Influence of Different Reward Functions on the Performance of an Agent's Reinforcement Learning Model in the Game Snake

GROUP 32:
ELIZABETH DWENGER - 2704183 - e.a.dwenger@student.vu.nl
AMBER HAWKINS - 2712318 - a.f.y.c.hawkins@student.vu.nl
VANSHITA KUMAR - 2687732 - v.s.kumar@student.vu.nl
KARIM NASR - 2507575 - k.nasr@student.vu.nl
CORINNA TRIEBOLD - 2703751 - c.e.h.triebold@student.vu.nl

ABSTRACT: Reward functions are an indispensable part of reinforcement learning. They represent, quantify and determine the reward which an agent tries to maximize and what it adjusts its policy for. Through this they play a major role in how well an agent performs at a given task. Through their simplicity and observability video games like Snake present a great opportunity for studying reinforcement learning agents under varying conditions like changes in the reward function. For this paper an agent which plays Snake was trained and tested with five different reward functions to observe and analyze their influence on the performance of that agent. The selected reward functions represent different strategies and heuristics.

1 INTRODUCTION

Games have long since been a staple of machine learning, from Deep Blue which beat world chess champion Garry Kasparov in 1996 to modern programs such as AlphaGo in 2016 [16]. This is true, in particular for reinforcement learning, in this field games provide critical features for training agents. They have easily observable environments, meaning that the agent can see the playing field, and games also follow predetermined rules. Additionally, they allow for an easy quantification of performance and rewards. Reward functions are an important aspect of reinforcement learning as they allow the agents to learn and develop strategies to maximize the expected reward [1]. This paper will focus on the influence of different reward functions on an agent's learning behavior. To this end the game, Snake, was chosen due to its relatively simplistic environment. A player has 3 goals: 1. Collide headfirst with the apple to 'eat' it. 2. Do not collide with its own body. 3. Do not collide with the border wall.

In this paper, the learning is done through reinforcement learning, more specifically, deep Q-learning (DQN). In the implementation, DQN determines what the best action is based on observations from the environment and previous knowledge acquired by the agent [11]. It takes the current state (position, direction, apple position) represented as cells as an input for the network, applies a ReLU activation function, and outputs the action that the agent should take. While training, the Bellman Equation is used to produce a Q-value based on the old state and a Q-value regarding the current state [12]. It takes the loss between the target and the predicted value and performs backpropagation to readjust its weights regarding the action and its reward.

This environment is partially deterministic, meaning it is dependent on the snake's actions; the only stochastic element is the apple [13]. Additionally, the environment is observable, which makes the training phase easy to manipulate. Due to the uncomplicated nature of the game, it is an ideal environment in which to investigate the influence of different reward functions. The influences of each reward function becomes evident through the learning performance of the snake during training.

2 BACKGROUND INFORMATION

2.1 Machine Learning in the Gaming Industry

Machine learning is utilized to create agents which are able to produce human-level or beyond performance in a respective game. Games have been used as a means to challenge artificial intelligence due to their complexity;

conversely, games have the benefit of being low-risk and rule-based. The variety of games available today also provide a variety of challenges for the available machine learning algorithms. [14]

2.2 Reinforcement Learning

Reinforcement learning, an area of machine learning which utilizes rewards to influence the actions of an agent, is used to train these agents to play games. One benefit of reinforcement learning is that the implementation of the rules of a game is not necessarily required. Rather, rules and strategies are learned through trial and error, with support from the rewards. [11] Moreover, unlike supervised learning, reinforcement learning does not need explicit corrections of poor actions. However, reinforcement learning may suffer from the exploration-exploitation trade-off, where a better action may remain hidden in favor of an action that has already proven to lead to some reward. [14] Additionally, another pitfall of reinforcement learning is the credit assignment problem. This is the issue of determining the magnitude of the reward that is given for any particular action [9].

2.3 Deep Q-Learning

The algorithm known as Q-learning uses a Markov decision process and decides on which action to take by updating a Q-function from the agent's experience [11]. The current state of the agent is the only thing that determines its succeeding action (as dictated by the Markov model). A reward function is implemented into the algorithm which rewards or punishes the agent based on the outcome of its action. The most advantageous Q-function is found by updating the Q-value using the Bellman Optimality Equation [12]. Measuring the quality of a Q-function is computationally burdensome; this aspect is typically considered the learning part of Q-learning. Q-learning has proven successful in multi-player games, such as StarCraft, where a Q-learning algorithm was trained to conquer opponents [11]. An extension to Q-learning is deep Q-learning, an algorithm that approximates the action-value function using deep neural networks [5]. It has the ability to learn policies from high-dimensional inputs and can approximate future rewards [14]. The approximation of these rewards is done by mini-batch sampling of states, actions, and rewards, which then are used to estimate the action-value function.

3 RESEARCH QUESTION

How and why do different reward functions influence the performance of an agent's reinforcement learning model during the training phase in the game Snake?

4 HYPOTHESIS

Based on the research question the following null and alternative hypothesis were created. The measure of performance was quantified through the average score. The hypothesis is purposefully undirected to account for both improvements or deterioration in performance.

- Null hypothesis: There is no difference between the mean scores whilst training with different reward functions; $\mu_1 = \mu_2 = \mu_3 = \mu_4 = \mu_5$.
- Alternative hypothesis: There is a difference between the mean scores whilst training with different reward functions; $\mu_1 \neq \mu_2 \neq \mu_3 \neq \mu_4 \neq \mu_5$.

5 METHODS

To implement the different reward functions a simple snake environment with a DQN network was selected as the basis. This environment was created by Patrick Loeber [10]. After understanding of the code, and determining if it would have the potential to help in answering the research question, the code was modified to suit the needs of this paper. Ensuring that the code worked and the methods were applicable required thorough understanding

Reward functions	Reward		
	Apple	Collision (Death)	Additional actions
Simple reward function	+ 10	- 10	0
Manhattan based reward function	+ 10	- 10	+ 0.25 (correct direction)
Inverse Manhattan based reward function	+ 10	- 10	+ ((furthest distance from apple - manhattan distance snake to apple) / 55.0)
Increased Score based reward function	+ (10 + ((score – (score % 5)) / 2))	- 10	0
Punishment based reward function	+ 1	- 10	0

Table 1. Rewards and penalties for all implement reward functions

of the underlying mechanisms implemented. Every reward function, other than the simple reward, was manually implemented. Table 1 includes a short overview of the functions.

All implementations can be found in the appendix. The simplest implementations were done for the simple reward function and the punishment based reward function. These required a change in the respective assigned values. The Manhattan distance reward function and the Inverse of the Manhattan distance both required the addition of the calculation of the Manhattan distance. This distance then is used to judge whether the snake makes a move that is in line with the Manhattan heuristic. The Manhattan distance is determined by calculating all possible values of the next state (snake turning up, down, left, or right). Then it calculates the distance between the food and the body depending on those possible values. From this it determines the direction that has the minimum distance and in which direction the snakehead should turn, to fulfill this shortest distance. In the implementation, the snake is rewarded if the direction it turns to is the same as the best direction calculated by the Manhattan distance. If it does not turn in the same direction no reward is given. The Manhattan reward is an additional reward to the ones already implemented, reward for the apple and punishment for death.

For the inverse of the Manhattan distance every cell on the map holds a certain reward. The closer this is to the apple the higher this reward is, with the apple representing the maximum reward. A good visualization of this would be a heat map where a single source of heat is coloured red with heat bleeding out into the surrounding area. An alternative perspective is that the inverse of the Manhattan distance turns the map into the inverse of a loss surface where the lightest point represents a global maximum. This was implemented by taking the remaining distance, calculated through the Manhattan heuristic, to the apple. The remaining distance of where the head is located was deducted from the absolute maximum number of cells away from the apple and a reward is calculated from it. Thus, a snakehead that is farther away will have a smaller difference between the maximum steps away which results in a smaller reward, compared to a snake closer to the apple. The difference between the maximum block steps away and the current block steps away would be larger, thus resulting in a larger reward. This reward, like the Manhattan reward, is additional to the existing reward.

The increased reward function required the addition of a dynamic calculation of the reward that increases and updates it for every five points the snake obtains. Thus, the reward the snake can obtain continuously grows with the length of its body. Long-term survival thus maximizes the reward an apple can bring. The reward ensures that for every 5 points it gains the reward regarding the apple increases by 2.5 points to the base reward of 10. To implement this, the module of the score is taken and then the result is subtracted from the current score. This ensures that the score is always rounded to the floor of the numbers divisible by 5. In addition to the baseline

code and the extra reward functions, implementation regarding automatically transferring all the results from the experiments into a CSV files was added.

6 EXPERIMENTAL SETUP

To test the hypothesis about the differences in performance between the reward functions during the training phase, the following experimental setup was created and followed. In the snake reinforcement environment only the reward functions were changed, thus the reward functions are the independent variable of this test. The hyperparameters and other factors were kept the same to ensure that the observed effect was caused only by the reward functions. The simple reward function was chosen as the control variable against which the other reward functions could be compared. The simple reward function was tested prior to the other reward functions and achieved what is considered a reasonable performance given the limitations of the agent, thus it represents a good baseline for the models based on other reward functions.

For each reward function, 5 trials were run with 5000 games each. More than 1 trial was chosen to ensure that the results of the trials were not caused by random fluctuations [2]. The number of trials and games was limited by training time and available computational power. For each episode of each trial, the score was recorded in a CSV file and for each trial, the average score was calculated; the scores were then averaged for each reward function. The overall average of a reward function is the independent variable of this experiment.

Since the research question also aims to understand how potential differences in performance occur, the number of episodes the agent needed to start performing well was also observed. The threshold for this was set at the number of games at which the agent reaches an average score of ten. This number was then also averaged over across the trials. It represents how quickly the different reward functions can lead the agent to start playing better games. Since the agent's trained for this experiment will not be deployed no considerations were made towards a real-world use case or production.

7 RESULTS

During the training phases the agent continuously learned from previous experiences. All agents began with a cold-start, meaning that they had no prior knowledge about the environment or the rules of the game. The initial model that was run was the simple reward function, of which the results can be seen in Figure 1. According to the results shown in the graph, we can see that the learning process starts slowly but rises steeply between around 100-300 games and slowly plateaus at between 30 to 35 points on average after about 1000 games in all of the trials.

The Manhattan distance was the next reward function which was trained. The results can be seen in Figure 2. The graph shows a very similar behavior compared to the simple reward function in Figure 1 with two minor differences. The results do not plateau as completely as the simple reward function does; there is a noticeable slight continuous upward trend in all the trials with the average scores between 30 and 35. Furthermore, the performance improves slightly slower than the simple reward function.

The increased score based reward function was the next model for which results were collected. As can be seen in the graph in Figure 3, the agent starts off relatively similar to the previous two, but eventually the average score plateaus around 25 and 30 points as opposed to the two initial graphs which plateaued between 30 and 35 points. The performance improvement does seem to occur at a similar rate as the Manhattan based reward function.

The punishment reward function was the next one for which results were collected for. As can be seen in the graph plotted in Figure 4, the graph shows very varied results. Two of the trials never managed to collect any rewards, and stayed at a 0 point average. The other three trials each took longer to improve their performance than the previously tested functions and their results plateaued between 25 to 27 points towards the end. This

average is close to the average of the increased reward based function but smaller than the averages of the simple reward and the Manhattan distance reward function.

The final reward function which was used for training is the Inverse of the Manhattan distance reward function. As can be seen in Figure 5, the results show that this reward function generally did not perform well compared to the others; the average score never went over 1 for any of the trials. They all performed quite equally for approximately the first 500 games, after that only two of the trials, Trial 5 and Trial 3, manage to show any improvement in the average score.

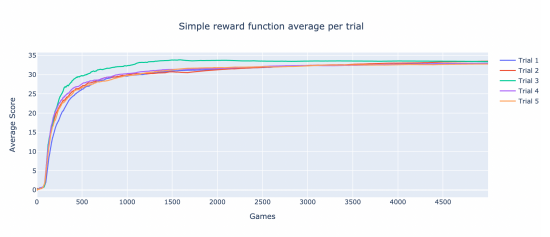

Fig. 1. Simple reward function - score of 5000 games throughout training phase

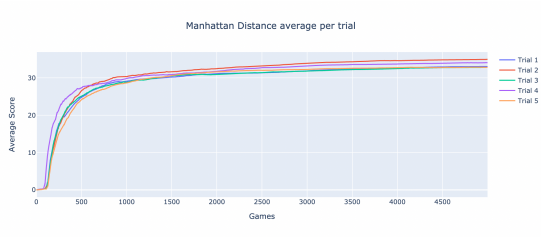

Fig. 2. Manhattan based reward function - score of 5000 games throughout training phase

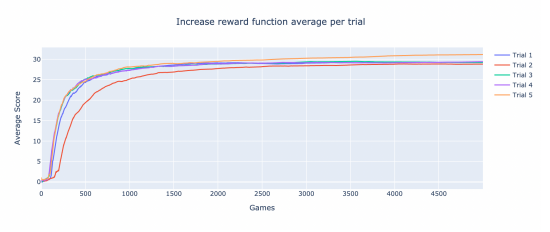

Fig. 3. Increased score based reward function - score of 5000 games throughout training phase

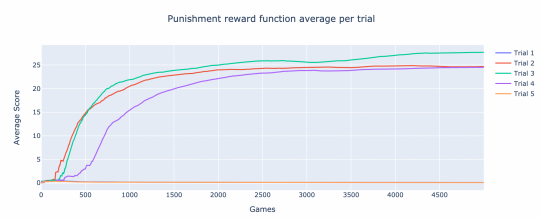

Fig. 4. Punishment based reward function - score of 5000 games throughout training phase

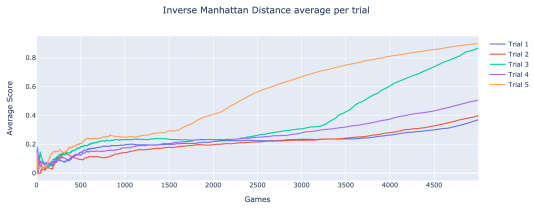

Fig. 5. Inverse Manhattan based reward function - score of 5000 games throughout training phase

Figure 6 displays the average performance across all trials for each reward function. As can be seen, the simple reward function and the Manhattan distance were the most similar, with the noticeable differences being that simple reward picks up faster early on, but plateaus slightly sooner and is eventually out performed by the Manhattan distance. The increased reward function plateaus sooner than these two functions, while the punishment reward function plateaus later, at around 2000 games, and at a much lower average score. Finally, the inverse Manhattan distance performed poorly compared to the rest of the reward function models.

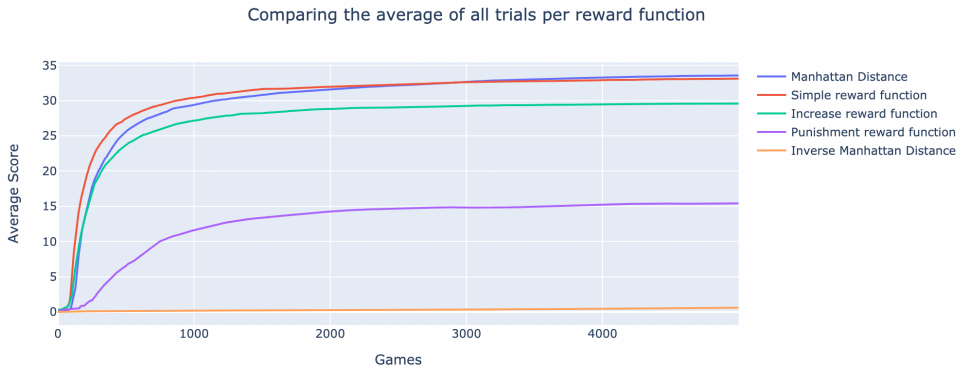

Fig. 6. Average score of every reward functions during the training phase

In addition to the average score the number of games that it takes an agent to cross the threshold of a 10 point average was also recorded. This represents the number of games it takes an agent to start performing better. According to the results the simple reward function improved the fastest, followed by the Manhattan distance and the Increased reward function which have the same average number of games to cross the threshold. The punishment function is the slowest with two trials that never reach an average score of 10 and above. The Inverse function does not reach a 10 point average in any trial. The table with this data can be found in the appendix.

8 STATISTICAL ANALYSIS

Following the qualitative analysis of the results, two significance tests were performed to check the hypothesis quantitatively. The first test that was performed was the ANOVA one-way test [8]. This test is used to test for significant differences in means between three or more independent groups. This test was chosen to check if further testing of the hypothesis is appropriate. If this test had returned no significant differences between the means this would already indicate that the reward functions do not have a significant influence on the performance of the agent. To conduct this test the scores for each trial of a reward function were averaged over; this was done for each reward function leading to five groups that were tested using the ANOVA test. The significance level was set to 5%; $\alpha = 0.05$.

Anova: Single Factor

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Manhattan	5000	167807,4	33,56148	61,94976216
Simple	5000	165599	33,1198	52,1977115
Increased Reward	5000	147893	29,5786	49,78491502
Punishment Reward	5000	77065	15,413	29,53155331
Inverse of the Manhattan	5000	3046,8	0,60936	0,224629316

ANOVA

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	4073296,373	4	1018324,09	26287,66597	0	2,372287568
Within Groups	968249,168	24995	38,7377143			
Total	5041545,541	24999				

Fig. 7. ANOVA one-way test

As figure 7 shows, the test results of the ANOVA test indicate a significant difference between the means of all tested reward functions; the calculated p-value is so small that the software used to conduct the test rounded it to 0. Therefore, the p-value is smaller than the significance level ; $0 < 0.05$. The null-hypothesis is rejected as there is significant difference between the performances of the reward functions. However, the ANOVA test gives no indication as to which of the tested groups are significantly different. Therefore, further testing was conducted. The simple reward function was chosen as the baseline against which to test the other models; for this reason, a t-test for independent means was performed for each model against the simple model [6]. The significance level was set at 5%, $\alpha = 0.05$. A two-tailed test was selected since the hypothesis is undirected. This test returned the following results:

Reward functions compared to the simple reward	P-value	Conclusion
Manhattan reward function	P-value = 0,003472284	P-value $<\alpha$
Inverse reward function	P-value = 9.13e-130	P-value $<\alpha$
Increasing reward function	P-value = 0	P-value $<\alpha$
Punishment function	P-value = 0	P-value $<\alpha$

Table 2. Results of t-test for independent means for the reward functions

For every reward function tested against the simple reward function the p-value is smaller than the significance level α, this means that the mean of every reward function is significantly different compared to the mean of the simple reward function, this gives no indication as to the direction of the significance. In the case of the increasing reward and the punishment function the p-value was very small and thus rounded to 0 by the software used to conduct the test. Both significance tests lead to a rejection of the null hypothesis, there is evidence for a difference in performance between the reward functions. The full results of the conducted tests can be found in appendix B.

9 DISCUSSION

The results allow for an opportunity to discuss the reward functions and their importance for an agent's performance. The simple reward function and the Manhattan distance performed best. With the Manhattan distance, the agent gains 0.25 points for each step it takes in the direction of the Manhattan distance heuristic. This approach proved to be the most effective in maximizing the agent's performance, as it averaged the highest, illustrated in Figure 6. Relative to the Manhattan distance, the implementation of the simple reward function performances nearly as proficient. We can see both Manhattan and the simple reward function to be close in average points for each of the 5 trials. For the simple reward function, the agent reaches its peak performance faster than in the Manhattan reward function, yet it also plateaus. Meanwhile, the Manhattan reward function continues to improve its performance. In the Manhattan reward function, the agent will optimally take a maximum of 2 turns to reach the apple from its current position. The agent receives a reward when it turns in the correct direction that correlates to the Manhattan distance heuristic. Thus, it appears as if it developed a strategy that causes fewer self-collisions by reducing the number of overall turns it takes; this decreases the likelihood of the snake colliding with itself. This reward function was created to lessen the influence of the delayed reward problem that is present in reinforcement learning. By adding intermediate rewards, it minimizes the influence of not receiving immediate feedback for each action it takes. The snake's ability to create good strategies can also be based on intermediate actions within the Manhattan heuristic instead of just knowing that getting the apple is advantageous and death is not.

The increased reward function does not perform as well relative to the simple reward function or the Manhattan reward function. However, it does outperform the Inverse and the Punishment reward function. The increased

reward steadily improves the agents performance. The increased reward function performed less well than expectations. It was predicted that the increased reward of the apple for every 5 points obtained would incentivize the snake to grow. As the snake length increases, the chance of death increases, thus at a certain length, it would start accumulating negative rewards causing the snake's performance to plateau. The increase in the apple reward was expected to help stimulate it to develop a better long-term strategy. This, however, was not what was observed. This could be due to sparse reward: while the reward does increase every 5 points the snake obtains, the intermediate states has no meaningful reward. This illustrates the problem of credit assignment rewards, as the expectation of the reward cannot be maximized to develop a good strategy. Even if the apple reward does increase, the moment the snake collides and dies, it considers the game a bad game. It could have completed a series of perfect moves, but if the last action resulted in death, the previous actions are considered bad.

Lastly, the reward functions, inverse Manhattan distance and punishment reward function, have the worst overall performance relative to the other reward functions. For the inverse Manhattan reward function, with every ring of cells closer to the apple, the cell increases in the reward. This was done with the expectation that the snake would learn to move to the apple to maximize its reward. After a few games, it seemed to immediately go straight and hit the border. It would maintain this strategy for the remainder of the games, thus not accumulating any score points and thereby performing extremely poor. It is suspected that this strategy is caused by the exploration vs. exploitation trade-off. While the agent does not receive the large reward of the apple, it still gains a reward by moving across the cells. While it is not a lot of points, it is still regarded as doing well as it is accumulating points. This results in exploitation of the minimal points, and not exploring to find rewards that would be worth a lot, such as the apple. The punishment reward function performs better than the inverse Manhattan reward function, presumably due to its clear goal and avoidable actions. Even if the reward regarding the apple is smaller than in the Simple reward function, it still forms a workable strategy to maximize the apple points. Yet it is clear that the snake is less incentivised to accumulate a higher score in comparison to the higher apple rewards.

Overall, the obtained results depict that the reward function significantly influences the performance of a reinforcement learning agent. The results allow an overview and understanding of how various implementations of reward functions can improve or impair an agent's performance. This conclusion is supported by the statistical analysis that concluded that the average performance for each reward function was significantly different and thus allowed for the null hypothesis to be rejected.

Throughout the experimentation of various reward functions, the agent was confronted with a technical limitation; it developed the tendency to collide with itself. The snake's collision with its own body is due to it's inability to be aware of its own body. As the snake's length grows it was not aware of how long it is, it would then take turns that would trap itself between its own body. To solve this, a convolutional layer needs to be implemented into the DQN. Convolution layers are a type of deep neural network layers which are often utilized to classify or generate pictures. The shape of this layer is determined by some knowledge of its purpose, for example, acquiring the apple or keeping track of its body. Neural networks are often used in image classification or generation, however, they have further been applied to games in order to gain an understanding of how algorithms influence agents and the way they interact with their respective environments in order to improve and maximize their performance. [15] The first layer in a convolution neural network applies a convolution operation to the input, passing the result to the next layer. With the application of the convolution layer in the respective algorithm, this would significantly improve the performance of snake [15]. As the agent continues learning from each output layer, it is able to record the improvement and further create long-term strategies which would maximize its performance. In the respective technical issues which occurred during the testing of the agent, it sometimes fails to efficiently learn from its previous outcomes, resulting in the snake devising inefficient long-term solutions that failed to maximize its performance. For example, the agent managed to achieve high scores of around 70-80 but failed to do so consistently. For further research, implementing a convolution layer would be an interesting approach to improve the agent.

Although it has been discussed that convolution layers have a significant influence on the performance of a reinforcement learning agent, overfitting the data can also affect the snake's performance [17]. While environment overfitting can occur in any learning algorithm, it is particularly pernicious in reinforcement learning. Unlike supervised learning, where evaluations are based on fixed data sets, reinforcement learning relies on observations of the environment. In reinforcement learning, there is thus no data set the agent trains on, so it must create its own data. Overfitting can occur, by over-training because the agent is constantly being trained on one reward function and this results in the agent being unable to learn an optimal policy by maximizing a reward function rather than minimizing a loss function, which leads to inefficient performance.

10 CONCLUSION

In conclusion, this paper illustrates the influence of different reward-focused functions and the influence it has on the agent and the reinforcement model created during the training phase. Due to the fully observable and deterministic nature of the Snake environment, it allowed for a transparent environment in which manipulation would be easily observable while the noise of additional factors was kept to a minimum. Through this, the research question - "How and why do different reward functions influence the performance of an agent's reinforcement learning model during the training phase in the game Snake?" - and its hypothesis can be answered. The reasoning for the difference between reward functions was already highlighted in the discussion, it is clear that the Manhattan-based reward function was the best overall, it continued to improve even after the maximum number of games was reached while all the other functions had plateaued in their performance. The Manhattan reward-based function performs well because it minimizes the sparse rewards problem, it gives rewards for intermediate states. Giving a reward when the snake turns in the correct direction that will minimize the distance to the apple, helps the agent create a strategy that is based on one of the best Snake game heuristics. While the Inverse Manhattan reward-based function also uses reward shaping as a mechanism to incentivize traveling close to the apple, the training model suffers from the exploitation vs. exploration problem in which it is content with the minimal points obtained instead of finding a greater sum of points further away. The other reward functions were still quite sparse, and this created the delay reward problem to become apparent and plateau the snake's performance. Overall the performance could have been better analyzed if the implementation of a convolutional layer was added to the DQN, this would have most likely prevented the stagnation in the maximum average score that is apparent in the current results.

11 FUTURE WORK

In potential future work, the work done in this paper about testing an agent's performance under varying reward functions could be extended with hyperparameter tuning to test and observe its influences. Hyperparameters play a significant role in reinforcement learning and directly influence the performance of an agent. Hyperparameters are set by the developers of the model and can be quite time consuming. Nevertheless, hyperparameter tuning could change the performance of the presented reward functions and it would be worthwhile to test if poorly performing reward functions could be improved. In addition to the hyperparameter tuning DeepMind's EPIC method could be used to quantify differences between reward functions prior to the optimization step to avoid wasting training time on poorly performing functions [7].

Another interesting area to explore is training with multiple snakes. Training with multiple snakes is an area of machine learning which involves multi-agent systems. This exploration would allow an understanding of how snakes interact and observe their respective environment and each other. Multi-agent reinforcement learning expands upon conventional reinforcement learning by adding the challenge of other agents. One of these challenges comes from the fact that adding more agents means that a previously deterministic environment is turned into a stochastic one from a single agent's perspective [4]. In addition, generating a learning goal, keeping

track of other agents' learning, coordinating behavior, and scalability can hinder the success of multi-agent reinforcement learning models [3]. There have, however, been models which have been developed to deal with multiple learners successfully, such as coordination-free methods, which include the Team Q-learning algorithm and Distributed Q-learning algorithm, as well as coordination-based methods, and indirect coordination methods [3]. These models could represent interesting research areas to explore.

Another area of potential research is changing the environment. This could lead to an in-depth understanding of how agents interact in dynamic environments. Moreover, it could be explored how changes in the environment influence not only one snake, but also the learning of multiple snakes. The changes in the environment could be the amount of food, the number of snakes or changes in the game map by adding more walls or labyrinths. Furthermore, the application of different reward functions and different algorithms in different games, such as Pacman or Attari, could be researched. This field of research would allow a new perspective on how specific reward functions and algorithms influence and track the outcome of the long-term strategies developed by the agent in the respective games.

REFERENCES

[1] Christian Biseinere. 2019. Reinforcement learning algorithms in a multi-agent snake environment by Christian Biseinere. https://www.illuminatenrhc.com/post/reinforcement-learning-algorithms-in-a-multi-agent-snake-environment-by-christian-biseinere

[2] Jason Brownlee. 2019. Controlled experiments in machine learning. https://machinelearningmastery.com/controlled-experiments-in-machine-learning/

[3] Lucian Busoniu, Robert Babuska, and Bart De Schutter. 2010. *Multi-agent Reinforcement Learning: An Overview*. Vol. 310. 183–221. https://doi.org/10.1007/978-3-642-14435-6_7

[4] Lorenzo Canese, Gian Carlo Cardarilli, Luca Di Nunzio, Rocco Fazzolari, Daniele Giardino, Marco Re, and Sergio Spanò. 2021. Multi-agent reinforcement learning: A review of challenges and applications. *Applied Sciences* 11, 11 (2021), 4948.

[5] Jianqing Fan, Zhaoran Wang, Yuchen Xie, and Zhuoran Yang. 2020. A theoretical analysis of deep Q-learning. In *Learning for Dynamics and Control*. PMLR, 486–489.

[6] Banda Gerald. 2018. A brief review of independent, dependent and one sample t-test. *International Journal of Applied Mathematics and Theoretical Physics* 4, 2 (2018), 50–54.

[7] Adam Gleave, Michael Dennis, Shane Legg, Stuart Russell, and Jan Leike. 2020. Quantifying differences in reward functions. *arXiv preprint arXiv:2006.13900* (2020).

[8] Jörg Kaufmann and AG Schering. 2007. Analysis of variance ANOVA. *Wiley Encyclopedia of Clinical Trials* (2007).

[9] Benjamin James Lansdell, Prashanth Ravi Prakash, and Konrad Paul Kording. 2019. Learning to solve the credit assignment problem. *arXiv preprint arXiv:1906.00889* (2019).

[10] Patrick Loeber. 2021. snake-ai-pytorch. https://github.com/python-engineer/snake-ai-pytorch.

[11] Vlad Ovidiu Chelcea and Björn Ståhl. 2018. Deep Reinforcement Learning for Snake.

[12] Brendan O'Donoghue, Ian Osband, Remi Munos, and Volodymyr Mnih. 2018. The uncertainty bellman equation and exploration. In *International Conference on Machine Learning*. 3836–3845.

[13] Jesus Rodriguez. 2017. 6 types of artificial intelligence environments. https://jrodthoughts.medium.com/6-types-of-artificial-intelligence-environments-825e3c47d998

[14] Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. 2019. A survey of deep reinforcement learning in video games. *arXiv preprint arXiv:1912.10944* (2019).

[15] Sharanya Suvain Goyal, Vaibhav Somani. 2020. Reinforcement Learning using Convolutional Neural Network for Game Prediction. https://doi.org/DOI:10.35940/ijitee.G5698.069820

[16] Shehab Yasser. 2020. A brief history of reinforcement learning in Game Play. https://medium.com/swlh/a-brief-history-of-reinforcement-learning-in-game-play-d0861b2b74ef

[17] Chiyuan Zhang, Oriol Vinyals, Remi Munos, and Samy Bengio. 2018. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893* (2018).

A DATA FOR THE 10 POINT THRESHOLD

Crossed avg of 10 at game:	1	2	3	4	5	Avg
Manhattan	168	177	173	129	189	167,2
Simple	145	124	117	118	125	125,8
Punishment Reward	-	355	386	717	-	486
Increased Reward	166	267	139	133	141	167,2
Inverse of the Manhattan	-	-	-	-	-	Never

Fig. 8. Number of games at which an agent crossed the 10 point average threshold

B DETAILED STATISTICAL RESULTS

t-Test: Two-Sample Assuming Unequal Variances

	Simple	*Manhattan*
Mean	33,1198	33,56148
Variance	52,1977115	61,94976216
Observations	5000	5000
Hypothesized Mean Difference	0	
df	9926	
t Stat	-2,923207096	
P(T<=t) one-tail	0,001736142	
t Critical one-tail	1,645007154	
P(T<=t) two-tail	0,003472284	
t Critical two-tail	1,960203009	

Fig. 9. t-test between the simple reward function and the Manhattan distance

t-Test: Two-Sample Assuming Unequal Variances

	Simple	*Increased Reward*
Mean	33,1198	29,5786
Variance	52,1977115	49,78491502
Observations	5000	5000
Hypothesized Mean Difference	0	
df	9992	
t Stat	24,79547094	
P(T<=t) one-tail	4,5667E-132	
t Critical one-tail	1,64500614	
P(T<=t) two-tail	9,13E-132	
t Critical two-tail	1,96020143	

Fig. 10. t-test between the simple reward function and the Increased reward

t-Test: Two-Sample Assuming Unequal Variances

	Simple	*Punishment reward*
Mean	33,1198	15,413
Variance	52,1977115	29,53155331
Observations	5000	5000
Hypothesized Mean Difference	0	
df	9284	
t Stat	138,4956994	
P(T<=t) one-tail	0	
t Critical one-tail	1,645017772	
P(T<=t) two-tail	0	
t Critical two-tail	1,96021954	

Fig. 11. t-test between the simple reward function and the Punishment reward function

t-Test: Two-Sample Assuming Unequal Variances

	Simple	*Inverse of the Manhattan*
Mean	**33,1198**	**0,60936**
Variance	**52,1977115**	**0,224629316**
Observations	**5000**	**5000**
Hypothesized Mean Difference	**0**	
df	**5042**	
t Stat	**317,5043244**	
P(T<=t) one-tail	**0**	
t Critical one-tail	**1,645155898**	
P(T<=t) two-tail	**0**	
t Critical two-tail	**1,960434598**	

Fig. 12. t-test between the simple reward function and the Inverse of the Manhattan distance

C CODE FOR THE SIMPLE REWARD

```
81          # 3. check if game over
82          reward = 0
83          game_over = False
84          if self.is_collision() or self.frame_iteration > 100 * len(self.snake):
85              game_over = True
86              reward = -10
87              return reward, game_over, self.score
88
89          # 4. place new food worth 10 reward or just move
90          if self.head == self.food:
91              self.score += 1
92              reward = 10
93              self._place_food()
94          else:
95              self.snake.pop()
```

Fig. 13. Code for the simple reward

D CODE FOR THE MANHATTAN DISTANCE

```
        # calculate all possible vals of next state
        cur_up = (self.head.x, self.head.y + 20) if (
                self.head.y <= 480 and not self.collides(self.head.x, self.head.y + 20)) else (INF, INF)
        cur_down = (self.head.x, self.head.y - 20) if (
                self.head.y >= -480 and not self.collides(self.head.x, self.head.y - 20)) else (INF, INF)
        cur_right = (self.head.x + 20, self.head.y) if (
            self.head.x <= 480 and not self.collides(self.head.x + 20, self.head.y) + 20, self.head.y) else (
            INF, INF)
        cur_left = (self.head.x - 20, self.head.y) if (
                self.head.x >= -480 and not self.collides(self.head.x - 20, self.head.y)) else (INF, INF)

        # apply manhattan distance heuristic on all four possibilities
        man_dist_up = abs(self.food.x - cur_up[0]) + abs(self.food.y - cur_up[1])
        man_dist_down = abs(self.food.x - cur_down[0]) + abs(self.food.y - cur_down[1])
        man_dist_right = abs(self.food.x - cur_right[0]) + abs(self.food.y - cur_right[1])
        man_dist_left = abs(self.food.x - cur_left[0]) + abs(self.food.y - cur_left[1])

        # obtain index of move with lowest manahttan distance heuristic
        v, i = min((v, i) for (i, v) in enumerate([man_dist_up, man_dist_down, man_dist_right, man_dist_left]))
```

Fig. 14. Code for the Manhattan distance

```
        # obtain index of move with lowest manahttan distance heuristic
        v, i = min((v, i) for (i, v) in enumerate([man_dist_up, man_dist_down, man_dist_right, man_dist_left]))

        # determines direction based on lwest manhattan distance heuristic
        if i == 0:
            man_direction = Direction.UP if self.direction != "down" else Direction.RIGHT
        elif i == 1:
            man_direction = Direction.DOWN if self.direction != "up" else Direction.LEFT
        elif i == 2:
            man_direction = Direction.RIGHT if self.direction != "left" else Direction.UP
        else:
            man_direction = Direction.LEFT if self.direction != "right" else Direction.DOWN

        # 2. move
        self._move(action)  # update the head
        self.snake.insert(0, self.head)
```

Fig. 15. Code for the Manhattan distance

```
        # reward based on Manhattan function
        if self.direction == man_direction:
            reward = 1
```

Fig. 16. Code for the Manhattan distance

E CODE FOR THE INVERSE OF THE MANHATTAN DISTANCE

```
        # calculate all possible values of next state
        cur_up = (self.head.x, self.head.y + 20) if (
                self.head.y <= 480 and not self.collides(self.head.x, self.head.y + 20)) else (INF, INF)
        cur_down = (self.head.x, self.head.y - 20) if (
                self.head.y >= -480 and not self.collides(self.head.x, self.head.y - 20)) else (INF, INF)
        cur_right = (self.head.x + 20, self.head.y) if (
            self.head.x <= 480 and not self.collides(self.head.x + 20, self.head.y) + 20, self.head.y) else (
            INF, INF)
        cur_left = (self.head.x - 20, self.head.y) if (
                self.head.x >= -480 and not self.collides(self.head.x - 20, self.head.y)) else (INF, INF)

        # apply manhattan distance heuristic on all four possibilities
        man_dist_up = abs(self.food.x - cur_up[0]) + abs(self.food.y - cur_up[1])
        man_dist_down = abs(self.food.x - cur_down[0]) + abs(self.food.y - cur_down[1])
        man_dist_right = abs(self.food.x - cur_right[0]) + abs(self.food.y - cur_right[1])
        man_dist_left = abs(self.food.x - cur_left[0]) + abs(self.food.y - cur_left[1])

        # obtain index of move with lowest manahttan distance heuristic
        v, i = min((v, i) for (i, v) in enumerate([man_dist_up, man_dist_down, man_dist_right, man_dist_left]))

```

Fig. 17. Code for the Inverse of theManhattan distance

```
108
109         # Inverse Manhattan reward based function, reward based on distance to apple
110         reward = (float((MAX_BLOCK_STEPS - v))/55.0)
```

Fig. 18. Code for the Inverse of the Manhattan distance

F CODE FOR THE PUNISHMENT REWARD

```
        reward = 0
        game_over = False
        if self.is_collision() or self.frame_iteration > 100 * len(self.snake):
            game_over = True
            reward = -10
            return reward, game_over, self.score


        # 4. place new food worth 1 reward or just move
        if self.head == self.food:
            self.score += 1
            reward = 1
            self._place_food()
        else:
            self.snake.pop()

```

Fig. 19. Code for the punishment reward

G CODE FOR THE INCREASING REWARD

```
88          # 4. increase apple reward by 2.5 every 5 points scored, place new food or just move
89          if self.head == self.food:
90              self.score += 1
91              games_rounded_5 = self.score - (self.score % 5)
92              reward = 10 + (games_rounded_5/2)
93              self._place_food()
94          else:
95              self.snake.pop()
```

Fig. 20. Code for the increasing reward