CSC413 PA4

YINJUN ZHENG

Part 1: Deep Convolutional GAN (DCGAN)

Generator

Question 1

The code is shown as follows:

```
class DCGenerator(nn.Module):
    def __init__(self, noise_size, conv_dim, spectral_norm=False):
        super(DCGenerator, self).__init__()

        self.conv_dim = conv_dim
        ##########################################
        ##   FILL THIS IN: CREATE ARCHITECTURE   ##
        ##########################################

        self.linear_bn = upconv(in_channels=100, out_channels=self.conv_dim*4, kernel_size=3, stride=2, padding=2, spectral_norm=spectral_norm)
        self.upconv1 = upconv(in_channels=self.conv_dim*4, out_channels=self.conv_dim*2, kernel_size=5, stride=2, spectral_norm=spectral_norm)
        self.upconv2 = upconv(in_channels=self.conv_dim*2, out_channels=self.conv_dim, kernel_size=5, stride=2, spectral_norm=spectral_norm)
        self.upconv3 = upconv(in_channels=self.conv_dim, out_channels=3, kernel_size=5, stride=2, spectral_norm=spectral_norm)
```

Training loop

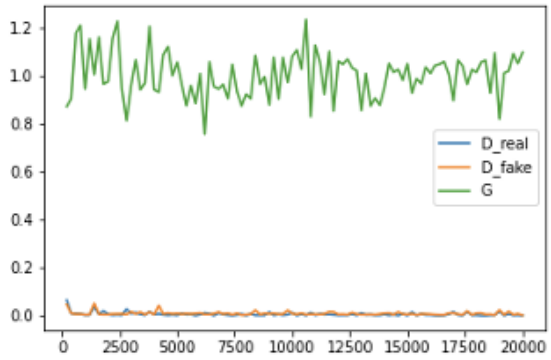
Question 1

The code is shown as follows:

```
# FILL THIS IN
# 1. Compute the discriminator loss on real images
D_real_loss = torch.mean((D(real_images) - 1)**2)

# 2. Sample noise
noise = sample_noise(real_images.shape[0], opts.noise_size)

# 3. Generate fake images from the noise
fake_images = G(noise)

# 4. Compute the discriminator loss on the fake images
D_fake_loss = torch.mean(D(fake_images)**2)
```

```
    # -------------------------
    # 5. Compute the total discriminator loss
    D_total_loss = D_real_loss + D_fake_loss

    D_total_loss.backward()
    d_optimizer.step()

##########################################
###          TRAIN THE GENERATOR       ###
##########################################

g_optimizer.zero_grad()

# FILL THIS IN
# 1. Sample noise
noise = sample_noise(real_images.shape[0], opts.noise_size)

# 2. Generate fake images from the noise
fake_images = G(noise)

# 3. Compute the generator loss
G_loss = torch.mean((D(fake_images)-1)**2)

G_loss.backward()
g_optimizer.step()
```

Experiment
Question 1
The loss function looks as follows:



The generator performance is not so stable as the iteration goes. We can see the the green line fluctuates as through the iterations.

Here are some representation samples:

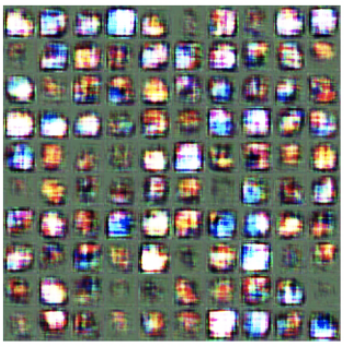


Figure 600

figure 12800

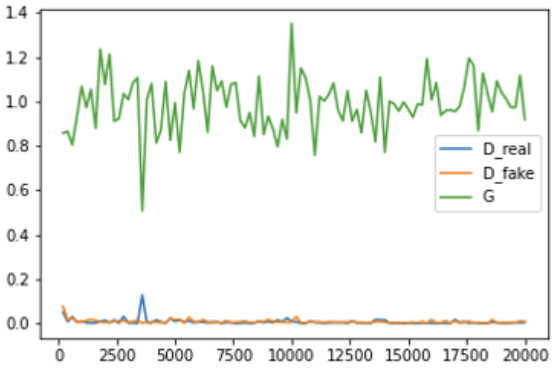
Figure 10200

figure 20000

Comment:
We can see that the in all the above four graphs, the qualities of the samples are not so good. We cannot clearly tell what the images are. But the images show different shapes and different patterns through the iterations. At first, each image is obscure at the middle, but as the iteration goes, the middle of each images become clearer and display some features.

Question 2

If we use the gradient penalty, the loss is shown as follows:



Here are some sample images.

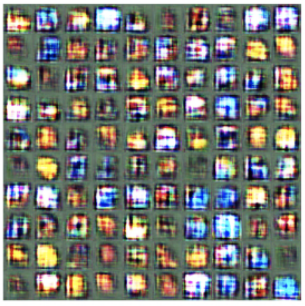


Figure 1000

figure 12600

Figure 12600

figure 20000

Comment:

As we can see above, some of the figures look better than the result without gradient penalty. In most of the figures, the middle of these figures are clearer, while in the results without gradient penalty are more obscure at the middle of the images.

The reason why adding gradient penalty help stabilize training is shown as follows:

(1) It helps avoid the problem of gradient explosion

(2) The penalty guarantees the generalization and convergence of GANs.

PART 2 StyleGAN2-Ada
Experiment
Question 1
The code are shown as follows:

```
# Sample a batch of latent codes {z_1, ...., z_B}, B is your batch size.
def generate_latent_code(SEED, BATCH, LATENT_DIMENSION = 512):
  """
  This function returns a sample a batch of 512 dimensional random latent code

  - SEED: int
  - BATCH: int that specifies the number of latent codes, Recommended batch_size is 3 - 6
  - LATENT_DIMENSION is by default 512 (see Karras et al.)

  You should use np.random.RandomState to construct a random number generator, say rnd
  Then use rnd.randn along with your BATCH and LATENT_DIMENSION to generate your latent codes.
  This samples a batch of latent codes from a normal distribution
  https://numpy.org/doc/stable/reference/random/generated/numpy.random.RandomState.randn.html

  Return latent_codes, which is a 2D array with dimensions BATCH times LATENT_DIMENSION
  """
  ################################################################################
  ######################### COMPLETE THE FOLLOWING ###############################
  ################################################################################
  rnd = np.random.RandomState(SEED)
  latent_codes = rnd.randn(BATCH, LATENT_DIMENSION)
  ################################################################################
  return latent_codes
```

```
def generate_images(SEED, BATCH, TRUNCATION = 0.7):
  """
  This function generates a batch of images from latent codes.

  - SEED: int
  - BATCH: int that specifies the number of latent codes to be generated
  - TRUNCATION: float between [-1, 1] that decides the amount of clipping to apply to the latent code distribution
              recommended setting is 0.7

  You will use Gs.run() to sample images. See https://github.com/NVlabs/stylegan for details
  You may use their default setting.
  """
  # Sample a batch of latent code z using generate_latent_code function
  latent_codes = generate_latent_code(SEED, BATCH)

  # Convert latent code into images by following https://github.com/NVlabs/stylegan
  fmt = dict(func=tflib.convert_images_to_uint8, nchw_to_nhwc=True)
  images = Gs.run(latent_codes, None, truncation_psi= TRUNCATION, randomize_noise=True, output_transform=fmt)
  return PIL.Image.fromarray(np.concatenate(images, axis=1) , 'RGB')
################################################################################
```

Question 2
The code is shown as follows:

```
def interpolate_images(SEED1, SEED2, INTERPOLATION, BATCH = 1, TRUNCATION = 0.7):
  """
  - SEED1, SEED2: int, seed to use to generate the two latent codes
  - INTERPOLATION: int, the number of interpolation between the two images, recommended setting 6 - 10
  - BATCH: int, the number of latent code to generate. In this experiment, it is 1.
  - TRUNCATION: float between [-1, 1] that decides the amount of clipping to apply to the latent code distribution
              recommended setting is 0.7

  You will interpolate between two latent code that you generate using the above formula
  You can generate an interpolation variable using np.linspace
  https://numpy.org/doc/stable/reference/generated/numpy.linspace.html

  This function should return an interpolated image. Include a screenshot in your submission.
  """
  latent_code_1 = generate_latent_code(SEED1, BATCH)
  latent_code_2 = generate_latent_code(SEED2, BATCH)

  latents = np.linspace(latent_code_1, latent_code_2, num = INTERPOLATION).reshape(INTERPOLATION, 512)

  fmt = dict(func=tflib.convert_images_to_uint8, nchw_to_nhwc=True)
  images = Gs.run(latents, None, truncation_psi= TRUNCATION, randomize_noise=True, output_transform=fmt)

  return PIL.Image.fromarray(np.concatenate(images, axis=1) , 'RGB')
```

Row of interpolation is shown as follows:

```
# Create an interpolation of your generated images
interpolate_images(1000, 2000, INTERPOLATION = 6)
```

Question 3

Experiment 1

col_seeds = [1, 2, 3, 4, 5]

row_seeds = [6]

col_styles = [1, 2, 3, 4, 5]

The images are:

Experiment 2

col_seeds = [1, 2, 3, 4, 5]

row_seeds = [6]

col_styles = [8, 9, 10, 11, 12]

The images are shown as follows:

Conclusion:

From the above experiment, we find that large values of col_styles are responsible for changing the background color of the images, as shown in the result of the second experiment. The images are lighter as col_styles is closer to 8, and the images are darker as col style is closer to 12.

Also, as shown in the first experiment, smaller values of col styles affect the contour of the images. While the face of the people looks similar, but the contour of these faces are changes according to the values of col_styles.

Part 3: Deep Q-Learning Network (DQN)

Question 1

The code is shown as follows:

```
def get_action(model, state, action_space_len, epsilon):
    # We do not require gradient at this point, because this function will be used either
    # during experience collection or during inference

    with torch.no_grad():
        Qp = model.policy_net(torch.from_numpy(state).float())
    Q_value, action = torch.max(Qp, axis=0)

    ## TODO: select action and action
    prob = random.random()
    if prob < epsilon:
      action = torch.randint(action_space_len, (1,))
    return action
```

Question 2

The code is shown as follows:

```
def train(model, batch_size):
    state, action, reward, next_state = memory.sample_from_experience(sample_size=batch_size)

    # TODO: predict expected return of current state using main network
    Qp = model.policy_net(state)

    num_state = len(action)
    pred = torch.zeros((num_state))
    for i in range(num_state):
      pred[i] = Qp[i][int(action[i])]

    # TODO: get target return using target network
    Qt = model.target_net(next_state)
    target, next_action = torch.max(Qt, axis = 1)

    # TODO: compute the loss
    loss = model.loss_fn(pred, reward + model.gamma * target)
    model.optimizer.zero_grad()
    loss.backward(retain_graph=True)
    model.optimizer.step()

    model.step += 1
    if model.step % 5 == 0:
        model.target_net.load_state_dict(model.policy_net.state_dict())

    return loss.item()
```

Question 3
Choose the hyperparamtres as follows:

```
# TODO: try different values, it normally takes more than 6k episodes to train
exp_replay_size = 2000
memory = ExperienceReplay(exp_replay_size)
episodes = 12000
epsilon = 1 # episilon start from 1 and decay gradually.
```

Set the decay of epsilon as follows:

```
    # TODO: add epsilon decay rule here!
    epsilon = epsilon * 0.6
```

The resulting loss changes as follows:
See the detailed video in the code for the final training result.