Unity: A General Platform for Intelligent Agents

Unity: A General Platform for Intelligent Agents

Arthur Juliani
Unity Technologies
arthurj@unity3d.com

Vincent-Pierre Berges
Unity Technologies
vincentpierre@unity3d.com

Esh Vckay
Unity Technologies
esh@unity3d.com

Yuan Gao
Unity Technologies
vincentg@unity3d.com

Hunter Henry
Unity Technologies
brandonh@unity3d.com

Marwan Mattar
Unity Technologies
marwan@unity3d.com

Danny Lange
Unity Technologies
dlange@unity3d.com

Abstract

Recent advances in Deep Reinforcement Learning and Robotics have been driven by the presence of increasingly realistic and complex simulation environments. Many of the existing platforms, however, provide either unrealistic visuals, inaccurate physics, low task complexity, or a limited capacity for interaction among artificial agents. Furthermore, many platforms lack the ability to flexibly configure the simulation, hence turning the simulation environment into a black-box from the perspective of the learning system. Here we describe a new open source toolkit for creating and interacting with simulation environments using the Unity platform: Unity ML-Agents Toolkit[1]. By taking advantage of Unity as a simulation platform, the toolkit enables the development of learning environments which are rich in sensory and physical complexity, provide compelling cognitive challenges, and support dynamic multi-agent interaction. We detail the platform design, communication protocol, set of example environments, and variety of training scenarios made possible via the toolkit.

1 Introduction

1.1 Background

In recent years, there have been significant advances in the state of Deep Reinforcement Learning research and algorithm design (Mnih et al., 2015; Schulman et al., 2017; Silver et al., 2018; Espeholt et al., 2018). Essential to this rapid development has been the presence of challenging, easy to use, and scalable simulation platforms, such as the Arcade Learning Environment (Bellemare et al., 2013), VizDoom (Kempka et al., 2016), Mujoco (Todorov et al., 2012), and others (Beattie et al., 2016; Johnson et al., 2016). The existence of the Arcade Learning Environment (ALE), for example, which contained a set of fixed environments, was essential for providing a means of benchmarking the control-from-pixels approach of the Deep Q-Network (Mnih et al., 2013). Similarly, other platforms have helped motivate research into more efficient and powerful algorithms (Oh et al., 2016; Andrychowicz et al., 2017). These simulation platforms serve not only to enable algorithmic improvements, but also as a starting point for training models which may subsequently be deployed in the real world. A prime example of this is the work being done to train autonomous robots within

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

a simulator, and transfer that model to a real-world robot (OpenAI 2018; Sadeghi et al., 2016). In these cases, the simulator provides a safe, controlled, and accelerated training environment.

As the state of the field becomes more developed, existing environments and the benchmarks based on them become less informative, the need for novel environments presents itself. For example, most environments in the Arcade Learning Environment have been solved well above human-level performance, making the continued use of the benchmark less valuable (Machado et al., 2017). The complementary point created by this state of algorithmic progress is that there exists a virtuous circle in which the development of novel environments can enable the development of novel algorithms, and vice versa. We can expect the research community to continue to provide high-quality algorithms, but it is unclear from where these researchers should expect high-quality environments to come from, since the creation of such environments is often time-intensive and requires specialized domain knowledge.

Many of the current research platforms described above are based on popular video games or game engines, such as Atari 2600, Quake III, Doom, and Minecraft. This is part of a much longer-term trend in which games have served as a platform for Artificial Intelligence (AI) research. This trend can be traced back to the earliest work in AI around playing games such as chess and checkers (Shannon, 1950; Samuel, 1959), or work in the 1990s on applying Reinforcement Learning to the game of Backgammon (Tesauro, 1995). The qualities found in many video games which make them engaging challenges for human players are also the same challenges which AI researchers are interested in themselves (Laird & VanLent, 2001). This insight has motivated a wide range of research into the intersection of video games and AI from the diverse perspectives of game playing, player modeling, as well as content generation (Yannakakis & Togelius, 2018).

All simulations or games are not created equal in their ability to meaningfully contribute to the field. The question naturally arises concerning which properties of an environment make it a worthwhile platform for research. When thinking of the kinds of environments that provide the complexity necessary for challenging current and to-be-developed algorithms, the physical world appears as a primary candidate. It is in the physical world where mammalian, and more specifically, human, intelligence developed, and it is this kind of intelligence which researchers are often interested in replicating (Lake et al., 2017).

When examining the areas of human intelligence typically studied, we find four major ones which have been the focus in recent years, and which we expect to continue to be the focus into the future: sensory, physical, cognitive, and social. These aspects of intelligence have traditionally been studied separately within the separate domains of Computer Vision (Krizhevsky et al., 2012), Optimal Control (Zhou et al., 1996), Planning (Browne et al., 2012), and Multi-Agent Systems (Ferber & Weiss, 1999). Recently these fields have begun to be studied holistically within the context of Deep Reinforcement Learning (Mnih et al., 2015). To challenge algorithms within these four domains, environments should be able to test all four axes of intelligence simultaneously.

In this paper, we begin by investigating the desired properties of a simulation platform, then presenting a review of existing platforms, along with their limitations at fully addressing the kinds of intelligences mentioned above. We then introduce the Unity platform as a simulator, discussing the extent to which it possesses the desired properties for enabling research. We next outline the functionality and tools provided by our open source toolkit for creating high-quality and flexible simulation environments using Unity, and finally provide a set of benchmark results on our example example learning environments. We describe the toolkit from the perspective of both researchers interested in benchmarking their algorithm, as well as researchers interested in developing novel environments for use in experimentation. For more extensive information on the toolkit, see our GitHub documentation page[2].

1.2 Anatomy of a Simulation

Here we describe in detail some of the requirements of a simulation we believe are needed to advance the state of the field. These requirements make up a taxonomy which can be used to examine the fidelity and capability of a variety of simulators. Simulators enable the ability to create and run simulations of environments. As such, we can example this taxonomy of properties both from the

[2] `https://github.com/Unity-Technologies/ml-agents/tree/master/docs`

perspective of the environments being simulated as well as from the perspective of the simulator providing the simulation of the environment.

1.2.1 Environment Properties

As learned models are able to perform increasingly complex tasks, the complexity of the simulated environments themselves should increase in order to continue to meaningfully challenge the algorithms and methods being explored. We identify the key dimensions of complexity we believe are essential for the continued development of AI systems: sensory, physical, cognitive, and social.

Sensory Complexity - The recent advances in Deep Learning have largely been driven by the ability of neural networks to process large amounts of visual, auditory, and text-based data (LeCun et al., 2015). ImageNet, a large database of natural images with associated labels, was essential in enabling models such as ResNet (He et al., 2016), and Inception (Ioffe & Szegedy, 2015) to be trained to near human-level performance (Russakovsky et al., 2015). While ImageNet was mainly used for static image recognition tasks, its key component of visual complexity is necessary for many real-world decision making problems, such as self-driving cars, household robots, and Unmanned Autonomous Vehicles (Zhu et al., 2017). Additionally, advances in image processing algorithms, specifically around Convolutional Neural Networks, were the motivation for the pixel-to-control approach eventually found in the Deep-Q network (Mnih et al., 2015).

Physical Complexity - Many of the applied tasks researchers are interested in solving using AI involve not only rich sensory information, but a rich control scheme in which agents can interact with their dynamic environments in complex ways (Bicchi & Kumar, 2000; Levine et al., 2016). The need for complex interaction often comes with the need for environments which replicate the physical properties of the target domain, typically the real world. This realism is essential to problems where the goal is to transfer a policy learned within a simulator to the real world, as would be the case for most robotics applications (Rusu et al., 2016; Tobin et al., 2017; OpenAI, 2018).

Cognitive Complexity - A third track of complexity can be thought of as cognitive, or combinatorial complexity. The game of Go for example, which has long served as a test-bed for AI research, contains neither complex visuals nor complex physical interactions. Rather, the complexity comes from the large search space of possibilities open to the agent at any given time (Muller, 2002; Silver et al., 2016). Meaningful simulation environments should enable designers to naturally create such problems for the learning agents within them. These complex tasks might display hierarchical structure, a hallmark of human intelligence (Botvinick, 2008), or vary from instance to instance, thus requiring meta-learning or generalization to solve (Wang et al., 2016). The tasks may also be presented in a sequential manner, where independent sampling from a fixed distribution is not possible. This is often the case for human task acquisition in the real world, and the ability to learn new tasks over time is seen as a key-component of Continual Learning (Ring, 1994), and ultimately systems capable of Artificial General Intelligence (Schmidhuber, 2015; Schmidhuber, 2018).

Social Complexity - The acquisition of complex skills via learning in mammals is believed to have evolved hand in hand with their ability to hold relationships within their social groups (Arbib et al., 2008). At least one strong example of this exists within the human species, with language primarily being the development of a tool for communication in a social setting. As such, the development of social behavior among groups of agents is of particular interest to many researches in the field of AI. There are also classes of complex behavior which can only be carried out at the population level, such as the coordination needed to build modern cities. Additionally, the ability for multiple species to interact with one another is a hallmark of the development of ecosystems in the world, and would be desirable to simulate as well. A simulation environment designed to allow the study of communication and social behavior should then provide a robust multi-agent framework which enables interaction between agents of both the same population as well as interaction between groups of agents drawn from separate distributions.

1.2.2 Simulation Properties

While the properties defined above are ideal for a modern AI research environment, there are additional practical constraints imposed by the simulator itself which must be taken into consideration when designing environments and experimental pipelines. Specifically, simulation environments must be flexibly controlled by the researcher, and they must run in a fast and distributed manner in order to provide the iteration time required for experimental research.

Fast & Distributed Simulation - Depending on the sample efficiency of the method used, modern machine learning algorithms often require up to tens of billions of pieces of data in order to converge to an optimal solution. As such, the ability to collect that data as quickly as possible is paramount (Espeholt et al., 2018). One of the most appealing properties of a simulation is the ability for it to be run at a speed often orders of magnitude greater than that of the physical world. In addition to this increase in speed, simulations can often be run in parallel, allowing for orders of magnitude greater data collection than a real-time serial experience in the physical world would. The faster such algorithms can be trained, the greater the speed of iteration and experimentation that can take place, leading to faster development of novel methods.

Flexible Control - A simulator must also allow the researcher or developer a flexible level of control over the configuration of the simulation itself, both during development and at runtime. While treating the simulation as a black-box has been sufficient for certain advances in recent years (Mnih et al., 2015), it also prevents the usage of a number of advanced research-level machine learning approaches such as Curriculum Learning, Domain Randomization, or adaptive task selection, in which more dynamic feedback between the training process and the agents is essential.

One such method is Curriculum Learning, which entails initially providing a simplified version of a task to an agent, and slowly increasing the task complexity as the agent's performance increases (Bengio et al., 2009). This method was used to achieve near human-level performance in a recent VizDoom competition (Wu & Tian, 2016). Such approaches are predicated on the assumption that the researcher has the capacity to alter the simulation to create such curricula in the first place.

Domain Randomization is another technique requiring flexible control of the simulator. The approach involves introducing enough variability into the simulation so that the models learned within the simulation can generalize to the real world. This often works by ensuring that the data distribution of the real world is covered within all of the variations presented within the simulation (Tobin et al., 2017). This variation is especially important if the agent depends on visual properties of the environment to perform its task. It is often the case that without domain randomization, models trained in simulation suffer from a "reality gap" and perform poorly. Concretely, performing Domain Randomization often involves dynamically manipulating textures, lighting, physics, and object placement within a scene.

2 Other Simulation Platforms

In recent years there have been a number of simulation platforms developed for the purposes of providing challenges and benchmarks for Deep Reinforcement Learning algorithms. Many of these platforms have been based on existing games or game engines, and carry with them specific strengths and weaknesses. While not exhaustive of all of the currently available platforms, below we survey a few of the more prominent ones, and the role they have played in the development of more advanced algorithms and models.

2.1 Arcade Learning Environment

The release of the Arcade Learning Environment (ALE) contributed to much of the recent resurgence of interest in Reinforcement Learning. This was thanks to the development of the Deep Q-Network, which was able to achieve superhuman level performance on dozens of emulated Atari console games within the ALE by learning only from pixel inputs (Mnih et al., 2015). The ALE provides a Python interface for launching and controlling simulations of any of dozens of Atari games. When considering the simulation criteria described above, the ALE provided visual input through pixel-based rendering, hierarchical problems within some of the games such as Montezuma's Revenge, and high-performance simulation, with an emulated game being able to run at thousands of frames per second (Bellemare et al., 2013). Its downsides included deterministic environments, relatively simple visuals, a lack of realistic physics, single-agent control, and a lack of flexible control of the simulation configuration. Furthermore, the majority of the environments provided in the ALE have been solved with greater than human performance, with the exception of a few sparse-reward tasks such as Montezuma's Revenge and Pitfall (Bellemare et al., 2017). As such, there is little room for novel algorithms to meaningfully outperform the current state of the art.

2.2 DeepMind Lab

Built from the Quake III game engine, DeepMind Lab (Lab) was released in 2016 as the external version of the research platform used by DeepMind (Beattie et al., 2016). Being designed in the wake of public adoption of the ALE, Lab contains a number of features designed to address the other platform's shortcomings. By using a 3D game-engine, complex navigation tasks similar to those studied in robotics and animal psychology could be studied within Lab (Leibo et al., 2018). The platform also contains primitive physics enabling a level of prediction about the quality of the world, allows researchers to define their own environmental variations, and allows for basic multi-agent interactions using language (Espeholt et al., 2018). The limitations of this platform, however, are largely tied to the dated nature of the underlying rendering and physics engine, which was built using decades-old technology. As such, the gap in quality between the physical world and the simulation provided via Lab is relatively large. Furthermore, the engine was also designed to enable first-person shooter games, and as such the environments built using Lab are limited to agents with a first-person perspective.

2.3 Project Malmo

Another popular simulation platform is Project Malmo (Malmo). Based on the exploration and building game Minecraft, the platform provides a large amount of flexibility in defining scenarios and environment types (Johnson et al., 2016). As a result, there has been a number of research projects exploring multi-agent communication, hierarchical control, and planning using the platform (Oh et al., 2016; Shu & Socher, 2017; Tessler et al., 2017). The limitations of the platform, however, are bound tightly with the underlying limitations of the Minecraft engine itself. Due to the low-polygon pixelated visuals, as well as the rudimentary physics system, Minecraft lacks both the visual as well as the physical complexity that would be desirable from a modern platform.

2.4 Mujoco

The Mujoco physics engine has become a popular simulation environment for benchmarking model-free continuous control tasks, thanks to the a set of standard tasks built on top of Mujoco being provided with OpenAI Gym and the DeepMind Control Suite (Todorov et al., 2012; Brockman et al., 2016; Tassa et al., 2018). The high quality physics simulation combined with a number of standardized benchmarks has led to the platform being the primary choice for researchers interested in examining the performance of continuous control algorithms. The nature of the Mujoco engine, however, poses limitations for more general artificial intelligence research. The first is around the limited visual rendering capabilities of the engine, preventing the use of complex lighting, textures, and the use of shaders. The second are the restrictions of the physics engine itself which make more difficult the creation of dynamic "game-like" environments where many different objects would be instantiated and destroyed in real-time during the simulation. More dynamic environments are often necessary to pose tasks which require greater planning or coordination to solve.

2.5 VizDoom

Based on the game Doom, VizDoom provides researchers with the ability to create tasks which involve first-person navigation and control (Kempka et al., 2016). Through a 2017 AI deathmatch competition, the platform enabled the development of a number of compelling approaches to Deep Reinforcement Learning, including utilizing learning curricula (Wu & Tian, 2016), novel algorithm design (Dosovitskiy & Koltun, 2016), and memory systems (Lample & Chaplot, 2016). Like DeepMind Lab, the platform is mainly restricted by the underlying game engine, which was built for the purpose of a decades old first-person shooter game. As such the visual and physical complexity possible in the environments created using VizDoom are relatively limited. It also is restricted to simulating artificial agents with only a first-person perspective.

3 Unity Platform

Unity is a game development platform that consists of a game engine and graphical user interface called the Unity Editor. Unity was originally created in 2005 to enable developers to make video games and to help make the video game GooBall. Since then, Unity has grown and is now used by a large community of game developers to make a variety of interactive simulations, ranging from small mobile and browser-based games to high-budget console games and AR/VR experiences. This historical focus on developing a general-purpose engine to support a variety of platforms, developer experience levels, and game types make the Unity engine an ideal simulation platform.

The flexibility of the underlying engine makes everything from simple gridworld problems to complex strategy games, physics-based puzzles, or multi-agent competitive games possible. Unlike many of the research platforms defined above, the underlying engine is not restricted to any specific genre of gameplay or simulation. Furthermore, the Unity Editor enables rapid prototyping and development of games and simulation environments.

3.1 Unity Terminology

A Unity Project consists of a collection of Assets. These typically correspond to files within the Project. Scenes are a special type of Asset which define the environment or level of a Project. Scenes contain a definition of a hierarchical composition of GameObjects, which correspond to the actual objects, either physical or purely logical, within the environment. The behavior and function of each GameObject is determined by the components attached to it. There are a variety of built-in components provided with the Unity Editor, including Cameras, Meshes, Renderers, RigidBodies, and many others. It is also possible to define custom components using C# scripts or external plugins.

3.2 Engine Properties

This section examines the properties of the Unity engine from the perspective of a simulator and set of environments, as described in Section 1.2, and compares it to other simulators currently used in the field.

3.2.1 Environment Properties

Sensory Complexity - The Unity engine enables high-fidelity graphical rendering. It supports pre-baked as well as real-time lighting and the ability to define custom shaders, either programmatically or via a visual scripting language. As such, it is possible to quickly render near-photorealistic imagery to be used as training data for a Machine Learning model. It is also possible to render depth information, object masks, infrared, or images with noise injected into it through the use of custom shaders. Furthermore, the engine provides a means of defining audio signals which can serve as potential additional observational information to learning agents, as well as ray-cast based detection systems, which can simulate Lidar.

Physical Complexity - The Unity engine contains built-in physics provided by Nvidia PhysX. This enables interactions between RigidBodies of various sizes and shapes, enabling research on locomotion and physical interactions. Furthermore, the extensible nature of the platform enables the use of additional 3rd party physics engines if desired. For example, there are plugins available for Unity which provide both the Bullet and Mujoco physics engines (Deane, 2018; Todorov, 2018) as alternatives to PhysX.

Cognitive Complexity - The Unity Engine provides a rich and flexible scripting system via C#. This system enables any form of gameplay or simulation to be defined and dynamically controlled via the scripting system. In addition to the scripting language is the GameObject and component system, which enables managing multiple instances of agents, policies and environments, making it possible to define complex hierarchical tasks, or tasks which would require meta-learning to solve.

Social Complexity - In addition to enabling complex task definitions, the nature of the Unity scripting language and component system make the posing of multi-agent scenarios simple and straightforward. Indeed, because the platform was designed to support the development of multi-player video games, a number of useful abstractions are already provided out of the box, such as the Multiplayer Networking system[3].

3.2.2 Simulation Properties

Fast & Distributed Simulation - The physics and frame rendering of the Unity engine take place in a de-synchronized fashion. As such, it is possible to greatly increase the speed of the underlying simulation without the need to increase the frame rate of the rendering process. It is also possible to run Unity simulations without rendering, when it is not necessary to the simulation. In scenarios where rendering is desirable, such as learning from pixels, it is possible to control the frame rate and speed of game logic. Extensive control of the rendering quality also makes it possible to greatly increase the frame rate when desired.

The added capabilities of the Unity Engine does add additional overhead when attempting to simulate in a large-scale distributed fashion. It is also currently not possible to natively perform rendering in the absence of a virtual screen environment such as X-Server or xvfb, although this capability will be available in the future. Also, the memory footprint of a Unity simulation is larger than that of an Atari game in the ALE, for example.

Flexible Control - It is possible to control most aspects of the simulation programmatically, enabling researchers to define curricula, adversarial scenarios, or other complex methods of changing the learning environment during the training process over time. In Section 4.2, we discuss ways in which further control of the simulation is made possible via exposed simulation parameters and a Python API.

3.3 Unity Editor and Services

The Unity Editor is a graphical user interface used to create the content for 2D, 3D and AR / VR experiences. It is available on Windows, Mac and Linux. (See Figure 1 for a screenshot of the Unity Editor window.) The Unity Editor and its services provide additional benefits for AI research:

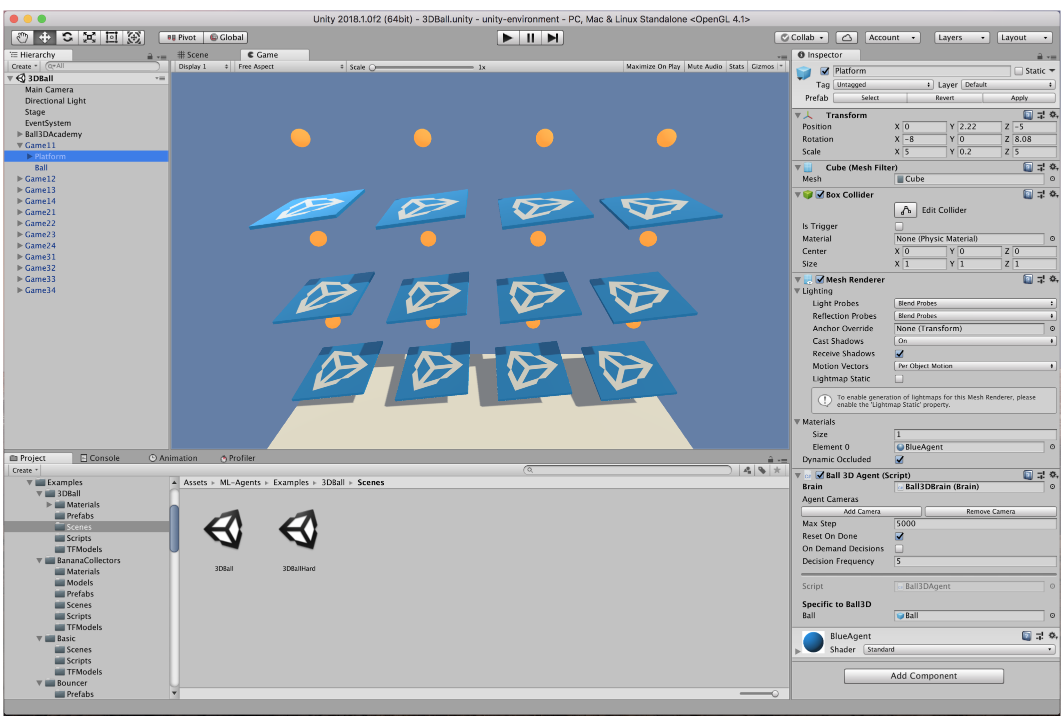

Figure 1: The Unity Editor window on macOS.

[3] https://unity3d.com/learn/tutorials/s/multiplayer-networking

1. *Create custom Scenes* - Unity provides a large number of guides and tutorials on how to create Scenes within the Editor. This enables developers to quickly experiment with new environments of varying complexities, or novel tasks. Furthermore, the availability of the Asset Store which contains 50,000 free and paid assets has the potential to decrease the barrier to entry for the creation of new environments.
2. *Record local, expert demonstrations* - The Unity Editor includes a Play mode which enables a developer to begin a simulation and control one or more of the agents in the Scene via a keyboard or game controller. This can be used for generating expert demonstrations to train and evaluate imitation learning algorithms. To facilitate generating expert demonstrations from multiple experts, each with their own Unity Editor, a project can be easily shared and versioned using Unity Teams. Unity Teams is a Git-like versioning system built into the Unity Editor that is free of charge for small teams.
3. *Record large-scale demonstrations* - One the most powerful features of the Unity Editor is the ability to build a Scene to run on more than 20 platforms ranging from wearables to mobile and consoles. This enables developers to distribute their Scenes to a large number of devices (either privately or publicly through stores such as the Apple App Store or Google Play). This can facilitate recording expert demonstrations from a large number of experts or measuring human-level performance from a user (or player) population.

3.4 Previous Usage

One platform that has already validated the use of the Unity engine as a simulation is the AI2Thor simulator (Kolve et al., 2018). The platform provides a set of pre-built indoor Scenes which are rendered using the Unity engine, and a Python API for interacting with those environments using a first-person agent. Research using AI2Thor simulator has validated the viability of the Unity engine as a simulator by demonstrating that it is possible to transfer a policy learned within the simulation to a physical robot in the real world to complete an indoor-navigation task (Zhu et al., 2017). A similar platform has also been developed by Cornell using Unity to provide an additional set of indoor navigation environments (Yan et al., 2018). Recent work at OpenAI has also taken advantage of the rendering capabilities of the Unity engine to aid in the development of a system used to transfer a robotic hand's grasping policy from a simulator to a physical robot (OpenAI, 2018). Below we present the ML-Agents Toolkit, a more generalized set of tools for creating and interacting with simulation environments built with Unity.

4 Ml-Agents Toolkit

The ML-Agents toolkit is an open source project which enables researchers and developers to create simulation environments using the Unity Editor and interact with them using a Python API. The toolkit is built to take full advantage of the properties of the Unity Engine described above which make it a potentially strong research platform.

The toolkit provides a set of core and additional functionalities. The core functionality enables developers and researchers to define environments with the Unity Editor and associated C# scripts, and then expose these built environments to a straightforward Python API for interaction. Additional functionality includes a set of example environments and baseline algorithms. The example environments can be used either as a means of benchmarking Reinforcement Learning algorithms, or as templates upon which to build or modify novel environments and control challenges for ML systems. The baseline algorithms can be used to train agents within any environment created using the core platform, and serve as a starting point for those interested in developing more advanced algorithms.

4.1 Core Functionality

The ML-Agents Core Toolkit provides everything necessary to create and interact with simulation environments built using the Unity Editor. It is composed of two pieces: an ML-Agents SDK, which contains all functionality necessary to define environments within the Unity Editor and associated C# scripts, and a Python package which enables interfacing with environments built using the SDK.

4.1.1 ML-Agents SDK

Once the ML-Agents SDK is imported into the Unity project, Scenes within Unity can be made into learning environments. This happens through the use of three entities available in the SDK: Agent, Brain, and Academy.

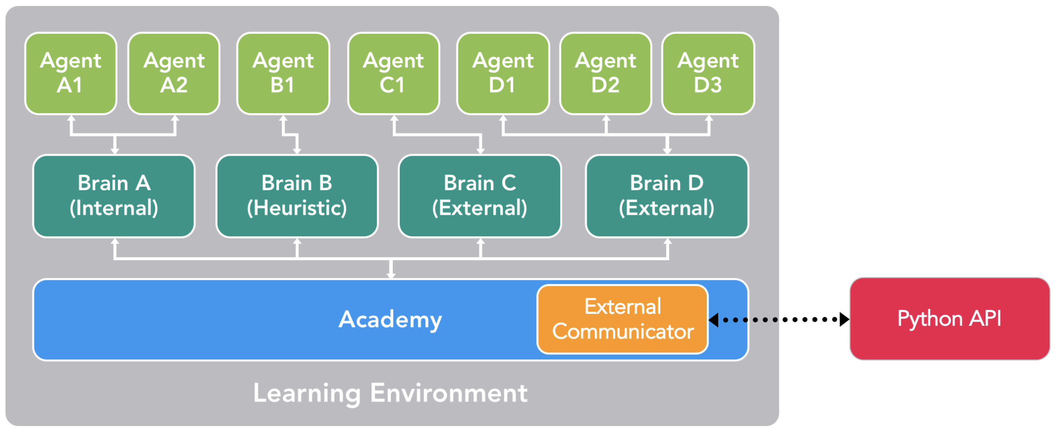

Figure 2: A Learning Environment created using the Unity Editor contains Agents, Brains, and an Academy. The Agents are responsible for collecting observations and taking actions. Each Brain is responsible for providing a policy for decision making for each of its associated agents. The Academy is responsible for global coordination of the environment simulation.

Scene Components

The Agent component is used to directly indicate that a GameObject within a scene is an Agent. This component is responsible for collecting observations from the environment, and for executing actions within that environment as well. Each Agent component is linked to a single Brain at a time. These linked Brains are responsible for making decisions for all of their linked Agents. There can be any number of GameObjects with Agent components attached, making the creation of multi-agent environments as simple as duplicating GameObjects within a scene.

Brain components make decisions for all linked Agents. From the perspective of the Agent, it is the Brain which provides the policy that the Agent follows. Multiple Agents can be connected to a single Brain, and the Brain will make decisions for all of them. Brains contain a definition of the observation and action spaces within which the policy of the Brain is valid. Only Agents which contain matching observation and action space configurations can be linked to a Brain. A Brain's decision making can be conducted in various ways, including through player input, predefined scripts, internally embedded neural network models, or via interaction through the Python API. These means of providing a policy to the agents are referred to as Player, Heuristic, Internal, and External, respectively. Adding multiple Brains to a single scene enables scenarios in which multiple policies can be learned simultaneously.

The Academy component within a scene is used to keep track of the steps of the simulation, and to provide functionality such as the ability to reset the environment, set the target simulation speed and framerate. The Academy also contains the ability to define reset parameters, which can be used to change the configuration of the environment during runtime.

As a motivating example, imagine defining an environment modeled after the African savanna. We can start simple by creating a single agent within this environment, and a single brain to accompany it. In this case, our agent will be a Zebra, and we can define its task as collecting shrubs within the environment. This means creating a Zebra GameObject, and attaching a Zebra Agent component to it. We would then create a Zebra Brain, and link the two. If we set the brain to Player mode, we will be able to control the actions of the Zebra ourselves. If we set it to External mode, we will be able to control it using the Python API (described below in Section 4.2.2).

We may, however, want to model a more complex environment, one which includes multiple different species, such as Zebras, Elephants, and Lions. These three animal types would each correspond to unique Agent components. These components can be attached to multiple different GameObjects, making it possible for there to be an arbitrary number of each animal within the environment. Brains can then be associated with different Agents to provide different policies. For example, there may be a single Zebra which is player controlled, which would utilize a Player Brain. There may be aggressive and peaceful elephants, and in this case there would be two Elephant Brains, each containing a different policy. See Figure 2 for a diagram describing the way in which each of these components work together.

Defining a Learning Environment

Once the GameObjects and components are in place within the scene, it is then possible to define the learning environment and create a Markov Decision Process (MDP) or Partially Observable MDP (POMDP) which will serve as the basis of a variety of possible Reinforcement Learning tasks (Sutton & Barto, 1998). This is done through the definition of the observations, actions, and a reward function. All of these can be defined from within the Agent component itself. Furthermore, any game logic which has access to the Agent component can also manipulate the agent's reward function as well as define whether the agent has reached a terminal state.

Observations provide the agent with a context about the state of the environment. They can take the form of both a vector of floating point numbers, as well as any number of rendered camera outputs from within the scene which the agent contains a reference to. As such, it is possible to define observations which can correspond to any numerical information made available to the Agent component. More concretely, information such as the results of ray-casts, auditory signals, Agent position, and more, can be provided as observations. Actions can take the form of either discrete (array of integers integer) or continuous variables (array of floating point numbers).

It is possible for agents to ask for decisions from Brains either at a fixed or dynamic interval, as defined by the developer of the environment. The reward function, used to provide a learning signal to the agent, can be defined or modified at any time during the simulation using the Unity scripting system. Likewise, simulation can be placed into a done state either at the level of an individual agent, or the environment as a whole. This happens by either calling via a Unity script call, or by reaching a predefined max step count. It is also possible to define a set of reset environment parameters which can be accessed and manipulated from the Python API upon resetting the environment.

4.1.2 Python Package

The provided Python package[4] contains a class called UnityEnvironment that can be used to launch and interface with Unity executables (as well as the Editor) which contain the required components described above. Communication between Python and Unity takes place via a gRPC communication protocol, and utilizes protobuf messages. Interacting with the Python API works as follows:

- $env = UnityEnvironment(filename)$ – Launches a learning environment given a file or path name, and establishes communication with python API.
- $env.reset()$ – Resets the environment and returns a dictionary of $BrainInfo$ objects containing initial observations of all agents.
- $env.step(actions)$ – Steps the learning environment by providing a set of actions for all agents present in the previously received BrainInfo dictionary and returns a dictionary of $BrainInfo$ objects containing new observations from the agents that require decisions.
- $env.close()$ – Sends termination signal to learning environment.

$BrainInfo$ is a class which contains lists of observations, previous actions, rewards, and miscellaneous state variables. At each simulation step, $BrainInfo$ contains information for all active agents within the scene which require decisions in order for the simulation to proceed.

We also provide a set of wrapper APIs, which can be used to communicate with and control Unity Learning Environments through the standard gym interface used by many researchers and algorithm developers (Brockman et al., 2016). These gym wrappers are designed to enable researchers to more easily swap in Unity Learning Environments to an existing Reinforcement Learning system already designed around the gym interface.

[4]`https://pypi.org/project/mlagents/`

4.1.3 Performance Metrics

It is essential that an environment be able to provide greater than real-time simulation speed. It is possible to increase Unity ML-Agents simulations up to one hundred times real-time. The possible speed increase in practice, however, will vary based on the computational resources available, as well as the complexity of the environment. In the Unity Engine, game logic, including physics, can be run independently from the rendering of frames. As such, environments which do not rely on visual observations, such as those that use ray-casts for example, can benefit from simulation at speeds greater than those that do. See Table 1 for performance metrics when controlling environments from the Python API.

Environment	Observation Type	Number Agents	Mean (ms)	Std (ms)
Basic	Vector(1)	1	0.803	0.005
3D Ball	Vector(8)	12	5.05	0.039
GridWorld	Visual(84x84x3)	1	2.04	0.038
Visual Banana Collector	Visual(84x84x3)	4	9.23	0.556

Table 1: Performance benchmark when using the Python API to control a Learning Environment from the same machine by calling $env.step(action)$. Mean and standard deviation in time averaged over 1000 simulation steps. See Section 4.3.1 below for descriptions of example environments used.

4.2 Additional Functionality

In addition to the Unity SDK and Python interface, the ML-Agents Toolkit also includes a set of example environments, and compatible reinforcement learning training algorithms to be used with the platform. We outline these below.

4.2.1 Example Environments

The Unity ML-Agents GitHub repository contains a number of example environments in addition to the core functionality. These environments are designed to both be usable for benchmarking RL algorithms as well as templates to develop novel environments and tasks. These environments contain examples of single and multi-agent scenarios, with agents using either vector or visual observations, taking either discrete or continuous actions, and receiving either dense or sparse rewards. See Figure 3.a-3.n. for images of the included example environments at the time of writing, and read below for descriptions of the corresponding environments.

(a) *Basic* - A linear movement task where the agent (blue cube) must move left or right to rewarding states. The goal is to move to the most rewarding state.

(b) *3D Ball* - A balance-ball task where the agent controls the rotation of the platform. The goal is to balance the platform in order to keep the ball on it for as long as possible.

(c) *Push Block* - A platforming environment where the agent can push a block around. The goal is to push the block to the target area (black white grid).

(d) *Tennis* - Two-player game where agents control rackets to bounce ball over a net. The goal is to bounce ball to the other side instead of dropping the ball or sending ball out of bounds.

(e) *Bouncer* - A bouncing task where the agent (blue cube) can jump with a certain speed and angle when it touches the ground. The goal is to catch the floating banana with as few jumps as possible.

(f) *Grid World* - A version of the classic grid-world task. Scene contains agent (blue square), target, and obstacles. The goal is to navigate to the target while avoiding the obstacles.

(g) *Reacher* - Double-jointed arm which can move to target locations. The goal is to move its hand to the target location (green sphere), and keep it there.

(h) *Banana Collector* - A multi-agent environment where agents (blue cube) compete to collect bananas. The goal is to move to as many yellow bananas as possible while avoiding blue bananas.

(i) *Wall Jump* - A platforming environment with a wall and a yellow block that can be pushed around, and an agent (blue cube) that can move, rotate and jump. The goal is to reach the target (white

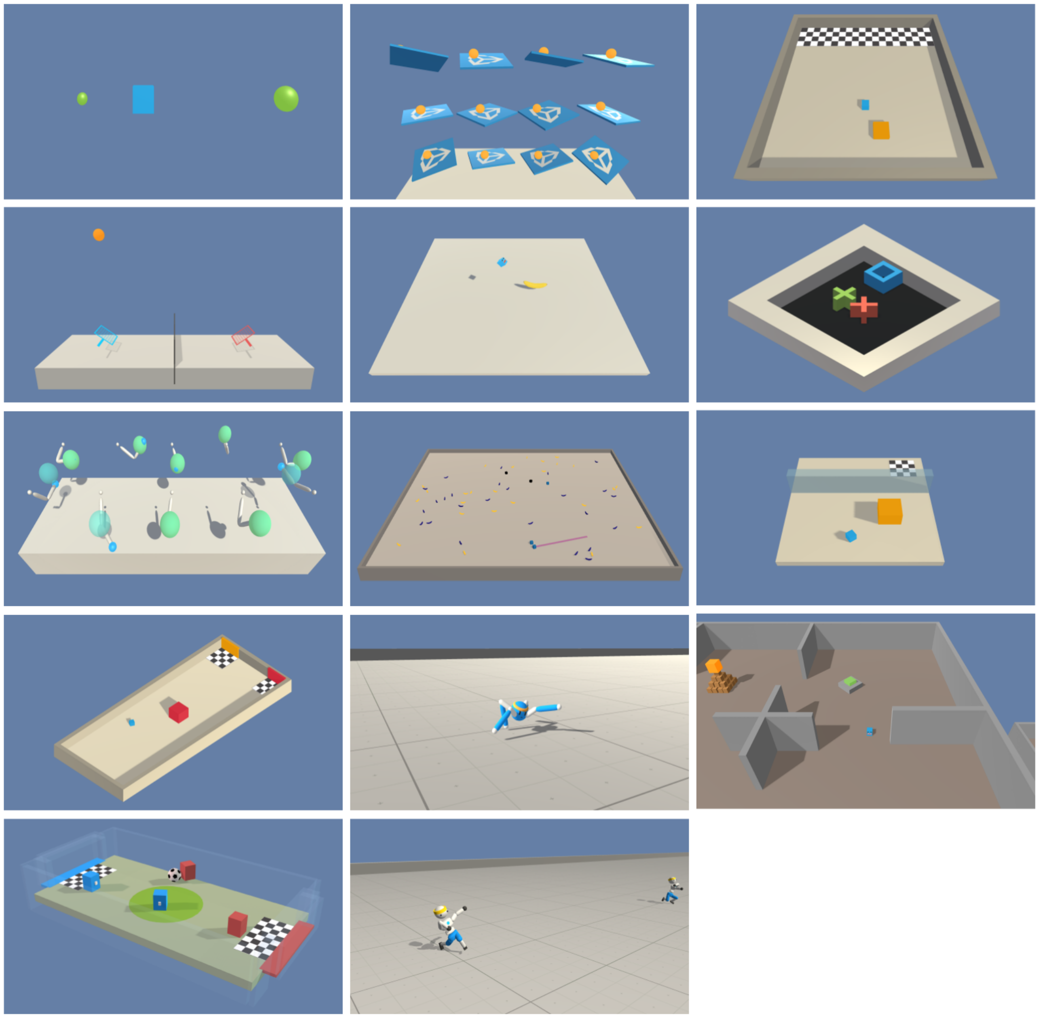

Figure 3: Images of the fourteen included example environments as of the v0.4 release of the Unity ML-Agents Toolkit. From Left-to-right, up-to-down: (a) Basic, (b) 3DBall, (c) Push Block, (d) Tennis, (e) Bouncer, (f) Grid World, (g) Reacher, (h) Banana Collector, (i) Wall Jump, (j) Hallway, (k) Crawler, (l) Pyramids, (m) Walker, (n) Soccer Twos.

black grid) on the other side of the wall. The agent sometimes needs to push the yellow block near the wall, jump onto it to reach its target.

(j) *Hallway* - Environment where the agent (blue cube) needs to find information in a room, remember it, and use it to move to the correct target. The goal is to move to the target (black white grid) which corresponds to the color of the block in the room.

(k) *Crawler* - Physics-based creatures with 4 arms and 4 forearms. The goal is to move toward the goal direction as quickly as possible without falling.

(l) *Pyramids* - Environment where the agent (blue cube) needs to press a button to spawn a pyramid, then navigate to the pyramid, knock it over, and move to the gold brick at the top. The goal is to move to the golden brick on top of the spawned pyramid.

(m) *Walker* - Physics-based humanoids with 26 degrees of freedom of its body-parts. The goal is to move toward the goal direction as quickly as possible without falling.

(n) *Soccer Twos* - Environment where four agents compete in a 2 vs 2 toy soccer game. There are two kinds of agents in the environment. The goal of the Striker agent is to push the ball into the opponent's goal area, the goal of the Goalie is to prevent the ball from entering its own goal area.

For more information on the specifics of each of the environments, including the observations, actions, and reward functions, see our GitHub documentation[5].

4.2.2 Baseline Algorithms

We provide a set of baseline algorithms which can be used to validate new environments as well as starting points for the development of novel algorithms. We currently provide an implementation of Proximal Policy Optimization (PPO) (Schulman et al., 2017), a state of the art Deep Reinforcement Learning algorithm, with the option to extend it using an Intrinsic Curiosity Module (ICM) (Pathak et al., 2017), and a Long-Short-Term Cell (LSTM) (Hochreiter & Schmidhuber, 1997). We also provide an implementation of Behavioral Cloning, a simple Imitation Learning algorithm (Hussien et al., 2017) to showcase the Unity Editor as a tool for recording expert demonstrations.

We provide trained model files as well as hyperparameter specifications for replicating all of our results on the example environments provided with the toolkit. As the platform grows, we intend to provide additional algorithms and model types. See Figure 4 below for baseline results on each of our provided example environments. These results describe the mean cumulative reward per-episode over five runs using PPO (plus relevant modifications), human performance over ten minutes of control, and a random baseline over a single learning session. While our baseline algorithm is able to consistently solve most environments, it does poorly on a few environments, such as VisualPyramids, VisualPushBlock, VisualHallway, and SoccerTwos. We encourage researchers to develop solutions which achieve greater means reward, or learn more quickly than the results provided here.

Environment	PPO (mean)	PPO (std)	Random	Human (mean)	Human (std)
3DBall	100.00	0.00	1.24	2.06	0.38
3DBallHard	85.42	8.41	1.17	3.53	2.24
Banana	4.27	1.33	0.20	N/A	N/A
Basic	0.94	0.00	0.10	0.94	0
Bouncer	13.63	0.21	-0.41	2.81	1.23
CrawlerDynamic	371.91	44.27	2.21	N/A	N/A
CrawlerStatic	2655.18	68.13	4.34	N/A	N/A
GridWorld	0.96	0.03	-0.10	0.45	0.14
Hallway	0.93	0.00	-0.99	N/A	N/A
PushBlock	4.98	0.00	-1.00	N/A	N/A
Pyramids	1.73	0.04	-1.00	N/A	N/A
Reacher	38.84	0.69	0.27	N/A	N/A
SoccerTwos-Goalie	-0.09	0.11	0.42	N/A	N/A
SoccerTwos-Striker	0.09	0.11	-0.42	N/A	N/A
Tennis	0.91	0.51	0.01	N/A	N/A
VisualBanana	8.05	2.99	0.44	10.16	6.36
VisualHallway	0.35	0.02	-1.00	0.93	0.01
VisualPushBlock	0.56	1.42	-1.00	4.94	0.02
VisualPyramids	-0.62	0.48	-1.00	1.67	0.31
Walker	2652.05	149.73	-5.15	N/A	N/A
WallJump-BigWall	0.90	0.01	-1.14	0.36	0.33
WallJump-SmallWall	0.95	0	-1.12	0.36	0.33

Table 2: Table of cumulative episodic reward for the various example environments provided with the ML-Agents Toolkit. PPO results are averaged over final score on five separate runs. Human results are averaged over three individual runs, each lasting five minutes of play-time.

[5] `https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Examples.md`

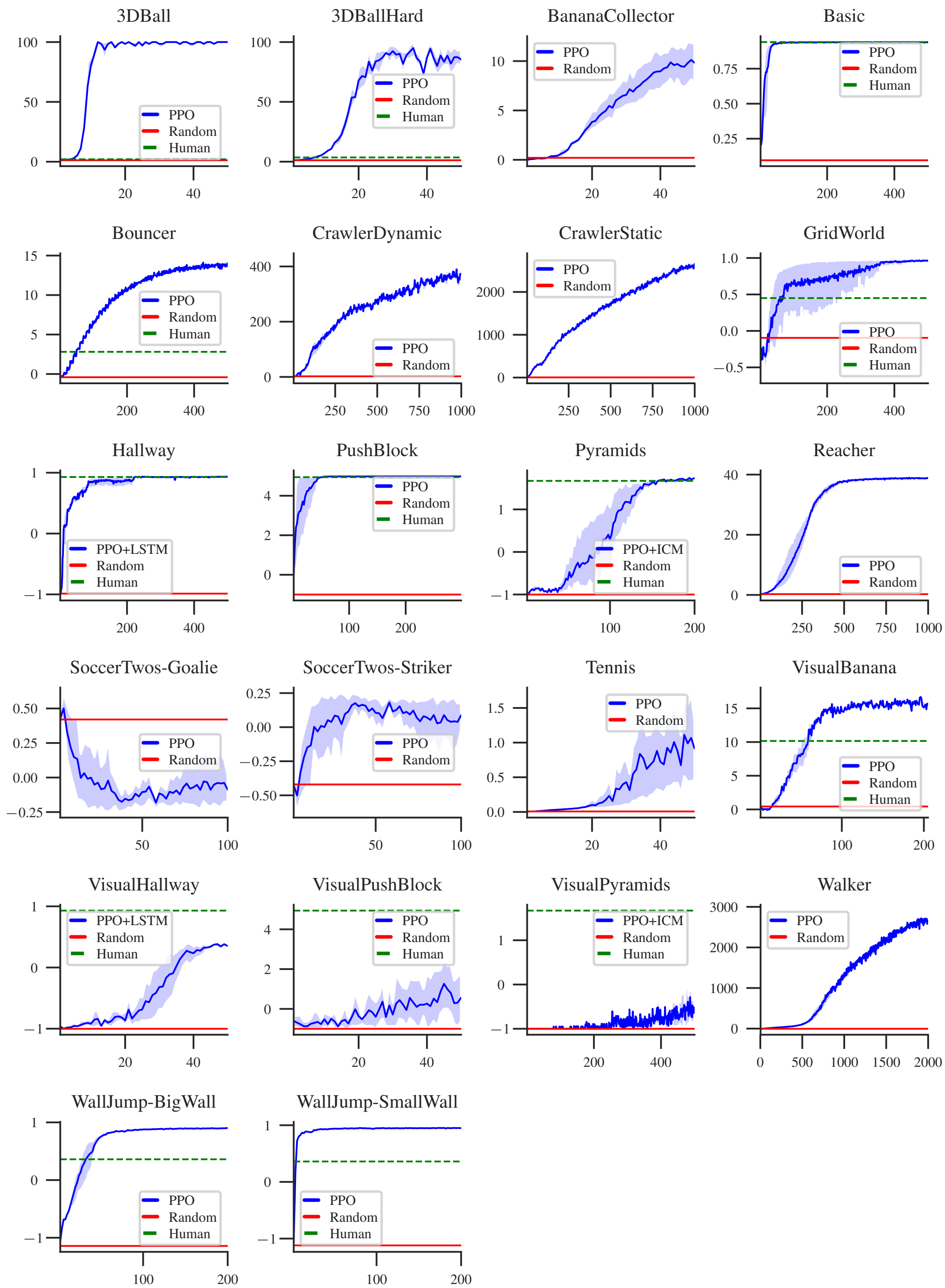

Figure 4: Graphs depicting mean cumulative episodic reward (y-axis) over time-steps of simulation (in thousands, x-axis) during training and evaluation. We compare Proximal Policy Optimization (blue line), a fixed random policy (red line) and human (green dashed line) performance. PPO results presented based on five separate runs, with a 95% confidence interval. PPO+LSTM indicates LSTM unit used in network. PPO+ICM indicates Intrinsic Curiosity Module used to generate additional intrinsic reward. Human results are averaged over three individual runs, each lasting five minutes of play-time.

4.3 Future Direction

Going forward, it is our intention to continue to improve the platform both as a tool for research, and also as a tool for game development. This includes providing more flexible means of defining observations and actions of a variety of types, providing increasingly complex environments to serve as benchmarks, and improving the overall performance of the system to provide a highly optimized means of simulating large numbers of agents in parallel. We are also interested in expanding the framework beyond Reinforcement Learning use-cases, eventually supporting a broader set of Machine Learning tasks including Computer Vision and Generative Modeling tasks.

5 Conclusion

The existing simulation platforms used for AI research are insufficient for enabling the development of truly human-like artificial intelligence. In this paper we introduced Unity as a simulation platform for such research that can provide a greater level of sensory, physical, cognitive, and social complexity than other platforms, moving closer to the above mentioned ideal. We believe this platform and toolkit can help to enable continued algorithmic development and progression in the field. The ML-Agents toolkit has already been used by a number of research groups, including those studying Reinforcement Learning (Jain & Lindsey, 2018; Burda et al., 2018), and Neural Attention Mechanisms (Ghani et al., 2018). While the current toolkit provides a basic set of functionalities, we believe the Unity Engine has the right sets of properties to serve as a platform for the development of increasingly complex and realistic learning environments for many years to come. As the project is open-source, we welcome those in the community of researchers and developers who are interested to join us in improving the platform going forward.

6 Acknowledgements

We would like to thank Jeff Shih, Deric Pang, and Sachin Dharashivkar for their contributions to the benchmarks presented in this paper, Praneet Dutta, Christopher Lu, and Cesar Romero for their feedback during the initial toolkit design process, Trevor Santarra, Vladimir Oster, Samuel Warren, Adam Crespi, YouCyuan Jhang, Joe Ward, Catherine Morrison, and Jose De Oliveira for their feedback on a draft of this paper, and Kai Arulkumaran for his assistance in making the publication of this paper possible.

References

Andrychowicz, M., et al. (2017). Hindsight experience replay. In *Advances in Neural Information Processing Systems*(pp. 5048-5058).

Arbib, M. A., et al. (2008). Primate vocalization, gesture, and the evolution of human language. *Current anthropology*, 49(6), 1053-1076.

Beattie, C., et al. (2016). DeepMind lab. *arXiv preprint arXiv:1612.03801*.

Bellemare, M. G., Naddaf, Y., Veness, J., & Bowling, M. (2013). The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47, 253-279.

Bellemare, M. G., Dabney, W., & Munos, R. (2017). A distributional perspective on reinforcement learning. *arXiv preprint arXiv:1707.06887*.

Bengio, Y., Louradour, J., Collobert, R., & Weston, J. (2009). Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning* (pp. 41-48). ACM.

Bicchi, A., & Kumar, V. (2000, April). Robotic grasping and contact: A review. In *ICRA* (Vol. 348, p. 353).

Botvinick, M. M. (2008). Hierarchical models of behavior and prefrontal function. *Trends in cognitive sciences*, 12(5), 201-208.

Brockman, G., et al. (2016). Openai gym. *arXiv preprint arXiv:1606.01540*.

Browne, C. B., et al. (2012). A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1), 1-43.

Burda, Y., et al. (2018). Large-Scale Study of Curiosity-Driven Learning. In *arXiv:1808.04355 2018*.

Deane, I. (2018). Bullet Physics For Unity. *https://assetstore.unity.com/packages/tools/physics/bullet-physics-for-unity-62991*

Dosovitskiy, A., & Koltun, V. (2016). Learning to act by predicting the future. *arXiv preprint arXiv:1611.01779*.

Espeholt, L., et al. (2018). IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures. *arXiv preprint arXiv:1802.01561*.

Ghani, A. R. A., et al. (2018). Designing Efficient Neural Attention Systems Towards Achieving Human-level Sharp Vision.

Ferber, J., & Weiss, G. (1999). Multi-agent systems: an introduction to distributed artificial intelligence (Vol. 1). Reading: Addison-Wesley.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 770-778).

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8), 1735-1780.

Hussein, A., Gaber, M. M., Elyan, E., & Jayne, C. (2017). Imitation learning: A survey of learning methods. *ACM Computing Surveys* (CSUR), 50(2), 21.

Kempka, M., et al. (2016). Vizdoom: A doom-based ai research platform for visual reinforcement learning. In *Computational Intelligence and Games (CIG), 2016 IEEE Conference on (pp. 1-8). IEEE*.

Kolve, E., et al. (2017). AI2-THOR: An Interactive 3D Environment for Visual AI. *arXiv preprint arXiv:1712.05474*.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (pp. 1097-1105).

Laird, J., & VanLent, M. (2001). Human-level AI's killer application: Interactive computer games. *AI magazine*, 22(2), 15.

Lake, B. M., Ullman, T. D., Tenenbaum, J. B., & Gershman, S. J. (2017). Building machines that learn and think like people. *Behavioral and Brain Sciences*, 40.

Lample, G., & Chaplot, D. S. (2017). Playing FPS Games with Deep Reinforcement Learning. In *AAAI* (pp. 2140-2146).

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep learning. *Nature*, 521(7553), 436.

Leibo, J. Z., et al. (2018). Psychlab: a psychology laboratory for deep reinforcement learning agents. *arXiv preprint arXiv:1801.08116*.

Jain, M. S., & Lindsey, J. (2018). Semiparametric reinforcement learning.

Johnson, M., Hofmann, K., Hutton, T., & Bignell, D. (2016). The Malmo Platform for Artificial Intelligence Experimentation. In *IJCAI* (pp. 4246-4247).

Levine, S., Finn, C., Darrell, T., & Abbeel, P. (2016). End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1), 1334-1373.

Machado, M. C., et al. (2017). Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents. *arXiv preprint arXiv:1709.06009*.

Mnih, V., et al. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mnih, V., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540), 529.

Müller, M. (2002). Computer go. *Artificial Intelligence*, 134(1-2), 145-179.

Oh, J., Chockalingam, V., Singh, S., & Lee, H. (2016). Control of memory, active perception, and action in minecraft. *arXiv preprint arXiv:1605.09128*.

OpenAI. (2018). Learning Dexterous In-Hand Manipulation. *https://d4mucfpksywv.cloudfront.net/research-covers/learning-dexterity/learning-dexterity-paper.pdf*.

Pathak, D., Agrawal, P., Efros, A. A., & Darrell, T. (2017). Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning (ICML)* (Vol. 2017).

Russakovsky, O., et al. (2015). Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3), 211-252.

Rusu, A. A., et al. (2016). Sim-to-real robot learning from pixels with progressive nets. *arXiv preprint arXiv:1610.04286*.

Sadeghi, F., & Levine, S. (2016). Cad2rl: Real single-image flight without a single real image. *arXiv preprint arXiv:1611.04201*.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3), 210-229.

Schmidhuber, J. (2015). On learning to think: Algorithmic information theory for novel combinations of reinforcement learning controllers and recurrent neural world models. *arXiv preprint arXiv:1511.09249*.

Schmidhuber, J. (2018). One Big Net For Everything. *arXiv preprint arXiv:1802.08864*.

Shannon, C. E. (1950). XXII. Programming a computer for playing chess. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 41(314), 256-275.

Shu, T., Xiong, C., & Socher, R. (2017). Hierarchical and Interpretable Skill Acquisition in Multi-task Reinforcement Learning. *arXiv preprint arXiv:1712.07294*.

Schulman, J., et al. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Silver, D., et al. (2017). Mastering the game of go without human knowledge. *Nature*, 550(7676), 354.

Sutton, R. S., & Barto, A. G. (1998). *Reinforcement learning: An introduction*. MIT press.

Tassa, Y., et al. (2018). DeepMind Control Suite. *arXiv preprint arXiv:1801.00690*.

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3), 58-68.

Tessler, C., et al. (2017, February). A Deep Hierarchical Approach to Lifelong Learning in Minecraft. In *AAAI* (Vol. 3, p. 6).

Tobin, J., et al. (2017). Domain randomization for transferring deep neural networks from simulation to the real world. In *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on (pp. 23-30). IEEE.*

Todorov, E., Erez, T., & Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on (pp. 5026-5033). IEEE.*

Todorov, E. (2018). MuJoCo Plugin and Unity Integration. *http://www.mujoco.org/book/unity.html*

Wang, J. X., et al. (2016). Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*.

Wu, Y., & Tian, Y. (2016). Training agent for first-person shooter game with actor-critic curriculum learning.

Yan, C., et al. (2018). CHALET: Cornell house agent learning environment. *arXiv preprint arXiv:1801.07357*.

Yannakakis, G. N., & Togelius, J. (2018). *Artificial Intelligence and Games*. New York: Springer.

Zhou, K., Doyle, J. C., & Glover, K. (1996). *Robust and optimal control* (Vol. 40, p. 146). New Jersey: Prentice hall.

Zhu, Y., et al. (2017, May). Target-driven visual navigation in indoor scenes using deep reinforcement learning. In *Robotics and Automation (ICRA), 2017 IEEE International Conference on (pp. 3357-3364). IEEE.*