Programming Assignment 4

Part 1: Deep Convolutional GAN (DCGAN)

Generator Implementation

```
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 = nn.Sequential(
            nn.Flatten(),
            nn.Linear(noise_size, conv_dim*4*4*4, bias=False)
        )
        self.upconv1 = upconv(in_channels=conv_dim*4, out_channels=conv_dim*2, kernel_size=5)
        self.upconv2 = upconv(in_channels=conv_dim*2, out_channels=conv_dim, kernel_size=5)
        self.upconv3 = upconv(in_channels=conv_dim, out_channels=3, kernel_size=5, batch_norm=False)

    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
```

Training Loop Implementation

```
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 = 1/2*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 =  1/2*torch.mean((D(fake_images))**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 = 1/2*torch.mean((D(fake_images)-1)**2)

G_loss.backward()
g_optimizer.step()

# Print the log info
if iteration % opts.log_step == 0:
    losses['iteration'].append(iteration)
    losses['D_real_loss'].append(D_real_loss.item())
    losses['D_fake_loss'].append(D_fake_loss.item())
    losses['G_loss'].append(G_loss.item())
    print('Iteration [{:4d}/{:4d}] | D_real_loss: {:6.4f} | D_fake_loss: {:6.4f} | G_loss: {:6.4f}'.format(
        iteration, total_train_iters, D_real_loss.item(), D_fake_loss.item(), G_loss.item()))
```

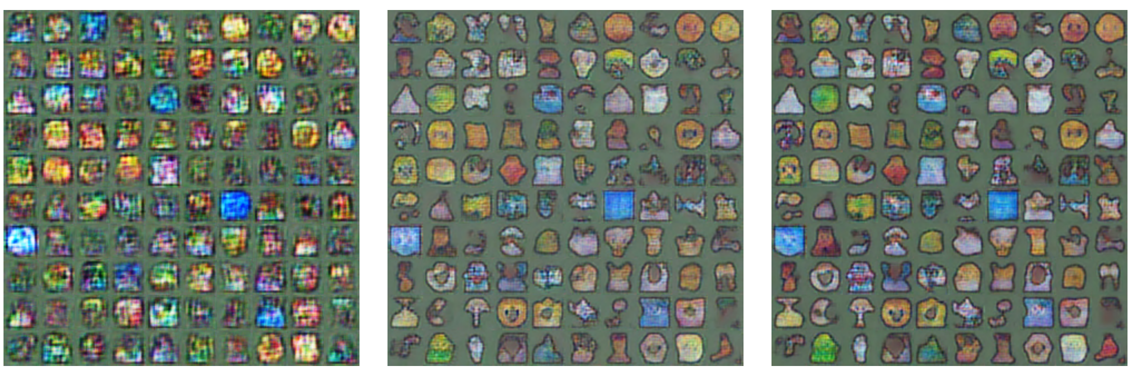
Experiments

1. The samples get better with more iterations - the images start off very pixelated and there is low contrast between the emoji and background, but with each iteration, the image becomes clearer with less pixelation and higher contrast.

(a) Iteration 1000 (b) Iteration 10000 (c) Iteration 20000

Figure 1: Sample progression over iterations without gradient penalty

2. Gradient penalty improves stability of the training as it prevent the gradient from exploding, improves generalization of the model, and guarantees convergence [1]. Comparing the images below with the images in Figure 1, we can see that the emojis become clearer with less iterations.



(a) Iteration 1000 (b) Iteration 10000 (c) Iteration 20000

Figure 2: Sample progression over iterations with gradient penalty

Part 2: StyleGAN2-Ada

Experiments

1.

```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
```

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

2.

```
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)
  interpolated = np.vstack(np.linspace(latent_code_1, latent_code_2, INTERPOLATION))
  fmt = dict(func=tflib.convert_images_to_uint8, nchw_to_nhwc=True)
  images = Gs.run(interpolated, None, truncation_psi=TRUNCATION, randomize_noise=True, output_transform=fmt)

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

Figure 3: Interpolation results

3.

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

Figure 4: With col_styles = [1,2,3,4,5]

Figure 5: With col_styles = [8,9,10,11,12]

Changing the col_styles values changes which details from the other image to imitate. For example, Figure 4 was generated with col_styles = [1,2,3,4,5] and so coarser details like the shape, angle, and size were emulated. However, as Figure 5 shows, with col_styles = [8,9,10,11,12] shows, finer details such as colour and texture were emulated.

Part 3: Deep Q-Learning Network (DQN)

Experiments

1.

```
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
    if (np.random.random() < epsilon):
      action = randint(0, env.action_space.n, (1,))
      return action
    return action
```

2.

```
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

    exp_ret = model.policy_net(state).gather(1, action.unsqueeze(1).long()).squeeze()


    # TODO: get target return using target network

    target_ret = model.target_net(next_state).max(1)[0]


    # TODO: compute the loss
    r = reward + model.gamma * target_ret
    loss = model.loss_fn(exp_ret, r)

    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.

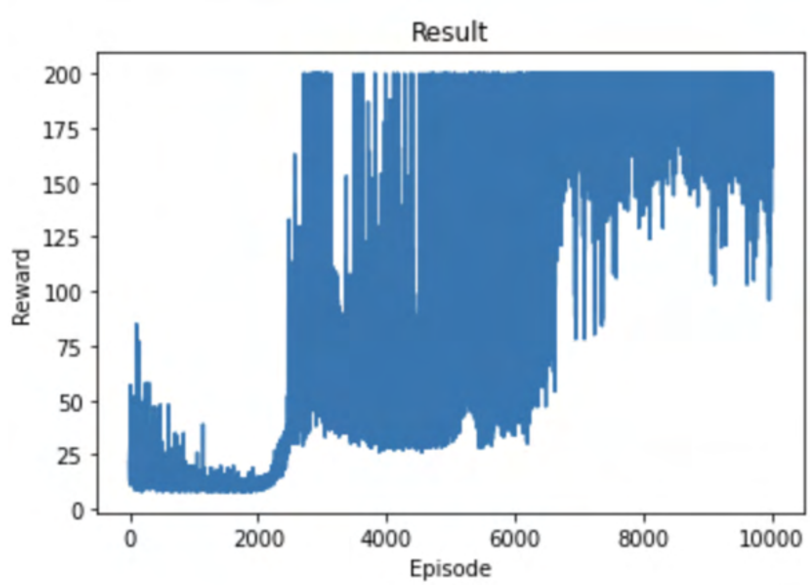
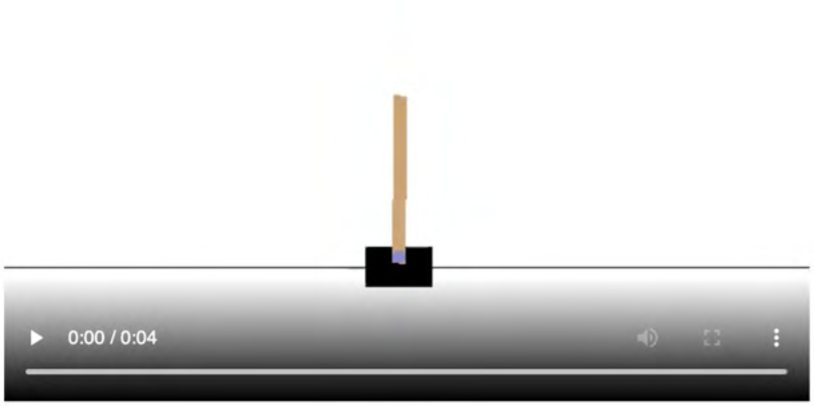


Figure 6: Reward vs. episode for trained model

```
100%|██████████| 300/300 [00:18<00:00, 16.44it/s]average reward per episode : 199.82333333333332
```



References

[1] H. Thanh-Tung, T. Tran, and S. Venkatesh, "Improving generalization and stability of generative adversarial networks," 2019. arXiv: 1902.03984 [cs.LG].