Exp6: Generative Adversarial Networks

使用生成对抗网络(GAN) 生成 图片

In [1]:

```
from __future__ import print_function

# import argparse
import os
import random
import torch
import torch.nn as nn
# import torch.nn.parallel
# import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML
%matplotlib inline
%config InlineBackend.figure_format = 'svg'

Seed = 220103
random.seed(Seed)
torch.manual_seed(Seed)
```

executed in 1.78s, finished 22:42:17 2022-01-05

Out[1]: <torch._C.Generator at 0x1eaa2a16730>

1. 数据准备

- 将所有人脸图片放入 face 文件夹（共13233）
- 创建数据迭代器

In [2]:

```
dataroot = r"F:\2021研一上学期\深度学习\assignment6-GAN\image_data"
image_size = 64
batch_size = 32

# Create the dataset
dataset = dset.ImageFolder(root=dataroot,
                           transform=transforms.Compose([
                               transforms.Resize(image_size),
                               transforms.CenterCrop(image_size),
                               transforms.ToTensor(),
                               transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ]))
# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=2)

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available()) else "cpu")

# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis("off")
plt.title("Training Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64], padding=2, normalize=True).cpu(),
```

executed in 6.15s, finished 22:42:23 2022-01-05

Out[2]: <matplotlib.image.AxesImage at 0x1eaabfbb490>

In [3]:

```
# Number of channels in the training images. For color images this is 3
nc = 3

# Size of z latent vector (i.e. size of generator input)
nz = 300

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 300

# Learning rate for optimizers
lr = 0.0002
```

executed in 14ms, finished 22:42:23 2022-01-05

2. 模型搭建

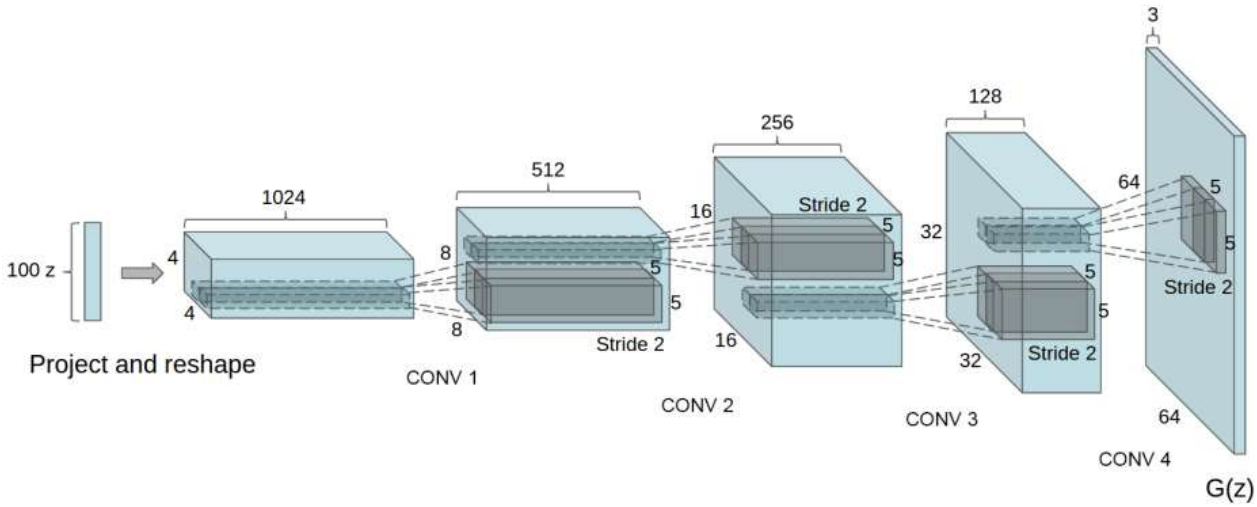
在生成网络G和对抗网络D上调用自定义权重初始化
在DCGAN论文中，作者指定所有模型的权值都应该从均值为0,标准差0.02的正态分布随机初始化。

weights_init函数接受一个初始化的模型作为输入，并重新初始化所有卷积、卷积转置和批处理规范化层，以满足这个标准。此函数在初始化后立即应用于模型。

In [4]:
```
# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

executed in 14ms, finished 22:42:23 2022-01-05

生成器 G 的结构如下



其中，nz为输入向量z的长度，ngf与通过生成器传播的特征图的大小有关，nc为输出图像中的通道数(对于RGB图像，设置为3)。

In [5]:
```
# Generator Code

class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

    def forward(self, input):
        return self.main(input)
```

executed in 14ms, finished 22:42:23 2022-01-05

实例化生成器并应用weights_init函数

In [6]:

```
# Create the generator
netG = Generator().to(device)

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.02.
netG.apply(weights_init)

# Print the model
print(netG)
```

executed in 45ms, finished 22:42:23 2022-01-05

```
Generator(
  (main): Sequential(
    (0): ConvTranspose2d(300, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

鉴定器 D 是一个二值分类网络，它以图像作为输入，并输出输入图像是真实的(而不是假的)的概率。
在这里，D 获取一个3x64x64的输入图像，通过一系列Conv2d、BatchNorm2d和LeakyReLU层处理它，并通过Sigmoid激活函数输出最终的概率。

In [7]:

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 16 x 16
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 8 x 8
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)
```

executed in 13ms, finished 22:42:23 2022-01-05

In [8]:
```
# Create the Discriminator
netD = Discriminator().to(device)

# Apply the weights_init function to randomly initialize all weights
#  to mean=0, stdev=0.2.
netD.apply(weights_init)

# Print the model
print(netD)
```

executed in 29ms, finished 22:42:23 2022-01-05

```
Discriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

接下来，我们将真正的标签定义为1，假标签定义为0。在计算生成器和鉴定器的loss时会用到这些标签，这也是原GAN论文中使用的约定。
最后，设置了两个单独的优化器，一个用于D，一个用于G，如DCGAN论文所述，都是Adam优化器，学习速率为0.0002,Beta1 = 0.5
为了跟踪生成器的学习进程，从高斯分布(即fixed_noise)中生成一批固定的潜在向量。

In [9]:
```
# Initialize BCELoss function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
#  the progression of the generator
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1.
fake_label = 0.

# Setup Adam optimizers for both G and D
beta1 = 0.5
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
```

executed in 14ms, finished 22:42:23 2022-01-05

3. 模型训练

In [10]:

```
# Training Loop

# Lists to keep track of progress
img_list = []
fake_list= []
G_losses = []
D_losses = []
iters = 0

print("Starting Training Loop...")
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        ############################
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        ###########################
        ## Train with all-real batch
        netD.zero_grad()
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size,), real_label, dtype=torch.float, device=device)
        # Forward pass real batch through D
        output = netD(real_cpu).view(-1)
        # Calculate loss on all-real batch
        errD_real = criterion(output, label)
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()

        ## Train with all-fake batch
        # Generate batch of latent vectors
        noise = torch.randn(b_size, nz, 1, 1, device=device)
        # Generate fake image batch with G
        fake = netG(noise)
        label.fill_(fake_label)
        # Classify all fake batch with D
        output = netD(fake.detach()).view(-1)
        # Calculate D's loss on the all-fake batch
        errD_fake = criterion(output, label)
        # Calculate the gradients for this batch, accumulated (summed) with previous gradients
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        # Compute error of D as sum over the fake and the real batches
        errD = errD_real + errD_fake
        # Update D
        optimizerD.step()

        ############################
        # (2) Update G network: maximize log(D(G(z)))
        ###########################
        netG.zero_grad()
        label.fill_(real_label)  # fake labels are real for generator cost
        # Since we just updated D, perform another forward pass of all-fake batch through D
        output = netD(fake).view(-1)
        # Calculate G's loss based on this output
        errG = criterion(output, label)
        # Calculate gradients for G
        errG.backward()
        D_G_z2 = output.mean().item()
        # Update G
        optimizerG.step()

        # Output training stats
        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G(z)): %.4f / %.4f'
                  % (epoch, num_epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))

        # Save Losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

        # Check how the generator is doing by saving G's output on fixed_noise
```

```
        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i == len(dataloader)-1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

        iters += 1

torch.save(netD.state_dict(), 'model/netD.pth')
torch.save(netG.state_dict(), 'model/netG.pth')
```

executed in 2h 13m 55s, finished 00:56:18 2022-01-06

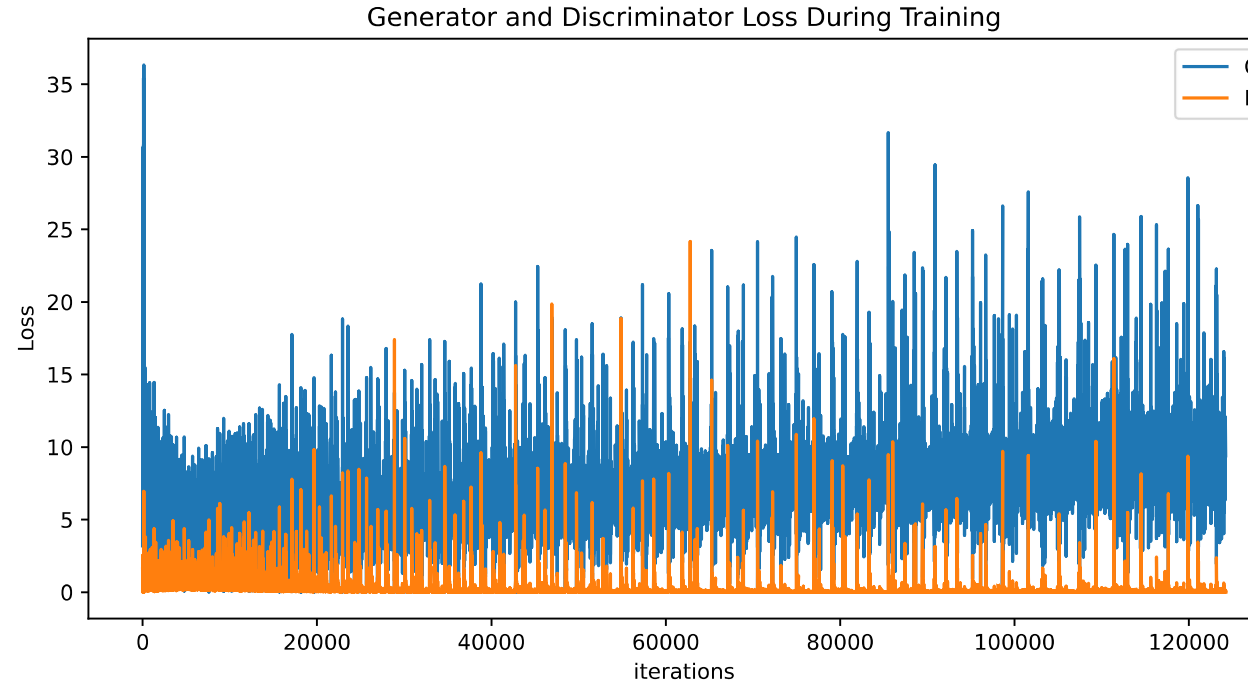
...

4. 结果查看与保存

In [11]:

```
plt.figure(figsize=(10,5))
plt.title("Generator and Discriminator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
# plt.savefig("./result/Loss.jpg")
plt.show()
```

executed in 208ms, finished 00:56:18 2022-01-06



Visualization of G's progression

Remember how we saved the generator's output on the fixed_noise batch after every epoch of training. Now, we can visualize the training progression of G with an animation. Press the play button to start the animation.

In [12]:

```
#%%capture
fig = plt.figure(figsize=(8,8))
plt.axis("off")
ims = [[plt.imshow(np.transpose(i,(1,2,0)), animated=True)] for i in img_list]
ani = animation.ArtistAnimation(fig, ims, interval=1000, repeat_delay=1000, blit=True)

HTML(ani.to_jshtml())
```

executed in 19.7s, finished 00:56:38 2022-01-06

```
Animation size has reached 21398016 bytes, exceeding the limit of 20971520.0. If you're sure you want a larger animation embedded, set the animation.embed_limit rc parameter to a larger value (in MB). This and further frames will be dropped.
```

Out[12]:

Once Loop Reflect

Real Images vs. Fake Images

Finally, lets take a look at some real images and fake images side by side.

In [13]:

```
# Grab a batch of real images from the dataloader
real_batch = next(iter(dataloader))

# Plot the real images
plt.figure(figsize=(15,15))
plt.subplot(1,2,1)
plt.axis("off")
plt.title("Real Images")
plt.imshow(np.transpose(vutils.make_grid(real_batch[0].to(device)[:64], padding=5, normalize=True).cpu(),

# Plot the fake images from the last epoch
plt.subplot(1,2,2)
plt.axis("off")
plt.title("Fake Images")
plt.imshow(np.transpose(img_list[-1],(1,2,0)))
plt.savefig("./result/face.jpg")
plt.show()
```

executed in 3.19s, finished 00:56:41 2022-01-06

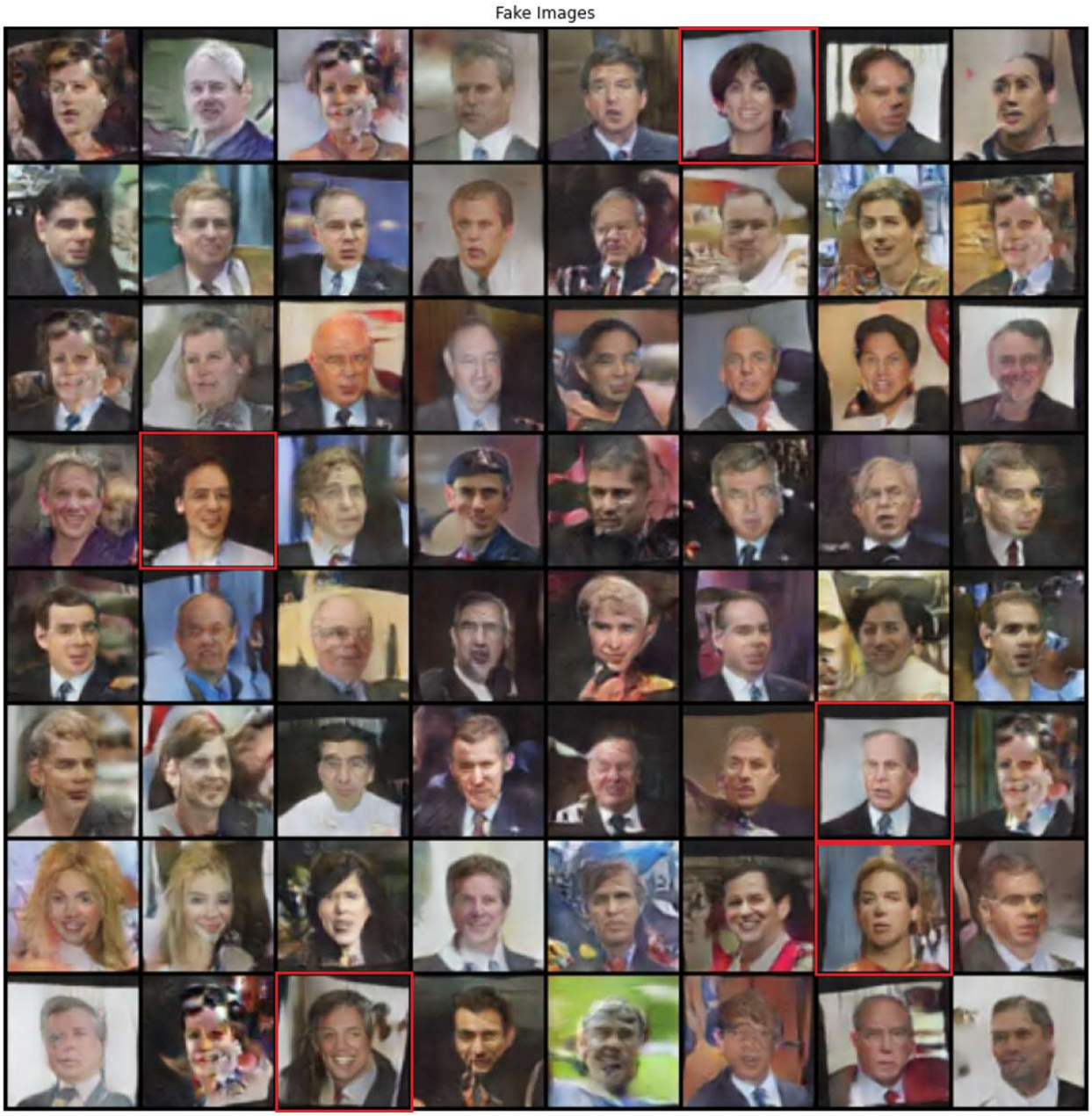
In [14]:

```
for i in range(1,len(img_list)):
    plt.figure(figsize=(15,15))
    plt.axis("off")
    plt.title("Fake Images")
    plt.imshow(np.transpose(img_list[i],(1,2,0)))
    plt.savefig(f"./result/face_{i}.jpg")
```

executed in 2m 11s, finished 00:58:52 2022-01-06

...

最终选出的我最满意的5张图片如下：



5. 计算 FID

In [ ]:

```
test = vutils.make_grid(fake, padding=0, normalize=True)
vutils.save_image(test, './test.jpg', normalize=False)
```

In [68]:

```
'''
将一张图片切为64张图
'''
from PIL import Image
import sys

#将图片填充为正方形
def fill_image(image):
    width, height = image.size
    #选取长和宽中较大值作为新图片的
    new_image_length = width if width > height else height
    #生成新图片[白底]
    new_image = Image.new(image.mode, (new_image_length, new_image_length), color='white')
    #将之前的图粘贴在新图上，居中
    if width > height:#原图宽大于高，则填充图片的竖直维度
        #(x,y)二元组表示粘贴上图相对下图的起始位置
        new_image.paste(image, (0, int((new_image_length - height) / 2)))
    else:
        new_image.paste(image,(int((new_image_length - width) / 2),0))

    return new_image

#切图
def cut_image(image):
    width, height = image.size
    item_width = 64
    box_list = []
    # (left, upper, right, lower)
    for i in range(0,8):#两重循环，生成64张图片基于原图的位置
        for j in range(0,8):
            #print((i*item_width,j*item_width,(i+1)*item_width,(j+1)*item_width))
            box = (j*item_width+k,i*item_width+k,(j +1)*item_width+k,(i+1)*item_width+k)
            box_list.append(box)

    image_list = [image.crop(box) for box in box_list]
    return image_list

#保存
def save_images(image_list):
    index = 1
    for image in image_list:
        image.save('./img/'+str(index) + '.png', 'PNG')
        index += 1

file_path = "test.jpg"
image = Image.open(file_path)
# image.show()
image_list = cut_image(image)
save_images(image_list)

```

executed in 88ms, finished 10:57:36 2022-01-06

In [66]:

```
! python fid_score.py image_data/face img --gpu 0
```

executed in 27ms, finished 10:57:23 2022-01-06

计算得到结果为 194.7550522848991

In [ ]:

```

```