CSC413 Programming Assignment 4 April 1, 2021

Part 1: Deep Convolutional GAN (DCGAN)

1. Generator

1.1. The code for this question is shown below.

```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=conv_dim*4, kernel_size=5, stride=4, batch_norm=True, spectral_norm=spectral_norm)
        self.upconv1 = upconv(in_channels=conv_dim*4, out_channels=conv_dim*2, kernel_size=5, stride=2, spectral_norm=spectral_norm)
        self.upconv2 = upconv(in_channels=conv_dim*2, out_channels=conv_dim, kernel_size=5, stride=2, spectral_norm=spectral_norm)
        self.upconv3 = upconv(in_channels=conv_dim, out_channels=3, kernel_size=5, stride=2, batch_norm=False, spectral_norm=spectral_norm)

```

2. Training Loop

2.1. The code for this question is shown below.

```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 ----
                ...

                # -------------------------
                # 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)


```

3. Experiment

3.1. Without gradient penalty.

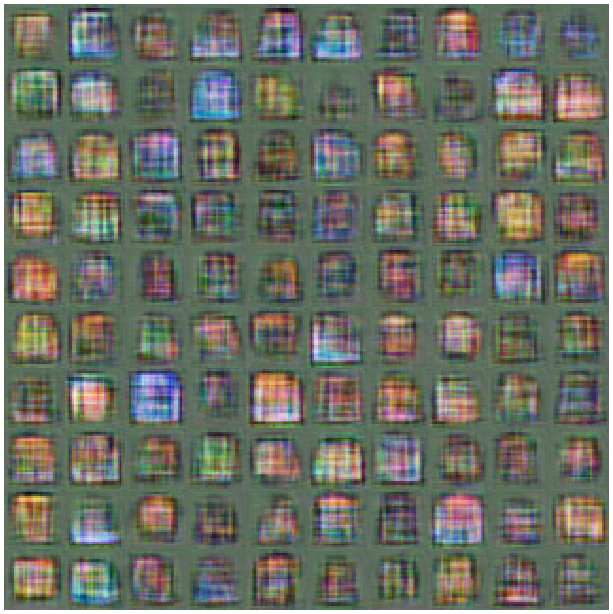

Figure 1: The sample at 1000-th iteration.

Figure 2: The sample at 19200-th iteration.

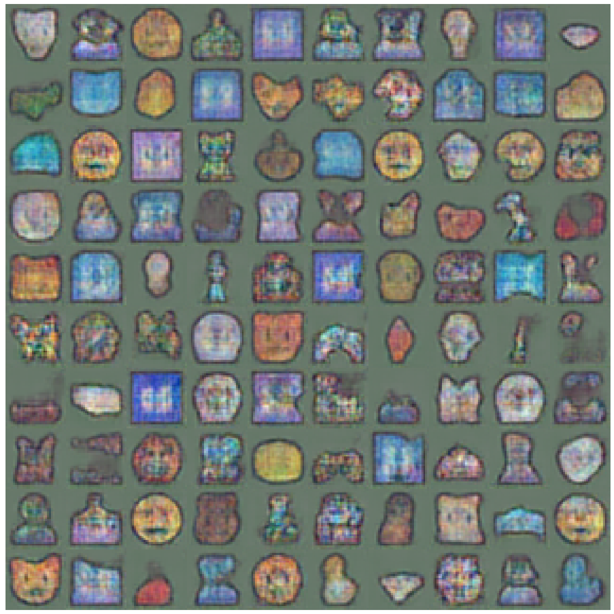

Figure 3: The sample at 20000-th iteration.

Figure 1-3 show the samples of the output of the generator at the 1000-th, 19200-th, and 20000-th iteration. As the number of iteration goes up, we can see that the emoji becomes clearer and more colorful in general. According to Figure 1, we are unable to distinguish each emoji. At the end of training, we can basically see their appearance. For the satisfactory sample (Figure 2), we can nearly see the smile face of some emoji.

3.2. Use gradient penalty.

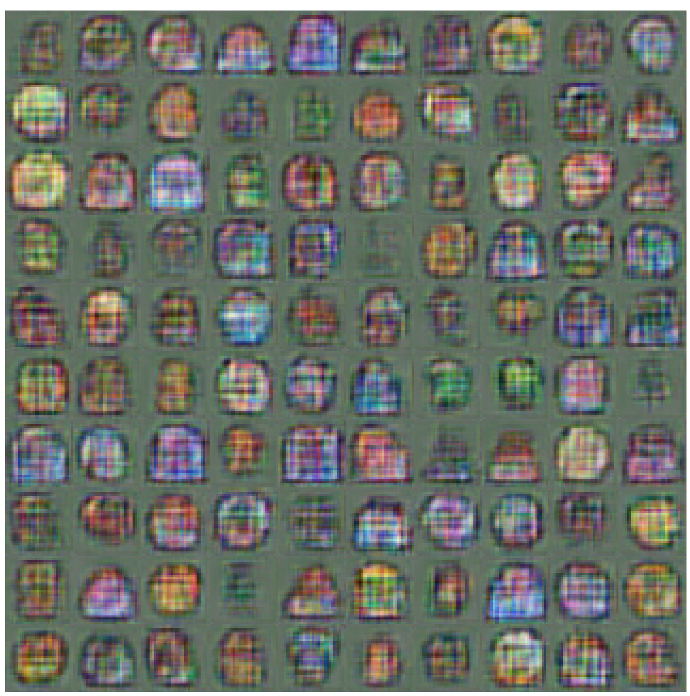

Figure 4: The sample at 1000-th iteration.

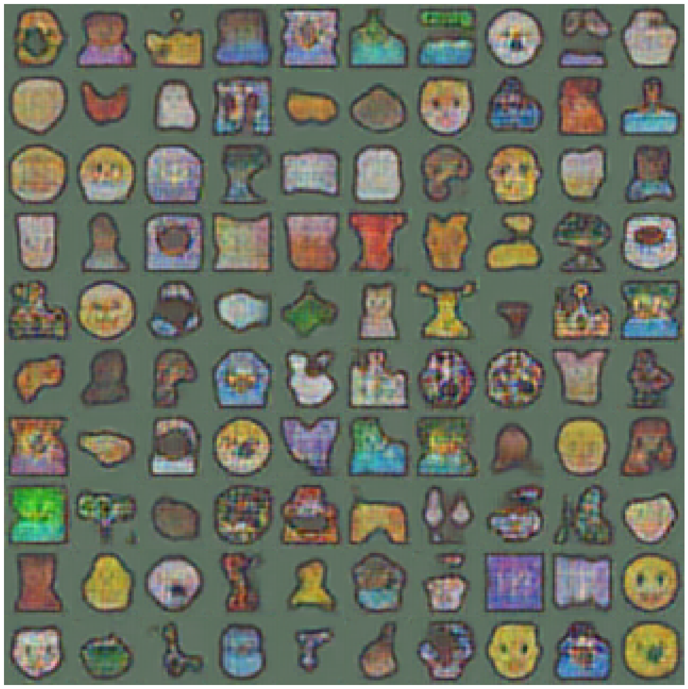

Figure 5: The sample at 18600-th iteration.

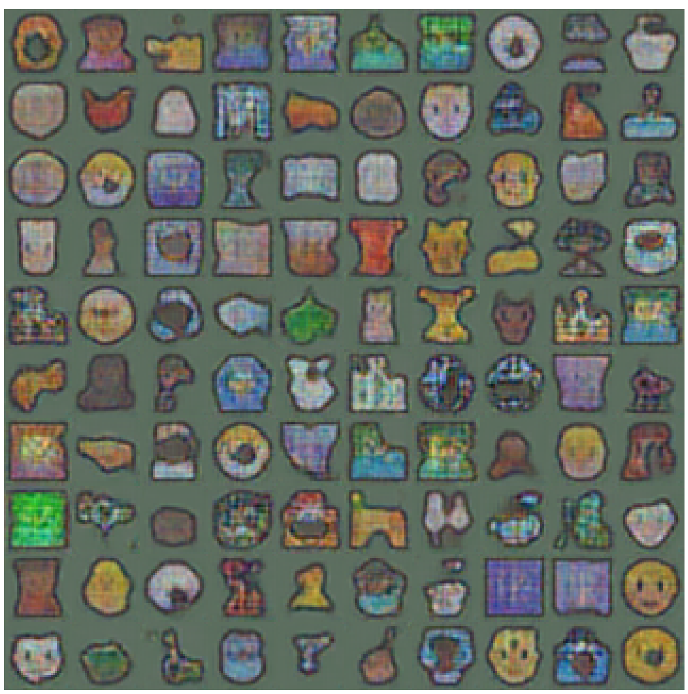

Figure 6: The sample at 20000-th iteration.

Figure 4-6 show the samples of the output of the generator at the 1000-th, 18600-th, and 20000-th iteration. Compared with Figure 1-3, we can see that by applying the gradient penalty, more emoji can be identified with the similar quality. Thus, the model using gradient penalty has a better performance. However, according to Figure 7 and 8, it is difficult to identify which model provide more stabilized training process. Actually, based on the Thanh-

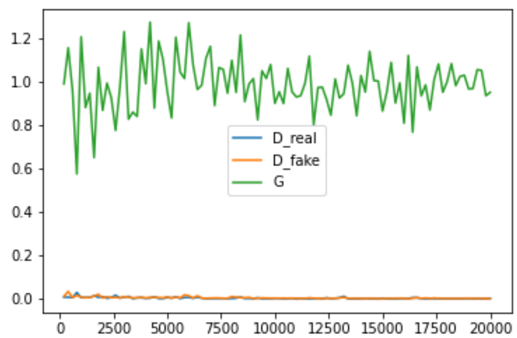

Figure 7: The loss result of the model without using gradient penalty.

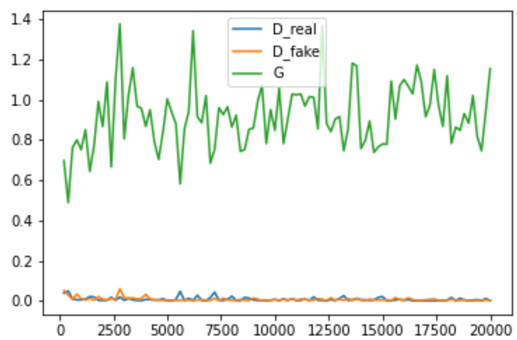

Figure 8: The loss result of the model using the gradient penalty.

Tung et al., (2019)'s paper, we know that since adding the gradient penalty could enforce the potential diverge values to be zero, it helps to improve the generation capability of the discriminator during the training, which could prevent the potential gradient exploding and stabilize the training process.

3.3. (Optional)

Part 2: StyleGAN2-Ada

1. Sampling and Identifying Fakes

   For this part, I choose to use Flickr-Faces-HQ dataset which is about human face. The code for this question is shown below.

```
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

###############################################################################
########################## 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)

  # 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')
###############################################################################
# Generate your images
generate_images(5, 5)

```

2. Interpolation
   The code for this question is shown below.

```
###############################################################################
########################## 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 = generate_latent_code(SEED1, BATCH, Gs.input_shape[1])
    latent_code_2 = generate_latent_code(SEED2, BATCH, Gs.input_shape[1])
    r_lst = np.linspace(0, 1, num=INTERPOLATION)
    fmt = dict(func=tflib.convert_images_to_uint8, nchw_to_nhwc=True)
    interpolated_image = np.zeros((len(r_lst), Gs.input_shape[1]))
    for i in range(len(r_lst)):
      interpolated_image[i, :] = r_lst[i] * latent_code_1 + (1-r_lst[i]) * latent_code_2
    images = Gs.run(interpolated_image, 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(6, 8, 10)

```

Figure 9: The interpolation example.

3. Style Mixing and Fine Control
   The code for Step one is shown below.

```
# 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.synthesis, 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(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')

```

The code for Step two is shown below.

```
##########################################################################
####################COMPLETE THE NEXT THREE LINES##########################
##########################################################################
col_seeds = [1, 2, 3, 4, 5]
row_seeds = [6]
col_styles = [1, 2, 3, 4, 5]
##########################################################################
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(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)
##########################################################################
...
# col_style 2
col_seeds = [1, 2, 3, 4, 5]
row_seeds = [6]
col_styles = [8, 9, 10, 11, 12]

```

Figure 10: The experiment 1.

For my experiment, I choose $1-5$ as my *col_seeds*, and 6 as my *row_seeds*. I tried two column styles $[1, 2, 3, 4, 5]$ and $[8, 9, 10, 11, 12]$ as recommended. Figure 10 show the result using

Figure 11: The experiment 2.

col_styles $= [1, 2, 3, 4, 5]$ and Figure 11 show the result using *col_styles* $= [8, 9, 10, 11, 12]$. Based on these experiments, we figure out that *col_styles* seem like to represent the resolution, which represents different features of the picture in *col_seeds*, where larger number gives lower level of the features. Also, by additional experiments, we found that the number of *col_styles* is also considered. In specific, according to Figure 10, we can see that the most high level features of human face including eyes, nose, mouse, hair style, and face structure in mixed pictures come from the *col_seeds*, while the color of hair, background, and other features are from the *row_seeds*. For Figure 11, we observe that the mixed pictures only take low level features from the *col_seeds* like the color of the background, hair, and clothes, and a little bit of facial expression, and remain most features of human face from the *row_seeds*.

Part 3: Deep Q-Learning Network (DQN)

1. The code for this question is shown below.

```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

    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
    action_2 = torch.randint(0, action_space_len, (1,))
    next = random.choices([action, action_2], weights=[1-epsilon, epsilon], k=1)[0]
    return next

```

2. The code for this question is shown below.

```python
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
    Qp = model.policy_net(state)
    Qsa = Qp.gather(1, action.long().view(-1, 1)).flatten()

    # TODO: get target return using target network
    Qt = model.target_net(next_state)
    Qsta = Qt.max(1)[0]

    # TODO: compute the loss
```

```python
    # print("types q:{}{}, s:{}, a:{}, r:{}, n:{}".format(Qsa.shape, (reward + model.gamma *Qsta).shape, state.shape, action.shape, reward.shape, next_state.shape))
    loss = model.loss_fn(Qsa, reward + model.gamma * Qsta)
    # loss = Qsa - reward + model.gamma * Qsta
    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

```python
# Create the model
...

# 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 = 200
memory = ExperienceReplay(exp_replay_size)
episodes = 8000
rew_thres = 0
epsilon = 1 # episilon start from 1 and decay gradually.

# initiliaze experiance replay
...

    # TODO: add epsilon decay rule here!
    if epsilon > 0.01 and rew >= rew_thres:
      epsilon *= 0.97
      rew_thres += 1

    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")

```

For my epsilon decay rule, I add a new parameter called "rew_thres" to store the current threshold for the reward. If the current epsilon is larger than 0.01 and the current reward is greater than or equal to the reward threshold, then the epsilon would reduce to 0.97 times and the reward threshold would increase one. After tuning hyperparameters many times, I got a good model when *exp_replay_size* = 200, *episodes* = 8000, initial *epsilon* = 1, minimum *episodes* = 0.01, and *epsilon_decay* = 0.97. Figure 12 shows the reward plot and Figure 13 shows the result by the trained model.

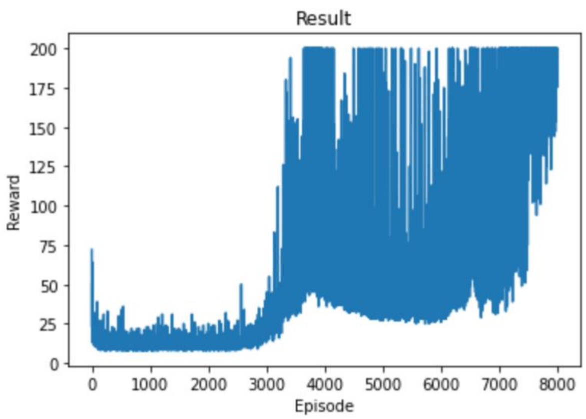

Figure 12: The reward plot of the final trained model.

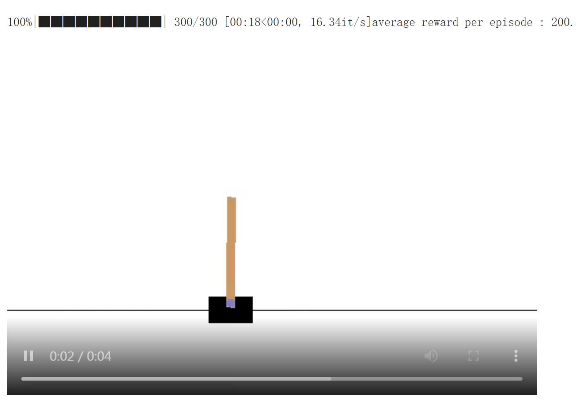

Figure 13: The validation result.