Programming Assignment 4

1 Part 1: Deep Convolutinal GAN (DCGAN)

1.1 DCGAN generator implementation

Look at figure 1

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

        self.conv_dim = conv_dim
        ########################################
        self.linear_bn = upconv(noise_size, self.conv_dim*4*4*4, kernel_size=1, stride=1, padding = 0, batch_norm= True)
        self.upconv1 = upconv(self.conv_dim*4, conv_dim*2, kernel_size= 5, spectral_norm= spectral_norm)
        self.upconv2 = upconv(self.conv_dim*2, conv_dim, kernel_size= 5, spectral_norm= spectral_norm)
        self.upconv3 = upconv(self.conv_dim, 3, kernel_size= 5, batch_norm=False, spectral_norm=spectral_norm)
        ########################################

        # self.linear_bn = ...
        # self.upconv1 = ...
        # self.upconv2 = ...
        # self.upconv3 = ...
```

Figure 1:

1.2 Training Loop implementation

Look at figure 2 and 3

1.3 Experiment 1

Look at Figures 4, 5 and 6. Figrues 4, 5 and 6 show the generated out images (Windows Emojis) by our generator at iteration 200, 10000 and 20000 respectively. Looking at the generated output images and the way the loss changes over iterations, we can see that the generator performance becomes better over time; we can see that the Generator Loss is higher than the Discriminator Loss. This is exactly how we expect a good Generator to perform, i.e. it tries to 'fool' the discriminator by making it misclassify fake generated images as real which leads to an adversarial effect where generator is trying to fool the discrimator and the discrimator is trying perform its best, i.e. minimize the loss.

Furthermore, if we look at figure 4, which shows generated images at the 400th iteration, we can see that the images look like nothing more than colorful blobs with no definite structure of meaning; We can see that the quality of these images improves over time as at the 10000th iteration (figure 5), the images look much better than the ones at the 400th iteration. Finally comparing all 3 images, i.e. the images generated at iteration 20000 (figure 6) to the rest, we can clearly see how the images have become better over time where images in figure 6 seems to show a very clear improvement (improvement in definite structure and clear representation of emojis) from images in figures 4 and 5.

```
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 = 0.5 * (torch.mean((D(real_images)- 1)**2))

    # 2. Sample noise
    noise = fixed_noise[torch.randint(100, (real_images.shape[0],))]


    # 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))

    # ---- 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()

########################################
```

Figure 2: Discriminator Loss

```
    ##########################################
    ###          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()
```

Figure 3: Generator Loss

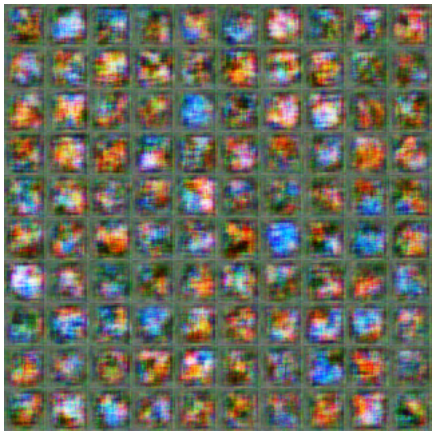

Figure 4: Generated Image at iteration 400

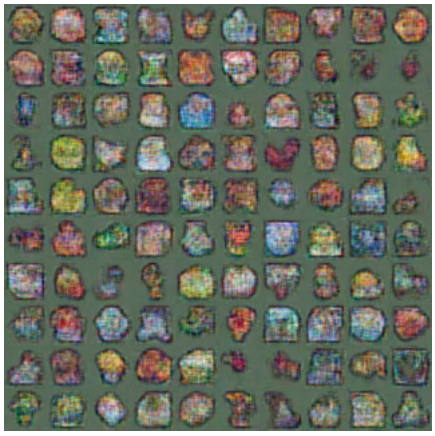

Figure 5: Generated Image at iteration 10000

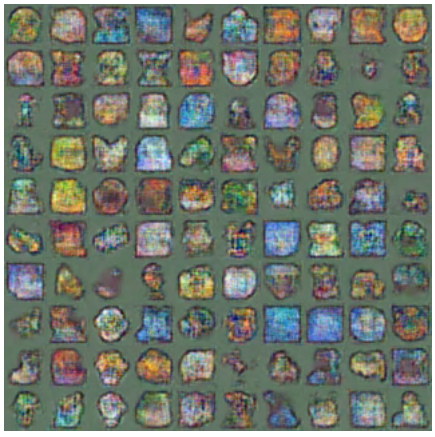

Figure 6: Generated Image at iteration 20000

1.4 Experiment 2

Looking at figure 7 which shows the output generated (at the 20000th iteration) by the generator with gradient penalty turned on, we can see the output looks somewhat better than the output

images shown in figure 6 which shows output image at 20000th iteration when gradient penalty is turned False. Turning gradient penalty on seems to give us generated images with a somewhat sharper color scheme, and sharper features; looking at the images side by side and making comparison shows the subtle differences in the generated image which give us idea about the fact that turning gradient penalty on improves our performance as we get sharper, better looking images.

Looking at the loss plot, the training stabilizes (with presence of some general stochasticity) as number of iterations go up. There is no absolute convergence given the nature of GANs but still, we can say that on a general level, the training stabilizes. Gradient penalty limits the norm of the gradient so it can improve performance by tackling the problem of exploding gradients (which can ofcourse occur in GANs given the presence of stochasticity and noise). Gradient penalty can stabilize loss minimization, i.e., it can decrease oscillation when we approach the minima and in a lot of cases, it can guarantee convergence. Furthermore, it even penalizes generated examples that belong to the 'tails' of our generation distribution, i.e. the more 'unlikely' generated examples. The gradient penalty overall improves the generalization and convergence in training for our model.

Figure 7: Generated image with gradient penalty on

2 Part 2: StyleGAN2-Ada

2.1 Sampling and Identifying Fakes

Look at figures 8 and 9

```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
```

Figure 8:

```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 = SEED, BATCH= 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')
################################################################################
```

Figure 9:

2.2 Interpolation

Look at figures 10 and 11

```
##########################################################################
######################### COMPLETE THE FOLLOWING #########################
##########################################################################
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 = np.repeat(generate_latent_code(SEED = SEED1, BATCH = BATCH), INTERPOLATION, axis = 0)
  latent_code_2 = np.repeat(generate_latent_code(SEED = SEED2, BATCH = BATCH), INTERPOLATION, axis = 0)
  interpolate_var = np.linspace(0,1, num=INTERPOLATION).reshape(INTERPOLATION, 1)
  interpolated_latent = (interpolate_var * latent_code_1) + (np.flip(interpolate_var) * latent_code_2)
  fmt = dict(func=tflib.convert_images_to_uint8, nchw_to_nhwc=True)
  images = Gs.run(interpolated_latent, None, truncation_psi=0.7, randomize_noise=True, output_transform=fmt)

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

Figure 10:

Figure 11: Interpolated Images

2.3 Style Mixing and Fine Control

Step 1: Look at figure 12

Step 2: When 'col_style' is set to [1,2,3,4,5], we can see that the 'style mixed' images generated (shown in figure 13) seem to 'combine' features from the row and column images in a way such that the 'features' from image in a row (in the human face example, this refers to the figure of the girl child) has its features 'imposed' (in a manner of speaking) on images (different human faces shown across columns), i.e. the predominant features in the 'style mixed' images seems to come from the column images. There seems to be an involvement of mxing the finer features from the row image to the coarse features from the column images. When 'col_style' set to [8, 9, 10, 11, 12], we can see a different kind of set of style mixed images are generated (shown in 14) where the features from the column images seem to be 'imposed' on the row image (the girl child in our case). There seems to be an involvement of mixing the finer features from the column images to the coarse features of row image. This gives us an idea that these numbers in 'col_style' refer to how/what features are going to be 'mixed' from row and column images to generate the desired 'style mixed' images.

```python
# You will generate images from sub-networks of the StyleGAN generator
# Similar to Gs, the sub-networks are represented as independent instances of dnnlib.tflib.Network
# Complete the function by following \url{https://github.com/NVlabs/stylegan}
# And Look up Gs.components.mapping,  Gs.components.synthesism, Gs.get_var
# Remember to use the truncation trick as described in the handout after you obtain src_dlatents from Gs.components.mapping.run
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 12:

Figure 13:

Figure 14:

3 Deep Q-Learning Network (DQN)

3.1 Implementation of ϵ-greedy

Look at figure 15

```
[ ]  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 torch.rand(1) <= epsilon:
           action = torch.randint(action_space_len, (1,))
         else:
           action = action

         return action
```

Figure 15:

3.2 Implementation of DQN traning step

Look at figure 16

```
[ ]  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
         action_ind = action.view(batch_size, 1).long()
         all_q = model.policy_net(state)
         q_value = torch.gather(all_q, 1, index = action_ind)

         # TODO: get target return using target network
         qr = model.target_net(next_state)
         expected_return, _ = torch.max(qr, axis = 1)
         expected_return = expected_return.view(batch_size, 1)


         # TODO: compute the loss
         loss = model.loss_fn(q_value, model.gamma*expected_return + reward)


         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()
```

Figure 16:

3.3 Train your DQN Agent

Look at figures 17, 18, 19 and 20

```
# Create the model
env = gym.make('CartPole-v0')
input_dim = env.observation_space.shape[0]
output_dim = env.action_space.n
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 = 64
memory = ExperienceReplay(exp_replay_size)
episodes = 13000
epsilon = 1 # episilon start from 1 and decay gradually.

# initiliaze experiance replay
index = 0
for i in range(exp_replay_size):
    obs = env.reset()
    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
```

Figure 17: Hyperparameter tuning

```
# TODO: add epsilon decay rule here!
if (i%1000 == 0) and (i != 0):
  epsilon = epsilon * 0.60
  #print(epsilon)
```

Figure 18: Epsilon decay implementation

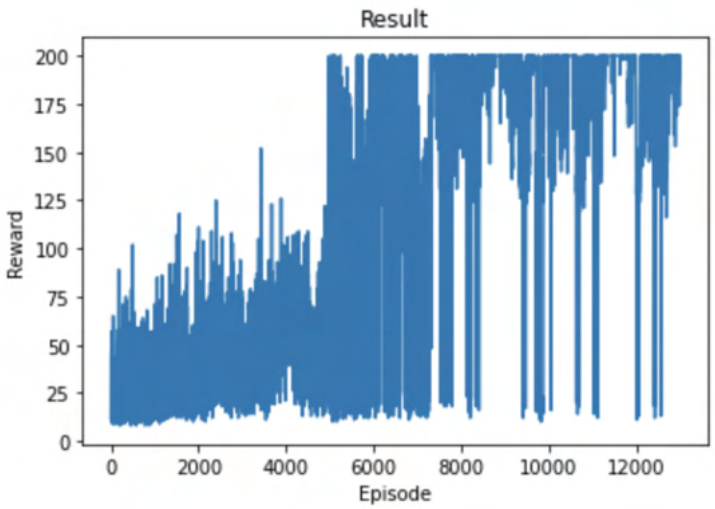

Figure 19: Generated reward plot

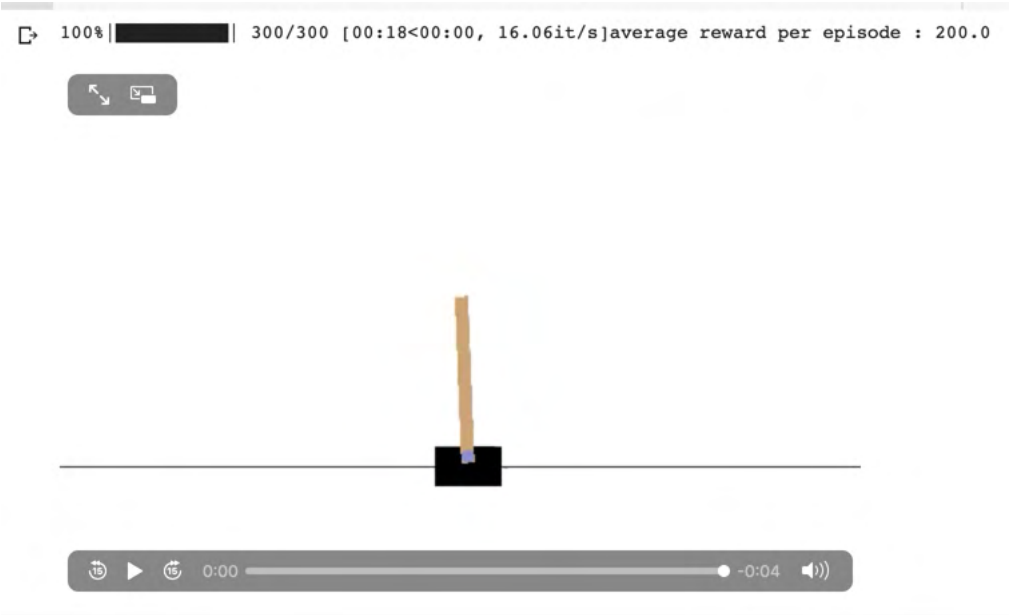

Figure 20: Generated video (agent playing game)