7.강 Deep Q-learning

Contents

- NN Architecture for V, Q, π
- Deep SARSA
- Deep Q-Learning
- Code Exercise

Neural network optimization

- Mean squared error:

$$L(\theta) = \frac{1}{N}\sum_{i=1}^{N}[y_i - \hat{y}_i]^2$$

- We want to minimize the square of the errors of the neural network estimates

Neural network optimization

- We calculate the gradient vector of the cost function with respect to the θ parameters:

$$\nabla \mathrm{L}(\theta) = \left[\frac{\partial L}{\partial \theta 1}, \frac{\partial L}{\partial \theta 2}, \ldots, \frac{\partial L}{\partial \theta \mathrm{n}}\right]$$

- With the gradient vector, we will make a SGD step:

$$\theta \leftarrow \theta - \alpha \nabla \hat{\mathrm{L}}(\theta)$$

Neural Net. Architecture for V, Q

- St vector input → NN → V scalar output

- St, At vector input → NN → Q scalar output
  - Continuous case

- St vector input → Q vector output
  - Discrete action space only
  - Output size is $|A|$

Neural Net. Architecture for policy, π

- St input → NN → vector output
  - Discrete action space case
  - Output size is $|A|$
  - SOFTMAX turns the output into prob. (sum of prob. is 1)

- St input → NN → $\mu_\theta(S_t), \delta_\theta(St)$ output
  - Continuous action space case
  - Represented with Gaussian Distribution

Deep SARSA

Neural network optimization

$$L(\theta) = \frac{1}{|K|}\sum_{\mathrm{i}=0}^{|K|}[\mathrm{R}_i + \gamma\hat{\mathrm{q}}(\mathrm{S}_i{'}_{,}\,\mathrm{A}_i{'}|\theta_{targ}) - \hat{\mathrm{q}}(\mathrm{S}_i, \mathrm{A}_i|\theta)]^2$$

- Target is the value towards which we want to push the estimates.

$$\mathrm{R}_i + \gamma\hat{\mathrm{q}}(\mathrm{S}_i{'}_{,}\,\mathrm{A}_i{'}|\theta_{targ})$$

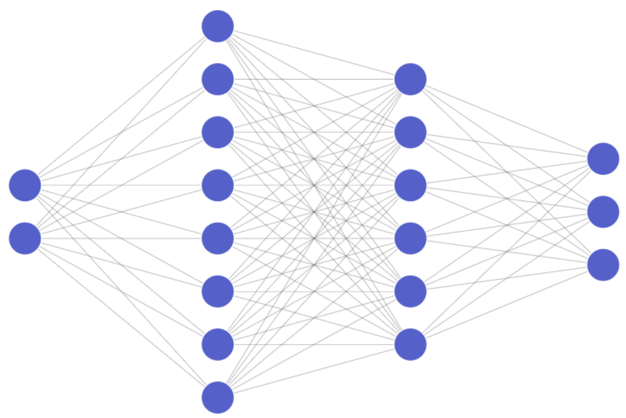
- Estimate is the estimate of the q-value of a state-action pair

$$\hat{\mathrm{q}}(\mathrm{S}_i, \mathrm{A}_i|\theta)$$

Target network

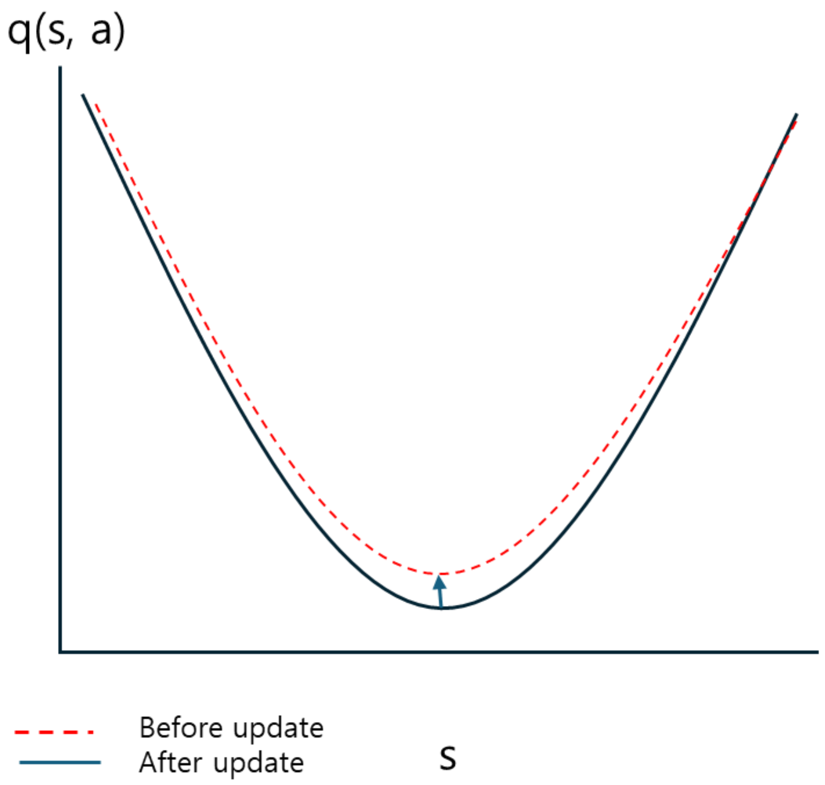
Bootstrapping ✚ Function approximator

$$y_i = R_i + \gamma \hat{q}(S_i{}', A_i{}' | \theta_{targ})$$



Target network

- When a value is changed, nearby values will also be affected.
- By modifying a $\hat{q}(S_i, Ai|\theta)$ estimate we also modify its $\hat{q}(S_i', \ A_i'|\theta_{targ})$ target
- For the learning process to be stable, the target must also be stable
- power of neural networks



Target network

- We make a copy of the neural network to calculate the targets.

$$\theta_{targ} \leftarrow \theta$$

- This neural network does not change with SGD. Its θ parameters remain the same
- The estimated value of $\mathrm{S}_i{}'{}_,\ \mathrm{A}_i{}'$ is calculated with the target network:

$$L(\theta) = \frac{1}{\mathrm{N}}\sum_{\mathrm{i=0}}^{\mathrm{N}}[\mathrm{R}_i + \boxed{\gamma\hat{\mathrm{q}}(\mathrm{S}_i{}'{}_,\ \mathrm{A}_i{}'|\theta_{targ})} - \hat{\mathrm{q}}(\mathrm{S}_i, \mathrm{A}_i|\theta)]^2$$

Neural network optimization

Algorithm 1 Deep SARSA

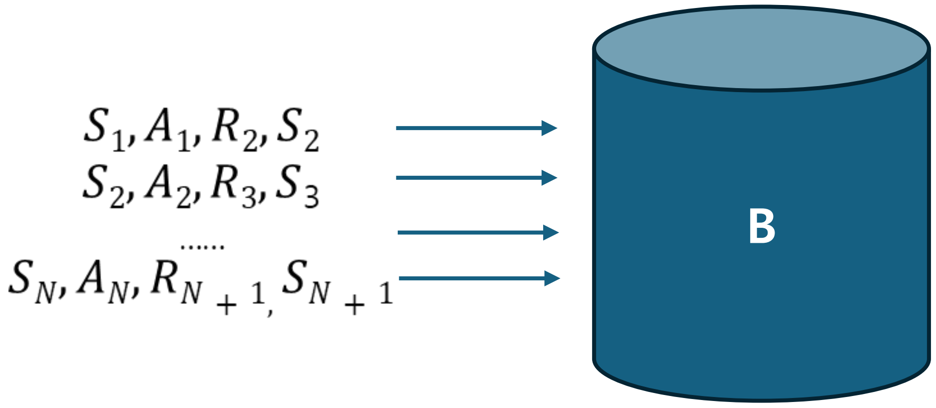
Input: α learning rate, ϵ random action probability,
  γ discount factor,
Initialize q-value parameters θ and target parameters $\theta_{targ} \leftarrow \theta$
$\pi \leftarrow \epsilon$-greedy policy w.r.t $\hat{q}(s, a|\theta)$
Initialize replay buffer B
for episode $\in 1..N$ **do**
  Restart environment and observe the initial state S_0
  for $t \in 0..T-1$ **do**
    Select action $A_t \sim \pi(S_t)$
    Execute action A_t and observe S_{t+1}, R_{t+1}
    Insert transition $(S_t, A_t, R_{t+1}, S_{t+1})$ into the buffer B
    $K = (S, A, R, S') \sim B$
    Select actions $A' \sim \pi(S')$
    Compute loss function over the batch of experiences:

$$\boxed{L(\theta) = \frac{1}{|K|}\sum_{i=1}^{|K|}\left[R_i + \boxed{\gamma\hat{q}(S'_i, A'_i|\theta_{targ})} - \hat{q}(S_i, A_i|\theta)\right]^2} \quad (1)$$

  end for
  Every k episodes synchronize $\theta_{targ} \leftarrow \theta$
end for
Output: Near optimal policy π and q-value approximations $\hat{q}(s, a|\theta)$

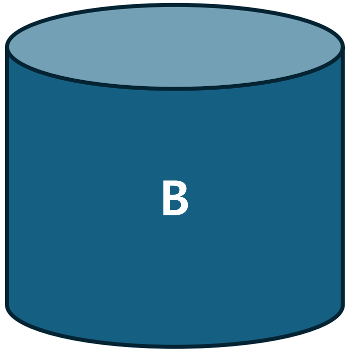
Experience Replay

- Memory that stores the state transition that the agent experiences
- The memory has a limited size and when it fills up, it replaces old transitions with new ones



Experience Replay

- To update the neural network, we randomly chose a batch of transitions from the memory



$K = (S, A, R, S')\sim B$

- The batch of transitions obtained from the memory is used to calculate the cost function and update the θ parameters
- $L(\theta) = \frac{1}{|K|}\sum_{\mathrm{i=0}}^{|K|}[\mathrm{R}_i + \gamma\hat{\mathrm{q}}(\mathrm{S}_i', \mathrm{A}_i'|\theta_{targ}) - \hat{\mathrm{q}}(\mathrm{S}_i, \mathrm{A}_i|\theta)]^2$

Neural network optimization

Algorithm 1 Deep SARSA

Input: α learning rate, ϵ random action probability,
    γ discount factor,
Initialize q-value parameters θ and target parameters $\theta_{targ} \leftarrow \theta$
$\pi \leftarrow \epsilon$-greedy policy w.r.t $\hat{q}(s, a|\theta)$
Initialize replay buffer B
for episode $\in 1..N$ **do**
    Restart environment and observe the initial state S_0
    for $t \in 0..T-1$ **do**
        Select action $A_t \sim \pi(S_t)$
        Execute action A_t and observe S_{t+1}, R_{t+1}
        Insert transition $(S_t, A_t, R_{t+1}, S_{t+1})$ into the buffer B
        $K = (S, A, R, S') \sim B$
        Select actions $A' \sim \pi(S')$
        Compute loss function over the batch of experiences:

$$L(\theta) = \frac{1}{|K|} \sum_{i=1}^{|K|} \left[R_i + \gamma \hat{q}(S'_i, A'_i|\theta_{targ}) - \hat{q}(S_i, A_i|\theta)\right]^2 \qquad (1)$$

    end for
    Every k episodes synchronize $\theta_{targ} \leftarrow \theta$
end for
Output: Near optimal policy π and q-value approximations $\hat{q}(s, a|\theta)$

Code Ex.

- MountainCar: Reach the goal from the bottom of the valley

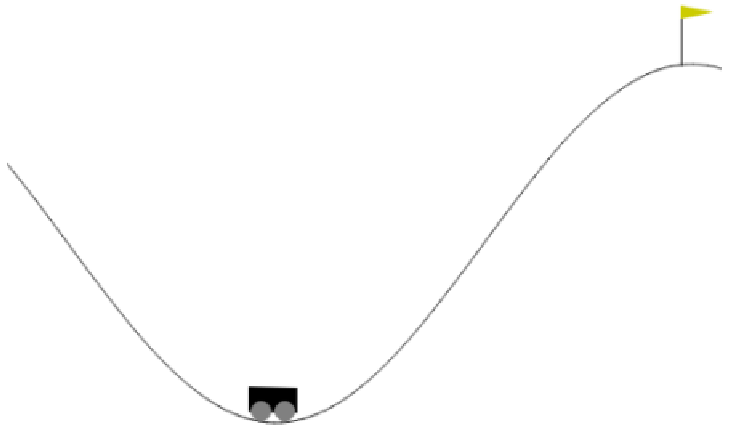
The state

The observation space consists of the car position $\in [-1.2, 0.6]$ and car velocity $\in [-0.07, 0.07]$

The actions available

The actions available three:

```
0    Accelerate to the left.
1    Don't accelerate.
2    Accelerate to the right.
```



Code Ex.

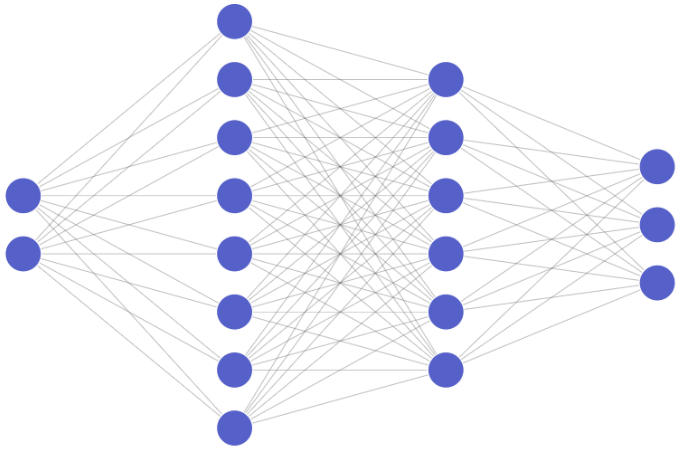
- deep_sarsa_colab.ipynb

Deep Q-learning

Deep Q-learning

- Q-learning + Neural Network
- Swap the q-value table for a neural network
- Tackle more difficult problems
- Leverage generalization power of neural networks

S=[s1,s2]



$\hat{q} = [\hat{q}(s,s_1), \hat{q}(s,s_2), \hat{q}(s,s_3)]$

Neural network optimization

- We calculate the gradient vector of the cost function with respect to the θ parameters:

$$\nabla \hat{\mathrm{L}}(\theta) = \left[\frac{\partial L}{\partial \theta 1}, \frac{\partial L}{\partial \theta 2}, \ldots, \frac{\partial L}{\partial \theta \mathrm{n}}\right]$$

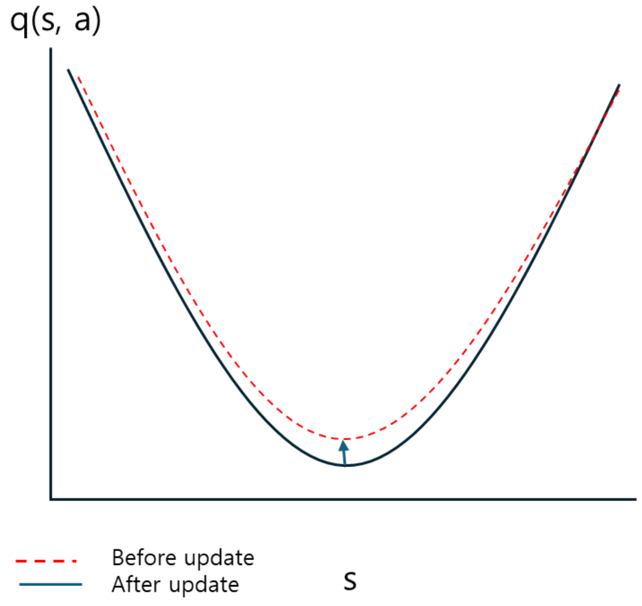
- With the gradient vector, we will make a SGD step:

$$\theta \leftarrow \theta - \alpha \nabla \hat{\mathrm{L}}(\theta)$$

Target network

- Two techniques combined generate an unstable learning process.
- Bootstrapping($y_i = R_i + \gamma\hat{q}(S_i', A_i'|\theta_{targ})$) + function approximator
- Why? : When a value is changed, nearby values will also be affected.

- When a value is changed, nearby values will also be affected.
- By modifying a $\hat{q}(S_i, Ai|\theta)$ estimate we also modify its $\hat{q}(S_i,' Ai'|\theta_{targ})$ target
- For the learning process to be stable, the target must also be stable



Target network

- We make a copy of the neural network to calculate the targets.

$$\theta_{targ} \leftarrow \theta$$

- This neural network does not change with SGD. Its θ parameters remain the same
- The estimated value of $S_i{'}, A_i{'}$ is calculated with the target network:

$$\hat{L}(\theta) = \frac{1}{N}\sum_{i=0}^{N}[R_i + \boxed{\gamma\hat{q}(S_i{'}, A_i{'}|\theta_{targ})} - \hat{q}(S_i, A_i|\theta)]^2$$

Deep Q-learning

Algorithm 1 Deep Q-Learning

Input: α learning rate, ϵ random action probability,
    γ discount factor,
Initialize q-value parameters θ and target parameters $\theta_{targ} \leftarrow \theta$
$b \leftarrow \epsilon$-greedy policy w.r.t $\hat{q}(s, a|\theta)$
$\pi \leftarrow$ greedy policy w.r.t $\hat{q}(s, a|\theta)$
Initialize replay buffer B
for episode $\in 1..N$ **do**
    Restart environment and observe the initial state S_0
    for $t \in 0..T-1$ **do**
        Select action $A_t \sim b(S_t)$
        Execute action A_t and observe S_{t+1}, R_{t+1}
        Insert transition $(S_t, A_t, R_{t+1}, S_{t+1})$ into the buffer B
        $K = (S, A, R, S') \sim B$
        Select actions $A' \sim \pi(S')$
        Compute loss function over the batch of experiences:

$$L(\theta) = \frac{1}{|K|} \sum_{i=1}^{|K|} \left[\boxed{R_i + \gamma \hat{q}(S_i', A_i' | \theta_{targ})} - \hat{q}(S_i, A_i | \theta)\right]^2 \tag{1}$$

    end for
    Every k episodes synchronize $\theta_{targ} \leftarrow \theta$
end for
Output: Near optimal policy π and q-value approximations $\hat{q}(s, a|\theta)$

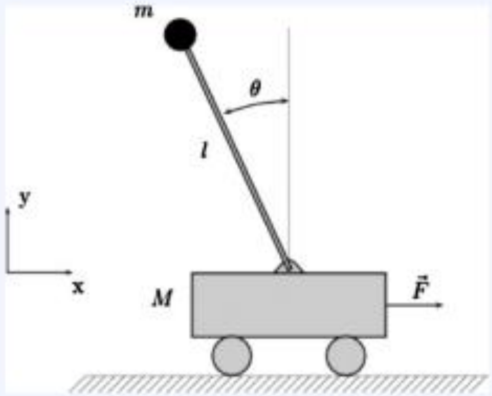
Code Ex.

- Cartpole: Keep the tip of the pole straight

- States: Pole angle and angular velocity
- Actions: Move left, right
- Transition model:

$$\begin{bmatrix} \dot{x} \\ \ddot{x} \\ \dot{\theta} \\ \ddot{\theta} \end{bmatrix} = \begin{bmatrix} \dot{x} \\ \frac{-ml\sin(\theta)\dot{\theta}^2 + F + mg\cos(\theta)\sin(\theta)}{M+m-m\cos(\theta)^2} \\ \dot{\theta} \\ \frac{-ml\cos(\theta))\sin(\theta)\dot{\theta}^2 + F\cos(\theta) + mg\sin(\theta) + Mg\sin(\theta)}{l*(M+m-m\cos(\theta)^2} \end{bmatrix}$$

- Rewards:
  - +1: Survive
  - 0: Fall



https://danielpiedrahita.wordpress.com/portfolio/cart-pole-control/

If we know angle and velocity, we can calculate accelerations (Able to describe the whole dynamics)

Code Ex.

The state

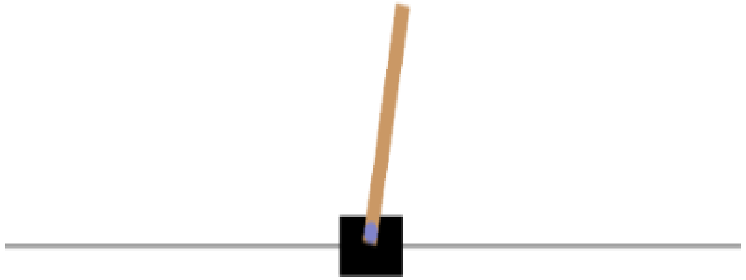
The states of the cartpole task will be represented by a vector of four real numbers:

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	-0.418 rad (-24 deg)	0.418 rad (24 deg)
3	Pole Angular Velocity	-Inf	Inf

The actions available

We can perform two actions in this environment:

```
0    Apply +1 torque on the joint between the links.
1    Do nothing
2    Apply -1 torque on the joint between the links.
```



Reward: +1 for every step

Code Ex.

```
env = gym.make('CartPole-v0')
#seed_everything(env)
env.reset()
plt.imshow(env.render(mode='rgb_array'))
```

```
state_dims = env.observation_space.shape[0]
num_actions = env.action_space.n
print(f"CartPole env: State dimensions: {state_dims}, Number of actions: {num_actions}")
```

```
CartPole env: State dimensions: 4, Number of actions: 2
```

Code Ex.

Create the Q-Network: $\hat{q}(s, a|\theta)$

```
q_network = nn.Sequential(
    nn.Linear(state_dims, 128),
    nn.ReLU(),
    nn.Linear(128, 64),
    nn.ReLU(),
    nn.Linear(64, num_actions))
```

액션의 개수만큼 output이 나옴

Create the target Q-Network: $\hat{q}(s, a|\theta_{targ})$

```
target_q_network = copy.deepcopy(q_network).eval()
```

Code Ex.

Unsqueeze(dim=0) put a extra. dim. in front.
View(1,-1) put a extra. dim. in front.

For every step,

- **Sample state:** tensor([[-0.0220, -0.0468, 0.0114, -0.0126]])
- **Next state:** tensor([[-0.0230, -0.2421, 0.0112, 0.2836]])
- **Reward:** tensor([[1.]])
- **Done:** tensor([[False]])

```python
class PreprocessEnv(gym.Wrapper):

    def __init__(self, env):
        gym.Wrapper.__init__(self, env)

    def reset(self):
        obs = self.env.reset()
        return torch.from_numpy(obs).unsqueeze(dim=0).float()

    def step(self, action):
        action = action.item()
        next_state, reward, done, info = self.env.step(action)
        next_state = torch.from_numpy(next_state).unsqueeze(dim=0).float()
        reward = torch.tensor(reward).view(1, -1).float()
        done = torch.tensor(done).view(1, -1)
        return next_state, reward, done, info
```

Experience Replay

$[[S_1, A_1, R_2, S_2],\ [S_2, A_2, R_3, S_3]]$

$[[S_1, S_2], [A_1, A_2], [R_2, R_3], [S_2, S_3]]$

$[[[S_1], [S_2]], [[A_1], [A_2]], [[R_2], [R_3]], [[S_2], [S_3]]]$
A batch shape : N x D for each

Torch.cat은 default가 행(0) 방향으로 붙임
S, A, R, 에 대해 각각 batch(N x D) 로 바꿔 줌

```
class ReplayMemory:

    def __init__(self, capacity=1000000):
        self.capacity = capacity
        self.memory = []
        self.position = 0

    def insert(self, transition):
        if len(self.memory) < self.capacity:
            self.memory.append(None)
        self.memory[self.position] = transition
        self.position = (self.position + 1) % self.capacity

    def sample(self, batch_size):
        assert self.can_sample(batch_size)

        batch = random.sample(self.memory, batch_size)
        batch = zip(*batch)
        return [torch.cat(items) for items in batch]

    def can_sample(self, batch_size):
        return len(self.memory) >= batch_size * 10

    def __len__(self):
        return len(self.memory)
```

Deep Q-learning

$\pi \leftarrow$ greedy policy w.r.t $\hat{q}(s, a|\theta)$

A batch shape : N x D.

$[[[S_1], [S_2]], [[A_1], [A_2]], [[R_2], [R_3]],[[S_2], [S_3]]]$
A batch shape : N x D for each

각 state당 액션의 개수만큼 output이 나옴.
예를들어 $\hat{q}(S_i|\theta)$ 는 $[[(\hat{q}(si)\ for\ A_1)], [\hat{q}(si)\ for\ A_2]]$

$\hat{q}(S_i,' Ai'|\theta_{targ})$, where $A_i' \sim \pi(S')$

$$\hat{L}(\theta) = \frac{1}{N}\sum_{i=0}^{N}[R_i + \gamma\hat{q}(S_i{}', A_i'|\theta_{targ}) - \hat{q}(S_i, A_i|\theta)]^2$$

$$\nabla\hat{L}(\theta) = [\frac{\partial L}{\partial\theta 1}, \frac{\partial L}{\partial\theta 2}, \ldots, \frac{\partial L}{\partial\theta n}]$$

$$\theta \leftarrow \theta - \alpha\nabla\hat{L}(\theta)$$

```python
def deep_q_learning(q_network, policy, episodes,
                    alpha=0.0001, batch_size=32, gamma=0.99, epsilon=0.2):

    optim = AdamW(q_network.parameters(), lr=alpha)
    memory = ReplayMemory()
    stats = {'MSE Loss': [], 'Returns': []}

    for episode in tqdm(range(1, episodes + 1)):
        state = env.reset()
        done = False
        ep_return = 0
        while not done:
            action = policy(state, epsilon)
            next_state, reward, done, _ = env.step(action)

            memory.insert([state, action, reward, done, next_state])

            if memory.can_sample(batch_size):
                state_b, action_b, reward_b, done_b, next_state_b = memory.sample(batch_size)
                qsa_b = q_network(state_b).gather(1, action_b)

                next_qsa_b = target_q_network(next_state_b)
                next_qsa_b = torch.max(next_qsa_b, dim=-1, keepdim=True)[0]

                target_b = reward_b + ~done_b * gamma * next_qsa_b
                loss = F.mse_loss(qsa_b, target_b)
                q_network.zero_grad()
                loss.backward()
                optim.step()

                stats['MSE Loss'].append(loss)

            state = next_state
            ep_return += reward.item()

        stats['Returns'].append(ep_return)

        if episode % 10 == 0:
            target_q_network.load_state_dict(q_network.state_dict())

    return stats
```

Deep Q-learning

- You can cut a gradient by calling `y.detach()`, which will return a new tensor with `required_grad=False`. Note that `detach` is not an in-place operation! You would want this during evaluation.
- You also can't convert a tensor with `requires_grad=True` to numpy (for the same reason as above). Instead, you need to detach it first, e.g. `y.detach().numpy()`

- y.clone() vs. y.detach():
- If y change, y.clone() not change.
- If y change y.detach() change

Create the exploratory policy: $b(s)$

```
def policy(state, epsilon=0.):
    if torch.rand(1) < epsilon:
        return torch.randint(num_actions, (1, 1))
    else:
        av = q_network(state).detach()
        return torch.argmax(av, dim=-1, keepdim=True)
```

Code Ex.

- Open deep_q_learning_colab.ipynb