T.C.

CUKUROVA UNIVERSITY

FACULTY OF ENGINEERING

DEPARTMENT OF COMPUTER ENGINEERING

GRADUATION THESIS

Training a 3D Dynamic Table Model

Orhun Eren Yalçınkaya

2019556064

Advisor: Prof. Dr. Umut Orhan

June, 2024

Contents

Figures

Tables

Equations

Images

Abstract

This study explores the application of deep reinforcement learning to the 'Dynamic Table' model, designed for efficient manipulation of objects. Reinforcement learning, a field that has received significant attention within machine learning and AI, enables an agent to learn optimal strategies for sequential decision-making tasks through trial and error. The Dynamic Table, trained using these advanced techniques, shows promise in various domains, including logistics, industrial production, gaming, and home automation. The model's performance was thoroughly assessed, demonstrating robustness in object placement and movement capabilities within constrained environments. The results underline the potential of the model to improve robotic manipulation tasks, laying a solid foundation for future research and practical implementations. Further studies could extend these findings to more complex real-world scenarios, emphasizing the practical applications of the Dynamic Table model in diverse fields.

Introduction

Reinforcement learning has gained much attention in the machine learning and artificial intelligence communities over the past decade. It is the process by which an agent learns from its environment through trial and error to develop the best strategy for sequential decision-making tasks. This approach has wide-ranging applications in fields such as social sciences and engineering, etc. Presenting the intriguing potential of its agents through the reward mechanism without explicit task specifications, reinforcement learning combines elements from various disciplines, including the trial-and-error tradition in psychology and optimal control theory in engineering. Recent advances, particularly in deep reinforcement learning, have demonstrated superhuman performance in various challenging domains. Efforts to overcome this challenge continue by developing deep RL methods that can quickly adapt to new tasks, which points to a very important goal in this field.[1][2][3][4]

This project aims to provide initial training using deep learning and reinforcement learning techniques for the 'Dynamic Table' designed to simply manipulate the movement of objects. A system designed with dynamic table design and trained with reinforcement learning can be utilized as an optimized system for placing and moving items in narrow spaces or storage areas with simple geometries in storage and logistics. Additionally, it can be used for transporting, placing or rotating parts in industrial production lines. In the gaming industry, it can be employed in designing interactive environments. devices. In home automation, it can facilitate tasks such as moving furniture or relocating items within the home. Furthermore, it can provide opportunities for practical application in the design and control of robotic systems in education and research. This system offers usability in various fields, making daily life and industry more efficient, flexible, and intelligent. In this paper, unless otherwise specified, all information provided pertains to Dynamic Table version 090.

Materials and Methods

Dynamic Table

Dynamic Table has a structure consisting of different numbers of rectangular prism parts that can move up and down independently of each other. The Dynamic Table designed in this project can be shown as follows on the *Image 1*.

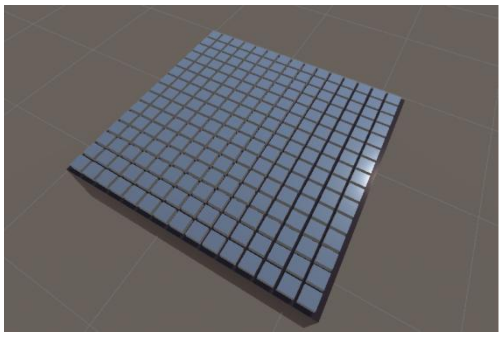


Image 1: Dynamic Table Design [36]

Some of the projects that inspired the Dynamic Table design are inFORM [5], TRANSFORM [6] and shapeShift [7] projects. In these projects, the design does not remain in the digital environment but is processed as a prototype. inFORM is a Dynamic Shape Display, which uses this design for 3D modelling, visualization bar charts, media control, object rotation and movement, etc. purposes. TRANSFORM and inFORM projects transform the tabletop into a dynamic tangible display, a tangible user interface. The name of the table designed in this project can be distinguished from this point since it is not used as a display or as a user interface.

Unity ML-Agents Toolkit

Unity is a real-time 2D/3D development platform that allows developers to create and modify their games or simulations using the engine's features. It was released in 2005 and consists of a rendering and physics engine that powers the engine called Unity Engine as well as a graphical user interface called Unity Editor. At the start, the engine was designed for game development. However, with the opening of Unity Lab, research in various fields begin to be developed. ML-Agents can be considered one of the new researches in this matter. It is a toolkit that was first released in 2017. ML-Agents was released to create a suitable environment for researchers and developers to transform games and simulations into environments where intelligent agents can be trained with the help of machine learning algorithms. [8][9]

The Unity Engine is capable of rendering a wide variety of content such as 2/3D graphics with photo-realistic images, visual effects and shaders, animations, lightning, a physics-based rendering which allows the rendering of rigid bodies, etc., audio and more. In this way, it can provide a lot of information that can potentially be used as training data in reinforcement learning algorithms and can support a wide variety of training types, whether complex or simple. In the Engine, physics and frame rendering are handled asynchronously, allowing the simulations to run at a much higher speed independently of the rendering frame rate. Unity offers a robust and flexible scripting system with C#. The component system allows the creation and management of various instances of agents, policies and environments. It makes it possible to define complicated tasks and, leads to dynamic control of any type of game or simulation and makes it possible to test various training scenarios such as single-agent, simultaneous single-agent, cooperative/competitive multi-agent, adversarial self-play and ecosystems. Environments can be run without rendering, which can be helpful in some situations, such as when testing is needed. With Unity's free license, extensive supporting documentation, and comprehensive tutorials, ML-Agents stands out as a handy and customizable tool for testing, research, or personal projects. This accessibility and support make it a useful choice for a wide range of applications. [10]

How It Works?

ML-Agents has some core entities in the ML-Agents SDK. In the entities numbered and written below, the run process was explained in no order. *Figure 1* below shows an example block diagram of the ML-Agents Toolkit.

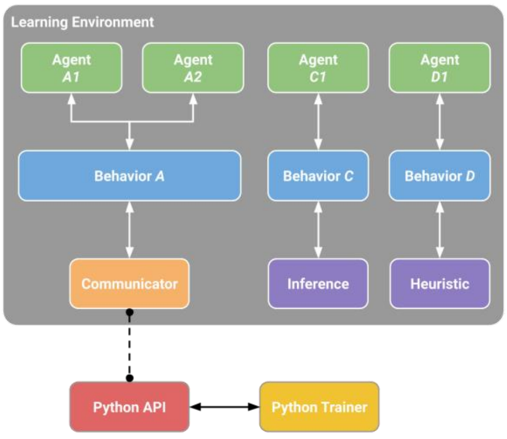


Figure 1 : ML-Agents Working Principle [11]

Agent

The agent is the main component that is trained to collect observations, take actions, and receive rewards. ML-Agents has various sensors that help the agent collect observations. While some of these sensors can collect rendered images of desired quality, some can collect ray-cast results. A reward of any value can be given at any time in the process. It can be calculated through a reward function, which can be of varying complexity and modified at any time during training using the Unity scripting system. Actions are decided by brains and taken into action by agents. [9][10]

Brains

Brains are assigned to perform the decision process by managing the observations and actions. It also defines the states and action spaces. At the start of the training process, It must be connected to the brain with the specified behavior name correctly. [10]

The brain of the agent is in the Behavior Parameters script, which also contains the brain parameters, model, etc. It also decides the behavior type to be used in the run-time. There are 3 Behavior Types: Inference, where the actions are selected by the trained model; Heuristic, where the actions are hand-coded inside the function Heuristic in the agent’s script, and Default, where the actions are selected by the connected brain. The brain's function is to use observation and reward information to instruct the agent on what action to take through communication with the trainer's API. [8]

Policies are labelled with behavior names uniquely. Any number of agents can have the policy with the same behavior name. Those agents will execute the same behavior and share the collected data during training. Additionally, There can be any number of behaviors in the scene which can be useful like multi-agent situations. [10]

Learning Environment

The Learning Environment is one of the high-level components. It contains the Unity scene that includes the environment, which contains the agent or agents and where they observe, act and learn. [11]

Python Low-Level API

Python API contains a low-level Python interface for interacting and manipulating a learning environment. It is not a part of Unity and lives outside. Python API communicates with Unity through an external communicator. The API is contained in a dedicated `mlagents_env` Python package and is used by the Python training process to communicate with and control the Academy during training. It can also be used as the simulation engine for custom machine learning algorithms. [11]

Academy

Each environment has one Academy instance. Academy is used to keep track of the steps of the simulation and is responsible for coordinating and managing agents and their decision-making process by handling the requests from the Low-Level Python API. It can also define the environment parameters/engine configuration, such as speed and rendering quality, frame number needed before taking the next decision, or global episode length. [9][10][12]

External Communicator

External communicator connects the Learning Environment with the Low-Level Python API. It lives within the Learning Environment. [11]

Python Trainers

Python trainers contain all the machine learning algorithms that enable training agents. The algorithms are implemented in Python and are a part of their own `mlagents` Python package. The package exposes a single command-line utility `mlagents-learn` that supports all the training methods and options. [10]

Machine Learning Algorithms in ML-Agents

ML-Agents has four algorithms. While PPO and SAC are the main reinforcement learning algorithms, GAIL and BC are the algorithms are used for imitation learning/behavioral cloning and can be activated to be used alone or in conjunction with reinforcement learning algorithms. [13]

Proximal Policy Optimization (PPO) is the algorithm optimizes the policy using a stochastic gradient ascent. PPO is explained detailed in the section Proximal Policy Optimization.

Soft Actor-Critic (SAC) is the algorithm that maximizes the expected return and entropy to optimize the stochastic policy in an off-policy way. SAC learns one policy and two Q-functions simultaneously. [14] [15]

Generative Adversarial Imitation Learning (GAIL) is an imitation learning algorithm that confronts a generative neural network and an expert discriminator to generate an optimal policy that is optimized by the generative neural network. It can be used with or without environment, reward, and works when there are enough of demonstrations. [15][16]

Behavioral Cloning (BC) is an Imitation Learning algorithm used in combination with PPO, SAC or GAIL to obtain a policy. BC feature can be enabled on the PPO or SAC trainers. Demonstrations of agent behavior can be recorded from the Unity Editor. [15]

Proximal Policy Optimization

Proximal policy optimization, or PPO for short, is a reinforcement learning algorithm used in agent training and is classified as a policy gradient method. The algorithm was introduced by OpenAI in 2017 and now can be considered state-of-the-art in Reinforcement Learning Algorithms, in OpenAI's words. The biggest reason why the PPO algorithm has achieved such good performance is that the previous algorithms were not scalable, inefficient with data, or too complicated. Meanwhile, PPO design focuses on 3 things: simple implementation, sample efficiency, and stability. [15][18][19]

The agent's PPO-trained policy network is the function used to make decisions. PPO's focus lies in optimizing policy improvement with current data while avoiding performance collapse due to overly aggressive steps. In this case, while making these decisions, PPO follows a small-step policy and alternates between gathering data from environment interactions using stochastic gradient methods. So, this method allows for the calculation the direction and size of the steps taken and avoids taking steps that are too large or too small. The reason for this is that while too many steps may lead the agent to be misled, very small steps are not efficient. Unlike standard policy gradient methods, PPO enables multiple epochs of minibatch updates, enhancing its online learning capability. [20][21][22][23]

While Q-learning struggles with simple problems and continuous control benchmarks, TRPO's complexity and limited compatibility with certain architectures does not outperform PPO's approach. Tested across benchmark tasks like simulated robotic locomotion and Atari game playing, PPO demonstrates superior performance over other online policy gradient methods, showcasing a favourable balance between sample complexity, simplicity, and efficiency. [20]

Algorithm Explained

PPO is an actor-critic algorithm in which the actor is the policy that chooses the actions, and the critic evaluates how well the agent’s actions match the expected reward. It updates the policy using the actor-critic structure and applies a constraint that limits these updates through its objective function. An objective function is a mechanism that helps an agent to execute multiple epochs based on a batch of data. The algorithm enhances the training stability through a Clipped surrogate objective function shown in *Equation 1*. PPO’s clipped surrogate objective function clips the coefficient of the ratio function. [22][24][25][26]

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

Equation 1: Clipped Surrogate Objective Function [20]

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}$$

Equation 2: Ratio Function [20]

The function $r_t(\theta)$ refers to the ratio function (*Equation 2*). The ratio function decides if the action at the state is more or less probable in the current policy than the old one. If It was observed that the action becomes less likely under the current policy and within the state than it previously was under the previous state of the earlier policy. [18][24][25]

$$\hat{A}_t = V_t^{target} - V_{\omega_{old}}(s_t)$$

Equation 3: Advantage Function [20]

$\hat{A}_t$ refers to the advantage function (*Equation 3*), which decides whether a specific action of the agent is better or worse than the other possible actions. Here V_t^{target} is computed from the observations and represents the expected discounted cumulative reward. $V_{\omega_{\text{old}}}(s_t)$ is the state value which was is predicted by the state value network and represents the expected return that an agent can achieve within the corresponding state. [18][24][25][27]

$$L_t^{\text{CLIP+VF+S}}(\theta) = \hat{E}_t[L_t^{\text{CLIP}}(\theta) - c_1 L_t^{\text{VF}}(\theta) + c_2 S[\pi_\theta](s_t)]$$

Equation 4: PPO Actor Critic Objective Function [20]

The overall result of the PPO's actor critic objective function (*Equation 4*) combines surrogate objective function and more and maximized each iteration but within limits. In the function, c_1 and c_2 are coefficients, S is the entropy bonus to encourage exploration and $L_t^{\text{VF}}$is a squared-error loss that measures the discrepancy between the predicted value and the target value. [20][28][29]

A* Path Finding Algorithm

A* is a graph traversal and path-finding algorithm which is used to find the shortest path in graph-based search problems. It searches for the path with the lowest total cost by calculating the cost to the current node called g cost and the estimated cost from the current node to the destination called h cost.

It cleverly combines elements from two other well-known algorithms: Dijkstra's Algorithm and Greedy Best-First-Search. A* inherits the strategy of preferring nodes close to the start point from Dijkstra's Algorithm. This involves tracking the cost of reaching each point from the start and ensuring lower-cost routes are explored first. As a result, this approach guarantees that the algorithm identifies the shortest possible path based on the distance travelled from the starting point. A* algorithm's heuristic strategy focuses on nodes, that are closest to the goal. This is done by estimating the cost from each vertex to the goal, also used in the Greedy Best-First-Search Algorithm. As a result, the algorithm is directed towards the goal more directly, potentially minimizing the number of nodes it needs to be explored. [30][31][32]

By combining these two strategies, A* algorithm balances between exploring paths that seem immediately promising and those that are potentially more optimal in the long run. This dual focus enables A* to efficiently find the shortest path. [30][31][32]

The Pseudocode for A*

```
1.   Initialize open and close sets
2.   Add start node to open set
3.   Set all the nodes's g cost to Infinity
4.   Set start node's g cost to 0
```

```
5.   While open list is not empty:
6.      Find the node with the least f cost in the open list and call it the current node
7.      Remove the current node from open list and add it to the closed list
8.      If current node is the goal
9.            Stop searching
10.     For neighbor the current node has:
11.           If neighbor is not in the closed set:
12.                 Calculate the g cost for the neighbor and call it new g cost
13.                 If the neighbor not in open set or new g cost is smaller then the neighbor’s
14.                       Set the neighbors g cost to the new g cost
15.                       Set the neighbors parent’s to the current node
16.                       If the neighbor not in the open set:
17.                             Add neighbor to open set
18.  Initialize the path
19.  Set current node to the goal node
20.  While the current node is not the start node
21.  Add current node to path
22.  Set current node to current node’s parent
23.  Return reversed path
```

Figure 2: Pseudocode of A* Algorithm [30]

Model and Training Architecture

The 3D dynamic model consists of a system designed to manipulate the movement of objects through vertically moving parts. The system is configured to interact with a dynamic environment, enabling the agent to learn effective strategies for object manipulation.

Model Components

Dynamic Table Structure

The table consists of vertically moving quadrangular pieces. Every piece can move individually. The Table has a board under it that keeps the pieces.

Environment

Simulated Environment: This environment replicates near real-world physics to ensure the agent learns applicable skills. There is nothing in the environment but a table, objects on the table and lightning.

Objects: Besides the table, also a red ball (As it is said in the codes and in the rest of this thesis, the product.) that is moved by the table and a green ball (As it is said in the codes and in the rest of this thesis, the target.) which is the destination for the red ball are placed on the Table which can be seen on the *Image 2*. The table is surrounded by walls on 4 sides. The same wall objects are also used to create mazes on the table. Mazes are already prepared and chosen through training builds. (The mazes before version 090 was created randomly in order to some dynamically changing difficulty values during training. Also, in some versions, a frequency value had been calculated to decide when to randomize and change the maze.)

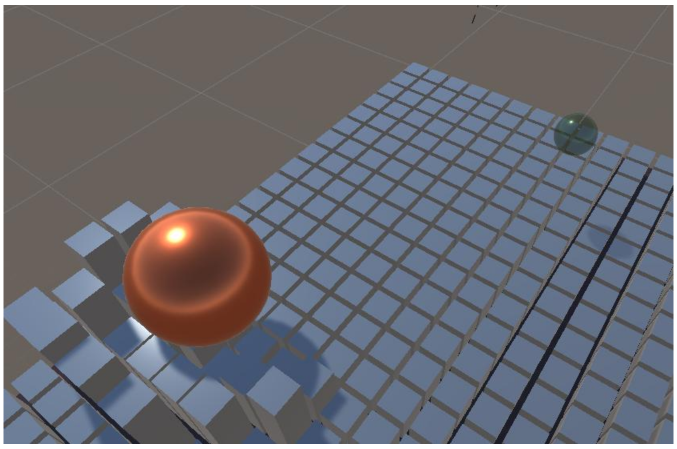


Image 2: Product and Target on the Table [36]

Environment Set: At the start of each episode, the `public override void OnEpisodeBegin()` function is called. This function creates or changes mazes, in which cases mazes are open. Custom training data are collected here via the function `recorder.Add()`. The object's locations are randomized, and A* nodes are reset.

Observation and Action Spaces

Action Space: The action space includes only the movement of the 44 pieces. Actions are continuous values that can be between -1 and 1, including both endpoints, which indicate the speed and direction of each piece's movement. Action spaces vary from version to version. The action space is collected by the `public void OnActionReceived(ActionBuffers actions)` function. In this function, firstly `GetActiveArray()` method is called to find the closest 44 pieces, which will take action. The new positions of those pieces are calculated as `child.localPosition.y + MoveSpeed * actions.ContinuousActions[movingPartsIndex] * Time.deltaTime`.

Observation Space: 61 observations are collected. Vertical position of the 44 pieces, the location and the speed of the product, the location of the target, Ray Perception Sensor results are the collected observations. When the A* algorithm is used, the target location observation replaced with the next target node on the A* path. Observation spaces varies from version to version. The observations are collected using the `public void CollectObservations(VectorSensor sensor)` function.

A* Path Finding Algorithm

The algorithm is managed by `PathFinder.cs`. The node graph consists of 52x52 nodes slightly nested within each other to achieve smoother movement. Each node object has the `Node.cs` script that keeps the h and g costs and neighbors while using the information collected through rays to decide whether the node is too close to a wall in case the node must be closed or is hospitable. The `PathFinder.cs` script recalculates the shortest path every time the product moves a distance of at least one unit, and returns the location of the first node on the new path. In the *Image 3* below, It is presented that the path is colored green, the close nodes are red, and the next node in the path is purple.

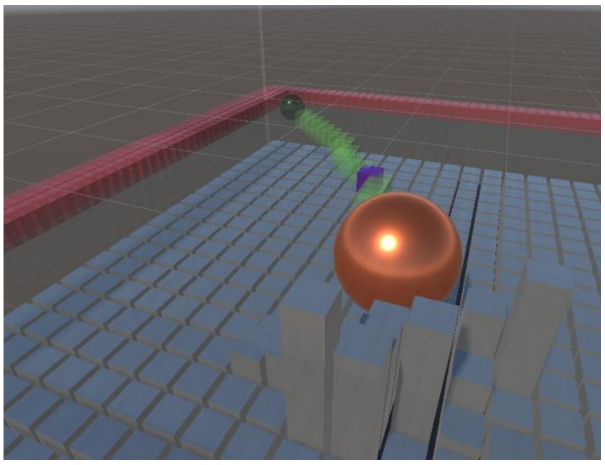


Image 3: A* Algorithm on the Table [36]

Sensors

Product Ray Perception Sensors: Ray perception sensors send rays in a number of directions and return the distance between a specific type of object. In this case, They return the distance between the product and the walls.

A* Node Rays: The nodes send rays to set the node grid which in a state that walls are detected and those nodes that are close to the walls are signed as closed nodes by setting their h cost to Math.Infinity.

Policy Network

The policy network is the function that is to be used to make decisions by mapping the observation space to a probability distribution over actions. [33][34]

Value Network:

It is a neural network that serves to estimate the value function and to predict the expected cumulative reward of a given state. [33]

Reward Function

The reward function represented in *Equation 5* is designed to encourage behaviors that lead to successful object manipulation in which situations the product gets to closer to the target without hitting any wall after getting an action space.

$$\text{reward} = \frac{\text{directionPoint} \times \text{speed}^2 \times \text{multiplier}}{\text{closeness}^{0.1}} + \text{heighPoint} \times \text{multiplier} \times 0.2$$

Equation 5: Reward Function of Dynamic Table 090 [36]

Where,

The direction point is the given point to the movement direction of the product whereas it indicates if it is through to the target or else. Its scope is between -1 and 1. However while It is between 0 and 0.65, It is multiplied by -1.

The speed is the speed of the product. Its minimum value is 0.1 and the expected scope is between 0.1 and 10~12.

The heighPoint is the vertical location of the product. It is added to rewards function to avoid the pushing behavior of the model which was observed in the early versions.

The multiplier is 0.0001. It is added to keep rewards in a smaller scope.

The closeness is the distance between the product and the target. Its minimum value is 0.1 and the expected scope is between 0.1 and 10~12.

Penalties: Penalties are given when the product hits a wall. In this case a reward of -3 is added.

Heuristic Function

The heuristic function randomized each action and sets the action space randomized.

Training Components

Training Configurations

In this section, the configuration of the Dynamic Table model was explained. Detailed information about the configuration settings can be found at unity-technologies.github.io/ml-agents/Training-Configuration-File.

Configuration Setting: Current Value, Description
trainer_type: ppo, Specifies the type of trainer algorithm being used. The information for PPO is available in the Section Proximal Policy Optimization.
learning_rate: 0.0009, Represents the initial learning rate to be used in training.
batch_size: 1024, Denotes the number of experiences processed in each training iteration.
buffer_size: 10240, Refers to the number of experiences accumulated before updating the model parameters.
learning _rate_schedule: linear, Decides how learning rate changes over time.
hidden_units: 64, Indicates the number of units within the hidden layers of the neural network architecture.
num_layers: 3,Specifies the number of hidden layers in the neural network.
beta: 0.005, Strength of the entropy regularization
epsilon: 0.2, Influences the speed at which the policy can evolve during the training process.
num_epoch: 3, Number of passes to make through the experience buffer when performing gradient descent optimization
gamma: 0.99, Multiplier factor for rewards received from the environment.
strength: 1.0, Denotes the discount factor for future rewards originating from the environment. [35]

Training Process

The trainings are completed in the Colab environment (*https://colab.research.google.com/*). To use Colab in order to train models in it, a dedicated server build with target platform Linux is needed. An example training file is shown below; and can be found at the *GitHub Page*.

```
1- Set Required Python Modules
   !rm -r /content/sample_data
```

```
        !pip install mlagents
        !pip3 install torch torchvision torchaudio
        !pip install protobuf==3.19.6 install numpy==1.22.3 onnx==1.12.0
        !pip install tensorboard
    2- Check Environment
        2.1- Test Tensorflow-Pytorch GPU
              (Availability of the GPU devices are checked here.)
    2.2- Test MLAgents
        !mlagents-learn -h
    3-Set Build and Config
        3.1- Set the permissions
            env = [Path to your env]
            exe = 'build' # the name of the exe file
            !chmod -R 755 /{env}/{exe}.x86_64
            !chmod -R 755 /{env}/UnityPlayer.so
            !ls -l /{env}
    3.2- Create configuration.yaml
        (Check the Section Training Configurations)
    4- Tenserboard (It is an optional section which let's you track your model during
training..)
        %load_ext tensorboard
        %tensorboard --logdir /content/drive/MyDrive/results
        %reload_ext tensorboard
    5- Train
        !mlagents-learn /pathto/configuration.yaml
```

Results

The dynamic table model underwent training and testing using various reward functions and configurations. This section presents the findings of the dynamic table model, specifically focusing on its performance in manipulating objects. The following table, *Table 1,* provides a concise overview of the performance measures for several model iterations. The metrics encompass learning rate, beta, lambda, epsilon, number of epochs, hidden units/layers, episode duration, training step counts, and observation/action spaces.

Version	Test Point	LR 0.-	Bet 0.-	Lambda 0.-	Epsilon 0.-	Epoch	Hidden Units / Hidden Layer	Episode Length	Training Step	Observation / Action Space
074	~89.5	0003	005	95	2	3	64/2	~60	~10,000,000	52/44
074	~10	0008	005	95	2	3	40/2	~100	~1,500,000	52/44
074	~39	0009	005	95	2	3	64/3	~50	~5,000,000	52/44
074	~89.8	001	005	95	2	3	64/3	~50	~8,000,000	52/44
074	~87.9	0003	005	95	2	3	128/3	~60	~5,000,000	52/44
076	~88.5	0009	005	95	2	3	64/3	~40	~10,000,000	62/44
076 (TR)	~64.8	0006	005	95	2	3	64/3	~50	~10,000,000	62/44
076 (M)	~45.6	0006	005	95	2	3	64/3	~50	~24,000,000	62/44
077	~77.3	0009	005	95	2	3	64/3	~50	~8,000,000	62/44
077	~74	0009	005	944	4	4	64/2	~45	~10,000,000	62/44
077	~68.1	0009	005	95	3	3	64/2	~50	~4,000,000	62/44
077	~81	0008	005	95	4	3	64/3	~50	~10,000,000	62/44
077	~76.5	0008	005	95	35	3	64/3	~35	~4,500,000	62/44
078	~98.7	0009	005	95	25	3	64/3	~35	~10,000,000	62/44
078	~86.7	0009	005	95	2	3	64/3	~45	~4,000,000	62/44
090	~98	0009	005	95	2	3	64/3	~35	~5,000,000	61/44
090 (M) (A)	~90	0009	005	95	2	3	64/3	~40	<~25,000,000	61/44
090 (M) (A)	~90	0009	005	95	2	3	64/3	~40	<~35,000,000	61/44

Table 1: Results for Different versions of Dynamic Table Models [36]

About the *Table 1*,

(M) means the the dynamic table is trained to move the product towards the target with a known location by finding its way through the maze,

(A) means the A* Path Finding Algorithm was used during training,

(TR) means the target the target was able to move and tended to escape the product throughout training,

Training Points were obtained by testing the trained model for more than 200 episodes and evaluating it out of 100 by getting a point of 1 when the product reached the target and a point of 0 at any other possible end,

The version names are changed in order to increase readability and not all the versions is represented in the table. The original names of the versions which also can be found in the project folder on the GitHub page [36], and are as follows in the table: 074, 074Config1, 074Config2, 074ConfigLRate, 074ConfigUnitsL, 076, 076targetrun, 076maze3fromrun, 077, 077config2, 077config3, 077config4, 077n, 078colab, 078n3, 090, 090rereremaze22, 090rereremaze222.

A comparative analysis of the different model versions reveals key improvements and areas for further enhancement. The table above highlights the differences between versions in terms of performance and features. Notably, versions 078 and 090 demonstrate the highest test points, indicating superior performance in terms of accuracy. However, the importance of version 090 among 078 is explained with the other key observations below.

Observations

There is a positive relationship between longer episode lengths and a higher number of completed episodes, which tends to result in improved performance measures. For example, versions that have episode lengths of approximately 50 and have completed millions of episodes demonstrate strong performance.

The presence of a configuration consisting of 64 units and 3 hidden layers is often observed in the high-performing versions, suggesting that it represents an optimal balance for the model's architecture.

Comparing 090 and 078

Version 078 obtained a test score of around 98.7, surpassing all other tested versions. This indicates significant improvements in managing complex tasks using the dynamic table model.

Version 090 demonstrated outstanding performance, achieving a test score of approximately 98. It consistently performed effectively across many setups.

In contrast, to achieve the difference between the versions 078 and 090, It is important to note that the operational mechanisms of maze systems, with the exception of the 090 version, are as follows: The occurrences of these events are unpredictable, and their levels of difficulty fluctuate dynamically throughout the training process. As the model achieves greater success, its challenges multiply. The increase in complexity is accomplished by growing the quantity of randomly generated walls. Various iterations of mazes have been attempted, with detailed modifications to the difficulty level in each iteration. However, none of these attempts had successful outcomes.

When comparing the 090 and 078 versions, the 078 version achieved a higher score and also facilitated faster product delivery. However, in contrast to the 090 version, the other versions lacked the use of an A* algorithm, which made them insufficient in situations when there was an obstacle blocking the path between the product and the target. As a result, these versions rarely achieved a successful arrival of the product to the target. Despite scoring 88.5, even the 076 version failed to achieve adequate outcomes when trained for these scenarios. Among models trained for the maze system, it achieved the highest score of 45.6, second only to the 090 version. The primary factor that significantly influenced this score, beyond initial expectations, was an error in the adjustment of maze difficulty. Specifically, the difficulty wasn't increasing at an enough rapid rate, whereas the main issue was that it decreased too swiftly. The 090 version successfully achieves the desired results in both pre-prepared mazes and on an empty table.

Graphs

The initial set of graphs includes beta, entropy, epsilon, extrinsic reward, extrinsic value estimate, learning rate and more. The graphs presented below relate to the model that was trained for the maze of the 090 version. The current model was initialized based on the 090 model and underwent training for 20,000,000 steps. In the first 10,000,000 steps, the mazes were used sequentially, while thereafter, they were presented randomly. This model is represented as 090rereremaze222 in the GitHub project documents.

Beta

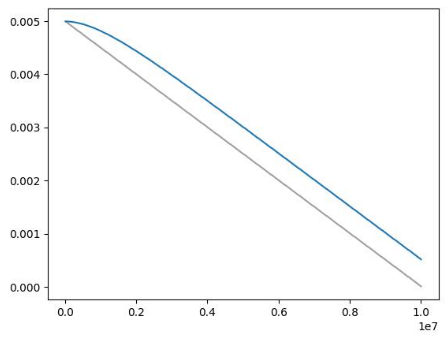


Image 4: Beta [36]

The Beta graph illustrates the change of the beta parameter with training steps. The beta parameter in reinforcement learning is frequently associated with factors such as the balance between exploration and exploitation or the inclusion of regularization components.

The initial beta value is approximately 0.005 and, it decreases linearly over the training process, finally approaching zero. This slight decrease indicates the implementation of a planned annealing process. The fact that beta tends to decrease indicates that the model starts by actively exploring the environment, and as training progresses, it gradually shifts towards employing learned policy. This approach enables the initial collection of various observations, which later serve to improve the policy. By gradually decreasing to almost zero, the model ensures that at the end of training, the policy is being stabilized.

Entropy

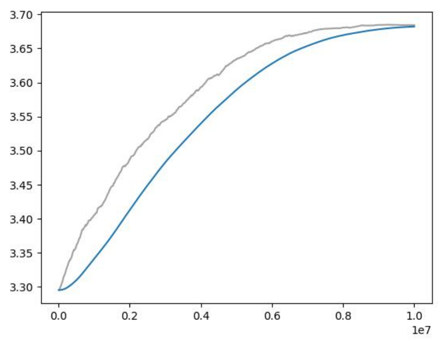


Image 5: Entropy [36]

Entropy is commonly used to determine the level of randomness or unpredictability in the distribution of actions taken by the policy.

The entropy value gradually increases from roughly 3.3 to 3.7 during the training process. The rise is shown as a continuing curve, suggesting a continual increase in the model's process of selecting actions that are more random or unpredictable. A rising entropy indicates that the model is being motivated to keep participating in exploration, even as it gathers knowledge. This may prevent the model from rapidly converging towards inefficient policies.

Epsilon

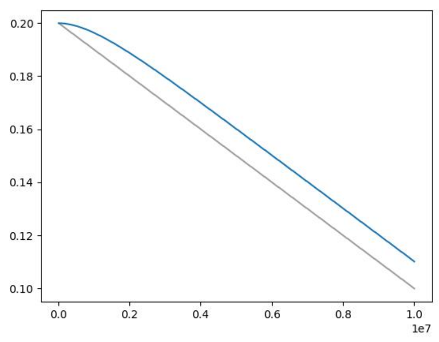


Image 6: Epsilon [36]

The Epsilon graph illustrates the change of the epsilon parameter, commonly related to epsilon-greedy exploration methods used in reinforcement learning.

Like the Beta graph, the epsilon value initially begins at roughly 0.2 and gradually decreases in a linear fashion to around 0.1 during the training period. The gradual decrease of epsilon signifies a planned decrease in the likelihood of picking random actions. At first, the model engages in more exploration (with a greater value of epsilon), but as it obtains knowledge, it rapidly prioritizes exploiting the activities that are known to be the best by decreasing epsilon.

Extrinsic Reward

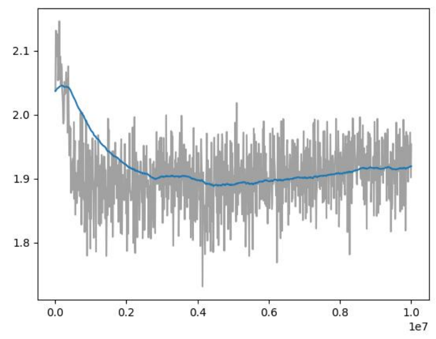


Image 7: Extrinsic Reward [36]

This graph illustrates the pattern of extrinsic rewards that the model was obtained. The extrinsic reward demonstrates a changing pattern, but overall, it decreases from roughly 2.1 to around 1.9 over time. The cycles signify a degree of variability in the rewards obtained during the training. The falling may indicate that tasks or environments become more complex as training advances, leading to reduced rewards. The oscillations are a result of the inherent variability in the environment or tasks that the model is attempting to solve.

Extrinsic Value Estimate

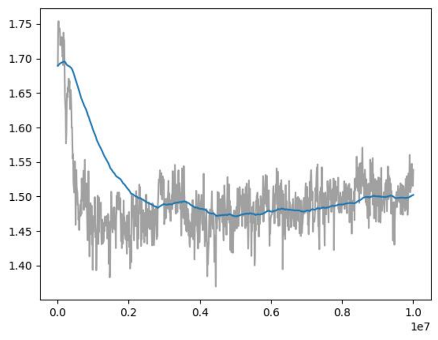


Image 8: Extrinsic Value Estimate [36]

The extrinsic value estimate is a projection made by the model, which predicts the value of future rewards. The initial value estimate is approximately 1.75 and thereafter lowers to roughly 1.45. It then reaches a point of stability with minor changes. The initial decrease in the estimated value may suggest that the model is updating its expectations as it gains a deeper understanding of the environment.

Learning Rate

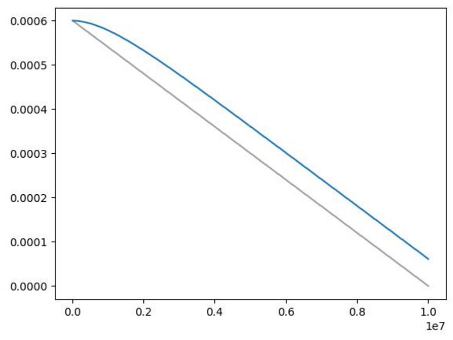


Image 9: Learning Rate [36]

The Learning Rate graph shows the variation in the learning rate over time. The learning rate starts at approximately 0.0006 and decreases linearly to about 0.0001. The linear decay of the learning rate is a common practice in training deep learning models, including reinforcement learning, to ensure stable convergence. By reducing the learning rate, the model makes smaller updates to the policy and value estimates as training progresses, which helps in fine-tuning the learned parameters and avoiding large, destabilizing changes.

Cumulative Reward

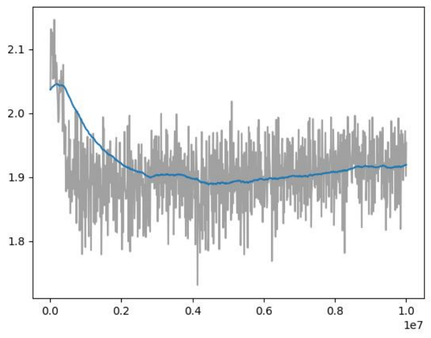


Image 10: Cumulative Reward [36]

The cumulative reward graph demonstrates an initial decline, decreasing from around 2.1 to around 1.8, and subsequently stabilizes at approximately 1.9 throughout the later phases of training. The reward values exhibit significant fluctuations, suggesting that performance varies across different episodes. These fluctuations may be due to the random use of mazes at different difficulty levels during training. The early reduction in cumulative reward indicates the moment when the model started to create mazes. The model's subsequent stabilization around 1.9 suggests that it has discovered a generally constant strategy to make the product make sharp turns at different angles to overcome maze obstacles.

Episode Length

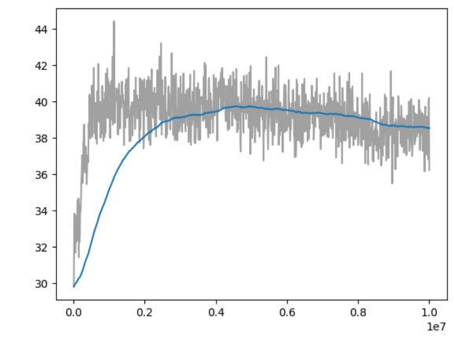


Image 11: Episode Length [36]

The graph representing episode length which is the duration of each individual episode. The duration of each episode experiences an initial rise from roughly 30 to about 42, followed by a slow decline and eventual stabilization at around 38. The decline and stabilization may suggest that the model has improved its actions to accomplish tasks with more efficiency, hence lowering the necessity for extended periods. The variations in the duration of each episode are a result of the differences in the level of complexity of the mazes.

Policy Loss

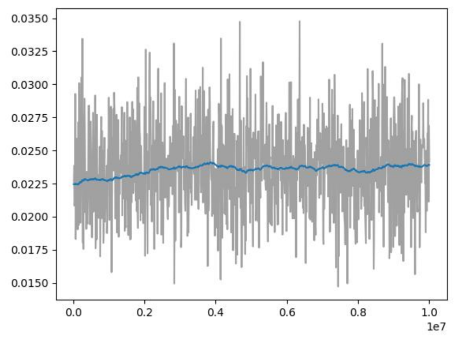


Image 12: Policy Loss [36]

The policy loss graph shows the loss amount captured during policy updates. In the policy loss graph, it can be observed that the values oscillate between 0.0175 and 0.035. There is a slow rise in the mean value of policy loss over time.

Value Loss

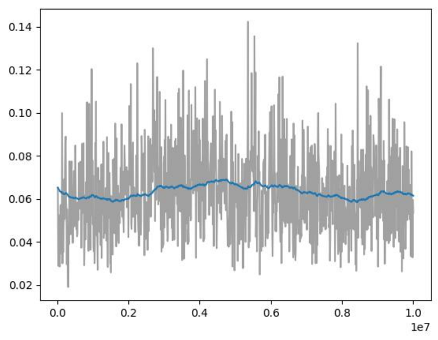


Image 13: Value Loss [36]

The value loss is the loss amount that was produced by the value function during the training process. The fluctuation in value loss ranges from around 0.04 to 0.14, with no noticeable upward or downward trend. The wide range of value loss indicates that the model is experiencing many circumstances that could be a result of randomized the use of mazes while maintaining a stable learning process.

In summary, the graphs indicate that the dynamic table model is being trained using an effective and adaptable technique to achieve an expected performance in handling objects that are vertically moving elements to be used to manipulate the object's movement.

Conclusion

The study presents and evaluates a dynamic table model that is specifically designed for the initial phases of possible future works, particularly in scenarios that need precise control over the movement of objects. The outcomes were deliberated, and selected versions were compared. The model demonstrated robust performance throughout a range of training programs. *Image 14* serves as an example where the moving parts of the table model are evident, while *Image 15* demonstrates a working example of the final version of the 090 model. The training process exhibited examples of stabilization and demonstrated effective learning of skills in terms of the outcomes or training points.

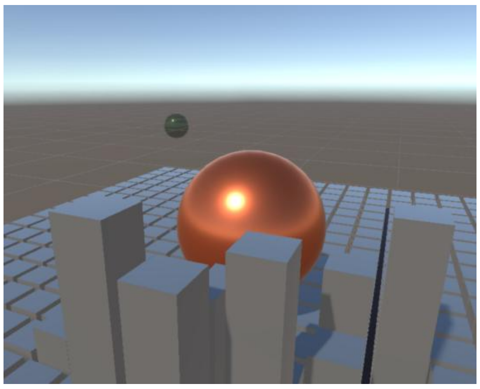
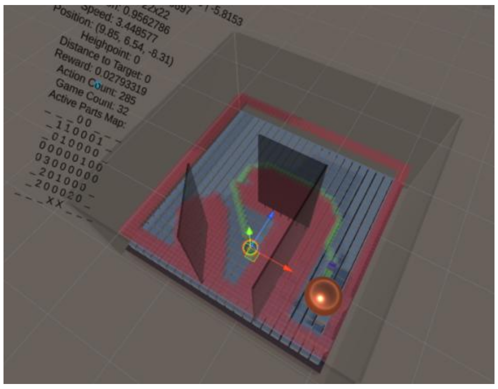


Image 14: Dynamic Table [36]

Extending the model to real-world scenarios that require more complicated and various object manipulation situations could provide significant findings and further demonstrate the success of the model. This study shows the feasibility of using the dynamic table model to enhance the abilities of robotic manipulation using machine learning. The findings obtained from this research establish a solid foundation for future advancements and implementations, supporting the significance and potential of our dynamic table model in performing object manipulation.



```
22x22
0.9562786        -5.8153
Speed: 3.448577
Position: (9.85, 6.54, -8.31)
Heighpoint: 0
Distance to Target: 0
Reward: 0.02793319
Action Count: 285
Game Count: 32
Active Parts Map:
_ _ _ 0 0
_ 1 1 0 0 0 1 _ _ _
_ 0 1 0 0 0 0 _
0 0 0 0 0 1 0 0 _
0 3 0 0 0 0 0 0
_ 2 0 1 0 0 0
_ 2 0 0 0 2 0 _
_ _ _ X X _ _ _ _
```

Image 15: Dynamic Table [36]

References

1. Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement learning: A survey. Journal of artificial intelligence research, 4, 237-285. https://doi.org/10.1613/jair.301
2. Sutton, R. S., & Barto, A. G. (1999). Reinforcement learning. Journal of Cognitive Neuroscience, 11(1), 126-134.
3. Wang, J. X., Kurth-Nelson, Z., Tirumala, D., Soyer, H., Leibo, J. Z., Munos, R., ... & Botvinick, M. (2016). Learning to reinforcement learn. arXiv preprint arXiv:1611.05763.
4. Li, Y. (2017). Deep reinforcement learning: An overview. arXiv preprint arXiv:1701.07274.
5. Follmer, Sean & Leithinger, Daniel & Olwal, Alex & Hogge, Akimitsu & Ishii, Hiroshi. (2013). inFORM: Dynamic Physical Affordances and Constraints through Shape and Object Actuation. UIST 2013 - Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology. 417-426. 10.1145/2501-988.2502032
6. TRANSFORM. (n.d.). Tangible.media.mit.edu. Retrieved May 30, 2024, from https://tangible.media.mit.edu/project/transform/
7. shapeShift. (n.d.). Eric J. Gonzalez. Retrieved May 30, 2024, from https://www.ejgonzalez.me/shapeshift-1/
8. Metz, L. A. E. P. (2020). An evaluation of unity ML-Agents toolkit for learning boss strategies (Doctoral dissertation).
9. Introducing Unity Machine Learning Agents Toolkit. (n.d.). Unity Blog. Retrieved May 30, 2024, from https://blog.unity.com/engine-platform/introducing-unity-machine-learning-agents
10. Juliani, A., Berges, V. P., Teng, E., Cohen, A., Harper, J., Elion, C., ... & Lange, D. (2018). Unity: A general platform for intelligent agents. arXiv preprint arXiv:1809.02627.
11. ML-Agents Overview - Unity ML-Agents Toolkit. (n.d.). Unity-Technologies.github.io. Retrieved May 30, 2024, from https://unity-technologies.github.io/ml-agents/ML-Agents-Overview/#key-components
12. Simonini, T. (2020, March 25). An Introduction to Unity ML-Agents. Medium. https://towardsdatascience.com/an-introduction-to-unity-ml-agents-6238452fcf4c
13. ml-agents/docs/ML-Agents-Overview.md at develop · Unity-Technologies/ml-agents. (n.d.). GitHub. Retrieved May 30, 2024, from https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/ML-Agents-Overview.md#imitation-learning
14. Soft Actor-Critic — Spinning Up documentation. (n.d.). Spinningup.openai.com. https://spinningup.openai.com/en/latest/algorithms/sac.html
15. Proximal Policy Optimization. (n.d.). Proximal Policy Optimization. Retrieved May 12, 2024, from https://toloka.ai/blog/proximal-policy-optimization/
16. ml-agents/docs/ML-Agents-Overview.md at develop · Unity-Technologies/ml-agents. (n.d.). GitHub. Retrieved May 30, 2024, from https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/ML-Agents-Overview.md#gail-generative-adversarial-imitation-learning
17. ml-agents/docs/ML-Agents-Overview.md at develop · Unity-Technologies/ml-agents. (n.d.). GitHub. Retrieved May 30, 2024, from https://github.com/Unity-Technologies/ml-agents/blob/develop/docs/ML-Agents-Overview.md#behavioral-cloning-bc
18. Proximal Policy Optimization. (2023, December 15). Wikipedia. https://en.wikipedia.org/wiki/Proximal_Policy_Optimization

19. PhD, W. van H. (2023, August 16). Proximal Policy Optimization (PPO) Explained. Medium. https://towardsdatascience.com/proximal-policy-optimization-ppo-explained-abed1952457b?gi=b28568be48d4
20. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.
21. Chadi, M. A., & Mousannif, H. (2023). Understanding Reinforcement Learning Algorithms: The Progress from Basic Q-learning to Proximal Policy Optimization. arXiv preprint arXiv:2304.00026.
22. Simonini, T. (2022, August 5). Proximal Policy Optimization (PPO). Huggingface.co. https://huggingface.co/blog/deep-rl-ppo
23. Proximal Policy Optimization — Spinning Up documentation. (n.d.). Spinningup.openai.com. Retrieved May 12, 2024, from https://spinningup.openai.com/en/latest/algorithms/ppo.html#background
24. Bick, D. (2021). Towards Delivering a Coherent Self-Contained Explanation of Proximal Policy Optimization (Doctoral dissertation).
25. Proximal Policy Optimization. (n.d.). Proximal Policy Optimization. https://toloka.ai/blog/proximal-policy-optimization/
26. Proximal Policy Optimization (PPO) Agents - MATLAB & Simulink. (n.d.). www.mathworks.com. Retrieved May 12, 2024, from https://www.mathworks.com/help/reinforcement-learning/ug/proximal-policy-optimization-agents.html
27. Egon, A., & Temiloluwa, F. (2023). Deep Reinforcement Learning: Training Intelligent Agents to Make Complex Decisions.
28. Vayani, A. (2023, April 15). Loss Functions, when to use them? Medium. https://medium.com/@ashmalanis08/loss-functions-when-to-use-them-5b41345b3698
29. How Does Maximum Entropy Help Exploration in Reinforcement Learning? (2020, May 17). Reinforcement Learning. https://rl-book.com/learn/entropy/exploration/
30. Wikipedia Contributors. (2019, March 10). A* search algorithm. Wikipedia; Wikimedia Foundation. https://en.wikipedia.org/wiki/A*_search_algorithm
31. Patel, A. (2019). Introduction to A*. Stanford.edu. http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html
32. Belwariar, R. (2018, September 7). A* Search Algorithm - GeeksforGeeks. GeeksforGeeks. https://www.geeksforgeeks.org/a-search-algorithm
33. DhanushKumar. (2024, February 21). PPO Algorithm. Medium. https://medium.com/@danushidk507/ppo-algorithm-3b33195de14a#:~:text=Policy%20Network%3A%20The%20policy%20network
34. Proximal policy optimization. (2024, May 6). Wikipedia. https://en.wikipedia.org/wiki/Proximal_policy_optimization#:~:text=PPO%20is%20classified%20as%20a
35. Training Configuration File - Unity ML-Agents Toolkit. (n.d.). Unity-Technologies.github.io. Retrieved May 30, 2024, from https://unity-technologies.github.io/ml-agents/Training-Configuration-File/
36. Yalçınkaya, O. E. (2024, May 30). elymsyr/Thesis-Dynamic-Table. GitHub. https://github.com/elymsyr/Thesis-Dynamic-Table