# Project for the course of "Computational Intelligence - Deep Reinforcement Learning" of CSD AUTh

Project on Deep Reinforcement Learning

**Θωμάς Αχιλλέας Μπίλλας**
(ΑΕΜ: **10366 - ΗΜΜΥ ΑΠΘ**)
**Thomas Achilleas Billas**
(ΑΕΜ: **10366 - ECE AUTh**)
**tmpillas@ece.auth.gr**

Department of Electrical and Computer Engineering
May 2025

# Project on Deep Reinforcement Learning

Thomas Achilleas Billas
(AEM: 10366 - ECE AUTh)

May 2025

## Contents

# 1 Introduction

The goal of this project is the development of an agent that uses Deep Reinforcement Learning (RL) to learn to play Prisoner's Dillema in an Iterated manner. More specifically the agent will participate in Iterated Prisoner's Dilemma Tournaments called Axelrod Tournaments. We use Reinforcement Learning, in particular the algorithm of Deep Q-Learning to train an agent against a broad pool of strategies that correspond to a multitude of approaches to the game of Iterated Prisoner's Dillema. We try different architectures for the Neural Network in Deep Q-Learning and we represent experimental results for conclusions for each case. We conclude that with RL we can produce agents that perform really well and even dominate in Axelrod Tournaments.

# 2 Necessary knowledge to understand the problem and solution

We now present the mathematical and theoretical background that is necessary to have in order to understand the task at hand and be able to interpret the results:

## 2.1 Prisoner's Dillema

The Prisoner's Dilemma (PD) is a two player game used to model a variety of strategic interactions. Each player chooses between cooperation (C) or defection (D). The payoffs of the game are defined by the matrix

$$\begin{pmatrix} R & S \\ T & P \end{pmatrix}$$

where $T > R > P > S$ and $2R > T + S$. The PD is a one round game, but is commonly studied in a manner where the prior outcomes matter. After each round, both players receive their reward based on their selected action. If both players choose C they receive reward R, if both choose D they both receive reward P and if one chooses D and the other C, the player that opted to defect receives the maximum reward T while the player that opted to cooperate receives the minimum reward S. The repeated version of PD is called the Iterated Prisoner's Dilemma (IPD). The IPD is frequently used to understand the evolution of cooperative behavior from complex dynamics. [8]

## 2.2 Axelrod Tournaments

In 1979, Robert Axelrod (a political scientist) organized a tournament between strategies for the Prisoner's Dilemma (PD). He invited several game theory researchers to submit strategies (in the form of computer programs), which were used in an IPD. Specifically, each strategy played against every other strategy and was evaluated based on the total number of points it accumulated in order to assess which behaviors were most successful over time.

The first Axelrod Tournament invited researchers to submit strategies in the form of computer programs. Each strategy played a repeated game of Prisoner's Dilemma against every other strategy (including itself), and the total scores were recorded. A total of 14 strategies were submitted by researchers from the fields of Mathematics, Psychology, Political Science, and Sociology. Each strategy/program had access to the history of all the games that had been played and used this history to decide whether to cooperate or defect in the current game. There was also a random strategy (i.e., one that randomly selected 1 or 2 in each round with equal probability). Each game consisted of 200 rounds. The entire tournament was repeated 5 times, and the performance of each strategy was summed across repetitions. Remarkably, the simplest strategy, Tit For Tat-which begins with cooperation and then mimics the opponent's last move—emerged as the winner.

A second tournament followed, in which participants had access to the results of the first and could adapt or improve their strategies. Despite this opportunity, Tit For Tat still performed exceptionally well, reinforcing the idea that cooperation can evolve and be robust even in competitive environments. These tournaments demonstrated that cooperation could thrive under specific conditions, such as repeated interactions, recognition, and reciprocity, and have since influenced fields ranging from evolutionary biology to economics and political science. He presented his results in [2] and in [1] respectively.

## 2.3 Reinforcement Learning & Deep Reinforcement Learning

Reinforcement learning (RL) is an interdisciplinary area of machine learning and optimal control concerned with how an intelligent agent should take actions in a dynamic environment in order to maximize a reward signal. Reinforcement learning differs from supervised learning in not needing labeled input-output pairs to be presented, and in not needing sub-optimal actions to be explicitly corrected. Instead, the focus is on finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge) with the goal of maximizing the cumulative reward (the feedback of which might be incomplete or delayed). [9] In the context of the IPD, RL can be used to evolve strategies that learn to cooperate, defect, or punish, depending on what yields the best long-term payoff.

Deep Reinforcement Learning (Deep RL) extends classical reinforcement learning by combining it with deep neural networks, allowing agents to learn directly from high-dimensional inputs such as images, raw observations, or large feature spaces. This enables the agent to approximate complex value functions, policies, or models of the environment that would be otherwise intractable with traditional RL techniques. Deep RL has been successfully applied to a wide range of problems including robotics, game playing, and strategic decision-making in multi-agent systems.

In the context of the IPD, Deep RL allows for more expressive and adaptive strategies, especially when learning from large amounts of simulated interaction data or when competing against a diverse population of opponents.

Some key components-terms of RL, that are known to us from the course of Deep RL and Computational Intelligence are:

- **Agent**: The learner/decision-maker.
- **Environment**: The world the agent interacts with.
- **State**: A representation of the current situation.
- **Action**: A move the agent can make.
- **Reward**: A signal telling the agent how good its action was.
- **Policy**: A strategy that maps states to actions.
- **Value Function**: Expected future rewards from a state.

While the RL loop can be summed up to be:

1. **State** – the agent observes the current situation.
2. **Action** – it chooses an action.
3. **Reward** – it receives a numerical reward from the environment.
4. **Next state** – the environment changes.
5. **Update** – the agent updates its knowledge or policy.

We present the theoretical background of the Deep RL algorithms used to train our agent and present the results in the next sections of the project:

#### 2.3.1 Deep Q-Learning

Let's first explain what the Q-Learning algorithm does in simple terms. It is a classic tabular RL algorithm that learns $Q(s, a)$ for every state-action pair using the Bellman equation.

Let $Q(s, a)$ be the function that calculates the discounted future reward given that the agent is in state $s$ and chooses an action $a$ and continues optimally from this point on.

$$Q(s_t, a_t) = \max_{\pi} R_{t+1}$$

where $\pi$ the possible policies to follow in the future. This function is called the Q-function and represents the quality of every choice. It expresses the best possible outcome after the choice of action $a$.
Given the Q-function we choice the policy $\pi(s)$ of the agent to be:

$$\pi(s) = \arg\max_{a} Q(s, a)$$

And the Q-function can also be computed as:

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

where $\langle s, a, r, s' \rangle$ a known transition of the agent from one state $s$ to another state $s'$ and $\gamma$ the discount factor. The main idea of Q-learning is to iteratively approximate the Q-function using the Bellman equation (the function above).

In order to deal with problems where the state space is so big that the Q-table (a lookup table with values $Q(s, a)$ for each state-action pair) cannot be calculated, we use a Neural Network (NN) to approximate the Q-table (Deep Q-Learning).

$$Q(s, a) \approx \mathrm{NN}(s, a; \theta)$$

The NN is trained to approximate the target Q-value using a loss function, e.g.:

$$\text{loss} = \sum (Q - Q_{\text{target}})^2$$

The table look-up can be replaced by the feed-forward process and storing the new value using backpropagation. To avoid local minima, we use an $\epsilon$-greedy policy, meaning that with probability $\epsilon$ we execute a random action (exploration) instead of the one suggested by the network (exploitation). [7]

# 3 Implementation

## 3.1 Axelrod Tournaments in Python

In this project we use the Axelrod Python library [4,5], open source software for conducting IPD tournaments with reproducibility and ease to use and check the performance of our agent as main motives. The library is continuously developed and, as of version 4.13.1, contains over 250 strategies, many of which are derived from the scientific literature, including classic strategies like Win Stay Lose Shift and previous tournament winners such as OmegaTFT and Adaptive Pavlov. The growing strategy set also includes a variety of reinforcement learning-based strategies, zero-determinant strategies, and human-submitted designs. We will use some of them to train and test the performance of our agent, in tournaments where it will face a broad spectrum of strategies. More specifically it will face basic strategies, like Always Cooperate or Always Defect, reactive strategies like TitForTat or Win Stay Lose Shift and stochastic strategies like Gambler or Joss. A analytical description of all strategies available in the Axelrod library can be found on the library document page [3].

## 3.2 Environment

The environment used simulates the IPD and is implemented as a custom class, `IPDEnv`, built on top of the Axelrod Python library. Each episode represents a match between a learning agent and a fixed opponent strategy provided by Axelrod.

The environment is initialized with three main parameters: the opponent strategy, a history length for state representation, and a maximum number of turns per episode. At each time step, the agent chooses between cooperation (C) and defection (D), represented as 0 and 1 respectively. The opponent responds according to its own strategy logic, and both actions are recorded.

The agent's state is a fixed-length binary vector encoding the last $h$ actions taken by both the agent and the opponent (where $h$ is the history length). If fewer than $h$ rounds have occurred, the state is filled with zeros.

A standard Prisoner's Dilemma reward matrix is used, which is the one that Axelrod library uses by default:

$$\begin{bmatrix} (3,3) & (0,5) \\ (5,0) & (1,1) \end{bmatrix}$$

The agent receives its respective reward based on the joint action of both players. The episode ends after a predefined number of turns that we choose to be 100.

## 3.3 Training Data

A very big challenge is figuring out what the training data will and should be. The challenge here is to train the agent versus a multitude of strategies that covers the full behavioral spectrum (exploitable/naive, punishing/reactive, stochastic/noisy, deceptive/meta) with just enough examples to avoid memorizing the data. Moreover regarding the training opponent pool, we will use the following strategies that we will arbitrarily categorize in the following groups, in order to create a balanced mix of opponents:

- **Exploitable & Naive:** Strategies that are too forgiving, static, do not punish and can be exploited easily by defecting. We use `CyclerCCCCCD`, `EasyGo`, `Alternator`.
- **Punishing/Retaliatory & Reactive:** Strategies that react based on the opponent moves, punish defection and teach restraint. We use `Grudger`, `ForgivingTitForTat`, `HardTitForTat`.
- **Stochastic & Noisy:** Strategies that include randomness and probabilistic behavior. We use `Gamlber`, `SoftJoss`, `StochasticWSLS`.
- **Deceptive & Pattern-Based:** Strategies that require adaptation, bluff, or long memory to exploit. We use `WinStayLoseShift`, `TitFor2Tats`, `FoolMeOnce`.

Obviously, this categorization is done somewhat heuristically and it can be argued that some strategies fit in more than one category or that there needs to be a larger variant of strategies in order to be more accurate. However, we conclude that by classifying the strategies in this manner we train an agent that manages to deal well with all different kinds of strategies included in the Axelrod library.

## 3.4 Evaluation Data

The same challenge presents regarding the data that we will use to assess how well our agent performs. We have to test the agent's generalization ability and not its memory so of course we do not test on the same strategies. However, in order to correctly evaluate the agent we place it in a tournament that we use strategies that are different versions of the training, that are similar in style but not seen before, in order to again create a balanced mix of opponents. Thus, we use:

- **Exploitable & Naive:** We use `Cooperator`, `Defector`, `CyclerCCD`.
- **Punishing/Retaliatory & Reactive:** We use `TitForTat`, `SneakyTitForTat`, `SuspiciousTitForTat`.
- **Stochastic & Noisy:** We use `FirstByJoss`, `StochasticCooperator`, `MathConstantHunter`.
- **Deceptive & Pattern-Based:** We use `BackStabber`, `Calculator`, `Handshake`.

## 3.5 Deep Q-Learning Agent

We implement a Deep Q-Learning (DQN) agent based on the algorithm introduced by DeepMind in their 2015 paper, *Human-level control through deep reinforcement learning* [6]. As mentioned in subsection 2.3.1, in Deep Q-Learning a NN is used to approximate the values of the Q-table that includes values of the Q-function, which estimates the expected cumulative reward for taking action $a$ in state $s$ and following the learned policy thereafter. We will analyze the choice for the architecture of the DQN network in the following subsection 3.5.1.

The network is trained using the mean squared error between predicted Q-values and target Q-values computed via the Bellman equation. To stabilize training, we use a target network, which is a copy of the policy network and is periodically updated with the policy network's weights. We also use a replay buffer to store transitions $(s, a, r, s', \text{done})$ and sample mini-batches uniformly.

The agent follows an $\epsilon$-greedy policy, selecting a random action with probability $\epsilon$ and the greedy action (the one with the highest Q-value) otherwise. We will discuss more regarding the way we deal with exploration-exploitation dilemma in subsection 3.5.2.

During training the agent interacts with the diverse pool of opponents defined in subsection3.3, updating its Q-values via mini-batches sampled from the replay buffer and using the Adam optimizer.

### 3.5.1 Architecture of the Neural Network

We will explore some options regarding the architecture of the DQN network.

1. **The simple case:** In the first simple case, we use a fully connected Feedforward NN (MLP). It consists of two fully connected layers: an input layer of dimension equal to the state size, a hidden layer with ReLU activation, and an output layer with two units representing the Q-values for the two possible actions (cooperate or defect). This architecture is commonly used in DQN when we have non-image, low-dimensional input spaces, like the ones in our case. More specifically we use:
    - Input layer with 10 dimensions (5 for each of the players last moves).
    - Hidden layer with 64 dimensions with a ReLU activation function.
    - Output layer with 2 dimensions, one for each action C or D.
2. **Simple case with more inputs:** We again follow the same architecture as in case 1. with the only change being that instead of using just 5 of each of the players last moves we increase the input dimension to 20 in order to consider the last 10 moves of each player.
3. **Deeper MLP:** We also explore the option of creating a deeper MLP with 2 hidden layers and an input dimension space that resembles the one in case 2. More specifically we use:
    - Input layer with 20 dimensions (10 for each of the players last moves).
    - Hidden layer 1 with 128 dimensions with a ReLU activation function.

- Hidden layer 2 with 64 dimensions with a ReLU activation function.
- Output layer with 2 dimensions, one for each action C or D.

4. **A CNN:** Even though we get better results with a simpler architecture, we will also analyze the case of using a Convolutional NN (CNN) to estimate the Q-Table. As discussed elaborately during the semester, a CNN is a type of feedforward neural network that learns features via filter (or kernel) optimization. While CNNs are typically used for spatial data (like images), we can apply them to 1D sequential patterns in the IPD state representation, in order to enhance the agent's ability to learn temporal patterns from the opponent and agent histories. More specifically:
   - We will reshape the input vector of size 20 (10 for each players last moves) into a $2 \times 10$ 1D image, so the convolution operates across time, independently to the player.
   - Convulotional 1d layer 1 with 16 filters, kernel size 3 and padding 1 with a ReLU Activation function.
   - Convulotional 1d layer 2 with 32 filters, kernel size 3 and padding 1 with a ReLU Activation function.
   - Flatten layer into a 320-dimensional vector ($32 \times 10$).
   - Fully connected hidden layer with 64 dimensions followed by a ReLU activation function.
   - Fully connected output layer with 2-dimensional output, one for each action C or D.

We present some results for each of the 4 cases mentioned.

### 3.5.2 Exploration-Exploitation dilemma

As mentioned before, the agent follows an $\epsilon$-greedy policy, selecting a random action with probability $\epsilon$ and the greedy action (the one with the highest Q-value) otherwise. $\epsilon$ is decayed gradually during training to ensure exploration in the early stages and exploitation in the later stages. How drastically we will opt to reduce the value of $\epsilon$ during our training is the main issue here, known as exploration-exploitation dilemma. We of course start with $\epsilon = 1$ and we have set $\epsilon\_min = 0.05$. In order to determine the rate that $\epsilon$ will decay we will use the following line of reasoning:

- Determine the number of the episodes that we want our training to last.
- Given the initial $\epsilon$ value and $\epsilon\_min$ that we want to achieve in the last episodes of the training, let's determine what percentage of the total episodes we want to achieve $\epsilon = \epsilon\_min$.
- After we decide the percentage we use the following formula:

$$r_\epsilon = \left( \frac{\epsilon_{\min}}{\epsilon} \right)^{\frac{1}{N \cdot M}}$$

  where $r_\epsilon$ is the decay rate of $\epsilon$, $N$ is the desired number of episodes and $M = 1 - P$ where P is the percentage we decided. For example if we want our $\epsilon$ to be minimum during the last 10% of the training we will use $M = 0.9$.

After some experimentation we have concluded that we achieve the best results when $M \in (0.8, 0.9)$. That choice gives our agent enough time to learn and also lock in its best behavior. We will also present some results that we twist $\epsilon$ manually while still remaining in that range to get some better results. For most experiments we choose $\epsilon$ to be minimum for the last 15% of training episodes.

### 3.5.3 Batch size choice

For all of our experiments we will use a batch size of 64, meaning that during training we will sample 64 experiences `(state, action, reward, next_state, done)` form the replay buffer to update the network. We use this common and safe choice because it has empirically shown to work well in DQN and is also small enough to update frequently and fit comfortably in memory while also being large to enough to give a good average.

# 4 Experiments & Results

We present experiments and results for all the aforementioned cases. In order to reproduce the results presented the following code is provided:

- For section 4.1 use `DQN_case1.ipynb`.
- For section 4.2 use `DQN_case2.ipynb`.
- For section 4.3 use `DQN_case3.ipynb`.
- For section 4.4 use `DQN_case4.ipynb`.

In general, we choose $\epsilon$ to be minimum for the last 15% of training episodes. Due to the fact that some strategies are stochastic, it is obvious that if we run the tournament multiple times we will possibly see different rankings since differences in points are small. However we can get a feeling of the general performance of our agent by also taking in consideration the difference between rankings.

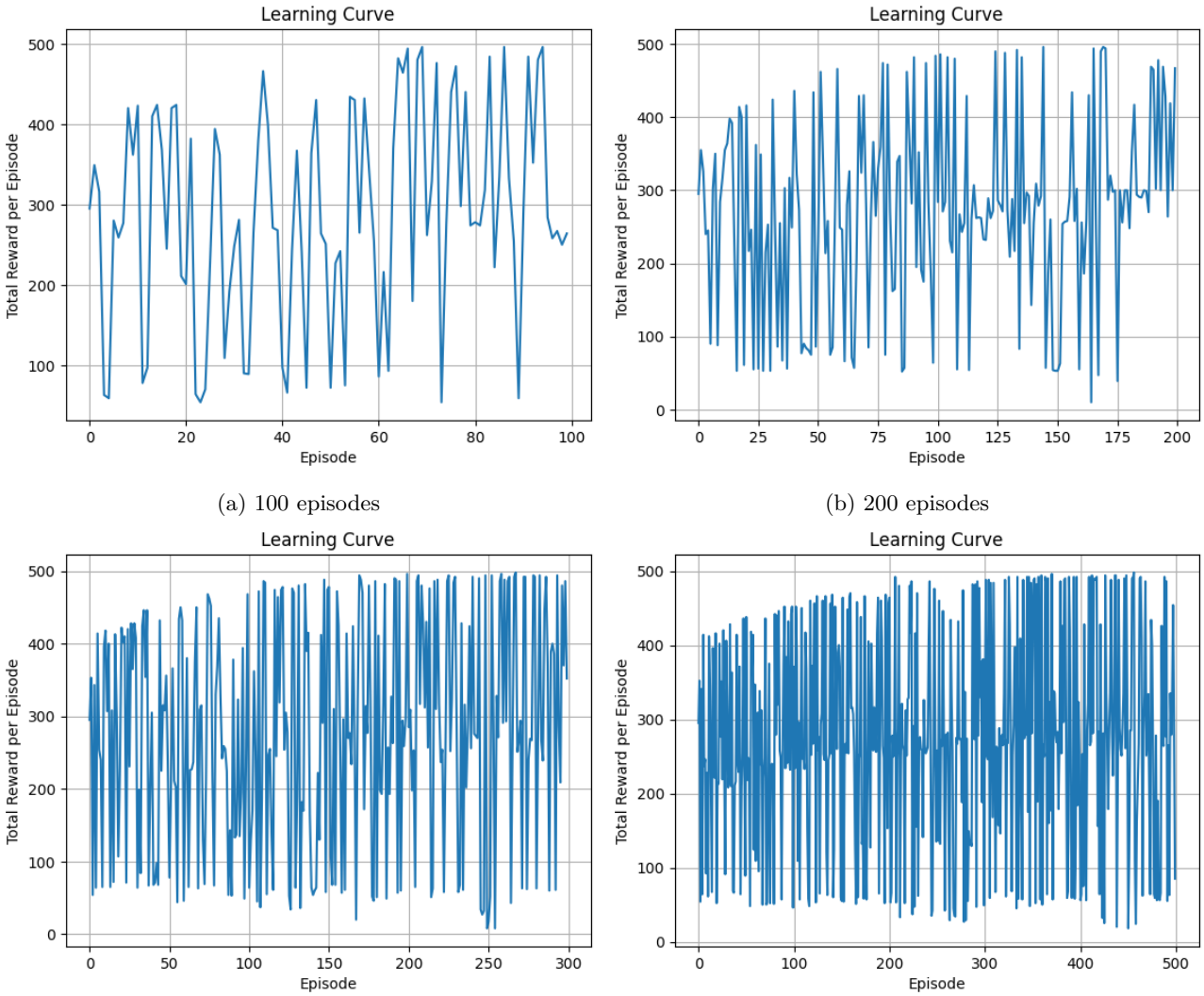
## 4.1 DQN agent using MLP with 1 hidden layer and 10-dimensional input

We present the results for the agent that is described as *1. The simple case* in subsection 3.5.1.

| # of episodes | $r_\epsilon$ | Ranking (out of 13) | Agent Score | Score of Winner |
|---|---|---|---|---|
| 100 | 0.965369925 | 2nd | 2916.00 | 2926.00 |
| 200 | 0.982532404 | 1st | 2963.30 | 2963.30 |
| 300 | 0.988320768 | 10th | 2577.70 | 2858.4 |
| 400 | 0.991227726 | 11th | 2382.30 | 3071.00 |
| 500 | 0.992976003 | 6th | 2631.70 | 2840.20 |
| 600 | 0.994143233 | 11th | 2334.20 | 3068.30 |
| 800 | 0.995604201 | 1st | 2914.60 | 2914.60 |
| 1000 | 0.996481812 | 1st | 3031.60 | 3031.60 |
| 1500 | 0.997653164 | 1st | 2933.20 | 2933.20 |
| 2000 | 0.998239356 | 1st | 3050.70 | 3050.70 |
| 3000 | 0.998825893 | 1st | 2943.50 | 2943.50 |
| 6000 | 0.999412774 | 3rd | 2788.90 | 3030.40 |
| 10000 | 0.999647623 | 2nd | 2882.60 | 2891.20 |

Table 1: Number of episodes, decay rate and results

We will also present some diagrams that show the full learning curve, meaning the total reward per episode as a function of the episode number.




(a) 100 episodes

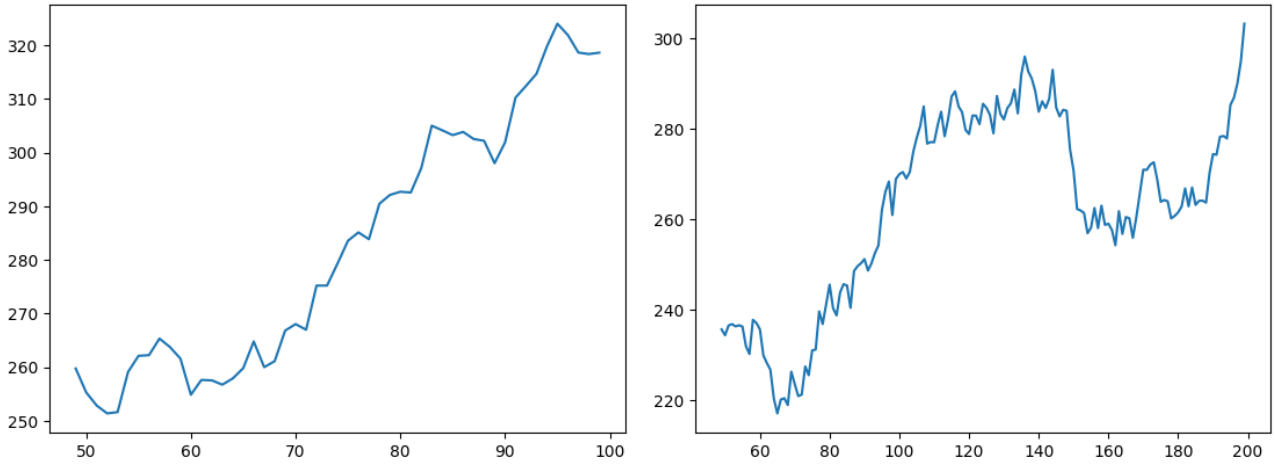
(b) 200 episodes

(c) 300 episodes

(d) 500 episodes

Figure 1: Full learning curve

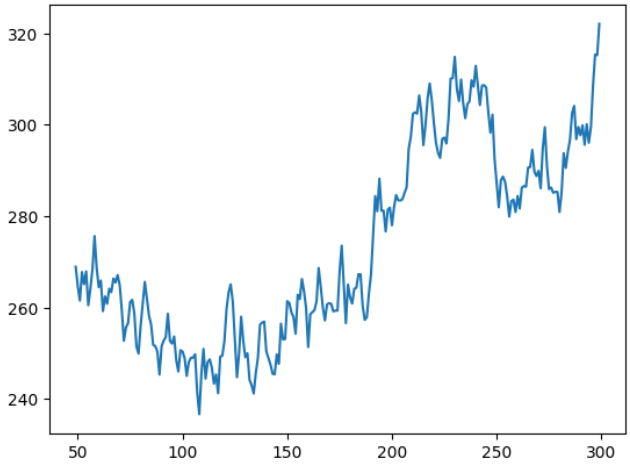
We observe some big fluctuations in the total reward per episode throughout the whole process of training! **However this phenomenon is expected and it can be attributed to the nature of our task!** We can understand this by giving a simple example. When our agent plays versus a Cooperator (a strategy that always plays C) the worst it can do is gather 300 points and the best is to gather 500 points, but when it plays versus a Defector (a strategy that always plays D) the worst it can do is gather 0 points and the best is to gather 100! So these fluctuations do not concern us. In order to better visualize the learning curve we will show a moving average plot of it, that includes the average reward for the current and previous 49 rewards. Also, after 500 episodes the full learning curve becomes difficult to interpret due to its density.



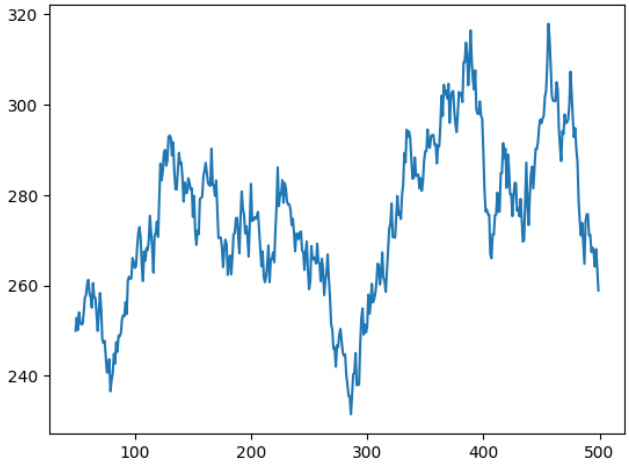

(a) 100 episodes




(b) 200 episodes

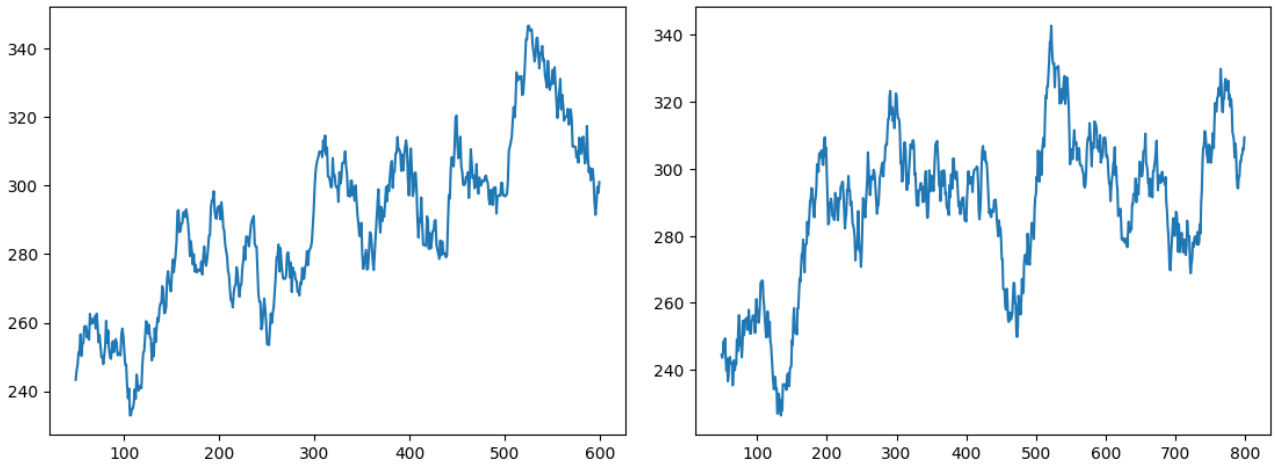



(c) 300 episodes




(d) 500 episodes

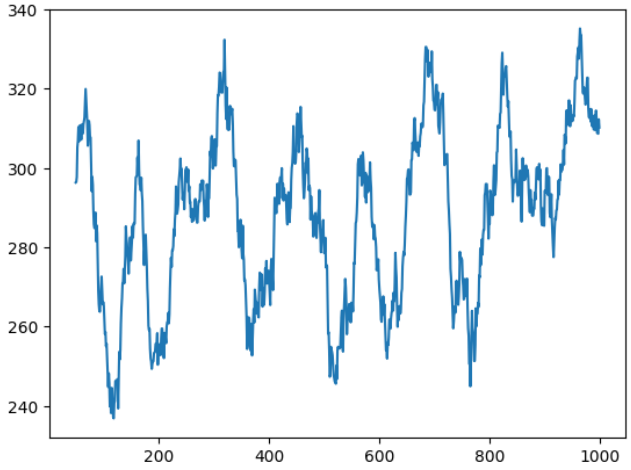
Figure 2: Moving average for 100, 200, 300 and 500 episode training



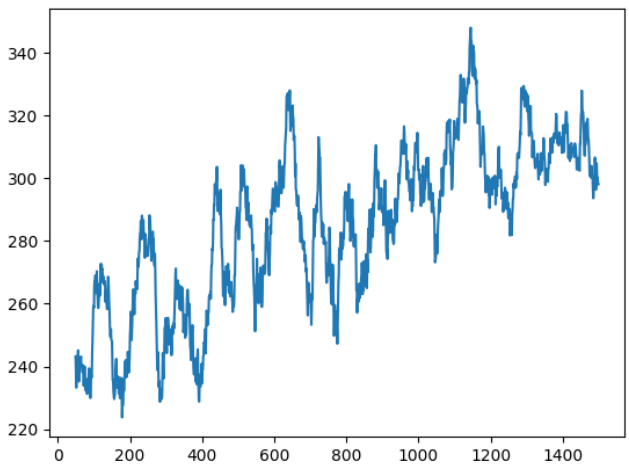

(a) 600 episodes




(b) 800 episodes

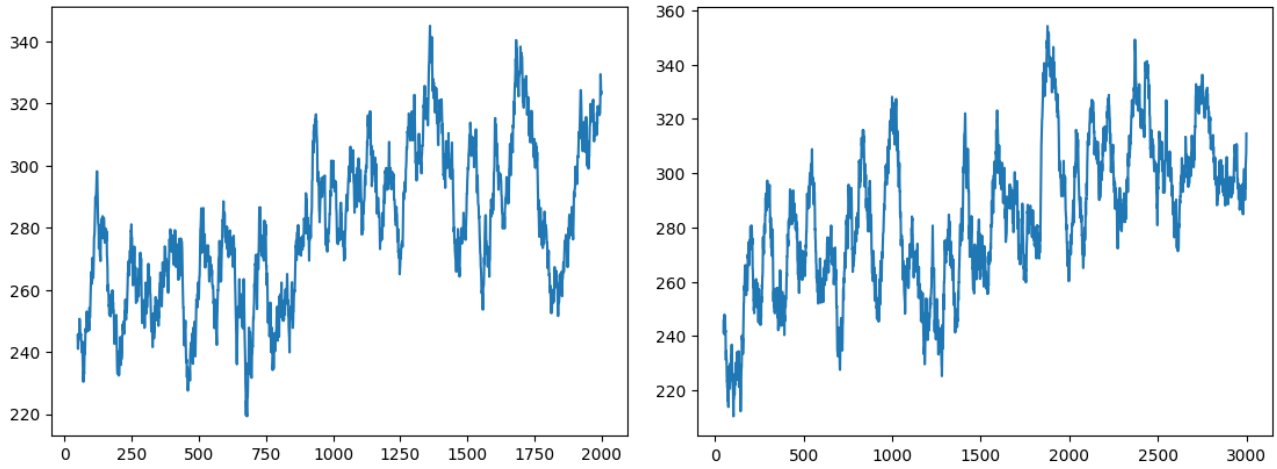



(c) 1000 episodes




(d) 1500 episodes

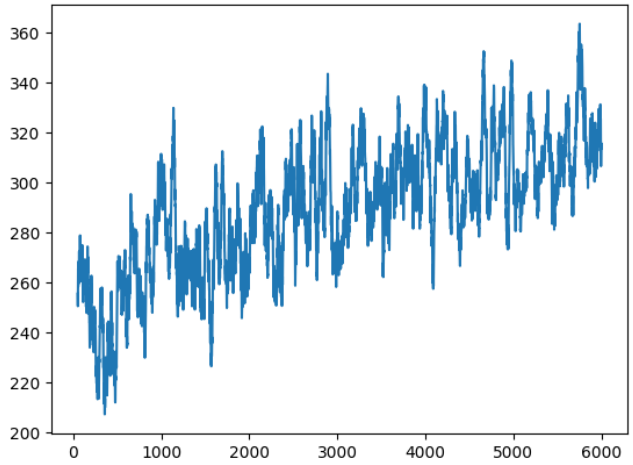
Figure 3: Moving average for 600, 800, 1000 and 1500 episode training



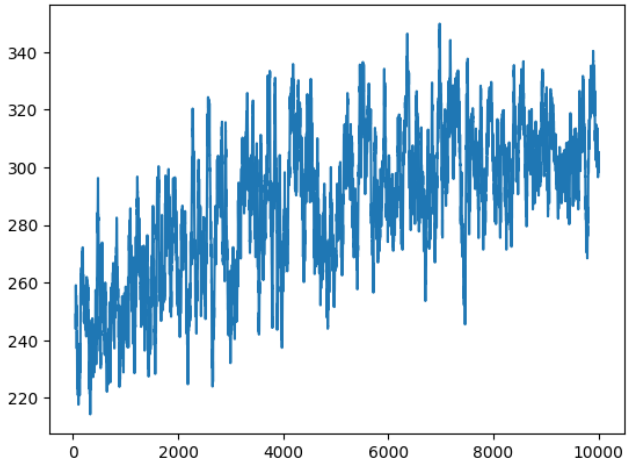

(a) 2000 episodes




(b) 3000 episodes




(c) 6000 episodes

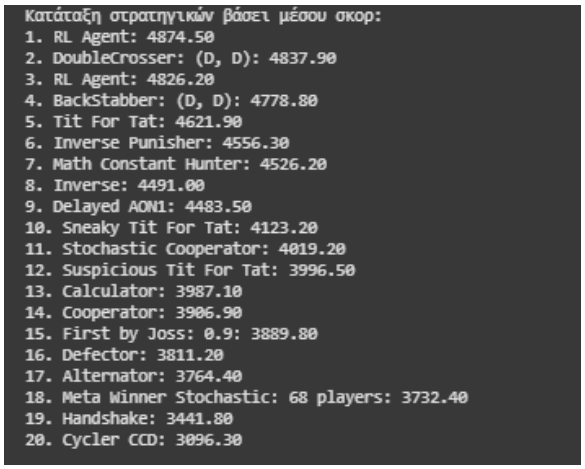



(d) 10000 episodes

Figure 4: Moving average for 2000, 3000, 6000 and 10000 episode training

We can extract the following observations to further extend our analysis:

- In general we can deduce that our agent learns how to be better, as the average reward is increasing during training in most cases in the long run.
- Our agent seems to be in the best shape, with the parameters that we have set, when training last a few thousand episodes, something which is also expected based on our knowledge on RL.
- Our agent performs good but not explicitly well when we try to train him for 6000 episodes using the methodology described above regarding the handling of $\epsilon$. If we let the agent explore for a few more episodes and we increase the $r_\epsilon$ to 0.999430 we observe that our agent performs really well an ranks 1st in the evaluation tournament once again.
- When training for 10000 episodes we see that our agent finishes second. However we re-run the tournament 5 times and 3 out of the 5 our agent is the winner. This result is a bit better than the one produced with 6000 episodes, and it is still acceptable behavior, considering the fact that it also plays good in the extended version of the tournament.
- We place our agent, when it seems to have the most dominant behavior, in an extended version of the tournament that includes more strategies and a second agent with the same policy, in order to further evaluate the performance.




(a) 1st run of the extended tournament


(b) 2nd run of the extended tournament

Figure 5: 6000 episodes of training with $r_\epsilon = 0.999430$


(a) 1st run of the extended tournament


(b) 2nd run of the extended tournament

Figure 6: 3000 episodes of training with $r_\epsilon = 0.998825893$

Due to the stochastic nature of some strategies the result is expected to not be replicated for each rerun, however it is visible that our agent has adopted a policy that performs really well and both of them end up in a favorable position in the final ranking!

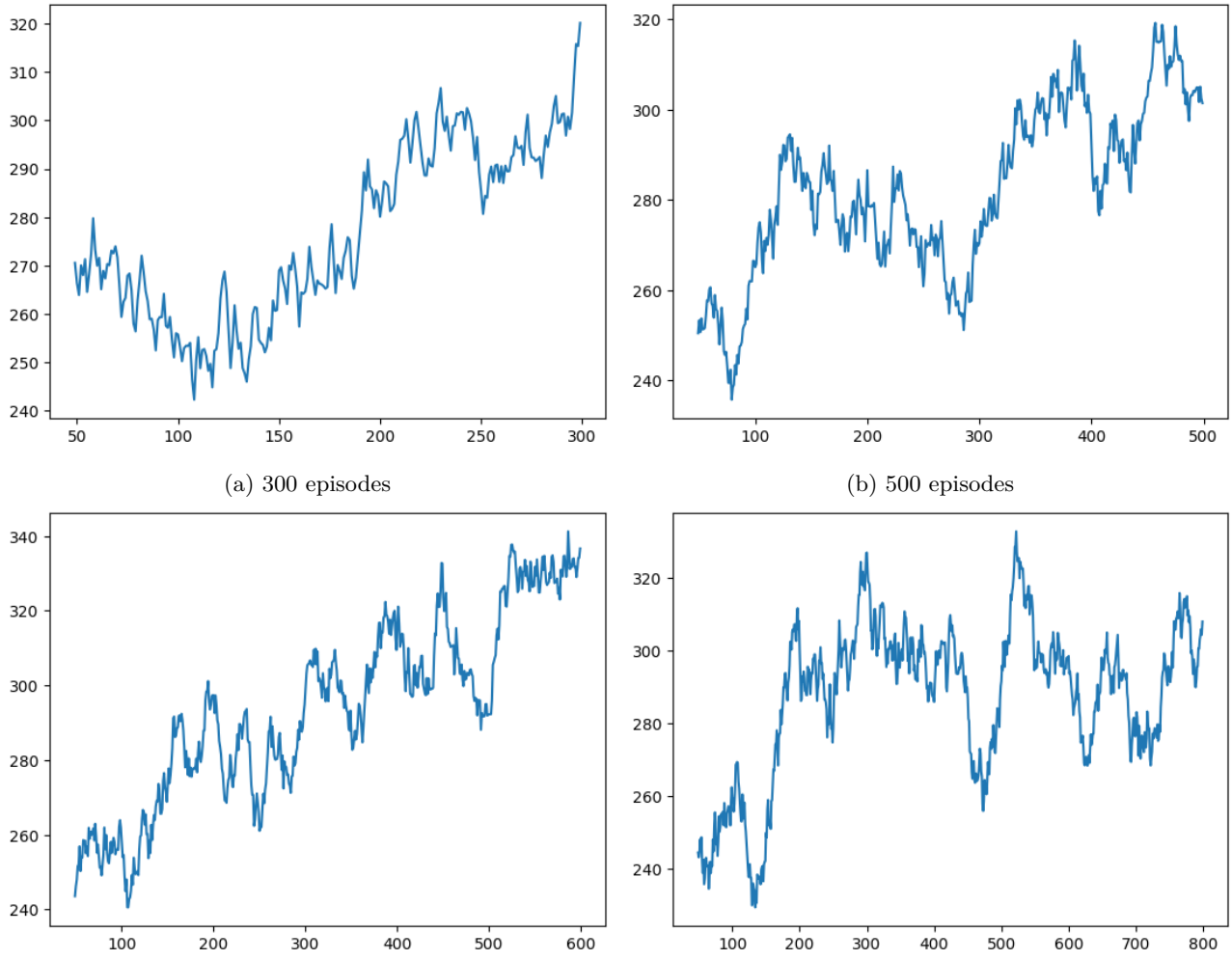
## 4.2 DQN agent using MLP with 1 hidden layer and 20-dimensional input

We now double the size of the input in our NN, in order to take in consideration more of the opponent's and the agent's history. We use the architecture mentioned in case 2. of subsection 3.5.1. We again present the same form of results.

| # of episodes | $r_\epsilon$ | Ranking (out of 13) | Agent Score | Score of Winner |
|---|---|---|---|---|
| 300 | 0.988966021 | 1st | 2942.70 | 2942.70 |
| 500 | 0.992976003 | 2nd | 2867.30 | 2890.00 |
| 600 | 0.994143233 | 1st | 2890.80 | 2890.80 |
| 800 | 0.995604201 | 3rd | 2859.20 | 2885.20 |
| 1000 | 0.996481812 | 2nd | 2869.50 | 3021.20 |
| 1500 | 0.997653164 | 1st | 2924.70 | 2924.70 |
| 2000 | 0.998239356 | 1st | 3030.80 | 3030.80 dominant |
| 3000 | 0.998825893 | 1st | 3002.60 | 3002.60 dominant |
| 6000 | 0.999412774 | 3rd | 2844.50 | 3033.60 |
| 10000 | 0.999647623 | 3rd | 2838.50 | 3033.20 |

Table 2: Number of episodes, decay rate and results

Again, to better visualize the learning curve we will show a moving average plot of it, that includes the average reward for the current and previous 49 rewards.



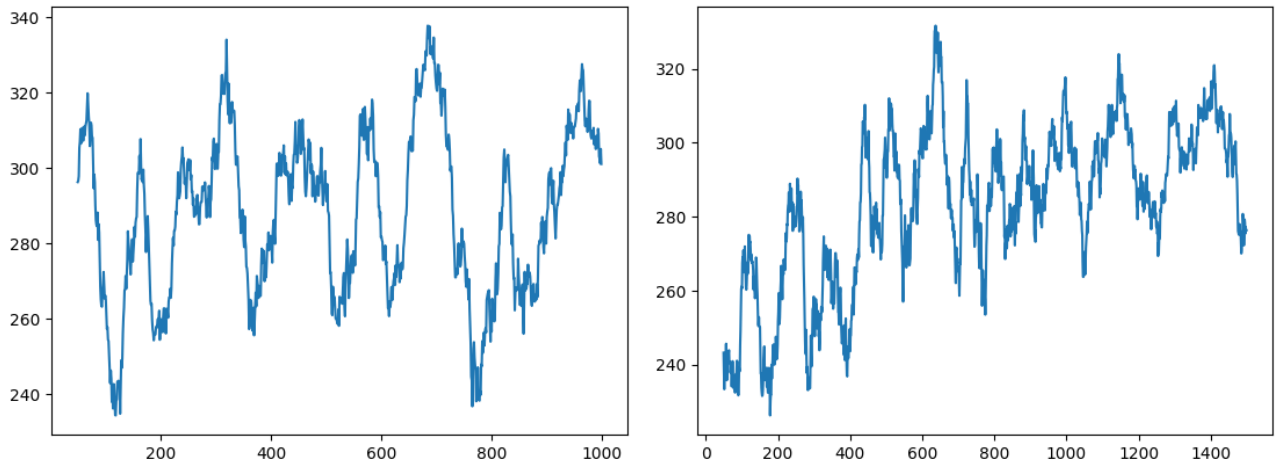
(a) 300 episodes

(b) 500 episodes

(c) 600 episodes

(d) 800 episodes

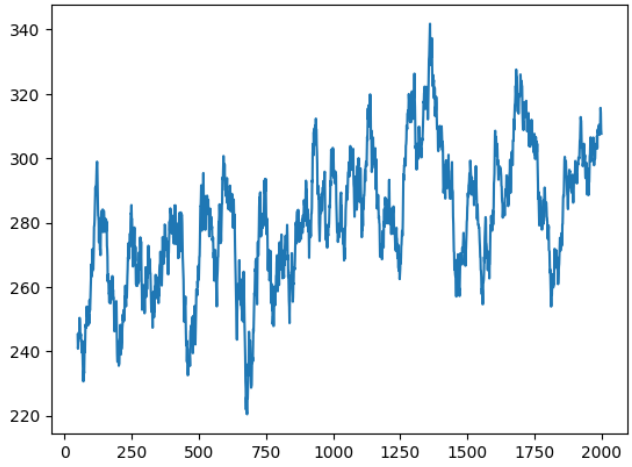
Figure 7: Moving average for 300, 500, 600 and 800 episode training



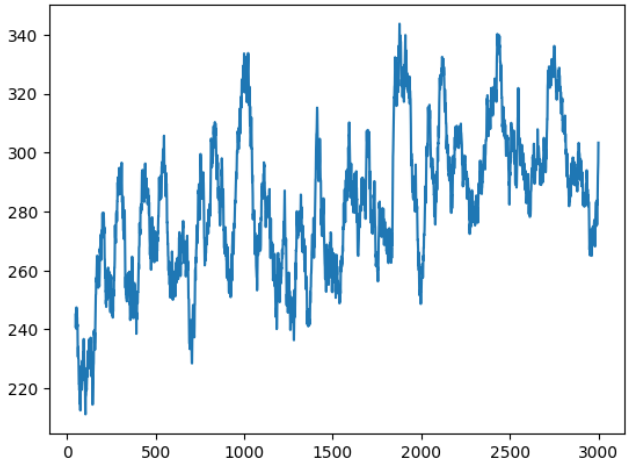

(a) 1000 episodes




(b) 1500 episodes

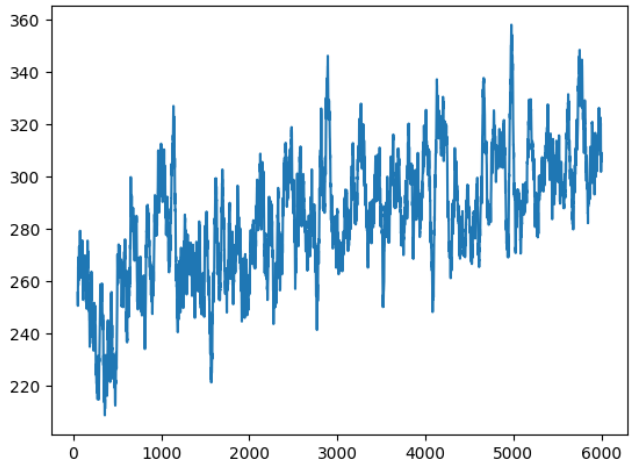



(c) 2000 episodes

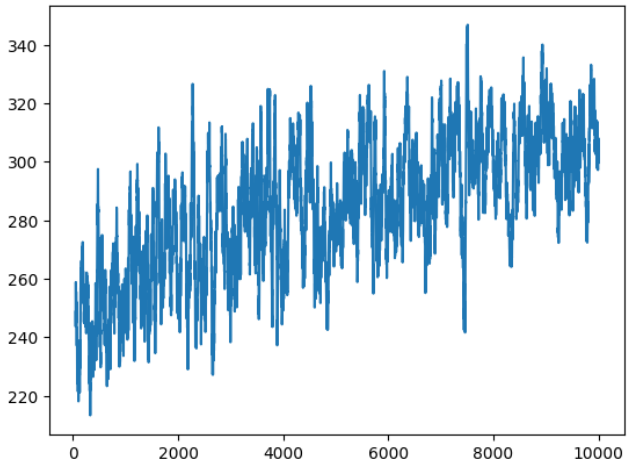



(d) 3000 episodes

Figure 8: Moving average for 1000, 1500, 2000 and 3000 episode training




(a) 6000 episodes

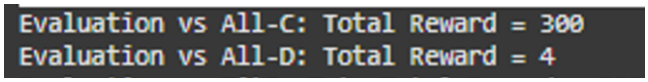



(b) 10000 episodes

Figure 9: Moving average for 6000 and 10000 episode training

Some interesting points can be made again:

- The increasing nature of the learning curve is not as robust as in the version with fewer number of inputs. Even though our agent still performs well, we have not seen an increase in the ranking or in the total score compared to before when training last more than 600 episodes.
- When we train with 6000 or more episodes we see that our agent's performance once again becomes worse. It learns to basically cooperate a lot and defects very frequently meaning that due to overfit it doesn't generalize well. This is evident when we place it in a match versus a Defector, where it only gathers 4 and 3 points for 6000 and 10000 episodes respectively. This is very problematic since Defector is one of the few fully deterministic strategies with very predictable behavior and an agent that generalizes well should face problems with dealing with it. However if we expand the tournament to include more strategies, the effect of the aggressive defections minimizes and our agent performs better.

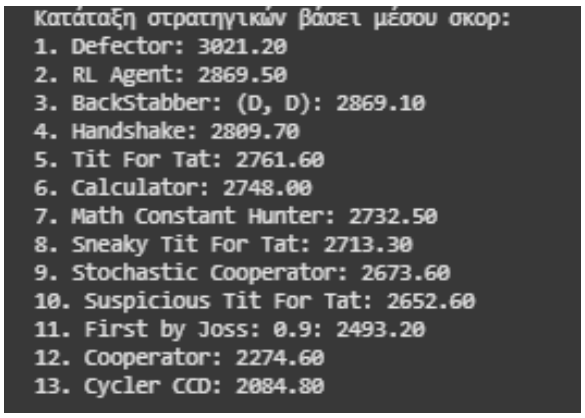



(a) Defector dominates the agent when heads up

| Round | Agent | Opponent |
|---|---|---|
| 1 | C | D |
| 2 | D | D |
| 3 | D | D |
| 4 | D | D |
| 5 | C | D |
| 6 | C | D |
| 7 | C | D |
| 8 | C | D |
| 9 | D | D |
| 10 | C | D |
| 11 | C | D |
| 12 | C | D |
| 13 | C | D |
| 14 | C | D |
| 15 | C | D |
| 16 | C | D |
| 17 | C | D |
| 18 | C | D |
| 19 | C | D |
| 20 | C | D |

(b) Log of the first 20 moves of the match

Figure 10: 6000 episodes of training



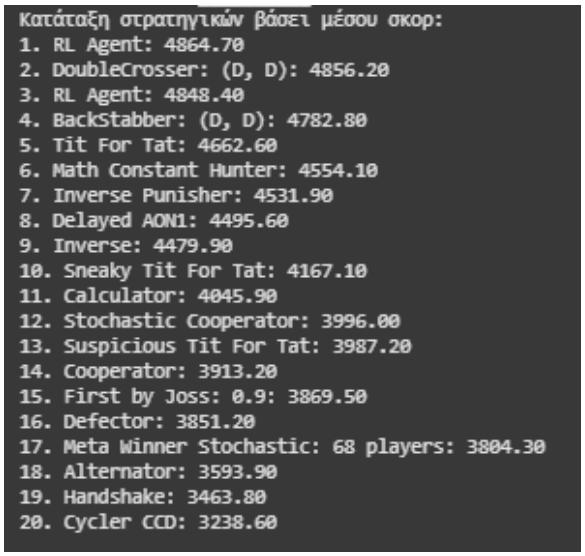

(a) Defectors beat the agent in simple tournament


(b) Defectors beat the agent less in expanded tournament

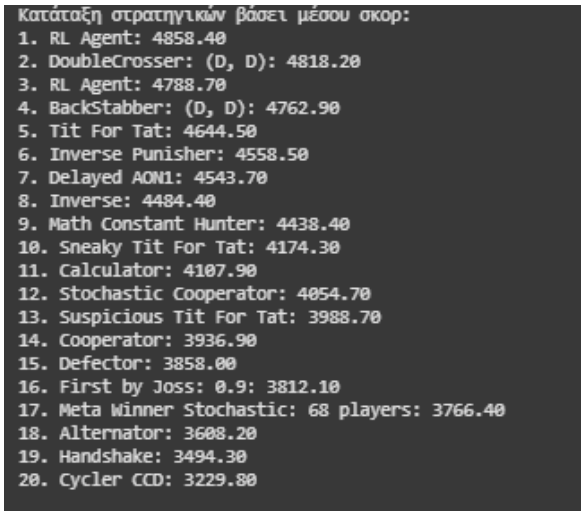
Figure 11: 10000 episodes of training

- We place our agent, when trained for 2000 episodes, in an extended version of the tournament that

includes more strategies and a second agent with the same policy, in order to further evaluate the performance.




(a) 1st run of the extended tournament




(b) 2nd run of the extended tournament

Figure 12: Improved behavior with more strategies added

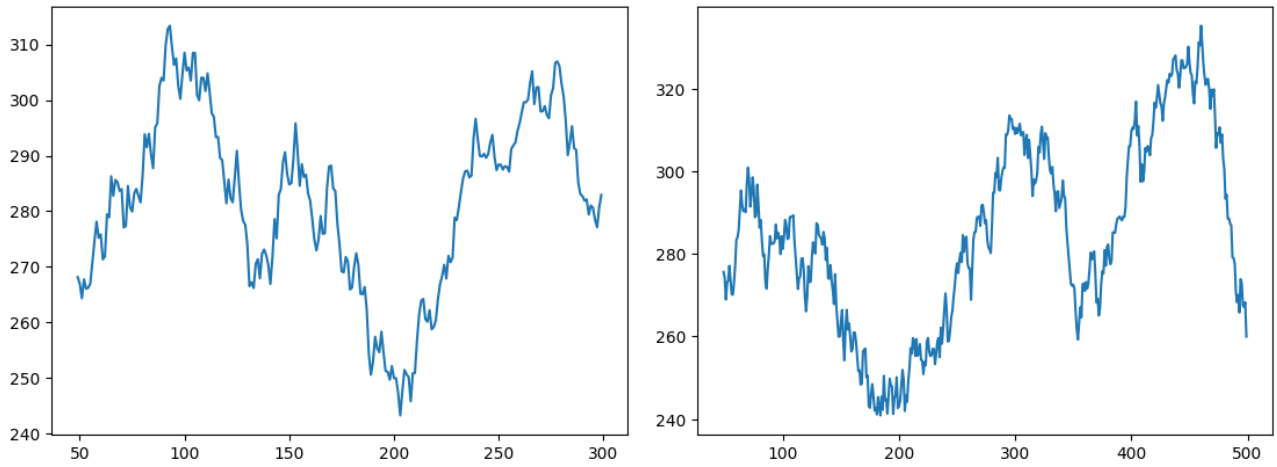
### 4.3 DQN agent using MLP with 2 hidden layers and 20-dimensional input

We now have an input space of 20 dimensions and a deeper MLP. We use the architecture mentioned in case 3. of subsection 3.5.1. We again present the same form of results.

| # of episodes | $r_\epsilon$ | Ranking (out of 13) | Agent Score | Score of Winner |
|---|---|---|---|---|
| 300 | 0.988966021 | 3rd | 2821.60 | 2917.20 |
| 500 | 0.992976003 | 5th | 2787.70 | 3013.20 |
| 600 | 0.994143233 | 3rd | 2892.80 | 2911.20 |
| 800 | 0.995604201 | 1st | 2898.80 | 2898.80 |
| 1000 | 0.996481812 | 4th | 2745.80 | 2908.60 |
| 1500 | 0.997653164 | 1st | 2899.00 | 2899.00 |
| 2000 | 0.998239356 | 3rd | 2785.20 | 3016.00 |
| 3000 | 0.998825893 | 1st | 2934.00 | 2934.00 |
| 6000 | 0.999412774 | 2nd | 2851.60 | 2875.60 |
| 10000 | 0.999647623 | 9th | 2636.60 | 3041.60 |

Table 3: Number of episodes, decay rate and results

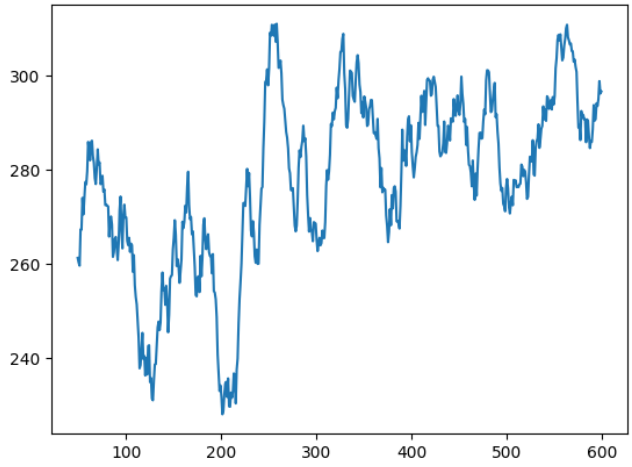
Once again, in order to better visualize the learning curve we will show a moving average plot of it, that includes the average reward for the current and previous 49 rewards.



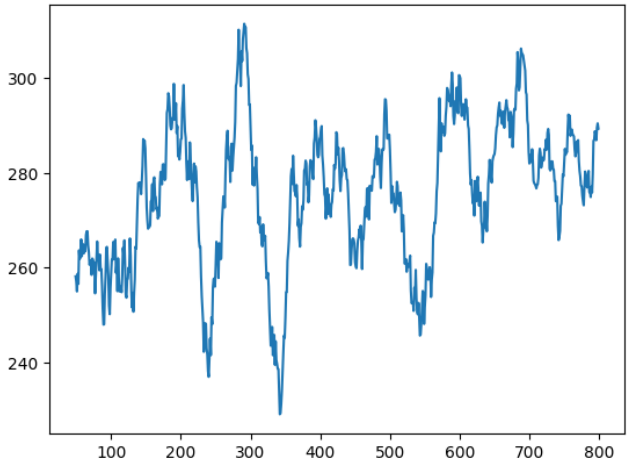

(a) 300 episodes




(b) 500 episodes

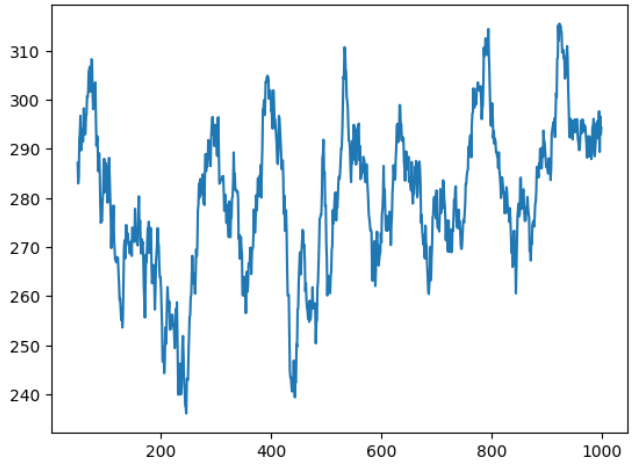



(c) 600 episodes

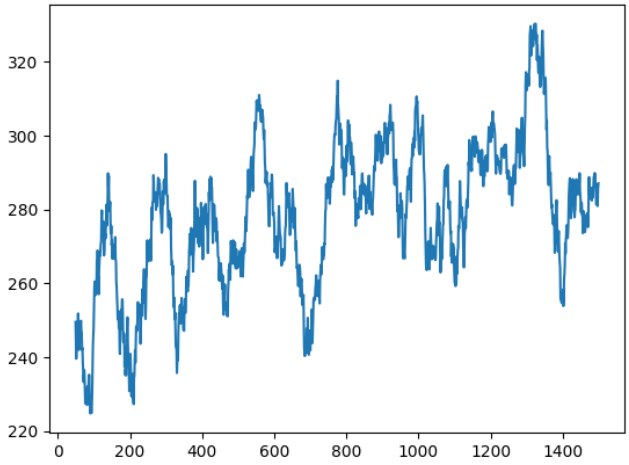



(d) 800 episodes

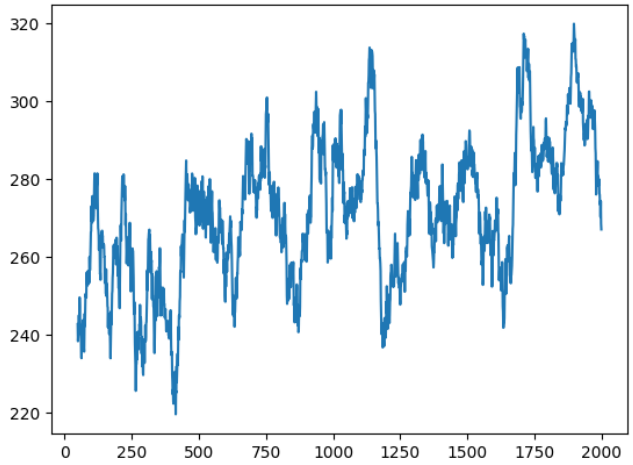
Figure 13: Moving average for 300, 500, 600 and 800 episode training



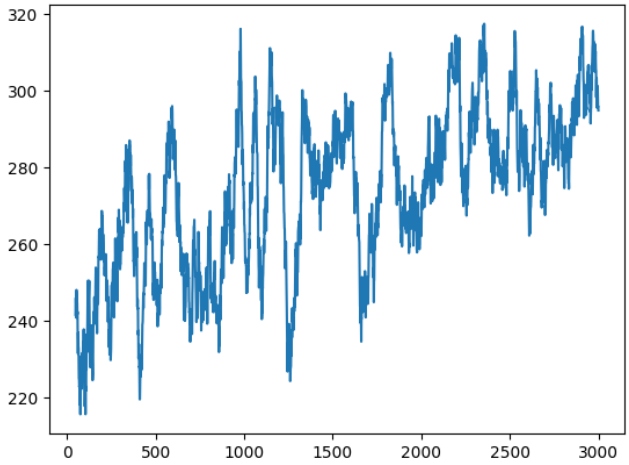

(a) 1000 episodes




(b) 1500 episodes

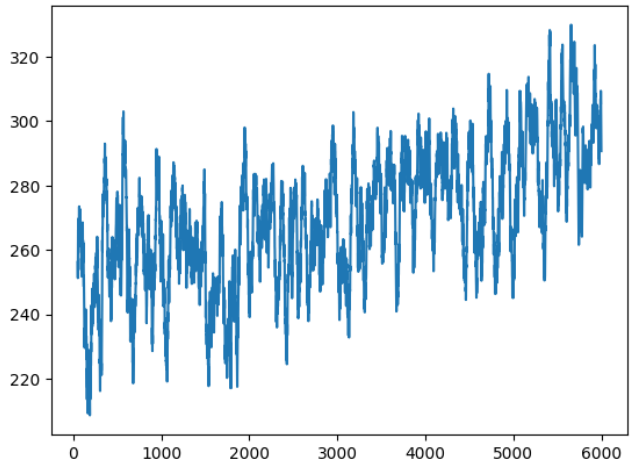



(c) 2000 episodes

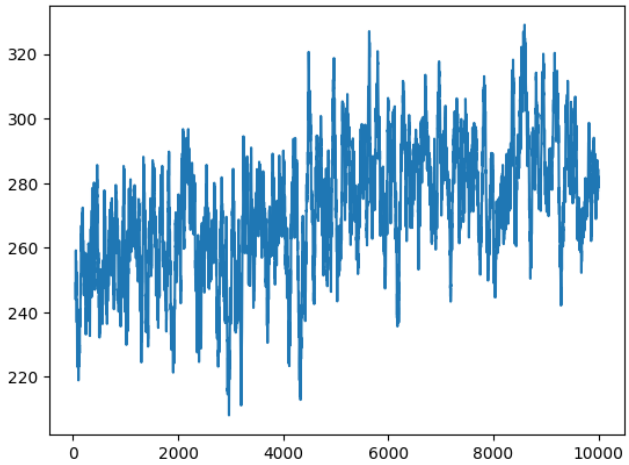



(d) 3000 episodes

Figure 14: Moving average for 1000, 1500, 2000 and 3000 episode training




(a) 6000 episodes




(b) 10000 episodes

Figure 15: Moving average for 6000 and 10000 episode training

Seeing the results at this stage, we can make some interesting points about the process of training and

the architecture of the NN:

- It is obvious that the use of a MLP with a simple architecture with 1 hidden layer and smaller input dimension performs better than both of the 2 other options.
- When training for 10000 episodes we observe an interesting behavior. When our agent plays against a Defector it gathers **1** point, almost the minimum available reward. Our agent learns that cooperation is favorable in the long run and spams it, thus becoming exploitable by opponents that act aggressively and defect a lot. So, we can conclude that our agent does not generalize well in that case. We can see that in the tournament we winner is the Defector, a very simple strategy that manages do win, due to the naiveness of our agent and other strategies like Cooperator.

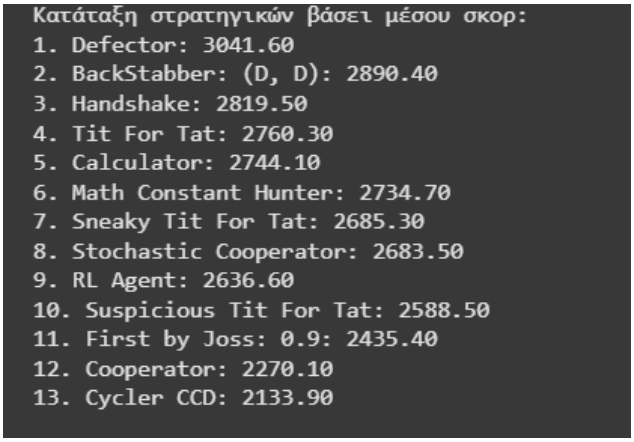
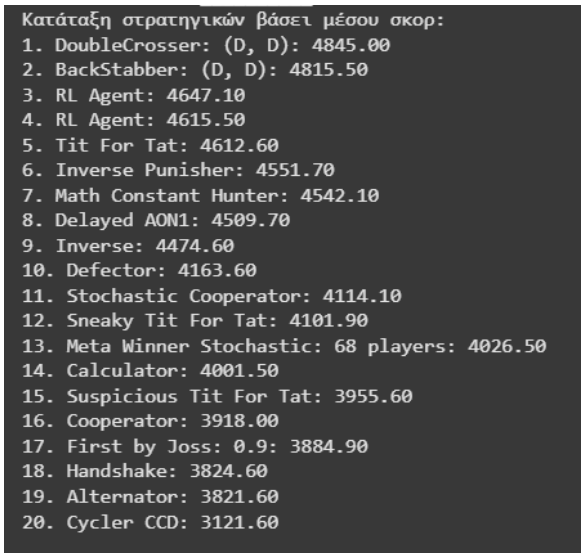
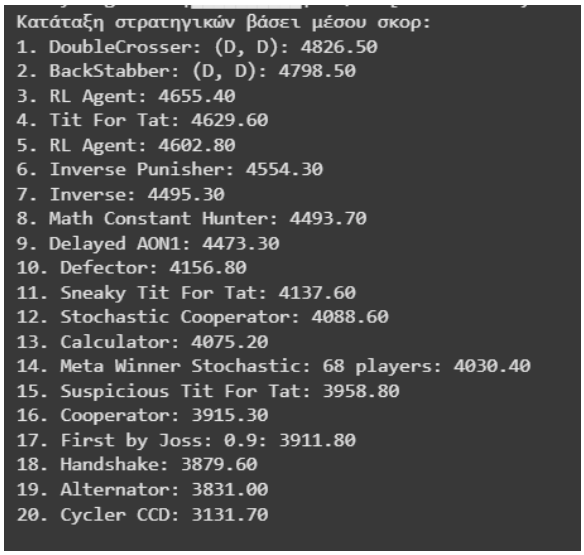



Figure 16: Ranking with 10000 episodes of training. Defectors take advantage of naiveness.

- Our agent exhibits the best behavior when training for 1500 and 3000 episodes, so we place it in the extended version of the tournament and run each tournament twice.

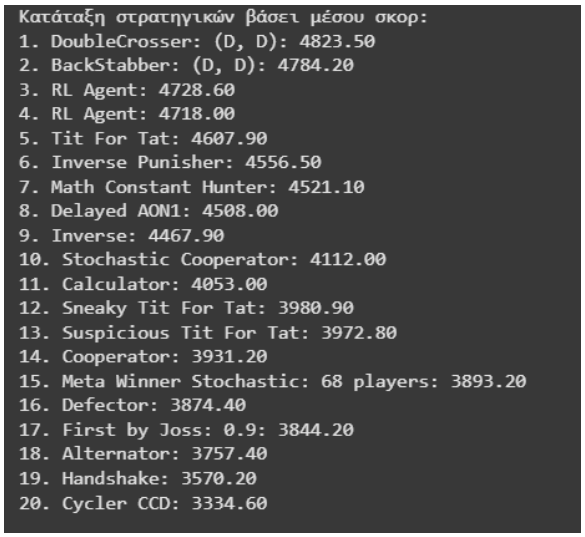



(a) 1st run of the extended tournament




(b) 2nd run of the extended tournament

Figure 17: Trained for 1500 episodes




(a) 1st run of the extended tournament


(b) 2nd run of the extended tournament

Figure 18: Trained for 3000 episodes

- It is now evident that the best architecture choice out of the 3 presented so far is the simplest with fewer inputs and one hidden layer.

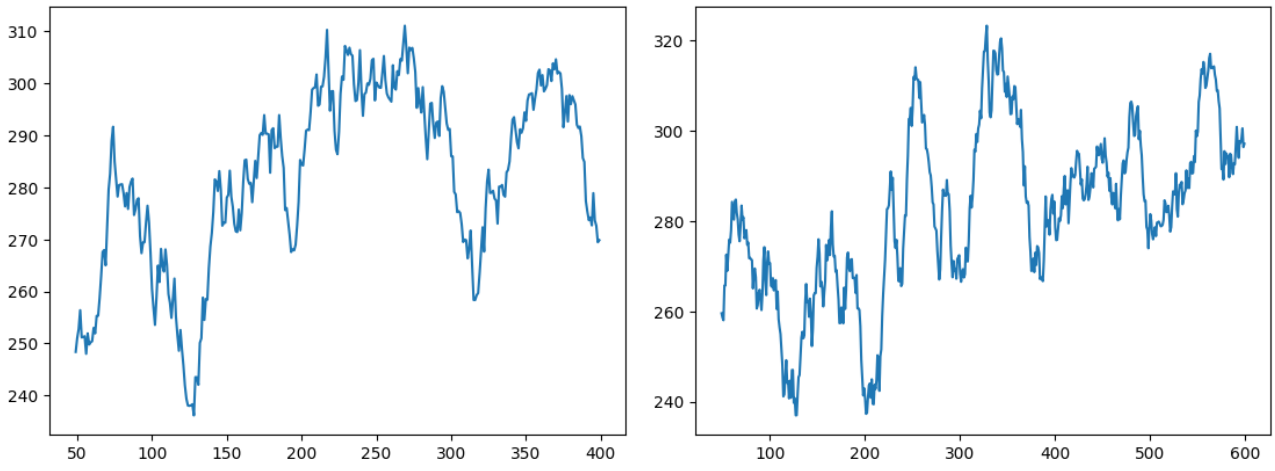
## 4.4 DQN agent using a CNN

We use the architecture mentioned in case 4. of subsection 3.5.1 with 20-dimensional input and a CNN. We again present the same form of results.

| # of episodes | $r_\epsilon$ | Ranking (out of 13) | Agent Score | Score of Winner |
|---|---|---|---|---|
| 400 | 0.991227726 | 11th | 2270.60 | 3048.60 |
| 600 | 0.994143233 | 5th | 2771.40 | 3029.20 |
| 800 | 0.995604201 | 2nd | 2856.60 | 2893.20 |
| 1000 | 0.996481812 | 1st | 2957.10 | 2957.10 |
| 2000 | 0.998239356 | 4th | 2674.40 | 2786.10 |
| 4000 | 0.999168198 | 4th | 2761.70 | 2888.30 |
| 6000 | 0.999412774 | 2nd | 2849.90 | 2886.80 |
| 10000 | 0.999647623 | 1st | 2923.40 | 2923.40 |

Table 4: Number of episodes, decay rate and results

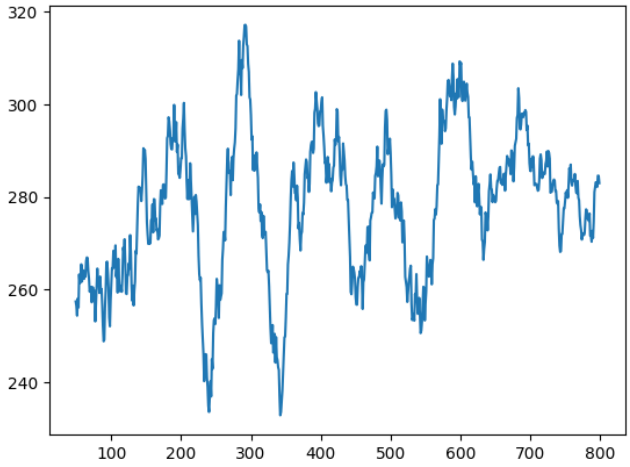
Finally, one last time, in order to better visualize the learning curve we will show a moving average plot of it, that includes the average reward for the current and previous 49 rewards.



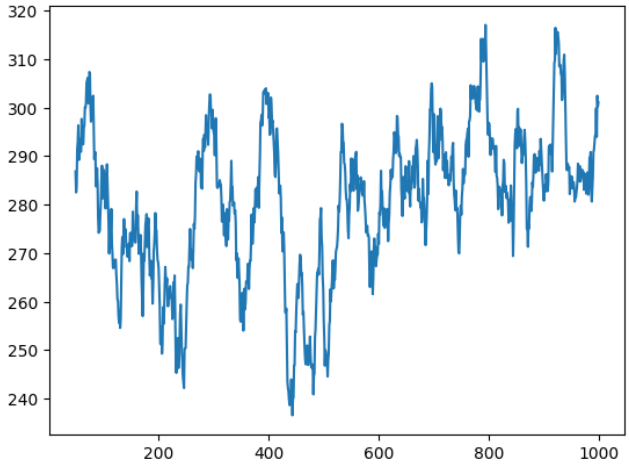

(a) 400 episodes




(b) 600 episodes

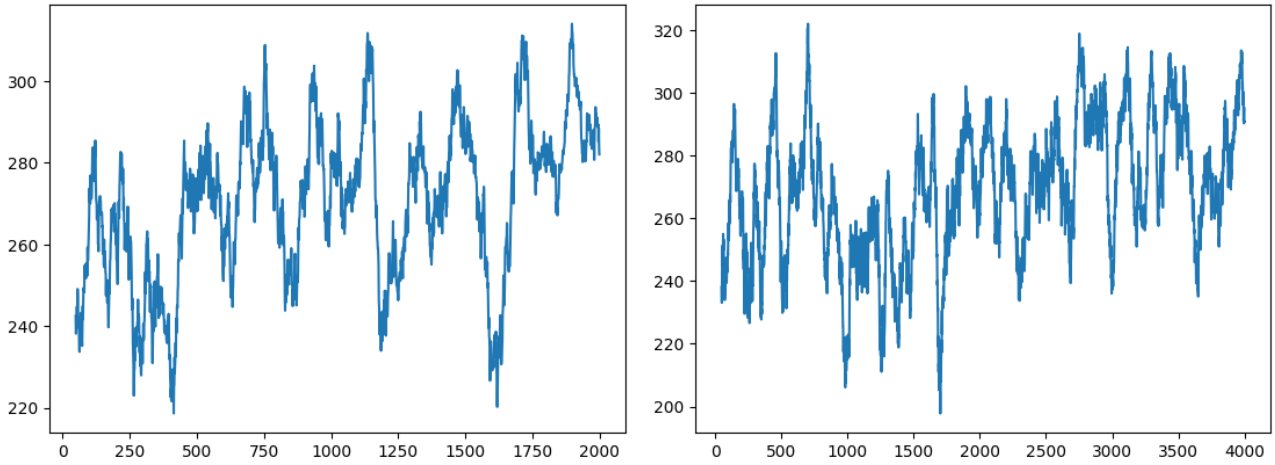



(c) 800 episodes




(d) 1000 episodes

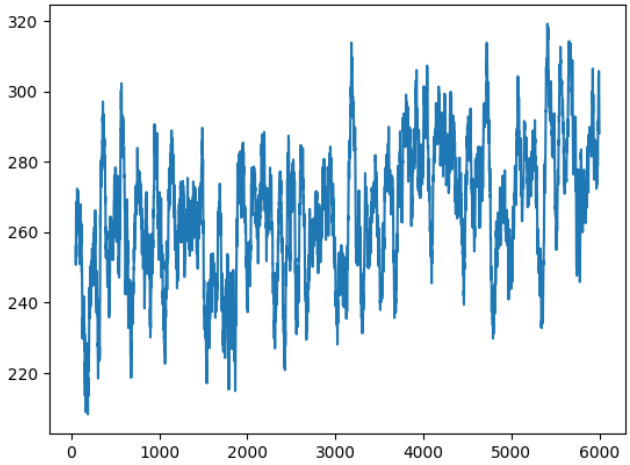
Figure 19: Moving average for 400, 600, 800 and 1000 episode training



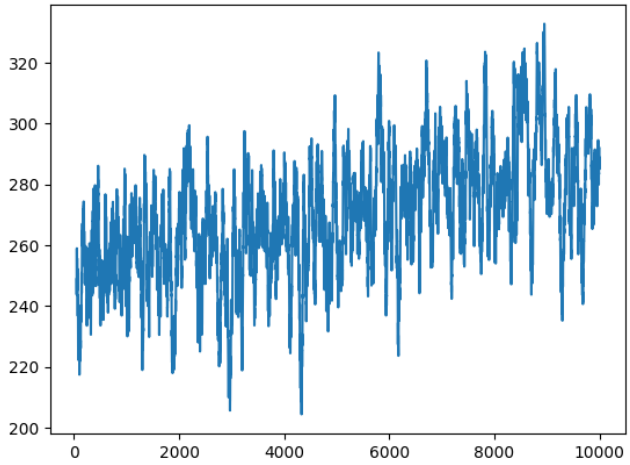

(a) 2000 episodes




(b) 4000 episodes




(c) 6000 episodes

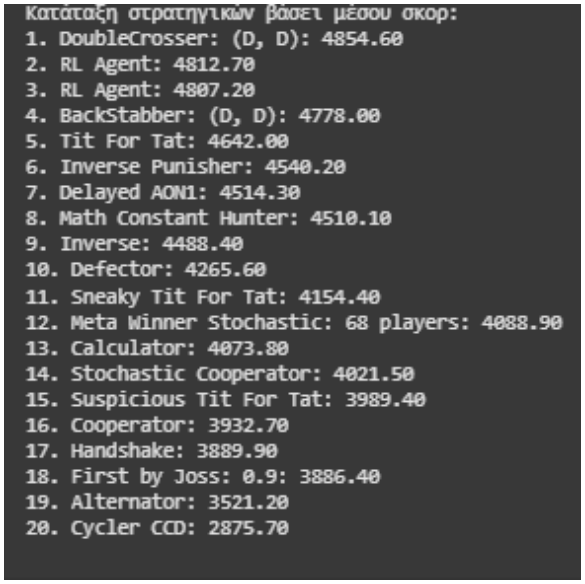



(d) 10000 episodes

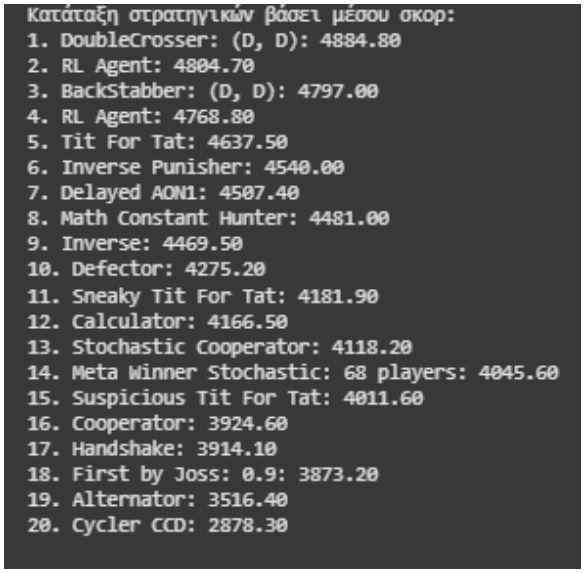
Figure 20: Moving average for 2000, 4000, 6000 and 10000 episode training

Having the results we can make some interesting points once again:

- We conclude that the use of a more complex NN like a Convolutional NN in DQN is unnecessary and doesn't lead to better results for our task.
- We will again place our top performers in an extended version of the tournament and see how they perform.

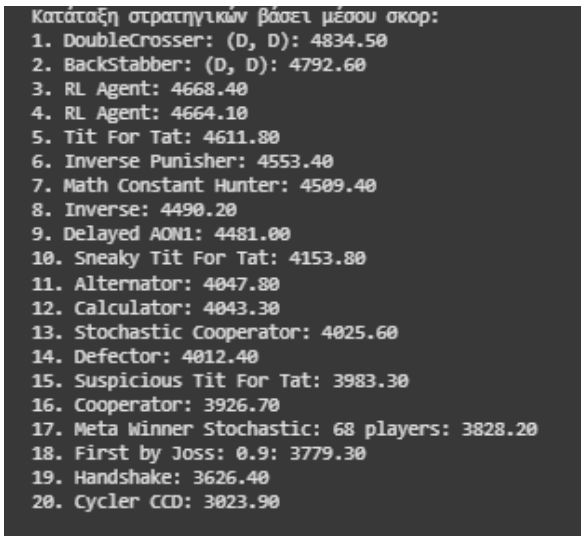



(a) 1st run of the extended tournament




(b) 2nd run of the extended tournament

Figure 21: Trained for 1000 episodes




(a) 1st run of the extended tournament


(b) 2nd run of the extended tournament

Figure 22: Trained for 10000 episodes

## 4.5 Conclusions & General comments

After all the simulations and experiments some general comments and conclusions can be made:

- The agent learns to perform well and generalize, which was the main challenge of this project so we can deem it as successful.
- Obviously to further confirm our results it is necessary to expand the opponent pool during evaluation to an even larger group, however initial results are encouraging.
- We have observed that our agent exhibits the best behavior when he adopts a nice strategy. A strategy is deemed as nice in Game Theory terms when it is not the first to defect. When logging each move for our agent, we can see that when it has the most dominant performance it does not opt to defect first!

This is an interesting observation that comes in accordance to our knowledge of Axelrod Tournaments and the ranking of our agent, since in the original Tournaments, the top performing strategies where always nice strategies.

- Evolutionary algorithms and Swarm Optimization techniques such as Particle Swarm Optimization can also be used to create an agent that learns to play Iterated Prisoner's Dillema in a favorable manner.

### 4.6 Extras

- We have defined a `set_seed(seed)` function that we call with a specific seed (42) before all of our experiments in order to guarantee reproducibility of the results presented in this report.
- In the end of every notebook we have included a cell containing a script that logs every move of every opponent and our agent to help us visualize our agent's performance against a specific opponent, to further analyze patterns and derive conclusions.
- In order to place our agent in an Axelrod tournament, we have also develop the `RLWrapper(axl.Player)` class, which is a subclass of axl.Player, in order to allow the agent to be compatible and smoothly participate in the tournament and be evaluated and ranked against standard strategies by the native Axelrod mechanics.
- Every .ipynb file can be executed from the top to bottom to visualize the results.

# References


[1] Robert Axelrod and Douglas Dion. The further evolution of cooperation. *Science*, 242(4884):1385–1390, 1988.

[2] Robert Axelrod and William D. Hamilton. The evolution of cooperation. *Science*, 211(4489):1390–1396, 1981.

[3] Axelrod Python Documentation. Axelrod strategy index. `https://axelrod.readthedocs.io/en/stable/reference/strategy_index.html`.

[4] Vince Knight et al. Axelrod-python: Repository for the iterated prisoner's dilemma library. `https://github.com/Axelrod-Python/Axelrod`, 2017.

[5] Vince Knight, Owen Campbell, Marc Harper, et al. The axelrod library: Reproducible research into the iterated prisoner's dilemma. *Journal of Open Research Software*, 4(1):A11, 2016.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[7] Anastasios Tefas Nikolaos Passalis. Εισαγωγή στη Βαθιά Ενισχυτική Μάθηση (deep reinforcement learning). `https://elearning.auth.gr/pluginfile.php/120583/mod_resource/content/1/reinforcement-intro.pdf`, 2024. Course material, Aristotle University of Thessaloniki, Instructor: Anastasios Tefas.

[8] Martin A. Nowak and Karl Sigmund. Evolution of cooperation. *Nature*, 437(7063):1291–1298, 2005.

[9] Wikipedia contributors. Reinforcement learning – wikipedia, the free encyclopedia, 2024. [Online; accessed 18-May-2025].