StarGAN v2

Jonathon Chow [*] Ke Chern

（中国科学技术大学数学科学学院）

2022/7/11

1 目的

复现 Clova Research 研究团队的论文《StarGAN v2: Diverse Image Synthesis for Multiple Domains》[1]，该论文被收录于 CVPR 2020。下称“文章”。

2 Abstract 部分翻译

原文的 Abstract 部分为

A good image-to-image translation model should learn a mapping between different visual domains while satisfying the following properties:1)diversity of generated images and 2) scalability over multiple domains. Existing methods address either of the issues, having limited diversity or multiple models for all domains. We propose StarGAN v2, a single framework that tackles both and shows significantly improved results over the baselines. Experiments on CelebA- HQ and a new animal faces dataset (AFHQ) validate our superiority in terms of visual quality, diversity, and scalability. To better assess image-to-image tr- anslation models, we release AFHQ, high-quality animal faces with large inter- and intra-domain differences. The code, pretrained models, and dataset are ava- ilable at https://github.com/clovaai/stargan-v2.

以下是我们对上面的翻译：

[*]E-mail: jonathonchow23@gmail.com

一个理想的图像转换器应该兼容不同视觉域的图像，并且需要满足以下性质：1) 生成图像的多样性和 2) 多个视觉域的可扩展性。然而目前现有的方法都仅解决了其中一个问题，即要么生成的图像缺乏多样性，要么对所有视觉域采用各样模型。于是我们提出了 StarGAN v2 框架，它可以同时解决以上两个问题，并能够实现出比基础模型更加好效果。在 CelebAHQ 数据集和新动物面部数据集 (AFHQ) 上的测试验证了我们的框架在视觉质量、多样性和可扩展性方面的优势。为了更好地评估图像转换模型，我们发布了 AFHQ 数据集，它具有高质量的在视觉域间和域内有较大差异的动物面部数据。框架的代码、预训练模型以及数据集可以在 https://github.com/clovaai/stargan-v2 上获得。

3 文章的贡献与我们的创新

3.1 文章的贡献

文章中定义了“域”（domain）：可以分组为视觉上有独特风格的不同组图片；“风格”（style）：每组图片中的每张图片都有独特的外观和特征，这称之为风格。比如，可以根据性别将人分为男性和女性两个域，而人物的装扮，胡子，发型等特征可以视为风格。大概就是范围更大的可区分特征叫做域，范围小的叫做风格。一个理想的图像转换模型应该考虑域内的多样化的风格。但这种模型的设计和学习很困难，因为一种域内可以有大量的任意风格。这给我们的问题解决带来了很大的难度。

为了解决风格多样性问题，人们开展了大量关于图像对图像生成的研究。这些方法将低维的 latent code 注入到生成器（generator）中，该代码可以从标准高斯分布中随机抽样。它们特定于领域的解码器在生成图像时将 latent code 解释为各种风格的配方。但是，因为这些方法只考虑了两个域之间的映射，所以它们不能扩展到不断增加的域。例如，如果有 K 个域，这些方法需要训练 K(K-1) 个生成器来处理每个域之间的转换，从而限制了它们的实际使用。

为了解决可扩展性，一些研究提出了一些框架。StarGAN 是最早的模型之一，它使用单个生成器学习所有可行域之间的映射。生成器将域标签作为额外的输入，并学习将图像转换为相应的域。然而，StarGAN 仍然学习每个域的确定性映射，这没有捕获数据分布的多模态性质。这个限制是因为每个域都由一个预先确定的标签表示。

所以为了解决这两个问题，本文提出了 StarGANv2，它可以在多个域生成不同的图像。特别地，从 StarGAN 开始，用文章建议的域特定的 style code 替换它的域标签，该代码可以表示特定域的不同风格。为此，文章引入了两个模块，mapping network 和 style encoder。mapping network 学习将随机高斯噪声转换为 style code，而 style encoder 学习从给定的参考图像中提取 style code。考虑到多个域，两个模块都有多个输出分支，每个分

支都为特定域提供风格代码。最后，利用这些 style code，我们的生成器将学习如何成功地在多个领域合成不同的图像。

3.2 我们的创新

首先，我们利用课程所提供的的算力平台 Bitahub，对论文给出的代码进行了复现，并提取出训练的部分，增加了代码功能的专一性。

其次，为了实现可视化监测训练过程，我们在代码中加入了在 TensorBoard 画出 Loss 曲线的代码。

再次，对论文提供的 CelebAHQ 数据集进行了训练，利用训练保存的 checkpoint 进行测试，直观的观察训练效果的变化。

接下来，我们利用预训练模型，选择了一些有特色的图片，比如不同肤色、不同发型、不同人脸朝向，对训练效果进行评估。

最后，我们利用预训练模型复现了 CelebAHQ 数据集的评估，指标是 FID 和 LPIPS。

4 文章细节

4.1 框架

4.1.1 Generator

我们的 Generator G 将输入图像 $\mathbf{x}$ 转换为一个输出图像 $G(\mathbf{x}, \mathbf{s})$ 并且将其会反映出域内的 style code$\mathbf{s}$ 反映出域内年代风格的代码，这将会由 mapping network F 或 style encode E 提供。我们将利用自适应实例正常化方法 (AdaIN) 把 $\mathbf{s}$ 融入到 G。我们观察到 $\mathbf{s}$ 的目的是代表一个特定的域 (如 y) 的风格，可见没有必要向 G 提供 y 和允许 G 去综合所有域的图片。以下为 Generator 对 AFHQ 数据集的结构，4 个下采样块，4 个中间块以及 4 个上采样块。对 CelebA HQ 数据集，下采样以及上采样块数加 1。

4.1.2 Mapping network

给定一个 latent code $\mathbf{z}$ 和一个域 y，我们的 Mapping network F 会生成一个 style code $\mathbf{s} = F_y(\mathbf{z})$，这里 $F_y(\cdot)$ 表示 F 对应于域 y 的输出。F 由一个 MLP 组成，它有多个输出分支，为所有可行的域提供 style code。F 可以通过对 latent 向量 $\in \mathcal{Z}$ 和域 $y \in \mathcal{Y}$ 随机抽样从而产生多样的 style code。我们的多任务架构将允许 F 高效地学习所有域的 style 表示。以下为 Mapping network 的结构。

LAYER	RESAMPLE	NORM	OUTPUT SHAPE
Image $\mathbf{x}$	-	-	$256 \times 256 \times 3$
Conv1×1	-	-	$256 \times 256 \times 64$
ResBlk	AvgPool	IN	$128 \times 128 \times 128$
ResBlk	AvgPool	IN	$64 \times 64 \times 256$
ResBlk	AvgPool	IN	$32 \times 32 \times 512$
ResBlk	AvgPool	IN	$16 \times 16 \times 512$
ResBlk	-	IN	$16 \times 16 \times 512$
ResBlk	-	IN	$16 \times 16 \times 512$
ResBlk	-	AdaIN	$16 \times 16 \times 512$
ResBlk	-	AdaIN	$16 \times 16 \times 512$
ResBlk	Upsample	AdaIN	$32 \times 32 \times 512$
ResBlk	Upsample	AdaIN	$64 \times 64 \times 256$
ResBlk	Upsample	AdaIN	$128 \times 128 \times 128$
ResBlk	Upsample	AdaIN	$256 \times 256 \times 64$
Conv1×1	-	-	$256 \times 256 \times 3$

TYPE	LAYER	ACTVATION	OUTPUT SHAPE
Shared	Latent $\mathbf{z}$	-	16
Shared	Linear	ReLU	512
Shared	Linear	ReLU	512
Shared	Linear	ReLU	512
Shared	Linear	ReLU	512
Unshared	Linear	ReLU	512
Unshared	Linear	ReLU	512
Unshared	Linear	ReLU	512
Unshared	Linear	-	64

4.1.3 Style encoder

给定图像 $\mathbf{x}$ 及其对应的域 y，我们的 Style encoder E 提取 $\mathbf{x}$ 的风格代码 $\mathbf{s} = E_y(\mathbf{x})$，这里 $E_y(\cdot)$ 表示 E 对应于域 y 的输出。与 F 类似，我们的 Style encoder E 受益于多任务学习的设置。E 可以使用不同的参考图像产生不同的 style code。这允许了 G 合成一个反映出参考图像 $\mathbf{x}$ 的风格的输出图像。以下为 Style encoder 的结构，其中 D 和 K 分别代表输出维数和域的总数。

LAYER	RESAMPLE	NORM	OUTPUT SHAPE
Image $\mathbf{x}$	-	-	$256 \times 256 \times 3$
Conv1×1	-	-	$256 \times 256 \times 64$
ResBlk	AvgPool	-	$128 \times 128 \times 128$
ResBlk	AvgPool	-	$64 \times 64 \times 256$
ResBlk	AvgPool	-	$32 \times 32 \times 512$
ResBlk	AvgPool	-	$16 \times 16 \times 512$
ResBlk	AvgPool	-	$8 \times 8 \times 512$
ResBlk	AvgPool	-	$4 \times 4 \times 512$
LReLU	-	-	$4 \times 4 \times 512$
Conv4×4	-	-	$1 \times 1 \times 512$
LReLU	-	-	$1 \times 1 \times 512$
Reshape	-	-	512
Linear $\star$ K	-	-	D $\star$ K

4.1.4 Discriminator

我们的鉴别器 D 是一个多任务鉴别器, 它由多个输出分支组成。它的每个分支 D_y 会学习一个二分类来鉴别一个图片 $\mathbf{x}$ 是其域 y 的 real 图片还是由 G 生成的 fake 图片 $G(\mathbf{x}, \mathbf{s})$。Discriminator 的结构与上面的 Style encoder 相同。

4.2 训练目标函数

文章引入了几个 Loss：

1.Adversarial Loss

$$\mathcal{L}_{adv} = \mathbb{E}_{\mathbf{x},y}[\log D_y(\mathbf{x})] + \mathbb{E}_{\mathbf{x},\tilde{y},\mathbf{z}}[\log(1 - D_{\tilde{y}}(G(e\mathbf{x}, \widetilde{\mathbf{s}})))]$$

这是 GAN 的一般损失。

2.style reconstruction loss

$$\mathcal{L}_{adv} = \mathbb{E}_{\mathbf{x},\tilde{y},\mathbf{z}}[\|\widetilde{\mathbf{s}} - E_{\tilde{y}}(G(\mathbf{x}, \widetilde{\mathbf{s}}))\|_1]$$

style reconstruction loss 要求转换后的图片也能编码出一致的 style code，也就是强制要求 Generator G 去使用我们生成的 style code。

3.diversity sensitive loss

$$\mathcal{L}_{ds} = \mathbb{E}_{\mathbf{x},\tilde{y},\mathbf{z_1},\mathbf{z_2}}[\|G(\mathbf{x}, \widetilde{\mathbf{s_1}}) - G(\mathbf{x}, \widetilde{\mathbf{s_2}})\|_1]$$

diversity sensitive loss 要求尽可能使合成图像多样性高，也就是我们要求 Generator G 去生成更多样化的图片。

4.cycle consistency loss

$$\mathcal{L}_{cyc} = \mathbb{E}_{\mathbf{x},y,\tilde{y},\mathbf{z}}[\|\mathbf{x} - G(G(\mathbf{x}, \widetilde{\mathbf{s}}), \hat{\mathbf{s}})\|_1]$$

cycle consistency loss 保证两次转换后，图像能复原；这也是为了保证生成的图片能够正确地保持原域不变的特征。

我们最终的总损失函数为：

$$\min_{G,F,E} \max_{D} \mathcal{L}_{adv} + \lambda_{sty}\mathcal{L}_{sty} - \lambda_{ds}\mathcal{L}_{ds} + \lambda_{cyc}\mathcal{L}_{cyc}$$

4.3 训练过程

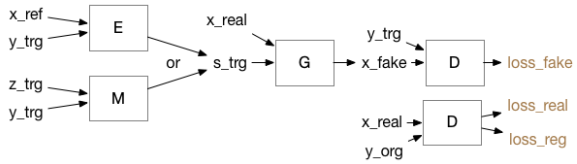
这里用图大概展示一下训练的流程。其中 G,M,S,D 分别代表 Generator、Mapping network、Style encoder、Discriminator。

4.3.1 训练 Discriminator

计算各个 loss 后，更新 D 的参数。

4.3.2 训练 Generator

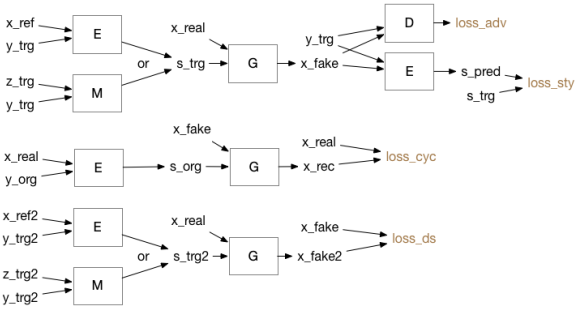
计算各个 loss 后，更新 E、M、G 的参数。



x_real: 源domain (y_org) 的真实图像

x_ref: 目标domain (y_trg) 的参考图像，用于生成目标domain的style code (s_trg)

z_trg: 随机噪声，用于生成目标domain的style code (s_trg)



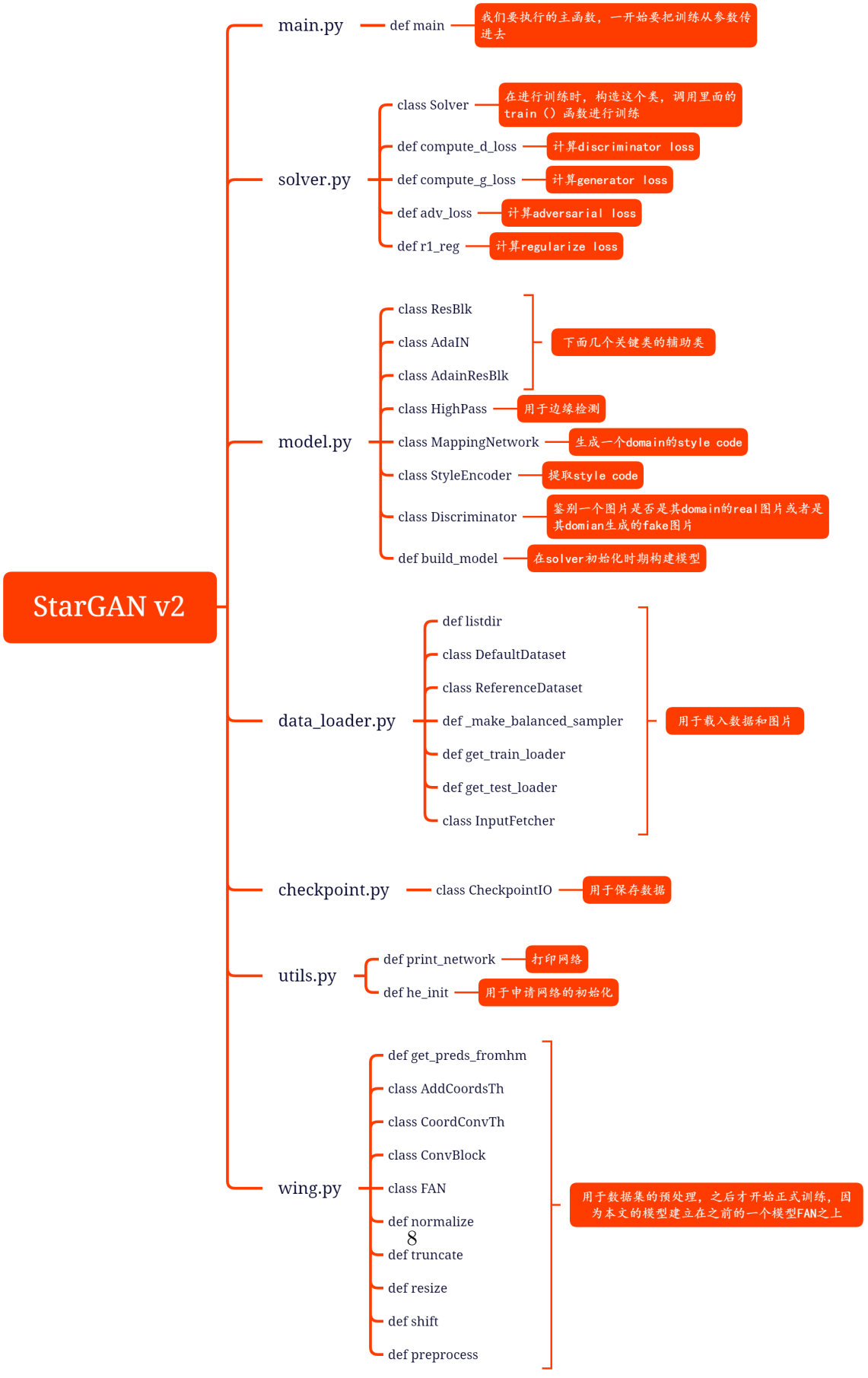
x_real: 源domain (y_org) 的真实图像

x_ref: 目标domain (y_trg) 的参考图像，用于生成目标domain的style code (s_trg)

z_trg: 随机噪声，用于生成目标domain的style code (s_trg)

5 代码结构描述

我们将官方代码改成得更加精简，只留下了训练数据集的部分，也就是去掉了 sample mode 和 eval mode。我们在算力平台 Bitahub 上执行的就是我们如此更改后的版本。下面我们描述一下各.py 文件的代码结构，以及一些代码细节。


StarGAN v2
main.py
def main
我们要执行的主函数，一开始要把训练从参数传进去
solver.py
class Solver
在进行训练时，构造这个类，调用里面的train（）函数进行训练
def compute_d_loss
计算discriminator loss
def compute_g_loss
计算generator loss
def adv_loss
计算adversarial loss
def r1_reg
计算regularize loss
model.py
class ResBlk
class AdaIN
class AdainResBlk
下面几个关键类的辅助类
class HighPass
用于边缘检测
class MappingNetwork
生成一个domain的style code
class StyleEncoder
提取style code
class Discriminator
鉴别一个图片是否是其domain的real图片或者是其domian生成的fake图片
def build_model
在solver初始化时期构建模型
data_loader.py
def listdir
class DefaultDataset
class ReferenceDataset
def _make_balanced_sampler
def get_train_loader
def get_test_loader
class InputFetcher
用于载入数据和图片
checkpoint.py
class CheckpointIO
用于保存数据
utils.py
def print_network
打印网络
def he_init
用于申请网络的初始化
wing.py
def get_preds_fromhm
class AddCoordsTh
class CoordConvTh
class ConvBlock
class FAN
def normalize
def truncate
def resize
def shift
def preprocess
用于数据集的预处理，之后才开始正式训练，因为本文的模型建立在之前的一个模型FAN之上

5.1 main.py

```
def main(args)
```

函数输入的参数 args 为 python 标准库推荐的“命令行解析模块”command-line parsing module，它可以指定程序运行不同的设置。

```
cudnn.benchmark = True
```

这句代码对模型结构以及输入大小固定的算法有加速作用，其原理大概可以这样描述：当该标识位设置为 True 时，cudnn 库会根据不同的模型设置与输入大小找出最优的卷积算法，但如果模型是变化的，则每次都要重新优化找到最佳算法（候选算法包括有 GEMM，FFT 等），反复寻找反而会浪费时间；当该标识位设置为 False 时，cudnn 库会启发式地选择卷积算法，执行速度不一定最快。

```
loaders = Munch(...)
```

Munch 类能实现属性风格的访问，类似于 Javascript，同时属于 Dictionary 的子类，有字典的所有特性。这里定义的 Munch 对象的 loaders 中包含了 src、ref 以及 val 的 dataloader，便于之后的训练过程调用。

5.2 model.py

```
class Discriminator(nn.Module)
```

鉴别器 Discriminator 有多个输出分支, 每个分支对应一个 domain，每个分支输出一个值，即属于该 domain 的概率，最终 D 的输出为 x 是否属于 domain y 的概率。

```
class StyleEncoder(nn.Module)
```

其结构与鉴别器相同，区别在于结构图中最后一个 Linear 层，鉴别器是用一个 Conv1x1 实现，Style Encoder 则是用多个 nn.Linear() 代替。

```
class MappingNetwork(nn.Module)
```

Mapping network 有 8 层 MLP，其输入为随机噪声 z 以及目标 domain y，输出为对应的风格编码 s 。

5.3 solver.py

```
self.device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

用于表示 torch.Tensor 在或者将会被分配到哪个设备上。

```
# compute moving average of network parameters
moving_average(nets.generator, nets_ema.generator, beta=0.999)
moving_average(nets.mapping_network, nets_ema.mapping_network, beta=0.999)
moving_average(nets.style_encoder, nets_ema.style_encoder, beta=0.999)
```

这个函数输入 model，是真正在训练的模型（参数一直更新）；model_test (XXX_ema) 为滑动平均值，torch.lerp() 计算结果为 beta * (model_test- model)+ model。

```
self.nets, self.nets_ema = build_model(args)
```

在这个函数中定义了所有网络，包括 Generator，MappingNetwork，StyleEncoder，Discriminator。这里的 copy.deepcopy() 为深拷贝，对模型 generator 创建一个独立的复制 generator_ema。该复制用于之后训练时对模型参数做滑动平均, 但是可惜的是，这里文章没有解释原因。

这句代码出现在初始化 solver 的时候。其中其实还定义了一个预训练好的人脸关键点模型 FAN（ICCV2019 AdaptiveWingLoss），其作用为产生关键部位的 mask，使得原图像 mask 区域在转换后仍能得以保留。

```
setattr(self, name, module)
```

这个 setattr 用于设置属性的值。self.nets 为字典对象，里面包含了各个模型网络，我们需要直接使各个模型为 Solver 类的属性，以使得后续可使用 self.to(device) 将模型参数

分配到 GPU 上。经过测试，不加 setattr 确实对分配到 GPU 有影响。原因在于 self.to() 只能将 float 型参数移动到 GPU，无法移动字典类型。另外，nn.Module 的.to() 是 inplace 操作，而 Tensor 的.to() 是在拷贝上操作。

nn.Module

nn.Module 类中.named_children() 返回子模块名及子模块本身；.apply(fn) 将 fn 迭代地应用到该模块及其子模块，最典型的用法就是用于模型初始化。

def compute_d_loss()

用于训练鉴别器，分为两部分。以 latent code 为输入以及以 refenrence 为输入。

其中的.requires_grad_() 表示让 autograd 开始记录该 Tensor 上的 operation。对 x_real 进行该操作是为了后续计算 r1_reg 需要求 out 对 x_real 的导数。

其中的 with torch.no_grad() 下的内容不计算梯度，这样做是因为当前只训练鉴别器，除鉴别器外的其他模型无需产生梯度用于反向传播，故可以减少计算以显存占用。

def compute_g_loss()

这个函数类似上面的函数，用于训练生成器。同样分两部分，以 latent code 和 refenrence 为输入。值得注意的是，在以 latent_code 为输入时，优化了 generator、mapping_network 以及 style_encoder；但在以 reference img 为输入时，只优化了 generator，为什么不优化 style_encoder？文章同样没有解释。

def r1_reg()

这个函数源自 zero-centered gradient penalty [2]，其公式如下，即鉴别器输出对真实图像的导数的模的平方：

$$R_1(\psi) = \frac{\gamma}{2} E_{p\mathcal{D}(x)} \left[\|\nabla D_\psi(x)\|^2 \right]$$

5.4 checkpoint.py

其中定义的 CheckpointIO 类用于保存、加载模型。其中初始化参数 **kwargs 表示输入为多个关键词的参数（可以理解成字典），CheckpointIO 中对应输入为 Munch 类（属于字典类）的 self.nets 以及 self.optims。这个用法可增加代码灵活性。

5.5 data_loader.py

```
class InputFetcher
```

try 部分用于不断从 loader 中取出数据。第一次进入 try，因为还没定义迭代器，所以会产生 AttributeError，进入 except 部分定义 self.iter；当取完迭代器中所有数据后，再次进入 try 取数据，会产生 StopIteration 而进入 except 重新加载 loader 迭代器。含有 ___next___() 函数的对象都可以看成一个迭代器。可以使用 next() 依次访问其中的内容。

```
def get_train_loader()
```

训练数据的预处理包括:

1. 随机裁剪后缩放到 256 固定大小；
2. 随机水平翻转；
3. 像素归一化（均值方差为 0.5）

所以在这个函数里，对应 source 的 dataset 函数使用 torchvision.datasets.ImageFolder 产生。数据集 CelebA HQ 的文件夹包括 female 和 male 两个文件夹，在文件夹下为对应的文件，因而该 dataset 函数返回为 (x,y) 对应取出来的图像以及其对应的 domain 标签。2、对应 source 的 dataset 函数使用 class ReferenceDataset 产生，它将返回两张参考图像（ref）以及其对应的 label，这是为了后续训练生成器时，计算 diversity sensitive loss。

6 对官方代码的修改

6.1 TensorBoard 部分

为了实现可视化检测，我们在函数 train() 中加入了用 TensorBoard 记录 Loss 的代码：

```
from torch.utils.tensorboard import SummaryWriter
writer = SummaryWriter(log_dir="/data/JONATHONCHOW/celeba_hq/runs/result",
flush_secs=120)
# write all the Loss values into tensorboard
for key, value in all_losses.items():
writer.add_scalar('%s' % key, value, i)
```

6.2 删减部分

我们将所有与 mode sample 和 mode eval 的函数与参数都删减掉了，从而训练部分的代码结构更加清晰。

6.3 参数修改部分

我们修改了原代码中的以下参数：1. 各种文件的读取和保存的地址:

```
# directory for training
parser.add_argument('–train_img_dir', type=str,
default='/data/JONATHONCHOW/celeba_hq/train',
help='Directory containing training images')
parser.add_argument('–val_img_dir', type=str,
default='/data/JONATHONCHOW/celeba_hq/val',
help='Directory containing validation images')
parser.add_argument('–sample_dir', type=str,
default='/data/JONATHONCHOW/celeba_hq/expr/samples',
help='Directory for saving generated images')
parser.add_argument('–checkpoint_dir', type=str,
default='/data/JONATHONCHOW/celeba_hq/expr/checkpoints',
help='Directory for saving network checkpoints')
# face alignment
parser.add_argument('–wing_path', type=str,
default='/data/JONATHONCHOW/celeba_hq/expr/checkpoints/wing.ckpt')
parser.add_argument('–lm_path', type=str,
default='/data/JONATHONCHOW/celeba_hq/expr/checkpoints/celeba_lm_mean.npz')
```

2. 将 ds_iter 从 100000 改为 500000, 将 total_iters 从 100000 改为 500000，这是训练的迭代次数:

```
parser.add_argument('–ds_iter', type=int, default=500000,
help='Number of iterations to optimize diversity sensitive loss')
parser.add_argument('–total_iters', type=int, default=500000,
help='Number of total iterations')
```

3. 将 batch_size 从 8 改为 4，不然平台会显示“CUDA OUT OF MEMORY”:

```
parser.add_argument('–batch_size', type=int, default=4,
help='Batch size for training')
```

4. 将 save_every 从 10000 改为 5000，这是相邻两次保存 Checkpoint 的迭代次数间隔：

```
parser.add_argument('–save_every', type=int, default=5000)
```

5. 将保存 Checkpoint 的文件名格式从 {:6d} 改为 {}(我们发现在 Windows 系统上跑代码时会将冒号“:”识别为路径从而因找不到路径而报错)：

```
CheckpointIO(ospj(args.checkpoint_dir, '{}_nets.ckpt'),
data_parallel=True, **self.nets),
CheckpointIO(ospj(args.checkpoint_dir, '{}_nets_ema.ckpt'),
data_parallel=True, **self.nets_ema),
CheckpointIO(ospj(args.checkpoint_dir, '{}_optims.ckpt'),
**self.optims)]
```

6. 在以下代码中加了一个“False”：

```
module.module.load_state_dict(module_dict[name], False)
```

7 训练过程描述

配置虚拟环境

```
conda create -n stargan-v2 python=3.6.7
conda activate stargan-v2
conda install pytorch==1.5.0 torchvision==0.6.0 cudatoolkit=10.1 -c pytorch
conda install -c conda-forge x264
conda install -c conda-forge ffmpeg=4.0.2
pip install opencv-python==4.1.2.30 ffmpeg-python==0.2.0 scikit-image==0.16.2
pip install pillow==7.0.0 scipy==1.2.1 tqdm==4.43.0 munch==2.5.0
```

启动代码

```
python main.py –mode train –num_domains 2 –w_hpf 1
–lambda_reg 1 –lambda_sty 1 –lambda_ds 1 –lambda_cyc 1
–train_img_dir data/celeba_hq/train
–val_img_dir data/celeba_hq/val
```

7.1 BitaHub 训练

建立项目 StarGAN_v2：Python 3.6，PyTorch 1.4，celeba_hq。

创建数据集 celeba_hq：分批手动传输数据集压缩包，并解压缩文件，然后移动文件位置，重构数据集。

远程调试

```
ssh -i application_1657005726244_2313.txt -p 10001 root@10.11.0.11
```

启动代码

```
pip install pillow==7.0.0 scipy==1.2.1 tqdm==4.43.0 munch==2.5.0 python main.py –mode train –num_domains 2 –w_hpf 1
–lambda_reg 1 –lambda_sty 1 –lambda_ds 1 –lambda_cyc 1
–train_img_dir ../data/JONATHONCHOW/celeba_hq/train
–val_img_dir ../data/JONATHONCHOW/celeba_hq/val
```

7.2 测试

剪裁代码

```
python main.py –mode align
–inp_dir assets/representative/custom/female
–out_dir assets/representative/celeba_hq/src/female
```

人物启动代码

```
python main.py –mode sample –num_domains 2 –resume_iter 100000 –w_hpf 1
–checkpoint_dir expr/checkpoints/celeba_hq
–result_dir expr/results/celeba_hq
–src_dir assets/representative/celeba_hq/src
–ref_dir assets/representative/celeba_hq/ref
```

动物启动代码

```
python main.py –mode sample –num_domains 3 –resume_iter 100000 –w_hpf 0
–checkpoint_dir expr/checkpoints/afhq
–result_dir expr/results/afhq
–src_dir assets/representative/afhq/src
–ref_dir assets/representative/afhq/ref
```

7.3 评估

启动代码

```
python main.py –mode eval –num_domains 2 –w_hpf 1
–resume_iter 100000
–train_img_dir data/celeba_hq/train
–val_img_dir data/celeba_hq/val
–checkpoint_dir expr/checkpoints/celeba_hq
–eval_dir expr/eval/celeba_hq
```

8 测试结果

以下我们展示我们在训练到不同迭代次数 iter 时保存的模型所生成的图像结果：

图 1: iter=1000

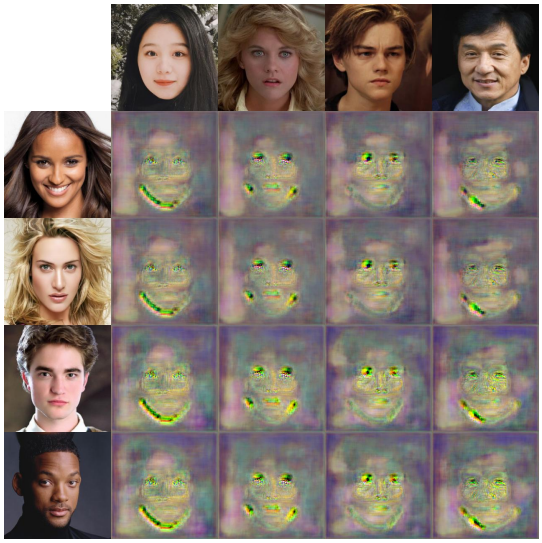


图 2: iter=5000

图 3: iter=10000

图 4: iter=20000

图 5: iter=40000

图 6: iter=80000

图 7: iter=100000

图 8: CelebA-HQ，iter=100000

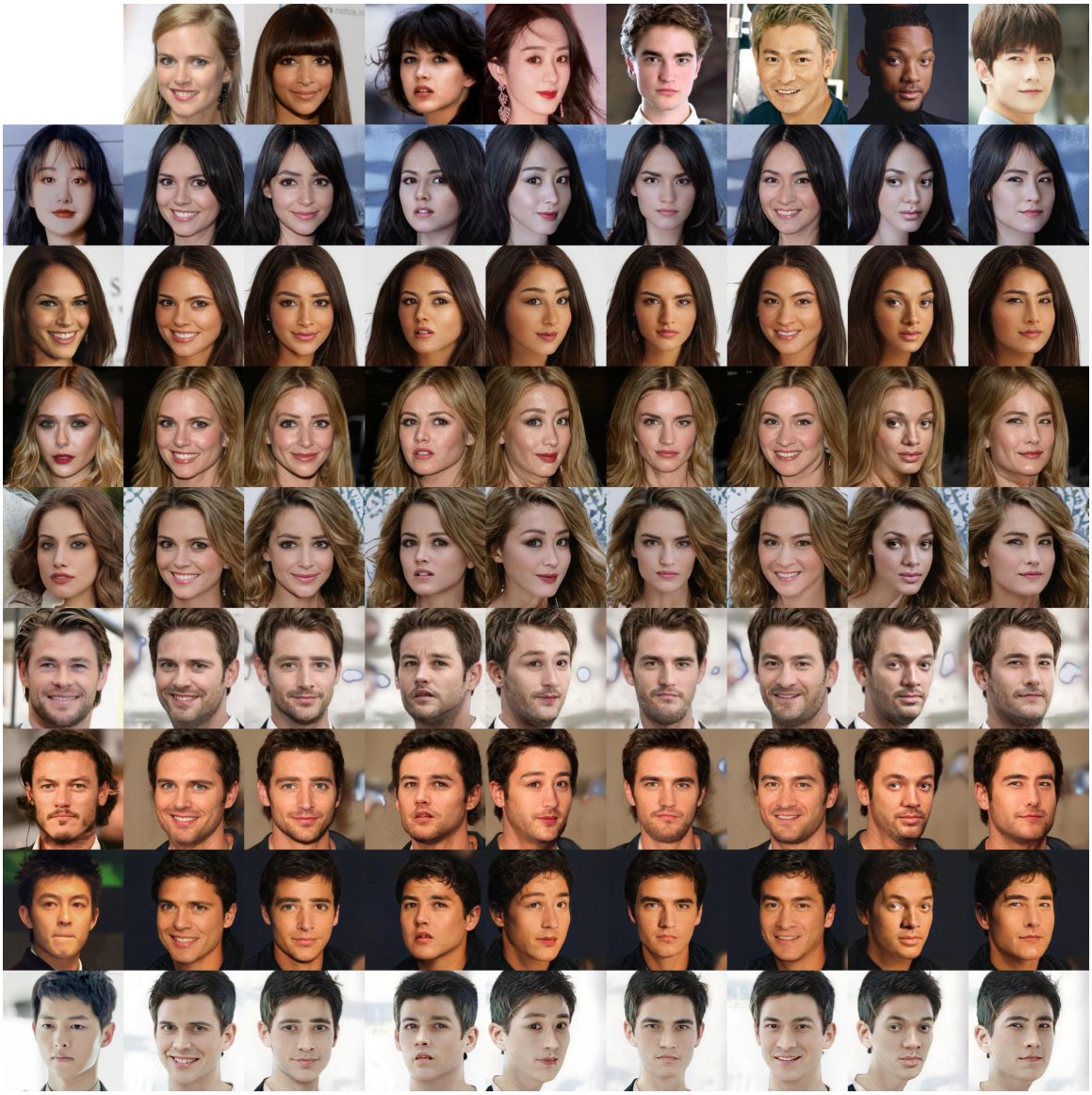


图 9: AFHQ，iter=100000

以下为 TensorBoard 中各参数随迭代次数的走势（迭代次数为 95000）。

图 10: latent

(1).png (1).bb

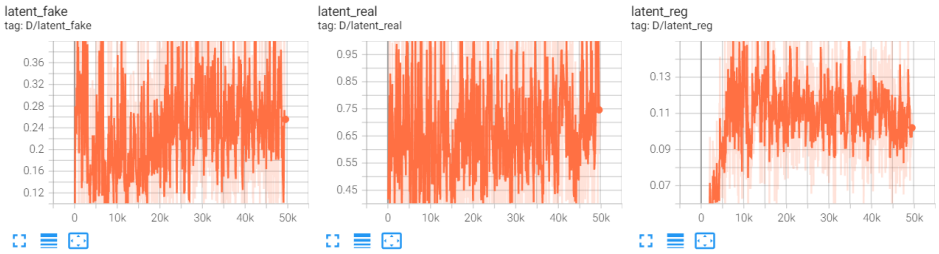


图 11: ref

(2).png (2).bb

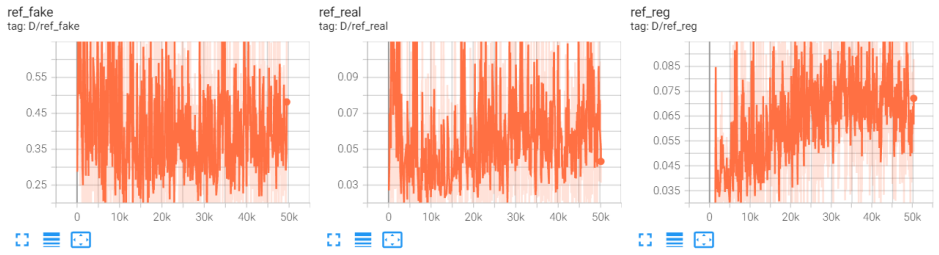


图 12: 各 loss

(3).png (3).bb

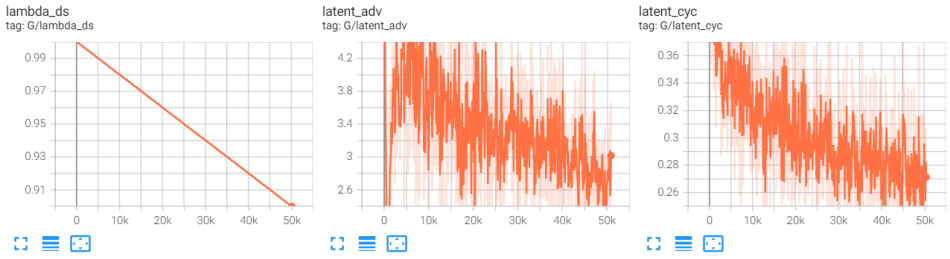


图 13: 各 loss

(4).png (4).bb

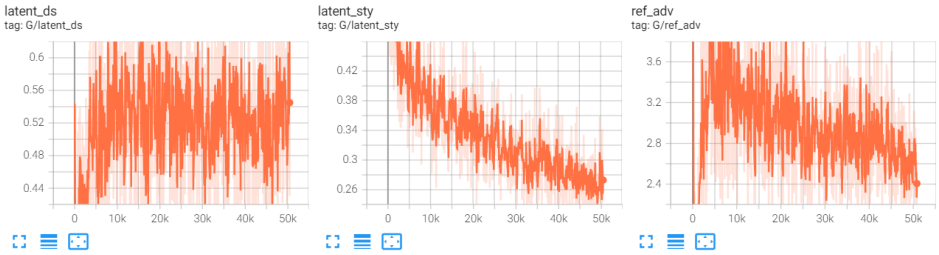
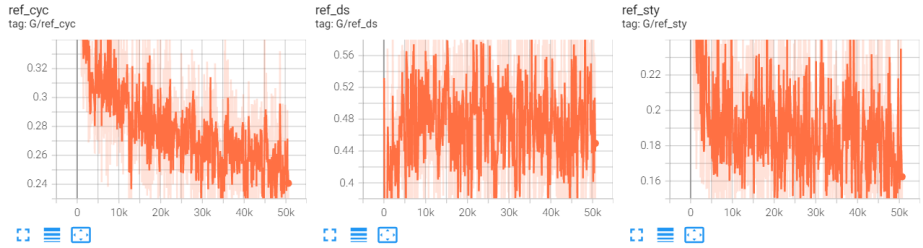


图 14: 各 loss

(5).png (5).bb



从上面的图线可以看出，latent 和 ref 中 fake 和 real 图片属于其对应 domain 的概率近乎为白噪声，这一点我们似乎没办法解释。但是后面的各种文章中定义的 loss 都明显呈现下降趋势，但是大范围内起伏和波动有点大。

8.1 FID 和 LPIPS

文章中提到了 2 个评估指标：

1.Frechet inception distance (FID)

FID 用于衡量真实图像分布与合成图像分布之间的差异，具体是指不同图像在 InceptionV3 分类器的高维特征空间中分布密度的差异，该差异用 Frechet Distance 进行计算，FID 值越小越好。Fréchet Distance 计算公式如下：

$$d^2((\boldsymbol{m}, \boldsymbol{C}), (\boldsymbol{m}_\omega, \boldsymbol{C}_\omega)) = \|\boldsymbol{m} - \boldsymbol{m}_\omega\|_2^2 + Tr(\boldsymbol{C} + \boldsymbol{C}_\omega - 2(\boldsymbol{C}\boldsymbol{C}_\omega)^{1/2})$$

2.Learned perceptual image patch similarity (LPIPS)

LPIPS 用于衡量影像的多样性，其值越大代表多样性越高。其具体计算方法为：

$$d(x, x_0) = \sum_l \frac{1}{H_l W_l} \sum_{h,w} \|w_l \odot (\hat{y}^l_{h,w} - \hat{y}^l_{0,h,w})\|_2^2$$

我们通过命令行也计算了我们的训练结果的这两个评估指标。

表 1: 评估指标（celeba-hq）

	FID (latent)	LPIPS (latent)	FID (reference)	LPIPS (reference)	Elapsed time
Official	13.73 ± 0.06	0.4515 ± 0.0006	23.84 ± 0.03	0.3880 ± 0.0001	49min 51s
us	13.1756	0.452713	23.7035	0.389031	152min 8s

从上面的对比我们可以看到个指标我们的训练在前 2 为有效小数上都一样。遗憾的是我们评估的耗时非常长。

9 本工作的分工

本次任务由 Jonathon Chow 和 Ke Chern 共同完成。

10 代码库链接

https://github.com/JONATHONCHOW/stargan-v2_train
（其中我们也生成了改动报告）

参考文献

[1] Y. Choi, Y. Uh, J. Yoo, and J.-W. Ha, "StarGAN v2: Diverse Image Synthesis for Multiple Domains," *arXiv e-prints*, p. arXiv:1912.01865, Dec. 2019.

[2] L. Mescheder, A. Geiger, and S. Nowozin, "Which Training Methods for GANs do actually Converge?" *arXiv e-prints*, p. arXiv:1801.04406, Jan. 2018.