Vilnius University
Faculty of Mathematics and Informatics
Informatics

# Self-Play Training Using Deep Reinforcement Learning

# Besirungiančių agentų strategijos taikymas naudojant gilųjį skatinamąjį mokymąsi

Project Work

By: Jokūbas Kondrackas
Supervisor: asist. dr. Linas Petkevičius

Vilnius - 2021

# Table of Contents


1 Reinforcement Learning 4
- 1.1 Action and Observation Space . . . 4
- 1.2 Algorithms . . . 5
  - 1.2.1 Proximal Policy Optimization(PPO) . . . 5
  - 1.2.2 Soft Actor-Critic(SAC) . . . 5
- 1.3 Competitive Self-Play . . . 5

2 Unity 7
- 2.1 Short introduction to Unity . . . 7
- 2.2 ML-Agents . . . 7
- 2.3 Hyperparameters and other settings . . . 7
  - 2.3.1 Hyperparameters . . . 8
  - 2.3.2 Network Settings . . . 9
  - 2.3.3 Self-Play . . . 9

3 Experiments 10
- 3.1 The Environment . . . 10
  - 3.1.1 Environment Details . . . 10
  - 3.1.2 Agent Details . . . 11
- 3.2 Hyperparameters . . . 12
- 3.3 Results Explanation . . . 12
- 3.4 Experiment 1 . . . 13
  - 3.4.1 Reward table . . . 13
  - 3.4.2 Results . . . 13
  - 3.4.3 Experiment Conclusion . . . 14
- 3.5 Experiment 2 . . . 15
  - 3.5.1 Reward table . . . 15
  - 3.5.2 Results . . . 15
  - 3.5.3 Experiment Conclusion . . . 16
- 3.6 Experiment 3 . . . 17
  - 3.6.1 Reward table . . . 17
  - 3.6.2 Results . . . 17
  - 3.6.3 Experiment Conclusion . . . 18

4 Conclusion 18

# Introduction

Reinforcement learning (RL) is a computational approach to understanding and automating goal-directed learning and decision making. It is distinguished from other computational approaches by its emphasis on learning by an agent from direct interaction with its environment, without requiring exemplary supervision or complete models of the environment [SB18].

The advantage of reinforcement learning versus other machine learning are is that it does not need any training date beforehand and learns as it goes along.

# Purpose

The aim of this study is to investigate how to use deep reinforcement learning to train agents using Unity's ML-Agents package in an environment that would allow for competitive self-play. In addition to that, multiple reward systems are investigated to compare their training efficiency.

# Objectives

Goals:

- Design a mini-game that the agents could compete in.
- Design multiple reward structures for the agents.
- Tune the hyperparameters in order to enable self-play.
- Compare different reward structures in the same environment with the same hyperparameters.
- Present results and provide recommendations.

# 1 Reinforcement Learning

Reinforcement learning is one of the main machine learning (ML) branches alongside supervised learning and unsupervised learning. The simplest RL model consists from agent, which observes environment and makes decisions, and environment, which takes input, changes state and gives reward to the agent [SB18].

Simply put, the agent and environment interact at each of a sequence of discrete time steps. At each time step t, the agent receives some representation of the environments state, and on that basis selects an action. One-time step later, in part as a consequence of its action, the agent receives a numerical reward, and finds itself in a new state [SB18], see Figure 1.

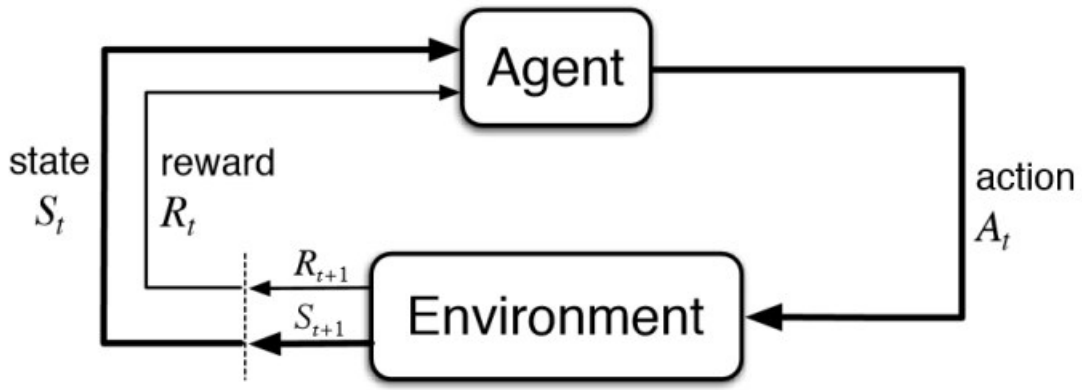


Figure 1: Agent-Environment Interface, taken from [SB18]

## 1.1 Action and Observation Space

To interact with the environment the agent has to take an action. This action type can vary depending on the chosen environment, however, it is either discrete or continuous. In discrete action space, each action is an integer from finite natural numbers range, where each number represents a specific action. Continuous actions space is a real number/vector [KSH20].

Observation space classification is identical to action space, however, it represents not action, which agent takes, but observation, which agent receives after taking action.

## 1.2 Algorithms

There are many available algorithms that can be used to train agents. However, in this study, game engine Unity is used to train agents. At the moment, it supports 2 training algorithms which are Proximal Policy Optimization and Soft Actor-Critic.

### 1.2.1 Proximal Policy Optimization(PPO)

The Proximal Policy Optimization (PPO) was introduced by the OpenAI team in 2017 and became one of the most popular Reinforcement Learning method that pushed all other RL methods at that moment aside. PPO involves collecting a small batch of experiences interacting with the environment and using that batch to update its decision-making policy. Once the policy is updated with that batch, the experiences are thrown away and a newer batch is collected with the newly updated policy. This is the reason why it is an "on-policy learning" approach where the experience samples collected are only useful for updating the current policy.

The main idea is that after an update, the new policy should be not too far from the old policy. For that, PPO uses clipping to avoid too large updates. This leads to less variance in training at the cost of some bias, but ensures smoother training and also makes sure the agent does not go down to an unrecoverable path of taking senseless actions [SWD+].

### 1.2.2 Soft Actor-Critic(SAC)

The Soft Actor-Critic (SAC) is an algorithm that optimizes a stochastic policy in an off-policy way, forming a bridge between stochastic policy optimization and Deep Deterministic Policy Gradient approaches. A central feature of SAC is entropy regularization. The policy is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy. This has a close connection to the exploration-exploitation trade-off: increasing entropy results in more exploration, which can accelerate learning later on. It can also prevent the policy from prematurely converging to a bad local optimum [HZAL18].

## 1.3 Competitive Self-Play

Competitive Self-Play is basically an agent training method that allows the agents to play with themselves [BPS+18]. It is analogous to how humans structure competition. For example, a human learning to play tennis would train against opponents of similar skill level because an opponent that is

too strong or too weak is not as conducive to learning the game. From the standpoint of improving one's skills, it would be far more valuable for a beginner-level tennis player to compete against other beginners than, say, against a newborn child or a professional player. The former couldn't return the ball, and the latter wouldn't serve them a ball they could return. When the beginner has achieved sufficient strength, they move on to the next tier of tournament play to compete with stronger opponents[1].

[1]https://blogs.unity3d.com/2020/02/28/training-intelligent-adversaries-using-self-play-with-ml-agents/

# 2 Unity

## 2.1 Short introduction to Unity

Unity[2] is a powerful cross-platform 3D engine. It is a complete 3D environment, suitable for laying out levels, creating menus, doing animation, writing scripts, organizing projects and so on. Thanks to the Unity ML-Agents[3] package, it is also an excellent tool for training agents and creating custom environments for those agents to train in.

## 2.2 ML-Agents

Unity ML-Agents trains agents with reinforcement learning and evolutionary methods [MSG99] via a Python API. It is an open-source project that enables games and simulations to serve as environments for training agents. Many of the used algorithms in ML-Agents toolkit leverage some form of deep learning. More specifically, Unity implementations are built on top of the open-source library TensorFlow[4].

The ML-Agents toolkit ships with several implementations algorithms for training agents. More specifically, during training, all of the agents in the scene send their observations to the Python API through the external Communicator. The Python API processes these observations and sends back actions for each agent to take. During training these actions are mostly exploratory to help the Python API learn the best policy for each agent. Once training concludes, the learnt policy for each agent can be exported as a model file. Then during the inference phase, the agents still continue to generate their observations, but instead of being sent to the Python API, they will be fed into their model to generate the optimal action for each agent to take at every point in time[5].

## 2.3 Hyperparameters and other settings

One of the first decisions that one needs to make regarding the training run is which trainer to use: Proximal Policy Optimization(PPO) or Soft Actor-Critic(SAC). In this project, I've used PPO, therefore the hyperparameters that I'll explain in this section will be either common to both trainers

[2]https://unity.com/

[3]https://github.com/Unity-Technologies/ml-agents

[4]https://blogs.unity3d.com/2020/05/12/announcing-ml-agents-unity-package-v1-0/

[5]https://github.com/Unity-Technologies/ml-agents/blob/master/docs/ML-Agents-Overview.md

or exclusive to PPO[6].

### 2.3.1 Hyperparameters

See Table 1 for hyperparameter explanation.

| Setting | Description |
|---|---|
| Buffer Size | Number of experiences to collect before updating the policy model. Typically a larger buffer size corresponds to more stable training updates. |
| Learning Rate | Initial learning rate for gradient descent. Corresponds to the strength of each gradient descent update step. |
| Beta | Strength of the entropy regularization, which makes the policy "more random." This ensures that agents properly explore the action space during training. |
| Epsilon | Influences how rapidly the policy can evolve during training. Corresponds to the acceptable threshold of divergence between the old and new policies during gradient descent updating. |
| Lambda | Regularization parameter (lambda) used when calculating the Generalized Advantage Estimate (GAE). This can be thought of as how much the agent relies on its current value estimate when calculating an updated value estimate. |
| Epoch Number | Number of passes to make through the experience buffer when performing gradient descent optimization. |
| Learning Rate Schedule | Determines how learning rate changes over time. |

Table 1: Hyperparameter explanation

[6]https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Configuration-File.md

#### 2.3.2 Network Settings

See Table 2 for network settings explanation.

| Setting | Description |
|---|---|
| Hidden Units | Number of units in the hidden layers of the neural network. Correspond to how many units are in each fully connected layer of the neural network. |
| Layers | The number of hidden layers in the neural network. Corresponds to how many hidden layers are present after the observation input, or after the CNN encoding of the visual observation. |

Table 2: Network Settings explanation

#### 2.3.3 Self-Play

See Table 3 for self-play settings explanation.

| Setting | Description |
|---|---|
| Team Change | Number of steps between switching the learning team. This is the number of trainer steps the teams associated with a specific ghost trainer will train before a different team becomes the new learning team. |
| Swap Steps | Number of ghost steps (not trainer steps) between swapping the opponents policy with a different snapshot. A "ghost step" refers to a step taken by an agent that is following a fixed policy and not learning. |
| Play Against Latest Model Ratio | Probability an agent will play against the latest opponent policy. |

Table 3: Self-Play settings explanation

# 3 Experiments

## 3.1 The Environment

The environment for these experiments is a football mini-game that consists of 4 agents in total, 2 on each team, trying to push a ball into the opposing team goal, see Figure 2. The agents are using competitive self-play to play against each other and train in that way. The experiments will all use the same hyperparameters but will change in reward structure.

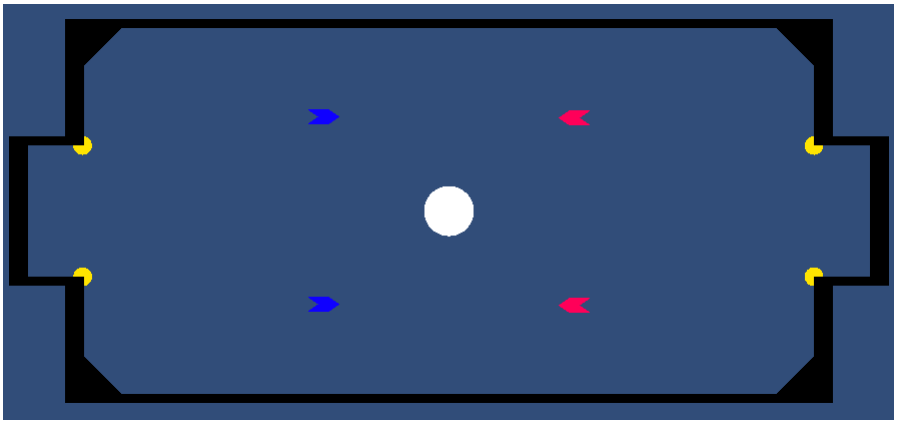

Figure 2: Screenshot of a starting position

### 3.1.1 Environment Details

There are two teams competing against each other. One is Blue (left) the other is Red (Right). The teams are trying to score a goal by pushing the ball to the enemy goal. The episode will end either after 4500 steps or a goal.

Goal Conditions:

- Blue team scores if the ball's X coordinate is $\geq 30$
- Red team scores if the ball's X coordinate is $\leq -30$

For arena bounds see Table 4.

| Coordinates | X | Y |
|---|---|---|
| Min | -32.5 | -13.1 |
| Max | 32.5 | 13.1 |

Table 4: Arena Bounds

### 3.1.2 Agent Details

Agent uses discrete action space. The Action Vector consists of the tree shown in Figure 3.

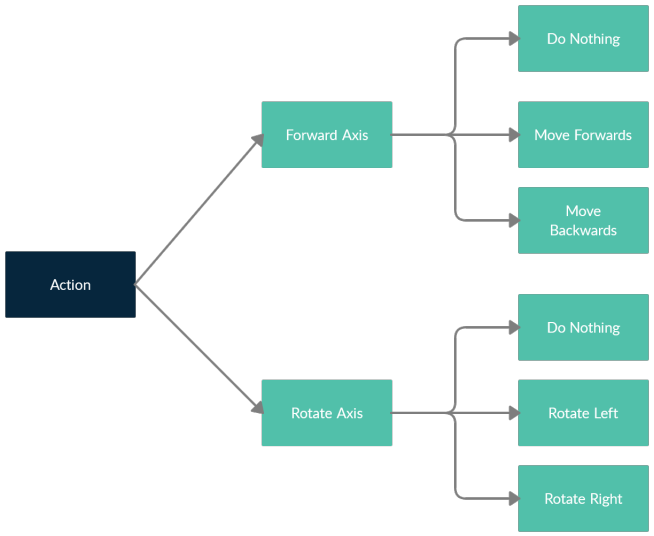


Figure 3: Action Vector Tree

The agent also uses Observation Vector that is of size 25. The agent observes the following data:

- Their own position
- Their own rotation
- Their own velocity
- Their own angular velocity
- Teammates position
- Teammates rotation

- Teammates velocity
- All of the opponents position
- All of the opponents rotation
- All of the opponents velocity
- Balls position
- Balls velocity

The data is normalized before being observed.

## 3.2 Hyperparameters

See Table 5 for hyperparameters. Same hyperparameters were used for all experiments.

| Algorithm | PPO |
|---|---|
| Hyperparameters | |
| Batch Size | 256 |
| Buffer Size | 8192 |
| Learning Rate | 0.003 |
| Beta | 0.005 |
| epsilon | 0.2 |
| lambda | 0.95 |
| Epoch Number | 6 |
| Learning Rate Schedule | Constant |
| Network Settings | |
| Hidden Units | 256 |
| Layers | 2 |
| Self Play | |
| Team Change | 250000 |
| Swap Steps | 10000 |
| Play Against Latest Model Ratio | 0.5 |

Table 5: Configuration

## 3.3 Results Explanation

The result subsection of experiments will contain the Cumulative Reward and Episode Length of the training process. Due to the fact that there are 2 teams playing against each other the Episode length might be a better indication than Cumulative Reward of the agents performance.

## 3.4 Experiment 1

For the first experiment I've created a simple reward system that would reward the team that scored and penalize the other one. This is considered good practice for reinforcement learning, however it may not always converge.

### 3.4.1 Reward table

See Table 6 for the reward table for this experiment.

| Reward | Team Left | Team Right |
|---|---|---|
| Score Left | 1 | -1 |
| Score Right | -1 | 1 |

Table 6: Reward structure table

### 3.4.2 Results

Data for Cumulative Reward and Episode Length for the first $3 \cdot 10^7$ steps, see Figure 4 and Figure 5.

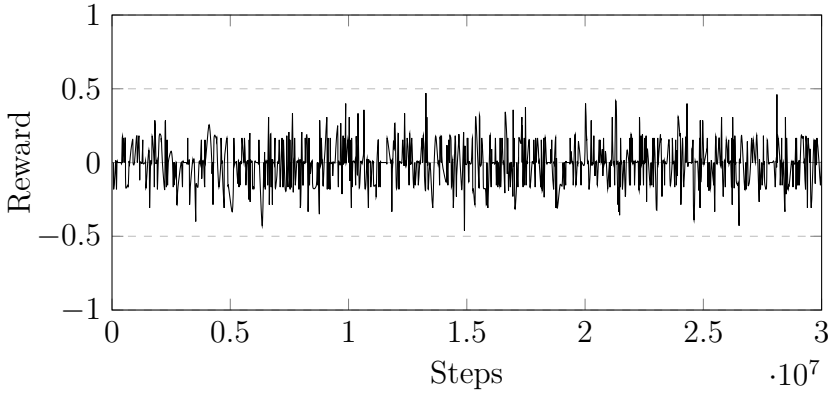


Figure 4: Cumulative Reward

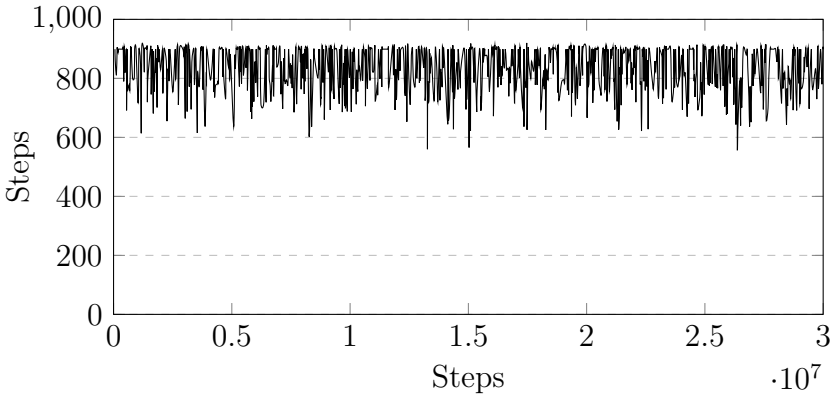


Figure 5: Episode Length

#### 3.4.3 Experiment Conclusion

As we can see from the Cumulative Reward and Episode Length graphs, the agents aren't converging towards anything and this is evident when we use this model: it moves and rotates randomly.

## 3.5 Experiment 2

For the second experiment I've modified the reward system so that the agents would also be rewarded for touching the ball.

### 3.5.1 Reward table

See Table 7 for reward table for this experiment.

| Reward | Team Left | Team Right |
|---|---|---|
| Score Left | 1 | -1 |
| Score Right | -1 | 1 |
| Ball touch | 0.15 | 0.15 |

Table 7: Reward structure table

### 3.5.2 Results

Data for Cumulative Reward and Episode Length for the first $3 \cdot 10^7$ steps, see Figure 6 and Figure 7.

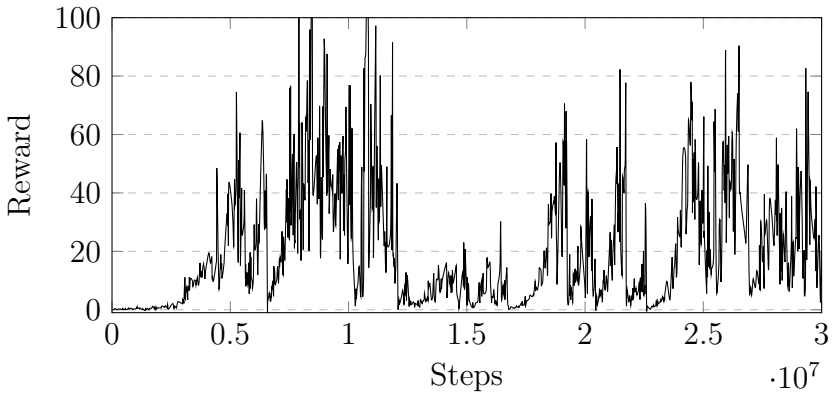


Figure 6: Cumulative Reward

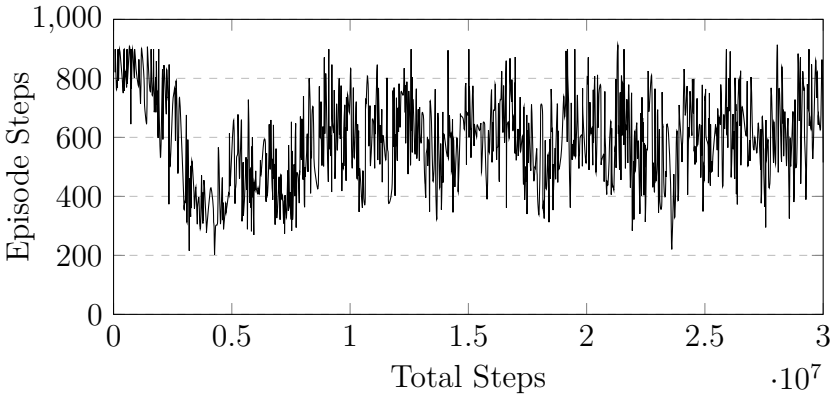


Figure 7: Episode Length

#### 3.5.3 Experiment Conclusion

The agents started converging very quickly. They also learnt that that they can maximize the reward by pushing the ball into a corner and keep colliding with it until the episode ends. They sometimes do score some goals but it's probably accidental.

## 3.6 Experiment 3

For the third experiment I've modified the reward system again, so that the agent will only get rewarded for touching the ball if the absolute Y coordinate of the ball would be less than 10. Also, on every step, the agents will now receive passive rewards or penalties according to the ball's position.

### 3.6.1 Reward table

See Table 8 for reward table for this experiment.

| Reward | Team Left | Team Right |
|---|---|---|
| Score Left | 1 | -1 |
| Score Right | -1 | 1 |
| Ball touch | 0.15 | 0.15 |
| Ball position X >= 0 | $\frac{0.005}{Distance(BallPos,RightGoalPos)}$ | $-\frac{0.005}{Distance(BallPos,RightGoalPos)}$ |
| Ball position X <= 0 | $-\frac{0.005}{Distance(BallPos,LeftGoalPos)}$ | $\frac{0.005}{Distance(BallPos,LeftGoalPos)}$ |

Table 8: Reward structure table

Note: agents only get reward for touching the ball if BallPos.Y <= 10

### 3.6.2 Results

This is the data for Cumulative Reward and Episode Length for the first $5 \cdot 10^7$ steps, see Figure 8 and Figure 9.

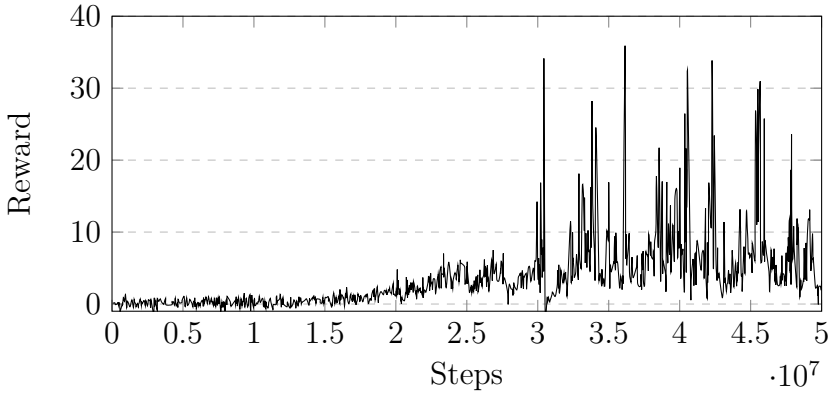


Figure 8: Cumulative Reward

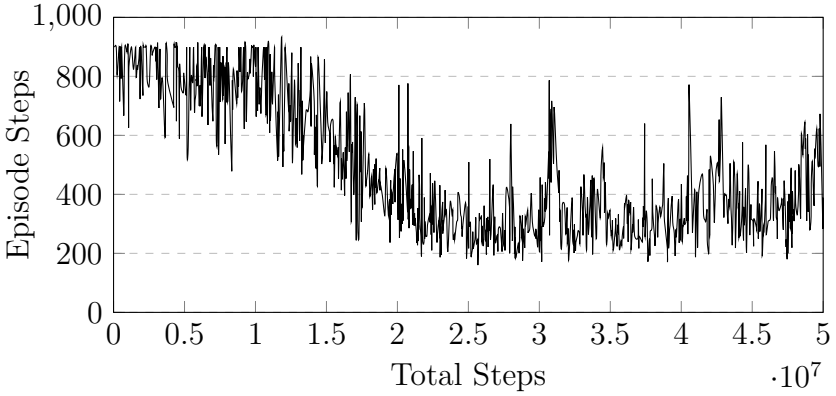


Figure 9: Episode Length

The training started to converge at around $1.5 \cdot 10^7$ steps. The spikes in the reward might be due to the fact that both teams are trying to push the ball in the opposite direction, thus continually getting the reward for touching the ball.

### 3.6.3 Experiment Conclusion

Looking at both, cumulative rewards and episode lengths, the agent started to converge at around $1.5 \cdot 10^7$ steps and the training could have been stopped at around $2.5 \cdot 10^7$ - $2.7 \cdot 10^7$ steps. After that point, agents started overfitting and exploiting the reward system.

# 4 Conclusion

- In this specific scenario, reward structure needs to be more complex. It is not enough to have a single reward for scoring a goal.
- Adding more rewards helps agents converge on the correct policy.
- The reward system created in experiment 2 was flawed, therefore the agents didn't converge on the correct policy for the environment, but converged on the policy that exploited the rewards.
- Adding more restrictions to rewards and passive rewards allowed agents to converge on policy that would play out the minigame that was setup for them.

# References


[BPS+18] Trapit Bansal, Jakub Pachocki, Szymon Sidor, Ilya Sutskever, and Igor Mordatch. Emergent complexity via multi-agent competition, 2018.

[HZAL18] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

[KSH20] Anssi Kanervisto, Christian Scheller, and Ville Hautamäki. Action space shaping in deep reinforcement learning, 2020.

[MSG99] D. E. Moriarty, A. C. Schultz, and J. J. Grefenstette. Evolutionary algorithms for reinforcement learning. Journal of Artificial Intelligence Research, 11:241–276, Sep 1999.

[SB18] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction. The MIT Press, second edition, 2018.

[SWD+] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms.