Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Xiaoxiao Ma

Extending a Game Engine with Machine Learning and Artificial Intelligence

Master's Thesis
Espoo, May 14, 2019

Supervisor:	Professor Perttu Hämäläinen
Advisor:	Professor Perttu Hämäläinen

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

ABSTRACT OF MASTER'S THESIS

Author:	Xiaoxiao Ma		
Title: Extending a Game Engine with Machine Learning and Artificial Intelligence			
Date:	May 14, 2019	**Pages:**	54
Major:	Computer Science	**Code:**	SCI3046
Supervisor:	Professor Perttu Hämäläinen		
Advisor:	Professor Perttu Hämäläinen		

Since the early days of Artificial Intelligence (AI), video games have been a popular testbed for evaluating methods. However, not until a long time ago, this only included building AI agents for playing board games like chess. In the last decade, researchers have found out that games are also a rich source for other types of AI problems. At the same time, Machine Learning (ML) has entered its golden age with the advancements of deep learning. In games, this has led to a wide range of novel AI methods for solving various problems like playing games with super-human performance.

Despite the significant advances, state-of-the-art AI methods are still far from being used in commercial games. One of the main reasons is that the tools used by researchers and game developers are different, which makes it difficult to use open-source codes in game projects. Furthermore, implementing these methods requires a moderate understanding of ML, which is not among the skill set of a regular game programmer. This calls for plug-and-play tools that enable game developers to deploy AI methods with minimum cost.

In this thesis, we develop a library that enables game developers to use state-of-the-art ML methods in their commercial projects. This library integrates Tensorflow, a modern ML toolbox, into Unity, the most common game engine in the industry. This library uses C# with intuitive Keras-like API for building and training models. We have also implemented several state-of-the-art algorithms, including Proximal Policy Optimization (PPO) and Matrix Adaptation Evolution Strategies (MA-ES).

This library can also be used along with Unity ML-Agents, the Unity plugin for building AI-training environments. Moreover, we provide various examples that demonstrate the library and the algorithms. One important example is a game called Calamachine Union, where the core game mechanic includes training the AI.

Keywords:	machine learning, games, AI, neural networks
Language:	English

Acknowledgements

The contents of this thesis is largely based on my work as a research assistant in professor Professor Perttu Hämäläinen's group in the Department of Computer Science of Aalto University. I would like to thank him for his great support and giving me the opportunity to work on what I am interested in. I also wish to thank Koray Tahiroglu for the financial support from the Department of Media.

Thanks to Lassi Vapaakallio for making the Calamachine Union game together with me in a game Jam.

Thanks to Amin Babadi and John Weston for their help in revising this thesis. The writing of this thesis would not have been complete without their help.

I would also like to thank Miikka Junnila and all my teammates and classmates during the study of Game Design and Production. It has been a pleasure to work with and learn from them.

Finally, I wish to thank my friends and family who have been great supports during my study in Finland. Special thanks to my parents Shunqin Dong and Pengfei Ma. They are always considerate and patient with me, giving me endless supports regardless of the choices I made.

Espoo, May 14, 2019

Xiaoxiao Ma

Contents

Chapter 1

Introduction

Artificial intelligence (AI) is the concept of machines being able to think like humans. It has been an active field of research for decades. Examples of traditional AI tools include searching, logic programming, finite state machine, and machine learning. Those tools combined with other algorithms have been successfully applied in fields such as robotics, finance, medical and games.

In recent years, machine learning has been one of the hottest topics in both research and industry. With the help of devices with increasing computational power and the vast amount of data available on the internet, it is possible nowadays to run computational-heavy and data-intensive machine learning algorithms such as ones that use deep neural networks. Those algorithms have been showing astonishing results on many tasks. For example, Google's FaceNet could already achieve 99.6% accuracy on the Labeled Faces in the Wild (LFW) dataset in 2015 [32]. Google Multilingual Neural Machine Translation system is used to increase fluency and accuracy in Google Translate [18]. In finance, machine learning is wildly used for fraud detection and insurance underwriting.

In the game industry, AI algorithms are mostly used to create the desired behaviors for Non-Player Characters (NPC) and opponents, and to generate procedural contents. Traditional AI algorithms are already wildly applied in many games. The Civilization series (MicroProse, Activision, Infogrames Entertainment, SA and 2K Games, 1991-2016) has sophisticated procedural algorithms for world generation and searching algorithms for opponent AI; Halo 2 (Microsoft, 2004) uses behavior tree for NPC's combat behavior and team tactics [17]; Half Life uses finite state machine for advanced opponent tactics [40].

With the development of machine learning, the latest game-related research is following the steps as well. In 2013, researchers showed that computers were capable of learning to play multiple Atari games by just looking at the pixels, with one fixed reinforcement learning (RL) algorithm, using deep neural network [26] . In 2017 Google released AlphaGo, a computer program that beat the No. 1 ranked Go player in the world at that time. AlphaGo uses Monte Carlo tree search with knowledge learned by machine learning from human and computer play. Soon later, another version of AlphaGo called AlphaGo Zero, beat the best AlphaGo by learning from playing Go with itself [35]. In 2018, OpenAI Five, a team of five AI players, has started to defeat amateur human teams at Dota 2. They are trained by self-playing using deep reinforcement learning.

However, few commercial games are using those state-of-the-art technologies like machine learning and deep neural networks. Creatures (Millennium Interactive, 1996) was the first popular game that used neural networks to model the creatures' behavior; Black and White (EA, 2000) used reinforcement learning coupled with other AI algorithms. Nevertheless, those games did not use any of the advanced algorithms or computational power from modern hardware. Therefore the AI capability was quite limited. Supercell is using modern machine learning to analyze players' behavior and improve the monetization [6], but it is not directly related to the game itself.

Regardless of whether modern AI and machine learning methods are useful for making a fun game, one reason why we hardly see any application of them in games is the lack of a proper modern machine learning tool that can be easily utilized during the game development, especially for games from smaller studios. Typically, those more advanced methods require more complicated neural network structures and more efficient implementation of math operations.

The current situation of available libraries in machine learning is that those libraries are either not powerful enough for modern algorithms (mainly deep neural networks), not compatible with game development tools, or too low-level for advanced usage. General math libraries such as Eigen and Accord.Net do not support deep neural networks very well. Popular machine learning libraries such as Tensorflow [1] usually have their high-level features written in script languages and do not support most of the mainstream game engines such as Unity. Even though they usually have low-level c interfaces as well, they lack many features for advanced usage and are not necessarily compatible with game engines neither. CNTK C# [34] can be integrated with Unity, but it only supports Windows operating system.

Some other libraries and tools are relatively more accessible in game development. We are going to focus on those libraries in later sections of this thesis. TensorflowSharp is a C# wrapper of Tensorflow's C library. It is simpler to use than the Tensorflow's C++ library, and with some modifications, it can be used directly in Unity. KerasSharp is a reimplementation of Keras [5] in C# using TensorflowSharp as the low-level library. However, the development of KerasSharp is not complete and has been stopped for a long time. Unity's ML-Agents [19] is a toolset which enables building learning environments in Unity, training the AI in Python using Tensorflow, and deploying the trained AI back in Unity. In its latest version released in March 2019, it provides the new Unity Inference Engine, which can load any standard neural network and use it on any platform supported by Unity. However, it does not allow training neural networks in a game.

1.1 Thesis Goals and Scope

The motivation of the thesis is to encourage more game developers to use state-of-the-art AI and machine learning methods in their games. As discussed in the previous section, currently there does not exist any machine learning library that is both powerful and easy to use in a game engine. Moreover, latest machine learning methods, especially game related algorithms such as reinforcement learning, are mostly not implemented in any form that is ready to be used in game production.

Therefore, to make machine learning more accessible to game developers, the primary goal of this thesis is to extend the Unity engine with well-functioning machine learning and AI tools, and provide some implementations of state-of-the-art algorithms with intuitive user interfaces. In this way, game developers without in-depth knowledge of machine learning can utilize this tool to add novel AI to their games.

To be more specific, the thesis goal includes three parts as following:

1. Integrate machine learning libraries into the Unity game engine using TensorflowSharp and KerasSharp, so that developers can implement machine learning methods, especially neural networks with training capability, that run in both the Unity editor and standalone games.

2. Implement modern machine learning algorithms with interfaces that are intuitive to use and compatible with the Unity ML-Agents library.

3. Provide examples of how to use the integrated library in Unity game engine and showcase the potential of using machine learning in actual gameplay.

The thesis is mainly focused on a few state-of-the-art algorithms including reinforcement learning, MAES and supervised learning because of the limitation of resources and time. However, the library is open-source and other ML algorithms could be added to it in the future.

Note that Unity launched ML-Agents after the start of the thesis work. It is trying to accomplish similar goals as this thesis, such as to encourage the use of machine learning in games. Even thought ML-Agents has been in active development and new features keep being added, it is still not enough yet. The main reason is that ML-Agents uses Python-based libraries for training, while the design philosophy of the thesis is to minimize the dependency on Python libraries. The benefit is the capability of training neural networks in games.

1.2 Structure of the Thesis

The rest of the thesis is organized as follows.

Chapter 2 gives the background. It includes two sections - the first section tells the basic of the machine learning and AI algorithms implemented in the thesis while the second section tells the detailed description of the tools used in the thesis.

Chapter 3 firstly gives the design and implementation details of the tools and algorithms. Then it analyses the performance of the implementation by testing it on personal devices and comparing it with Unity ML-Agents.

Chapter 4 gives a summary of the examples implemented to demonstrate the usage and capability of the library of this thesis.

Chapter 5 discusses the advantages, disadvantages and limitations of the thesis over existing tools, and proposes potential future improvements.

Chapter 6 is the conclusion of the whole thesis.

Chapter 2

Background

This chapter contains two sections. The first section tells the essentials of machine learning that are needed to understand the thesis, including some basic machine learning algorithms that the implementations of this thesis are based on. The second section summarizes the existing tools and libraries related to machine learning in game engines.

2.1 Machine Learning

Machine learning (ML) is the general method that computer systems use to perform tasks without using explicit instructions. Instead, it learns to discover the patterns in data automatically and infer the desired results using mathematical models [29]. A machine learning problem usually consists of a data set, a loss function, a model and an algorithm [11]. A machine learning algorithm builds a mathematical model of the training data from the dataset, minimizing the loss function.

The dataset is where the patterns need to be detected. For example, in medical research, one classic problem is to classify cancer patients into different risk groups from medical test data, such as regular blood test results of those patients. In this case, the dataset is the medical test data from many patients and the final results of their cancers after a certain period of time. People want the machine to learn from the dataset to classify patients' risk groups and to generalize the results so that for new patients, the machine can still correctly infer their risk groups as well.

A loss function is a function that maps some events or data into a single number which can intuitively represent the cost or wellness of the events or the data. Usually, a smaller loss value means a better result. For instance, a typical loss function can be the mean square error between the predicted result and the actual result.

There are many models in machine learning, for example, artificial neural network (ANN), support vector machine and Bayesian networks. Only ANN will be discussed in later sections since it is the only model used in the library of this thesis.

Usually, a machine learning algorithm is intended to reduce the loss. If the model is an artificial neural network, the algorithms are usually Stochastic Gradient Descent (SGD) or its modifications. For learning tasks that lack training dataset, such as reinforcement learning, the algorithms can also be the methods of producing training data or defining loss functions.

Typically, machine learning is divided into three categories: **supervised learning**, **reinforcement learning**, and **unsupervised learning**. The thesis involves only supervised learning and reinforcement learning.

In supervised learning, the dataset is organized as input-output pairs. The algorithms try to build a model that match the inputs to their corresponding outputs in the dataset. The loss function is usually a mathematical formula that represents the difference between the given outputs and the outputs coming from the model while fed with corresponding inputs. For discrete outputs, such as true/false or different categories, the loss functions can be cross entropy loss [24]. For continuous outputs, the loss function can be Mean Square Error (MSE).

In reinforcement learning, there is no dataset with input-output pairs. Instead, there is usually an environment where the machine needs to finish specific tasks. The data related to the environments are states, actions, and rewards. For instance, training an AI to control walking robot can be a reinforcement learning problem. The states are the observations of the environment perceived by the robot AI. The actions are the forces/torques applied to each joint of the robot. Rewards are given to the robot if it is walking instead of falling. The AI should learn how to choose the actions in each state such that the sum of rewards obtained during the whole period is maximized.

2.1.1 Artificial Neural Networks

Artificial neural networks (ANN), so-called neural networks (NN), is the most wildly used model in machine learning nowadays. It is inspired by the neural network structure in human brains. Depending on the architectures and hyperparameters of a neural network, it can be either large enough to represent complicated functions or simple enough to run on low-end processors in real time.

Figure 2.1 shows the basic concept of a neural network. Typically, a

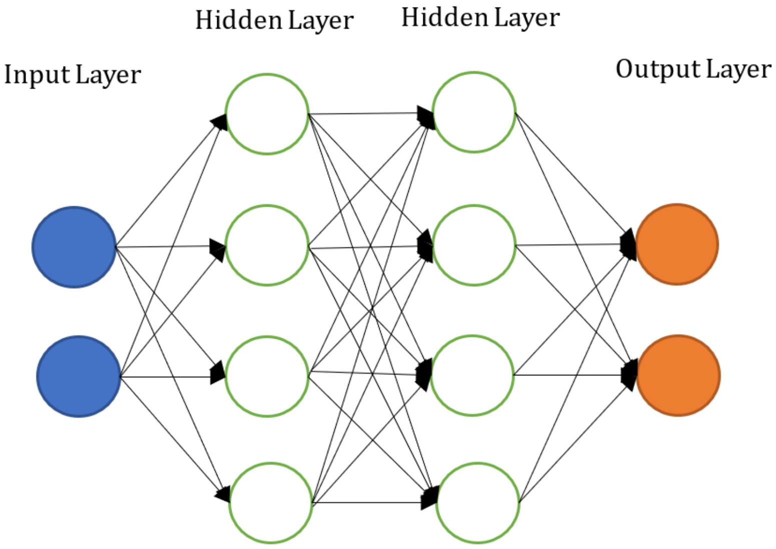


Figure 2.1: An illustration of the structure of an artificial neural network with two hidden layers. Input data is fed into the neurons in the input layer. It is then propagated through the neurons in hidden layers and finally reaches the output layer. Each neuron will modify the data going through it in some way.

neural network consists of structure of neurons. Each neuron in hidden and output layers represents a set of mathematical operations on its input value. The results of the operations in hidden layers go to the neurons in the next layer. The math operations usually include multiplication, addition and then a nonlinear activation function. Mathematically they can be represented as follows:

$$y = f(x) = \sigma(wx + b) \tag{2.1}$$

In the equation above, x is the input, y is the output. w and b are parameters, and σ is the activation function.

Since a layer usually contains more than one neuron, the operations of all neurons in one layer can be represented in forms of vector and matrix operations. Also, if the layers are connected one after each other, the input of a layers is the output of the previous layer. Then, the overall equations are:

$$\begin{gathered} f(\mathbf{x};\theta) = f_n(\dots f_1(f_0(\mathbf{x}))) \\ \text{where } f_i(\mathbf{x}) = \sigma_i(\mathbf{W}_i\mathbf{x} + \mathbf{b}_i) \end{gathered} \tag{2.2}$$

In the equations above, $\mathbf{x}$ and $\mathbf{b}$ are vectors while $\mathbf{W}$ is a matrix. Usually, $\mathbf{b}$

is called the bias and $\mathbf{W}$ is called the weight. θ represents the set of all parameters including $\mathbf{W}$ and $\mathbf{b}$. Each function f_i represents the mathematical operations of one layer. The type of layers that matches the equations above is called dense layers. There are other types of layers such as convolutional layers as well. The equations of them are different, but the concepts are similar.

The Universal Approximation Theorem [7] states that a feedforward network with a linear output layer and a least one hidden layer with some nonlinear activation function (such as the logistic sigmoid activation function) can approximate any function with any degree of accuracy, with the right parameters. Montufar et al.[27] shows that a deep neural network needs less hidden neurons than a shallow (one hidden layer) network to represent the same function. Therefore nowadays, deep learning, which is essentially machine learning with deep neural networks, is the most successful and popular research field.

2.1.2 Stochastic Gradient Descent

To train a neural network to represent the desired function, we need to update the parameters properly. The updating method is usually Gradient Descent (GD) or its modifications. Suppose we have a loss function $L(\theta)$ where θ represents the parameters of the neural network. The goal is to find the θ that minimizes $L(\theta)$:

$$\arg\min_{\theta} L(\theta) \tag{2.3}$$

The gradient of L, $\nabla L(\theta)$, is the direction of θ movement where L increases most rapidly. Therefore, we can iteratively move θ to the opposite of this direction until it reaches a point close enough to the point where the gradient is zero, a local minimum. Equation 2.4 represents the operation on the nth iteration update.

$$\theta_{n+1} = \theta_n - \alpha \nabla L(\theta) \tag{2.4}$$

In the equation above, α is called the learning rate. The learning rate is a small constant, which represents the step size of each iteration. Usually the smaller the learning rate is, the more stable the updates are. However, if α is too small, it might take too long to reach the local minimum.

$\nabla L(\theta)$ is calculated using backpropogation [30], which requires the derivative of the math operations in each layer. Therefore, the activation functions need to be continuous and differentiable nearly everywhere. A popular activation function is Rectified Linear Units (ReLU) [2]

$$\mathrm{ReLU}(x) = \max(0, x) \tag{2.5}$$

When training a neural network, it is desired to minimize the average loss for all x-y pairs in the dataset. However, if the dataset is too large, it requires too much computational power in each update, which slows down the training process a lot. One fact we can use to solve the problem is that the gradient is an expectation, and the expectation may be approximately estimated using a small set of samples. Therefore, on each iteration of the algorithm, we can sample a minibatch of examples drawn uniformly from the training dataset [11]. This method is called Stochastic Gradient Descent (SGD).

There are many modifications of SGD that improves the training. For example, SGD with momentum [31], which adds a momentum term, derived from physics law. It treats the gradient as force and the momentum as speed, and it works better than vanilla SGD in some cases. Some other algorithms adaptively change the learning rate during the training. Adaptive Moment Estimation (ADAM) [20] is the most popular and prominent of all nowadays.

2.1.3 Proximal Policy Optimization

Reinforcement learning was invented long before current deep reinforcement learning. It tries to solve how software agents take actions in an environment to maximize the cumulated rewards. The environment is typically a Markov Decision Process (MDP), which is shown by a tuple (S, A, P_a, R_a). S is a finite set of states, A is a finite set of actions that can be taken, $P_a(s, s')$ is the probability that action a in state s will lead to state s' and $R_a(s, s')$ is the reward to received after transitioning from state s to state s' due to action a.

In reinforcement learning, a policy represents how the agent takes actions based on the current state, which can be represented in Equation 2.6 as π. π returns the probability of taking action a in state s. Sometimes π can also return a random distribution of possible actions.

$$\begin{aligned} &\pi : S \times A \rightarrow [0, 1] \\ &\pi(a \mid s) = P(a_t = a \mid s_t = s) \end{aligned} \tag{2.6}$$

A value function $V_\pi(s)$ is defined as the expected discounted cumulated rewards starting with state s, using policy π:

$$V_\pi(s) = E[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s] \tag{2.7}$$

where r_t is the reward obtained at time step t and γ is a constant called the discount factor. The value γ is between 0 and 1 and defined by the user. It represents how important the future rewards are to the current value.

If the numbers of states and actions are finite, it is possible to use lookup tables to represent $V_\pi(s)$ and $\pi(a \mid s)$. Algorithms such as value iteration, policy iteration, and some temporal difference methods [37] can iteratively find the policy or value functions, therefore solving the reinforcement learning problem. However, those methods do not work if the states and actions have high dimensionality or are continuous, or the policy function is too complicated to be represented using simple models like lookup tables.

Neural networks solve the problems above. Ever since deep Q-learning [26] showed its success in playing Atari games from pixel observations, reinforcement learning with neural networks has been a hot research field, and many algorithms have been proposed for better performance in different scenarios. For example, vanilla Policy Gradient [38], Deep Deterministic Policy Gradient (DDPG) [22] and Proximal Policy Optimization (PPO) [33]. PPO is both efficient and reliable. The baseline algorithm of the thesis is a modification of PPO.

A basic PPO algorithm is shown below:

Algorithm 1: PPO

```
Initialize parameters $\theta$ in the policy network $\pi(s)$ and the value
  network $V(s)$.
for iteration=1,2,... do
    for actor=1,2,... do
        Run policy $\pi_{\theta_{old}}$ in the environment for $T$ timesteps;
        Compute advantages estimates $\hat{A}_1, \ldots, \hat{A}_T$;
    end
    Maximize surrogate $L$ with respect to $\theta$, with $K$ epochs, and
      minibatch size $M \leq NT$, using SGD;
    $\theta_{old} \leftarrow \theta$
end
```

In Algorithm 1, the advantages $\hat{A}_t$ is calculated using Equation 2.8:

$$\begin{aligned} &\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \cdots + \cdots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \\ &\text{where } \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \end{aligned} \tag{2.8}$$

where r_t is the reward obtained at time step t, γ is the discount factor, and λ is the generalize advantage factor, which is usually from 0 to 1. The advantage represents how much the value of sampled actions in the previous simulation is better than the estimated value from the value network.

The surrogate loss L for proposed Algorithm 1 in the original PPO paper is as following:

$$\begin{aligned} L_t(\theta) &= E_t[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t)] \\ \text{where } L_t^{CLIP}(\theta) &= E_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \\ \text{and } L_t^{Value}(\theta) &= (V_\theta(s_t) - V_t^{target})^2 \end{aligned} \tag{2.9}$$

In Equation 2.9 above, c_1 and c_2 are coefficients. S is an entropy bonus such as suggested in [25]. The entropy bonus is useful to encourage exploration during the training. V_t^{target} is a value calculated by $V_t^{target} = V_\theta(s_t) + \hat{A}_t$.

2.1.4 Covariance Matrix Adaptation Evolution Strategy

Covariance Matrix Adaptation Evolution Strategy (CMA-ES) is a stochastic algorithm to deal with non-linear, non-convex optimization problems in continuous domains. It tries to search for the input parameters that produce the optimal output, and it works even if the input-output mapping is a black box.

Consider the following object function:

$$f : \mathbb{R}^n \to \mathbb{R}, \mathbf{x} \to f(\mathbf{x}) \tag{2.10}$$

Assume that $f(\mathbf{x})$ is unknown, but we can still give it an input and get the output value ($f(\mathbf{x})$ is a black box). The goal is to find the solution vector $\mathbf{x}$ which minimizes $f(\mathbf{x})$.

The simplified steps of the CMA-ES algorithm to solve the problem is shown below:

Algorithm 2: CMA-ES

```
Initialize parameters $m$ and $C$ for a multivariate normal distribution
  $\mathbb{N}(m, c)$, where $m$ is the mean and $C$ is the covariance matrix.
for For generation g = 0,1,2,... do
    Sample $N$ input points $(x_1, x_2, x_3, \ldots, x_N)$ from distribution
      $\mathbb{N}(m, c)$;
    Evaluate samples$(x_1, x_2, x_3, \ldots, x_N)$ on $f(\mathbf{x})$;
    Update parameters $m$,$C$ using an adaptive method based on the
      evaluation results;
    If the termination criterion is met, stop the algorithm, otherwise,
      continue with the next generation;
end
```

The methods to update m and C are too complicated to show here. One popular version can be accessed from [14]. The term CMA in CMA-ES comes from the adaptive nature of the covariance matrix updating method.

Recent research shows that the core component of CMA-ES, the covariance matrix itself and some expensive operations in the updating method can be removed from the algorithm without any loss of performance [3]. The final algorithm is called Matrix Adaptation Evolution Strategy (MA-ES), and it reduces the time and space complexity to $O(n^2)$ per sample. For large-scale optimization problems, a variant of MA-ES called Limited Memory Matrix Adaptation Evolution Strategy (LM-MA-ES) was proposed, which has a moderate cost of $O(n \log(n))$ time and space complexity per sample [23].

2.2 Existing Libraries

2.2.1 Tensorflow and TensorflowSharp

Tensorflow [1], initially developed by Google, is one of the most wildly used open-source machine learning libraries in both research and industry. It has strong support for numerical computation, especially for machine learning and deep learning. The core Tensorflow library is in C++, which ensures high performance. Its high-level API and features are primarily in Python. Based on its core, TensorFlow also has Tensorflow.js that runs on web browsers, Tensorflow Lite that can be deployed on mobile devices, and C++/Java API that can be used for general purposes.

TensorflowSharp is a C# wrapper of Tensorflows's C++ core library developed by Miguel de Icaza, using .Net framework. With TensorflowSharp, one can write managed C# codes to use Tensorflow's core features, which is much more intuitive than C++. Original TensorflowSharp only supports Windows, Linux, and MacOS operation systems because those are the supported operation system of Tensorflow standard C++ library. However, with modifications, it can also be built for mobile platforms such as Android and IOS, but with less features, using Tensorflow Lite core library.

TensorflowSharp can be integrated easily into recent versions of Unity with .Net 4.6 support, with only small modifications of TensorflowSharp's source codes for .Net version compatibility. Even though, it still has many limitations regarding its functionality. Because TensorflowSharp uses Tensorflow's low-level C++ API, a lot of high-level features are missing. For example, although it has the function to obtain gradients of mathematical operations, it does not contain any implementation of training algorithms such as SGD at the time of writing the thesis. Also, it does not have operator override, which means one cannot use $+$, $-$, $*$, $/$ symbols in codes to build computational graphs like in Tensoflow's Python API. Moreover, since

Tensorflow Lite C++ library for Android does not have any gradient related feature, building a training scenario on Android devices is not possible (But it is still possible to train model on computers and load it on Android). Furthermore, some of the gradient related operations are missing in Tensorflow C++ API.

2.2.2 KerasSharp

Keras is a high-level API for ML written in Python. It does not have its own ML core library. Instead, it is running on top of other famous libraries such as Tensorflow. One advantage of Keras over other libraries is its user-friendly interface. As the developers of Keras state in its documentation [5] - "Keras is an API designed for human beings, not machines."

KerasSharp is a library that tries to port the Keras line-by-line into C#. It uses TensorflowSharp as one of its backends. However, there has been no update of this project since December 8th, 2017 according to its Github history at the time of writing this thesis. Many of the Keras' features are not ported, and the existing codes are not well tested. Keras had a couple of significant updates in its software architecture after the last update of KerasSharp. Therefore, the architecture and API in KerasSharp are mostly out of date.

Although the KerasSharp project has been stopped already and is probably never going to be continued, it is still useful for this thesis. The backend API for creating a basic computational graph is mostly done in KerasSharp, which means building and evaluating machine learning models are much simpler than in TensorflowSharp. A couple of commonly used optimizers such as ADAM are implemented, which makes it possible to train neural networks in C#. Some commonly used neural network layers such as convolutional layers are also implemented.

However, a moderate amount of work still needs to be done to integrate KerasSharp into Unity. For example, it uses new .Net syntax that is not supported by the .Net version in Unity 2018. The details of the integration and customization process of KerasSharp are discussed in Section 3.1

2.2.3 Unity ML-Agents

Unity is one of the most popular game engines in the world nowadays. The Unity Machine Learning Agents Toolkit (ML-Agents) [19] is an open-source plugin for Unity, maintained by a dedicated team in the Unity company. ML-Agents enables games and simulations that are made with Unity to serve as environments for training intelligent agents. The agents and environments can be accessed through a Python API, which enables researchers and developers to use other popular Python-based machine learning libraries to test their algorithms. ML-Agents also provides implementations of state-of-the-art algorithms for training intelligent agents, and the trained agents can be used in games directly using their Unity inference engine.

Unity ML-Agents has been in active development, and its latest version is V0.7 at the time of writing this thesis. All contents of this thesis are based on that version.

The core ML-Agents is composed of two main parts: the ML-Agents Software Development Kit (SDK), which contains functionality to define the training environments and agents within the Unity engine; a Python package which provides API to interact with the environments built using the ML-Agents SDK.

ML-Agents SDK

The ML-Agents SDK contains three main components: Agent, Brain, and Academy. The Agent component is used to define the intelligent agent in the scene. Developers need to override the Agent component for collecting observations from the environment and updating the agent based on the decided actions sent from its associated Brain component. There can be multiple Agent components in a scene.

Brain components contain the definition of the observation and action spaces, and they make decisions for all associated Agents. Multiple agents can be associated with the same Brain if they share the same observation and action spaces. In ML-Agents, a Brain's decision comes from either player input (Player Brain), predefined scripts (Heuristic Brain), embedded neural network model (Internal Brain) or interaction from ML-Agents' Python API (External Brain).

The Academy component handles the update of the whole scene, including simulation stepping, communication, and other environment configurations.

Figure 2.2 describes how those components work together. The interface design of the thesis is based on ML-Agent SDK and is discussed in Section 3.2.

ML-Agents SDK also provides many example environments that can be used as starting points for designing new environments or as benchmarks for learning algorithms.

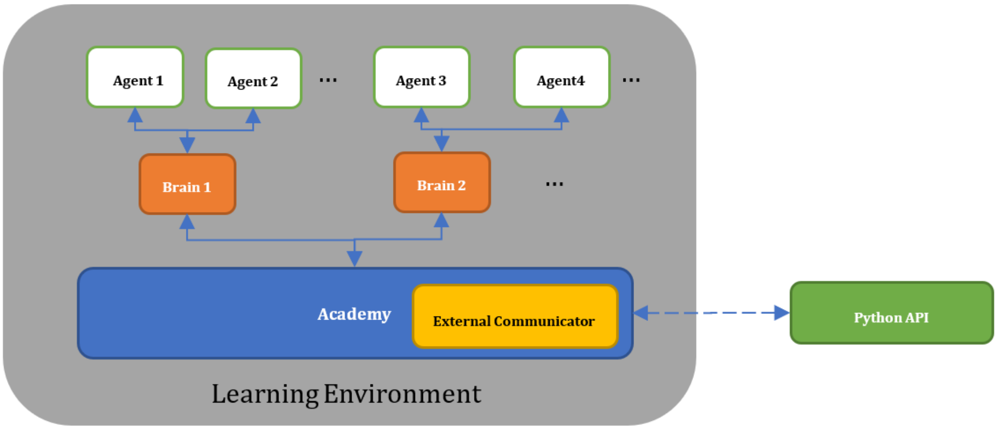


Figure 2.2: Architecture of ML-Agents. Agents are responsible for collecting information and taking actions. Brains are responsible for providing actions using specific policies. The Academy handles the global simulation.

Python Package

The Python package provided by ML-Agents contains interfaces to launch and interact with the learning environments created using ML-Agents SDK as mentioned above. The package also contains a set of wrapper APIs with standard OpenAI Gym interface [4], which is one of the main testing environments used by reinforcement learning researchers. People can easily plug ML-Agents environments into existing codes designed for OpenAI Gym.

ML-Agents provides a set of baseline algorithms in the Python package. Currently, it provides a well-optimized implementation of PPO with the option to extend it using Intrinsic Curiosity Module(ICM) [28] and Long-Short-Term-Memory (LSTM) [15]. It also provides an implementation of Behavioral Cloning (BC) [16]. The implementation of the PPO algorithm in the thesis in C# is based on ML-Agents' Python codes.

Chapter 3

Implementation and Performance

The library presented in this thesis is an open-source project that enables developers to create standalone applications with advanced machine learning features using the Unity game engine. The library was designed with the philosophy that developers can have the advantages of the Unity editor and C# programming language, without worrying about problems like multi-platform support, data communication, and algorithm details.

This library provides some core features and a set of example environments. The core features include three parts: a customized KerasSharp library; some tools for integrating Unity ML-Agents SDK; and several implementations of AI algorithms. The customized KerasSharp library enables developers to write machine learning codes using a Keras-like interface in the Unity editor, and to build it directly into standalone applications. The ML-Agents SDK integration enables developers to use AI algorithms implemented in C# on existing ML-Agents environments. The algorithm implementations provided in this thesis can be directly plugged into an existing environment to train AI without extra coding.

This chapter contains a detailed description of all core features and their implementations. Section 3.1 discusses how KerasSharp was customized and integrated into Unity. Section 3.2 discusses the design of this library's software architecture. Section 3.3 describes the implementation details of provided algorithms. The performance of this library is also analyzed in the later parts of this chapter, namely Section 3.4.

3.1 KerasSharp Customization and Integration

As mentioned in previous sections, KerasSharp enables developers to use all functionalities of TensorflowSharp and other advanced features such as training and constructing neural networks, with intuitive Keras-like interfaces in C#. However, KerasSharp has two main problems that prevent us from using it within Unity.

One problem is that KerasSharp uses .Net 4.7 and C# 7.0 but the Unity game engine only supports .Net 4.6 and C# 6.0 at the time of writing the thesis. The C# 7.0 features which are most commonly used by the original KerasSharp are the syntax for ValueTuple structure and local functions. I had to go over the whole library and convert the codes manually into the C# 6.0 syntax.

Another problem is that the original KerasSharp contains many bugs. One example is that it does not dispose of the memory of the tensor data during model evaluation or training, which causes memory leaks. Another example is that it does not infer the array dimension correctly when building a Tensorflow variable from an array of arbitrary dimension. During the development, I fixed all the bugs that I encountered. However, it is very likely that more bugs remain undiscovered where I did not test or evaluate.

Besides fixing those problems that make KerasSharp unusable in the Unity game engine, I also extended KerasSharp and its relevant libraries with more features.

Support for mobile platforms is added to KerasSharp. TensorflowSharp, the backend of KerasSharp, does not support mobile devices. However, the open source community ported TensorflowSharp to Xamarin using Tensorflow Lite, which supports Android and IOS on mobile devices. I updated the Xamarin version of TensorflowSharp and modified KerasSharp for Unity to add Android and IOS support.

I added the gradient function of the concatenation operation to the Tensorflow C++ library. TensorflowSharp uses the Tensorflow C++ library, which lacks the function to compute the gradient of the concatenation operation. The concatenation operation is used to merge tensors of different shapes into one tensor, and the operation's gradient is needed in order to train the neural networks. The concatenation operation is essential for machine learning algorithms that handle input data with various shapes. Therefore, I added a gradient-computing function for the concatenation operation to the C++ source code of Tensorflow, and then I rebuilt the library.

KerasSharp for Unity also provides the GPU version of the Tensorflow

C++ library for Windows. The GPU library is much faster than the CPU library for running large-scale neural networks such as convolutional neural networks. However, the GPU library needs Nvidia' cuDNN and CUDA to be already installed on PC.

Regardless of these improvements, there are still many limitations with KerasSharp for Unity. For example, training is not supported on mobile devices because Tensorflow Lite for mobile does not include functions to compute gradients. Moreover, some advanced neural network layers such as Long Short-Term Memory (LSTM) [15] are not implemented. Regardless of those limitations, KerasSharp for Unity is already powerful enough for basic machine learning algorithms that can be used in games.

3.2 Software Structure and Unity ML-Agents SDK Integration

The customized version of KerasSharp for Unity described in Section 3.1 works independently in Unity without dependencies such as Unity ML-Agents SDK. While it is possible to implement AI algorithms without ML-Agents SDK, I have decided to design the software structure based on ML-Agents. I also provided Unity editor tools so that the algorithm implementations can be used in any ML-Agents environment directly without extra coding. The two main reasons for using Unity Ml-Agents SDK are as follows.

Firstly, Unity ML-Agents SDK is powerful due to the fact that it supports many different types of learning environments. For example, it supports both supervised learning and reinforcement learning, with either continuous or discrete action/observation space. The library is officially maintained by a team of Unity and is guaranteed to have long-term support in the future. There is no good reason to provide my own software library for building learning environments.

Secondly, Unity ML-Agents is a popular open-source library (over 5000 stars on Github). It is easy to get enough learning environments for testing and benchmarking. Also, the documentation of ML-Agents provides good tutorials for developers to make new learning environments.

Even though I designed the software structure and editor tools mainly for usage in learning environments built with Unity ML-Agents SDK, it is still desired for the trained AI to be portable to any game. For this reason, during the design process, the AIs and the training process were kept relatively independent. The communication with the ML-Agents environments is handled by a separate software layer. This design guarantees that developers can easily remove the dependency on ML-Agents SDK at any time.

This rest of this section consists of a detailed review of the software structure and the tools made for the Unity editor.

3.2.1 Trainer and LearningModel

The library introduces two types of components that make up the core logic of AI: one is the Trainer class, and the other is the LearningModel class. Those two components are represented by two base classes in C#, which contains interfaces and virtual methods that need implementations for specific AI algorithms. The library provides the implementations for some algorithms.

The Trainer and LearningModel classes need to be instantiated for being used in the library. Therefore in the following context, a Trainer/LearningModel means a instance of the Trainer/LearningModel class.

A Trainer is used to execute the training algorithm. It is also the interface through which a LearningModel communicates with the learning environment. In general, at each update step of the training process, the environment will first give information of the agents, such as the observations, to the Trainer, and then query for an action to take from it. Then the environment takes the action, executes its update logic, and finally gives the observation and other information back to the Trainer after the update step. The Trainer automatically processes the incoming information and executes the learning algorithm whenever necessary. The Trainer acts as a bridge between the environment and the LearningModel, and it decides when to train and how to process the training data. However, the Trainer does not deal with the optimization of neural networks.

In this library, a LearningModel contains the information of a machine learning model (usually a neural network), and handles the evaluation and optimization process. For example, if there is a reinforcement learning algorithm using neural networks, the LearningModel contains the hyperparameters and weights of the neural network, the loss function, and the optimizer. One can query actions to take from a LearningModel and call its update function to train the AI, giving it the necessary training data. Once a LearningModel is trained, it can be used as an AI independently without a Trainer.

The relationship between Trainer, LearningModel and learning environments is shown in Figure 3.1.

This library contains implementations of the Trainer and LearningModel classes for baseline algorithms including PPO, Supervised Learning and MAES. For algorithms that use neural networks, there is also a helper class called UnityNetwork, which helps define the neural network architecture in a LearningModel. Given the desired hyperparameters, UnityNetwork can automati-

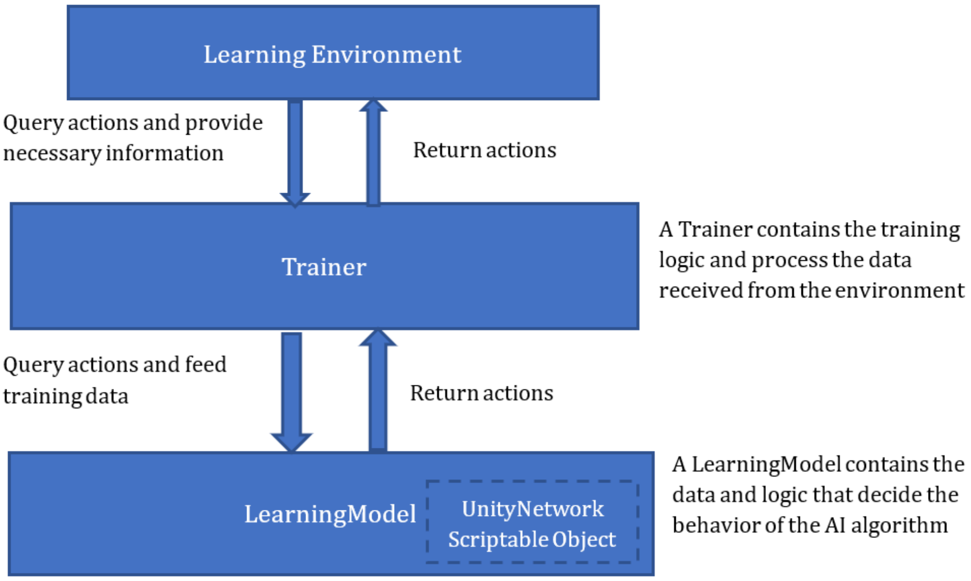
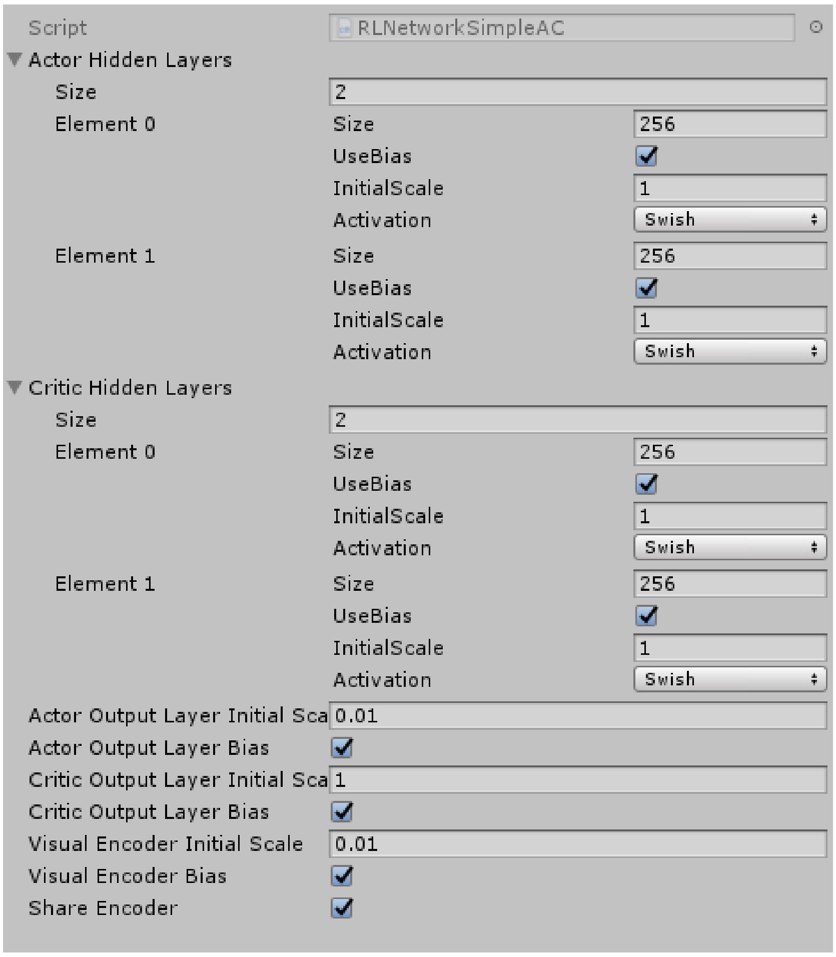

Learning Environment
Query actions and provide necessary information
Return actions
Trainer
A Trainer contains the training logic and process the data received from the environment
Query actions and feed training data
Return actions
LearningModel
UnityNetwork Scriptable Object
A LearningModel contains the data and logic that decide the behavior of the AI algorithm

Figure 3.1: Relationship of Model, Trainer and Learning Environment.

cally build a typical neural network with multiple layers. This neural network can be plugged into other machine learning LearningModels with the same input and output sizes. Since the UnityNetwork is a scriptable object in the Unity game engine, users can quickly change the hyperparameters using the graphic user interface and save them as a file. This feature helps in the tuning of hyperparameters, and makes it much easier to switch between different neural networks for a LearningModel by mouse dragging and click.

3.2.2 Unity ML-Agents SDK Integration

Unity ML-Agents SDK can be used to build learning environments using the Unity engine. The communication of the environments and the AI algorithm goes through the Brain component (See Section 2.2.3). Since ML-Agents version 0.6, there are four types of Brains: the Player Brain, the Heuristic Brain, the Internal Brain, and the External Brain. The Player Brain directly sends player input to the environment. The Heuristic Brain sends observations to a customized C# script and receives actions from it. The Internal Brain uses a trained neural network model to infer actions. The External Brain sends out all necessary observations and information to an external Python process and receives actions from it. Unity ML-Agents uses External Brain and a Python library to train neural network models.



Script	RLNetworkSimpleAC	
▼ Actor Hidden Layers		
Size	2	
Element 0	Size	256
	UseBias	☑
	InitialScale	1
	Activation	Swish
Element 1	Size	256
	UseBias	☑
	InitialScale	1
	Activation	Swish
▼ Critic Hidden Layers		
Size	2	
Element 0	Size	256
	UseBias	☑
	InitialScale	1
	Activation	Swish
Element 1	Size	256
	UseBias	☑
	InitialScale	1
	Activation	Swish
Actor Output Layer Initial Sca	0.01	
Actor Output Layer Bias	☑	
Critic Output Layer Initial Sca	1	
Critic Output Layer Bias	☑	
Visual Encoder Initial Scale	0.01	
Visual Encoder Bias	☑	
Share Encoder	☑	

Figure 3.2: The graphic user interface of a UnityNetwork scriptable object to define a neural network for PPO.

All the Brains are Unity scriptable objects. This allows developers to create Brain files with different types and parameters, and to set Brains of a learning environment easily with mouse dragging. It also allows developers to switch and copy Brains of any learning environment with only a couple of clicks.

To integrate the library provided in this thesis, another Brain called Internal Learning Brain was added. The Internal Learning Brain can be used the same way as other Brains, and it communicates with the Trainer through its interface. Once it is linked to a Trainer, the Internal Learning Brain will collect and process observations, send them to the Trainer and execute the training functions.

With the Unity ML-Agents integration and the AI algorithm implementations, developers can train and use AI with different advanced machine learning methods easily. If there is already a learning environment made with Unity ML-Agents SDK, developers only need to add Internal Learning Brains, Trainers and Models to the scenes by clicking the mouse before they can run the AI algorithms.

3.3 Algorithms Details

Several baseline algorithms were implemented including PPO, supervised learning and MAES for the library. This section contains the details of those implementations.

3.3.1 Implementation Details of PPO

The implementation of PPO in the thesis is based on Unity ML-Agents' Python codes. It is highly optimized and supports multiple types of learning environments. Unity's Python implementation of PPO is different from the vanilla PPO mentioned in Section 2.1.3. This section discusses those differences and provides the implementation details. The next part is organized as follows:

1. An overview of the implemented PPO algorithm.
2. Neural network architecture for inputs encoding (vector/visual observations).
3. Neural network architecture for outputs (discrete action space with branching and masking/continuous action space).
4. Parameter decaying.
5. Input processing (running normalization).
6. Others

An Overview of the Modified PPO Algorithm

Algorithm 3: Unity ML-Agents' PPO

```
Initialize parameters $\theta$ in the policy network $\pi(s)$ and the value
  network $V(s)$;
Reset the data buffer;
for step=1,2,... do
    for agent=1,2,..., running in parallel do
        Run one step with policy $\pi_{\theta_{old}}$ in the environment;
        if the current simulation step of the agent, $t_{agent}$, is a
          multiple of $H$ (the time horizon) then
            Compute advantages estimates from $t_{agent} - H$ to $t_{agent}$
              and add the history of this time period to the data buffer;
        end
        if $t_{\text{agent}}$ reaches $T_{max}$ or agent is done then
            reset the agent;
        end
    end
    if the data buffer is full then
        Maximize surrogate $L$ with respect to $\theta$, with $K$ epochs, and
          minibatch size $M \leq$data buffer size, using SGD;
        $\theta_{old} \leftarrow \theta$ ;
        Reset the data buffer;
    end
end
```

The main difference in the training process between the vanilla PPO and Unity ML-Agents' PPO is when to stop the simulation of an agent and how to collect the training data.

In the vanilla PPO, each agent runs for at most T timesteps, or until the game ends, before the agent is stopped and reset. When resetting the agent, its initial state needs to be randomized. The calculation of advantages uses the whole history of this agent.

However, in Unity's PPO, the agents keep running until T_{max} steps, which is much bigger than T usually. However, the training data is still collected at least every T timesteps from each agent, and the calculation of advantages uses the latest T steps of the history instead of the whole. In this way, the whole simulation history of an agent is separated into several segments, and each segment has T timesteps. A segment is equivalent to a simulation of T timesteps in the vanilla PPO algorithm. Therefore, there is no need to randomize the initial states of each agent because each segment already has different states at the start.

Another difference is that Unity's PPO keeps collecting training data until a data buffer is full, after which the model is updated. This guarantees that for each training epoch, the amount of data is always the same. In the basic PPO, if all agents end earlier than T timesteps because of failure, the data amount for that training epoch will be less than the case when all agents reach T timesteps. However, whether this change in Unity ML-Agents' PPO influences the training efficiency is unknown.

Neural Network Architecture for Inputs Encoding

The implementation of PPO in the thesis uses an actor-critic style architecture the same as Unity's PPO. Both the actor network and the critic network have the same layer hyperparameters for input encoding layers. They can also be configured to share the layer weights if needed.

The neural network can take both visual and vector observations as the inputs. Unity's implementation also takes memory inputs for recurrent neural networks, but it is not supported in our library. The visual observations go through the visual encoder which has some convolutional layers with predefined hyperparameters, and then the output is flattened into 1D neurons. The vector observation goes through dense layers with customizable hyperparameters. In the end, results from dense and convolutional layers are concatenated together to form the encoded input vector. Figure 3.3 shows the overall architecture of the inputs encoding layers.

Neural Network Architecture for Output Values and Output Actions

Unity's implementation of PPO supports both continuous action space and discrete action space. The value output layers in the critic networks are the same for both action space while the action outputs layers in the actor networks are different.

For continuous action space, unity uses the vanilla version of PPO, where the actor network outputs the means and variances of all actions. The final actions to take are sampled from Gaussian distribution with that mean and variance.

In the actor network, the last layer is a dense layer after the encoded inputs as described in the previous section. This layer has an output size the same as the number of required actions. The output values represent the means of all actions.

The variances of the actions do not come from the neural network. Instead, each action has an independent variable which represents the natural

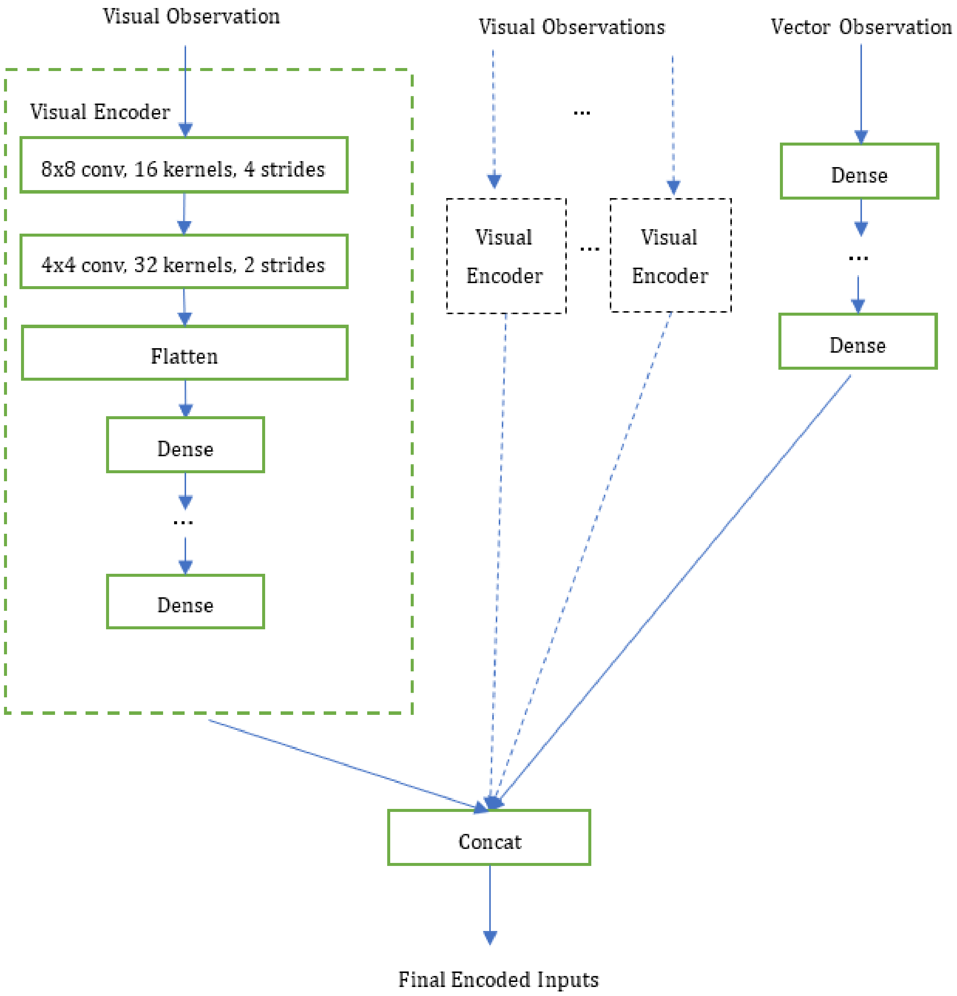


Figure 3.3: The input encoder of the neural networks used in PPO.

logarithm of that action's variance. Those variables are optimized during the training.

For discrete action space, the actor network outputs the probabilities of each possible action. The implementation also supports action branching and masking [39]. It uses masked softmax layer to calculate the probabilities of each action in each branch. See the reference [39] for more details.

Parameter Decaying

Some training related hyperparameters can be modified based on the training steps during the training to make the result better. For example, one can decay the learning rate as the training goes on to make the convergence of the neural network more stable. In the PPO implementation of this thesis, there are two optional methods of modifying training parameters during training automatically: polynomial and Unity animation curve. The decaying equation is:

$$x_t = d(r) = d(t/T_{max}) \tag{3.1}$$

where t is the current training steps, and T_{max} is the maximal training steps. For the polynomial method, $d(r) = (1 - r)^p(x_{init} - x_{end}) + x_{end}$, where x_{init} and x_{end} are the desired initial and end value of that parameter, and p is a constant which can be 1 for linear decay. For Unity animation method, $d(r)$ is defined in a animation curve which can be modified via GUI. Normally, the parameters that should be decayed during the training are c_1, c_2 and ϵ in Equation 2.9 and the learning rate α for SGD.

Others

Unity ML-Agents' PPO uses other tricks in their implementation to improve the training. The benefits of those tricks are not strictly verified but might be useful in some cases. Therefore three of them were implemented in the library.

Firstly, the vector observation can be automatically normalized if the raw observation data is not normalized and causes undesired results. The equation for the running normalization is:

$$\begin{aligned}
\mathbf{m}_t &= \mathbf{m}_{t-1} + (\mathbf{x} - \mathbf{m}_{t-1})/(t+1) \\
\mathbf{V}_t &= \mathbf{V}_{t-1} + (\mathbf{x} - \mathbf{m}_t)(\overline{\mathbf{x}} - \mathbf{m}_{t-1})) \\
\hat{\mathbf{x}} &= (\mathbf{x} - \mathbf{m}_{t-1})/\sqrt{\mathbf{V}_{t-1}/(t+1)}
\end{aligned} \tag{3.2}$$

where $\hat{x}$ is the normalized inputs, t is the time step, m is the running mean and V is the running variance.

Secondly, Unity ML-Agents' implementation normalizes the advantages used for each neural network optimization step as in Equation 2.8 before each training epoch. The normalization of advantages might avoid the failure when advantages are too small or too large, or there are so many negative advantages. People have shown that sometimes negative advantages might make the training unstable [13], but there have not been bad results yet even though the negative advantages are kept after the normalization.

Thirdly, the final actions to take on the agents can be clipped and scaled down to a specific range if the action space is continuous. This feature is useful when only actions within a specific range are acceptable by the agents.

3.3.2 Implementation Details of Other algorithms

Besides PPO, this library also has implementations of other algorithms including supervised learning, Matrix Adaptation Evolution Strategy (MA-ES) [23], neural evolution [36], Generative Adversarial Network (GAN) [12], and PPO-CMA [13]. Those implementations are only the vanilla versions, but they are enough for simple applications in games and are also useful as educational materials. In this section, the work on supervised learning and MA-ES are briefly introduced.

Supervised Learning

It is also possible to train an agent by demonstrating the correct behavior to it instead of training the agent with the help of reward functions. Unity ML-Agents' Python library provides an algorithm called Behavioral Cloning [16]. This algorithm is essentially supervised learning. It works by collecting demonstrations of observations/actions pairs and using them directly to train the policy network. The thesis also provides the implementation of the supervised learning algorithm using the library, and has integrated it with ML-Agents SDK as well.

There are two steps in a supervised learning process. Firstly, the player or another AI plays the game and the demonstration data of observations/actions pairs is recorded. The library provides tools for recording the demonstration data in either the Unity editor or game executables. Secondly, one trains the neural network model using SGD with the collected observations/actions pairs. After the training, giving the neural network an observation similar to one in a collected pair, it is supported to output an action close to the action in that pair.

The loss function of the supervised learning implementation has a couple of options. For discrete action space, which means the neural network outputs the probability of each possible action using softmax function, categorical crossentropy loss is used. For continuous actions space, users can choose to use either mean square error or Bayesian loss as in Equation 3.3. If users need the neural network to output both the mean and the variance of actions, the loss function needs to be Bayesian loss. If only the mean is needed, the loss function needs to be mean square error.

$$L_{Bayesian} = \frac{||y - \hat{y}||_2}{2\sigma^2} + \frac{1}{2}\log\sigma^2 \tag{3.3}$$

The neural network architecture for the policy network of supervised learning can be the same as the one used in PPO. In this case, users can first train a neural network with supervised learning, then use the trained neural network as the initial one for reinforcement learning algorithms such as PPO. This method might shorten the training process of reinforcement learning if it is difficult to make progress at the start [8]. Google's AlphaStar [9] uses a similar concept even though the algorithm is much more complicated.

MA-ES

It is also possible to use MA-ES/LM-MA-ES to search for the best actions without training. The implementation of MA-ES and LM-MA-ES in C# comes from Perttu Hämäläinen, and it is ported from Ilya Loschilov's C++ code provided with his paper [23].

A helper class called ESOptimizer is provided for using MA-ES in Unity. Developers only need to implement a C# interface for their game system to evaluate the value of actions at any state, which is needed by the algorithm. Then they can start the optimizing process by attaching the ESOptimizer class to any Unity Gameobject and call the start method in C# scripts. The hyperparameters of the algorithm can be adjusted in the Unity editor.

3.4 Performance

This section of the thesis analyses the performance of the library. The library might be used for AI research, but it is mainly designed for developers who want to utilize machine learning algorithms in games. It is useful to know how fast the training and evaluation execute on a regular computer under different situations so that developers can design proper agents and learning environments that run with an acceptable frame rate.

This section firstly compares the sample efficiency of the PPO implementation of the thesis with the Python version of Unity ML-Agents on PC. Then the time efficiency for training and evaluating of neural networks with different hyperparameters is analyzed on both PC and mobile devices. The test PC has an Intel Core i5-6300HQ CPU and a NVIDIA GeForce GTX 960M GPU, and the test mobile phone is an Honor 10V with a HiSilicon Kirin 970 CPU. Note that GPU is only utilized by the GPU version of Tensorflow C++ library when processing on visual observations during the test.

3.4.1 Comparison with ML-Agents

To verify that the implementation of PPO is almost identical to Unity ML-Agents, it is necessary to compare the sample efficiency of those two implementations. Some of the example environments from Unity ML-Agents are used to train the agents with the same hyperparameters for both implementations. Figure 3.4 shows the results. In Figure 3.4, the plots for both PPO implementations are close to each other, which demonstrates that those two implementations are comparable.

Note that even though the sample efficiency is similar, the time efficiency of this PPO in C# is better when the data amount generated in each step is large. The reason is that ML-Agents uses web socket to transfer data between the game process and the Python process, which has limited bandwidth. The codes of our library are running within one process, and no inter-process data transfer is required. Therefore the data transfer bandwidth becomes the bottleneck and slows down the training of ML-Agents when the data amount is large.

3.4.2 Time Efficiency on Different Devices

Many factors decide how long it takes to call the evaluation or the training on a neural network. It is essential for developers to have a rough idea about the performance to make sure the designed game can reach a reasonable FPS. Two of the main factors that affect the performance are the number of parameters in the neural network and the batch size for evaluation and training. I ran some experiments to analyze the effects of those factors.

A learning environment called Empty is made for benchmarking. In the Empty environment, the agents do not execute any logic based on their actions. The observations sent to the Brain are vectors of random numbers. This makes sure that the execution time of the environment is minimal. The action and observation sizes are both 64, and the action space is continuous.

In the first experiment, the input/output size and the batch size are fixed, while the depth and the width of the neural network are the variables. For evaluation, the batch size (number of agents) is 10. The time of each

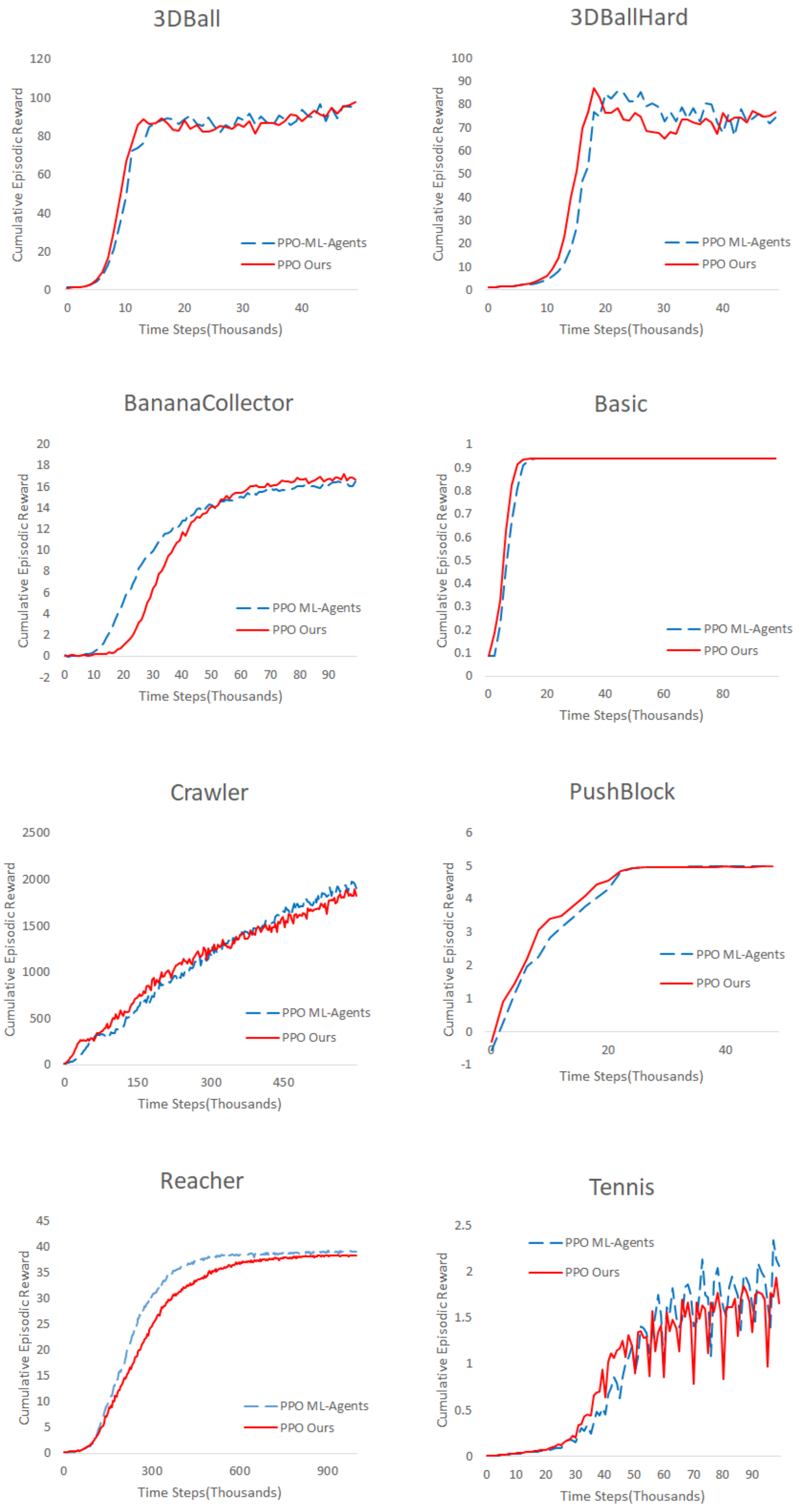
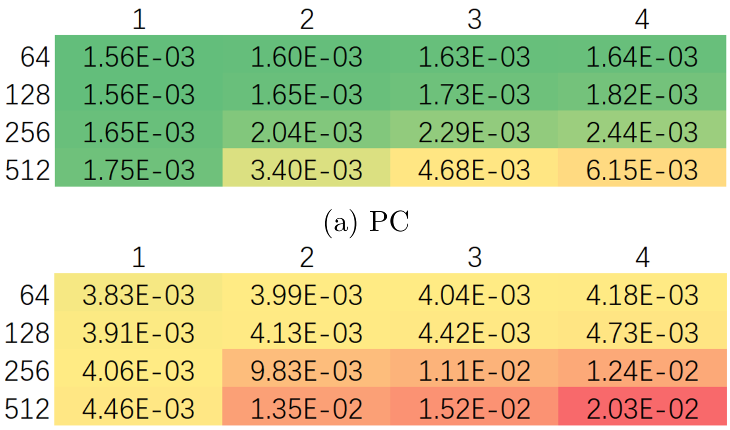


Figure 3.4: Comparison between the implementation of PPO in the thesis and Unity ML-Agents. The results are the mean of five separate runs.



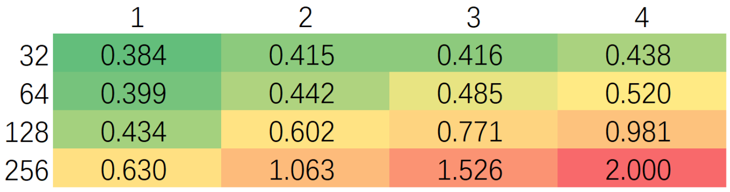
	1	2	3	4
64	1.56E-03	1.60E-03	1.63E-03	1.64E-03
128	1.56E-03	1.65E-03	1.73E-03	1.82E-03
256	1.65E-03	2.04E-03	2.29E-03	2.44E-03
512	1.75E-03	3.40E-03	4.68E-03	6.15E-03

(a) PC

	1	2	3	4
64	3.83E-03	3.99E-03	4.04E-03	4.18E-03
128	3.91E-03	4.13E-03	4.42E-03	4.73E-03
256	4.06E-03	9.83E-03	1.11E-02	1.24E-02
512	4.46E-03	1.35E-02	1.52E-02	2.03E-02

(b) Mobile

Figure 3.5: Time cost (second) in each evaluation step with different depth (column labels) and width (row labels) of the neural network.



	1	2	3	4
32	0.384	0.415	0.416	0.438
64	0.399	0.442	0.485	0.520
128	0.434	0.602	0.771	0.981
256	0.630	1.063	1.526	2.000

Figure 3.6: Time cost (second) in each updating step with different depth (column labels) and width (row labels) of the neural network.

evaluation step is measured on both PC and mobile devices, summarized in Figure 3.5. For training, the time of each update step is measured on PC only, summarized in Figure 3.6. Note that during the training process of Algorithm 3, an update is only performed on the steps when the data buffer is full; otherwise, the evaluation is performed. In the test to measure the update time of training, the data buffer size is 2048, the epoch number is 3, and the minibatch size is 128.

In the second experiment, the neural network has fixed depth and width (2 and 128), while the number of agents changes. This experiment is designed to figure out how many agents at most can use the neural network for evaluation at the same time. Figure 3.7 shows the results for both PC and mobile device.

In a game made with Unity, the evaluation for querying actions of agents is called at most 50 times per second, which gives a 0.02 seconds interval. According to the results, the evaluation time is much less than 0.02 seconds on a regular PC, even if the neural network or the number of agents is large. Therefore the performance is not an issue on PC for a regular game. However,

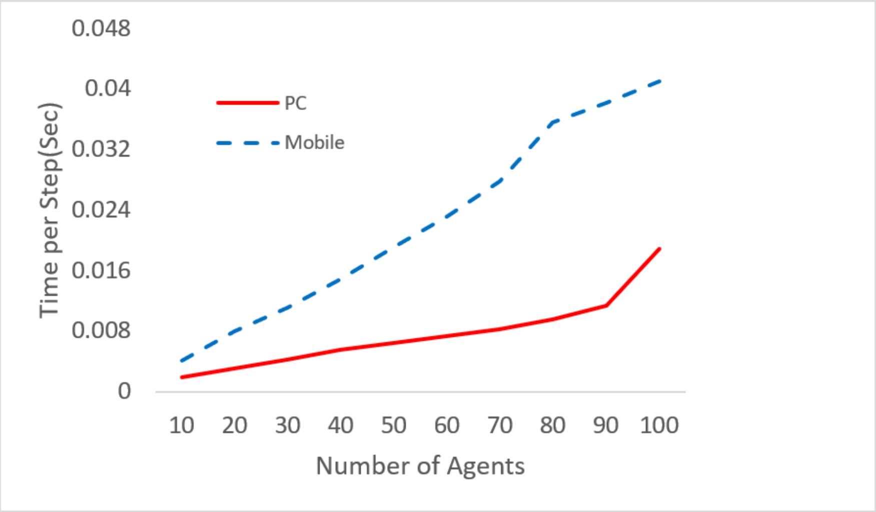


Figure 3.7: The evaluation time versus the number of agents.

on mobile devices, the limitation on the number of agents or the size of neural networks is more strict if an acceptable FPS is required.

Chapter 4

Examples

The library in this thesis provides plenty of examples. They demonstrate the capability of this library as well as how to use different machine learning tools provided. Some of the examples also have intuitive visualization to help developers understand the principles behind those algorithms. Next, there is a summary of all the examples. Figure 4.1 shows the screenshots of all the examples. After the summary, two of the primary examples will be discussed with more details.

3D Ball

The 3D Ball environment is copied from Unity ML-Agents' examples, but the AI uses this library so that training and evaluation can run in the Unity editor or a game executable. The agents have continuous action space (tilting angles) and vector observations (position and velocity of the ball). The AI needs to balance the ball and keep it on the platform as long as possible by controlling the tilt of the platform. The algorithm used in this example is PPO.

Banana Collectors

The Banana Collectors environment is copied from Unity ML-Agents' examples and modified to use this library as well. The agents have discrete action space with branching and vector observations. The AI can see what objects are in front of the agents by casting several rays ahead. It tries to move the agents to collect good bananas, avoiding bad bananas while shooting at each other. The algorithm used in this example is PPO.

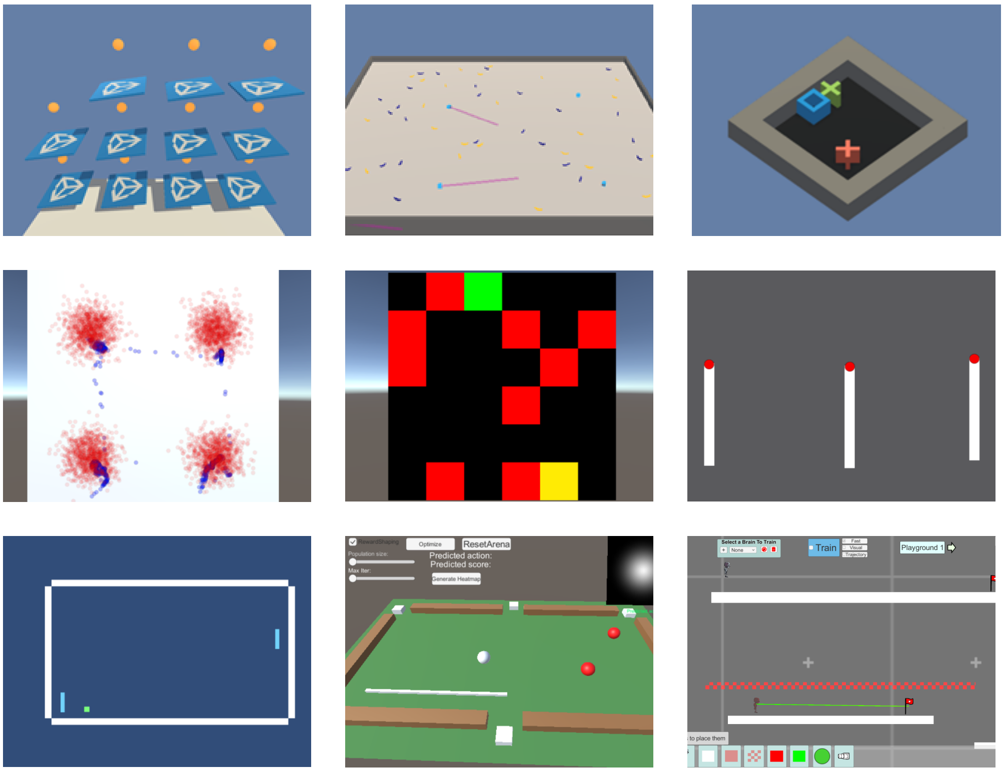


Figure 4.1: Images of example environments and games. Some of the environments are cloned and modified from ML-Agents. From Left-to-right, up-to-down: (a) 3DBall, (b) Banana Collectors, (c) Grid World, (d) GAN 2D Plane, (e) Maze, (f) Pole, (g) Pong, (h) Intelligent Pool, (i) Calamachine Union.

Grid World

The Grid World environment is copied from Unity ML-Agents' examples and modified to use this library as well. The agents have visual observations and discrete action space with masking. The AI needs to move (four directions on a grid) the blue square towards the green target without touching the red cross. The algorithm used in this example is PPO.

GAN 2D Plane

The GAN 2D Plane environment demonstrates how to train a GAN to generate data points from a 2D multimodal Gaussian distribution.

Maze

In the Maze environment, the yellow agent needs to reach to the green destination and avoid the red obstacles. It is a more complicated version of the Grid World. The agents have visual or vector observations, and discrete action space. The algorithm used in this example is PPO.

Pole

In the Pole environment, the agents need to keep inverted pendulums balanced by applying torques on them. The agents have either visual (the image of the pendulum) or vector (current angular velocity and angle of the pendulum) observations and continuous action space. The algorithm used in this example is PPO.

Pong

The Pong environment is similar to the classic Pong (Atari, 1972) game. Two agents are playing against each other in the arena. The agents have vector observations (positions of balls and both agents), and discrete actions space (moving up, down or staying).

The Pong environment includes multiple scenes that use different learning algorithms: PPO, supervised learning and PPO with neural networks initialized from supervised learning.

Intelligent Pool

In the Intelligent Pool environment, the AI tries to play a simple pool game automatically. It combines MAES and supervised learning algorithms, and provides nice visualizations for people to understand the principle and the limitation of those algorithms. The details are discussed in Section 4.1.

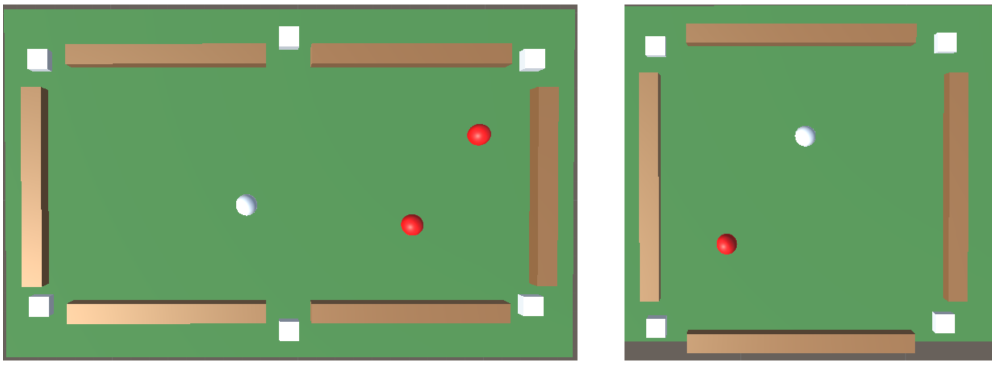
Calamachine Union

Calamachine Union is a game using reinforcement learning as the main mechanic. The base of this game is made by Lassi Vappakallio and me in a game jam. It can be used as an educational material which teaches people the concepts and capability of reinforcement learning. The details are discussed in Section 4.2.

4.1 Intelligent Pool

The Intelligent Pool, a set of examples of the billiard game, is mainly developed as the materials for the Intelligent Computational Media course. The examples use different algorithms to showcase their concepts and capability. The start of the examples is a simple environment where the AI needs to hit the white ball once and tries to make both red balls on the table into pockets, using MAES algorithm. The final goal is to develop an AI that can play a complete game with itself. However, the final goal is not achieved. Instead, two simplified environments were used in the end.

Each example is running in one of those two environments: environment 1 has two red balls and six pockets; environment 2 has one red ball and four pockets.



(a) Environment 1

(b) Environment 2

Figure 4.2: Two environments used in the Intelligent Billiard. Note that the positions of balls are randomized on start.

There are five examples as listed below.

1. Use MAES to find the optimized solution for one shot in environment 1.

2. Use MAES to find the optimized solution for two consecutive shots in environment 1.

3. Use supervised learning together with MAES for one shot in environment 1.

4. Use PPO in environment 1.

5. Use supervised learning together with MAES for one shot in environment 2.

Example 1 and Example 2

The only difference between example 1 and example 2 is that in example 1 the AI needs to find the best single shot while in example 2 it needs to find the best two consecutive shots, both with random initial ball positions.

Those examples show the importance of reward shaping when using MAES. Reward shaping is the idea of modifying the reward of each action so that MAES can find the optimized solution more easily. In the examples, not shaping the reward means the value of a shot is evaluated by how many red balls are in pockets after the shot. In this case, the MAES optimizer is not able to find the best shot. However, if we shape the reward function so that the value of a shot is also determined by how close the red balls are to the pockets, the MAES optimizer is able to find the best shot.

Both example 1 and example 2 work well with reward shaping enabled. Example 2 takes more time for optimization because the action dimension is larger for two consecutive shots than one shot.

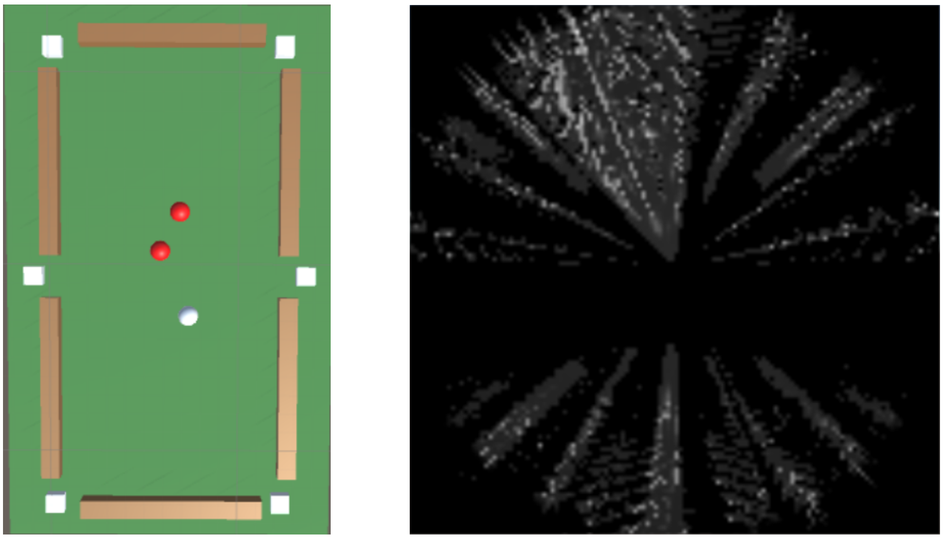
Example 3

The goal of this example is to use supervised learning on neural networks for generating meaningful actions and use those as the initial guesses of the MAES optimizer. With the initial guess, the optimizer should be able to find the best result faster. The data for supervised learning is collected from running the environment with MAES as in example 1 and 2.

This idea has been proven to be helpful [29]. However, it does not work in this case. The trained neural network often outputs an action that is not even close to the best action that a MAES optimizer might choose. It is misleading the MAES optimizer rather than helping it.

One reason why the supervised learning does not work well is the discontinuous distribution of the optimal solutions. Figure 4.3 shows the heatmap of all possible shots in a scenario. The whitest pixels, which indicate this shot can pocket two red balls, are very rare on the heatmap. However, there are plenty of less white pixels scattered. Those pixels mean only one red ball gets in the pocket.

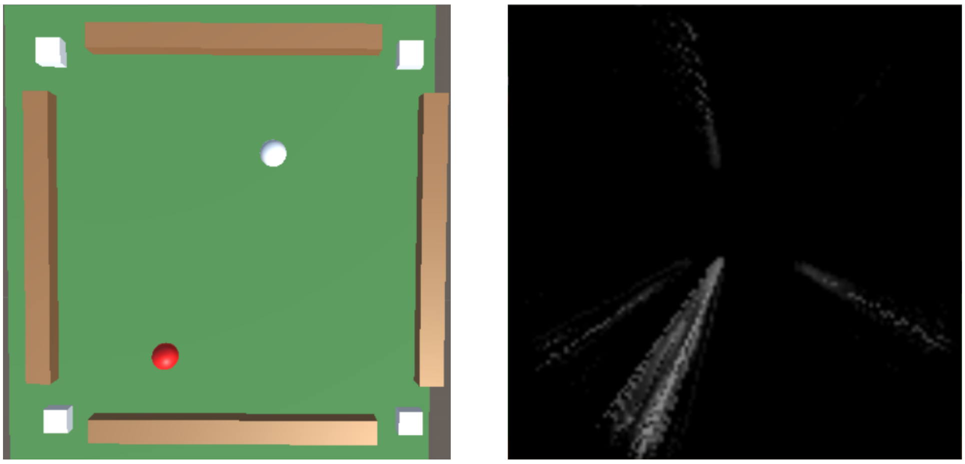
The training data used for supervised learning is collected by running MAES optimization. Since the perfect white pixels are rare, when optimizing with MAES, it is not very likely that the optimal solutions can be found every time. Also, because white pixels are scattered around, the sub-optimal solution found by MAES might far from each other every time as well. Therefore, in the collected data of state-solution pairs, sometimes even if the states are very similar, the solutions might still vary a lot.



(a) Environment

(b) Heatmap

Figure 4.3: A scenario in environment 1 and its heatmap. The pixel coordinate relative to the center represents the shooting direction and force, and the color represents the objective score of that shot. The whiter a pixel is, the higher the score is.



(a) Environment

(b) Heatmap

Figure 4.4: A scenario in environment 2 and its heatmap. The pixel coordinate relative to the center represents the shooting direction and force, and the color represents the objective score of that shot. The whiter a pixel is, the higher the score is.

The discrepancy in the training data causes problems for supervised learning. The supervised learning usually tries to reduce the error between the neural network's outputs and the outputs in the training data. If there are multiple different outputs for the same input in the training data, the neural network will try to generate the mean of all those outputs given this input. This minimizes the error but the actual output from the neural network might be meaningless. In this case, the supervised learning neural network learns to output the mean of some white pixels' positions on the heat map. It is likely that this mean position is a dark pixel instead of a white pixel, and a dark pixel means a bad action. Above is a reason why this neural network cannot learn anything helpful.

There might be other reasons for the bad performance of the neural network, such as the neural network architecture is not good enough, the neural network size is not big enough, or the training data set is not large enough. Those can be topics for future research.

Example 4

No good result was obtained using PPO during the experiments. The reason remains undiscovered and can be analyzed in the future. The hypothesis is that it might be the lack of training time or better/larger neural network architecture.

Example 5

This example uses environment 2, which is much simpler than environment 1. The same algorithms as in example 3 are used as well.

According to the heatmap in Figure 4.4, the white pixels are more aggregated than in example 3. This alleviates the problems mentioned in example 3. Also, a simpler scenario requires a smaller neural network and less training time to reach a good result.

After collecting 20000 samples and training the neural network for 30 minutes, the AI using only the neural network can at least shoot the white ball to touch the red ball every time, and sometimes pocket it.

If using the output from the neural network as the initial guess of the MAES optimizer, the performance improvement is noticeable. The average iteration count to find a satisfying solution is reduced from 9.72 to 4.72, and the average objective score also increases by about 0.1, as shown in Table 4.1.

	Mean MAES Iterations	**Mean Score**
MAES with NN	4.72 ± 1.47	0.89101 ± 0.0401
MAES only	9.72 ± 1.33	0.79048 ± 0.0690

Table 4.1: Comparison of performance between MAES with NN and MAES only, in Intelligent Pool environment 2. The results are based on 50 runs for each, with a 95% confidence interval.

4.2 Calamachine Union

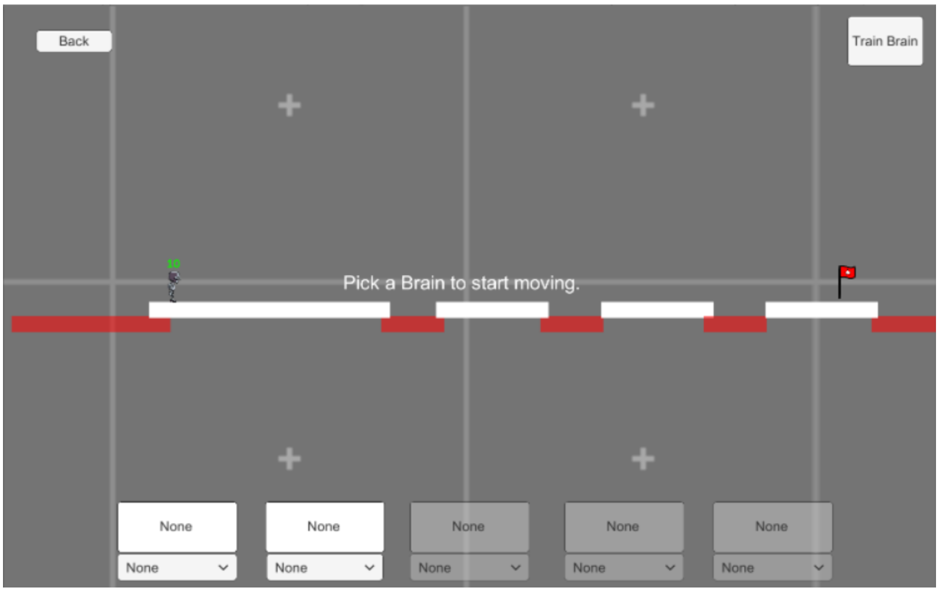
Calamachine Union is a game that uses deep reinforcement learning as one of the game mechanics. In the game, the player leads a group of robots called Frank from point A to point B on 2D platforms. There is no way to control the characters, and they are entirely controlled by the AI trained by the player. What the player can do is to create learning environments where AI learns how to earn rewards. These trained AIs are saved, and the player can choose which one to use on their adventures. For example, one AI might know how to move to the right and another might know how to jump. Though these AIs can get a lot more complicated than that.

In the game, there are several pre-designed levels where the player needs to use the trained AIs to play. There is also a playground, where the player can put items to build learning environments to train the AIs.

Figure 4.5 is a screenshot of a level. In the levels, the AI needs to control those robots to move and jump to reach the goal position and to avoid being killed by red killing zones. The action space of the AI is discrete, and there are five possible actions: move right, move left, jump, crunch or do nothing. The observation of the agents is a vector of 88 numbers, which represent the ray casting results of 44 directions from the agent, each of which contains a distance value and a color value.

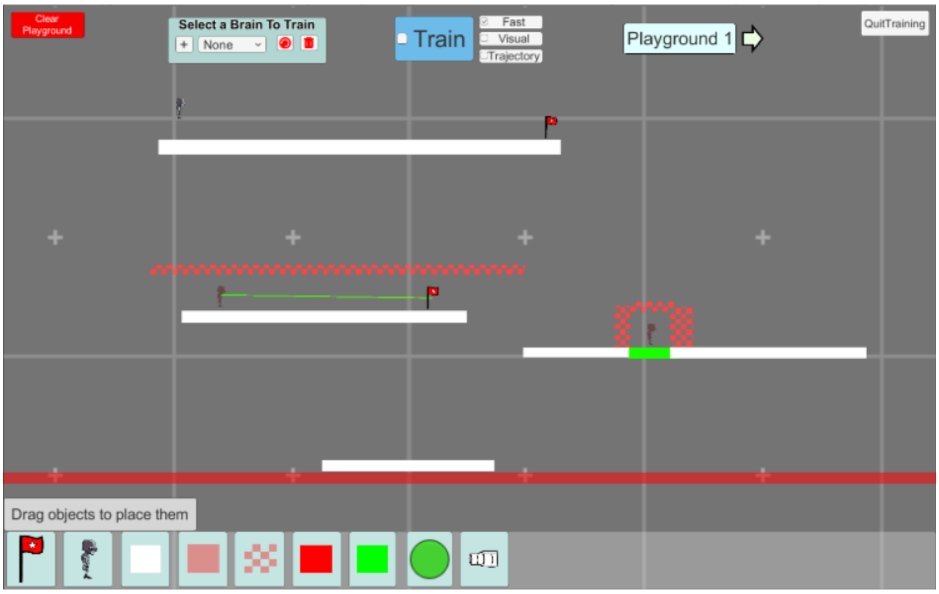
Figure 4.6 is a screenshot of the playground scene. The player can put items like platforms, killing zones, spawn positions, and target positions by drag and drop. During the training, agents will spawn at the spawn position initially. They obtain positive rewards when moving closer to the target positions or touching the reward block, and negative rewards when being killed. Items have different colors, and some of the items are even invisible by the agents. When the learning starts, players can see the visualization of agents' trajectories and observations.

The game is a good learning material for people to know what reinforcement learning is. It also showcases the capability of this library for game development. With this example, game developers can start to think about



Pick a Brain to start moving.

Figure 4.5: A level in Calamachine Union.



Drag objects to place them

Figure 4.6: The playground in Calamachine Union.

utilizing reinforcement learning in real games.

Chapter 5

Discussion

This thesis provides an open-source library for using state-of-the-art machine learning algorithms in the Unity game engine. This integrates Unity and machine learning without needing to run algorithms in Python. In this chapter, we firstly discuss the advantages and disadvantages of the library. Then two improvements are suggested to make in the future.

The main advantage for users or players is that the library minimizes the software dependencies and allows training of neural networks during the gameplay. Other libraries, such as Unity ML-Agents, require Python and some machine learning tools to be installed. They also need multiple processes to be opened during the training. Those are not practical for the distribution of commercial games which needs to minimize the installation complexity for players.

There are three advantages for developers to use the library for running machine learning algorithms in the game engine. Firstly, it is easier to handle the data transfer when developing learning environments and learning algorithms if everything is running in one computer process. If using Python, the data transfer between the game and the Python process requires inter-process communication and encoding/decoding of data, which increases the working load and slows down the iteration. Secondly, the powerful and flexible editor tools provided by the Unity game engine make it easier to setup the training process or tuning the hyperparameters. The library also provides a customizable tools for visualizing the training process. Thirdly, the programming language used in Unity, C#, is better for developing software applications than Python. Developers have more access to low-level data structures to optimize the running time. C# is also a strong-typed language which is easier to debug than Python.

Developing machine learning in the game engine also has some disadvantages over using Python, and we discuss two of the main ones next. Firstly, there are numerous machine learning libraries, tools and the-state-of-art algorithms in Python, while there are very few in C#. For example, the Tensorflow Python library has tools and features such as Tensorboard [1], Open Neural Network Exchange (ONNX), distributed training and a considerable amount of pre-implemented advanced neural network architecture, which the library in the thesis does not have. Secondly, C# lacks some of the simplified syntax for mathematical operations found in Python. This means that C# needs more lines of codes than Python to implement the same machine learning method.

Even though the library already has good support for utilizing machine learning in games, a lot of commonly used features are still missing. Two major improvements are suggested to make in the future.

The first improvement is to add more neural network architectures and training algorithms. For example, the Unity ML-Agents library has two features that are not supported in the library of the thesis: the recurrent neural networks, and the curiosity module. The first of these, the recurrent neural network, is critical for AI that needs memory. The second feature, the curiosity module, improves the training if the rewards obtained from the environments are sparse and extrinsic. Also, other popular algorithms such as Model-Agnostic Meta-Learning [10], Q-Learning and style transfer [21] can also be included in the library.

Secondly, we can improve the multiplatform support of the library. Currently, training and convolutional neural networks are not supported on mobile platforms because the standard way of building the Tensorflow Lite library for mobile platforms does not include those advanced mathematical operations. This can be fixed by re-configuring the building process of Tensorflow Lite, which is difficult without enough time and resources. Another platform-related issue is that some basic operations such as concatenation are not supported on Mac or Linux operation systems. This can be fixed by rebuilding the Tensorflow C++ libraries with modifications on corresponding operating systems.

Chapter 6

Conclusions

The library in the thesis provides machine learning tools for the Unity game engine. The existing KerasSharp was customized for Unity so that it can be used directly in Unity editor and standalone games built with Unity. This library does not require the installation of other libraries, or communicating with other computer processes. It enables the developers to develop games or other applications that contain advanced machine learning capabilities, using the C# programming language in Unity. The elegance of C# and the intuitive graphic user interface in the Unity editor make the development process more straightforward than using other tools and languages such as Python.

Several state-of-the-art machine learning algorithms are also implemented and integrated with the popular Unity ML-Agents SDK. The algorithms include PPO (based on Unity ML-Agents' Python implementation), MAES, supervised learning and others. For any learning environment built with ML-Agents, developers can choose to use the implemented algorithms to train the AI without any coding, or implement other algorithms easily using KerasSharp and our software code base. The implementation of PPO of the thesis has similar performance to Unity ML-Agents' regarding sample efficiency, while faster in time when running on a regular PC.

The thesis also contains some example environments and games for educational purposes. The Intelligent Pool example provides the visualization of the optimizing process using MAES, and the heatmap of the 2D searching space of a game. It also analyses the method of combining MAES with supervised learning to improve the optimization performance. The Calamchine Union game is an example of how machine learning can be used as the core mechanic in a game. It also helps the player to learn the concepts of reinforcement learning by allowing them to build the learning environments easily and visualizing the training process in an intuitive and interesting way.

Other environments together provide examples of using different algorithms for different types of agents.

The main limitations of the work are as follows. Firstly, more other state-of-the-art algorithms need to be implemented and integrated into the library. Secondly, some of the features such as training a neural network are not supported on some platforms such as Android and iOS. Thirdly, some popular neural network architectures such as recurrent neural networks are not supported by the library.

The mentioned limitations can be addressed in the future work. Experiments show that this library can be used as a straightforward toolbox for studying machine learning and deploying machine learning in games.

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Agarap, A. F. Deep learning using rectified linear units (relu). *CoRR abs/1803.08375* (2018).

[3] Beyer, H., and Sendhoff, B. Simplify your covariance matrix adaptation evolution strategy. *IEEE Transactions on Evolutionary Computation 21*, 5 (Oct 2017), 746–759.

[4] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym. *CoRR abs/1606.01540* (2016).

[5] Chollet, F., et al. Keras. `https://keras.io`, 2015.

[6] Clayton, N. How supercell uses machine learning to automate monetisation in clash royale, 2018. [Online; accessed 18-February-2019].

[7] Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems 2*, 4 (Dec 1989), 303–314.

[8] de la Cruz, G. V., Du, Y., and Taylor, M. E. Pre-training neural networks with human demonstrations for deep reinforcement learning. *CoRR abs/1709.04083* (2017).

[9] Deepmind. Alphastar: Mastering the real-time strategy game starcraft ii, 2019. [Online; accessed 22-February-2019].

[10] Finn, C., Abbeel, P., and Levine, S. Model-agnostic meta-learning for fast adaptation of deep networks. *CoRR abs/1703.03400* (2017).

[11] Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[12] Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. Generative adversarial nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2* (Cambridge, MA, USA, 2014), NIPS'14, MIT Press, pp. 2672–2680.

[13] Hämäläinen, P., Babadi, A., Ma, X., and Lehtinen, J. PPO-CMA: proximal policy optimization with covariance matrix adaptation. *CoRR abs/1810.02541* (2018).

[14] Hansen, N. The CMA evolution strategy: A tutorial. *CoRR abs/1604.00772* (2016).

[15] Hochreiter, S., and Schmidhuber, J. Long short-term memory. *Neural Comput. 9*, 8 (Nov. 1997), 1735–1780.

[16] Hussein, A., Gaber, M. M., Elyan, E., and Jayne, C. Imitation learning: A survey of learning methods. *ACM Comput. Surv. 50*, 2 (Apr. 2017), 21:1–21:35.

[17] Isla, D. Gdc 2005 proceeding: Handling complexity in the halo 2 ai, 2005. [Online; accessed 18-February-2019].

[18] Johnson, M., Schuster, M., Le, Q. V., Krikun, M., Wu, Y., Chen, Z., Thorat, N., Viégas, F. B., Wattenberg, M., Corrado, G., Hughes, M., and Dean, J. Google's multilingual neural machine translation system: Enabling zero-shot translation. *CoRR abs/1611.04558* (2016).

[19] Juliani, A., Berges, V., Vckay, E., Gao, Y., Henry, H., Mattar, M., and Lange, D. Unity: A general platform for intelligent agents. *CoRR abs/1809.02627* (2018).

[20] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. *CoRR abs/1412.6980* (2014).

[21] Li, Y., Fang, C., Yang, J., Wang, Z., Lu, X., and Yang, M. Universal style transfer via feature transforms. *CoRR abs/1705.08086* (2017).

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

[23] Loshchilov, I., Glasmachers, T., and Beyer, H. Limited-memory matrix adaptation for large scale black-box optimization. *CoRR abs/1705.06693* (2017).

[24] Mannor, S., Peleg, D., and Rubinstein, R. The cross entropy method for classification. In *Proceedings of the 22Nd International Conference on Machine Learning* (New York, NY, USA, 2005), ICML '05, ACM, pp. 561–568.

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

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

[27] Montúfar, G., Pascanu, R., Cho, K., and Bengio, Y. On the Number of Linear Regions of Deep Neural Networks. *arXiv e-prints* (Feb 2014), arXiv:1402.1869.

[28] Pathak, D., Agrawal, P., Efros, A. A., and Darrell, T. Curiosity-driven exploration by self-supervised prediction. In *ICML* (2017).

[29] Rajamaki, J., Hamalainen, P., Kyrki, V., and Korkeakoulu, P. *Random Search Algorithms for Optimal Control*. PhD thesis, Aalto University, 2012.

[30] Rumelhart, D. E., Durbin, R., Golden, R., and Chauvin, Y. Backpropagation. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1995, ch. Backpropagation: The Basic Theory, pp. 1–34.

[31] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Neurocomputing: Foundations of research. MIT Press, Cambridge, MA,

USA, 1988, ch. Learning Representations by Back-propagating Errors, pp. 696–699.

[32] Schroff, F., Kalenichenko, D., and Philbin, J. Facenet: A unified embedding for face recognition and clustering. *CoRR abs/1503.03832* (2015).

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

[34] Seide, F., and Agarwal, A. Cntk: Microsoft's open-source deep-learning toolkit. In *KDD* (2016).

[35] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR abs/1712.01815* (2017).

[36] Such, F. P., Madhavan, V., Conti, E., Lehman, J., Stanley, K. O., and Clune, J. Deep neuroevolution: Genetic algorithms are a competitive alternative for training deep neural networks for reinforcement learning. *CoRR abs/1712.06567* (2017).

[37] Sutton, R. S., and Barto, A. G. *Introduction to Reinforcement Learning*, 1st ed. MIT Press, Cambridge, MA, USA, 1998.

[38] Sutton, R. S., McAllester, D., Singh, S., and Mansour, Y. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Neural Information Processing Systems* (Cambridge, MA, USA, 1999), NIPS'99, MIT Press, pp. 1057–1063.

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

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