Ben-Gurion University of the Negev

Deep Reinforcement Learning

DQN - CartPole

August 2020

Lecturer: Armin Biess, Ph.D.

MSc Student: Tom Landman

1. Introduction

This report composed of two main sections: Theory and Evaluation. In the first section, some key concepts regarding Deep Reinforcement Learning, are defined, explained, and mainly referred from Google DeepMind's Playing Atari article [2] and Sutton and Barto Book of Reinforcement Learning [1]. In the second section, the evaluation procedure, design of experiments, and results of the given CartPole agent are displayed, compared, and discussed after the completion of the DQN skeleton and ReplayBuffer classes implementations according to the provided CartPole environment. Also note that using Pytorch, the implemented DQN model was trained on internal NVIDIA GeForce GTX 1060 with Max-Q Design GPU with CUDA support.

2. Theory

This section presents important key concepts, algorithms, and optimization methods relate to the theory of *Reinforcement Learning* (RL) in general and *Deep Reinforcement Learning* (DRL) in particular. In the first subsection, an elaborated explanation regarding the *Epsilon-Greedy* policy is provided. Second, the Q-Learning algorithm is presented and discussed due to its contribution that served as one of the core elements for developing the Google DeepMind's Deep Q-Networks (DQN) algorithm, introduced in the third and final subsection including information regarding the use of several optimization improvements techniques based on Experience Replay Buffer and target network updates.

2.1 Epsilon-Greedy

In the 1st assignment, in order to find the optimal policy, a Markov Decision Process (MDP) model was used for implementing *Value Iteration* and *Policy Iteration* algorithms. Based on Sutton and Barto [1] In both of these algorithms the optimal policy derived by using the simplest action selection rule of selecting greedily an action based on the highest estimated state value function v or action-value of the Q function respectively. For example, in the case of the action-value function, that is, to select at step t one of the greedy actions, A_t^*, for which $Q_t(A_t^*) = \max_a Q_t(a)$. Therefore, this greedy action selection method can be written as $A_t = \underset{a}{argmax}\, Q_t(a)$. Since greedy action selection always exploits the agent's current knowledge to maximize its immediate reward, it doesn't spend any time to select less desired actions in order to explore and see if they might be better and retrieve a better policy. One of the common methods from which the issue of balancing between exploration and exploitation of the agent can be addressed is by choosing an action randomly in a small fraction of time and denoted as the *ε-greedy*. The *ε-greedy* method causes the agent to behave greedily most of the time, but sometimes, say with small probability ε, instead the agent selects an action randomly amongst all the actions with equal probability independently of the action-value estimates. As a result, the agent can both explore the environment and exploit its knowledge. There are several approaches for an agent to learn a certain policy in which the task of tuning the ratio between exploration and exploitation is being addressed. For example, Singh et al. [3] discuss the decaying exploration learning policy that becomes more and more like the greedy learning policy over time, and a persistent exploration learning policy that does not. The advantage of decaying exploration policies is that the actions taken by the system may converge to the optimal ones eventually, but with the price that their ability to adapt slows down. Further, the class of decaying exploration learning policies characterized by the following two properties:

1. each action is executed infinitely often in every state that is visited infinitely often.

2. In the limit, the learning policy is greedy with respect to its Q-value function with probability 1.

Learning policies that satisfy the above conditions are denoted as *GLIE*, which stands for "greedy in the limit with infinite exploration". For *ε-greedy* exploration [1], at timestep t in state s picks a random exploration action a with probability $\varepsilon_t(s)$ and the greedy action with probability $1 - \varepsilon_t(s)$. When using GLIE the *ε-greedy* initially performs worse because it explores more, however, it performs better because its exploration decreases with time (e.g. exploitation increases).

2.2 Q-Learning

Q-Learning is an off-policy temporal difference (TD) control algorithm, meaning the learned action-value function, Q, directly approximates q_* which is the optimal action-value function, independent of the policy being followed. As a result, early convergence proof is enabled, moreover, the analysis of this algorithm is simplified. One-step Q-learning is defined by the following update Eq. 1:

$$Q(S_t, A_t) \doteq Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right] \quad (1)$$

The policy still has an effect in the manner it determines and updates which state-action pairs need to be visited. Under the assumption that all state-action pairs continue to be updated in addition to a variant of stochastic approximation conditions on the sequence of step-size parameters (e.g. α), Q has been shown to successfully converge with probability 1 to q_*. The Q-learning algorithm is presented in Fig. 1.

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$;
  until S is terminal

Figure 1 - Q-Learning: An off-policy TD control algorithm (Sutton and Barto, 2018)

From the aspect of Q-learning's sample backup diagram, as can be seen in Fig. 2 below, the update rule updates a state-action pair, hence, the top node which is the root of the backup must be a small filled action node. Whereas the backup is also from action nodes, maximizing over all possible actions in the next state represented by the bottom nodes of the backup diagram.

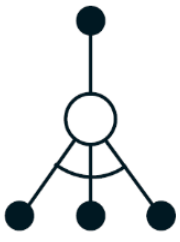

Figure 2 - Q-Learning's Sample Backup Diagram (Sutton and Barto, 2018)

2.3 Deep Q–Networks (DQN)

From a Deep Learning (DL) point of view, RL presents several challenges. First, most of the successful DL-based applications require large amounts of labeled training data which is different from the case of RL algorithm where the agent's goal is to be able to learn from a scalar reward signal that is frequently sparse, noisy and delayed between the action and the resulting rewards. Second, most DL algorithms assume that data samples to be independent (I.I.D), while in RL, the data distribution changes as the algorithm learn new behaviors, and further, sequences of highly correlated states can be encountered. Hence, can be problematic for DL algorithms that assume a fixed underlying distribution.

In order to overcome these challenges, the authors from DeepMind introduced *Deep Q-Network* (DQN), a *Deep Reinforcement Learning* (DRL) value-based method using *Convolutional Neural Networks* (CNN) trained with a variant of the Q-learning algorithm, with stochastic gradient descent to update the weights. Furthermore, to address the issues of correlated data and non-stationary distributions, the authors used an experience replay mechanism which randomly samples previous transitions, and thereby smooths the training distribution over many past behaviors. Also, note that the proposed method was evaluated and applied to a range of Atari 2600 games implemented in The Arcade Learning Environment (ALE). Furthermore, as can be seen in the DQN algorithm presented in Fig 3., at each time-step t the agent's experiences are stored as a tuple e_t, with the following structure $e_t = (s_t, a_t, r_t, s_{t+1})$. Moreover, these tuples are being stored into a replay memory dataset $D = e_1, e_2, \ldots\ldots, e_N$ pooled over many episodes. Note that during the inner loop of the algorithm, Q-learning updates or minibatch updates are applied to samples of experience, $e \sim D$, drawn at random from the replay memory of stored samples. Then, after performing experience replay, an action a is selected and executed by the agent according to an ε-greedy policy. Furthermore, using arbitrary length histories as inputs to a neural network can be difficult, therefore, the Q-function implemented on a fixed-length representation of histories produced by a function ϕ.

Algorithm: Deep Q-Network (DQN) with Experience Replay

Input: the pixels and the game score, a preprocessing map ϕ
Output: Q action value function (from which we obtain policy and select action)
Initialize replay buffer $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights $\boldsymbol{\theta}$
Initialize target action-value function Q with random weights $\boldsymbol{\theta}^- = \boldsymbol{\theta}$
  for $episode = 1, to\ M$ **do**
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    for $t = 1, T$ **do**
      ε-greedy: select $a_t = \begin{cases} \text{a random action}, & \text{with prob } \varepsilon \\ \text{argmax}_a\, Q(\phi(s_t), a; \boldsymbol{\theta}), & \text{otherwise} \end{cases}$
      Execute action a_i in emulator and observe reward r_t and image x_{t+1}
      Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
      Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
      Sample random minibatch B of transitions $(\phi_j, a_j, r_j, \phi_j)$ from $\mathcal{D}$
      Set $y_j = \begin{cases} r_j, & \text{episode terminates at } j+1 \\ r_j, + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \boldsymbol{\theta}^-) & \text{otherwise} \end{cases}$
      Perform a gradient descent step on $(y_i - Q(\phi_j, a_j; \boldsymbol{\theta}))^2$ wrt to the network parameter $\boldsymbol{\theta}$
      Every C steps reset $\hat{Q} = Q$, $\boldsymbol{\theta} = \boldsymbol{\theta}^-$
    end
  end

Figure 3 - DQN Algorithm (DRL Class Lectures, 2020)

In addition to the replay buffer technique, as appeared in the 15th line of the algorithm, the *target network*'s weights can be updated by replicating and copying the learned parameters (e.g. weights) of the policy network every fixed amount of steps C. For instance, in this assignment, we were asked to implement this procedure, denoted as a *hard target update* for C=15, hence, the policy network's weights will be copied to the target network every 15 steps (see Eq. 2 below). Furthermore, an implementation of *soft target update* also provided, with C=1 where unlike hard update, in this scenario, as can be seen in Eq. 3 below, in each update the importance ratio between storing the new policy network weights and keeping the existed target network's weights are being tuned using the parameter τ. For a small τ, the target network will move slightly to the value of Q-network and in the case of higher τ, the target network will be much closer to the values of the Q-network, since the policy network values has more weight.

- $Hard\ Target\ Network\ Update$: $\theta' = \theta$ (2)
- $Soft\ Target\ Network\ Update$: $\theta' = \tau\theta + (1-\tau)\theta'$ (3)

3. Evaluation

In this section, after implementing the rest of the DQN skeleton and the ReplayBuffer class (as can be seen in the appendix), I evaluated tha agent's performance. Agent's performance evaluation is demonstrated using six experiments that differ from each other based on their configuration settings relates to the use of replay buffer (with\without) and the target network update method (Hard\\None\Soft). Each experiment conducted five times, hence, to reduce the noise derived due to the CartPole stochastic environment, by averaging the task scores achieved per each one of the five executions set. For each experiment, a performance plot is being presented and evaluated using the measurement of Score over episodes, Loss over episodes, and Mean Q error over episodes. Finally, as can be seen in Table 1 – Design of Experiments and results, a summary of both experiments configurations, notations, and their averaged results are presented.

a. Experiment I – Hard Target Update with Replay Buffer

As can be seen in the plot presented in Fig. 4 below, from the perspective of the score over episodes measurement, it seems that the agent learning curve started after 50 episodes and improved noisily, however, significantly until the 65th episode, and from that moment on the score was pretty stationary around the score of 200 but with a noisy bias to both directions. Simultaneously, both the loss and the Mean Q error over episodes are unstable and overshoot during the initial 50 episodes, and then they start to converge towards zero in an increasing stable trend line. Furthermore, from the perspective of Mean Q error over episodes, note the differences in the height of the peaks along the trend line especially for the initial episodes and their appearance after every 10-15 episodes where the highest of the peak is substantially decreased. The resulted task scores belong to each of the five executions of this experiment were 205, 216, 158, 115, and 220. Hence, the average score was ~183 and further, the average training time for a total of 200 episodes was 2.2 minutes. Also, note this configuration achieved the second-highest average score due to the use of a replay buffer with a capacity of 100,000 experiences tuples, and the use of a hard target update after every 15 steps.

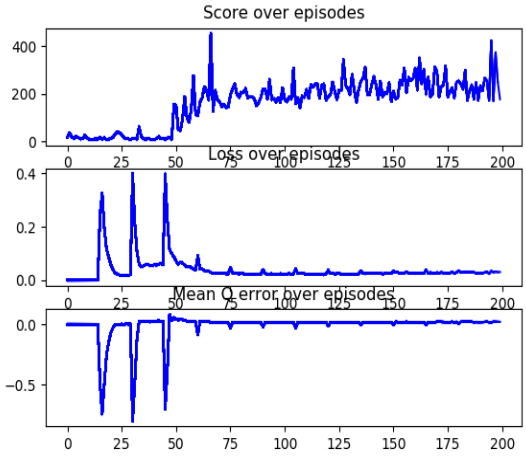

Figure 4 - Experiment I - Performance Metrics

b. Experiment II – Soft Target Update with Replay Buffer

As can be seen in the plot presented in Fig. 5 below, from the perspective of the score over episodes measurement, it seems that the agent learning curve started after 50 episodes and improved noisily, however, significantly until the 110th episode. Then, the interval until the 115th episode is reduced substantially and afterward jumps back until the 125th episode. Last, the score over episodes is exponentially reduced and stays noisy around the score value of 190. Simultaneously, both the loss and the Mean Q error over episodes overshoot around the 15th episode and then reduced towards zero. However, unlike the stable convergence of the hard target update in the 1st experiment, the convergence is less stable and the loss over the episode is slightly higher. Furthermore, unlike the peaks that appeared among the trendline in the 1st experiment, except the highest peak around the 10th episode, despite looks smooth the trendline has more bias that can be caused by the soft update rule and the ratio parameter τ of 0.01. The resulted task scores belong to each of the five executions of this experiment were 205, 304, 156, 162, and 156. Hence, the average score was ~197 and further, the average training time for a total of 200 episodes was 2.4 minutes. Also, note this configuration outperformed the rest and achieved the highest average score due to the use of a replay buffer with a capacity of 100,000 experiences tuples, and the use of a soft target update after every step.

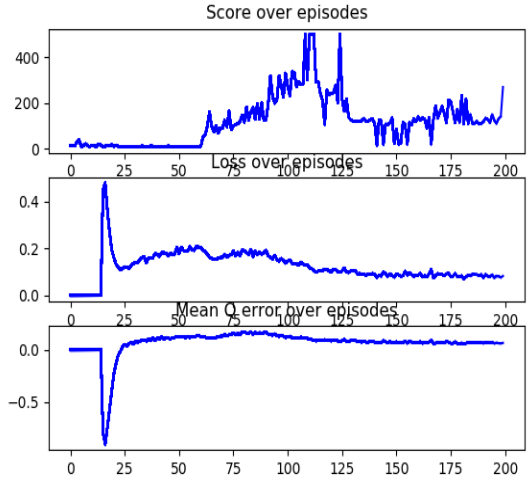

Figure 5 - Experiment II - Performance Metrics

c. Experiment III – Replay Buffer without any Target Update

In this experiment, similar to the last two experiments, a replay buffer with a capacity of 100,000 experiences is used however, unlike the last experiments, the target network isn't updated during the episodes. As can be seen in the plot presented in Fig. 6 below, despite the low loss and mean Q error, the score over episodes is noisy and stationery around the score of 10. Besides the two main peaks located around the initial episodes and the 25th episodes, it looks like no learning curve trend causes the agent to improve its performance, hence, the agent learned a poor policy which resulted in low task scores. Therefore, it's reasonable to assume that since the target network is not updated and not synchronized with the policy network's weights during all of the episodes, the agent's learning and performance are significantly low compared to the results of the above-mentioned experiments. regardless of the use of the replay buffer. The resulted task scores belong to each of the five executions of this experiment were 22, 14, 46, 17, and 15. Hence, the average score was ~23 and further, the average training time for a total of 200 episodes was 48 seconds.

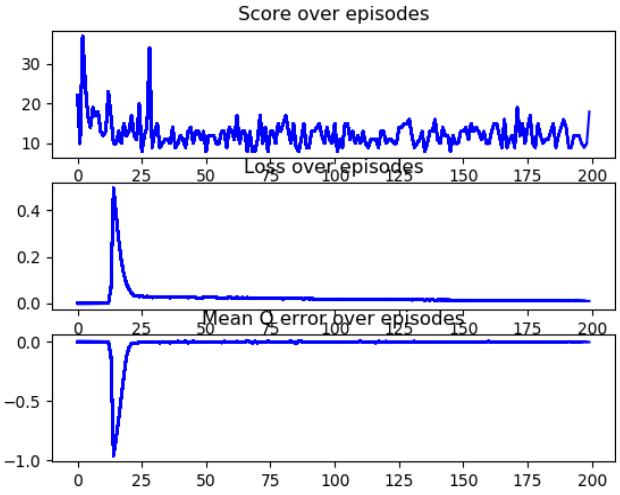

Figure 6 - Experiment III - Performance Metrics

d. Experiment IV – Hard Target Update without Replay Buffer

In this experiment, similar to the 1st experiment, a hard target network update was used, but without replay buffer mechanism, meaning that agent's action was selected based on the last experience in each episode, hence, a batch size of 1, e.g. *Stochastic Gradient Descent* (SGD). Also, the target network was updated every 15 steps using a hard target update. As can be seen in the plot presented in Fig. 7 below, the score over episodes is decreased as long the agent experiences more episodes and stabilize with a noisy trend around the score of 10. Further, note both the loss and the mean Q error are linearly increased from an overall perspective, however, with a relatively similar seasonality every ~15 episodes. Therefore, the resulted task scores belong to each of the five executions of this experiment were 24, 14, 18, 22, and 12. This resulted in the lowest average task score of 18 after an averaged training time of 45 seconds for 200 episodes. It's reasonable to assume that the root cause is due to the lack of replay buffer mechanism which causes the agent to be dependent on the last experience which does not well represent the whole agent's journey. Further, the seasonality phenomenon can be related to the fact of using a hard target update every

15 episodes, hence, increases the chance for the agent to forget its past, since unlike in soft target update in which the target network weights values are taken also into consideration, in the case of hard target update, only the policy network weights are taking into account.

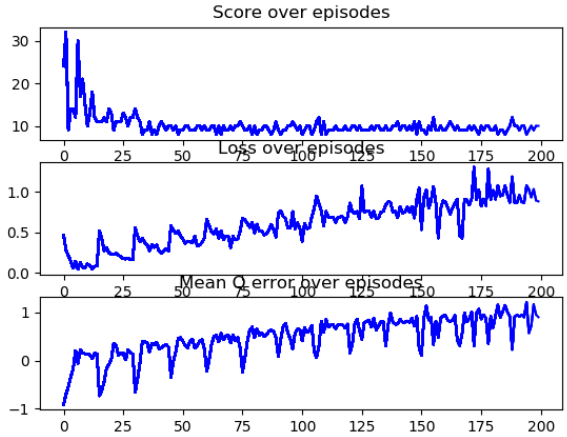

Figure 7 - Experiment IV - Performance Metrics

e. Experiment V – Soft Target Update without Replay Buffer

In this experiment, similar to the 4th experiment, no replay buffer was used, however using a soft target network every step. As can be seen in the plot presented in Fig. 8 below, the results trend is relatively similar to the results of the 4th experiment since the score over episodes is also decreased as long the agent experiences more episodes and stabilize with a noisy trend around the score of 10. Furthermore, both the loss and the mean Q error are linearly increased from an overall perspective, however, noisier and without any seasonality. This noisier loss evidence can be attributed to the soft update rule which updates the target weight every step and therefore, causes a high variance between different intervals during the agent's journey. Moreover, due to the lack of replay buffer and the use of a tuning ratio parameter τ equal to 0.01, more weight is attributed to the target network and therefore, last affected by the agent's newly learned policies from which their experiences encoded in the newly modified policy network weights. The resulted task scores belong to each of the five executions of this experiment were 26, 27, 15, 22, and 24. This resulted in the second-lowest average task score of 23 after an averaged training time of 43 seconds for 200 episodes.

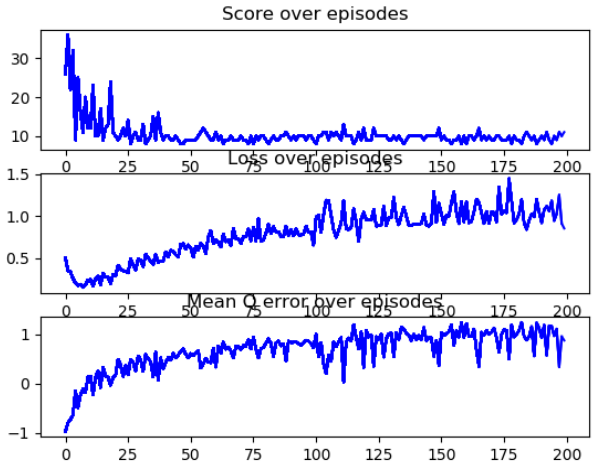

Figure 8 - Experiment V - Performance Metrics

f. Experiment VI – Without both Replay Buffer & Target Update

In this experiment, no replay buffer nor target network update were used. As a result, in each step, the agent is mainly dependent on its last experience since the lack of replay buffer causes the policy network to be trained with a batch size of 1. Yet, unlike the 4th and 5th experiments, the target network is not being updated except at the beginning where the initial weights from the policy network are replicated and copied to the target network. As a result, despite achieving a low score, the agent's performance was slightly better compared to those of the 4th and 5th experiments, probably because the target network was not affected from the policy network inductive bias obtained during the agent journey, due to the lack of replay experience mechanism. As can be seen in the plot presented in Fig. 9 below, the results trend is relatively similar to the results of the 5th experiment since the score over episodes is also decreased as long the agent experiences more episodes and stabilize with a trend around the score of 10, however, it is much smoother. From the aspect of loss and mean Q error it seems that despite the loss is stable it still relatively fixed around 0.1, which represents the inductive bias of the agent that causes him to learn a poor policy. The resulted task scores belong to each of the five executions of this experiment were 14, 59, 18, 24, and 18. This resulted in an averaged task score of 27 after an averaged training time of 43 seconds for 200 episodes.

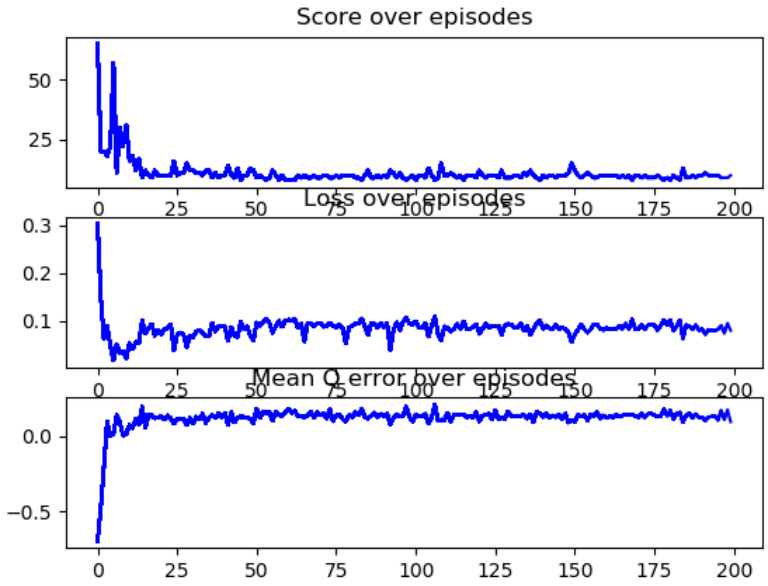

Figure 9 - Experiment VI - Performance Metrics

As can be seen in Table 1 – Design of Experiments and resultsbelow, the results are compared and presented in the DOE table where the best-averaged score of 197 in Exp. II configuration (With Replay & Soft Update) is marked in green and the lowest-averaged score of 18 achieved in Exp. IV is marked in red. Finally, as can be seen in Fig. 10 below, I generated a movie of the best performing agent derived using Exp. II configuration and resulted in a final task score of 309 and a total reward of 200 uniformly distributed between actions, e.g. 100 left actions and 100 right actions. Moreover, after watching the video it seems that the agent balancing the pole by clicking left and right repeatedly to stay in the middle of the surface and avoids sharp movement that can increase the chance to fail in the task.

Table 1 – Design of Experiments and results

Target Update	With Replay With Target	With Replay Without Target	Without Replay With Target	Without Replay Without Target
Hard	**Exp. I** Average Score: 183 Average Time: 2.2 min.	X	**Exp. IV** Average Score: 18 Average Time: 45 sec.	X
None	X	**Exp. III** Average Score: 23 Average Time: 48 sec.	X	**Exp. VI** Average Score: 27 Average Time: 42 sec.
Soft	**Exp. II** Average Score: 197 Average Time: 2.4 min.	X	**Exp. V** Average Score: 23 Average Time: 43 sec.	X

```
Final task score =  309.0
load best model ...
make movie ...
Total reward: 200.00
Action counts: Counter({0: 100, 1: 100})
```

Figure 10 - Final Model Results

4. Appendix

a. DQN Class

i. Build the target network and set its weights to policy net's weights

```
# Build neural networks - Policy Network & Target Network
policy_net = Network(network_params, device).to(device)
target_net = Network(network_params, device).to(device)
# Setting Target Network's initial weights same as Policy Network's weights
target_net.load_state_dict(policy_net.state_dict())
```

ii. Action selection function implementation

```
def select_action(s):
    '''
    This function gets a state and returns an action.
    The function uses an epsilon-greedy policy.
    :param s: the current state of the environment
    :return: a tensor of size [1,1] (use 'return torch.tensor([[action]], device=device, dtype=torch.long)')
    '''
    global epsilon
    # In case the generated random number is less than epsilon => Select Random Action
    if np.random.uniform(0, 1) < epsilon:

        return torch.tensor([np.random.choice(network_params["action_dim"], 1)], device=device, dtype=torch.long)
    # In case the generated random number is higher than epsilon => Select Action Greedily
    else:
        policy_prediction = policy_net(s)

        return torch.tensor([[torch.argmax(policy_prediction)]], device=device, dtype=torch.long)
```

iii. Q_curr and expected Q calculations

```
# Compute curr_Q = Q(s, a) - the model computes Q(s), then we select the columns of the taken actions.
# Pros tips: First pass all s_batch through the network
#            and then choose the relevant action for each state using the method 'gather'
curr_Q = policy_net(state_batch).gather(1, action_batch).reshape(training_params["batch_size"])

# Compute expected_Q (target value) for all states.
# Don't forget that for terminal states we don't add the value of the next state.
# Pros tips: Calculate the values for all next states ( Q_(s', max_a(Q_(s')) )
#            and then mask next state's value with 0, where not_done is False (i.e., done).
next_state_values = target_net(next_states_batch).max(1)[0].detach()
expected_Q = reward_batch + (training_params["gamma"] * next_state_values)
expected_Q[~ not_done_batch] = 0
```

iv. Soft Target Update Implementation

```
# soft target update
if params.target_update == 'soft':
    # Copies soft updated weights from Policy Network to Target Network every each episode
    for target_param, policy_param in zip(target_net.parameters(), policy_net.parameters()):
        target_param.data.copy_(
            training_params["tau"] * policy_param.data + (1.0 - training_params["tau"]) * target_param.data)
```

v. Hard Target Update Implementation

```
# hard target update. Copying all weights and biases in DQN
if params.target_update == 'hard':
    # Copies the weights from Policy Network to Target Network every 15 episodes (+1 is for indices adjustment)
    if (i_episode + 1) % training_params['target_update_period'] == 0:
        target_net.load_state_dict(policy_net.state_dict())
```

vi. Storing Weights

```
# update task score
if min(all_scores[-5:]) > task_score:
    task_score = min(all_scores[-5:])

    # Store weights
    torch.save(policy_net.state_dict(), "best.dat")
```

b. Replay Buffer Class – Push & Sample functions

```
def push(self, *args):
    """Saves a transition."""

    # Check if current position lower than replay buffer's size, hence, to avoid cases of out of bound index
    if len(self.memory) < self.capacity:
        self.memory.append(Transition(*args))
        self.position += 1

    # In case of out of bound index, initialize position for overriding past buffers and storing new buffers
    else:
        self.position = self.position % self.capacity
        self.memory[self.position] = Transition(*args)

def sample(self, batch_size):

    # Sample buffers from replay buffer. Buffer's amount based on the batch size
    return random.sample(self.memory, batch_size)
```

5. Reference

[1] R. S. Sutton and A. G. Barto, “Reinforcement Learning: An Introduction Second edition, in progress,” 2018.

[2] V. Mnih, D. Silver, and M. Riedmiller, “Deep Q Network (Google),” pp. 1–9.

[3] S. Singh, T. Jaakkola, and M. L. Littman, “Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms,” 2000.