Conferenceon Multimedia Retrie-val. 2017:269G277.

[4]FAN L,NG K W,CHAN C S. Rethinking deep neuralnetwork ownership verification: Embedding passports to defeat ambiguityattacks[C] // Proceedings of the Advances in Neural InformationProcessing Systems. 2019:4714G4723.

[5]ADI Y,BAUM C,CISSE M, et al. Turning your weaks into astrength: Watermarking deep neural networks by backdooring [C] // Proceedings of the 27th {USENIX} Security Symposium. 2018:1615G1631.

[6] Zhang J, Dongdong C, etal. (2021, March 8). Deep Model Intellectual Property Protection via DeepWatermarking. ArXiv.Org. https://arxiv.org/abs/2103.04980

[7] Zhu J, Kaplan R, Johnson J, et al. Hidden: Hiding data with deep networks[C]//Proceedings of the European conference on computer vision (ECCV). 2018: 657-672.