2021-6-13

二次元头像生成器

多媒体技术小组项目汇报

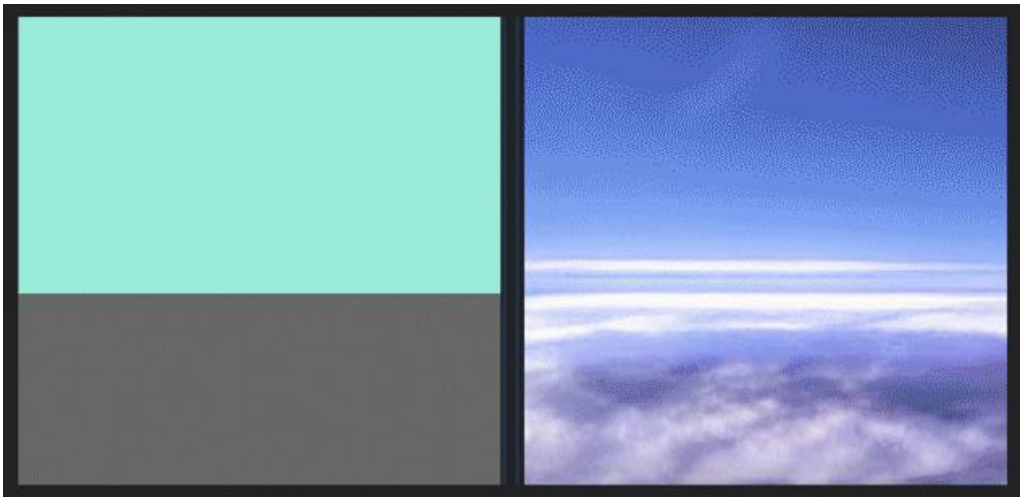
watermellye / gzy02

目录

二次元头像生成器

选题背景

第一次听到 GAN 这个词，是在视频网站上看到一个介绍 AI 作图生成器的视频。



人只需在空白的画布上指定天空、草地等元素，并勾勒出物体的轮廓，便可生成一张好似实景拍摄的风景画。甚至，还可以改变色调和主题。

视频链接：https://www.bilibili.com/video/BV1d4411G76N

展示网站：https://www.nvidia.com/en-us/research/ai-playground/?ncid=so-twi-nz-92489&fbclid=IwAR3cf-s5JzcVLxMRLAL4pi9DH4zT_OKRL4U_6kXio3FqP1M-vEuiriHBHl0

论文：https://arxiv.org/abs/1903.07291

这个视频可以说在当时颠覆了我对人工智能的认知，好似变魔术般的实现了真正的智能。大二的我们自认为，尝试复现这样的算法可谓不自量力。但以多媒体技术这门课为契机，我们决定挑战一番。

经过论文阅读和资料搜集得知，该功能的核心算法为 GAUGAN，是生成对抗网络 GAN 的一种应用。最终，我们选定了本次的主题：使用 GAN 生成二次元头像。

实验流程

为了达到逐步深入的效果，本次报告分享中，我将采用时间线的方式，逐步介绍实现过程。报告的前半段简要介绍环境搭建、训练集搜集以及网络调参；后半段介绍我们对该网络的核心算法的剖析，以及基于此而自己重新编写的网络。介绍过程中，将会对每个优化阶段给出详细的成果图片。

算法简介

这一部分简要概括我们所做的工作的基本原理，即 GAN 到底是什么。这里以我们小组研究的生成图片为例进行说明。

以下内容部分参考于：https://zhuanlan.zhihu.com/p/24767059

假设我们有两个网络，G(Generator)和 D(Discriminator)。 正如它的名字所暗示的那样，它们的功能分别是：

- G 是一个生成图片的网络，它接收一个随机的噪声 z，通过这个噪声生成图片，记做 G(z)。
- D 是一个判别网络，判别一张图片是不是“真实的”。它的输入参数是 x，x 代表一张图片，输出 D(x)代表 x 为真实图片的概率，如果为 1，就代表 100%是真实的图片，而输出为 0，就代表不可能是真实的图片。

在训练过程中，生成网络 G 的目标就是尽量生成真实的图片去欺骗判别网络 D。而 D 的目标就是尽量把 G 生成的图片和真实的图片分别开来。这样，G 和 D 构成了一个动态的“博弈过程”。 最后博弈的结果是什么？在最理想的状态下，G 可以生成足以“以假乱真”的图片 G(z)。对于 D 来说，它难以判定 G 生成的图片究竟是不是真实的，因此 D(G(z)) = 0.5。

GAN 的开山之作：Ian Goodfellow - Generative Adversarial Networks（arxiv：https://arxiv.org/abs/1406.2661）

环境搭建、运行例程

决定该选题后，我们开始找程序和训练集跑结果。

我们从一篇国内博客中找到了一个最简单的生成二次元头像的 GAN 算法的程序。

https://blog.csdn.net/qq_43815869/article/details/109121772

该博客附带训练集，为 1w6 张 96×96 分辨率的图像。

在搭建好 python 环境、装载 PyTorch 依赖后，我们使用 CPU 跑了一个晚上，得到了第一份结果：

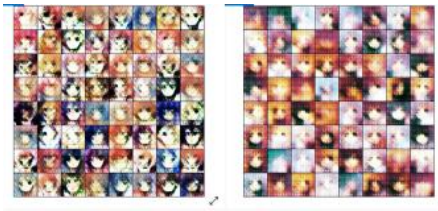
 *CPU 80 轮

训练速度太慢，经过了解，首要事项是改用 GPU 提高速度。

由于项目处于初期，模型和数据都有大量修改，不适宜租赁服务器来运行；同时由于现在正处于矿潮，实在买不到显卡。最终，我们从同学临时借用到一块宝贵的 2070 显卡。

环境：python=3.8.3 / Cuda=11.3 with Cudnn / pyTorch for Cuda 11.1

GPU/CPU 结果对比

 *29 轮 左边为 GPU 右边为 CPU*

安装好 Cuda、装载 Cudnn 后，我们真正跑出了一份结果：

 *900 轮，已收敛

可以看到，图像充满扭曲，质感粗糙，远未达到预期。

容易想到，接下来有两个优化方向：从模型的角度，和从训练集的角度。因为我们对模型的了解还不足，我们决定由简入繁，先处理训练集，也期望从处理的过程中总结出一定的经验。

训练集优化

更多的图像！

首先，我们怀疑例程中的图像数量不足。为此，我们爬取了多个网站的动漫图，使用 python 和 opencv 的识别头像模块得到 14w 张图。

爬虫代码见文件 webspider.py（以 konachan.net 为例）

截头代码见文件 Capture_head_portrait.py

尝试过的所有二次元图片分享网站链接：

http://safebooru.donmai.us/

http://konachan.net/

https://www.pixiv.net/

python-opencv 头部识别：

https://github.com/nagadomi/lbpcascade_animeface

 *270 轮，已收敛。

可以看出，图像效果有所提升。64 张图中已经有少许完成度较高的图像。但是，图像总体的根本性缺陷仍然没有改善。

只训练一个人会发生什么？

首先我们尝试，只训练一个角色的不同角度和表情的头部。

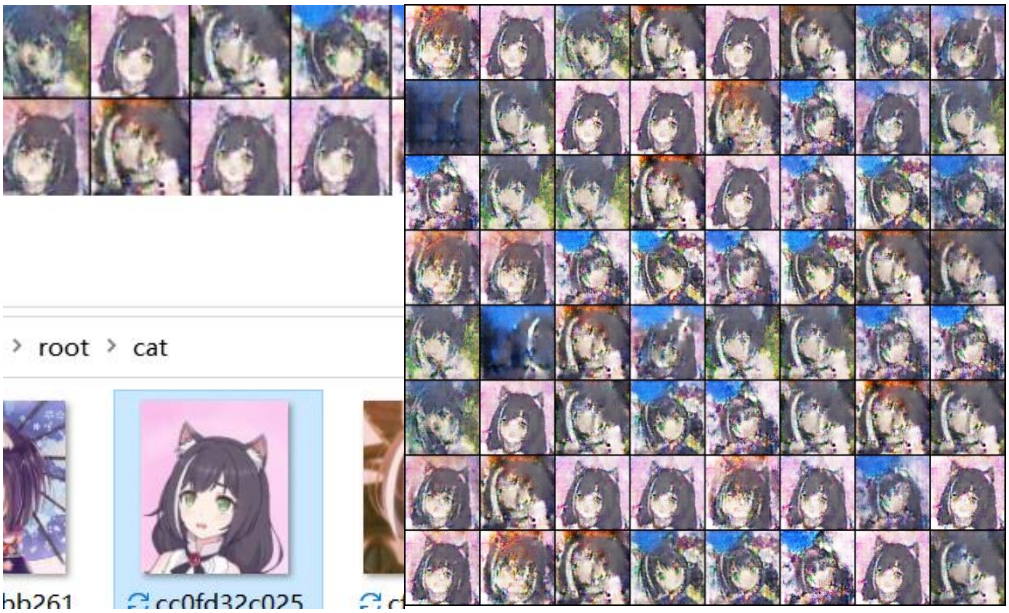
在角色选取方面，我们最终选定了一个二次元游戏角色“凯留”。她有一个称号叫做“接头霸王”。顾名思义，该角色的头部常被接到各种其它人物上，且几乎没有违和感。

这是我所参与修改并搭建到 QQ 机器人上的“凯留接头”功能的图片展示。

接头功能链接：https://github.com/pcrbot/plugins-for-Hoshino/tree/master/shebot/conhead

我们对凯留截取了 60 张不同的头部图片，进行三千轮的训练。

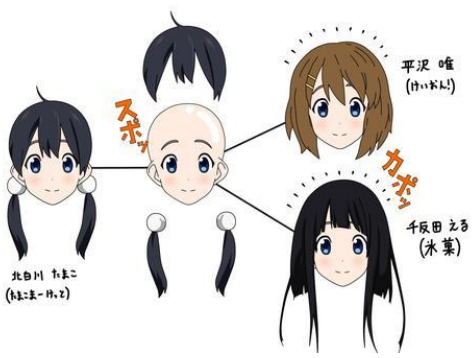
我们猜测，生成器为了能“欺骗”判别器，生成的头像一定是越来越接近原图。而最终结果和我们的猜测是相符的。



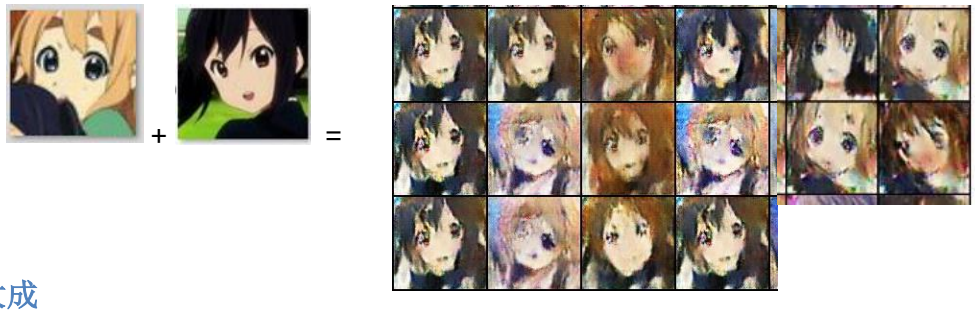
*3000 轮

那么，训练一组画风相近的人物呢？

接着，我们筛选出 200 张同一个动漫公司（京都动画）的人物。该公司的多部作品画风统一且有辨识度，被业界和粉丝称作“京都脸”。



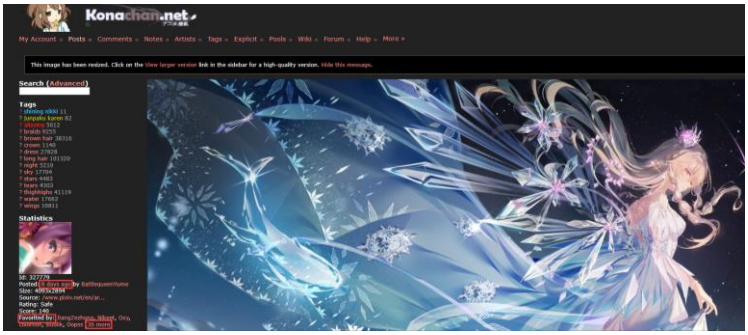
经过一千轮的训练，我们发现生成结果和单个人物出现了不同。比如这张图片，他将两个不同的原图中的两个部分，即第一张图中的发色，与第二张图的脸型和眼睛合二为一，生成了与原图不完全相同的新的图像。



集大成

由此，我们得出结论：为了最终生成的头像更加美观，我们需要对训练集做出筛选。之前爬取的训练集中有不良好的图片。所谓的不良好，包括把其它器官识别成头像、把两个头框在了一起、画风差距过大的图片。

二次元人物的头像画风随着时间的变化是非常明显的。为此，我们限定了图像的最低浏览量/点赞量（以确保这是一张被大家认可的美观的图片）、以及作画年份（为近三年），重新爬虫。



同时，我们改用训练好的 yoloV3 模型来更为精确的识别头部。

最终，我们得到了新的，质量更高的训练集。其中包含 7w 张图片。

yolo3 动漫人脸检测：https://github.com/wmylxmj/YOLO-V3-IOU

这里是上述、优化到第三个版本的训练集训练 180 轮后的结果展示。根据训练好的模型，生成 512 组随机噪点，喂入训练好的生成器，用判别器选出其认为最好的 64 张并展示。

*180 轮，机能受限未训练到拟合。

可以看出，其中已经有完成度较好的头像，极端丑陋的图像已经消失。当前最大的问题包括极其不自然的勾线、以及直接糊成一团的生成头像而被判别器认为是好的头像。

网络优化

观察中间成果发现，糊成一团的头像的生成原因之一，是其眼睛的位置不确定。通常是前一次生成时眼睛在下方，后一次生成时眼睛在上方，再训练后变成了四个眼睛、然后变成 16 个眼睛，最后整张图像扭曲。

带着以上经验和结论，我们开始了对模型的修改。

了解基本原理

之前我们介绍了构成 GAN 的生成器和判别器的基本工作，而对他们具体的工作机制一无所知。

> *我们在外网找到了一个能精确且完整的解答我们疑惑的论文：*
> *Image_Completion_with_Deep_Learning：http://bamos.github.io/2016/08/09/deep-completion/*

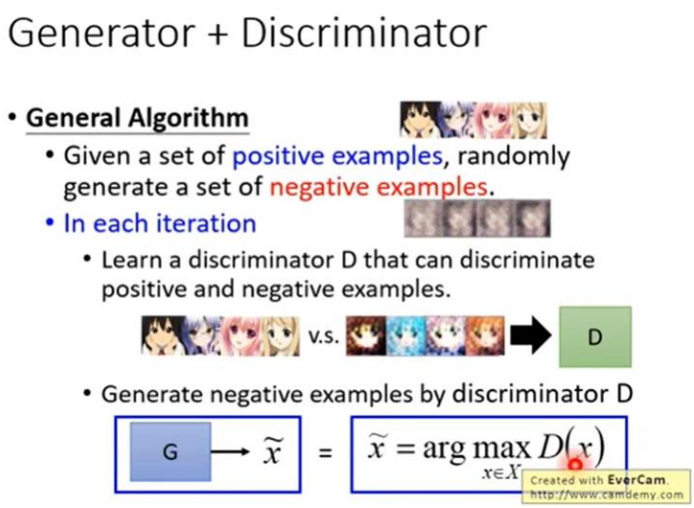
假设，一幅图像中给挖去了一块，作为一个人类，我们会如何去填补缺失掉的这块图像呢？ 首先，我们可以根据周围像素提供的信息，来推断丢失的像素。 其次，我们判断我们填入的像素是“正常的”，是符合日常生活中的所见所闻的。 在机器学习中，我们称前一种为“上下文信息”（Contextual information），后一种为“感知信息”（Perceptual information）

我们可以认为，头像的一些特征，比如眼睛的位置，头发的走向，在让人感到比例协调时，其对应的像素矩阵也服从某种复杂分布的。而各种“抽象派”、难看的头像，是处于这个分布之外的。因此，生成器实质上是在学习如何解这个非线性方程。 辨别器实质上是在不断完善这个非线性方程（细化分布的边界条件）。他们相互配合，以找出二次元头像的图片特征。

于是我们明白，我们给机器喂入的图像，被解构为概率分布的样本。机器通过学习这些样本，逐渐能回答出，在不同的上下文环境下，最符合“感知”的信息是什么，也就是哪些生成的图像是“最符合常理的”。

我们在“计算机科学导论”这门课中曾尝试过，光是训练一个判别人、自行车、汽车的分类器，想要达到一个较好的效果，除非进行大量的预训练，或者人工提取特征，不然往往需要大量的图像，至少是万张数量级。现在我们希望同样以万张数量级的图像，让机器能自主生成一个新图像，肯定是不够的。

这，就 GAN 网络的独到之处：除了训练判别器，也同时训练一个生成图像的生成器。 在训练判别器的时候，训练集中的图片向判别器送入真图像；生成器生成的图片向判别器送入假头像。训练完后，固定判别器，再以判别器类似的去训练生成器。由此，源源不断，生生不息。



对比之下可以发现：直接用辨别器“造假”、“画画”的“成本过高”，需要解一个非线性方程，找到 argmaxD(x)。 而对于生成器来说，“造假”只需要做几遍反卷积即可。

以下是算法的伪代码：

for number of training iterations **do**
  for k steps **do**
    • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

上面我们所阐释的的，GAN 算法核心原理的公式化描述如下：（当理解了以后，可以说是非常简洁）

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))].$$

DCGAN

基于对深度学习中图像处理最常用的 CNN 模型的拓展，我们很容易想到，GAN 中判别生成的图像是真是假的判别器，是否可以用 CNN 二元分类器来进行替代？查阅得知，这便是深度卷积生成对抗网络，DCGAN。

DCGAN 论文地址：https://arxiv.org/abs/1511.06434

DCGAN 把上述的 G 和 D 换成了两个卷积神经网络（CNN）。为了提高样本的质量和收敛的速度，DCGAN 的神经网络的结构有以下特点：

- 取消所有池化层。G 网络中使用转置卷积（transposed convolutional layer）进行上采样，D 网络中用加入 stride 的卷积代替池化。
- 在 D 和 G 中均使用 batch normalization，防止初始化时随机种子不优秀导致梯度爆炸。
- 去掉全连接层，使网络变为全卷积网络

卷积神经网络中，有以下几个对我们较为陌生的参数：

- 输入的宽度 i
- 卷积核的宽度 k
- 单边填充宽度 p
- 输出宽度 o
- 步长 s

在学习其含义时，我们分别参考了论文、博客、视频。由于时间原因不展开详述：

论文：https://arxiv.org/pdf/1511.06434.pdf

博客：https://blog.csdn.net/weixin_43794311/article/details/109195285

视频：https://www.bilibili.com/video/BV1P4411f7hK?p=48

最终，我们得到两组公式：

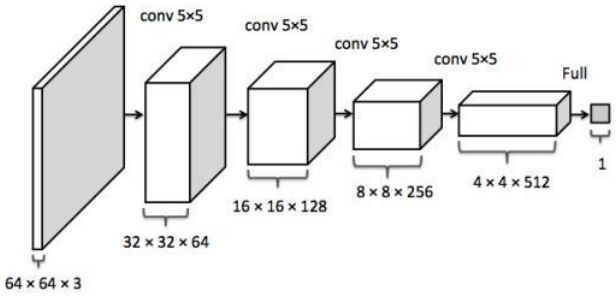
在正向卷积中，有：o=(i+2p-k)/s+1

在反向卷积中，有：o=s(i-1)-2p+k+(o+2p-k)%s

因为 DCGAN 中全部采用的是全卷积。因此，根据以上公式，我们在确认好输入和输出图像的长宽后，可以快速计算出合适的步长、填充和卷积核。最终我们确定并重写了网络结构。以反卷积为例，从上到下为：

```
        self.Gene = nn.Sequential(
            # 假定输入为opt.nz*1*1 维的数据，opt.nz 维的向量
            # output = (input - 1)*stride + output_padding - 2*padding + kernel_size
            # 把一个像素点扩充卷积，让机器自己学会去理解噪声的每个元素是什么意思。
            nn.ConvTranspose2d(in_channels=opt.nz,
                               out_channels=self.ngf * 8,
                               kernel_size=4,
                               stride=1,
                               padding=0,
                               bias=False),
            nn.BatchNorm2d(self.ngf * 8),
            nn.ReLU(inplace=True),

            # 输入一个4*4*ngf*8
            nn.ConvTranspose2d(in_channels=self.ngf * 8,
                               out_channels=self.ngf * 4,
                               kernel_size=4,
                               stride=2,
                               padding=1,
                               bias=False),
            nn.BatchNorm2d(self.ngf * 4),
            nn.ReLU(inplace=True),

            # 输入一个8*8*ngf*4
            nn.ConvTranspose2d(in_channels=self.ngf * 4,
                               out_channels=self.ngf * 2,
                               kernel_size=4,
                               stride=2,
                               padding=1,
                               bias=False),
            nn.BatchNorm2d(self.ngf * 2),
            nn.ReLU(inplace=True),

            # 输入一个16*16*ngf*2
            nn.ConvTranspose2d(in_channels=self.ngf * 2,
                               out_channels=self.ngf,
                               kernel_size=4,
                               stride=2,
                               padding=1,
                               bias=False),
            nn.BatchNorm2d(self.ngf),
            nn.ReLU(inplace=True),

            # 输入一张32*32*ngf
            nn.ConvTranspose2d(in_channels=self.ngf,
                               out_channels=3,
                               kernel_size=5,
                               stride=3,
                               padding=1,
                               bias=False),

            # Tanh 收敛速度快于sigmoid,远慢于relu,输出范围为[-1,1]，输出均值为0
            nn.Tanh(),
        )  # 输出一张96*96*3
```

与之类似，判别器卷积顺序如图所示：（与生成器相反）



其它技术

ReLU 和 LeakyReLU

DCGAN 中，我们测试激活函数发现：Tanh 收敛速度快于 sigmoid。最终，我们决定在 G 网络中使用 ReLU 作为激活函数，但最后一层改用 tanh。 在 D 网络中使用 LeakyReLU 作为激活函数。Leaky 相对原版据称削弱了 ReLU 的负数硬饱和问题。

*然而在我们的测试中没有发现其对结果有肉眼可见的改变。

loss 和学习率控制

采用 loss 计算的公式如下：

BCELOSS

CLASS `torch.nn.BCELoss(weight=None, size_average=None, reduce=None, reduction='mean')` [SOURCE]

Creates a criterion that measures the Binary Cross Entropy between the target and the output:

The unreduced (i.e. with `reduction` set to `'none'`) loss can be described as:

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_n \left[y_n \cdot \log x_n + (1 - y_n) \cdot \log(1 - x_n) \right],$$

where N is the batch size. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \text{mean}(L), & \text{if reduction} = \text{'mean'}; \\ \text{sum}(L), & \text{if reduction} = \text{'sum'}. \end{cases}$$

This is used for measuring the error of a reconstruction in for example an auto-encoder. Note that the targets y should be numbers between 0 and 1.

Notice that if x_n is either 0 or 1, one of the log terms would be mathematically undefined in the above loss equation. PyTorch chooses to set $\log(0) = -\infty$, since $\lim_{x \to 0} \log(x) = -\infty$. However, an infinite term in the loss equation is not desirable for several reasons.

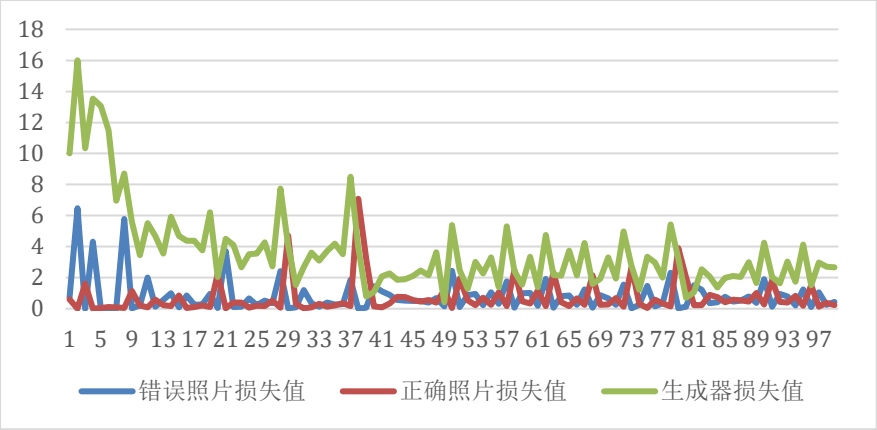
For one, if either $y_n = 0$ or $(1 - y_n) = 0$, then we would be multiplying 0 with infinity. Secondly, if we have an infinite loss value, then we would also have an infinite term in our gradient, since $\lim_{x \to 0} \frac{d}{dx} \log(x) = \infty$. This would make BCELoss's backward method nonlinear with respect to x_n, and using it for things like linear regression would not be straight-forward.

Our solution is that BCELoss clamps its log function outputs to be greater than or equal to -100. This way, we can always have a finite loss value and a linear backward method.

来源：

https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html?highlight=bceloss

由于算力限制，我们采取了动态输出 loss 值，并据此动态调整学习率的训练方法。以下展示的是我们将 loss 数据制成的图表，以及就此采取的对学习率的改变。



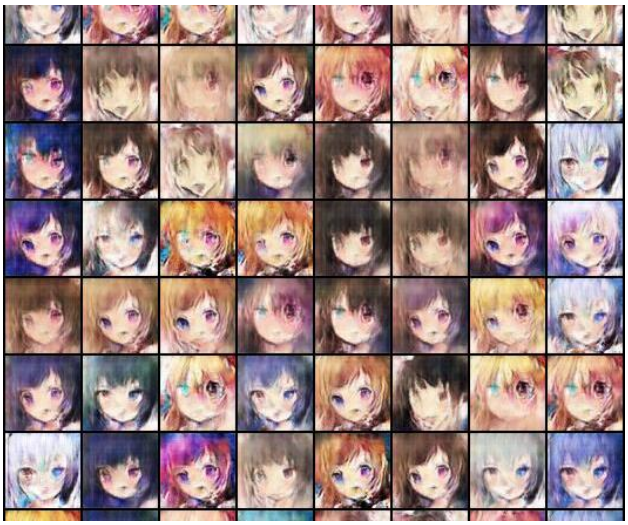
随机梯度下降法

随机梯度下降法用于减少陷入局部最优的风险，进而可以更接近全局最优的方法。 时间所限，我们未其进行深入研究，直接使用了原项目所采取的 Adam 优化。

最终 DCGAN 源代码见文件 DCGAN.py

成果展示

最终，经过以上所有优化，得到的最终结果如下图：



*160 轮
5 月 31 日 08:00
epoch1-10：learning rate=1e-3
epoch10-30：learning rate =5e-4
epoch30-50：learning rate =3e-4
epoch50-160：learning rate =2e-4

*290 轮
6 月 12 日 22:00
Epoch160-290：learning rate =1e-4

其他参数：
d_every = 1
g_every = 5
batch_size = 64
Adam optimizer beta1 = 0.5
Leaky ReLU=0.2

总结分析

在整个项目的周期中，我们大多数的时间用于搭建和复现各种程序：至少包括 python 环境、openCV、yoloV3、图像爬虫、Cuda、GAN、DCGAN 等；其次，便是在遇到问题时，搜索各种论文、博客、视频，查阅并找到解答。因此、可以说本次挑战给我们最大的收获，便是信息检索，以及自主学习的能力。

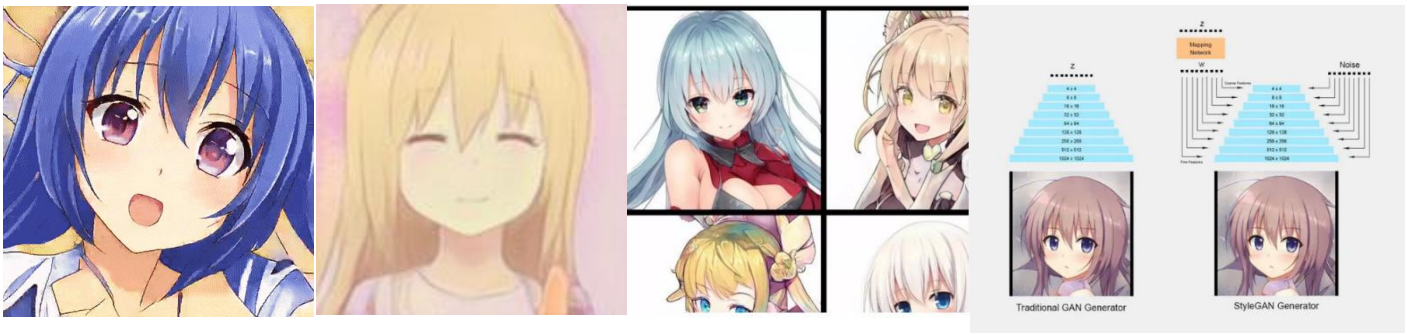
从参考列表的长度可以也看出，我们首先需要感谢广袤的互联网。

其次，感谢老师以及多媒体技术这门课程，给予我们一个尝试未曾设想的算法的平台，一个拓宽知识面的契机。

最后，是感谢我们自己。这次项目肯定了我们对计算机视觉处理方向的能力。在漫长的调试和 debug 中，这份热爱战胜了知识缺乏的胆怯，和队友一起磨砺着骨子里的热忱和眼睛里的光。

回望这个项目，确实存在值得惋惜之处：比如，图像的分辨率不足、缺乏对 DCGAN 模型进一步修改与调试的时间、无法通过修改低层特征获得瞳色相同的双眼，等等。

本项目所基于的基础论文（GAN）距今已有八年。最新的 StyleGAN（2019）在人像生成方面取得了跨越性的突破，被美誉为“GAN2.0”。其生成器应用风格迁移的思路重写；通过在各自的图层上给予噪声来控制不同层次的细节的随机变化，允许控制生成样本的不同细节水平，从粗糙的细节（如头部形状）到更精细的细节（如眼睛颜色）；在训练集标签不足（例如男性角色）时截断潜伏向量迫使其接近平均值。



StyleGAN 论文：https://arxiv.org/abs/1812.04948

生成左上图的作者论文：https://www.gwern.net/Faces

左中图：https://twitter.com/ak92501

waifu 生成在线网站：

https://www.thiswaifudoesnotexist.net/

https://waifulabs.com/

然而，其数据集需求也超过百万（标签数量更是近亿），受限于知识储备和机能不足，只得“日后再战”。

谢谢！

参考文献

https://www.bilibili.com/video/BV1d4411G76N

https://www.nvidia.com/en-us/research/ai-playground/?ncid=so-twi-nz-92489fbclid=IwAR3cf-s5JzcVLxMRLAL4pi9DH4zT_OKRL4U_6kXio3FqP1M-vEuiriHBHl0

https://arxiv.org/abs/1903.07291

https://zhuanlan.zhihu.com/p/24767059

https://blog.csdn.net/qq_43815869/article/details/109121772

http://safebooru.donmai.us/

http://konachan.net/

https://www.pixiv.net/

https://github.com/nagadomi/lbpcascade_animeface

https://github.com/pcrbot/plugins-for-Hoshino/tree/master/shebot/conhead

https://github.com/wmylxmj/YOLO-V3-IOU

http://bamos.github.io/2016/08/09/deep-completion/

https://arxiv.org/abs/1511.06434

https://arxiv.org/pdf/1511.06434.pdf

https://blog.csdn.net/weixin_43794311/article/details/109195285

https://www.bilibili.com/video/BV1P4411f7hK?p=48

https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html?highlight=bceloss