# Identifying how agents behave in the ‘Warlords’ game environment when trained using different multi-agent reinforcement learning techniques. Can interesting behaviours occur or be promoted?

**Joe Down**
Department of Computer Science
University College London
joe.down.20@ucl.ac.uk

**Hector Devie**
Department of Computer Science
University College London
hector.devie.23@ucl.ac.uk

**David Lu**
Department of Computer Science
University College London
david.lu.20@ucl.ac.uk

## 1 Introduction

This project investigates how agents trained using different multi-agent reinforcement learning techniques behave in the "Warlords" game environment. We explore the potential to promote interesting and strategic behaviors through these techniques.

First, let us describe the Warlords game environment. The objective of this game is to defend your base located in one corner of the board, while simultaneously trying to destroy your opponents' bases by launching the ball at them. Each player controls a paddle, which they move up and down to deflect or catch the ball and protect their base. Once the ball touches a player's base, the player is eliminated. The last player standing wins.

Multi-agent reinforcement learning (MARL) holds promise for training agents to excel in complex, competitive environments. "Warlords," a simple yet engaging 4-player Atari game, provides an ideal testbed for exploring MARL techniques. Our central question is: How do different algorithms and reward structures influence the behaviors exhibited by agents within the Warlords environment? Can these algorithms be used to promote the emergence of interesting and strategic behaviors?

The initial phase of our project involved conducting a thorough literature review aimed at providing a comprehensive examination and analysis of existing research and literature pertinent to our inquiry. The literature review primarily focuses on two significant areas: the application of MARL methods in Atari games via PettingZoo API and Python libraries and the exploration of algorithms Q-Learning and Multi-Agent Deep Deterministic Policy Gradient (MADDPG). These discussions are encapsulated within the first section of our report.

Subsequently, we outline the overarching methodology devised to execute our research, which is divided into four distinct sub-sections:

- Data Processing: This subsection elaborates on the process of handling data sourced from the PettingZoo API to ensure compatibility with the algorithms implemented in our study.
- Reward Structure Design: Here, we delineate the development of a custom reward structure aimed at facilitating seamless modifications to explore the emergence of various behavioral patterns.
- Multi-Agent Reinforcement Learning Techniques Implementation: This section provides insights into the implementation of our chosen MARL techniques, namely Q-Learning and MADDPG.

- Performance Metrics: Lastly, we describe the implementation of metrics designed to assess agent performance across different algorithms and configurations.

Concluding the report, we present our results, followed by a analysis and discussion.

## 2 Literature Review

### 2.1 Warlords

We were inspired to investigate the Atari Warlords game environment after reading "Multiagent cooperation and competition with deep reinforcement learning"[9]. This paper investigates performing Deep Q-Learning in a multi-agent setting with the Atari game Pong. In their future work section, they suggest the game Warlords as a possible setting in which a larger (than 2) number of agents may be able to learn to exhibit competitive or cooperative behaviours. They succeeded at producing collaborative and cooperative behaviours between their agents by modifying reward structures, as we had also aimed to investigate in the warlords environment.

### 2.2 Libraries

We used Python 3.8 to code our project due to library dependency issues with later versions. For simulating the Warlords game, we used an existing implementation provided by the petting zoo library[10], with supersuit[11] to perform frame skipping to reduce training times. We used Pytorch[8] as a framework for building and training our neural network. For game frame processing and building our parsed observation vector, we used a combination of numpy[6] and pillow[3].

### 2.3 Deep Q Learning

As mentioned above, "Multiagent cooperation and competition with deep reinforcement learning"[9] has been our first motivation to implement Deep Q-Learning algorithm inside Warlords environment. We found another very interesting research paper "Deep reinforcement learning variants of multi-agent learning algorithms"[2] using DQN variants trained from Atari Games, suggesting again in the future work section to apply these techniques to Warlords. Finally, the excellent and renowned "Reinforcement learning: An introduction by Richards' Sutton"[1] was used as a basis to fully understand how to understand and implement this algorithm.

### 2.4 MADDPG

The MADDPG algorithm originates from the paper "Multi-agent actor-critic for mixed cooperative-competitive environments"[5]. Traditional reinforcement learning methods like Q-Learning face challenges in multi-agent environments because policies evolve continuously, causing the environment to become non-stationary. This variability can destabilize learning and hinder the use of experience replay. To address these challenges, Lowe et al. introduce the Multi-Agent Deep Deterministic Policy Gradient (MADDPG), designed for both cooperative and competitive settings. MADDPG employs centralized training with decentralized execution, allowing the use of global information during training without relying on it during execution. It uniquely enhances actor-critic methods by incorporating insights about other agents' policies into the critic, while the actor uses only local information. The algorithm's ability to approximate and integrate other agents' policies was found to enhance strategic cooperation and competition among agents. Empirical evidence shows MADDPG outperforms traditional methods, enabling agents to develop complex coordination strategies. This makes it seem particularly suitable for environments like the Warlords game, where exploring its potential could significantly enhance multi-agent interaction dynamics.

## 3 Method

### 3.1 Parsing Observations

At each time step of the game, for each agent, PettingZoo provides an RGB image view of the board. In order to reduce the complexity of the observation space we aimed to simplify this into a parsed observation of only key features. In this section we discuss how this parsed observation is produced.

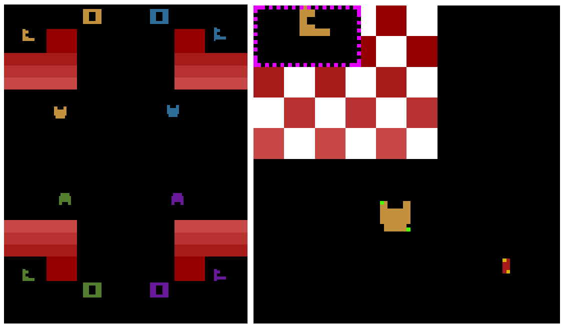
(a) Warlords game board. (b) Detection boundary guide.

Figure 1: Visualisation of the Warlords game board and a key object guide. Green dots indicate paddle bounds, yellow dots indicate ball bounds, pink square represents base area, checkerboard shows unique block location.

The Warlords game board (shown in fig. 1a) is symmetrical horizontally and vertically. Therefore, the game board can be considered identical for each agent under reflections. We exploit this by splitting the board into quadrants and transforming as shown in fig. 2. Each agent's view of the board is uniquely represented by one of 4 arrangements of these quadrants as shown.

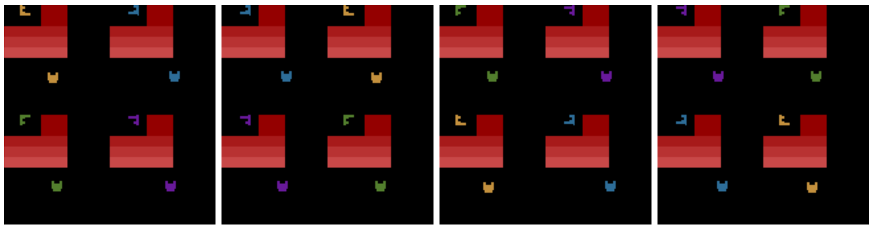
(a) Player 1 board. (b) Player 2 board (c) Player 3 board (d) Player 4 board

Figure 2: Visualisation of how the game board is represented for parsing each player's point of view

Observation vectors are obtained by extracting features from each quadrant. This process is identical for each quadrant after transformation since blocks and the base are placed in corresponding positions, and the paddle now moves along the same path. An example observation for some player at some time step $t$ is shown in fig. 3. The first 4 lines correspond to the statuses of each of the players on the board, while the final line corresponds to the ball status. For each agent the layout is: base status (1 or -1), paddle bounding box (x_min, y_min, x_max, y_max), paddle velocity (x_min_velocity, y_min_velocity, x_max_velocity, y_max_velocity), and the remaining values are block statuses (1 or -1). The ball is represented by (x_min, y_min, x_max, y_max, x_min_velocity, y_min_velocity, x_max_velocity, y_max_velocity).

```
[ 1 51 33 59 41 52 34 60 42  1  1  1  1  1  1  1  1  1  1 -1 -1 -1  1  1  1  1  1 -1  1  1  1  1  1
  1 50 43 58 51 51 44 59 52  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1 -1  1  1  1  1  1  1  1  1
  1 50 33 58 41 51 34 59 42  1  1  1  1  1  1  1  1  1  1 -1  1  1  1 -1  1  1  1  1  1  1  1  1  1
  1 50 42 58 50 51 43 59 51  1  1  1  1  1  1  1  1  1  1 -1  1  1  1  1  1  1 -1  1  1  1  1 -1  1
 68 43 70 47  4  8  4  8]
```

Figure 3: An example observation vector from the point of view of some arbitrary agent. Line 1 is the observation of the segment corresponding to the player, line 2 is their relative player 2, line 3 is their relative player 3, and line 4 is their relative player 4. Line 5 is the ball position boundary.

A breakdown of how each observation is made is attained is broken down in the following subsections. Descriptions will be assisted using fig. 1b:

**Base Status:** A 4 colour palette has been made corresponding to the game's 4 player colours. To determine base status, the pink square region in fig. 1b is checked for pixels of one of these 4 colours. If any pixels of this colour are present, the base is present (1), otherwise the base has been destroyed (-1).

**Paddle Boundary:** Using the same 4 colour palette, pixel coordinates outside the pink square in fig. 1b (to avoid selecting pixels from the base) are checked and must belong to a paddle if they fit the palette. Of these coordinates, a boundary is defined using the smallest pixel x and y values, and the largest pixel x and y values (i.e. 4 integer values). These are indicated by the green dots on the paddle in the fig. 1b.

**Paddle Velocity:** At each time step, the paddle boundary is stored. At the next step, the previous paddle boundary is subtracted from the current paddle boundary to give a velocity (4 values).

**Block Statuses:** A 4 colour palette has been made corresponding to the game's 4 possible block colours. The block area is split into segments according to the checkerboard shown in fig. 1b (the white pixels in this image are purely for visualisation and are actually block coloured). For each segment, it is checked that all pixels are of block colour. If this is the case, then the block must be present (1), otherwise it has been destroyed (-1).

**Ball Boundary:** Rather than by quadrant, ball position is determined by searching the entire image since the block is rarely in more than one quadrant at once. A boundary is determined as the ball's smallest pixel x and y values, and its largest pixel x and y values (i.e. 4 integer values). Since the ball can change colour and is often the same colour as the blocks, a straightforward colour detection technique like for the paddles couldn't be used. Each row and column is counted for pixels which are not player or background coloured. This gives sums per column and row for all pixels containing either a block or the ball. Since stacks of blocks are always a multiple of 8 pixels high, or a multiple of 4 wide (due to the thinner blocks), and the ball is 2 pixels wide and 4 tall, rows and columns containing ball pixels can be determined when a row has a non-multiple of 4 valid pixels and a column has a non-multiple of 8 valid pixels. From these valid row and column values, max and min x and y values can be determined.

A further transformation step is required to map the ball into each player's view of the board. Ball pixel coordinates are reflected according to the board reflections shown in fig. 2.

**Ball Velocity:** At each time step, the ball boundary is stored. At the next step, the previous ball boundary is subtracted from the current ball boundary to give a velocity (4 values).

### 3.2 Reward Structure

Rewards at some time $t$ are generated simply by applying a transformation to a parsed observation (of format in fig. 3) at that time step. In positions corresponding to coordinates, 0s are placed since this does not contribute to rewards. In all remaining spaces in the reward transformation vector, values are assigned to weigh the value of each feature. The result of the reward transformation vector applied to the observation is summed to give a reward value. An additional reward is added under the condition that the player base is the only one alive since this indicates a win. We assigned reward -1 if a player's block is destroyed, -100 if their base is destroyed, 100 if they win, and 0 for all other observations.

### 3.3 Training Models

#### 3.3.1 Deep Q-Learning

The implementation of Deep Q-Learning algorithm can be divided into two parts: Network Class implementation and Agent Class update:

**Network:** To estimate q-values for each action, we implemented a deep fully connected network using Pytorch library. Our model architecture is sequential, containing one input layer (with observation size) and 4 hidden layers with ReLU activation function, and one output layer (with action space size). To get action-value estimates, we call the `forward(self, x)` function, where $x$ is the observation parsed converted as a torch Tensor. Saving and loading methods have also been implemented to load and save the model permitting infinite use of the same parameters over a large number of runs.

**Agent:** Each Agent object has a method to select an action, a replay buffer, and a training method. At each game step, an agent samples an action given an observation parsed by calling the `action(self, observation)` method. This method follows Q-Learning action sampling algorithm. We sample a uniformly random value between 0 and 1:

- If sampled value is lower than $\epsilon$: select an action in the action space randomly,
- Otherwise, select the action with the highest q-value model estimate performing forward.

Where $\epsilon$ is the probability of exploration, i.e. selecting a random action. The exploration is not constant since agents should explore almost always during the first games (i.e when our model is untrained) and decreases as the number of games increases (i.e exploit trained model). Therefore, we set an initial `epsilon = 1`. At each step, update `epsilon *= epsilon_decay`, with `epsilon_decay = 0.99999` until it reaches `min_epsilon = 0.1`. Note that the epsilon decays very slowly. We justify such choice due to the unexpected very slow convergence of our network towards the optimal values.

A replay `Buffer` is attached to each `Agent` object to store transitions of the form $S_t, A_t, R_{t+1}, S_{t+1}$ obtained during each step of a run, using the `add_to_buffer` method. These transitions are then used during the training process performed also at each run by calling `train(self)`. Inside this method, we randomly sample a mini-batch of transitions from the replay buffer. From this mini-batch, we perform an optimization step by performing backpropagation on the Deep Q-Learning model:

```
current_Q = self.model(state_batch).gather(1, action_batch)
pred = self.model(next_state_batch).max(1)[0].view(-1, 1)
next_Q = reward_batch + (1 - done_batch) * self.gamma * pred
loss = self.loss(next_Q, current_Q)
self.optimizer.zero_grad()
loss.backward()
self.optimizer.step()
```

Where `self.gamma = 0.9` is our discount factor. Setting high discount grants higher importance to future rewards. For such game where the main objective of an agent is not getting its base destroyed with no early game strategies that can provide benefits, it seemed an optimal choice. For optimization, we used a Mean Squared Error (MSE) as loss and Adam as optimizer with learning rate $1e^{-7}$.

### 3.3.2 MADDPG

Discussing the implementation of the MADDPG algorithm, it's crucial to highlight that the foundational logic is adapted from the repository available at *maddpg-pettingzoo-pytorch*[4].

The implementation code for `run_maddpg()` is as follows, where we will focus on the parts that differ from our `q_learning run()` method:

```
maddpg = MADDPG(dim_info, capacity=10000, batch_size=64, actor_lr=1e4,
    critic_lr=1e-3, res_dir="./results")
```

In this segment, a MADDPG instance is created, incorporating crucial parameters such as observation and action dimensions per agent, capacity for the replay buffers, batch size for learning, and learning rates for both actor and critic networks.

```
actions = maddpg.select_action(last_observations_parsed)
```

The `select_action()` method diverges from traditional Q-Learning by utilizing the action network of MADDPG to decide actions after a few random steps. It converts observations to a format the model can use, predicts actions with the model, chooses the action with the highest predicted value, and returns a dictionary mapping each agent to their chosen action.

Following this, we feed the updated observations and rewards back into our MADDPG model for learning and updating the agent policies. This step is crucial for adapting the agents to the environment and refining their strategies over time:

```
    maddpg.add(last_observations_parsed, actions, rewards_dict,
        next_observation_parsed_dict, terminations)
```

The add function stores the interaction data of each agent with the environment into their respective replay buffers for later training use.

**Parameters:**

- `last_observations_parsed`: The current observations of each agent.
- `actions`: The actions taken by each agent.
- `rewards_dict`: The rewards received by each agent.
- `next_observation_parsed_dict`: The observations of the next state after taking the action.
- `terminations`: Signals if the episode has ended for each agent

The final step would be the learning and updating processes commence as follows:

```
    if step >= random_steps:
        maddpg.learn(16, 0.99)
        maddpg.update_target(0.01)
```

The `learn` function orchestrates the agents' learning process by updating the critic and actor networks based on a sampled batch of experiences. This process adjusts the agents' policies towards more effective strategies: Subsequently, the `update_target` function applies soft updates to the target networks, ensuring the stability of the learning process by gradually incorporating the latest changes.

### 3.4 Metrics

To assess agent performance, we mainly designed metrics related to survival statistics: win rate, average game position, and average survival time (in game environment steps). These metrics are stored, updated, and printed after each run is finished. We encountered several difficulties during metrics implementation:

- Determining winner: API provides termination status for each players, which can be easily used to determine position for first players eliminated. However, once the game terminates, i.e when one of the two last players is eliminated, both terminates similarly. Only the scores displayed on the board indicate a win. Instead of processing this part of the observation, we use the last ball position to determine which base has been destroyed.
- Add destruction metrics: the metrics mentioned above are purely related to survival statistics, illustrating the defense capabilities of agents. The initial idea was to incorporate destructive metrics to illustrate offensive capabilities, which failed as too large number of edge cases needed to be treated under short time constraint.

## 4 Results

We trained an agent for 10 hours with all 4 agents controlled by and contributing to the model. We then ran 250 games with Agent 1 controlled by the model, and the rest random. We did this for Q-Learning, then MADDPG. Results are shown in table 1. For all random agents over infinite runs, since the game is completely symmetrical, we would expect an average win rate of 0.25 for all agents (equal probability of each agent winning), average position of 2.5 for all agents (equal probability of each position so $\frac{1+2+3+4}{4} = 2.5$), and some same average survival time for all agents.

| Metric | Trained Agent 1 | Random Agent 2 | Random Agent 3 | Random Agent 4 |
|---|---|---|---|---|
| Average Win Rate (Q-Learning) | 0.16 | 0.24 | 0.17 | 0.43 |
| Average Position (Q-Learning) | 2.537 | 2.741 | 2.624 | 2.091 |
| Average Survival Time (Q-Learning) | 1340.227 | 1108.311 | 1235.206 | 1390.149 |
| Average Win Rate (MADDPG) | 0.26 | 0.16 | 0.19 | 0.4 |
| Average Position (MADDPG) | 2.178 | 2.856 | 2.828 | 2.136 |
| Average Survival Time (MADDPG) | 1373.662 | 1042.517 | 1120.933 | 1348.032 |

Table 1: Metrics captured using setup described in section 4. Trained agent 1's model is defined in brackets after metric name.

**Q-Learning Results:** Weak results. Our trained agent win rate is the lowest and worse than expected for random. However, its average position and survival time is better than Random Agents 2 and 3, and worse than expected for random. It's worth noting however that it also reduces these two opponents' average position metric to significantly worse than expectation. In all metrics, Agent 4 performed significantly better than all others, and better than expectation. We conclude that our model performs well against adjacent agents, but completely under-performs against its diagonal rival Agent 4 which it in fact manages to assist.

**MADDPG Results:** Slightly less weak results. Our trained agent win rate is second, better than Q-Learning in last, and slightly better than expected for random (though within margin for error). Like Q-Learning, its average position and survival time is better than Random Agents 2 and 3, however it also achieved a slightly larger average survival time and a similar average position compared to Agent 4. We conclude that, like Q-Learning, this model performs well against adjacent opponents. Unlike Q-Learning, it performs similarly to Agent 4 for all metrics but win rate where it significantly loses out.

**Anecdotal Observations:** Watching either model's agent play, we noticed it would stick to one of three positions: directly facing their relative player 2's base on the right, directly facing their relative player 3's base at the bottom, and directly facing their relative player 4's base on the diagonal. The position facing player 4 was chosen far less frequently. This makes sense as these are the three positions from which an agent can directly shot at an opponent base, or have their base shot at by an opponent. We saw the Agent would often, though not always, move to the position opposite an opponent if that opponent took up the position opposite one of the three mentioned positions. These observations are reasonably consistent with the metrics we observed, since the agents it performed well against correspond to the more frequent positions taken, while the agent it performed poorly against is in opposite the less frequently taken position.

We were unable to observe any clear intelligent ball bouncing behaviour as we had hoped. This was observed by Tampuu et al. [9] for the simpler, but similar, game of Pong.

## 5 Discussion

Since our results were weaker than we had hoped, this section will discuss the difficulties we faced and how further attempts could aim to avoid these problems. While the strength of both models against relative agents 2 and 3, and weakness against agent 4, could perhaps indicate emergent exploitation and cooperation, we do not believe that our anecdotal observations and our inability to modify behaviour using different reward structures support this. This makes it difficult to provide any answer to our research question beyond the observations stated in section 4.

### 5.1 Implementation

Implementing the Q-learning algorithm and neural network proved relatively simple. Within a few days we had these fully implemented. Interpreting with the game environment proved the main implementation difficulty. We initially thought this would be easy since the game setup is relatively simple, however we later identified lots of edge cases which had to be dealt with. This took focus away from the actual training and tuning process, slowing down our acquisition of results. Some difficulties were as follows:

- Screen flashing when bases are destroyed or a block is destroyed at high speed resulted in mis-detections in our board parsing logic.
- Petting zoo stops providing outputs for agents which are dead on the frame before their death. This meant that for a while our agents were not seeing this state. We had to create an artificial dead state immediately following the agent's termination using the previous observation with the base status inverted from alive (1) to dead (-1).
- When the full game ends, again petting zoo terminates the frame before the result is shown. We therefore had to artificially create a final state at the game end where the second place player's base is manually switched to dead (-1). This required us to use ball detection to determine which player base is destroyed at the last step based on the quadrant it was in.

### 5.2 Model Parameterisation

The largest difficulty came with trying to parameterise both the Q-Learning and MADDPG neural networks. In both cases, learning rate needed careful adjusting to avoid either neural network weights exploding, or learning being too small to reduce loss. In our best cases for Q-Learning we found it took at least 5-6 hours before small losses were achieved, making this process difficult. With MADDPG, we found the problem was further complicated since both the actor and critic have a separate learning rate which both need to be sensibly picked. We failed to identify any combination which resulted in a long term reduction in loss after the first few steps, even when left for a long time.

For Q-Learning we also had to select an appropriate $\gamma$ value (discount factor). We found that with larger values for this we were not able to achieve sufficiently small losses to indicate our model was converging towards a reasonable solution. With smaller $\gamma$ values, achieving reasonably small ( 0.5 MSE) losses was possible, however we were not viewing particularly intelligent behaviours. This makes sense since we are restricting the models consideration of long term rewards and therefore it will not care for long term future rewards (i.e. winning).

### 5.3 Game Limitations

One explanation for our results is that the game is perhaps too complex for the techniques we applied. Since there are 4 agents, each with a large number of possible positions, as well as blocks to track, the state space is large and there is a lot of unpredictability, at least relative to the pong example of Tampuu et al. [9].

### 5.4 Potential Improvements

We suggest a few possible improvements which could lead to stronger future results.

Using a larger neural network architecture could perhaps yield improvements since our models were perhaps not expressive enough. This would require more powerful hardware than we had available to train within a reasonable time-frame.

We also suggest implementing some form of automatic hyper-parameter tuning. This would require training a large number of differently parameterised models which could not be done in our available time-frame.

A final suggestion is an alternate approach to the solution implementation. Rather than training a model to select an action to given the current state, perhaps a model could be trained to decide an optimal paddle position and whether the ball should be held. From this optimal position, an appropriate action can then be selected to move the paddle toward the position.

## 6 Code

Code for this project is available with a README at the following GitHub repository: `https://github.com/COMP0124-Group-9/Coursework-2`[7]

## References

[1] Andrew G Barto. “Reinforcement learning: An introduction by Richards’ Sutton”. In: *SIAM Rev* 6.2 (2021), p. 423.

[2] Alvaro Ovalle Castaneda. “Deep reinforcement learning variants of multi-agent learning algorithms”. In: *Edinburgh: School of Informatics, University of Edinburgh* (2016).

[3] Alex Clark et al. “Pillow (pil fork) documentation”. In: *readthedocs* (2015).

[4] Git-123-Hub. *maddpg-pettingzoo-pytorch*. URL: `https://github.com/Git-123-Hub/maddpg-pettingzoo-pytorch`.

[5] Ryan Lowe et al. “Multi-agent actor-critic for mixed cooperative-competitive environments”. In: *Advances in neural information processing systems* 30 (2017).

[6] Travis E Oliphant et al. *Guide to numpy*. Vol. 1. Trelgol Publishing USA, 2006.

[7] *Our project GitHub repository*. URL: `https://github.com/COMP0124-Group-9/Coursework-2`.

[8] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019).

[9] Ardi Tampuu et al. “Multiagent cooperation and competition with deep reinforcement learning”. In: *PloS one* 12.4 (2017), e0172395.

[10] Jordan Terry et al. “Pettingzoo: Gym for multi-agent reinforcement learning”. In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 15032–15043.

[11] Justin K Terry, Benjamin Black, and Ananth Hari. “Supersuit: Simple microwrappers for reinforcement learning environments”. In: *arXiv preprint arXiv:2008.08932* (2020).