Reinforcement Learning Assignment

Igor Kolasa, *University of Sheffield, Department of Computer Science*

I. Introduction

REINFORCEMENT LEARNING (RL) has emerged as a powerful paradigm for training intelligent agents to autonomously learn optimal decision-making strategies through interactions with their environment. RL leverages the principles of trial and error, where an agent learns from feedback in the form of rewards or punishments. These rewards guide the agent's exploration in the environment, shaping its behavior over time.

The fundamental ideas behind RL were established years ago, with significant contributions from the fields of control theory, dynamic programming, and artificial intelligence. Concepts such as Markov Decision Processes [1], the Bellman equations [2], and Temporal-Difference Learning [3] laid the groundwork for RL algorithms to compute optimal policies and value functions.

However, it is in recent years that we have witnessed an exponential growth in the field of RL. This surge can be attributed to the integration of classical RL techniques with deep learning. By combining deep learning with RL, researchers have been able to tackle complex and high-dimensional problems that were previously deemed intractable. For example, algorithms such as Deep Mind's AlphaZero [4] are able to achieve superhuman performance in incredibly complex games like Go and chess.

This report aims to provide a brief overview of two classical RL algorithms, namely Q-Learning [5] and SARSA [6]. Next, the performance evaluation of the Double Deep Q-Network (DDQN) [7] will be conducted in the specially designed chess endgame task. Furthermore, the standard Deep Q-Network (DQN) [8], will be applied to the same problem, and a comparative analysis will be presented between the results obtained from the DDQN method and the DQN approach. The experimental code can be accessed at https://github.com/KOLA16/COM3240-Reinforcement-Learning-Assignment.

II. Q-Learning and SARSA: Background and Algorithms

A. Markov Decision Process

The Markov Decision Process (MDP) is a fundamental framework used in reinforcement learning to model sequential decision-making problems. It comprises a set of states, actions, transition probabilities, and rewards. At each time step, an agent, operating within the environment, takes an action based on its current state. The environment then transitions to a new state, and the agent receives a reward accordingly. The transition from one state to another is governed by probabilistic rules, known as the transition probabilities. MDPs assume the Markov property, which states that the future state depends only on the current state and action, independent of the past. This property enables the formulation of optimal policies that maximize cumulative rewards.

B. Bellman Optimality Equation

RL literature [9] distinguishes two types of value functions. The *state-value* function $V_\pi(s)$ is the expected cumulative reward an agent can achieve starting from a state s and following a policy π thereafter. The *action-value* $Q_\pi(s,a)$ (also referred to as a Q-value) is the expected cumulative reward starting from state s, taking the action a, and then following the policy π. The Bellman optimality equation is a key concept in dynamic programming and reinforcement learning, providing a recursive formulation for the optimal state-value and action-value functions. The equation relates the values of states or state-action pairs to the values of their successor states, taking into account the transition probabilities and rewards. For the state-value, the equation can be written as:

$$V_*(s) = \max_a \mathbb{E}[R(s,a) + \gamma V_*(s')]$$

where $V_*(s)$ is the expected value of state s under optimal policy $*$, $R(s,a)$ is the immediate reward obtained in the sate s when taking an action a, γ is a discount factor between 0 and 1 that determines the importance of future rewards compared to immediate rewards, and $V_*(s')$ is the value of the next state s'. Similarly, the Bellman optimality equation for the Q-value can be expressed as:

$$Q_*(s,a) = \mathbb{E}[R(s,a) + \gamma \max_{a'} Q_*(s',a')]$$

where $Q_*(s,a)$ represents the optimal expected Q-value of an action a taken from current state s, and $\max_{a'} Q_*(s',a')$ denotes the maximum Q-value over all possible actions a' in the next state s'.

C. Temporal-Difference Learning

Temporal-Difference (TD) learning is a reinforcement learning method that combines ideas from dynamic programming and Monte Carlo (MC) [10] methods. While the MC methods provide a way to update value functions based on the actual values of successor states, TD learning introduces the concept of bootstrapping, updating the value function based on an estimate of the future value. The difference between the current estimate and an estimate obtained by experiencing a transition to the next state is known as the TD error. TD learning provides an effective way to learn from incomplete sequences of experiences without requiring a complete knowledge of the

environment dynamics. TD methods, such as SARSA and Q-Learning, utilize the TD error to update the estimated values and learn optimal policies during ongoing interactions with the environment.

D. Classical SARSA and Q-Learning

SARSA is an *on-policy* TD control algorithm that stands for State-Action-Reward-State-Action. The key idea behind SARSA is to estimate the action-value function $Q(s, a)$ and learn an optimal policy directly from the observed experiences. The algorithm conventionally follows an epsilon-greedy policy, which means that it selects the action with the highest Q-value with a probability of $(1-\epsilon)$ and selects a random action with a probability of ϵ to encourage exploration of the state space. SARSA uses the following update function to iteratively update the Q-values:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[R(s, a) + \gamma Q(s', a') - Q(s, a)\right]$$

where $Q(s, a)$ is the Q-value of action a taken in state s, α is the learning rate that determines the step size of the update, $R(s, a)$ is the immediate reward received after taking action a in state s, γ is the discount factor, s' is the next state, and a' is the action taken in the next state s'. The update equation computes the TD error, which is the difference between the observed value of $Q(s, a)$ and the estimated value based on the observed reward and the next state's action-value $Q(s', a')$.

Q-Learning is an *off-policy* TD control algorithm that aims to learn the optimal action-value function $Q(s, a)$ without explicitly following a policy during learning. The key idea behind Q-Learning is to estimate the optimal action-value function by iteratively updating the Q-values based on the maximum Q-value over all possible actions in the next state. The update function is similar to the one followed by SARSA:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[R + \gamma \max_{a'} Q(s', a') - Q(s, a)\right]$$

However, the main difference between SARSA and Q-Learning lies in the way they update the Q-values. SARSA updates the Q-value based on the action actually taken in the next state, while Q-Learning updates the Q-value based on the maximum Q-value over all possible actions in the next state. This difference leads to distinct characteristics in their learning behaviors. SARSA learns the Q-values and policy simultaneously by following the current policy during training. This makes SARSA more cautious and well-suited for environments where the exploration must be controlled, such as in safety-critical applications. SARSA tends to converge to a policy that performs well under the current exploration strategy. On the other hand, Q-Learning learns the optimal Q-values regardless of the policy followed during training. Q-Learning is known to be more explorative and can converge to an optimal policy even if the exploration is not carefully controlled. Q-Learning can potentially discover a more optimal policy than SARSA if provided with enough exploration.

E. Deep Reinforcement Learning

Deep reinforcement learning combines the principles of deep neural networks with reinforcement learning algorithms, such as Q-learning, to tackle the challenges posed by complex and high-dimensional state spaces. By leveraging deep neural networks as function approximators, deep Q-learning eliminates the need for explicit Q-value tabulation and instead utilizes a deep neural network to estimate Q-values for state-action pairs.

One prominent and widely utilized deep reinforcement learning algorithm is the Deep Q-Network (DQN). Training a DQN involves employing an *experience replay buffer*, which stores observed transitions (experiences) consisting of the state s, action a, immediate reward $R(s, a)$, and subsequent state s'. Actions can be selected using a standard epsilon-greedy strategy. During training, mini-batches of experiences are randomly sampled from the replay buffer, and the obtained states s are fed into a neural network. This network outputs estimated Q-values for each possible state-action pair, based on the input state. The calculated loss function is then given by:

$$L = Q_*(s, a) - Q(s, a)$$

Here, $Q(s, a)$ represents the Q-value associated with the taken action, while $Q_*(s, a)$ denotes the optimal Q-value. As discussed earlier, the optimal Q-value can be estimated using the Bellman optimality equation:

$$Q_*(s, a) = \mathbb{E}[R(s, a) + \gamma \max_{a'} Q_*(s', a')]$$

The term $\max_{a'} Q_*(s', a')$ is obtained by passing the subsequent state s' to the network and selecting the maximum Q-value from the output. After computing the loss, standard optimization techniques such as Stochastic Gradient Descent (SGD) or Adam are employed to update the network's weights, minimizing the loss. Consequently, the neural network gradually refines its Q-value estimations, approaching the optimal Q-value, and subsequently identifying the most valuable actions for each encountered state. In order to stabilize the learning process, the conventional DQN algorithm uses two separate neural networks. The first network is usually referred to as the *main* neural network and it is used to calculate the current state and action Q-values, while the other network, called *target* network, is employed to estimate the subsequent state and action Q-values.

Despite application of the above technique, the classical DQN algorithm tends to dramatically overestimate the Q-values if the training is not stopped after a proper number of steps. To address this issue, the Double DQN has been proposed that decouples the action selection from the action evaluation. In the standard DQN, while computing the Bellman optimality equation, the target network is used for both the optimal action selection and Q-value evaluation. DDQN modifies this process, such that the main network is used to select the optimal action a', while the target network is used only to evaluate Q-value of the selected action. This trick leads to more accurate Q-value

estimates and can result in improved learning performance and stability.

III. Project Objectives

The task addressed in this project was to train a deep neural network capable of achieving checkmate in a simplified chess game scenario. We were given a 4x4 chessboard with three pieces: 1x King and 1x Queen controlled by the trained network, and the opponent's King performing random moves to a safe position. Each game could continue until a checkmate or a draw (where the opponent's King is not threatened, but there are no squares for the King to move to). Additionally, the chess environment provided specifically for this assignment was set to end a game after 20 moves had been made. All moves performed must comply with the rules of chess.

The first objective of the conducted experiments was to evaluate the performance of the Double Deep Q-Network (DDQN) in the presented task and also test how changing values of selected DDQN hyperparameters affect its results. The first hyperparameter was the previously introduced discount factor, denoted as γ, which controls the importance of future rewards in the Q-value estimation. In other words, the closer γ is to 0, the less the agent cares about rewards in the distant future and only learns actions that yield maximum immediate reward. The second hyperparameter is the speed, denoted as β, of the decaying trend of ϵ in the epsilon-greedy policy. The common practice in training DQN policies is to decay the exploration probability as the training progresses and encourage exploitation of the learned optimal Q-values.

The second part of the project aimed to apply the vanilla DQN to the same chess task setting and investigate if the discussed overestimation bias significantly affects the learning performance.

IV. Double Deep Q-Network Experiments

The network architecture used for the experiments was a fully connected neural network with a single hidden layer of size 200, and it employed a standard ReLU activation function. The network weights were updated using the Adam optimizer with a learning rate of 0.001. The default discount factor γ was set to 0.95. The initial epsilon value ϵ was set to 0.1, and its default decay speed β was set to 10. The target network's parameters were updated after every 320 training steps. The training ran for 10 epochs, with each epoch consisting of 5000 steps. The training collector collected 20 steps in each iteration, and the main network was updated once every 20 steps using a batch of 30 samples from the training collector's reply buffer. Every 200 steps, the trained policy's performance was evaluated on 100 episodes (games) emulated with a separate test collector. Code snippets including the setup of hyperparameters and training loop can be found in **Appendix A**. Details of the libraries used are described in **Appendix B**.

In total, three runs of the full training pipeline were executed. In the first run, the default values of γ and β were used. In the second run, the γ parameter was reduced to 0.05, while β remained unchanged. In the third run, β was increased to 10000, while γ retained its default value. The settings of the second run aimed to emulate a scenario where the agent focused mostly on immediate rewards, which may not be the most effective strategy in the game of chess, where evaluating the value of a position should consider the potential future advantages that a move can bring. On the other hand, the third scenario aimed to minimize the exploration probability and force the agent to exploit strategies that were initially found to be most beneficial, which could potentially lead to slower discovery of the globally optimal strategies. **Figures 1** and **2** depict the results observed in this set of experiments:

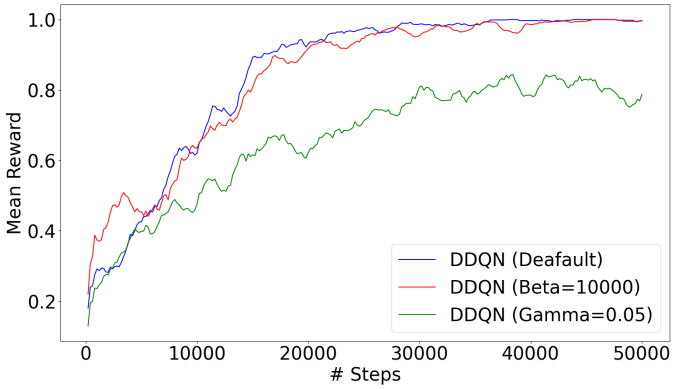

Fig. 1. Average reward over training steps obtained with DDQN algorithm in three different hyperparameters settings: I. ($\beta = 10$, $\gamma = 0.95$), II. ($\beta = 10000$, $\gamma = 0.95$), III. ($\beta = 10$, $\gamma = 0.05$)

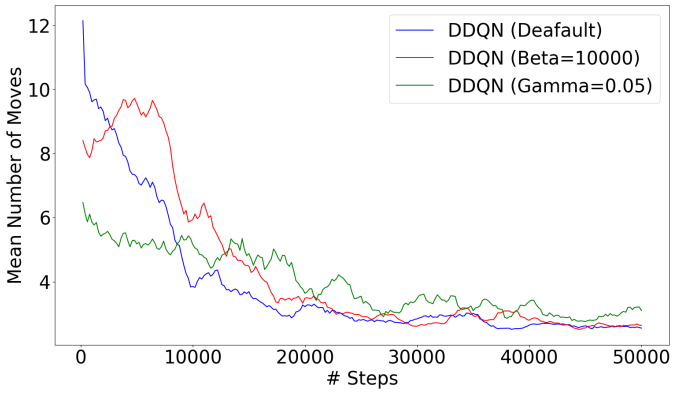

Fig. 2. Average number of moves per game over training steps obtained with DDQN algorithm in three different hyperparameters settings: I. ($\beta = 10$, $\gamma = 0.95$), II. ($\beta = 10000$, $\gamma = 0.95$), III. ($\beta = 10$, $\gamma = 0.05$)

As shown in **Figure 1**, the DDQN with default parameters converges to the global optimum after approximately 30000 training steps, and the trained network becomes capable of winning almost every game thereafter. This is in contrast to the performance of the DDQN with a reduced γ factor. The performance of this network seems to plateau around an average reward value of 0.8, indicating that focusing on immediate rewards reduces the probability of winning by

approximately 20%. It is interesting to observe the behavior of the policy with the reduced exploration probability. Its average reward spikes rapidly in the early stages of training as the policy exploits the best-known strategies at that time. However, following this approach leads to a suboptimal approach that guarantees winning in only 50% of the games. After escaping the local optimum, the policy eventually learns the optimal strategy, but it converges slower than the default DDQN.

Changes in the average number of moves per game, as depicted in **Figure 2**, reflect the trends observed in the reward dynamics. The default DDQN starts with the largest number of moves, which can be attributed to a stronger tendency for exploration. It is also possible that it considers strategies that require more moves but have greater future value.

V. Vanilla Deep Q-Network Experiments

The same exact default hyperparameters were used in training the DQN policy as for the DDQN. Performance observed over 50000 training steps is presented in **Figures 3** and **4**:

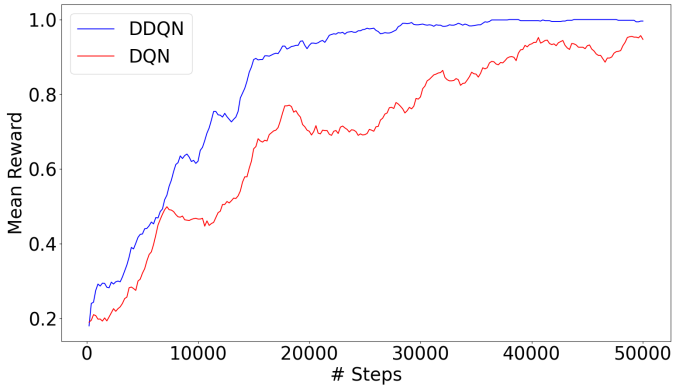

Fig. 3. Average reward over training steps obtained with DDQN and DQN algorithm in the default hyperparameters setting: ($\beta = 10$, $\gamma = 0.95$)

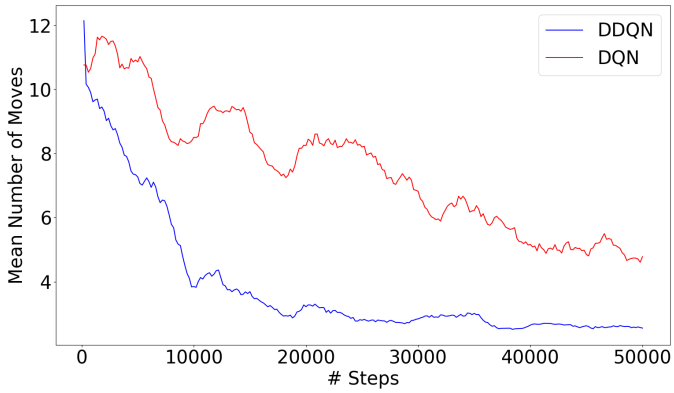

Fig. 4. Average number of moves per game over training steps obtained with DDQN and DQN algorithm in the default hyperparameters setting: ($\beta = 10$, $\gamma = 0.95$)

The results obtained clearly demonstrate that learning with the vanilla DQN is less effective compared to its DDQN modification. This is likely due to the overestimation bias discussed earlier, which leads to an unstable training process and slower convergence. Despite the slower progress, a positive trend can be observed in the average reward, and the average number of moves decreases as training advances. This suggests that the DQN has the potential to eventually learn the optimal strategy.

VI. Conclusions

Modern reinforcement learning is characterized by the remarkable diversity of methods that draw and combine ideas from various research fields. The fusion of classical RL algorithms with deep neural networks has enabled the training of autonomous systems capable of excelling in complex tasks, such as playing the game of chess. The conducted experiments have revealed that, although incredibly powerful, deep RL algorithms like the Double Deep Q-Network are also sensitive to the selection of hyperparameters. Specifically, it has been demonstrated that extreme changes in the discount factor for future rewards or the probability of exploration can impede the learning process or even make it impossible to learn the optimal policies. Additionally, the experiments have shown that DDQN guarantees more efficient learning compared to the vanilla Deep Q-Network, which is susceptible to the overestimation bias.

Appendix A
Code Snippets

```
## INITIALISE THE TRAINING AND TESTING ENVIRONMENTS
# I had to initialise two separate environments as using the same
# environment for training and testing resulted in error, which led
# the agent to choose illegal actions
size_board = 4
train_env = Chess_Env_Gym(size_board)
test_env = Chess_Env_Gym(size_board)
state_shape = train_env.state_shape
action_shape = train_env.action_shape

# set up the Q-net, optimizer and policy
########### fill in your code ###########
net = Net(
    state_shape,
    action_shape,
    hidden_sizes=[200],
    activation=torch.nn.ReLU
)
optim = torch.optim.Adam(net.parameters(), 1e-3)

# Double DQN
policy = DQNPolicy(
    net,
    optim,
    discount_factor=0.95,  # GAMMA
    target_update_freq=320,
    is_double=True
)

# set up the training and testing replay buffers and collectors
buffer_train = ReplayBuffer(size=20000)
train_collector = Collector(policy, train_env, buffer_train)
buffer_test = ReplayBuffer(size=20000)
test_collector = Collector(policy, test_env, buffer_test)

# customized training loop for Double DQN
epoch = 10
eps_train = 0.1   # epsilon value of the epsilon-greedy policy in the training phase
beta = 10          # BETA - controls speed of the epsilon decaying trend
step_per_epoch = 5000
step_per_collect = 20
update_per_step = 1 / 20  # update policy once after every 20 steps
batch_size = 30
```

Fig. 5. Setting of the Double Deep Q-Network parameters. Experiments for the vanilla Deep Q-Network used the exact same setting except the is_double attribute in the DQNPolicy definition which was set to False.

```python
steps = []
test_rews = []
test_n_moves = []
eps_train_0 = eps_train

for i in range(epoch):
    policy.set_eps(eps_train)
    eps_train = eps_train_0 / (1 + beta * i)  # decaying epsilon
    step_acc = 0
    while step_acc < step_per_epoch:
        # collect `step_per_collect` steps of traning transitions
        train_collect_result = train_collector.collect(n_step=step_per_collect)

        # accumulate number of steps `step_acc`
        step_acc += train_collect_result['n/st']

        for j in range(round(update_per_step * train_collect_result['n/st'])):
            # train policy with a sampled batch data from buffer
            loss = policy.update(batch_size, train_collector.buffer)
            loss = loss['loss']


        # After 200 steps, test the policy performance on 100 games
        if step_acc % 200 == 0:
            test_collect_result = test_collector.collect(n_episode=100)

            # mean reward per game
            test_rew = test_collect_result['rew']

            # mean number of moves per game (max is set to 20)
            test_len = test_collect_result['len']

            step = step_acc + step_per_epoch*i
            steps.append(step)
            test_rews.append(test_rew)
            test_n_moves.append(test_len)

            print(f'Step: {step}, train loss: {loss}, mean test reward per game
```

Fig. 6. Customised training loop for the Double Deep Q-Network policy. The same training loop was used for the vanilla Deep Q-Network learning.

APPENDIX B
LIBRARIES

Experiments were conducted in a Python 3.9.13 environment using a deep reinforcement learning library Tianshou in version 0.4.10. The library is based on the pure PyTorch which was installed in version 1.13.0 with CUDA 11.6. The chess environment was created with the OpenAI Gym 0.26.2 toolkit.

Seed value of 42 was set in the Python random variable generator, NumPy random generator and PyTorch manual seed to allow reproducibility of experiments.

REFERENCES

[1] Puterman, M. L. (1990). Markov decision processes. *Handbooks in operations research and management science*, 2:331–434. Elsevier.

[2] Bellman, R. (1966). Dynamic programming. *Science*, 153(3731):34–37. American Association for the Advancement of Science.

[3] Tesauro, G. et al. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68.

[4] Silver, D. et al. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419):1140–1144. American Association for the Advancement of Science.

[5] Watkins, C. J. C. H. (1989). Learning from delayed rewards. King's College, Cambridge United Kingdom.

[6] Rummery, G. A. and Niranjan, M. (1994). On-line Q-learning using connectionist systems. *University of Cambridge, Department of Engineering Cambridge, UK*, 37.

[7] Van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30, number 1.

[8] Mnih, V. et al. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[9] Sutton, R. S. and Barto, A. G. (2018). Reinforcement learning: An introduction. MIT Press.

[10] Hammersley, J. (2013). Monte carlo methods. Springer Science & Business Media.