Programming Assignment 4: DCGAN, StyleGAN and DQN

Part 1: Deep Convolutional GAN (DCGAN)

1. **Generator:** Implementation of *DCGenerator*

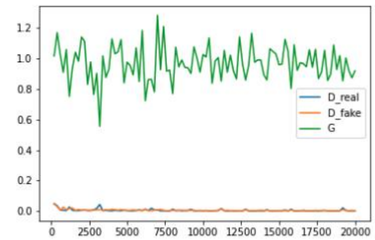
```python
class DCGenerator(nn.Module):
    def __init__(self, noise_size, conv_dim, spectral_norm=False):
        super(DCGenerator, self).__init__()

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

        self.linear_bn = upconv(in_channels=noise_size, 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)


    def forward(self, z):
        """Generates an image given a sample of random noise.

            Input
            -----
                z: BS x noise_size x 1 x 1   -->  BSx100x1x1 (during training)

            Output
            ------
                out: BS x channels x image_width x image_height  -->  BSx3x32x32 (during training)
        """
        batch_size = z.size(0)

        out = F.relu(self.linear_bn(z)).view(-1, self.conv_dim*4, 4, 4)    # BS x 128 x 4 x 4
        out = F.relu(self.upconv1(out))  # BS x 64 x 8 x 8
        out = F.relu(self.upconv2(out))  # BS x 32 x 16 x 16
        out = F.tanh(self.upconv3(out))  # BS x 3 x 32 x 32

        out_size = out.size()
        if out_size != torch.Size([batch_size, 3, 32, 32]):
            raise ValueError("expect {} x 3 x 32 x 32, but get {}".format(batch_size, out_size))
        return out
```

2. **Training Loop:** Implementation of *gan_training_loop*

```python
        for d_i in range(opts.d_train_iters):
            d_optimizer.zero_grad()

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

            # ---- Gradient Penalty ----
            if opts.gradient_penalty:
                alpha = torch.rand(real_images.shape[0], 1, 1, 1)
                alpha = alpha.expand_as(real_images).cuda()
                interp_images = Variable(alpha * real_images.data + (1 - alpha) * fake_images.data, requires_grad=True).cuda()
                D_interp_output = D(interp_images)

                gradients = torch.autograd.grad(outputs=D_interp_output, inputs=interp_images,
                                                grad_outputs=torch.ones(D_interp_output.size()).cuda(),
                                                create_graph=True, retain_graph=True)[0]
                gradients = gradients.view(real_images.shape[0], -1)
                gradients_norm = torch.sqrt(torch.sum(gradients ** 2, dim=1) + 1e-12)

                gp = gp_weight * gradients_norm.mean()
            else:
                gp = 0.0

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

            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()
```

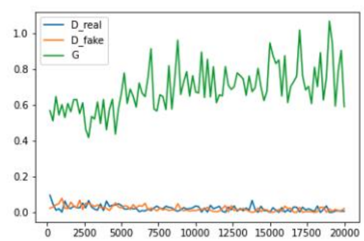
3. Experiment

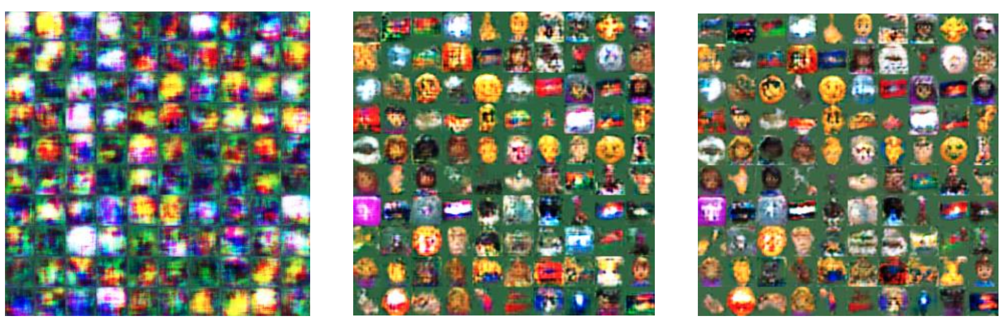
Without gradient_penalty:



The images are captured at iteration 200, 10000 and 20000, respectively from left to right. At the beginning, the image contains a lot of noises since the network hasn't learned anything yet; in the middle of training, the emojis are showing up and having distinguishable colors but some emojis are still not identifiable; at the end of training, almost all the emojis are showing up the correct shapes, especially for the 'faces' emojis, eyes and nose are more clearly showed.

With gradient_penalty:





With gradient penalty, the generated images do not seem to be more stable… By looking at the loss curve, it seems to be more fluctuating than before, and the loss is trending up as iterations go up. In theory, gradient penalty penalizes the exploding gradients to make the gradients small during training by penalizing the norm of the gradient w.r.t so that the gradients will never be zero between fake and real images in the discriminator and the generator can always learn.

Part 2: StyleGAN2-Ada

1. Sampling and Identifying Fakes
   a. *Generate_latent_code*

```python
# 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
```

   b. *Generate_images*

```python
# Sample images from your latent codes https://github.com/NVlabs/stylegan
# You can use their default settings

################################################################################
########################### COMPLETE THE FOLLOWING ##############################
################################################################################
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,512)

  # 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')
################################################################################
```

2. Interpolation

```python
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,512)
  latent_code_2 = generate_latent_code(SEED2,BATCH,512)
  fmt = dict(func=tflib.convert_images_to_uint8, nchw_to_nhwc=True)
  latent_codes = np.vstack([r*latent_code_1 + (1-r)*latent_code_2 for r in np.linspace(0,1,INTERPOLATION)])
  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. Style Mixing and Fine Control

- Step 1

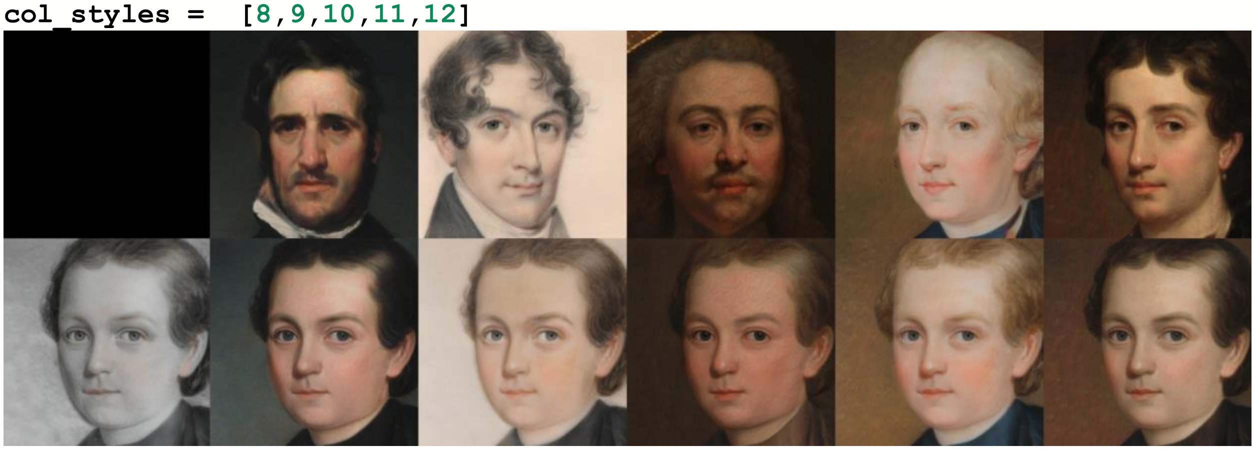
```python
def generate_from_subnetwork(src_seeds, LATENT_DIMENSION = 512):
    """
    - src_seeds: a list of int, where each int is used to generate a latent code, e.g., [1,2,3]
    - LATENT_DIMENSION: by default 512

    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(LATENT_DIMENSION) 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) * truncation
    all_images = Gs.components.synthesis.run(src_dlatents, **synthesis_kwargs)
    ############################################################################
    return PIL.Image.fromarray(np.concatenate(all_images, axis=1) , 'RGB')
```

- Step 2

```python
col_seeds = [1, 2, 3, 4, 5]
row_seeds = [6]
col_styles =  [0,2,4,8,12]
################################################################################
src_seeds = list(set(row_seeds + col_seeds))

# default arguments to Gs.components.synthesis.run, do not change
synthesis_kwargs = {
    'output_transform': dict(func=tflib.convert_images_to_uint8, nchw_to_nhwc=True),
    'randomize_noise': False,
    'minibatch_size': 4
}
################################################################################
########################## COMPLETE THE FOLLOWING ##############################
################################################################################
# Copy the   #### WRITE YOUR CODE HERE #### portion from generate_from_subnetwork()
truncation = 0.7
src_latents =  np.stack(np.random.RandomState(seed).randn(512) 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) * truncation
all_images = Gs.components.synthesis.run(src_dlatents, **synthesis_kwargs)
################################################################################

# (Do not change)
image_dict = {(seed, seed): image for seed, image in zip(src_seeds, list(all_images))}
w_dict = {seed: w for seed, w in zip(src_seeds, list(src_dlatents))}

# Generating Images (Do not Change)
for row_seed in row_seeds:
    for col_seed in col_seeds:
        w = w_dict[row_seed].copy()
        w[col_styles] = w_dict[col_seed][col_styles]
        image = Gs.components.synthesis.run(w[np.newaxis], **synthesis_kwargs)[0]
        image_dict[(row_seed, col_seed)] = image

# Create an Image Grid (Do not Change)
def create_grid_images():
  _N, _C, H, W = Gs.output_shape
  canvas = PIL.Image.new('RGB', (W * (len(col_seeds) + 1), H * (len(row_seeds) + 1)), 'black')
  for row_idx, row_seed in enumerate([None] + row_seeds):
      for col_idx, col_seed in enumerate([None] + col_seeds):
          if row_seed is None and col_seed is None:
              continue
          key = (row_seed, col_seed)
          if row_seed is None:
              key = (col_seed, col_seed)
          if col_seed is None:
              key = (row_seed, row_seed)
          canvas.paste(PIL.Image.fromarray(image_dict[key], 'RGB'), (W * col_idx, H * row_idx))
  return canvas

# The following code will create your image, save it as a png, and display the image
# Run the following code after you have set your row_seed, col_seed and col_style
image_grid = create_grid_images()
image_grid.save('image_grid.png')
im = Image.open("image_grid.png")
im
```

```
col_styles =  [1,3,5,7,9]
```

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



Compare two screenshots above, it clearly shows that different col_styles will generate different styles of pictures. Two sources latent codes are generated: one for the pictures in the first row and another one for the first graph at the second row ('child'), other pictures are generated by copying a portion of the styles from first row and apply to the 'child'.

With low values of col_styles, it copied some coarse spatial resolutions from first row which means the generated graphs can only capture some high-level styles such as shape of face, hair style and eyebrows; with higher values, the generated pictures capture some fine details such as colors and some microstructures in eyes.

Part 3: Deep Q-Learning Network (DQN)

1. 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
    sample = random.random()
    if sample > epsilon:
      return action
    else:
      return torch.randint(0,action_space_len,(1,))
```

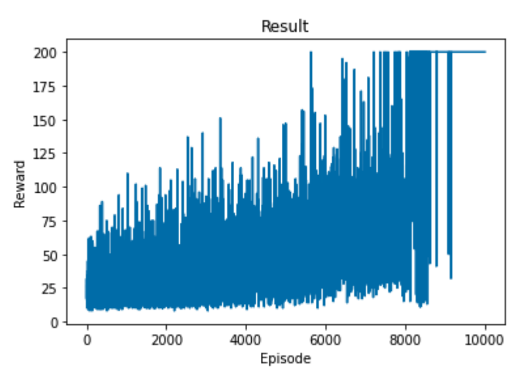
2. Implementation of DQN training step

```
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
    state_action_values, _ = torch.max(model.policy_net(state), axis = 1)

    # TODO: get target return using target network
    Q_next, _ = torch.max(model.target_net(next_state), axis=1)
    next_state_action_values = Q_next * model.gamma + reward

    # TODO: compute the loss
    loss = model.loss_fn(state_action_values, next_state_action_values)

    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. Train your DQN Agent

Training Result: it worked!

```
100%|██████████| 300/300 [00:17<00:00, 17.36it/s]
average reward per episode : 200.0
```



Hyperparameters:

```
# Create the model
env = gym.make('CartPole-v0')
input_dim = env.observation_space.shape[0]        #4
output_dim = env.action_space.n                   #2
agent = DQN_Network(layer_size_list=[input_dim, 64, output_dim], lr=1e-3)

# Main training loop
losses_list, reward_list, episode_len_list, epsilon_list = [], [], [], []

# TODO: try different values, it normally takes more than 6k episodes to train
exp_replay_size = 256
memory = ExperienceReplay(exp_replay_size)
episodes = 10000
epsilon = 1 # episilon start from 1 and decay gradually.

# initiliaze experiance replay
index = 0
for i in range(exp_replay_size):
    obs = env.reset()  #len(obs)=4
    done = False
    while not done:
        A = get_action(agent, obs, env.action_space.n, epsilon=1)
        obs_next, reward, done, _ = env.step(A.item())
        memory.collect([obs, A.item(), reward, obs_next])
        obs = obs_next
        index += 1
        if index > exp_replay_size:
            break

index = 128
for i in tqdm(range(episodes)):
    obs, done, losses, ep_len, rew = env.reset(), False, 0, 0, 0
    while not done:
        ep_len += 1
        A = get_action(agent, obs, env.action_space.n, epsilon)
        obs_next, reward, done, _ = env.step(A.item())
        memory.collect([obs, A.item(), reward, obs_next])

        obs = obs_next
        rew += reward
        index += 1

        if index > 128:
            index = 0
            for j in range(4):
                loss = train(agent, batch_size=16)
                losses += loss

    # TODO: add epsilon decay rule here!
    epsilon = max(0.05, epsilon - 0.0001)

    losses_list.append(losses / ep_len), reward_list.append(rew)
    episode_len_list.append(ep_len), epsilon_list.append(epsilon)

print("Saving trained model")
agent.save_trained_model("cartpole-dqn.pth")
```

```
100%|██████████| 10000/10000 [01:22<00:00, 121.22it/s]Saving trained model
```