Mutliplayer Snake AI

CS221 Project Final Report

Felix CREVIER, Sebastien DUBOIS, Sebastien LEVY

12/16/2016

Abstract

This project is focused on the implementation of AI strategies for a tailor-made game of multiplayer snake, inspired by the online hit *Slither.io*. We successfully implemented reinforcement learning (Q-learning) and adversarial-based (Minimax, Expectimax) strategies. For the former, we investigated methods to speed-up the learning process as well as the impact of the agent's opponents during learning trials. For the latter, we focused on increasing the run-time performances by designing threat-adaptive search depth functions and other pruning methods. Both approaches largely outperformed our hand-coded baselines and yielded comparable performances.
Keywords: *multiplayer snake, adversarial tree, reinforcement learning, adaptive search depth*

1 The Game

The proposed Multiplayer Snake game is an extension to the Snake game inspired by the popular online game *Slither.io*. In short, multiple snakes move on a 2D grid on which candies randomly appear, and grow by 1 cell every second candy eaten. Snakes die when their heads bump into borders or other snakes. This adds an interesting complexity to the classic, as snakes can try to make others collide with them. The game stops when a single snake remains or when the clock runs out, whichever comes first. The final score depends on the length at the time of death or at the end of the game and the number of snakes still alive at the time of death:

$$\textbf{SCORE} = \textbf{Length} \times \frac{1}{\textbf{\# Snakes Remaining}}$$

For full statistics, we also compute the percentage of wins and the average length at the end of the game (in general or only when the agent won). The other rules are:

- Candies appear according to a predefined appearance ratio;
- Snakes can cross their own tail, but not twice in two time steps. On the second successive crossing, the snake dies;
- A snake's head cannot move backwards;
- When a snake dies, every cell in its tail transforms into a special candy worth 3 regular candies.

A screen shot of the game is shown in 1. White tiles are heads, bold colored tiles are tails, bronze tiles are candies and golden tiles are special candies created from a dead snake's body.

2 Motivation

The main motivation behind this project is to assess the relative performance of adversarial versus reinforcement learning strategies and compare snake behaviors inherent to each. In addition, this game setting comprises a number of challenges, such as simultaneous player actions, multiple opponents and large state spaces. Finally, there is no single conspicuous objective, thus making it difficult to predict the opponents' best moves in search trees and makes RL policies very dependent on the strategies used for training.

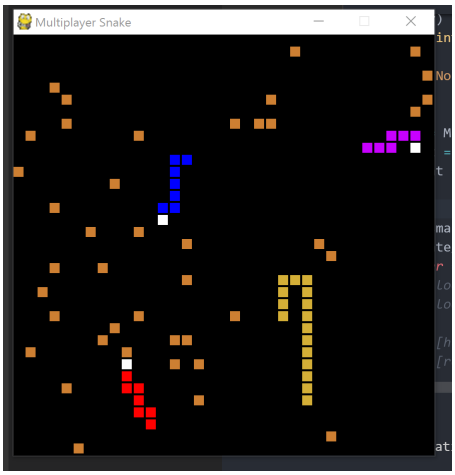

Figure 1: Screenshot of game interface

3 Related Work

There has been some work done on the traditional Snake game, mostly based on path finding. We found two projects which apply reinforcement learning techniques (Q-learning and SARSA) to implement an intelligent agent [1] [2]. We also found a project addressing the multiplayer setting [3], yet the *intelligence* in the agent's strategy consists only of a path-finding algorithm.

4 The Model

We represent a snake as a list of coordinate tuples of the cells making its head and tail, with the head cell at the head of the list. For computation time reduction on large grids, we also store an array of integers indicating for each cell if a snakes is present and if it has crossed its tail there. We define a state by a dictionary of all snakes alive, a list of all candy positions, and the current iteration number. The goal of our project is to learn optimal policies, therefore the inputs are the game states and the outputs are the agents' actions (*straight*, *turn left*, *turn right*). We implemented all our code in Python and it is available on Github [4].

5 Baselines: Static Strategies

We have implemented several basic strategies that will serve as baselines. It is possible for different snakes to follow different strategies.

- Smart Greedy. Snakes move towards the closest candy, but move randomly to unoccupied cells if an opponent is in the way;
- Opportunist. Improving again, snakes now move towards the candy closer to themselves than to all opponents;
- Random. Snakes move randomly, only avoiding grid boundaries.

We ran 1000 simulations of the 3 baselines together on a grid of size 20 and $max\_iteration = 1000$, and reported the results in table 1.

Our first oracle was a human player with moderate game experience. Over the course of 20 games against baseline strategies, the human player won 75% of the time, with a final score ranging from 50 to 100. Because of the high variance in the final human score, we set our score oracle to be the number of iterations, assuming it

[1] http://spranesh.github.io/rl-snake/
[2] http://cs229.stanford.edu/proj2016spr/report/060.pdf
[3] http://isnake.sourceforge.net/docs/
[4] `https://github.com/sds-dubois/snake.ai`

Strategy	Random	Smart Greedy	Opportunist
Wins (%)	21	25	38
Avg Points if Win	8	83	81
Avg Points @ End	11	63	66
Avg Score	5	41	47

Table 1: Baselines statistics with a 1000 simulations on a grid of size 20

eats a candy at every time step. This is slightly inferior to the maximum obtainable score since special candies come into play. Nonetheless, eating a candy at every time step is already extremely unlikely and would only result from shear luck. Hence, our oracle is 75% win and 318 points (the average number of iterations for baseline versus baseline games).

6 Adversarial Approaches

6.1 Settings

Our first approach to artificial intelligence consists of adversarial strategies. First of all, for adversarial methods like Minimax and Expectimax to function properly, we need to handle synchrony. In the case of Minimax, this is done by learning the *a priori* worst case scenario, i.e. the strategy assumes other snakes have already moved to the most menacing position. Thus, the snake will be more cautious than deemed necessary in a real synchronous setting. In the case of Expectimax, opponents are seen as random, and therefore the agent assumes it plays first.

In this game, it can be unclear what the opponents' agendas are. Are they attempting to trap other snakes or achieve good scores by eating candy, minding their own business? One thing is certain, dead snakes provide the highest reward, and the special candies created are beneficial for the remaining snakes. However, every snakes' primary objective is fundamentally to eat a maximum number of candies, which can easily be done without interfering too much with opponents. This ambiguity justifies both Minimax and Expectimax strategies: the former performs well when opponents are "offensive" and the latter may lead to adventurous exploration, which is better if opponents demonstrate a peaceful behavior.

Given the large state space, the number of moves (3) and adversaries (at least 3), it is critical to optimize computations. In this line of thought, Alpha-beta pruning was used for the Minimax agent, but was still slow. A simple Minimax agent with constant speed was also implemented to assess the reward/computation time trade-off of acceleration.

6.2 Evaluation Functions

Let's first define the maximum points a snake can achieve on a given grid. Because special candies are worth 3 points, we have:

$$MaxPoints = 3 \times \textbf{Grid Size}^2$$

We then define the naive evaluation function as:

$$NaiveEval(snake) = \begin{cases} MaxPoints & \text{if snake wins} \\ -MaxPoints & \text{if snake loses} \\ \textbf{Length}(snake) & \text{otherwise} \end{cases}$$

To account for the advantage of being close to candies, we use the greedy evaluation function that slightly penalizes a long distance to the closest candy:

$$GreedEval = NaiveEval - \frac{\min_{c \in Candies} d(\textbf{head}, c)}{2 \times \textbf{Grid Size}}$$

Minimax Depth Strategy	Depth 1	Depth 2	Smart Coward	Claustrophobic	Survivor
Wins (%)	44	59	61	59	63
Avg Computation Time	1.4	28.8	4.8	5.8	5.2
Avg Final Score	38	54	58	55	60

Table 2: Rate of victory, average final score and average computation time for different depth strategies when doing 1000 simulations of Minimax with radius 2 against Opportunist and Smart Greedy.

Expectimax Depth Strategy	Depth 1	Depth 2	Smart Coward	Claustrophobic	Survivor
Wins (%)	29	31	32	31	31
Avg Computation Time	1.3	31.8	1.4	1.5	1.4
Avg Final Score	20	22	21	21	21

Table 3: Rate of victory, average final score and average computation time for different depth strategies when doing 1000 simulations of Expectimax with radius 2 against Opportunist and Smart Greedy.

6.3 Adaptive Search

Due to a large state space, a large number of moves and adversaries, the search computations are very time-consuming and thus we cannot look deep into the Minimax/Expectimax trees. However, in most situations, most opponents are not a threat to the agent and can be considered immobile. This is equivalent to not considering them at all in the search tree. The best search depth can also depend on the state: when a snake is small, far from its opponents and far from the borders, just going to the closest candy is likely to be optimal. These two ideas can be implemented in an adaptive search function which returns the list of opponents to consider and the depth of the tree when given a state and an agent. We have considered 4 different strategies:

- **Coward**: If the head of the snake is too close to a snake, we increase the depth. We only consider opponents in the vicinity;
- **Smart Coward**: Improvement on Coward. We now consider an opponent only if its head is close to the agent's head;
- **Claustrophobic**: Improvement on Smart Coward. We now increase the depth if the agent's head is close to the border of the grid;
- **Survivor**: Improvement on Smart Coward. We now increase the depth when the agent's tail is curled up around its head. Formally, we define the compactness of a snake for given radius ρ to be:

$$\text{compactness } (snake) = \frac{\left|\left\{ \; c \in \text{tail} \;\middle|\; d(\mathbf{head}, c) \leq \rho \; \right\}\right|}{\rho^2 - 1}$$

  and we increase the depth when the compactness goes beyond a given threshold (0.5 or 0.6).

6.4 Results and Discussion

In total, we ran 1000 simulations of each strategy against Opportunist and Smart Greedy snakes on a grid of size 20, and reported the statistics in table 2 and 3. We report the final score as well as the rate of victory, the average computation time of game and the average length – when the snake wins and in general. We compare different depth strategies for Minimax and Expectimax with a radius of 2. We also report the full results for the best adversarial strategy in table 4. It is interesting to see that the best adversarial agent is longer on average at the end than when it is winning. This suggests that its opponents tend to die too early.

We observe that Greedy Minimax outperforms both Smart Greedy and Opportunist. The snakes following this strategy tend to stay a little shorter, which enables them to survive longer in a crowded grid. Because of its cautious approach, the strategy leads to few draws, with an estimated 10% of games ending with a head-to-head collision.

Strategy	Minimax	Smart Greedy	Opportunist
Wins (%)	**69**	9	18
Avg Points if Win	79	84	**91**
Avg Points @ End	**81**	59	71
Avg Final Score	**66**	28	40

Table 4: Full report for the best adversarial strategy: Minimax with Survivor depth function, radius 4 and compactness 0.5

Minimax Survivor Radius	1	2	3	4	5
Wins (%)	36	63	63	**69**	66
Avg Computation Time	**2.7**	3.9	4.3	7.6	14.8
Avg Final Score	27	60	60	**66**	62

Table 5: Influence of the radius with a compactness of 0.5 for Minimax Survivor

Minimax Survivor Compactness	0.4	0.5	0.6	0.7	0.8
Wins (%)	61	63	**67**	65	62
Avg Computation Time	4.5	4.3	4.3	**4.0**	4.3
Avg Final Score	61	60	**63**	57	59

Table 6: Influence of the compactness with a radius of 3 for Minimax Survivor

Strategy	Minimax rad 2	Expectimax rad 2	Minimax rad 1	Expecti. rad 2	Mini. rad 2
Wins (%)	**53**	18	24	**37**	36
Avg Points if Win	45	**90**	72	118	**129**
Avg Points @ End	55	56	**56**	105	**109**
Avg Final Score	**37**	32	32	74	**78**

Table 7: Minimax against Expectimax. On the right 1vs1 and on the left one Expectimax against two Minimax with different radius

Expectimax also outperforms both baselines. Compared to Minimax, it leads to more draws due to its adventurous approach. The average number of iterations is also lower confirming that Expectimax tends to die quicker.

The adaptive depth approach allows us to keep a minimal depth of 2 with a reasonable run-time and to explore the search tree deeper in more complicated situations. In a run-time similar to a depth 1 agent, we can slightly improve both the rate of victories and the average final score. Tuning the radius leads to a trade-off between computation time and final score (or rate of victory) (see table 5). Increasing the radius does not lead to better results beyond 4, meaning that it provides sufficient local information to choose optimal move. Finally, because the adaptive depth acts locally, its is relatively independent of the size of the grid (a longer grid still implies longer snakes which would make the process slower only when they are taken into account).

We can see on table 6, that changing the compactness does not change the computation time significantly. The optimal value seems to be around 0.6. We find interesting that for compactness thresholds inferior to 0.6, the performance does not increase.

On table 7, we observe that on 1v1, strategies tend to perform similarly with a lot of draws due to the adventurous behavior of Expectimax. With 3 agents (2 Minimax with different radii and 1 Expectimax), Minimax with radius 2 performs slightly better. This is principally because it wins much more often. However, Expectimax still performs well because it is generally much longer when it wins (more variance in its score). This can be due to its riskier but greedier approach, i.e. it tends to aim for clusters of candies and often kills and eats opponents.

6.5 What Could Be Improved?

Naturally, some improvements could still be made to the current strategies.

- Adaptive evaluation functions. They would be based on the current score to reflect the importance of winning and eating candies;
- Better evaluation functions. We could assign a bonus to various situations, such as proximity from other snakes tails, proximity of tail form other heads, special candies or clusters of candies. It could also penalize being in a corner, as we know corners are deadly;
- Improved evaluation functions with TD learning.
- Situation checks for Expectimax. We can fix the disadvantages of a "best case scenario" approach by adding a check for situations that could yield immediate death. For example, the AI could avoid head-to-head collisions, which are one of the main causes of draws;
- Layer adaptive depth. We could choose search depths while searching through the Minimax tree instead of at the head (trade-off between computation time and optimal strategy).

7 Reinforcement Learning

7.1 Settings

We have implemented AI agents by learning a Q-function with linear function approximation, i.e. $Q(s,a) = \theta \cdot \phi(s,a)$. We used the following indicator features:

- Agent's head x and y position;
- Indicator coding if the agent is trapped;
- Indicator coding if the agent is crossing its tail;
- Relative agent tail positions;
- Relative candy positions and their value;
- Opponents' head and tail positions relative to agent.

These last features are only considered if within Manhattan distance 11 or less of the agent's head. In addition, these are computed exclusively considering the agent's position after taking action a.
The Q-function is learned by stochastic gradient descent over a large number of trials, while the agent is using an ϵ-greedy exploration strategy. When not otherwise specified, we train the RL agent with the same opponents as the ones it is tested against.

A key item in our RL settings is the way rewards are attributed to the agent while learning the Q-function. We explored a few options and finally settled on attributing rewards following the game's rules (i.e. candy's value if eaten and a bonus/penalty of 10 points when winning/dying).

Another important modeling element was the discount factor. When using $\gamma = 1$, we found the learned strategy was to wrap around itself: the snake would not grow and thus be impossible to kill. In contrary, with a discount factor lower than 0.6, the learned strategy performed poorly, most likely because the snake would be indifferent to dying if it was to eat a candy. We obtained good results for $\gamma = 0.9$ and this is the discount factor used for the results presented below.

7.2 Eligibility Traces

Eligibility traces is an adaptation of the classic Q-learning update method. When observing (s,a,r,s'), we not only update the weights with respect to (s,a) but also for all previous (s_{-i},a_{-i}) as follows:

$$\begin{aligned}
\Delta &\leftarrow Q(s,a) - \Big(r + \gamma * \max_{a'} Q(s',a')\Big) \\
\theta &\leftarrow \theta - \eta\Delta\ \phi(s,a) \\
\theta &\leftarrow \theta - \eta(\eta\lambda)^i\Delta\ \phi(s_{-i},a_{-i})\ \ \forall i \geq 1
\end{aligned}$$

Strategy	Smart Greedy	Opportunist	RL
Wins (%)	12	22	**63**
Avg Points if Win	**97**	96	65
Avg Points @ End	61	70	**73**
Avg Final Score	31	42	**56**

Table 8: Detailed statistics for configuration 1

where s_{-i} denotes the i^{th} last state visited. In other words, the observed difference Δ is propagated back to previous state with an exponentially decreasing factor λ. Note than when using an ϵ-greedy exploration strategy, we perform such updates only for the history of states visited based on a *greedy* decision.

Eligibility traces are suppose to speed-up the learning phase since it will update Q for previous state and not only based on the generalization contained in the representation ϕ. This is especially suited to handle delayed rewards such as in games. However our game is special since it has short-term rewards (candies) and not only long-term ones (final score).

We experimented with different values of λ and found that this could yield better results for mid-range number of learning trials, when it was quite small ($\lambda \in [0.1, 0.2]$). When using a small number of learning trials, its influence was not clear, which could be explained by the noise in the updates and lack of time to average it. And with large number of learning trials, λ had to be smaller and smaller to be useful. Our intuitions is that again, eligibility traces can introduce some noise in the updates, and if the number of trials is large enough the classic update suffices to compute expected utilities. In the next section, we therefore present results for weights learned without eligibility traces since for equivalent performances, we preferred to increase the number of learning trials in favor of fine tuning λ.

7.3 Results and Discussion

In this section we simply refer to a "Minimax Survivor with radius 2 and compactness 0.5" as *Minimax*. We chose to train an RL agent against this Minimax agent because it seemed a good trade-off between performance and computation time. In section 8, however, we let trained RL agents play against better Minimax strategies.

We experimented with the following combinations of opponents to train the RL agent:

- Config 1: Smart Greedy, Opportunist;
- Config 2: Smart Greedy, Minimax;
- Config 3: Opportunist, Minimax;
- Config 4: Smart Greedy, Opportunist, Minimax;
- Config 5: Smart Greedy, two Minimax;
- Config 6: Opportunist, two Minimax.

Configurations 4a and 4b differ by the number of learning trials, 10,000 and 20,000 respectively. For all other configurations, we used 10,000 learning trials. All tests were made with 2,000 simulations.

Figure 2 presents the average final score obtained by each player in each configuration. We first notice that the RL agent has the highest final score except for configuration 3. Second, we observe that as soon as we introduce a Minimax player, the RL agent's final score increases considerably. This happens because the Minimax strategy outperforms both baselines and therefore the game can last longer, enabling the RL agents to grow more. Table 8 presents the detailed statistics of each player in configuration 1. We indeed observe that the RL agents wins most of the games (63 %) but does not have enough time to grow. Surprisingly, in this configuration, it is on average smaller when it wins that in general. This may be explained by the fact that it is better at avoiding the other snakes than its own tail.

Tables 9 and 10 present the detailed statistics for configurations 4a and 4b. Recall that both differ only by the number of learning trials (10,000 vs 20,000) and that the RL agent was trained against the same opponents (Smart Greedy, Opportunist, and Minimax). As expected, increasing the number of learning trials yielded better scores when playing against the same opponents. Between both sets of statistics, the main difference is

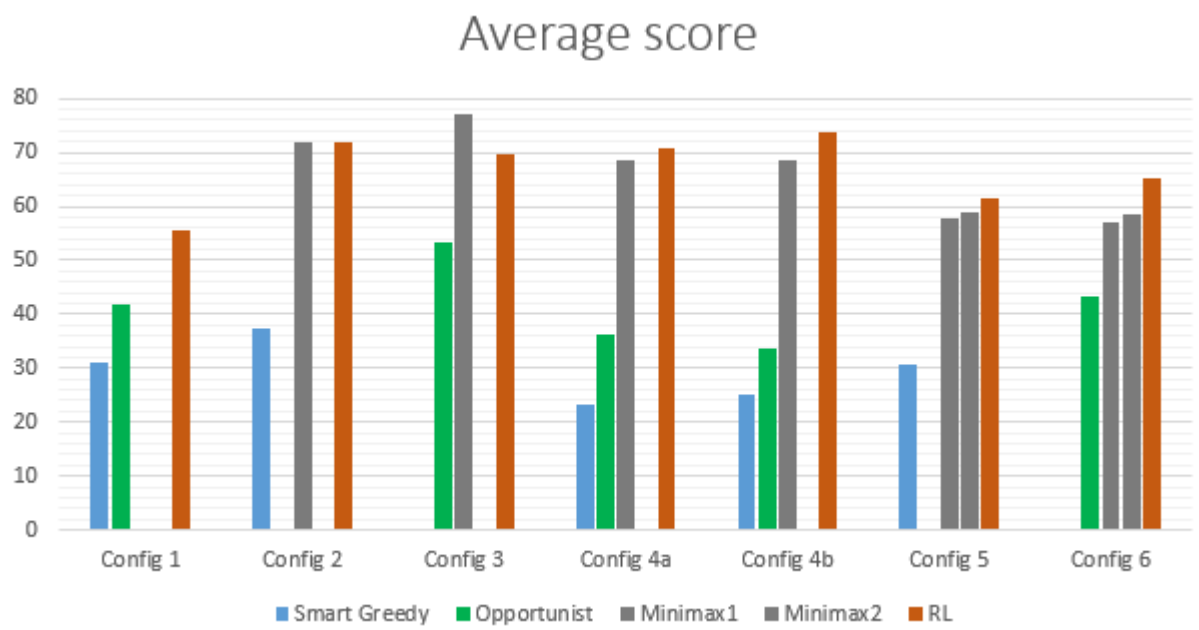

Figure 2: Average score over 2,000 simulations of each strategy for different game configurations

the average points the RL agent has when winning (which increases from 118 to 127). Our intuition is that the overall behavior of the RL agent does not change much, since average point at the end does not vary much, but that it gets better at playing when it has a long tail. Finally, we tested each learned strategy against Smart Greedy and Opportunist, and results are reported in Table 11. First, we notice it is difficult to correlate these results with those in Figure 2 (i.e. the performance of the RL agent when tested against its opponents used to train it). In particular configuration 3 did not seem promising at first but performs well against the two baselines. In addition, training against Minimax seem to benefit in general but does not yield a clear improvement (e.g. for configurations 2 and 5).

Hence, we can conclude that our RL algorithm enables us to learn good strategies that perform well in comparison to other baselines and Minimax, in a variety of configurations. However we also observe that the opponents used at training time can have a relatively important influence on the learned strategy's performances, depending on the opponents at testing time. This is logical since the best strategy should depend on the other players' strategy, as it is wise to be cautious if the opponent is aggressive and reversely.

Strategy	Smart Greedy	Opportunist	Minimax	RL
Wins (%)	7	14	**39**	36
Avg Points if Win	100	109	**125**	118
Avg Points @ End	53	70	96	**106**
Avg Final Score	23	36	69	**71**

Table 9: Detailed statistics for configuration 4a

Strategy	Smart Greedy	Opportunist	Minimax	RL
Wins (%)	8	12	**38**	37
Avg Points if Win	107	106	125	**127**
Avg Points @ End	55	66	97	**107**
Avg Final Score	25	34	69	**74**

Table 10: Detailed statistics for configuration 4b

Configuration	Config 1	Config 2	Config 3	Config 4a	Config 4b	Config 5	Config 6
Wins (%)	63	61	62	60	63	61	**64**
Avg Points if Win	65	68	**70**	67	**70**	67	69
Avg Points @ End	73	76	**78**	77	**78**	75	77
Avg Final Score	56	57	59	58	**60**	56	59

Table 11: Performances of an RL agent playing against baselines, when learned against different opponents

7.4 What Could Be Improved?

Below are a few of things we think would have been interesting to try.

- Rotational invariance. Since we play on square grids, the strategy should be invariant by any 90° rotation. So we could extract the features $\phi(s, a)$ by first rotating the board so that the snake is moving up. This would reduce the state space by a factor 4, thus increasing the ratio performance - number of learning trials.
- Non-linear Q-function. Although it makes sense to model Q has a linear function of our indicator features, we feel that some decisions should take into account more abstract elements. For examples, an agent should steer left if there are several obstacles to its right side but not on its left side, or move according to the shape of the opponents' tails. In this mindset, we could have learned Q using small neural networks, which would have allowed more complex functions. Moreover, SGD updates would have been equally simple, making them well suited for our Q-learning framework.
- Handcrafted short-term goals. We would have liked to implement these through the reward function used in the learning phase. This could have helped avoiding specific scenarios or forcing it to learn specific behavior. For example, with our current implementation it is difficult to learn to avoid getting into *tunnels* where it gets stuck. In addition, we did not observe any aggressive behavior, such as trying to surround an opponent to kill it, since such scenarios are highly unlikely to happen by chance in training. Therefore, we could give partial rewards when partially surrounding opponents to incite such tactics.
- Learning schema. We observe that the quality of learned strategy depends on the opponents trained against. Therefore, we would have liked to study this more in-depth as well as designing a learning schema. For example, we could learn the weights by training repeatedly against different opponents and different combinations of them. We could even design specific handcrafted strategies just for the RL agent to play against and learn, such as one that aims for head-to-head collisions to teach our RL agent how to avoid them.

8 Ultimate Match Up

In this section, we make our best Minimax strategy (Survivor with radius 4 and compactness 0.5) compete against the best learned RL one (configuration 4b).

Table 12 presents the statistics for duels between these two. Note that when using Config. 1 for the RL strategy, the average final score was only 79 (whereas Minimax's score was the same) - it thus appears crucial to train against the Minimax agent.

Table 13 shows the results when we add two baseline players: Smart Greedy and Opportunist. The Minimax agent obtained the highest final score once again, but recall that the RL agent was trained against a simpler Minimax version (Survivor with radius 2).

Strategy	Minimax	RL
Wins (%)	52	45
Avg Points if Win	133	125
Avg Points @ End	130	126
Avg Final Score	100	91

Table 12: Detailed statistics - best Minimax vs. best RL

Strategy	Smart Greedy	Opportunist	Minimax	RL
Wins (%)	7.2	11	42	35
Avg Points if Win	107	107	129	131
Avg Points @ End	54	67	106	109
Avg Final Score	24	33	77	74

Table 13: Detailed statistics - baselines vs. best Minimax vs. best RL

9 Conclusion

In the scope of this project, we developed a game of multiplayer snake inspired by the online sensation *Slither.io*, implemented reinforcement learning and adversarial-based AI strategies, and finally analyzed their relative performance. Computationally greedy by nature, our adversarial algorithms were sped up by the use of pruning and threat-adaptive search depth and locally trimmed search spaces. On the other hand, reinforcement learning (RL) parameters and features were tuned to obtain optimal policies. From extensive learning tests, we noticed that the RL policy depends greatly on the opponents against which it is trained, as their behaviors vary significantly. We attribute this to the absence of a clear objective, or in other words a fuzzy definition of victory, which is clearly one of the challenging aspects of the game. In the end, with our current effort and available computation power, we conclude that the best agent follows a Minimax strategy with a *Survivor* depth function of radius 4 and a compactness parameter of 0.5. It managed to slightly surpass our best RL agent in an ultimate four player match up. In the future, we wish to add snake acceleration to the game and implement non-linear function approximation for Q-learning and TD-learning to allow and incite aggressive encirclement tactics observed in human play.