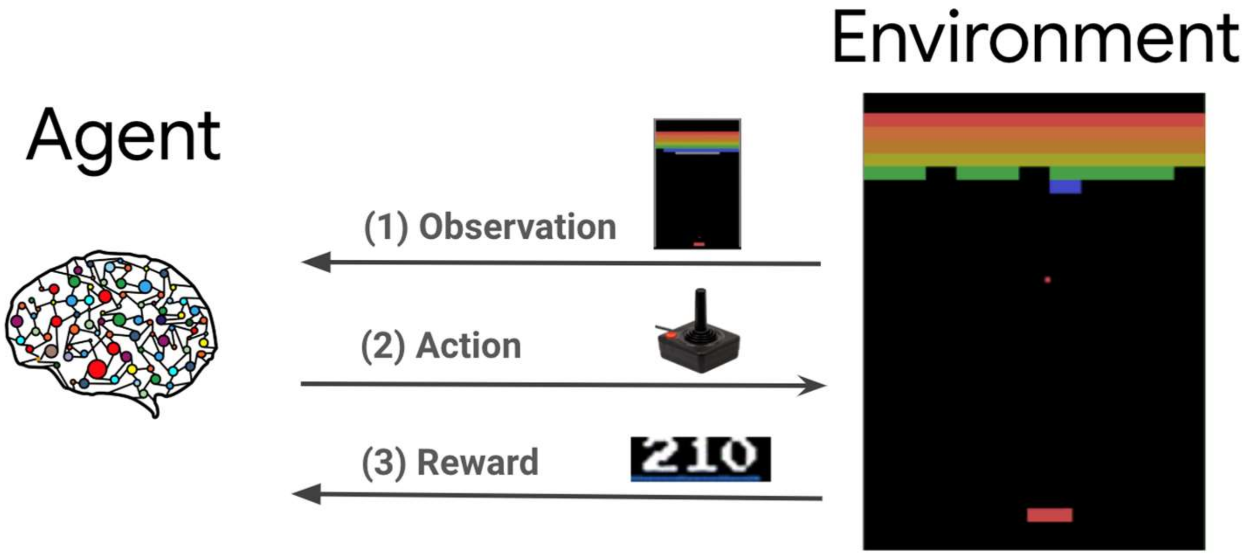
Deep Q Network

Deep Q Network (DQN)

- A representative Deep Reinforcement Learning (DRL) algorithm
- Proposed in 2013 paper "Playing Atari with Deep Reinforcement Learning".
- The paper showed that a DQN can be trained to play Atari games with human-level accuracy.



What is DQN?

- The objective of reinforcement learning is to find the optimal policy.
- Optimal policy: policy that gives the maximum return.
- To get the policy, we compute the Q function.
- Once we have a Q function, we can extract a policy by choosing actions with maximum Q values.

- For example, if we have a Q table like this:

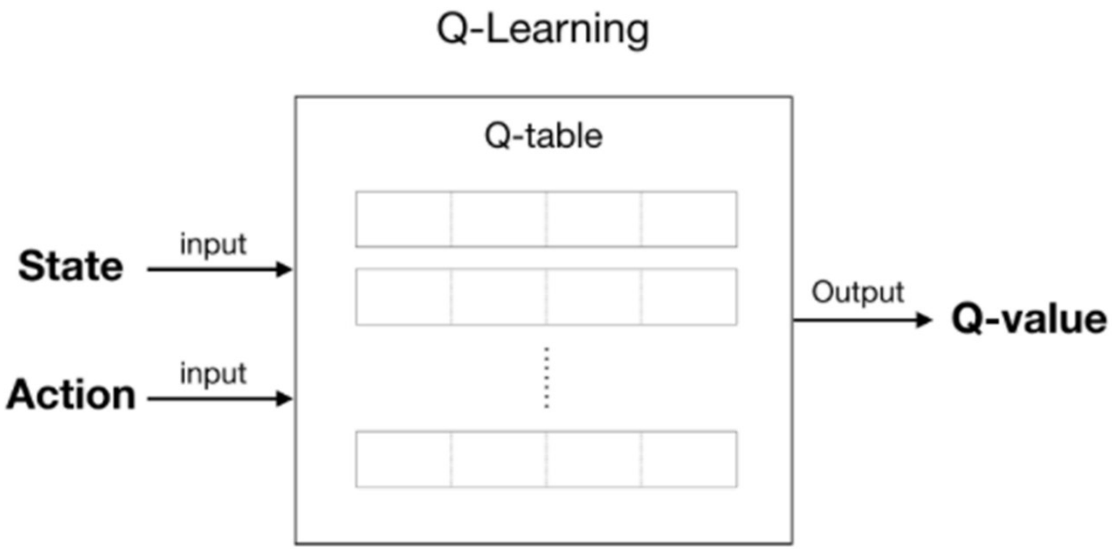
State	Action	Value
A	up	17
A	down	10
B	up	11
B	down	20

- Our policy is **{A: up, B: down}**.

What is DQN?

- In previous chapters, we used a Q table.
- "Learning" was done by updating the Q table iteratively.
- In a Q table, an entry consists of a (state, action) pair and its Q value.

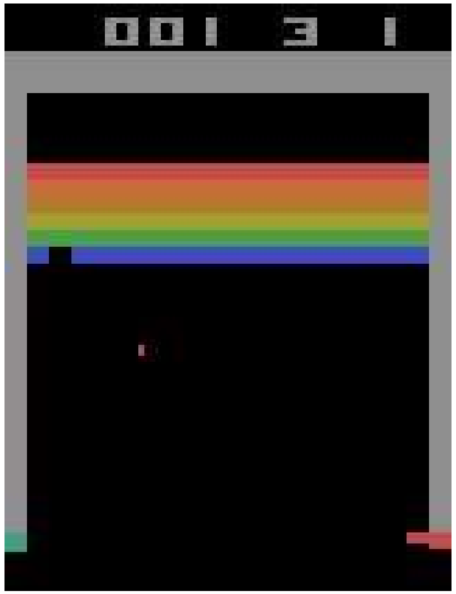
- Basically, a Q table is a function.
- The input to the table is a (station, action).
- The output from the table is a Q value.



What is DQN?

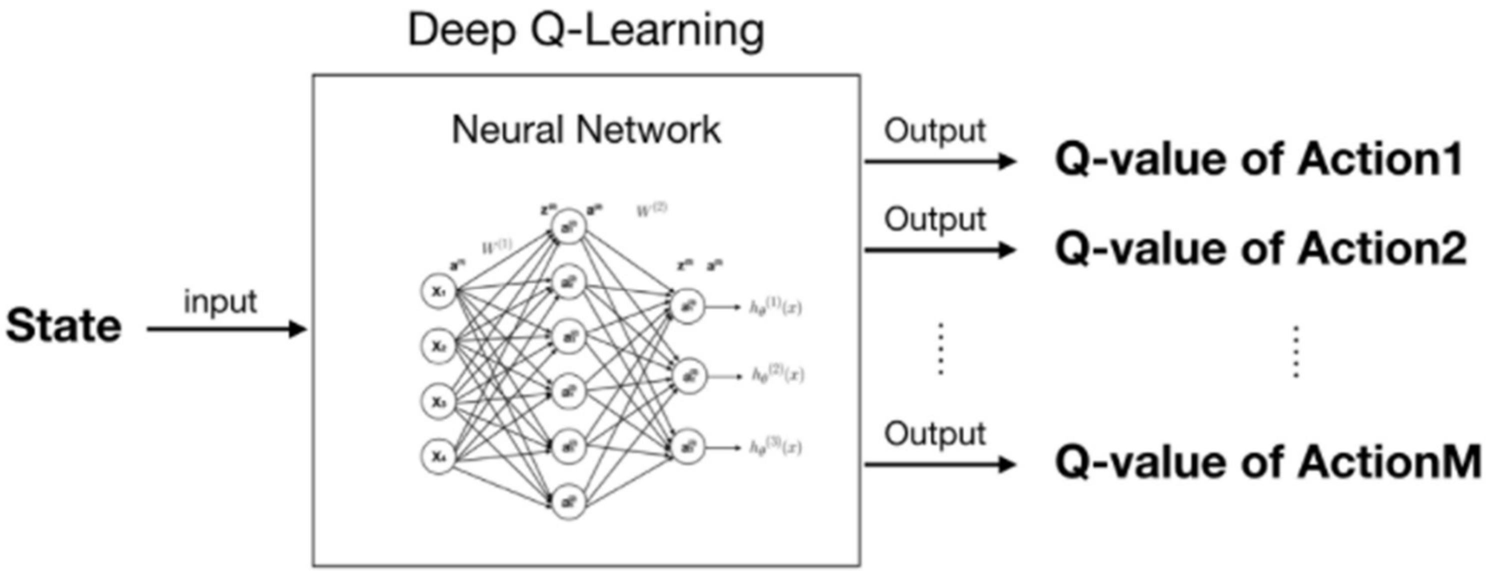
- The problem with using a Q table is when the number of states is too large.
- Say we have an environment with 100,000 states and 10 possible actions. Then we have 1,000,000 entries in our Q table.
- It will be very expensive to compute the Q values of all state-actions pairs.

- e.g.) The 'Breakout-v0' environment has 210x160x3 different states.



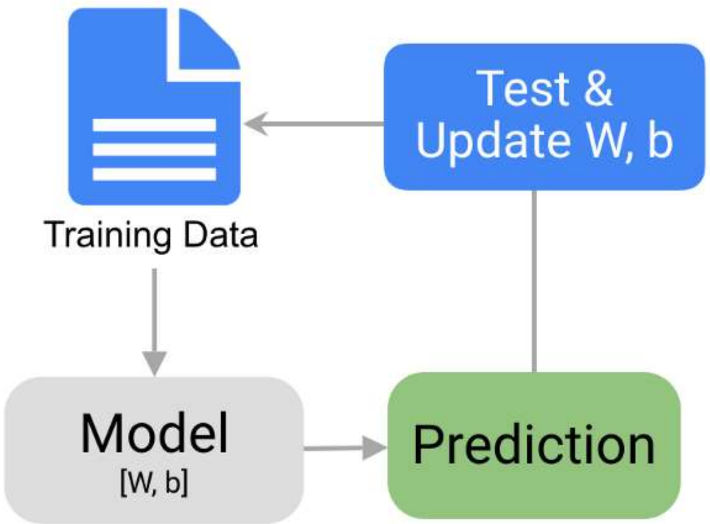
What is DQN?

- Instead of using a Q-table, we use a neural network.
- The neural network is basically a function approximator for the Q function.
- The input to neural network is a vector representing the state.
- The output of the network is the Q-values for each action in the action space.
- The neural network can estimate Q values for unvisited states, based on the experience from similar states.



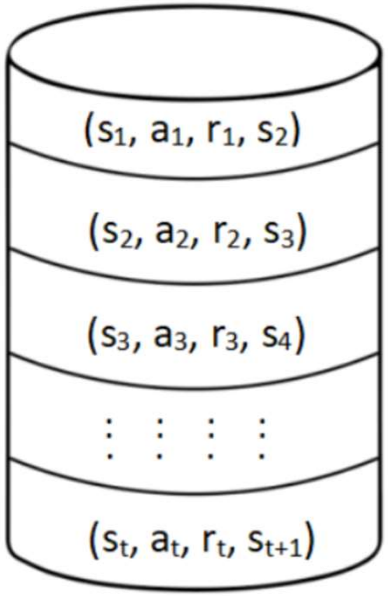
Understanding DQN

- As Q tables need updates, a DQN must be trained.
- The DQN is trained using supervised learning, where we provide training samples and their labels (target value).
- In supervised learning, stochastic gradient descent is typically used, where we provide a batch of samples to the network.



Training Samples for DQN

- What are training samples we use to train a DQN?
- In Q-learning, we learn experience by going through episodes.
- In a state, the agent performs an action according to the policy (such as an epsilon-greedy policy), and moves to the next state.
- From this we get a transition, which is (s, a, r, s'). This transition becomes a training sample.
- We save the transitions in a buffer called replay buffer.



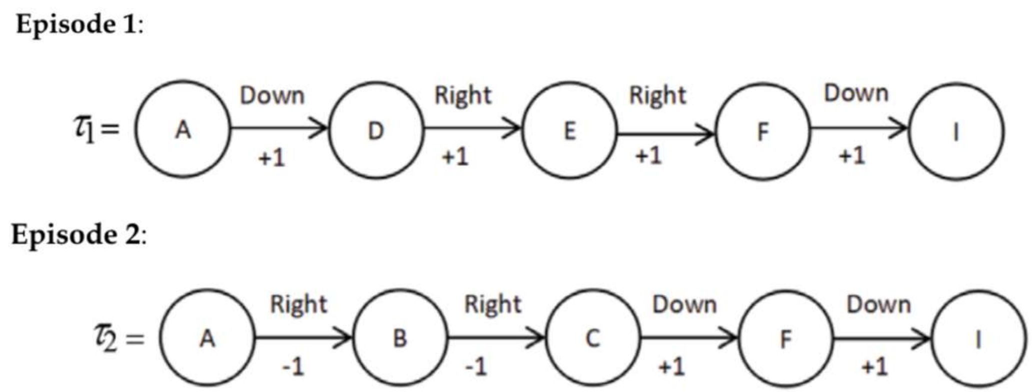
Training Samples for DQN

- Process for constructing a replay buffer

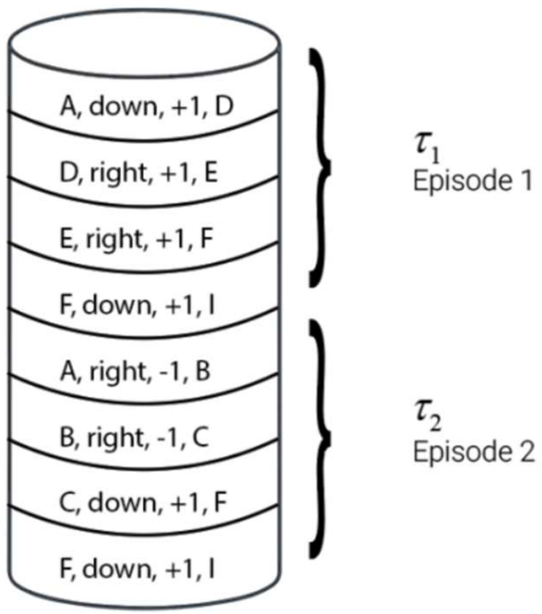
1. Initialize the replay buffer $\mathcal{D}$.
2. For each episode perform *step 3*.
3. For each step in the episode:
    1. Make a transition, that is, perform an action a in the state s, move to the next state s', and receive the reward r.
    2. Store the transition information (s, a, r, s') in the replay buffer $\mathcal{D}$.

Training Samples for DQN

- Example: running episodes in the Grid World



replay buffer



Training Samples for DQN

- We collect transitions from many episodes and store them in the reply buffer.
- When training a model, we select minibatches from the replay buffer.
- In order to avoid correlation, we select random samples to create a minibatch.
- This process is called Experience Replay.

- Since we have a limited size for the replay buffer, old transition samples are replaced with new transition samples.

Loss Function

- Our objective of using a neural network is to estimate Q values of state-action pairs. This is a regression task.
- For regression task, we generally use the mean squared error (MSE) as the loss function.

$$\text{MSE} = \frac{1}{K}\sum_{i=1}^{K}(y_i - \hat{y}_i)^2$$

- In the equation, y is the target value, $\hat{y}$ is the predicted value, and K is the number of training samples in the minibatch.

Loss Function

- The predicted value $\hat{y}$ is the outcome of the model.
- What is the target value y?

- In the Bellman optimality equation, the optimal Q value can be obtained using the following equation.

$$Q^*(s,a) = \mathbb{E}_{s' \sim P}[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')]$$

- We remove the expectation from the equation. We will approximate the expectation by sampling K number of transitions from the replay buffer and taking the average value.

$$Q^*(s,a) = r + \gamma \max_{a'} Q^*(s',a')$$

  – We denote $R(s,a,s')$ as r.

Loss Function

- Since we want our Q values to become optimal, we set the target Q value as:
  - $Q^*(s, a)$

- Also, we denote the predicted value as:
  - $Q_\theta(s, a)$

- Thus, the difference between target value and predicted value is:
  - $Q^*(s, a) - Q_\theta(s, a) = r + \gamma \max_{a'} Q(s', a') - Q_\theta(s, a)$

- This is the temporal difference error used in the update rule of Q-learning.

Loss Function

- We use the MSE loss. Thus, our loss function can be expressed as:

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(r_i + \gamma \max_{a'} Q_\theta(s_i', a') - Q_\theta(s_i, a_i))^2$$

  - $y_i = r_i + \gamma \max_{a'} Q_\theta(s_i', a')$
    - Since we only have a model θ we extract the maximum Q value from the model.
  - $\hat{y}_i = Q_\theta(s_i, a_i)$

- If the next state s' is a terminal state, we cannot compute the Q value because we do not take any action in the terminal state.
- Thus, if s' is terminal, we define $y_i = r_i$.
- In summary, our loss function will be:

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(y_i - Q_\theta(s_i, a_i))^2 \qquad y_i = \begin{cases} r_i & \text{if } s' \text{ is terminal} \\ r_i + \gamma \max_{a'} Q_\theta(s_i', a') & \text{if } s' \text{ is not terminal} \end{cases}$$

The Target Network

- In our loss function, both the target value and the predicted value come from the same model.

$$L(\theta)=\frac{1}{K}\sum_{i=1}^{K}(r_i+\gamma\max_{a'}\underbrace{Q_\theta(s'_i,a')}_{\text{Compute using }\theta}-\underbrace{Q_\theta(s_i,a_i)}_{\text{Compute using }\theta})^2$$

- This causes instability in the MSE and the network learns poorly. It also causes a lot of divergence during training.
  - When we update the network parameters θ, both the target and the predicted value changes.
  - The predicted value keeps on trying to be the same as the target value, but the target value keeps on changing due to the update on the network parameter θ.

The Target Network

- It helps if we freeze the target value for a while and compute only the predicted value so that our predicted value matches the target value.

- In order to apply this idea, we prepare two models represented by parameters θ and θ'. They have exactly the same architecture.

- We freeze the target Q-network θ' for a while, and update the main Q-network θ.

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(r_i + \gamma \max_{a'} \underbrace{Q_{\theta'}(s_i', a')}_{\text{Compute using } \theta'} - \underbrace{Q_{\theta}(s_i, a_i)}_{\text{Compute using } \theta})^2$$

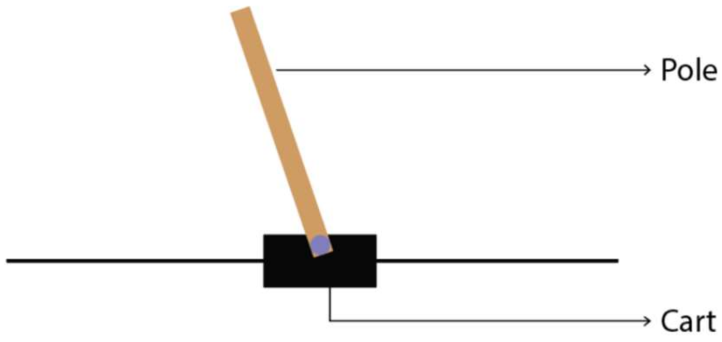
- Every once in a while, the parameters θ is copied to θ'.

The DQN algorithm

1. Initialize the main network parameter θ with random values
2. Initialize the target network parameter θ' by copying the main network parameter θ
3. Initialize the replay buffer $\mathcal{D}$
4. For N number of episodes, perform *step 5*
5. For each step in the episode, that is, for $t = 0, \ldots, T$-1:
    1. Observe the state s and select an action using the epsilon-greedy policy, that is, with probability epsilon, select random action a and with probability 1-epsilon, select the action $a = \arg\max_a Q_\theta(s, a)$
    2. Perform the selected action and move to the next state s' and obtain the reward r
    3. Store the transition information in the replay buffer $\mathcal{D}$
    4. Randomly sample a minibatch of K transitions from the replay buffer $\mathcal{D}$
    5. Compute the target value, that is, $y_i = r_i + \gamma \max_{a'} Q_{\theta'}(s'_i, a')$
    6. Compute the loss, $L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(y_i - Q_\theta(s_i, a_i))^2$
    7. Compute the gradients of the loss and update the main network parameter θ using gradient descent: $\theta = \theta - \alpha\nabla_\theta L(\theta)$
    8. Freeze the target network parameter θ' for several time steps and then update it by just copying the main network parameter θ

Cart-Pole Balancing using DQN

- State space: 4 continuous values
  - cart position
  - cart velocity
  - pole angle
  - pole velocity at the tip
- Action space: 2 discrete actions
  - push cart to the right
  - push cart to the left
- Reward
  - The agent acquires +1 reward for every timestep until the termination
- Terminating condition
  - The pole is more than 15 degress from vertical
  - The cart moves more than 2.4 units from the center



- Since we have a continuous state space, it is difficult to use a Q-table for learning.

Cart-Pole Balancing using DQN [ex016]

- libraries
  - we use the 'collections' libary to manage the replay buffer.
    - we are going to use a double-ended queue (deque) for the buffer.
  - we use the 'random' libary to sample a random subset from a list

```
# libraries
import gym
import collections
import random
```

  - we use the Pytorch library to train a neural network.
    - other possibility is to use Keras/Tensorflow.

```
# pytorch library is used for deep learning
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
```

Cart-Pole Balancing using DQN

- hyperparameters
  - learning rate: 0.0005
  - discount factor (γ): 0.98
  - batch size: 32
  - size of the replay buffer: 50000

```
# hyperparameters
learning_rate = 0.0005
gamma = 0.98
buffer_limit = 50000        # size of replay buffer
batch_size = 32
```

Cart-Pole Balancing using DQN

- class ReplayBuffer
  - '__init__' initializes the replay buffer
  - 'put' adds a new transition to the buffer
  - 'sample' creates a batch by randomly selecting transitions from the buffer
    - It makes lists of s, a, r, s', and done.
    - It also converts the list into tensors.
  - 'size' returns the number of transitions stored in the buffer

Cart-Pole Balancing using DQN

```python
class ReplayBuffer():
    def __init__(self):
        self.buffer = collections.deque(maxlen=buffer_limit)    # double-ended queue

    def put(self, transition):
        self.buffer.append(transition)

    def sample(self, n):
        mini_batch = random.sample(self.buffer, n)
        s_lst, a_lst, r_lst, s_prime_lst, done_mask_lst = [], [], [], [], []

        for transition in mini_batch:
            s, a, r, s_prime, done_mask = transition
            s_lst.append(s)
            a_lst.append([a])
            r_lst.append([r])
            s_prime_lst.append(s_prime)
            done_mask_lst.append([done_mask])

        return torch.tensor(s_lst, dtype=torch.float), torch.tensor(a_lst), \
               torch.tensor(r_lst), torch.tensor(s_prime_lst, dtype=torch.float), \
               torch.tensor(done_mask_lst)

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

Cart-Pole Balancing using DQN

- class Qnet
  - '__init__' defines layers of the model
    - nn.Linear(4, 128) is a fully connected layer with 4 inputs and 128 outputs
    - The model used here has one hidden layer with 128 neurons
  - 'forward' is called when an input is passed to the Qnet object
    - it takes a single argument, x, which is the input vectors
    - F.relu is applies ReLU to the input
  - 'sample_action' selects an action according to the epsilon-greedy policy.
    - If 'coin' is smaller than epsilon, either 0 or 1 is chosen randomly.
    - If 'coin' is larger than epsilon, the action with the maximum Q value is chosen.

Cart-Pole Balancing using DQN

```python
class Qnet(nn.Module):
    def __init__(self):
        super(Qnet, self).__init__()
        self.fc1 = nn.Linear(4, 128)
        self.fc2 = nn.Linear(128, 128)
        self.fc3 = nn.Linear(128, 2)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def sample_action(self, obs, epsilon):
        out = self.forward(obs)
        coin = random.random()
        if coin < epsilon:
            return random.randint(0,1)
        else :
            return out.argmax().item()
```

Cart-Pole Balancing using DQN

- def train: trains the Q network using minibatches in the replay buffer
  - First, sample a batch from the replay buffer.
  - Pass the states s as the input to the model q and get the output q_out.
  - From q_out, select values for actions taken in each sample and assign to q_a.
  - Calculate $\mathrm{max\_q\_prime}$ which is $\max_{a'} Q_\theta(s_i', a')$ for all samples in the batch.
  - Calculate the target value $target$.
    - If done_mask is 1, the target value is equal to r.
    - If done_mask is 0, the target value is equal to $r + \gamma \max_{a'} Q_\theta(s_i', a')$.
  - Calculate the MSE loss.

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(r_i + \gamma \max_{a'} Q_{\theta'}(s_i', a') - Q_\theta(s_i, a_i))^2$$

  - Calculate the gradient.

$$\theta = \theta - \alpha \nabla_\theta L(\theta)$$

  - Update the parameters.

Cart-Pole Balancing using DQN

```python
def train(q, q_target, memory, optimizer):
    for i in range(10):
        s,a,r,s_prime,done_mask = memory.sample(batch_size)

        q_out = q(s)
        q_a = q_out.gather(1,a)
        max_q_prime = q_target(s_prime).max(1)[0].unsqueeze(1)
        target = r + gamma * max_q_prime * done_mask
        loss = F.mse_loss(q_a, target)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Cart-Pole Balancing using DQN

- The main function
- Create the environment
- Create two Q networks: q, and q_target.
- Create and initialize a replay buffer

```python
def main():
    env = gym.make('CartPole-v1')
    q = Qnet()
    q_target = Qnet()
    q_target.load_state_dict(q.state_dict())
    memory = ReplayBuffer()
```

Cart-Pole Balancing using DQN

- Initialize variables
  - print_interval: the interval for printing out the progress
    - After print_interval, we also copy parameters from q to q_target.
  - score: average score during a duration of print_interval.
  - optimizer: the gradient descent algorithm
    - We use the Adam optimizer here.
    - Learning rate is given as an argument to the optimizer.

```python
print_interval = 20
score = 0.0
optimizer = optim.Adam(q.parameters(), lr=learning_rate)
```

Cart-Pole Balancing using DQN

- For each episode,
- Set the epsilon for the episode
  - epsilon is used for selecting actions using the epsilon-greedy policy.
  - In the first episode, epsilon is set to 0.08.
  - In the later episodes, epsilon is decreased linearly until it reaches 0.01.
  - We promote more exploration in the initial stage of training, but reduce exploration as we go.
- Reset the environment to the initial state.
- Set variable 'done' to False. The episode will end when 'done' becomes True.

```python
for n_epi in range(2000):
    epsilon = max(0.01, 0.08 - 0.01*(n_epi/200)) #Linear annealing from 8% to 1%
    s = env.reset()
    done = False
```

Cart-Pole Balancing using DQN

- In the episode,
  - Sample an action using the epsilon-greedy policy.
  - Perform action and get the transition result (s', r, done).
  - Insert the transition into the replay buffer.
  - Move on to the next state.
  - Add reward to 'score'.
  - If the new state is a terminal state, end the episode.

```python
while not done:
    a = q.sample_action(torch.from_numpy(s).float(), epsilon)
    s_prime, r, done, info = env.step(a)
    done_mask = 0.0 if done else 1.0
    memory.put((s,a,r/100.0,s_prime, done_mask))
    s = s_prime

    score += r
    if done:
        break
```

Cart-Pole Balancing using DQN

- After one episode is over,
- Train model q using samples from the replay buffer.
  - We only do training if there are more than 2000 samples in the replay buffer.
- For each print_interval,
- Copy the parameters from q to q_target.
- Print the average score, memory size, and the current epsilon value.
- Reset score to 0.

```
if memory.size()>2000:
    train(q, q_target, memory, optimizer)

if n_epi%print_interval==0 and n_epi!=0:
    q_target.load_state_dict(q.state_dict())
    print("n_episode :{}, score : {:.1f}, n_buffer : {}, eps : {:.1f}%".format(
                                                    n_epi, score/print_interval, memory.size(), epsilon*100))
    score = 0.0
```

Cart-Pole Balancing using DQN

- When we are done running the predefined number of episodes, we close the environment.
- Let the function main() be called when the file is executed.

```
    env.close()

if __name__ == '__main__':
    main()
```

- Try running the file
  - It is considered successful if the average score exceeds 200 constantly.

End of Class

Questions?

Email: jso1@sogang.ac.kr