Training intelligent tennis adversaries using self-play with ML agents

by

Bakhtiyar Ospanov

Submitted to the Department of Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

at the

NAZARBAYEV UNIVERSITY

April 2021

Author .....

Department of Computer Science
Apr 11, 2021

Certified by .....

M. Fatih Demirci
Associate Professor
Thesis Supervisor

Accepted by .....

Vassilios D. Tourassis
Dean, School of Engineering and Digital Sciences

Training intelligent tennis adversaries using self-play with ML agents

by

Bakhtiyar Ospanov

Submitted to the Department of Computer Science
on Apr 11, 2021, in partial fulfillment of the
requirements for the degree of
Master of Science in Computer Science

Abstract

In the game and robotics industries, the design of intelligent and interactive characters can be greatly enriched by advances in artificial intelligence. This approach is in need due to the biased and pre-programmed, and thus limited, nature of conventional algorithms. In contrast, machine learning approaches can educate these characters to have creative and independent behavior even in complex games. This study explores the possibility of training intelligent adversaries using self-play in a game of tennis, which is not yet competently researched. The agent is provided with the basic tennis rules and informed about the outcome (winning/losing). Given that, it is up to the agent to find out suitable behavior. The agents are placed into a visually, physically, and cognitively rich surrounding environment implemented in Unity. Reinforcement learning with proximal policy optimization is used to train one brain for two adversarial agents. The training is stabilized using incremental complexity modification based on curriculum learning. By having itself as a level matching opponent, the agent consistently improves its skills. The Elo rating system helps to quantitatively assess the performance by computing the relative skill level between two agents in a zero-sum game. The liberty in behavior during the training process has opened the possibility for agents to discover robust tactics that helped them to play on the same level as human opponents. Later, an agent's brain can be employed for researching, benchmarking, and using as virtual non-playable characters.

Thesis Supervisor: M. Fatih Demirci
Title: Associate Professor

Acknowledgments

Thanks to Prof. Demirci for continuous support and weekly meetings, Albina Li and Makar Lezhnikov for insightful comments, and my family for patience.

Contents

List of Figures

Chapter 1

Introduction

Machine Learning (ML) algorithms have been introduced to multiple industries over the last decade and are projected to play an essential role in improving them. A simulation of the real world is one of such complex domains that may greatly benefit from ML. Though the design of a simulation environment is one difficult aspect, it is more challenging to create logical interactions and meaningful behaviors among objects that resemble realistic ones. Those must be done relevant to the context and rather based on experience than hard-coded rules. Such results will create a high fidelity simulation of real-world which can later boost robotic research (e.g. training a robotic hand to solve Rubik's cube [20]) and entertainment industries (e.g. outperforming average human in Doom game [16]). For instance, playing against an intelligent, creative, and context-aware opponent will make almost any game more entertaining and engaging. So, this study is trying to explore the possibility of training intelligent agents using self-play in a game of tennis. This idea belongs to the set of emerging and unresearched experiments. Besides, the ability to produce solid results will help to support Juliani A. [15] claim of game engines being suited to be a general platform for simulation environments, machine learning research, and more. The most feasible application is the ability to employ an agent's brain for creating an opponent player in games with a customizable level of difficulty and play style. Regarding the implementation, an environment is designed in the state-of-the-art Unity Real-Time

Development Platform[1] which provides an accurate simulation of the physical world and offers convenient machine learning tools. Those are accessed via Python API to implement Reinforcement Learning (RL) training for the agent's brain. Using the same brain, two ML-Agents control rackets to hit a ball to the opposite side of the net according to tennis rules[2]. The players should hit the ball in a way that another player could not hit it back. The ability to train such a model in a realistic environment will satisfy the research goal of this project. Though replicating an exact court area with proper physical properties and training intelligent behavior require meticulous setup and tuning due to complex physical interactions within a large action space, it is important to introduce such an approach in the game industry which otherwise will be stalled by explicitly programmed algorithms. This work contributes to the domain by providing an environment, training setup and trained model. The comprehensive simulation environment has a physically and visually rich context and can be easily adapted to serve as a training platform for similar sports games, such as table tennis, badminton, pickleball, etc. The training setup can be used to research observation complexity in games, the influence of physical parameters, logical interactions (strategies), and social interactions (friendly/aggressive playstyle). The resulted model can be served as a baseline performance model for future research in the area as well as used in a simple production application.

Firstly, this study considers a machine learning area and its approaches, introduces deep reinforcement learning and its components. Later, types of policies for RL are discussed with the introduction of Proximal Policy Optimization (PPO) used for this project and alternatives. The Curriculum Learning approach is briefly explained as it is a helpful technique in improving performance and speeding up the training process. An environment creation and training tool, Unity and ML-Agents Toolkit, are presented with a description of their features and advantages. Secondly, major methodologies for this project are outlined, including the description of physical world setup and learning environment interactions. Following that, training configu-

[1] https://unity.com/

[2] http://protennistips.net/tennis-rules/

ration is examined with feature selection and hyperparameter setup explanation. The experiments and results section outlines statistical observations during the training and comparative analysis of runs with various configurations. The resulted model is used in the test application to evaluate the performance of the ML agent and compare it with user experience. Finally, limitations of the study and several methods for improving the models are discussed.

Chapter 2

Related Work

2.0.1 Machine Learning

Machine learning is an instrument for extracting knowledge from data. Formally, it is a large subfield of Artificial Intelligence (AI) in which a machine can predict an outcome based on given data without being explicitly programmed [2]. It is achieved by selecting actions that lead to correct solutions which drive adjustments in an action-decision algorithm. This approach can be applied in a wide range of disciplines, especially those with high complexity. A simulation of real-world behavior is one of these and a game industry diligently tries to achieve it. This industry can greatly benefit from the machine learning approach by providing a more diversified experience to users. Though not every approach can be gainfully employed to solve existing issues in the simulation area. Machine learning is well-developed by extensive research with time-proved and widely-used approaches. The underlying concept of each is driven by input data, optimization, and target predictions. ML can be classified into three major types with their own taxonomy: supervised learning, unsupervised learning, and reinforcement learning [3]. The first one requires a great deal of correctly labeled data and is frequently used for classification and regression [6]. The unsupervised one does clustering and association with no use of labels and prior categorization [8]. The last one uses trial-and-error agents which explore and interact with an environment. An agent tries to maximize a reward that is given in case of correct performance

and minimize a penalty for improper behavior. By adapting to a reward signal, an agent improves knowledge about an environment and its rules, and as a result, it can predict the next optimal action. This experience is later expressed in the form of a single model, called policy [26]. Given the complexity of the simulation world, the last approach, reinforcement learning, is the most suitable to create an agent with dynamic behavior. There are great examples of how AI has disrupted the game industry [18], including RL agents that play chess, shogi and Go [24], more general video games [28] (including 3D multiplayer games [14]), and game research frameworks [17].

Reinforcement learning was greatly enhanced by the introduction of deep learning into the approach. Deep Learning (DL) has in its core neural networks with diverse layer structure for learning from extensive unstructured data [12]. The collection of methodologies suggested by deep learning is an efficient way to handle unpredictable environments with a high influx of data for extracting meaningful patterns. DL allows to recognize a problem but not yet make a decision. So, by integrating these neural networks into RL, a high-performance technique, Deep Reinforcement Learning (DRL), is designed.

DRL is about making decisions within an uncertain dynamic environment though with defined action space. An Agent is a single-purpose entity for deciding on the action to solve the problem in the given environment. The responder to these decisions is an environment that creates consequences by changing the agent's observation, reward, or own state. A state is an instance of all the possible parameters that an environment posses in the given time. The limited version of this state, information that an agent perceives, is referred to as an observation. Given this observation, an agent decides on the action from action space - a pool of actions granted by an environment within each state. In response to this action, an environment sends a signal to an agent, called reward. An agent processes this feedback as described previously. This cycle of observing an environment and adapting decision-making based on reward signal repeats until optimal mapping (policy) is not achieved [12].

The most valuable goal of RL is to form an optimal policy. It is a function that maps observations to actions and is generated using a neural network for approxima-

tion in this scenario. It is defined in terms of Markov Decision Process represented in the form of <S,A,P,R> with S - state of the environment accessed by an agent (observation), A - action space, P - the probability of moving between states and R - reward function. The data collected by iterative interactions with the environment in such a manner are called experiences. These are later fed into a neural network to train a policy. There are mainly two types of approaches in defining a policy: on-policy and off-policy. As for the first one, the same policy is used for both improving during the training and for an agent to select the next action. An agent must interact with the environment to collect experiences that are provided to create and refine the latest available policy [25]. In contrast, off-policy learning needs a separate policy for improving and another one for selecting a next action [19]. An agent collects experiences that are stored in a replay buffer. Later, this past data is used to calculate and update the policy. One of the possible off-policy methods that can be adopted for this project is state-of-the-art Soft Actor-Critic (SAC) [10]. Unfortunately, the behavior of the opponent is continuously modified which creates unfavorable dynamics for the replay buffer as noticed by [7]. Though off-policy is sample efficient (since no new samples are collected when a policy is updated), the on-policy approach is selected for this project due to its more reliable and faster convergence.

One of the most widely used on-policy algorithms is Proximal Policy Optimization by OpenAI [23]. This policy can be easily maintained and adjusted while having a relatively solid performance. It makes use of policy gradient algorithms as those are efficiently utilized for control problems, games, and simulations. Though their performance is greatly affected by the selection of hyperparameters and requires a substantial amount of collected data. To overcome these issues, the OpenAI team initially introduced Trust Region Policy Optimization (TRPO) [22], and following that, a simplified and generalized version, called PPO, was introduced. It demonstrated supremacy over other on-policy gradient algorithms while being balanced. This advantage is achieved by using a stochastic gradient ascent in updating a trust-region and eliminating the Kulllback-Liebler divergence penalty. Overall, this algorithm is most suitable for this project as it belongs to the continuous domain problems where

similar projects, as 3D locomotion [11], Atari [23] and more, have demonstrated high performance using it.

Though PPO is a state-of-the-art approach, an environment can have complex interactions which an agent may not be able to guess. The exploration starts with random actions which may lead to receiving a spare reward signal if a target action requires an understanding of multiple concepts about an environment. And this project requires a complex action to consistently exploit a reward. So, the Curriculum Learning concept [4] is introduced to alleviate the agent's random space exploration. It is one of the methodologies adopted for machine learning training where the difficulty of the task is progressively incremented so that an agent is optimally challenged. Both studies [4] and [9] demonstrated a noticeable decrease in convergence time (up to half-time) and improved quality of local minima. In the case of using ML-Agents Toolkit [15], environment parameters may be introduced and adjusted during the lessons. A curriculum consists of series of lessons that are triggered by selected completion criteria. Each criterion should have a threshold to determine the end of the lesson for the selected measure (e.g. cumulative reward or step progress). A minimum lesson length and signal smoothing can also be defined. The current lesson can be tracked in TensorBoard during the training. All in all, an optimal generalization can be achieved by fine-tuning curriculum parameters.

2.0.2 Unity

Reinforcement learning provides a great performance given an environment that supplies informative and realistic observations for an agent. The environment design requires an intuitive and highly customizable tool to simulate real-world concepts and test researchers' hypotheses One of the world-leading game engines, Unity, positions itself as an eco-system that provides a universal real-time platform with elaborated physics and comprehensive usability to satisfy research needs [15]. The application of those research outputs can be found in engineering, entertainment, customer service, and more, which later, appear in the form of educational simulators, mobile or VR software with multi-platform support. Unity Editor is used in developing a

project which comprises user-provided assets files, referred to as 'Assets'. An environment is created within a 'Scene' by allocating a new 'GameObject' or 'Prefab' in the hierarchy. A GameObject is a basic logical and physical entity in the scene. The 'Component' can be attached to a GameObject to define its behavior, visual appearance (e.g. Mesh), physical properties (e.g. RigidBody, BoxCollider), or custom functions (C# script). Prefab is configured version of a collection of GameObject with saved components which is stored as an asset file and can be reused. Unity supports external and supplementary packages that can greatly enhance its functionality. Unity ML-Agent Toolkit is one of those which provides an open-source package with several widely-used algorithms based on Pytorch, including reinforcement and imitation learning, that are provided to train intelligent agents using environment interaction. A more detailed explanation of Unity capabilities can be found in their comprehensive documentation[1].

[1]https://docs.unity3d.com/Manual/index.html

Chapter 3

Methodology

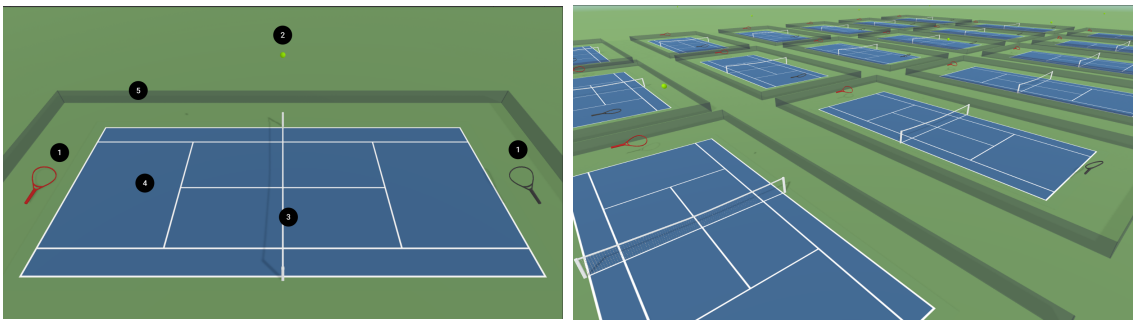
To satisfy the research goal of the project and train an intelligent tennis player, an environment with rich complexity should be built. Following that, logical behaviors and interactions are created along with the training environment. The practical usage of this training in the test application is also described in this section.

3.0.1 Environment Design

Physical environment

An environment should have a high-fidelity resemblance to the real world both in terms of physical properties and visual representation to successfully train an agent. As mentioned before, Unity allows the construction of a high-quality environment with the ability to introduce run-time adjustable parameters. Firstly, a prefab of Tennis Area was created and served as a main container for the ball, rackets, net, scenery, etc 3-1a. This container allowed to design independent instance with own local coordinate system for positioning and tracking agents and their dependent components. This instancing has facilitated the training process by creating 18 independent duplicates of these areas 3-1b.

Each of these prefabs contained two agents which are represented graphically as rackets (these two words are used interchangeably in this work). The collider component was attached to track collisions with a ball, net, surface, and borders. It consists



(a) Tennis Area prefab: 1 - agents, 2 - ball, 3 - net, 4 - court area, 5 - borders

(b) Training setup: 18 instances of the tennis area

Figure 3-1: Environment setup

of two 'Box Collider'-s which cover a racket's head and handle while being optimal in calculating collision physics instead of resource-consuming mesh colliders. The physic material was created for rackets to handle collision interaction with other colliders. It had 0.45 dynamic friction, 0.5 static friction, and 0.2 bounciness. The 'Minimum' friction combine and 'Maximum' bounciness combine modes were used to calculate values when two bodies collide. These parameters were selected experimentally based on visual trials on behavior interactions. The 'RigidBody' component was used to simulate the physical properties of real-world objects. The mass was set to 350 grams which is an average racket weight. The angular drag represents the impact of air resistance while rotating (where zero means no resistance) and it was set to 0.05. The computationally-intensive 'Continuous' collision detection was used for precise calculations due to fast-moving interactions. Finally, this component helped to simulate the realistic movement of an agent by controlling velocity in all three dimensions. In terms of velocity and inertia constraints, no axes were frozen for position while x, y-axes were limited for rotation due to the specificity of training configuration. The dimensions of rackets: 1.37m length, 0.57m head width, 0.021m thickness.

A ball is another essential entity in the environment setup. It has a similar set of components attached. The 'Sphere Collider' with the following physic material properties: 0.2 dynamic friction, 0.1 static friction, 0.8 bounciness, and the same combine modes as for the racket. The mass of the ball is 60 grams as claimed to be average weight and the rest of the 'RigidBody' properties are similar to the one of

the racket. The radius of the ball is 0.2 meters.

The scenery consists of supplementary objects that build a realistic tennis environment and help to track interactions according to tennis rules. The floor defines sectors for valid points (i.e. in/out areas) and also participate in ball rebound from the surface. Its physic material has 0.4 dynamic friction, 0.4 static friction, 0.1 bounciness, and both 'Average' friction and bounciness combine modes. The court dimensions are 23x8 meters. The net separates a court on two identical sides and contains a collider above to track the side change of the ball. Finally, the walls surround the whole court area to restrict rackets' movement beyond reasonable position.

Overall, the project settings have the default values which include 9.834 m/s^2 for gravity, time scale during the gameplay is 1 while for training process is 10 to speed up the training process, and fixed delta time 0.02 second for physics updates.

Learning Environment

The collection of observations and inference are managed by the Academy class. There is one instance that controls all the agents' decisions during both training and gaming. The Academy is also responsible for communicating with Python API to transfer observed data and train a neural network.

The simulation process comprises episodes in which agents try to win a point. The episode goes on until one of the termination conditions is satisfied. In this scenario, either of the agents wins a point or times out (the number of maximum steps is reached). Each episode starts with re-initializing some parts of the environment setup by calling 'OnEpisodeBegin()'. It resets the position and rotation of the rackets and zeros out their velocities. Their initial positions are randomized along court baseline to expose agents to diverse conditions which they may face later.

The next step is to observe the environment by collecting data that may have an impact on the potential decision. The training process requires data to be sent in feature vector format, so the neural network can operate on it. This information is collected by adding observation values to 'VectorSensor' in the 'CollectObservations' method. The space size of the observation vector is also explicitly declared in behavior

parameters. The feature selection was done by analytical estimation of the importance of each potential parameter.

Based on observed information, the model produces the next action for each agent depending on its policy decision. The space size of the action vector is also stated in behavior parameters. The 'Continuous' action space type is required for a given setup as agents constantly move around the court. The decision-making process is accompanied by allocation rewards for taken actions as intended in reinforcement learning. These rewards are used both during training and simulation to determine the optimality of selected actions by the neural network. Finally, 'OnEpisodeEnd()' is called once a winning point is allocated towards either agent and the whole cycle for a new episode starts again.

3.0.2 Training Configuration

As all static parameters were introduced above, this section describes dynamic variables that were experimented with during the training. Regarding an observation vector, it contains 14 floating values, namely:

- p_x^a, p_y^a, p_z^a - position of an agent
- v_x^a, v_y^a, v_z^a - velocity of an agent
- r_z^a - rotation of an agent
- p_x^b, p_y^b, p_z^b - position of a ball
- v_x^b, v_y^b, v_z^b - velocity of a ball
- estimatedSectorId

All of the values are normalized within the [-1,1] range as it helps an algorithm to converge faster in neural networks. Instead of introducing a complex recurrent neural network architecture, stacking is used to augment current observations with previous steps' data. This enlarged observation imitates a short "memory" to make data more context-aware. The optimal size of stacked vectors is three for this project.

Using provided input information, the network produces an output action vector of size 4. The values from this action buffer clamped in range [-1,1] and used to control v_x^a, v_y^a, v_z^a and r_z^a of a racket. As for the rewards allocation system, 0.5 is given to a winner of a point within a game and -0.5 is taken from a loser. To guide an agent towards reasonable behavior faster, a small 0.0005 reward is given when a racket touches a ball and 0.001 is added if the ball goes over a net and hits an opponent's floor. While to encourage understanding of tennis rules and more consecutive wins, $\pm$ 0.5 is added for winning/losing a game point and $\pm$ 1.0 is for winning/losing a set point.

This configuration setup wraps up the Unity side of the training, while the actual training is executed externally via the ML-Agent Toolkit python package. The package uses a .yaml configuration file with 27 hyperparameters for this project. Based on selected training algorithms, hyperparameter tuning can greatly affect agents' ability to come up with optimal policy while accomplishing a goal.

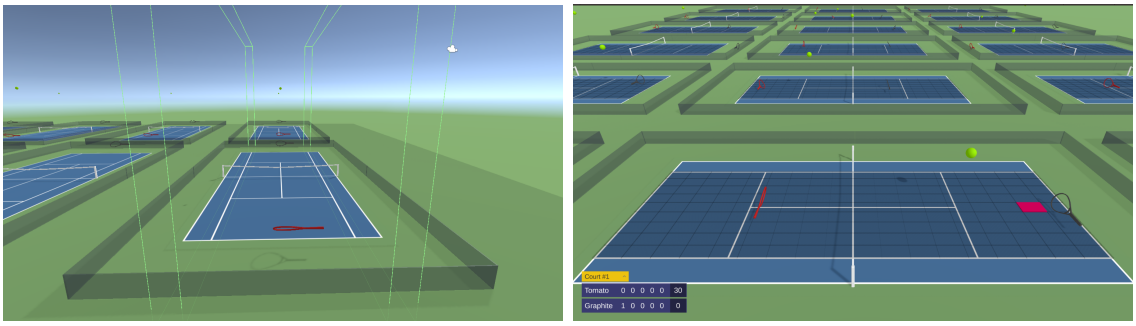
As described before, the main policy in use is PPO for this project as it supports self-play training mode and allows the introduction of imitating learning, GAIL, etc. later. For the trainer, the following hyperparameters were selected based on the best practice guideline and tuning experiments: batch_size = 2048, buffer_size = 20480, learning_rate = 0.0002 (constant), beta = .003, epsilon = 0.15, lambd = 0.93, num_epoch = 4. The neural network contains 256 hidden units and consists of two layers with applied normalization. For the reward signal only extrinsic reward configuration is defined with gamma .96 and strength 1.0 (no change of reward). This gamma allows controlling the importance of future rewards that an agent can get, so it should perform well in the present and expect to receive rewards further in the future.

There are special adjustments introduced for self-play setup. The learning team plays with a ghost trainer for 100000 steps, and then, the other team becomes a new learning team. This number of steps leads to a longer game with the same opponent and playing against it for a large amount of time increases the chances of winning over. On other hand, it is not long enough to adapt to a specific strategy and helps to

stay away from overfitting. The probability of an agent playing against the opponent with the most recent policy is 50%, otherwise, a model from previous iterations is taken. Each iteration an agent revises its policy and the same holds for the opponent. The window size is 10, i.e. this number of previous policy snapshots is available in the pool of opponents. The pool has diverse enough behaviors to expose a learning agent to difficult challenges and design a generalized policy though the training process will take more time.

The proper agent behavior cannot be achieved straightforwardly with targeted physical parameters. The real-life sizes of a racket and a ball are small relative to the court area. It means an immense search area for an agent, so the reward signal is too sparse and not received most of the time. An agent with a small racket's surface area should at least touch a ball which can be located anywhere on a large court. Given a random policy as an initial strategy, it was not possible to achieve. Instead, curriculum learning was applied to progressively increase the difficulty of the task. The training was initiated with enlarged versions of rackets, upscaled by 25%, ball scale by 150%, and frozen velocity along the z-axis (v_z^a). Later, every 200,000 steps the scales were decreased proportionally until the target values. So, the agent was able to exhibit proper behavior with given scale parameters. After 1 million steps the velocity was freed in all axes through with limiting walls on both sides of the z-axis (Fig. 3-2a). The distance between walls, i.e. available region for the agent to move, was increasing every 200,000 steps as a part of curriculum learning as well.

The 'estimatedSectorId' parameter is introduced to decrease search space for an agent. Once an opponent hits a ball, an agent estimates where the ball is going to land on his side. Though instead of feeding the precise location of the calculated position, each side of the court is divided on a 9x9 grid, and a corresponding sector index is used for observation. A sample landing sector prediction is illustrated in Figure 3-2b.



(a) Limitings walls along court sidelines. (b) Landing sector prediction.

Figure 3-2: The proposed measures to decrease a search space.

3.0.3 Test application

The trained model is used in a simple game of tennis built in Unity. The neural network is processed by Unity Inference Engine which can run it both on CPU and GPU. The game is built for the WebGL platform though it can be executed on a diverse range of devices, The application is hosted on a website[1] for user testing. The environment is similar to the one used during the training. Though for gaming mode, the velocity of an agent along the z-axis is limited for user convenience. It is challenging for a user to spatially evaluate the position of the ball in relation to its own position along all three axes simultaneously. Thus, the camera view is set up in a way to be able to understand at least a height and a distance to the net properly. The height range for a user to control is imperceptible in relation to the court scale. So, instead of controlling it, a user can rotate a racket along the z-axis and move it along the x-axis with keyboard input. The movement of an agent is not driven by a change of its position but rather the velocity is applied in a selected direction. It implies the natural behavior, including a little initial delay due to slow and steadily accelerating start. These conditions are identical for both parties. In addition to limited action space, a trajectory line of a ball movement is displayed as a hint for a user which is not available to the opposing ML agent. The scoring system is based on actual tennis rules with a 5-sets match. A user can also switch between ML-Agent and User modes to either observe two ML agents playing or take control over one of the rackets. Once

[1] https://bakhtiyar-ospanov.github.io/MLAT/index.html

users get acquainted with gameplay and played several games against the ML agent, they can proceed with a survey which is later discussed in the next section.

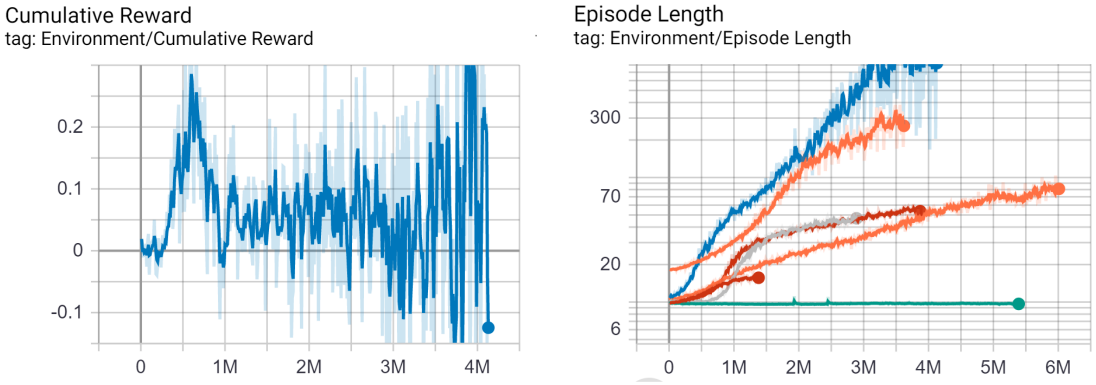
Chapter 4

Experiments and Results

The statistics during the training process and experiments were recorded by ML-Agents Toolkit and tracked via TensorBoard[1]. TensorBoard provides tools for visualizing and observing custom metrics. A graph depicts each independent training run with selected metric over overall step count. A cumulative reward is the default metric that is usually used in reinforcement learning to demonstrate success and expected to lean towards 1. Though this is not the case for this training as self-play is involved. As depicted in Figure 4-1a, a reward signal fluctuates slightly over zero at the beginning, and as the training progresses, the amplitude increases and shifts closer to zero. It happens due to +1/-1 reward each game while having an opponent with the most recent policy most of the time (defined by play_against_latest_model_ratio = 0.5). As policy improves, a pool of opponents contains mostly strong policies which lead to more frequent switches between win/lose endings. Thus, a cumulative reward is not a meaningful metric for a such case. The mean length of the episode (Figure 4-1b) is somehow helpful as it shows the ability of agents to hit a ball and balance it with each other. However, a longer episode length does not imply a good performance but rather means not aggressive playstyle with fewer winning shots. Another important metric from policy statistics is entropy (Fig. 4-2a) which tracks the randomness of agent's decisions. It gradually decreases as the training continues which indicates a well-selected beta hyperparameter. Beta is the rate of entropy regularization that

[1]https://www.tensorflow.org/tensorboard

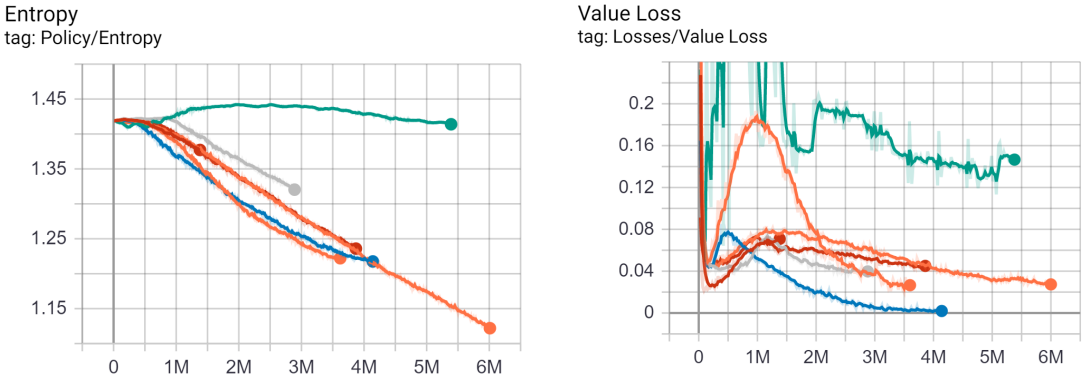
encourages agents to randomly research the action space. Regarding loss functions, a value loss (Fig. 4-2b) is tracked to evaluate the prediction of each state value produced by the model, The graphs start with a rise as an agent explores space and learns, followed by the progressive decline as receiving a reward is stabilized.



(a) Sample Cumulative Reward over a test run.

(b) Episodes Lengths for the various configuration runs.

Figure 4-1: Tracking of Environment metrics



(a) Entropy values for the various configuration runs.

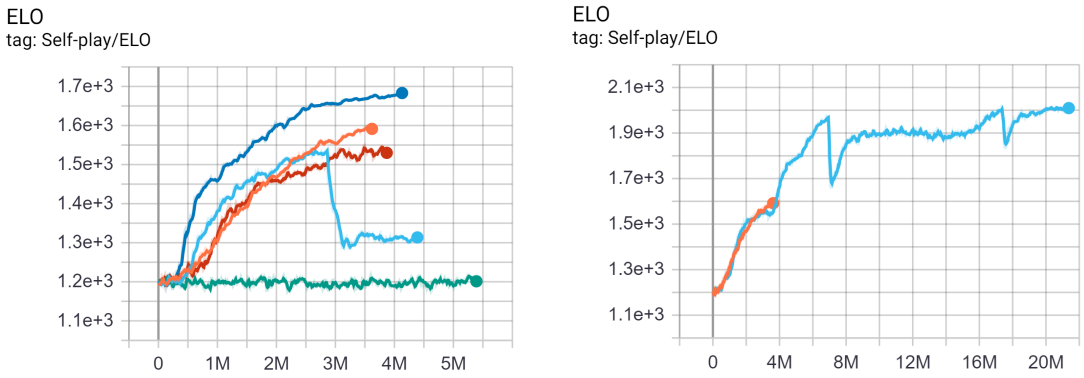
(b) Value Losses for the various configuration runs.

Figure 4-2: Tracking of Policy and Losses metrics

The primary metric for evaluating the success of the training is specific to self-play games and called the Elo rating system [1], This approach allows to evaluate relative proficiency levels of two participants in a zero-sum game. The rating is calculated based on the current value and match outcome (win/loss/draw). If the highly skilled player wins, a few points will be given as a victory was expected, and a

few points will be deducted from the lower-skilled player as a defeat was forecasted. In contrast, more points are awarded to the lower-rated player in case of victory over the higher-rated player which is penalized with more points respectively. If a game is ended with a draw, a case will be considered in the same way as a higher rated player defeat though with proportionally lowered penalty/award points. The initial Elo rating score is 1200. The Figure 4-3a demonstrates Elo rating for several runs with various configurations. The orange line represents the initial setup with locked v_x^a, enlarged version of rackets and ball, and basic reward signal with +1/-1 for winning/losing a game. An agent performed well in this setup and was able to balance a ball continuously. Thus, this performance was used as a baseline. The blue line shows the performance of the improved reward signal version. Additional rewards were given for touching, getting a ball over the net, winning a whole match, etc. as described in the Training Configuration section. There was a noticeable performance increase in terms of Elo rating and training speed. Once receiving a reward signal was stabilized, the physical properties were adjusted to correlate with real-world dimensions as mentioned previously. An agent performed poorly and was not able to hit a ball since a rapid decrease of racket surface and ball size created a larger search space. Curriculum learning was introduced at this stage and its influence can be observed in the Figure 4-3b in comparison with the baseline. There are sharp falls that correspond to lesson changes as the agent needed time to adapt to the modified environment. A significant increase was achieved in terms of Elo score, but at the same time, a step count was increased multiple times. After all, the agent's ability to perform in the targeted scenario was prioritized over training time. Following that, v_x^a was unlocked and the agent was able to move in all three dimensions freely. This freedom substantially enlarged a search space and required more complex interactions from the agent. The agent was rarely able to touch a ball yet not hit it properly, as illustrated by the flat green line in Figure 4-3a. To overcome this issue, an enlarged version of items was returned and limiting walls were installed on both sides of the agent along the z-axis to shrink a search space. The distance between walls was part of curriculum learning and increased over time. The light blue line outlines the

performance in this setup. An agent performed well up until the certain point when walls were not far apart, and after, a search space became unmanageable again. To further guide the agent towards the target action, the 'estimatedSectorId' parameter was introduced to indirectly hint at a potential landing position of the ball (described in Training Configuration). The red line indicates the benefit of using this approach where steep downfall at the end was replaced with continuous increase. However, enlarged versions of items were used as well as limiting walls which were located at court sidelines.



(a) Elo ratings for the various configuration runs.

(b) Elo rating for the Curriculum Learning run.

Figure 4-3: Tracking of Elo rating metric

Overall, there are two successfully trained models as the output of the experiments. The first one is with reliable performance and realistically-sized rackets and ball though with locked v_x^a. The second one simulates freedom that the agent would have in the real world but having an enlarged version of primary objects and unstable behavior. There are several ways how to further improve this model which will be discussed in the next section. As for the first model, it was used to build the test application for users to play with and evaluate the agent's performance.

The test application was hosted on the web platform and contained a questionnaire. The participants of the survey were asked to play the game as they needed to get acquainted with the gameplay, interaction physics, and rules. The total number of responses was 13. In the questionnaire, the participants were presented with two

sample video clips, each 1-minute duration. One of them contained a play of an expert user against the ML agent, another one was with two ML agents playing against each other. The task was to distinguish between two videos and indicate how the difference was spotted. The statistical data from the survey is available in Appendix A. Generally, 38.5% of participants were able to spot the behavior difference and 30.8% confused ML agent with an expert user and wrongly answered this question (Fig.A-1a). The last 30.8% of the responders could not notice enough variance in the two behaviors and see no substantial difference. To sum up, the participants who mistakenly answered and who could not distinguish the difference can be grouped as these answers imply a high resemblance of two behaviors. It means the majority of the participants believe that an intelligent agent can simulate expert user behavior with high fidelity. This claim is also supported by the statistics from the next question (Fig.A-1b). It asked responders to evaluate resemblance according to 1 (not similar) - 5 (highly similar) scale. The majority gave 4 (53.8%) and 3 (38.5%) points with an average of 3.46 points which indicates the satisfactory performance of the ML agent. Additionally, the survey asked to separately assess the performance of the player from Video#1 (the expert user, Fig.A-1c) and from Video#2 (the ML agent, Fig.A-1d). The expert user received 3.23 points and 3.39 points were given to the ML agent. Though these scores are not that high, most importantly, they are equal in being moderately average. Overall, this survey demonstrated that the ML agent behavior can simulate the expert user's one to the extent that the responders could not tell the difference or confused them.

Chapter 5

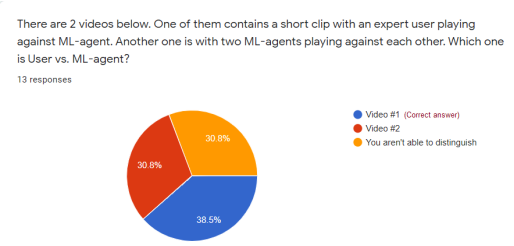
Conclusion

This study proposes a simulation environment with rich visual and physical context to train intelligent agents for a game of tennis. The environment is created with Unity Real-Time Development Platform which made a possible precise simulation of physical interaction, graphical representation, and logical behavior of real-world objects. The well-established environment facilitated the collection of accurate and versatile observations during the training. These data were delivered to the training site using the bridge by ML-Agents Toolkit. This package provided tools to conduct training using reinforcement learning and formulate policy. There are two satisfactory models which are produced as the output of this study. The first one is used in the production as overviewed above and the second one is still a subject of research. The last model is work-in-progress due to imposed limitations by the environment and selected algorithms. The tennis environment requires complex interactions from the ML agent, including the ability to hit a ball from any point within a court area. It implies complete freedom in movement and rotation along all three axes. The last policy is trained with these unlocked properties but with a modified environment as described before. Otherwise, the algorithm could not handle such an immense search space with a sparse reward signal. The combination of reinforcement and curriculum learning is not enough for the agent to learn complex interactions with extended conditions. One way of possible improvements is to introduce imitation learning into this combination. It requires recorded demonstrations from an expert

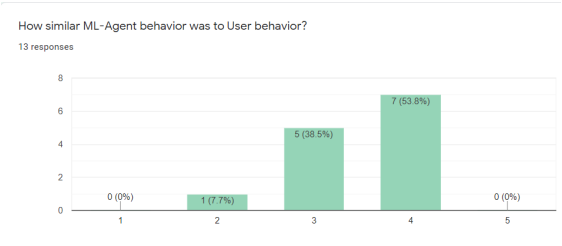
user which can be done with the existing environment and toolkit. If the resulted set of demonstrations contains a limited number of possible states, it is advised to use Generative Adversarial Imitation Learning [13], otherwise, Behavioral Cloning [27] can be utilized. In such complex environments with sparse rewards, it is also recommended to introduce intrinsic rewards, such as Curiosity [21] reward signal and Random Network Distillation [5]. All of these approaches can be used separately as well as integrated simultaneously. There is another potential direction in which the current setup can be readjusted for the new experiments. The doubles (paired) version of tennis can be trained. It implies introducing two more ML agents and extending the court area. The currently trained models will fail to handle this scenario as the collision of two agents from the same team most likely to occur. Though the given environment setup and tools will suffice to experiment with this idea. Overall, without taking into consideration possible improvements for presented limitations, it was possible to use Unity as a general simulation platform for the research purpose. It has resulted in training an intelligent tennis agent which exhibits satisfactory behavior and was used as a non-playable opponent in the test application.

Appendix A

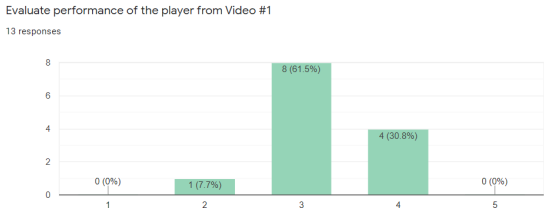
Figures



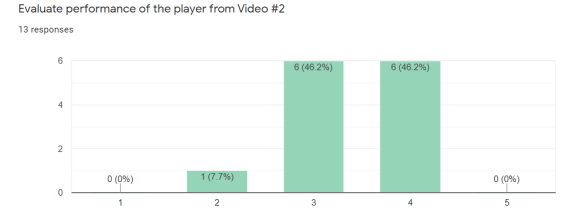
(a) Video clips differentiation results



(b) Behaviour resemblance evaluation



(c) Expert user performance evaluation.



(d) ML agent performance evaluation.

Figure A-1: The performance questionnaire results.

Bibliography

[1] Paul CH Albers and Han de Vries. Elo-rating as a tool in the sequential estimation of dominance strengths. *Animal Behaviour*, pages 489–495, 2001.

[2] Jafar Alzubi, Anand Nayyar, and Akshi Kumar. Machine learning from theory to algorithms: an overview. In *Journal of physics: conference series*, volume 1142, page 012012. IOP Publishing, 2018.

[3] Taiwo Oladipupo Ayodele. Types of machine learning algorithms. *New advances in machine learning*, 3:19–48, 2010.

[4] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.

[5] Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. *arXiv preprint arXiv:1810.12894*, 2018.

[6] Rich Caruana and Alexandru Niculescu-Mizil. An empirical comparison of supervised learning algorithms. In *Proceedings of the 23rd international conference on Machine learning*, pages 161–168, 2006.

[7] Jakob Foerster, Nantas Nardelli, Gregory Farquhar, Triantafyllos Afouras, Philip HS Torr, Pushmeet Kohli, and Shimon Whiteson. Stabilising experience replay for deep multi-agent reinforcement learning. In *International conference on machine learning*, pages 1146–1155. PMLR, 2017.

[8] Zoubin Ghahramani. Unsupervised learning. In *Summer School on Machine Learning*, pages 72–112. Springer, 2003.

[9] Alex Graves, Marc G Bellemare, Jacob Menick, Remi Munos, and Koray Kavukcuoglu. Automated curriculum learning for neural networks. In *international conference on machine learning*, pages 1311–1320. PMLR, 2017.

[10] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1861–1870. PMLR, 2018.

[11] Nicolas Heess, Dhruva TB, Srinivasan Sriram, Jay Lemmon, Josh Merel, Greg Wayne, Yuval Tassa, Tom Erez, Ziyu Wang, SM Eslami, et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017.

[12] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[13] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. *arXiv preprint arXiv:1606.03476*, 2016.

[14] Max Jaderberg, Wojciech M Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio Garcia Castaneda, Charles Beattie, Neil C Rabinowitz, Ari S Morcos, Avraham Ruderman, et al. Human-level performance in 3d multiplayer games with population-based reinforcement learning. *Science*, 364(6443):859–865, 2019.

[15] Arthur Juliani, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar, et al. Unity: A general platform for intelligent agents. *arXiv preprint arXiv:1809.02627*, 2018.

[16] Guillaume Lample and Devendra Singh Chaplot. Playing fps games with deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[17] Marc Lanctot, Edward Lockhart, Jean-Baptiste Lespiau, Vinicius Zambaldi, Satyaki Upadhyay, Julien Pérolat, Sriram Srinivasan, Finbarr Timbers, Karl Tuyls, Shayegan Omidshafiei, et al. Openspiel: A framework for reinforcement learning in games. *arXiv preprint arXiv:1908.09453*, 2019.

[18] Ian Millington and John Funge. *Artificial intelligence for games*. CRC Press, 2009.

[19] Rémi Munos, Tom Stepleton, Anna Harutyunyan, and Marc G Bellemare. Safe and efficient off-policy reinforcement learning. *arXiv preprint arXiv:1606.02647*, 2016.

[20] I Akkaya OpenAI, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 10, 2019.

[21] Deepak Pathak, Pulkit Agrawal, Alexei A Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning*, pages 2778–2787. PMLR, 2017.

[22] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015.

[23] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[24] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[25] Satinder Singh, Tommi Jaakkola, Michael L Littman, and Csaba Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine learning*, 38(3):287–308, 2000.

[26] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis lectures on artificial intelligence and machine learning*, 4(1):1–103, 2010.

[27] Faraz Torabi, Garrett Warnell, and Peter Stone. Behavioral cloning from observation. *arXiv preprint arXiv:1805.01954*, 2018.

[28] Ruben Rodriguez Torrado, Philip Bontrager, Julian Togelius, Jialin Liu, and Diego Perez-Liebana. Deep reinforcement learning for general video game ai. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2018.