Deep Reinforcement Learning

17 DQN code

Yunseong Cho

목차

CartPole - v1

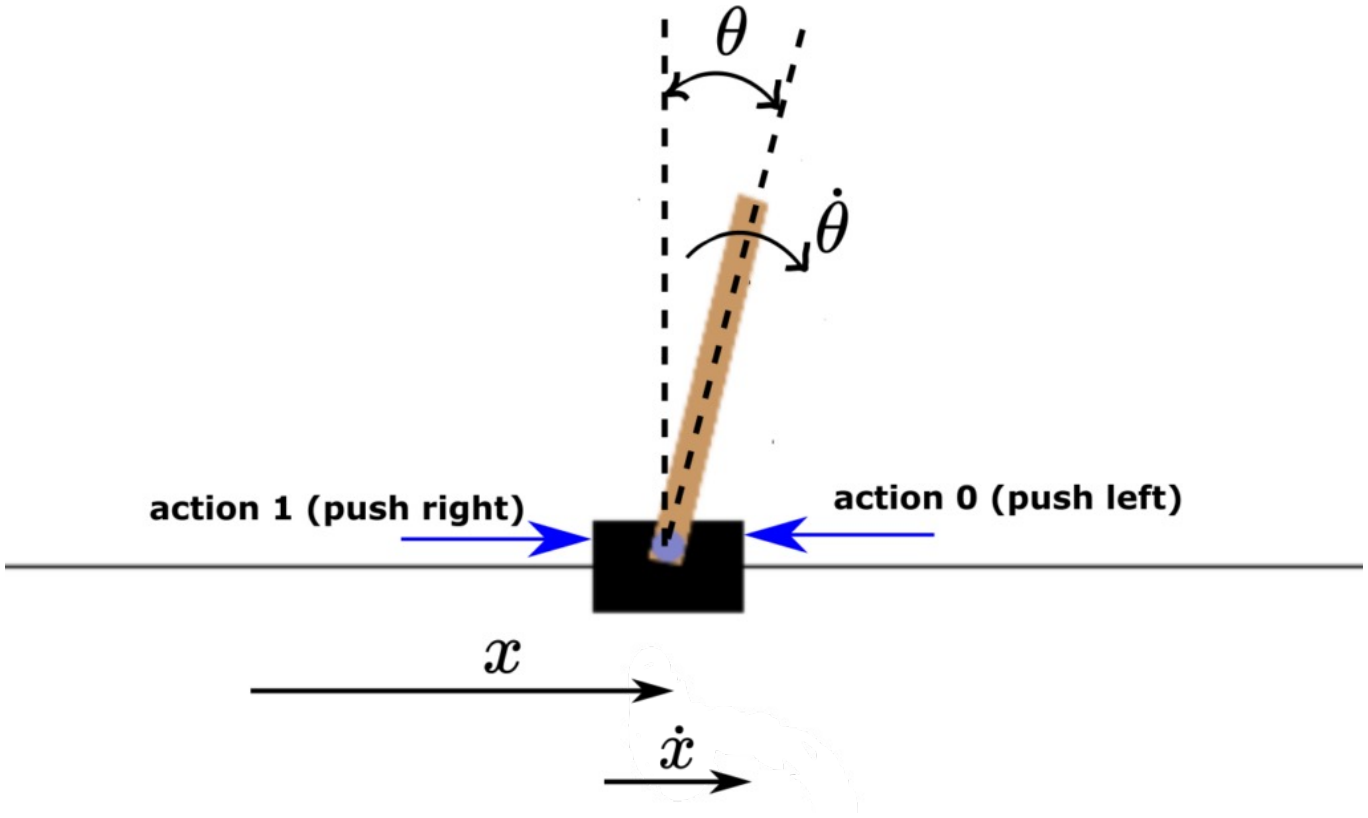
CartPole - v1

정해진 시간동안 카트에 달린 기둥이 쓰러지지 않게 균형을 잡는 게임

1

2

CartPole - v1



Time step 200을 넘거나, 기둥이 기울어진 각도가 12도가 넘거나, 카드가 중심으로부터 2.4이상 벗어나면 끝

DQN

DQN

```python
class DQN_Agent():
    def __init__(self, input_shape=(4,), num_actions=2, gamma=0.99, epsilon=1.0, epsilon_min=0.1, memory_type='PER', buffer_size=2**15, pretrained=''):

        self.num_actions = num_actions
        self.agent = self.nn_model(input_size=input_shape, action_dim=num_actions)
        self.target_agent = self.nn_model(input_size=input_shape, action_dim=num_actions)
        self.update_target()
        self.optimizer = tf.keras.optimizers.Adam()

        self.memory_type = memory_type
        if memory_type == 'PER':
            self.buffer = Prioritized_Experience_ReplayBuffer(capacity=buffer_size)
        else:
            self.buffer = ReplayBuffer(capacity=buffer_size)

        self.gamma = gamma
        self.epsilon_start = epsilon
        self.epsilon = epsilon
        self.epsilon_min = epsilon_min

        if pretrained:
            self.continue_training(pretrained)

    def nn_model(self, input_size, action_dim):
        input_layer = Input(shape=input_size)
        x = Dense(128, activation='tanh', kernel_initializer='glorot_uniform')(input_layer)
        x = Dense(64, activation='tanh', kernel_initializer='glorot_uniform')(x)
        output_layer = Dense(action_dim, activation='linear')(x)

        model = Model(input_layer, outputs = output_layer)
        return model
```

객체 생성

사전 학습

네트워크를 생성하는 메소드

Xavier Initialization

- Xavier Normal Initialization

$$W \sim N(0, Var(W))$$

$$Var(W) = \sqrt{\frac{2}{n_{in} + n_{out}}}$$

(n_{in} : 이전 layer(input)의 노드 수, n_{out} : 다음 layer의 노드 수)

- Xavier Uniform Initialization

$$W \sim U(-\sqrt{\frac{6}{n_{in} + n_{out}}}, \ +\sqrt{\frac{6}{n_{in} + n_{out}}})$$

(n_{in} : 이전 layer(input)의 노드 수, n_{out} : 다음 layer의 노드 수)

- Xaiver함수는 비선형함수(ex. sigmoid, tanh)에서 효과적인 결과를 보여준다. 하지만 ReLU 함수에서 사용 시 출력 값이 0으로 수렴하게 되는 현상을 확인 할 수 있다. 따라서 ReLU함수에는 또 다른 초기화 방법을 사용해야 한다.

DQN - Buffer

```python
class ReplayBuffer():
    def __init__(self, capacity=10000):
        self.capacity = capacity
        self.buffer = deque(maxlen=capacity)

    def store(self, state, action, reward, next_state, done):
        self.buffer.append([state, action, reward, next_state, done])

    def replay_buffer_sampling(self, batch_size):
        experience_samples = random.sample(self.buffer, batch_size)
        state_arr, action_arr, reward_arr, next_state_arr, done_arr = map(np.asarray, zip(*experience_samples))
        return state_arr, action_arr, reward_arr, next_state_arr, done_arr

    def size(self):
        return len(self.buffer)
```

변수 정의

버퍼에 저장

하나의 샘플

버퍼 사이즈 만큼 샘플링

DQN

target network를 업데이트

'Tau' 파라미터를 통해서 target network를 업데이트

배치 샘플을 가져와서 loss를 계산한다.

```python
def update_target(self):
    self.target_agent.set_weights(self.agent.get_weights())

def soft_update_target(self, TAU):
    for t, e in zip(self.target_agent.trainable_variables, self.agent.trainable_variables):
        t.assign(t * (1 - TAU) + e * TAU)

def replay_experience(self, batch_size):
    if self.memory_type == 'PER':
        with tf.GradientTape() as tape:
            state_arr, action_arr, reward_arr, next_state_arr, done_arr, sampled_idxs, is_weights = self.buffer.replay_buffer_sampling(batch_size)
            predicts = tf.reduce_sum(self.agent(state_arr, training=True)*action_arr, axis=1)
            next_q_values = np.max(self.target_agent(next_state_arr, training=False), axis=1)
            targets = reward_arr + self.gamma*next_q_values*(1-done_arr)
            td = targets - predicts
            loss = tf.reduce_mean(is_weights * td**2)
        grads = tape.gradient(loss, self.agent.trainable_variables)
        self.optimizer.apply_gradients(zip(grads, self.agent.trainable_variables))

        return td, sampled_idxs

    else:
        with tf.GradientTape() as tape:
            state_arr, action_arr, reward_arr, next_state_arr, done_arr = self.buffer.replay_buffer_sampling(batch_size)
            predicts = tf.reduce_sum(self.agent(state_arr, training=True)*action_arr, axis=1)
            next_q_values = np.max(self.target_agent(next_state_arr, training=False), axis=1)
            targets = reward_arr + self.gamma*next_q_values*(1-done_arr)
            td = targets - predicts
            loss = tf.reduce_mean(td**2)

        grads = tape.gradient(loss, self.agent.trainable_variables)
        self.optimizer.apply_gradients(zip(grads, self.agent.trainable_variables))

        return
```

$$L(\theta) = \underbrace{[r_{t+1} + \gamma \max_a Q(s_{t+1}, a; \theta)}_{\text{target}} - \underbrace{Q(s_t, a_t; \theta)}_{\text{predict}}]^2$$

DQN

State을 보고
ε-greedy에 따라
액션을 선택.

```python
def get_action(self, observation):
    action_logits = self.agent.predict_on_batch(observation.reshape(1,-1))

    should_explore = np.random.rand()
    if should_explore < self.epsilon:
        action = np.random.choice(self.num_actions)
    else:
        action = np.argmax(action_logits, axis=1)[0]
    return action
```

state

에피소드가 진행됨에 따라서
ε을 0에 가깝게 설정

```python
def linear_schedule_epsilon(self, episode:int, max_episode:int):
    start_episode = 0
    start, end = self.epsilon_start, self.epsilon_min
    if episode < max_episode:
        return (start*(max_episode-episode) + end*(episode-start_episode)) / (max_episode - start_episode)
    else:
        return end

def exp_schedule_epsilon(self, decay):
    return self.epsilon * decay

def save_model(self, mdir):
    self.agent.save_weights(mdir)

def continue_training(self, mdir):
    self.agent.load_weights(mdir)
```

Double DQN

Double DQN

```python
class DoubleDQN_Agent(DQN_Agent):
    def replay_experience(self, batch_size):
        if self.memory_type == 'PER':
            with tf.GradientTape() as tape:
                state_arr, action_arr, reward_arr, next_state_arr, done_arr, sampled_idxs, is_weights = self.buffer.replay_buffer_sampling(batch_size)
                predicts = tf.reduce_sum(self.agent(state_arr, training=True)*action_arr, axis=1)

                next_q_targets = self.target_agent(next_state_arr, training=False)
                next_q_values  = next_q_targets.numpy()[range(batch_size), np.argmax(self.agent(next_state_arr), axis=1)]

                targets = reward_arr + self.gamma*next_q_values*(1-done_arr)
                td = targets - predicts
                loss = tf.reduce_mean(is_weights * td**2)
            grads = tape.gradient(loss, self.agent.trainable_variables)
            self.optimizer.apply_gradients(zip(grads, self.agent.trainable_variables))

            return td, sampled_idxs

        else:
            with tf.GradientTape() as tape:
                state_arr, action_arr, reward_arr, next_state_arr, done_arr = self.buffer.replay_buffer_sampling(batch_size)
                predicts = tf.reduce_sum(self.agent(state_arr, training=True)*action_arr, axis=1)

                next_q_targets = self.target_agent(next_state_arr, training=False)
                next_q_values = next_q_targets.numpy()[range(batch_size), np.argmax(self.agent(next_state_arr), axis=1)]

                targets = reward_arr + self.gamma*next_q_values*(1-done_arr)
                td = targets - predicts
                loss = tf.reduce_mean(td**2)
            grads = tape.gradient(loss, self.agent.trainable_variables)
            self.optimizer.apply_gradients(zip(grads, self.agent.trainable_variables))

            return
```

Double DQN 아닐때 loss 계산

$$L(\theta) = [r_{t+1} + \gamma \hat{Q}(s_{t+1}, \underset{a}{\arg\max}\, Q(s_{t+1}, a; \theta); \hat{\theta}) - Q(s_t, a_t; \theta)]^2$$

target　　predict

Double DQN

```
            self.optimizer.apply_gradients(zip(grads, self.agent.trainable_variables))

            return td, sampled_idxs

        else:
            with tf.GradientTape() as tape:
                state_arr, action_arr, reward_arr, next_state_arr, done_arr = self.buffer.replay_buffer_sampling(batch_size)
                predicts = tf.reduce_sum(self.agent(state_arr, training=True)*action_arr, axis=1)

                next_q_targets = self.target_agent(next_state_arr, training=False)
                next_q_values = next_q_targets.numpy()[range(batch_size), np.argmax(self.agent(next_state_arr), axis=1)]

                targets = reward_arr + self.gamma*next_q_values*(1-done_arr)
                td = targets - predicts
                loss = tf.reduce_mean(td**2)
            grads = tape.gradient(loss, self.agent.trainable_variables)
            self.optimizer.apply_gradients(zip(grads, self.agent.trainable_variables))

            return
```

```
def get_action(self, observation):
    action_logits1 = self.agent.predict_on_batch(observation.reshape(1,-1))
    action_logits2 = self.target_agent.predict_on_batch(observation.reshape(1,-1))

    action_logits = action_logits1 + action_logits2

    should_explore = np.random.rand()
    if should_explore < self.epsilon:
        action = np.random.choice(self.num_actions)
    else:
        action = np.argmax(action_logits, axis=1)[0]

    return action
```

state

behavior network와 target network를 통해서 나온 최종적인 Q-value를 바탕으로 ε-greedy를 통해서 action을 선택

Dueling DQN

Dueling DQN

value function, advantage function 과 advantage function의 mean을 구함

```
class DuelingDQN_Agent(DQN_Agent):
    def nn_model(self, input_size, action_dim):

        input_layer = Input(shape=input_size)
        x = Dense(128, activation='tanh', kernel_initializer='glorot_uniform')(input_layer)
        x = Dense(64, activation='tanh', kernel_initializer='glorot_uniform')(x)

        v_out = Dense(1, activation='linear')(x)
        adv_out = Dense(action_dim, activation='linear')(x)
        adv_mean = -tf.reduce_mean(adv_out, axis=1)

        output_layer = Add()([v_out, adv_out, adv_mean])
        model = Model(input_layer, outputs = output_layer)

        return model
```

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left[A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\alpha)\right]$$

Double Dueling DQN

Double Dueling DQN

value function, advantage function 과 advantage function의 mean을 구한다

```python
class DoubleDuelingDQN_Agent(DoubleDQN_Agent):
    def nn_model(self, input_size, action_dim):

        input_layer = Input(shape=input_size)
        x = Dense(128, activation='tanh', kernel_initializer='glorot_uniform')(input_layer)
        x = Dense(64, activation='tanh', kernel_initializer='glorot_uniform')(x)

        v_out = Dense(1, activation='linear')(x)
        adv_out = Dense(action_dim, activation='linear')(x)
        adv_mean = -tf.reduce_mean(adv_out, axis=1)

        output_layer = Add()([v_out, adv_out, adv_mean])
        model = Model(input_layer, outputs = output_layer)

        return model
```

Training

Training

```python
def train(env, agent, num_episodes, beta_anneal_episodes, replay_period, batch_size, index:int=0):

    agent_name = agent.__class__.__name__.split('_')[0]
    if not os.path.exists(agent_name):
        os.makedirs(agent_name)

    memory_type = agent.memory_type
    if memory_type:
        fig_path = agent_name+'/figs_PER/'
    else:
        fig_path = agent_name+'/figs/'

    if not os.path.exists(fig_path):
        os.makedirs(fig_path)

    reward_list = []
    action_num = env.action_space.n
    one_hot_action = np.eye(action_num)

    for episode in range(num_episodes):

        counter = 0
        done, total_reward = False, 0

        state, _ = env.reset()

        while not done:
            action = agent.get_action(state)
            next_state, reward, terminated, truncated, _ = env.step(action)
            done = terminated or truncated
            total_reward += reward

            agent.buffer.store(state, one_hot_action[action], reward, next_state, done) # self.buffer.append([state, action, reward, next_state, done])
```

총 에피소드 개수만큼 루프

초기상태

막대가 넘어지거나
카트가 범위를 벗어나거나
시간이 지날경우 에피소드를 끝.

Training

총 에피소드 개수만큼 루프

초기상태

막대가 넘어지거나
카트가 범위를 벗어나거나
시간이 지날경우 에피소드를 끝.

버퍼에 배치 사이즈 이상으로 샘플이 쌓이면
일정 주기로 랜덤 샘플링을 해서
behavior network를 업데이트

더 큰 주기로 target network를 업데이트

```python
for episode in range(num_episodes):

    counter = 0
    done, total_reward = False, 0

    state, _ = env.reset()

    while not done:
        action = agent.get_action(state)
        next_state, reward, terminated, truncated, _ = env.step(action)
        done = terminated or truncated
        total_reward += reward

        agent.buffer.store(state, one_hot_action[action], reward, next_state, done) # 버퍼에 에피소드 저장

        if agent.buffer.size() >= batch_size and counter%replay_period[0] == 0:
            if agent.memory_type == 'PER':
                td, idxs = agent.replay_experience(batch_size) # 버퍼에서 샘플링하고 그것을 업데이트에 사용
                for i in range(batch_size):
                    agent.buffer.update(idxs[i], abs(td[i])) # 업데이트 후에 변경된 td값을 샘플들에게 다시 할당
            else:
                agent.replay_experience(batch_size) # 버퍼에서 샘플링하고 그것을 업데이트에 사용

        if agent.buffer.size() >= batch_size and counter%replay_period[1] == 0:
            agent.update_target()
        # agent.soft_update_target(TAU=0.005)

        state = next_state

    reward_list.append(total_reward)
```

Training

버퍼에 배치 사이즈 이상으로 샘플이 쌓이면
일정 주기로 랜덤 샘플링을 해서
behavior network를 업데이트

더 큰 주기로 target network를 업데이트

```
    if agent.buffer.size() >= batch_size and counter%replay_period[0] == 0:
        if agent.memory_type == 'PER':
            td, idxs = agent.replay_experience(batch_size) # 버퍼에서 샘플링하고 그것을 업데이트에 사용
            for i in range(batch_size):
                agent.buffer.update(idxs[i], abs(td[i])) # 업데이트 후에 변경된 td값을 샘플들에게 다시 할당
        else:
            agent.replay_experience(batch_size) # 버퍼에서 샘플링하고 그것을 업데이트에 사용

    if agent.buffer.size() >= batch_size and counter%replay_period[1] == 0:
        agent.update_target()
    # agent.soft_update_target(TAU=0.005)

    state = next_state

reward_list.append(total_reward)

if agent.memory_type == 'PER':
    if episode >= beta_anneal_episodes[0]:
        agent.buffer.beta = agent.buffer.anneal_beta(episode, beta_anneal_episodes[0], beta_anneal_episodes[1], 0.4, 1)
    if episode > beta_anneal_episodes[1]:
        agent.buffer.beta = 1

agent.epsilon = agent.linear_schedule_epsilon(episode=episode, max_episode = 300) # epsilon을 에피소드가 진행될수록 min_epsilon에 linear하게 가까워지도록 설

print(f'Episode: {episode+1}, total reward: {total_reward}, buffer size: {agent.buffer.size()}, epsilon: {agent.epsilon}')
```

감사합니다