UNIVERSITY OF BUCHAREST
FACULTY OF MATHEMATICS AND INFORMATICS
DISSERTATION DOMAIN: DISTRIBUTED SYSTEMS

DISSERTATION WORK

A Study of Reinforcement Learning for 3D Games

SCIENTIFIC COORDINATOR
Păduraru Ionuț Ciprian
STUDENT
Roșcăneanu George

BUCHAREST
June 2019

Table of Contents

Introduction

Motivation

Some of the main hobbies in my life are programming and playing computer games. Since I started to understand the basics of programming, I was wondering how the features in games are being done.

In the days when I was playing on my Nintendo Entertainment System console, I saw how the characters in the screen would be moving, how they reacted when I pressed the buttons on the controller, how the points on the screen are affected and how the enemy characters are moving. Back in those days, the behavior of the enemies were very simple. Most of the time they moved all the way to one side, hit an obstacle, then start walking to the other side.

Back then, that simple strategy was enough to keep players on the edge. Eventually better technology was available, stronger hardware was cheaper, and the demands of the players grew.

The characters in video games slowly became smarter and smarter. They are now interacting within a 3D environment, are actively searching for the player and becoming more and more life-like.

Racing games would have the enemy drivers trying to surpass you, to be faster then you, but it has to do all sorts of complex calculations. How to get ahead of you, without necessarily colliding with you or the walls, while moving at relatively high speeds.

Real time strategy games such as StarCraft II give the possibility to human players to fight against one or more enemy commanders controlled by the computer. I personally do not even begin to understand the size and complexity of the code that was written for the enemy to be able to build entire bases and command their armies.

Simple tasks and routines for the enemy AIs do not take a significant effort to be written. Writing the code for a competent AI that is able to challenge even the best humans at complicated tasks is incredibly difficult. Thus I started searching for other ways of writing these AIs.

One thing that I found was how machine learning is used to train AIs that became so good they are surpassing humans.

A simple Google search for the term “ai defeats human” will reveal several interesting websites:

- **Google AI defeats human Go champion**
  “Google's DeepMind AlphaGo artificial intelligence has defeated the world's number one Go player Ke Jie.” (2017 May 25, https://www.bbc.com/news/technology-40042581)
- **DeepMind AI Beats Professional Human StarCraft II Players**
  “The Google-owned artificial intelligence lab announced on Thursday that its new "AlphaStar" AI had beaten two of the world's best StarCraft II players.”

(2019 January 25, https://www.forbes.com/sites/samshead/2019/01/25/deepmind-ai-beats-professional-human-starcraft-ii-players/#32d881077cec)

- **AI trained on 3500 years of games finally beats humans at Dota 2**
  "They say 10,000 hours makes an expert, but for video-game playing AIs much more is needed. After playing thousands of years' worth of the video game Dota 2, artificial intelligence is now able to beat the world's top amateurs."
  (2018 June 25, https://www.newscientist.com/article/2172612-ai-trained-on-3500-years-of-games-finally-beats-humans-at-dota-2/)

This has shown me that machine learning is really creating some incredible AIs, so good that even the humans that trained them could not create a better code by hand to beat the respective AIs.

But one more hurdle had to be surpassed, I do not have experience with machine learning, perceptrons, neural networks, reinforcement learning and so on. Trying to learn all such techniques and algorithms would be too complicated and take too much time to implement them correctly. Even then, when I will try to train whatever AI, I would not even know if the implementation I have done would be correct or not, as training them can take anywhere from minutes to days.

Here comes ML-Agents, a plugin for Unity that lets developers and hobbyists jump over the difficulty of implementing their own machine learning code and use the best implementations currently available in a popular and powerful machine learning library called TensorFlow.

No longer I have to worry about code correctness, time of training and possible hidden bugs. The respective plugin is hosted on GitHub, where everyone can see what is happening to the code base, able to report whenever they see a problem, and also able to improve it.

After copying and doing the setup for the plugin, I started trying all of the samples.

It contains several environment samples, with already trained machine agents that solve sample problems more efficiently then I could do by playing the games themselves, or by writing a hard-coded AI to solve the respective problems.

Illustration 1: Walker Agent screenshot

Some of the problems I cannot even begin to imagine how to solve them. For example, there is an environment with a ragdoll model where each of the important joints are motorized and controlled by the agent, in an attempt to reach a golden target somewhere far away. How does one even take into consideration all the angles, physics and calculations that have to be done for ragdoll to even just simply stay upright. Yet in the sample we can see the ragdoll running like a crazy cartoon character towards the target. Even though it looks ridiculous, it is still a ragdoll controlled only by it's joints, running decently fast while balancing itself!

A screenshot of the walker agent can be seen in Illustration 1.

After trying every sample, I started following the guide for creating a brand new Learning Environment (https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Create-New.md). I was beginning the grasp the idea how the scene should be organized, and how the scripts should look like. I observed how the training time was pretty slow, until I added the suggested training hyperparameters (batch_size: 10 buffer_size: 100) which indeed has reduced the number of training steps from 300000 to 20000 just like how the guide has said.

I was amazed at the significant change, it was at least 10 times faster!

Slowly, I began adding more complexity to the task, and experimented with the different hyperparameters, the reward system, with how the agent gets information from the environment and so on.

I documented here my findings, in hope people will use it to get a better idea on how the different values affect the training time and the performance of the agents.

Definitions and short history

Artificial Intelligence research has been focusing on teaching the machines to understand and recreate the human behavior, but with better performances. Back in the 2000s, the main challenges of AI were to fly helicopters and walk up flights of stairs, but during the past years, the list has become longer. Nowadays, people have the advantage of being capable of solving problems that are not well defined over the machines, which need to have first the information about the world in order to be able to learn how to solve the problems.

Inverse Reinforcement Learning (IRL) is related to these kinds of discrepancies and its solution might be a significant step forward towards of learning human behavior.

Reinforcement learning process requires to study the decision process in order to recreate the comportment that magnifies the predefined reward function. In the moment when the agent executes an action in a certain state, the reward function sends the results of the agent's achievement. The purpose of the reward function is to generate an ideal policy, where the future reward is most favorable.

On the other hand, Inverse Reinforcement Learning has another perspective over the problem and it tries to extract the reward function from the studied agent's behavior. In other words, based on the agent's behavior history, the reward function has to be identified, one which will explain the comportment. Under the hypothesis that the actions of the agent were optimal and it always chooses the best action for the outcome, the reward function can be estimated to indicated this behavior.

An example for this can be self driving cars. The fast forward approach could be designing the reward function based on the wanted behavior of the driver. This includes the following the driving rules, as red light stopping, staying away of people walking on the sidewalks etc. This approach has a lot of rules to be weighted and it would need to an extensive list of behaviors to be considered.

In the IRL strategy for the autonomous car, based on a data load of human driving data a similarity of the reward function for the task can be extracted. The approximation is dealing with a basic driving model, but a lot of information needed for figuring out the problem is gathered when calculating the reward function. This approach evaluates the actions, good or bad and in the moment when the correct reward function has been identified, the problem concentrates on discovering the right guideline.

In this example, the task is mentioned in the reward function, so it’s not required to have the exact aspects of the drivers policy as the correct reward function is optimizing. IRL algorithms can be defined as a technique to weight expert knowledge to transform a task specifications into a reward function.

The distinction that occurs in this situation when a complex task is converted in a basic reward function is that the behavior could become optimal for a lot of particular reward functions. Even the actions are taken from an expert, might happen that the expert is trying to maximize many reward functions. In order to figure out this, Andrew Y. Ng and Stuart Russel formulate IRL like an optimization problem (reference Algorithms for Inverse Reinforcement Learning https://ai.stanford.edu/~ang/papers/icml00-irl.pdf).

Furthermore, considering the constraint of choosing one reward function where the behavior is ideal, it must be possible to select a reward function that increases the main properties.

Inverse Reinforcement Learning problem can be described as follows based on Stuart’s Russell, 1998 definition:

“Given

1) measurements of an agent’s behavior over time, in a variety of circumstances;

2) if needed measurements of the sensory inputs to that agent;

3) if available a model of environment

Determine the reward function being optimized”

Used tools

About Unity

Unity is a game engine developed by Unity Technologies. This tool gave users the ability to create games and other applications, in both 2D and 3D, for a wide selection of platforms (Windows, iOS, Android, macOS, Linux, PlayStation 4, Xbox One and many others).

Games are not the only things made with Unity. It is also used in the automotive industry to render aesthetically pleasing car models, in the animation industry for producing high quality animations while still permitting lots of artistic freedom, and also for creating educational applications and VR/AR experiences.

Here is a short list of popular games that were made with Unity: Cuphead, Monument Valley 2, Rick and Morty: Virtual Rick-ality, Inside, Ori and the Blind Forest, Hearthstone and Cities: Skylines.

The main programming language used in the game engine is C#, which is converted to C++ on platforms such as Windows by using a scripting back-end called IL2CPP.

About machine learning

Machine learning (ML) is the scientific study of algorithms and statistical models that computer systems use to effectively perform a specific task without using explicit instructions, relying on patterns and inference instead. It is seen as a subset of artificial intelligence.

This type of algorithms increases the accuracy of the software applications when predicting outcomes.

About ML-Agents

The Unity Machine Learning Agents Toolkit (ML-Agents) is an open-source Unity plugin that enables games and simulations to serve as environments for training intelligent agents. Agents can be trained using reinforcement learning, imitation learning, neuroevolution, or other machine learning methods through a simple-to-use Python API.

These trained agents can be used for multiple purposes, including controlling NPC behavior (in a variety of settings such as multi-agent and adversarial), automated testing of game builds and evaluating different game design decisions pre-release. The ML-Agents toolkit is mutually beneficial for both game developers and AI researchers as it provides a central platform where advances in AI can be evaluated on Unity’s rich environments and then made accessible to the wider research and game developer communities.[ML Agents GitHub page]

Features

- Unity environment control from Python
- 10+ sample Unity environments
- Support for multiple environment configurations and training scenarios
- Train memory-enhanced agents using deep reinforcement learning
- Easily definable Curriculum Learning scenarios
- Broadcasting of agent behavior for supervised learning
- Built-in support for Imitation Learning
- Flexible agent control with On Demand Decision Making
- Visualizing network outputs within the environment
- Simplified set-up with Docker

- Wrap learning environments as a gym

About TensorFlow

In this project, TensorFlow is used indirectly through the ML-Agents plugin.

TensorFlow is an open source software library for numerical computation using data flow graphs. The graph nodes represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) that flow between them. This flexible architecture enables you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device without rewriting code. TensorFlow also includes TensorBoard, a data visualization toolkit.

TensorFlow was originally developed by researchers and engineers working on the Google Brain team within Google's Machine Intelligence Research organization for the purposes of conducting machine learning and deep neural networks research. The system is general enough to be applicable in a wide variety of other domains, as well.

TensorFlow provides stable Python and C APIs as well as non-guaranteed backwards compatible API's for C++, Go, Java, JavaScript, and Swift.[TensorFlow GitHub page]

About GitHub and Git

Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity, and support for distributed, non-linear workflows.

Git was created by Linus Torvalds in 2005 for development of the Linux kernel, with other kernel developers contributing to its initial development. Its current maintainer since 2005 is Junio Hamano.

GitHub is a web-based hosting service for version control using Git. It is mostly used for computer code. It offers all of the distributed version control and source code management (SCM) functionality of Git as well as adding its own features.

It provides access control and several collaboration features such as bug tracking, feature requests, task management, and wikis for every project.

GitHub offers plans for enterprise, team, pro and free accounts which are commonly used to host open-source software projects. As of January 2019, GitHub offers unlimited private repositories to all plans, including free accounts.

As of June 2018, GitHub reports having over 28 million users and 57 million repositories (including 28 million public repositories), making it the largest host of source code in the world.

As with most other distributed version-control systems, and unlike most client–server systems, every Git directory on every computer is a full-fledged repository with complete history and full version-tracking abilities, independent of network access or a central server.

Git is free and open-source software distributed under the terms of the GNU General Public License version 2.

Used hardware

Most of the machine learning sessions have been done on my 2 personal computers. First a desktop that has these respective hardware specifications:

- **CPU** : AMD Ryzen 7 2700X Eight-Core Processor

  8 physical cores, 16 logical cores, no overclocking, base speed 3.70 GHz
- **RAM** : 16 GB, 2400 MHz
- **GPU** : NVIDIA Geforce GTX 1080, 8GB
- **SSD #1** : KINGSTON SV300S37A120G
- **SSD #2** : Samsung SSD 850 EVO 250GB
- **OS** : Windows 10 Pro, 64-bit version

Second, a Lenovo ideapad 330S laptop:

- **CPU** : Intel® Core(TM) i5-8250U CPU

  4 physical cores, 8 logical cores, no overclocking, base speed 1.80 GHz
- **RAM** : 8 GB, 2400 MHz
- **SSD** : SanDisk SD9SB8W512G1101
- **OS** : Windows 10 Home, 32-bit version

Implementation

How does the training environment look like?

In order to keep the environment simple to train, I have decided to create a simple game for the agent to train with.

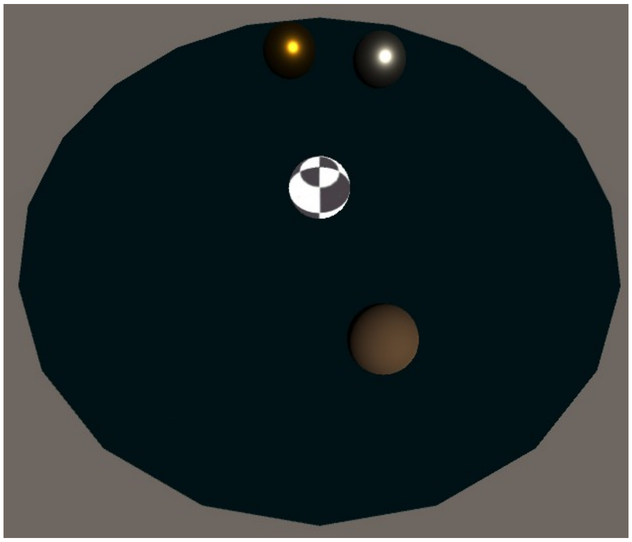


Illustration 2: Screenshot of an arena

The idea of the game for the agent or player is to control the black-and-white checkered sphere and push out of the stage all of the other colored spheres in a certain order. (seen in Illustration 2)

First the brown sphere must be pushed out, then the gray-metallic sphere, then lastly the golden sphere. If all three balls are pushed out in the correct order, the agent/player has won, if any of the balls have fallen when it was not supposed to fall (for example, it has attempted to push out the golden sphere) or if the black-and-white sphere has fallen out, then the agent/player lost.

All of the spheres have the same mass, drag and angular drag. And also all of them are affected by the same gravitational force. They are also the same size, all having a radius of 0,5 units.

The platform that all the spheres are standing on is a very thin cylinder with a radius of 5. From a physics point of view, it is immovable and the spheres cannot pass through it or push it no matter how hard they hit it.

How is the main sphere agent controlled?

The main sphere agent can only be controlled via two possible actions: a force applied on the X axis and a force applied on the Z axis. The code that applies these respective forces look a little like this.1

```
Vector3 controlSignal = Vector3.zero;
controlSignal.x = vectorAction[0];
controlSignal.z = vectorAction[1];
rBody.AddForce(controlSignal * speed);
```

Text 1: Force applied on the agent sphere

The physics calculations are being handled by the Unity game engine. The force is just applied, and the engine handles the rest.

Those ***vectorAction*** values that can be seen at Text 1 are being generated by the neural network after it has received the observations of the current game state. The machine agents brain does not inherently know that the first vectorAction value controls the X axis movement, and the second vectorAction value controls the Z axis movement. In the very first training steps, it will most likely just create random values that will the main sphere off the stage.

After many attempts of simply falling off the stage, it might accidentally knock the first brown sphere, and receive an award value. When it starts to receive the first rewards, it might also start to understand what the vectorAction values also control.

What can the machine agent “see” in the environment?

In order for the machine agent to be able to win at the game that was proposed, it must receive information about what it is playing. One way to do that is to simply give it everything that a human player can also see, like a camera render of the entire stage.

Indeed that would be possible, but training such a machine where it is receiving as an input the entire image, frame by frame, every time it has to decide what to do would mean an astonishing amount of time for it to train to a competent performance.

Surely such a mass of information will help train the best agent, but in order for it to finish training faster, a more limited set of data must be sent.

When a human is playing this particular game, it can see where it is currently positioned in stage, where all the other spheres are positioned. It can also approximate the speed that it is currently traveling that and a pretty general idea of how far it is from the center of the stage and how close it is to other spheres.

And all of this information is unconsciously deduced by the human. In order to help the machine agent to learn faster, we shall send all of this distilled information directly to it.

Observations that are sent to machine agent:

- the agents current position on the X and Z axis
- the agents distance from the center of the stage
- the agents velocity on the X and Z axis
- the agents “mission” (a number that symbolizes which of the colored ball should it attempt to knock out)
- the position of the brown sphere relative to the agent on the X and Z axis
- the distance from the brown sphere to the agent
- the brown spheres distance from the center of the stage
- the brown spheres velocity on the X and Z axis
- the position of the gray sphere relative to the agent on the X and Z axis
- the distance from the gray sphere to the agent
- the gray spheres distance from the center of the stage
- the gray spheres velocity on the X and Z axis

- the position of the gold sphere relative to the agent on the X and Z axis
- the distance from the gold sphere to the agent
- the gold spheres distance from the center of the stage
- the gold spheres velocity on the X and Z axis

One interesting thing that be seen is that data that is relevant to the Y axis is never being sent. That is because for the machine agent, that information is not necessary. The Y coordinate for all of the spheres will only change when the spheres are falling out of the stage.

The information that the spheres have fallen off the stage can be deduced just from the distance observations or from the “current mission” observation.

Another thing that might be odd to some is why are all of the positions of the other spheres are being as sent as relative to the main sphere. This is just to help the machine agent to better understand where are the other spheres. For example it is much simpler for the machine agent to deduce that it has to go to the right to hit the golden sphere if it has received that the relative position of it on the X axis is greater than 0. The alternative would have been for the machine agent to deduce to do additional computation, such as subtracting the absolute position of the golden sphere from the absolute position of the main sphere.

There are in total 24 observations sent to the machine agents brain.

How is the game stage generated?

In order for the machine agent to experience as many situations as possible, the target spheres positions are generated randomly in such a way so that they do not overlap.

Inside Text 2 there is a snippet of the code that attempts to generate a valid random position for the first target sphere, the brown sphere.

```
do
{
    Vector2 r = Random.insideUnitCircle * arenaRadius;
    targetPosition = new Vector3(r.x, spawnHeight,
        r.y);
    distanceToAgent =
Vector3.Distance(transform.position, targetPosition);
} while (distanceToAgent < 1.1f);
target1.transform.position = targetPosition;
```

Text 2: Position randomizer

Attempting to move any gameObject that has a RigidBody component in such away that it overlaps another gameObject with a RigidBody will cause them to sporadically move away from each other. The more overlapped they are, the more violent the reaction between them will be.

This position randomizer will start to take into consideration all the previously moved spheres when generating positions for the gray and golden spheres.

There is ample room on the stage, so it doesn’t take a significant amount of retries in order to generate valid non-overlapping positions.

Another thing to take into consideration is that simply moving the spheres back on the stage is not enough, the physics engine still has data about their velocity and angularVelocity.

Simply moving them back on stage will cause them to keep moving with the same velocity as before, so in order to prevent that, the velocity and angularVelocity of all the target spheres must be set to Vector3.zero.

What rewards are given to the machine agent?

The reward system is simple and straight forward, every sphere that is successfully knocked of the stage in the correct order will give a reward of 1/3f to the agent.

When all 3 spheres are knocked off, the agent is marked as Done, and Tensorflow will adjust the weights in the neural network in order to encourage the same behavior.

Anytime the agent does something wrong: knocking off one the spheres too early or dropping off stage; the agent will be marked as done, and it’s currently accumulated reward will be taken into consideration by Tensorflow.

More complicated reward systems can be implemented, such as on every action that the machine agent is doing, add a small constant negative reward that will encourage the machine agent to finish the task faster before it accumulates a time penalty that is too big.

Another example would be to give weighted rewards for each of the spheres being dropped off, such as:

- knocking out the brown sphere: 1/7 reward
- knocking out the gray sphere: 2/7 reward
- knocking out the golden sphere: 4/7 reward

This way, making further and further progress is exponentially better that just doing the first and second task and be contempt with just that.

There is also another way that could be done. I saw that in the “Pyramids” sample from the MLAgents even though there are two tasks to be done, the positive reward is given only when both of the tasks are completed in the correct order. With this the reward system could be like this:

- knock out the brown, gray and golden spheres in order: 1 reward

Speeding up the training process

One way to speed up training is by creating several agents and for each one of them to create an arena to train into. This can be seen to be also applied in several of the example scenes of the ML-Agents plugin and also described in the documentation [Making a New Learning Environment] found on the GitHub page for the respective plugin.

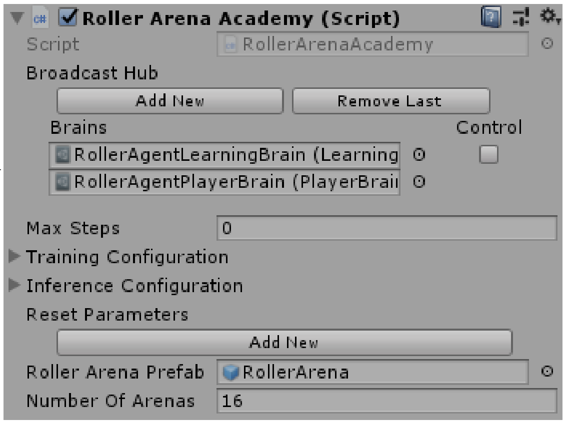


Illustration 3: Roller Arena Academy Unity settings

The responsibility of generating all of the agents and the arenas where they will train has been given to the ***RollerArenaAcademy*** class, which extends the Academy.

Inside of the InitializeAcademy method, a dynamic amount of arenas are generated.

The number of generated arenas is set with the help of the Unity Editor [Illustration 3]. The same is also said about the arena themselves, which are just copies instantiated from a prefabricated roller arena game object.

The code that handles the creation of the arenas is

```
public GameObject m_RollerArenaPrefab;
public int m_NumberOfArenas;
public override void InitializeAcademy()
{
  for(int i = 0; i < m_NumberOfArenas; i++)
  {
    Instantiate(m_RollerArenaPrefab, new Vector3(0, -4*i, 0), Quaternion.identity);
  }
}
```

At the moment of writing this document, I found that creating 16 arenas is an adequate amount that does not overload my system when training and also leaves enough resource for my PC to do something else also.

What are the used training parameters?

A list of all the training parameters that can be chosen are displayed on a documentation page called [Training ML-Agents].

The ones that are current used by default for training the roller agents are:

- use_curiosity: true => train using an additional intrinsic reward signal generated from Intrinsic Curiosity Module[Solving sparse-reward tasks with Curiosity]
  Seeing that the Pyramids example inside of the ML-Agents plugin was able to train to competent levels I decided to try this value to see if it affects the training in a positive way.
- curiosity_strength: 0.01 => magnitude of intrinsic reward generated by Intrinsic Curiosity Module
- curiosity_enc_size: 256 => the size of the encoding to use in the forward and inverse models in the Curiosity module
- summary_freq: 2000 => how often, in steps, to save training statistics
- time_horizon: 128 => how many steps of experience to collect per-agent before adding it to the experience buffer
- batch_size: 128 => the number of experiences in each iteration of gradient descent

- buffer_size: 2048 => the number of experiences to collect before updating the policy model
- hidden_units: 512 => the number of units in the hidden layers of the neural network
  In my attempt of making the roller agent able to learn complicated tasks, I have decided to give it a large amount of hidden units.
- num_layers: 3 => the number of hidden layers in the neural network
  Again, I have chosen this number in order to attempt to make the agents learn more complicated problems.
- beta: 1.0e-2 => corresponds to the strength of the entropy regularization, which makes the policy "more random."
- max_steps: 5.0e5 => how many steps of simulations are run in total
- num_epoch: 3 => the number of passes through the experience buffer during gradient descent

The chosen reinforcement learning technique is called Proximal Policy Optimization (PPO). All of the above values have been chosen by either copying them from other ML-Agents examples or after reading more about them from the GitHub documentation page named [Training with Proximal Policy Optimization]

State of the art

What is Proximal Policy Optimization?

The Proximal Policy Optimization algorithm, which from now on shall be refered as PPO, is an algorithm that uses a neural network to create behavior for machine agents, taking as input the observations of the environment it currently is in, and outputting the actions of the agent.

This algorithm is currently used by the ML-Agents plugin, which is implemented in TensorFlow.

Due to the fact that it is performing as good as or better than other state-of-the-art algorithms, it was also chosen by the OpenAI team as the default reinforcement learning algorithm.

This reinforcement learning technique was invented by John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford and Oleg Klimov and is described in the article named Proximal Policy Optimization Algorithms ([PPO]).

Up until recently, policy gradient methods have been mostly used for training deep neural networks with the task of controlling machine agents, such as in video games. But their main disadvantage depends on the chosen step size. If it is too small the machine agent is learning too slowly, if it is too large, the performance of the agent might suffer sudden significant drops. Even if the step size is picked carefully, the progress is still incredibly slow, taking millions (or billions) of time steps to learn simple tasks.

Other reinforcement learning techniques have been developed, such as [TRPO], and [ACER], with the hope of eliminating these flaws. But they come with disadvantages also, [ACER] is more complicated than [PPO] (even though [ACER] is a bit better than [PPO]), and [TRPO] isn’t so good at solving problems that require visual input, such as video games.

Here is the objective function that is currently used in [PPO].

$$L^{CLIP}(\theta)=\hat{E}^{t}[min(r_t(\theta)\hat{A}^{t}, clip(r_t(\theta),1-\varepsilon,1+\varepsilon)\hat{A}^{t})]$$

- θ is the policy parameter
- $\hat{E}_t$ denotes the empirical expectation over time steps
- r_t is the ratio of the probability under the new and old policies, respectively
- $\hat{A}_t$ is the estimated advantage at time t
- ε is a hyperparameter, usually 0.1 or 0.2

The team at OpenAI, was able to train a human shaped ragdoll that is able to: stand up straight without falling, walk fast towards a target (it was not running, it’s legs were never really off the ground at the same time), turn towards a target, maintain up-right position when hit small projectiles and finally when toppled over because it couldn’t balance, it was able to stand up by itself after being knocked over.

By using a keyboard, the user is able to direct where the rag-doll human/robot should be walking to. The target is simply placed in direction where to agent is supposed to walk to. Even though the way it walks is amusing, it has potential to become more lifelike than currently hand-made walking and turning animations that are usually made for games.

What is Curriculum Learning?

Curriculum learning in machine learning is a way of teaching machines to become proficient at solving problem by giving them at the start easy challenges that they have to overcome, and slowly increase the difficulty of the challenges as the machine masters the problem.

This idea is based on how human children learn, they are given at the very beginning simple lessons, and once they understand the basics, they are introduced to more complex lessons.

Easing up the children to more and more complicated problem gradually helps them understand the intricacies. Seeing how well this can work in the real world, an attempt was made to simulate this for machines.

One of the ML-Agents sample training scenarios that uses curriculum learning is Wall Jump. The training scenario can be described as blue cube shaped machine agents is tasked with reaching a green target on the arena that it is placed on.

Usually there between the agent and the target there is gray wall of varying heights. The machine agent is able to jump a fixed height. Sometimes the wall is small enough that a simple jump is enough to go over it, sometimes the wall is non-existent, sometimes the wall is too tall to jump over it with a simple jump.

The interesting bit in this scenario, is that there is also an orange block, that can be pushed around by the agent. The orange block is big enough for the agent to jump on it, and then over the wall if needed. The training scenario is setup in a such a way that the wall is never taller than the height the agent is able to jump while starting on top of the orange block.

It doesn’t look too difficult to learn, even for a neural network to learn. Simply generating arenas with random wall height, random starting position for the agent and random position for the target (the target is always behind where the wall generated, even if it’s height is non-existent) should be enough for the neural network to learn.

But something interesting happens when generating the arenas in a more controlled way. If the first few arenas have a non-existent wall height, and in time the wall height increases, the neural network learns significantly faster how to jump even the tallest wall in order to reach the target. It is faster than just picking random heights for the wall at any time.

In ML-Agents, Curriculum Learning defines the "lessons" that the machine agent has to go through with the help of "metacurriculum" files (explained in more detail at [Training with Curriculum Learning]).

Parameters have to be defined in the respective file that enables the TensorFlow API to control them.

```
{
  "measure" : "progress",
  "thresholds" : [0.1, 0.3, 0.5],
  "min_lesson_length" : 100,
  "signal_smoothing" : true,
  "parameters" :
  {
    "big_wall_min_height" : [0.0, 4.0, 6.0, 8.0],
    "big_wall_max_height" : [4.0, 7.0, 8.0, 8.0]
  }
}
```

Text 3: Curriculum structure example

For example in the Wall Jump sample training scenario there are the big_wall_min_height and big_wall_max_height parameters which are used in the Unity code to decide how tall to generate the wall.

What is Deep Reinforcement Learning?

In order to answer the question of "What is Deep Reinforcement Learning?", first we have to understand what is Reinforcement Learning by itself.

Reinforcement Learning refers to all algorithms that are goal-oriented and learn to execute complex tasks. As we know until now, they use positive rewards to encourage an actor to do things that we deem to be correct, and negative rewards in order to discourage it from doing things that we consider bad for solving the task.

In order to make it smarter at solving tasks, Deep Learning is used. In essence Deep Learning refers to using multiple neural network layers that help the agent extract higher level features from the observation input that it is receiving. One classical example is in image processing, where the first layers are very good at extracting very basic information, such as edges in the image, which is then used by the subsequent layers to understand more complex features such faces.

Combining Deep Learning algorithms and Reinforcement Learning algorithms, one can obtain a Deep Reinforcement Learning algorithm. Such an algorithm would be able to extract meaningful information from a broad observation input, and combine with the reward system in order to better decide its next action.

The machine agents in the ML-Agents plugin for Unity use deep reinforcement learning.

[Udacity Deep Reinforcement Learning]

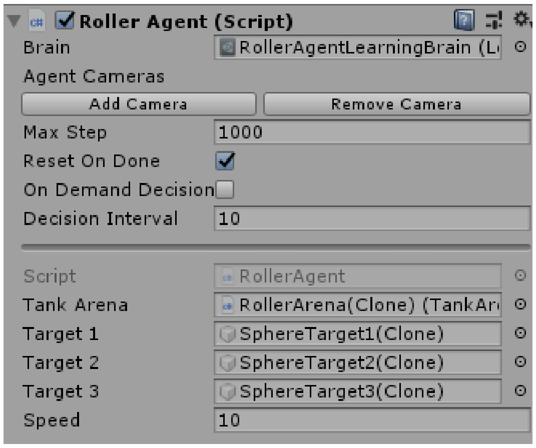
Discovered observations

Default setting used for training

Before the observations and differences between them are being described, first there must be a common default set of settings or particularities defined for them.

First here is the training hyper-parameters that are being used

```
batch_size:     128
beta:   0.01
buffer_size:    2048
epsilon:        0.2
gamma:  0.99
hidden_units:   512
lambd:  0.95
learning_rate:  0.0003
max_steps:      5.0e5
normalize:      False
num_epoch:      3
num_layers:     3
time_horizon:   128
sequence_length:        64
summary_freq:   2000
use_recurrent:  False
memory_size:    256
use_curiosity:  True
curiosity_strength:     0.01
curiosity_enc_size:     256
```



Secondly is to describe the particularities of environment created in Unity

- all of the spheres have the same mass
- the main agent sphere moves by applying a force of 10 multiplied with vector that indicates where it desires to move
- the main agent sphere is given the relative position of all the other target spheres as observation (not the absolute position in the game world)
- the "current mission" of the main agent sphere is coded as
  - 1 => push off the brown sphere
  - 2 => push off the gray sphere
  - 3 => push off the golden sphere
- failing to push the correct sphere just sets the agent as done and does not impose a negative reward
- the main agent sphere falling of the arena sets the agent as done and does not impose a negative reward

- going over the “Max Step” value because the too much time was taken trying to complete the task does not impose a negative reward
- the simple passage of time does not a negative reward
- in this version of the training scenario, there are in total three spheres that have to be pushed off
- rewards are given gradually and proportionally to the number of spheres that have been pushed off
  - 1/3 reward for each of the 3 spheres that have been pushed off

Training when using the default settings

By using these training parameters, the mean reward slowly raised to these respective values:

- 0.124 at step 4000
- 0.200 at step 10000
- 0.322 at step 26000
- 0.410 at step 112000
- 0.510 at step 258000
- 0.610 at step 370000

Only stabilized mean reward values are taken into consideration. There is no need to report reaching a milestone such as for example a mean reward of 0.5 when it is not immediately improving.

```
batch_size:      128
beta:   0.01
buffer_size:     2048
epsilon:         0.2
gamma:  0.99
hidden_units:    512
lambd:  0.95
learning_rate:   0.0003
max_steps:       5.0e5
normalize:       False
num_epoch:       3
num_layers:      3
time_horizon:    128
sequence_length:         64
summary_freq:    2000
use_recurrent:   False
memory_size:     256
use_curiosity:   True
curiosity_strength:      0.01
curiosity_enc_size:      256
```

Default training parameters

In this particular training session, the best mean reward that it was able to get was 0.690 at step 472000, but it was not stable. The reward did not approach maximum value of 1.

Behavior

The agent seems to have developed several interesting positive behaviors:

- it is proactively trying to always stay on the arena
- in the vast majority of times it will first chase after the brown sphere to knock it off
- after the first brown sphere is knocked off it will chase after the gray one to also knock it off
- sometimes when attempting to push the brown or gray sphere, it may avoid the gold sphere

Negative behaviors:

- when chasing the brown sphere, it sometimes moves the gray sphere by accidentally

- after successfully knocking off the gray sphere, it just moves around and around the stage, it does not actively search for the gold sphere to knock it off the arena
- sometimes when attempting to push the brown or gray sphere, it may push the gold sphere out the stage too early by mistake

Training when the reward is given only at the end

In this training scenario, instead of giving a reward of 1/3 to the agent for every sphere that pushes of correctly, now it will receive only once a reward of 1, only if it is pushes all of the spheres in the correct order.

Because of this, the chances of it ever receiving the reward are very slim when the agent is just learning. It is highly probable that it will just knock the spheres by mistake in the wrong order. But such a strict training scenario might help it discover eventually that the task is fully complete only when pushing all of spheres in the correct order.

In the first 20000 training steps, the mean reward only reached a maximum of 0.002, most of the time it's just 0.000.

After reaching 40000 training steps, the mean reward reached a maximum of 0.017. It seems that it is starting to discover some sort of pattern or behavior.

At training step 54000, the mean reward reached a value of 0.049. The training is incredibly slow, it seems as if it's just pure luck.

When reaching training step 80000, it seems as if the mean reward does not increase, rather it decreased over time, now being at a value of 0.018.

At training step 94000, the mean reward has recovered back to 0.052. Maybe that curiosity feature is indeed exploring wider and more random weights for the neural network.

At training step 108000 it has finally reached a mean reward of 0.108. Compared to the default settings, which already had a mean reward of 0.124 at step 4000, it is 27 times slower taking into consideration the progress made so far.

While going to training step 236000, the mean reward value kept growing and shrinking, at worst being a value of 0.057, at best 0.155. Again, the curiosity feature is exploring possible solutions.

At training step 278000, it has finally reached a mean reward value of 0.206, which is still incredibly slow compared to the default settings that reached this value at training step 10000. Thus again making it around 27 times slower.

The trend of rising and lowering mean reward continues to happen. At training step 296000 it reached a mean reward of 0.265, and at 304000 it dropped to 0.194.

At training step 360000, it has reached a mean value of 0.300. It close to reaching a success rate of 33% percent.

At around training step 450000, it has managed to somewhat stabilize a mean reward value of 0.332. Thus making it able to win the game at 33% of the time.

Finally at the end of the training session, with 500000 steps done, the best mean reward that it has achieved was of 0.404, which is worse the defaults case with 0.610.

Attempting to give a reward only at the end slowed down the training significantly. It might be possible that a much longer training period will make it achieve better performance.

Behavior

In comparison to the default case, this machine agent is moving more carefully. Instead of just flinging around very fast, it seems sometimes it's slowing down before coming into contact with the spheres.

Positive behaviors:

- once it manages to start pushing a sphere, sometimes it lets it roll a bit, then starts going for another sphere while the previous one has not fallen of the stage yet; this reduces the total time for all three spheres to fall down
- it became incredibly good at not falling of the stage
- in comparison to the default case, once it pushed of the first two spheres, it is better at finding the last golden sphere to push it away
- sometimes is seen to avoid pushing other spheres thus avoiding a hard to mitigate billiard effect

Negative behaviors:

- sometimes the machine agent slams into the golden sphere at the start of a round, thus knocking it out too early
- was observed one time to be too afraid to push a brown sphere because it was incredibly close to the edge of stage and not moving (the brown sphere had no velocity because it was just spawned)
- sometimes push off the metallic sphere first because the brown sphere was too close to the golden sphere at the start of the round

Because the reward is now fully given only at the end of task, this agent is sometimes performing better than the default agent by making smarter decisions that take it consideration the velocity of the spheres. Sadly with an average success rate of 33-40% it is not good enough.

Training when using 4 hidden layers with 32 units each

In this training session, the number of hidden units will be change from 512 to 32, and the number of hidden layers from 3 to 4.

Having a smaller number of hidden units in total might make it simpler for TensorFlow to explore relevant weights for the neural network. After all, the proposed problem does not need an incredibly complicated neural network to chase some spheres.

Increasing the number of layers might help it also create some interesting and complex functions.

Let’s not forget that this training scenario still use the default training scenario, the rewards are still cumulative (1/3 for each sphere).

Instead of being 512*3 = 1536 hidden nodes, there are now only 128 hidden nodes.

Already beginning from the training step 4000, the mean reward has become 0.132. This is very similar to the default case.

At training step 12000 it has reached a mean reward of 0.200. It is slightly slower than the default case.

At training step 36000 it is now at 0.314, and it seems to have some difficulty surpassing this value. Because of the cumulative reward system, the agent has mastered pushing off the brown sphere, now it must continue to learn to push after that the metallic gray sphere.

At 54000 it’s starting to make a breakthrough, the mean reward is 0.363, so that means that some agents are starting to discover how push off the brown sphere and gray sphere in the correct order.

The simpler neural network is starting to show it’s problems. When the training reached step 76000, the mean reward is still around 0.333. The default case reached that value at training step 26000. At this point it looks like having a network that is too simple is detrimental to learning more complex behavior.

When it reached training step 190000, the mean reward was 0.408. Reaching this mean reward took almost twice as many training steps then the default training scenario. The mean reward is now increasing only very slowly, occasionally dipping under 0.400.

Even at training step 360000, the mean reward has not changed by significant amount, only about reaching a value of 0.420.

At around training step 392000, the agent finally starts to grasp what it has to do in order to obtain the next chunk of 1/3 reward, thus approaching the mean reward of around 0.450.

At the end of the training session, the best mean reward had a value of 0.480, and the last mean reward at step 500000 had a value of 0.464.

In comparison with the default training session with a value of 0.610, it performed worse. The cause of why this happened might be because of pure bad luck, as in the possible space of values for the neural network was not explored sufficiently.

Or it might have been because the configuration of 32 hidden nodes by 4 layers was actually detrimental to generating useful behavior for the proposed problem.

Behavior

In comparison to the previous test case, where the result was given fully only at the end of a round, this machine agent is moving more chaotically. Most of the time its moving fast in circles, using it's high velocity to smack out the other spheres.

Positive behaviors:

- actively chasing the first brown sphere to push it out
- it became incredibly good at not falling of the stage
- after taking out the brown sphere, it starts it's chaotic spinning pattern, sometimes kicking out the metallic sphere
- the fact that it's moving fast helps it finish the task faster in some cases

Negative behaviors:

- sometimes the machine agent slams into the golden sphere at the start of a round, thus knocking it out too early
- due to the chaotic movement, it sometimes does mistakes at the very beginning of the round, such as knocking out the metallic or golden sphere
- does not actively search for the final golden sphere after pushing out the metallic sphere
- does not really seem to take into consideration the velocity of the other spheres

Training when using 4 hidden layers and more steps

From what I saw in the default case, the machine agent was able to reach a mean reward of around 0.600, thus I though, what if I let it train overnight, and also give it more hidden layers.

Having more hidden layers should help it learn more complex behaviors, and letting it train more time should help it reach better average rewards.

All the training settings are the same, except that there are now 4 hidden layers, and the max steps value is now set to 5.0e6 (5 million).

At step 6000, it has already reached a mean reward of over 0.139. This looks good so far, the training started really well.

At step 50000, it was able to stabilize the mean reward to be above 0.300. It took it almost twice as many training steps than the default case, but it is expected for this to happen, taking into consideration the extra hidden layer.

After a long time, it has reached a stable mean reward of at least 0.400, at training step 214000. Again it took twice as many training steps to reach this compared to the default case.

I thought everything will be well, but something very interesting has happened, the mean reward started to drop. And it didn’t stop. After it reached a decent mean reward of 0.467 at training step 272000, the mean reward just started to get lower and lower.

It dropped all the way down to a mean reward perfect 0.000, at training step 418000. It was as if all progress was lost. I am not completely sure why this happened, but this might be because of the curiosity flag being set to true.

I personally do not fully understand how this feature works, so its best to just quote them;

“encourage agents to explore the environment more effectively when the rewards are infrequent and sparsely distributed. These agents can do this using a reward they give themselves based on how surprised they are about the outcome of their actions” (taken from [Solving sparse-reward tasks with Curiosity])

So it seems like the agent is more prone to wildly exploring different strategies, giving itself some kind of special reward to continue searching, in hope of receiving a real reward.

For a very long time, this strategy does not seem to really work. The mean reward remained under 0.020 until training step 1168000. That means 896000 training steps have been spent randomly trying bad strategies. That is an amazing amount of time, considering all my previous training setups stopped when reaching 500000.

After this long pause of trying and failing for a very long time, the agent seems to start recovering some of it’s progress.

At training step 1216000 it reached a mean reward of 0.105.

At 1286000 it reached 0.200.

At 1340000 it got to 0.304.

Its now taking tens of thousands of steps to learn useful strategies.

It managed to stabilize a reward of at least 0.400 after 1538000 steps. The growth was slow, but at least it’s not dropping back to zero.

At training step 1948000, it has managed to stabilize a new best mean reward of at least 0.500. Maybe the new strategy that it was discovered due to the curiosity system might help it after all.

The incredibly slow learning process is still continuing. At training step 3352000, it finally managed to stabilize to at least 0.600. It took at least 1000000 steps of training for just that little increase in performance.

My hope in leaving my computer run over night was to discover a fully developed agent that was an expert at it’s task. But it was taking too much time. I stopped the training at step 3352000, the time was 19:07, the time when I started the training was 01:16, and all of that was to just reach a mean reward of 0.626.

It took almost 18 hours, only to reach that meager value.

Behavior

The movements of this agents are not chaotic, they may rather be consider to be prudent. The speed is very controlled at most times.

Positive behaviors:

- actively chasing the first brown sphere to push it out
- it became incredibly good at not falling off the stage
- sometimes is seen to chase metallic sphere in order to get the second reward
- sometimes seen to actively push on the final golden sphere if it comes into solid contact with it
- the more prudent nature of this agent makes it give a small push to the other spheres, as if it was played by a player

Negative behaviors:

- sometimes the machine agent slams into the golden sphere at the start of a round, thus knocking it out too early
- sometimes seen to be too afraid to approach the metallic sphere if the golden one is nearby to it
- in about 50% of times, it does not attempt to chase after the last golden sphere, only to spin around in the safety of the arena
- sometimes too afraid of the spheres that are very close to the edge of the arena

Training when separating the mission observation into three values

Taking into consideration the progress of the other training attempts, it always seems to have difficulty in mastering pushing all three spheres.

It only seems to be reaching an average reward value of around 0.610, which is close to the default given reward when knocking off the first two spheres.

One attempt at coercing the agent to learn better is to better split up the observations regarding the current mission for it.

Instead of just using a single float value that signals to the agent the current mission it has, now there are 3 separate values.

When the current mission is to push off the brown sphere, instead of just sending a single float value of 1, there are now the three float values 1, 0 and 0.

The same idea is used on the next spheres, for the metallic sphere being 0,1 and 0. And in the end when the agent has to push off the golden sphere, the value shall be 0,0 and 1.

Separating the values should now be easier for the agent to understand, it doesn't have to make an awkward comparison for the mission being equal to 1, 2 or 3. Now it will have these values that plainly say which one to attempt to chase.

The training process is going pretty smoothly. The mean reward is slowly increasing and a stable rate. There are some places where it dips down for a moment, but then it continues to increase. This phenomenon might be due to the curiosity flag being set to true, thus making it more likely to change it's strategy more radically.

Here is just a simple list of the training process:

- training step 6000, mean reward 0.117
- training step 12000, mean reward 0.219
- training step 24000, mean reward 0.329; most of the agents are now able to successfully push the first brown sphere off the stage.
- training step 84000, mean reward 0.401
- training step 242000, mean reward 0.524
- training step 326000, mean reward 0.601
- training step 400000, mean reward 0.650; now that it is close to reaching two thirds of the reward, it means most agents are proficient at proficient at pushing both the brown and metallic sphere off the stage in the correct order.
- training step 500000, mean reward 0.671; the end of this training session

The best achieved mean reward until training step 500000 was of 0.708, which is slightly better than the value of 0.690 from the default training session.

Compared to the default training session, the "milestones" are reached slightly faster. Maybe the clearer separation of mission observation value does indeed help.

Or it could just be the fact that better weights have been randomly selected by chance for the neural network of the agent.

One argument why it isn't just a random chance for the slightly faster training, is because even though they are random, they are spread over a very significant amount of time. That makes it more evenly spread in the long run.

Behavior

The movements of this agents are neither very chaotic, neither are too prudent. The speed is very controlled at most times.

Positive behaviors:

- actively chasing the first brown sphere to push it out
- the spheres tend to be more confidently pushed; not afraid to go to spheres that are on the edge of the arena
- seen one attempt of trying to stop the golden sphere from being pushed off the stage when touching it by accident; it's almost is as if it doesn't want to lose the game too early
- it became incredibly good at not falling off the stage
- observed sometimes to push multiple spheres in order to give them different velocities before they fall of the stage; the brown sphere to be pushed the fastest, metallic sphere given a medium speed, and golden sphere just a very light tap; the agent might have understood that it doesn't have to fully confirm that sphere has fallen off, and that it understands their order
- sometimes when the golden sphere is left last, the slightly chaotic movement of the machine agent might give a light tap to the respective sphere and knock it off

Negative behaviors:

- it doesn't seem to understand that when it's pushing the brown sphere and it hits another sphere there might be a chance that the second sphere is knocked off too early
- sometimes seen to be too afraid to approach the metallic sphere if the golden one is nearby to it
- has difficulty understanding how to chase the last golden sphere when it is its turn to be knocked off
- in about 50% of times, it does not attempt to chase after the last golden sphere, only to spin around in the safety of the arena
- sometimes too afraid of the spheres that are very close to the edge of the arena

Extending the training time

Taking into consideration the slightly better success of this training scenario, I was wondering what would happen if I let it train more.

Thus it was left again to train over the course of a day.

The average mean reward is sometimes decreasing significantly. This can be seen around training step 998001 where it reached a value of 0.370.

The reason why this is happening might due to the curiosity feature rewarding the agent by exploring different strategies that might help it finish the entire course.

Even though it was given copious amounts of time to train, it only reached a maximum mean reward value of 0.825 at training step 3298001.

While this is better than the default case, so much time being spent only training the agent is too much taking into consideration the simplicity of the proposed challenge.

Behavior after extending training time

The movements of the agent have become slightly more prudent than before. It seem to be pushing the sphere with a slight "scare", as if, when it first forcefully one the spheres, the next moment is seen to jolt momentarily in the other direction.

At other times is seen to over enthusiastically push the spheres off stage, thus resulting in the agent itself falling off the stage.

Positive behaviors:

- sometimes observed to expertly avoid spheres that don't have be pushed yet in order to correctly go for the right one
- sometimes observed that it gives only a slight velocity to some of the spheres so that it may move to other spheres to finish the task faster
- the machine agent is still good at staying on the stage
- when the last golden sphere is left alone on the stage, it seems to understand that and goes directly to it to push it off, instead of just spinning around in the stage and hitting it by accident
- most of the time not afraid to go after spheres that are near the edge of the arena

Negative behaviors:

- seen many times to push the brown sphere by mistake into the golden or metallic sphere, thus triggering a chain reaction of movement, making them fall too early
- the machine agent is seen sometimes to fall of stage because it accumulated too much velocity
- it doesn't seem to understand that when it's pushing the brown sphere and it hits another sphere there might be a chance that the second sphere is knocked off too early
- sometimes is seen to be too afraid to push the first brown sphere off the stage

In conclusion of this training setup, adding more training time does eventually help the machine agent become better at given tasks.

Training when using no curiosity and smaller batch_size and buffer_size values

Training these machine agents take a lot of time. Most of the time they take at least 2 hours for them reach the 500000 step.

One suggestion I found on the internet was to use smaller batch_size and buffer_size values, and because I want the training process to be more steady, I also disabled the curiosity feature.

Now that the curiosity feature is turned off, it should no longer wildly increase and decrease the mean reward over time.

All the characteristics of this training are the same as the default training scenario, with the only differences being the used hyperparameters.

The updated values are:

- use_curiosity: false
- batch_size: 64
- hidden_units: 512

Around training step 6000, the machine agent has reached a mean reward of 0.106. A pretty decent start.

At training step 12000, it reached a mean reward of 0.206. This is very similar to how the default training scenario manifested.

At training step 42000, it reached a mean reward of 0.309, which sadly didn't continue to increase. It then decrease to 0.211 when reaching training step 44000.

The mean reward seems to have reached some sort of ceiling of around 0.333, even at training step 80000. It is taking more time to train compared to the default training scenario. At around this step it seems the machine agent mastered pushing off only the brown sphere.

When reaching training step 148000, the mean reward is around 0.400. The agent is slowly learning the next step of exercise, pushing the metallic sphere after the brown sphere.

At training step 208000 it seems to be picking up the pace. The mean reward is around 0.450.

A stable mean reward of 0.500 is reached at step number 272000. It's only slowly lagging behind the default training session, which achieved this value at 258000.

When reaching the training step 416000, it's getting close to mastering pushing the first two spheres correctly, by having a stable mean reward values of approximately 0.640.

Behavior

This particular machine agent is seen to move more confidently then others but is at the seen to stagnate very often when it was to push off the last golden sphere.

Positive behaviors:

- actively chasing the first brown sphere to push it out
- sometimes observed to give a very solid hit to the other spheres when it sees that it has clear path to do that
- it became incredibly good at not falling off the stage

- seems to understand that it has to push the spheres off the stage in the correct order
- seems to understand sometimes that the velocity of the spheres is a good indicator for finding out if they are going to fall off soon

Negative behaviors:

- sometimes locks up (makes only very small movements, as if it's stuck) when it has to push the last golden sphere
- doesn't really understand that colliding spheres together might send them off the stage too early
- when the stage resets, the machine agent might have a very great velocity from the previous stage, thus accidentally slamming into a new sphere, which might not be a brown one
- sometimes seen to spin around the stage when only the golden sphere is left on stage; it doesn't seem to go directly for it

Training when using normalization

In this training session, all the training parameters and characteristics are the same as the default training session, with the exception of the normalization parameter being set to true.

Normally this is set to false, and I was curios what effects does it have when its enabled. I saw that this parameter is set to true also in these samples: BouncerLearning, 3DBallLearning, 3DBallHardLearning, TennisLearning, CrawlerStaticLearning, CrawlerDynamicLearning, WalkerLearning and ReacherLearning.

Taking into consideration the fact that a pretty wide variety of learning environment are taking advantage of this normalize flag, I decided that it is worth a try.

"normalize corresponds to whether normalization is applied to the vector observation inputs. This normalization is based on the running average and variance of the vector observation. Normalization can be helpful in cases with complex continuous control problems, but may be harmful with simpler discrete control problems." (the definition was taken from the [Training with Proximal Policy Optimization] web page).

The mean reward of 0.120 was reached at training step 6000, which is a similar achievement to the one made by the default training scenario.

At training step 12000, the mean reward stabilized above 0.200. This is slightly slower than the default training scenario.

At training step 92000, the mean reward stabilized above 0.333. It seem that it is significantly slower then the default scenario, which achieved this around training step 26000. It is hard to tell if this is caused by the normalize flag, or by simple bad luck during the random selection of weights for neural network.

The trend of the slow training seems to continue. The mean reward has stabilized above 0.400 at around training step 136000.

But as time continues, this testing scenario seems to have reached the stabilization of mean reward 0.500 faster, at training step 226000.

At around training step 318000, it has achieved a mean reward of 0.594, but it has not stabilized yet. The mean reward still tends to drop down a little from time to time. The machine agent should be soon mastering pushing off the brown sphere and the metallic sphere.

Only when reaching training step 440000 it can be said that it is safely over 0.600.

At the end of this training session, after 500000 steps, the best average reward was of 0.646. Which is slightly smaller than the default training scenario.

The addition of the normalize flag does not seem to significantly improve the speed of training or the rewarding.

Behavior

The way this machine agents moves is rather peculiar. Instead of moving in a circle-like pattern, it tends to much faster in X axis, instead of the Y axis. This makes it a bit more reluctant to change its current Z axis position in order to better collide with the other spheres.

Positive behaviors:

- actively chasing the first brown sphere to push it out
- sometimes observed to give a very solid hit to the other spheres when it sees that it has clear path to do that
- it became good at not falling off the stage
- seems to understand that it has to push the spheres off the stage in the correct order

Negative behaviors:

- doesn't really understand that colliding the goal spheres between them might result in them falling too early
- overly favoring movement on the X axis makes it very reluctant to chase rare cases when the current goal sphere is directly above or below on Z axis
- does not seem to understand the velocity of the spheres; it's not taken into consideration so that it may finish the task faster

Training by rewarding only just touching the spheres

Just for the sake of trying, it is worth to see if a simpler mission for the machine agent can be mastered.

Here the stakes are simpler, the brown sphere, metallic sphere and golden sphere only have to be touched in the correct order. They disappear instantly after they are touched, and the reward is given as usual, as if they were pushed off the stage.

All the other training parameters and characteristics are the same as in the default training scenario.

This training scenario is significantly faster in getting a high mean reward value.

By step 4000, it has already reached a mean reward of 0.229.

At training step 12000, it reached a mean reward value of 0.362. Most of the agents are now able to reach for the first brown sphere.

At training step 70000, a mean reward of 0.668 was achieved. Now most agents are reaching one after the other the brown sphere and the metallic sphere.

From here on out it should be very straight forward for TensorFlow to discover the last adjustment to go for the golden sphere.

At training step 108000, it reached a mean reward value of 0.791, which is a considerably high value, but does not manage to raise it higher than that for a while. Occasionally dipping and increasing.

The mean reward does not increase, instead it keep increasing and decreasing for a very long time.

At training step 500000, it didn’t manage to improve the mean reward over 0.791. Considering the potential of the mission being simpler, I modified the training parameter max_steps to 5.0e6.

The mean reward just doesn’t seem to just go upwards, there are many times where it reaches some sort of local maximum and then starts dipping down.

Such “local maximum” and “local minimum” moments are observed around the training steps:

- 962090, mean reward 0.809
- 1132090, mean reward 0.098
- 1256090, mean reward 0.380
- 1298090, mean reward 0.175

One reason why these weird mean reward values manifest, could be because of the curiosity feature from the ML Agents plugin. It encourages the machine agent to try new strategies with the scope of discovering new interesting observations that might advantage it.

This might have paid off eventually. The best mean reward had a value of 0.956 which is incredibly close to a perfect reward value of 1. This value was reached at training step 2846090, which took almost an entire day to do so.

The training was allowed to continue in hope for it become even close to a perfect reward of 1, but I saw it started to dip again, and thus I stopped the training.

The last training step was done around 2874090, and had a mean reward value of 0.929. Currently I do not know if the model that is kept is the last one that was trained, or if it is the best one that ever performed.

Luckily the achieved performance was pretty good even if the observation values still contained the velocity of the other spheres. They were not useful at all, because once they were touched they disappear. They velocity is always 0, and it could have been removed in order to further increase the training speed.

Behavior

The machine agent is performing marvelously. It has learned how to control it’s velocity, and a pretty good sense of how to dodge for example the metallic or golden sphere when it has to touch the first brown sphere.

The only times it is losing is either due to bad luck when generating the sphere positions (for example: brown sphere is very close to edge and the rest of the spheres surround it), either due to the machine agent having too much velocity from the previous run.

Positive behaviors:

- it has learned to dodge spheres that must not be touched yet in order to chase the one that the current mission points it to
- it has learned to go the spheres in the correct order
- it became good at not falling off the stage
- when the line between the machine agent and the current mission sphere is clear, it moves with a confident speed towards it

Negative behaviors:

- sometimes it has bad luck because of the way the stage is generated; any remaining velocity from the machine agent might propel it in a newly spawned sphere that might not be correct one at that time
- sometimes seen to be too afraid of dodging both the metallic and golden sphere in order to reach the brown sphere, even though there is sufficient space between them
- rarely seen to fall outside the stage because the remaining velocity of the machine agent after hitting the last sphere from the previous run is still too great

Training when only having to push two spheres

One obvious thing that can be seen from most of the other training sessions so far is that the machine agent has a difficulty in mastering pushing off all three spheres.

Thus it is worth seeing if simplifying the problem in different ways increases the learning rate of the machine agent.

In this session, all machine agents now only have to push the brown sphere off the stage, then the metallic sphere. There is no more a golden sphere to take into consideration.

Compared to the default training session, the number of spheres have been changed, and along with that, the reward for each sphere is now 0.5, and the number of observations have been reduced. There are no observations relevant to the golden sphere being sent, because it doesn't exist anymore.

This training session is significantly faster at gaining a good mean reward.

At training step 12000, the mean reward is already at 0.508, which mean the majority of the machines agents have figured out how to push off the brown sphere, even some of them were lucky enough to also push the metallic sphere.

Now that the golden sphere is out of the picture, the stage is also less cluttered and simpler. The agent can't make as many accidental movements as before.

At training step 24000, the mean reward is at 0.660. The machine agent is starting to understand how to chase metallic sphere after the brown one.

At training step 58000, the mean reward has stabilized at 0.855. It is rather interesting how after completing a small sub task, it is difficult for machine agents to continue with learning the other sub tasks successfully. Over 30000 steps were necessary in order to attain a meager extra reward of around 0.200.

Training when using two stacked vectors

All of the training scenarios until now only passed observations of the present state of the environment. The default machine agents have no ability to remember events from the past. Every movement that they make is always based on the present.

My original guess was that passing sufficient observation values of the present state of the spheres would be enough in order to manage to find the best way to push them off in the correct order.

While searching for ways to improve the machine agent, I discovered an interesting slider in the Unity editor (as in Illustration 4), the one named Stacked Vectors.

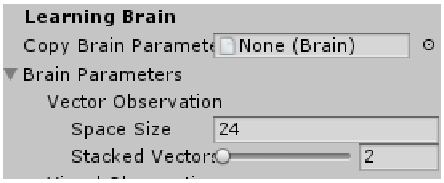


Illustration 4: Learning Brain Settings

The Stacked Vectors value represents "the number of previous vector observations that will be stacked and used collectively for decision making. This results in the effective size of the vector observation being passed to the Brain being: Space Size x Stacked Vectors” (taken from the [Learning Environment Design Brains] web-page).

This value lets a machine agent “remember” values from the past, and use them in current time decision calculation. But one disadvantage of this would be the vast increase in size of number of observations. This would mean a typical learning brain that would have normally 24 observations, if its stacked vector value is 2, it would now have 48 values being used as input for its neural network.

A vast amount of inputs will indeed help an agent train and gain a better performance for the task , but this results in more time being needed for the respective agent to understand the meaning of the new values.

One way the machine agent might use the values from the previous frame is to see if the distance of the target spheres has increased or decreased. With this information it can decide if it has to pursue that respective sphere to not fall yet, to let it fall off in order to begin pushing the next sphere or to push it harder if the its speed is not high enough.

But with this there might be some redundant information, having an update on the current and previous distance is nice, but is it really necessary to know the previous frame mission value? Or to know the previous distance? A lot of information might simply not be worth to be repeated, but taken into consideration the only thing that needed to be changed was that slider in the Unity editor, and no change to the script, it was a try to see the results.

The training scenario is all the same as the default training scenario, except that the maximum amount of steps was increased to 5.0e6, and the stacked vectors value is set to 2.

At the very beginning of the training session, it’s pretty much the same as the default training session, reaching a mean reward of 0.132 at training step 6000.

The trend continues, at training step 10000, the mean reward reached a value of 0.217.

At training step 16000, we see something surprising, the mean reward already stabilized above 0.300, while the default training scenario managed this at training step 26000.

The rate at which the mean reward increase at the beginning seem to be faster at the beginning, reaching a value of 0.403 at training step 54000. In comparison the default scenario managed this at 112000. This scenario managed to master pushing off the brown sphere in almost half the training steps compared to the default training scenario (it reached that value at 112000)!

For the next few tens of thousands of steps, the mean reward value increases to 0.500, but does not stabilize yet, dipping down to 0.349 at training step 210000. It finally manages to smash discover a better strategy, thus stabilizing the mean reward above 0.500, beginning with training step 250000.

This is a shame that this happened so late, as the default training case managed this around training step 258000.

The stacked vector value of 2 seemed to help the machine agent learn in a very small time how to avoid the metallic and golden sphere in order to chase the brown sphere, but then got stuck with that strategy for a very long time.

When reaching training 500000, the mean reward is at 0.554. All that time it wasn't quite able to master pushing both the brown and the metallic sphere in the right order. The default training surpassed that mean value a long time ago, with a stabilized value of 0.610 at training step 370000.

The next hundreds of thousands of steps, the machine agent is still struggling to stabilize the mean reward to 0.600. The mean reward is slowly varying around the value of 0.550.

The firs time it reached a mean value of 0.602, the training step was 448000, but it quickly lost some ability.

At around training step 2000000, it has finally overcome some sort of hurdle, stabilizing its mean reward over 0.600. The very fast learning speed from the beginning for the simpler task for pushing the brown sphere has been traded off for a very sluggish incrementation in the long run.

The machine agents seems to have mastered pushing off the first two spheres at around training step 2500000. This is incredibly slow compared to the default training scenario. Luckily the mean reward seem to be slowly increasing for there on.

At training step 3328000, the mean reward has stabilized over 0.700. Bigger and bigger time frames are required in order to become better.

A mean reward of over 0.750 is starting to stabilize around training step 4000000. Over 600000 steps just to increase another 0.050 in the mean reward.

When the training session has stopped, it has reached training step 4166000, where it was an average reward value of 0.786. At training step 4156000, it reached a the biggest mean reward value of 0.807. In these final training steps, the mean reward seems to have stabilized around 0.785.

This training session took almost 18 hours. Being left over night, and continuing to run during the day.

Behavior

Positive behaviors:

- it became good at chasing the first brown sphere
- it has learned to go the spheres in the correct order
- it became incredibly good at not falling off the stage
- after pushing off the brown sphere, it is seen to chase after the metallic sphere
- it is seen sometimes to give first a small impact to the metallic sphere in order to make it move slowly, and then slam hard into the brown sphere to take it out first

- rarely seen to give a small velocity to each of the spheres in order to push them all out sequentially in the right order, even though the first brown sphere didn't even fall out yet
- seen to avoid the metallic sphere when it's current mission is to push off the brown sphere

Negative behaviors:

- when its current mission is to push off the brown or metallic sphere, the machine agent doesn't really take into consideration the position of the golden sphere, thus hitting it many times by accident, and not correcting its velocity
- seen several times that the strategy of pushing first the metallic sphere fails horribly because it doesn't chase the brown sphere fast enough
- when the golden sphere is the last sphere to be pushed, it doesn't go straight to it, the machine agent just moves around the stage in big circles, thus hitting it occasionally by accident
- seen sometimes to attempt to stop spheres from falling off when it's not their turn, only to push them even faster then intended out

Training when using a visual input

There are many ways of conveying information about the environment to a machine agent. Most strategies involve sending some form of distilled data to the respective agent. Such as relevant positions, nearby walls, some distance to important things and maybe some prepossessed that help the agent understand faster what it's supposed to do.

There is also another way, make it see how humans see the game. That is to give an actual graphically rendered input, similar to what a human would see.

When a human plays a video game, it is constantly aware of all the queue and events that happen on the screen. Things like it's own position in the world, the position of the enemies and their health values are conveyed visually and sometimes by sounds.

The human can deduce how well it is doing just from these queues, all there is left for it is to desire to become better and to win.

As amazing as it may sound, there are drawbacks to send a visual input of the game itself. One of the first problems is the sheer size of such an input. Typical video games of 2019 are rendered at a resolution of 1920*1080 RGB pixels, with image refresh rates that can range from 30 Hz to 60 Hz.

That would mean the machine agent would typically have to receive and digest an input of

1920 pixels on width * 1080 pixels on height * 3 colors per pixel = 6220800 **inputs per action**

6220800 inputs per action * 30 frames per second = **186624000 inputs per second!!!**

And that value was calculated using a refresh rate of 30 Hz, if it were 60 Hz that number would be twice as big. And if the resolution was 4K, that number would be 4 times as big!

That is simply way too much data to be processed. There must be a way to simplify the input.

The typical route that is taken, is to simply use a smaller resolution. The machine agent does not have to see the entire image in perfect clarity, it just needs enough to distinguish the important things. Another thing that could be done is to not send RGB values, just use a gray scale image. Gray scale images are typically 3 times smaller than RGB images because they only need one value per pixel, instead of 3 values per pixel, one for each color (Red Green Blue).

And finally that machine agent wouldn't probably need to know everything that happens every single frame. It is enough for it to understand what is happening when it wants to take an action. Taking into consideration that all our machine agents had a decision interval of 10, that means it's requesting an input and sending an output every 10 frames, instead of 30Hz, 60Hz or more.

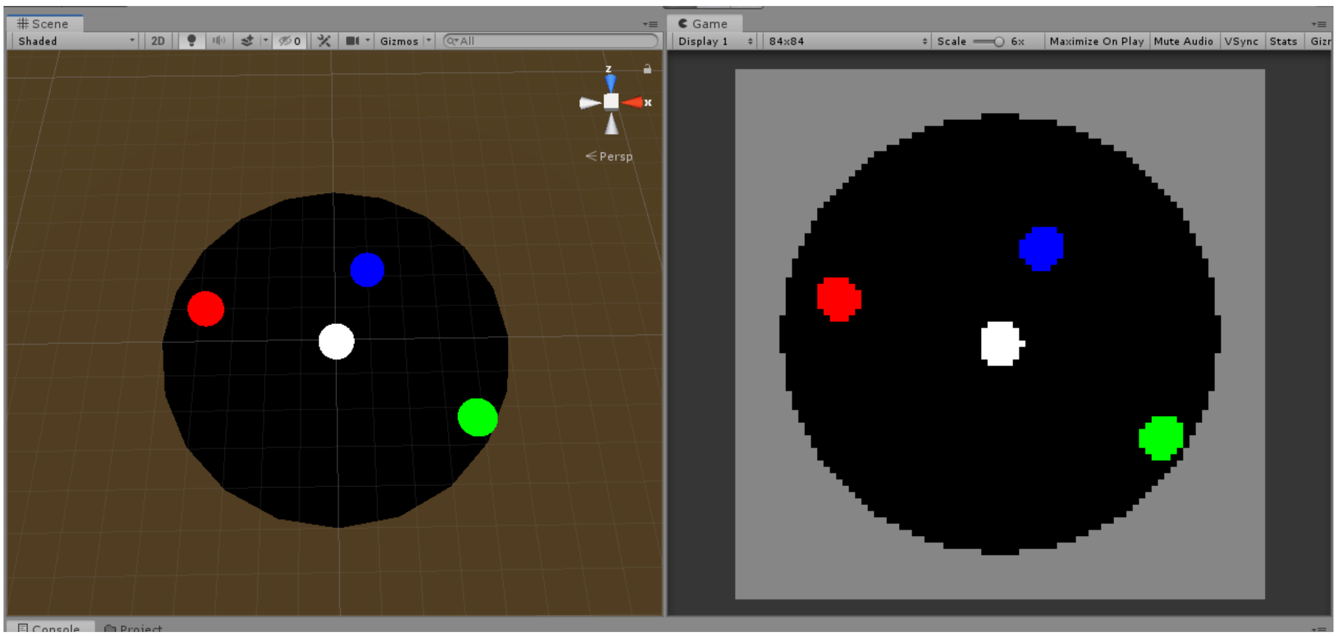


Illustration 5: Roller Camera Agent Training Environment

In this particular training scenario, several things have been changed. The colors of all of the spheres, the arena and the background have been changed. The agent sphere itself is now plain white, the arena is plain black, and the rest of spheres are plain red, green and blue.

The reason why I chose is in order to make it simpler for the agent to distinguish the objects in the arena. In this training scenario the agent does not receive any kind of other observation! It no longer receives any of the positions, distance or velocities of itself or all the spheres. Now it has to find out by itself what are each of the colored blobs on the screen.

In Illustration 5, on the left side of the screen can be seen the game world as in the editor. On the right side of the screen is game camera, but it manually selected resolution of only 84 by 84. The respective camera represents pretty accurately what the machine agent is receiving as an input.

84 pixels on width * 84 pixels on height * 3 colors per pixel= 21168 **inputs per action**

This value is relatively better than 6220800 inputs. Approximately 300 times lesser inputs per action. But still a significant amount of input.

In order to help it a bit with the training, I set up more hidden layers for the neural networks, it now uses 6 of them. More layers might help it better understand the feature of the present in the camera input.

Source code

RollerAgent.cs

The RollerAgent.cs file contains the RollerAgent class which extends the Agent class from the ML-Agents plugin. It is responsible for collecting the observations of the environment, interpreting the action values that come from the neural network, and resetting the position and velocity of itself and the target spheres.

The class was designed to have some flexibility regarding how the training environment is handled. Features can be enabled and disabled by changing boolean and integer variables that will be shown below.

Initialization and reset

In Text 4 can be seen the objects and variables that every RollerAgent object needs. The arenaPosition is needed in order to know where to position all the spheres after a reset. The public modifier makes it easy for other classes, or for the user, to set what GameObjects have to be chased and handled.

All of the RigidBody objects are for convenience; reduce further down in the code the amount of calls to the function GetComponent<Rigidbody>().

```
Rigidbody rBody;
public Vector3 arenaPosition;
public GameObject target1;
public GameObject target2;
public GameObject target3;
private Rigidbody target1RigidBody;
private Rigidbody target2RigidBody;
private Rigidbody target3RigidBody;
public float speed;
```

Text 4: RollerAgent objects and variable

The speed variable is for controlling how fast should the RollerAgent be allowed to move.

In Text 5 can be seen the variable that control how the training environment is interacting with the RollerAgent itself.

```
private const bool giveGradualRewardForEachSphere = true;
private const bool eachMissionIsItsOwnBoolean = false;
private const bool rewardOnTouch = false;
private int numberOfTargets = 3;
private int currentMission = 1;
```

Text 5: RollerAgent training environment modifiers and current mission variable

The giveGradualRewardForEachSphere flag controls whether to give an incremantal reward to the agent after pushing off every target sphere, or to give a big reward only when all the target spheres have been pushed off.

The eachMissionIsItsOwnBoolean flag controls whether to send only a single value for the current mission of the agent, or to send a boolean value for each of the target spheres, with a value of true only if when it has to chase it and push it off, or false when it must not push it off the stage yet.

The rewardOnTouch flag controls whether the RollerAgent has to push off the stage the target spheres or if it is enough to just simply touch in order to get a reward and complete a mission.

The numberOfTargets value controls how many sphere targets are on the arena for the agent to push off. Currently it can only have a value between 1 and 3. The majority of the testing scenarios that were done had a numberOfTargets value of 3, thus meaning that all three target spheres were present.

When for example the numberOfTargets is 2, every target sphere being pushed off gives a reward of ½ = 0.5.

The currentMission variable stores a value that represents which one of the missions the agent must do right now. A currentMission value of 1 would mean to chase the brown sphere. A value of 2 is for the metallic sphere. And finally a value of 3 is for the golden sphere.

```
public override void InitializeAgent() {
  if(numberOfTargets < 1) {
    numberOfTargets = 1;
  }else if(numberOfTargets > 3) {
    numberOfTargets = 3;
  }
  rBody = GetComponent<Rigidbody>();
}
```

Text 6: RollerAgent initialization

Using a variable to store the current mission instead of inferring it from the current state of the target spheres simplifies the code.

The RollerAgent initialization is shown in Text 6. Nothing much is done here, only to check that the numberOfTargets value is clamped between 1 and 3, and store a reference to the RigidBody component of the RollerAgent game object.

Every time an agent is initialized, or when it has been reset, it’s AgentReset method is called. Usually in this method there is implemented some logic for moving everything in the training environment back to relevant positions.

```
public override void AgentReset(){
  if (!target1RigidBody){
    target1RigidBody = target1.GetComponent<Rigidbody>();
  }
  if (!target2RigidBody){
    target2RigidBody = target2.GetComponent<Rigidbody>();
  }
  if (!target3RigidBody){
    target3RigidBody = target3.GetComponent<Rigidbody>();
  }

  if (transform.position.y < arenaPosition.y + 0.45f){
    rBody.angularVelocity = Vector3.zero;
    rBody.velocity = Vector3.zero;
    transform.position = new Vector3(0, arenaPosition.y +
0.5f, 0);
  }
  currentMission = 1;
  ResetAllTargets();
}
```

Text 7: RollerAgent AgentReset method

In Text 7 there can be seen the implementation of the AgentReset method. In here the RigidBody component of every target sphere is being obtained if it wasn’t obtained until now, the agent sphere is placed back in the center of the stage if it fell of stage for some reason. The currentMission value is set back to 1 so that the agent will know to start chasing for the first target sphere. At the end of this method, the ResetAllTargets method is called.

```
target1RigidBody.velocity = Vector3.zero;
target1RigidBody.angularVelocity = Vector3.zero;
target2RigidBody.velocity = Vector3.zero;
target2RigidBody.angularVelocity = Vector3.zero;
target3RigidBody.velocity = Vector3.zero;
target3RigidBody.angularVelocity = Vector3.zero;
```

Text 8: Reset target spheres velocity and angular velocity

```
target1.SetActive(numberOfTargets >= 1);
target2.SetActive(numberOfTargets >= 2);
target3.SetActive(numberOfTargets >= 3);
```

Text 9: Selective activation of the target spheres

In the ResetAllTargets method, there are many things done, the first being canceling all of the velocity and angular velocity of all the target spheres as in Text 8.

```
float spawnHeight = arenaPosition.y + 0.5f;
float arenaRadius = 4.5f;
```

Text 10: Auxiliary variables in ResetAllTargets

In Text 9 the target spheres are selectively activated depending on the value of numberOfTarges.

In Text 10 there can be seen two auxiliary variables that simplify the code for the next lines of code. The arenaRadius value of 4.5 was hard-coded for convenience, but has to be changed if the actual size of the arena is changed.

The position of each of the target spheres is selected randomly until they have a distance of at least 1.1 between each one of them and the agent itself. (show in Text 11)

This way no training scenario will be ever start with target spheres already overlapping with others or with the agent.

After each target has been successfully placed in the environment, a check is done on the numberOfTargets variable to see if the code logic has to find a position for the next targets too.

Because some training environments only use two spheres, it is not worth finding a valid position for the third sphere.

```
do{
  r = Random.insideUnitCircle*arenaRadius;
  targetPosition = new Vector3(r.x,spawnHeight,r.y);
  distanceToAgent = Vector3.Distance(
    transform.position, targetPosition);
} while (distanceToAgent < 1.1f);
target1.transform.position = targetPosition;

if (numberOfTargets < 2) return;
do{
  r = Random.insideUnitCircle*arenaRadius;
  targetPosition = new Vector3(r.x,spawnHeight,r.y);
  distanceToAgent = Vector3.Distance(
    transform.position, targetPosition);
  distanceToTarget1 = Vector3.Distance(
    target1.transform.position, targetPosition);
} while (distanceToAgent < 1.1f
         || distanceToTarget1 < 1.1f);
target2.transform.position = targetPosition;

if (numberOfTargets < 3) return;
do{
  r = Random.insideUnitCircle*arenaRadius;
  targetPosition = new Vector3(r.x,spawnHeight,r.y);
  distanceToAgent = Vector3.Distance(
    transform.position, targetPosition);
  distanceToTarget1 = Vector3.Distance(
    target1.transform.position, targetPosition);
  distanceToTarget2 = Vector3.Distance(
    target2.transform.position, targetPosition);
} while (distanceToAgent < 1.1f
         || distanceToTarget1 < 1.1f
         || distanceToTarget2 < 1.1f);
target3.transform.position = targetPosition;
```

Text 11: Finding positions for target spheres

Also something to observe that the more spheres are placed in the environment, the more distance checks have to be made to ensure that they do not collide, which might make the loops last a bit longer. But they should not really be of any problem considering the radius of the arena is 5 and the radius of the agent and target spheres is 0.5, which leaves for them more than enough room to find a valid spot.

Interacting with the environment

The Goal1Achieved method, shown in Text 12, is called whenever the RollerAgent managed to push out the first target sphere. The method then deactivates the respective sphere, adds the reward if necessary and marks the agent as done if that was also the last sphere to be pushed (can only happen if the numberOfTargets is set to 1). If the respective sphere was pushed out when it wasn't its turn, the agent is marked as done without it receive the reward.

Similar methods can be seen for the other two targets spheres as well, in Text 13 and in Text 14.

```
private void Goal1Achieved() {
  if (currentMission == 1) {
    target1.SetActive(false);
    if (giveGradualRewardForEachSphere) {
      AddReward(1f / numberOfTargets);
    }
    if(currentMission == numberOfTargets) {
      if (!giveGradualRewardForEachSphere) {
         AddReward(1);
      }
      Done();
    }
    currentMission++;
  }
  else {
    Done();
  }
}
```

Text 12: Goal1Achieved method

```
private void Goal2Achieved() {
  if (currentMission == 2) {
    target2.SetActive(false);
    if (giveGradualRewardForEachSphere) {
      AddReward(1f / numberOfTargets);
    }
    if (currentMission == numberOfTargets){
      if (!giveGradualRewardForEachSphere){
        AddReward(1);
      }
      Done();
    }
    currentMission++;
  }
  else {
    Done();
  }
}
```

Text 13: Goal2Achieved method

```
private void Goal3Achieved() {
  if (currentMission == 3) {
    target3.SetActive(false);
    if (giveGradualRewardForEachSphere) {
      AddReward(1f / numberOfTargets);
    }
    if (currentMission == numberOfTargets){
      if (!giveGradualRewardForEachSphere){
        AddReward(1);
      }
      Done();
    }
    currentMission++;
  }
  else {
    Done();
  }
}
```

Text 14: Goal3Achieved method

Each of the 3 methods also take care of incrementing the currentMission variable each time the correct sphere is pushed off, thus better encapsulating the functionality.

```
public override void AgentAction(float[] vectorAction, string textAction) {
  Vector3 controlSignal = new Vector3(vectorAction[0], 0, vectorAction[1]);
  rBody.AddForce(controlSignal * speed);

  if (target1.activeSelf && numberOfTargets>=1 && target1.transform.position.y+1<arenaPosition.y){
    Goal1Achieved();return;
  }
  if (target2.activeSelf && numberOfTargets>=2 && target2.transform.position.y+1<arenaPosition.y){
    Goal2Achieved();return;
  }
  if (target3.activeSelf && numberOfTargets>=3 && target3.transform.position.y+1<arenaPosition.y){
    Goal3Achieved();return;
  }
  if (transform.position.y < arenaPosition.y + 0.45f){
    Done();
  }
}
```

Text 15: RollerAgent AgentAction method

The AgentAction method, shown in Text 15, is called every time TensorFlow sends the vectorAction output generated from the trained neural network to the agent running in the environment. Because in our examples all of the machine agents have been moving along the X and Z axis, only two outputs are sent.

The first output is interpreted as the force applied on the X axis, and the second output is for the Z axis. After finding the vector describing the direction of the agent, a force is applied on the RigidBody component of the agent, making it move.

Taking into consideration the fact that the AgentAction method is called periodically, it is a nice place to place some code logic that normally would be put in an Update method.

In this respective method, it is checked if the target spheres have fallen of the arena, but not just be any amount. The center of the target spheres, which is normally 0.5 units above the arena, has to fall 1 unit below the arena before it is considered gone.

This will ensure that the target spheres are clearly out of bounds. While the agent sphere is given a more restrictive condition, if it drop even 0.05 units below its usual height, it is considered gone.

```
void OnCollisionEnter(Collision collision){
  if (!rewardOnTouch) { return;}
  GameObject otherGameObject = collision.gameObject;
  if(otherGameObject == target1) { Goal1Achieved();}
  if(otherGameObject == target2) { Goal2Achieved();}
  if(otherGameObject == target3) { Goal3Achieved();}
}
```

Text 16: Reward on touch alternative behavior

Punishing the agent faster might help it understand that it's most recent action has led to it falling, thus learning faster to not fall off the stage.

An alternative behavior is when enabling the rewardOnTouch flag. It cause the goal achieved methods to trigger instantly on touch instead of pushing them off the stage.

The OnCollisionEnter method is called by the game engine whenever the RigidBody or Collider component starts touching another game object that also has RigidBody or Collider component.

Once called, the game object that has collided with can be obtained and then checked. For each of the target spheres, a check is done, and if true, will call their respective goal methods. The goal methods will handle the rest of the logic.

This alternative was added for users who wanted to train machine agents who only had to collide once with the target spheres.

Obtaining observations

In order to send observations to TensorFlow which will then use them as inputs in the neural network, the developer must implement a method called CollectObservations.

```
if(agentParameters.agentCameras.Count > 0){
  return;
}
```

Text 17: Do not add observations if camera is present

In the CollectObservations method, the developer must call the AddVectorObs in order to send data to the input layer of the neural network.

```
AddVectorObs(transform.position.x);
AddVectorObs(transform.position.z);
float distanceFromCenter = Vector3.Distance(
  transform.position,
  new Vector3(0, arenaPosition.y + 0.5f, 0)
);
AddVectorObs(distanceFromCenter);
AddVectorObs(rBody.velocity.x);
AddVectorObs(rBody.velocity.z);
```

Text 18: Observations of the agent itself

In one training environment, a camera is the only input that is used. In order to keep it fair, no other observations are sent, the neural network must figure everything out only from the camera. (code shown in Text 17).

In the code shown at Text 18, there can be seen how the observations of the machine agent itself are sent. The coordinate and velocity on the Y axis is ignored as the agents or the target spheres are not meant to jump.

A distance value from the center of the arena helps the agent find out how far it is from center, and in how much danger is from falling out.

```
if (eachMissionIsItsOwnBoolean){
  for(int i = 1; i <= 3; i++){
    AddVectorObs(currentMission == i);
  }
}
else {
  AddVectorObs(currentMission);
}
```

Text 19: Mission observations

The next observations that have to be taken into consideration is how the agent is signaled what its current mission is. In Text 19 can be seen that there are two ways to do that. The first one is to send three boolean values, each one of them being either true or false, which is converted to 1 or 0 by the AddVectorObs method. The boolean values will be true if the current mission is equal to index of the boolean value, leaving the rest false.

The second way to do this is to simply send an integer value of 1, 2 or 3 to plainly say the current mission.

These two options can be toggled with the help of the eachMissionIsItsOwnBoolean flag.

The observations regarding the first target can be seen in Text 20. The values that are sent are the targets position, velocity, distance from the agent and the distance from the center.

Such a wealth of information should be enough for a neural network to understand how to interact with the respective target object.

```
Vector3 relativePosition = target1.transform.position -
  transform.position;
AddVectorObs(relativePosition.x);
AddVectorObs(relativePosition.z);
float distanceToTarget = Vector3.Distance(
  transform.position, target1.transform.position);
AddVectorObs(distanceToTarget);
float targetDistanceFromCenter = Vector3.Distance(
  target1.transform.position,
  new Vector3(0, arenaPosition.y + 0.5f, 0)
);
AddVectorObs(targetDistanceFromCenter);
AddVectorObs(target1RigidBody.velocity.x);
AddVectorObs(target1RigidBody.velocity.z)
```

Text 20: First target sphere observations

```
if (numberOfTargets < 2) return;
relativePosition = target2.transform.position -
  transform.position;
AddVectorObs(relativePosition.x);
AddVectorObs(relativePosition.z);


distanceToTarget =
  Vector3.Distance(transform.position,
    target2.transform.position);
AddVectorObs(distanceToTarget);


targetDistanceFromCenter = Vector3.Distance(
  target2.transform.position,
  new Vector3(0, arenaPosition.y + 0.5f, 0)
);
AddVectorObs(targetDistanceFromCenter);


AddVectorObs(target2RigidBody.velocity.x);
AddVectorObs(target2RigidBody.velocity.z);
```

Text 21: Second target sphere observations

```
if (numberOfTargets < 3) return;
relativePosition = target3.transform.position -
  transform.position;
AddVectorObs(relativePosition.x);
AddVectorObs(relativePosition.z);


distanceToTarget =
  Vector3.Distance(transform.position,
    target3.transform.position);
AddVectorObs(distanceToTarget);


targetDistanceFromCenter = Vector3.Distance(
target3.transform.position,
new Vector3(0, arenaPosition.y + 0.5f, 0)
);
AddVectorObs(targetDistanceFromCenter);


AddVectorObs(target3RigidBody.velocity.x);
AddVectorObs(target3RigidBody.velocity.z);
```

Text 22: Third target sphere observations

Similar code is defined also for the second and third target spheres, with the exception that they have an if condition at the beginning of their code section, that will stop sending observations about them if the current numberOfTargets value is small enough.

RollerArena.cs

The RollerArena class is responsible for creating instances of the RollerAgent and target sphere game objects based on the prefab game objects that are set in the Unity Editor.

```
public GameObject rollerAgentPrefab;
public GameObject target1Prefab;
public GameObject target2Prefab;
public GameObject target3Prefab;

private GameObject rollerAgentObject;
private GameObject target1Object;
private GameObject target2Object;
private GameObject target3Object;

private RollerAgent rollerAgent;
```

Text 23: Objects used by RollerArena

In Text 23, there can be seen the objects that are being maintained by this class. A prefab for each of the needed game objects, and the game objects themselves after they are created.

The Start method of the RollerArena class (seen in Text 24), is responsible for instantiating every game object that is needed for the training environment.

```
private void Start() {
  rollerAgentObject = Instantiate(rollerAgentPrefab,
    new Vector3(0, transform.position.y + 0.5f, 0),
    Quaternion.identity);

  rollerAgent = rollerAgentObject.GetComponent<RollerAgent>();
  rollerAgent.arenaPosition = transform.position;
  target1Object = Instantiate(target1Prefab);
  target2Object = Instantiate(target2Prefab);
  target3Object = Instantiate(target3Prefab);
  rollerAgent.target1 = target1Object;
  rollerAgent.target2 = target2Object;
  rollerAgent.target3 = target3Object;
}
```

Text 24: RollerArena Start method

The RollerAgent object was intentionally generated at a specific position of the arena so that it may be valid from the very beginning.

After creating every new instance, the target sphere objects are sent as references to the freshly generated RollerAgent game object, so that it may position them accordingly.

The Instantiate method is provided by the Unity game engine, and it’s functionality is to simply clone whatever Unity Object is sent to it as a parameter.

If that respective Unity Object is of type GameObject, developers may set other values such as transformation and rotation so that it may be immediately placed in the game world.

Dedicating the generation of the game objects to a particular class eases the difficulty of the rest of code in the long run because it enables the developer to better compartmentalize the structure of the project.

RollerArenaAcademy.cs

The RollerArenaAcademy class extends the Academy class provided by the ML-Agents, and it's role is to dynamically generate several arenas so that multiple agent may train at the same time.

The content of this class is shown in Text 25, where we can see how the arena can receive via editor the prefab objects for the arena that it has to instantiate and how many.

```
public GameObject m_RollerArenaPrefab;
public int m_NumberOfArenas;
public override void InitializeAcademy() {
  for(int i = 0; i < m_NumberOfArenas; i++) {
    Instantiate(
      m_RollerArenaPrefab,
      new Vector3(0, -4 * i, 0),
      Quaternion.identity);
  }
}
```

Text 25: RollerArenaAcademy code contents

Sometimes Academy classes override the AcademyReset and AcademyStep methods in order to add more interesting interactions with the possible training environment. But in this project these are not need as the RollerAgent class is already doing most of the important logic already per step and at every reset.

The developer must be careful to set a prefab that will eventually generate some agents that the academy will have to train. And the m_NumberOfArenas must also be chosen according to the PC system that it's going to run on.

Usually more arenas can be generated if the system is particularly strong and/or the training scenarios are simple and non-intensive for the CPU.

Bibliography

ML Agents GitHub page: , , , https://github.com/Unity-Technologies/ml-agents
TensorFlow GitHub page: , , , https://github.com/tensorflow/tensorflow
Making a New Learning Environment: , , , https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Create-New.md
Training ML-Agents: , , , https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-ML-Agents.md
Solving sparse-reward tasks with Curiosity: , , , https://blogs.unity3d.com/2018/06/26/solving-sparse-reward-tasks-with-curiosity/
Training with Proximal Policy Optimization: , , , https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md
PPO: John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, Oleg Klimov, Proximal Policy Optimization Algorithms, 2017,https://arxiv.org/abs/1707.06347
TRPO: John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, Pieter Abbeel, Trust Region Policy Optimization, 2017,https://arxiv.org/abs/1502.05477
ACER: Ziyu Wang, Victor Bapst, Nicolas Heess, Volodymyr Mnih, Remi Munos, Koray Kavukcuoglu, Nando de Freitas, Sample Efficient Actor-Critic with Experience Replay, 2017,https://arxiv.org/abs/1611.01224
Training with Curriculum Learning: , , , https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Curriculum-Learning.md
Udacity Deep Reinforcement Learning: Nisheed Rama, , , https://medium.com/@nisheed/udacity-deep-reinforcement-learning-project-1-navigation-d16b43793af5
Learning Environment Design Brains: , , , https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Design-Brains.md