Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

Accelerating Multi-Agent Reinforcement Learning with Scalable Environments

Author:
Thomas Alner

Supervisor:
Dr. Antoine Cully

Second Marker:
Prof. Andrew Davison

PhD Supervisor:
Bryan Lim

June 19, 2023

Abstract

Competitive Multi-Agent Reinforcement Learning (MARL) tasks have proved promising mediums for learning complex and intelligent strategy. However, researchers in the field face a significant barrier to entry: the sampling complexity is even greater than with standard RL tasks. Indeed, with the agents' emergent complexity inducing multiple stages of curricula, these methods are inevitably more computationally expensive. Unfortunately, unable to navigate this impasse with existing solutions and hardware, research in the area has diminished over recent years.

However, the journey of massive parallelism in Deep Reinforcement Learning has now culminated in engines providing end-to-end training pipelines on a single device. Whilst this has been utilised for accelerating control task simulation and many RL algorithms, there has yet to be an investigation or implementation pertaining to the multi-agent scenario.

We present MAax: a novel and accelerated framework for creating rich and diverse simulation environments for MARL tasks. With the full exposure of independent domain randomisation parameters to users, we facilitate incremental complexity scaling over a range of modular entities. We demonstrate the capabilities by tackling a sample task, Hide-And-Seek, and fully implement the game's mechanics in JAX. By leveraging massive parallelism, we find our approach surpasses the performance of existing solutions by a significant margin. Furthermore, we evidence the scalability of our solution and how this performance difference extends to several orders of magnitude on specialised hardware.

Acknowledgements

I'd like to express my gratitude to my supervisor, Dr. Antoine Cully, for lending his guidance and expertise throughout this project.

I also wish to thank the entirety of the Adaptive and Intelligent Robotics Lab for welcoming me into their workspace, especially the ever-enthusiastic Bryan Lim, who has offered me continual support throughout the project.

Contents

1 Introduction **5**

1.1 Motivation . . . 5

1.2 Contributions . . . 5

2 Preliminaries **7**

2.1 Machine Learning . . . 7

2.1.1 Artificial Neural Networks . . . 7

2.2 Reinforcement Learning . . . 8

2.2.1 Markov Decision Processes . . . 8

2.2.2 Deep Reinforcement Learning . . . 10

2.2.3 Multi-Agent Reinforcement Learning . . . 12

2.3 Physical Simulation . . . 13

3 Related Work **15**

3.1 Learning Complex Strategy . . . 15

3.1.1 Self-Play and Curricula . . . 15

3.2 Procedural Content Generation . . . 17

3.2.1 Applications to Machine Learning . . . 17

3.2.2 Domain Randomisation . . . 17

3.2.3 Creating Environments . . . 18

3.3 Accelerated RL via Massive Parallelism . . . 19

3.4 Summary . . . 20

3.4.1 Our Approach . . . 21

4 Procedural Environment Generation **22**

4.1 Problem Statement . . . 22

4.2 Initial Work . . . 23

4.2.1 Object Distribution - Brax V1 . . . 23

4.2.2 Early Benchmarks . . . 24
4.3 World Generation . . . 25
4.3.1 Brax-V2 Update . . . 25
4.3.2 MuJoCo-Worldgen: Full Acknowledgements[64] . . . 25
4.3.3 Modular Environments . . . 27

5 Tasks and Mechanics **30**
5.1 Mechanics . . . 30
5.1.1 Wrappers . . . 30
5.2 Tasks . . . 30
5.2.1 Hide-and-Seek . . . 30

6 Evaluation **34**
6.1 Modularity . . . 34
6.2 Scalability . . . 34
6.3 Performance . . . 35
6.3.1 Internal Analysis . . . 35
6.3.2 Engine Comparison . . . 37
6.4 Usability . . . 38

7 Conclusion **39**
7.1 Summary . . . 39
7.2 Ethical Considerations . . . 39
7.3 Future Work . . . 40

A Domain Randomisation Parameters **41**

B Experimentation Resources **43**

List of Figures

2.1 Structure of a artificial neuron . . . 8
2.2 Example architecture of a Artifical Neural Network containing a single hidden layer 8
2.3 The agent-environment interaction loop in Reinforcement Learning [13] . . . 9
2.4 Abstract architecture of a Deep Q Network . . . 11
2.5 The RLLab control environments, later adapted to form OpenAi Gym's MuJoCo tasks (Duan et al., 2016) [26] . . . 13

3.1 Emergent skill progression in multi-agent hide-and-seek[4] . . . 16
3.2 The OpenAI Rapid Framework as presented in 2018[40] . . . 17
3.3 Comparison of the domain randomised renders with the real world test images (Tobin et al., 2019) [45] . . . 18
3.4 Performance when leveraging massive parallelism for both simulation (Figure 3.4a) and training (Figure 3.4b) in Brax.[9] . . . 20

4.1 Example MAax environment. 2 hiders, 2 seekers, 2 boxes, 1 ramp, quadrant walls . 22
4.2 Randomised environment in Brax V1 via a minor modification to Algorithm 2 . . . 23
4.3 Performance of initialising a `System` from `Config` and executing a 1000 timestep rollout – Figure 4.3a. For context, we include Figure 4.3b to illustrate the average object count resulting from Poisson point sampling with separation $r = 5$. . . . 24
4.4 Performance between poisson sampling and naive rejection based sampling for object distribution, with identical rejection threshold $k = 30$ . . . 25
4.5 State and World-building interaction of modular MAax environments. During transitions, each module sequentially adds observations to the state. At the world-building stage, object modules append their geoms to the floor object. . . . 27
4.6 Sample environment for each of the wall scenarios. . . . 28
4.7 Environments illustrating a range and combination of placement functions. . . . 29

5.1 How MAax observations are embedded for policy training. Supplementary options such as door locations and remaining preparation time are omitted for simplicty. . 32

6.1 Performance of rollout execution for various object types and quantities. We use measure the non-batched JIT compiled run-time with $\mathcal{T} = 1000$. The environments also contain 2 hiders and 2 seekers both to provide the existence of an action space, and place the result within context of our task. . . . 35
6.2 Rendering of the standard MAax environment $\mathcal{M}$ with the external walls stripped. This represents the 'no walls' environment referenced in later experiments. . . . 36
6.3 Comparison of average total simulation runtime(Figure 6.3a) and throughput(Figure 6.3b) between MAax environments and the Brax Ant task with increasing batch size. Note that both axes use a logarithmic scale. . . . 37
6.4 Comparison of average total simulation runtime(Figure 6.3a) and throughput(Figure 6.3b) between MAax environments and an equivalent MuJoCo system with increasing batch size. Note that both axes use a logarithmic scale. . . . 37

Chapter 1

Introduction

1.1 Motivation

In recent years, Reinforcement Learning (RL) has been applied across various domains such as healthcare[1], robotics[2] and gaming[3]. Unsurprisingly, of particular interest to researchers are agents that exhibit human intelligence by interacting with the objects around them. For many of these explicit and physically-grounded problems, training a single RL agent is sufficient to produce extremely promising results. However, this approach fails to extend to more complex and open-ended tasks, even under the guidance of unsupervised exploration incentives, such as intrinsic motivation[4].

In many cases, the complexity of a trained agent is directly contingent on the complexity of the environment[5]. However, the skills of such independent agents will always remain bounded by the task description. Instead, through introducing multiple agents, and competition and cooperation between them, they are able to learn far more complex behaviour [6, 7]. Attacking problems in this way loosens the bindings on learning induced by the task; these competing agents enforce new challenges upon each other over time and thus a new pressure to adapt, improving both the quality and diversity of solutions.

Such multi-agent scenarios demand even higher sampling complexity than single agent reinforcement learning tasks: not only do we have multiple actors, but each must progress through the many phases of implicit curricula. As a result, research into the area is comparatively far more limited, especially beyond major research institutions. Indeed, they observe days of training time, even whilst leveraging over 100,000 distributed CPU cores and hundreds of GPUs[4]. Additionally, we note that little research has been directed into improving the capabilities and performance of their training environments.

However, recent work has leveraged massive parallelism to obtain significant improvements on algorithm run times for computationally expensive sub-domains of reinforcement learning [8, 9, 10]. We argue that applying such methods to multi-agent reinforcement learning will provide greater accessibility and accelerate research in the field.

1.2 Contributions

We present **MAax**: a novel framework facilitating the creation of scalable and parallelisable simulation environments for researching Multi-Agent Reinforcement Learning. We categorise the main contributions into two divisions:

- **Modular Procedural World Generation**
  We introduce an API for procedurally generating environments within the accelerated Brax[9] simulation engine. By generating environmental components in a modular fashion, the resulting systems are not only entirely re-creatable, but – through customising the distribution, complexity and variety of objects in the scene – also provide revolutionary diversity.

- **Differentiable Suite of Task Mechanics**
  We demonstrate how the framework can be leveraged for multi-agent competitive games by implementing the game mechanics for a sample task, Hide-and-Seek. We achieve this through a suite of stateless environment wrappers, which we then extend with a set of utility classes, allowing researchers to easily reshape global state into unique and specific agent observations for policy training.

Under evaluation, we find our environments to be suitable candidates for leveraging massive parallelism, and as such, lay a novel and accessible landscape for researching multi-agent interactions and how they shape intelligent behaviour.

Chapter 2

Preliminaries

In this chapter, we begin by introduce the requisite material surrounding the core technical concepts of the project. First, an overview of Machine Learning, and in particular the area of Reinforcement Learning; a formal definition of the problem is presented before moving onto recent works concerning Multi-Agent and Deep Reinforcement Learning. We then briefly survey the physical simulation landscape, detailing the various solutions available and their respective limitations.

2.1 Machine Learning

Machine Learning (ML) is a sub-field of Artificial Intelligence (AI) that concerns enabling systems to learn from data and experiences, and using this information to make their own independent decisions. We divide Machine Learning into three distinct groups:

- **Supervised Learning** : The system learns a predictive model using labelled data points. The model is learnt through an optimisation routine: a prediction is made on the input sample and then validated against the true value. From this, the system adjusts its internal parameters to more closely fit the correct outcome. We classify supervised learning into two problem classes:
  - **Classification** : The model predicts a discrete class label from the input data.
  - **Regression** : The model predicts a continuous real output variable from the input data.
- **Unsupervised Learning** : The system independently learns a predictive model using unlabelled data points, finding structure and patterns in the output space. Common algorithms include clustering and dimensionality reduction[11, 12].
- **Reinforcement Learning** : The system learns optimal behaviour in an environment. Rather than being explicitly taught, learning occurs through observing the consequence of actions on the reward received from the environment.

Whilst Unsupervised Learning algorithms often involve optimisation processes that take considerable time to converge, Reinforcement Learning relies on a trial-and-error process of interaction with the environment to determine an optimal policy. As a consequence of this, and the complexity of the learning algorithms discussed later, Reinforcement Learning is comparatively the slowest.

2.1.1 Artificial Neural Networks

For many ML tasks we wish to approximate the function f that represents the relationship between our input and prediction spaces. Inspired by the structure and behaviour of the human brain, modern techniques often model these functions using compounded networks of artificial neurons – Artifical Neural Networks (ANNs). Each neuron in the graph performs a simple computation on its inputs, as visualised in Equation(2.1) and Figure 2.1

$$y = g(w_1x_1 + w_2x_2 + w_3x_3 + b) \tag{2.1}$$

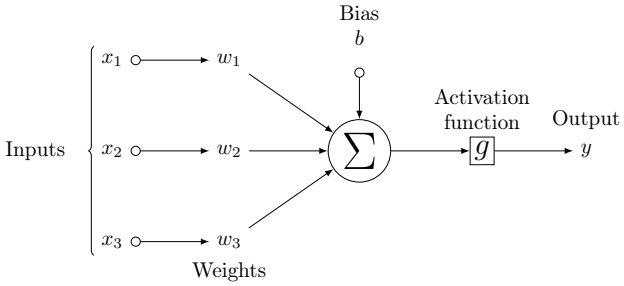


Figure 2.1: Structure of a artificial neuron

Single-layer artificial neurons are however only capable of learning linearly separable patterns in data; many of the more complex problems we wish to tackle in ML require non-linearity to approximate. By connecting several layers of neurons into a network, we increase the size of our function class and obtain non-linearity.

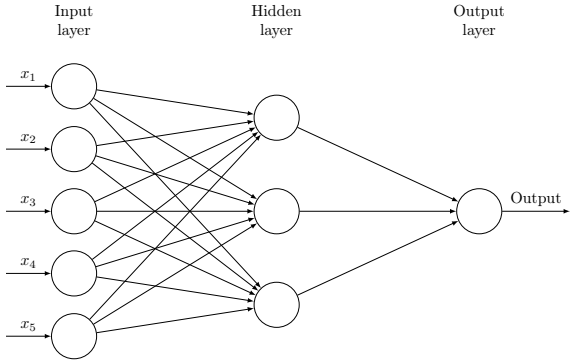


Figure 2.2: Example architecture of a Artifical Neural Network containing a single hidden layer

2.2 Reinforcement Learning

As previously mentioned, Reinforcement Learning (RL) is a subset of Machine Learning concerning the problem of goal-directed learning in an often uncertain environment[13]. The learner (*agent*) interacts directly with the *environment*, receiving at each timestep a *reward signal*. By observing the influence of its actions on this reward signal, the agent seeks to optimise its behaviour in the environment and maximise rewards over time.

2.2.1 Markov Decision Processes

We often frame RL problems as partially observable Markov Decision Processes (POMDPs). These are a formalisation of an optimal control problem concerning sequential decision making and notably the consequences on both the immediate and subsequent rewards. Within the context of reinforcement learning, we construct MDPs over the quadruple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$.

- **State space** $(\mathcal{S})$: represents the space of all states to which the agent can transition to.
- **Action space** $(\mathcal{A})$: represents the space of all actions the agent can take in the environment. We often reference, more specifically, $\mathcal{A}(s) \subseteq \mathcal{A}$, the set of actions available given the agent is currently in state $s \in \mathcal{S}$.
- **Transition probability function** $(\mathcal{P})$: represents the environment dynamics. Formally, the probability of transitioning to state $s' \in \mathcal{S}$ when taking action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$.
- **Reward function** $(\mathcal{R})$: represents the numerical reward the agent receives upon transitioning into state $s' \in \mathcal{S}$ from state $s \in \mathcal{S}$ via action $a \in \mathcal{A}$.

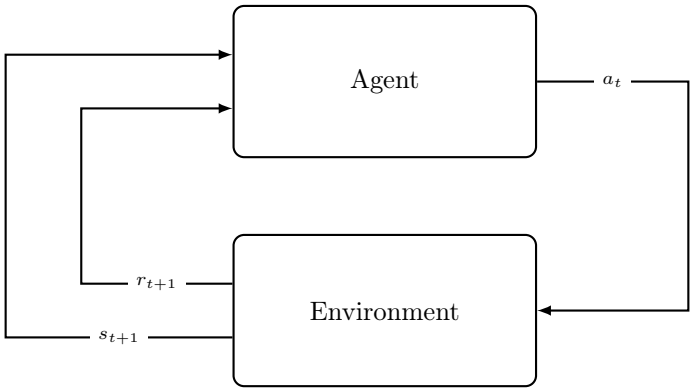


Figure 2.3: The agent-environment interaction loop in Reinforcement Learning [13]

The agent and environment interact continually, in an exchange of action and response. More specifically, this interaction is considered over a sequence of discrete timesteps, $t = 0, 1, 2, ...$[13]:

- At each discrete timestep t, the agent holds a representation of the environment's state $s_t \in \mathcal{S}$, then chooses an action $a_t \in \mathcal{A}(s_t)$ based on this observation.
- On the subsequent timestep $t + 1$, the agent receives a numerical reward signal from the environment as a consequence of its previous action, $\mathcal{R}_{t+1} \in \mathcal{R}$, and falls into a destination state $s_{t+1} \in \mathcal{S}$.

Agents use this iterative process to learn an optimal policy (also illustrated in Figure 2.3). For each state, the policy π represents a mapping $\pi(a|s)$ from the perceived state $s \in \mathcal{S}$, to a probability distribution over the actions available in that state $a \in \mathcal{A}(s)$. As such, it essentially defines the agent's way of behaving at any given time.

The optimal policy π^* is then the one that maximises the expected total reward obtained from the environment. Consider a finite sequence of states and actions, $s_0, a_0, ..., s_t, a_t, s_{t+1}$, we define the cumulative reward over this *trace* as the *return* $R_t = \sum_{t=0}^{T} \gamma^t r(s_t, a_t, a_{t+1})$. Here $\gamma \in [0, 1)$ denotes the discount factor, representing how much the agent values immediate rewards relative to future ones.

Optimality

From this, we can introduce the notion of state-value: how 'good' it is for the agent to be in a given state s under a policy π. We formally define the state-value as follows:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[R_t|S_t = s] = \mathbb{E}_{\pi}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|S_t = s] \tag{2.2}$$

Similarly, we define the action-value function: how 'good' it is for the agent to be in a given state s and take a given action a under a policy π:

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[R_t|S_t = s, A_t = a] = \mathbb{E}_{\pi}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1}|S_t = s, A_t = a] \tag{2.3}$$

We can now also define a partial ordering on polices; a policy π is strictly better than another policy π' if it represents a greater expected return for every state. Formally, $\pi > \pi'$ if and only if $V^{\pi}(s) > V^{\pi'}(s)\, \forall s \in S$. Thus, a policy is optimal if it is better than or equal to all other policies and one such must exist. Additionally, this policy π^* will maximise the value function $V^*(s)$.

$$\pi^* = \underset{\pi}{argmax}\, [V^{\pi}(s)]\, , \forall s \in S \tag{2.4}$$

$$\pi^* = \underset{a \in A}{argmax}\, [Q^*(s, a)]\, , \forall s \in S \tag{2.5}$$

In the deterministic case, learning an optimal policy is therefore synonymous with learning the optimal action-value function. This is expressed in the *Bellman Optimality Equation* (Equations 2.6, 2.7): the value of a state under an optimal policy must equal the expected return of the best action from that state.

$$V^*(s) = \max_{a \in A(s)} \left[Q^*(s,a) \right], \forall s \in S \tag{2.6}$$

$$Q^*(s,a) = \mathbb{E}_\pi [R_{t+1} + \gamma \cdot \max_{a'} Q^*(S_{t+1}, a') | S_t = s, A_t = a] \tag{2.7}$$

Solution Classes

Given a perfect model of the environment as a finite MDP, we can leverage the Bellman equations to iteratively compute the optimal policy. Such *dynamic programming* algorithms include Policy and Value Iteration, aiming to learn the optimal policy and value function respectively. However, often we do not have complete knowledge of an underlying MDP. We consider both these scenarios and their associated solutions as separate classes:

- **Model-based learning** : We have knowledge of the environment dynamics $\mathcal{P}$ or are able to closely estimate them. We construct a model of the environment from this information and consult it to guide decision making. These methods update value estimates based on neighbouring estimates to closer approximate the optimal value function.
- **Model-free learning** : There is no explicit model of the environment or dynamics; we instead learn directly from sample interactions. We refer to these experience-based algorithms, such as Q-Learning or Actor-Critic methods, as *Monte-Carlo Algorithms*.

Model-based learning is more data efficient: we can update the entire value-function or policy with a single sample. Though computationally, model-free methods prove more efficient: in avoiding full-backups, the cost remains constant and independent of $|\mathcal{S}|$. Furthermore, model-free methods incur a lower memory cost: representing the state-value function requires $|\mathcal{S}| \times |\mathcal{A}|$ space compared to $|\mathcal{S}| \times |\mathcal{A}| \times |\mathcal{S}|$ required to represent the underlying model. Thus, model-free methods escape the *Curse of Dimensionality* and are much more scalable as a result. However, they require an immense amount of data to learn from and are, as such, limited by that quantity. We refer to the amount necessary to train a successful model as the *sampling complexity* of an algorithm and in reinforcement learning it is especially high.

2.2.2 Deep Reinforcement Learning

In many reinforcement learning tasks, the state space is continuous or impractical to represent as a discrete set. In such cases, it is infeasible to utilise the previously discussed tabular reinforcement learning methods. Deep Reinforcement Learning (DRL) applies artificial neural networks as function approximators of the value or policy function. Representing functions in this way alleviates the exploding memory requirements associated with high-dimensional tabular state spaces. This has driven many of the recent successes in reinforcement learning for both discrete[3] and continuous[14] control tasks.

Discrete Action Control - Deep Q Learning

A revolutionary work in Deep Reinforcement Learning space was the Deep Q-Learning algorithm proposed by Mnih et al.[15]. Q-learning represents a model-free approach to approximating the action-value function. Through directly sampling the environment, the agent updates an parameterised deep convolutional neural network Q_θ, taking as input the observation vector, and outputting Q-value estimates for each discrete action (illustrated in Figure 2.4).

The training objective of the Q-network is to minimise the mean-squared error in the Bellman Equation (2.7), substituting optimal target values $r + \gamma \cdot \max_{a'} Q^*(s_{t+1}, a')$ for approximate target values $r + \gamma \cdot \max_{a'} Q_{\theta_i^-}(s_{t+1}, a')$, conditioned on previous iteration parameters θ_i^-. This results in the loss function $L_i(\theta_i)$:

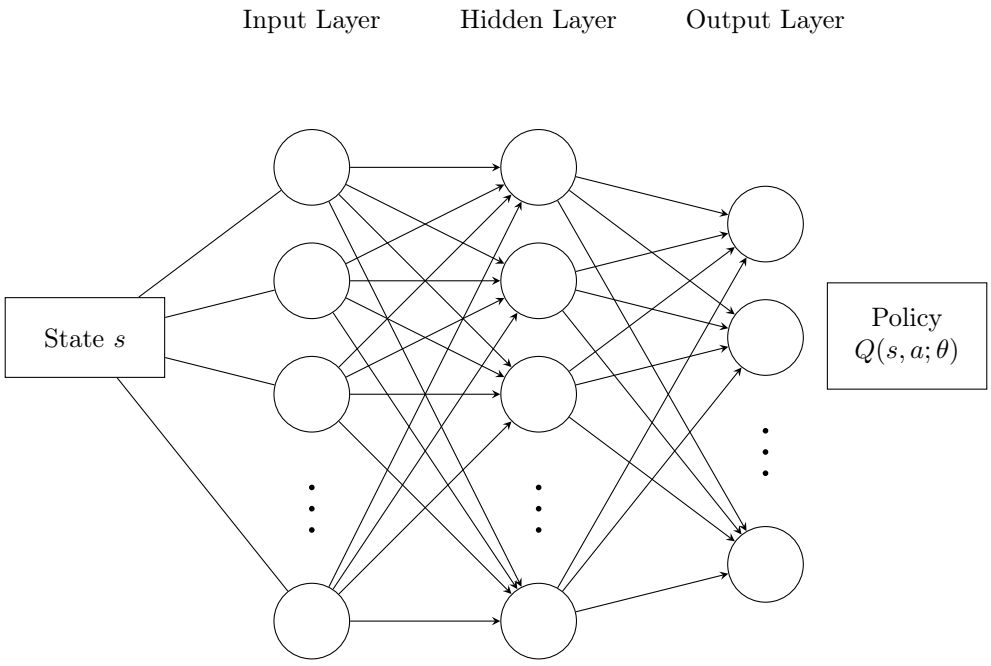


Figure 2.4: Abstract architecture of a Deep Q Network

$$
\begin{aligned}
y_j &= \begin{cases} r_j & s_{j+1} \text{ is a terminal state} \\ r_j + \gamma \hat{Q}_\theta \left(s_{j+1}, a'\right) & \text{otherwise} \end{cases} \\
L_i(\theta_i) &= \frac{1}{2}\mathbb{E}\left[\left(y_j - \hat{Q}_\theta \left(s_j, a'\right)\right)^2\right]
\end{aligned}
\tag{2.8}
$$

We can then optimise the parameters of a Deep Neural Network approximator via gradient descent. Differentiating with respect to the network weights we obtain the gradient:

$$
+_{\theta_i} L_i(\theta_i) = \mathbb{E}\left[\left(y_j - \hat{Q}_\theta \left(s_j, a'\right)\right) \nabla_\theta \hat{Q}_\theta \left(s_j, a'\right)\right] \tag{2.9}
$$

Using non-linear function approximators such as CNNs for the Q-function is known to be more unstable in convergence than the linear case[16]. Another key contribution of the authors was in addressing this instability; the *experience replay* mechanism alters the standard Q-learning algorithm to utilise a saved transition buffer. During a rollout, the agent stores experienced transition quadruples $(s_t, a_t, s_{t+1}, r_{t+1})$ in the fixed size buffer for each timestep t. Then during learning, the agent applies Q-learning updates on sampled uniform random batches from the buffer dataset D (Algorithm 1). In this way, the algorithm is able to disconnect learning from the correlations present in sequential observations and thus reduce variance and avoid local minimums. A recent improvement is the Double DQN[17] algorithm; where the Double Q-Learning algorithm[18] is leveraged to address the overestimation problem. We decouple the role of action selection to a target network $\hat{Q}_{\hat{\theta}}$ and evaluate the action-value under the primary network Q_θ. By periodically copying $\hat{\theta} \leftarrow \theta$, we help stabilise learning, achieving more accurate value estimations and a better policy as a result.

Continuous Action Control - Policy Gradient Methods

Although a set of discrete actions suffices for many reinforcement learning tasks – particularly in the domain of games[3] – many real-world applications require continuous and high-dimensional action spaces. DQN cannot be naively applied to these domains: finding the action to optimise the action-value function at each step now becomes a slow iterative optimisation process. Thus, rather than approximating a value function, we instead directly learn a stochastic policy π using an independent function approximator with its own parameters θ[19]. Quite often a neural network is chosen for this, taking as input a representation of the state and outputting action selection probabilities. For the physical simulations of joints and actuators we discuss later, this is particularly

Algorithm 1: Deep-Q-Learning

Initialise replay memory D to capacity N;
Initialise action-value function Q_θ with random parameters θ;
for *episode = 1,M* **do**
  Initialize S;
  for t = *1,T* **do**
    Select action a_t using ϵ-greedy policy on Q_θ;
    Execute action a_t and observe reward r_{t+1} and state s_{t+1};
    Choose A from S using policy derived from Q;
    Store transition (s_t, a_t, r_{t+1}, s_{t+1}) in D;
    Sample mini-batch B of size N from D;
    Set $y_j = \begin{cases} r_j & s_{j+1} \text{ is a terminal state} \\ r_j + \gamma \hat{Q}_\theta\left(s_{j+1}, a'\right) & \text{otherwise} \end{cases}$;
    Perform gradient descent step on $(y_j - \hat{Q}_\theta(s_j, a')$ as in Equation 2.9
  end for
end for

relevant.

For a state-action trace τ, the goal of these *policy gradient methods* is to find the optimal policy weights θ^*, namely those that maximise the expected return: $\theta^* = argmax_\theta \, \mathbb{E}[\sum_t^T r(s_t, a_t)]_{(s_t,a_t)\sim p_\theta(\tau)}$. We define the performance measure $J(\theta)$ of the model as this expected return, or equivalently, the value function of the start state under the parameterised policy. Practically, this can be approximated by the mean empirical return over N traces:

$$J(\theta) = \mathbb{E}[\sum_t^T r(s_t, a_t)]_{(s_t,a_t)\sim p_\theta(\tau)} \approx \frac{1}{N} \sum_{i=1}^{N} \sum_t r(s_{i,t}, a_{i,t}) \tag{2.10}$$

We derive the gradient and optimise via gradient ascent:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_t \nabla_\theta \log \pi_\theta \left(a_{i,t} \mid s_{i,t}\right) \right) \left(\sum_t r\left(s_{i,t}, a_{i,t}\right) \right) \tag{2.11}$$

Actor-critic algorithms learn both a policy and state-value function, where the value function approximator is used for bootstrapping returns in place of exclusively sampling from the environment, resulting in decreased variance[13]. The DDPG[2] algorithm represents an actor-critic, model-free extension of the DQN[3] for continuous action spaces, utilising experience replay and target networks alongside the parameterised actor function presented in the DPG algorithm[20]. Further pursuit of improvements has led to the emergence of a suite of algorithms that currently stand at the forefront for learning continuous action control – A3C[21], TD3[22], SAC[23] and PPO[24]. Each offers different properties in convergence and efficiency, and such physical simulators often provide training frameworks with custom implementations for each.

2.2.3 Multi-Agent Reinforcement Learning

In the methods discussed so far, we consider only a single agent solving the sequential decision-making problem. However, as it provides greater likeness to real-world scenarios, of particular interest to many researchers is the Multi-Agent case, where each agent possesses their own objectives and action spaces. As such, the environment state and reward function is now determined by the agents' joint actions $\boldsymbol{a}_t$.

Modelling

This slight change has a significant impact on the underlying models, violating several of our previous assumptions. In a traditional MDP, we assume the environment to be stationary, meaning

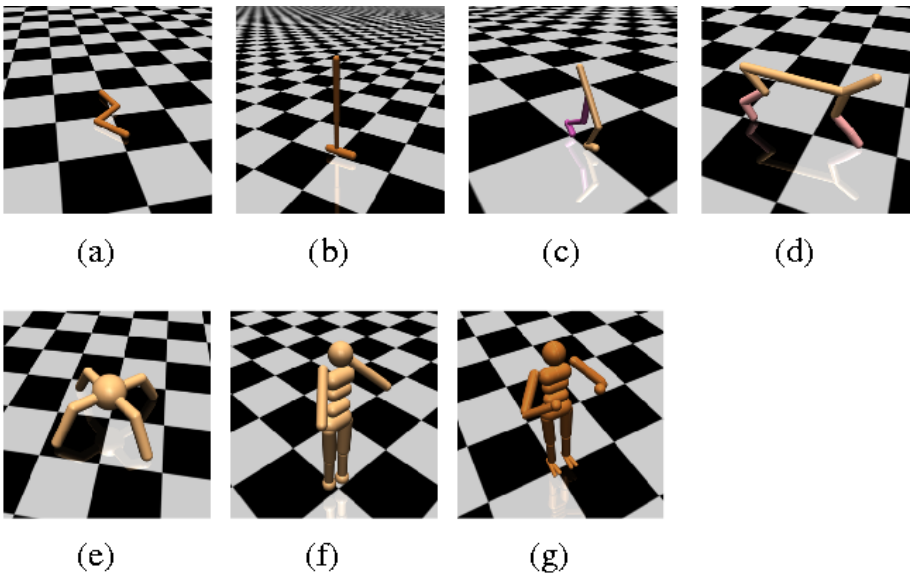


Figure 2.5: The RLLab control environments, later adapted to form OpenAi Gym's MuJoCo tasks (Duan et al., 2016) [26]

.

that the transition dynamics $\mathcal{P}$ and rewards $\mathcal{R}$ do not change over time. For a given agent, we can consider all the others as part of the environment; as they influence our reward and transitions, we have a non-stationary condition.

Similarly, we also violate the Markov property: the state is no longer *sufficient*. We may now need to consider the intentions and behaviour of other agents, requiring partial observability of their state. Therefore, to reason about these scenarios, we usually model the process as a Stochastic/-Markov game, which can be considered to be a multi-agent extension to an MDP[25]. Essentially, the Stochastic game framework allows simultaneous actions from agents. Each agent then tries to learn an optimal policy under the influence of all other policies, expressed in Equation 2.12.

$$V^{\pi^i,\pi^{-i}}(s) = \mathbb{E}_{s_{t+1}\sim P(\cdot|s_t,a_t),a^{-i}\sim\pi^{-i}(\cdot|s_t)}\left[\sum_{t=0}^{\infty}\gamma^t R_t^i\left(s_t,\boldsymbol{a}_t,s_{t+1}\right) \mid a_t^i\sim\pi^i\left(\cdot\mid s_t\right),s_0\right]. \tag{2.12}$$

Cooperation and Competition

It follows probabilistically that a set of independent agents will outperform a given single agent. However, we can observe new behaviours by allowing agents to cooperate and pursue a common goal. Tan[6] proposes three methods of cooperation: sharing sensation, sharing episodes, and sharing policies. It was observed that communication in this way can improve both the speed of learning, and quality of solution. In contrast, we can explore agents in a competitive setting, where they seek only to maximise their own reward. This is a characteristic inherent in human interaction, for example, in the field of economics.

2.3 Physical Simulation

Simulators play a crucial role in developing robots for deployment in the real world; training via the RL paradigm in physical space can easily lead to damage to machinery or the environment itself. There exists a broad and varied selection of engines to power these simulated environments; we detail the most prominent in the current moment of robotics research.

MuJoCo

MuJoco[27] has long represented the state-of-the-art for accuracy when simulating constrained multi-joint systems. Rather than representing state as Cartesian coordinates with numerical joint constraints like many game-oriented engines[28], MuJoCo maintains a representation using joint coordinates and simulates contacts more accurately.

For broadening the scope of Gym[29] research to robotics, OpenAI uses MuJoCo to implement a suite of 10 control tasks — an augmentation of the RLLab[26] control environments, as seen in Figure 4.2.

Brax

Brax [9] is a fully differentiable engine for simulating physical systems made up of rigid-bodies, joints and actuators, with a strong focus on parallelism and hardware acceleration. To achieve this Brax is implemented in JAX: 'a JIT compiler for generating high-performance accelerator code'[30]. To provide a baseline for comparison, Brax ships with a reconstruction of the community-adopted Gym MuJoCo tasks.

Freeman et al. motivate Brax with three key insights into the existing domain:

- **To address the latency problem:** Whilst RL algorithms generally run on the GPU or TPU (or another machine entirely), most simulation engines still operate on the CPU. Marshalling and then transferring this data across machines proves a major component in the time an RL experiment takes to run.
- **To provide differentiability:** In order to make predictions about consequences, model-based RL algorithms require a gradient for the sampled environment state. Without providing this derivative, non-differentiable simulation engines restrict the researcher to model-free RL and less efficient optimisation approaches.
- **To enable introspection:** Freeman et al.[9] describe most simulation engines as *black boxes*: it's very unclear exactly how they work. Many are closed source and built on a different technical stack to the RL algorithms, thus, it's difficult to understand the relationship between the state and action spaces. Publishing a packaged open source library provides far greater accessibility to the research area and simplifies the debugging process.

These are not novel contributions individually – many differentiable simulation engines exist[31, 32] – and Freeman et al. acknowledge this. However, they argue that, unlike other simulators, Brax represents a solution to all three problems at once.

IsaacGym

An alternative accelerated simulator that has been successfully utilised in several RL domains[33, 34] is NVIDIA's IsaacGym[10]. Like Brax, the motivation involves addressing the simulation speed bottleneck faced by engines like MuJoCo[27] and PyBullet[28], and in particular to address the expensive physics state and action transfers between the CPU and GPU. Through a GPU-accelerated simulation back-end - leveraging NVIDIA's PhyX engine - and a PyTorch[35] tensor-based API to access state results natively on the GPU, IsaacGym provides an end-to-end GPU training pipeline.

Chapter 3

Related Work

Whilst the Multi-Agent domain is still eminent in contemporary RL research, truly esteemed works have not appeared at the same frequency as several years ago[4, 36]. In this chapter, we analyse existing works in the field, particularly those pertaining to the emergence of complex strategies in competitive tasks. In this process, we identify a limitation that may have contributed to the diminution of research in the area. Then, by considering the successes of massively parallelised methods in RL, we suggest a solution to the MARL problem.

3.1 Learning Complex Strategy

3.1.1 Self-Play and Curricula

In many cases, the complexity of a trained agent is directly contingent on the complexity of the environment. This is evidenced in works applying rich and incremental complexity[37, 5] for both quality and generalisation purposes. However, competitive multi-agent environments can produce behaviours that eclipse the complexity of the environment itself. This class of environments can be achieved via *self-play*, whereby an agent competes against former copies of itself. Such environments have two desirable properties for learning complex behaviour[7]:

- **Enhanced Environments:** There are many environments with very simple rules that need complex strategies to win, for example, Go. We require advanced strategies because the adversary produces complexity; as mentioned in Section 2.2.3, we can consider all other agents as part of the environment. Therefore, we have, by nature, an implementation of incremental complexity scaling.

- **Induced Curriculum:** A competitive multi-agent environment using self-play induces a near-perfect curriculum. By populating the environment with agents of equivalent competency, an agent undergoes suitable challenge and can learn regardless of current strength.

With regard to modern DRL methods, self-play was first introduced to train competitive video-game agents AlphaGo[38] and OpenAI Five[39]. In both cases, self-play induced behaviour more complex than the games themselves.

In a later work applying self-play to continuous control tasks, Bansal et al.[7] introduce several competitive tasks using the MuJoCo simulator[27], and trained agents via a distributed PPO implementation. A key observation from the experiment was that the agents learned a range of dexterous skills, often due to simply using a different random seed. Furthermore, these skills were not only valuable for competition, but also in independent tasks. For example, skills learned in the Sumo task enabled an agent to remain upright under strong wind perturbation, despite having never been exposed to such forces before.

The other crucial concept throughout the work is the notion that complex reward functions are not necessary to learn intelligent behaviour. It has long been standard practice in the community to carefully engineer reward functions, especially for locomotion tasks where the "correct" reward is

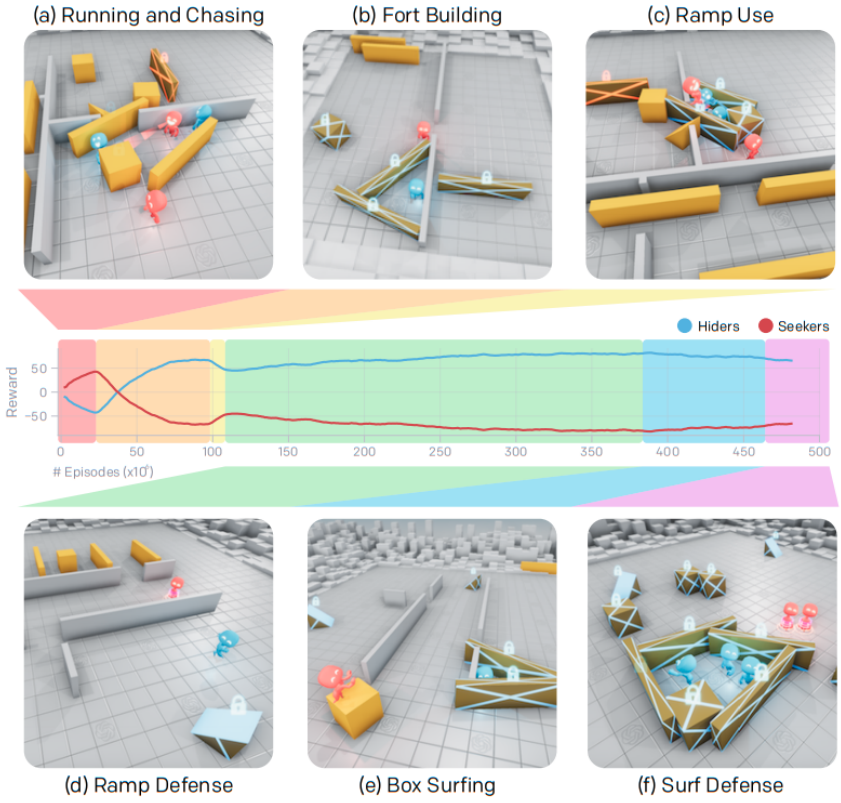


Figure 3.1: Emergent skill progression in multi-agent hide-and-seek[4]

difficult to define. In demonstrating that complex behaviour can emerge from simple reward functions, Bansal et al.[7] echo the sentiment of Heess et al.[5]: environmental diversity and richness can be sufficient.

Human Games

Continuing in the plane of physically grounded environments, Jaderberg et al.[36] trained a program to successfully compete against humans at Capture-The-Flag in the video-game Quake III Arena. As they had hypothesised, highly competent agents learned a rich representation of the game without explicit motivation. By assessing agents on binary features of the current state, such as flag possession and base proximity, they found high classification correctness amongst the elite "FTW" trained agents.

The work of Baker et al.[4] challenges agents in another human game – hide-and-seek. Most significantly, they found clear evidence of an evolving auto-curriculum over the course of training; six phases of emergent strategy appeared, as the development of each new strategy forced the opposing team to adapt (illustrated in Figure 3.1). Furthermore, they provide evidence that – for discovering human-relevant skills – multi-agent competition scales better with environmental complexity compared with self-supervised exploration methods such as intrinsic motivation.

However, we highlight the expensive compute of their training model. The default model, with 1.6 million parameters and a batch-size of 64,000, still required 132.3 million episodes to reach Stage 4 of progression (ramp-defense). Using their extremely powerful Rapid framework (displayed in Figure 3.2), this translates to 34 hours of training. Given this duration is only achieved whilst leveraging ∼100,000 CPU cores and hundreds of GPUs, it is evident that such research is currently inaccessible for groups outside of major global research institutions.

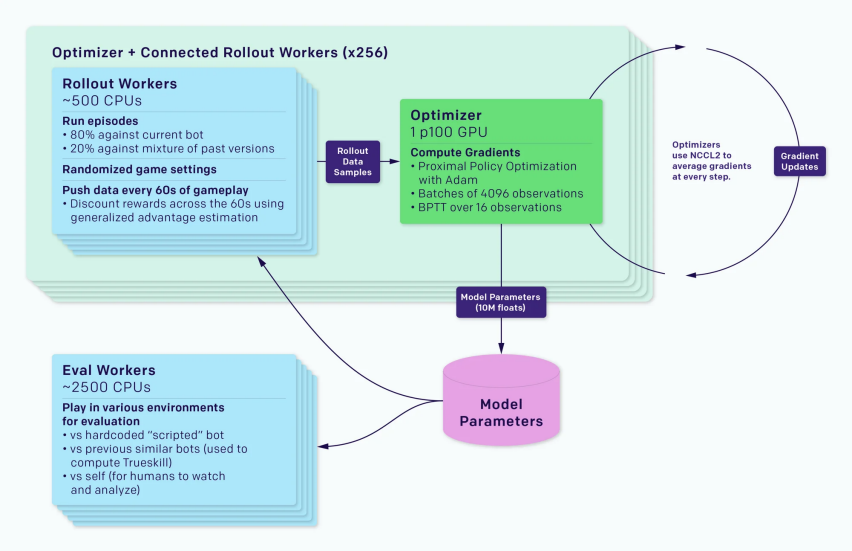


Figure 3.2: The OpenAI Rapid Framework as presented in 2018[40]

3.2 Procedural Content Generation

Procedural Content Generation (PCG) refers to the pseudo-random and algorithmic generation of data. While traditionally focused on content creation for video-games[41], PCG has many use cases in other domains leveraging virtual environments. Risi and Togelius[42] provide a broad summary of these existing applications, as well as a discussion on future opportunities and challenges. We consider here those pertaining to the space of machine learning.

3.2.1 Applications to Machine Learning

The resulting diversity of PCG that produces novel game environments can also serve to tackle the most frequently discussed issue in ML – the generality problem.

Data-Augmentation

In actuality, data-augmentation represents a PCG adjacent procedure, but remains the simplest and most common application in ML. The aim is to increase the size and diversity of the training set by making modifications to the original samples. In the field of supervised learning – more specifically image classification – these often include geometric transformations, colour space augmentations and additionally mixing between images. It is substantially evidenced that such methods result in significantly less overfitting to training data[43, 44].

3.2.2 Domain Randomisation

As previously discussed, the sampling required to enable random exploration and large state spaces imposes huge demands on hardware. This is most relevant in the field of Robot Learning, where the physical nature of robots often makes data collection infeasible. Consequently, researchers in the field must often rely on the Sim2Real transfer: learning a policy in physical simulation, then rolling it out in the real world.

Physical simulators have not only discrepancies with one another, but more importantly, with the real world. Whilst they do offer a scalable and cheap solution for training robotics models, tuning the simulation parameters to better reflect the real world is time-consuming and offers no guarantees in translation. Moreover, there are many physical effects that escape modelling by simulators, such as gear backlash and fluid dynamics[45]. These inconsistencies are collectively referred to as the *reality gap* and form a considerable hindrance on the Sim2Real transfer. Domain randomisation aims to address the reality gap through providing a randomised range of environments during training, whereby the diversity in simulation is sufficient to induce generalisation to

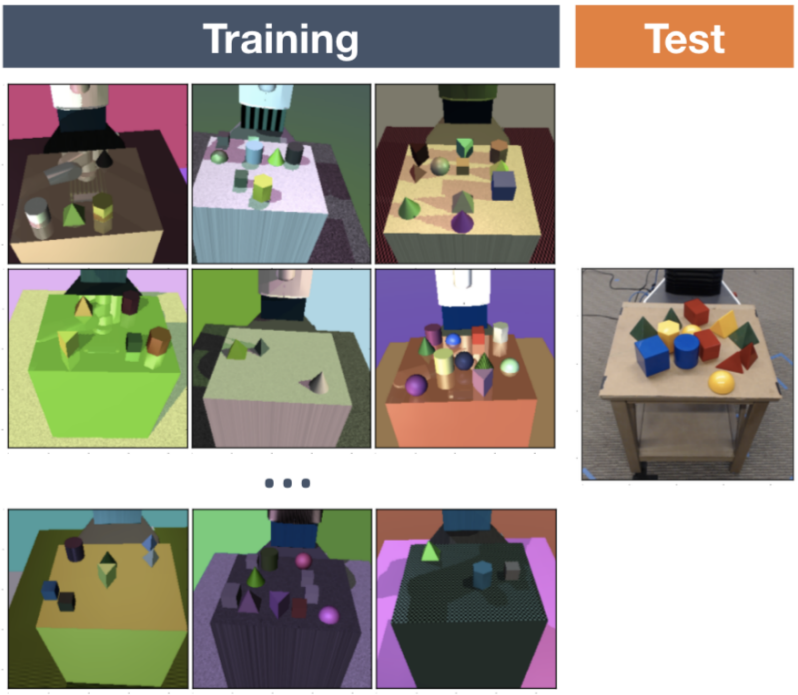


Figure 3.3: Comparison of the domain randomised renders with the real world test images (Tobin et al., 2019) [45]

the real world. In essence, this is a method for tackling the *state distribution problem* by increasing the state-space coverage of the training data.

Initial works in this field focused on the transfer between image spaces and have shown extremely promising results. Tobin et al.[45] produced a real world object detector trained solely on low-fidelity rendered images (Figure 3.3), randomising between them, camera and object positions, lighting, and textures.

Following Weng [46], the domain randomisation that we see from both Tobin et al.[45] and Sadeghi et al.[47] exhibits interval bounded randomisation parameters, $\zeta_i \in [\zeta_i^{low}, \zeta_i^{high}], i = 1, ..., N$, uniformly sampled within that range.

Guided Domain Randomisation

Whilst simply sampling the parameters can provide sufficient coverage for several problems, designing appropriate distributions for simulation parameters requires expert knowledge for tuning, and yet still provides no guarantees that the randomisation will lead to a sensible real-world policy. We can instead choose to direct the randomisation prior using task performance or known information about the real environment and achieve a much better solution, a concept closely intertwined with the curriculum-based learning seen in works discussed earlier[48, 37, 7]. In this way, we also reduce the computational burden by focusing only on training in realistic and feasible environments. One such example is the work of Chebotar et al.[49]. Through tuning the parameter distribution by interleaving real-world rollouts during policy training, they demonstrated improved Sim2real policy transfer compared with standard domain randomisation.

3.2.3 Creating Environments

Indeed, many of the environments utilised for competitive MARL are procedurally generated[7, 5, 4, 36], and thus inherently domain randomised. We observe in each that environmental diversity is equally necessary for complex behaviour; agents must learn rich[36] and generalised representations of the environment to develop truly intelligent strategies.

We also see PCG used to create bench-marking suites for RL algorithms; the ProcGen[50] environments are heavily randomised to compel agents to learn policies robust to multiple axes of variation. However, as discovered by Heess et al.[5], excessive randomisation may hinder learning in early policy training as there are too many things for an agent to explore. For instance, when randomising the position of the ball and agent in the kick-and-defend task, agents were unable to learn the skill of kicking. Conversely, with no randomisation, agents overfit to certain ball positions. We arrive at the conclusion that incremental randomisation can attribute to learning intelligent behaviours in the same way as incremental complexity scaling.

3.3 Accelerated RL via Massive Parallelism

As discussed in Section 2.2, sampling complexity imposes a significant bottleneck on Reinforcement Learning methods. For sub-fields like robotics, where we can execute rollouts in physical simulators, this issue is mitigated slightly.

Deep Reinforcement Learning

In the context of DRL, the training time issue is exacerbated by the necessity of hyper-parameter tuning, which requires sequentially rerunning the expensive training process. As a result, efforts to parallelise and accelerate the training of RL agents have been ongoing for several years. The first of these works was Gorila[51]: a distributed DQN[3] implementation. Using hundreds of CPU cores, it achieved significant, though sublinear, speedups. Following Gorila, Mnih et al.[21] proposed the A3C algorithm, which is by nature parallelised, and, using 16 CPUs, achieved similar results to Gorila.

A subsequent extension was the GA3C[52] algorithm, where the policy is moved to the GPU for accelerated inference and training, resulting in a 4x speedup on CPU-only A3C. In an investigation to further optimise existing RL algorithms, Stooke et al.[53] produced a framework for multi-GPU DRL, adapting state-of-the-art policy-gradient and Q-learning algorithms to utilise parallel simulator instances. With these optimisations, they were able to learn effective strategies for many Atari games in under 10 minutes.

Physics Simulation

Having been initially applied to DRL methods for policy training, the logical advancement was the application of GPU-acceleration to physical simulation. Liang et al.[54] demonstrated the promise of this approach to several continuous control locomotion tasks, notably training the high-DOF Humanoid task to run in less than 20 minutes.

Using an NVIDIA Tesla V100 GPU on their internal compute cluster, and a single CPU core of an Intel Xeon E5-2698 running at 2.2GHz, they observed similar learning time to ES[55] and ARS methods, with 10x and 1000x less CPU cores respectively. However, we note that their results are not directly comparable to those obtained in works using other engines, due to the difference in simulation physics.

In an extension of their method to the Multi-GPU case, they observe more apparent scaling benefits with the "Humanoid Flagrun Harder" (HFH) and "HFH on Complex Terain" tasks. The implication here is that more complex tasks achieve greater performance gains from distributed massive parallelism, and are as such ideal candidates for hardware accelerated RL.

The authors' final contribution is the development of a GPU-accelerated RL simulator to facilitate further research in the community. This would later be released as IsaacGym[10].

Fully On-Device

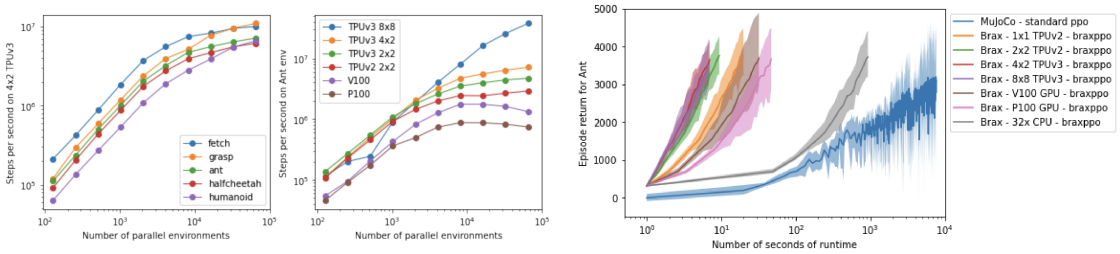
Whilst leveraging GPU acceleration for either simulation or policy update stages has shown considerable performance gain compared to CPU based methods, the data collection stage of on-policy

algorithms is not easily parallelisable. Consequently, simulation and reward/observation calculation must be transferred over PCIe, which has been shown to be up to 50 times slower than the GPU processing time itself[56]. Motivated by this problem, Rudin et al.[48] examine the potential of massive parallelism for on-policy algorithms. By leveraging the fully on-device capability of their recently released simulator IsaacGym[10], the authors evade the latency of data transfer to the CPU, and escape the core-limits and memory constraints of CPU parallelisation. They evaluate this approach on a terrain walking task with a quadrupedal robot, learning effective policies for flat and uneven terrain in 4 and 20 minutes respectively.

As a secondary contribution, the authors present a novel game-inspired curriculum suitable for parallelism. All robots are assigned a terrain type from flat, sloped, rough, discrete objects and stairs. Each is also allocated a level representing the difficulty of their terrain, for slopes this is the incline, and for stairs/obstacles, it is the step height. Escaping the borders of its terrain marks a 'level-up' for the robot, relocating it for the next episode. In this way, instead of learning the distribution of the policy's performance via a generator network, it can instead be directly represented by curriculum progress.

As mentioned in Section 2.3, the alternative fully on-device simulator is Google's Brax[9]. In the initial publication, the authors present the performance benefits of massive parallelism for both the training and simulation stages. We see in Figure 3.4, the steps per second improvement during simulation, and a comparison of Brax's parallelised PPO vs a standard implementation in MuJoCo.

In a recent work utilising Brax, Lim et al.[8] present an accelerated implementation of MAP-Elites[57] leveraging massive parallelism on hardware – QDax. They demonstrate the scalability of QD algorithms to large batch sizes and observe run-time improvements by two orders of magnitude compared with existing CPU-parallelised implementations.



(a) Scaling of environment steps/s with batch-size

(b) Training curves for PPO implementations

Figure 3.4: Performance when leveraging massive parallelism for both simulation (Figure 3.4a) and training (Figure 3.4b) in Brax.[9]

3.4 Summary

To summarise, we have seen repeated evidence that competitive MARL tasks can induce complex strategies[36, 4]. We note this to be contingent on both environmental complexity and diversity, and often, the adversary itself can impose a challenging and evolving auto-curriculum to facilitate this[7]. Regardless, the exact domain randomisation for such tasks needs to be considered to guide exploration in positive initial directions whilst maintaining sufficient diversity. A suitable method is incrementally scaling environmental complexity and randomisation as task performance improves. In essence, evolving both the environment and adversary over time yields policies that are intelligent and transferable.

Although, as such methods involve complex environments with potentially multiple stages of skill emergence, they are more computationally expensive than standard RL problems and can only be solved with substantial clusters. Consequently, we find the area outside the reach of many researchers and to this we attribute the lack of progress in recent years. However, with the gradual journey of massive parallelism for DRL having now culminated in fully on-device training pipelines,

there exists the potential to accelerate MARL research using these simulators. In fact, with indications that hardware acceleration is ideal for both curriculum-based learning[48] and complex tasks with high variance[54], we can apply this approach with reasonable confidence.

3.4.1 Our Approach

Thus, we propose a framework for exploring competitive MARL tasks, using the end-to-end physics simulator Brax[9] and high-performance computing library JAX[30]. With support for diverse domain randomisation and scaling, we hope to accelerate research in MARL and facilitate new discoveries of complex behaviour.

Why Brax?

As mentioned in Section 2.3, Brax represents a synthesis of desirable properties in a physics engine, especially when concerned with optimising performance. Accelerated alternatives such as IsaacGym[10] are non-differentiable, and differentiable options are non-parallelisable [31, 32]. Additionally, with the recent addition of the generalised physics pipeline to match the accurate contact dynamics of engines like MuJoCo[27], we don't compromise on accuracy either. As identified by Erez et al.[58], this inconsistency plagues the game-oriented engines[10, 28] using positional/impulse or spring-based dynamics. Furthermore, with identical physics, we can draw direct comparisons between engines, as was not possible in previous works regarding massive parallelism[54].

We also justify this decision as it provides the most 'future-proofing' to our work. It has been identified that the MuJoCo Gym environments are in an unmaintainable state: they contain multiple bugs in simulation configuration and depend on the fully deprecated MuJoCo-Py[59]. As a result, there has been considerable effort in recent years to replace these environments with Brax[60]. Terry[59] references a discussion with the DeepMind MuJoCo and Google Brax teams, stating that the differentiability and hardware acceleration capabilities of Brax made it 'the preferable option', and confirming internal plans at Google to maintain Brax for 'at least five years'.

Chapter 4

Procedural Environment Generation

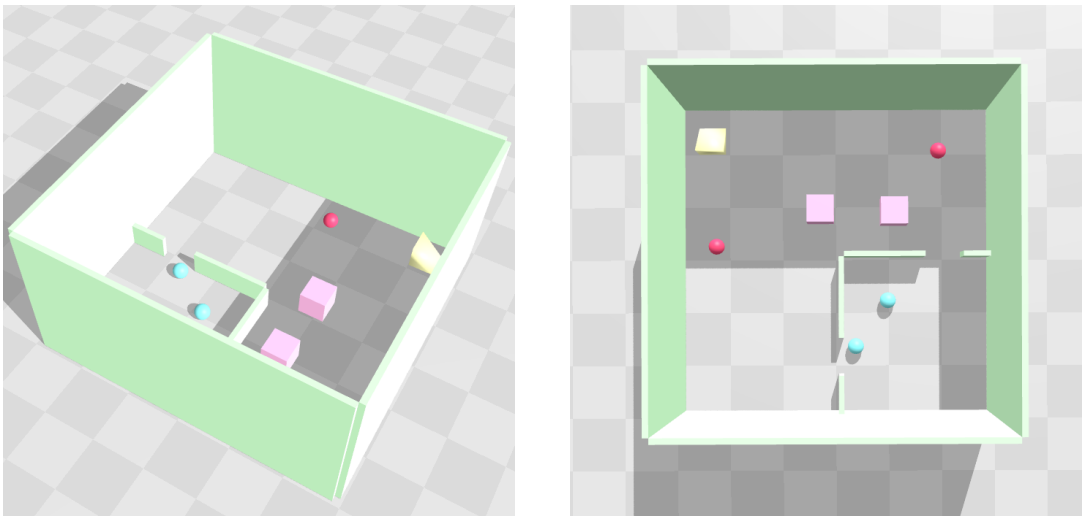


Figure 4.1: Example MAax environment. 2 hiders, 2 seekers, 2 boxes, 1 ramp, quadrant walls

We introduce here a new framework for accelerating Multi-Agent Reinforcement Learning – MAax. In this chapter, we detail the implementation of Procedural World Generation within the MAax framework, and demonstrate its capabilities in facilitating the creation of environments that are both structured and diverse.

4.1 Problem Statement

In order to facilitate learning in a broad range of tasks, we have identified three key properties that we require of our environments.

- **Modular:** The environment generation exposes all static and randomisation parameters for customisation. Procedural methods utilise these parameters independently such that specific functionality can be interchanged and isolated. As a result, a researcher can be confident in the exact properties of a given random sample.
- **Scalable:** Adjustment and incrementation of the size and complexity of the environments are facilitated both before and during policy training. In this way, domain randomisation can be guided to induce curriculum-based learning.
- **Parallelisable:** The framework supports leveraging parallelism on accelerated hardware during rollouts. Furthermore, all environment logic is functional under JIT compilation, and minimal in memory requirements in order to maximise benefits.

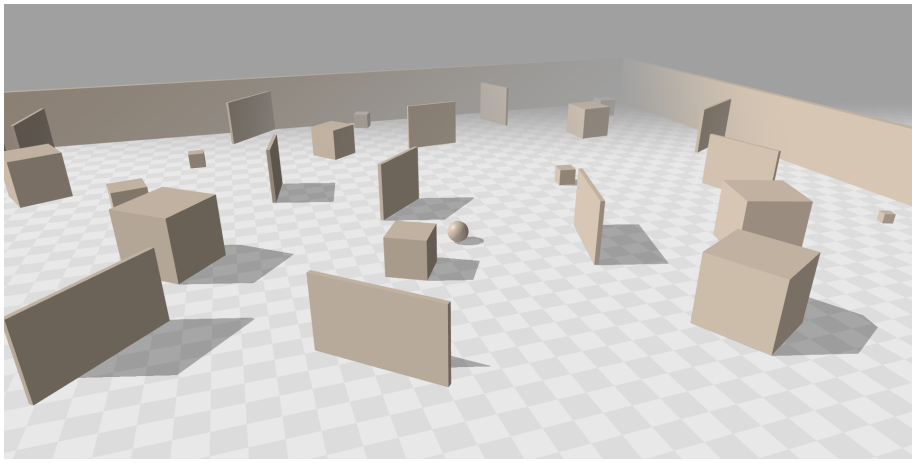


Figure 4.2: Randomised environment in Brax V1 via a minor modification to Algorithm 2

.

4.2 Initial Work

Our initial objective was to familiarise ourselves with the Brax framework. Of course, we wanted to progress towards our goal in this process; we began by implementing a scalable point distribution algorithm and basic parsing functions to convert the points into objects within a protocol buffer representing the physical system. With this, we would be able to ascertain the limiting factors of the simulator and thus the scale of environments we could realistically implement.

4.2.1 Object Distribution - Brax V1

Sampling algorithms prove effective methods for distributing points in N-dimensional spaces. We can of course leverage this in our 2D setting as a means of distributing objects in the environment. One such approach, proposed by Bridson[61], is a Fast Poisson Disc Sampling algorithm, illustrated in Algorithm 2.

Algorithm 2: Fast Poisson Disc Sampling in N-Dimensions

Initialise k: the rejection threshold;
Initialise background grid array G for storing samples = -1 indicating no sample, with cell-size bounded by $\frac{r}{\sqrt{n}}$;
Select an initial sample $p_0 \sim \mathcal{U}[\zeta_i^{low}, \zeta_i^{high}], i = 1, ..., n$;
Insert x_0 into the background grid G and list of points P;
Initialise the active list $A = 0$;
while $A \neq \emptyset$ **do**
  Sample a random index i from the active list A;
  for $j \leftarrow 0, k$ **do**
    Generate a sample point p_j within the spherical annulus $[r, 2r]$ of x_i;
    if p_j *not within distance* r *of any existing sample in* G **then**
      emit p_j as the next sample and add to A, G and P;
      break;
    end if
  end for
  If we have found no such point p_j, remove i from A
end while

Through the strictly defined cell size, validating the placement of a point requires checking only cells in 5x5 mini-grid around a sample. Then, via the index based lookup, the algorithm can operate in linear time – producing N samples takes $2N + 1$ iterations of the $\mathcal{O}(k)$ loop. This is preferable to the non-linear complexity of naive *dart-throwing* mechanisms, especially in a system concerned with optimising performance.

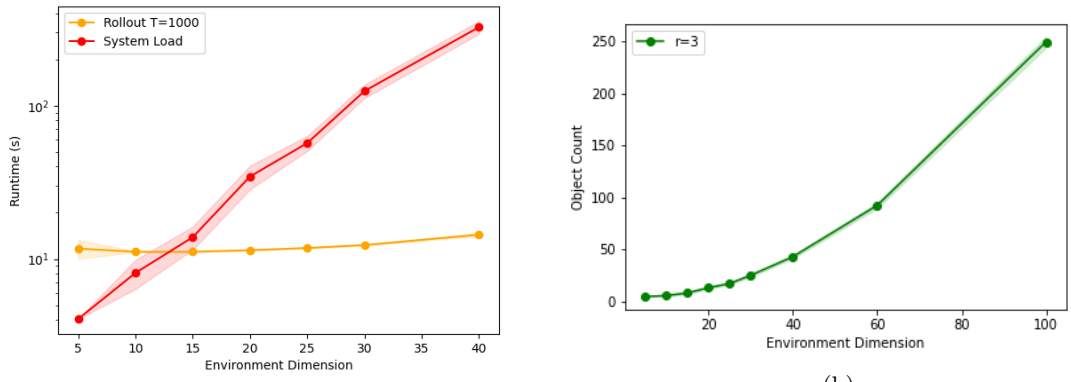
Extending this to distribute objects in a 3D environment is naturally quite simple. Firstly, our agents operate on a grounded plane, and such we only care about sampling in 2 dimensions here. Then, to ensure separation by r, we sample instead within the annulus defined by $[r + h_{max}, 2r + h_{max}]$, where h_{max} represents the maximum potential value of the randomised parameter governing the half-size of the spawned primitive bodies.

4.2.2 Early Benchmarks

Now with a scalable environment construction algorithm, we conducted experiments to benchmark the performance of object distribution, system loading and rollout execution in Brax environments of varying dimensions — *(width, height)*. As such, we could identify the limiting factors of the process; and with this foresight, then guide the project to either address them or better navigate their ineradicable constraints.

Brax Operations

Regarding Brax operations that fall within the scope of our framework, there are two that are largely dictated by our environments: loading a system from our `Config`, and executing a rollout. Whilst we have minimal influence on their performance, it's crucial to understand and consider their behaviour when designing a solution. Figure 4.3a shows the run-time increase of each operation, as the densely populated environment grows in size.



(a) (b)

Figure 4.3: Performance of initialising a `System` from `Config` and executing a 1000 timestep rollout – Figure 4.3a. For context, we include Figure 4.3b to illustrate the average object count resulting from Poisson point sampling with separation $r = 5$.

We observe from the plot that – in Brax-V1 – executing a JIT complied rollout remains computationally cheap, irrespective of environmental scale, whereas loading the `System` object has exponential time-complexity with respect to the environment dimensions. It's important to note the logarithmic scale of Figure 4.3a to appreciate the magnitude between the two operations.

Point Sampling

There are two critical observations from Figure 4.4: firstly, that naive and algorithmic dart-throwing have fundamentally identical runtime at any scale; secondly, as expected, that sampling run-time is negligible in its impact on the system performance prior to rollout, given that we must pay the cost of loading the system regardless.

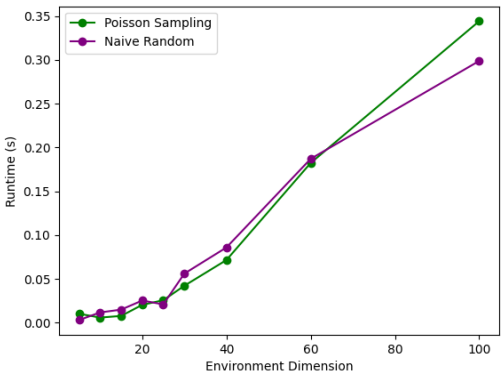


Figure 4.4: Performance between poisson sampling and naive rejection based sampling for object distribution, with identical rejection threshold $k = 30$

4.3 World Generation

4.3.1 Brax-V2 Update

On March 30th 2023, Google officially released the transition to Brax V2 in the v0.9.0 update[62]. This update deprecated a considerable amount of the previous functionality, opting for a design that is more analogous with the API of other prevalent physical simulators in the community [27, 10]. We detail here those changes relevant to the project direction, and illustrate the benefits and challenges that result from pivoting to the new API:

- **MJCF XML Parsing:** Brax previously created `System` objects through parsing their native `Config` objects, stored as Protocol Buffers. These objects defined the bodies present in the environment, and their respective colliders, joints and actuators, as well as the physics dynamics to use. Conversely, other simulators create models by loading XML files, specifically loading those in MuJoCo's native MJCF format, as well as the popular but limited URDF format. In deprecating the `Config` mechanism, Brax now fully supports and endorses loading models through XML instead. This allows us to both benchmark equivalent environments in different simulators, as well as directly utilise open-source code for parsing to MJCF strings in place of our original `Config` purposed implementation.
- **Generalised Physics Pipeline:** In the pursuit of unrivalled speed, Brax-V1 provided two fast – but inaccurate – physics pipelines to use for simulation: positional, which uses position-based dynamics[63]; and spring, which uses impulse-based methods as found in many video-games. With the addition of the generalised pipeline providing the accurate equivalent robot dynamics to MuJoCo, Brax-V2 enables more faithful comparisons for benchmarking across simulators.

4.3.2 MuJoCo-Worldgen: Full Acknowledgements[64]

As discussed, Brax requires an MJCF input file in order to load a `System`. To create these, we build a hierarchical tree of objects to parse into MJCF by employing open-source XML generation code published by OpenAI[64]. We detail only the code's behaviour in relation to our system, omitting low-level internal functionality; and note the minor contributions made to re-purpose the software for Brax compatibility.

Overview

The package generates environments in five distinct phases, we will explain only the initial and final stages:

1. Create a tree of all the objects.
2. Recursively compute names for every element in the tree.

3. Determine the sizes and relative positions of the objects.

4. Determine the absolute position of the objects.

5. Generate the system XML and initial environment state.

Stage 1

The world-generation is built around the base `Obj` class from which all objects inherit. One of the key external methods is the `append()` function. It takes as input a `child_obj`, a `placement_name` and a `placement_xy`. Placements are world-aligned rectangular prism spaces where we can place and orientate objects. An object can define multiple placements for its children e.g. "top" or "inside". In the `append()` method, the `child_obj` is then added to the object tree as a child node of the caller in the specified `placement_name`. If a `placement_xy` value is provided, the object will be placed at the provided (x, y) position within the placement bounds; otherwise, the position will be random.

The `Geom(Obj)` class is used to represent physical primitive types in the object tree, for example: "box", "sphere" and "cylinder". When constructing a `Geom` object, we can pass also a size parameter, or lower and upper bounds for domain randomisation instead.

The developers propose the following paradigm for world-building: create a `Worldbuilder` object and append a `Floor` object to it as the root node. From there, continue appending desired objects and as such, the environment is constructed upwards from the floor; this ensures all objects are directly or indirectly connected to the ground.

Stage 5

The final phase culminates the process with two tree traversal functions:

- `to_xml_dict()` – the object recursively builds an XML dictionary: it adds XML representing itself, then updates the dictionary with the XML of all its children by invoking their texttttto_xml_dict() function. It then finally applies any stored transforms, modifying the XML in place to set generic attributes, for instance, mass, friction or contact group.

- `to_xinit()` – the object recursively builds a dictionary of initial `q` and `qd` values for the joints in the system. Once again, the object adds its own information, then updates the dictionary with the data of its children.

Finally we note the `ObjFromXML(Obj)` and `ObjFromSTL(Obj)` classes, that enable loading objects from XML and STL files respectively. These allow us to load assets for non-primitive types, such as ramps, that must be defined using a mesh.

MAax Contributions

- **Introduction of Free Joints:** With a view to maintain compatibility with the `spring` and `positional` back-ends, only bodies with 1, 2, 3 and 6 DOF are supported in Brax[65]. Since MuJoCo supports bodies with 0-6 DOF, the `worldgen` package is very loose on joint constraints. We refactor the code to support object generation with Brax's only option for 6-DOF – the "free" joint, and introduce the relevant constraints for "slide" and "hinge" joints.

- **Static Objects:** Only a single ground body can have 0-DOF in Brax; in MuJoCo a body with 0-DOF can exist as a fixed object. We replace the package's static object mechanism: instead of stripping joints from marked bodies, we define a new `Fixed` object class that imposes joint `limit` attributes to fix objects in place.

- **Naming Conventions:** We reform the recursive body and joint naming conventions that conflict with the rollout visualisation of the Brax HTML renderer.

4.3.3 Modular Environments

We begin modular environment construction from our `Base` class. When creating the environment's physical system we begin by instantiating a `worldgen.Worldbuilder` object and append a `worldgen.Floor` object to it as the root node. From here, we desire an abstraction of the low-level object placement, in both a modular and scalable fashion. We achieve this separation of concerns by dividing environment features into distinct components that we can add to the `Base` environment – modules.

Modules

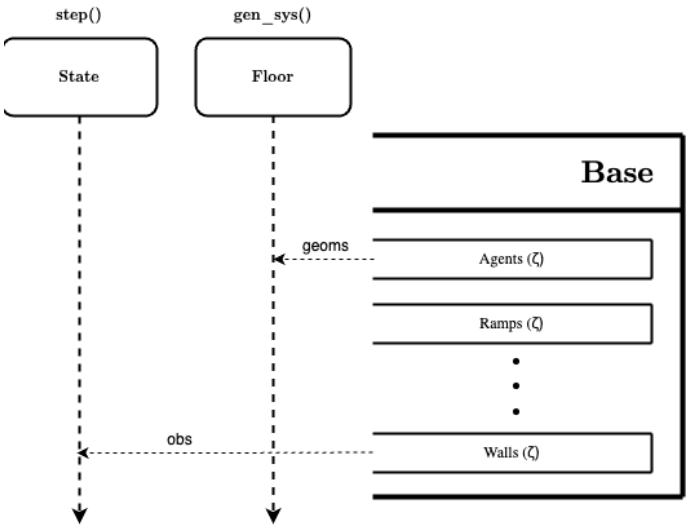


Figure 4.5: State and World-building interaction of modular MAax environments. During transitions, each module sequentially adds observations to the state. At the world-building stage, object modules append their geoms to the floor object.

In our paradigm, modules serve to specify the classes of objects present to generate in the environment, as well as their associated domain randomisation parameters ζ, we detail these in Appendix A. Each module inherits from the abstract `Module` class, implementing three methods:

- `module.build_step()` – tries to find a valid placement for the module respective objects, and appends them to the `worldgen.Floor` object. Returns whether the placement was successful or not.
- `module.cache_step()` – once the `System` is generated, cache module constants for instant lookup during rollouts, e.g. `q` and `qd` indices of the module's respective objects in the system.
- `module.observation_step()` – create and/or reformat any observations specific to the module. For example, adding wall position observations to the `state`, representing them as the centre and width of the wall gaps.

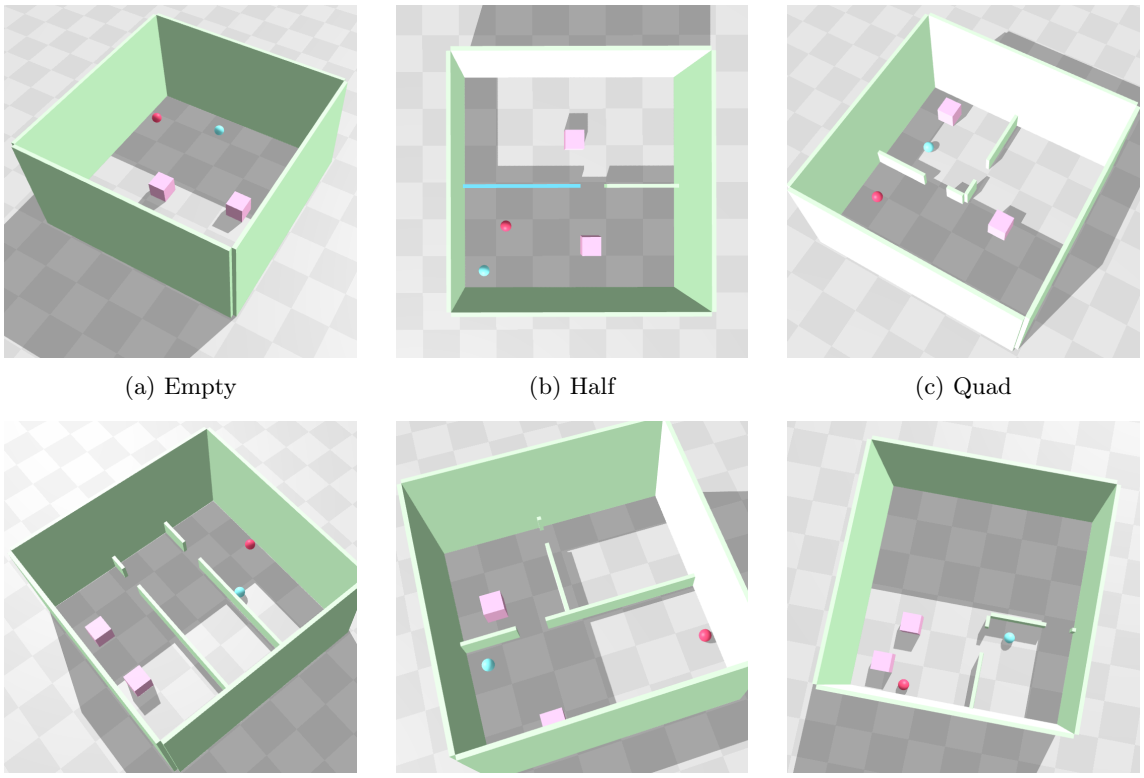
When constructing a physical system or handling a state transition, the environment iterates through each of the applied modules and invokes the relevant `step()` function, this interaction is visually captured in Figure 4.5. In this way, each module entirely encapsulates its own functionality and thus can be interchanged in any and all combinations.

Rooms

The wall modules enable a variety of room types, with differing structural complexity. We implement random wall placements and a selection of preset wall scenarios that create room-like areas in the space. Currently, we implement five different wall scenarios, presented in Figure 4.6:

- `empty` – the environment contains no walls

- `quad` – an exact quadrant of the environment is sectioned, with random doors placed along the walls.
- `half` – a wall divides the environment in half, with a single random door.
- `tri` – the environment is sectioned into three rooms. A wall divides the environment in half, with a single random door, then another wall divides one of the halves into quadrants, each with a random door.
- `var_quad` – a randomly sized quadrant of the environment is sectioned, with random doors placed along the walls.



(a) Empty (b) Half (c) Quad

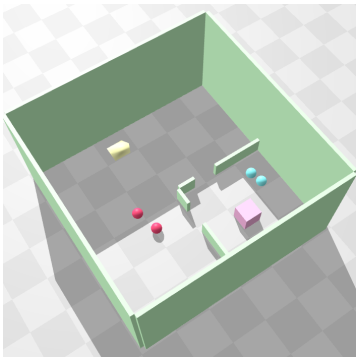
(d) Random (e) Tri (f) Variable Quad

Figure 4.6: Sample environment for each of the wall scenarios.

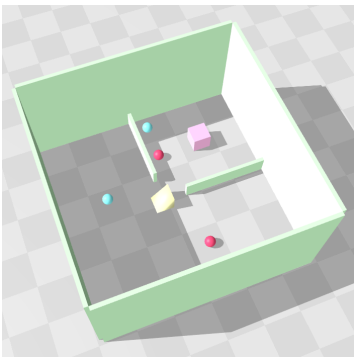
Placement Functions

Many of the object modules are initialised with a `placement_fn` argument. This function specifies how to sample the location of objects on the placement grid with respect to the room scenario. We include six functions within the current framework, though any additions are easily implementable for researchers adding custom tasks:

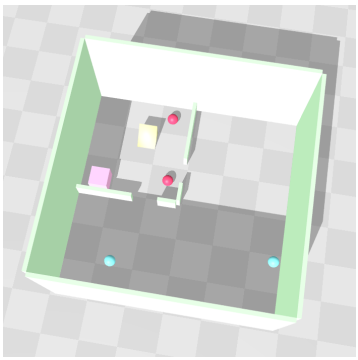
- `uniform_placement()` – uniformly samples a random position within the constraints of the grid:
- `centre_placement()` – samples the position in the exact centre of the placement grid:
- `separation_placement()` – attempt to sample a position outside radius r of all other placed objects:
- `proximity_placement()` – samples a position within radius r of a specified placed object $q = (x_j, y_j)$:
- `quad_placement()` – given we are using a quadrant room scenario, uniformly sample a random position within the walls defining the quadrant:



(a) Hider and Seeker proximity. Initial hider placed in the quadrant, initial seeker placed outside.



(b) Uniform placements with a central ramp.



(c) Single separated hider, with seekers placed in the quadrant.

Figure 4.7: Environments illustrating a range and combination of placement functions.

- `outside_quad_placement()` – given we are using a quadrant room scenario, uniformly sample a random position outside the walls defining the quadrant:

In the current implementation, all the placement functions apply to every object specified by the module, i.e. objects of the same type will be placed in the same manner. Figure 4.7 displays identically seeded environments, using these different functions.

We also provide here an abstraction of the `make_env` function to demonstrate the simplicity of our API, Listing 4.1. We present this same function in our documentation as exemplar usage of creating MAax environments.

```
seed = 1
def make_env(n_frames, floor_size, grid_size, door_size, n_hiders, n_seekers,
    n_boxes, n_ramps, scenario="quad", ...):
    """
    Create an environment
    """

    n_agents = n_seekers + n_hiders
    env = Base(n_agents, n_frames, grid_size, seed)

    # Define the placement functions
    agent_placement_fn = quad_placement
    ramp_placement_fn = uniform_placement


    # Add all desired modules
    env.add_module(WallScenarios(grid_size,
                                 door_size,
                                 scenario,
                                 ...)

    env.add_module(Agents(n_agents,
                          agent_placement_fn,
                          ...)

    ...

    if jp.max(n_ramps) > 0:
        env.add_module(Ramps(n_ramps,
                             ramp_placement_fn,
                             free,
                             ...)
    return env
```

Listing 4.1: Simplicity of the MAax API for creating custom environments, some parameters omitted for clarity.

Chapter 5

Tasks and Mechanics

In this chapter, we present how the MAax framework facilitates task design and underlying mechanics by proceeding through the implementation of our sample task, hide-and-seek.

5.1 Mechanics

To facilitate the complex human-relevant tasks we are interested in, we may wish to extend the agents' action space beyond motor control to provide the capacity for abstract interaction with objects in the environment. Alternatively, we may want to constrain agents to a specific area of the environment, or perhaps restrict their observations to activity within their central vision. Here, we refer to all such augmentations escaping the representation of the underlying physical system as the task *mechanics*.

5.1.1 Wrappers

In our paradigm, adding additional task mechanics is achieved via implementing a `Wrapper` class. This is a commonplace method within RL software, particularly for tasks implemented in Gym, which provides extensive documentation on the subject[66]. Ultimately, wrappers implement modular transformations to an environment, and within the scope of adding mechanics, there are three things we may want to perturb:

- **Actions**, before applying a physics step in the base environment.
- **Observations**, returned from the base environment.
- **Rewards**, returned from the base environment.

In Gym, these are implemented by inheriting from an `ActionWrapper`, `ObservationWrapper` and `RewardWrapper` respectively. As such wrappers are currently not defined in Brax, we implemented a suite with equivalent functionality within MAax to help standardise our software where possible.

5.2 Tasks

5.2.1 Hide-and-Seek

For our case study task – hide-and-seek (H&S) – there are several fundamental game mechanics, as well as plethora of potential extensions that we see in the work of Baker et al.[4].

Actions

Considering only the core game, the action space of H&S is extremely simple and fully captured by the physical system: the hiders must move around the space and avoid contact with a seeker. However, when played by humans, we often profit through interacting with the environment, for instance, covering ourselves with objects or closing a wardrobe door. To learn such complex human-relevant behaviours, we require the capability of interactive tool-use.

To balance the game and broaden hider strategy, we allow agents to 'lock' a subset of objects in the scene. Alongside their movement, each agent has a 'locking action' holding a binary value for each object in the scene; a 1 represents that the agent desires to lock that object, and 0 to either unlock or leave the object undisturbed. For each object, we compute a mask describing, which agents are within a suitable distance to lock the object and desire to. The behaviour is then determined by the chosen locking mechanic:

- `any` – if any agent that is permitted wants to lock the object, it will be locked.
- `all` – all agents within the proximity must want to lock the object.
- `any_specific` – any agent may lock the object, however only that agent may release the lock.
- `team_` – like `all`, however, all team members of the locker must concur to unlock.

To achieve this internally, we set the joint limits of target objects to be equal to their current `q` value. Essentially this fixes the object in place, however we do sometimes observe oscillations when these objects experience a considerable external force. We imagine this is related to performing multiple pipeline substeps within each abstract environmental timestep. In the current implementation, we facilitate this mechanic only for boxes.

We also implement the mechanic of a 'preparation phase', mimicking the short period where hiders disperse and the seeker performs a countdown. Generally, in this phase, the seeker is restricted in both mobility and sight. In our simulation, we use an `ActionWrapper` to null seeker actions for a designated initial percentage of each episode.

Observations

The observation space is comparatively more intricate. A considerable proportion of the observations are constructed by modules, being the global position and rotation of the objects. Regarding wrappers, we currently provide agents with three additional pieces of information. Firstly, a binary array informing agents of the team allocation of all the competitors. Secondly, we expose the remaining duration of the preparation phase to allow hiders to accommodate this knowledge into their strategy. Finally, we provide agents with a contact mask: a matrix informing who they are in contact with. We also later use this for reward calculation.

We then offer a `SplitObservations` wrapper, this divides observations into four categories and constructs a tailored observation array for each agent:

- `self` – observations that are agent-specific. For instance, we permute agent (`q`,`qd`) values and the `hider` observation such that each agent sees its own values in the first position, with consistent ordering on all observations regarding the other agents.
- `copy` – observations that are passed through unchanged as they are already correctly shaped.
- `self_matrix` – observations representing a custom observation of another agent, where the pairwise interaction may be asymmetric. i.e. an observation shaped (`n_agents, n_agents, d`).
- `external` – observations that are copied over to each agent.

To facilitate environmental complexity scaling, whereby the number of objects or agents in the scene can change between episodes, we implement a wrapper to spoof entities. In this way, we can match the observation shape that would occur with the maximum quantity of each variable. This is mandatory for preparing the usage of a policy network, as we require a fixed input shape for both training and inference.

Rewards

Since Brax isn't equipped with native ray-casting, we chose not to introduce a likely expensive custom implementation for LIDAR-based observations. As a result, the objective of our task is not to evade detection, but rather contact. We implement a selection of dense reward functions to explore the consequences of incentivising either selfish or cooperative play:

- `selfish` – Seekers receive a reward of 1 if they are in contact with any hider and -1 otherwise. Hiders receive a reward of 1 if they are distanced from all seekers and -1 otherwise.
- `team_mean` – Hiders and seekers receive the mean reward of their team, calculated as above.
- `team_zero_sum` – Hiders receive 1 only if all are currently uncaught. Seekers receive 1 if any seeker catches a hider.

To clarify, catching a seeker doesn't signify the end of episode for either party; such a sparse reward makes the task very difficult to solve. Furthermore, as Brax is built upon JAX and uses static arrays, we are constrained by their immutability and must simulate all the steps for each episode. Thus, we argue fully leveraging a dense reward is in this case optimal. We can consider this mechanic to engineer two phases of game-play: approach and pursuit, from which we believe different behaviours will emerge.

The second feature of the 'preparation phase'is to disable rewards for its duration. As such, there is no explicit reward for preparation strategies such as pushing boxes or ramps. After all, we have no notion of intrinsic motivation in our investigations; we instead wish for our agents to discover meaningful interactions through induced curricula, which we know to scale better with increased environmental complexity[4].

Policy

For intelligibility, we internally handle our observations in a dictionary mapping strings to `jax.ndarrays`; we achieve this by wrapping the higher level Brax `State` data-class with additional fields. However, the internal implementations of training algorithms such as PPO expect a flattened array for each network. Figure 5.1 visualises how we prepare observations for a policy network, as well as the output action space of the task.

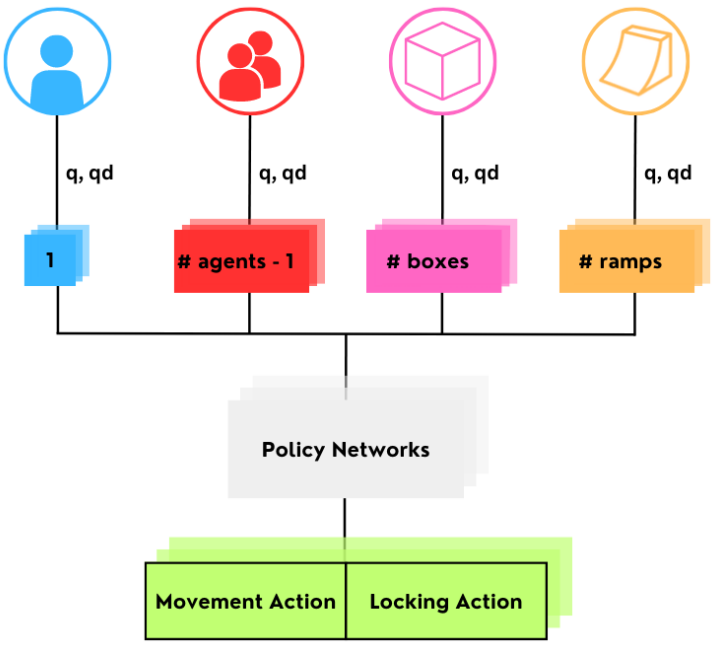


Figure 5.1: How MAax observations are embedded for policy training. Supplementary options such as door locations and remaining preparation time are omitted for simplicty.

We include Listing 5.1 to demonstrate how mechanics can be sequentially applied when creating a MAax environment. Once again this occurs in the example `make_env` function.

```
def make_env(..., n_hiders, n_seekers, prep_fraction, ...):
    """
    Create an environment
    """

    ...
    # Module-relevant code
    ...

    env = TeamMembership(env, team_indices)
    env = AgentAgentContactMask2D(env)

    # Define the task-reward
    env = HideAndSeekRewardWrapper(env, n_hiders, n_seekers,
                                   rew_type)

    # Define the preparation phase, and its relative duration
    env = PreparationPhase(env, prep_fraction)

    # Assign agent specific and global observations
    env = SelectObsWrapper(env, keys_self, keys_other)

    # Flattens observations for Brax training pipeline
    env = FlattenDictObsWrapper(n_agents)

    return env
```

Listing 5.1: API of MAax for applying task-mechanics, some parameters are abstracted for clarity.

Chapter 6

Evaluation

In this chapter, we evaluate the MAax framework against our problem statement, as defined in Section 4.1. We consider a range of quantitative metrics and conduct several benchmarks to carry out both internal and external analysis of the performance. Finally, we offer some brief qualitative thoughts on the software with respect to current state-of-the-art alternatives.

6.1 Modularity

Measuring the modularity of our software involves considering dependencies and conflicts between modules of code. We consider first the module classes themselves.

WorldGen

Regarding the four main modules for environment generation: agents, walls, boxes and ramps, there exists no explicit dependencies between them. Indeed, whilst several of the placement functions presume the existence of walls in the environment, they are not required for functionality. However, we note that the `proximity_placement` depends on an external object, and is not yet implemented to handle its absence. Finally, the Brax pipeline requires at least one active body, and as such, we can consider the agents module to be mandatory.

Mechanics

Our mechanics wrappers are comparatively slightly less independent. We identified several dependencies between them that we believe could potentially be refactored during future work:

- **Preparation Phase:** The preparation phase is implemented via an `ActionWrapper` to nullify the movement, and a `RewardWrapper` to nullify the reward. We could likely unite these and eliminate the dependency.
- **Selecting Observations:** When selecting which observations to export for training, we assume that they have been reshaped and assigned amongst the agents by the `SplitObservations` wrapper.
- **Catching Agents:** The wrapper implementing H&S reward assumes the agent-agent contact mask has been calculated, and as such, depends on that wrapper.

In summary, we believe that this list is extremely minimal, demonstrating considerable success in implementing a modular framework. We also emphasise again the modular nature of environment generation that facilitates spawning objects in any desired combination.

6.2 Scalability

To evaluate the scalability of our environments, we reflect on the extent of the customisation available for each of the world generation modules:

- **Base**: We are able to scale the environment size, and customise world parameters such as floor friction and gravity.
- **Agents**: We can select any number of agents on each team, although this must remain constant throughout training.
- **Objects**: We can set a quantity range for each object type, and gradually increase this lower bound over time for incremental complexity scaling.

In conclusion, we facilitate environments of any size or complexity, limited only by the memory constraints of the simulator and performance degradation at very large sizes. We urge readers to scan Appendix A for full details on domain randomisation.

6.3 Performance

The critical metric given the context of our work is performance; we evaluate this by analysing the execution time of rollouts under varying environmental conditions. For all experiments, we use a horizon $\mathcal{T}$=1000, unless stated otherwise. The hardware and parameters used for each experiment are detailed in Appendix B. We also maintain a constant environment throughout the experiments; we define here the standard MAax environment:

$$\mathcal{M} = [(2, 2), Quad, 1, 1] \tag{6.1}$$

Environment Generation: Agents, Walls, Boxes and Ramps.

6.3.1 Internal Analysis

Object Influence

In order to investigate the impact of the quantity and type of objects in the scene on rollout performance, we conducted a basic experiment comparing the runtime of episodes containing either: boxes, a primitive type; or ramps, a non-primitive geometry created using a mesh. This informs us of their influence on the performance of the internal Brax physics pipeline.

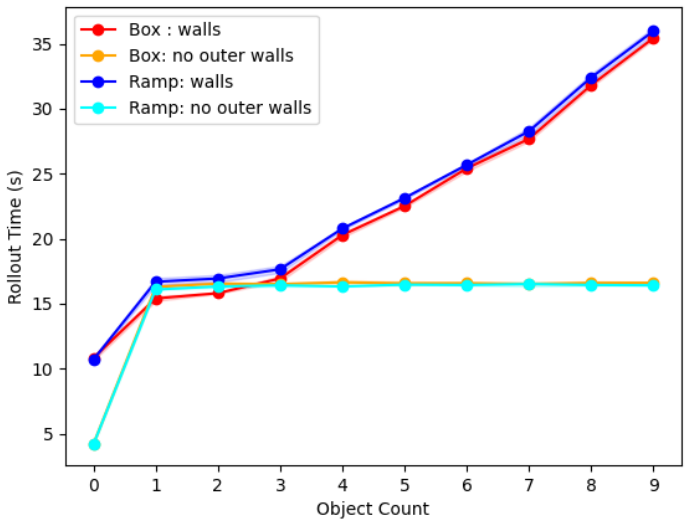


Figure 6.1: Performance of rollout execution for various object types and quantities. We use measure the non-batched JIT compiled run-time with $\mathcal{T} = 1000$. The environments also contain 2 hiders and 2 seekers both to provide the existence of an action space, and place the result within context of our task.

We observe from Figure 6.1 that introducing more objects has a detrimental effect on simulation performance, likely as more collisions need to be calculated. We also note that, despite ramps consisting of a mesh that is likely more graphically complex, they have almost identical impact on performance.

Most significantly though, we notice that the presence of walls is considerably detrimental to performance, regardless of other entities in the scene. Following this, we considered whether we could alleviate their burden slightly and study the performance of these lighter environments. Without the time or requisite context to analyse the internal workings of Brax, we instead introduced a new mechanic to imitate the behaviour of the bounding walls. By penalising agents who stray outside the numerical bounds, we can constrain their location to the game-area without a physical enclosing. We provide a visualisation of such an environment in Figure 6.2

Finally, we mention that replacing the 6-DOF ('free' joint) used in the experiment with 3-DOF (double 'slide' and a 'hinge' joint) had negligible impact on the execution time for either object.

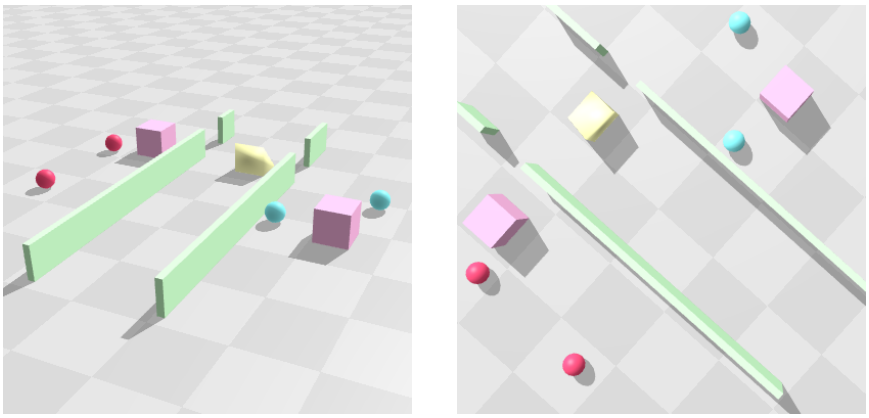


Figure 6.2: Rendering of the standard MAax environment $\mathcal{M}$ with the external walls stripped. This represents the 'no walls' environment referenced in later experiments.

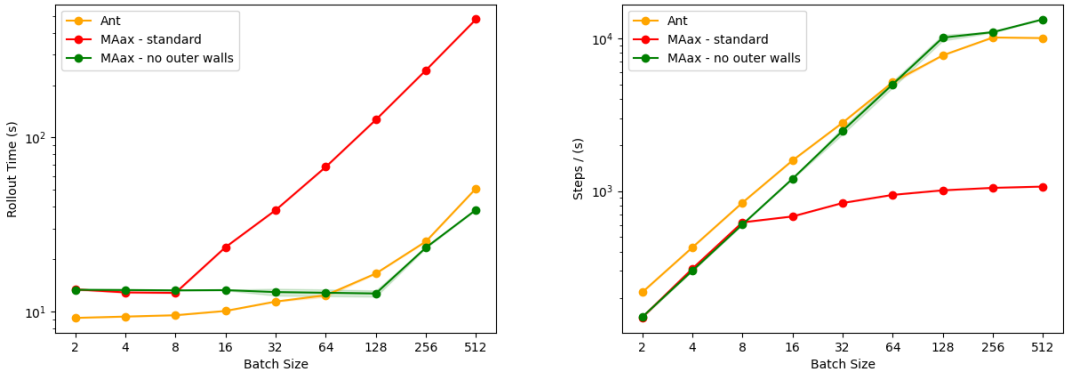
Parallelism

We next evaluate the key motivation of our work: the suitability of MAax environments to massive parallelism. To measure this, we investigate the effect that increasing batch size has on the simulation runtime and throughput of MAax environments. We start from a batch size $\mathcal{B} = 2$ and double it until we reach a plateau, or run out of GPU memory. In our first experiment, we compare the standard environment $\mathcal{M}$ to the common Ant task in Brax. We modify the constraint solver iterations and `dt` to match for both tasks. Finally, having seen the dramatic influence the walls have on our performance and presented a reasonable alternative, we also measure an identical MAax environment without these external physical bounds.

We quantify this throughput through the number of timesteps per second (steps/s). To obtain this metric, we take the total number of timesteps performed $\mathcal{B} \cdot \mathcal{T}$ and divide this by the runtime of the simulation t_n. We then take the mean of this value across N iterations:

$$steps/s = \frac{1}{N} \sum_{n=1}^{N} \frac{mathcalB \cdot \mathcal{T}}{t_n} \tag{6.2}$$

We observe the results of this experiment in Figure 6.3. We note that the standard environment $\mathcal{M}$ plateaus significantly beyond a batch size of 8; evidently there are too many rigid bodies for the Brax system to be performant. We are very keen to target this limitation in future research. Although, we emphasise that, without physical external walls, the MAax environment displays similar (if not better) scalability with increasing batch size to the extremely performant Ant task, only plateauing once we reach the limit of the device. Since we have evidence of the scalability of the Ant task with improved hardware[9], we claim that these results demonstrate MAax's capacity



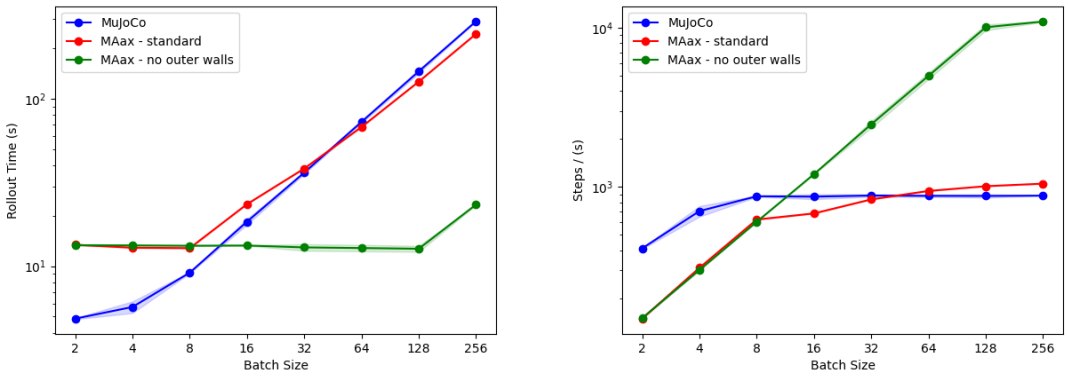
(a) Total Simulation Runtime (b) Simulation Throughput

Figure 6.3: Comparison of average total simulation runtime(Figure 6.3a) and throughput(Figure 6.3b) between MAax environments and the Brax Ant task with increasing batch size. Note that both axes use a logarithmic scale.

to leverage massive parallelism and lie amongst the Brax tasks performing millions of simulation steps per second on advanced TPUs[9].

6.3.2 Engine Comparison

Perhaps more importantly than analysing any internal metrics is to ascertain where our framework lies in relation to the existing state-of-the-art. We draw attention to the work of Baker et al[4] who also implemented a hide-and-seek task, however, in MuJoCo[27]. Since we represent our environments as MJCF files, we can load identical simulations into MuJoCo and perform a direct comparison. Evidently, we can no longer apply any mechanics to the simulation as they are implemented to be Brax specific, however, these would only serve to slow the simulation anyway. Furthermore, given our environments utilise the same XML generation code[64] as Baker et al., we can have confidence that both environments are constructed upon the same physics primitives, therefore, our results are not skewed by discrepancies that may have existed otherwise.



(a) Total Simulation Runtime (b) Simulation Throughput

Figure 6.4: Comparison of average total simulation runtime(Figure 6.3a) and throughput(Figure 6.3b) between MAax environments and an equivalent MuJoCo system with increasing batch size. Note that both axes use a logarithmic scale.

We perform an identical experiment as with the Ant task, observing both the total runtime and throughput across batch sizes against a MuJoCo environment utilising parallelism across CPU cores. We observe the results of this experiment in Figure 6.4 and summarise them, alongside the Ant task, in Table 6.1. We notice that the scaling of the MuJoCo instance is far more limited

than the unbounded MAax environment and is out-performed by an order of magnitude. This is an expected result; each CPU core is running a separate rollout instance, and therefore, once the batch size reaches the fixed number of CPU cores (8), we no longer receive benefits from parallelism. Additionally, we note that the MuJoCo environment experienced negligible benefits when stripping the external walls. It's likely that the engine handles rigid-bodies very differently to Brax.

We emphasise that the standard MAax environment is limited only by the engine performance at high rigid body counts. As such, given that this can likely be optimised in future work or mitigated through instead employing an incorporeal boundary, we demonstrate that MAax is limited only by the hardware available and thus represents a far more scalable and performant solution than current alternatives.

Physical System	Simulator	Resources	Steps/s	Batch size
MAax - $\mathcal{M}$	Brax[9]	GPU Quadro 6000	**1,069**	512
MAax - no outer walls	Brax[9]	GPU Quadro 6000	**13,302**	512
Ant	Brax[9]	GPU Quadro 6000	10,115	256
MAax - $\mathcal{M}$	MuJoCo[27]	8 CPU-cores	883	32

Table 6.1: Maximum simulation throughput obtained and their corresponding batch size across tasks. We report the mean value across 5 iterations.

6.4 Usability

As illustrated in the code listings of Sections 4 and 5, the MAax API requires a user to sequentially demand environmental features and mechanics for a given task. We believe this represents a simplistic paradigm for external use, however we note that for more complex tasks, definition could be cluttered. Whilst it may not fall under our influence, as we desire for researchers to create their own tasks and mechanics, we recognise that perhaps we could package common modules or mechanics into a composite module. However, this requires a decomposition of the task itself, which is not always a trivial process.

Often, we use the MAax API within a Jupyter Notebook; this is analogous with how Brax is utilised within current research[8, 62]. As such, we claim that – in adhering to the community standard – our API is intelligible and can be readily incorporated into ongoing research.

Chapter 7

Conclusion

7.1 Summary

In this research, we have introduced MAax: a novel accelerated framework for creating modular and scalable environments for MARL tasks.

In Chapter 4, we constructed the modular world generation paradigm and demonstrated the ability to create extremely diverse environments from only 5 different types of entity. Furthermore, we parameterised the domain randomisation with value ranges to facilitate incremental complexity and diversity scaling.

In Chapter 5, we considered how the framework could be leveraged for competitive multi-agent tasks, conducting our investigation via a case study of the Hide-And-Seek game. We met this challenge with a suite of differentiable components implementing a range of task mechanics. We then extended this with a set of utility classes, enabling users to easily select and reshape observations to be used in policy training.

Finally, we evaluated our method across differing tasks and engines. Our experimentation uncovered several undocumented properties of Brax and, more significantly, classified the performance of our environments in relation to the state-of-the-art. We find MAax environments can leverage massive parallelism to execute thousands of simulation steps per second, limited only by the hardware itself; this places them alongside the established performant tasks that reach millions of steps per second on advanced TPUs[9]. As such, our approach proves a considerably more scalable solution than existing implementations[4], with the potential to outperform them by several orders of magnitude. We believe this can revolutionise the landscape for researching intelligent behaviour in competitive multi-agent tasks and are excited to see developments that arise as a result.

7.2 Ethical Considerations

As discussed, reinforcement learning algorithms are becoming increasing more powerful, especially in their ability to generalise to unforeseen problems[3]. As such, although we have no specific application in mind, this still raises the potential issue of misuse. Additionally, with our contribution and ongoing research in the Sim2real domain, malicious actors could utilise the agents to propose a real threat in the physical world.

Another consideration is our use and handling of data. The bench-marking tasks and simulators we use for our experiments contain no personal data. Though, as we drive these agents to learn more human-centric behaviours, we expect them to influence and interact with real people in the near-future. As work in this area continues, we urge researchers to be mindful of both the practical applications and potential training bias of these agents.

Whilst we hope the scalability of our work provides greater accessibility into the domain of Multi-Agent Reinforcement Learning, particularly for those in emerging economies and developing academic groups, we recognise that we also benefit holders of large amounts of computational resources.

Often, these will be people in positions of power who can leverage reinforcement learning for political or personal goals, disregarding the detrimental effect on wider society. Furthermore, this risks the notion of 'buying' results[67] and tilting the research landscape away from a desirable meritocracy.

Finally, we note the environmental impact of machine learning. By reducing previous computation times by several orders of magnitude, we once again hope our work can contribute in a positive manner. Nonetheless, there is a possibility that the introduction of an efficient framework could lead to a surge in computation dedicated to Reinforcement Learning problems, bearing similarity to the phenomenon observed when constructing a new major road, where counter-intuitively, traffic flow often increases[68].

7.3 Future Work

As this is very much an enabling project, the landscape for future work is especially broad. We briefly present several directions in which the project could be extended, potentially for a PhD program.

Policy Training

Primarily, having implemented the capacity to easily modify the environment and output observation space for each agent, we are particularly interested in training policies for the hide-and-seek task, as we believe our environments and mechanics expose the requisite complexity to learn intelligent strategies concerning tool-use. This would require extending the relatively simple-to-use native Brax PPO implementation to the multi-agent case by using independent policy networks.

New Tasks

Following this, we should explore introducing new tasks. Although our paradigm essentially delegates the freedom of task definition to a user, we feel a responsibility to assess the transferability of the framework beyond the case study we have currently constrained ourselves to. Reasonably adjacent games include Capture-The-Flag, as researched by Jaderberg et al.[36], or Tag, in which we could investigate more actively utilising the 3-dimensional space when creating structures.

Physics Pipelines

Operating under the generalised physics pipeline provides the most accurate physics and allows us to draw direct comparisons to state-of-the-art alternatives[27]. However, accurate physics are likely not critical to a task such as hide-and-seek. We should explore the impact of transitioning to a different pipeline and observe if the performance increases justify the accuracy decline. This would involve several minor changes to the XML generation and `System` logic.

Open-Source Contribution

In the development of our work, we have communicated with several of the Brax developers to better understand some of the underlying design decisions of the software. Since our framework sits upon the simulator, we would be interested in making performance related contributions to the engine in areas we identified as bottlenecks in our research. For example, introducing new physics primitives or improving performance when loading from MJCF.

Quality-Diversity

Much of the research in our group is not only concerned with intelligence, but also adaptability. We believe the diverse behaviours learned through QD algorithms could translate well into the MAax environments. For instance, with the removal of a ramp, wall or teammate, agents would be forced to adapt by employing an alternative behaviour in their repertoire.

Appendix A

Domain Randomisation Parameters

Base

- `floor_size`: (`float`, `float`) – size (in Brax units) of the bounded environment.
- `grid_size`: (`int`) – size of the grid used to place objects on the floor.

RandomWalls

- `n_rooms`: (`int`) – number of rooms to create.
- `min_room_size`: (`int`) – minimum size a room (in grid cells).
- `door_size`: (`int`) – door width (in grid cells).
- `outside_walls`: (`bool`) – whether to place bounding walls around the environment.
- `random_room_count`: (`bool`) – if `True` randomise the number of rooms between 1 and `n_rooms`

WallScenarios

- `door_size`: (`int`) – door width (in grid cells).
- `scenario`: (`string`) – room wall scenario to select
- `p_door_droput`: (`int`) – probability that we don't place one of the doors

Agents

- `n_agents`: (`int`) – number of agents.
- `placement_fn`: (`fn` or [`fn`]) – placement to use for each agent, if a list is supplied, we assume one function per agent.

Boxes

- `n_boxes`: (`int` or (`int`, `int`)) – number of boxes, if Tuple, we sample uniformly between the given range.
- `placement_fn`: (`fn` or [`fn`]) – placement to use for each box, if a list is supplied, we assume one function per box.
- `box_size`: (`float`) – size of the box (in Brax units).
- `box_mass`: (`float`) – mass of the box (in Brax units).
- `box_friction`: (`float`) – box friction.
- `free`: (`string`) – if `True` we use a 'free' joint, otherwise two 'slide' joints and a single 'hinge' joint.

Ramps

- `n_ramps`: (`int` or (`int`, `int`)) – number of ramps, if Tuple, we sample uniformly between the given range.
- `placement_fn`: (`fn` or [`fn`]) – placement to use for each ramp, if a list is supplied, we assume one function per ramp.
- `ramp_friction`: (`float`) – ramp friction.

Appendix B

Experimentation Resources

Attribute	MAax (Brax)	MAax(MuJoCo)
Horizon $\mathcal{T}$	1000	1000
dt	0.005	0.005
Solver Iterations	5	5
Action Repeat	1	1
CPU Cores	4	8
GPU	Quadro 6000	*Unused*

Table B.1: Simulation parameters and hardware specifications for parallelism experiments.

Bibliography

[1] Alex Ray TM Glenn Powell. Automatic object XML generation for MuJoCo; 2019. Available from: https://github.com/openai/mujoco-worldgen.

[2] Sutton RS, Barto AG. Reinforcement learning: An introduction. MIT press; 2018.

[3] Duan Y, Chen X, Houthooft R, Schulman J, Abbeel P. Benchmarking deep reinforcement learning for continuous control. In: International conference on machine learning. PMLR; 2016. p. 1329-38.

[4] Baker B, Kanitscheider I, Markov T, Wu Y, Powell G, McGrew B, et al. Emergent tool use from multi-agent autocurricula. arXiv preprint arXiv:190907528. 2019.

[5] Brockman G, Dennison C, Zhang S. OpenAI-Five; 2018. Accessed on 05.06.2023. Available from: https://openai.com/research/openai-five.

[6] Tobin J, Fong R, Ray A, Schneider J, Zaremba W, Abbeel P. Domain randomization for transferring deep neural networks from simulation to the real world. In: 2017 IEEE/RSJ international conference on intelligent robots and systems (IROS). IEEE; 2017. p. 23-30.

[7] Freeman CD, Frey E, Raichuk A, Girgin S, Mordatch I, Bachem O. Brax–A Differentiable Physics Engine for Large Scale Rigid Body Simulation. arXiv preprint arXiv:210613281. 2021.

[8] Raghu A, Komorowski M, Ahmed I, Celi L, Szolovits P, Ghassemi M. Deep reinforcement learning for sepsis treatment. arXiv preprint arXiv:171109602. 2017.

[9] Lillicrap TP, Hunt JJ, Pritzel A, Heess N, Erez T, Tassa Y, et al. Continuous control with deep reinforcement learning. arXiv preprint arXiv:150902971. 2015.

[10] Mnih V, Kavukcuoglu K, Silver D, Rusu AA, Veness J, Bellemare MG, et al. Human-level control through deep reinforcement learning. nature. 2015;518(7540):529-33.

[11] Heess N, TB D, Sriram S, Lemmon J, Merel J, Wayne G, et al. Emergence of locomotion behaviours in rich environments. arXiv preprint arXiv:170702286. 2017.

[12] Tan M. Multi-agent reinforcement learning: Independent vs. cooperative agents. In: Proceedings of the tenth international conference on machine learning; 1993. p. 330-7.

[13] Bansal T, Pachocki J, Sidor S, Sutskever I, Mordatch I. Emergent complexity via multi-agent competition. arXiv preprint arXiv:171003748. 2017.

[14] Lim B, Allard M, Grillotti L, Cully A. QDax: on the benefits of massive parallelization for quality-diversity. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion; 2022. p. 128-31.

[15] Makoviychuk V, Wawrzyniak L, Guo Y, Lu M, Storey K, Macklin M, et al. Isaac gym: High performance gpu-based physics simulation for robot learning. arXiv preprint arXiv:210810470. 2021.

[16] Hartigan JA, Wong MA. Algorithm AS 136: A k-means clustering algorithm. Journal of the royal statistical society series c (applied statistics). 1979;28(1):100-8.

[17] Van Der Maaten L, Postma E, Van den Herik J, et al. Dimensionality reduction: a comparative. J Mach Learn Res. 2009;10(66-71):13.

[18] Andrychowicz OM, Baker B, Chociej M, Jozefowicz R, McGrew B, Pachocki J, et al. Learning dexterous in-hand manipulation. The International Journal of Robotics Research. 2020;39(1):3-20.

[19] Mnih V, Kavukcuoglu K, Silver D, Graves A, Antonoglou I, Wierstra D, et al. Playing atari with deep reinforcement learning. arXiv preprint arXiv:13125602. 2013.

[20] Tsitsiklis J, Van Roy B. Analysis of temporal-diffference learning with function approximation. Advances in neural information processing systems. 1996;9.

[21] Van Hasselt H, Guez A, Silver D. Deep reinforcement learning with double q-learning. In: Proceedings of the AAAI conference on artificial intelligence. vol. 30; 2016. .

[22] Hasselt H. Double Q-learning. Advances in neural information processing systems. 2010;23.

[23] Sutton RS, McAllester D, Singh S, Mansour Y. Policy gradient methods for reinforcement learning with function approximation. Advances in neural information processing systems. 1999;12.

[24] Silver D, Lever G, Heess N, Degris T, Wierstra D, Riedmiller M. Deterministic policy gradient algorithms. In: International conference on machine learning. Pmlr; 2014. p. 387-95.

[25] Mnih V, Badia AP, Mirza M, Graves A, Lillicrap T, Harley T, et al. Asynchronous methods for deep reinforcement learning. In: International conference on machine learning. PMLR; 2016. p. 1928-37.

[26] Fujimoto S, Hoof H, Meger D. Addressing function approximation error in actor-critic methods. In: International conference on machine learning. PMLR; 2018. p. 1587-96.

[27] Haarnoja T, Zhou A, Abbeel P, Levine S. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In: International conference on machine learning. PMLR; 2018. p. 1861-70.

[28] Schulman J, Wolski F, Dhariwal P, Radford A, Klimov O. Proximal policy optimization algorithms. arXiv preprint arXiv:170706347. 2017.

[29] Yang Y, Wang J. An overview of multi-agent reinforcement learning from game theoretical perspective. arXiv preprint arXiv:201100583. 2020.

[30] Todorov E, Erez T, Tassa Y. Mujoco: A physics engine for model-based control. In: 2012 IEEE/RSJ international conference on intelligent robots and systems. IEEE; 2012. p. 5026-33.

[31] Coumans E, Bai Y. PyBullet, a Python module for physics simulation for games, robotics and machine learning; 2016–2021. http://pybullet.org.

[32] Brockman G, Cheung V, Pettersson L, Schneider J, Schulman J, Tang J, et al. Openai gym. arXiv preprint arXiv:160601540. 2016.

[33] Frostig R, Johnson MJ, Leary C. Compiling machine learning programs via high-level tracing. Systems for Machine Learning. 2018;4(9).

[34] Gradu P, Hallman J, Suo D, Yu A, Agarwal N, Ghai U, et al. Deluca–A Differentiable Control Library: Environments, Methods, and Benchmarking. arXiv preprint arXiv:210209968. 2021.

[35] Heiden E, Millard D, Coumans E, Sheng Y, Sukhatme GS. NeuralSim: Augmenting differentiable simulators with neural networks. In: 2021 IEEE International Conference on Robotics and Automation (ICRA). IEEE; 2021. p. 9474-81.

[36] Pan Y, Cheng CA, Saigol K, Lee K, Yan X, Theodorou E, et al.. Agile Autonomous Driving using End-to-End Deep Imitation Learning; 2019.

[37] Chen T, Xu J, Agrawal P. A system for general in-hand object re-orientation. In: Conference on Robot Learning. PMLR; 2022. p. 297-307.

[38] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. Pytorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems. 2019;32.

[39] Jaderberg M, Czarnecki WM, Dunning I, Marris L, Lever G, Castaneda AG, et al. Human-level performance in 3D multiplayer games with population-based reinforcement learning. Science. 2019;364(6443):859-65.

[40] Chaffre T, Moras J, Chan-Hon-Tong A, Marzat J. Sim-to-real transfer with incremental environment complexity for reinforcement learning of depth-based robot navigation. arXiv preprint arXiv:200414684. 2020.

[41] Silver D, Schrittwieser J, Simonyan K, Antonoglou I, Huang A, Guez A, et al. Mastering the game of go without human knowledge. nature. 2017;550(7676):354-9.

[42] Berner C, Brockman G, Chan B, Cheung V, Dębiak P, Dennison C, et al. Dota 2 with large scale deep reinforcement learning. arXiv preprint arXiv:191206680. 2019.

[43] Hendrikx M, Meijer S, Van Der Velden J, Iosup A. Procedural content generation for games: A survey. ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM). 2013;9(1):1-22.

[44] Risi S, Togelius J. Increasing generality in machine learning through procedural content generation. Nature Machine Intelligence. 2020;2(8):428-36.

[45] Krizhevsky A, Sutskever I, Hinton GE. Imagenet classification with deep convolutional neural networks. Communications of the ACM. 2017;60(6):84-90.

[46] Perez L, Wang J. The effectiveness of data augmentation in image classification using deep learning. arXiv preprint arXiv:171204621. 2017.

[47] Weng L. Domain Randomization for Sim2Real Transfer. lilianwenggithubio. 2019. Accessed on 02.01.2023. Available from: https://lilianweng.github.io/posts/2019-05-05-domain-randomization/.

[48] Sadeghi F, Levine S. Cad2rl: Real single-image flight without a single real image. arXiv preprint arXiv:161104201. 2016.

[49] Rudin N, Hoeller D, Reist P, Hutter M. Learning to walk in minutes using massively parallel deep reinforcement learning. In: Conference on Robot Learning. PMLR; 2022. p. 91-100.

[50] Chebotar Y, Handa A, Makoviychuk V, Macklin M, Issac J, Ratliff N, et al. Closing the sim-to-real loop: Adapting simulation randomization with real world experience. In: 2019 International Conference on Robotics and Automation (ICRA). IEEE; 2019. p. 8973-9.

[51] Cobbe K, Hesse C, Hilton J, Schulman J. Leveraging procedural generation to benchmark reinforcement learning. In: International conference on machine learning. PMLR; 2020. p. 2048-56.

[52] Nair A, Srinivasan P, Blackwell S, Alcicek C, Fearon R, De Maria A, et al. Massively parallel methods for deep reinforcement learning. arXiv preprint arXiv:150704296. 2015.

[53] Babaeizadeh M, Frosio I, Tyree S, Clemons J, Kautz J. GA3C: GPU-based A3C for deep reinforcement learning. CoRR abs/161106256. 2016.

[54] Stooke A, Abbeel P. Accelerated methods for deep reinforcement learning. arXiv preprint arXiv:180302811. 2018.

[55] Liang J, Makoviychuk V, Handa A, Chentanez N, Macklin M, Fox D. Gpu-accelerated robotic simulation for distributed reinforcement learning. In: Conference on Robot Learning. PMLR; 2018. p. 270-82.

[56] Salimans T, Ho J, Chen X, Sidor S, Sutskever I. Evolution strategies as a scalable alternative to reinforcement learning. arXiv preprint arXiv:170303864. 2017.

[57] Gregg C, Hazelwood K. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer. In: (IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software. IEEE; 2011. p. 134-44.

[58] Mouret JB, Clune J. Illuminating search spaces by mapping elites. arXiv preprint arXiv:150404909. 2015.

[59] Erez T, Tassa Y, Todorov E. Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. In: 2015 IEEE international conference on robotics and automation (ICRA). IEEE; 2015. p. 4397-404.

[60] Terry J, Coumans E, Huang C. Update on Plans for MuJoCo; 2021. Available from: https://github.com/openai/gym/issues/2456.

[61] Huang C, Coumans E, Terry J. Replacing gym's MuJoCo envs with brax envs; 2021. Available from: https://github.com/google/brax/issues/49.

[62] Bridson R. Fast Poisson disk sampling in arbitrary dimensions. SIGGRAPH sketches. 2007;10(1):1.

[63] Freeman CD, Frey E, Raichuk A, Girgin S, Mordatch I, Bachem O. Brax V2 Release; 2023. Available from: https://github.com/google/brax/releases/tag/v0.9.0.

[64] Müller M, Heidelberger B, Hennix M, Ratcliff J. Position based dynamics. Journal of Visual Communication and Image Representation. 2007;18(2):109-18.

[65] Alner T, Frey E. Unsupported Joints in Brax; 2023. Available from: https://github.com/google/brax/issues/358.

[66] OpenAI. Gym Wrappers; 2022. Accessed on 07.06.2023. Available from: https://www.gymlibrary.dev/api/wrappers/.

[67] Schwartz R, Dodge J, Smith NA, Etzioni O. Green ai. Communications of the ACM. 2020;63(12):54-63.

[68] Goodwin P, Noland RB. Building new roads really does create extra traffic: a response to Prakash et al. Applied Economics. 2003;35(13):1451-7.