Folding Blocks

1st Alexandre Carqueja
Informatics and Computer Engineering
Faculty of Engineering of UPorto
Porto, Portugal
up201705049@fe.up.pt

2nd Henrique Santos
Informatics and Computer Engineering
Faculty of Engineering of UPorto
Porto, Portugal
up201706898@fe.up.pt

3rd Tiago Alves
Informatics and Computer Engineering
Faculty of Engineering of UPorto
Porto, Portugal
up201603820@fe.up.pt

***Abstract*—This paper's goal is to analyse Reinforcement Learning to solve the prototypical combinatorial puzzle "Folding Blocks", with a solution that aims to be general, independent of board size and efficient. It will break down important topics, such as the choosing of the representation, Reinforcement learning algorithms and correspondent parameters. An in-depth analysis will also be made regarding the efficiency and applicability of the implemented work under several criteria.**

I. Introduction

Reinforcement Learning is a type of learning, in which the agents learns what to do, namely how to make a match between a situation and an action, with the final objective of maximizing the numerical reward received. When taking an action, the agent should not only consider the next immediate action, but also the next future situations and the total reward. In the field of artificial intelligence, classical puzzles, such as the N-Queens or the n-puzzle problem are often used as toy problems, and despite all the differences this puzzle might have, the approaches to these problems helped reaching a conclusion on what is the best way to solve this puzzle. In this paper, we study how Reinforcement Learning can be employed to solve the Folding Blocks puzzle. Thus, we seek to learn a general strategy for finding solutions for all solvable puzzle configurations.

Regarding the organization of the present paper, Section 2 is the Description of the puzzle, containing a brief explanation of the game and all the move restrictions. Section 3 shows and explains the approach chosen to solve problems of this matter, taking into account the Representation of the puzzle, how some puzzles were generated and the best Reinforcement Learning algorithms found. Finally, Section 4 provides analytical and statistical analysis of the final results, regarding the use of different algorithms, reward functions and parameters.

II. Description of the Puzzle

The Folding Blocks puzzle was invented in 2018 by Popcore Games. In this game, groups of tiles of different colors are arranged in a grid, in a way that the majority of the grid is empty. The state of the puzzle can be changed by unfolding one of the colors, in one direction, making the color rotate through an axis and double its size. The color can only be moved if the symmetric tiles of the color are empty, making the set of possible actions, if every color can unfold in any direction a subset of (up,down,right,left) for each color in the board. The goal is to unfold the blocks in a way that the grid is completely filled, with no empty squares. This puzzle has not been target of much investigation, but traditional search algorithms like A* can be employed to find feasible solutions. A puzzle configuration is expressed in two main variables, S and C, where S is the size of the board and C is the number of colors. The state space for each move is thus given by 4^C for each given move.

A. Restrictions

For each move, there are a set of restrictions. Firstly, when the user folds a specific tile, T, it has to have a value of $0 < Val < C$, C being the number of colors of the puzzle. In addition to this, after calculating the symmetric rotation axis, it is only possible to make a move if all the symmetric tiles of the correspondent color are empty. The moves can only be made in 4 directions, up,down,left and right.

More formally, the problem can be expressed in the following constraints:

TABLE I
Puzzle Restrictions

Name	Pre-Conditions	Effects
Flip Up	$Val > 0$ $axis = min(y)$ $Matrix[x][y] == Val$ $\forall piece \in Matrix[$ piece.x,2*axis.piece.y-1] == 0	$\forall piece$ $Matrix[piece.x,$ 2*axis-piece.y-1] = Val
Flip Down	$Val > 0$, axis =max(y) where $Matrix[x][y] == Val$ $\forall piece \in Matrix[$ piece.x,2*axis-piece.y+ 1] == 0	$\forall piece$, $Matrix[piece.x,$ 2*axis-piece.y+ 1] = Val
Flip Right	$Val > 0$, $axis = max(x)$ where $Matrix[x][y] == Val$ $\forall piece \in Matrix$[2*axis-piece.x- 1, piece.x] == 0	$\forall piece$ $Matrix$[2*axis-piece.x- 1, piece.y] = Val
Flip Left	$Val > 0$, $axis = min(x)$, where $Matrix[x][y] = Val$ $\forall piece \in Matrix$[2*axis-piece.x+ 1, piece.y] == 0	$\forall piece$ $Matrix$[2*axis-piece.x+ 1, piece.y] =Val

III. Approach

As mentioned above, a Reinforcement Learning approach was used to solve the Folding Blocks Puzzle. In particular,

we used Proximal-Policy-Optimization, (PPO) and Soft-Actor-Critic, (SAC), making use of "ML-Agents", an open-source project that enables Unity games and simulations to serve as environment for training intelligent agents. All the logic of the game was developed in C#, using Unity for the graphical interface.

A. Representation of the Puzzle

The internal representation of a puzzle is a grid of integers, which may vary in size. The only decision variables associated with the puzzle are the numbers of each tile, these being represented in the following form

- -1, when the tile is null, not being part of the final solution.
- 0, when the tile is empty and part of the filled board.
- Positive Integer, all the other colors have a correspondent positive id.

An example of a puzzle still unsolved, with its correspondent solution can be viewed in the matrix below.

$$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 2 \\ -1 & -1 & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 1 & -1 \\ 1 & 1 & 2 \\ -1 & -1 & 2 \end{bmatrix}$$

As said before, the graphical representation of the game was implemented in Unity, due to the fact that is relatively easy to use and is able to make a decent display of the puzzle without a lot of work. The final representation of the puzzle is the following.

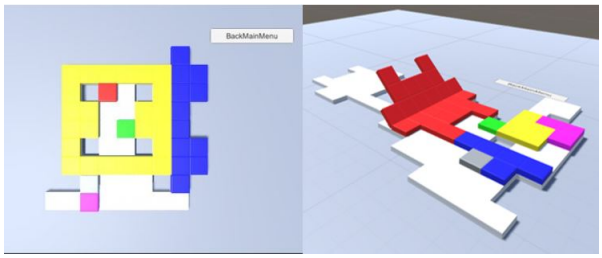

Fig. 1. Graphical Representation of the Game

B. Dynamic Generation

When solving a Puzzle using Reinforcement Learning, if the same puzzles are used for training the agent becomes very prone to overfitting, not being able to solve puzzles that it has never seen, while having almost 100% accuracy on the training samples. To solve this problem and increase the variety of training levels, a random puzzle generator was created.

The implemented solution receives as arguments the size of the matrix (N), the number of colors (C) and the total number of moves (M) to define the difficulty of the puzzle. The first step for the puzzle generator was to create an empty board with a size of $N * N$, and placing a tile of a given type randomly in the board. Secondly, the block is unfolded in a random order X times, where X is a random number that follows $0 <= X <= M/C$.

The next step is to choose a new random color and put it in a random position adjacent to one of the colors already present in the board, with the objective of not creating boards that are completely divided by a null row or column. After this the new color is unfolded Y times (Y being calculated by dividing the number of Moves still to be done by the number of the colors that are still not in the board), so that the total of moves are equal to M. This process is repeated for each new color.

Finally, after fulfilling all the conditions, all the empty squares are set to null and all the colored squares, apart from the starting tiles of each color are set to empty, giving a enormous variety of valid puzzles.

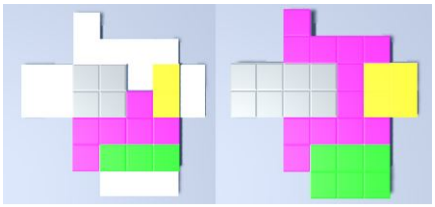

Fig. 2. Simple puzzle randomly generated

C. Reward Function

During the development of this project, various reward functions were implemented, each one having different results. Due to the fact that Folding Blocks can't be easily sub-divided in other smaller problems, like other games, the most promising solutions were the ones that would reward the final result, the extrinsic ones. 1 for Success -0.8 for Failure 0.1 for valid step -0.15 for invalid tiles

The final solution, gives a small reward of -0.15 when the agent does an invalid move, like trying to move a color to a null tile, and a slightly bigger reward when it does a valid move, of 0.1. This way the agent has the tendency to do the least possible amount of invalid moves. When the agent can solve the board, it is given a reward of 1, to incentive to complete the puzzle successfully. On the other hand, it is punished when there are still empty spaces in the board and there aren't any moves possible.

IV. Implemented Algorithm

Proximal Policy Optimization

Proximal policy optimization (PPO) is one of the most successful deep reinforcement-learning methods, achieving state-of-the-art performance across a wide range of challenging tasks. However, its optimization behavior is still far from being fully understood, which may result in performance instability. However, due to it's great successes in recent years in video games, board games and robotics, it was decided it should be considered to solve this puzzle.This algorithm attempts to restrict the policy by clipping the likelihood ratio between the

new policy and the old one. This algorithm is defined by the following formula:

$$L^{\text{CLIP}}(\pi) = \mathbb{E}\left[\min\left(r_{s,a}(\pi)A_{s,a}, \mathcal{F}^{\text{CLIP}}\left(r_{s,a}(\pi), \epsilon\right)A_{s,a}\right)\right]$$

where F^{CLIP} is defined as

$$\mathcal{F}^{\text{CLIP}}(r_{s,a}(\pi), \epsilon) = \begin{cases} 1-\epsilon & r_{s,a}(\pi) \leq 1-\epsilon \\ 1+\epsilon & r_{s,a}(\pi) \geq 1+\epsilon \\ r_{s,a}(\pi) & \text{otherwise} \end{cases}$$

Soft Actor Critic

Soft Actor Critic (SAC) is an algorithm that optimizes a stochastic policy in an off-policy way, forming a bridge between stochastic policy optimization and DDPG-style approaches.This algorithm incorporates the clipped double-Q trick, and due to the inherent stochastic of the policy in SAC, it also winds up benefiting from something like target policy smoothing.

A central feature of SAC is entropy regularization. The policy is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy. This has a close connection to the exploration-exploitation trade-off: increasing entropy results in more exploration, which can accelerate learning later on. It can also prevent the policy from prematurely converging to a bad local optimum.

$$J(\pi) = \mathbb{E}_{\pi}\left[\sum_{t} r(\mathbf{s}_t, \mathbf{a}_t) - \alpha \log(\pi(\mathbf{a}_t|\mathbf{s}_t))\right]$$

Q-Learning and SARSA

Q-learning is an off policy reinforcement learning algorithm that seeks to find the best action to take given the current state. It's considered off-policy because the q-learning function learns from actions that are outside the current policy, like taking random actions, and therefore a policy isn't needed. More specifically, q-learning seeks to learn a policy that maximizes the total reward. Having the following formula

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma max_a Q(s_{t+1}, a) - Q(s_t, a_t))$$

On-policy SARSA learns action values relative to the policy it follows, while off-policy Q-Learning does it relative to the greedy policy. Under some common conditions, they both converge to the real value function, but at different rates. Q-Learning tends to converge a little slower, but has the capability to continue learning while changing policies. Also, Q-Learning is not guaranteed to converge when combined with linear approximation.

In practical terms, under the ϵ-greedy policy, Q-Learning computes the difference between Q(s,a) and the maximum action value, while SARSA computes the difference between Q(s,a) and the weighted sum of the average action value and the maximum. SARSA upadating technique formula is:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t))$$

Both algorithms seemed very promising compared to PPO and SAC, since they are usually applied to more discrete action spaces, like the one from Folding Blocks. Unfortunately, due to the fact that ml-agents still does not support Q-Learning and SARSA, and due to the impossibility of using ml-agents in two of the three members of the group computer's, it was impossible to implement them. Despite all of this, although we think they would both have excellent results, even better than the ones obtained, the solution using more complex algorithms like PPO and SAC is still very interesting and achieved quite satisfying results.

V. Results

Statistical analysis was conducted, in order to extract conclusions regarding the solution's efficiency and applicability. It's separated in three parts: data collected using different algorithms, PPO and SAC, results from using different reward functions using PPO, and finaly changing the parameters of PPO to optimize the learning speed of the agent.

The results presented in the following sections were obtained by testing one hundred times a set of different puzzles.

A. PPO vs SAC

When it comes to the reinforcement learning algorithm, the following tests tried to change the least amount of parameters between them, so it would be easy to compare them and reach a conclusion. Therefore, the following results were compared using the same reward function and very similar parameters, apart from the ones that are specific to each algorithm. The average results comparing both algorithms in the training set are presented in Table II and for the Test set III.

TABLE II
STATISTICS FOR PPO AND SAC IN TRAINING SET

Algorithm	**PPO**	**SAC**
Success Rate	100%	90%
Valid Moves Rate	97.12%	18%
Avg Number Moves	8.0	43.95
Time(s)	2m 52s	3m 55s

It's Possible to conclude that the use of the PPO algorithm is significantly more effective than SAC, both in the training set and in the test set. Although both algorithms achieved excellent results in the training set, PPO success rate was higher and converged faster than SAC. In addition to this, SAC appears to not have learned what were the valid moves at all, while PPO almost exclusively made valid moves. Consequently, the

TABLE III
STATISTICS FOR PPO AND SAC IN TRAINING SET

Algorithm	PPO	SAC
Success Rate	0.5%	0%
Valid Moves Rate	1.8%	0.1%
Avg Number Moves	154.77	448.3

number of moves to solve the PPO were considerably lower. The following graph represents the training of both algorithms, the curved regular line corresponds to the PPO training and the irregular line to the SAC training.

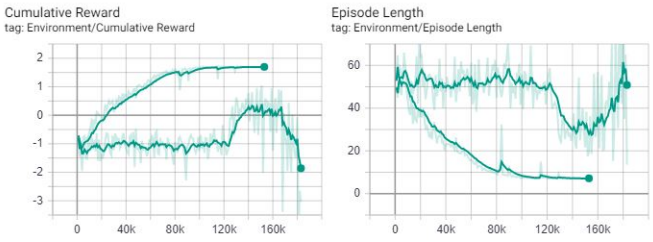

Fig. 3. Algorithm Statistics

B. Reward Functions

After concluding that PPO had overall better results than SAC, the following tests were made only for this algorithm, so it would be possible to show only the most interesting results.

The following tables compare two reward functions for the algorithm PPO. Although they are quite similar, as they both receive a value of 1 when they successfully solve a Puzzle and both receive a value of -0.8 when they cannot, there exists a core difference. While the first reward function punishes the agent with a value of -0.05, to motivate the least number of movements possible, the second one rewards the agent with a value of 0.1, so it is encouraged to do valid moves. The first function also has a bigger penalization for invalid moves, with a value of -0.15, while the second one just punishes with a modest value of -0.05

TABLE IV
STATISTICS FOR BOTH REWARD FUNCTIONS IN THE TRAINING SET

Algorithm	1	2
Success Rate	100%	100%
Valid Moves Rate	98%	93%
Avg Number Moves	8.5	8.8

TABLE V
STATISTICS FOR BOTH REWARD FUNCTIONS IN THE TEST SET

Algorithm	1	2
Success Rate	0.5%	0%
Valid Moves Rate	1.4%	4.42%
Avg Number Moves	213.45	71.84

After analysing both tables, it was concluded that this slight variation in the reward function did not change significantly the final results. While both have a 100% success rate in the training test, the first functions seems to be slightly better than the second one, as it has a higher percentage of valid moves and makes less moves than the second one. In the Test set, the results are quite different, with both functions almost having a 0% success rate, not being able to generalize the network at all. This can also be seen in the extremely low valid move rate, and in the extraordinary number of moves it needs to solve the puzzle. Despite all of this, the second reward function needed much less moves comparing to the first function.

Considering all of this, and by watching the graphic bellow, which represents the algorithm 1 as orange line and the algorithm 2 as blue line, we opted to test the other variances using the first reward function.

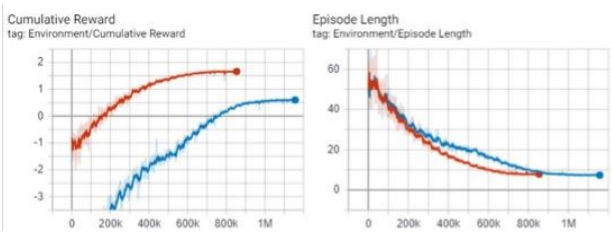

Fig. 4. Comparison of the Reward Functions

C. Parameters

When it comes to the parameters of the algorithm, there are an enormous number of possible variations, being impossible to reach a good result without extensively exploring each one of them. Although there were dozens of tests regarding these parameters, in this section it is only discussed three of the most impacting ones.

After having analysed carefully each parameter and of doing some of the initial tests, we reached some of the following conclusions regarding some parameters. The ones that made the biggest difference were:

- **Batch_Size** - The number of experiences in each iteration of gradient descent should be very low in this case, as we are dealing with a discrete action space.
- **Buffer_Size** - The number of experiences to collect before updating the policy model. In the first tests, the agent was taking an enormous amount of time to complete the puzzle successfully, so this parameter was lowered, sacrificing some stability in the training updates for a faster model.
- **Max_Steps** - The number of steps that must be taken in the environment before ending the training process. This value had to be set to its maximum as the other values were not big enough to train the agent.
- **Sequence_Length** - Define how long the sequences of experiences must be while training. By lowering this value it was possible to have a much faster training without affecting the final solution.

- **Beta** - Strength of the entropy regularization, which makes the policy more random. Initially this value was to large, not being able to solve the puzzles of the training set, this was changed to an intermediate value.
- **Epsilon** - Influences how rapidly the policy can evolve during training. This value was set to the maximum, to maximize the speed of the training process.
- **Lambd** - This can be thought as of how much the agent relies on its current value estimate when calculating an updated value estimates. Surprisingly, the lowest possible value for this parameter was the best solution, relying more on the actual award received in the environment.

The following results represent the refinement of the parameters as we studied and tested everything more extensively, 1 representing one of the first tests made, 3 some of the final tests done and 2 in the middle.

TABLE VI
STATISTICS FOR DIFFERENT PARAMETERS IN TRAINING SET

Algorithm	**1**	**2**	**3**
Success Rate	100%	100%	100%
Valid Moves Rate	98%	93%	97.12
Avg Number Moves	8.5	8.8	8.0
Time	1h:17m	18m	3m

TABLE VII
STATISTICS FOR DIFFERENT PARAMETERS IN TEST SET

Algorithm	**1**	**2**	**3**
Success Rate	5%	0.5%	0.5%
Valid Moves Rate	2.1%	1.4%	1.8%
Avg Number Moves	169.72	213.45	154.77

After analysing the results, it is safe to conclude that the parameters have an enormous impact on the algorithm, with 3 being overall much better than the others. Although the three of them have a 100% success rate, the first algorithm took approximately 70 times more than the final one, while still requiring a higher number of necessary moves to complete the puzzle. On the other hand, the first algorithm was able to generalize to puzzle outside of the training set slightly better than the final one, proving to have some advantages and that we sacrificed some generalization on behalf of speed. Since the values were not satisfying for the third puzzle, only having a 0.5% success rate, it was concluded the benefits of lowering the parameters explained above had extreme advantages regarding speed, while only having slight disadvantages regarding generalization. In figure 5 we can see the reward graphs correspondent to the different training parameters discussed above. 1 - Orange; 2 - Red; 3 - Turquoise

D. Puzzle Difficulty

Given that the provided tests were all made for puzzles with a similar difficulty, this section changes this factor, thus allowing to compare how the agent behaves when confronted

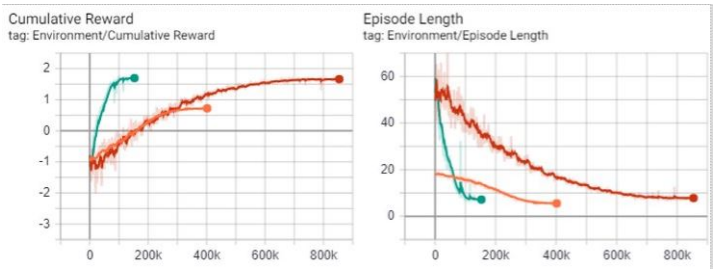

Fig. 5. Parameters Graph

with different Levels. All the previous tests were made using puzzles with a size of $6 * 6$, while the results in this sections are the result of comparing different puzzles with a size of $8 * 8$. The difficulty is defined by the ratio of how many wrong moves are possible, compared to correct moves, all of them being valid. The following table presents four different agents, one that only solves easy puzzles, other medium, other hard and finally one that solves all the puzzles from all difficulties. The results of the tests made in the training set are presented in table VIII and the ones for the test set in table IX.

TABLE VIII
STATISTICS COMPARING DIFFERENT DIFFICULTIES TRAINING SET

Training Set	**Easy**	**Medium**	**Difficult**	**General**
Success Rate	100%	100%	100%	99%
Valid Moves Rate	99%	99%	98%	63.16%
Avg Number Moves	8.2	8.12	9.06	14.08
Time	5m 53s	4m 4s	6m 15s	7m

TABLE IX
STATISTICS COMPARING DIFFERENT DIFFICULTIES TEST SET

Test Set	**Easy**	**Medium**	**Difficult**	**General**
Success Rate	0%	0%	0%	0.2%
Valid Moves Rate	4.1%	0.6%	1.8%	0.2%
Avg Number Moves	73.16	434.19	563.5	654.13

It is possible to observe that the difficulty did not have a major impact for puzzles of the same size, having excellent results in the training set. Despite the small difference, it is possible to notice in the graph below that the easier the difficulty, the sooner it converged. It is also important to notice the strong exploration component after the graph have converged, which can be explained by the random factor explained in the previous section. The General puzzle had slightly worse results, having a 99% success rate and a 63.16% valid move rate. Since this agent can solve all puzzles from the other difficulties, we concluded the results are very satisfying. Fig. 6's graph shows the progression of our training in the different difficulty settings. As expected the easy difficulty converged faster (grey line), next it was the medium (orange line), afterwards the hard (red line), and lastly the training that learned from all the difficulty settings (the blue line). This last

one experienced the slowest convergence and took the longest to get good results, as was to be expected.

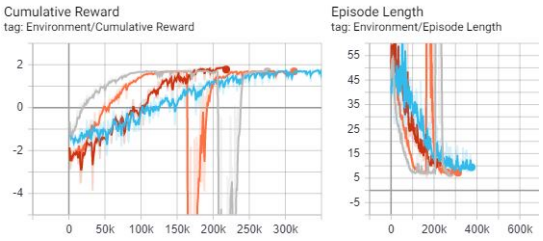

Fig. 6. Difficulty Graph

VI. Conclusions and Future Work

The network turned out to be quiet efficient solving the puzzle, solving the puzzles it trained on in a short amount of time with a 100% success rate. Although Q-Learning and SARSA would be more appropriate for Folding-Blocks, PPO proved to be a valuable candidate, proving once again it can have amazing State of the Art results. During the implementation of this project, we realized that Reinforcement Learning is not as advanced as we initially predicted, being much more difficult to have good results when comparing to supervised learning for example. On the other hand, it also made us realize the potential of this programming area and that is one of the areas with more potential in the future, to solve an enormous amount of problems.

Our knowledge in this kind of area was deeply increased during the development of this work, and opened our minds to new ways of programming.

To sum up,despite all the difficulties, we got used to this programming paradigm and learnt to appreciate both its cons and advantages.

References

[1] S. McAleer, F. Agostinelli, A. Shmakov, P. Baldi , Solving the Rubik's Cube Without Human Knowledge

[2] D. Budakova, V. Vasilev , Applying Reinforcement learning to find the logic puzzlessolution
Technical University of Sofia

[3] B.Bischoff1,D.Nguyen-Tuong and H.Markert1,A.Knoll //Solving the 15-Puzzle GameUsing Local Value-Iteration
Jun 2012

[4] Borga, M Reinforcement Learning Using Local Adaptive Models. ISY, Link oopingUniversity, ISBN 91-7871-590-3, LiU-Tek-Lic-1995:39 (1995)

[5] Noppon Choosri, Solving Sheduling Problems as the Puzzle Games Using Constraint Programming
October 2004

[6] Sutton, R. S., Barto, A. G.: Reinforcement Learning: An Introduction (AdaptiveComputation and Machine Learning). The MIT Press (1998)

[7] Y. Wang and H.He,C.Wen X. Tan .Truly Proximal Policy Optimization. In Proc. of AAAI-94, pp. 301–306.

[8] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The arcade learning
environment: An evaluation platform for general agents. Journal of Artificial Intelligence
Research, 47:253–279, 2013.

[9] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Offpolicy maximum entropy deep reinforcement learning with a stochastic actor. In International Conference on Machine Learning (ICML), pages 1856–1865, 2018.

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