1. Generative adversarial network (GAN) :
   - Network Architecture and Performance:
     - Network Architecture :

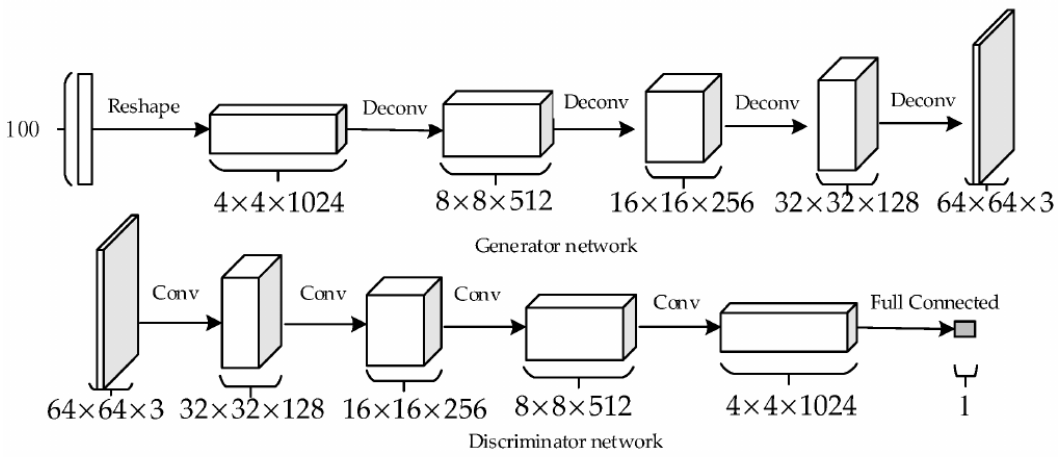
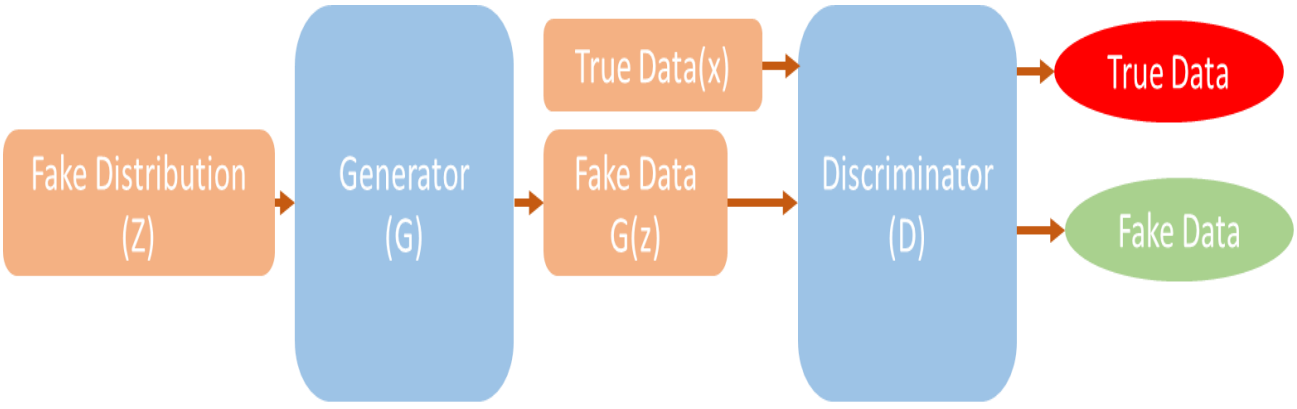


Figure 1: Structure of DCGAN

GAN network is the game between Generator and Discriminator. Target of Generator is aim to generate a fake data then make the Discriminator judges these data as true data. However the mission of the Discriminator is to determine the generating data is fake data and the true data is ground turh. That is we call this procedure is Genrative Adversarial . The following diagram is the whole training procedure for this network.



Generator is aim to make Discriminator determine its generating data as true data

Discriminator is aim to determine generating data from generator as fake data

Discriminator and Generator have different task therefore their loss function should designed in different way to meet their requirement.

- Discriminator Cost :

$$C^{(D)} = -E_{x\sim P_{data}} log\, D(x) - E_{z\sim P_z} log\, (1 - D(G(z)))$$

  - X : true data
  - $D(x)$: Distribution of the true data
  - Z : Fake distribution of data , we can view it as a randomized noise.
  - G(z) : Generator Encode a fake distribution.

- Generator Cost :

$$C^{(G)} = -C^{(D)}$$

Discriminator cost form is a general Cross-Entropy Cost. This cost is aim to make $D(x)$ as large as possible. On the other hand , it also make the$(1 - D(G(z)))$ as small as possible which represents discriminator favors more than generating data.

Our goal is to find the Generator to make a fake data. To summerize the both cost above, we can get the below form :

$$\theta^{(G)} = arg \min_{\theta^{(G)}} \max_{\theta^{(D)}} C^{(G)}$$

The algoritm procedure can represented as the below psudo code :

Implementation

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**
  for k steps **do**
    - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
    - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for
  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

◆ Step of Preprcessing image and Experiment :

- Step 1 :

  Use PIL Package to transform the image to the RGB format.

- Step 2:

  To balance all the training process , I reshape all the image to the same size and unify them to the numpy matrix.

- Step 3 :

  After unifying them into a matirx. I use a random list to shuffle the data in the matrix.

- Step4 :

  Normalizing this data then feed the data to the training procedure.

- Step 5 :

  Initialize the hyperparametor :

  Both discriminator and generator network parametor should be initialized with random normal . There normal distribution are also in different way.

	Mean	Variance
Generator	0.0	0.02
Discriminator	1.0	0.02

  We view the input of generator as a Gaussian noise therefore we should set the mena vaule of it as zero.

- Step 6 :

  Sample $z^1.z^2$ ….z^n and $x^1.x^2$ ….x^n to update the loss :

$$C^{(D)} = -E_{x\sim P_{data}} log\, D(x) - E_{z\sim P_z} log\, (1 - D(G(z)))$$

- Step 7 :

  Sample $z^1.z^2$ ….z^n to update the loss :

$$C^{(D)} = -E_{z\sim P_z} log\, (1 - D(G(z)))$$

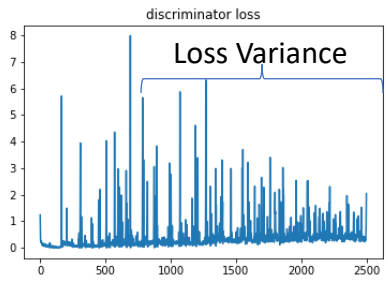
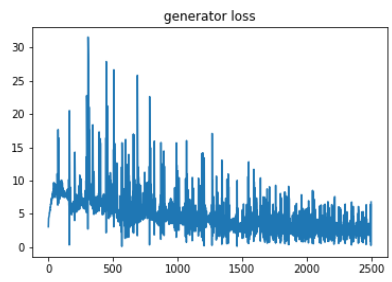
■ Experiment Result & Analysis : (utilize main.py to plot)

◆ Train without image folder :

■ Patametor :

Batch_size	200
learning_rate	0.001
Training_data	10000

■ Loss performance :

Discriminator	Generator
	

Both discriminator and generator have high variance during training procedure. The image array didn't shuffle at first. I guess that I can't only normalize the graph with the following formula.

$$Grapic\ array\ /\ 255$$

This is a roughly normalization. We should delpoly the procedure such as below code.

```
dataset = dset.ImageFolder(root = args.dataroot,
                           transform = transforms.Compose([
                           transforms.Resize(args.image_size),
                           transforms.CenterCrop(args.image_size),
                           transforms.ToTensor(),
                           transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
                           ])
                           )
```

■ Graph Result :

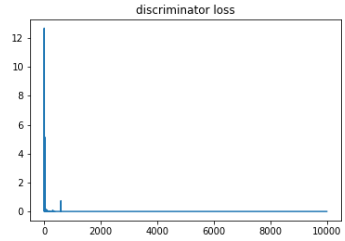
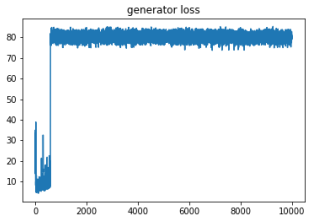
	Example	List of graph
graph		
Analysis	1. Incorrect normalization result in incorrect generation。 2. The transforms.Normalize method is adopt the mean=(0.5,0.5,0.5), std = (0.5,0.5,0.5) which indicate the nomalization is operated in the RGB three dimensions. Comapring to the traditional method. This opration can precisely control the normalization manner of the mean and standard deviation.	

◆ Train without ImageFolder with incorrect parametors :

■ Patametor :

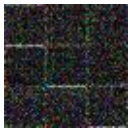
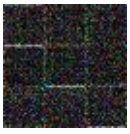
Batch_size	50
learning_rate	0.0001
Training_data	12000

■ Loss performance :

Discriminator	Generator
	

The generator loss remain increasing while loss of discriminator remain zeros. It implies that the generator can not be against with discriminator. The corresponding generating data will be always justified as fake data by the discriminator.

■ Graph Result :

	Example	List of graph
graph		

Analysis

1. I still adopt the incorrect way to normalize the result in this experiment. Addition to this operation, I also increase the training epoch to learn the pattern. It generalize a ovefitting in both discriminator and generator.
2. In order to solve this problem , we should adopt a correct nomalization and precise batch amount to solve such this problem.
3. To increase the generation ability of the generator for being against , we can add some white gaussian noise to the real data to confuse the discriminator to achive the such goal.
4. We can still use the cost function to vrify our observation.
   - Discriminator Cost :

     $$C^{(D)} = -E_{x\sim P_{data}} log\, D(x) - E_{z\sim P_z} log\,(1 - D(G(z)))$$

     Discriminator is powerful to justify the real data that make the $D(x) \cong 1$

     Discriminator is powerful to justify the real data that make the $D(G(z)) \cong 0$ so that the $1 - D(G(z)) \cong 1$
   - Generator Cost :

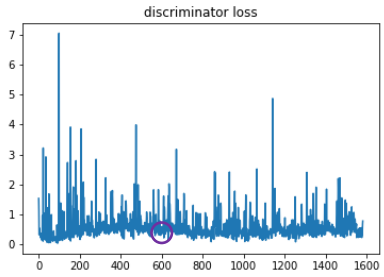
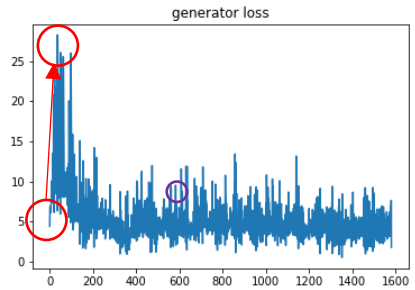
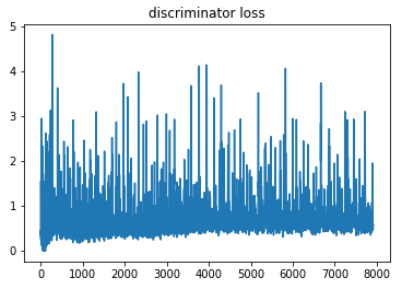
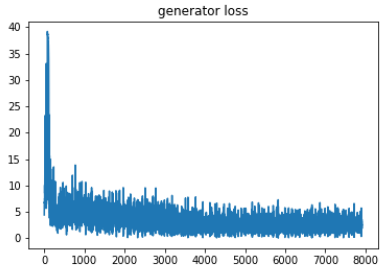
     $$C^{(G)} = -C^{(D)}$$

   This cost will result in opossite manner because of the negtive term.

- Training with imagefolder in lower epoch way :
  - Patametor :

Batch_size	103
learning_rate	0.0002

  - Loss performance :

	Discriminator	Generator
Epoch = 1		
Epoch = 5		

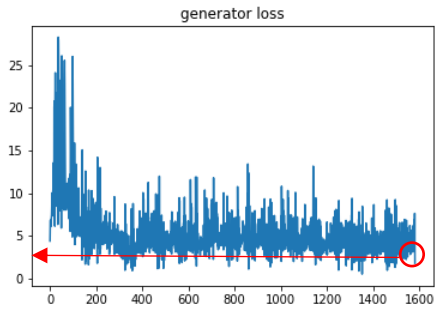
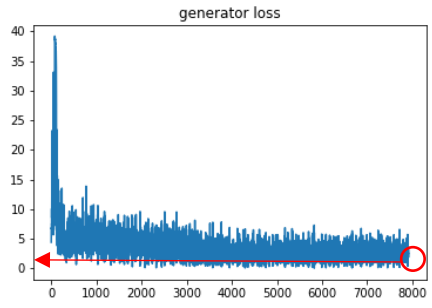
- There exists a interesting phenomenon in this procedure. In the previous network, the losses are always be highest in the initial timming. However, we can observe that the generator loss is low in the beginning of training. This impiles that the generating data can be not justified as fake data by the untrained model. Therefore, the first step of training procedure will make the generator loss is lowest.

- We can also observe that the 600 th step of training. The corresponding loss of generator and discriminator are performed in a opposite way. This physical meaning implies that generating data is powerful enough to be against to the discriminator.

■ Graph Result :

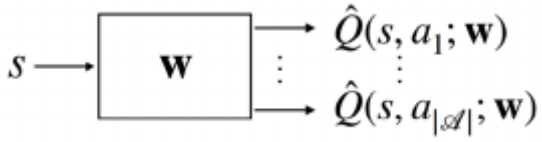
	Example	List of graph
Epoch = 1		
Epoch = 5		

We can observe that he generization ability of the epoch eqauling to 5 model is better than the other one. We can also verfy the result by comparing their corresponding generating loss.

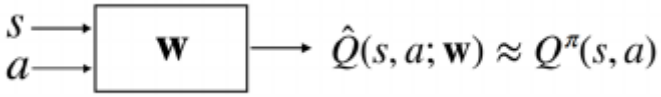
Epoch = 1	Epoch = 5
	

The samller loss is the precise answer in deep learning knowledge. We can observe that with higher epoch we will get lower loss in this pattern.

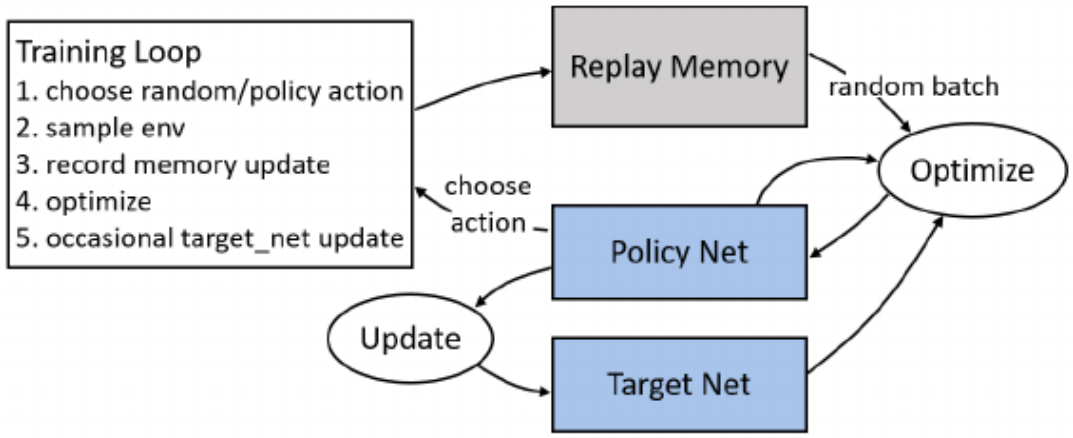
2. Deep Q-learning :
   - Network Architecture:
     - Target net



     - Policy net :

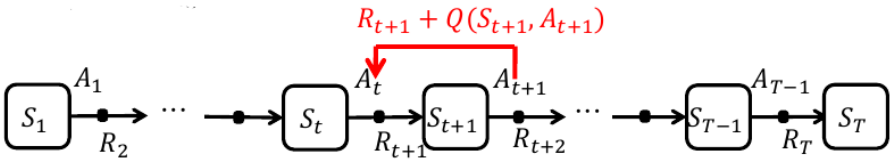


   - Training Procedure :



   - Loss function design and explaintation :

     DQN is a value based reinforcement learning. It deploy the single deep network to estimate the action value dinction of each discrete action.



     We use the following notation to represent the all network architecture.

     - Action Value :

$$Q(s_t, a_t|\theta)$$

       Action value is a core index for Reinforcement learning neural network. Based on the Action Value , we can deploy the policy by the select action form.

- Select Action :

$$arg\max_{\alpha'} Q(s_t, \alpha'|\theta)$$

Our target is to find the optimal α' that will maximize the Q value. The reward will combine with the select action value as the below form.

- Target Q :

$$Y_t^Q = r_{t+1} + \gamma \max_{\alpha'} Q(s_{t+1}, \alpha'|\theta)$$

We want to minimize the distaance between target Q and Q value. Therefore the loss can be desinged as their Euclidian distance.

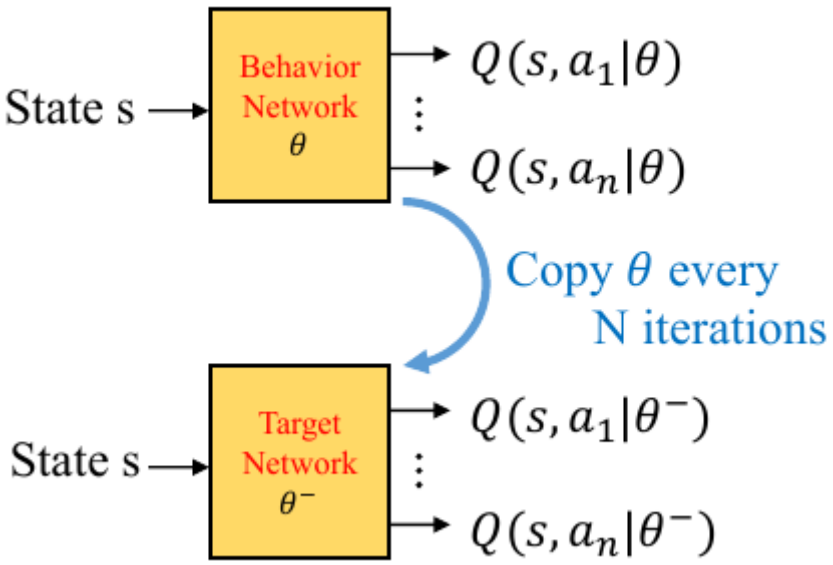
- Loss funtion :

$$L_Q(s_t, a_t|\theta) = \left(Y_t^Q - Q(s_t, a_t|\theta)\right)^2$$

Our goal is designing a Q- function that will minimize the reward r_{t+1} to make the different state be balanced. Driative of the loss function is the gradient decent direction. Therefore we can get the gradient decent form as.

- Gradient Decent :

$$\nabla_\theta L_Q(s_t, a_t|\theta) = \left(Y_t^Q - Q(s_t, a_t|\theta)\right) \nabla_\theta Q(s_t, a_t|\theta)$$



- Answer :
  - Problem 1 :
    - ✧ Updating step α :

      It is analog to the learing rate in the stochastic gradient decent.

      When it comes to training the procedure :

      $$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_t + \gamma \cdot \max_a Q(s_{t+1}, a_t) - Q(s_t, a_t))$$

      α will determine the revision velocity of the $Q(s_t, a_t)$

    - ✧ Discount factor γ :

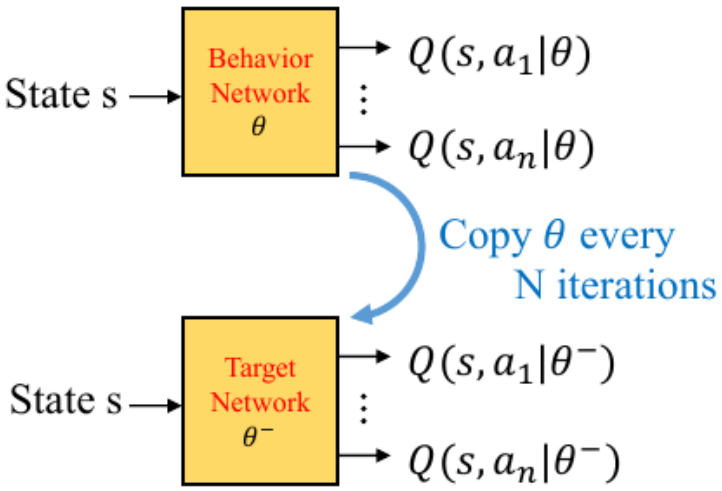
      From the previous discussion, we can get the below form

      - ➢ Target Q :

        $$Y_t^Q = r_{t+1} + \gamma \max_{\alpha'} Q(s_{t+1}, \alpha' | \theta)$$

        where target Q is our algrithm goal that will make the reward r_{t+1} is approximate as zero. γ is a weight that play a role of importance that can represent a reward function cause of reward r_{t+1} is unkown. We use this approach to get the reward function.

    - ✧ Update period τ :

      The updating iteration will make the target network revise the network parametors. We use the updating period to catch the record of the previous state for convience.

    - ✧ ϵ for ϵ greedy policy :

      To guarantee the best action for adopting a series policy. Both exploration and exploitation remain ϵ and $1 - \epsilon$ probability to select the action in a random manner.

The above configuration is showed as below :

```
def select_action(self, state):
    self.interaction_steps += 1
    self.epsilon = self.EPS_END + np.maximum( (self.EPS_START-self.EPS_END) * (1 - self.interaction_steps/self.EPS_DECAY), 0) # linear decay
    if random.random() < self.epsilon:
        #assert False, "You should check the source code for your homework!!" # random select for epsilon greedy (Dependent on the exploration probability)
        # return torch.tensor( [random.sample([0,1,2],1)], device=device, dtype=torch.long ) #torch.tensor([[random.randrange(self.action_dim)]], device=device
        choices = [0]*3+[1]*6+[2]*1
        return torch.tensor( [random.sample(choices,1)], device=device, dtype=torch.long ) #torch.tensor([[random.randrange(self.action_dim)]], device=device,
    else:
        with torch.no_grad():
            return self.policy_net(state).max(1)[1].view(1, 1)
```

- Problem 2 :

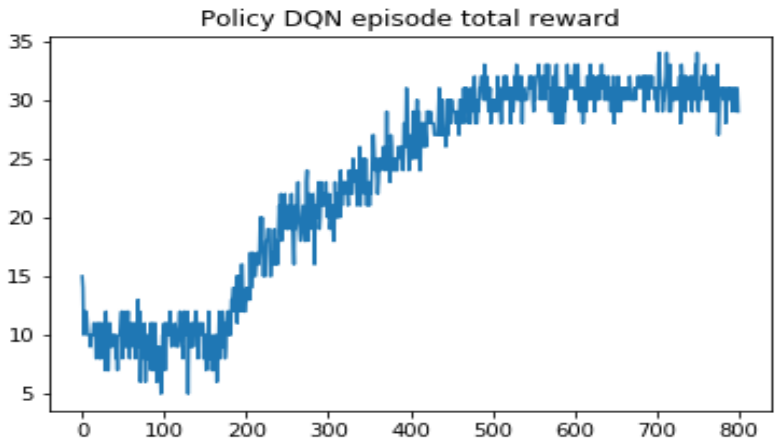
```
Episode:      1, interaction_steps:   2048, reward: 10, epsilon: 0.998157
[Info] Save model at './HW3/model' !
Evaluation: True, episode:      1, interaction_steps:   2048, evaluate reward:  0
Episode:      2, interaction_steps:   4096, reward: 10, epsilon: 0.996314
Episode:      3, interaction_steps:   6144, reward: 13, epsilon: 0.994470
Episode:      4, interaction_steps:   8192, reward:  9, epsilon: 0.992627
Episode:      5, interaction_steps:  10240, reward: 11, epsilon: 0.990784
Episode:      6, interaction_steps:  12288, reward: 11, epsilon: 0.988941
Episode:      7, interaction_steps:  14336, reward:  9, epsilon: 0.987098
Episode:      8, interaction_steps:  16384, reward: 12, epsilon: 0.985254
Episode:      9, interaction_steps:  18432, reward: 12, epsilon: 0.983411
Episode:     10, interaction_steps:  20480, reward: 12, epsilon: 0.981568
```

- Problem 3 :

My configuration to select the manner of the [NOOP(0.3), UP(0.6), DOWN(0.1)] is to use a list distributed as the frequency of the above manner. This configuration is showed as below :

```
def select_action(self, state):
    self.interaction_steps += 1
    self.epsilon = self.EPS_END + np.maximum( (self.EPS_START-self.EPS_END) * (1 - self.interaction_steps/self.EPS_DECAY), 0) #
    if random.random() < self.epsilon:
        #assert False, "You should check the source code for your homework!!" # random select for epsilon greedy (Dependent on
        # return torch.tensor( [random.sample([0,1,2],1)], device=device, dtype=torch.long ) #torch.tensor([[random.randrange(s
        choices = [0]*3+[1]*6+[2]*1
        return torch.tensor( [random.sample(choices,1)], device=device, dtype=torch.long ) #torch.tensor([[random.randrange(sel
```

We use the choice list to achieve this goal.



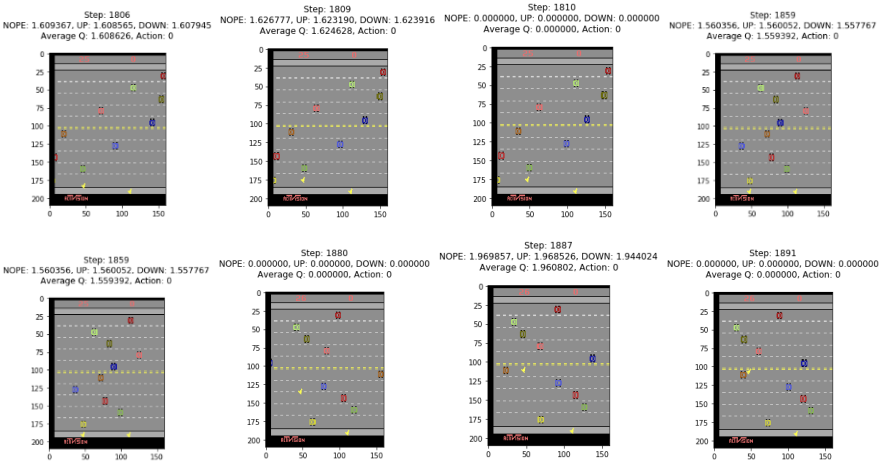
The reward function willl increase in a sigmoid shape like manner.

- Problem 4 :

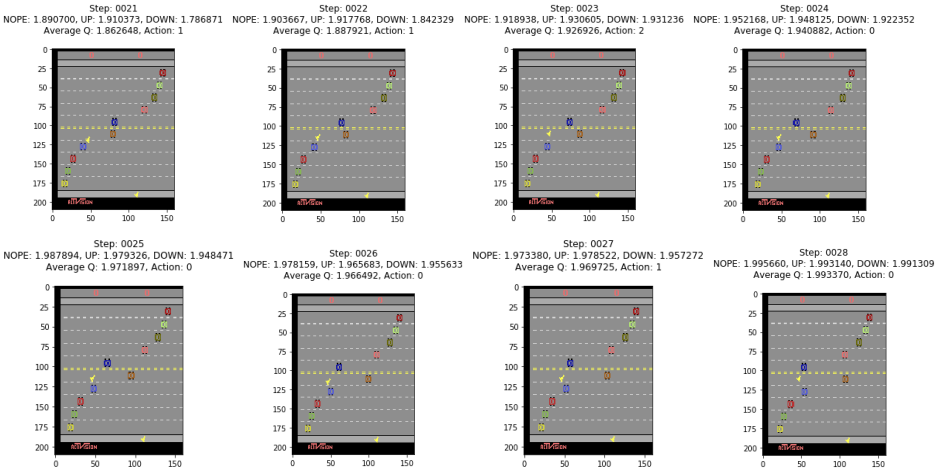
```
Episode:      0, interaction_steps:      0, reward: 29, epsilon: 0.100000
Episode:      1, interaction_steps:      0, reward: 29, epsilon: 0.100000
Episode:      2, interaction_steps:      0, reward: 29, epsilon: 0.100000
Episode:      3, interaction_steps:      0, reward: 29, epsilon: 0.100000
Episode:      4, interaction_steps:      0, reward: 31, epsilon: 0.100000
Episode:      5, interaction_steps:      0, reward: 28, epsilon: 0.100000
Episode:      6, interaction_steps:      0, reward: 30, epsilon: 0.100000
Episode:      7, interaction_steps:      0, reward: 30, epsilon: 0.100000
Episode:      8, interaction_steps:      0, reward: 28, epsilon: 0.100000
Episode:      9, interaction_steps:      0, reward: 29, epsilon: 0.100000
```

- Problem 5 :
  - Sample some states :

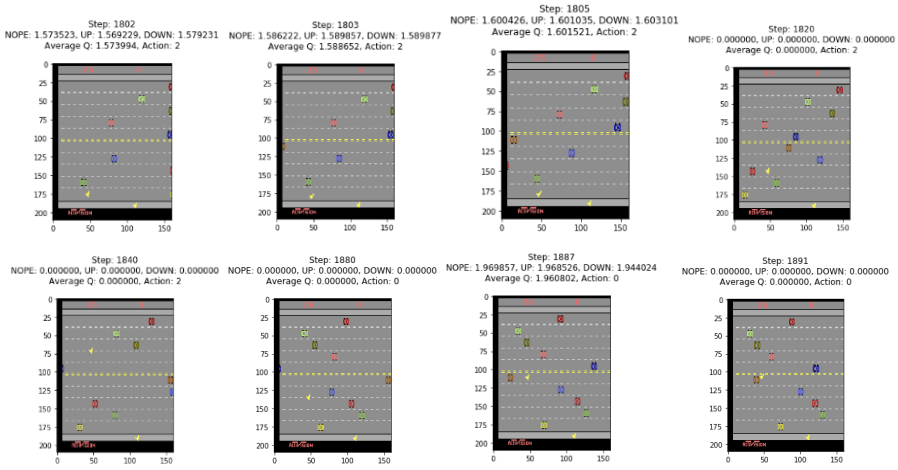
NOOP



UP



DOWN



- DQN decision meet the genral decision like a human decision. Select a proper model will follow the game rule perfectly.
- The decision manner will follow the rule we set previosly :

  [NOOP(0.3), UP(0.6), DOWN(0.1)]

We can observe this phenomenon by the distribution of the separate action by the animation.

- Q value will help the model decide the action.Larger Q value indicate that current step prfer such the action it select.Less Q value is in the opossite manner relative to the larger Q value.