5-2018

Training Machine Learning Agents in a 3D Game Engine

Diego Calderon
University of Arkansas, Fayetteville

Training Machine Learning Agents in a 3D Game Engine

Diego Calderon

4/9/2018

Abstract

Artificial intelligence (AI) and video games benefit from each other. Games provide a challenging domain for testing learning algorithms, and AI provides a framework to designing and implementing intelligent behavior, which reinforces meaningful play. Medium and small studios, and independent game developers, have limited resources to design, implement, and maintain agents with reactive behavior. In this research, we trained agents using machine learning (ML), aiming to find an alternative to expensive traditional algorithms for intelligent behavior used in video games. We use Unity [14] as a game engine to implement the environments and TensorFlow [13] for the neural network training.

1. Introduction

AI is the science and engineering of making intelligent machines. AI is a very broad area of study addressing a wide range of topics; which include planning, learning, natural language processing, perception, and the ability to manipulate objects (see figure 2). As you may suspect, the effort of advancing knowledge in AI is

multidisciplinary, attracting talent from computer science, mathematics, neuroscience, and many other fields. Even though AI does not yet present itself with characteristics of human intelligence, there are technologies that perform tasks better than humans. These tasks are under the concept of "narrow AI," which only have one narrow task. A good example of narrow AI is the task of image classification on a service like face recognition of Facebook. These technologies exhibit similarities of human intelligence, which come from ML [5].

One important field of study within AI is the development of intelligent agents. Any device that perceives its environment and can make decisions and act in a rational way (which maximizes its chances of success at a defined goal) is an intelligent agent. The diagram below shows how an agent perceives its environment, assesses its current state, evaluates the consequence of its actions, makes a decision, and finally, takes an action that will affect the environment. This process is repeated until a goal is reached.

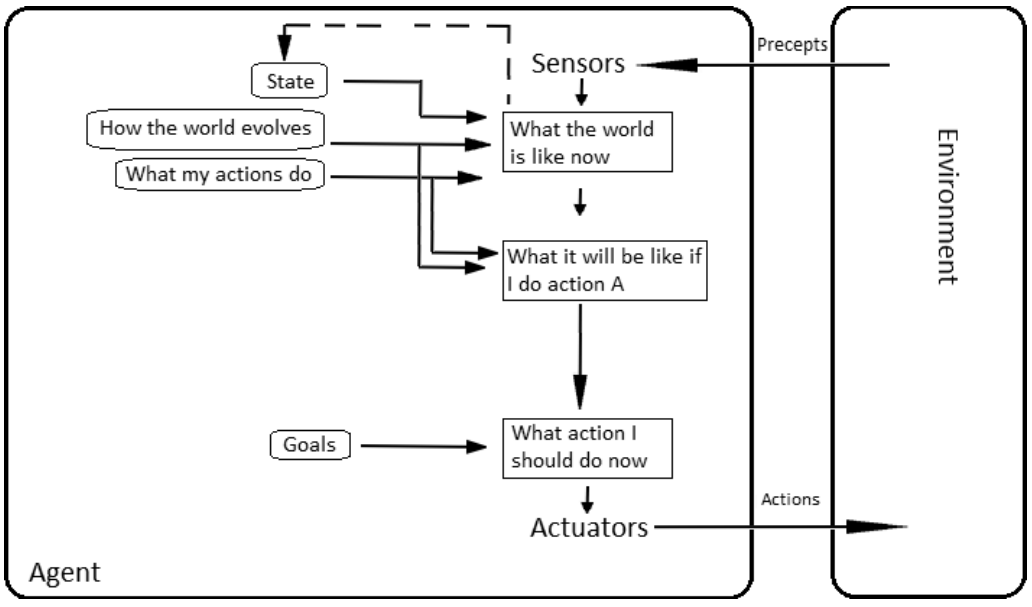


Figure 1: Model and goal based agent. Originally from Stuart J. Russell and Peter Norvig, Artificial Intelligence: A Modern Approach p.52

1.1 Machine Learning

ML is a subfield of AI that aims to provide machines with the ability to learn without explicit programming. Conceptually, the goal of ML is to automatically discover the mapping function that takes a set of inputs A and calculates output B. This could mean given an input, we want to predict its output. For example, given a loan application, we want to know if it will be accepted. Basically, ML is the practice of using algorithms to parse data, learn from it, and predict a new outcome given access to new data. This means that instead of having a static hand-coded set of instructions with a specific purpose, the machine is trained with a large dataset and algorithms that result in the ability to perform certain tasks. Most ML problems fall into one of two categories: supervised or unsupervised learning [5]. The example of mapping set A to B falls into the supervised learning domain. In this setting we wish to fit a model that relates observations to responses, with the purpose of accurately predicting the response of observations not seen before. On the other hand, unsupervised learning describes the scenario in which for every observation $i = 1, \dots, n,$ we observe a vector of measurements x_i, but no associated response y_i. In other words, we cannot supervise the analysis because we lack response variables.

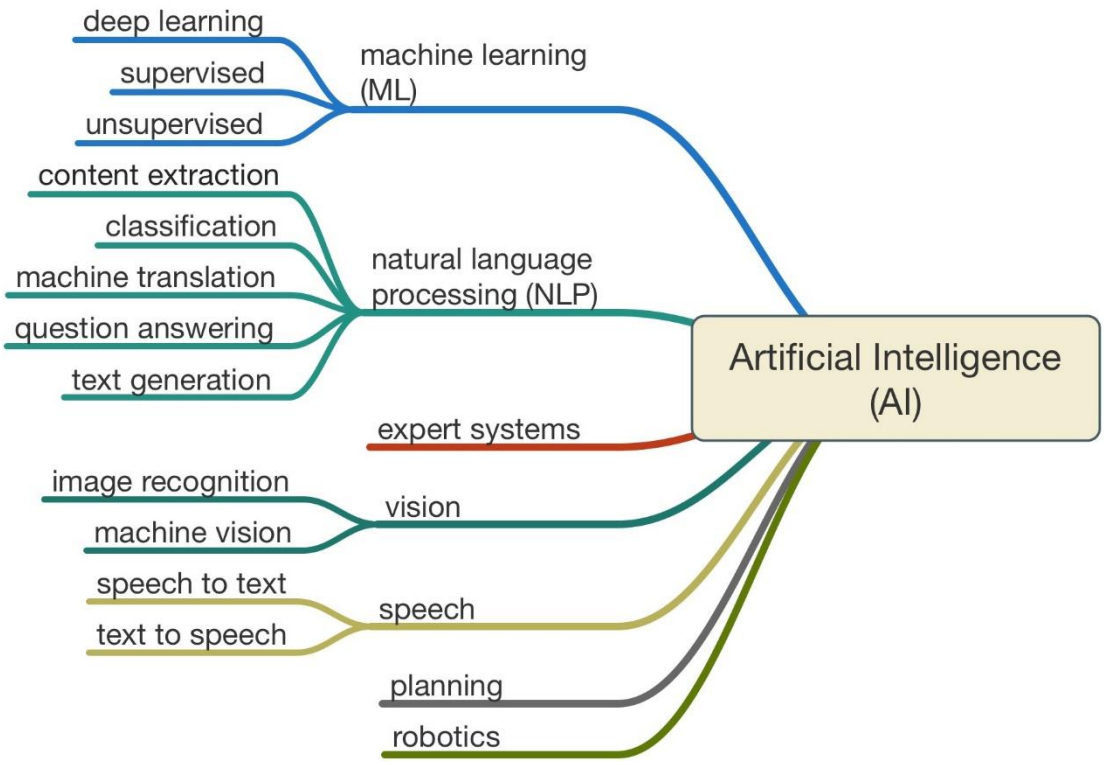


Figure 2 Different branches of AI.

Retrieved from https://mse238blog.stanford.edu/2017/08/jgokani/the-evolution-of-banking-ai/

1.2 Artificial Neural Networks

A popular technique to build programs that learn from data is using artificial neural networks. At its most basic level, a neural network is inspired by our understanding of the brain [7]. To implement it, a collection of neurons is created and connected (allowing communication between them). Mathematically speaking, the simplest artificial neuron takes binary inputs, and produces a single binary output (note that artificial neurons also compute continuous inputs and outputs). In this model, the concept of weights was introduced to compute the output. These weights, $w_1, w_2, \ldots, w_n$, are real numbers that represent the importance of the respective inputs to the output. The neuron's output is determined by the following algebraic term:

$$output = \begin{cases} 0 & if \sum_j w_j x_j \leq treshold \\ 1 & otherwise \end{cases}$$

To illustrate this concept, let us consider a toy example. Imagine we have a non-player character in a video game whose objective is to attack the player. The agent might make decisions by weighting up three factors: Is the agent's life low? Is the player's life low? Is there an ally near? If the agent's life is low, the player's life is not low, and there are no allies nearby, our input would look like $x_1 = 1, x_2 = 0, \ and \ x_3 = 0$. Now, let us suppose that we greatly value whether the player's life is low, we moderately value the agent's life, and we do not care if there are nearby allies. In this case, the decision to attack or not could be represented as $w_1 = 3, w_2 = 6, and \ w_3 = 1$. Note that other weights would produce different attack behaviors.

1.3 Reinforcement Learning

Furthermore, a successful technique for ML is reinforcement learning (learning from rewards or punishments). Reinforcement learning involves an agent, a set of states, and a set of actions [11]. By performing actions, the agents transitions between states. This method allows agents to learn how to achieve successful strategies (a sequence of actions) that lead to the greatest utility in the long term (they seek to maximize long-term rewards) without any hand-coded features. These agents are now beating humans at playing Atari games [6], and they even beat champions at the strategy board game Go. DeepMind presented a successful deep learning model that learns control policies from high-dimensional input using reinforcement learning. The authors used this method to play seven Atari 2600 games using the Arcade Learning

Environment (without adjusting the architecture or learning algorithm to play the different games).

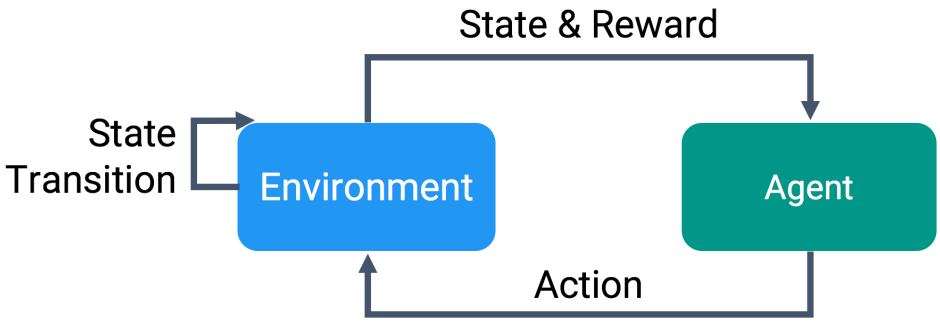


Figure 3:Reinforcement Learning training cycle. Retrieved from https://blogs.unity3d.com/2017/09/19/introducing-unity-machine-learning-agents/

In other words, their agent interacts with an environment (the Atari emulator) in a sequence of actions, observations, and rewards. At each step, the agent selects a valid action, which modifies the game's internal state and score [15]. A simple example of a reinforcement learning task is playing Pong. The agent controls a paddle in a vertical motion (up and down). The objective is to bounce a ball in the paddle and make the opponent miss the ball. For this task, the agent receives a byte array with pixel values (a 210x160x3 byte array with integer values ranging [0,255]), which represents the current information on the computer screen, and learns to decide whether to move the paddle up or down [6]. After each decision, the simulator executes the action and rewards the agent: 1 if the opponent missed the ball, -1 if the agent missed the ball, and 0 otherwise. What makes reinforcement learning even more challenging is that we do not have the correct label for the input (we do not know what is the right action given the game state beforehand). In the supervised learning task, we train over known data, mapping input to output, and then testing it with unseen data. A common reinforcement

technique is Q-learning, which does not require a model of the environment [11]. Recall that the reinforcement learning tasks consists of an agent, a set of states $s \in S$, and a set of actions $a \in A$. On a high level, the Q-learning algorithm performs the following computation:

- Perform action $a\ in\ s$
- Receive consequences of the actions performed: s'
- Compute an evaluation of s': $V(s')$
- Update crossbar value $W'(a,s) = W(a,s) + V(s')$.

Another popular reinforcement learning approach (moving away from Q-learning) is using Policy Gradients [10]. On a high level, policy gradients make good actions more likely and bad actions less likely. We can think of it as a formalization of trial and error: a stochastic policy that samples actions and then actions that lead to good outcomes are encouraged in the future, and actions that lead to bad outcomes are discouraged. Open AI [9] recently proposed a new family of policy gradient methods for reinforcement learning, which alternates between sampling data through interaction with an environment, and optimizing an objective function using stochastic gradient ascent. We will use this method to train our agents.

The fact that there are algorithms that allows agents to learn how to effectively interact with its environment is exciting, especially for video games. However, AI in games is not only a matter of developing good algorithms, but also a matter of good design. We do not want AI to frustrate the player, but to elevate gameplay. Every system should address AI in a different manner. AI in games should be predictable, but not easy to beat. Furthermore, it should be designed in a manner that allows players to

find behavioral patterns, which would allow them to devise their own meaningful goals (through their understanding of the game mechanics), and to formulate meaningful plans. In practice, it is common for video games to use scripted computer opponents. This means that the designers must build a behavior tree (BT), which describes the switching between a finite set of tasks in a modular manner. Even though BT can create complex tasks composed of simple ones, they can grow very large. As you can see in Figure 4, the complexity of a BT increases with each different action. This means that if a designer, down the road, decides that it would be appropriate to add a new action the agent needs to perform, the BT must be modified almost entirely, which is a large investment of resources.

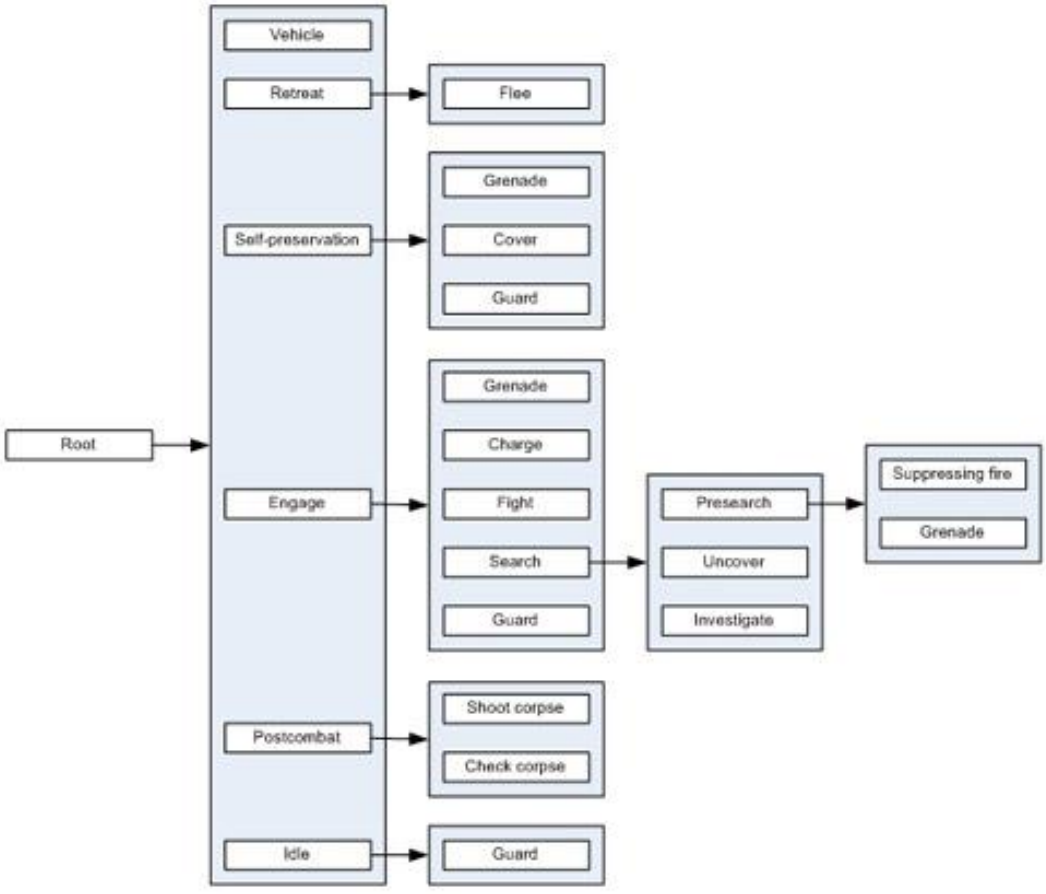


Figure 4 Handling Complexity in the Halo 2 AI, GDC 2005. Retrieved from https://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php

1.3 Game Engine

Unity is a cross-platform game engine which supports the development of 3D and 2D video games and simulations with drag-and-drop functionality. Originally, Unity supported scripting using C# and Javascript, but with their latest release they have started deprecating Javascript. Due to recent advances in graphics hardware, the Unity Engine is evolving into a complete 3D modeling tool, able to generate high-quality simulations; which caught the attention of both academia and industry [4].

Unity recently released a toolkit for training ML agents. This tool allows developers and researchers train and test AI algorithms in a 3D world. We are particularly interested in using Unity to train agents because of the simplicity in generating games (a well implemented API and a large, and collaborative, community). As we mentioned before, with the rise of processing power, simulated environments are becoming a more common testing platform across disciplines. Simulated 3D environments allow researchers to test their ideas in a safe manner. Unity's tool allows the training of agents using TensorFlow and to export them directly to the environment (to visualize results, analyze behaviors, and test the agents). Within the tool, the communication between the game's environment and the library for numerical computation is displayed in Figure 5.

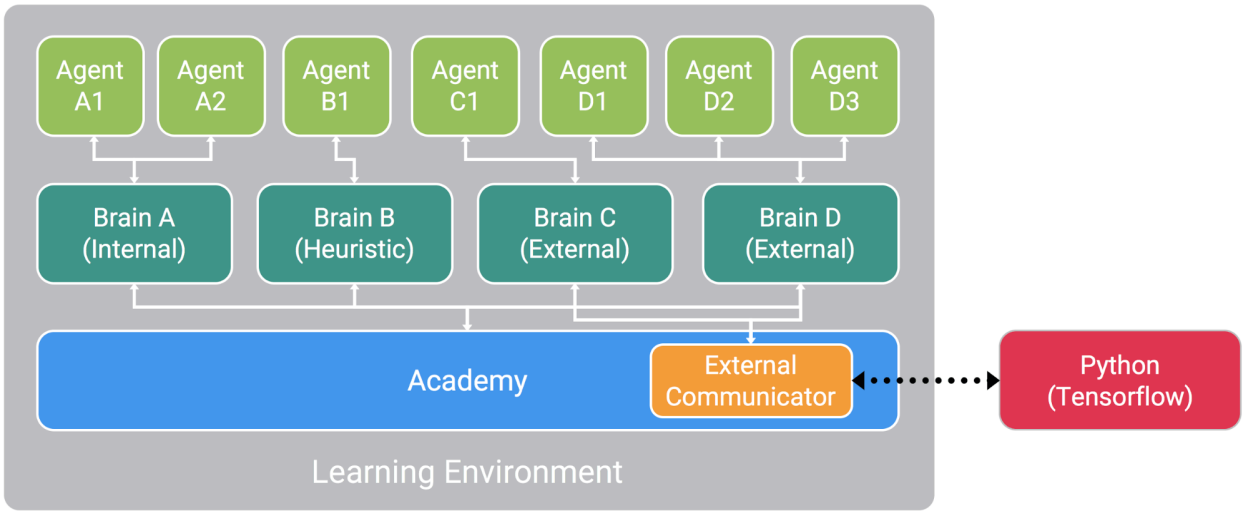


Figure 5:Unity's learning environment. Retrieved from https://blogs.unity3d.com/2017/09/19/introducing-unity-machine-learning-agents/

1.3.1 ML-Agents Key Components

Following ML-Agents' documentation [14], the plugin contains three high-level components:

- **Learning Environment:** contains the Unity scene (3D or 2D environment), and all characters and agents.
- **Python API:** contains all the ML algorithms used for training (the API is not part of Unity, and needs an external communicator).
- **External Communicator:** connects the Learning Environment and the Python API. It lives inside the Learning Environment.

With this plugin, we can train agents using a variety of methods.Basically, we define three entities at every moment of the environment:

- **Observations:** what the agent perceives of the environment, which can be visual and/or numeric. Observations can be discrete or continuous, depending on the

complexity of the environment-- typically, more complex environments require numerous continuous observations.

- **Actions:** what actions the agent can take. Actions can also be discrete or continuous numerical values, depending on the complexity of the environment and the agent. For example, if the actions of the agent depend on more than one factor, for example speed and direction, the agent might benefit from having continuous actions.
- **Reward Signals:** a scalar value that indicates how well an agent is doing in its environment. Recall that the reward signals is how the objectives of the task are communicated to the agent. Therefore, the reward system must be set up such that maximizing the reward leads to an optimal behavior.

Using these entities, we can train the agent's behavior. Using the notation of reinforcement learning, the behavior an agent learns is called a policy, which is an optimal mapping between observations and actions. Furthermore, the ML-Agents plugin contains three additional components to help organize the scene:

- **Brain:** encapsulates the logic for making decisions for the Agent. It is the component that receives the observations and rewards from the Agent and returns an action.
- **Agent:** handles generating observations, performing the actions it receives from the brain, and assigning a reward when appropriate. Each Agent is linked to exactly one Brain.

- **Academy:** orchestrates the observation and decision-making process. Within the Academy, several environment-wide parameters such as the rendering quality can be specified. The External Communicator lives within the Academy.

To summarize, each agent must be linked to a brain, and it is possible to have multiple agents linked to a single brain. Note that the brain defines the space of all possible actions and observations, and the agents that are connected to a specific brain can have unique observation and action values. In other words, the brain receives observations and rewards from the agents and returns actions. Each learning environment will have a global academy, which ensures that all the agents and brains are synchronized and controls environment-wide settings.

2. Approach

The first step is to download and install Unity 2017.1. There are several options for downloading it, but for using for using Unity's ml-agents, the free personal version suffices. To access the downloading site, follow the link: https://unity3d.com/get-unity/download. There are several tutorials online on how to get started with Unity. A step-by-step guide to getting started can be found here: https://unity3d.com/learn. Since Unity's ML-agents is constantly evolving, I would suggest checking their documentation here: https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Readme.md. This work was implemented using ML-Agents V0.1, and currently, they released the third version, which is not compatible with earlier versions of Unity and ML-Agents. The newer versions introduced new learning algorithms, such as curriculum and imitation learning, which we did not use to train our agents.

Also, it is good to note that the ML-Agents developing team constantly update their documentation, making it easier for developers to interface with their code. Once Unity 2017.1 is installed, we need to install TensorFlow. On 2015, Google released an open source software library called TensorFlow, which is used for ML and Deep Learning for numerical computations. TensorFlow can run on multiple CPUs and GPUs, and since we will work with large datasets, it will be beneficial to run these algorithms on CUDA enabled Nvidia GPUs to achieve faster execution. We used Windows 10, therefore, we will explain the installation process. The first step is to check if your Nvidia GPU is CUDA compatible, which you can check here: https://developer.nvidia.com/cuda-gpus. If your GPU is compatible, the next step is to set it up. First, you need to install Cuda Toolkit 8.0 and cuDNN v5.1. You can download the toolkit from https://developer.nvidia.com/cuda-downloads, and the cuDNN library for Windows 10 from https://developer.nvidia.com/cudnn.

For a graphical explanation of how to install those components, you can follow Nitish Mutha's guide [8]. Once the toolkit and library are installed, you need to install Python 3: download and install Anaconda for Windows. For the V0.1 of ML-Agents, you could use Python 2 or Phyton 3. Note that for newer versions, Python 2 is no longer supported. Once Anaconda is installed, you need to activate a new conda environment, and install all the dependencies. The most straight forward way of installing dependencies is to clone the following repository: git clone git@github.com:Unity-Technologies/ml-agents.git, navigate to the directory in anaconda prompt, and then run the command "pip install ." Once those requirements are installed, run the command

“pip install tensorflow-gpu.” Now we are ready to train agents in Unity! For a deeper insight into Unity, refer to [14].

3. Learning Environment Evaluation

The goal of this research is to demonstrate the benefits of using ML to create intelligent behavior in video games. To demonstrate the benefits of training intelligent agents using ML techniques, we designed, implemented, and evaluated 2D environments (with simple tasks) using Unity as a game engine. We will describe them in detail in the following subsections.

3.1 3D Ball

To familiarize ourselves with this new tool, we followed Unity’s example environment “3D Ball.” The task is to balance a ball in a platform, and keep it on as long as possible. To summarize (using Unity’s description), the task is as follows:

- Set-up: Balance a ball – the agent controls the platform.
- Goal: Keep the ball on the platform as long as possible.
- Agents: The environment consists of 12 agents linked to a single brain.
- Agent reward function:
    - +0.1 for every step that the ball remains on the platform.
    - -1.0 if the ball falls.
- Brains: One brain
- State space: Continuous – 8 variables corresponding to: rotation of platform, and position, rotation, and velocity of the ball.
- Action space: Continuous – 2 possible actions (rotate x axis, and rotate z axis).
- Observations: 0.

- Reset parameters: None.

After two hours of training the agents with these settings, we were able to successfully train an agent to balance the ball. What is unique to this tool is that once we trained the agent, we were able to modify the environment and see how the trained agents reacted. For example, we made a platform rotate constantly in the y axis (adding more challenge to the task). We also made modifications to add random forces to the balls. This showed us how the agents reacted to “unexpected” events that did not happen during training.

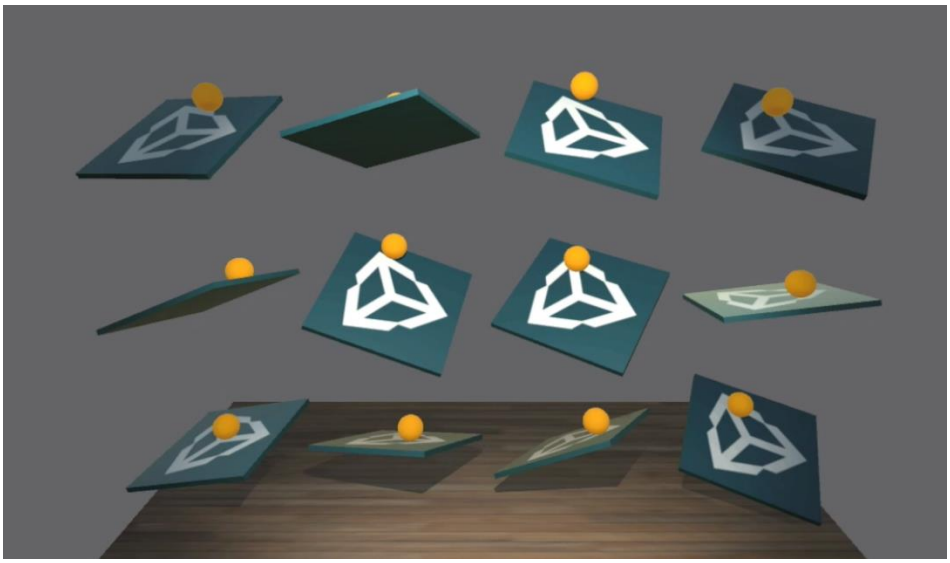


Figure 6: Agents linked to one brain balancing a ball

3.2 Linear Movement

After training the example environment (where the observations and actions were given to us), we decided to set a simple environment to test the features of the platform. The task is to linearly move a character to reach a goal. The setting is as follows:

- Set-up: Move left or right.
- Goal: Reach the goal.

- Agents: The environment consists of 1 agent linked to a single brain.
- Agent reward function:
    - -0.1 for every action.
    - -1 for going out of bounds.
    - +1 for arriving at the goal.
- Brains: One brain
- State space: Continuous – 2 variables corresponding to the position of goal, and the agent's position. Our setting ranges from 0 to 100 in the x coordinate. The position of the goal and agent is generated randomly each simulation.
- Action space: Discrete – 2 possible actions (move left, and move right). The agents move at a constant speed, with no acceleration.
- Observations: 0.
- Reset parameters: Randomly place the goal and agent.

Figure 7:Simple task of reaching goal

After 20 minutes of training, the trained agent was able to move towards the goal every time. With more training time, the agent would move to the target faster. Early versions of the trained agents would alternate between moving left and right, but would eventually reach the target. Even in those early versions, the agent would avoid going out of bounds.

3.3 Linear movement with obstacles

To increase the challenge of the previous approach, we decided to add an obstacle between the agent and the goal. In this process, we learned that our locomotion implementation (and the way we handled moving from left to right or vice versa) had conflicts with the obstacle's collider. To solve this problem, we changed the implementation to negatively reward the agent if it collided with the obstacle. The agent's task is as follows:

- Set-up: Move left, right, do nothing, and jump.
- Goal: Reach an objective.
- Agents: The environment consists of 1 agent linked to a single brain.
- Agent reward function:
    - -0.1 for locomotion.
    - -1 for colliding with an obstacle or going out of bounds.
    - +1 for reaching an objective.
- Brains: One brain
- State space: Continuous – 5 variables corresponding to the position of the agent, the position of the goal, the distance between the agent and the goal, the value of a raycast, and 1 if the raycast hit an obstacle (0 otherwise).

- Action space: Discrete – 4 possible actions (move left, right, jump, and do nothing).
- Observations: Local ray-cast perception.
- Reset parameters: Randomly place the goal, the obstacle, and agent.

Figure 8 Agent avoiding obstacles by jumping

First, we designed and developed the environment, defined possible actions of characters, set the brain to external, exported the environment created with Unity, and trained the agents using TensorFlow. With this approach, after 2 hours of training, our agent learned to jump obstacles when they are in the way of the goal. We added more obstacles in different locations to test the agent, and it successfully overcame most of the settings. The most significant problem we had with this approach was the rotating of the sprites when the agent changed directions. When the agent was close to the obstacle's collider, and it moved right and left rapidly, the colliders would overlap and it would make the agent float (disconnecting the actions from the consequences and rewards). Once we fixed that problem, the training process was smooth.

3.4 Combat environment

In our combat environment we designed and implemented two agents: a range attacker (an archer), and a melee attacker (in our setting, a ninja). We wanted the archer to shoot arrows (which follow a parabolic trajectory) to enemies, and the ninja to attack with a sword to enemies. The idea was to use these two agents to train against each other. Our first approach was to use a deep neural network to learn features from raw pixels to allow each agent to decide on what actions to take. First, we designed and developed an environment, defined possible actions of characters, exported the environment created with Unity, and trained the agents using TensorFlow. The main problem was that Unity's engine physics system and the agent's step (their actions) had a different update cycle, resulting in the inability of properly training the agents. Furthermore, each made decisions at a constant rate, disregarding whether the action had an immediate consequence or not. Additionally, we learned that learning from raw pixels is too expensive, and the agents can learn accurately with other parameters (such as raycasts, and information about the position of objects).

Archer's task:

- Set-up: Throw arrows, setting a force, and angle.
- Goal: Hit an objective.
- Agents: The environment consists of 1 agent linked to a single brain.
- Agent reward function:
    - +0.1 for setting and angle or force.
    - +0.2 for shooting an arrow.
    - -0.2 for missing an objective.
    - +1.0 for hitting the objective.

    - -0.01 for doing nothing.
- Brains: One brain
- State space: Continuous – 3 variables corresponding to the angle, force, and position of the objective.
- Action space: Discrete – 4 possible actions (set an angle, set a force, do nothing, and shoot an arrow).
- Observations: 1 corresponding to the game view.
- Reset parameters: Randomly place the goal and agent.

Ninja's task:

- Set-up: Move left or right, and attack.
- Goal: Kill an objective.
- Agents: The environment consists of 1 agent linked to a single brain.
- Agent reward function:
    - -0.1 for locomotion.
    - +0.2 for attacking and hitting an objective.
    - -0.2 for attacking and missing an objective.
    - +1.0 for hitting the objective and killing it.
    - -0.01 for doing nothing.
- Brains: One brain
- State space: Continuous – 4 variables corresponding to the agent's health, x position, opponent's health, and opponent's x position.
- Action space: Discrete – 4 possible actions (move left, right, attack, and do nothing).

- Observations: 1 corresponding to the game view.
- Reset parameters: Randomly place the goal and agent.

Figure 9: Archer and ninja set-up

After several hours of training, the archer agent was not able to learn the behavior we desired. It would eventually hit a target, but an agent that picks angles at random, and shoots at random moments would achieve the same results as the trained agents. As we mentioned before, we believe that training the archer was not as we expected because we could not reward the shooting arrow action properly. Between the frame when the agent shoots the arrow, to when the arrow hits or miss, up to several hundred frames may pass, and with each frame, an action is performed. Unity's team acknowledged this shortcoming, and said that they will address it in the future (allowing more control on when actions are performed).

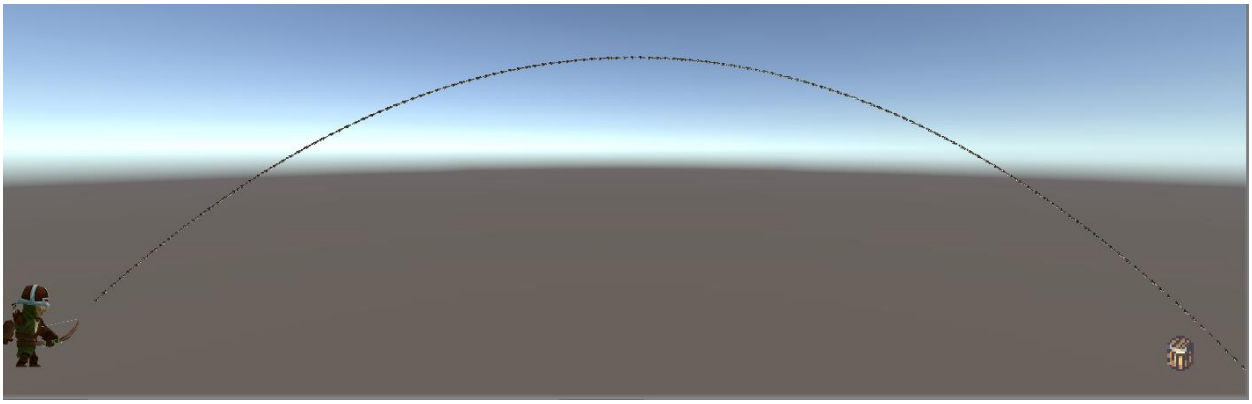


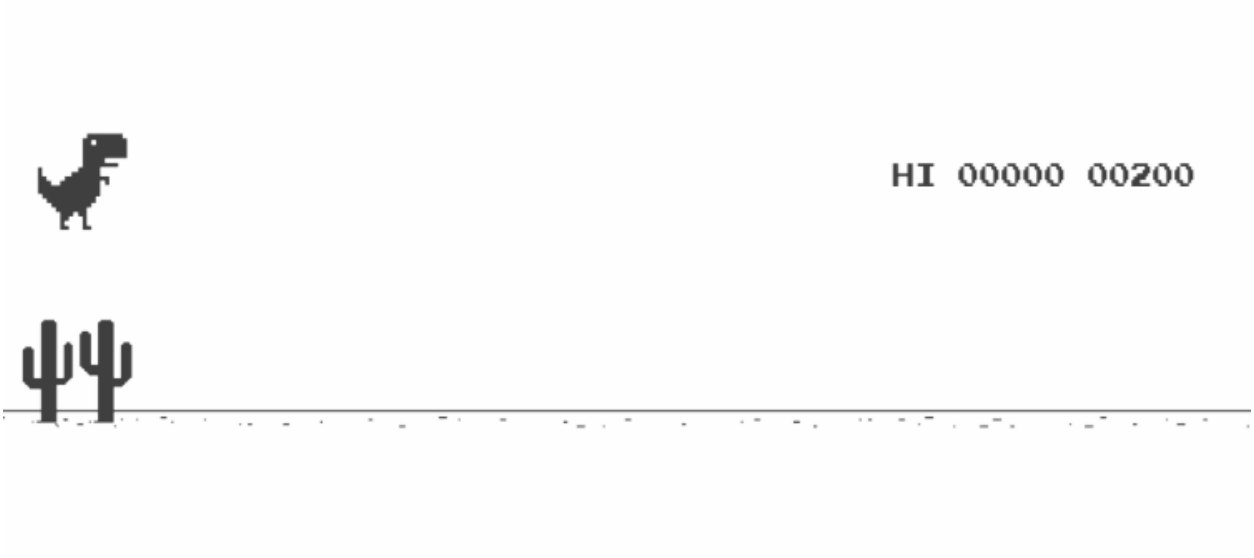
Figure 10 Trajectory of an arrow

3.5 Google's T-Rex Game

We also trained an agent to play Google Chrome's dinosaur game [12]. The purpose of this agent is to test if an agent is able to learn how to play a game that is based on a highest score system (meaning that there is one level with increased difficulty as time passes, but with no concrete end). Besides learning that it is possible to train agents to play these type of games, we realized that using agents helped us note shortcomings of our own environments. We learned if there is "a hack" in our game (a shortcut that the designers did not plan), the agent will find it, and exploit it to get to the end-game reward faster. The dinosaur agent task is as follows:

- Set-up: Jump, duck, or do nothing.
- Goal: Accumulate points and beat the high score.
- Agents: The environment consists of 1 agent linked to a single brain.
- Agent reward function:
    - +0.1 for every action.
    - -1 for dying.
- Brains: One brain

- State space: Continuous – 15 variables corresponding to horizontal and vertical raycasts, and the speed at which the obstacles approach the dinosaur.
- Action space: Discrete – 3 possible actions (jump, duck, and do nothing).
- Observations: Local ray-cast perception.
- Reset parameters: none.



We had several approaches to training the dinosaur agent. First, we used the main camera to capture the screen and train over the pixels. We added the level's speed as a state. One error on this approach was that we were not normalizing the reward (which was based on the score). Using this implementation, we had an agent that reached a maximum score of 159. There was no significant improvement with more training time, so we acknowledged that it was a shortcoming of our reward and state system. We refactored the way the agent perceived the environment. We got rid of the observations (we no longer used the camera), and added horizontal and vertical raycasts. We kept the level's speed, and normalized the reward system. With this new approach, after 20 minutes of training we reached a high-score of 330. We kept training for over 12 hours,

and we saw that the mean reward would fluctuate from 4 (which is very low for our reward system) to 600. After analyzing the agent's behavior, we realized that at higher level speeds, the agent did not have time to react. We then realized that the length of the horizontal raycasts were not long enough. Unfortunately, training the agent with the extended raycast lengths did not improve the agent's highest score.

4. Conclusions

Unity's effort of facilitating the training of agents in a 3D environment is promising. They have been constant in their iterations, fixing bugs rapidly and responding to the community's inquiries. The version this project is based is deprecated and I would advise waiting for the end of the tool's beta phase to invest time in development. Most of the main issues were solved in later versions of the tool, and those that remain are going to be addressed in future versions. Unity made a contest to incentivize developers to use their tools, for which they just announced a winner. The project, "Pancake bot," consists of an agent that balances a pancake in a pan, throws it to a plate, and a robot takes butter to the pancake. The winners used the same concept as we did for moving the agent through obstacles (using raycasts to detect obstacles). For the balancing and throwing of the pancake, they used Curriculum Learning [2]. It is exciting to see that Unity is investing resources in improving these tools, and I look forward to seeing agents trained within Unity and used in games. I believe that in order to develop agents that are feasible in games, agents need a mix between machine learned behavior (throw some imitation learning [1] in there, too), heuristics, and hard coded behavior that is too hard to train.

5. References

[1] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. 2017. Imitation Learning: A Survey of Learning Methods. ACM Comput. Surv. 50, 2, Article 21 (April 2017), 35 pages. DOI: https://doi.org/10.1145/3054912/. [Accessed: 15-Apr-2018]

[2] Alex, Bellemare, M. G., Menick, Jacob, Munos, Remi, and Koray, "Automated Curriculum Learning for Neural Networks," *[1704.03003] Automated Curriculum Learning for Neural Networks*, 10-Apr-2017. [Online]. Available: https://arxiv.org/abs/1704.03003. [Accessed: 15-Apr-2018].

[3] "Deep Reinforcement Learning," *DeepMind*. [Online]. Available: https://deepmind.com/blog/deep-reinforcement-learning/. [Accessed: 15-Apr-2018].

[4] "Designing safer cities through simulations – Unity Blog," *Unity Technologies Blog*. [Online]. Available: https://blogs.unity3d.com/2018/01/23/designing-safer-cities-through-simulations. [Accessed: 16-Apr-2018].

[5] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning: with applications in R*. New York: Springer, 2013.

[6] Karpathy, Andrej. *Deep Reinforcement Learning: Pong from Pixels*. [Online]. Available: http://karpathy.github.io/2016/05/31/rl/. [Accessed: 15-Apr-2018].

[7] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, (2015).

[8] Mutha, Nitish, "Install TensorFlow with GPU for Windows 10." [Online]. Available: http://blog.nitishmutha.com/tensorflow/2017/01/22/TensorFlow-with-gpu-for-windows.html. [Accessed: 15-Apr-2018].

[9] "OpenAI," *OpenAI*. [Online]. Available: https://openai.com/. [Accessed: 18-Apr-2018].

[10] Schulman, John, Wolski, Filip, Prafulla, Radford, Alec, Klimov, and Oleg, "Proximal Policy Optimization Algorithms," *[1707.06347] Proximal Policy Optimization Algorithms*, 28-Aug-2017. [Online]. Available: https://arxiv.org/abs/1707.06347. [Accessed: 18-Apr-2018].

[11] *Reinforcement Learning Repository at University of Massachusetts, Amherst*. [Online]. Available: http://www-all.cs.umass.edu/rlr/. [Accessed: 15-Apr-2018].

[12] T-Rex Google game Unity project, "saiichihashimoto/dinosaur-game," *GitHub*. [Online]. Available: https://github.com/saiichihashimoto/dinosaur-game. [Accessed: 16-Apr-2018].

[13] "TensorFlow," *TensorFlow*. [Online]. Available: https://www.tensorflow.org/. [Accessed: 16-Apr-2018].

[14] Unity-Technologies, “Unity-Technologies/ml-agents,” *GitHub*. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Background-Unity.md. [Accessed: 15-Apr-2018].

[15] Volodymyr, Koray, David, Alex, Ioannis, Daan, Riedmiller, and Martin, “Playing Atari with Deep Reinforcement Learning,” *[1312.5602] Playing Atari with Deep Reinforcement Learning*, 19-Dec-2013. [Online]. Available: https://arxiv.org/abs/1312.5602. [Accessed: 15-Apr-2018].