Deep Reinforcement Learning for Fighting Games

Timoleon Latinopoulos[1] and Anastasios Tefas[1]

[1]Department of Computer Science, A.U.Th.

***Abstract*—In recent years there has been significant progress in the use of deep reinforcement learning for solving complex tasks, from robotics and self-driving cars to personalised recommendations and computer games. Some of the most notable advancements in the area of video game agents include programs like AlphaGo and OpenAI5, which benefit from the use of different neural network architectures. In this thesis, we study and experiment with one of those architectures in an attempt to address some of the problems of teaching a neural network to play a fighting game.**

I. Introduction

THE use of raw visual data for controlling artificial agents has been a crucial task in the field of deep reinforcement learning. Many of the most remarkable advances in this research area, which include subjects such as self-driving cars and robotics typically require interaction with the real world. One of the many challenges of this approach is correctly handling noisy and complex data. For that reason, virtual environments are traditionally used to simulate an accurate training scenario.

Those simulated environments usually incorporate video games. Board games such as Chess, Backgammon and Go have been used as a benchmark for various deep reinforcement learning algorithms [7]. Recently, studies were made regarding retro games from the Atari 2600 [3] to the SNES game consoles as well as modern games like Starcraft 2 and DOTA 2. Most of those studies were met with great success, with agents able to achieve superhuman performance and surpass even pro players.

In this thesis, we attempt to use deep reinforcement learning in order to teach an agent to beat opponents in the original Mortal Kombat from the SNES directly from the game screen's pixels. The method that was used in all the experiments was Double Dueling Deep Q Learning and the game was emulated in the retro environment from OpenAI's gym library using Python [11].

II. Background

A. Markov Decision Processes

The method which allows us to formalise a reinforcement learning problem is called Markov Decision Process (MDP). In this process, we consider an agent who interacts with an environment $\mathcal{E}$ by getting feedback from a distinct set of observations $\mathcal{S}$ obtained from this environment. Then the agent decides what action to take from a finite set $\mathcal{A}$ and receives a corresponding reward $\mathcal{R}_a(s, s')$, where s is the current state of the environment and s' is the new state which occurs after taking action a. Given a state s we can refer to $P_a(s, s')$ as the probability of state s' happening if action a is chosen. Thus, we can conclude that state s' is dependent on the preceding state s and action a.

Regarding a video game environment, the observation refers to an image, which consists of the raw pixel values taken from the game's screen [10], [12]. The actions correspond to all the legal button presses that the game console will translate into an in-game action. As for the reward appointed for a particular set of observations and actions, it is typically defined by a reward function specific to the game scenario.

The main problem of Markov Decision Processes is finding a policy π for the agent which specifies what action to pick when in a specific state s. This policy's goal is to maximise the cumulative reward usually derived from the discounted sum of all the rewards over time:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \textstyle\sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

where γ is the discount factor satisfying $0 \leq \gamma \leq 1$. This factor determines how important an action in the future is to the agent in contrast to any immediate ones.

B. Deep Q-Learning

The goal of reinforcement learning is finding an optimal action-value function $Q^*(s, a) = \max_\pi Q^\pi(s, a)$ which gives us the best action for every possible state. This function obeys a fundamental property known as the Bellman equation:

$$Q^*(s, a) = E\left[R_{t+1} + \gamma \max_{a'} Q^*(s', a')\right],$$

which intuitively suggests that if the optimal value $Q^*(s', a')$ at state s' was known for all possible actions a', then the optimal strategy is picking the action which maximises the expected reward of the state s'. Attempting to learn this value function in an environment such as a video game can be extremely complex because of the high dimensionality of the state/action spaces. Thus, non-linear function approximators are typically used, such as neural networks to extract this value function through iterative updating. Those neural network architectures are traditionally called Q-networks and the method which takes advantage of them Deep Q-Learning [18].

This method is model-free meaning that the task is solved directly from the observations of the environment. These observations are fed into the network in random batches from a replay memory rather than utilizing the most recent experiences. The advantage of this approach called experience

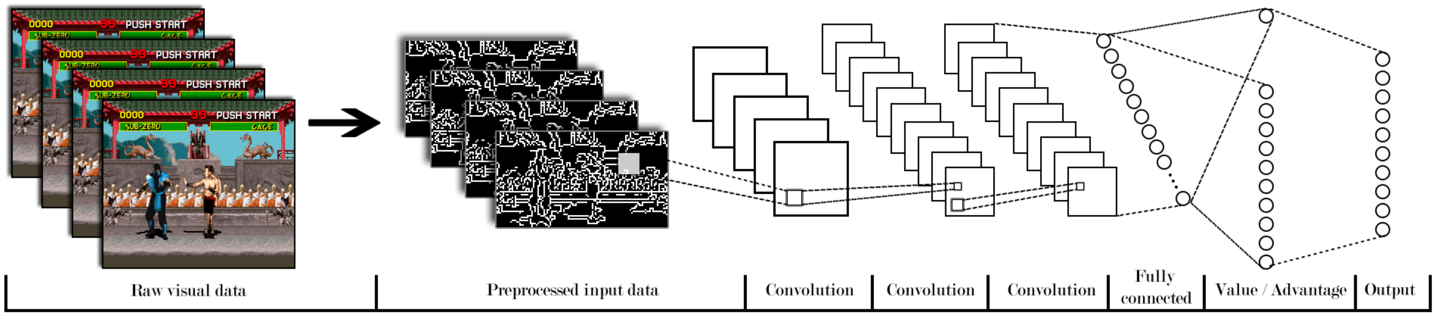


Fig. 1: An example of the neural network architecture used for game-playing with pixel input. The raw visual data is preprocessed and fed into the network which consists of some convolutional and fully connected layers. The output of the last layer is split into two different streams which estimate the state value and action advantage and then they are combined to calculate the final Q-values.

replay is to avoid correlations between recent observations [14]. Another strategy in Deep Q-Learning is ϵ-greedy which refers to a random selection of actions during observations. The probability of random selections is determined by the variable ϵ satisfying $0 \leq \epsilon \leq 1$. Otherwise, the agent selects the optimal action with probability $1-\epsilon$. This allows the agent to try different approaches in hopes of achieving the optimal solution to its task. Usually, training starts with an ϵ close to 1 and is gradually lowered to a value close to 0 where the agent is rarely acting randomly.

In this thesis, the method used is called Double Dueling Deep Q-Learning and it has some important differences from the original Deep Q-Learning. The dueling part of the neural network used is referring to the split of the state values and action advantages from the original state/action values into two different streams [9]. The formula which is used to compute the resulting Q-values is:

$$Q(s,a) = V(s) + (A(s,a) - \frac{1}{|A|}\sum_{a'} A(s,a'))$$

where $A(s,a)$ is the advantage of taking an action a in a state s, $V(s)$ is the value of that state and the rest is the average of all the action advantages of the current state. The average action advantage is used to reduce the advantage of the action chosen at that state to 0 and avoid the problem of identifiability of the Q-value given the two separate streams. The overall benefit of this strategy is to generalise learning across different actions by keeping a separate value for each state. This way we can recognise how valuable the current state is without having to learn the effect of each action at that state.

Another improvement over the original Deep Q-Learning is the use of two separate neural networks, which is called Double DQN [2], [5] and is used to decouple the action selection from the target Q-value generation. This can solve the problem of overoptimistic value estimates which can arise from using the same values for both selecting and evaluating an action, therefore leading to faster training and more stable learning.

III. Proposed Methods

A. Similar Works

There has been a handful of articles trying different methods for training agents on fighting video games. In the article ”Deep Learning for Video Game Playing” [13], there was an attempt of training a DQN agent on a fighting game called FightingICE which is used as a benchmark for different AI implementations. In addition, in the article ”An Exploration of Neural Networks Playing Video Games” [19], there were experiments on different games, such as Super Mario Bros, Sonic the Hedgehog and Mortal Kombat 3 using methods such as NEAT and PPO2. Finally, in the article ”How We Built an AI to Play Street Fighter II - Can you beat it?” [15], there was presented a DQN agent playing Street Fighter 2.

B. Neural Network Architecture and Action Space

The network architecture is similar to the one used by DeepMind regarding artificial agents achieving human level control on Atari 2600 games [6]. It was utilised for all the experiments and consists of three convolutional layers and one fully-connected linear layer. In detail:

- Input Layer: $63 * 113 * 4$
- First Convolutional Layer: 32 8x8 filters, stride 4, relu
- Second Convolutional Layer: 64 4x4 filters, stride 3, relu
- Third Convolutional Layer: 64 3x3 filters, stride 2, relu
- Fully Connected Layer: 512 rectifier units
- Output Layer: 1 state value and 12 action advantages

The output of the last fully connected layer is split into two separate streams to represent the state value and the action advantages. An example of the network used can be seen in Figure 1. The loss function that was selected was Log-Cosh [17] and the optimiser was Adam [4] with a learning rate of $5 * 10^{-5}$.

Regarding the agent's actions, we tried to reduce the action space to just 12 different button combinations [16], [19]. These actions consist of no button presses, up, down, left, right, L and the following combinations:

- {Down, A},
- {Down, B},
- {Right, A},
- {Right, B},
- {Left, A},
- {Left, B}

Moreover, the e-greedy exploration was set to start from 100 percent and gradually decrease to 10 percent random actions in the span of 1.000.000 frames. For the rest of the training this value stayed the same.

C. Input Preprocessing

In order to train the agent, we used a stack of four consecutive preprocessed frames of the game's screen as an input to the convolutional neural network. However, handling the SNES game screen can be computationally heavy as well as noisy. The preprocessing applied involves converting the individual pixels from RGB to black and white, cropping part of the screen that does not contain significant information, reducing the resolution in half both in width and height and finally applying an edge detection filter on those frames [1], [19]. Edge detection proved to be a reliable way of discarding all the unneeded noise from the input and helped achieve significantly quicker results. After edge detection, the data was normalised by dividing the pixel values with 255 so that each one was either 1 or 0. The final result consists of an input size of $63 * 113 * 4$ in contrast to the original size of $3 * 224 * 256 * 4$.

In addition, we considered flipping the game screen whenever the agent's player moved to the opposing side of the frame so that it would always stay on left. However, flipping the game screen meant that we had to manually flip the agent's left and right action buttons to adjust his movement. This whole process reduced the observation space because of the mirrored frames and achieved a significant boost to the speed of the agent's training. Finally, the target network was updated by the weights of the main Q-network every 10.000 frames.

D. Reward Function and Replay Memory

One major component of training an artificial agent is picking a suitable reward function. We ultimately decided to go with the simpler but most effective one which was the difference between each player's health points (HP) per frame observed. This way we managed to avoid sparse rewards and give more emphasis on actions such as arrow keys that would otherwise not offer an immediate reward to the agent. The discount factor γ was set to 0.999. Frame skipping of 10 frames was also implemented to shorten the length of each episode and emulate a human player's reflexes [8], [19].

In order to store the observations, actions and rewards of each time step, a replay memory was utilised. In addition, it was set to fit 600.000 observations and was used every 4 frames to extract a random batch of 32 experiences in order to train the network.

IV. Experiments

A. Dueling DDQN VS CPU

The first experiment's scenario was training the artificial agent against several CPU opponents set on hard difficulty. The agent's selected player was Sub-Zero and the three different CPU opponents were Johnny Cage, Lyu Kang and Scorpion. The agent trained on each opponent on a separate session, training took 30.000 episodes of playing through one round fights and the evaluation of the agent occurred every 100 episodes on a traditional best-of-three fight. The results can be seen in Figure 2.

In all three of the test cases, the agent learned how to play against the CPU opponent and consistently win its matches. In the late stages of training, mostly on the Johnny Cage and Scorpion fights, the agent managed to exploit specific behaviours of the opponent and while using a very limited set of moves, win convincingly. What is notable about Mortal Kombat is that the CPU opponent behaved exactly the same way on every replay of a particular evaluation step, meaning that the environment proved to be deterministic.

B. Dueling DDQN VS Random Move Opponent

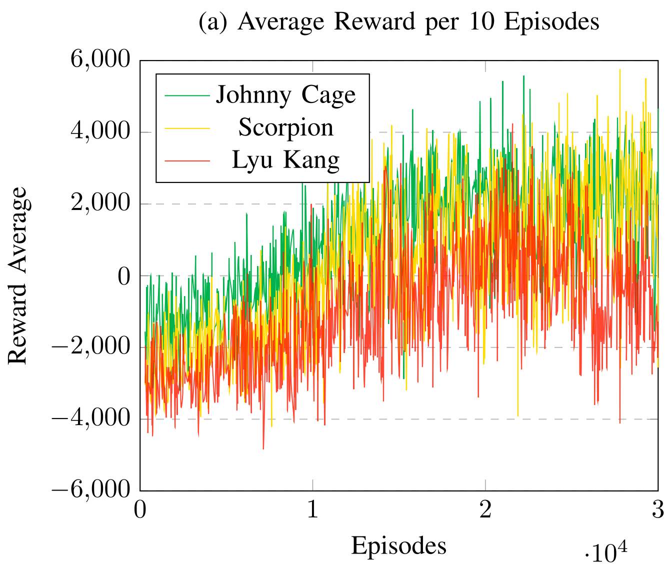
In this example, we tried to experiment with a non-deterministic environment by introducing an opponent which acted on random button presses. This proved to be extremely chaotic and difficult for the network to handle since the opponent would repeatedly press all of the damaging buttons, leaving no space for close combat. To compensate for this, the opponent was set to act once for every 6 actions of our agent and the opponent was set to play as Sub-Zero, the same player as our agent. Training was done similarly to the previous experiment with 30.000 episodes of training and progress evaluation on every 100 episodes. The results can be seen in Figure 3.

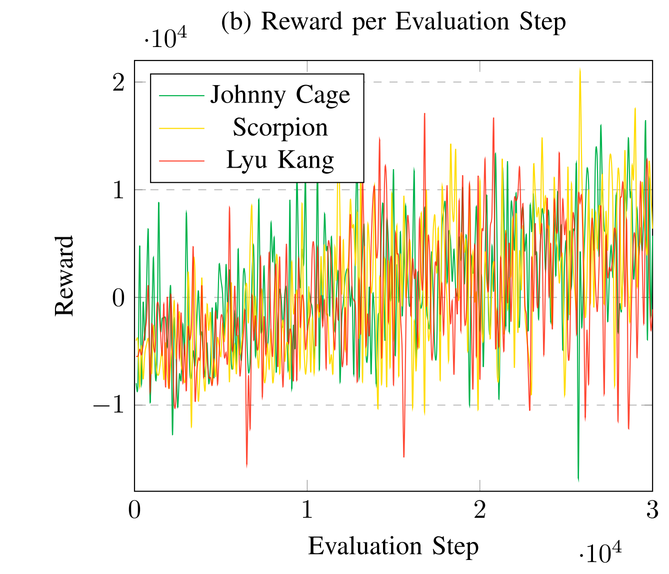
This approach had a reward peak near the end of the training with our agent finding the best way to approach, damaging its opponent and then holding back and waiting for the time to run out.

C. Dueling DDQN VS Dueling DDQN

As a last experiment, we trained two different artificial agents fighting against each other. Both agents used a separate neural network model of the same general architecture and the selected player for both was Sub-Zero. Training took place for 10.000 episodes with the same strategy as the previous experiments with 30.000 episodes of training and progress evaluation on every 100 episodes. The results can be seen in Figure 4.

In the early stages of training, both players stayed mainly on their starting position and most of the score accumulated by them was due to someone randomly damaging the other and then finishing the match because of timeout. This meant that the agents gained high rewards as the matches went on for longer. Over time, they started approaching each other more frequently and the matches were usually much shorter.





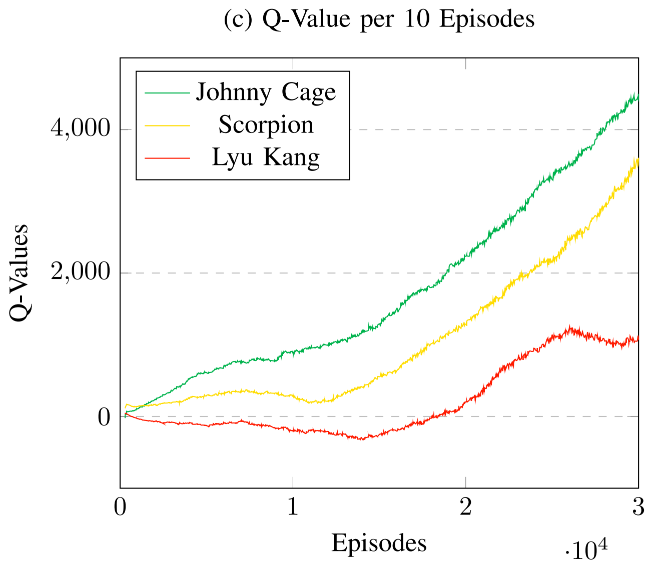
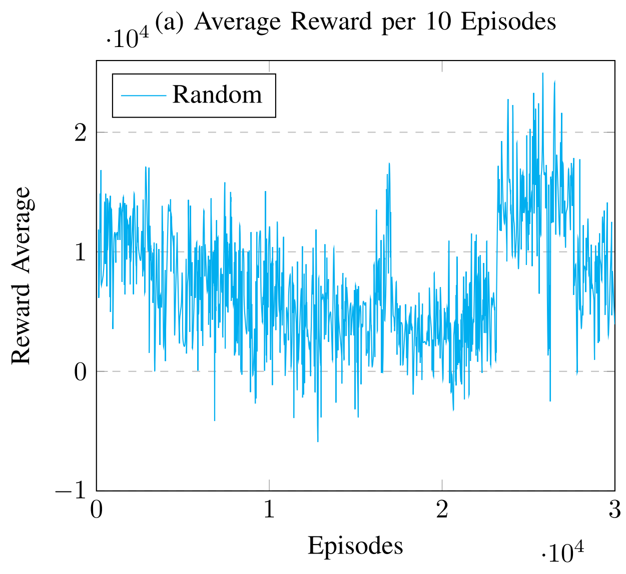
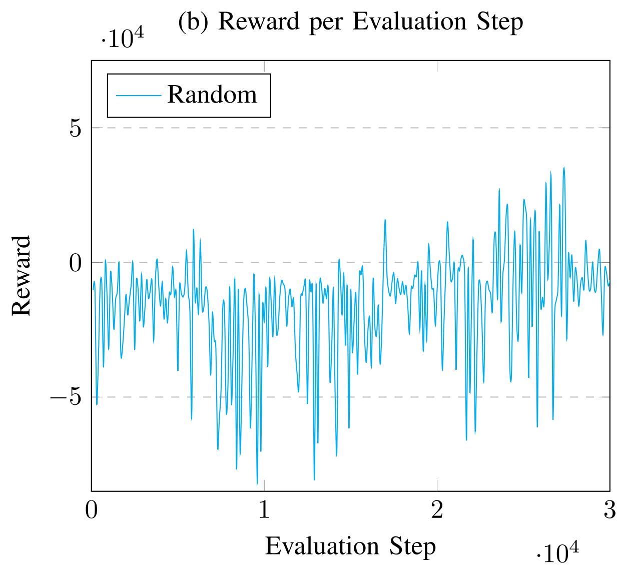


Fig. 2: Dueling DDQN VS CPU.





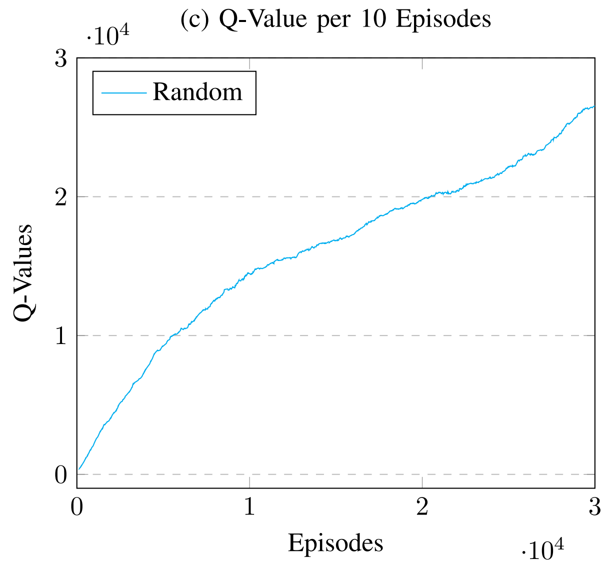


Fig. 3: Dueling DDQN VS Random Move Opponent.

When the training was done, both agents managed to learn to approach their opponent cautiously while also trying to land some hits to increase their overall reward.

V. Conclusion

Dueling Double Deep Q-Learning proved to be an exceptional technique when used in dealing with different scenarios in fighting game environments. In most of our tests the agents showed how they can handle diverse and difficult tasks while achieving significant performance improvements during training. This shows that the artificial agents presented are more than capable of challenging and winning matches against human opponents on those types of environments.

References

[1] J. Canny, *A Computational Approach to Edge Detection,* in *IEEE Transactions on Pattern Analysis and Machine Intelligence*, ISSN: 1939-3539, pp-679-698, 1986.

[2] Hado van Hasselt, *Double Q-learning,* in *Advances in Neural Information Processing Systems 23*, J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel and A. Culotta. Curran Associates, Inc. pp-2613-2621, 2010.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra and Martin Riedmiller, *Playing Atari with Deep Reinforcement Learning*. arXiv:1312.5602v1 [cs.LG], 2013.

[4] Diederik P. Kingma and Jimmy Ba, *Adam: A Method for Stochastic Optimization*. arXiv:1412.6980 [cs.LG], 2014.

[5] Hado van Hasselt , Arthur Guez, and David Silver, *Deep Reinforcement Learning with Double Q-Learning*. arXiv:1509.06461 [cs.LG], 2015.

[6] Mnih, V., Kavukcuoglu, K., Silver, D. et al, *Human-level control through deep reinforcement learning*. Nature 518, 529–533 (2015).

[7] Silver, D., Huang, A., Maddison, C. et al, *Mastering the game of Go with deep neural networks and tree search.* Nature 529, 484–489 (2016).

[8] Jakub Sygnowski and Henryk Michalewskiy, *Learning from the memory of Atari 2600.* arXiv:1605.01335v1 [cs.LG], 2016.

[9] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot and Nando de Freitas, *Dueling Network Architectures for Deep Reinforcement Learning*. arXiv:1511.06581v3 [cs.LG], 2016.

[10] K. Kunanusont, S. M. Lucas, and D. Perez-Liebana, *General video game ai: Learning from screen capture*. 2017 IEEE Congress on, pages 2078–2085, 2017.

[11] Nadav Bhonker, Shai Rozenberg and Itay Hubara, *Playing SNES in the Retro Learning Environment*. arXiv:1611.02205v2 [cs.LG], 2017.

[12] Ivan Pereira1 and Gabriel Sousa1, *Machine Learning Directly from Pixels*. ISSN: 2594-486X, 2017

[13] Seonghun Yoon and Kyung-Joong Kim, *Deep Q Networks for Visual Fighting Game AI*. CIG 2017.

[14] Ruishan Liu and James Zou, *The Effects of Memory Replay in Reinforcement Learning*. arXiv:1710.06574 [cs.AI], 2017

[15] Adam Fletcher, *"How We Built an AI to Play Street Fighter II - Can you beat it?"*, Oct 25, 2017. Available: https://medium.com/gyroscopesoftware/how-we-built-an-ai-to-play-street-fighter-ii-can-you-beat-it-9542ba43f02b

[16] Tom Zahavy, Matan Haroush, Nadav Merlis, Daniel J. Mankowitz and Shie Mannor, *Learn What Not to Learn: Action Elimination with Deep Reinforcement Learning*. arXiv:1809.02121 [cs.LG], 2018.

[17] Pengfei Chen, Guangyong Chen and Shengyu Zhang, *Log Hyperbolic Cosine Loss Improves Variational Auto-Encoder*. ICLR 2019 Conference, 2019.

[18] Niels Justesen, Philip Bontrager, Julian Togelius and Sebastian Risi, *Deep Learning for Video Game Playing*. arXiv:1708.07902v3 [cs.AI], 2019.

[19] Joshua J Luo, *"An Exploration of Neural Networks Playing Video Games"*, May 22, 2019. Available: https://towardsdatascience.com/an-exploration-of-neural-networks-playing-video-games-3910dcee8e4a

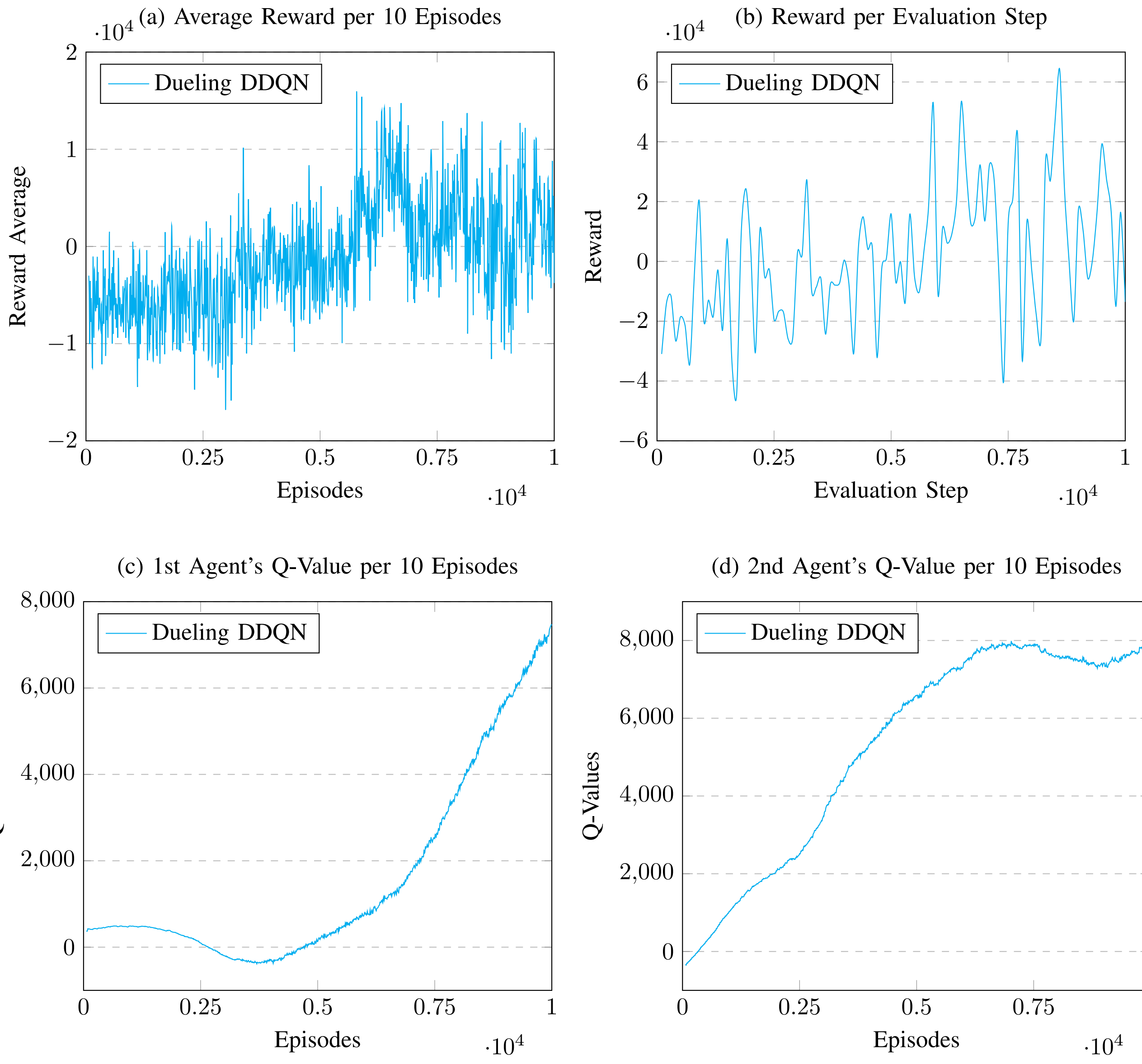


Fig. 4: Dueling DDQN VS Dueling DDQN. (a) and (b) show the rewards for the first agent which represent the reversed rewards of the second agent.