Xiang Chen
March 29, 2021

CSC413 Report

Part 1: Deep Convolutional GAN (DCGAN) [4pt]

[2pt] Implementation:

1. [1pt]

```
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=128, kernel_size=1, stride=2, padding=1, spectral_norm=spectral_norm)
    self.upconv1 = upconv(in_channels=128, out_channels=64, kernel_size=5, stride=2, spectral_norm=spectral_norm)
    self.upconv2 = upconv(in_channels=64, out_channels=32, kernel_size=5, stride=2, spectral_norm=spectral_norm)
    self.upconv3 = upconv(in_channels=32, out_channels=3, kernel_size=5, stride=2, batch_norm=False, spectral_norm=spectral_norm)
```

2. [1pt]

```
# FILL THIS IN
# 1. Compute the discriminator loss on real images
D_real_loss = 0.5 * torch.mean((D(real_images) - 1) ** 2)

# 2. Sample noise
noise = sample_noise(dim, 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 = 0.5 * torch.mean(D(fake_images) ** 2)

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

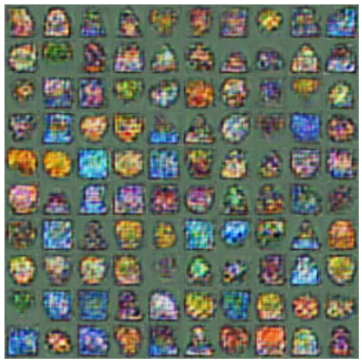
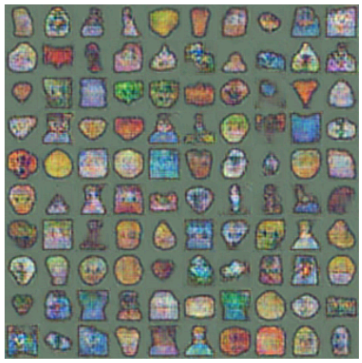
```
# FILL THIS IN
# 1. Sample noise
noise = sample_noise(dim, opts.noise_size)

# 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)
```

[2pt] Experiment:

1. [1pt]


The first image is the output of iteration 1600. The second image is the output of iteration 8200. The third image is the output of iteration 20000. We can see the output of 1600 is really blur and hard to see it is an emoji. The output of 8200 actually has a huge improvement than that of 1600. We can see the emoji becomes more clear but we can still see some artifacts. The output of 20000 is very good. It has less artifacts and the colour is actually really good.

2. [1pt]

After my experiments, I find the output with on for gradient_penalty actually is better than that without on. I am able to stabilize the training. By adding gradient penalty, our model can prevent gradient exploding while training the discriminator, and it can also help stabilize the training.

3. [0pt]

Pass.

Part 2: StyleGAN2-Ada [4pt]

[4pt] Experiments

1. [1pt] Sampling and Identifying Fakes

```
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=0.7, randomize_noise=True, output_transform=fmt)
  return PIL.Image.fromarray(np.concatenate(images, axis=1) , 'RGB')
```

2. [1pt] Interpolation

```
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)
  latent_codes = np.vstack(np.linspace(latent_code_1, latent_code_2, num=INTERPOLATION))
  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')
```

3. [2pt] Style Mixing and Fine Control

```
def generate_from_subnetwork(src_seeds, LATENT_DIMENSION = 512):
    """
    - src_seeds: a list of int, where each int is used to generate a latent code, e.g., [1,2,3]
    - LATENT_DIMENSION: by default 512

    You will complete the code snippet in the Write Your Code Here block
    This generates several images from a sub-network of the genrator.

    To prevent mistakes, we have provided the variable names which corresponds to the ones in the StyleGAN documentation
    You should use their convention.
    """

    # default arguments to Gs.components.synthesis.run, this is given to you.
    synthesis_kwargs = {
        'output_transform': dict(func=tflib.convert_images_to_uint8, nchw_to_nhwc=True),
        'randomize_noise': False,
        'minibatch_size': 4
    }
    ############################################################################
    ######################### WRITE YOUR CODE HERE #############################
    ############################################################################
    truncation = 0.7
    src_latents = np.stack(np.random.RandomState(seed).randn(Gs.input_shape[1]) for seed in src_seeds)
    src_dlatents = Gs.components.mapping.run(src_latents, None)
    w_avg = Gs.get_var('dlatent_avg')
    src_dlatents = w_avg + (src_dlatents - w_avg) * 0.7
    all_images = Gs.components.synthesis.run(src_dlatents, **synthesis_kwargs)
    ############################################################################
    return PIL.Image.fromarray(np.concatenate(all_images, axis=1) , 'RGB')
```

```
truncation = 0.7
src_latents = np.stack(np.random.RandomState(seed).randn(Gs.input_shape[1]) for seed in src_seeds)
src_dlatents = Gs.components.mapping.run(src_latents, None)
w_avg = Gs.get_var('dlatent_avg')
src_dlatents = w_avg + (src_dlatents - w_avg) * 0.7

all_images = all_images = Gs.components.synthesis.run(src_dlatents, **synthesis_kwargs)
```

From two set images above, we can see when we have a larger value for col_style, the image becomes more fine in details. In the contrast situation, if we have a smaller value for col_style, our images will become coarse. We can also see when we have a larger value for col_style, we will have more features.

Part 3: Deep Q-Learning Network (DQN) [4pt]

1. [1pt] Implementation of ϵ — greedy

```
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
    A = None
    s = random.random()
    if s > epsilon:
        A = action
    else: A = torch.tensor(random.randint(0, action_space_len - 1))
    return A
```

2. [2pt] Implementation of DQN training step

```
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
    predictVal = model.policy_net(state)
    Qp = torch.gather(predictVal, 1, torch.unsqueeze(action, 1).to(torch.int64))[:,0]
    # TODO: get target return using target network
    targetVal = model.target_net(next_state)
    Qt = torch.max(targetVal, axis=1)[0]
    Qt = reward + Qt * model.gamma
    # TODO: compute the loss
    loss = model.loss_fn(Qt, Qp)
    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()
```

3. [1pt] Train your DQN Agent