Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Swarm Intelligence Cup

Distributed Systems Lab

Marc Lundwall

mlundwall@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Benjamin Estermann, Florian Grötschla
Prof. Dr. Roger Wattenhofer

September 10, 2023

Acknowledgements

I would like to thank my supervisors, Benjamin Estermann and Florian Grötschla, for their endless support throughout this project. I appreciated very much our weekly meetings and their invaluable advice. Even though our results weren't always as good as we had hoped, their optimism was contagious, and motivated me to keep trying even harder. I could not have hoped for better supervisors.

I would also like to thank Prof. Dr. Roger Wattenhofer for giving me the opportunity to work on this particularly fun project.

Abstract

The Swiss AI company Lab42 has organised popular contests around machine learning, such as challenges with the ARC corpus to learn how to solve some pixel-based logic puzzles automatically. Their most recent objective was launching the Swarm Intelligence Cup, where the aim is to demonstrate emergent behaviour: by having hundreds of small agents acting locally, a harder, global objective might be solvable. Lab42 have asked us to help design their Swarm Intelligence Cup. We first created a custom environment to investigate various strategies for competitors. Our focus quickly turned to using neural network architectures, to see if reinforcement learning could be competitive with a manual, hardcoded approach. Although we were not quite successful, we present here some lessons learned in improving the performance of reinforcement learning agents, and new neural network architectures adapted to process communication more effectively.

Contents

Chapter 1

Introduction

Reinforcement learning has emerged at the forefront of the AI revolution, being associated with highly mediatised events such as the defeat of the best human Go player Lee Sedol. However, reinforcement learning remains difficult to access for even programming-minded people. The complexities involved in setting up the environment, choosing the algorithms to use, the observation and action spaces, as well as hand-crafting a custom reward function, all turn a fun idea of trying a beat a video game with AI into a nightmare of sleepless nights. Our objective here is also to make reinforcement learning more accessible and fun to general programmers, or even beginners learning to code.

Swarm intelligence, the theme of the competition, is all about achieving great things through the combined efforts of many small and weak agents. Alone they are useless, but together become more powerful than the sum of their parts. Simulations of swarm intelligence are always fascinating to observe: something simple, like Boids, where each bird follows trivial rules, becomes a hypnotic simulation of birds' flocking behaviour. This leads us directly into the subfield of RL known as multi-agent reinforcement learning, or MARL, where numerous agents work collaboratively, or even competitively, to achieve their objective. This is what we have decided to focus on in this project.

Chapter 2

Our Bee Environment

The first step was creating a custom MARL environment, which we will highlight here, as well as the challenges that we prepared.

2.1 Related Work

We were inspired by previous competitions and existing frameworks.

2.1.1 MIT Battlecode

Battlecode [1] is MIT's biggest and longest running programming competition, where competitors code the behaviour of an army of agents, to destroy the opposing team's army. This includes devising a strategy and tactics, as well as implementing algorithms for pathfinding, efficient communication, and resource management. The game mechanics are often very fun, such as in the 2017 edition, where farmers plant trees to increase the production of in-game currency, and soldiers shoot bullets to destroy the enemy, as seen in Figure 2.1. However, each agent has a very limited amount of computation available, so machine learning isn't a feasible approach.

2.1.2 MAgent2

MAgent2 [2] is a library to create simple competitive environments, where agents are trained through reinforcement learning. There are multiple objectives available, including pursuit, resource gathering, and battle. The agents learn to collaborate to achieve their objective, such as working together to block escape routes in the adversarial pursuit game shown in Figure 2.2. However, there is no communication, so the extent of the collaboration is limited, and at a local level only.

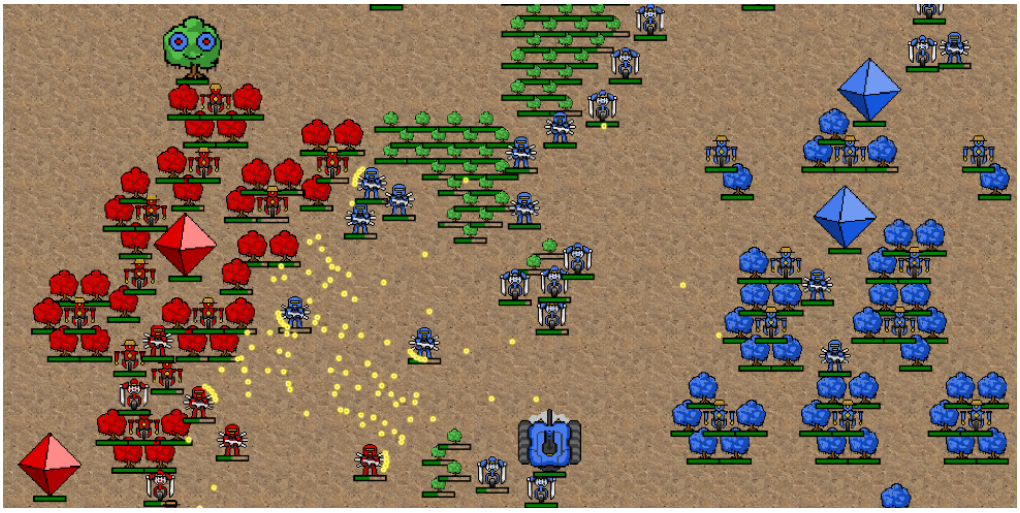

Figure 2.1: MIT Battlecode, 2017 edition

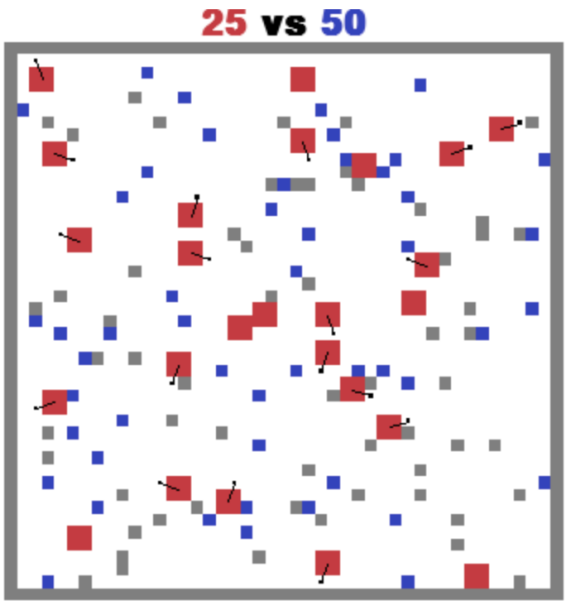

Figure 2.2: MAgent2: Adversarial Pursuit

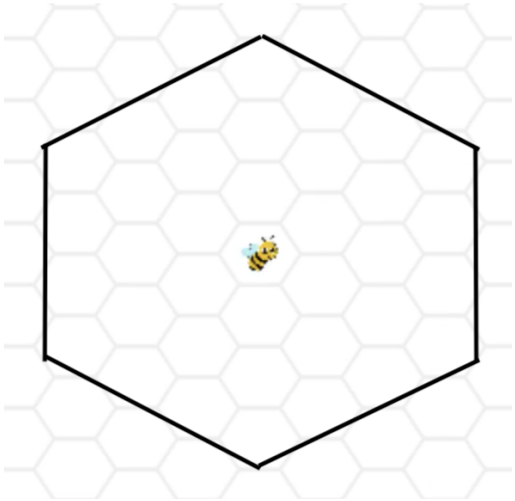

Figure 2.3: Bee Vision Range

2.2 Requirements

We wanted an environment where both manually coding an agent's behaviour and using machine learning are valid strategies, and we would try to tweak what we can to make it so. Communication between agents is also preferred: this allows for more interesting challenges and strategies. In other words, we would be combining the learned approach of MAgent2 with the communication strategies of Battlecode. Although we are open to challenges pitting players directly against each other in the future, we decided to start by proposing simpler single-player tasks. Multiple challenges are needed, some harder than others, potentially culminating in a combination of challenges where agents need to adapt to the most pressing objective.

2.3 Bees

Inspired by Lab42's poster for the Swarm Intelligence Cup, we have developed a bee-themed environment and challenges. The bees are our agents, and each turn can move one step in any of the six cardinal directions of the hex grid, provided that it is free. During the turn, the bees sequentially make their 'move', to avoid any conflicts if two bees choose to go to the same cell. The bees have a limited field of view: they only see up to a Manhattan distance of 3 cells, as seen in image 2.3. For reference, the environment is always of size 20×20.

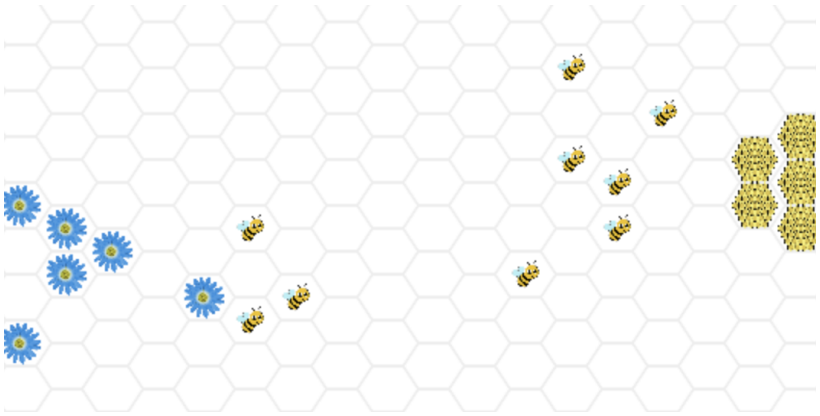

Figure 2.4: Bees, flowers, and a hive

2.3.1 Making Honey

The first challenge adds two features to the map, each in a random position: flowers, and a hive. The aim is for the bees to find the flowers, collect their nectar, and bring it back to the hive, where it is transformed into honey. The nectar in the flower regenerates, and the aim is to make as much honey as possible in a fixed number of steps. In an ideal scenario, bees would try to explore the map, and share with other bees the locations of the flowers and hive, before going back and forth between the two locations, as can be imagined from Figure 2.4.

2.3.2 Eliminating Wasps

The other challenge conceived is about eliminating any pesky wasp that may show up. For this, bees need to surround the wasp, which causes it to overheat and suffocate after 3 turns. The objective here is to eliminate all wasps as quickly as possible. It takes at least 6 bees to eliminate a wasp, so bees should coordinate to get help from the other bees, and encircle the wasp, giving it no way out. A yet unsuccessful attempt in shown in Figure 2.5.

2.4 Implementation

Our environment was created by combining a few useful frameworks.

We use Mesa [3] as our simplified 'game engine'. Originally designed for Agent-Based Modelling (ABM), this project allows for easily simulating the environment described so far. It takes care of all the details that are not particularly in the scope of this project, such as creating the map as a hex grid, placing agents on it, defining each agent type's behaviour, scheduling them during a run, and vi-

Figure 2.5: Bees attempting to kill a wasp

sualizing the simulation in real time through a web browser. Being Python-based, it is straightforward to use, and the performance tradeoffs with a considerable number of agents aren't too relevant to us because the bottleneck would be the model inference time anyway, which needs to be done sequentially on every agent.

The standard way of preparing environments for reinforcement learning is by wrapping them with OpenAI's Gym API, now known as Gymnasium [4]. However, Gymnasium only works with single-agent reinforcement learning, and we are dealing with multi-agent reinforcement learning (MARL), so we use the PettingZoo API [5]. This standardised API makes our environment plug-and-play with the various reinforcement learning frameworks.

The generally recommended reinforcement learning framework is StableBaselines3 [6], but once again, StableBaselines3 is designed for single-agent reinforcement learning. Even if in theory there are ways around this to adapt it to multi-agent environments, we prefer using Ray's RLlib [7], for its multi-agent compatibility (it works almost directly with our PettingZoo-wrapped environment), industry-grade algorithm implementations, and easy distributed computing on a cluster.

Chapter 3

Reinforcement Learning

3.1 Algorithm

We use OpenAI's Proximal Policy Optimization (PPO) algorithm [8] with parameter-sharing. Each agent uses the same model weights, and all observations and rewards contribute to its improvement. This algorithm is considered the state-of-the-art in reinforcement learning, and supposedly works well out-of-the-box with a wide range of environments, requiring little hyperparameter tuning. Through our testing, we found that a learning rate between 2.5e-5 and 5e-5 works slightly better, and we use a schedule to slowly lower it throughout training. The default training batch size of 4000 in RLlib's implementation was reasonable. This corresponds to running four times the environment, stopping each time after 100 rounds, which is 1000 agent steps (also known as the episode length). Every episode's environment is randomly generated each time.

The observation is completely flattened and fed through a fully-connected neural network (FCNN), having by default 2 hidden layers, each with 256 neurons, with the action as the output layer. Discrete values (such as directions) are one-hot encoded, and continuous values are gives two floats: a mean, and a standard deviation.

3.2 Rewards

Choosing the appropriate rewards is essential to ensure that the agents learn the right thing.

3.2.1 Goal-only Rewards

The most straightforward reward strategy would be to only give a reward to the agent when it achieves the goal we set for it. In our case, this would be only when it brings back the nectar to the hive to make honey, or when it kills a

wasp. This is theoretically the ideal strategy: we ensure that there is no way for any unintended behaviour to surface. However, it is impossible in practice, since this reward would be too sparse, and wouldn't ever happen when starting with a 'random' policy.

3.2.2 Subgoal Rewards

The natural solution here is to give rewards when subtasks are achieved. These smaller goals need to be able to happen sometimes with random policies, for the agent to have the opportunity to learn at the beginning of training. For us, this includes giving an additional reward when the bee takes nectar from the flower, or when it stands directly next to a wasp. We extend this for wasps by increasing the size of this reward the more the bee is surrounded.

3.2.3 Reward Shaping

Reward shaping takes the subgoals reward system further, where smaller rewards are always given to agents as they seem to get closer to the goal. The standard way of formulating this reward function is through a potential-based shaping function [9], and computing the difference between the current and the previous potentials:

$$F(s, a, s') = \gamma\Phi(s') - \Phi(s)$$

Here, $F(s, a, s')$ is the reward given for taking action a when transitioning from state s to state s', Φ is the potential function (which can be interpreted as how 'close' the agent is to its goal), and γ is the discount factor, between 0 and 1. γ makes the reward seem more urgent, since the net reward of going in circles becomes negative, but PPO already has such a heuristic built-in, so we kept the default $\gamma = 1$ to skip discounting the rewards. A reasonable choice for Φ is $-distance(target)$, setting $target$ to the nearest wasp, flower, or hive depending on the bee's current objective, and that it what we have chosen in our experiments.

3.2.4 Curriculum Learning

An alternative way of dealing with the sparse goal-only rewards without the problem of accidentally introducing unwanted behaviour by the agent (if it focuses too much on subgoal rewards), is with curriculum learning: we start the training in a purposefully easier environment, and gradually make it harder. A simple way of doing this in our case is by starting with smaller maps, such as 10×10, and slowly increasing it every time the learning stabilises, until we get to our default map size of 20×20.

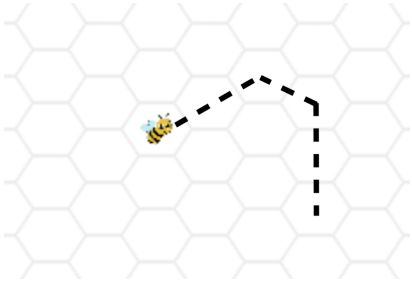

Figure 3.1: A bee's 'trace'

3.3 Observations

We have to show our bee its neighbouring environment, in a way in which it can best extract the information it needs to make decisions.

In video games, reinforcement learning is usually done by giving the pixels of the screen directly to the agent, which uses a CNN to extract the relevant details. In our case, we have access to the internal state of the game, so we can skip the CNN and directly encode the elements in the cells.

3.3.1 Trace

In reinforcement learning, agents only have access to their current observation, and have no 'memory' beyond that. This can be a problem in our case, since a bee might want to know the path that it has taken in the last few rounds. This is particularly important for the exploration of the map: if the bee is completely alone, then its best bet is to continue in its current direction until it hits an obstacle. The usual way of adding this information is through frame-stacking, where the previous 'frames' of the game (usually 4) are also added to the observation. However, this is very inefficient in our case, because we don't need to estimate velocities or predict trajectories. Our solution was to add a 'trace' to the bee's observation, as shown in Figure 3.1, reminiscent of the retro arcade game 'snake'.

3.3.2 Channels

There are 37 cells around a bee, including the centre. The simplest way of representing these cell's contents is by using an ordinal number for each type of agent surrounding the bee. However, this kind of label-encoding usually implies some ordering between the types, which makes no sense here. Therefore, we chose to represent each type of agent in a different channel, using 'presence bits' to indicate a type's presence at a specified location. So for each of the 37 locations, we need a vector of 6 binary values (bee, hive, flower, wasp, trace, obstacle). This

approach was also used in MAgent2 [2].

3.4 Communication

We have designed our environment with communication in mind: the ideal strategies should result from agents sharing between themselves what they have learned and collaborating. Our approaches centre around passive communication. Each bee has its 8-value flag of floats, and can see the other bee's flags, where they encode information as part of their action, alongside their move direction. This also serves as a way of having some sort of memory, shared in this case.

3.4.1 Baseline: No Communication

To accurately measure the contribution of communication, we trained a baseline model with no communication at all. The bees behave as well as can be expected, considering the circumstances. They wander off, sometimes staying grouped together, and have to find the locations again every time, since they don't have memory either. During this exploration phase, they often move in a straight line, changing direction every time they encounter an obstacle.

3.4.2 Hardcoded Communication

Then we experimented with hardcoding all communication in a way that seems optimal. When a bee sees flowers, it records their relative location, updates it when moving, and shares it with all neighbouring bees. This is all done automatically, for all three potential targets: hives, flowers, and wasps. We see here that bees explore the map, share their findings, and go back-and-forth between the flowers and the hive, although not always very efficiently.

3.4.3 Naïve Communication

The simplest way of communicating is by encoding in the observation all the 8-value bee messages, for every location around it, thus creating a 37×8 matrix. We call this naïve communication.

3.4.4 Communication-based Architecture

We designed another architecture custom-made for trying to improve communication. We encode the bee's observation as a sequence of vectors, containing a one-hot encoding of the type of thing observed, its message if it's a bee, and the thing's position in cube hex coordinates. We encode each vector through the

same multi-layer perceptron (MLP) to create an embedding. Then, we sum up all of these embedding vectors, before passing the result through another MLP for the action, and another one for the value function (needed in PPO). The embeddings are hopefully learned to be summable in some way, to represent a synthesis of the broadcasted communications around the bee.

3.4.5 Simple Attention Variant

We also extend the previously explained architecture to use simple attention instead of a sum operation, using multiple heads. Each head has only one learnable query, which it uses on each of the input vector's keys to compute a score for each vector. The corresponding values of the vectors are combined into a score-weighted average, and the averages (one per head) are concatenated and linearly transformed into a smaller vector, before going through the same action and value MLPs as before.

3.4.6 Self-Attention Variant

We can further extend the attention variant by using self-attention instead of simple attention. Now each head has as many queries as input vectors, and the weighed averages (for each head, one per query) are also averaged again per head, before being linearly transformed into a smaller vector passed through the same MLPs as with the simple attention.

Chapter 4

Analysis of Results

4.1 Trace

After experimenting with adding the trace of previous locations to the bee's observation, we confirm that it does give a significant boost in the mean final score in the context of nectar gathering. This is the case in both extremes in communication: with no communication (a 40% higher honey score), and with the hardcoded ideal communication (a 62% improvement).

4.2 Reward Shaping

With a variety of different architecture, we performed an extensive comparison of reward shaping. We noticed an interesting phenomenon in numerous instances using reward shaping: after several rounds, the mean score started to decrease, whereas the mean reward kept increasing, as can be seen by comparing Figures 4.2 and 4.1. This suggests that bees were able to game the system and maximise their local rewards, even if it ended up harming the final score. In the vast majority of cases, the performance was better when using a sparser reward function, such as the subgoal rewards as described previously. This finding highlights the difficulty of selecting a good reward shaping function: even though ours was carefully chosen, it still subtly encouraged the wrong behaviour, and even now, we are still not sure what that behaviour was exactly.

There are a few exceptions, as can be observed in the Appendix, in tables A.3 and A.2, with the communication network in the honey-making scenario, and the self-attention network in the wasp situation. However, the variance is too high to be confident in these results, whose differences could be attributed to unlucky starting seeds.

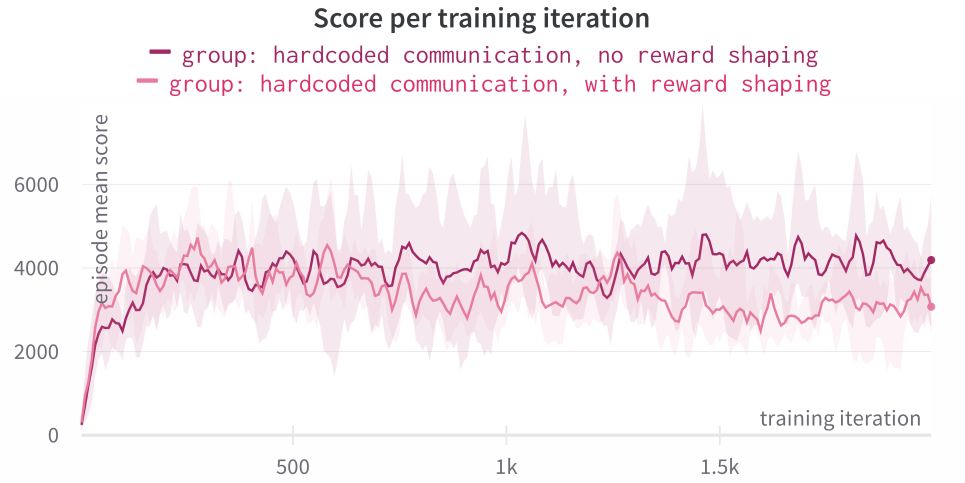

Figure 4.1: Mean episode score. Reward shaping slowly worsens the final score.

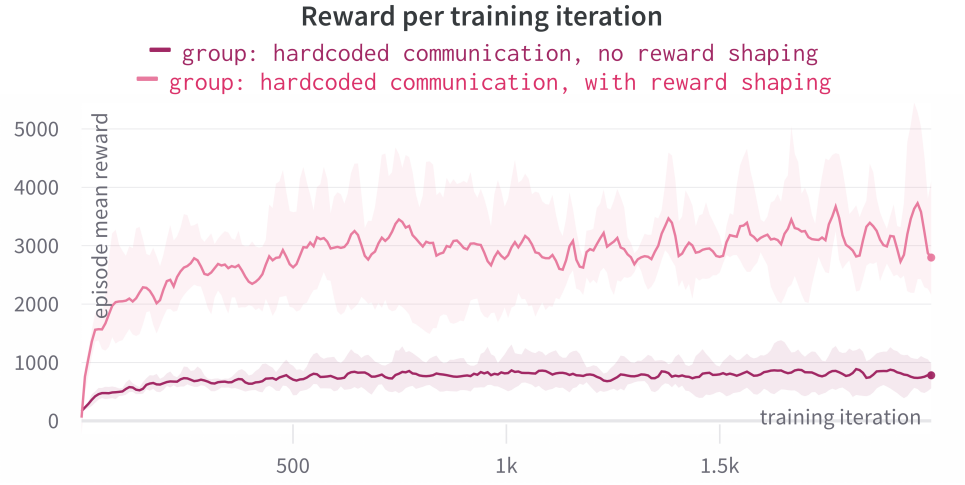

Figure 4.2: Mean episode reward. The agents keep increasing their local rewards.

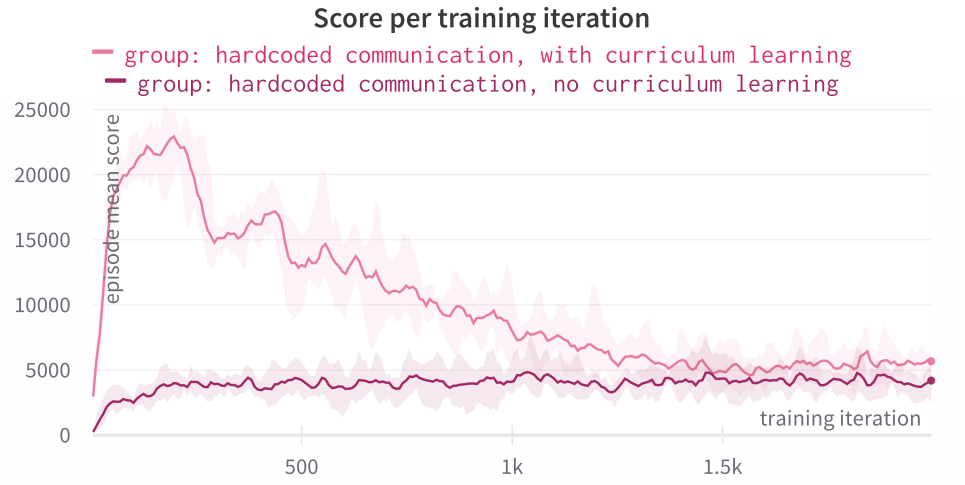

Figure 4.3: Mean episode score for curriculum learning, compared to without. Note that the lines plot the average of the 100 previous iterations, which makes the transitions when increasing the map size seem gradual. In reality, the actual score instantly drops.

4.3 Curriculum Learning

Curriculum learning leads to some improvements in the score, as can be seen in Figure 4.3, where with hardcoded communication the score with curriculum learning is 36% better. The score starts high since it is easier to make honey when the flowers and hive are closer. The map increases in size until around the iteration 1250, where it matches the size of the map in the experiment without curriculum learning. All the data is in the Appendix, in table A.4.

4.4 Communication Models

We have decided to avoid reward shaping when comparing model architectures after discovering its pitfalls. We also do not use curriculum learning, even though it gives better results, because we also want to better understand the convergence properties of the models. Models include architectures without communication in the observations, a naïve implementation of communication, communication through embedding aggregation, and variants with attention and self-attention, as well as models with hardcoded communication and completely manual behaviour, without reinforcement learning. The results are given in the Appendix, in tables A.5 and A.6. The best learned communication for nectar gathering was the communication-focused network with attention on top, scoring 2134. It is

still far from the performance of the hardcoded communication model at 4190, or even of the manually coded bee at almost 8000, but an encouraging fact is that the most encouraging model hasn't yet fully converged, as can be seen in Figure 4.4, and maybe only needs more training time to get closer to these hardcoded models.

The best performing model in the wasp-eliminating situation is a smaller naïve communication network, with only two hidden layers of 256 neurons, which is a bit surprising. In general, the bigger models always win, and we tried to adapt their sizes to reign in the training time. The wasp challenge is probably a little simpler: the only thing the bees need to do, in theory, is to go towards the wasp, and ask surrounding bees for help.

Between all three communication-centric architecture variants, the most promising is the simple attention one. Self-attention's poorer performance might be because its ability to find subtle interactions between vectors is overkill here, in this simple environment with straightforward objectives. The normal attention's multiple heads can track different features of the vectors, and the simpler architecture compared to self-attention most likely helps it converge quicker as well.

4.5 Analysing the learned behaviour

Beyond just looking at the numbers, the learned agents are somewhat disappointing when seen trying to solve the challenges. Even with the best models, bees don't consistently go back and forth between the hive and the flowers, although that would be a simple strategy to achieve a much greater score than what they are currently able of. For the wasps, even when it is almost surrounded, a bee often decides to move away from the wasp, thus letting it escape. This is also something that is obviously bad: the most basic thing that they should learn is to stay close to the wasp when they're next to it.

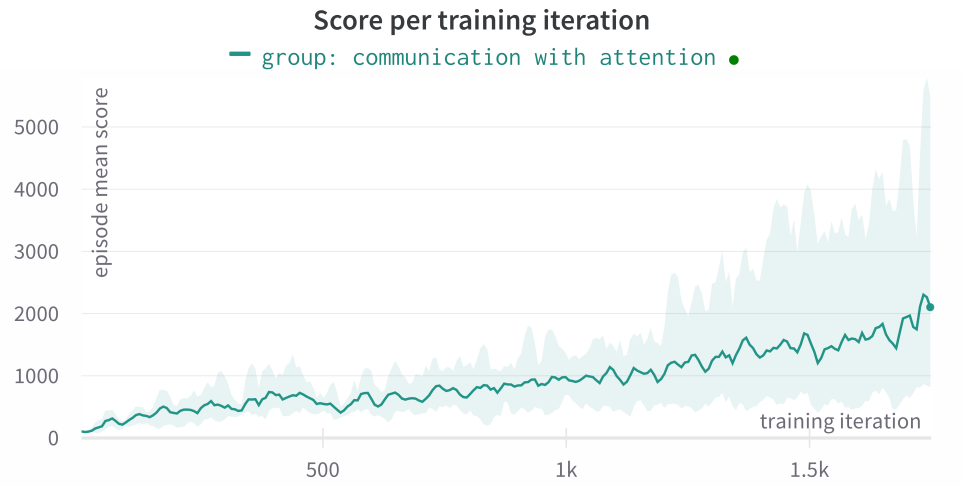

Figure 4.4: Attention Not Yet Converged

Chapter 5

Conclusion

All in all, this project allowed us to experiment with reinforcement learning, and test some commonly recommended techniques to get better results. The results were far from being competitive with a manually-coded agent, so we were probably too ambitious in wanting to design a competition where players compete from two camps, the ones that prefer to manually code all behaviour, and the ones that prefer training a RL model.

We have learned that reward shaping needs to be designed very cautiously, and so is best avoided. Furthermore, curriculum learning offers a boost in performance, jumpstarting the progress of the model. Moreover, keeping in the agent's observation its previous positions as a 'trace' is a way of compressing a memory of previous observations into its most essential components, which makes it considerably more efficient than quadrupling the size of the observation by keeping the last 4 around, for example.

Communication in general has proven to be tricky to learn. Since the agent cannot directly associate its learned communication action with the communication that it sees from other bees, it needs to learn two things simultaneously: both sending and receiving the information, which might not be trivial. Our new custom neural network architectures show how the communications around a bee might be aggregated, and the simple attention one in particular has been quite promising.

5.1 Further Work

Our custom models might benefit from more experimentation with hyperparameters, and there are many things that can still be tweaked, and that could unlock our bee's potential. Something that has been discussed, but not yet implemented, is using the coordinates in the input vectors of these architectures as positional encodings in the attention mechanism, to have it learn the positions of the agents in the same way in which Large Language Models know the position of words in a sentence.

There may be ways of separating the training of the communication, which frees up the agent to fully focus on communicating, instead of being interrupted by rewards when it gets close to its target, making it 'forget' any interesting communication breakthrough. This is an extension of curriculum learning: first learn to communicate, and maybe use judicious rewards in some way to encourage communication. Then the agent is dropped into an environment where it needs to accomplish things, and it has already bootstrapped some communication skills to help.

This project can be extended with a more diverse set of challenges, to benchmark RL models with different types of objectives. Our two current tasks have already shown us that some models might be better suited to some tasks than others. For example, the wasp-elimination didn't need an enormous neural network to get reasonable results. We have already implemented the foundations of obstacles in the map, and this can be easily extended to create maze-escaping challenges, adding an interesting pathfinding dimension.

We're hopeful that some challenge can be born out of this project, perhaps something strictly RL focused. Teams would have to compete in figuring out the best observation spaces, reward functions, and model architectures. Such a competitive environment would foster creativity and progress, and this would be very fun to both participate in, and analyse the winning solutions after the competition.

Bibliography

[1] "MIT Battlecode," https://battlecode.org, accessed: 2023-09-04.

[2] "MAgent2," https://magent2.farama.org, accessed: 2023-09-04.

[3] J. Kazil, D. Masad, and A. Crooks, "Utilizing python for agent-based modeling: The mesa framework," in *Social, Cultural, and Behavioral Modeling*, R. Thomson, H. Bisgin, C. Dancy, A. Hyder, and M. Hussain, Eds. Cham: Springer International Publishing, 2020, pp. 308–317.

[4] M. Towers, J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, T. Deleu, M. Goulão, A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, A. T. J. Shen, and O. G. Younis, "Gymnasium," Mar. 2023. [Online]. Available: https://zenodo.org/record/8127025

[5] J. Terry, B. Black, N. Grammel, M. Jayakumar, A. Hari, R. Sullivan, L. S. Santos, C. Dieffendahl, C. Horsch, R. Perez-Vicente *et al.*, "Pettingzoo: Gym for multi-agent reinforcement learning," *Advances in Neural Information Processing Systems*, vol. 34, pp. 15 032–15 043, 2021.

[6] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021. [Online]. Available: http://jmlr.org/papers/v22/20-1364.html

[7] E. Liang, R. Liaw, P. Moritz, R. Nishihara, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica, "Rllib: Abstractions for distributed reinforcement learning," 2018.

[8] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[9] A. Y. Ng, D. Harada, and S. J. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *Proceedings of the Sixteenth International Conference on Machine Learning*, ser. ICML '99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, p. 278–287.

Appendix A

Training Results Data

Model type	Mean score	Standard Deviation
Hardcoded communication, with trace	**4190**	1297
Hardcoded communication, no trace	2582	381
No communication, with trace	**1521**	344
No communication, no trace	1080	215

Table A.1: Comparison between trace and no trace, in a nectar scenario

Model type	Mean score	Standard deviation
Hardcoded communication, no reward shaping	**4190**	1297
Hardcoded communication, reward shaping	3358	532
Attention network, no reward shaping	**2134**	1566
Attention network, reward shaping	1480	360
Self-attention network, no reward shaping	**1186**	668
Self-attention network, reward shaping	1023	715
Communication network, no reward shaping	822	358
Communication network, reward shaping	**1302**	385

Table A.2: Comparison between reward shaping and no reward shaping, in a nectar scenario

Model type	Mean episode length	Standard deviation
Hardcoded communication, no reward shaping	**805**	45
Hardcoded communication, reward shaping	813	43
Attention network, no reward shaping	**926**	13
Attention network, reward shaping	998	21
Self-attention network, no reward shaping	962	42
Self-attention network, reward shaping	**943**	95
Communication network, no reward shaping	**955**	54
Communication network, reward shaping	968	12

Table A.3: Comparison between reward shaping and no reward shaping, in a wasp scenario

Model type	Mean score	Standard deviation
Hardcoded communication, with curriculum learning	**5678**	599
Hardcoded communication, without curriculum learning	4190	1297

Table A.4: Comparison between curriculum learning and no curriculum learning, in a nectar scenario

Model type	Mean	Standard deviation
No communication	1512	357
Naïve communication, small	1126	281
Naïve communication, large	1981	533
Communication network	822	358
Attention network	**2134**	1566
Self-attention network	1186	668
Hardcoded communication	4190	1297
Hardcoded behaviour	7801	13487

Table A.5: Communication architecture comparison, in a nectar scenario. Ordered from least to most communication

Model type	Mean	Standard deviation
No communication	947	22
Naïve communication, small	**822**	30
Naïve communication, large	890	30
Communication network	955	54
Attention network	926	13
Self-attention network	962	42
Hardcoded communication	805	45
Hardcoded behavior	784	25

Table A.6: Communication architecture comparison, in a wasp scenario. Ordered from least to most communication