# XAI613 Fall 2024
# Assignment 2
# Due: October 21 at 9:00 am (KST)

## 1 Toy Environment [6 pts]

In this assignment, we will implement DQN algorithm but with simple linear function approximator, and run the implemented algorithm on a toy domain. The followings describe the details of the corresponding toy environment. It is implemented as `EnvTest` class in `utils/test_env.py` in your provided skeleton code, so you can check it out by yourself.

- 4 states: $0, 1, 2, 3$
- 5 actions: $0, 1, 2, 3, 4$. Action $0 \leq i \leq 3$ goes to state $i$, while action 4 makes the agent stay in the same state.
- Rewards: Going to state $i$ from states 0, 1, and 3 gives a reward $R(i)$, where $R(0) = 0.2, R(1) = -0.1, R(2) = 0.0, R(3) = -0.3$. If we start in state 2, then the rewards defind above are multiplied by $-10$. See Table 1 for the full transition and reward structure.
- One episode lasts 5 time steps (for a total of 5 actions) and always starts in state 0 (no rewards at the initial state).

| State ($s$) | Action ($a$) | Next State ($s'$) | Reward ($R$) |
|---|---|---|---|
| 0 | 0 | 0 | 0.2 |
| 0 | 1 | 1 | -0.1 |
| 0 | 2 | 2 | 0.0 |
| 0 | 3 | 3 | -0.3 |
| 0 | 4 | 0 | 0.2 |
| 1 | 0 | 0 | 0.2 |
| 1 | 1 | 1 | -0.1 |
| 1 | 2 | 2 | 0.0 |
| 1 | 3 | 3 | -0.3 |
| 1 | 4 | 1 | -0.1 |
| 2 | 0 | 0 | -2.0 |
| 2 | 1 | 1 | 1.0 |
| 2 | 2 | 2 | 0.0 |
| 2 | 3 | 3 | 3.0 |
| 2 | 4 | 2 | 0.0 |
| 3 | 0 | 0 | 0.2 |
| 3 | 1 | 1 | -0.1 |
| 3 | 2 | 2 | 0.0 |
| 3 | 3 | 3 | -0.3 |
| 3 | 4 | 3 | -0.3 |

Table 1: Transition table for the Test Environment

(a) **(written)** What is the maximum sum of rewards that can be achieved in a single trajectory in the test environment, assuming $\gamma = 1$? Show first that this value is attainable in a single trajectory, and then briefly argue why no other trajectory can achieve greater cumulative reward.

## 2 Tabular Q-Learning [3 pts]

If the state and action spaces are sufficiently small, we can simply maintain a table containing the value of $Q(s, a)$, an estimate of $Q^*(s, a)$, for every $(s, a)$ pair. In this *tabular setting*, given an experience sample $(s, a, r, s')$, the update rule is

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left( r + \gamma \max_{a' \in \mathcal{A}} Q(s', a') - Q(s,a) \right)$$

where $\alpha > 0$ is the learning rate, $\gamma \in [0, 1)$ the discount factor.

**$\varepsilon$-Greedy Exploration Strategy** For exploration, we use an $\varepsilon$-greedy strategy. This means that with probability $\varepsilon$, an action is chosen uniformly at random from $\mathcal{A}$, and with probability $1 - \varepsilon$, the greedy action (i.e., $\arg\max_{a \in \mathcal{A}} Q(s, a)$) is chosen.

(a) **(coding)** Implement the `get_action` and `update` functions in `q2_schedule.py`. Test your implementation by running `python q2_schedule.py`.

## 3 Q-Learning with Function Approximation [18 pts]

For some complex environments, we cannot reasonably learn and store a Q value for each state-action tuple. We will instead represent our Q values as a parametric function $Q_\theta(s, a)$ where $\theta \in \mathbb{R}^p$ are the parameters of the function (typically the weights and biases of a linear function or a neural network). In this *approximation setting*, the update rule becomes

$$\theta \leftarrow \theta + \alpha \left( r + \gamma \max_{a' \in \mathcal{A}} Q_\theta(s', a') - Q_\theta(s,a) \right) \nabla_\theta Q_\theta(s,a)$$

where $(s, a, r, s')$ is a transition from the MDP.

To improve the data efficiency and stability of the training process, DQN employed two strategies:

- A *replay buffer* to store transitions observed during training. When updating the $Q$ function, transitions are drawn from this replay buffer. This improves data efficiency by allowing each transition to be used in multiple updates.
- A *target network* with parameters $\bar{\theta}$ to compute the target value of the next state, $\max_{a'} Q(s', a')$. The update becomes

$$\theta \leftarrow \theta + \alpha \left( r + \gamma \max_{a' \in \mathcal{A}} Q_{\bar{\theta}}\left(s', a'\right) - Q_\theta(s,a) \right) \nabla_\theta Q_\theta(s,a) \tag{1}$$

Updates of the form (1) applied to transitions sampled from a replay buffer $\mathcal{D}$ can be interpreted as performing stochastic gradient descent on the following objective function:

$$L_{\text{DQN}}(\theta) = \mathop{\mathbb{E}}_{(s,a,r,s') \sim \mathcal{D}} \left[ \left( r + \gamma \max_{a' \in \mathcal{A}} Q_{\bar{\theta}}(s', a') - Q_\theta(s,a) \right)^2 \right]$$

Note that this objective is also a function of both the replay buffer $\mathcal{D}$ and the target network $Q_{\bar{\theta}}$. The target network parameters $\bar{\theta}$ are held fixed and not updated by SGD, but periodically – every $C$ steps – we synchronize by copying $\bar{\theta} \leftarrow \theta$.

(a) **(coding)** We start by implementing linear approximation in PyTorch. This question will set up the pipeline for the remainder of the assignment. You'll need to implement the following functions in `q3_linear_torch.py` (please read through `q3_linear_torch.py`):

- `initialize_models`
- `get_q_values`
- `update_target`
- `calc_loss`
- `add_optimizer`

Test your code by running `python q3_linear_torch.py`. This will run linear approximation with PyTorch on the test environment from Problem 1. Running this implementation should only take a minute. [15 pts]

(b) **(written)** Do you reach the optimal achievable reward on the test environment? Attach the plot `scores.png` from the directory `results/q3_linear` to your writeup. [3 pts]

**How to submit** For the written question write your answers with any software that can export a `pdf` file. Zip `q2_schedule.py`, `q3_linear_torch.py` files and your `pdf` file into the zip file named `StudentID_YourName.zip`. Submit the `zip` file through Blackboard.