MICAS912: Sequential Decision-Making Processing
Part II - Reinforcement Learning

Mireille Sarkiss

Abdelaziz Bounhar, Ibrahim Djemai

Lab 2

In this Lab, we consider the same problem we studied in Lab 1, i.e. scheduling the communication of users in a network, see Figure 1

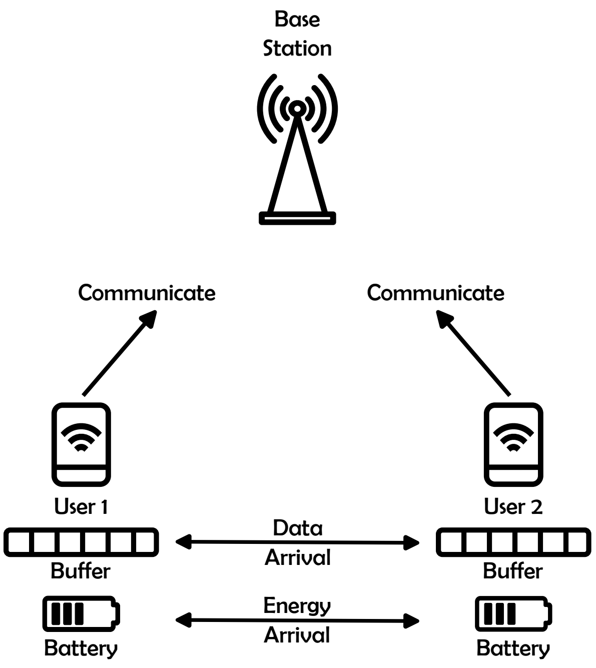

Figure 1: System model in the Uplink (UL) and Downlink (DL) for the centralized scheduling problem of one covert and one non-covert users

You are given the implementation of the Environment and you are asked to implement the **Q-Learning** algorithm as well as the **Deep Q-Learning** algorithm to solve this problem.
A template of these algorithms is given to you and you should complete it. A refresher on how **Deep Q-Learning** work is available in the Appendix A. Best of success!

A Deep Q-Learning

When the state and actions spaces grow larger, modeling them using Q-matrices can be memory consuming. In addition, the time required for exploring the state space sufficiently to determine what action to take for each circumstance increases, and thus making it harder for QL and 2QL to converge on a policy. Therefore, Deep Q-Learning [1] or Deep Q-Network (DQN) was introduced to tackle the curse of dimensionality that traditional algorithms face, by using NNs to model the environment rather than matrices. Thanks to the generalization abilities of NNs, DQN is able to approximate the Q-matrix using a function of parameters $\mathbf{w}$, even without exploring the entirety of the state space $\mathbf{S}$.

Specifically, a given state s is fed into the DQN's Neural Network of parameters $\mathbf{w}$. The output is an estimation of the Q-values denoted as $\mathbf{Q}^{\mathbf{w}}(s, a)$, which indicates the state-action values for the state s. An action a is chosen based on the obtained Q-values (in training, the action is sampled following the ϵ-Greedy method, whereas during the test, the action is the one that achieves the highest Q-value) and the environment transitions to a new state s', with a reward $R^a_{ss'}$, as illustrated in Figure 2.

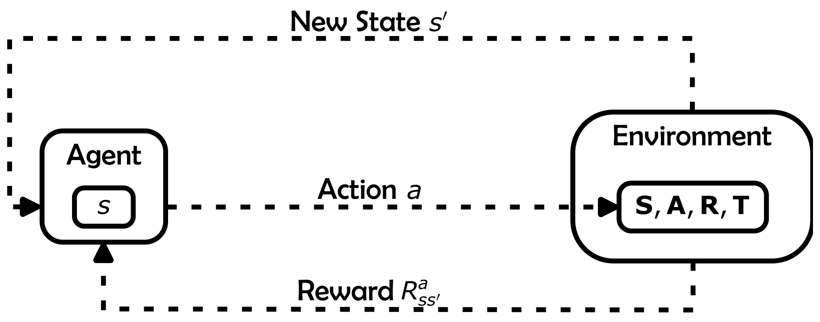

Figure 2: DQN agent exploring the environment.

To train the DQN agent, a loss function $\mathcal{L}^{\mathbf{w}}_{DQN}$ is used to evaluate the quality of the produced Q-values. More specifically, the estimate $\mathbf{Q}^{\mathbf{w}}(s, a)$ for a state s, is used to take an action a, transition to a next state s' and receive a reward $R^a_{ss'}$. The result is then compared with the Bellman equation of the Q-function using the DQN Q-values estimation of the next state $\mathbf{Q}^{\mathbf{w}}(s', a')$, expressed in the following equation:

$$\hat{\mathbf{Q}}^{\mathbf{w}}(s, a) = R^a_{ss'} + \gamma \cdot \max_{a' \in \mathbf{A}} \mathbf{Q}^{\mathbf{w}}(s', a'). \tag{1}$$

From that, the loss function computes the error between the two estimates $\mathbf{Q}^{\mathbf{w}}(s, a)$ and $\hat{\mathbf{Q}}^{\mathbf{w}}(s, a)$, and the DQN agent uses that to update its parameters $\mathbf{w}$ with Gradient Descent, and improve its estimation quality.

However, using the parameters $\mathbf{w}$, initially used to estimate $\mathbf{Q}^{\mathbf{w}}(s, a)$, to compute the Q-values of the next state $\mathbf{Q}^{\mathbf{w}}(s', a')$ when evaluating the Bellman estimation $\hat{\mathbf{Q}}^{\mathbf{w}}(s, a)$, to later update the same parameters can cause the issue of a moving target, i.e., the convergence point is constantly moving and reaching it becomes more difficult. To combat this issue, a second NN can be used with different parameters $\mathbf{w}'$. This network will be used to compute the Bellman equation and offer more stability to the training of the parameters $\mathbf{w}$ of the DQN agent. Notably, the first NN is named the **Evaluation Network** (EN), while the second NN is named the **Target Network** (TN). To avoid training two separate NNs, the TN parameters $\mathbf{w}'$ will be a **copy** of the EN parameters **$\mathbf{w}$ that do not get updated as often**. More specifically, for a certain number of episodes, the parameters $\mathbf{w}'$ are fixed to allow the parameters $\mathbf{w}$ to converge towards a fixed target. After that, an update on the TN parameters is done to make them equal to the EN parameters $\mathbf{w}' = \mathbf{w}$. This is done to improve the Bellman estimation using an updated version of the Q-values $\mathbf{Q}^{\mathbf{w}'}(s, a)$. The process is repeated until convergence.

Moreover, the sequential nature of MDP problems creates a correlation in the observed states used to train the DQN. If the system is unstable, the correlation aspect can yield to high variance updates of the agent's parameters, and therefore impact the performance and convergence of the DQN [3]. From that, DQN uses **Experience Replay** (ER), a technique implemented to train the agent more efficiently, and with more stability. ER consists of collecting the transitions of the agent in the environment, and storing them in a memory buffer. This memory buffer is then used to replay the transitions played in a randomized way (to break the correlation), and compute the loss function. Following this method, training the DQN agent becomes more stable against the variations in the transitions of the environment.
In what follows, we detail the training procedure of the DQN:

1. Starting with an initial state $S_t = s$ at timestep t, we compute the Q-values $\mathbf{Q}^{\mathbf{w}}(s,a)$ using the EN, choose an action $A_t = a$ following the ϵ-Greedy approach, transition to a new state $S_{t+1} = s'$ and receive a reward $R^a_{ss'}$. The tuple $\left(s, \mathbf{Q}^{\mathbf{w}}(s,a), s', R^a_{ss'}\right)$ is saved in the ER memory buffer, and the system transitions to a new timestep. The next state s' becomes the current state s and the process is repeated until the end of the episode, where a new initial state is used to start a new episode.

2. Once the memory buffer is filled with a sufficient number of tuples to form a mini-batch, the learning process starts, in which the sampled tuples are shuffled to break the correlation between the transitions to form a mini-batch.

3. The obtained mini-batch, denoted as b and of size $|b|$, is then used to compute the Bellman step using the TN for each tuple $\left(s_i, \mathbf{Q}^{\mathbf{w}}_i(s_i,a_i), s'_i, R^a_{ss',i}\right)$ of index i in the mini-batch, following the equation:
$$\hat{\mathbf{Q}}^{\mathbf{w}'}_i(s_i,a_i) = R^a_{ss',i} + \gamma \cdot \max_{a' \in \mathbf{A}} \mathbf{Q}^{\mathbf{w}'}_i(s'_i,a'). \tag{2}$$

4. The obtained Bellman estimations of the mini-batch b are then used to compute the loss $\mathcal{L}^{\mathbf{w},\mathbf{w}',b}_{DQN}$, which is the Mean Squared Error (MSE) between the Bellman estimates and the Q-values outputs of the EN. The loss function is expressed as follows:
$$\mathcal{L}^{\mathbf{w},\mathbf{w}',b}_{DQN} = \frac{1}{|b|}\sum_{i=1}^{|b|}\left(\hat{\mathbf{Q}}^{\mathbf{w}'}_i(s_i,a_i) - \mathbf{Q}^{\mathbf{w}}_i(s_i,a_i)\right)^2. \tag{3}$$

5. The parameters $\mathbf{w}$ are then updated using the loss $\mathcal{L}^{\mathbf{w},\mathbf{w}',b}_{DQN}$ following the Gradient Descent method with a learning rate α:
$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} \mathcal{L}^{\mathbf{w},\mathbf{w}',b}_{DQN}. \tag{4}$$

6. The parameters $\mathbf{w}'$ of the TN are updated every few episodes, to be the same as $\mathbf{w}$. The process described in the outlined steps is repeated until convergence.

Other features were added to DQN in order to improve its performance and accelerate its convergence. The work in [2] used prioritized ER to sample experiences that have high expected learning progress from the memory buffer with higher probability compared to experiences with lower expected learning progress. The estimation of the expected learning progress was done using TD-error of the experiences. Moreover, the paper [4] introduced a dual architecture to the DQN, where one network estimated the state value function and a separate network estimated the advantage function. Both network outputs were aggregated to produce the Q-values. The advantage of this approach was to make the distinction between actions in states where immediate reward is preferred (i.e., maximize the Advantage function), and actions where the long-term reward is better (i.e., maximize the value function).

References

[1] V. Mnih et al. “Human-level control through deep reinforcement learning”. In: *nature* 518.7540 (2015), pp. 529–533.

[2] Tom Schaul et al. “Prioritized experience replay”. In: *arXiv preprint arXiv:1511.05952* (2015).

[3] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[4] Ziyu Wang et al. “Dueling network architectures for deep reinforcement learning”. In: *International conference on machine learning.* PMLR. 2016, pp. 1995–2003.