Image Processing and computer vision HW3

Author: Kr.Cen
SID: 521021910151
Date: 2023.12.28

In this lab, we train DCGAN on water glasses. You can find my materials on https://github.com/Kr-Panghu/Image-Processing-CS3964

Dataset Collection

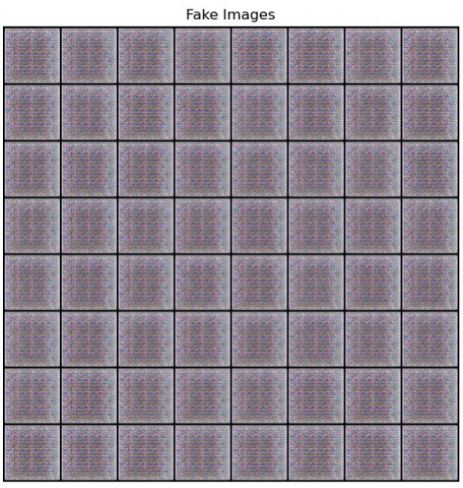
Our group has 5 members, so we collect 20 pictures respectively from amazon, decathlon, etc. As for myself, I picked 20 high-quality images on Google Images with keywords like Water glass, Vacuum cup, etc. After that, we deleted some of the images that were in the wrong format, and used a random deletion method to keep 90 images as a public dataset.

For the additional 10 images, I still selected them on Google Images, but this time I searched them using **Fancy water glass** as a keyword to **increase the diversity** of results I generated.

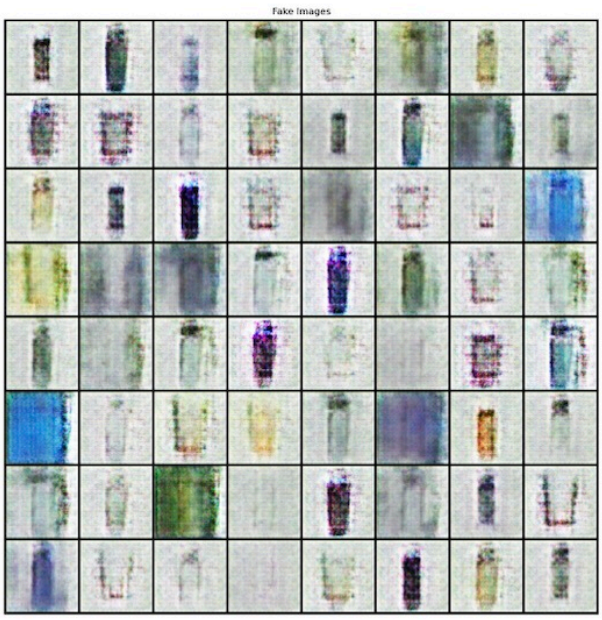
So far, I have obtained 100 pictures of water glass as the data set of this experiment.

DCGAN Experiment

First, I tried to train with 100 epoches in exp1 but failed to get even a real image, just got some unclear noise.



Obviously, the first thing we should do is increase the training epoch, thus I increase epoch=100 to epoch=1000 in exp2 and got not bad results this time.



At least some of the noise can be reconstructed into a more formed cup, but it is still in a very fuzzy state.

Then I attemped to improve the above-mentioned performance.

Adjust units inside network

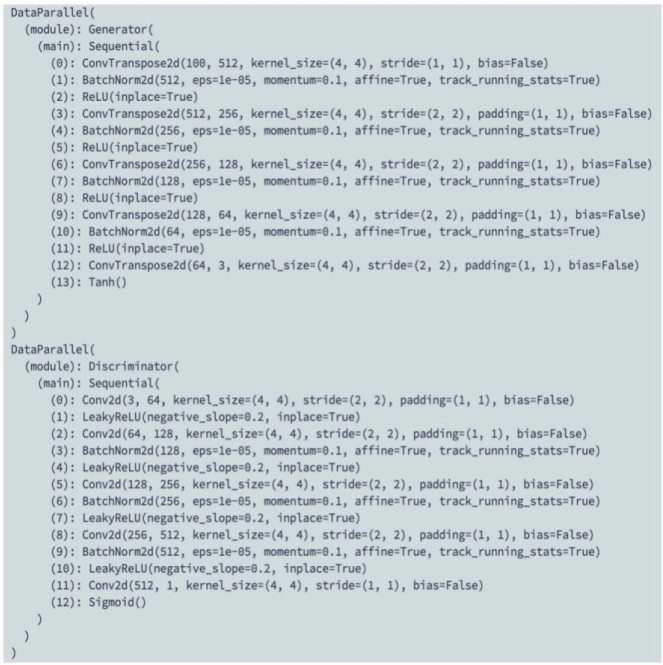
In the following experiments, I tried to increase the `image_size` to get a better results, which means I have to change the structure of the networks, thus the `Generator` and `Discriminator` will be like:

```
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            nn.ConvTranspose2d(nz, ngf * 16, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 16),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf * 16, ngf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
```

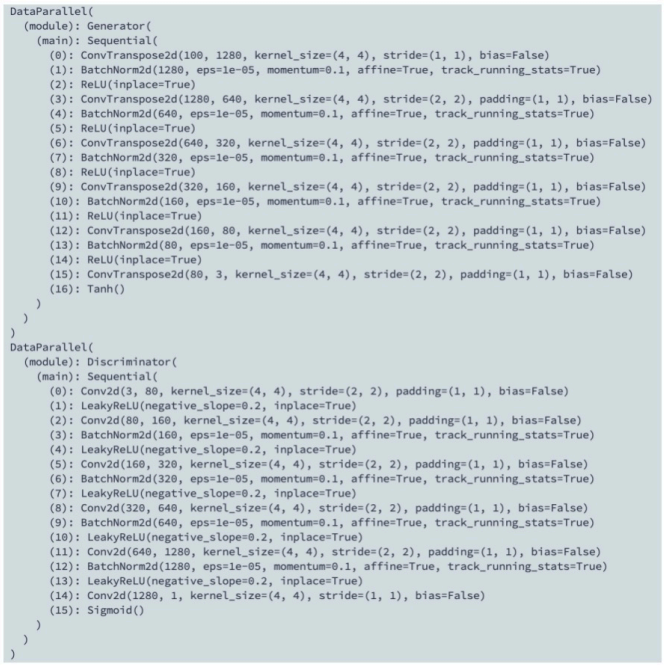
```
            nn.ConvTranspose2d(ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),

            nn.ConvTranspose2d(ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
        )

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

class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        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, ndf * 16, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 16),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. ``(ndf*16) x 4 x 4``
            nn.Conv2d(ndf * 16, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

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

You can find more details in exp3/dcgan.py.

The network structure will change like: (From left to right)



```
DataParallel(
  (module): Generator(
    (main): Sequential(
      (0): ConvTranspose2d(100, 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()
    )
  )
)
DataParallel(
  (module): 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()
    )
  )
)
```



```
DataParallel(
  (module): Generator(
    (main): Sequential(
      (0): ConvTranspose2d(100, 1280, kernel_size=(4, 4), stride=(1, 1), bias=False)
      (1): BatchNorm2d(1280, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
      (3): ConvTranspose2d(1280, 640, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (4): BatchNorm2d(640, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (5): ReLU(inplace=True)
      (6): ConvTranspose2d(640, 320, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (7): BatchNorm2d(320, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (8): ReLU(inplace=True)
      (9): ConvTranspose2d(320, 160, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (10): BatchNorm2d(160, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (11): ReLU(inplace=True)
      (12): ConvTranspose2d(160, 80, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (13): BatchNorm2d(80, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (14): ReLU(inplace=True)
      (15): ConvTranspose2d(80, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (16): Tanh()
    )
  )
)
DataParallel(
  (module): Discriminator(
    (main): Sequential(
      (0): Conv2d(3, 80, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (1): LeakyReLU(negative_slope=0.2, inplace=True)
      (2): Conv2d(80, 160, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (3): BatchNorm2d(160, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (4): LeakyReLU(negative_slope=0.2, inplace=True)
      (5): Conv2d(160, 320, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (6): BatchNorm2d(320, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (7): LeakyReLU(negative_slope=0.2, inplace=True)
      (8): Conv2d(320, 640, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (9): BatchNorm2d(640, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (10): LeakyReLU(negative_slope=0.2, inplace=True)
      (11): Conv2d(640, 1280, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
      (12): BatchNorm2d(1280, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (13): LeakyReLU(negative_slope=0.2, inplace=True)
      (14): Conv2d(1280, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
      (15): Sigmoid()
    )
  )
)
```

But unfortunately, hyperparameter adjustment of DCGAN is a challenging task. Additionally, I think the **limited size of the dataset** also led to poor generation (only 100 training images), so I did not get good results in `exp3-exp6`.

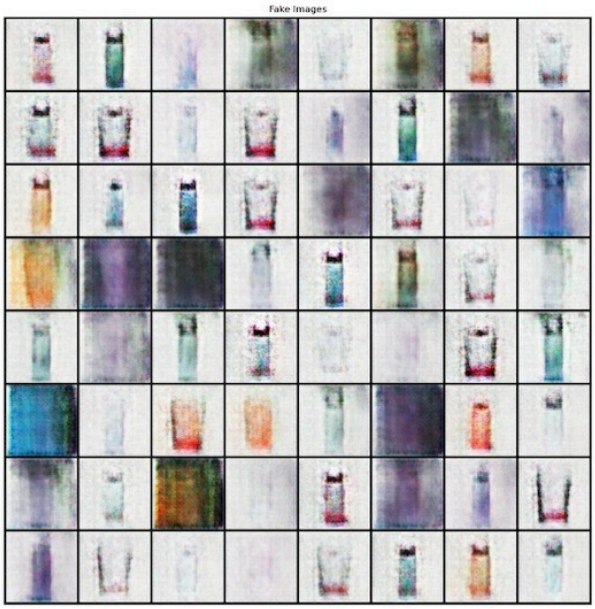
Adjust learning rate

In exp7 I tried to modify learning rate based on exp2 for better performance, which got an impressively greate result.

I added two learning rate schedulers.

```
from torch.optim import lr_scheduler # Adjust learning rate

optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))
schedulerD = lr_scheduler.StepLR(optimizerD, step_size=800, gamma=0.8) # Added here
schedulerG = lr_scheduler.StepLR(optimizerG, step_size=800, gamma=0.8) # Added here

for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):
        ......
        optimizerD.step()
        schedulerD.step() # Added here
        ......
        optimizerG.step()
        schedulerG.step() # Added here
        ......
```

In this way, the first 800 epoches of training will base on `learning_rate=0.0002`, the last 200 epoches of training will base on `learning_rate=0.0002 * 0.8`. This shows a better performance than exp2.



Result analysis

Merits: We can already see that most of the images can get a nearly complete shape of the water cup in exp7. To be honest, from my perspective, training on a data set of just 100 images to get this result is fairly impressive.

Weakness: Still lack of diversity, which is hard to solve **may due to the principles of GAN**, the modern algorithms like **Diffusion models** may help, which learn a backward diffusion process from a random gaussian noise to original picture.
