413 PA4

polo.li

March 2021

1 Deep Convolutional GAN

1.1 Implementation Part

1.1.1 DCGenerator

Note that although in the archtechture graph the last upcov layer does not contain a batchnorm layer i.e. *batch_norm* should be $False$ for *upconv*3, the predicted graph is brighter and higher quality with a batchnorm layer, so we set *batch_norm* to be $True$.

```
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, conv_dim*4, kernel_size=3, spectral_norm=spectral_norm)
        self.upconv1 = upconv(conv_dim*4, conv_dim*2, kernel_size = 5, spectral_norm=spectral_norm)
        self.upconv2 = upconv(conv_dim*2, conv_dim, kernel_size = 5, spectral_norm=spectral_norm)
        self.upconv3 = upconv(conv_dim, 3, kernel_size=5, spectral_norm=spectral_norm)
```

1.1.2 Training Loop

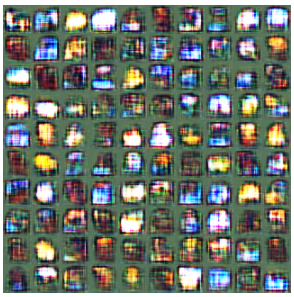
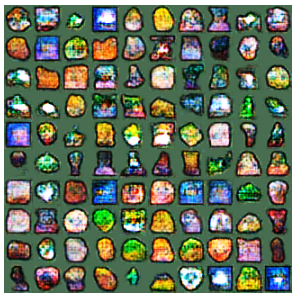
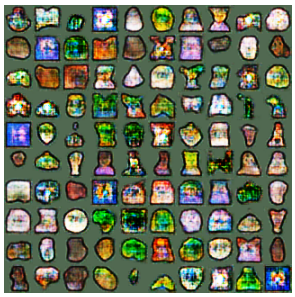
```
    m = real_images.shape[0]
    # ones = Variable(torch.Tensor(real_images.shape[0]).float().cuda().fill_(1.0), requires_grad=False)
    for d_i in range(opts.d_train_iters):
        d_optimizer.zero_grad()

        # 1. Compute the discriminator loss on real images
        D_real_loss = 1/(2*m) * torch.sum((D(real_images) - 1)**2,axis=0)

        # 2. Sample noise
        noise = sample_noise(m, 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*m) * torch.sum(D(fake_images) ** 2, axis=0)

        # ---- 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()
```

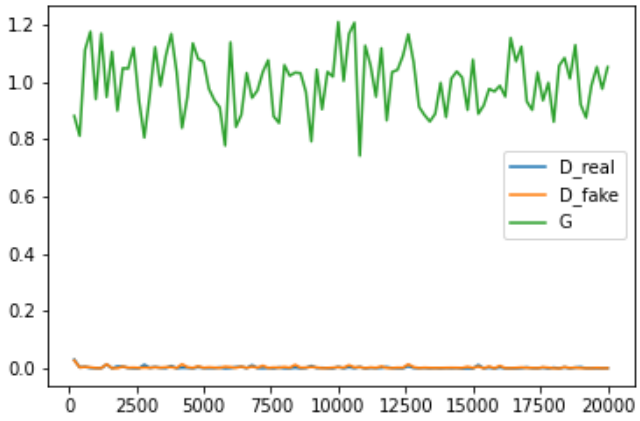
```
g_optimizer.zero_grad()

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

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

# 3. Compute the generator loss
G_loss = 1/m * torch.sum((D(fake_images) - 1) ** 2, axis=0) #
G_loss.backward()
g_optimizer.step()
```

1.2 Experiment Part

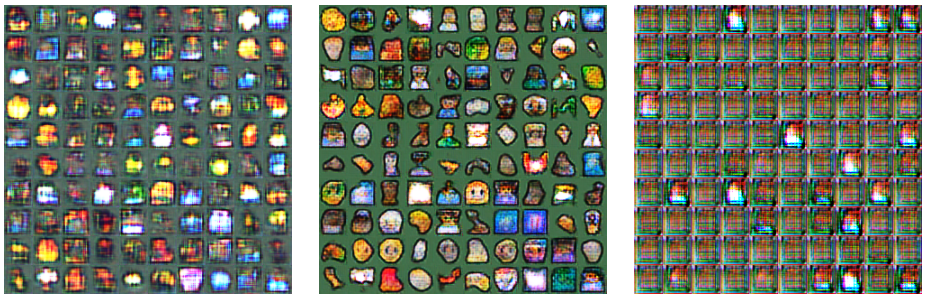
1.2.1

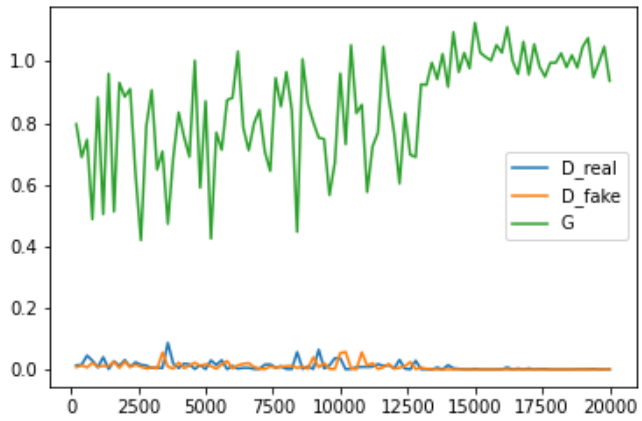
The following images respectively correspond to iteration 2000, 12000 and 20000, and the loss curve when gradient penalty is set to **False**. As can be seen, the produced image quality becomes better and better until a certain point, after which it stabilizes.



The following images respectively correspond to iteration 2000, 12000 and 20000 with gradient penalty set to **True**. As can be seen, the produced image quality becomes better and better until a certain point, after which it makes no sense. This can be due to overfitting.





1.2.2

As can be seen in the loss plots in Q1.1, gradient penalty actually makes the model more unstable and produce low quality result in the end.

Reason why gradient penalty can help:

In "Which Training Methods for GANs do actually Converge?", lemma 3.3 states that, by penalizing the Dirac-GAN, "the eigenvalues of the Jacobian of the gradient vector field at the equilibrium point are given by $\lambda = -\frac{\gamma}{2}a \pm \sqrt{\frac{\gamma^2}{4} - f'(0)^2}$". When $\gamma > 0$, the real part of the eigenvalues are negative, and GAN training converges locally for small enough learning rates.

1.2.3

2 StyleGAN2-Ada

2.1 Sampling and Identifying Fakes

```
# 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
```

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

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.
  With gradient clipping, pre-determined gradient threshold be introduced, and then gradients norms that exceed this
  This prevents any gradient to have norm greater than the threshold and thus the gradients are clipped.

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

2.1.1 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)
  r = np.linspace(0, 1, num=INTERPOLATION)
  print(latent_code_1.shape)
  #generate an array of multiple interpolations between two arrays. vectorized
  # def intpld_arr(arr1, arr2, num_points=INTERPOLATION):
  #   arr1_expanded = np.expand_dims(arr1, axis=0).repeat(num_points,axis=0)
  #   arr2_expanded = np.expand_dims(arr2, axis=0).repeat(num_points,axis=0)
  #   linspace = np.linspace(0, 1, num_points).reshape(num_points,1, 1)
  #   arr_intpld = arr1_expanded * linspace + arr2_expanded * (1 - linspace)
  #   return arr_intpld

  # latent_intplt = np.concatenate( intpld_arr(latent_code_1, latent_code_2), axis=0)
  latent_intplt = np.linspace(latent_code_1, latent_code_2, num=INTERPOLATION).squeeze()
  print(latent_intplt.shape)
  fmt = dict(func=tflib.convert_images_to_uint8, nchw_to_nhwc=True)
  images = Gs.run(latent_intplt, None, truncation_psi=TRUNCATION, randomize_noise=True, output_transform=fmt)

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

```
# Create an interpolation of your generated images
interpolate_images(SEED1 = 412,SEED2 = 413,INTERPOLATION = 6, BATCH=1)
```

```
(1, 512)
(6, 512)
```

2.1.2 Style Mixing and Fine Control

Higher the values in *col_style*, finer the features that will be brought into the original image.(i.e. bigger scale v.s. smaller scale features) The following two sets of pictures respectively correspond to $col\_style = [1, 2, 3, 4, 5]$ and $col\_style = [6, 7, 8, 9, 10]$. We see in the first set of pictures that coarse features like appearance, general faceshape, pose and smiles from the first row are brought in. Whereas in the second set of pictures, more detailed features such as eye expression, hairstyles are brought in.

3 Deep Q-learning Network

3.1

```python
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

    #action_space = {0, 1, 2, ... action_space_len - 1}
    with torch.no_grad():
        Qp = model.policy_net(torch.from_numpy(state).float())
    # print(f'Qp: {Qp}')
    Q_value, max_action = torch.max(Qp, axis=0) #(max, max_indices)
    # print(f'Q_value.shape: {Q_value.shape}, max_action.shape: {max_action.shape}')
    # print(f'Q_value: {Q_value}, max_action: {max_action}')
    # print(f'action_space_len: {action_space_len}')
    random_action = torch.randint(0, action_space_len, (1,))
    p = random.uniform(0,1)
    action = max_action if p > epsilon else random_action
    return action
```

3.2

```python
def train(model, batch_size):
    state, action, reward, next_state = memory.sample_from_experience(sample_size=batch_size)
    # print(f"action: {action.long()}")
    # print(f'state: {state}')
    # print(f'next_state: {next_state}')
    # print(f'reward: {reward.shape}')

    # TODO: predict expected return of current state using main network
    num_turns = action.shape[0]
    exp_ret = model.policy_net(state.float())[torch.arange(num_turns),action.long()]
    # print(f'model.policy_net(state): {model.policy_net(state)}')
    # print(f'model.policy_net(state)[torch.arange(num_turns),action.long()]: {model.policy_net(state)[
    # print(f'model.policy_net(state)[0][action.long().item()]: {model.policy_net(state)[0][action.long
    # print(f'exp_ret: {exp_ret.shape}')
    # print(action.long())

    # TODO: get target return using target network
    # print(model.target_net(next_state).max(axis=-1)[0])
    tar_ret = reward + model.gamma * model.target_net(next_state).max(axis=-1)[0]
    # print(f'tar_ret: {tar_ret.shape}')

    # TODO: compute the loss
    loss = model.loss_fn(exp_ret, tar_ret)
    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()
```

3.3

```
    # TODO: add epsilon decay rule here!
    epsilon = epsilon * 0.99
    losses_list.append(losses / ep_len), reward_list.append(rew)
    episode_len_list.append(ep_len), epsilon_list.append(epsilon)
```

```
# TODO: try different values, it normally takes more than 6k episodes to train
exp_replay_size = 15000
memory = ExperienceReplay(exp_replay_size)
episodes = 10000
epsilon = 1 # episilon start from 1 and decay gradually.
```