ECE-GY 9123 / Spring 2021 Project Report
Eat, Move and Learn

Bruno Coelho Özlem Yıldız

Tuesday 18th May, 2021

1 Introduction

Reinforcement Learning (RL) is a hot topic in machine learning as it can enable a model to learn directly from the environment and past experience instead of relying on a curated dataset. In this paper, we analyze a multiplayer RL game in an 11×7 grid game board (see figure 1) where multiple "agents" control a goose whose objective is to eat various types of food while avoiding crashing into other agents. We compare a wide range of different approaches to the problem, analyze their results and discuss possible shortcomings and improvements. Our source code is available on GitHub[1].

We structure this paper as follows. Beginning with section 2, we cover relevant literature on RL and its application to games. In section 3, we describe the general behavior and rules of our game, such as the board setup and how our agents win or lose. Section 4 goes in-depth about our methodology and details the models used, how they were trained, and any other relevant experimental information. Section 5 discusses the results from our previous project updates and how these led directly to our final work and insights presented in Section 6. Finally, we discuss limitations and conclude the paper in section 7.

2 Literature Survey

RL applied to multiplayer games has been a popular research topic these past few years, with Atari games being a common task in RL. Prior work used Deep Q-Networks [8] to play most of the games available in Atari. In AlphaGo Zero [14], arguably the most famous experiment in RL, the authors are able to beat world-class Go players with an RL agent. In its extensions, Alpha Zero [13] and later MuZero [10], the algorithm is adapted to learn from random play and with no domain knowledge except the game rules, which is shown to be enough for it to be able to various Atari games.

For Snake-like games specifically, heuristics and Q-Learning has been tentatively analyzed in a non-multiplayer setting [2]. Many open-source implementations of these algorithms exist, with Stable-baselines3 [9] implementing both neural network models we intend to use.

3 Data

As a reinforcement learning problem, we have no fixed data-set and instead sample from the environment every time step. The information comes within the 'Observation' object [5]. The game board is represented by a 1D array of size 77, representing an 11x7 board by first counting from the top-left and continuing first horizontally then vertically.

We are able to get the index of our player with the 'index' variable. The variable 'geese' holds the list of geese in order. By using the 'geese' variable with 'index', we can calculate the squares that are occupied by our goose, with the head of the goose being the first element of the array. The food placement information can be obtained in the 'food' variable. By default, there will always be

[1]https://github.com/BrunoGomesCoelho/Eat-Move-Learn

1 food item on the board 'min_food' and we have 40 turns before losing one segment ('hunger_rate' variable).

There is also a maximum length that a goose can reach and this information is in 'max_length', which is 99 by default. The total time that game takes is in the 'episodeSteps', which is 200 steps by default, with each step taking the 'actTimeout' seconds, which is 1 second by default. The total time of one game is in the 'runTimeout', which is 1200 seconds by default. The current step is in the 'step' variable.

The agent returns a string from ['NORTH', 'EAST', 'SOUTH', 'WEST'] for the 'action' after the evaluation. At the end of the game or when there is only one goose left, the reward is calculated as:

$$Total\_reward = (max\_length + 1) * steps\_survived + goose\_length \tag{1}$$

4 Models

In this work, we compare 5 different types of models: Random, Heuristic, DQN, PPO, and Evolutionary. We describe each one in detail below.

4.1 Random model

A random model is a baseline that chooses a random action from the ones available. Since moving backward is not allowed and results in an immediate loss, we limit our agent to sampling from just the 3 directions that are valid at the current time step. This is the same as Kaggle's default random model implementation.

4.2 Heuristic Model

Heuristic, also interchangeably called greedy, is a hard-coded model intended to serve as a baseline. Its implementation is 30 lines of Python code and it tries to move to the closest food source available while making sure to avoid any other snake's body cell and any cell that is adjacent to the head of a snake (since there is a good chance these will be occupied in the next time step and could cause our agent problem). This is the same as Kaggle's default "greedy" model implementation.

4.3 DQN

DQN is a Q-Learning value-based method where we try to map a (state, action) tuple to a value; We then iterate over all possible actions we can take and pick the one with the highest value. The mapper is represented by a deep neural network, that takes in the observations represented as a feature vector and outputs our value. The stables-baselines3 library [9] has support for both dense MLP and CNN architectures; For the preliminary results, we represent our board as a vector of size 77 (7 rows by 11 columns) and use a fully connected MLP with 9 layers, of size [100, 100, 300, 100, 100, 100, 100, 100, 4] respectively. All layers except the last one use a RELU activation, with the last one using a softmax, all implemented using Pytorch and trained for 100 thousand games. The loss function is given by $\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\Big[\big(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\big)^2\Big]$, which can be seen as the expected value of a frozen target Q-network and our unfrozen Q-network. For more information, we recommend the tutorial by Lilian Weng [1].

4.4 PPO

PPO [11] is an Actor-Critic RL method that combines and extends both the A2C [7] and TRPO [12] algorithms. We use two neural networks, one named the Critic which is tasked with measuring how good a particular action is (analogous to our value-based approach), and an Actor-network, tasked with controlling how our agent behaves (analogous to a policy-based approach). Note that the Stable-Baselines3 [9] implementation contains several modifications from the original paper not documented by OpenAI such as normalized advantages and value function being clipped.

4.5 Evolutionary Algorithms

Evolutionary algorithms(EA's) are a derivative-free way of optimization by evolving candidate solutions. The candidates are generated randomly and evaluated on some task (e.g. a grid world game). The candidates who have the least fitness or gain the least reward from this task are eliminated, while the fittest is reproduced and mutated. Quality Diversity EA's, which is a subset of EA's such as MAP-Elites optimize a set of diverse candidate solutions (e.g. that play the game in different ways). Covariant Matrix adaptation is an EA for continuous optimization tasks (e.g., neural nets). Specifically, we use Covariance Matrix Adaptation MAP-Elites [3], which is the combination of the above Quality Diversity EA's and Covariant Matrix adaptation. For more information, we recommend the CMA-ME tutorial [15].

4.6 Training Environment

For all DL experiments, we use NYU's Greene HPC cluster. We use 12 CPUs per experiment, 30 GB of RAM per CPU and train our models for 24 hours. For the Evolutionary Algorithm, we use Google Colab's GPUs.

5 Earlier Results

Here we discuss some of our earlier results and how these directly led to the final project. We hope this sequential view can give the reader insight into the general methodology and try-test approach used.

First, we can directly use Kaggle's open-source environment code [6], to visualize the game board at the beginning of a random match. This can be seen in Figure 1. In it, we can see there are two food items (pizzas in this case, at [2, 2] and [7, 3], considering row and column index) and 4 geese (the remaining colored cells). The colors of the geese in order of their indices are White, Blue, Green, and Red respectively.

In figure 2 we can see the average reward for our DQN, Heuristic, and Random model. This initial result indicated that we were on the right track and that DL models could potentially be applied to our problem.

Next, we analyze the earlier results from both DQN and PPO, implemented as explained in the Model section. Together with Random and Heuristic, these 4 agents are played against each other in one game for 1000 episodes. Firstly, the mean and standard deviation of 1000 episodes are calculated and can be seen in Figure 2. Additionally, the moving average of window size 100 for 1000 games is demonstrated in Figure 3a.

In this comparison, the reward function as defined in 1 has been used. As can be seen from the above results, the DQN agent reaches a higher reward value when compared to the heuristic and random agents.

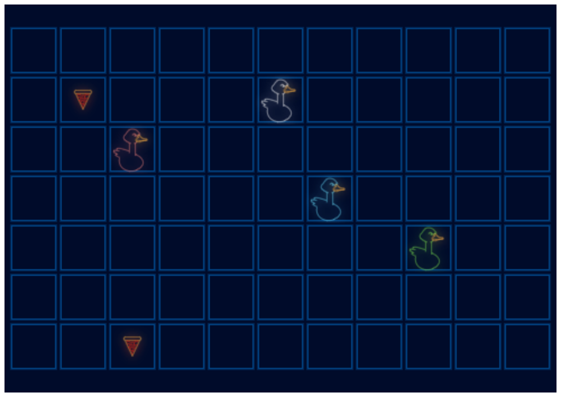


Figure 1: The game board

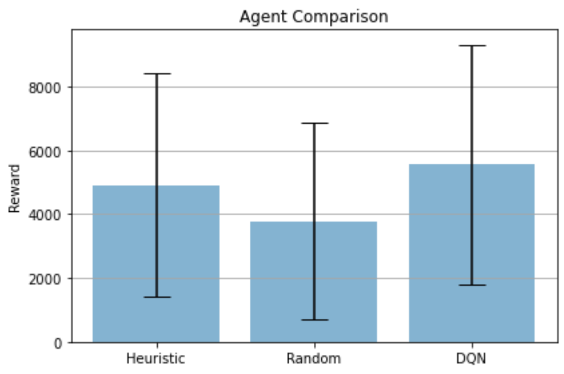
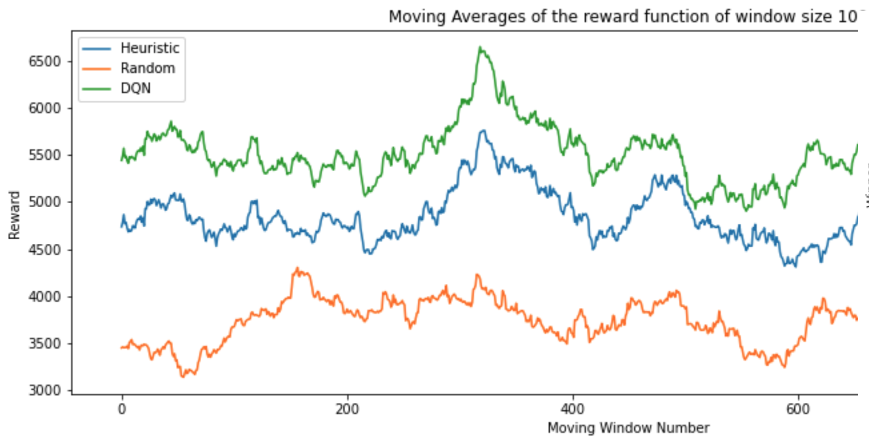
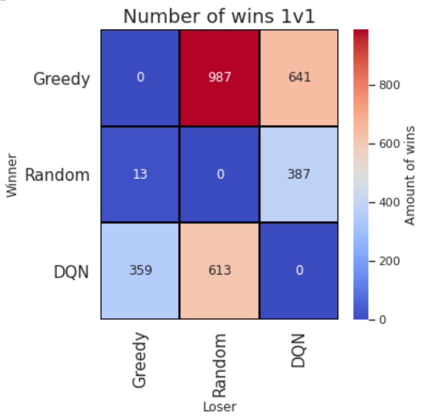


Figure 2: Mean and standard deviation (std.) graph



(a) Moving average of agent's reward



Winner \ Loser	Greedy	Random	DQN
Greedy	0	987	641
Random	13	0	387
DQN	359	613	0

(b) Amount of games won

Figure 3: Reward and amount of games won for different playing configurations

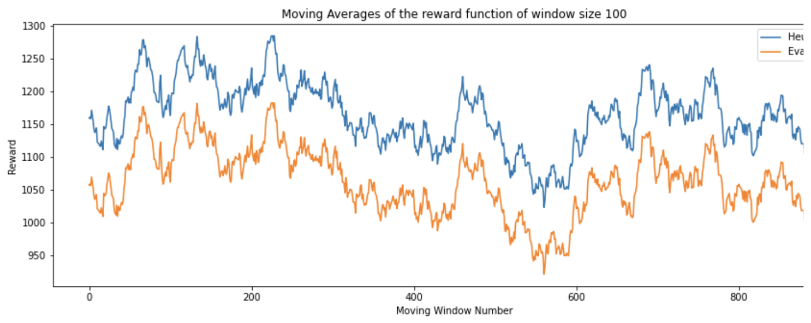
In Figure 3b we can see the number of wins for each 1 agent versus another agent out of a total of 1000 games per setup. The greedy model wins against the random model 987 times out of 1000, validating our intuition that the random baseline is not a particularly strong baseline.

Interestingly, the greedy agent tends to win against DQN more often (approx. 64%), even though in our 3 agents comparison, DQN had a higher reward. We believe this is due to the fact that DQN was originally trained to play against 4 agents and that the strategy when playing a 1v1 from the beginning is fundamentally different; With a fuller board, most of the strategy is making sure you don't move to a place that will be eventually cut off, while in a two-player game, due to the extra board space and fewer opponents, finding food is more important. We analyze this hypothesis in Section 6.3

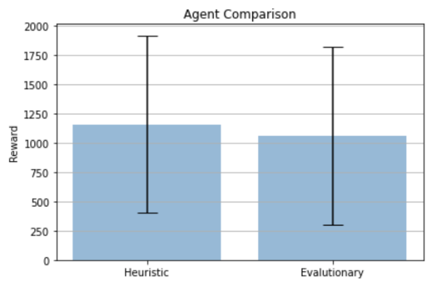
6 Final Result

6.1 Evolutionary Algorithms

As mentioned in the Project Update, besides trying traditional, gradient-based DL approaches, we also experimented with evolutionary algorithms as an agent. See the result obtained from a comparison with the Heuristic agent in Figure 4.



(a) Moving average



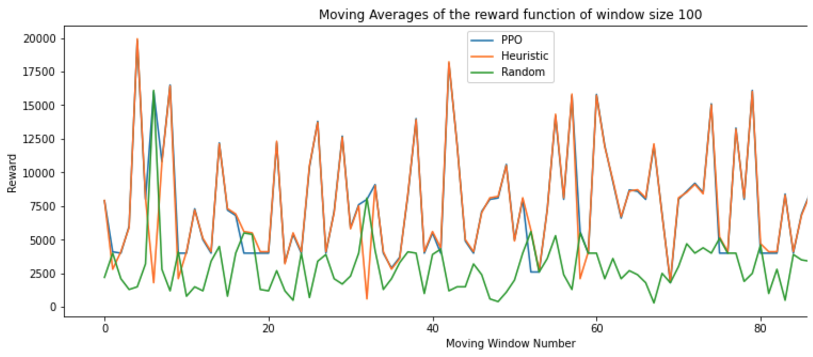
(b) Mean and std. graph

Figure 4: Evolutionary Algorithms and Heuristic Comparison

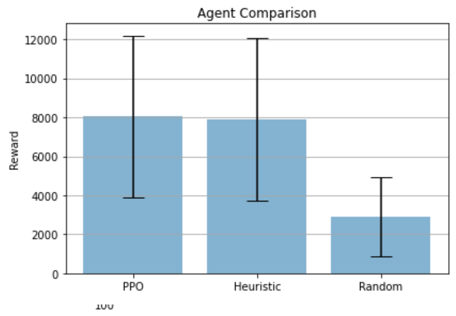
As can be seen from the results in Figure 4, the results for EA are constantly lower than the Heuristic and do not seem to be converging to better results. After this experiment, we did not continue using the evolutionary algorithm in future comparisons. We assume that this method may not be optimal for our purposes, might require more advanced techniques, or simply may require more training time.

6.2 PPO

In Figure 5 we can see the moving averages and mean/std. of our reward function between PPO, Heuristic, and the Random model. The PPO model seems to perform as well as the Heuristic, but not any better, differently than our previous results for DQN (Figure 3b). This trend was observed throughout all experiments, with DQN having an easier time training and converging and overall achieving better results.



(a) Moving average



(b) Mean

Figure 5: PPO and Heuristic model comparison

6.3 All models comparison

To validate our hypothesis that models trained with a different number of opponents end up learning different strategies, we create multiple experiments varying the number of opponents per model. More specifically, we train a model DQN-1 by playing it against a single Heuristic model. DQN-2 then represents a model trained with 2 Heuristic models, and likewise for DQN-3 and 3 opponents. DQN-Multi represents a model that was trained with all 3 amounts of opponents; all models were trained for the same amount of games and we perform the same experiments for the PPO model following the same naming conventions.

Figure 3b shows the results of all of our models evaluated on 1000 games on a 1v1. Notice that the number of wins does not necessarily add up to 1000 when comparing two pairs of models since we might have draws between our models; This happens more frequently for higher-performing models, e.g., DQN-1 versus heuristic.

As we can see, inside the DQN type of model, the DQN-1 constantly outperforms the other 3 DQN models, showing that its 1v1 strategy really was more effective and that the other models did not learn to play this specific scenario as well. The same happens with PPO-1 outperforming the other 3 PPO models, albeit the PPO-Multi is not so distant; we stipulate this is more due to the difficulty in convergence for the PPO algorithm than anything else since its results are constantly worse than the DQN approach.

Finally, we see that the Heuristic baseline is still a strong baseline to beat, with it performing better than our best DQN model, DQN-1.

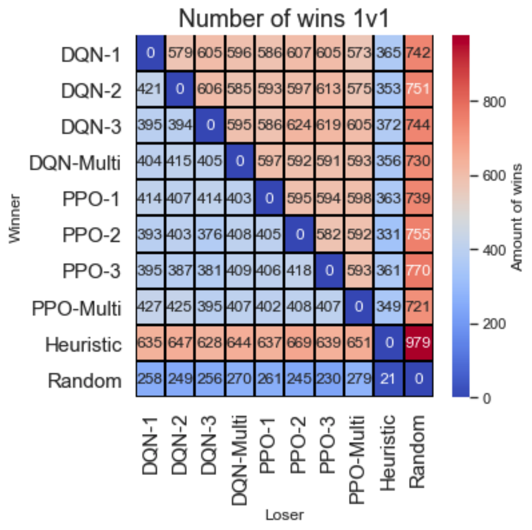
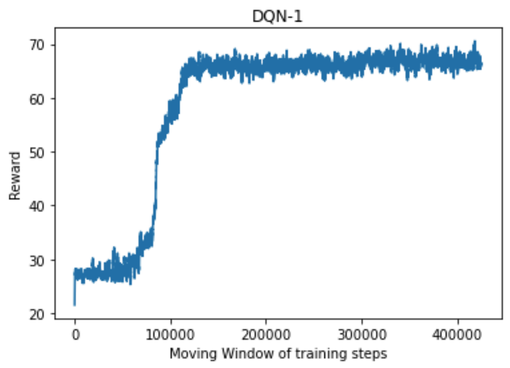


Figure 6: Results for all models

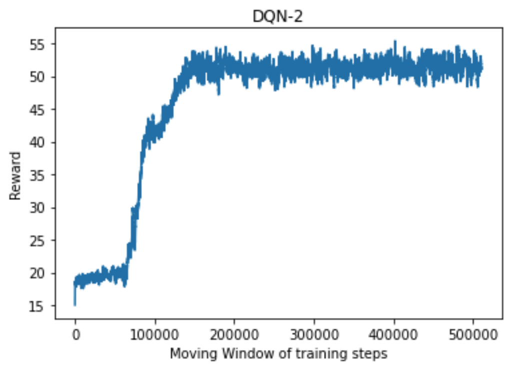
6.4 Training Reward Function

To analyze the convergence and general training properties of our models, we plot a moving window of the average reward for all of our DL models, as can be seen in Figure 7 and Figure 8. We notice a very similar trend in all of our models, where we see a big increase in the moving average reward at around the 50000 mark, which linearly increases very fast until a general convergence level. Our results of the average reward are consistent with our previous results, where DQN-1 ends up at the highest average reward and with less variance than its other versions.

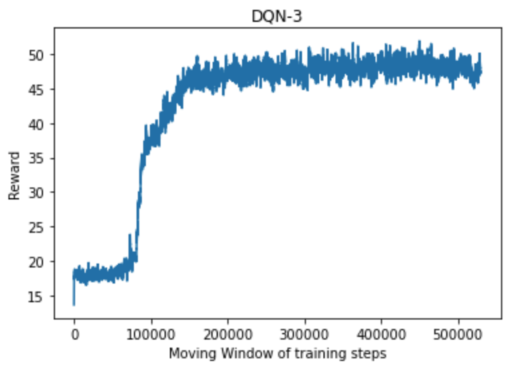
Interestingly for PPO, the results show an overall similar converge trend, but the individual model results tend to have a much higher variance. From these plots alone one might assume PPO-2 to be the best model, but as discussed previously it highly depends on the evaluating scenario, since it does not perform as well on a 1v1 scenario for example.



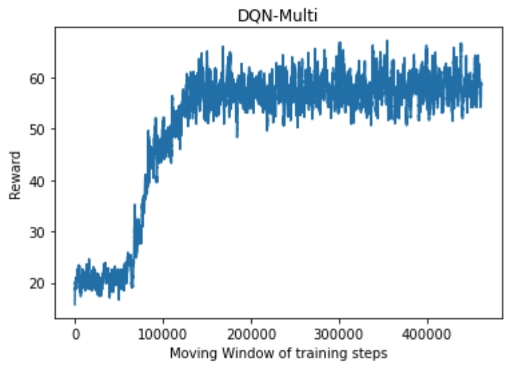
(a) Reward curve in the training for DQN-1



(b) Reward curve in the training for DQN-2

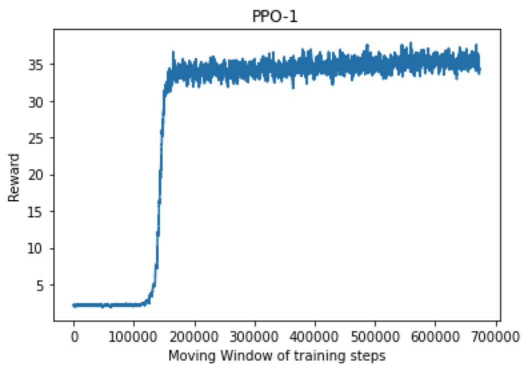


(c) Reward curve in the training for DQN-3

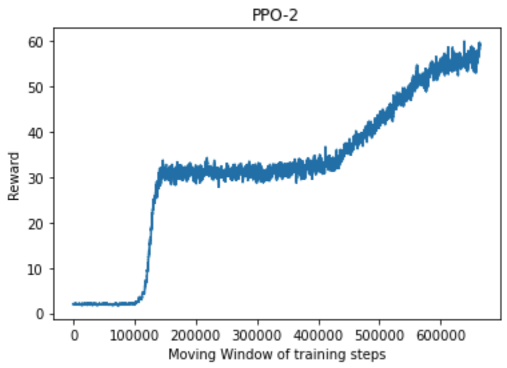


(d) Reward curve in the training for DQN-Multi

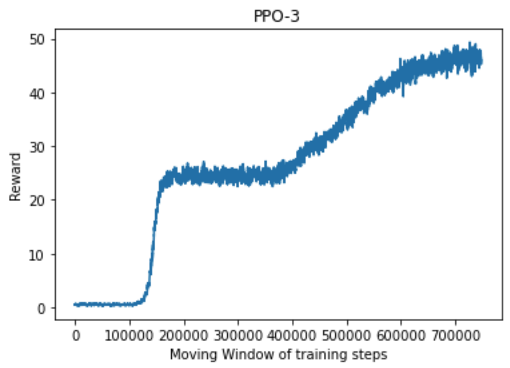
Figure 7: DQN reward curve comparison



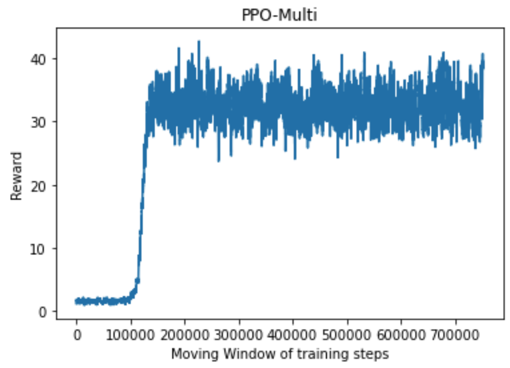
(a) Reward curve in the training for PPO-1



(b) Reward curve in the training for PPO-2



(c) Reward curve in the training for PPO-3



(d) Reward curve in the training for PPO-Multi

Figure 8: PPO reward curve comparison

6.5 Model Analysis

To analyze and better understand the behavior and trade-offs of our models, we perform multiple tests by directly manipulating the environment to create interesting examples. Below we discuss some of the more interesting findings, focused on comparing DQN-Multi with the Heuristic model.

In Figure 9 we can see two very similar scenarios but with different preferred actions, either just moving east on the first example (which we call "obvious" example) or moving west on the second one (which we call "wrapping") For the obvious example, both the Heuristic and DQN give the same results, to go east. For the wrapping example, only DQN correctly predicts west, while our Heuristic model goes to the east since it does not take wrapping of the board into consideration. Interestingly, the boundary for our DQN model seems to be 1 square after the middle. Further inspection of our models' weights showed us that our models tend to have a higher bias for going west versus east when all else is equal. This trait is most likely not desired since the board is vertically symmetric, showing us how these analyses are crucial towards finding faults in our models and correcting them.

7 Conclusion

Below we discuss some limitations of our work and an overall conclusion.

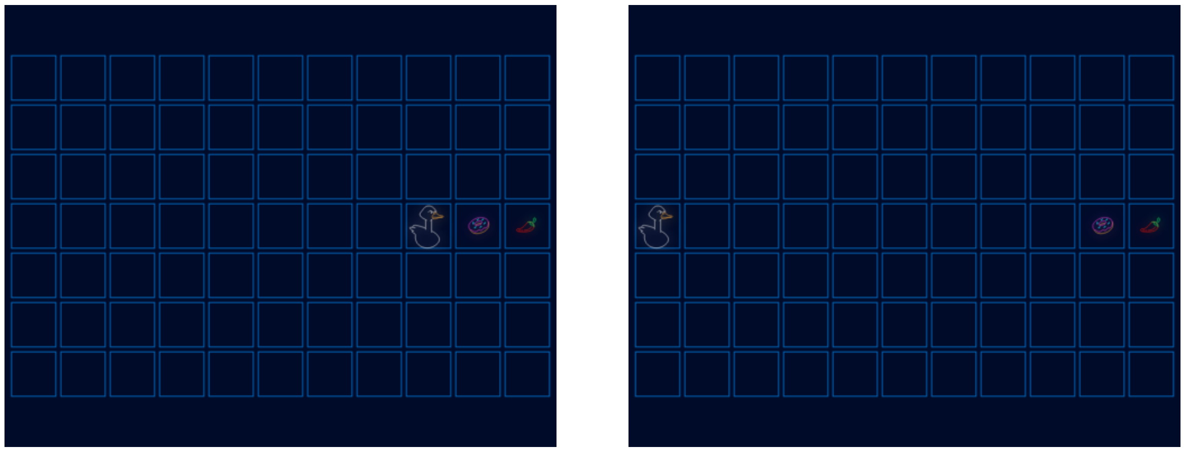


Figure 9: Obvious and non-obvious strategy decisions. Both require a single move to achieve a food item, the left on requiring the goose to move east, while the one in the right requires a model to understand "wrapping" and move west.

7.1 Limitations

RL as a whole is a very different area than traditional machine learning, due to the non i.i.d. data and the opportunity for the agent's interaction with the environment. Since this is the authors' first deep dive into RL, a significant part of the project involved learning about related topics.

The specific problem considered, Hungry geese, as both an RL environment and a Kaggle competition is very new, having only been around for the past 3 months. The authors find that this has a significant impact on the overall lack of helpful discussion and baseline RL implementations. While Kaggle does encourage both code sharing and discussion, we find that the competition aspect sometimes conflicts with our interest in applying RL models to the problem - many participants focus on 'fine tuning' hard-coded strategies to the problem or try to directly optimize against specific agents that take up a large proportion of the leader-board due to code sharing. [4]

Finally, we would have liked to see our EA work and converge, which unfortunately was not possible in this time frame due to the number of models and different experiments that were done for the project.

7.2 Takeaways

As mentioned in our original proposal, "Our goal is to dive deep into the RL literature and analyze different RL algorithms and their performance for this task." We feel we accomplished this goal and that this project has resulted in us understanding a lot better various traditional RL models and frameworks. We find training RL models to be a much trickier task than traditional models, which means proper training time inspecting, logging and analysis is extremely important and was the core of our time and attention.

References

[1] *A (Long) Peek into Reinforcement Learning.* en. Feb. 2018. URL: `https://lilianweng.github.io/2018/02/19/a-long-peek-into-reinforcement-learning.html` (visited on 04/23/2021).

[2] Ali Jaber Almalki and Pawel Wocjan. “Exploration of Reinforcement Learning to Play Snake Game”. In: IEEE. ISBN: 9781728155845.

[3] Matthew C Fontaine et al. “Covariance matrix adaptation for the rapid illumination of behavior space”. In: *Proceedings of the 2020 genetic and evolutionary computation conference.* 2020, pp. 94–102.

[4] *Hungry Geese - Public Code and the Evil Twin Effect.* en. URL: `https://www.kaggle.com/c/hungry-geese/discussion/230794` (visited on 04/23/2021).

[5] *Hungry Geese Go West!* URL: `https://www.kaggle.com/jamesmcguigan/hungry-geese-go-west`.

[6] Kaggle. *Kaggle/kaggle-environments.* Feb. 2021. URL: `https://github.com/Kaggle/kaggle-environments/blob/master/kaggle_environments/envs/hungry_geese/hungry_geese.py`.

[7] Volodymyr Mnih et al. *Asynchronous Methods for Deep Reinforcement Learning.* 2016. arXiv: `1602.01783 [cs.LG]`.

[8] Volodymyr Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: *arXiv:1312.5602 [cs]* (Dec. 2013). arXiv: 1312.5602. URL: `http://arxiv.org/abs/1312.5602` (visited on 02/27/2021).

[9] Antonin Raffin et al. *Stable Baselines3.* `https://github.com/DLR-RM/stable-baselines3`. 2019.

[10] Julian Schrittwieser et al. “Mastering Atari, Go, chess and shogi by planning with a learned model”. In: *Nature* 588.7839 (2020), pp. 604–609. ISSN: 0028-0836. DOI: `10.1038/s41586-020-03051-4`.

[11] John Schulman et al. “Proximal Policy Optimization Algorithms”. In: *CoRR* abs/1707.06347 (2017). arXiv: `1707.06347`. URL: `http://arxiv.org/abs/1707.06347`.

[12] John Schulman et al. *Trust Region Policy Optimization.* 2017. arXiv: `1502.05477 [cs.LG]`.

[13] David Silver et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* 362.6419 (2018), pp. 1140–1144. ISSN: 0036-8075. DOI: `10.1126/science.aar6404`. eprint: `https://science.sciencemag.org/content/362/6419/1140.full.pdf`. URL: `https://science.sciencemag.org/content/362/6419/1140`.

[14] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* (Jan. 2016), pp. 484–489.

[15] *Using CMA-ME to Land a Lunar Lander Like a Space Shuttle.* URL: `https://docs.pyribs.org/en/stable/tutorials/lunar_lander.html`.