Rhine-Waal University of Applied Sciences

Faculty Communication and Environment

Prof. Dr. Frank Zimmer

M. Sc. Vincent-Pierre Berges

Concurrent Discrete and Continuous Actions in Deep Reinforcement Learning

Master Thesis

by

Marco Pleines

Rhine-Waal University of Applied Sciences

Faculty Communication and Environment

Prof. Dr. Frank Zimmer

M. Sc. Vincent-Pierre Berges, Unity Technolgies

Concurrent Discrete and Continuous Actions in Deep Reinforcement Learning

A Thesis Submitted in

Partial Fulfillment of the

Requirements of the Degree of

Master of Science

in

Information Engineering and Computer Science

by

Marco Pleines

Spehstraße 74

47551 Bedburg-Hau

Matriculation Number:

12117

Submission Date:

November 19th, 2018

Abstract

The availability of concurrent action spaces in *Deep Reinforcement Learning* has many uses. For example, concurrently executed actions enable agents to learn richer and more complex behaviors, like driving a car, moving a robot arm, or playing video games. Especially for training agents to play video games like humans, it is beneficial to implement action spaces, which comprise discrete and continuous actions. This is useful to mimic the behavior of a computer mouse, where its movement is continuous and its clicks are discrete. Therefore this work investigates three approaches and introduces two novel environments to examine the application of concurrent discrete and continuous actions in video games.

One approach implements a threshold to discretize a continuous action, while another one divides a continuous action into multiple discrete actions (bucket approach). The third approach creates a multiagent to combine both action kinds. These approaches are benchmarked by two environments. In the first environment (*Shooting Birds*) the goal of the agent is to accurately shoot birds by controlling a cross-hair. The second environment is a simplification of the game *Beastly Rivals Onslaught*, where the agent is in charge of its controlled character's survival.

The outcome of the experiments show that all approaches successfully train the agents to reach their goals. Unstable results were achieved by the threshold approach. The multiagent scored the best, but took the longest to train. Overall, the bucket approach is recommended, because it is more stable and trains faster than the multiagent. Further observations, such as a biased locomotion of the resulted agent behaviors, are discussed. Future projects can investigate the examined approaches in combination with visual observations. Also, more complex environments can be developed based on this thesis' findings.

Key words: Artificial Intelligence, Deep Reinforcement Learning, Concurrent Actions, Continuous Actions, Discrete Actions

Table of Contents

List of Abbreviations

A3C	Asynchronous Advantage Actor-Critic
AI	Artificial Intelligence
BDQ	Branching Dueling Q-Network
BRO	Beastly Rivals Onslaught
DDPG	Deep Deterministic Policy Gradient
DDQN	Dueling Double Deep Q-Network
DL	Deep Learning
DotA	Defense of the Ancients 2
DRL	Deep Reinforcement Learning
FPS	First Person Shooter
MC	Monte Carlo
MDP	Markov Decision Process
ML	Machine Learning
neural network	Artificial Neural Network
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
SB	Shooting Birds
TD	Temporal Difference Learning
TRPO	Trust Region Policy Optimization

List of Figures

List of Tables

List of Code Listings

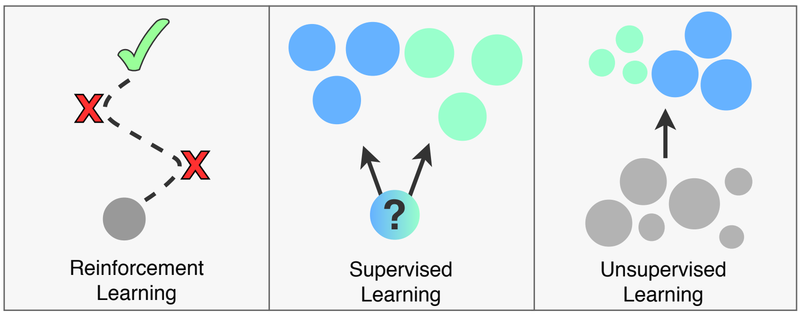


Figure 1.1: These are three major categories of machine learning. Reinforcement learning is concerned with sequential decision making, where an agent interacts with its environment to reach some goal in a trial-and-error manner. In supervised learning, a single decision is made to choose a class for a data record based on expert knowledge (i.e. training dataset). Unsupervised learning techniques derive patterns from datasets (e.g. clustering).

1 Introduction

Reinforcement Learning (RL), next to *Supervised* and *Unsupervised Learning*, is one of three major topics in the field of *Machine Learning* (ML) (Figure 1.1). It allows an agent to autonomously learn a policy, that tells the agent how to behave in a given state, by interacting with its environment in a trial-and-error manner. Based on the agent's behavior, the environment signals rewards to reinforce good behaviors and to diminish bad ones. In contrast to supervised learning, RL does not require expert knowledge nor is meant to classify data. Neither is its goal to find patterns in data like in unsupervised learning. With the successes of *Deep Learning* (DL), *Artificial Neural Networks* (*neural networks* for short) opened up new opportunities in the field of RL. Thus, *Deep Reinforcement Learning* (DRL) was introduced [Mnih et al., 2013]. This enabled agents to solve several games on a superhuman-like level, for example *Atari 2600* games [Mnih et al., 2015] and the board game *Go* [Silver et al., 2017]. Most recently, multiple agents outplayed human amateurs in the video game *Defense of the Ancients 2* (DotA) [OpenAI, 2018b], which has much more complex tasks to solve.

Nowadays, DRL is broadly researched on many aspects. One particular facet deals with action spaces. In this thesis, concurrent action spaces are examined, which eventually allow agents to make use of discrete and continuous actions concurrently. The motivation behind the emergence of this topic is established subsequently.

1.1 Motivation and Problem Statement

Multiple problems accompany RL research. Computational complexity and sample efficiency are some more general problems derived from DL. However, RL has its own challenging distinctivenesses. Environments may be partially observable. Rewards can be sparse and delayed. A strong temporal correlation might exist within sequences of actions [Arulkumaran et al., 2017]. Determining what action led to a positive outcome (credit assignment problem) is an issue as well [Arulkumaran et al., 2017]. The *exploration versus exploitation* balance issue is concerned with whether the agent shall exploit its learned policy or explore new strategies to potentially improve its policy [Yannakakis and Togelius, 2018].

Next to these obstacles, there are further challenging topics, which shall enable richer and more complex behaviors trained with RL. For instance, attention shall be given to agents' action spaces, which is the final context of this thesis. Concurrent action spaces are desirable for many complex environments and high-dimensional action spaces. For some first-person-shooter (FPS) video games, an eligible strategy is to run, strafe, and shoot concurrently [Harmer et al., 2018]. A similar motivation can be considered for autonomous driving, because a car usually has to slow down while steering through a curve. Moreover, if a robot arm had to rotate its joints one after the other, it would take much more time to accomplish its job (e.g. picking up and moving items). Such essential and rich behaviors are not possible when each action is carried out sequentially.

This context can be expanded to concurrent action spaces comprising discrete and continuous actions. Mimicking input devices like a computer mouse or a gamepad cannot be done by simply using a discrete or continuous action space alone. Providing a solution for this issue is of particular interest for video games, because most games are played by utilizing the aforementioned input devices. In FPS video games, players move the mouse or the thumbstick of a gamepad to change their view. This sort of action is continuous as it allows infinite directions and velocities as input. While moving these devices, players also need to press buttons concurrently to trigger an event such as shooting a gun, which is a discrete action.

Nonetheless, concurrent actions can draw restrictions into account. It should be considered that certain actions might conflict upon combination. For example, reloading and shooting a gun cannot be executed concurrently.

In the end, the availability of concurrent discrete and continuous actions shall lead to more feasible use-cases and more complex agent behaviors.

1.2 Objectives

The overall goal of this thesis is to utilize state-of-the-art DRL technology to demonstrate potential solutions for applying concurrent discrete and continuous actions in the context

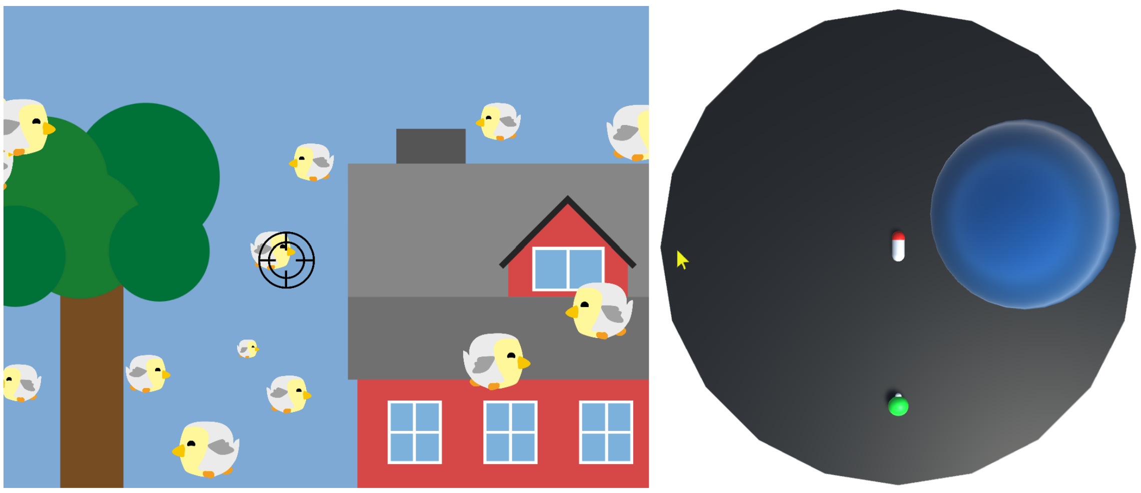


Figure 1.2: The environments Shooting Birds (left) and the simplification of Beastly Rivals Onslaught (right)

of video games. For this purpose, two novel environments, a simplified version of *Beastly Rivals Onslaught*[1] (BRO) and *Shooting Birds* (SB) (Figure 1.2), are developed and implemented. These environments feature action spaces, which allow the agent to control a mouse cursor to play typical video games. All accomplished contributions can be seen as work towards agent behaviors, which are capable of generally playing video games just like humans (i.e. controlling keyboard, mouse and gamepad input devices). Ultimately, consecutive projects shall benefit from the provided experience to enable the development of agents, which can play many more video game environments such as *MOBA* (Multi-Online Battle Arena) like games (e.g. the full game of BRO as potential next milestone). However, these goals come with constraints. Working on as complex environments as DotA is not feasible due to limited resources concerning time and computational power. Furthermore, the agents' observation space will not feature image input due to longer training times and thus longer iterations on the development of the introduced environments. It is out of scope to develop a novel algorithm, which natively supports the to-be-examined action space type. Also, it is not specifically intended to analyze the generalization capabilities of the resulting agent behaviors.

1.3 Approach and Overview

To achieve the set goals, this thesis starts out by building up necessary fundamental knowledge. It is intended to gather all the basic RL ingredients to understand the to-be-employed DRL algorithm *Proximal Policy Optimization* (PPO) [Schulman et al., 2017]. PPO is currently one of the most promising state-of-the-art algorithms, because of its

[1]Video of the game BRO `https://youtu.be/OTcDd7a_R0A`

successes and uses [Juliani et al., 2018, Bansal et al., 2017, OpenAI, 2018b, Schulman et al., 2017]. Also, relevant related work is briefly reviewed to support the taken approaches and to assist final discussions. After that, the main part commences by elaborating three approaches (Section 4):

- discretizing continuous actions using a threshold
- breaking down continuous actions into multiple discrete actions (bucket approach)
- multiagent setup to combine both action spaces

Previously gained knowledge and described contributions from related work and Unity's ML-Agents Toolkit [Juliani et al., 2018], are the key components to carry out the implementation and the setup of the proposed environments. Resulting insights are then thoroughly discussed and put into context to ultimately conclude with an outlook, which comprises future steps, suggestions, and ideas.

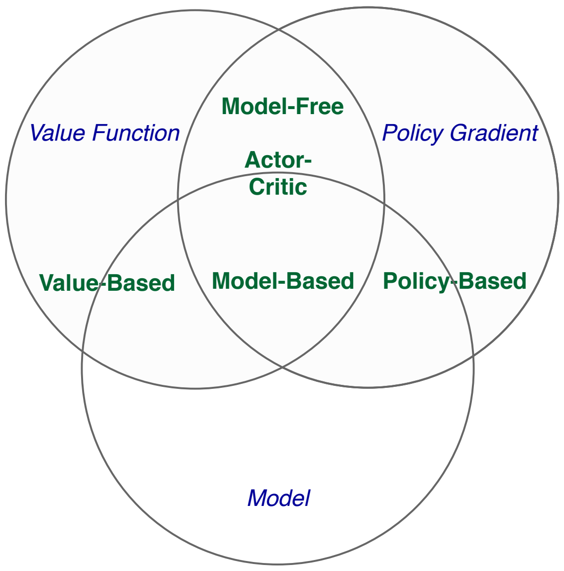


Figure 2.1: Three general areas (blue) of RL algorithms [Silver, 2015]. Each area's method can exist on their own, or can be combined to more complex approaches (e.g. actor-critic methods emerge from value- and policy-based methods.)

2 Fundamental Knowledge

A precondition for achieving the set goals is to completely comprehend the fundamental components of the DRL algorithm PPO. This requires starting with the basics of RL and to continue with value functions and policy gradients. Before elaborating PPO, its field of methods, actor-critic methods, is introduced. The notation is mostly based on Sutton and Barto's book: *Reinforcement Learning: An Introduction* (2nd edition, 2018).

2.1 Reinforcement Learning Basics

Value functions, policy gradients, and model-based approaches are three apparent areas in RL, which may overlap (Figure 2.1), for implementing agents. The value of an environment's state (expected return) is estimated by a value function (Section 2.1.3). Policy gradients do not estimate a value, instead they directly map states to actions. Methods, which build a model of the behavior of the environment, are model-based.

Due to focusing on PPO, actor-critic methods (coupling value functions with policy gradients) are considered in this thesis. The subsections from 2.1.1 to 2.1.7 intend to provide all RL ingredients, which are fundamental to PPO. This starts out by formalizing the *Markov decision process* (MDP) and explaining what is meant by the expected return. After that, value functions are introduced. Subsection 2.1.4 differentiates between episodic and continuing RL approaches, which is followed by distinguishing off-policy methods from on-policy ones. Policy gradients are detailed next. The last section is about actor-critic methods.

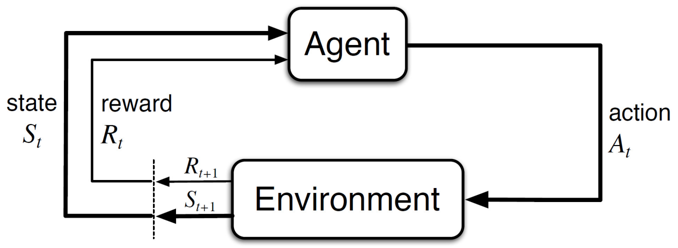


Figure 2.2: The agent-environment interaction in a Markov decision process [Sutton and Barto, 2018, p. 48]

2.1.1 Markov Decision Process

In general, RL comprises learning through interaction. An autonomous agent interacts with its environment by sensing its state S_t and taking an action A_t at time step t, which causes the environment to transition to a new state S_{t+1}. The agent then learns to modify its behavior based on a reward R_t signaled by the environment as consequence of S_{t-1} and A_{t-1}. Its goal "is to learn a policy π (control strategy) that maximizes the expected return (cumulative, discounted reward)" [Arulkumaran et al., 2017]. Based on the present state, the policy tells the agent what action to take in order to maximize the expected return in its environment.

Overall, the previously given description can be illustrated as an MDP as seen in Figure 2.2. The MDP, which is a "classical formalization of sequential decision making" [Sutton and Barto, 2018], comprises:

- a state s
- an action a,
- a reward r,
- a number of states S,
- a number of actions A,
- transitions $\mathcal{T}(S_{t+1}|S_t, A_t)$,
- a reward function $\mathcal{R}(S_t, A_t, S_{t+1})$,
- and a discount rate $\gamma \in [0, 1]$.

2.1.2 Expected Return

RL is centered on rewards and therefore it is sometimes described as *reward-driven behavior*. The definition of a reward can be found in Sutton and Barto's *reward hypothesis*:

"That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward)" [Sutton and Barto, 2018, p. 53].

The previously mentioned expected return, in its easiest form, is the sum of all rewards for a certain number of timesteps T:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T, \tag{1}$$

where R_t is the received scalar reward by the agent at timestep t. Environments have to be distinguished between *episodic* and *continuing tasks.* Episodic tasks terminate to terminal states and cause the environment to reset, whereas continuing tasks do not terminate. They keep running, which raises the issue of evergrowing cumulated returns. To avoid an infinite return, the return is *discounted* using the *discount rate* γ [Sutton and Barto, 2018]:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{2}$$

where γ is a parameter, which is selected from the range [0, 1]. This way, a high discount rate puts focus on long-term rewards, while low rates target more immediate rewards.

2.1.3 Value Functions

Estimating *value functions* is an essential part for many RL algorithms [Sutton and Barto, 2018]. These functions estimate "*how good* it is for the agent to be in a given state" [Sutton and Barto, 2018, p. 58]. This kind of utility of a state is determined by the expected return. Based on a policy π, the value function $v_\pi(s)$ of a state s is the expected return when starting in s and following π henceforth. Sutton and Barto (2018) formally define the *state-value function*:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right]. \tag{3}$$

As the agent acts according to its policy π at time step t, $v_\pi(s)$ expresses the expected value of the underlying state s, while following policy π.

A similar definition is considered for the so-called *action-value function*:

$$q_\pi(s,a) \doteq \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right], \tag{4}$$

where $q_\pi(s,a)$ denotes the expected return by starting in state s and carrying out action a, while following the policy π.

Some RL algorithms utilize the so-called *advantage function* $\hat{A}(s,a)$. Instead of taking the whole action-value $Q(s,a)$ into account[2], the advantage function subtracts an estimate of the state's value $V(s)$ from the action-value [Wang and Yu, 2016]:

$$\hat{A}(s,a) = Q(s,a) - V(s). \tag{5}$$

The advantage describes how much better the value of a state turned out to be in comparison to what was expected by the value function. An alternative to the action-value is the use of the discounted return as an estimation of the action-value.

2.1.4 Monte Carlo vs Temporal Difference Learning

Like mentioned in subsection 2.1.2, there are episodic and continuing tasks in RL. For instance, *Monte Carlo* methods are a typical example for episodic RL tasks [Yannakakis and Togelius, 2018]. These methods wait until a whole episode of the environment is simulated. The gathered experience, which comprises the discounted cumulative reward G_t, is then used to update the value function $V(S_t)$:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)], \tag{6}$$

where α is a step-size parameter (i.e. learning rate) [Sutton and Barto, 2018, p. 119].

In comparison to the just mentioned episodic method, *Temporal Difference Learning* (TD) updates the value function immediately at step $t+1$. This way, the currently observed reward R_{t+1} and the current estimate of the value function $V(S_{t+1})$ are used to update the value function straight away [Sutton and Barto, 2018, p. 120]:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]. \tag{7}$$

$[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$ is the so called TD error [Li, 2017]. α denotes the learning

[2]The state-value function $V(s)$ and the action-value function $Q(s,a)$ are denoted with capitals, because they are being learned. The lower-case ones represent the final functions. However, this distinction is not present in the underlying literature. Based on a discussion among AI researchers and hobbyists, the meaning of this notation is assumed.

rate. To sum up, G_t is the target of episodic methods, whereas $R_{t+1}+\gamma V(S_{t+1})$ is targeted by continuing methods.

2.1.5 On-Policy vs Off-Policy

On-Policy and *Off-Policy* algorithms are differentiated in RL. The distinctive difference between on-policy and off-policy algorithms emerges in the way the policy is updated. For on-policy algorithms, the agent chooses its actions itself and therefore follows and updates its own policy online. The same policy is updated over and over again, where as for off-policy algorithms, the agent learns an optimal policy, which can be updated offline upon a pool of experiences sampled from uncorrelated policies.

2.1.6 Policy Gradients

Instead of selecting actions based on learned action-value estimates, policy gradient algorithms utilize a *parameterized policy* [Sutton and Barto, 2018, p. 321]. Hence, the policy π selects an action a based on the state s and the policy's parameter vector θ:

$$\pi(a|s,\theta) = \Pr\{A_t = a|S_t = s, \theta_t = \theta\}, \ \theta \in \mathbb{R}^{d'}. \tag{8}$$

Pr represents the probability of taking an action over the present state and θ at time t. d' denotes the dimensionality of the policy parameter vector. An example for θ is the weights and biases of a neural network.

The general learning process comprises some scalar performance measure $J(\theta)$, which takes in the policy parameter. Maximizing the *performance* is the goal of policy gradient methods. This is approached by gradient *ascent* in J [Sutton and Barto, 2018]:

$$\theta_{t+1} = \theta_t + \alpha\widehat{\nabla J(\theta_t)}, \tag{9}$$

where the gradient of the performance measure, in conjunction with θ_t, is approximated by the expectation of the stochastic estimate $\widehat{\nabla J(\theta_t)}$ [Sutton and Barto, 2018]. α indicates the learning rate. Updating the gradients is challenging, because the performance relies on the selected actions and the distributions of the states, while both of these impact the policy parameter [Sutton and Barto, 2018, p. 324]. To overcome this issue, the *policy gradient theorem* [Sutton and Barto, 2018, p. 326] is defined and proven:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a)\nabla\pi(a|s,\theta). \tag{10}$$

$\mu_(s)$ is a distribution of states, which is sampled from the current policy [Sutton and Barto, 2018]. Using the performance measure this way, the derivative of the state distribution is not involved [Sutton and Barto, 2018]. All in all, this is the basis for many policy gradient algorithms like REINFORCE [Sutton and Barto, 2018].

2.1.7 Actor-Critic Methods

Policy gradient methods that additionally learn an approximation of a value function are called actor-critic methods [Sutton and Barto, 2018]. The learned policy is considered as actor, whereas the critic is usually represented by a value function. The critic measures how well the taken action performed (value-based) and helps the agent to improve its policy-based behavior. Learning an approximation of the value function is defined with a weight vector w [Sutton and Barto, 2018]:

$$\hat{v}(s, w), \ \ w \in \mathbb{R}^d. \tag{11}$$

In some cases, the approximators for the policy gradient and the value function share a set of parameters.

2.2 Proximal Policy Optimization

Schulman et al. (2017) introduced a new kind of policy gradient methods. These methods are called *Proximal Policy Optimization* (PPO). PPO emerged from the desire to create an RL algorithm, which features higher scalability (larger models and parallelized setups), data efficiency and robustness (e.g. less prone to hyperparameter selection) while easing the complexity [Schulman et al., 2017]. The preceding subsections describe the major components of PPO, which make PPO outperform other policy gradient approaches such as *Trust Region Policy Optimization* (TRPO) [Schulman et al., 2017].

2.2.1 Clipped Surrogate Objective

A typical objective function for policy gradients is formalized by [Schulman et al., 2017]

$$L^{PG}(\theta) = \hat{\mathbb{E}}_t[log\pi_\theta(A_t|S_t)\hat{A}_t]. \tag{12}$$

The stochastic policy is denoted by π_θ. $\hat{A}_t$ is an estimator of the advantage function at timestep t. Optimizing L^{PG} may cause destructively large policy updates [Schulman et al., 2017]. To solve this problem, Schulman et al. (2015) developed the TRPO algorithm, which implements a "surrogate" objective with a constraint or penalty to bound the policy updates. The maximization of this surrogate,

$$L^{CPI}(\theta) = \hat{\mathbb{E}}_t\left[\frac{\pi_\theta(A_t, S_t)}{\pi_{\theta\text{old}}(A_t, S_t)}\hat{A}_t\right] = \hat{\mathbb{E}}_t[r_t(\theta)\hat{A}_t], \tag{13}$$

is considered by PPO as well, where $r_t(\theta)$ represents the probability ratio $\frac{\pi_\theta(A_t,S_t)}{\pi_{\theta\text{old}}(A_t,S_t)}$. Instead of computing a constraint or penalty, the policy updates are limited by simply clipping these [Schulman et al., 2017]:

$$L^{CLIP} = \hat{\mathbb{E}}_t[min(r_t(\theta)\hat{A}_t,\ clip(r_t(\theta),\ 1-\epsilon,\ 1+\epsilon)\hat{A}_t)]. \tag{14}$$

In this equation, ϵ is a hyperparameter (e.g. $\epsilon = 0.2$). This clipping ensures that too large advantages do not overshoot the to-be-optimized objective. Other policy gradient algorithms may be suitable to adapt the clipping logic [Schulman et al., 2017].

2.2.2 Minibatch Updates

An example for the PPO algorithm is published as an actor-critic variant, which utilizes K epochs of minibatch updates. If both, the value and policy approximator, share common parameters of a neural net, the policy surrogate and the value function error should be combined [Schulman et al., 2017]. Another addition to the ultimately used objective is an entropy bonus, which encourages exploration. Each iteration, the following combined objective is (approximately) maximized [Schulman et al., 2017]:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 E[\pi_\theta](S_t)]. \tag{15}$$

c_1 and c_2 are coefficients, the entropy bonus is denoted by E and L_t^{VF} represents the squared-error loss of the value function [Schulman et al., 2017]. To reduce variance, the showcased algorithm incorporates the approach of generalized advantage estimation [Schulman et al., 2015b].

The subsequent (Listing 1) pseudo-code describes the flow of PPO that makes use of fixed-length trajectories. During each iteration, N (parallel) actors generate T timesteps of experience tuples using their old policy $\pi_{\theta_{old}}$. Besides that, the advantage estimates $\hat{A}_1, ..., \hat{A}_T$ are calculated. After each actor completed its trajectory, the aforementioned surrogate objective is optimized using minibatches for K epochs [Schulman et al., 2017]. Possible optimizers are *stochastic gradient descent* and *Adam*.

```
for iteration=1,2,...
        for actor=1,2,...,N
        Use old policy for T timesteps
        Compute advantage estimates
    Optimize surrogate objective with K epochs and minibatches
    Replace old policy by the updated one
```

Code Listing 1: Actor-critic variant of the PPO algorithm [Schulman et al., 2017]

3 Related Work

Varying work focused on action spaces in RL recently. Sharma et al. (2017) introduced the method of *Factored Action Space Representations*, which exploits the relationship among multi-actions (i.e. combination of single actions as one discrete action) to allow the agent to learn about the effect of multi-actions' individual sub-actions. Besides that, research on probability distributions for continuous action control is apparent. Chou et al. (2017) conducted research on using the Beta distribution instead of using the Gaussian one. Another group implemented a clipped Gaussian distribution [Fujita and Maeda, 2018]. Both approaches have shown to improve training performance for multiple, commonly used environments.

However, these scientific contributions are not concerned with concurrent action spaces explicitly. The most relevant related work, which has significant impact throughout this thesis, comprises OpenAI's DotA 2 achievements, the approach of action branching by Tavakoli et al. (2017) and the successes of using concurrent discrete actions in imitation learning and DRL [Harmer et al., 2018]. These three essential publications are elaborated further during the next subsections.

3.1 OpenAI Five

The non-profit AI research company, OpenAI, has been working on developing DRL agents for DotA 2. So far, they have published several blog posts about their results. Initially, they started out with games of DotA where two agents competitively faced each other [OpenAI, 2017]. As consecutive work, *OpenAI Five* was made public featuring 5 vs 5 matches [OpenAI, 2018b]. Their work is of significance, because they have to deal with the challenge of developing suitable action spaces for their agents to solve the game, which normally requires mouse and keyboard input devices. This and further challenges, while providing an overview of OpenAI's general approach and their results, are detailed next.

3.1.1 Challenges

Complex game rules, partially-observed states, long time horizons, high-dimensional continuous action, and observation spaces make DotA to a much greater challenge than the board games Go and chess [OpenAI, 2018b]. Motivation for taking on this challenge of developing agents for DotA is drawn from the rich representation of video games, which "start to capture the messiness and continuous nature of the real world" [OpenAI, 2018b]. An average game of DotA lasts for 45 minutes at 30 frames per second leading to overall

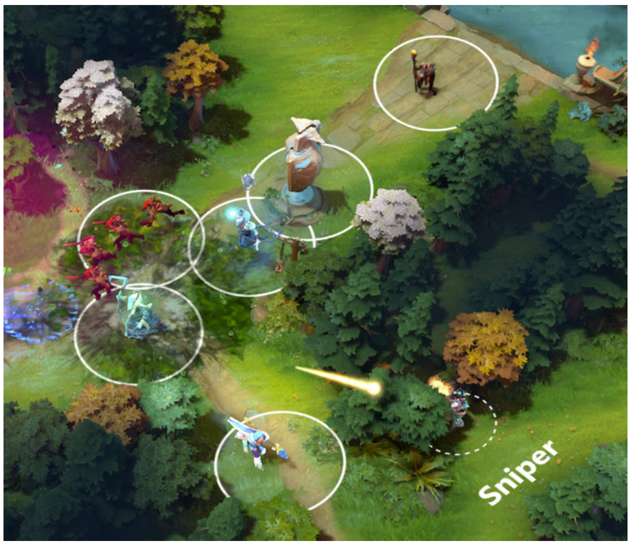


Figure 3.1: OpenAI Five: Discretized attack actions [OpenAI, 2018b]

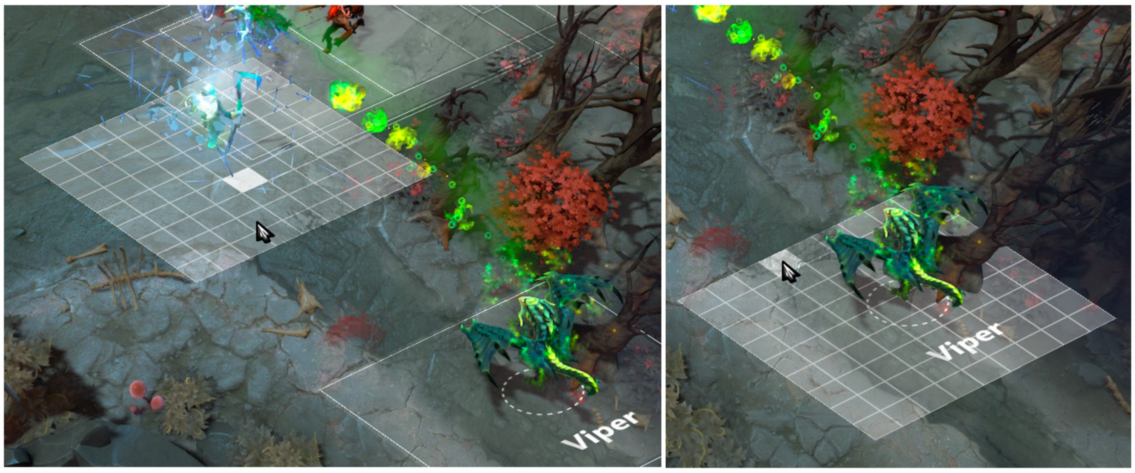


Figure 3.2: OpenAI Five: Discretized spell cast (left) and movement locations (right) [OpenAI, 2018b]

80,000 ticks. Every fourth frame is used to make a decision [OpenAI, 2018b]. Therefore, 20,000 moves are to be played by a DotA agent. In comparison, 150 moves are usually required to complete a Go match [OpenAI, 2018b]. Besides these long time horizons, more difficulties arise by the complexity of the environment's state representation (i.e. observation space). 20,000 values are observed by a DotA agent, whereas only 400 values are required to fully observe the environment in Go [OpenAI, 2018b].

3.1.2 Action Space Composition

Due to this thesis' ambitions, it is of particular interest to examine how OpenAI composed the action spaces of their agents. They did not implement a solution to let an agent mimic a mouse cursor and a keyboard, like it is intended by the underlying work. Instead, they created 170,000 discrete actions (per hero) to cover all actions, which are normally

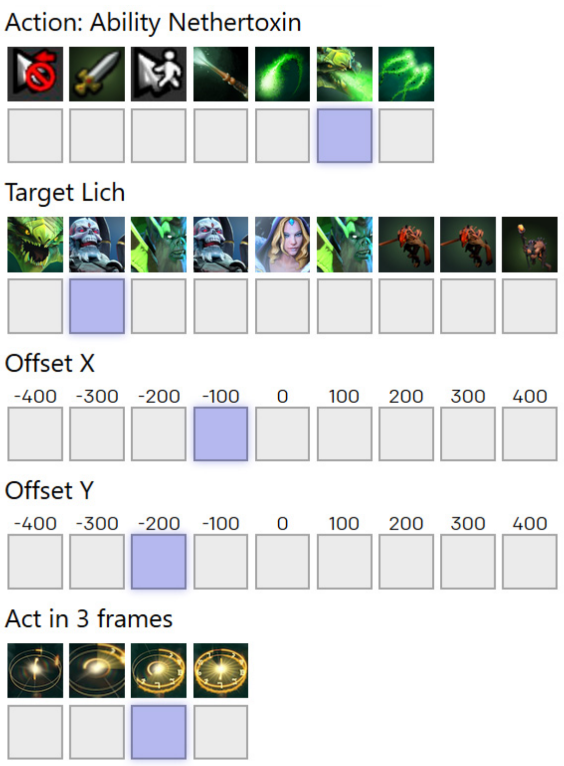


Figure 3.3: OpenAI Five: Exemplary composition of the action Nethertoxin, targeting the enemy hero Lich, choosing a positional offset on the x and y axis (grid usage) and acting in 3 frames. [OpenAI, 2018b]

continuous and discrete for human players.

Some abilities (e.g. spell casts) and attacks require an explicit game entity to be targeted (e.g. enemy hero or building). A human player would have to select the desired ability or attack move and then click on the intended entity. In OpenAI Five, these actions are hard coded. Given the scene in Figure 3.1, the agent (Sniper) has 6 potential foes to attack. So there are ultimately 6 attack actions, where each one represents a different target.

Whenever a location on the ground is to be selected (Figure 3.2), the continuous action space is discretized to a limited grid. For selecting a position for the agent to move to, the grid is limited to the agent's (Viper) position. If a location is to be selected for an ability, the grid is originated at game entities.

Another action dimension provides the possibility to execute the action on the next frame or with a delay of 1, 2, or 3 frames. An example for choosing one action is illustrated by Figure 3.3. At last, it is to be mentioned that not every single action may be available for every possible state.

3.1.3 Taken Approach

Before, it was believed that further advances in the field of RL (e.g. hierarchal RL) are required to solve environments with a long time horizon [OpenAI, 2018b]. However, the PPO algorithm was capable of achieving well performing agents in the 1 vs 1 scenario of DotA where agents outplayed human professionals [OpenAI, 2017]. For the 5 vs 5 scenario, PPO is scaled up to a massively distributed system featuring 256 GPUs and 128,000 CPU cores [OpenAI, 2018b]. Every hero (i.e. agent) trained its own recurrent neural network. This network architecture features so called action heads, which are computed independently. Each head has its own purpose (e.g. selecting action delay or target). Unfortunately, more information on the action selection architecture is not available yet.

Concerning the cooperation among the agents in their respective teams, they did not implement a solution for communication. Instead, a "team-spirit" hyperparameter, ranging from 0 to 1, indicated how much the agent cares about its own rewards or the rewards gathered as a team. Further details of the training setup and statistics can be found on OpenAI's blog post [OpenAI, 2018b]. Adjustments made to the PPO algorithm are not published yet.

3.1.4 A Few Results

Until June 2018, the trained behaviors of OpenAI Five won most of the games against amateur human teams. Later, the trained agents faced a more skilled human team [OpenAI, 2018c]. Two out of three games were won. One peculiarity (out of the many interesting findings) of the resulting behaviors is that the agents learned to utilize a strategy, which is known in the professional scene [OpenAI, 2018b]. In August at the *International* tournament, the agents lost two games against a professional human team [OpenAI, 2018a]. Bugfixes, more training, and removing remaining scripted logic, will be OpenAI's next steps on improving their team of five agents [OpenAI, 2018a].

3.2 Action Branching Architectures

Tavakoli et al. (2017) introduced the idea of partially distributed actions, called *action branching*. Action branching can be used to modify a neural network function approximator to enable concurrent discrete action spaces, which may be high-dimensional. Their contributions are utilized by Unity's ML-Agents toolkit and are required by this thesis to implement the bucket approach (Section 4.4).

The inspiration for action branching is drawn from octopuses' neural systems. Half

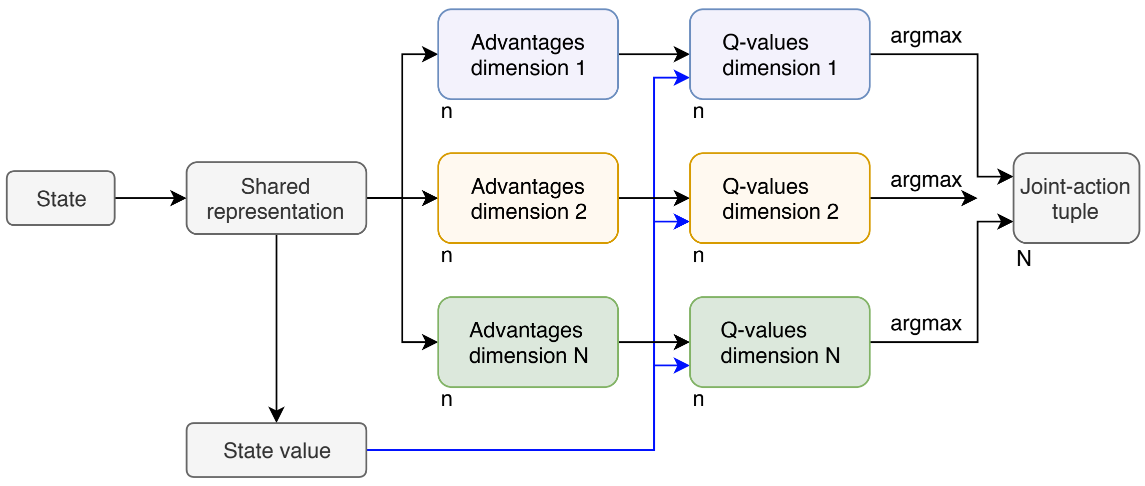


Figure 3.4: Branching Dueling Q-Network by Tavakoli et al. (2017)

of these octopuses' neurons are located in their arms, granting them a certain degree of autonomy [Tavakoli et al., 2017]. Using this idea, Tavakolie et al. implemented the *Branching Dueling Q-Network* (BDQ), which is a branching variant of the *Dueling Double Deep Q-Network* (DDQN) [Lillicrap et al., 2015]. However, it is stated that this architecture can be applied to other discrete RL algorithms [Tavakoli et al., 2017]. A Q-Learning variant was chosen due to its many researched extensions and sample efficiency [Tavakoli et al., 2017].

3.2.1 Motivation

The authors' main intention is to solve the issue of high-dimensional action spaces, caused by the exponential growth of combining possible actions. This strong increase of combined actions can be formalized for an N-dimensional action space, where each dimension d comprises n_d sub-actions:

$$\prod_{d=1}^{N} n_d. \tag{16}$$

Also, thoroughly exploring such large action spaces is challenging [Lillicrap et al., 2015].

3.2.2 Branching Dueling Q-Network Architecture

The architecture of BDQ, which allows a linear increase of the total number of network outputs, is shown in Figure 3.4. After providing the agent's observation (i.e. state) to the network, a shared decision module starts encoding the information, which is branched off the dueling part of the network (i.e. evaluation of the state value) and the individual action branches, where each one possesses an individual degree of freedom. Once action advantages and Q-values are computed for each dimension, the selected actions are concatenated to a joint-action tuple [Tavakoli et al., 2017].

Details of the final implementation affect these key components [Tavakoli et al., 2017]:

- the action-value function:

$$Q_d(s, a_d) = V(s) + (A_d(s, a_d) - \max_{a'_d \in \mathcal{A}_d} A_d(s, a'_d)), \tag{17}$$

- the temporal-difference target:

$$y = r + \gamma \frac{1}{N} \sum_d Q_d^{-}(s', \arg\max_{a'_d \in \mathcal{A}_d} Q_d(s', a'_d)), \tag{18}$$

- the loss function:

$$L = \mathbb{E}_{(s,a,r,s')\sim D}\left[\frac{1}{N} \sum_d (y_d - Q_d(s, a_d))^2\right], \tag{19}$$

- the error for experience prioritization
- and gradient rescaling

In the equations listed above, $d \in \{1, 2, ..., N\}$ specifies the action dimension containing n discrete sub-actions. The set of each dimensions' sub-actions is denoted as $\mathcal{A}_d$.

3.2.3 Outcome

BDQ is put to test against the RL algorithms DDQN and *Deep Deterministic Policy Gradients* (DDPG) on varying reacher environments and several locomotion benchmarks of OpenAI's MuJoCo Gym (e.g. Walker2D-v1). For the environments, which use a continuous action space, the continuous actions are discretized using a certain number of fine-grained sub-actions. Their results show strong superior performance of the contributed action branching architecture in the Humanoid-v1 environment. Mixed results were achieved on other environments. However, the authors see potential for further investigations.

3.3 Imitation Learning with Concurrent Actions

Harmer et al.'s (2018) work is worth exploring in greater detail as well due to their similar motivations and goals in relation to this thesis. To increase the entertainment and immersion of players, concurrent action spaces shall lead to richer agent behaviors as described in Section 1.1. Overall, Harmer et al. (2018) contributed an approach for concurrent discrete action policies. This approach makes use of imitation learning to inject decaying expert data into the RL optimization process. Moreover, the learned policy surpasses human skills. A batched version of the *Asynchronous Advantage Actor-Critic* (A3C) algorithm was implemented [Harmer et al., 2018].

3.3.1 Modified Loss Function

Typically, a softmax activation function is used for the policy output layer of a neural network approximator [Harmer et al., 2018]. While increasing the probability of a singular action due to a positive advantage, the other actions' likelihood are decreased as a cause of the softmax activation. This activation function is replaced with a sigmoid activation, which works like an independent probability for each single action. However, this activation only affects the selected action by the agent. This causes many actions to be selected at once, using the following loss:

$$-\nabla \sum_i a_i log(p_i)\hat{A}, \tag{20}$$

where a_i denotes the selected action i and p_i is the as (0,1) encoded policy for action i [Harmer et al., 2018]. Solving this issue is done by increasing "the probability of $(1-p)$ for the actions not selected $(1-a)$" [Harmer et al., 2018]. This simulates the effect of the softmax activation leading to the modified loss function [Harmer et al., 2018]:

$$-\nabla \sum_i a_i log(p_i)\hat{A} - \nabla \sum_i (1-a_i) log(1-p_i)\hat{A}. \tag{21}$$

3.3.2 Conducted Experiments

A simple 3D FPS video game is used to test agents, which utilize single actions, concurrent actions, and concurrent actions learned with the help of imitation learning. The observation space comprises a $128x128$ pixel RBG image and scalars like remaining health and ammunition. A total number of 13 actions can be selected by the policy. Furthermore, a recurrent neural network is employed.

Over the course of 70 million steps, training the concurrent action approach without imitation learning performed worse (score $\sim$ 25) than the single actions one (score $\sim$ 40) [Harmer et al., 2018]. The best result (score $\sim$ 100) was achieved by the concurrent action policy, which was trained with the help of imitation learning [Harmer et al., 2018]. Harmer et al. (2018) discussed this outcome on the basis of the issues of TD, which are detailed in their publication. As future work, they intend to develop concurrent continuous action policies to allow expert data being recorded from a mouse and keyboard or a gamepad. This way, agents shall be trained on modern FPS video games [Harmer et al., 2018].

4 Approach

This section describes the details of the taken approaches of developing two novel environments and applying concurrent discrete and continuous action spaces to them. First, the limitations and the criteria for developing the environments are detailed. After that, the three approaches, comprising a threshold, a bucket, and a multiagent approach, are elaborated and reasoned. The resulting environments, along with their observation spaces, action spaces, reward signals, and functionalities are detailed in the implementation section.

4.1 Limitations

Some apparent limitations have greater impact on how the environments are going to be designed. First of all, time is limited. If an agent takes too many hours to be trained, the development takes much longer. This is due to many training runs, which have to be conducted to tune hyperparameters, test the implemented features, and fix bugs. Another constraint is the available computational power. For this thesis, a PC (4 CPU cores) and a server (10 CPU cores) are employed. Therefore, too complex environments might demand too much training time, leading to a higher risk of defective results. For example, OpenAI Five was trained for weeks and Harmer et al.'s (2018) FPS video game was trained for 70 million steps. So complexity is a high concern and in general, it is advisable to start out as simple as possible when building RL environments. Once training succeeds, complexity can be raised. To keep the to-be-trained problem smaller, images are not used as observation.

4.2 Environment Criteria

As seen from the previous subsection, time is crucial and therefore a requirement for developing the environments. 1 million steps of training time is considered feasible. It is also important to provide more than one environment, because the approaches later introduced shall work for more than one use-case. Besides that, criteria has to be defined to suit this project's goals: develope an agent mimicking computer-input devices. Computer games that are played with these devices demand the player to play accurately and to time their actions. Thus, the environments have to challenge the agent in tasks that require precision. Precision is related to continuous actions (e.g. mouse cursor movement), and click events (discrete actions) require proper timing.

To give a brief overview, this is the final criteria:

- Implementation of more than one environment

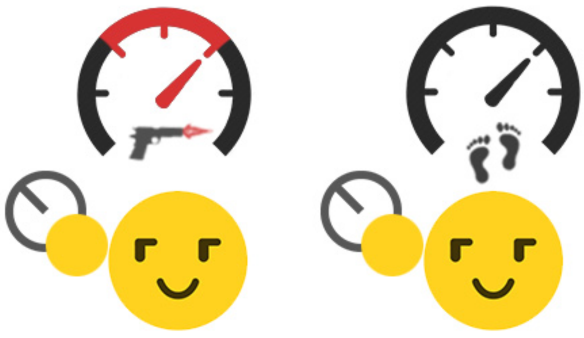


Figure 4.1: The threshold agent uses a range, specified by the threshold, to decide on whether to shoot or not to shoot (left), while simultaneously selecting a movement speed (right).

- Max training time: 1 million steps
- No image observation
- Build environments that ask the agent to be precise
- Continuous actions (e.g. movement)
- Discrete actions (e.g. shoot)

4.3 Threshold Discretization

This approach considers all actions to be continuous. Continuous actions, such as movement, are represented by a value from the range $[-1,\ 1]$ in the ML-Agents toolkit. These actions are carried out without further processing. Actions, which are supposed to be discrete, have to be derived from a continuous one by applying a threshold. The threshold is applied to the action's absolute value:

$$\begin{aligned} |action\ value| &< \text{threshold} \quad \rightarrow \quad \text{Shoot} \\ |action\ value| &\geq \text{threshold} \quad \rightarrow \quad \text{Do nothing} \end{aligned}$$

For example, if a threshold of 0.1 is applied, every value in the range $(-0.1,\ 0.1)$ will trigger the actual discrete action. Querying the threshold can be done either using the less-than or greater-than operator. So the discrete action can be triggered due to values inside or outside the range of the threshold's value. Thresholds, which are applied with the less-than operator are called "inner thresholds", whereas the opposite are called "outer thresholds". Figure 4.1 illustrates a threshold agent.

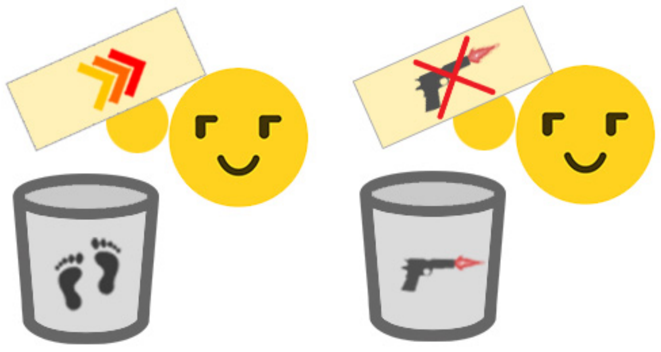


Figure 4.2: The bucket agent draws a movement speed value from a limited amount of options (left), while simultaneously deciding on the discrete actions to shoot or not to shoot (right).

For the conducted experimental training sessions, seven thresholds are selected:

$$thresholds \ = \ \{ \ 0.1, \ 0.2, \ 0.33 \ , 0.5 \ , 0.67, \ 0.8, \ 0.9 \ \}$$

14 training sessions are conducted for the selected values in the inner and outer range. More thresholds are not selected due to the time constraint. The chosen values are evenly distributed in the valid range $(0, \ 1)$ to cover a broad spectrum, except for the thresholds at the tails of the threshold set. This is because of a potentially high probability that this discrete action is triggered too frequently during exploration. If an agent receives a negative reward for failing such discretized actions frequently, it might end up in a policy which makes the agent too afraid of choosing these actions again. However, this is an assumption, which might be negated by the results.

4.4 Bucketed Continuous Action

The bucket approach can be understood as the opposite of the threshold approach. This time, actions which are meant to be continuous are now discretized by a set of discrete actions within a certain range, referred to as "bucket" from now on. This bucket also simplifies a continuous action, if its resolution (number of fine-grained actions) is not too high. Due to this simplification, attention has to be given to the loss of precision, which could result in a disadvantage for the present approach. Discrete actions stay the same for this action setup. Figure 4.2 shows an example of a bucket agent.

Buckets can be based on Tavakoli et al.'s (2017) action branching architecture, which is fortunately implemented by Unity's ML-Agents toolkit. One action branch shall represent one continuous action. Further branches can implement classical discrete actions. Also,

Figure 4.3: Both agents make use of their natural action spaces where the continuous agent chooses a movement speed (left), while simultaneously the other one decides whether to shoot or not to shoot (right).

the problem of incompatible joint-action tuples are avoided this way, because only one action per branch is selected to create the tuple. For example, shooting and reloading can be part of the same branch and therefore do not conflict.

A continuous action is bucketed by a certain number of sub-actions in a certain range. In the case of this thesis, the individual actions are selected from the range $[-1,\ 1]$. The policy tells the agent which sub-action to take from the bucket. The only reason for selecting the range $[-1,\ 1]$ is easing the implementation, because continuous actions work on this range, too. In general, sub-actions of a bucket can be composed to the developer's alike. Five buckets, with different levels of detail, are tried out throughout this thesis:

$$\begin{aligned}
Bucket7 &= \{-1.0,\ -0.5,\ -0.1,\ 0.0,\ 0.1,\ 0.5,\ 1.0\} \\
Bucket11 &= \{-1.0,\ -0.8,\ -0.6,\ -0.4,\ \cdots,\ 1.0\} \\
Bucket21 &= \{-1.0,\ -0.9,\ -0.8,\ -0.7,\ \cdots,\ 1.0\} \\
Bucket41 &= \{-1.0,\ -0.95,\ -0.9,\ -0.85,\ \cdots,\ 1.0\} \\
Bucket201 &= \{-1.0,\ -0.99,\ -0.98,\ -0.97,\ \cdots,\ 1.0\}
\end{aligned}$$

The smallest bucket contains 7 actions, whereas the biggest one consists of 201 actions. Smaller discrete actions near 0 are chosen for the smallest one to allow the agent to, for example, slow down its movement, if this might be of utility for the agent's precision. All other buckets are distributed evenly with different step sizes $\{0.2,\ 0.1,\ 0.05,\ 0.01\}$ leading to the quantities $11,\ 21,\ 41$ and 201.

4.5 Multiagent

Assembling a multiagent, using agents that only use one of the two available action space types, is the last approach. Each respective action space can remain the same without applying any buckets or thresholds. An example is given by Figure 4.3. However, using multiple agents has a few drawbacks. One deals with a redundant observation space, because each agent trains its own model, and in the worst case, both agents process the same amount of information. Depending on the environment, some information can be omitted. For example, a continuous agent, which is in charge of the agent's locomotion, does not need to know its remaining ammunition, because it is not capable of taking the reload action. Same accounts for the reward function. The aforementioned continuous agent cannot be punished for reloading a fully loaded gun. Thus, the agents may vary in all RL components, which make them hard to compare to the previously mentioned approaches. As all agents have the same goals, they all have to receive a shared reward signal to aid cooperation. Due to the many potential modifications, only observation spaces may vary for different training runs. Another drawback is concerned with a potential need for communication among the agents. The explicit multiagents setups are described in the next section.

5 Implementation

This section describes the implementation of the underlying thesis to carry out the previously described approach. At first, dependencies, especially the *ML-Agents Toolkit* (ml-agents) from the game engine Unity, are portrayed. After that, both proposed environments are detailed concerning their functionalities, observation spaces, action spaces, and rewards. The setup of the multiagent, bucket and threshold approaches are described at the end. The resulting code can be found in this GitHub repository[3].

5.1 Unity ML-Agents Toolkit

Unity's ML-Agents Toolkit [Juliani et al., 2018] is the fundamental basis for this project's implementation. It features an interface between the game engine Unity and Python to allow the application of state-of-the-art ML technologies. The game engine Unity can be easily used to develop new and complex RL environments. ML-agents include an implementation of the PPO algorithm, which differs from the original paper by Schulman et al. (2017). These details comprise:

- A linearly decaying learning rate, which is based on the maximum number of learning steps of the training run
- Action branching for concurrent discrete actions based on Takavoli et al. (2017) contributions
- A Gaussian distribution for continuous actions, which is clipped to the range $[-3,\ 3]$ and then scaled down to $[-1,\ 1]$

The used software versions are Unity 2018.2.2, ml-agents 0.5.0, Python 3.6.3, and *TensorFlow* 1.7.0. Further dependencies of the python environment can be found in the annex.

5.2 Environment: Shooting Birds

Shooting Birds (Figure 5.1) is inspired by the German game *Moorhuhn*[4]. In this environment, the agent steers a cross-hair to shoot birds. Birds of three different sizes are randomly spawned on the left and right side of the environment. Their velocities and flying behaviors are stochastically initialized. The agent's shooting functioning is limited by ammunition (8 shots). Therefore, the agent has to manually reload to be able to shoot again. Birds, which are hidden behind obstacles (e.g. tree and barn), cannot be hit.

[3]`https://github.com/MarcoMeter/DRL-Concurrent-Discrete-Actions`
[4]`https://www.moorhuhn.de/`

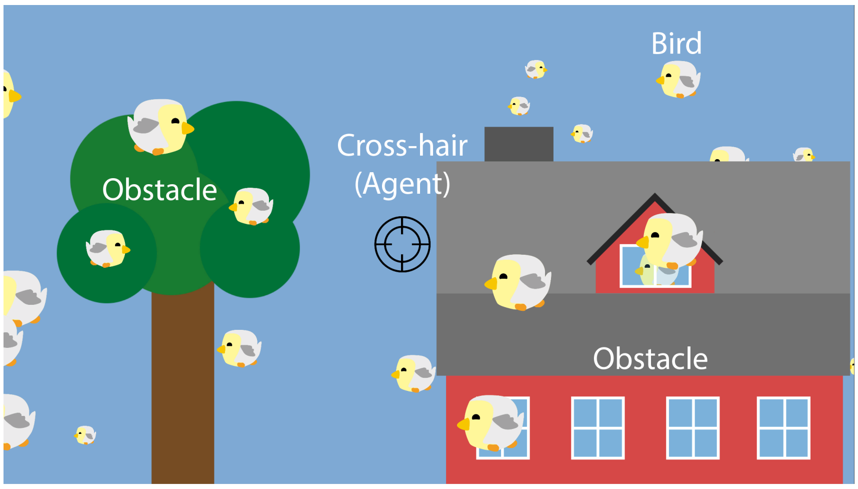


Figure 5.1: The environment Shooting Birds and its entities

5.2.1 Observation Space

To partially observe the environment, the agent collects 30 observations. 24 of these values are perceived by using raycasts (Figure 5.2). Starting from the center of the cross-hair, the raycasts are sent outwards at angular steps of 15 degrees. This way, only distances to detected birds are added to the observation space. If no birds are detected at all, the observation is described with -1 for that particular raycast. Other entities are ignored. The remaining 6 values comprise:

- Remaining ammunition
- Agent's position
- Agent's velocity
- Hovered entity

The position and velocity of the agent comprise the values for the x and y axis. Detectable entities, which are hovered by the cross-hair, can be either a bird, an obstacle, or nothing.

5.2.2 Reward Signals

Accurately shooting down birds as fast as possible is the agent's goal. Thus, 6 different rewards are signaled throughout the environment's execution. The agent is rewarded for hitting a bird. However, the size of the bird matters. Big birds handout +0.25, medium ones +0.5, and small ones +1. Hitting nothing is punished with a negative reward of -0.1. The other reward signals are related to the ammunition. If the agent tries to shoot without ammunition, it is punished by -1. Reloading while the agent's gun is still loaded is punished as well by -0.1.

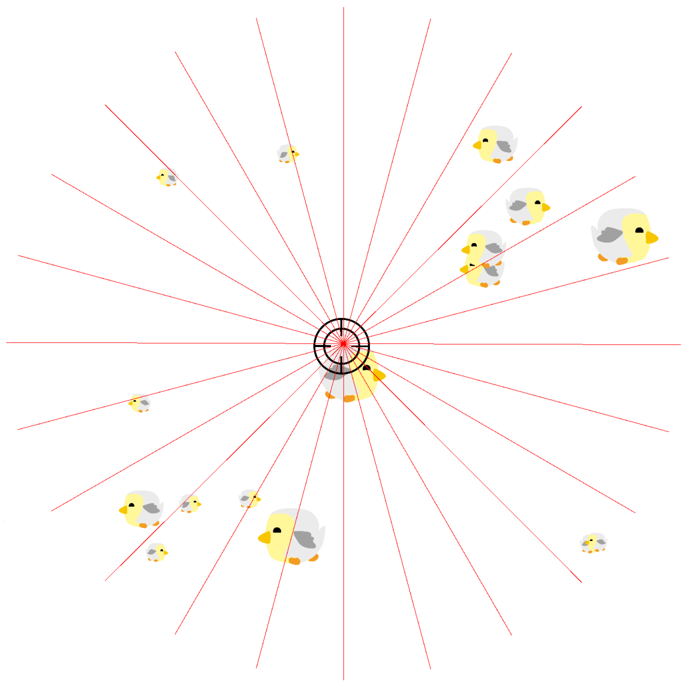


Figure 5.2: Raycasts to measure distances to perceived birds

5.2.3 Action Space

Two continuous and two discrete actions are available to the agent. The agent's horizontal and vertical velocities are dependent on the continuous ones. Shooting and reloading are triggered by the discrete ones. In the case of the bucket and multiagent approaches, a third discrete action is added, which does nothing at all. This is necessary for the agent to decide when not to shoot or not to reload. The agent makes a decision for every step of the environment.

5.2.4 Multiagent Setup

Two agents compose the multiagent solution. One carries out the agent's continuous movement and the other one decides when to shoot, to reload, or to do nothing. Both agents use, to some extent, different reward functions and observation spaces. The continuous agent senses the whole observation space as defined in Section 5.3.1, excluding the remaining ammunition. Only four values are observed by the discrete agent. These comprise the remaining ammunition, the agent's vertical and horizontal velocities, and the currently hovered entity. However, another experimental setup makes the whole observation space available to the discrete agent. Concerning rewards, the discrete agent receives all signals whereas the continuous agent is only rewarded for a bird being shot in relation to the bird's size. A punishing signal is not sent to the continuous agent, because it cannot be made responsible for reloading a loaded gun or shooting without ammunition.

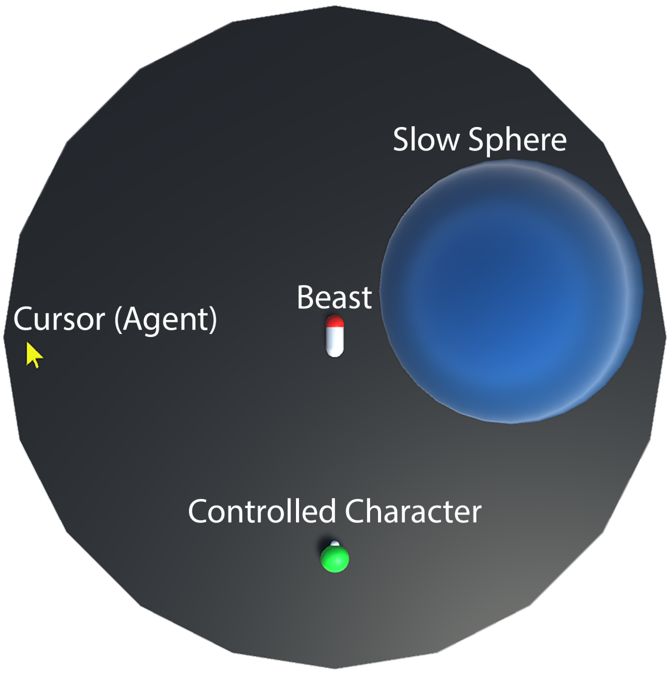


Figure 5.3: The simplified BRO environment and its entities

5.3 Environment: Beastly Rivals Onslaught

This implemented environment is a simplification of the game BRO[5]. To avoid the challenge of more complex multiagent systems, the original game is broken down to just one player with the goal of surviving as long as possible. Figure 5.3 shows the four main game entities:

- A mouse cursor, which represents the agent
- A character, which is controlled by the agent
- A beast, which chases and kills the character once reached
- A slow sphere, which wanders around and slows down the beast and the character

Over time, the beast gets more accurate and moves more quickly, and thus survival is more and more challenging. Another threat or potential support is the slow sphere. The character's and the beast's velocities are reduced while inside this sphere. The closer they are to its center, the stronger the effect is. To survive, the agent has to send suitable commands to its controlled character. By "clicking on the ground" the agent tells the character to move to that location. Another command, which could be implemented as a "double-click" for a human player, is the blink ability. Once commanded, the character is teleported instantly to the location of the cursor. The blink ability is then on cooldown (i.e.

[5]Video of the game BRO `https://youtu.be/OTcDd7a_R0A`

unavailable) for 200 steps of the environment. For visualization purposes, the character is colored green while not being able to blink.

BRO has some stochastic elements, which affect the environment's initial state. Random locations are chosen for the agent, the controlled character, the slow sphere and the beast. Additionally the beast's rotation is randomized.

5.3.1 Observation Space

21 observations are collected by the agent to fully observe its environment:

- Blink cooldown
- Mouse cursor position
- Mouse cursor velocity
- Character position
- Character velocity
- Relative position to the beast
- Beast position
- Beast velocity
- Beast speed
- Beast accuracy growth speed
- Relative position to slow sphere
- Slow sphere velocity

Position and velocity information are two dimensional features. Relative positions are related to the character. The whole observation space is stacked three times.

5.3.2 Reward Signals

Three kinds of reward signals are sent to the agent. As long as the character is alive, the agent gets rewarded with +0.01. Being killed is punished with -1 and the last reward signal punishes the agent with -0.025 for trying to use the blink ability during its cooldown time.

5.3.3 Action Space

Two continuous and two discrete actions are apparent. The continuous ones allow the agent to select horizontal and vertical velocities. One discrete action triggers the movement of the character to the "clicked" location commanded by the agent. The other one is the character's blink ability, which is a teleport to the agent's position. A third discrete action is considered for the bucket and multiagent approaches, because the agent has to be capable of deciding when not to move or not to blink at all. Every 6th step, the agent decides which action to take. In-between these steps the same action is triggered, which has to be drawn into consideration for punishing forbidden blink attempts. So the defined punishment is 6 times stronger than previously defined. This is a peculiarity of the ML-Agents toolkit.

5.3.4 Multiagent Setup

Both action space kinds are divided among two agents. One triggers the discrete actions and the other the continuous actions. So the discrete agent takes care of selecting movement locations, blink positions, and when not to move or not to blink at all. Continuous locomotions on the horizontal and vertical axis are left for the continuous agent. Both agents use the full observation space (Section 5.3.1). Only the reward signals are distributed differently, because the movement agent cannot influence the failed blinking attempts and therefore it will not experience the punishment of -0.025. Besides that, all reward signals from Section 5.3.2 apply to both agents as a team.

5.4 Threshold

Thresholds are a simple condition. The listing below shows an exemplary implementation.

```
if (Mathf.Abs(continuousActionValue) < threshold)
{
    Reload(); // e.g. Shoot, Blink...
}
```

Code Listing 2: Discretization of a continuous action using a threshold

5.5 Bucket

Implementing the bucket approach is easy, as illustrated in Listing 3. The agent selects the sub-action from the bucket based on its discrete action decision, which is an index for the to-be-chosen sub-action.

```
buckets = new float[] {-1.0f, -0.5f, 0.0f, 0.5f, 1.0f};
agentVelocity = buckets[selectedDiscreteAction];
// e.g. move horizontally at speed of -1
```

Code Listing 3: Bucket of discrete actions to replace a continuous action

6 Results

The final outcome of applying the proposed approaches to the implemented environments is covered in this chapter. At first, an overview is given on the conducted experiments, including their performance measures and hyperparameters. After that, a guideline is provided to guide through the results of each individual approach. Ultimately, all approaches are compared and the most interesting learned behaviors are described.

6.1 Conducted Experiments

6.1.1 Training Session Overview

Overall, 41 training sessions were run. Both environments underwent sessions on varying thresholds (14) and bucket sizes (5). Concerning multiagent experiments, one training run, using the full observation space, is conducted on the BRO environment. Two runs are done on SB, which are differentiated based on the perceived observations by the discrete agent. One uses the almost complete observation space (29/30) and the other one just utilizes 4 out of 30 observations.

6.1.2 Performance Measures

To rate and compare the outcomes of the different training sessions, these values, based on the defined criteria (Section 4.2), are measured over the course of each training run:

- General measures:
  - Training time (real-time and steps)
  - Mean cumulative reward
- Environment specific measures:
  - BRO: Episode length (i.e. duration of survival)
  - SB: Click accuracy

The velocity of the agent is another measure, which is recorded post-training for selected results, to observe if the agent actually slows down in order to be more successful in reaching its goals (e.g. hitting birds or choosing blink locations).

6.1.3 Hyperparameters and Training Parameters

Many hyperparameters and training parameters have to be selected for the PPO algorithm and the environment. Explanations of these parameters can be found in the annex. The parameters are initially selected based on experience and examples (e.g. previous projects and example environments of ML-Agents). Further tuning was conducted in a trial-and-error manner where usually one parameter was changed at a time. The next two tables (Tables 1 and 2) show the parameters, which are shared between both environments, except for the maximum episode length. Further parameters are detailed for each approach's distinctive subsection.

Table 1: Shared hyperparameters between both environments

Hyperparameter	**Value**
Gamma	0.99
Epsilon	0.2
Lambda	0.95
Time Horizon	64

Table 2: Training parameters used by the present environments

Parameter	**BRO**	**SB**
Max Episode Length	10,000	2,000
Time Scale	30	30
Agent Count	18	18
Run Count	10	10

6.1.4 Utilized Computer Hardware

The development of the environments was done on a PC using an Intel Core i7-3770k (4x 3,5GHz) CPU. All training sessions were deployed on a server utilizing an Intel Xeon E5-2640v4 (10x 2,4GHz). This way, 10 parallel runs were feasible to benchmark each experiment. A GPU was not used for training the environments.

6.2 Result Guideline

To ease the readability of the to-be-presented results, the preceding subsections cover the results of one approach at a time for both environments. Results of SB are displayed first, while BRO is shown second. Due to the many conducted training sessions, not all results are focused. However, a plot for each single run can be found in the annex. At first the plotted outcome of one environment is shown, which is followed by a description. Before moving on to the next environment or approach, selected hyperparameters are shown in a table. The final sections compare the approaches against each other, provide an overview of the training durations, and describe the learned behaviors.

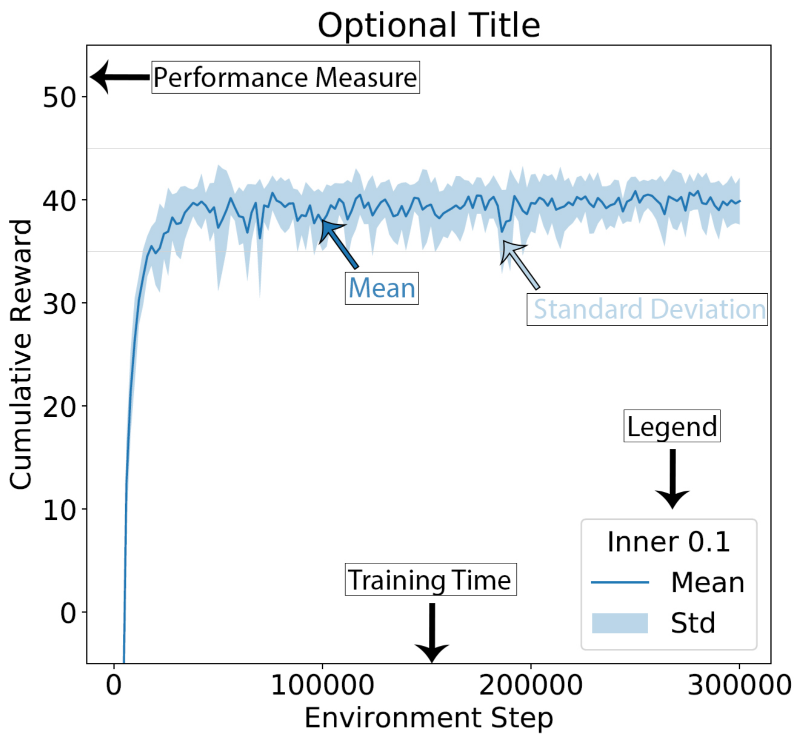


Figure 6.1: A guiding example on what information is provided by a plot

Figure 6.1 is a typical example of the to-be-shown plots. Usually, the caption of a figure is the only element, which distinctively describes what environment and what approach was used. However, in the case of the threshold results, a plot title is added to differentiate the inner and outer thresholds. The y-axis features the cumulative reward, the shooting accuracy (SB), or the episode length (BRO). Time, in terms of environment steps, is always indicated by the x-axis. If multiple plots are shown in one figure, the legend can be found in the bottom right corner of the overall figure. Lastly, the plotted solid line shows the mean over 10 runs of one experiment. Depending on the available space and the number of results shown, the standard deviation is visualized as well, which is denoted by the slightly lighter area.

6.3 Threshold Results

6.3.1 Shooting Birds

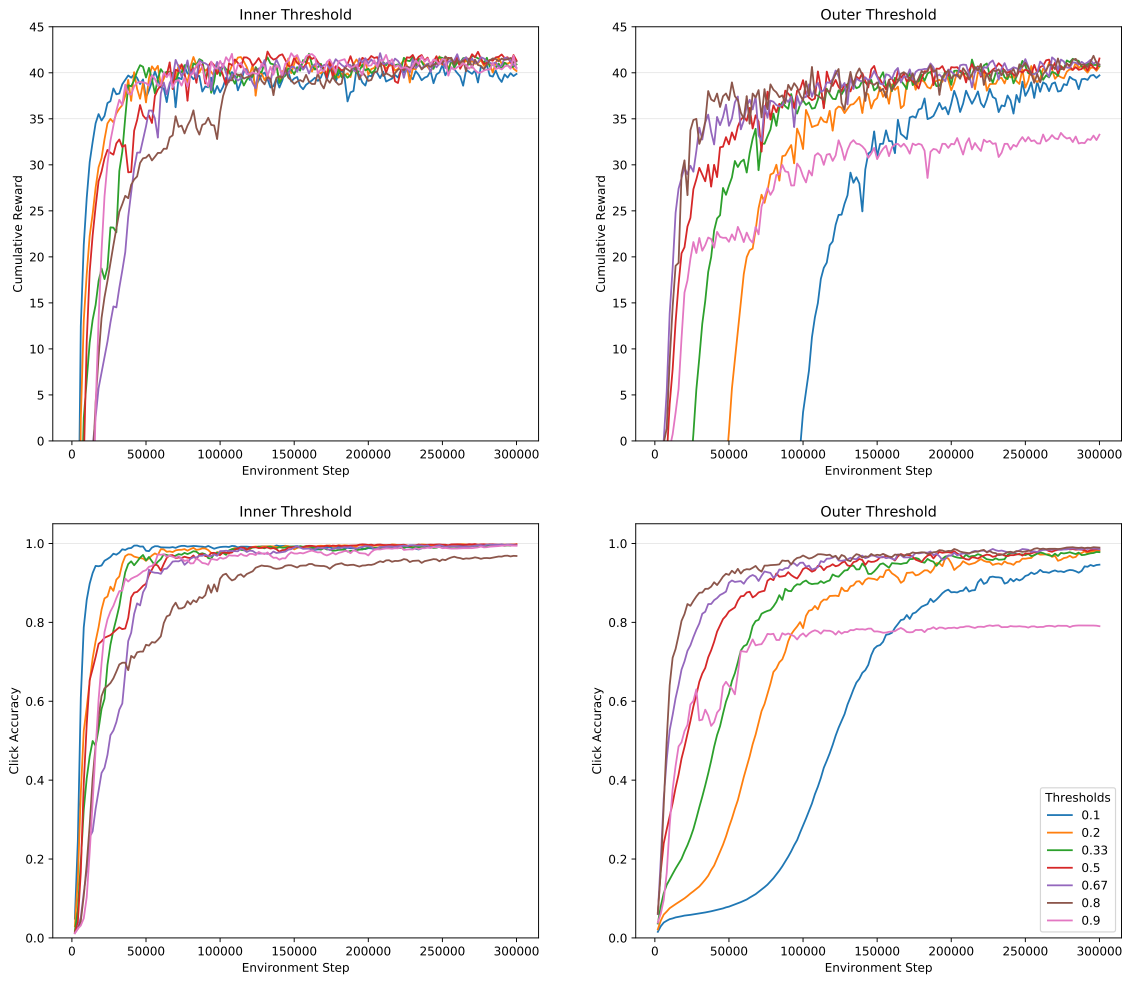


Figure 6.2: SB: Inner and outer threshold results

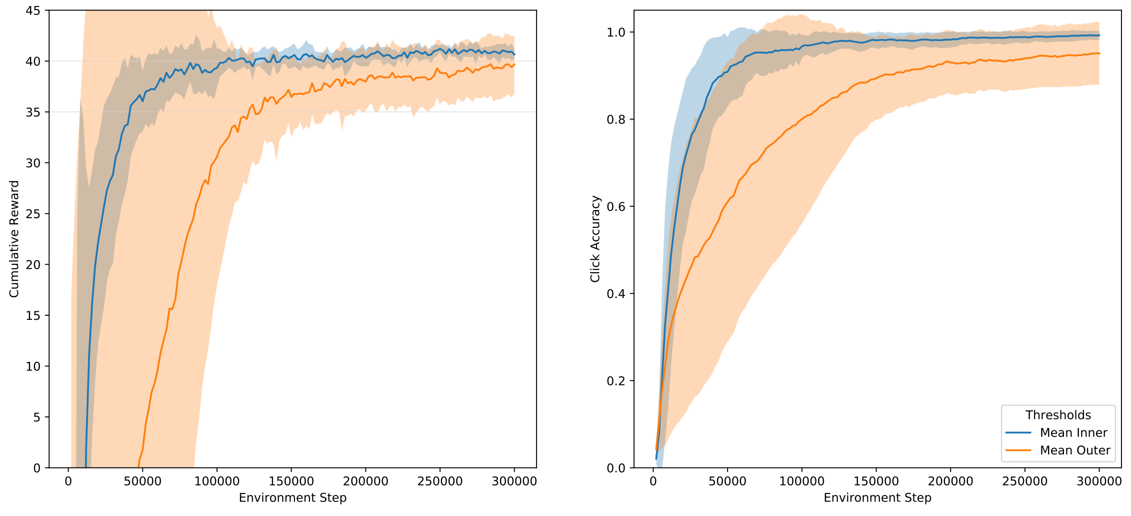


Figure 6.3: SB: Average inner vs average outer threshold values

Looking at the scored cumulative rewards, it can be seen that all inner thresholds converge somewhere in-between the range $(39, 42)$. Most of the time, less than 50,000 steps are required to reach this range of convergence. A more precise look on the annex suggests that the best inner threshold is approximately 0.33. The worst inner threshold is 0.8, due to its lower click accuracy.

The outer thresholds converge to a similar range, but need more time to converge. Concerning the click accuracy, these values do not get as close to 100% in comparison to the inner thresholds. The poorest performance is achieved by the value 0.9 (outer threshold), which does not reach an accuracy of 80% at all. The best outer value is 0.8.

Figure 6.3 illustrates the mean and standard deviation of the inner and outer thresholds. Obviously, inner thresholds outperform the outer ones, while being much more stable. A huge instability is observed during the first half of the training using the outer thresholds.

Table 3: Hyperparameters for the threshold approach (SB)

Hyperparameter	**Values**
Max Steps	300,000
Epochs	3
Batch Size	256
Buffer Size	1,024
Beta	5e-3
Learning Rate	3e-4
Layers	2
Hidden Units	256

6.3.2 Beastly Rivals Onslaught

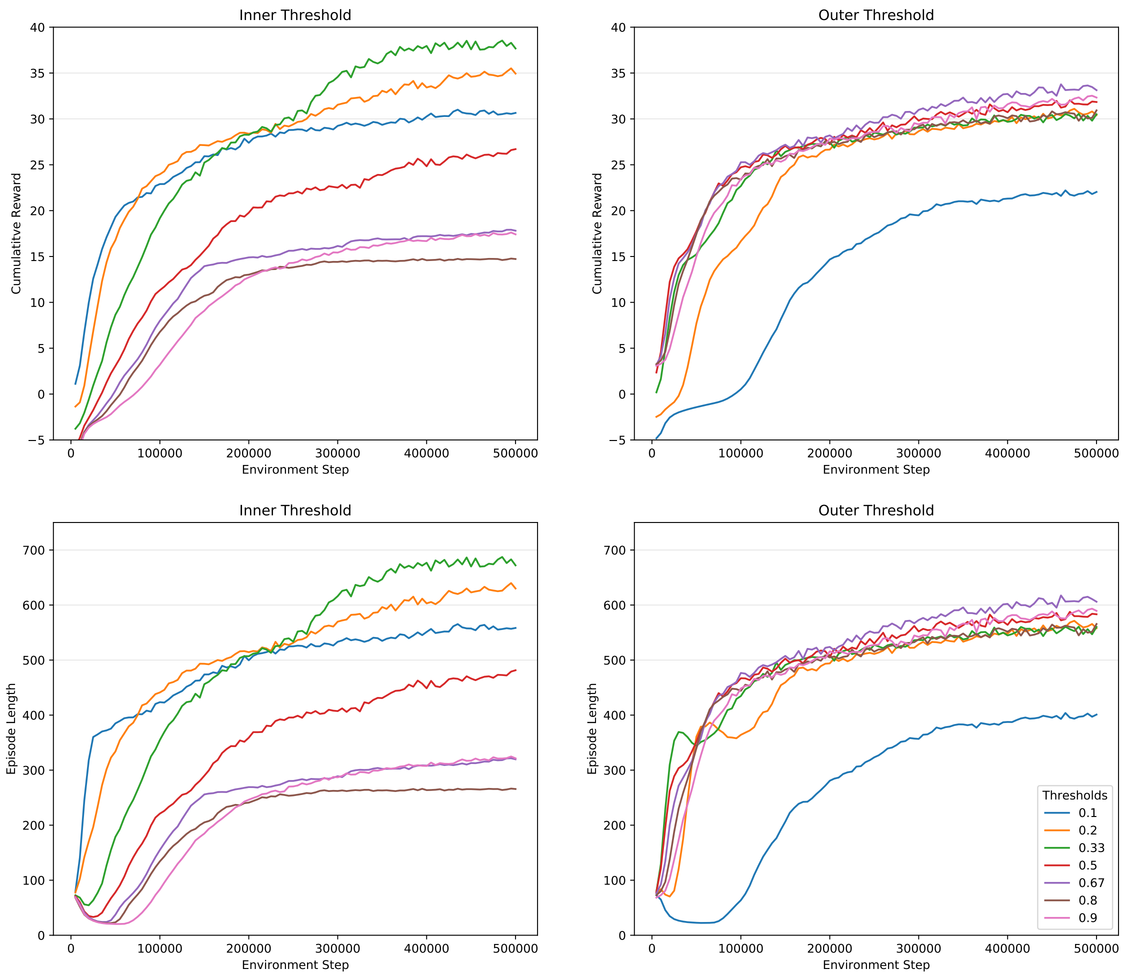


Figure 6.4: BRO: Inner and outer threshold results

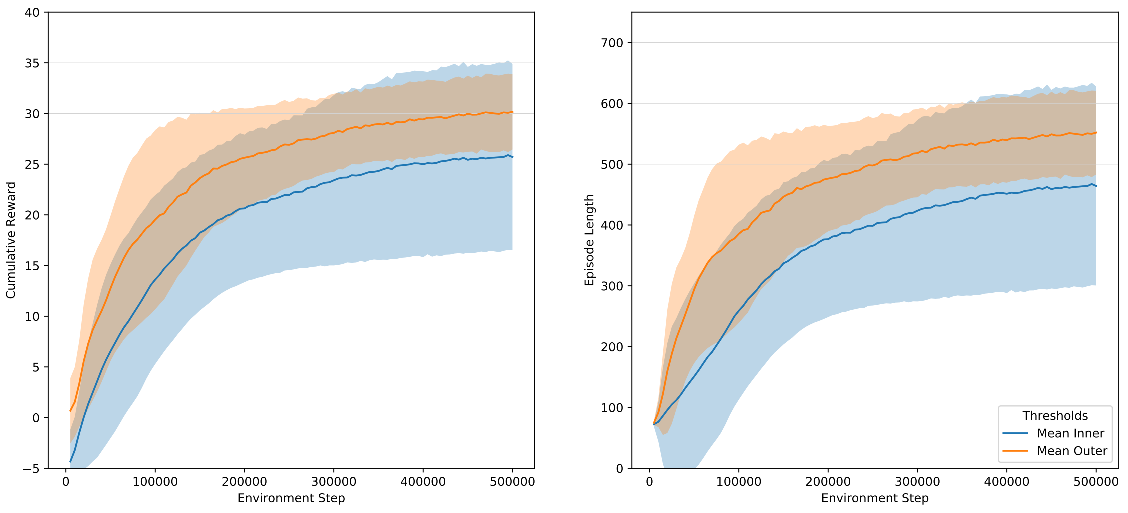


Figure 6.5: BRO: Average inner vs average outer threshold values

A completely different picture is drawn for the results of the BRO environment. At first, it can be seen that a distinction between the cumulative reward and the episode length is not required, because there is a high correlation between these measures (due to the reward signals) and therefore behave almost identical.

Inner threshold values have a lot of variance, whereas the outer ones converge between 30 and 34, except for the outer value of 0.1. It is difficult to estimate a point of convergence for the outer thresholds, because their lines keep slightly growing. But it can be considered that these would not get past the cumulative reward of 35. The best performance across all thresholds is delivered by the inner threshold 0.33, where the agent survives for about 680 steps of an episode. Inner thresholds, at least of the size 0.67, achieve the least compelling results. 0.67 works best for the outer values, while 0.1 is the worst.

Comparing the mean of the inner and outer thresholds (Figure 6.5), outer thresholds achieve longer enduring agent behaviors. However, these results still lack performance, because surviving for at least 680 steps is feasible for a single result. While the means for both threshold kinds do not achieve better results, their high variance indicates the instability of the training.

Table 4: Hyperparameters for the threshold approach (BRO)

Hyperparameter	**Values**
Max Steps	500,000
Epochs	10
Batch Size	2,500
Buffer Size	20,000
Beta	3.5e-3
Learning Rate	4e-4
Layers	1
Hidden Units	30

6.4 Bucket Results

6.4.1 Shooting Birds

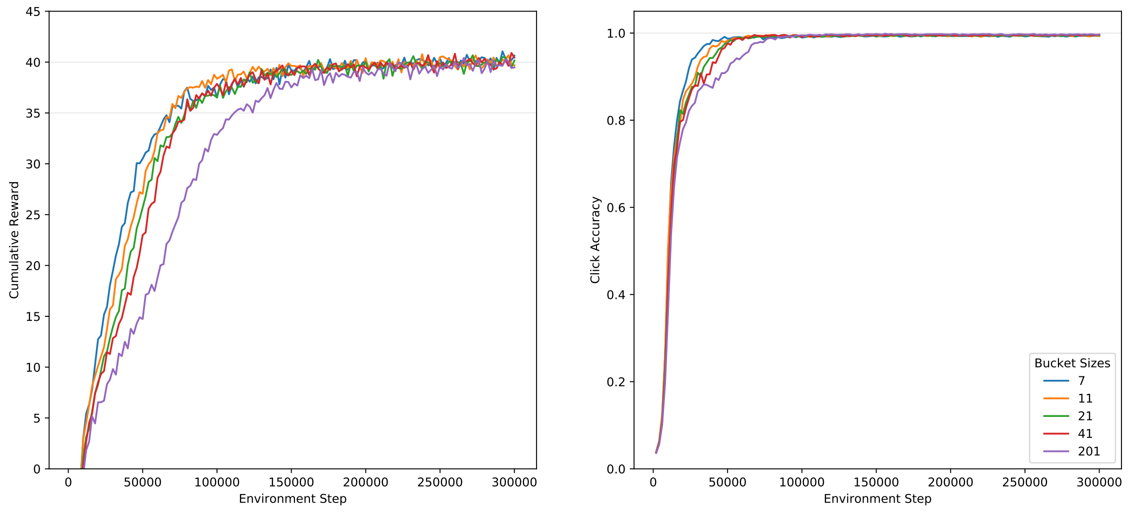


Figure 6.6: SB: Bucket results

All 5 bucket resolutions achieve a high and stable click accuracy (as seen in the annex), while they converge to a cumulative reward of about 40. The biggest bucket, comprising 201 movement speed actions, takes the longest to converge. It appears the two smallest buckets perform slightly better than the ones with 21 and 41 actions.

Table 5: Hyperparameters for all bucket sizes (SB)

Hyperparameter	**Values**
Max Steps	300,000
Epochs	3
Batch Size	96
Buffer Size	10,240
Beta	5e-3
Learning Rate	2e-3
Layers	2
Hidden Units	20

6.4.2 Beastly Rivals Onslaught

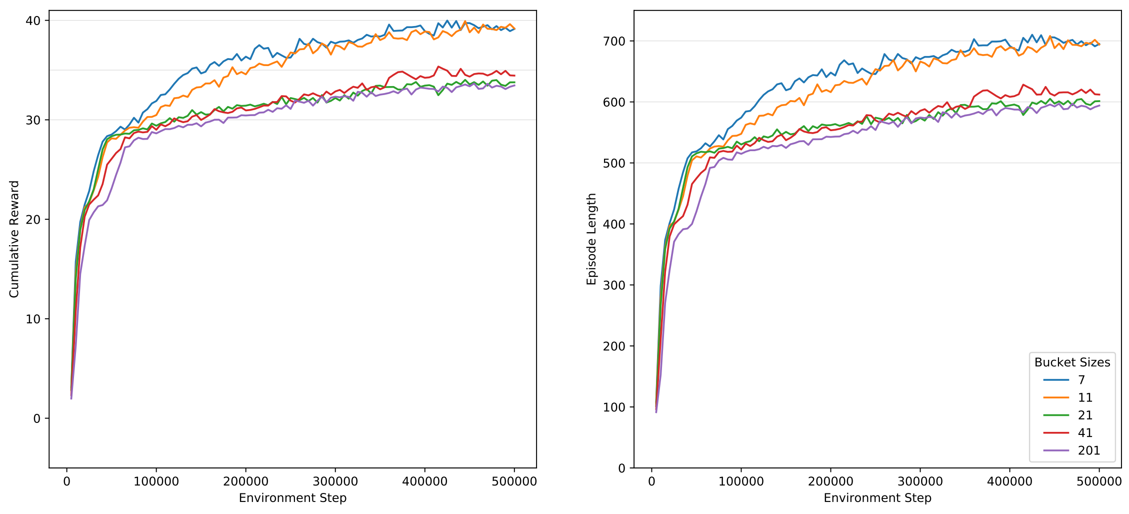


Figure 6.7: BRO: Bucket results

Due to the larger action spaces, different hyperparameters are chosen for the bigger buckets as listed in Table 6.

A clearer distinction of the results is possible for the BRO environment. The smallest buckets perform best by reaching an episode length of about 700. Episode lengths of about 600 are achieved by the larger action spaces. 7 movement speed actions are slightly superior to 11.

Table 6: Hyperparameters for the different bucket sizes (BRO)

Hyperparameter	**7**	**11**	**21**	**41**	**201**
Max Steps	500,000	500,000	500,000	500,000	500,000
Epochs	5	5	6	5	10
Batch Size	96	96	128	128	256
Buffer Size	10,240	10,240	10,240	12,288	12,288
Beta	3e-3	3e-3	3e-3	3e-3	3e-3
Learning Rate	3e-4	3e-4	3e-4	3e-4	3e-4
Layers	1	1	1	1	1
Hidden Units	20	20	20	30	30

6.5 Multiagent Results

6.5.1 Shooting Birds

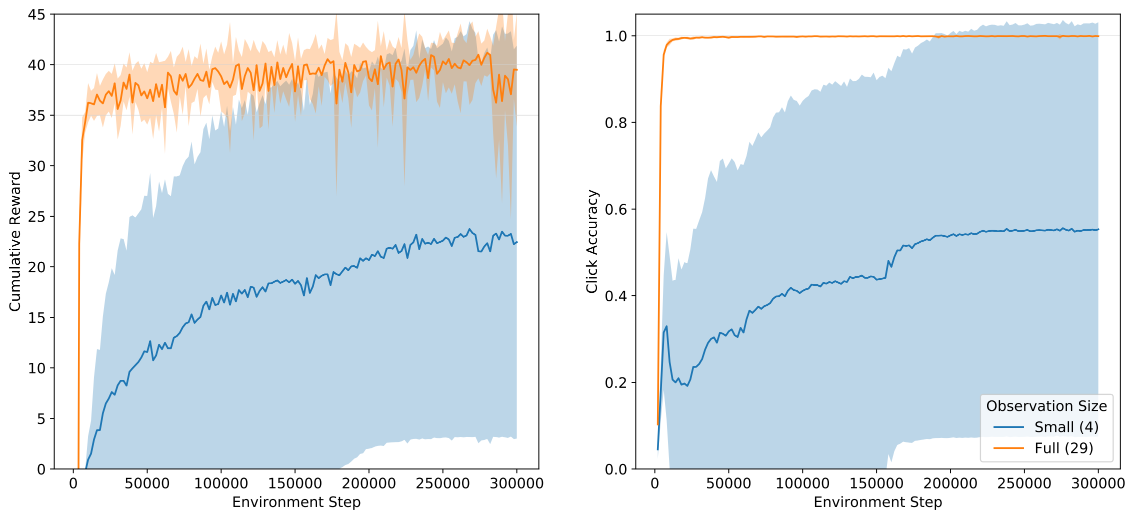


Figure 6.8: SB: Multiagent results

Out of the two training sessions for the SB environment, the one that utilizes the almost complete observation space performs much better than the one using only the least necessary observations (4). A strong variance is present for the multiagent, which uses the smaller action space, so there are really bad and really good results among the 10 run sessions. This observation is particularly focused in the discussion section.

Table 7: Hyperparameters for the multiagent approach (SB)

Hyperparameter	**Continuous Agent**	**Discrete Agent**
Max Steps	300,000	300,000
Epochs	3	3
Batch Size	128	48
Buffer Size	1,024	512
Beta	5e-3	3e-3
Learning Rate	3e-3	2e-3
Layers	2	1
Hidden Units	256	20

6.5.2 Beastly Rivals Onslaught

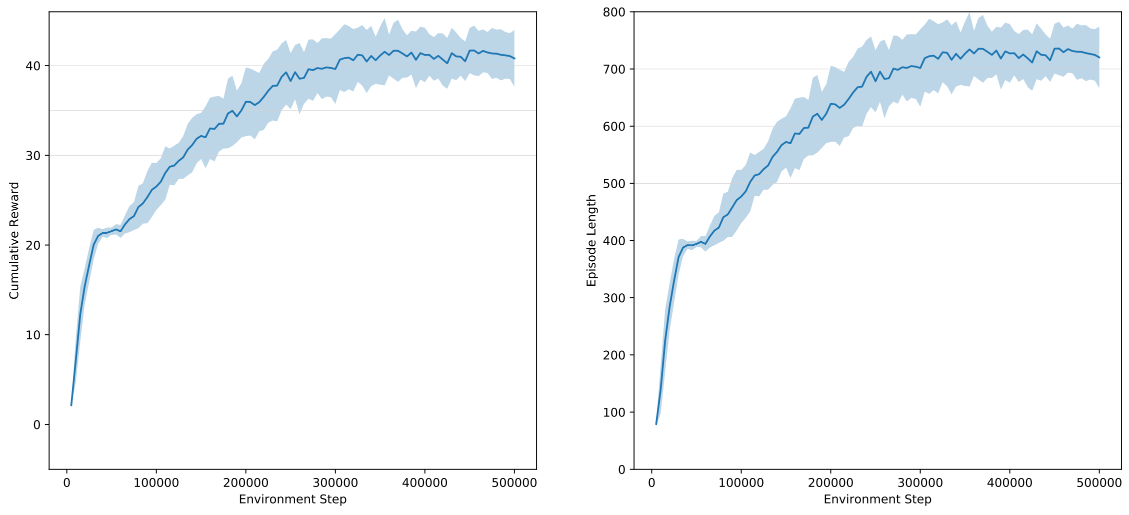


Figure 6.9: BRO: Multiagent results

While being somewhat unstable, the multiagent of the BRO environment achieved long enduring policies. The average time of survival is greater than 700. As usual for BRO, the training cannot be stated as stable.

Table 8: Hyperparameters for the multiagent approach (BRO)

Hyperparameter	**Continuous Agent**	**Discrete Agent**
Max Steps	500,000	500,000
Epochs	5	3
Batch Size	2,000	64
Buffer Size	20,000	8,192
Beta	3e-3	3e-3
Learning Rate	3e-3	3e-3
Layers	1	1
Hidden Units	20	20

6.6 Approach Comparison

6.6.1 Comparing Average Results

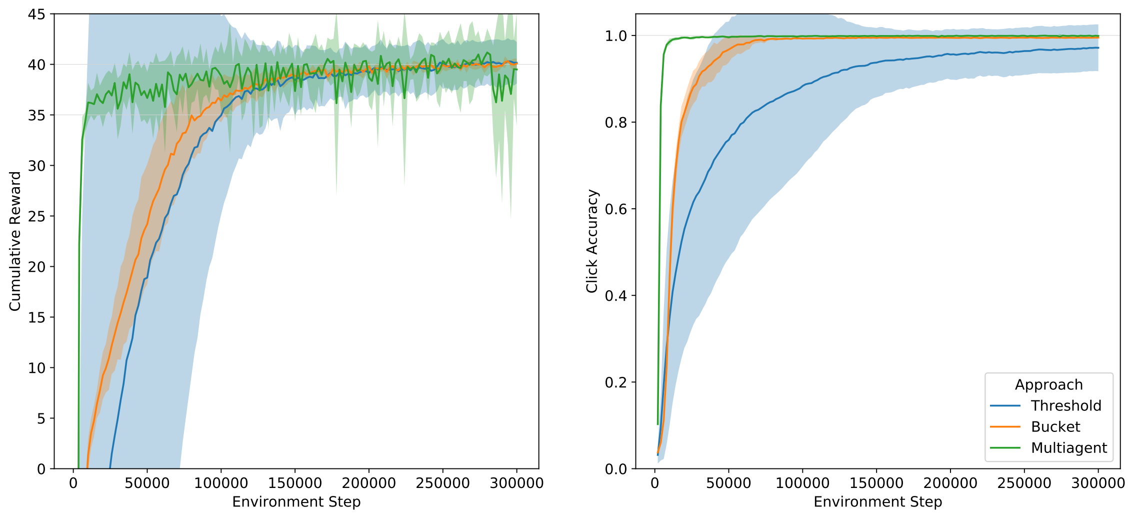


Figure 6.10: SB: Comparing all approaches' results based on their mean

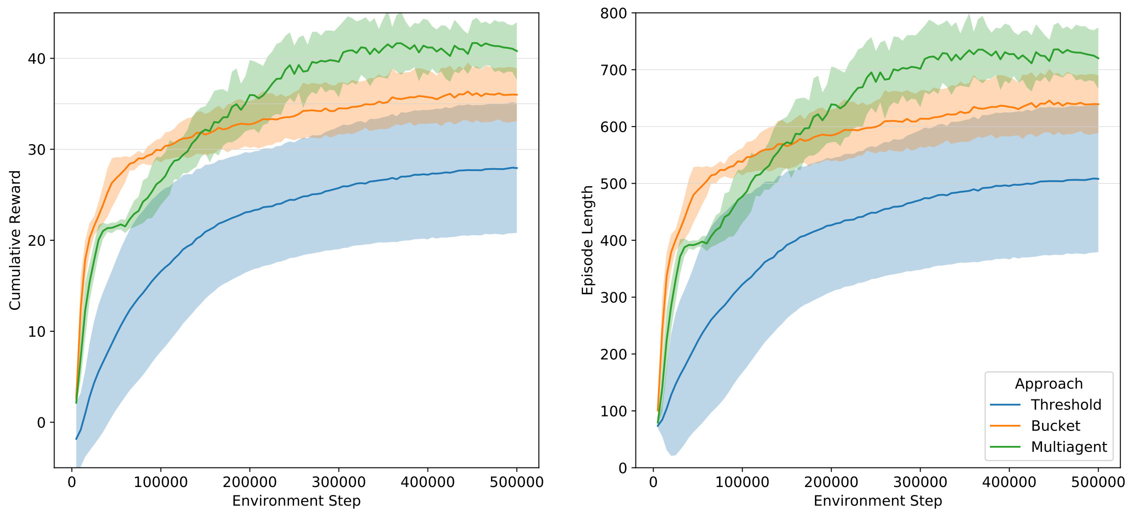


Figure 6.11: BRO: Comparing all approaches' results based on their mean

Figures 6.10 and 6.11 show the mean of all approaches, except for the multiagent run of SB. Concerning the multiagent results of SB, only the result, which uses the big observation space, is included, because the smaller one lacks in comparability due to its high variance and its small number of observations.

Both environments show that the most successful results were achieved using the multiagent approach. The bucket approach comes in second, while the threshold approach is inferior. Also, this order is followed by the strength of variance for these approaches. The average of the threshold approach is strongly unstable in contrast to the bucket and multiagent approach.

6.6.2 Comparing Best Results

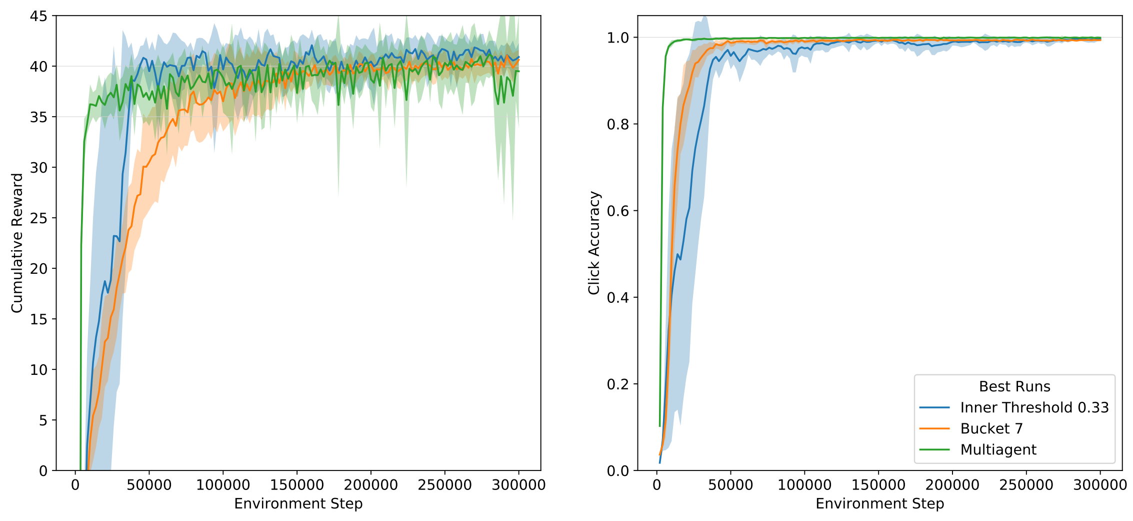


Figure 6.12: SB: Comparing the best results of each approach

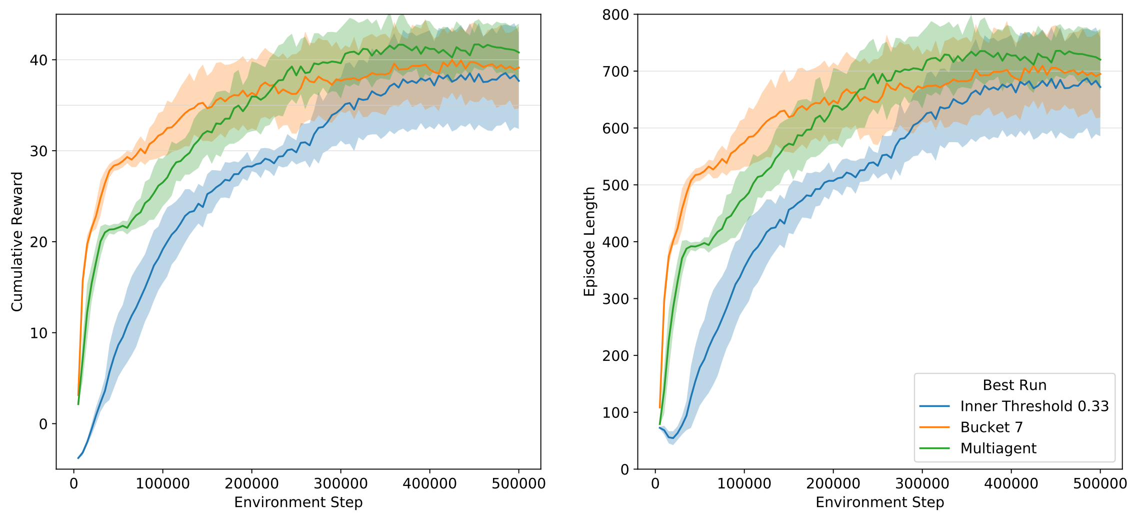


Figure 6.13: BRO: Comparing the best results of each approach

For both environments, an inner threshold of 0.33 and a bucket size of 7 achieved the best results. By comparing these results, it can be seen that the previous impressions apply again. The threshold approach is the most unstable one. However, for the SB environment, all approaches have similar performance. In terms of the click accuracy, the multiagent leads due to its earlier convergence time. Focusing on the BRO environment, it can be observed that the three approaches experience roughly the same variance, while the multiagent is superior to the other approaches.

6.7 Training Duration

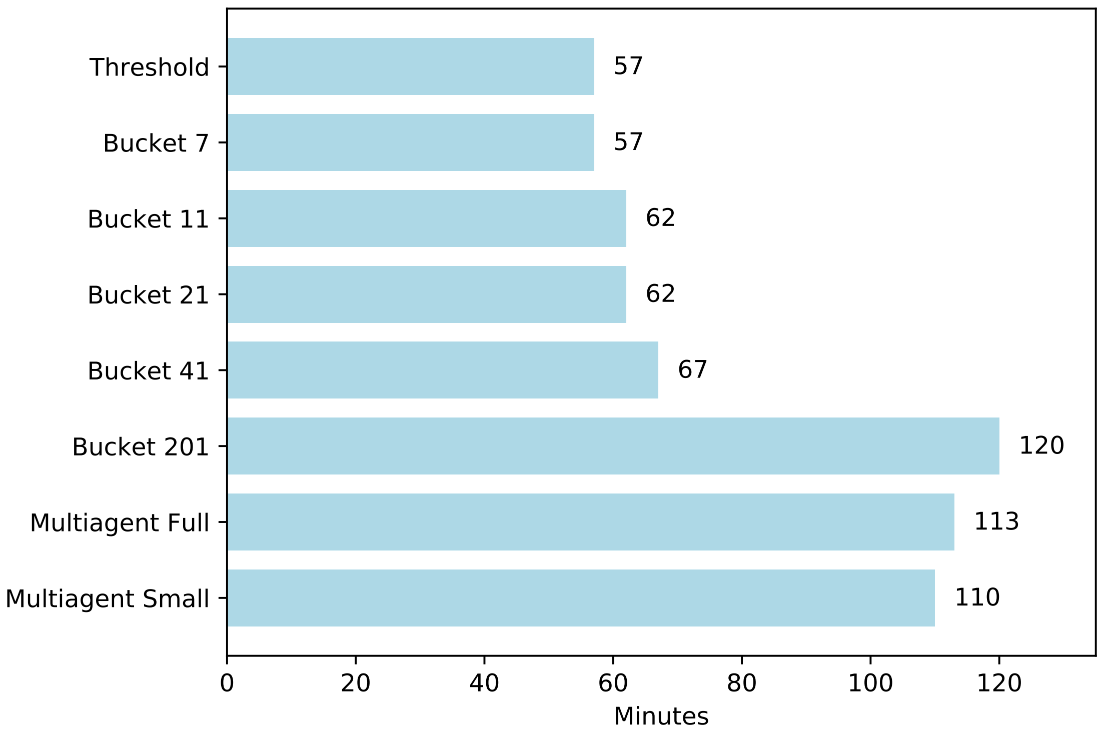


Figure 6.14: SB: Training duration

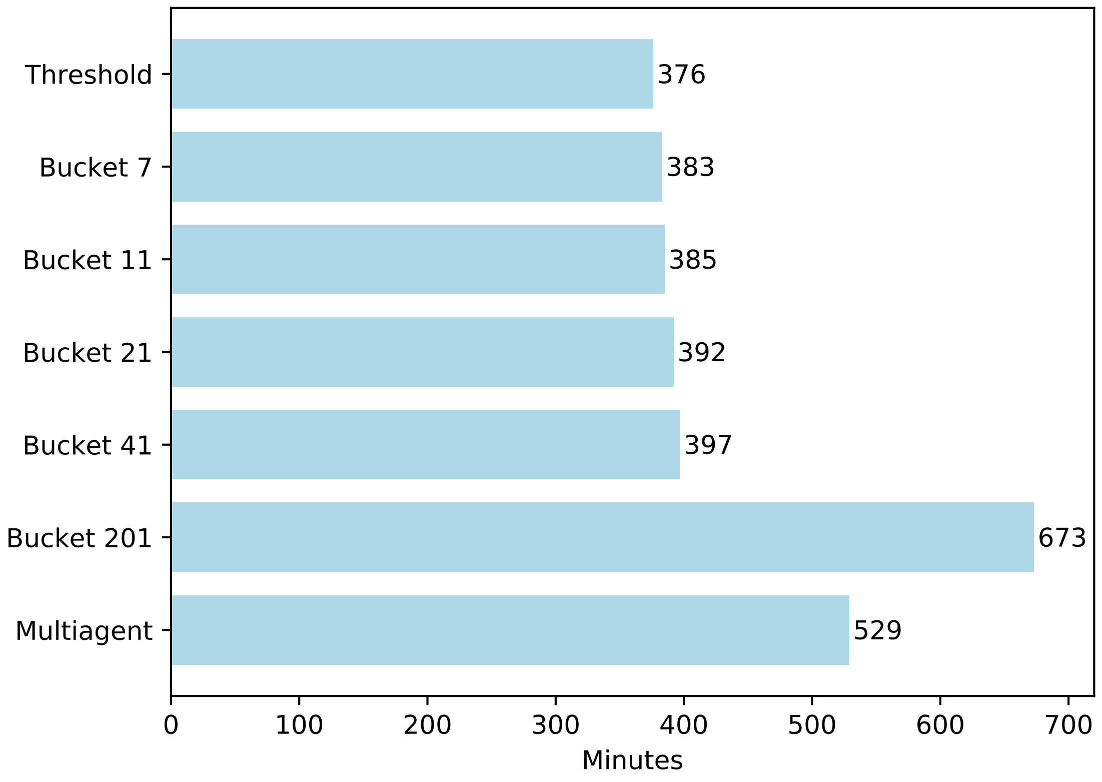


Figure 6.15: BRO: Training duration

The two bar charts from above illustrate the duration of the conducted training sessions in minutes. Concerning the threshold sessions, the average over the 14 sessions is shown. The least training time is required by the threshold approach for both environments. 120 (SB) and 673 minutes (BRO) are the longest times on the record obtained by the biggest bucket agent. On the SB environment, the multiagents used about twice as much time as the threshold approach and the smallest bucket. Not twice as much, but the multiagent took significantly longer to train in contrast to the threshold and the bucket sizes smaller than 201. Finally, it has to be stated that the threshold runs of the BRO environment have some variance as shown in the figure below. This behavior, which is later on discussed, did not occur for SB.

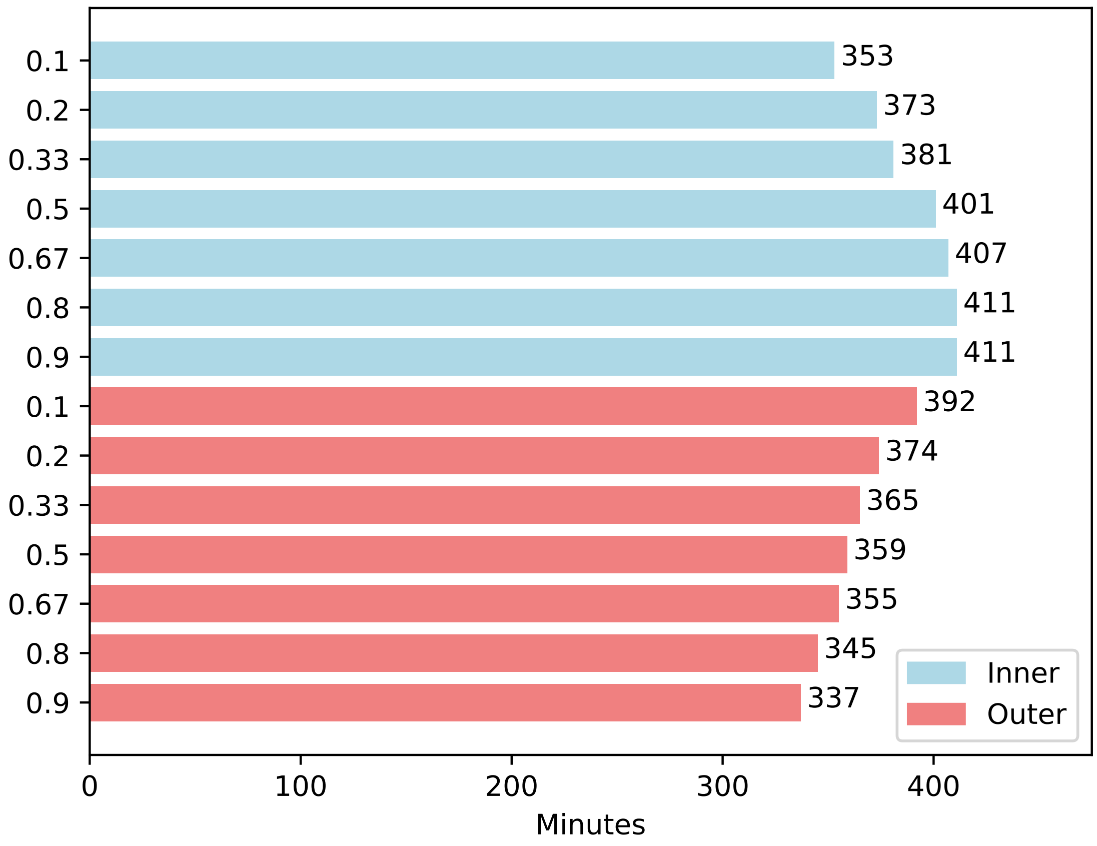


Figure 6.16: BRO: Training duration of each threshold value

6.8 Learned Behaviors

The previously compared best results (Section 6.6.2) are featured in this video[6]. Descriptions of these results are provided by the next two subsections.

6.8.1 Shooting Birds

A distinction between the showcased behaviors cannot be made. Given all approaches, the agent rapidly shoots down the birds accurately. Further, it can be observed that once no birds are remaining on one side of the environment, the agent moves to the other one, even though the agent's raycasts do not reach the other side to perceive birds.

Concerning the agent's average speed, the threshold approach moves with a speed of about 43 (horizontal movement) and 28 (vertical). About 47 and 44 are measured during the bucket approach. Finally, the multiagent travels at the speed of about 43 and 30. The maximum speed for each axis is 50.

6.8.2 Beastly Rivals Onslaught

All of BRO's recorded results have some observations in common. First, the agent clicks very frequently and therefore makes the character follow the agent. Second, keeping the beast most of the time in the slow sphere is tried by the agent. Also, the agent sticks to the edge of the pitch to let the character teleport or move to the edge and thus keep the distance to the beast as high as possible. Blinking is usually used once the beast is dodged. Dodging gets harder during the progression of the episode.

For the bucket and multiagent results, the agent keeps moving in circles. However, the agent, seen in the threshold results, sticks to one half of the battlefield's circumference. Also, it barely uses the blink ability in the early stages of the episode. In comparison, the multiagent uses this ability as soon as possible once the beast reached a certain velocity.

The maximum movement speed of the agent is 10. During the threshold result, the average speed on the horizontal axis is about 8. 6 is determined for the vertical one. About 7 and 7 is measured for the bucket result. The multiagent moves at a speed of about 6 on both axes.

[6]Video of the best results `https://youtu.be/f_zpNuj54PE`

6.9 Summary

Various training sessions have been conducted for all approaches on both environments. It can be stated that all approaches are capable of achieving reasonable agent behaviors. The best performance is delivered by the multiagent approach based on the selected performance measures. This comes at the cost of much longer training times in relation to the threshold approach and the less dimensional buckets. The bucket approach is rated second, because the threshold approach is unstable.

Concerning the individual results of the threshold approach, mixed results are achieved for inner and outer values. For both environments, an inner threshold of 0.33 worked best. Looking at the bucket approach, smaller buckets (7 and 11) performed best and the big one, using 201 actions, took the longest to train.

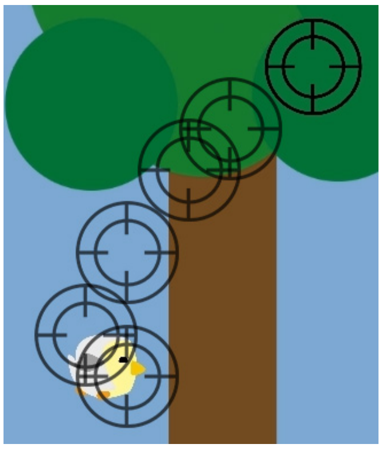


Figure 7.1: Diagonal-like movement of the agent in SB

7 Discussion

All but one of the presented results show that the taken approaches are feasible for both environments. Now, it has to be discussed why the agents successfully solve their tasks. Furthermore, there are more observations which have to be examined as well. One of them deals with the bad results caused by the one SB multiagent, which makes use of the smaller observation space. Another topic is the observed instability of the BRO environment. For this concern, three potential causes are described. Also, the implementation of the agents' locomotion is questioned, because it turned out to be biased. Throughout discussing the locomotion, a few alternatives are elaborated.

7.1 Precision and Timing of the Agent Performances

7.1.1 Shooting Birds

In the SB environment, high click accuracies were achieved for most training sessions. This indicates that the agent's actions are timed well, because accuracies close to 1.0 (i.e. agent barely misses birds) were measured most of the time, except for some threshold results. However, it can be doubted that the agent is completely precise, because it tends to move diagonally as seen in Figure 7.1. Moving diagonally is the fastest option for the agent to navigate its environment. The reason for this learned locomotion and potential solutions for this occurrence are discussed later on in Subsection 7.3. Also, the recorded average velocity of the best results suggest that the agent slows down from time to time. For example, the average velocity of the SB bucket agent is 47 and 44, while its maximum movement speed is 50 for both axes. So when needed, the agent slows down. In general, the agents of the SB environment clear their environments rapidly.

Another peculiarity can be observed in the recorded video[7] (0:13). It can be seen that the agent hits partially obscured birds. Unfortunately, this cannot be considered as valid outcome, because a bug in the implementation was detected, which allows the agent to shoot through obstacles (e.g. trees).

To provide a greater challenge to the agents, a small test was conducted on an SB environment, which only spawns small birds. The best resulted agents were able to solve this task accurately as well. Once birds are spawned at half of the size of the small ones, the agents have significant difficulty in shooting these down. This is most likely due to an overfitted behavior, because the agent did not encounter smaller birds during its training. It can be further reasoned that the observation space is not rich enough. For instance, the perceiving raycasts are more likely to loose track of these smaller targets.

Further thoughts deal with the complexity of the environment. The observation space includes the information of whether a bird is hovered or not. In conjunction with making decisions every frame, it is easy for the agent to learn when to shoot or when not to shoot. Observing the environment based on images would make it a much more complex task to solve for the agent. This will be considered at the end of this thesis as future work.

7.1.2 Failed Multiagent in Shooting Birds

The one SB multiagent, which only observes four values (e.g. hovered entity) of its environment, produced succeeding and failing outcomes. This heavy instability could be due to the negative reward of missing birds. While exploring the environment and therefore missing frequently, the agent could learn a behavior, which makes it too afraid of shooting at all. In comparison to using the full observation space, including the information of the perceiving raycasts might help the agent to reason the bad outcome of missing.

Another drawback could be the omitted negative reward signal for the continuous agent, which punishes the agent for missing its targets. Initially, it was considered that the discrete agent is the only one who could be blamed for missing. However, the discrete agent is dependent on the continuous agent's locomotion. More difficulties could arise if the discrete agent acts right after carrying out the movement of the continuous agent. Both receive their observations on the same timestep, but if the movement is done before clicking, the state of the environment is altered and therefore could make the discrete agent miss.

To ensure that no bugs interfere and the agents do not ruin each other's timing, a scripted heuristic was utilized to replace the discrete multiagent. This heuristic simply

[7] `https://youtu.be/f_zpNuj54PE?t=13`

triggers shots once a bird is hovered. Training a combination of the continuous agent and the heuristic created a successful behavior. Thus, the influence of bugs and an unfortunate sequence of the multiagent's actions is negated.

7.1.3 Beastly Rivals Onslaught

It is difficult to tell if the agent acts precisely in BRO, because it does not have to click on distinctive targets like a character. But it can be argued that the agent moves rather imprecisely, because it moves continuously on the edge of the pitch. The bounds of the pitch can be understood as an invisible wall. Once the agent decides to keep moving into this wall, it will rub itself forward along the wall.

This outcome can be further explained by another observation, which is about the agent's high frequency of selecting movement locations (i.e. clicking). As the character keeps following the agent, due to its high clicking frequency, it seems to be easier to survive by staying on the pitch's circumference. A behavior was not observed where the agent moves straight to the other side of the pitch, which could be an eligible strategy, but is only achieved with less clicks. For example, the character is commanded to move to the right side of the battlefield. After this command, the agent moves to the opposite site, while not executing any further clicks. Once it reaches a certain distance, the agent could trigger the blink ability to let the character dodge the beast. In the case of continuously clicking, the character would most likely run into the beast.

At least some evidence can be given to a well-timed usage of the blink ability. As observed in the video, the agent usually commands the character to blink once it dodged the beast. In particular the multiagent does not utilize this action until the beast gets too fast and accurate. The overall impression is that the agent learned to not waste this action.

A future consideration is to implement the whole game of BRO and therefore to challenge the agents in clicking on moving foes.

7.2 Instability of the BRO Environment

As seen in the results section, instability was observed for the BRO environment. A reason behind this outcome is hard to determine, but there are a few leads which can be tracked down.

One approach, to achieve more stable results, is to further optimize the hyperparameters. Especially the number of hidden units (20), in comparison to the observation space (21 x 3), leaves the impression of being too small. However, the results did not improve by increasing the number of hidden units to 80.

Another less significant cause of instability could be the stochastic nature of the environment. The positions of all game entities are randomized at the beginning of each episode. Therefore, it is possible, but unlikely, that the beast spawns right next to the character and kills it immediately. Modifying the logics of the random placement of the game entities should mitigate this minor threat.

The biggest question arises when looking at the training duration of the run experiments, especially the threshold sessions, which vary greatly. It was rather expected to observe a similar training duration for every threshold value, because it does not affect the structure of the neural network or the behavior of the learning algorithm. A reason could be that the server is lacking performance. In the game engine Unity, the performance affects how the physics operate. If performance starts lacking, physics may become unreliable. To verify this cause, the training was repeated for a time scale of 10 and 50. Surprisingly, the same results were achieved. It was expected that the training time goes by slower, while running on a time scale of 10. Also, the number of concurrent training sessions was decreased to 5, which led to the same results. Further investigations, like searching for bugs and optimizing the training parameters, have to be done in the future.

7.3 Biased Locomotion Actions

The locomotion of the agents are based on two continuous actions. One determines the velocity on the x-axis and the other one on the y-axis. Therefore, the fastest velocity is accomplished by choosing a magnitude of 1 for each axis. As a consequence, the agent is biased and thus learns to move diagonally most of the time as seen in Figure 7.1, because this is the fastest option to navigate its environment. Due to this behavior, the agent is less precise or takes longer to achieve its goal, while appearing less authentic to observers. Figure 7.2 illustrates how the decision is carried out for the agent's locomotion.

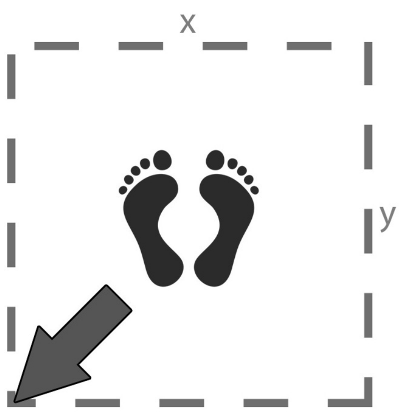


Figure 7.2: The agent's locomotion is driven by the velocities on the horizontal and vertical axis

To overcome this bias, a few alternatives can be considered. One alternative could be to use one continuous action to rotate the agent and a second one to select a speed to push the agent forward (Figure 7.3). This solution does not align with this thesis' goals, because a computer mouse is not intended to be rotated during its usage.

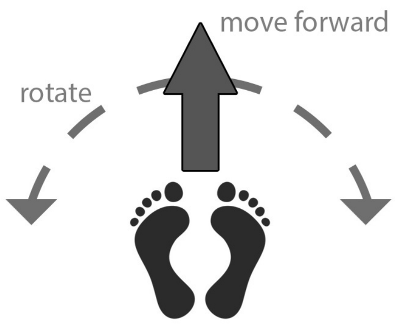


Figure 7.3: The agent rotates left and right and moves forward at a certain speed

Another option considers the discretization approach, which was taken by OpenAI for DotA as seen in Section 3.1. Following their strategy, the agent selects a grid cell to determine a position to move to. The grid cells are represented by two discrete action branches. An additional continuous action is in charge of selecting a movement speed. However, this is a valid solution for enabling an agent to solve such environments, but not to behave like a thumbstick of a gamepad or a computer mouse.

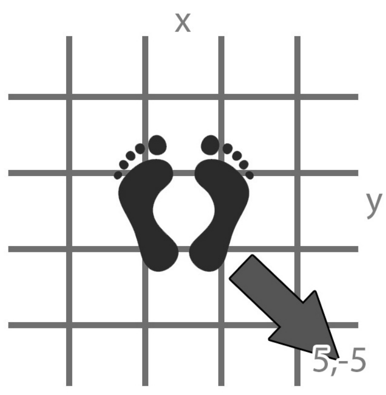


Figure 7.4: The agent moves at a certain speed towards a chosen grid cell

The last approach could make use of one continuous action to derive a direction from a position on a circle's circumference. This way, the action would determine an angle. A second one would choose the movement speed. An exemplary illustration is given by Figure 7.5. In the future, it has to be tested if the agent is capable of learning an appropriate behavior using this action space setup.

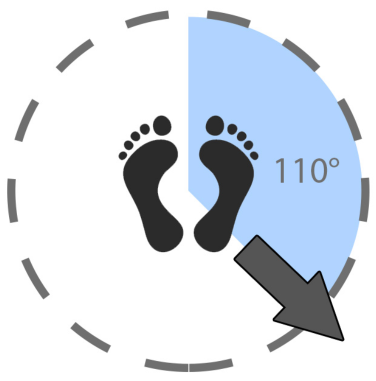


Figure 7.5: The agent selects an angle as moving direction and moves at a certain speed towards this direction

8 Conclusion

Throughout this project, three approaches for enabling concurrent discrete and continuous actions in DRL were examined. The threshold approach discretizes a continuous action to achieve the behavior of a discrete one. Multiple discrete actions replace a continuous one based on the bucket approach. Lastly, the multiagent combines several agents to support discrete and continuous actions concurrently. These approaches were challenged by two novel environments, where the agent is in charge of controlling a mouse cursor to play a video game. As for the outcome of the conducted experiments, it can be stated that all approaches achieved reasonably trained agents, which were capable of reaching their goals. Out of the three approaches, the bucket approach is the most recommendable, as it achieved stable training results while taking the least time to train. Another advantage is the trivial implementation of bucketed actions. Multiagents performed better in the end, but come at the cost of much longer training times, which is related to their redundant observation space. The threshold approach is inferior to its competitors because it showed a lot of variance.

In conclusion, the underlying contributions establish the first steps of solving the problem of enabling both action kinds concurrently. More investigations have to be done to stress test the explored approaches on more complex environments, because both of the implemented environments are simple and have their drawbacks. For instance, the agent's locomotion should be changed for both environments, because it turned out to be biased. As a result, the agents usually traveled diagonally through their environment, because its their fastest option. In addition, the agent, in the simplified version of BRO, learned an appropriate timing of its blink ability, but does not move the cursor precisely. Nevertheless, SB and BRO have shown to be suitable environments to get started on the matter of this thesis.

8.1 Outlook

In the future, plenty of work is left to advance the topic of this thesis. First of all, the discussed problems have to be solved. This is considered for the biased locomotion as well as the instability of the BRO environment and its suspicious threshold training durations. After that, the goal is to further approach the motivation of this thesis, which deals with agents playing video games similarly to humans. This includes action spaces mimicking human input devices and observation spaces, which comprise visual observations of the environment. The later aspect has been omitted due to limited computational resources and time. Because of the shown feasibility of the three utilized approaches, visual observations can be tackled next. Though one caveat comes with the use of visual observations; Harmer et al. (2018) included images of the dimension 128x128x3 in their environment. It is to be examined if greater image resolutions are necessary to capture a mouse cursor. If click-able targets are too challenging at first, it could be helpful to incorporate curriculum learning [Bengio et al., 2009, Matiisen et al., 2017]. This way, targets could be initially scaled up and as the agent improves, targets can shrink down over the course of several lessons. To get started on visual observations in combination with concurrent discrete and continuous actions, the SB environment could be a good use-case because it is fast to train.

Concerning BRO, building an environment that features the whole game is an interesting but challenging project. As multiple players competitively play against each other, this challenge is concerned with the context of multiagents, which is a distinctive field of research in RL. So intensifying the idea of agents playing BRO draws further problems and literature into account. For example, Tampuu et al (2017) showed how agents can play *Pong* in a competitive or cooperative setting.

Finally, training agents more quickly and more efficiently is another general concern. As visual observations increase the complexity of a model and thus need more time to be trained, distributed systems, as used by OpenAI Five [OpenAI, 2018b], should be considered to increase the pace of developing the environments and their agents. Utilizing the architecture of *IMPALA* is another possibility or a source for inspirations [Espeholt et al., 2018].

8.2 Critical Reflection

Initially, this thesis was supposed to develop a solution to natively allow the usage of concurrent discrete and continuous actions. This would have required developing a novel neural net architecture and coming up with an objective function. Due to a lack of knowledge of DRL fundamentals and the usage of TensorFlow, the scope of this project had to be adjusted. The focus shifted to building suitable environments and applying more simple approaches. After writing the fundamental knowledge chapter, PPO was understood in theory, but was still too challenging to practically implement it with TensorFlow. Therefore, developing a native approach was unfeasible given this skill base.

The lost time could have been beneficial to further elaborate the process of building RL environments, because the composition of the observation space, action space, and reward signals were very briefly described and reasoned. A complete section, dealing with options and examples for these three aspects, could have been possible and could have added more value to the underlying work. Also, the locomotion bias could have been detected much earlier. Another topic, which could have gained more attention, is the field of multiagents, because it is related to one of the three examined approaches.

Finally, it can be questioned if the implemented BRO environment is a suitable use-case, because it does not really challenge the agent to act precisely. Instead of implementing a slow sphere, clicking on the beast to trigger some ability, or to collect power-ups (e.g. temporary vulnerability of the character), could have been a better game mechanic for benchmarking the approaches.

In conclusion, the objectives of this thesis were met. Ultimately, the provided contributions can be utilized as reasonable foundation for promising subsequent projects.

References

[Arulkumaran et al., 2017] Arulkumaran, K., Deisenroth, M. P., Brundage, M., and Bharath, A. A. (2017). A brief survey of deep reinforcement learning. *CoRR*, abs/1708.05866.

[Bansal et al., 2017] Bansal, T., Pachocki, J., Sidor, S., Sutskever, I., and Mordatch, I. (2017). Emergent complexity via multi-agent competition. *CoRR*, abs/1710.03748.

[Bengio et al., 2009] Bengio, Y., Louradour, J., Collobert, R., and Weston, J. (2009). Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 41–48, New York, NY, USA. ACM.

[Chou et al., 2017] Chou, P.-W., Maturana, D., and Scherer, S. (2017). Improving stochastic policy gradients in continuous control with deep reinforcement learning using the beta distribution. In Precup, D. and Teh, Y. W., editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 834–843, International Convention Centre, Sydney, Australia. PMLR.

[Espeholt et al., 2018] Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., Doron, Y., Firoiu, V., Harley, T., Dunning, I., Legg, S., and Kavukcuoglu, K. (2018). IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. *ArXiv e-prints*.

[Fujita and Maeda, 2018] Fujita, Y. and Maeda, S. (2018). Clipped action policy gradient. *CoRR*, abs/1802.07564.

[Harmer et al., 2018] Harmer, J., Gisslén, L., Holst, H., Bergdahl, J., Olsson, T., Sjöö, K., and Nordin, M. (2018). Imitation learning with concurrent actions in 3d games. *CoRR*, abs/1803.05402.

[Juliani et al., 2018] Juliani, A., Berges, V.-P., Vckay, E., Gao, Y., Henry, H., Mattar, M., and Lange, D. (2018). Unity: A general platform for intelligent agents. **arXiv preprint arXiv:1809.02627.** `https://github.com/Unity-Technologies/ml-agents`.

[Li, 2017] Li, Y. (2017). Deep reinforcement learning: An overview. *CoRR*, abs/1701.07274.

[Lillicrap et al., 2015] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971.

[Matiisen et al., 2017] Matiisen, T., Oliver, A., Cohen, T., and Schulman, J. (2017). Teacher-student curriculum learning. *CoRR*, abs/1707.00183.

[Mnih et al., 2016] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *CoRR*, abs/1602.01783.

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602.

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). *Human-level control through deep reinforcement learning. Nature*, 518(7540):529–533. Letter.

[OpenAI, 2017] OpenAI (2017). More on dota 2. Available at `https://blog.openai.com/more-on-dota-2/` retrieved October 30, 2018.

[OpenAI, 2018a] OpenAI (2018a). The international 2018: Results. Available at `https://blog.openai.com/the-international-2018-results/` retrieved October 30, 2018.

[OpenAI, 2018b] OpenAI (2018b). Openai five. Available at `https://blog.openai.com/openai-five/` retrieved October 30, 2018.

[OpenAI, 2018c] OpenAI (2018c). Openai five benchmark: Results. Available at `https://blog.openai.com/openai-five-benchmark-results/` retrieved October 30, 2018.

[Schulman et al., 2015a] Schulman, J., Levine, S., Moritz, P., Jordan, M. I., and Abbeel, P. (2015a). Trust region policy optimization. *CoRR*, abs/1502.05477.

[Schulman et al., 2015b] Schulman, J., Moritz, P., Levine, S., Jordan, M. I., and Abbeel, P. (2015b). High-dimensional continuous control using generalized advantage estimation. *CoRR*, abs/1506.02438.

[Schulman et al., 2017] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms. *CoRR*, abs/1707.06347.

[Sharma et al., 2017] Sharma, S., Suresh, A., Ramesh, R., and Ravindran, B. (2017). Learning to factor policies and action-value functions: Factored action space representations for deep reinforcement learning. *CoRR*, abs/1705.07269.

[Silver, 2015] Silver, D. (2015). Lecture 1: Introduction to reinforcement learning. Available at `http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/intro_RL.pdf` retrieved October 29, 2018.

[Silver et al., 2017] Silver, D., Schrittwieser, J., Simonyan, K., ioannis Antonoglou, Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nature Publishing Group*, 550.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT Press, 2nd edition.

[Tampuu et al., 2015] Tampuu, A., Matiisen, T., Kodelja, D., Kuzovkin, I., Korjus, K., Aru, J., Aru, J., and Vicente, R. (2015). Multiagent cooperation and competition with deep reinforcement learning. *CoRR*, abs/1511.08779.

[Tampuu et al., 2017] Tampuu, A., Matiisen, T., Kodelja, D., Kuzovkin, I., Korjus, K., Aru, J., Aru, J., and Vicente, R. (2017). Multiagent cooperation and competition with deep reinforcement learning. *PLOS ONE*, 12(4):1–15.

[Tavakoli et al., 2017] Tavakoli, A., Pardo, F., and Kormushev, P. (2017). Action branching architectures for deep reinforcement learning. *CoRR*, abs/1711.08946.

[Wang and Yu, 2016] Wang, H. and Yu, Y. (2016). Exploring multi-action relationship in reinforcement learning. In Booth, R. and Zhang, M.-L., editors, *PRICAI 2016: Trends in Artificial Intelligence*, pages 574–587, Cham. Springer International Publishing.

[Yannakakis and Togelius, 2018] Yannakakis, G. N. and Togelius, J. (2018). *Artificial Intelligence and Games*. Springer. `http://gameaibook.org`.

Annex

Python 3.6.3 Environment

absl-py	0.4.0	alembic	0.9.9
appdirs	1.4.3	asn1crypto	0.24.0
astor	0.7.1	async-generator	1.10
atomicwrites	1.2.1	attrs	18.1.0
Automat	0.0.0	backcall	0.1.0
beautifulsoup4	4.6.3	bleach	1.5.0
bokeh	0.12.16	certifi	2018.8.24
cffi	1.11.5	chardet	3.0.4
cloudpickle	0.5.3	conda	4.5.8
constantly	15.1.0	cryptography	2.2.1
cycler	0.10.0	Cython	0.28.5
dask	0.18.2	decorator	4.3.0
dill	0.2.8.2	docopt	0.6.2
entrypoints	0.2.3	fastcache	1.0.2
gast	0.2.0	gmpy2	2.0.8
grpcio	1.11.1	h5py	2.7.1
html5lib	0.9999999	hyperlink	17.3.1
idna	2.7	imageio	2.3.0
incremental	17.5.0	ipykernel	4.8.2
ipython	6.5.0	ipython-genutils	0.2.0
ipywidgets	7.2.1	jedi	0.12.1
Jinja2	2.10	jsonschema	2.6.0
jupyter	1.0.0	jupyter-client	5.2.3
jupyter-console	5.0.0	jupyter-core	4.4.0
jupyter-tensorboard	0.1.7	jupyterhub	0.9.2
jupyterlab	0.34.0	jupyterlab-launcher	0.13.1
kiwisolver	1.0.1	llvmlite	0.23.0
Mako	1.0.7	Markdown	2.6.11
MarkupSafe	1.0	matplotlib	2.2.3
mistune	0.8.3	mlagents	0.5.0
more-itertools	4.3.0	nbconvert	5.3.1
nbformat	4.4.0	networkx	2.1
notebook	5.6.0	numba	0.38.1
numexpr	2.6.6	numpy	1.13.3
olefile	0.45.1	packaging	17.1
pamela	0.3.0	pandas	0.23.4
pandocfilters	1.4.2	parso	0.3.1
patsy	0.5.0	pexpect	4.6.0
pickleshare	0.7.4	Pillow	5.2.0
pip	18.0	pluggy	0.7.1
prometheus-client	0.3.0	prompt-toolkit	1.0.15
protobuf	3.6.0	ptyprocess	0.6.0
py	1.6.0	pyasn1	0.4.4
pyasn1-modules	0.2.1	pycosat	0.6.3

pycparser	2.18	pycurl	7.43.0.2
Pygments	2.2.0	pyOpenSSL	18.0.0
pyparsing	2.2.0	PySocks	1.6.8
pytest	3.8.2	python-dateutil	2.7.3
python-editor	1.0.3	python-oauth2	1.0.1
pytz	2018.5	PyWavelets	0.5.2
PyYAML	3.12	pyzmq	17.1.2
qtconsole	4.4.1	requests	2.19.1
ruamel-yaml	0.15.44	scikit-image	0.14.0
scikit-learn	0.19.2	scipy	1.1.0
seaborn	0.9.0	Send2Trash	1.5.0
service-identity	17.0.0	setuptools	40.0.0
simplegeneric	0.8.1	six	1.11.0
SQLAlchemy	1.2.11	statsmodels	0.9.0
sympy	1.1.1	tensorboard	1.7.0
tensorflow	1.7.0	termcolor	1.1.0
terminado	0.8.1	testpath	0.3.1
toolz	0.9.0	tornado	5.1
traitlets	4.3.2	Twisted	18.7.0
urllib3	1.23	vincent	0.4.4
wcwidth	0.1.7	webencodings	0.5
Werkzeug	0.14.1	wheel	0.31.1
widgetsnbextension	3.2.1	xlrd	1.1.0
zope.interface	4.5.0		

Hyperparameter and Training Parameter Explanations

The next two subsections provide a brief explanation of the used training parameters and hyperparameters. Explanations of the hyperparameters are based on the documentation of Unity's ML-Agents Toolkit[8].

Training Parameters

Max Episode Length:

Maximum number of steps by an agent until it is reset.

Time Scale:

Targeted simulation speed of the build made with the game engine Unity.

Agent Count:

Number of agents that run in parallel to collect experiences for training a single policy.

Run Count:

Number of training sessions to benchmark one experiment.

Hyperparameters

Max Steps:

Max Steps indicates the maximum number of steps, which are utilized for the whole training session.

Buffer Size:

Experience tuples (observations, actions, and rewards) are stored within the buffer. The learning process starts, once the buffer is full. Old experiences are replaced by new ones.

Batch Size:

The batch size specifies the number of samples from the buffer, which are used for one iteration gradient descent.

Epochs:

The number of epochs determines how many gradient descent iterations, where each iteration samples a new batch from the buffer, are done.

[8]`https://github.com/Unity-Technologies/ml-agents/blob/d6dad11c773a8598540a8896d0d48b2489b222ae/docs/Training-PPO.md`

Time Horizon:

The time horizon states how many steps of experiences are gathered, before they are added to the buffer.

Beta:

Beta is the strength of the entropy regularization, which influences the exploration of the agent.

Gamma:

Discount factor of the cumulative reward.

Epsilon:

The clipping threshold of the policy updates are determined by epsilon.

Lambda:

Lambda is involved in computing the generalized advantage estimate.

Learning Rate:

Step-size of the updates during gradient descent.

Layers:

Number of hidden layers of the neural network model.

Hidden Units:

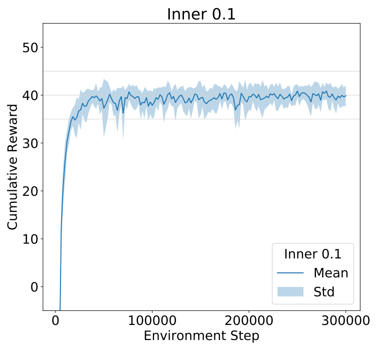
Number of hidden units on each hidden layer.

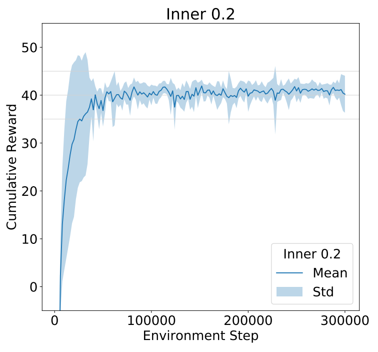
Single Results

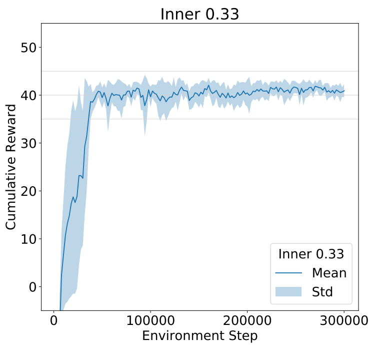
The remaining pages of the annex show every single result, which display the mean and standard deviation of each experiment. The following subsections are divided first by environments and then by approaches. The legend is found in the lower right corner. For each plot, it specifies the run experiment (e.g. inner threshold).

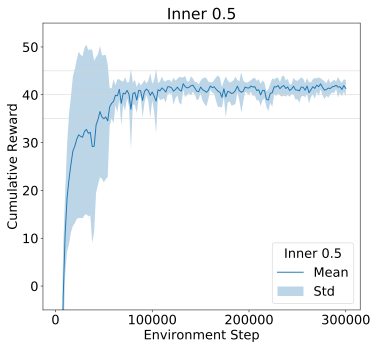
SB: Threshold

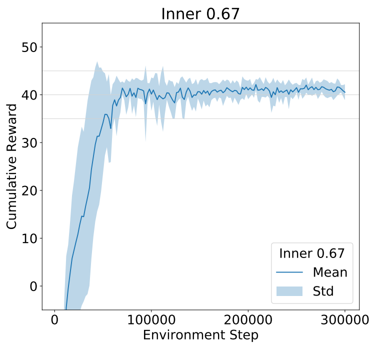
Cumulative Reward

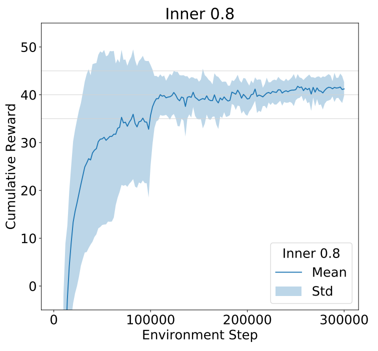


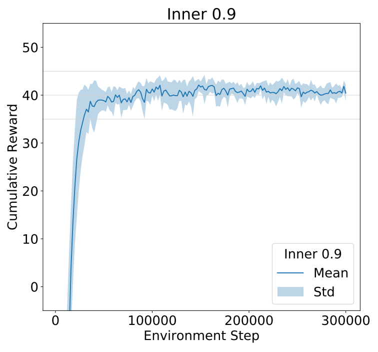


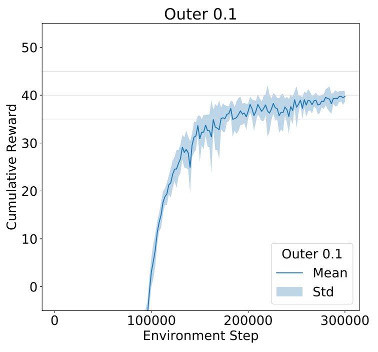


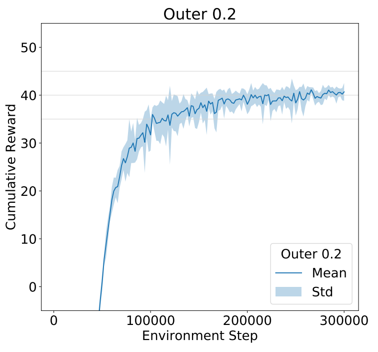


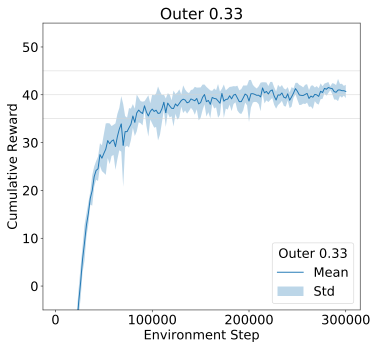


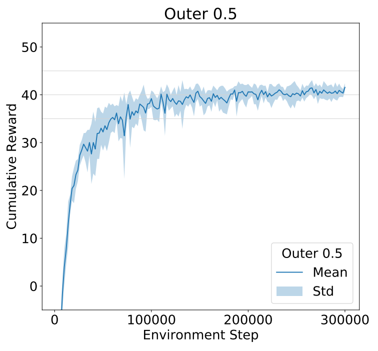


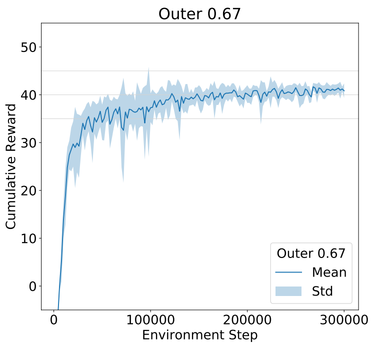


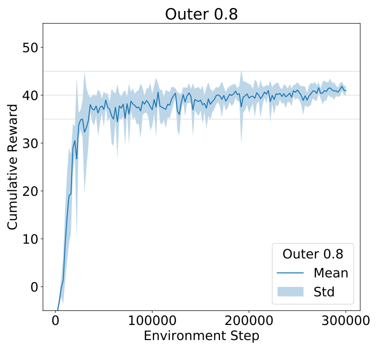


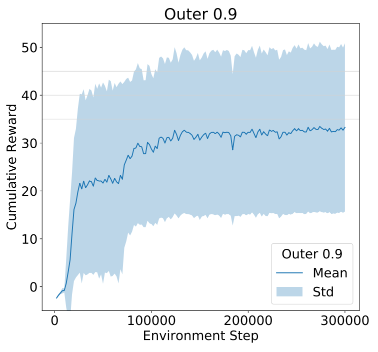




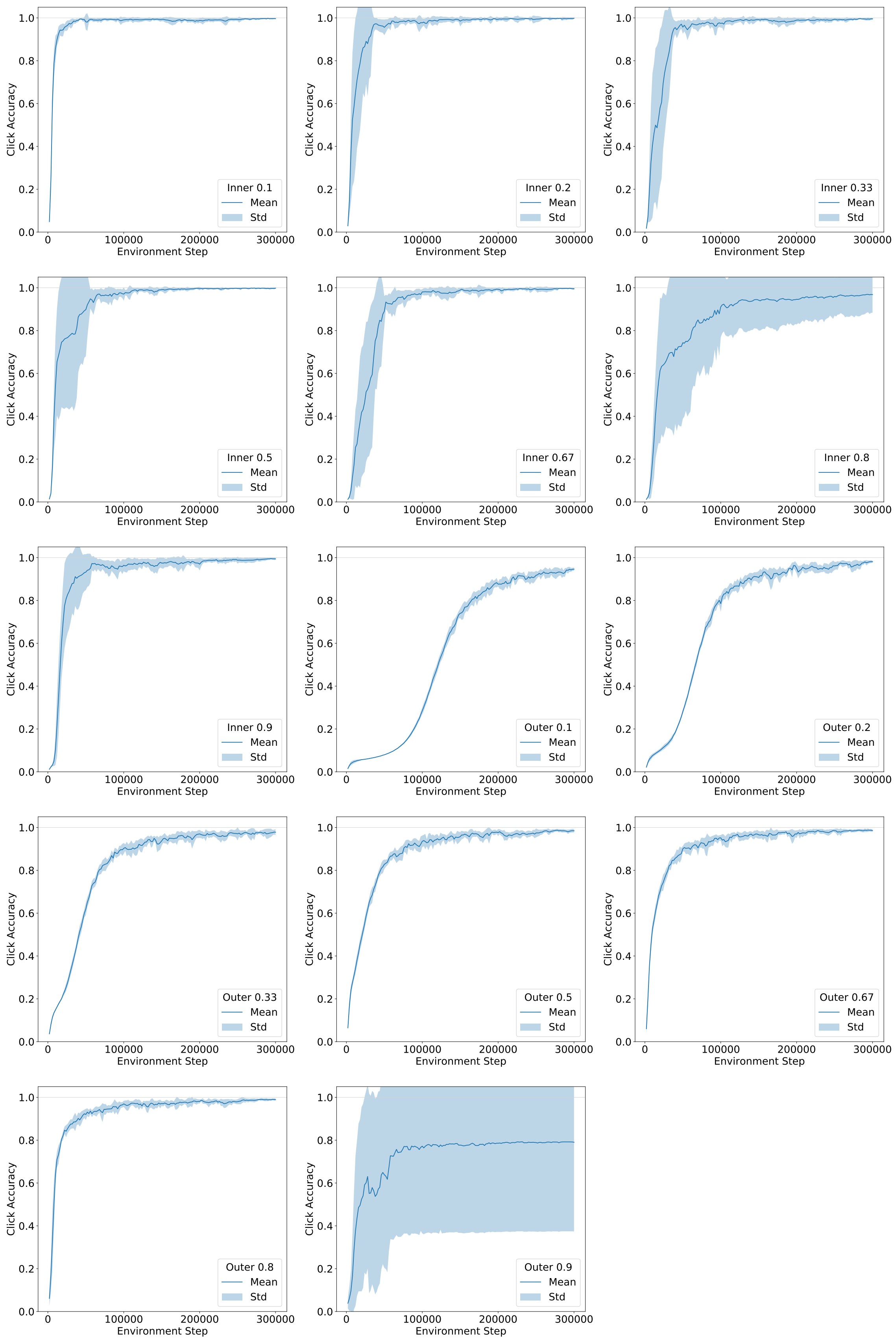






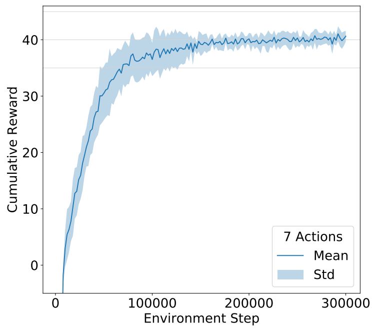


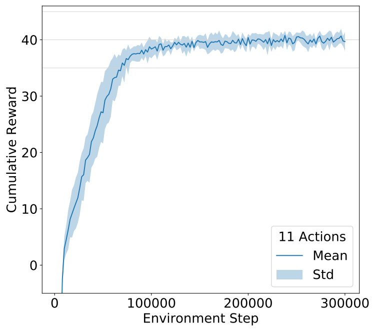
Click Accuracy

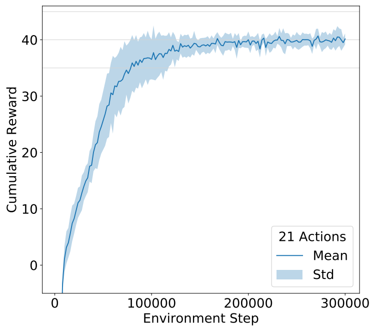


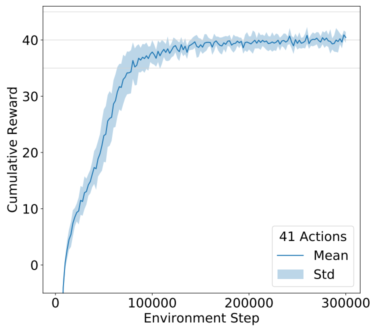
SB: Bucket

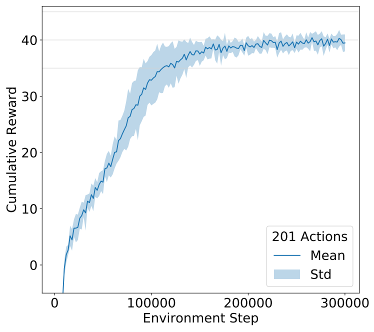
Cumulative Reward



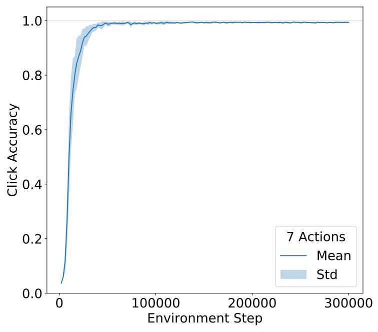


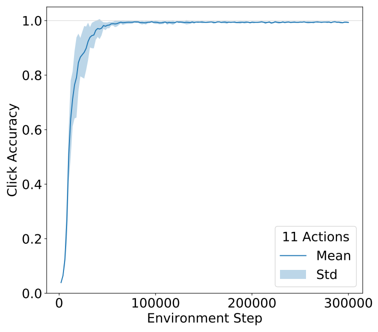


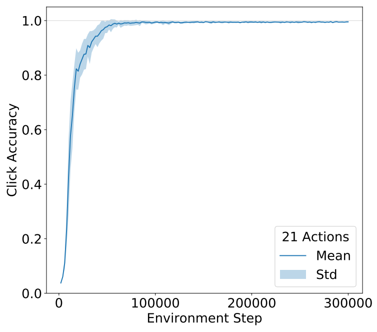


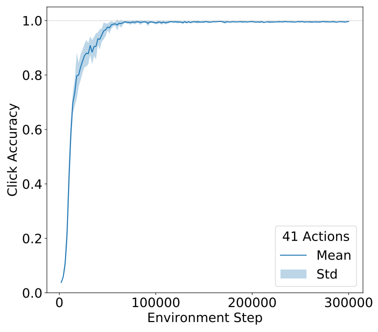


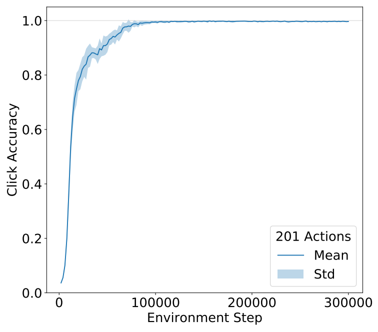
Click Accuracy





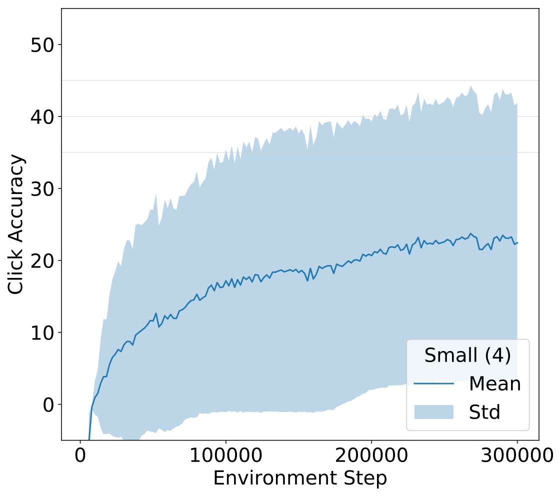


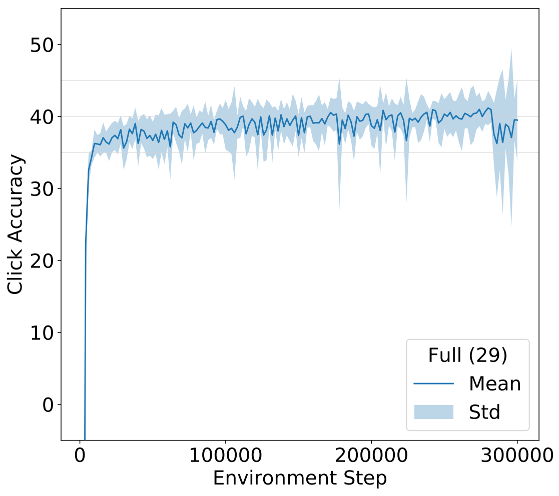




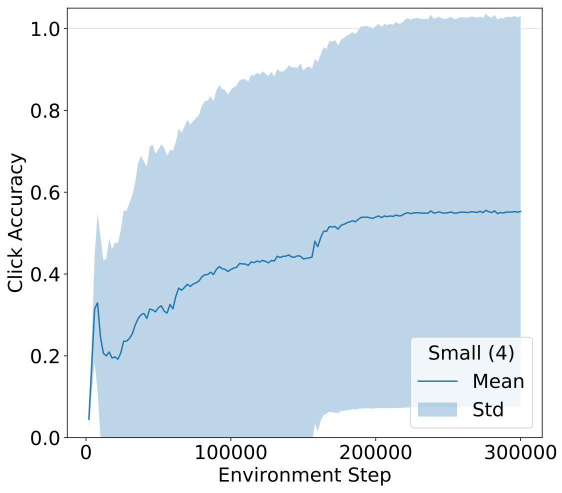
SB: Multiagent

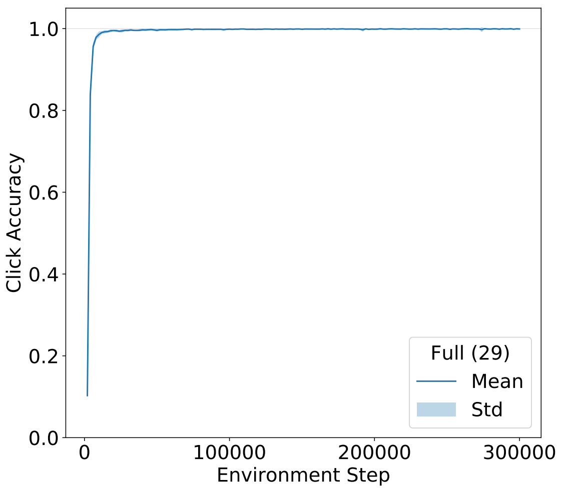
Cumulative Reward





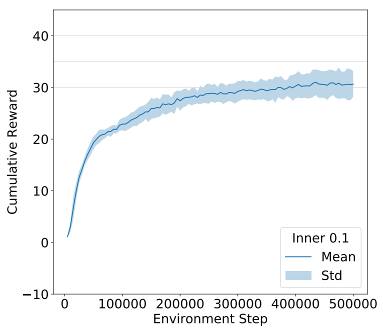
Click Accuracy

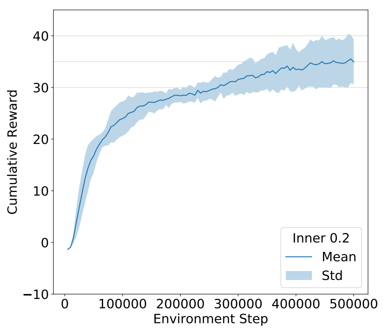


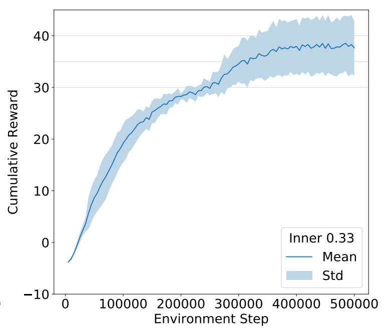


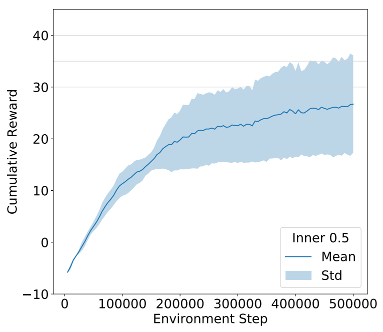
BRO: Threshold

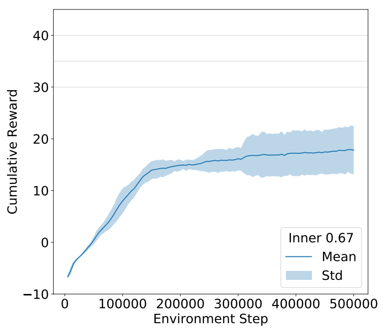
Cumulative Reward

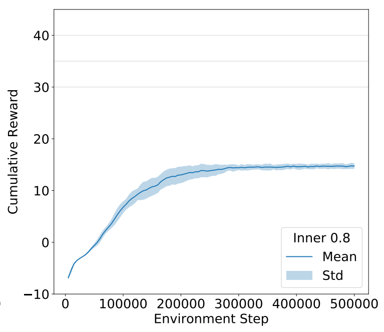


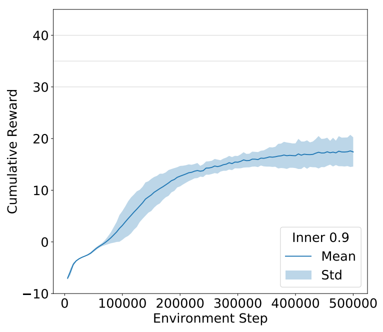


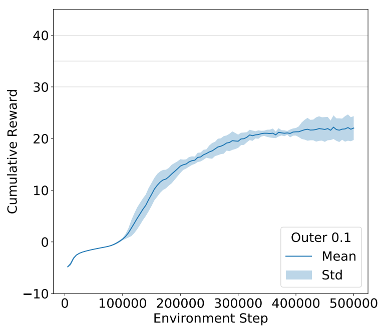


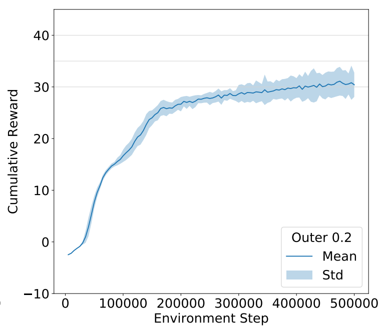


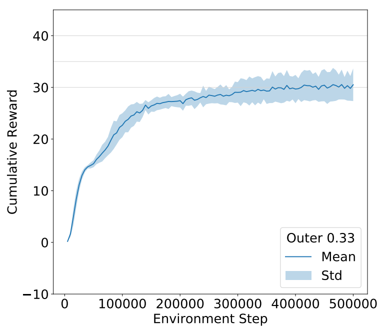


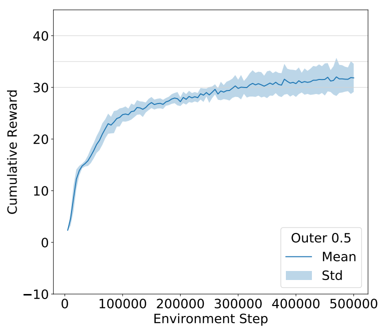


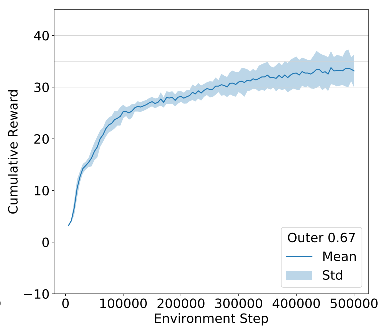


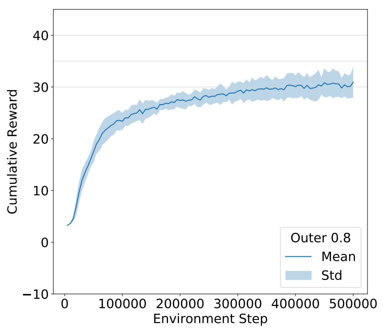


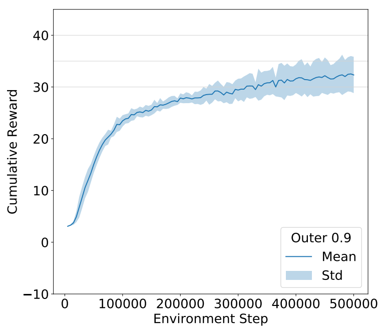




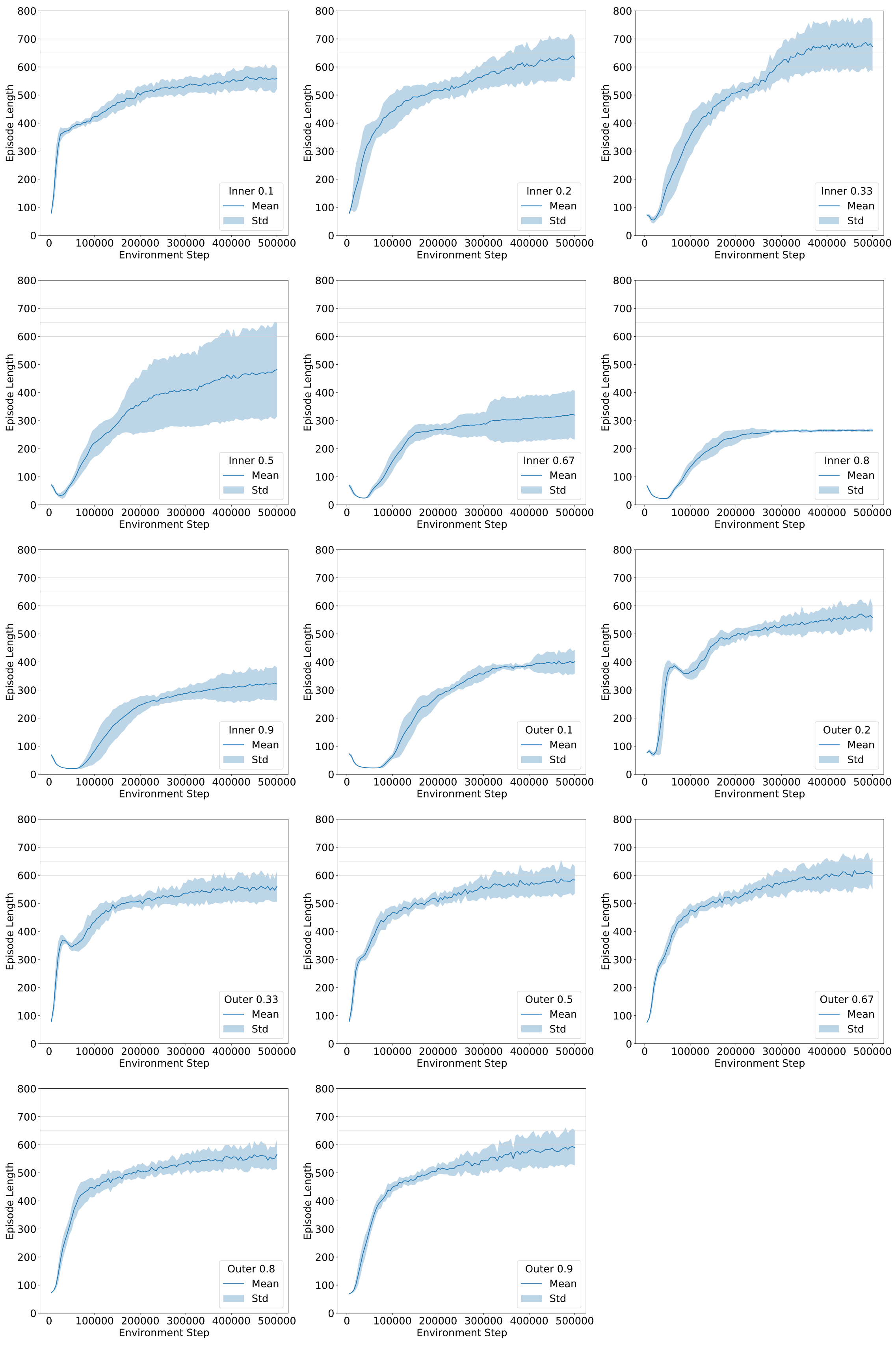






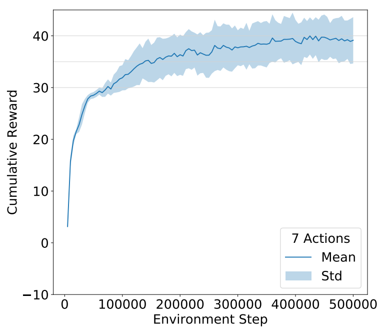


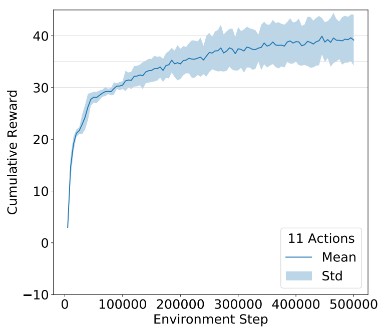
Episode Length

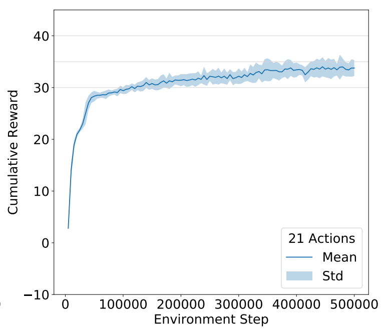


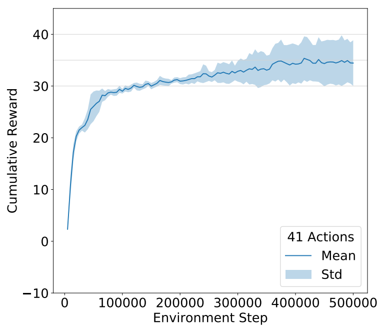
BRO: Bucket

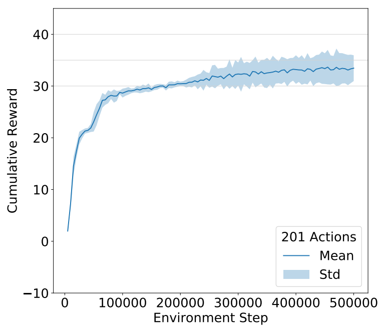
Cumulative Reward



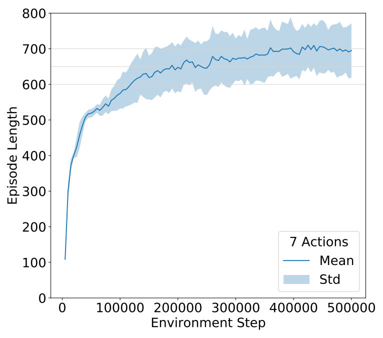


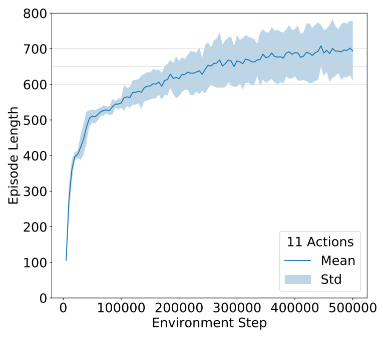


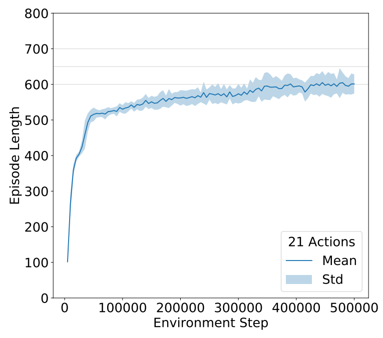


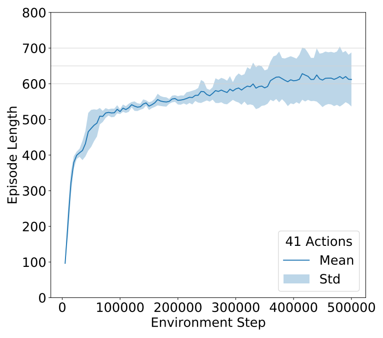


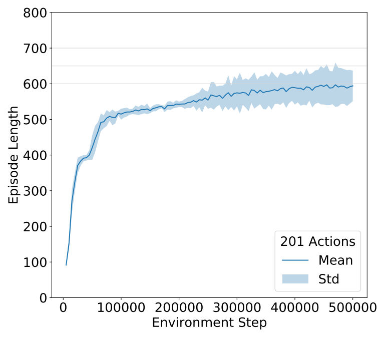
Episode Length





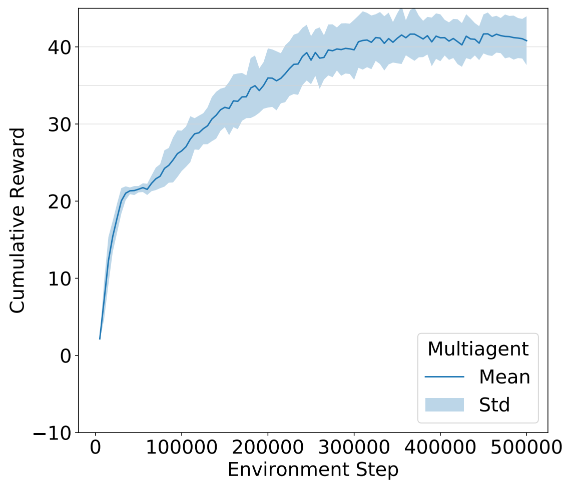




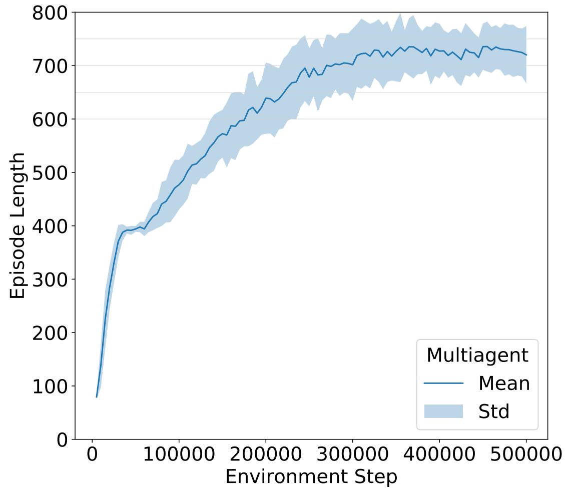


BRO: Multiagent

Cumulative Reward



Episode Length



Declaration of Authenticity