CS 4080/5080

Reinforcement Learning
Fall 2021

Class Project Papers:
Proposal, Midterm and Final

Undergraduate Papers

Proposal CS 4080 Fall 2021

Beating Snake Using Reinforcement Learning

Shawn Ringler and **Frank Torres**
University of Colorado Colorado Springs

Abstract

Reinforcement Learning has seen an increase of interest due to Google's Deepmind Alpha Go algorithm beating human players. This increase has shown light towards other games such as the game of Snake. Using reinforcement learning to learn the game of snake has many applications from path finding, obstacle avoidance, and even the potential to mimic the organic sense of odor detention found in mammals. In this paper we propose a method of creating a reinforcement learning agent that can learn how to play snake given little as information about the game as possible.

1 Intro

Snake is a 2D game where a snake is placed onto a grid and must fetch apples that are randomly placed onto the grid. The snake can move in four directions: up, down, left and right. As the snake moves on the grid it drags it's body behind it. The snake's body starts at a fix length, but increases in size when ever an apple is fetched. When an apple is fetched, another one is randomly placed onto the grid. The objective of the game is to grab as many apples as the snake can before the snake runs into it's own tail/body or the border of the grid.

Snake has been a game produced on many different systems and platforms through very different means over the years. This had lead to the question: can machines learn to play Snake, and can they perform well? By using reinforcement learning, we hope to answer this question.

2 Background

Reinforcement learning is a version of machine learning where the learner, in this case the machine, is not told what it must do but it must learn from it's own actions (Sutton and Barto 2020). There has been a recent interest with the rise of reinforcement learning such as Google's Deepmind Go playing algorithm called AlphaGo making headlines for doing the unthinkable and beating human players (Holcomb et al. 2018). This had led to a lot of other research into creating reinforcement learning models to play and learn how to be better then humans at certain games. One of these such games has been the game of snake.

To be able to use reinforcement learning, one must utilize Markov Decision processes. Which is a tuple of five distinct variables:

- Set of all states.
- Set of all actions that can be performed by the agent.
- The probability transition function given the current state and actions.
- The reward function that gives the reward given from going from one state to the next.
- The discount variable that motivates the agent towards an optimal path.

Snake has been used as an example for machine learning, not only to teach about reinforcement learning but to try and improve other algorithms as well. Such as trying to train a group of drones to navigate a 2D plane while learning to avoid other drones paths (Wu et al. 2019). Many other attempts at making reinforcement learning agents for the game of snake follow in the next section.

3 Related Work

One ambitious attempt at making a reinforcement learning agent for the game of snake was intended to showcase how drones could automatically navigate around a constantly changing environment (Wu et al. 2019). This kind of autonomy is proposed to have practical applications in the industrial field in terms of automated inspections of oil and gas fields. Another particularly interesting use case for these proposed autonomous drones is in search and rescue scenarios where injured people need to be located and materials (such as first aid, rescue equipment, etc.) need to be dispatched. Going further with this idea, the need for a collection of these drones to collaborate and work together in order to aid multiple people at different locations in a complex disaster scenario could be modeled using a multi-agent snake game where there are multiple snakes and multiple target locations. This of course creates an extra layer of complexity in terms of collision avoidance and collaborative pathing. This method used Deep Reinforcement Learning due to the vast state space. Specifically they used multiple Deep Q-Network (DQN) algorithms to achieve their results.

An interesting technique that helped speed up the learning process was the use of an "Odor effect" in which the apple is surrounded by a decreasing reward gradient which mimics the biological mechanism of smell, effectively reducing the blindness of the agent (Wu et al. 2019).

Another attempt at making a reinforcement learning was created by Yashshree Patil. Patil's verion of a reinforcement learning algorithm for snake was done with a unique set of states and rewards. The set of states given to the agent were 12 Booleans based on where the snake was relative the apple, obstacles and where the snake was heading. The set of rewards were then a collection of four reward states. First, if the snake obtained then apple, get a reward of +10. Second if the snake is heading towards the apple +0. Third, if the snake is heading away from the apple -1. Finally, if the snake hits a wall or it's own body give it a reward of -100 (Patil 2020).

In a paper titled: "Group Equivariant Deep Reinforcement Learning" Mondal et al explore the uses of games like snake and Pac-man to help improve the redundancy in for learning algorithms in symmetry-transformation equivariant representations of an input environment and its states. (Mondal, Nair, and Siddiqi 2020).

In another paper from 2016 titled "Exploration of Reinforcement Learning to SNAKE", Bowei Ma et al try to tackle the challenge of reducing the extremely large state space in snake to a minimum. This large state space comes from the fact that the location and size of the snake and its tail are constantly changing dynamically after every time step. This makes it difficult to achieve computationally fast learning results given the time complexity. They attempt to solve this by reducing the state space down to a 5-tuple only containing information on the relative position of the snake in relation to the target location. The Reinforcement Learning methods that they implement are Q-Learning and SARSA which seem to be effective in some sense. To measure the effectiveness of these methods, they use a non Reinforcement Learning "deterministic heuristic algorithm for Snake" which is able to approximate an optimal solution algorithm for playing the game of Snake (Ma, Tang, and Zhang 2016). This is then used as a benchmark to measure the success of their Q-Learning and SARSA methods. The SARSA algorithm seemed to outperform the Q-Learning algorithm in this case due to the faster rate of learning and overall effectiveness. (Ma, Tang, and Zhang 2016)

4 Methodology

For our system we will be using a system of reinforcement learning that incorporates Q-Learning. We shall be programming the game of snake and the machine learning agent in Python. We will be using the State-Action-Reward-State-Action method, or SARSA along side Q-learning for the training of the Agent similar to how Almaki et al. used in their version of "Exploration of Reinforcement Learning to Play Snake Game" (Almalki and Wocjan 2019). With the potential of later developing a Deep Q-Learning method as well.

Something to consider in terms of the mathematical complexity of Snake is that it is an NP-Hard problem. This can be seen in a paper by Viglietta in which it was proven that "any game exhibiting location traversal" as well as any game featuring "collectible tokens" is indeed NP-Hard (Viglietta 2014) . Obviously Snake contains both of these things which shows that creating an agent that can effectively play snake and learn an optimal policy isn't exactly a trivial matter.

What follows are more in depth sections on how we plan on implementing the individual distinct portions of the reinforcement learning algorithm.

4.1 States

In their paper titled: "Snake Game Using Reinforcement Learning" Yashshree Patil uses a state system based on a 12 Boolean system that we will use. These states include:

- Apple is above the snake
- Apple is below the snake
- Apple is to the right of the snake
- Apple is to the left of the snake
- Wall or snake body above snake
- Wall or snake body below snake
- Wall or snake body is to the right of the snake
- Wall or snake body is to the left of the snake
- Snake is facing up
- Snake is facing down
- Snake is facing right
- Snake is facing left

(Patil 2020) These states allow for the agent to have multiple ways on knowing where it is and what is going on without having to feed in the entire game grid to the agent. This states however will be needed to be changed and fixed by the game board between each move of the agent. However without telling the agent any info besides these 12 Booleans existing and being turned on or off, the agent will have to learn the deeper meaning behind these states and what is the action for it to take.

4.2 Reward

To help the snake agent learn what actions have good and bad outcomes, a simple reward function is going to be implemented. We want to include two different reward functions. The first of which will simply give a positive reward of +10 if the snake obtains an apple. Then if the agent hits a wall or hits it's own body the game will end and the agent will be rewarded with a -100 reward.

The second method we are planning on using uses the previous function as a base, but with the added ability of odor or smell to the apple. This was discussed in Chunxue Wu et. al's paper: "UAV Autonomous Target Search Based on Deep Reinforcement Learning in Complex Disaster Scene". This reward function not only gives a high reward for obtaining an apple, but also gives the surrounding areas around the apple an "odor" reward of decreasing values as the distance increases which essentially provides a reward gradient that mimics the sense of smell.

4.3 Actions

Snake is quite simple to control and therefor the actions are quite simple as well. Snake has only four actions:

- Move the snake up one space on the grid
- Move the snake down one space on the grid
- Move the snake left one space on the grid
- Move the snake right one space on the grid

With these four actions, the agent will be able to move the snake freely towards the fruit or around the grid.

4.4 Environment

Snake is a 2D grid based game. As such the environment will reflect as such. The environment will be the grid, or board of the game in size of 10x10. This way it is easy to generate and create the grid without the need for massive amount of computer power to generate large boards.

As mentioned previously snake is a game in which the goal of the game is to have a "snake" on the board obtain or consume randomly place apples on the board. When the snake eats the apple, the snake will grow in length. If the snake hits a boundary of the grid, or hits its own body, then the game will end. In terms of environment this means that the reward state will be completely random in its location. Meaning that the snake will have to figure out where the fruit is while trying not to enter a game over state.

4.5 Possible enhancements

One possible enhancement that we can see is to implement a DQN learning algorithm to this project. This has the possibility of enhancing and improving the network's ability to learn and remember it's actions betters (Wu et al. 2019).

5 Evaluation

We will be using a score based system to gauge the performance of the current model. Each time the snake obtains or eats the apple it will gain 1 point. If the snake hits it's own tail or hits a wall the game will be over and the score will be finalized. The higher the score of the agent, the better performance it would have.

Since we plan on having multiple types of agents, we can use this score function to compare how well the models do compared to one another based on their training, and reward functions.

6 Timeline

Below is a general time line table of how we believe progress on the project will be made. These dates, except for those with turn in requirements, are likely to move as the project is implemented.

- 9/22/2021: Turn in proposal.
- 10/01/2021: Have Snake game created and finished.
- 10/15/2021: Have first implementation of RL agent created.
- 10/20/2021: Complete working demo and midterm report.
- 11/05/2021: Update and improve RL agent.
- 12/3/2021: Update and improve RL agent in preparation for Final version.
- 12/13/2021: Complete Final version of project and report.

7 Conclusion

In this proposal we have outlined our plans for creating a reinforcement learning agent that will learn and play the game snake. This will be done by Q-learning and state-action-reward-state-action method of learning. This way we can generate two agents from these training methods to compare to at the end. We also plan to use potentially two different agents with two different rewards states. One based on the current state of the agent and the other using the distance the agent is away from the reward as an incentive to get closer to the reward through organic odor smelling, similar to that found in nature. Potentially if there is time, we also plan to use DQN as well to promote further learning possibilities. This would result in a total of six possible agents from 3 ways of training and 2 different ways of generating our reward function.

References

Almalki, A. J., and Wocjan, P. 2019. Exploration of reinforcement learning to play snake game. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, 377–381. IEEE.

Holcomb, S. D.; Porter, W. K.; Ault, S. V.; Mao, G.; and Wang, J. 2018. Overview on deepmind and its alphago zero ai. In *Proceedings of the 2018 international conference on big data and education*, 67–71.

Ma, B.; Tang, M.; and Zhang, J. 2016. Exploration of reinforcement learning to snake.

Mondal, A. K.; Nair, P.; and Siddiqi, K. 2020. Group equivariant deep reinforcement learning. *arXiv preprint arXiv:2007.03437*.

Patil, Y. 2020. Snake game using reinforcement learning. *Academia.edu*.

Sutton, R. S., and Barto, A. G. 2020. *Reinforcement Learning: An Introduction*. The MIT Press, 2nd edition.

Viglietta, G. 2014. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems* 54(4):595–621.

Wu, C.; Ju, B.; Wu, Y.; Lin, X.; Xiong, N.; Xu, G.; Li, H.; and Liang, X. 2019. Uav autonomous target search based on deep reinforcement learning in complex disaster scene. *IEEE Access* 7:117227–117245.

Midterm Paper CS 4080 Fall 2021

Beating Snake Using Reinforcement Learning Midterm

Shawn Ringler and **Frank Torres**
University of Colorado Colorado Springs

Abstract

Reinforcement Learning has seen an increase of interest due to Google's Deepmind Alpha Go algorithm beating human players. This increase has shown light towards other games such as the game of Snake. Using reinforcement learning to learn the game of Snake has many applications from path finding, obstacle avoidance, and even the potential to mimic the organic sense of odor detention found in mammals. In this paper we outline our method of creating a reinforcement learning agent that can learn how to play Snake given as little information about the game as possible.

1 Introduction

Snake is a 2D game where a snake is placed onto a grid and must fetch apples that are randomly placed onto the grid. The snake can move in four directions: up, down, left and right. As the snake moves on the grid it drags its body behind it. The snake's body starts at a fix length, but increases in size when ever an apple is fetched. When an apple is fetched, another one is randomly placed onto the grid. The objective of the game is to grab as many apples as the snake can before the snake runs into its own tail, body or the border of the grid.

Snake has been a game produced on many different systems and platforms through very different means over the years. This had lead to the question: can machines learn to play Snake, and can they perform well? By using reinforcement learning, we hope to answer this question.

2 Background

Reinforcement learning is a version of machine learning where the learner, in this case the machine, is not told what it must do but it must learn from its own actions (Sutton and Barto 2020). There has been a recent interest with the rise of reinforcement learning such as Google's Deepmind Go playing algorithm called AlphaGo making headlines for doing the unthinkable and beating human players (Holcomb et al. 2018). This had led to a lot of other research into creating reinforcement learning models to play and learn how to be better than humans at certain games. One such game has been the game of Snake.

To be able to use reinforcement learning, one must utilize Markov Decision processes. Which is a tuple of five distinct variables:

- Set of all states.
- Set of all actions that can be performed by the agent.
- The probability transition function given the current state and actions.
- The reward function that gives the reward given from going from one state to the next.
- The discount variable that motivates the agent towards an optimal path.

Snake has been used as an example for machine learning, not only to teach about reinforcement learning but to try and improve other algorithms as well, such as trying to train a group of drones to navigate a 2D plane while learning to avoid other drones paths (Wu et al. 2019). Many other attempts at making reinforcement learning agents for the game of Snake follow in the next section.

3 Related Work

One ambitious attempt at making a reinforcement learning agent for the game of Snake was intended to showcase how drones could automatically navigate around a constantly changing environment (Wu et al. 2019). This kind of autonomy is proposed to have practical applications in the industrial field in terms of automated inspections of oil and gas fields. Another particularly interesting use case for these proposed autonomous drones is in search and rescue scenarios where injured people need to be located and materials (such as first aid, rescue equipment, etc.) need to be dispatched. Going further with this idea, the need for a collection of these drones to collaborate and work together in order to aid multiple people at different locations in a complex disaster scenario could be modeled using a multi-agent Snake game where there are multiple snakes and multiple target locations. This of course creates an extra layer of complexity in terms of collision avoidance and collaborative pathing. This method used Deep Reinforcement Learning due to the vast state space. Specifically they used multiple Deep Q-Network (DQN) algorithms to achieve their results. An interesting technique that helped speed up the learning process was the use of an Odor effect in which the apple is

surrounded by a decreasing reward gradient which mimics the biological mechanism of smell, effectively reducing the blindness of the agent (Wu et al. 2019).

Another attempt at making a reinforcement learning was created by Patil (2020). Patil's version of a reinforcement learning algorithm for snake was done with a unique set of states and rewards. The set of states given to the agent were 12 Booleans based on where the snake was relative the apple, obstacles and where the snake was heading. The set of rewards were then a collection of four reward states. First, if the snake obtained then apple, get a reward of +10. Second if the snake is heading towards the apple +0. Third, if the snake is heading away from the apple -1. Finally, if the snake hits a wall or it's own body give it a reward of -100 (Patil 2020).

Mondal et al. also explored the uses of games like Snake and Pac-man to help improve the redundancy in learning algorithms for symmetry-transformation equivariant representations of an input environment and its states. (Mondal, Nair, and Siddiqi 2020).

In another paper from 2016, Bowei Ma et al. try to tackle the challenge of reducing the extremely large state space in Snake to a minimum. This large state space comes from the fact that the location and size of the snake and its tail are constantly changing dynamically after every time step. This makes it difficult to achieve computationally fast learning results given the time complexity. They attempt to solve this by reducing the state space down to a 5-tuple only containing information on the relative position of the snake in relation to the target location. The Reinforcement Learning methods that they implement are Q-Learning and SARSA which seem to be effective in some sense. To measure the effectiveness of these methods, they use a non Reinforcement Learning deterministic heuristic algorithm for Snake which is able to approximate an optimal solution algorithm for playing the game of Snake (Ma, Tang, and Zhang 2016). This is then used as a benchmark to measure the success of their Q-Learning and SARSA methods. The SARSA algorithm seemed to outperform the Q-Learning algorithm in this case due to the faster rate of learning and overall effectiveness. (Ma, Tang, and Zhang 2016)

4 Methodology

For our system, we will be using a system of reinforcement learning that incorporates Q-Learning (Watkins and Dayan 1992). We shall be programming the game of Snake and the machine learning agent in Python. We will be using Q-learning for the training of the Agent similar to how Almaki et al. used it in their version of a reinforcement learning agent, with the potential of later developing a SARSA and Deep Q-Learning method as well (Almalki and Wocjan 2019).

Something to consider in terms of the mathematical complexity of Snake is that it is an NP-Hard problem. This can be seen in a paper by Viglietta in which it was proven that "any game exhibiting location traversal" as well as any game featuring "collectible tokens" is indeed NP-Hard (Viglietta 2014) . Obviously Snake contains both of these things which shows that creating an agent that can effectively play snake and learn an optimal policy isn't exactly a trivial matter.

What follows are more in depth sections on how we plan on implementing the individual distinct portions of the reinforcement learning algorithm.

4.1 States

In their paper (Patil 2020) uses a state system based on a 12 Boolean system that we will use. These states include:

- Apple is above the snake
- Apple is below the snake
- Apple is to the right of the snake
- Apple is to the left of the snake
- Wall or snake body above snake
- Wall or snake body below snake
- Wall or snake body is to the right of the snake
- Wall or snake body is to the left of the snake
- Snake is facing up
- Snake is facing down
- Snake is facing right
- Snake is facing left

These states allow for the agent to have multiple ways on knowing where it is and what is going on without having to feed in the entire game grid to the agent. This states however will be needed to be changed and fixed by the game engine between each move of the agent. However without telling the agent any info besides these 12 Booleans existing and being turned on or off, the agent will have to learn the deeper meaning behind these states and what is the action for it to take.

Another thing to note is that while configuring the state space with this method will heavily reduce the size of the state space for this reinforcement learning agent, it will still leave the state space to have a size of 2^{12} or 4096 possible states.

Figure 1 shows how these states are being shown during the demo of the project for the optimal path solution.

4.2 Reward

To help the snake agent learn what actions have good and bad outcomes, a simple reward function is going to be implemented. We want to include two different reward functions. The first of which will simply give a positive reward of +100 if the snake obtains an apple. Then if the agent hits a wall or hits it's own body the game will end and the agent will be rewarded with a -100 reward.

The second method we are planning on using uses the previous function as a base, but with the added ability of odor to the apple (Wu et al. 2019). This reward function not only gives a high reward for obtaining an apple, but also gives the surrounding areas around the apple an "odor" reward of decreasing values as the distance increases which essentially provides a reward gradient that mimics the sense of smell found in nature.

Figure 1: States as shown in during the demo of the optimal path. The Green squares indicate a 1, or true, while the Black Squares indicate a 0, of false. In this example, the apple is down-right from the snake which is facing to the right with a wall to it's left.

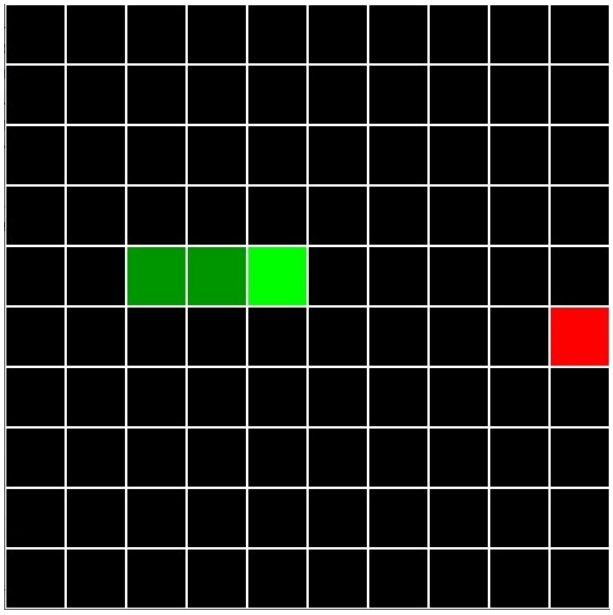

Figure 2: Grid Environment as shown during the demo of the learning agent. Which showcases the states shown from Figure 1.

4.3 Actions

Snake is quite simple to control and therefore the actions are quite simple as well. Snake has only four actions:

- Move the snake up one space on the grid
- Move the snake down one space on the grid
- Move the snake left one space on the grid
- Move the snake right one space on the grid

With these four actions, the agent will be able to move the snake freely towards the fruit or around the grid.

4.4 Environment

Snake is a 2D grid based game. As such the environment will reflect as such. The environment will be the grid, or board of the game in size of 10x10. This way it is easy to generate and create the grid without the need for massive amount of computer power to generate large boards. This grid can be showcased in Figure 2.

As mentioned previously Snake is a game in which the goal of the game is to have a "snake" on the board obtain or consume randomly place apples on the board. When the snake eats the apple, the snake will grow in length. If the snake hits a boundary of the grid, or hits its own body, then the game will end. In terms of environment this means that the reward state will be completely random in its location. Meaning that the snake will have to figure out where the fruit is while trying not to enter a game over state.

4.5 Possible enhancements

One current issue that is known with the current model is the use of Q-Learning, as Q-Learning has been shown to over fit data when training (Van Hasselt, Guez, and Silver 2016). This over fitting can become an issue with trying to learn with large amounts of episodes. When trying to learn a large amount of episodes for our current model of snake, it was found that the snake would sometimes learn incorrectly the wrong path early on and would then try to keep using that wrong path until training had stopped. One possible enhancement to Q-Learning that we can see is to implement both SARSA and a DQN learning algorithm to this project. The DQN has the possibility of enhancing and improving the network's ability to learn and remember it's actions better (Wu et al. 2019). This would greatly help with learning since one of the biggest issues with the current model would be how the model can get itself stuck in a loop, ending the game.

5 Evaluation

We will be using a score based system to gauge the performance of the current model. Each time the snake obtains or eats the apple it will gain 1 point. If the snake hits its own tail or hits a wall the game will be over and the score will be finalized. The higher the score of the agent, the better performance it would have.

Since we plan on having multiple types of agents, we can use this score function to compare how well the models do compared to one another based on their training, and reward functions.

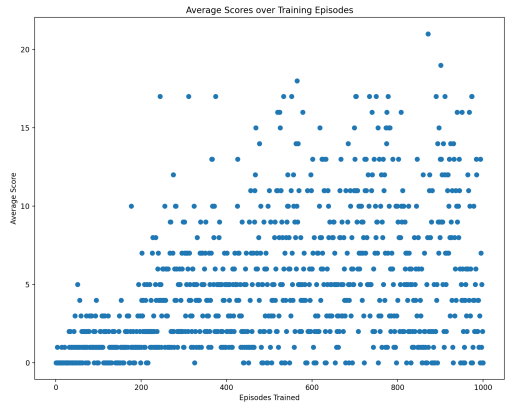

Figure 3: Scores represent 1 optimal run after each training episode for 1 agent training on 1000 episodes.

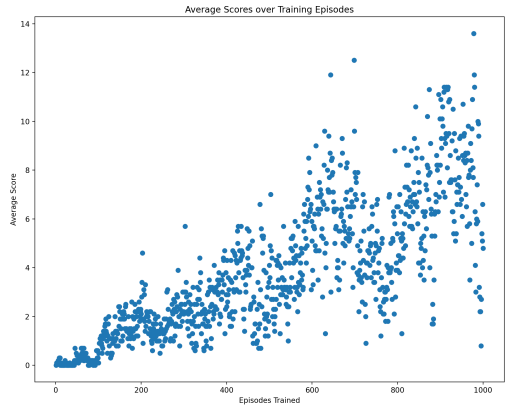

Figure 4: Scores represent the average of 10 optimal runs after each training episode for 1 agent training on 1000 episodes.

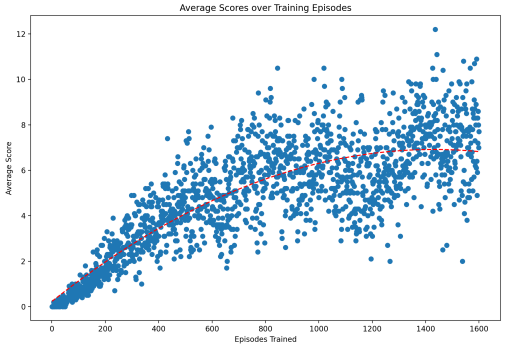

Figure 5: Scores represent the average of 1 optimal run per agent after each training episode for 10 agents training on 1600 episodes.

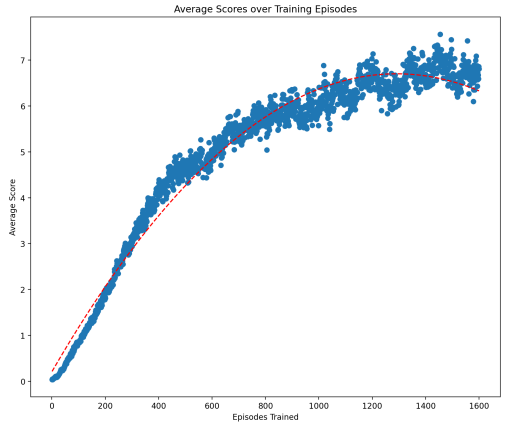

Figure 6: Scores represent the average of 10 optimal runs per agent after each training episode for 100 agents training on 1600 episodes.

6 Results

For the evaluation of the model we simply used the score system found within the game Snake. Each time the snake gets the apple, or fruit, a score of one is added to the total score. So far we have measured the performance of our Q-Learning implementation using several different methods. The graphs of using these measurements can be seen in figures 3, 4, 5, and 6.

Figure 3 showcases the discrete values of a single agent learning over 1,000 episodes. Very early on the snake is shown to not earn many points, but as the training continued the score varied quite a bit later on, resulting in a max score of 22 around episode 900. However, most of the scores generated by this agent were shown to be either low scoring or zero.

Figure 4 showcases 1 agent learning over 1,000 episodes and then taking the average score of 10 optimal runs after each episode of training. This results in a graph that better visually represents how the agent learns over a given amount of time. Very early on the agent does not score many points but as training goes on, the average score increases overall.

Figure 5 showcases 10 different agents being trained over 1,600 episodes and having the average of 10 optimal runs graphed. This better represents the learning curve of the average agent.

Figure 6 showcases 100 different agents being trained over 1,600 episodes and having the average of 10 optimal runs per agent graphed. This graph is the best showcase for showing the learning curve of the average agent. From this we can see a stark increase in score from the very beginning, but then a slow decrease over time maybe to even a decrease in overall performance towards the end.

Due Date	Event	Status
9/22/2021	Turn in Proposal	Completed
10/01/2021	Complete Snake Game	Completed
10/15/2021	First Implementation	Completed
10/27/2021	Complete Midterm	Completed
11/05/2021	Implement DQN	Not Done
12/03/2021	Finalize Model	Not Done
12/13/2021	Complete Final	Not Done

Table 1: Timeline of events planned for this project with due dates and status.

7 Updated Timeline

The updated timeline for this project can be seen in Table 1. This table showcases the events for this project, alongside their respective due date and current status.

8 Conclusion

In this paper we have shown our current progress with creating a reinforcement learning agent for the game of Snake. This agent utilizes a Q-Learning method with a 12 boolean state space. Through these methods the agent is able to perform quite well but quickly reaches a plateau in learning. For the future, using a DQN to replace the Q-Learning method may help to improve the overall performance of the agent.

References

Almalki, A. J., and Wocjan, P. 2019. Exploration of reinforcement learning to play snake game. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, 377–381. IEEE.

Holcomb, S. D.; Porter, W. K.; Ault, S. V.; Mao, G.; and Wang, J. 2018. Overview on deepmind and its alphago zero ai. In *Proceedings of the 2018 international conference on big data and education*, 67–71.

Ma, B.; Tang, M.; and Zhang, J. 2016. Exploration of reinforcement learning to snake.

Mondal, A. K.; Nair, P.; and Siddiqi, K. 2020. Group equivariant deep reinforcement learning. *arXiv preprint arXiv:2007.03437*.

Patil, Y. 2020. Snake game using reinforcement learning. *Academia.edu*.

Sutton, R. S., and Barto, A. G. 2020. *Reinforcement Learning: An Introduction*. The MIT Press, 2nd edition.

Van Hasselt, H.; Guez, A.; and Silver, D. 2016. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30.

Viglietta, G. 2014. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems* 54(4):595–621.

Watkins, C. J., and Dayan, P. 1992. Q-learning. *Machine learning* 8(3-4):279–292.

Wu, C.; Ju, B.; Wu, Y.; Lin, X.; Xiong, N.; Xu, G.; Li, H.; and Liang, X. 2019. Uav autonomous target search based on deep reinforcement learning in complex disaster scene. *IEEE Access* 7:117227–117245.

Final Paper CS 4080 Fall 2021

Beating Snake Using Reinforcement Learning Final

Shawn Ringler and **Frank Torres**
University of Colorado Colorado Springs

Abstract

Reinforcement Learning has seen an increase of interest due to Google's Deepmind Alpha Go algorithm beating human players. This increase has shown light towards other games such as the game of Snake. Using reinforcement learning to learn the game of Snake has many potential applications from path finding, obstacle avoidance, and even the potential to mimic the organic sense of odor detention found in mammals. In this paper we outline our methods of creating reinforcement learning agents using both Q-Learning and a DQN (Deep Q-Network) that can learn how to play Snake given as little information about the game as possible. After implementing the Snake environment and these reinforcement learning methods, we were able to reliably train an agent to achieve scores that potentially rival the scores of a human. It is also important to note that our minimized state space design lets us train our Q-table and Q-network very quickly although this puts a ceiling on how well the agent performs when the Snake becomes longer.

1 Introduction

Snake is a 2D game where a snake is placed onto a grid and must fetch apples that are randomly placed onto the grid. The snake can move in four directions: up, down, left and right. As the snake moves on the grid it drags its body behind it. The snake's body starts at a fix length, but increases in size when ever an apple is fetched. When an apple is fetched, another one is randomly placed onto the grid. The objective of the game is to grab as many apples as the snake can before the snake runs into its own tail, body or the border of the grid.

Snake has been a game produced on many different systems and platforms through very different means over the years. This had lead to the question: can machines learn to play Snake, and can they perform well? By using reinforcement learning, we hope to answer this question.

2 Background

Reinforcement learning is a version of machine learning where the learner, in this case the machine, is not told what it must do but it must learn from its own actions (Sutton and Barto 2020). There has been a recent interest with the rise of reinforcement learning such as Google's Deepmind Go playing algorithm called AlphaGo making headlines for doing the unthinkable and beating human players (Holcomb et al. 2018). This had led to a lot of other research into creating reinforcement learning models to play and learn how to be better than humans at certain games. One such game has been the game of Snake.

To be able to use reinforcement learning, one must utilize Markov Decision processes. Which is a tuple of five distinct variables:

- Set of all states.
- Set of all actions that can be performed by the agent.
- The probability transition function given the current state and actions.
- The reward function that gives the reward given from going from one state to the next.
- The discount variable that motivates the agent towards an optimal path.

Snake has been used as an example for machine learning, not only to teach about reinforcement learning but to try and improve other algorithms as well, such as trying to train a group of drones to navigate a 2D plane while learning to avoid other drones paths (Wu et al. 2019). Many other attempts at making reinforcement learning agents for the game of Snake follow in the next section.

3 Related Work

One ambitious attempt at making a reinforcement learning agent for the game of Snake was intended to showcase how drones could automatically navigate around a constantly changing environment (Wu et al. 2019). This kind of autonomy is proposed to have practical applications in the industrial field in terms of automated inspections of oil and gas fields. Another particularly interesting use case for these proposed autonomous drones is in search and rescue scenarios where injured people need to be located and materials (such as first aid, rescue equipment, etc.) need to be dispatched. Going further with this idea, the need for a collection of these drones to collaborate and work together in order to aid multiple people at different locations in a complex disaster scenario could be modeled using a multi-agent Snake game where there are multiple snakes and multiple

target locations. This of course creates an extra layer of complexity in terms of collision avoidance and collaborative pathing. This method used Deep Reinforcement Learning due to the vast state space. Specifically they used multiple Deep Q-Network (DQN) algorithms to achieve their results. An interesting technique that helped speed up the learning process was the use of an Odor effect in which the apple is surrounded by a decreasing reward gradient which mimics the biological mechanism of smell, effectively reducing the blindness of the agent (Wu et al. 2019).

Another attempt at making a reinforcement learning was created by (Patil 2020). Patil's version of a reinforcement learning algorithm for snake was done with a unique set of states and rewards. The set of states given to the agent were 12 Booleans based on where the snake was relative the apple, obstacles and where the snake was heading. The set of rewards were then a collection of four reward states. First, if the snake obtained then apple, get a reward of +10. Second if the snake is heading towards the apple +0. Third, if the snake is heading away from the apple -1. Finally, if the snake hits a wall or it's own body give it a reward of -100.

(Mondal, Nair, and Siddiqi 2020) also explored the uses of games like Snake and Pac-man to help improve the redundancy in learning algorithms for symmetry-transformation equivariant representations of an input environment and its states.

(Almalki and Wocjan 2019) used Deep Q-learning to train a reinforcement agent on the game of snake while a SARSA method is used to control the Snake. A new implementation that was also used was a version of snake where poisoned candies are introduced. These poisoned candies acted as a fake apple and will kill the agent immediately if the agent eats the candy. This introduces a new problem for the snake to learn of detecting which food on the grid is the sweet apple, or the poisonous apple.

(Ma, Tang, and Zhang 2016) try to tackle the challenge of reducing the extremely large state space in Snake to a minimum. This large state space comes from the fact that the location and size of the snake and its tail are constantly changing dynamically after every time step. This makes it difficult to achieve computationally fast learning results given the time complexity. They attempted to solve this by reducing the state space down to a 5-tuple only containing information on the relative position of the snake in relation to the target location. The Reinforcement Learning methods that they implemented were Q-Learning and SARSA which seem to have been effective in some sense. To measure the effectiveness of these methods, they use a non Reinforcement Learning deterministic heuristic algorithm for Snake which is able to approximate an optimal solution algorithm for playing the game of Snake (Ma, Tang, and Zhang 2016). This is then used as a benchmark to measure the success of their Q-Learning and SARSA methods. The SARSA algorithm seemed to outperform the Q-Learning algorithm in this case due to the faster rate of learning and overall effectiveness.

4 Methodology

For our system, we will be using a system of reinforcement learning that incorporates a Deep Q-learning Network or DQN (Roderick, MacGlashan, and Tellex 2017). Then comparing this model to a previous version that utilized only Q-learning. We shall be programming the game of Snake and the machine learning agent in Python. For the neural network part of the DQN we will be using the Tensorflow library, and for display purposes we shall be using the pygame library.

Something to consider in terms of the mathematical complexity of Snake is that it is an NP-Hard problem. (Viglietta 2014) proved that "any game exhibiting location traversal" as well as any game featuring "collectible tokens" is indeed NP-Hard. Snake contains both location traversal and collectible tokens which shows that creating an agent that can effectively play snake and learn an optimal policy isn't exactly a trivial matter.

What follows are more in depth sections on how we plan on implementing the individual distinct portions of the reinforcement learning algorithm.

4.1 State Space

(Patil 2020) uses a state system based on a 12 Boolean string that we will use. These states include:

- If the apple is above the snake
- If the apple is below the snake
- If the apple is to the right of the snake
- If the apple is to the left of the snake
- If a wall or snake body is above snake
- If a wall or snake body below snake
- If a wall or snake body is to the right of the snake
- If a wall or snake body is to the left of the snake
- If the Snake is facing up
- If the Snake is facing down
- If the Snake is facing right
- If the Snake is facing left

These states allow for the agent to have multiple ways to know where it is and what is going on without having to feed in the entire game grid to the agent. These states will need to be changed and fixed by the game engine or environment between each step of the agent. However without telling the agent any info besides these 12 Boolean States existing and being turned on or off, the agent will have to learn the deeper meaning behind these states and what is the correct action for it to take.

Another thing to note is that configuring the state space with this method will heavily reduce the size of the state space for this reinforcement learning agent from around having the entire grid space of 10 by 10 spaces with each grid having 3 different values possible. The environment is explained in more detail in Section 4.4 Environment. This means without space reduction the total state space is equivalent to 3^{100}. However with reducing the state space to this

Figure 1: States as shown in during the demo of the optimal path. The Green squares indicate a 1, or true, while the Black Squares indicate a 0, of false. In this example, the apple is down-right from the snake which is facing to the right with a wall to it's left.

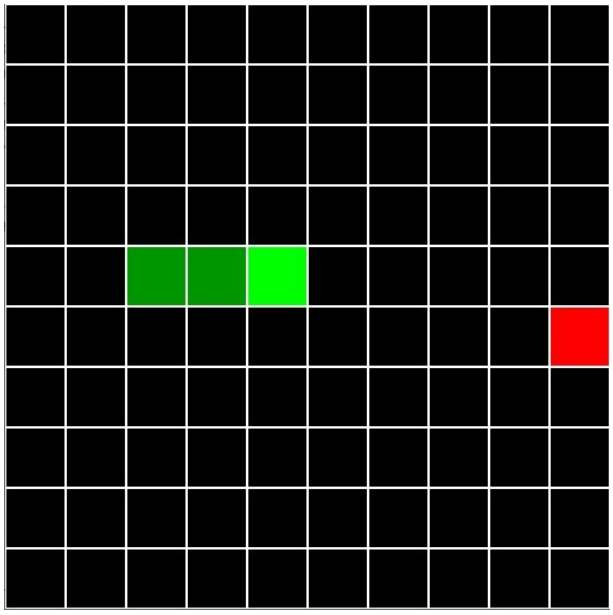

Figure 2: Grid Environment as shown during the demo of the learning agent. Which showcases the states shown from Figure 1. The apple is shown in red, while the snake is shown in green. The snake's head is colored bright green while the tail is shown in dark green.

12 Boolean logic, it will reduce the state space to have a size of 2^{12} or 4096 possible states.

Figure 1 shows how these states are being shown during the demo of the project for the optimal path solution.

4.2 Reward

To help the snake agent learn what actions have good and bad outcomes, a simple reward function is going to be implemented. We want to include two different reward functions. The first of which will simply give a positive reward of +100 if the snake obtains an apple. Then if the agent hits a wall or hits it's own body the game will end and the agent will be rewarded with a -100 reward.

The second method we are planning on using uses the previous function as a base, but with the added ability of a sense distance to the apple similar to (Patil 2020). This reward function gives similar rewards to the previous reward function, such as +100 if the snake obtains an apple and -100 if the snake dies. However this also function adds an additional reward of +1 if the snake is heading towards the apple, and a reward of -1 if the snake is heading away from the apple.

4.3 Actions

Snake is quite simple to control and therefore the actions are quite simple as well. Snake has only four actions:

- Move the snake up one space on the grid
- Move the snake down one space on the grid
- Move the snake left one space on the grid
- Move the snake right one space on the grid

With these four actions, the agent will be able to move the snake freely towards the fruit and can freely move around the grid.

4.4 Environment

Snake is a 2D grid based game. As such the environment will reflect this. The environment will be the grid, or board of the game in size of 10x10. This way it is easy to generate and create the grid without the need for massive amount of computer power to generate large boards. This grid is showcased in Figure 2.

As mentioned previously Snake is a game in which the goal of the game is to have a "snake" on the board obtain or consume randomly place apples on the board. When the snake eats the apple, the snake will grow in length. If the snake hits a boundary of the grid, or hits its own body, then the game will end. In terms of environment this means that the reward state will be completely random in its location. Meaning that the snake will have to figure out where the fruit is while trying not to enter a game over state.

4.5 Replay Buffer

The replay buffer is utilized in the DQN and stores entire episodes based on the actions moved during an episode of training. The information the replay buffer stores is:

- The current state of the agent
- The action taken by the agent
- The reward from the state-action pair
- The next state following the state-action pair

To ensure the DQN model has enough data to start using the data properly a set of data is generated at the beginning where a random policy is used to generate around 500 episodes of random actions in the replay buffer. This then allows for the creation of "mini-batches" of data from the replay buffer that can be used to fit the DQN to proper Q-values.

4.6 Learning Algorithms

For comparison to the DQN algorithm the Q-learning algorithm is shown in Figure 1 (Watkins and Dayan 1992). This algorithm was the basis for the an older learning agent of snake.

The algorithm shown in Algorithm 2, comes from (Roderick, MacGlashan, and Tellex 2017). This algorithm is the basis for learning with the DQN method that was implemented into this project. At it's core the DQN is merely an artificial neural network created from the Tensorflow library in python. However it is built with the combination of using the Replay Buffer, mentioned in section 4.5, and the environment, mentioned in section 4.4. This combination can be seen in figure 3 that showcases the data flow relationship between the DQN, the replay buffer and the Snake Environment.

The major difference that can be seen when looking at these two algorithms is the fact that the Q-learning stores it's values into a Q-table, where as the DQN predicts Q-values using a Neural Network. This prediction of Q-values allows the DQN algorithm to possibility have more learning capabilities then Q-learning, simply because the Neural Network found within the DQN has the ability to learn more in depth meaning to the states given to it then just a Q-table.

Algorithm 1 Q-Learning Algorithm

Initialize $Q(s, a)$ for each s and a
while Episode Count $\leq N$ **do**
    Initialize S
    while $s \neq terminal$ **do**
        Choose A from S using $Q(s, a)$ value greedily
        Take Action A
        Observe R and S'
        $Q(s, a)$ $Q(s, a) + \alpha[R + \Upsilon * Max_a Q(s', a) - Q(s, a)]$
        $S \leftarrow S'$
    end while
end while

Algorithm 2 DQN Algorithm

Initialize Replay Memory D
Initialize Model Q with random weights
Initialize Target Model Q with random weights
while Episode Count $\leq N$ **do**
    Initialize State S
    while $S \neq Terminal$ **do**
        Choose A from S using $Q(s, a)$ value greedily
        Take Action A
        Observe R and S'
        Store experience in Replay D(S, A, R, S')
        Get Sample Data from Replay D(S_j, A_j, R_j, S'_j)
        Set $Y_i \leftarrow R_j$ if S is $Terminal$
        Otherwise Set $Y_i \leftarrow R_j + \Upsilon * Max_{A_j} TargetQ(S_j, A_j)$
        Target Q(S) $\leftarrow$ Target Q(S) $+Y_i$
        Model Q.fit(S, Target Q(S))
        $S \leftarrow S'$
        Every C Steps Target Q = Model Q
    end while
    Episode Count $\leftarrow +1$
end while

Another thing to note about the DQN Algorithm is that it utilizes two Neural Network models at once, similar to double Q-learning. However in this DQN algorithm the Target model is used as values for the Model to be trained on. The two models will likely diverge however every C steps the Target Model is made equal to the Model.

5 Evaluation

We will be using a score based system to gauge the performance of the current model. Each time the snake obtains or eats the apple it will gain 1 point. If the snake hits its own tail or hits a wall the game will be over and the score will be finalized. The higher the score of the agent, the better performance it would have.

Since we plan on having multiple types of agents, we can use this score function to compare how well the models do compared to one another based on their training, and reward functions.

6 Results

For the evaluation of the model we simply used the score system found within the game Snake. Each time the snake gets the apple, or fruit, a score of one is added to the total score. When comparing the DQN algorithm to the Q-learning algorithm we found that the DQN performed about twice as well in terms of score. On average the Q-learning agent would converge to a policy that receives an average score of 7. The DQN model would average a score of around 15 which is quite an improvement.

6.1 Q-learning Results

So far we have measured the performance of our Q-Learning implementation using several different methods. The graphs of using these measurements can be seen in figures 4 and 5.

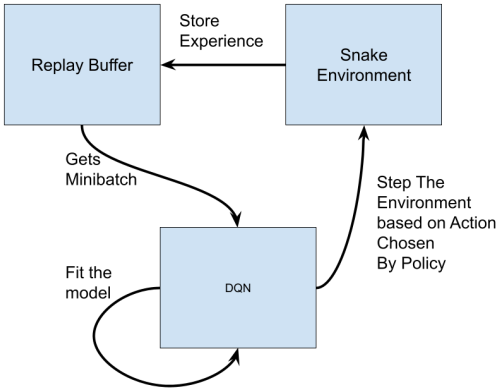

Figure 3: The data flow relationships between the DQN, the replay buffer and the Snake Environment.

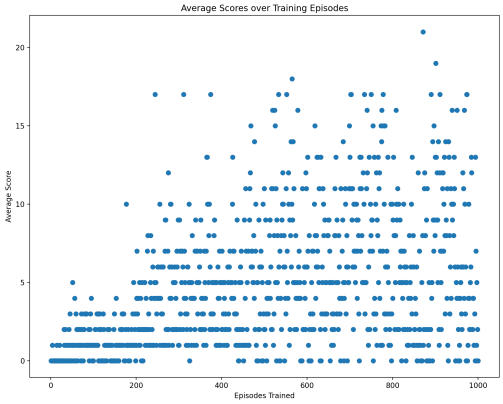

Figure 4: Q-Learning: Scores represent 1 optimal run after each training episode for 1 agent training on 1000 episodes.

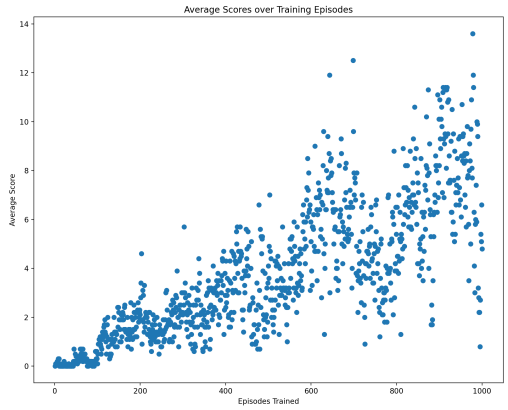

Figure 5: Q-Learning: Scores represent the average of 10 optimal runs after each training episode for 1 agent training on 1000 episodes.

Figure 4 showcases the discrete values of a single agent learning over 1,000 episodes in the Q-learning agent. Very early on the snake is shown to not earn many points, but as the training continued the score varied quite a bit later on, resulting in a max score of 22 around episode 900. However, most of the scores generated by this Q-learning agent were shown to be either low scoring or zero.

Figure 5 showcases 1 Q-learning agent learning over 1,000 episodes and then taking the average score of 10 optimal runs after each episode of training. This results in a graph that better visually represents how the agent learns over a given amount of time. Very early on the agent does not score many points but as training goes on, the average score increases overall.

6.2 DQN Results

The DQN implementation could probably be tweaked in several ways to potentially improve performance. Due to time constraints and long training times, it has been much more difficult to compare the performance effects of tweaking different parameters and neural network designs and architectures. Although, we were able to find a pretty good setup for our DQN given that we were able to double the average performance compared to regular Q-Learning.

Figure 6 showcases the current DQN model using the first reward function training over 10,000 episodes. Over the course of training, every 10 episodes an optimal run was done using the agent. The score from this optimal run was then stored into a list, which was then plotted alongside a trend line in Figure 6. This graph is a great showcase showing how the agent learns over time and is gaining more scores as the episodes of training increase.

Figure 7 showcases the DQN model after training is completed and is ran for 1,000 episodes. This showcases the high variance in the model and environment that stems from the randomness of the snake environment. However this figure is also a great showcase in how the model performs on av-

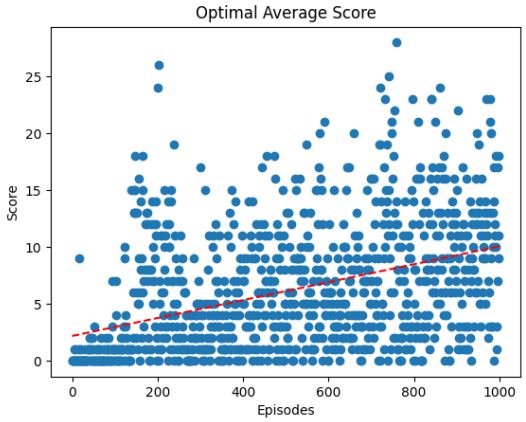

Figure 6: DQN model while training had an optimal run ran every 10 episodes. The scores from these optimal runs were then plotted, along with a trend line showcasing a great upwards tick in score.

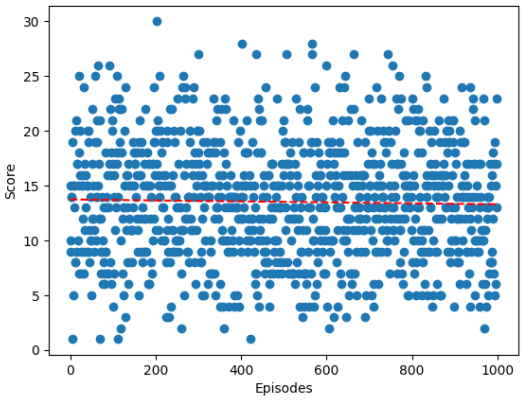

Figure 7: DQN Model running optimally for 1,000 episodes after training for over 10,000 episodes. The average of the scores obtained from this evaluation is around 13.

erage which is around 13 in this case. However most of the time the DQN model, on average, obtains a score of 15.

Figure 8 showcases the DQN agent's learning scores over 5000 episodes. The overall performance gain is easily seen in this graph. It even got a maximum high score of 33 around episode 4000 which is quite impressive given our minimized state space.

6.3 Comments

When comparing the DQN and the Q-learning methods, the average run of the Q-learning method would gain about 7 points, while the DQN would obtain an average of 15 points. This showcased a doubling in performance in terms of score. This means that the DQN model was able to outperform the Q-learning method while still using the limited state space. However, something to note was the massive amount of compute time needed to make the DQN work compared to Q-learning. Training the DQN model on 10,000 episodes took roughly 20 minutes, while training the Q-learning model took only around a few hundred seconds.

7 Updated Timeline

The updated timeline for this project can be seen in Table 1. This table showcases the events for this project, alongside their respective due date and current status. As of writing, the project has finished and concluded.

8 Possible Enhancements

While the current system for this project has been very fruitful, there are a few areas where we see possible enhancements could be placed in the future. One possible enhancement is to reduce the amount of possible actions from 4 actions down to 3 actions. Currently, the action space contains the actions for up, down, left, and right. This does cover the full movement possible for the grid-space but does leave one

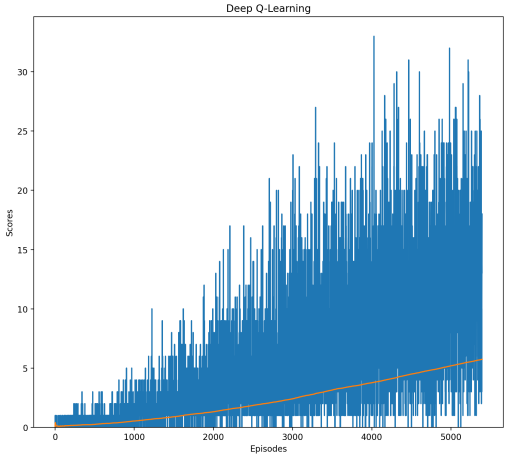

Figure 8: DQN model learning scores while training for over 5000 episodes.

Due Date	Event	Status
9/22/2021	Turn in Proposal	Completed
10/01/2021	Complete Snake Game	Completed
10/15/2021	First Implementation	Completed
10/27/2021	Complete Midterm	Completed
11/05/2021	Implement DQN	Completed
12/03/2021	Finalize Model	Completed
12/13/2021	Complete Final	Completed

Table 1: Timeline of events planned for this project with due dates and status.

major problem where a particular movement (moving backwards into the body of the snake) at any point can lead to the death of the agent. This means that at any one time we can remove one of the actions, resulting in a further reduced action space of just moving to the left, straight ahead, and to the right with respect to the direction that the snake is currently facing. This also means that we can remove one of the 12 booleans in the state space from the wall boolean section since we are no longer checking for there to be a wall behind the snake. Reducing the state space down to 11 boolean states for a state space of 2^{11} or 2048. This means that the total state-action space can be reduced from around 16,000 to 6,000. Which could greatly improve performance.

Another enhancement to this system would be to change the state space into the entire grid. This would greatly increase the state space to around 3^{100} and would require much more tweaking to the network structure to deal with this much larger state space. However this would allow the agent a full view of the grid allowing the agent to avoid it's own tail, which is a problem with the current implementation.

9 Conclusion

In this paper we have shown our accomplishments with creating a reinforcement learning agent for the game of Snake. This agent utilizes a Q-Learning method and a DQN method both with a 12 boolean state space. While the Q-learning method had proved more then useful for generating a working agent, the DQN performed greatly better. In the future possible enhancements could prove to make the DQN even faster or even perform greater then the current implementation.

References

Almalki, A. J., and Wocjan, P. 2019. Exploration of reinforcement learning to play snake game. In *2019 International Conference on Computational Science and Computational Intelligence (CSCI)*, 377–381. IEEE.

Holcomb, S. D.; Porter, W. K.; Ault, S. V.; Mao, G.; and Wang, J. 2018. Overview on deepmind and its alphago zero ai. In *Proceedings of the 2018 international conference on big data and education*, 67–71.

Ma, B.; Tang, M.; and Zhang, J. 2016. Exploration of reinforcement learning to snake.

Mondal, A. K.; Nair, P.; and Siddiqi, K. 2020. Group equivariant deep reinforcement learning. *arXiv preprint arXiv:2007.03437*.

Patil, Y. 2020. Snake game using reinforcement learning. *Academia.edu*.

Roderick, M.; MacGlashan, J.; and Tellex, S. 2017. Implementing the deep q-network. *arXiv preprint arXiv:1711.07478*.

Sutton, R. S., and Barto, A. G. 2020. *Reinforcement Learning: An Introduction*. The MIT Press, 2nd edition.

Viglietta, G. 2014. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems* 54(4):595–621.

Watkins, C. J., and Dayan, P. 1992. Q-learning. *Machine learning* 8(3-4):279–292.

Wu, C.; Ju, B.; Wu, Y.; Lin, X.; Xiong, N.; Xu, G.; Li, H.; and Liang, X. 2019. Uav autonomous target search based on deep reinforcement learning in complex disaster scene. *IEEE Access* 7:117227–117245.

Proposal CS 4080 Fall 2021

Computerized Pac-Man Using Q-Learning

Kaung Htet Myat, Jordan Wren
University of Colorado Colorado Springs

Abstract

Maze games have been popular in the entertainment industry for many decades. Originating from board games, they have developed into video games in many varieties. Pac-Man, a complex maze-solving video game, is considered one of the most successful of its genre. With its popularity, it has attracted a lot of research on reinforcement learning algorithms and their useful applications. In this paper, we will train an agent to play Pac-Man and attempt to attain consistently improving high scores. In order to train an agent, we will be using a case-based Q-Learning Algorithm.

I. Introduction

Developed in 1981 by Toru Iwatani, Pac-Man is a well-known, popular arcade game. The game has been a staple of culture in countries around the world, and people of all different ages enjoy playing it. Pac-Man is considered to be one of the most popular games of all time (Domínguez-Estévez et al., 2017).

Due to its popularity, many variations of it have been created over the years. Different versions may include different map levels, characters or endless game-modes. Like many maze games, it is a two-dimensional game where the player travels through the maze chasing dots for points while trying to avoid moving ghosts, which may kill Pac-Man.

In traditional Pac-Man games, as shown in Figure 1, the player will have three lives. Ghosts will spawn at a certain location and move around in patterns based on their 'aggression level' (indicated by color). If the player hits the ghost, they will lose one life. Meanwhile, the ghosts will respawn back at their starting location. The classic Pac-Man game will have four power pills, and eating these power pills will allow the player to eat the ghosts for points.The power pills come with a time limit so the player has to decide if he or she can get to the ghost location and eat it before the power time ends. In a typical maze game, the player needs to get to a certain point in order to win the game. In Pac-Man, however, winning the game is different as the game only ends when all the dots have been eaten or the player has lost all three of their lives (Gallagher and Ryan, 2003).

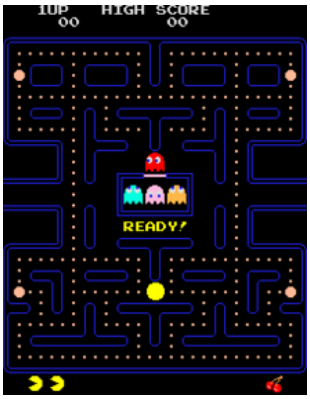

Figure 1: A classic Pac-Man game

Pac-Man has been an interesting game for research in video game AI. One big motivation for researchers is the simplicity of the environment, compared to relatively more complex behaviors in other video games. Additionally, it can be run on most machines which makes it convenient in terms of hardware requirements. (Domínguez-Estévez et al., 2017) Different strategies and algorithms have been tested on Pac-Man. Many research papers have pointed towards Q-Learning algorithms as proper solutions to training an agent to successfully complete the game. The game has drawn attraction to many academics and industries for AI game research and we have found an online course taught by UC Berkeley.

2. Background

Many AI game research has experiments with Pac-Man game. Genetic Algorithms and Reinforcement Learning are two ways to teach an agent to play video games. In Reinforcement Learning, an agent learns to differentiate good and bad actions with a reward system. We will be using Reinforcement Learning on teaching the agent to pay Pac-Man and develop our own reward system. The algorithm we will be using is a case-based Q-learning Algorithm.

2.1 Case-based Q-Learning

Q-Learning algorithms are special in that they do not require a policy table to function. The Q-table storing information

about states, or cases in this scenario, acts as the driving policy behind the agent's decision-making.

The advantage of case-based Q-learning is that cases are independent of states, so they can be applied in many different scenarios based on the current game state, as opposed to finding one 'best' action in every single state.

This algorithm also makes use of a discount rate, denoted by γ. The discount rate is applied to help find the optimal path for the agent; in other words, it encourages faster solutions(Sutton and Barto, 2018)

2.2 Policy Table

The Q table maps states with actions depending on the immediate rewards received in the current episode and in past. The states are assigned a number where the agent will follow the path it believes to be optimal from previous experience. This table is updated on every visit to a new state.

2.3 Updating the Q-Values

α represents the learning rate, ranging anywhere from 0 to 1. This is essentially a confidence score for the updated Q-values. γ represents discount factor, or the rate at which the agent will be penalized as more steps are taken towards the end of an episode.

$$Q(S,A) = Q(S,A+\alpha[R(S,A)+\gamma max_{a'}Q(S',A')-Q(S,A)] \quad (1)$$

When the action for reward is made, the next Action and State denoted by S', A' is observed. This is done until no episodes are left or the level is completed. Over time, the Q table will change, affecting the behavior of the agent to find an optimal path (Kendall and Lucas, 2005).

3. Related Work

There is much related work associated with Pac-Man. Many other maze games are also widely tested with reinforcement learning algorithms, such as Snake (Bom et al., 2013). Many Q-learning algorithms are popular for training agents to play maze games. This includes Approximate Q-learning and Deep Q-learning as some examples (Gnanasekaran et al., 2017). Certain algorithms work better in certain environments, and may not work as well in others. Some scholarly works in the past have managed to create agents that can match your average Pac-Man player, and others have even managed to exceed that boundary. Other unpublished works and student projects have also shown great promise in regards to reinforcement learning in the context of Pac-Man.

4. Methodology

Since the classic Pac-Man is complex and features multiple level designs, the simplest approach would be creating simple version of the game from scratch. It will be created with a simple graphics library, and will be much smaller in scale compared to the original Pac-Man.

4.1 Game Design

To start, the dimensions will be constrained to a 10x15 grid to create 150 total states. There will be just one ghost and it will move slower than Pac-Man, taking just one step for every two that Pac-Man takes. The Ghost movement will be randomized based on their location, as this will create a less aggressive behavior than the original's ghosts. The reward system will be as shown below.

- +1 for every dot consumed
- +20 for every power pill consumed
- +50 for eating a ghost
- +200 for level completion
- -1 for every state traversed without a dot
- -200 for dying

Instead of having Pac-Man make decisions in every state, it will only make decisions at every intersection, which have been labelled as 'decision-points'. Pac-Man cannot stop moving, and there will be 4 possible actions depending on the intersection. The 4 possible actions are "UP", "DOWN", "LEFT", and "RIGHT".

Since it is a maze, not all 4 possible actions will be available at every decision-point. The decision-points are visualized in Figure 2 below.

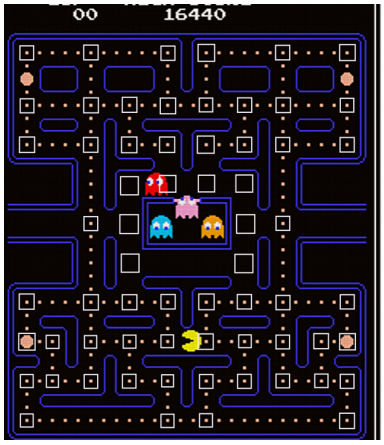

Figure 2: A classic Pac-Man game

4.2 Decision Making

In order to determine the most appropriate case at a decision-point, there will be a similarity function inside of the algorithm. The case found to be most similar to the current game state will be selected to choose an action. The agent will look back at its similar cases and select based on 4 factors. These 4 factors are distance to the closest pill, distance to the closest power pill, distance to the closest non-edible ghost, and distance to the closest edible ghost.

4.3 Implementation

The case-based Q-learning Algorithm will be integrated directly into the environment created. This means that all the

relevant information can be directly queried from the environment, and the most relevant case is then used to choose the highest value action based on previous experience.

There is a learning rate and a discount factor in the algorithm denoted by:

α = learning factor

γ = discount factor

The discount factor will start at 0.98. The learning factor will start at 0.2. The factor scores are random experimental numbers. Based on the learning curve of the agent, they may be adjusted for optimal learning curve.

5. Evaluation

We will first start the discount factor to 0.98. We will measure how many completions it takes for the agent to finally master the game. We will also record the total points that the agent gains in every completion. In order to evaluate the effectiveness of the algorithm, we will need to have a large sample size of different maps and data from other institutes that have trained agents in Pac-Man. Gathering a lot of data, and comparison would be very hard and time consuming. Alternatively, we will state that the agent has sufficiently mastered the game once it has gain 90 percent or above of all possible points in every episode, and the observed score growth has notably plateaued. We will then graph the points with the number of completions. Based on the learning curve, we will decide if the agent is learning efficiently or not. If the agent has a bad curve, we will adjust our reward systems and discount factor and decide if the new learning curve is optimal for the agent.

6. Timeline

This is our proposal timeline for the project. It may vary depending on the progress of the project.

- 26 Sept- 2 Oct Creation of Basic Environment
- 3 Oct - 9 Oct Case Base Q-Learning Algorithm
- 10 Oct - 16 Oct Environment adjustments and Midterm Report

7. Conclusion

This paper includes our proposal for training agent to play a simple version of Pac-Man game. Though there are many approaches to reinforcement learning in virtual environments, many scholarly papers seem to agree on Q-learning being both practical and efficient. Our work will advance based on how fast the agent appears to be learning, and the hope is that as the agent learns the environment can be expanded and slowly become more complex.

References

Bom, L., Henken, R., and Wiering, M. (2013). Reinforcement learning to train ms. pac-man using higher-order action-relative inputs. In *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 156–163. IEEE.

Domínguez-Estévez, F., Sánchez-Ruiz, A. A., and Gómez-Martín, P. P. (2017). Training pac-man bots using reinforcement learning and case-based reasoning. In *CoSECivi*, pages 144–156.

Gallagher, M. and Ryan, A. (2003). Learning to play pac-man: an evolutionary, rule-based approach. In *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, volume 4, pages 2462–2469 Vol.4.

Gnanasekaran, A., Faba, J. F., and An, J. (2017). Reinforcement learning in pacman. *See nalso URL http://cs229. stanford. edu/proj2017/final-reports/5241109. pdf*.

Kendall, G. and Lucas, S. (2005). Cig'05.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA.

Midterm Paper CS 4080 Fall 2021

Computerized Pac-Man Using Q-Learning

Kaung Htet Myat, Jordan Wren
University of Colorado Colorado Springs

Abstract

Maze games have been popular in the entertainment industry for many decades. Originating from board games, they have developed into video games in many varieties. Pac-Man, a complex maze-solving video game, is considered one of the most successful of its genre. With its popularity, it has attracted a lot of research on reinforcement learning algorithms and their useful applications. In this paper, we will train an agent to play Pac-Man and attempt to attain consistently improving high scores. In order to train an agent, we will be using a case-based Q-Learning Algorithm.

I. Introduction

Developed in 1981 by Toru Iwatani, Pac-Man is a well-known, popular arcade game. The game has been a staple of culture in countries around the world, and people of all different ages enjoy playing it. Pac-Man is considered to be one of the most popular games of all time (Domínguez-Estévez et al., 2017).

Due to its popularity, many variations of it have been created over the years. Different versions may include different map levels, characters or endless game-modes. Like many maze games, it is a two-dimensional game where the player travels through the maze chasing dots for points while trying to avoid moving ghosts, which may kill Pac-Man.

In traditional Pac-Man games, as shown in Figure 1, the player will have three lives. Ghosts will spawn at a certain location and move around in patterns based on their 'aggression level' (indicated by color). If the player hits the ghost, they will lose one life. Meanwhile, the ghosts will respawn back at their starting location. The classic Pac-Man game will have four power pills, and eating these power pills will allow the player to eat the ghosts for points.The power pills come with a time limit so the player has to decide if he or she can get to the ghost location and eat it before the power time ends. In a typical maze game, the player needs to get to a certain point in order to win the game. In Pac-Man, however, winning the game is different as the game only ends when all the dots have been eaten or the player has lost all three of their lives (Gallagher and Ryan, 2003).

Pac-Man has been an interesting game for research in video game AI. One big motivation for researchers is the simplicity of the environment, compared to relatively more complex behaviors in other video games. Additionally, it can be run on most machines which makes it convenient in terms of hardware requirements. (Domínguez-Estévez et al., 2017) Different strategies and algorithms have been tested on Pac-Man. Many research papers have pointed towards Q-Learning algorithms as proper solutions to training an agent to successfully complete the game. The game has drawn attraction to many academics and industries for AI game research and we have found an online course taught by UC Berkeley.

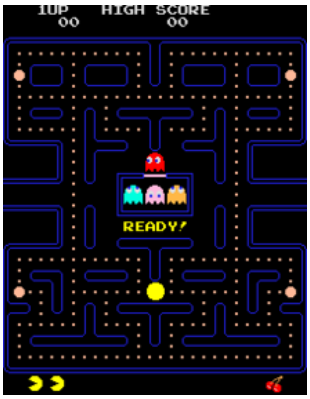

Figure 1: A classic Pac-Man game

2. Background

Many AI game research has experiments with Pac-Man game. Genetic Algorithms and Reinforcement Learning are two ways to teach an agent to play video games. In Reinforcement Learning, an agent learns to differentiate good and bad actions with a reward system. We will be using Reinforcement Learning on teaching the agent to pay Pac-Man and develop our own reward system. The algorithm we will be using is a case-based Q-learning Algorithm.

2.1 Case-based Q-Learning

Q-Learning algorithms are special in that they do not require a policy table to function. The Q-table storing information about states, or cases in this scenario, acts as the driving policy behind the agent's decision-making.

The advantage of case-based Q-learning is that cases are independent of states, so they can be applied in many different scenarios based on the current game state, as opposed to finding one 'best' action in every single state.

This algorithm also makes use of a discount rate, denoted by γ. The discount rate is applied to help find the optimal path for the agent; in other words, it encourages faster solutions(Sutton and Barto, 2018)

2.2 Policy Table

The Q table maps states with actions depending on the immediate rewards received in the current episode and in past. The states are assigned a number where the agent will follow the path it believes to be optimal from previous experience. This table is updated on every visit to a new state.

2.3 Updating the Q-Values

α represents the learning rate, ranging anywhere from 0 to 1. This is essentially a confidence score for the updated Q-values. γ represents discount factor, or the rate at which the agent will be penalized as more steps are taken towards the end of an episode.

$$Q(C, A) = Q(C, A + \alpha[R(C, A) + \gamma max_{a'} Q(C', A') - Q(C, A)] \quad (1)$$

When the action for reward is made, the next Action and State denoted by C', A' is observed. This is done until no episodes are left or the level is completed. Over time, the Q table will change, affecting the behavior of the agent to find an optimal path (Kendall and Lucas, 2005).

3. Related Work

There is much related work associated with Pac-Man. Many other maze games are also widely tested with reinforcement learning algorithms, such as Snake (Bom et al., 2013). Many Q-learning algorithms are popular for training agents to play maze games. This includes Approximate Q-learning and Deep Q-learning as some examples (Gnanasekaran et al., 2017). Certain algorithms work better in certain environments, and may not work as well in others. Some scholarly works in the past have managed to create agents that can match your average Pac-Man player, and others have even managed to exceed that boundary. Other unpublished works and student projects have also shown great promise in regards to reinforcement learning in the context of Pac-Man.

4. Methodology

Since the classic Pac-Man is complex and features multiple level designs, the simplest approach would be creating simple version of the game from scratch. It will be created with a simple graphics library, and will be much smaller in scale compared to the original Pac-Man.

4.1 Game Design

To start, the dimensions will be constrained to a 10x15 grid to create 150 total states. There will be just one ghost and it will move slower than Pac-Man, taking just one step for every two that Pac-Man takes. The Ghost movement will be randomized based on their location, as this will create a less aggressive behavior than the original's ghosts. The reward system will be as shown below.

- +1 for every dot consumed
- +20 for every power pill consumed
- +50 for eating a ghost
- +200 for level completion
- -1 for every state traversed without a dot
- -200 for dying

Instead of having Pac-Man make decisions in every state, it will only make decisions at every intersection, which have been labelled as 'decision-points'. Pac-Man cannot stop moving, and there will be 4 possible actions depending on the intersection. The 4 possible actions are "UP", "DOWN", "LEFT", and "RIGHT".

Since it is a maze, not all 4 possible actions will be available at every decision-point. The decision-points are visualized in Figure 2 below.

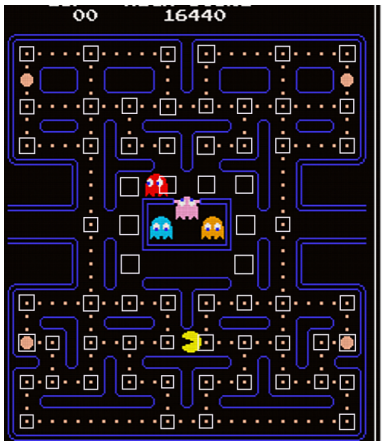

Figure 2: A classic Pac-Man game

4.2 Decision Making

In order to determine the most appropriate case at a decision-point, the algorithm makes use of a similarity function. By creating a case given the agent's current environment, the program can iterate through the case-base and calculate the percentage of similarity between each case, where the highest similarity will be selected and used to make the most appropriate decision. This requires that the cases hold distinct "features" that can be compared with a relatively low number of possible states. These 4 features are distance to the closest pill, distance to the closest power pill, distance to the closest non-edible ghost, and distance to the closest edible ghost.

To reduce the maximum number of possible states, each distance value will be prescribed one of five discrete enumerate values, reducing the maximum possible cases to 5^{16} (4 directions with 4 distinct distance values each). As the

size of this case-base increases, the agent's performance will increase.

4.3 Implementation

The case-based Q-learning Algorithm will be integrated directly into the environment created. This means that all the relevant information can be directly queried from the environment, and the most relevant case is then used to choose the highest value action based on previous experience.

There is a learning rate and a discount factor in the algorithm denoted by:

α = learning factor γ = discount factor

The discount factor is set at 0.98 and the learning factor at 0.1. Initial testing showed these numbers to be good enough to encourage quicker solutions with good confidence in our generated q-values.

4.4 Similarity Function

In order for the agent to make the best possible decision at any given point, it needs to reference its past experience in similar situations and use the outcome to move forward. The similarity function is the main driver behind case-based Q-Learning, and is critical to the entire implementation. By taking the features discussed previously, multiplying the distances by some weight and summing the scores, we can get a percentage value of similarity in our derived cases. We take the max similarity score and use that case to make the decision. If the action is impossible, the second best case is taken. The similarity function used can be seen below.

$$sim(c, c') = 1 - dist(c, c')$$

$$dist(c, c') = \sum_{i=1}^{4} w_i * dist_v(d_i, d_i')$$

$$dist_v(d_i, d_i') = \sum_{i=1}^{4} j \begin{cases} j = 1 \text{ if } d_i = d_i' \\ j = 0 \text{ if } d_i \neq d_i' \end{cases}$$

5. Evaluation

Starting with the discount factor at 0.98, we will measure how many completions it takes for the agent to finally master the game. We will also record the total points that the agent gains in every completion. In order to evaluate the effectiveness of the algorithm, we will need to have a large sample size of different maps and data from other institutes that have trained agents in Pac-Man. Gathering a lot of data, and comparison would be very hard and time consuming. Alternatively, we will state that the agent has sufficiently mastered the game once it has gain 90 percent or above of all possible points in every episode, and the observed score growth has notably plateaued. We will then graph the points with the number of completions. Based on the learning curve, we will decide if the agent is learning efficiently or not. If the agent has a bad curve, we will adjust our reward systems, discount factor, and strategies and decide if the new learning curve is optimal for the agent.

6. Initial Testing and Results

With the initial implementation in the Pac-Man environment, the simplest way to go about testing our case-based Q-Learning algorithm was to perform several trials and compare the results between each of them. At first, the belief was that modifying the discount factor and rewards/punishments would have the greatest impact on the agent's performance, but this proved to be incorrect. The weights used inside of the similarity function actually proved to be the most effective way of altering outcomes. The use of the similarity function also allowed the simulation of different strategies by modifying the weights used in the case calculations. In testing different strategies, we were able to find what our optimal strategy would be moving forward, attempting to replicate human-level play as closely as possible. All trials were completed over 500 episodes so the results were comparable.

6.1 The "Eat-Pills" Strategy

This initial trial involved settings the weight of pills in the similarity function to .9, where both edible and inedible ghosts held a smaller weight of just .05 each. Given that there was only one ghost to compete with inside of the environment, the expectations of this trial were that scores would manage to be fairly high before the ghost managed to kill Pac-Man. The results shown in figure 3 below show some initially low scores with consistent growth, but with large variance between episodes. Scoring high one game did not necessarily reflect in a higher score the next game. This strategy might work okay in the simpler environment, but it likely will not show similar results when the complexity increases.

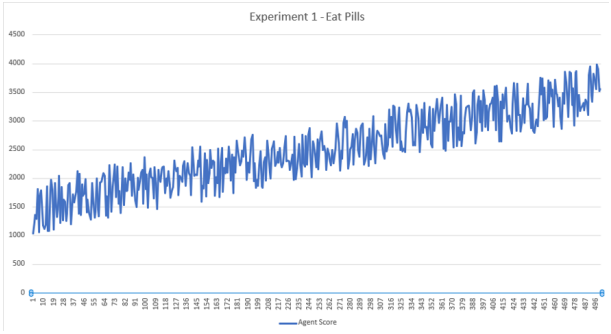

Figure 3: Collecting Pills Strategy

6.2 The "Avoid Ghosts" Strategy

The second trial was expected to be the worst among the three strategies attempted. To simulate this strategy, we set the weight of all pills to 0, inedible ghosts to .95, and edible ghosts to .05. The idea is that when at a decision point, the agent will only take ghosts into consideration, and the

collection of points will come passively as a result of running away from the ghost. The data shows lower scores than found with the first trial, with a much larger variance in score between games. The variance decreased as the agent learned, but after 500 episodes the agent was on-average worse than the first trial. fig:ghost strategy

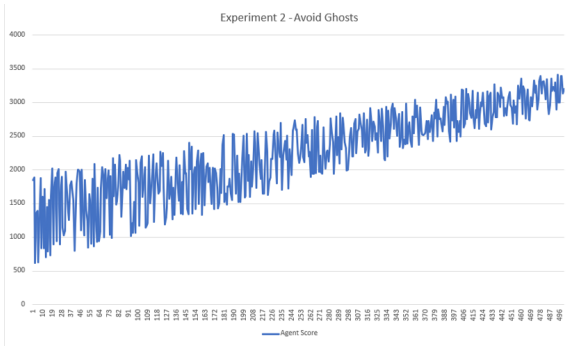

Figure 4: Avoiding Ghost Strategy

6.3 The "Hybrid" Strategy

This strategy came the closest to replicating human-levels of play within the agent. A large part of Pac-Man is avoiding ghosts with the collection of pills coming in as a second priority. To reflect this, the weight of inedible ghosts was set to .7, pills were given a weight of .2, edible ghosts and power pills were given a weight of .05 each. This very quickly showed to be the most promising strategy, with the highest average scores and the lowest variance between episodes.

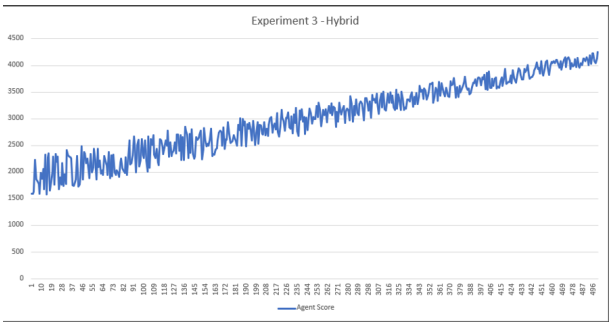

Figure 5: Hybrid Strategy

7. Progress Evaluation

Comparing the three strategies on one graph, the hybrid strategy is clearly the most effective choice to make forward progress on.

With a strategy in mind, the next steps become clearer as we hope to increase the complexity using our case-base established during the hybrid trial. To start, we hope to increase to a full-size Pac-Man grid meaning we will be scaling from a 10x15 to a 15x28. This may immediately reflect higher scores as the agent is still running from just one ghost

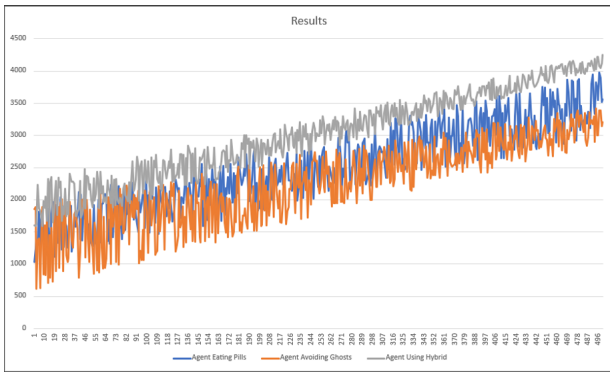

Figure 6: The three strategies combined

in a larger grid, so adding multiple, more aggressive ghosts is the next natural step. As we do this, we hope to begin seeing similar results to the initial testing phase, hopefully reaching a point where the agent will be able to complete multiple levels before dying.

8. Timeline

The finalized timeline moving forward from this mid-term progress report will look something like the following.

- 1 Nov - 7 Nov Building More Complex Environment
- 8 Nov - 15 Nov Begin trying different Q-Learning algorithms
- 16 Nov - 23 Nov Heavy trials to try and maximize agent performance
- 29 Nov - 5 Dec Compare Results and Finalize Report

9. Conclusion

The initial implementation and testing proved useful in providing direction for the final steps of our Q-Learning Pac-Man implementation. All initial tests were done using case-based Q-Learning, and the results are better than expected, but we would like to see if we can improve those results using different Q-Learning methods. Using the hybrid strategy, we are hopeful that we can create an agent that comes close to, or even outperforms the average human given enough episodes and a large-enough case base. Once we have expanded the simple environment into something more reflective of a true Pac-Man grid, the bulk of the time will be spent simply trying to improve results as quickly as possible, with the final results showing scores far beyond that of what we have found thus far.

References

Bom, L., Henken, R., and Wiering, M. (2013). Reinforcement learning to train ms. pac-man using higher-order action-relative inputs. In *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 156–163. IEEE.

Domínguez-Estévez, F., Sánchez-Ruiz, A. A., and Gómez-Martín, P. P. (2017). Training pac-man bots using reinforcement learning and case-based reasoning. In *CoSECivi*, pages 144–156.

Gallagher, M. and Ryan, A. (2003). Learning to play pac-man: an evolutionary, rule-based approach. In *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, volume 4, pages 2462–2469 Vol.4.

Gnanasekaran, A., Faba, J. F., and An, J. (2017). Reinforcement learning in pacman. *See nalso URL http://cs229. stanford. edu/proj2017/final-reports/5241109. pdf*.

Kendall, G. and Lucas, S. (2005). Cig'05.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA.

Final Paper CS 4080 Fall 2021

Computerized Pac-Man Using Q-Learning

Kaung Htet Myat, Jordan Wren
University of Colorado Colorado Springs

Abstract

Maze games have been popular in the entertainment industry for many decades. Originating from board games, they have developed into video games in many varieties. Pac-Man, a complex maze-solving video game, is considered one of the most successful of its genre. With its popularity, it has attracted a lot of research on reinforcement learning algorithms and their useful applications. In this paper, we will train an agent to play Pac-Man and attempt to attain consistently improving high scores. In order to train an agent, we will be using a case-based Q-Learning Algorithm.

I. Introduction

Developed in 1981 by Toru Iwatani, Pac-Man is a well-known, popular arcade game. The game has been a staple of culture in countries around the world, and people of all different ages enjoy playing it. Pac-Man is considered to be one of the most popular games of all time (Domínguez-Estévez et al., 2017).

Due to its popularity, many variations of it have been created over the years. Different versions may include different map levels, characters or endless game-modes. Like many maze games, it is a two-dimensional game where the player travels through the maze chasing dots for points while trying to avoid moving ghosts, which may kill Pac-Man.

In traditional Pac-Man games, as shown in Figure 1, the player will have three lives. Ghosts will spawn at a certain location and move around in patterns based on their 'aggression level' (indicated by color). If the player hits the ghost, they will lose one life. Meanwhile, the ghosts will respawn back at their starting location. The classic Pac-Man game will have four power pills, and eating these power pills will allow the player to eat the ghosts for points.The power pills come with a time limit so the player has to decide if he or she can get to the ghost location and eat it before the power time ends. In a typical maze game, the player needs to get to a certain point in order to win the game. In Pac-Man, however, winning the game is different as the game only ends when all the dots have been eaten or the player has lost all three of their lives (Gallagher and Ryan, 2003).

Pac-Man has been an interesting game for research in video game AI. One big motivation for researchers is the simplicity of the environment, compared to relatively more

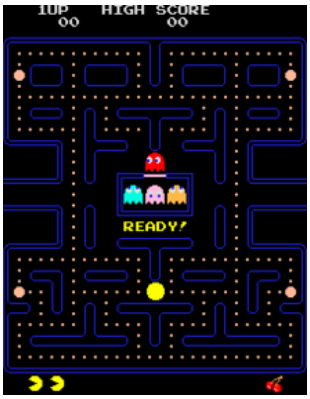

Figure 1: A classic Pac-Man game

complex behaviors in other video games. Additionally, it can be run on most machines which makes it convenient in terms of hardware requirements. (Domínguez-Estévez et al., 2017) Different strategies and algorithms have been tested on Pac-Man. Many research papers have pointed towards Q-Learning algorithms as proper solutions to training an agent to successfully complete the game. The game has drawn attraction to many academics and industries for AI game research and we have found an online course taught by UC Berkeley.

2. Background

Many AI game research has experiments with Pac-Man game. Genetic Algorithms and Reinforcement Learning are two ways to teach an agent to play video games. In Reinforcement Learning, an agent learns to differentiate good and bad actions with a reward system. We will be using Reinforcement Learning on teaching the agent to pay Pac-Man and develop our own reward system. The algorithm we will be using is a case-based Q-learning Algorithm. We will also be using SARSA Learning to compare the efficiency of case-base Q-Learning Algorithm.

2.1 Case-based Q-Learning

Q-Learning algorithms are special in that they do not require a policy table to function. The Q-table storing information

about states, or cases in this scenario, acts as the driving policy behind the agent's decision-making.

The advantage of case-based Q-learning is that cases are independent of states, so they can be applied in many different scenarios based on the current game state, as opposed to finding one 'best' action in every single state.

This algorithm also makes use of a discount rate, denoted by γ. The discount rate is applied to help find the optimal path for the agent; in other words, it encourages faster solutions(Sutton and Barto, 2018)

2.2 Policy Table

The Q table maps states with actions depending on the immediate rewards received in the current episode and in past. The states are assigned a number where the agent will follow the path it believes to be optimal from previous experience. This table is updated on every visit to a new state.

2.3 Updating the Q-Values

α represents the learning rate, ranging anywhere from 0 to 1. This is essentially a confidence score for the updated Q-values. γ represents discount factor, or the rate at which the agent will be penalized as more steps are taken towards the end of an episode.

$$Q(C,A) = Q(C,A+\alpha[R(C,A)+\gamma max_{a'} Q(C',A') - Q(C,A)] \quad (1)$$

When the action for reward is made, the next Action and State denoted by C', A' is observed. This is done until no episodes are left or the level is completed. Over time, the Q table will change, affecting the behavior of the agent to find an optimal path (Kendall and Lucas, 2005).

2.4 SARSA Learning

SARSA Learning is another type of temporal difference learning. SARSA Learning stands for State-Action-Reward-State-Action learning. SARSA Learning has similar functionality compare to Q-Learning. However, the main difference that separates SARSA Learning from Q-Learning is that after getting an immediate reward, the agent in SARSA learning would carry out an additional action and evaluate the first action along with the second reward.(Sewak, 2019) The equation for updating Q-values in SARSA learning is shown below:

$$Q(C,A) = Q(C,A+\alpha[R(C,A)+(C',A'+1) - Q(C,A)] \quad (2)$$

3. Related Work

There is much related work associated with Pac-Man. Many other maze games are also widely tested with reinforcement learning algorithms, such as Snake (Bom et al., 2013). Many Q-learning algorithms are popular for training agents to play maze games. This includes Approximate Q-learning and Deep Q-learning as some examples (Gnanasekaran et al., 2017). Certain algorithms work better in certain environments, and may not work as well in others. Some scholarly works in the past have managed to create agents that can match your average Pac-Man player, and others have even managed to exceed that boundary. One notable work that uses Q-Learning to train an agent to play Pac-Man is done by Anestis Fachantidis, Matthew E. Taylor and Ioannis Vlahavas. Their work include using transfer Q-Learning between two agents to evaluate how efficient an agent can train another agent to play Pac-Man. Their results shows that the student agent can learn significantly better with a higher discount factor. (Fachantidis et al., 2019) Other unpublished works and student projects have also shown great promise in regards to reinforcement learning in the context of Pac-Man.

4. Methodology

Since the classic Pac-Man is complex and features multiple level designs, the simplest approach would be creating simple version of the game from scratch. It will be created with a simple graphics library, and will be much smaller in scale compared to the original Pac-Man.

4.1 Game Design

To start, the dimensions will be constrained to a 10x15 grid to create 150 total states. There will be just one ghost and it will move slower than Pac-Man, taking just one step for every two that Pac-Man takes. The Ghost movement will be randomized based on their location, as this will create a less aggressive behavior than the original's ghosts. The reward system will be as shown below.

- +1 for every dot consumed
- +20 for every power pill consumed
- +50 for eating a ghost
- +200 for level completion
- -1 for every state traversed without a dot
- -200 for dying

Instead of having Pac-Man make decisions in every state, it will only make decisions at every intersection, which have been labelled as 'decision-points'. Pac-Man cannot stop moving, and there will be 4 possible actions depending on the intersection. The 4 possible actions are "UP", "DOWN", "LEFT", and "RIGHT".

Since it is a maze, not all 4 possible actions will be available at every decision-point. The decision-points are visualized in Figure 2 below.

4.2 Decision Making

In order to determine the most appropriate case at a decision-point, the algorithm makes use of a similarity function. By creating a case given the agent's current environment, the program can iterate through the case-base and calculate the percentage of similarity between each case, where the highest similarity will be selected and used to make the most appropriate decision. This requires that the cases hold distinct "features" that can be compared with a relatively low number of possible states. These 4 features are distance to the closest pill, distance to the closest power pill, distance to the closest non-edible ghost, and distance to the closest edible ghost.

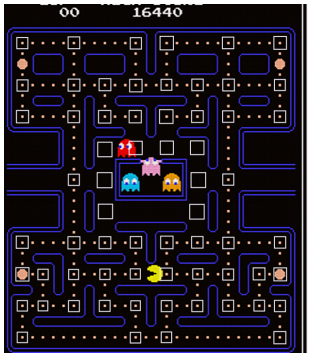

Figure 2: A classic Pac-Man game

To reduce the maximum number of possible states, each distance value will be prescribed one of five discrete enumerate values, reducing the maximum possible cases to 5^{16} (4 directions with 4 distinct distance values each). As the size of this case-base increases, the agent's performance will increase.

4.3 Q-Learning Implementation

The case-based Q-learning Algorithm will be integrated directly into the environment created. This means that all the relevant information can be directly queried from the environment, and the most relevant case is then used to choose the highest value action based on previous experience.

There is a learning rate and a discount factor in the algorithm denoted by:

α = learning factor γ = discount factor

The discount factor is set at 0.98 and the learning factor at 0.1. Initial testing showed these numbers to be good enough to encourage quicker solutions with good confidence in our generated q-values.

4.3 SARSA learning Implementation

The case-based Q-learning Algorithm will be integrated directly into the environment created. This means that all the relevant information can be directly queried from the environment, and the most relevant case is then used to choose the highest value action based on previous experience.

There is a learning rate and a discount factor in the algorithm denoted by:

α = learning factor γ = discount factor

The discount factor is set at 0.98 and the learning factor at 0.1. Initial testing showed these numbers to be good enough to encourage quicker solutions with good confidence in our generated q-values.

4.4 Similarity Function

In order for the agent to make the best possible decision at any given point, it needs to reference its past experience in similar situations and use the outcome to move forward. The similarity function is the main driver behind case-based Q-Learning, and is critical to the entire implementation. By taking the features discussed previously, multiplying the distances by some weight and summing the scores, we can get a percentage value of similarity in our derived cases. We take the max similarity score and use that case to make the decision. If the action is impossible, the second best case is taken. The similarity function used can be seen below.

$$sim(c, c') = 1 - dist(c, c')$$

$$dist(c, c') = \sum_{i=1}^{4} w_i * dist_v(d_i, d'_i)$$

$$dist_v(d_i, d'_i) = \sum_{i=1}^{4} j \begin{cases} j = 1 \text{ if } d_i = d'_i \\ j = 0 \text{ if } d_i \neq d'_i \end{cases}$$

5. Evaluation

Starting with the discount factor at 0.98, we will measure how many completions it takes for the agent to finally master the game. We will also record the total points that the agent gains in every completion. In order to evaluate the effectiveness of the algorithm, we will need to have a large sample size of different maps and data from other institutes that have trained agents in Pac-Man. Gathering a lot of data, and comparison would be very hard and time consuming. Alternatively, we will state that the agent has sufficiently mastered the game once it has gain 90 percent or above of all possible points in every episode, and the observed score growth has notably plateaued. We will then graph the points with the number of completions. Based on the learning curve, we will decide if the agent is learning efficiently or not. If the agent has a bad curve, we will adjust our reward systems, discount factor, and strategies and decide if the new learning curve is optimal for the agent.

6. Initial Testing and Results

With the initial implementation in the Pac-Man environment, the simplest way to go about testing our case-based Q-Learning algorithm was to perform several trials and compare the results between each of them. At first, the belief was that modifying the discount factor and rewards/punishments would have the greatest impact on the agent's performance, but this proved to be incorrect. The weights used inside of the similarity function actually proved to be the most effective way of altering outcomes. The use of the similarity function also allowed the simulation of different strategies by modifying the weights used in the case calculations. In testing different strategies, we were able to find what our optimal strategy would be moving forward, attempting to replicate human-level play as closely as possible. All trials were completed over 500 episodes so the results were comparable. After finding the optimal strategy of playing the game using Case-Based Q Learning Algorithm, we implemented SARSA Learning on another agent for comparison. By using the same environment, we train an agent 500 episodes using SARSA Learning. The results of two agents are compared to find out the optimal reinforcement learning algorithm for playing Pac-Man.

6.1 The "Eat-Pills" Strategy

This initial trial involved settings the weight of pills in the similarity function to .9, where both edible and inedible ghosts held a smaller weight of just .05 each. Given that there was only one ghost to compete with inside of the environment, the expectations of this trial were that scores would manage to be fairly high before the ghost managed to kill Pac-Man. The results shown in figure 3 below show some initially low scores with consistent growth, but with large variance between episodes. Scoring high one game did not necessarily reflect in a higher score the next game. This strategy might work okay in the simpler environment, but it likely will not show similar results when the complexity increases.

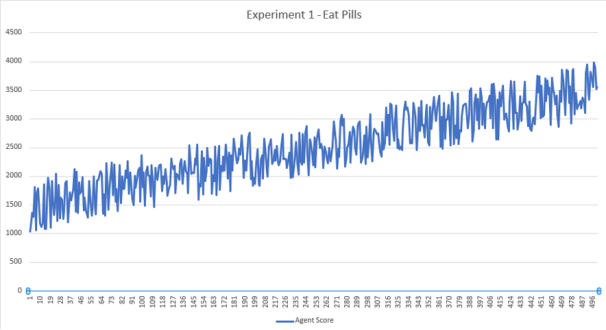

Figure 3: Collecting Pills Strategy

6.2 The "Avoid Ghosts" Strategy

The second trial was expected to be the worst among the three strategies attempted. To simulate this strategy, we set the weight of all pills to 0, inedible ghosts to .95, and edible ghosts to .05. The idea is that when at a decision point, the agent will only take ghosts into consideration, and the collection of points will come passively as a result of running away from the ghost. The data shows lower scores than found with the first trial, with a much larger variance in score between games. The variance decreased as the agent learned, but after 500 episodes the agent was on-average worse than the first trial. fig:ghost strategy

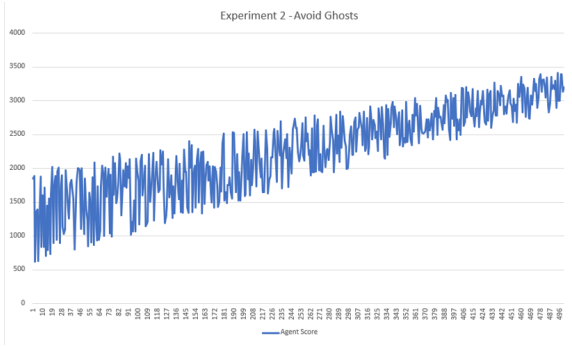

Figure 4: Avoiding Ghost Strategy

6.3 The "Hybrid" Strategy

This strategy came the closest to replicating human-levels of play within the agent. A large part of Pac-Man is avoiding ghosts with the collection of pills coming in as a second priority. To reflect this, the weight of inedible ghosts was set to .7, pills were given a weight of .2, edible ghosts and power pills were given a weight of .05 each. This very quickly showed to be the most promising strategy, with the highest average scores and the lowest variance between episodes.

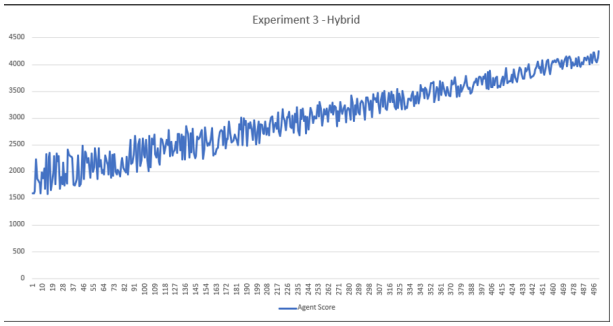

Figure 5: Hybrid Strategy

7. Progress Evaluation

After 500 episodes of training for all three Case-Based Q Learning Algorithms strategies, the results are compared to find out the best learning algorithm and appropriate strategy. Ultimately the metric we are comparing them by is highest average score in the same time-span to determine which will be best to use when making our environment more complex.

7.1 Evaluation of the three Case-Based Q Learning Algorithm

Comparing the three Q-Learning strategies on one graph, the hybrid strategy is clearly the most effective choice to make forward progress on.

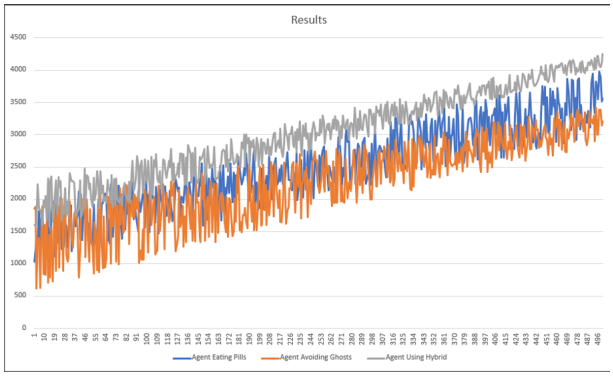

Figure 6: The three strategies combined

With a strategy in mind, the next steps become clearer as we hope to increase the complexity using our case-base

established during the hybrid trial. To start, we hope to increase to a full-size Pac-Man grid meaning we will be scaling from a 10x15 to a 28x31. This may immediately reflect higher scores as the agent is still running from just one ghost in a larger grid, so adding multiple, more aggressive ghosts is the next natural step. As we do this, we hope to begin seeing similar results to the initial testing phase, hopefully reaching a point where the agent will be able to complete multiple levels before dying.

7.2 Changing the Pac-Man Environment

The primary purpose behind using the case-based Q-Learning in a more complex environment is because most reference papers used when working on this project tend to prototype a proof-of-concept in which the environment is extremely simple using some form of Q-Learning. From there they tend to make a leap and begin using neural networks on full-scale versions of Pac-Man in order to test their select algorithm. We wanted to show that a neural network is not necessary for sampling the environment, and that the features stored by the cases are more than sufficient to derive high-level information about the game state.

Simply putting our agent inside of a bigger environment with the randomly acting ghost was going to prove too easy for the agent, thus we needed to populate the maze with a few more ghosts. Attempting to replicate ghost behavior as close as possible to an actual Pac-Man implementation, we introduced "aggression levels" which essentially indicate how likely a ghost is to select the shortest path to the agent at any given point. A ghost with an aggression level of 1.0 will always be seeking out the agent, whereas a ghost with an aggression of 0.1 will only decide to move towards the agent a tenth of the time.

With both aspects of the environment changed the agent is ready to test inside of the more complex environment to compare to what our initial findings told us.

8. New Findings

Our initial guess that the agent's learning would be slowed by the new environment turned out to be completely wrong. As evidenced by the figure below, the results from the trivial and complex environment are extremely comparable, with the agent managing to peak out at about 4,500 score in the first 500 episodes. The only real change between the two environments is the potential for a higher score, with the complex environment allowing for a maximum of about 14,000 if played perfectly.

This was widely unexpected by us, as it seemed despite there being a higher probability to die for the agent, it managed to adapt quick enough to begin experiencing a clear growth in average score over the 500 episodes. This aligns with findings in other papers such as those in the paper by Domınguez-Estevez et al.

The complexity of the environment is less important than good environment sampling like what has been done with our case-base.

The next step in our work would be to find some other Q-Learning algorithm to compare metrics with. We elected

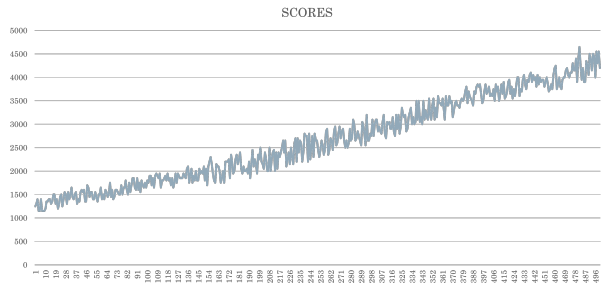

Figure 7: Agent's score over first 500 episodes in complex environment

to attempt a SARSA algorithm integrated directly into the environment to try and replicate the results we have already found.

8.1 Implementing SARSA

Unfortunately, mimicking the results created by our case-based Q-Learning algorithm would prove fairly difficult. For this implementation, we kept our decision-points and instead stored state-action pairs, where each decision-point is its own state; a typical Q-Learning implementation. What we forgot to account for was the importance of the features stored inside of cases. With no way to represent features like ghosts inside of the SARSA algorithm without an implementation of a neural network, the agent showed very little growth across the 500 episodes, barely managing to out-compete a completely random agent. As depicted in the figure below, the agent started scoring around the 800 point mark and after 500 episodes is only able to hit a score of about 1300, a mere 500 point improvement.

We were unable to find a proper Q-Learning solution to Pac-Man outside of the case-based algorithm that did not implement deep learning or a neural network of some sort. The need for a neural net in this circumstance became apparent after we saw these results, as its ability to take samples from the screen and convert the inputs through several layers was important for the agent to understand aspects of a dynamic environment.

Regardless, despite our inability to implement SARSA without a neural network, extensive work done into case-based Q-Learning has shown that on average it manages to outperform a neural network in certain situations like ours. The case-base effectively supplements the neural network when it converts feature distance into our 5 discrete values.

A. Q-learning (SARSA update)

SARSA is an algorithm for learning a Markov decision process policy, where the Q values are updated according to the following rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \cdot (r + \gamma \max_{a'} Q(s',a') - Q(s,a)) \quad (1)$$

Figure 8: Basic SARSA Algorithm

8.2 Evaluation between Case-Based Q Learning Learning Algorithm and SARSA Learning

The data in Figure 6 shows that SARSA Learning fails to train the agent in Pac-Man game. Therefore, we have found Case-Based Q Learning to be a better learning algorithm. A basic SARSA implementation simply does not have enough information regarding pills, power-pills, ghosts, and edible ghosts to create good and reliable quality values. Therefore, the location of the agent in the game is only relevant to the similarity of prior experiences, and not on the location of the map. When the agent only learns to choose an action based on the location of the map, it often hits the random moving ghost and thus ending the game with low points.

9. Finalized Project Timeline

The following is a timeline of recorded activities and progress involved in this project.

- 12 Sept - 19 Sept Research scholar articles relating to Reinforcement Learning in Pac-Man
- 20 Sept - 30 Sept Proposal Presentation and Report
- 1 Oct - 6 Oct Research additional resources
- 7 Oct - 15 Oct Building an Environment and Q-Learning algorithm
- 16 Oct - 23 Oct Testing different strategies in the agent
- 23 Oct - 31 Oct Mid-Term Presentations and report
- 1 Nov - 7 Nov Building More Complex Environment
- 8 Nov - 15 Nov Implementing SARSA Learning algorithm
- 16 Nov - 23 Nov Testing SARSA Learning algorithm
- 29 Nov - 5 Dec Compare Results and Finalize Report

10. Potential Improvements

If we were to re-visit aspects of this project, I believe we would change our hypothesis around a neural net not being necessary for the growth of an agent's ability. SARSA and other Q-Learning algorithms which do not store features are good for static environments in which the state-action pairing will almost always result in the same outcomes. This is not the case in an environment such as Pac-Man. If the agent chooses an action in a state and dies because of proximity to a ghost, that will devalue that action inside of that state, when in reality the issue was independent of the state and more-so relied on the proximity to a ghost in that moment.

Exploring the possibilities of deep learning in the same environment might be interesting to visit as well, imitating human thinking to see if the agent can improve in the same manner or better. Deep Learning also has strong evidence towards being an optimal learning policy for training agents to play games especially those involving a huge number of states. The Department of Computer Science from the University of Texas in Austin trained an agent to play Pac-Man using neural networks. The project results positively and showed that neural networks are a strong learning method for an agent to play a complex games involving a large number of states. (Schrum and Miikkulainen, 2014)

11. Conclusion

The agent's learning rate was better than anticipated for the Case-based Q-Learning in the more complex environment. After many episodes of training the agent using different game strategies, we find out that the hybrid strategy was the best. Using the hybrid strategy, we are hopeful that we can create an agent that comes close to, or even outperforms the average human given enough episodes and a large-enough case base. After creating a much more complex environment and re-comparing the different strategies, we continued to see consistently higher scores with the hybrid strategy as compared to the other two.

Based on our results it is safe to conclude that the hybrid strategy works better than the two other strategies tested. The efficiency of Q-Learning algorithm is also tested by comparing it with SARSA Learning. SARSA Learning showed no improvement in the agent's behavior over episodes of training. Therefore, Case-based Q Learning provides more significant impact on agent's learning.

Important to note is that our findings support the idea that a case-base can make a viable substitute for a neural network depending on the implementation. As we have shown here, our discrete features inside of cases are a good metric for finding similar experiences and using them to select a best action. Overall, the evidence says a case-based Q-Learning is generally a good solution for solving problems inside of dynamic environments.

References

Bom, L., Henken, R., and Wiering, M. (2013). Reinforcement learning to train ms. pac-man using higher-order action-relative inputs. In *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 156–163. IEEE.

Domínguez-Estévez, F., Sánchez-Ruiz, A. A., and Gómez-Martín, P. P. (2017). Training pac-man bots using reinforcement learning and case-based reasoning. In *CoSECivi*, pages 144–156.

Fachantidis, A., Taylor, M. E., and Vlahavas, I. (2019). Learning to teach reinforcement learning agents. *Machine Learning and Knowledge Extraction*, 1(1):21–42.

Gallagher, M. and Ryan, A. (2003). Learning to play pac-man: an evolutionary, rule-based approach. In *The 2003 Congress on Evolutionary Computation, 2003. CEC '03.*, volume 4, pages 2462–2469 Vol.4.

Gnanasekaran, A., Faba, J. F., and An, J. (2017). Reinforcement learning in pacman. *See nalso URL http://cs229. stanford. edu/proj2017/final-reports/5241109. pdf*.

Kendall, G. and Lucas, S. (2005). Cig'05.

Schrum, J. and Miikkulainen, R. (2014). Evolving multimodal behavior with modular neural networks in ms. pac-man. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO '14, page 325–332, New York, NY, USA. Association for Computing Machinery.

Sewak, M. (2019). *Temporal Difference Learning, SARSA,*

and Q-Learning, pages 51–63. Springer Singapore, Singapore.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA.

Graduate Papers

Proposal CS 5080 Fall 2021

Inverse Reinforcement Learning For The Inverse RNA Folding Problem

Van Hovenga
University of Colorado-Colorado Springs
1420 Austin Bluffs Pkwy
Colorado Springs, CO 80919
vhovenga@uccs.edu

Abstract

Introduction

Ribonucleic acid (RNA) is a molecule that is nearly ubiquitous across all living organisms. RNA is a poylmer that consists of four nucleotides: guanine (G), uracil (U), adenine (A), and cytosine (C). The sequences of these four nucleotides that are present in RNA are responsible for various biological functions such as coding, decoding, regulation and expression of genes (Boyer 2002). The crucial roles RNA play in living organisms has led to an abundance of research into RNA design and synthesis for drug discovery (Burke and Berzal-Herranz 1993), vaccine design (Verbeke et al. 2019), biosensing (Win and Smolke 2008), and many more applications (Delebecque et al. 2011), (Hao et al. 2014), (Dixon et al. 2010).

The structure of RNA is a particularly important in determining its mechanism of action (Mathews and Turner 2006). For this reason, several computational methods for predicting the structure of RNA from its corresponding sequence (Seetin and Mathews 2012). There are two levels of structure prediction: secondary and tertiary. Secondary structure refers to the bonding of individual base pairs within the sequence. Tertiary structure refers to how the sequence folds in three-dimensional (3D) space. See figure 1 for a visual representation of this difference. In many practical applications, the secondary structure of RNA is sufficient to infer function (Shijie et al. 2006). This paper is concerned only with the secondary structure of RNA.

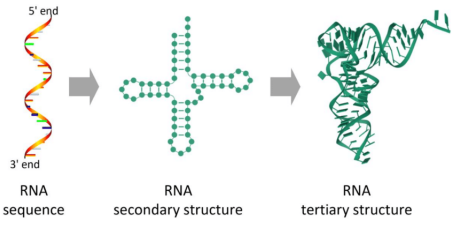

Figure 1: Visual depiction of the difference between the RNA sequence and the associated secondary and tertiary structures (Zhao et al. 2021)

The problem of predicting secondary RNA structure from a given sequence is important, but in many applications of RNA synthesis, it is already known that a specific type of secondary structure may be conducive to a particular function. In this case, it may be useful to ask the inverse question: given a desired secondary structure, how can we construct a sequence of nucleotides that confirms this structure? This is precisely the inverse RNA folding problem (IRFP).

Given a sequence of length n, there exists 4^n possible sequences of nucleotides, making the search space for the IRFP very large, even for small molecules. For this reason, several computational methods have been designed to address this problem (Churkin et al. 2018). In this work, we propose the use of inverse reinforcement learning for the IRFP.

Existing Methods

We now outline several of the computational methods for the IRFP. RNAinverse was the first method developed for the IRFP (Hofacker et al. 1994). This algorithm minimizes the difference between the minimal free energy (MFE) of a given sequence (i.e. the predicted structure) and that of the target structure. This optimization is performed by random mutation of the sequence via an adaptive random walk. RNAiFold (Garcia-Martin, Clote, and Dotu 2013) utilizes constraint programming to optimize one of three possible targets: the minimum free energy (MFE) structure, the total free energy, or the ensemble defect. Since the algorithm is deterministic, RNAiFold ensures optimality in its solutions. NUPACK utilizes partition function algorithms and stochastic kinetics simulations to mimic the paradigms of wet lab RNA design (Dirks et al. 2004). MODENA utilizes a multi-objective genetic algorithm that maximizes the stability of the output structures while minimizing their distance from the target structure. Reinforcement learning has also been utilized as an approach to the IRFP (Eastman et al. 2018).

A starkly different approach to the IRFP comes from the Eterna game (Lee et al. 2014). Eterna is a online open laboratory that formulates the IRFP as a puzzle to be solved by humans. Players are given a sequence and a target structure with the goal of altering the initial sequence such that the

predicted secondary structure matches the target structure. Eterna utilizes the Vienna algorithm for secondary structure prediction (Hofacker 2003). Eterna has over 250,000 registered players with over 225,000 unique puzzles solved.

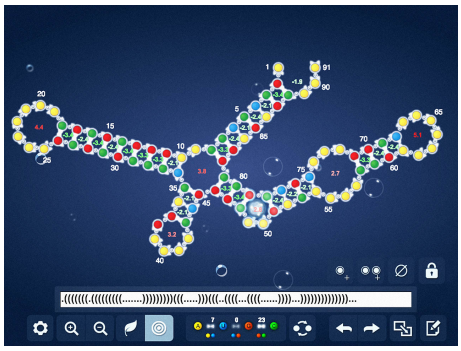

Figure 2: Screenshot of the Eterna game interface.

Eterna has made significant contributions to research of the IRFP. EternaBot is a rules-based algorithm designed by strategies informed by the top Eterna players (Lee et al. 2014). EternaBrain is a convolutional neural network (CNN) trained on 1.8 million player moves that utilizes single action payout of six strategies compiled by human players to predict the optimal sequence for a given target structure (Koodli et al. 2019).

Proposed Method

Prior to Eterna, all computational methods for the IRFP relied on utilizing physical simulation along with some form of optimization. In some cases, such as NUPACK, the method is guided by domain knowledge from wet lab research in RNA synthesis. The advent of Eterna, however, allows for an additional guiding principle for the IRFP; namely, that of human intuition from playing the Eterna game. EternaBot attempts to exploit this human knowledge via a rules-based algorithm guided by the strategies of players. Due to its rules-based nature, however, Eternabot fails to generalize to structures that are not common within the game. EternaBrain initially attempted to reconcile this lack of generalizability by training a CNN to recreate the moves of Eterna players in a supervized manner. Training the CNN alone, however, did not yield results that outperformed random guessing when tested on unseen data. Thus, the authors paired the CNN with a deterministic, single action payout algorithm comprised of six canonical strategies that are standard among Eterna players. This strategy was able to outperform many existing algorithms, suggesting that a combination of domain knowledge from gameplay with machine learning may be promising path for research into the IRFP.

Currently, all approaches that utilize domain knowledge from Eterna do so deterministically, i.e., from a set of predefined strategies. Although some of these methods yield impressive performance, they are restrictive in that they do not allow for modification of player strategy during optimization. Thus, these algorithms are bounded by the capabilities of the human strategies from which they were formed. To circumvent this limitation, we propose to utilize non-deterministic strategy recreation via inverse reinforcement learning (IRL) to address the IRFP. In this way, it will be possible to first train an agent to recreate canonical Eterna strategies, and then allow the agent to explore tangential strategies via direct interaction with the environment to further improve upon the original, human strategies.

Inverse Reinforcement Learning

Imitation learning is a general class of machine learning algorithms whereby an model is trained to imitate expert policies from known data. Assume we have a dataset $\mathcal{D} = \{\tau_1, \tau_2, \cdots, \tau_N\}$ of expert trajectories where $\tau_i = [(s_1, a_1), (s_2, a_2), \cdots, (s_m, a_m)]$ is a sequence of state-action pairs associated with the measured data of some expert. It is assumed that each trajectory τ_i is formed by some expert policy π^*. Informally, the goal of imitation learning is to learn some π_θ from $\mathcal{D}$ such that π^* and π_θ are similar. Behavioral cloning does this by assuming each state action pair is independent and identically distributed and simply minimizes $\mathcal{L}(\pi_\theta(s_i), a_i)$ where $\mathcal{L}$ is some loss function. This is the approach of the CNN of EternaBrain. Behavioral cloning can be difficult to generalize, however, due to distribution shift between the states within the training data and the states from trajectories of π_θ (Codevilla et al. 2019).

Inverse reinforcement learning (IRL) aims to address the lack of generalizability of behavior cloning. Rather than minimizing the loss between the expert policy and the agents policy directly, the reward function of the process itself is approximated from the expert's behavior. A policy is then derived from this estimated reward function and is then compared to the experts policy until the two are sufficiently similar. The potential benefits to using IRL in this setting are three-fold. Firstly, IRL allows us to exploit the human performance on the IRFP from the Eterna data. This aspect is not present in traditonal computational methods that rely on optimizing directly over the physical aspects (such as free energy) involved in the RNA folding. Secondly, IRL allows for an approximation of potentially highly non-linear rewards. The current RL approaches to the IRFP utilize free energy as a reward. In general, the free energy of a given RNA molecule is a very non-linear with respect to variation in the underlying sequence, making it difficult to optimize a policy from it efficiently. This is especially problematic given the size of the state space in the IRFP. Finally, given that we have access to interaction with the environment (via the Vienna engine), it is possible to further refine the learned policy via exploration. This is not possible in the case of rules-based algorithms designed around player policy.

Environment

There are several environments we may use to simulate the equilibrium secondary structure fro RNA sequences. Secondary structure is generally predicted by minimizing the free energy of the sequence. Since our method will be influenced by Eterna play, we will simulate secondary structure formation using the Vienna engine (Hofacker 2003). It is worth noting, however, that the performance of IRFP algorithms is dependent on the dynamics of the folding en-

gine (Koodli et al. 2021). This dependency is due to the fact that most methods optimize the MFE of structures, which is clearly biased by how MFE is estimated by the foldign engine. Given that our proposed method will rely on learned rewards rather than rewards stemming from free energy, we may test our method using several folding engines to see how robust it is with respect to changes in the folding engine.

Data Set

EternaBRain provides a data set of 1.8 million player moves. Each of these moves consists of a nucleotide sequence, a predicted and target structure in both dot-bracket notation and pairmap notation, and the free energy assocatied with both the predicted and target structure. See figure 3 for example data. This data set will be treated as the expert trajectories in our application.

Description	Example	Encoded example
Base Sequence	CCAGAAAAAAAAACUGG	[[0, 0, 0, 1], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 0], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0], [0, 0, 0, 1], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 1, 0]]
Predicted ('Nature Mode') Structure in dot-bracket notation	(((((.........))))) (dot-bracket)	2,2,2,2,1,1,1,1,1,1,1,1,1,1,3,3,3,3
Target Structure in dot-bracket notation	(((((.........))))) (dot-bracket)	2,2,2,2,1,1,1,1,1,1,1,1,3,3,3,3,3
Predicted Structure Energy	-2.2 kcal/mol	-2.2
Target Energy	4.9 kcal/mol	4.9
Predicted Structure in pairmap notation*	(((((.........))))) (pairmap)	16,15,14,13,-1,-1,-1,-1,-1,-1,-1,-1,-1,3,2,1,0
Target Structure in pairmap notation*	(((((.........))))) (pairmap)	16,15,14,13,12,-1,-1,-1,-1,-1,-1,-1,-1,4,3,2,1,0
Locked bases	xooooooooooooooox	2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2

* Additional features used for training on *eternamoves-select* and not *eternamoves-large* data set.

https://doi.org/10.1371/journal.pcbi.1007059.t001

Figure 3: Description and examples of attributes in the proposed training data (Koodli et al. 2019).

Evaluation

Eterna provides two benchmarks for performance of IRFP programs that each consist of 100 structures with known solutions of varying difficulty known as the Eterna100 and Eterna100v2. The two benchmarks are a mix of human created designs and known RNA structures. So far, no algorithm can solve all 100 structures on either data set. Time is also a consideration in the evaluation of the performance of IRFP algorithms due to the large search space. Figure 4 shows the performance of six algorithms on the Eterna100.

Conclusion

We propose the use of inverse reinforcement learning to address the inverse rna folding problem. Our algorithm will be initially trained on 1.8 million moves from human Eterna gameplay and be further refined using traditional RL via environment interaction. This approach is motivated by the fact that all current IRFP methods are either purely computational, or only incorporate human strategy via rules. We hope that our method allows for a policy guided by human strategy to be further optimized via reinforcement learning, thereby stitching the two traditional approaches together. We plan to evaluate our method with several other methods on the Eterna100 and Eterna100v2 benchmark.

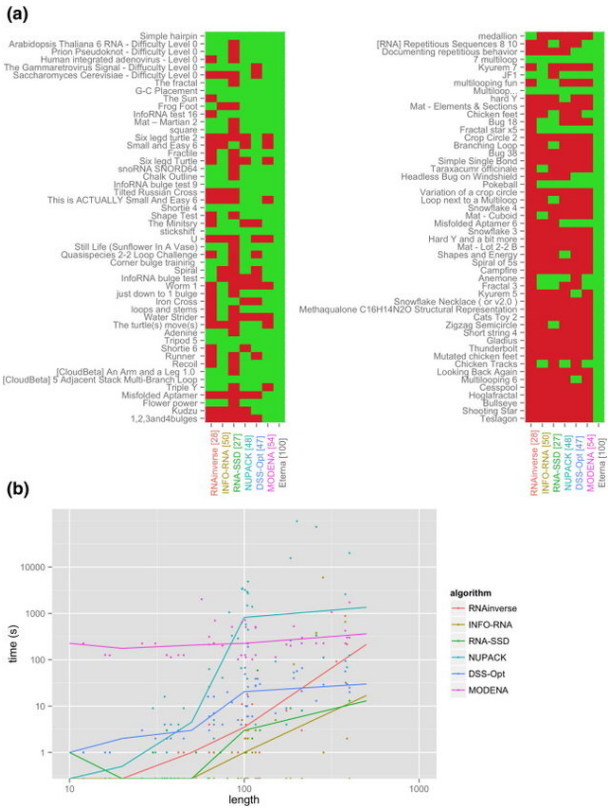

Figure 4: Comparison of performance on the Eterna100. **(a)** shows the solve status for each structure. **(b)** shows the solve time over sequence length.

Timeline

The proposed timeline for this project leading up to the midterm is as follows:

Date	Milestone
9/22-9/29	Data collection and Vienna environment set up.
9/29-10/6	Initial implementation of IRL algorithm.
3/6-3/13	Training and figure collection.
3/13-3/20	Prepare midterm report.

Table 1: Proposed Project Timeline

References

Boyer, R. F. 2002. *Concepts in biochemistry*, volume 139. Brooks/Cole Pacific Grove, CA.

Burke, J. M.; and Berzal-Herranz, A. 1993. In vitro selection and evolution of RNA: applications for catalytic RNA, molecular recognition, and drug discovery. *The FASEB journal* 7(1): 106–112.

Churkin, A.; Retwitzer, M. D.; Reinharz, V.; Ponty, Y.; Waldispühl, J.; and Barash, D. 2018. Design of RNAs: comparing programs for inverse RNA folding. *Briefings in bioinformatics* 19(2): 350–358.

Codevilla, F.; Santana, E.; López, A. M.; and Gaidon, A. 2019. Exploring the limitations of behavior cloning for autonomous driving. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 9329–9338.

Delebecque, C. J.; Lindner, A. B.; Silver, P. A.; and Aldaye, F. A. 2011. Organization of intracellular reactions with rationally designed RNA assemblies. *Science* 333(6041): 470–474.

Dirks, R. M.; Lin, M.; Winfree, E.; and Pierce, N. A. 2004. Paradigms for computational nucleic acid design. *Nucleic acids research* 32(4): 1392–1403.

Dixon, N.; Duncan, J. N.; Geerlings, T.; Dunstan, M. S.; McCarthy, J. E.; Leys, D.; and Micklefield, J. 2010. Reengineering orthogonally selective riboswitches. *Proceedings of the National Academy of Sciences* 107(7): 2830–2835.

Eastman, P.; Shi, J.; Ramsundar, B.; and Pande, V. S. 2018. Solving the RNA design problem with reinforcement learning. *PLoS computational biology* 14(6): e1006176.

Garcia-Martin, J. A.; Clote, P.; and Dotu, I. 2013. RNAiFOLD: a constraint programming algorithm for RNA inverse folding and molecular design. *Journal of bioinformatics and computational biology* 11(02): 1350001.

Hao, C.; Li, X.; Tian, C.; Jiang, W.; Wang, G.; and Mao, C. 2014. Construction of RNA nanocages by re-engineering the packaging RNA of Phi29 bacteriophage. *Nature communications* 5(1): 1–7.

Hofacker, I. L. 2003. Vienna RNA secondary structure server. *Nucleic acids research* 31(13): 3429–3431.

Hofacker, I. L.; Fontana, W.; Stadler, P. F.; Bonhoeffer, L. S.; Tacker, M.; and Schuster, P. 1994. Fast folding and comparison of RNA secondary structures. *Monatshefte für Chemie/Chemical Monthly* 125(2): 167–188.

Koodli, R. V.; Keep, B.; Coppess, K. R.; Portela, F.; participants, E.; and Das, R. 2019. EternaBrain: Automated RNA design through move sets and strategies from an Internet-scale RNA videogame. *PLoS computational biology* 15(6): e1007059.

Koodli, R. V.; Rudolfs, B.; Wayment-Steele, H. K.; and Das, R. 2021. Redesigning the Eterna100 for the Vienna 2 folding engine. *bioRxiv* .

Lee, J.; Kladwang, W.; Lee, M.; Cantu, D.; Azizyan, M.; Kim, H.; Limpaecher, A.; Gaikwad, S.; Yoon, S.; Treuille, A.; et al. 2014. RNA design rules from a massive open laboratory. *Proceedings of the National Academy of Sciences* 111(6): 2122–2127.

Mathews, D. H.; and Turner, D. H. 2006. Prediction of RNA secondary structure by free energy minimization. *Current opinion in structural biology* 16(3): 270–278.

Seetin, M. G.; and Mathews, D. H. 2012. RNA structure prediction: an overview of methods. *Bacterial regulatory RNA* 99–122.

Shijie, C.; Zhijie, T.; Song, C.; and Wenbing, Z. 2006. The statistical mechanics of RNA folding. *Physics* 3: 012.

Verbeke, R.; Lentacker, I.; De Smedt, S. C.; and Dewitte, H. 2019. Three decades of messenger RNA vaccine development. *Nano Today* 28: 100766.

Win, M. N.; and Smolke, C. D. 2008. Higher-order cellular information processing with synthetic RNA devices. *Science* 322(5900): 456–460.

Zhao, Q.; Zhao, Z.; Fan, X.; Yuan, Z.; Mao, Q.; and Yao, Y. 2021. Review of machine learning methods for RNA secondary structure prediction. *PLoS computational biology* 17(8): e1009291.

Midterm CS 5080 Fall 2021

Online-Offline Reinforcement Learning For The Inverse RNA Folding Problem

Van Hovenga
University of Colorado-Colorado Springs
1420 Austin Bluffs Pkwy
Colorado Springs, CO 80919
vhovenga@uccs.edu

Abstract

The secondary structure of ribonucleic acid (RNA), is important in many biological applications. There exists several computational methods that predict the secondary structure of RNA from a given sequence of nucleotides. It is often the case, however, that a desirable secondary structure is known a priori. The task then becomes predicting a sequence that conforms to the desired secondary structure. This is known as the inverse RNA folding problem (IRFP). In this work, we utilize data from humans solving the IRFP and offline-online reinforcement learning to address the IRFP. This particular iteration gives updates of our progress up until the current date on this project.

Introduction

Ribonucleic acid (RNA) is a molecule that is nearly ubiquitous across all living organisms. RNA is a poylmer that consists of four nucleotides: guanine (G), uracil (U), adenine (A), and cytosine (C). The sequences of these four nucleotides that are present in RNA are responsible for various biological functions such as coding, decoding, regulation and expression of genes (Boyer 2002). The crucial roles RNA play in living organisms has led to an abundance of research into RNA design and synthesis for drug discovery (Burke and Berzal-Herranz 1993), vaccine design (Verbeke et al. 2019), biosensing (Win and Smolke 2008), and many more applications (Delebecque et al. 2011), (Hao et al. 2014), (Dixon et al. 2010).

The structure of RNA is a particularly important in determining its mechanism of action (Mathews and Turner 2006). For this reason, several computational methods for predicting the structure of RNA from its corresponding sequence (Seetin and Mathews 2012). There are two levels of structure prediction: secondary and tertiary. Secondary structure refers to the bonding of individual base pairs within the sequence. Tertiary structure refers to how the sequence folds in three-dimensional (3D) space. See figure 1 for a visual representation of this difference. In many practical applications, the secondary structure of RNA is sufficient to infer function (Shijie et al. 2006). This paper is concerned only with the secondary structure of RNA.

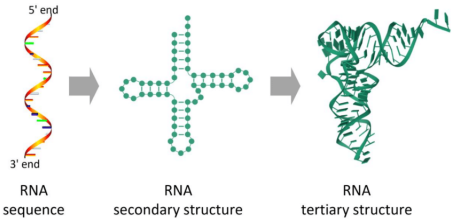

Figure 1: Visual depiction of the difference between the RNA sequence and the associated secondary and tertiary structures (Zhao et al. 2021)

The problem of predicting secondary RNA structure from a given sequence is important, but in many applications of RNA synthesis, it is already known that a specific type of secondary structure may be conducive to a particular function. In this case, it may be useful to ask the inverse question: given a desired secondary structure, how can we construct a sequence of nucleotides that confirms this structure? This is precisely the inverse RNA folding problem (IRFP).

Given a sequence of length n, there exists 4^n possible sequences of nucleotides, making the search space for the IRFP very large, even for small molecules. For this reason, several computational methods have been designed to address this problem (Churkin et al. 2018). In this work, we propose the use of online-offline reinforcement learning for the IRFP.

Existing Methods

We now outline several of the computational methods for the IRFP. RNAinverse was the first method developed for the IRFP (Hofacker et al. 1994). This algorithm minimizes the difference between the minimal free energy (MFE) of a given sequence (i.e. the predicted structure) and that of the target structure. This optimization is performed by random mutation of the sequence via an adaptive random walk. RNAiFold (Garcia-Martin, Clote, and Dotu 2013) utilizes constraint programming to optimize one of three possible targets: the minimum free energy (MFE) structure, the total free energy, or the ensemble defect. Since the algorithm is deterministic, RNAiFold ensures optimality in its solutions. NUPACK utilizes partition function algorithms and stochastic kinetics simulations to mimic the paradigms of wet lab

RNA design (Dirks et al. 2004). MODENA utilizes a multi-objective genetic algorithm that maximizes the stability of the output structures while minimizing their distance from the target structure. Reinforcement learning has also been utilized as an approach to the IRFP (Eastman et al. 2018).

A starkly different approach to the IRFP comes from the Eterna game (Lee et al. 2014). Eterna is a online open laboratory that formulates the IRFP as a puzzle to be solved by humans. Players are given a sequence and a target structure with the goal of altering the initial sequence such that the predicted secondary structure matches the target structure. Eterna utilizes the Vienna algorithm for secondary structure prediction (Hofacker 2003). Eterna has over 250,000 registered players with over 225,000 unique puzzles solved.

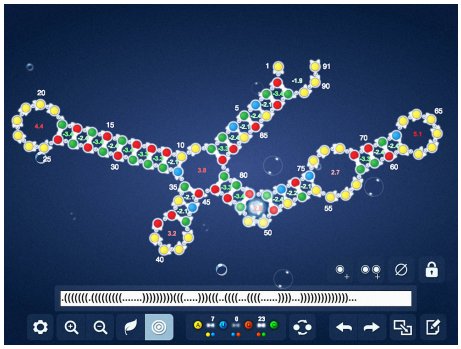

Figure 2: Screenshot of the Eterna game interface.

Eterna has made significant contributions to research of the IRFP. EternaBot is a rules-based algorithm designed by strategies informed by the top Eterna players (Lee et al. 2014). EternaBrain is a convolutional neural network (CNN) trained on 1.8 million player moves that utilizes single action payout of six strategies compiled by human players to predict the optimal sequence for a given target structure (Koodli et al. 2019).

Method

Prior to Eterna, all computational methods for the IRFP relied on utilizing physical simulation along with some form of optimization. In some cases, such as NUPACK, the method is guided by domain knowledge from wet lab research in RNA synthesis. The advent of Eterna, however, allows for an additional guiding principle for the IRFP; namely, that of human intuition from playing the Eterna game. EternaBot attempts to exploit this human knowledge via a rules-based algorithm guided by the strategies of players. Due to its rules-based nature, however, Eternabot fails to generalize to structures that are not common within the game. EternaBrain initially attempted to reconcile this lack of generalizability by training a CNN to recreate the moves of Eterna players in a supervized manner. Training the CNN alone, however, did not yield results that outperformed random guessing when tested on unseen data. Thus, the authors paired the CNN with a deterministic, single action payout algorithm comprised of six canonical strategies that are standard among Eterna players. This strategy was able to outperform many existing algorithms, suggesting that a combination of domain knowledge from gameplay with machine learning may be promising path for research into the IRFP.

Currently, all approaches that utilize domain knowledge from Eterna do so deterministically, i.e., from a set of predefined strategies. Although some of these methods yield impressive performance, they are restrictive in that they do not allow for modification of player strategy during optimization. Thus, these algorithms are bounded by the capabilities of the human strategies from which they were formed. To circumvent this limitation, we propose to utilize non-deterministic strategy recreation via offline-online reinforcement learning to address the IRFP. In this way, it will be possible to first train an agent to recreate canonical Eterna strategies, and then allow the agent to explore tangential strategies via direct interaction with the environment to further improve upon the original, human strategies.

Offline-Online Reinforcement Learning

Offline and online reinforcement learning define two distinct ways of training RL algorithms. Fully online RL refers to algorithms that are trained via direct interaction with an environment. Fully offline RL refers to algorithms that are trained using static data sets of historical interactions with an environment (Levine et al. 2020). Offline and online RL are not mutually exclusive, however. Several algorithms have been developed which allow for a policy initially estimated from static data to be refined via online learning (Nair et al. 2020a).

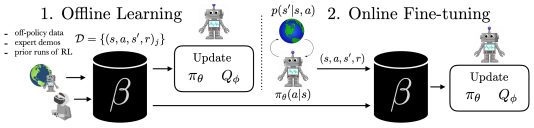

Figure 3: Visual depiction of offline-online RL. A policy is first estimated from expert data. The policy is then fine-tuned via interaction with the environment (Nair et al. 2020a).

One way of combining offline and online reinforcement learning is to treat the interaction data as a labeled data set and utilize it to pre-train a policy in a supervised setting. The pre-trained policy is then deployed to an online setting and is refined via direct interaction with the environment. We refer to this method as *behavior cloning behavior refinement* , or BCBR. The current work of this project has focused on the behavior cloning aspect of BCBR, which we outline below.

Behavior Cloning Architecture

In this section, we first outline the structure of the move data and the learning environment. We then describe the architecture of the policy network and discuss the training procedure. We conclude with the current results of the behavior cloning component of the project.

Data Set

EternaBrain provides a data set of 1.8 million player moves. Each of these moves consists of a nucleotide sequence, a

predicted and target structure in both dot-bracket notation and pairmap notation, and the free energy associatied with both the predicted and target structure. See figure 4 for example data. This data set will be treated as the expert trajectories in our application. From this data, we generate state action pairs associated with each move. In this context, the state is given by a sequence of nucleotides and an action is identified by a location and a base.

Description	Example	Encoded example
Base Sequence	CCAGAAAAAAAAACUGG	[[0, 0, 0, 1], [0, 0, 0, 1], [1, 0, 0, 0], [0, 0, 1, 0], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0], [1, 0, 0, 0], [0, 0, 0, 1], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 1, 0]]
Predicted ('Nature Mode') Structure in dot-bracket notation	(((((.........))))) (dot-bracket)	2,2,2,2,1,1,1,1,1,1,1,1,1,1,3,3,3,3,3
Target Structure in dot-bracket notation	((((((.........)))))) (dot-bracket)	2,2,2,2,2,1,1,1,1,1,1,1,1,1,3,3,3,3,3
Predicted Structure Energy	-2.2 kcal/mol	-2.2
Target Energy	4.9 kcal/mol	4.9
Predicted Structure in pairmap notation*	(((((.........))))) (pairmap)	16,15,14,13,-1,-1,-1,-1,-1,-1,-1,-1,-1,3,2,1,0
Target Structure in pairmap notation*	((((((.........)))))) (pairmap)	16,15,14,13,12,-1,-1,-1,-1,-1,-1,-1,-1,4,3,2,1,0
Locked bases	xooooooooooooooox	2,1,1,1,1,1,1,1,1,1,1,1,1,1,1,2

* Additional features used for training on *eternamoves-select* and not *eternamoves-large* data set.

https://doi.org/10.1371/journal.pcbi.1007059.t001

Figure 4: Description and examples of attributes in the proposed training data (Koodli et al. 2019).

Environment

There are several environments we may use to simulate the equilibrium secondary structure from RNA sequences. Secondary structure is generally predicted by minimizing the free energy of the sequence. Since our method will be influenced by Eterna play, we will simulate secondary structure formation using the Vienna engine (Hofacker 2003). It is worth noting, however, that the performance of IRFP algorithms is dependent on the dynamics of the folding engine (Koodli et al. 2021). This dependency is due to the fact that most methods optimize the MFE of structures, which is clearly biased by how MFE is estimated by the folding engine. Given that our proposed method will not rely on free energy as an objective, we may test our method using several folding engines to see how robust it is with respect to changes in the folding engine.

Graph Policy Network

We utilize a graph neural network architecture for the policy network. The reasons for this decision are two-fold. Firstly, variation in the lengths of each puzzle make it difficult to input the data into a traditional neural network. Secondly, graph neural networks allow us to incorporate the structure of the solution conformation via edges that would not be possible in a simple sequence model. With the graph neural network architecture, we formulate the task of predicting moves as a node classification task as follows.

Graph Structure The inputs to our graph neural network consist of an adjacency matrix defining connections between nodes, a node feature matrix, and a node label matrix. The adjacency matrix is defined via the bonds in the target structure. The ij^{th} entry of this matrix is 1 iff the bases in the i and j position of the sequence are bonded. For example, the adjacency matrix of the structure **((..))** is given by the matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

. The node feature matrix consists of one-hot encoded vectors corresponding to the nucleotides in the given state. The node label matrix consists of one-hot encoded vectors corresponding to five different classes: change to A, change to C, change to G, change to U, and don't change. With this formulation, the task of predicting player moves becomes a node classification problem. We may predict multiple player moves in unison simply by assigning multiple change labels to the graph. In our current implementation, we consider 12 player moves in each graph.

Architecture Our policy network follows a consolidate, aggregate, and predict scheme that is typical in many graph neural networks (ZHOU, JIE, and Publishers 2020). Assume that we have a state sequence, $(\mathbf{x}_0, \cdots, \mathbf{x}_N)$ and that we would like to predict the move on node $\mathbf{x_i}$. We refer to this as the target node. Since each base can bond with a maximum of three other bases, the target node has a maximum of three neighbors. Assuming that the target node is not located on either end of the sequence, there will be a sequence of nodes to the left of the target node $(\mathbf{x}_0, \cdots, \mathbf{x}_{i-1})$, a sequence of nodes to the right of the target node, $(\mathbf{x}_{i+1}, \cdots, \mathbf{x}_N)$, and potentially an additional node bonded to the target node, $\mathbf{x}_j$, depending on the target conformation. We refer to these as the left sequence, the right sequence, and the center sequence. Note that the target node and the center sequence always have length one.

The first step of the network is to consolidate the features of the target node along with the right, left, and center sequences into four separate vectors. We do this by sending each sequence through a five layer long-short-term-memory network (LSTM) (Hochreiter and Schmidhuber 1997). This gives us the left, right, center, and target node representations which we denote by $\widetilde{\mathbf{x}_L}$, $\widetilde{\mathbf{x}_R}$, $\widetilde{\mathbf{x}_C}$, and $\widetilde{\mathbf{x}_T}$ respectively. The second step is to aggregate these representations into a single representation. This is done simply by taking the sum of each representation. We denote this single representation as $\mathbf{x}'_i := \widetilde{\mathbf{x}_L} + \widetilde{\mathbf{x}_R} + \widetilde{\mathbf{x}_C} + \widetilde{\mathbf{x}_T}$. We finally predict the label of the target node by sending $\mathbf{x}'_i$ through a three layer multi-layer perceptron. Figure 5 is a diagram depicting the architecutre of the GNN.

Current Results

Following the processing of the raw move data, we have 150,000 graphs to be used as inputs to our GNN. The objective of the project up until now has been to pre-train our GNN using this data. We trained our GNN using batches of 100 graphs on an 80-20 train-test split of the data. We utilized categorical cross entropy loss as the objective and optimized the model parameters using the Adam optimizer (Kingma and Ba 2014).

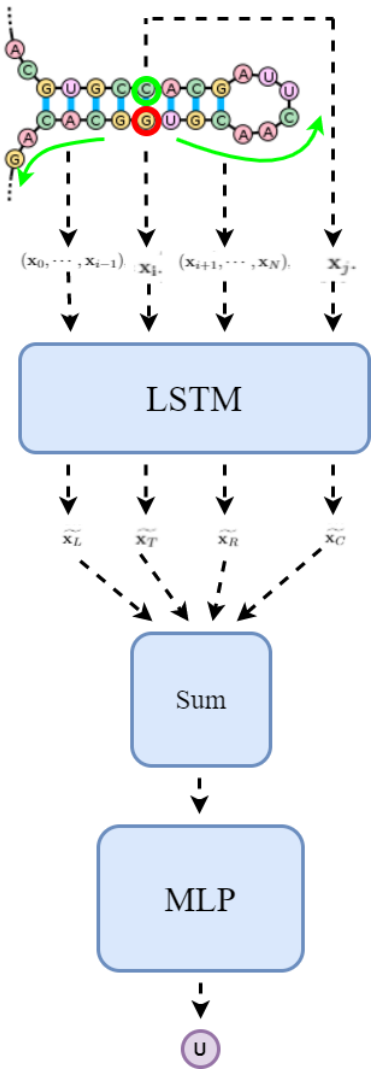

Figure 5: Visual depiction of the architecture of the GNN. In this case, the red outlined base is the target node.

After training for 1,000 epochs, the value of the loss was 15.6543, but the testing accuracy was around 97%. This discrepancy between the loss and the accuracy comes from the fact that the labels of our data are unbalanced. Recall that we only consider 12 player moves in a given graph. This implies that only 12 nodes have labels that are not simply "do nothing". This becomes problematic for large sequences because the majority of the labels correspond to the "do nothing" class. Figure 6 shows a bar chart corresponding to the counts of each label for a subset of the data. Due to the abundance of the "do nothing" class, the model simply results to predicting "do nothing" for every data instance, thereby leading to the high accuracy with a high loss.

A possible solution would be to weight the class probabilities in the loss based upon their frequency in the training data. We found that this does not work. In fact, the same problem persists even if the "do nothing" class is ignored all

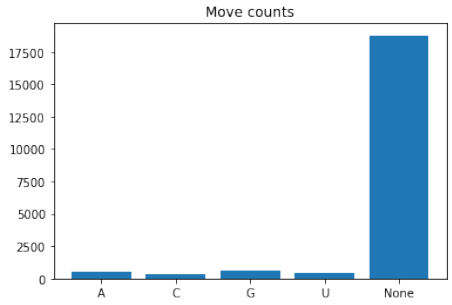

Figure 6: Counts for each class in a subset of the move data.

together. This leads us to believe that the core reason for this problem may not be the frequency of the data.

One possible reason for this problem may be due to the fact that the input features are not rich enough. We simply use one-hot encoded vectors as input features, but it may be possible to engineer features based upon chemical and physical properties of the bases within the sequence. Another cause of this problem may be due to high variance within the data set. It is possible that this particular data set consists of many different strategies so that the distribution of the expert behavior appears to be uniform. In this case, we would have to either abandon the idea of learning offline all together, or incorporate more sophisticated techniques to account for this issue (Nair et al. 2020b), (Peng et al. 2019).

Conclusion

This paper gives the details on our method to address the IRFP using online-offline reinforcement learning. We detailed the architecture of our policy network and described the current results from training on human moves from the Eterna game. Our current results are poor due to several possible reasons. The first reason is unbalanced data, the second reason insufficiently rich features, and the third reason is to high variance within the moves data set. The second part of this project will utilize online RL to train the policy network by using standard policy gradients.

Timeline

The proposed timeline for this project leading up to the final is as follows:

Date	Milestone
10/27-11/3	Define online RL environment.
11/3-11/10	Implement policy gradient algorithm.
11/10-11/17	Training and figure collection.
11/17-12/1	Experiment with different methods and architectures.
12/1-12/14	Generate results and write paper.

Table 1: Proposed Project Timeline

References

Boyer, R. F. 2002. *Concepts in biochemistry*, volume 139. Brooks/Cole Pacific Grove, CA.

Burke, J. M.; and Berzal-Herranz, A. 1993. In vitro selection and evolution of RNA: applications for catalytic RNA, molecular recognition, and drug discovery. *The FASEB journal* 7(1): 106–112.

Churkin, A.; Retwitzer, M. D.; Reinharz, V.; Ponty, Y.; Waldispühl, J.; and Barash, D. 2018. Design of RNAs: comparing programs for inverse RNA folding. *Briefings in bioinformatics* 19(2): 350–358.

Delebecque, C. J.; Lindner, A. B.; Silver, P. A.; and Aldaye, F. A. 2011. Organization of intracellular reactions with rationally designed RNA assemblies. *Science* 333(6041): 470–474.

Dirks, R. M.; Lin, M.; Winfree, E.; and Pierce, N. A. 2004. Paradigms for computational nucleic acid design. *Nucleic acids research* 32(4): 1392–1403.

Dixon, N.; Duncan, J. N.; Geerlings, T.; Dunstan, M. S.; McCarthy, J. E.; Leys, D.; and Micklefield, J. 2010. Reengineering orthogonally selective riboswitches. *Proceedings of the National Academy of Sciences* 107(7): 2830–2835.

Eastman, P.; Shi, J.; Ramsundar, B.; and Pande, V. S. 2018. Solving the RNA design problem with reinforcement learning. *PLoS computational biology* 14(6): e1006176.

Garcia-Martin, J. A.; Clote, P.; and Dotu, I. 2013. RNAiFOLD: a constraint programming algorithm for RNA inverse folding and molecular design. *Journal of bioinformatics and computational biology* 11(02): 1350001.

Hao, C.; Li, X.; Tian, C.; Jiang, W.; Wang, G.; and Mao, C. 2014. Construction of RNA nanocages by re-engineering the packaging RNA of Phi29 bacteriophage. *Nature communications* 5(1): 1–7.

Hochreiter, S.; and Schmidhuber, J. 1997. Long short-term memory. *Neural computation* 9(8): 1735–1780.

Hofacker, I. L. 2003. Vienna RNA secondary structure server. *Nucleic acids research* 31(13): 3429–3431.

Hofacker, I. L.; Fontana, W.; Stadler, P. F.; Bonhoeffer, L. S.; Tacker, M.; and Schuster, P. 1994. Fast folding and comparison of RNA secondary structures. *Monatshefte für Chemie/Chemical Monthly* 125(2): 167–188.

Kingma, D. P.; and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* .

Koodli, R. V.; Keep, B.; Coppess, K. R.; Portela, F.; participants, E.; and Das, R. 2019. EternaBrain: Automated RNA design through move sets and strategies from an Internet-scale RNA videogame. *PLoS computational biology* 15(6): e1007059.

Koodli, R. V.; Rudolfs, B.; Wayment-Steele, H. K.; and Das, R. 2021. Redesigning the Eterna100 for the Vienna 2 folding engine. *bioRxiv* .

Lee, J.; Kladwang, W.; Lee, M.; Cantu, D.; Azizyan, M.; Kim, H.; Limpaecher, A.; Gaikwad, S.; Yoon, S.; Treuille, A.; et al. 2014. RNA design rules from a massive open laboratory. *Proceedings of the National Academy of Sciences* 111(6): 2122–2127.

Levine, S.; Kumar, A.; Tucker, G.; and Fu, J. 2020. Offline reinforcement learning: Tutorial, review, and perspectives on open problems. *arXiv preprint arXiv:2005.01643* .

Mathews, D. H.; and Turner, D. H. 2006. Prediction of RNA secondary structure by free energy minimization. *Current opinion in structural biology* 16(3): 270–278.

Nair, A.; Dalal, M.; Gupta, A.; and Levine, S. 2020a. Accelerating online reinforcement learning with offline datasets. *arXiv preprint arXiv:2006.09359* .

Nair, A.; Dalal, M.; Gupta, A.; and Levine, S. 2020b. AWAC: Accelerating Online Reinforcement Learning with Offline Datasets .

Peng, X. B.; Kumar, A.; Zhang, G.; and Levine, S. 2019. Advantage-weighted regression: Simple and scalable off-policy reinforcement learning. *arXiv preprint arXiv:1910.00177* .

Seetin, M. G.; and Mathews, D. H. 2012. RNA structure prediction: an overview of methods. *Bacterial regulatory RNA* 99–122.

Shijie, C.; Zhijie, T.; Song, C.; and Wenbing, Z. 2006. The statistical mechanics of RNA folding. *Physics* 3: 012.

Verbeke, R.; Lentacker, I.; De Smedt, S. C.; and Dewitte, H. 2019. Three decades of messenger RNA vaccine development. *Nano Today* 28: 100766.

Win, M. N.; and Smolke, C. D. 2008. Higher-order cellular information processing with synthetic RNA devices. *Science* 322(5900): 456–460.

Zhao, Q.; Zhao, Z.; Fan, X.; Yuan, Z.; Mao, Q.; and Yao, Y. 2021. Review of machine learning methods for RNA secondary structure prediction. *PLoS computational biology* 17(8): e1009291.

ZHOU, Z. L.; JIE; and Publishers, M. . C. 2020. *INTRODUCTION TO GRAPH NEURAL NETWORKS*, volume 45. MORGAN CLAYPOOL PUBLISH, 1 edition. ISBN 1939-4608.

Final CS 5080 Fall 2021

Reinforcement Learning For The Inverse RNA Folding Problem

Van Hovenga
University of Colorado-Colorado Springs
1420 Austin Bluffs Pkwy
Colorado Springs, CO 80919
vhovenga@uccs.edu

Abstract

The secondary structure of ribonucleic acid (RNA), is important in many biological applications. There exists several computational methods that predict the secondary structure of RNA from a given sequence of nucleotides. It is often the case, however, that a desirable secondary structure is known a priori. The task then becomes predicting a sequence that conforms to the desired secondary structure. This is known as the inverse RNA folding problem (IRFP). In this work, we utilize reinforcement learning to address the IRFP.

Introduction

Ribonucleic acid (RNA) is a molecule that is nearly ubiquitous across all living organisms. RNA is a poylmer that consists of four nucleotides: guanine (G), uracil (U), adenine (A), and cytosine (C). The sequences of these four nucleotides that are present in RNA are responsible for various biological functions such as coding, decoding, regulation and expression of genes (Boyer 2002). The crucial roles RNA play in living organisms has led to an abundance of research into RNA design and synthesis for drug discovery (Burke and Berzal-Herranz 1993), vaccine design (Verbeke et al. 2019), biosensing (Win and Smolke 2008), and many more applications (Delebecque et al. 2011), (Hao et al. 2014), (Dixon et al. 2010).

The structure of RNA is a particularly important in determining its mechanism of action (Mathews and Turner 2006). For this reason, several computational methods for predicting the structure of RNA from its corresponding sequence (Seetin and Mathews 2012). There are two levels of structure prediction: secondary and tertiary. Secondary structure refers to the bonding of individual base pairs within the sequence. Tertiary structure refers to how the sequence folds in three-dimensional (3D) space. See figure 1 for a visual representation of this difference. In many practical applications, the secondary structure of RNA is sufficient to infer function (Shijie et al. 2006). This paper is concerned only with the secondary structure of RNA.

The problem of predicting secondary RNA structure from a given sequence is important, but in many applications of RNA synthesis, it is already known that a specific type of secondary structure may be conducive to a particular function. In this case, it may be useful to ask the inverse question: given a desired secondary structure, how can we construct a sequence of nucleotides that confirms this structure? This is precisely the inverse RNA folding problem (IRFP).

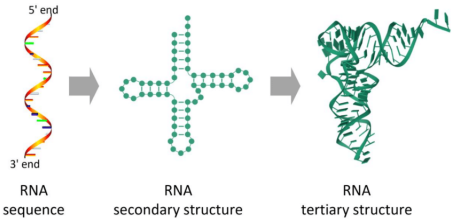

Figure 1: Visual depiction of the difference between the RNA sequence and the associated secondary and tertiary structures (Zhao et al. 2021)

Given a sequence of length n, there exists 4^n possible sequences of nucleotides, making the search space for the IRFP very large, even for small molecules. For this reason, several computational methods have been designed to address this problem (Churkin et al. 2018). In this work, we use reinforcement learning to address the IRFP.

Existing Methods

We now outline several of the computational methods for the IRFP. RNAinverse was the first method developed for the IRFP (Hofacker et al. 1994). This algorithm minimizes the difference between the minimal free energy (MFE) of a given sequence (i.e. the predicted structure) and that of the target structure. This optimization is performed by random mutation of the sequence via an adaptive random walk. RNAiFold (Garcia-Martin, Clote, and Dotu 2013) utilizes constraint programming to optimize one of three possible targets: the minimum free energy (MFE) structure, the total free energy, or the ensemble defect. Since the algorithm is deterministic, RNAiFold ensures optimality in its solutions. NUPACK utilizes partition function algorithms and stochastic kinetics simulations to mimic the paradigms of wet lab RNA design (Dirks et al. 2004). MODENA utilizes a multi-objective genetic algorithm that maximizes the stability of the output structures while minimizing their distance from the target structure. Reinforcement learning has also been

utilized as an approach to the IRFP (Eastman et al. 2018).

A starkly different approach to the IRFP comes from the Eterna game (Lee et al. 2014). Eterna is a online open laboratory that formulates the IRFP as a puzzle to be solved by humans. Players are given a sequence and a target structure with the goal of altering the initial sequence such that the predicted secondary structure matches the target structure. Eterna utilizes the Vienna algorithm for secondary structure prediction (Hofacker 2003). Eterna has over 250,000 registered players with over 225,000 unique puzzles solved.

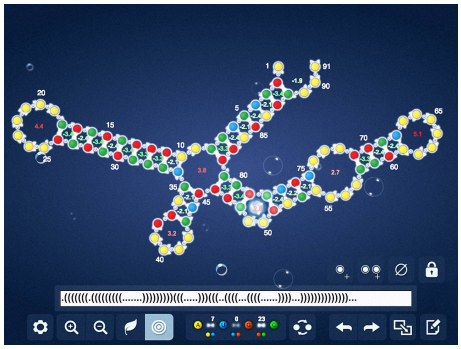

Figure 2: Screenshot of the Eterna game interface.

Eterna has made significant contributions to research of the IRFP. EternaBot is a rules-based algorithm designed by strategies informed by the top Eterna players (Lee et al. 2014). EternaBrain is a convolutional neural network (CNN) trained on 1.8 million player moves that utilizes single action payout of six strategies compiled by human players to predict the optimal sequence for a given target structure (Koodli et al. 2019).

Method

Prior to Eterna, all computational methods for the IRFP relied on utilizing physical simulation along with some form of optimization. In some cases, such as NUPACK, the method is guided by domain knowledge from wet lab research in RNA synthesis. The advent of Eterna, however, allows for an additional guiding principle for the IRFP; namely, that of human intuition from playing the Eterna game. EternaBot attempts to exploit this human knowledge via a rules-based algorithm guided by the strategies of players. Due to its rules-based nature, however, Eternabot fails to generalize to structures that are not common within the game. EternaBrain initially attempted to reconcile this lack of generalizability by training a CNN to recreate the moves of Eterna players in a supervized manner. Training the CNN alone, however, did not yield results that outperformed random guessing when tested on unseen data. Thus, the authors paired the CNN with a deterministic, single action payout algorithm comprised of six canonical strategies that are standard among Eterna players. This strategy was able to outperform many existing algorithms, suggesting that a combination of domain knowledge from gameplay with machine learning may be promising path for research into the IRFP.

One of the main limitations of the computational methods that do not utilize data from the Eterna game is that the optimization usually relies on some form of stochastic search. Typically, this is achieved by randomly mutating nucleotides in the underlying sequence and calculating their equillibrium secondary structures. Favorable mutations are then kept and unfavorable mutations are discarded. This method does not scale to larger sequences since the search space becomes prohibitively large. Indeed, the A-U, C-G, G-U base pairing scheme implies that a strand of size $n = p + u$ has $6^{p/2}4^u$ possible sequences, where p and u are the number of paired and unpaired nucleotides respectively. Thus, a strand of size 74 with 40 paired nucleotides and 34 unpaired nucleotides would have approximately 10^{36} sequences. Thus, it is highly desirable to reduce the size of this search space in order to allow methods to maintaing their performance on longer strands. The aforementioned methods do so by guiding the agent using player heuristics from the Eterna game.

Currently, all approaches that utilize domain knowledge from Eterna do so deterministically, i.e., from a set of predefined strategies. Although some of these methods yield impressive performance, they are restrictive in that they do not allow for modification of player strategy during optimization. Thus, these algorithms are bounded by the capabilities of the human strategies from which they were formed. To circumvent this limitation, we utilize a nondeterministic strategy learning via reinforcement learning to address the IRFP.

Asynchronous Advantage Actor Critic (A3C)

Due to the size of the search space in the IRFP, it is not possible to utilize standard reinforcement learning algorithms, such as Q-learning or value iteration, because it is simply not feasible to store the values of a Q-table or V-table with such a large search space. Thus, we rely on function approximation via neural networks. There currently exists several deep reinforcement learning frameworks. In this work, we chose to utilize the Asynchronous Advantage Actor Critic (A3C) algorithm (Mnih et al. 2016).

The A3C algorithm is a multi-agent, asynchronous, policy gradient method. Specifically, multiple policy networks are trained asynchronously in separate environments using an actor critic learning strategy which we will describe below. The parameters of each agent are shared with a global agent. The asynchronous nature of the algorithm allows for a more diverse set of experiences the global agent since its parameters are trained using the experiences of many, separate agents. This diversity mitigates sample bias and thereby reduces the bias of the global network. See figure 3 for a pictorial representation of the asynchronous aspect of the A3C algorithm (Sciforce 2021).

The A3C algorithm relies on two neural networks- and actor network $\pi_\theta(\cdot)$ and a critic network $V_v(\cdot)$. The actor network takes a state as an input and outputs a probability distribution over the admissible actions in that state. The value network takes a state as an input an outputs an estimate of the value of that input state. These two networks are trained in parallel as the agent interacts with the environment.

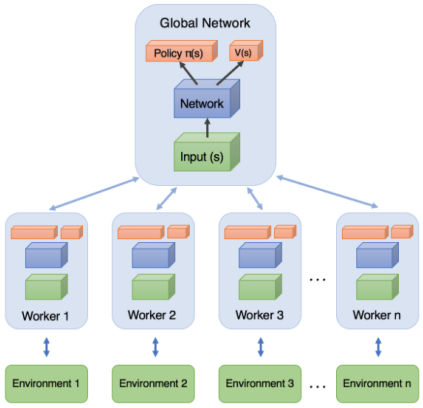

Figure 3: Diagram showing the how A3C learns asynchronously. Each worker network experiences interaction with separate environments, thereby leading to a more diverse set of experiences. The parameters of each worker is shared with a global network.

The critic network V_v can be trained by minimizing the squared error of its outputs with the observed rewards of a given episode. Specifically, let r_i be the reward at time step i. Then we may estimate the n-step discounted at time t, R_t, by using the equation

$$R_t = \sum_{k=0}^{n-1} \gamma^k r_{t+k+1} + \gamma^n V_v(s_{t+n+1}). \tag{1}$$

With this equation, we can train v_v by minimizing the loss

$$\mathcal{L}(v) = \sum_{t=0}^{T-1} (R_t - V_v(s_t))^2 \tag{2}$$

Since the actor network outputs over the space of actions, it is possible to train it using policy gradients. It has been shown, however, that policy gradients can lead to high variance in the parameter updates (Mnih et al. 2016). A3C mitigates this variance by introducing the advantage function. Intuitively, the advantage function is meant to measure how much it is to take a specific action relative to the average of all actions at a given state. Specifically, the advantage function is defined to be

$$\begin{aligned} A(s_t, a_t) &= Q(s_t, a_t) - V_v(s_t) \\ &= r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t) \end{aligned}$$

A3C uses the advantage function to reduce the variance of the policy gradients simply by multiplying it with the standard policy gradient. Specifically, we calculate the gradients of the actor network using

$$\nabla\theta J_\theta = \sum_{t=0}^{T-1} \nabla\theta \log \pi_\theta(a_t, s_t) A(s_t, a_t) \tag{3}$$

Architecture

In this section, we outline the structure of the learning environment. We then describe the architecture of the policy network and discuss the training procedure.

Environment

There are several environments we may use to simulate the equilibrium secondary structure from RNA sequences. Secondary structure is generally predicted by minimizing the free energy of the sequence. Since our method will be influenced by Eterna play, we will simulate secondary structure formation using the Vienna engine (Hofacker 2003). It is worth noting, however, that the performance of IRFP algorithms is dependent on the dynamics of the folding engine (Koodli et al. 2021). This dependency is due to the fact that most methods optimize the MFE of structures, which is clearly biased by how MFE is estimated by the folding engine.

Graph Policy Network

We utilize a graph neural network architecture for the policy network. The reasons for this decision are two-fold. Firstly, variation in the lengths of each puzzle make it difficult to input the data into a traditional neural network. Secondly, graph neural networks allow us to incorporate the structure of the solution conformation via edges that would not be possible in a simple sequence model. With the graph neural network architecture, we formulate the task of predicting moves as a node classification task as follows.

Graph Structure The inputs to our graph neural network consist of an adjacency matrix defining connections between nodes, a node feature matrix, and a node label matrix. The adjacency matrix is defined via the bonds in the target structure. The ij^{th} entry of this matrix is 1 iff the bases in the i and j position of the sequence are bonded. For example, the adjacency matrix of the structure **((..))** is given by the matrix

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

. The node feature matrix consists of one-hot encoded vectors corresponding to the nucleotides in the given state. The node label matrix consists of one-hot encoded vectors corresponding to five different classes: change to A, change to C, change to G, change to U, and don't change. With this formulation, the task of predicting player moves becomes a node classification problem. We may predict multiple player moves in unison simply by assigning multiple change labels to the graph. In our current implementation, we consider 12 player moves in each graph.

Architecture Our policy network follows a consolidate, aggregate, and predict scheme that is typical in many graph neural networks (ZHOU, JIE, and Publishers 2020). Assume that we have a state sequence, $(\mathbf{x}_0, \cdots, \mathbf{x}_N)$ and that we would like to predict the move on node $\mathbf{x_i}$. We refer to this as the target node. Since each base can bond with a maximum of three other bases, the target node has a maximum of three neighbors. Assuming that the target node is not located on either end of the sequence, there will be a sequence of nodes

to the left of the target node $(\mathbf{x}_0, \cdots, \mathbf{x}_{i-1})$, a sequence of nodes to the right of the target node, $(\mathbf{x}_{i+1}, \cdots, \mathbf{x}_N)$, and potentially an additional node bonded to the target node, $\mathbf{x}_j$, depending on the target conformation. We refer to these as the left sequence, the right sequence, and the center sequence. Note that the target node and the center sequence always have length one.

The first step of the network is to consolidate the features of the target node along with the right, left, and center sequences into four separate vectors. We do this by sending each sequence through a five layer convolutional neural network (CNN). This gives us the left, right, center, and target node representations which we denote by $\widetilde{\mathbf{x}_L}$, $\widetilde{\mathbf{x}_R}$, $\widetilde{\mathbf{x}_C}$, and $\widetilde{\mathbf{x}_T}$ respectively. The second step is to aggregate these representations into a single representation. This is done simply by concatenating each representation. We denote this single representation as $\mathbf{x}'_i$. We finally predict the label of the target node by sending $\mathbf{x}'_i$ through a three layer multi-layer perceptron. The actor network outputs a distribution over the states by applying the softmax function to this output. The critic network outputs a scalar value by sending the output through a single linear layer with no activation. Figure 4 is a diagram depicting the architecture of the GNN.

Evaluation

Eterna provides a benchmark for performance of IRFP programs that consists of 100 structures with known solutions of varying difficulty known as the Eterna100. This benchmark consists of a mix of human created designs and known RNA structures. So far, no algorithm can solve all 100 structures on either data set. We use this benchmark to evaluate the performance of our algorithm. For sake of simplicity, we only consider whether or not our method can solve a given puzzle and we ignore other metrics such as free energy and base pair distance.

Training Procedure

The A3C involves several hyperparameters. Table 1 lists the hyperparameters and their corresponding values which we used during the training process. Due to limitations on the time for completing this project, these parameters were simply selected ad-hoc and no hyperparameter optimization was performed.

We trained the algorithm using 50,000 target structures of varying difficulty from the Eterna game. We employed a simple curriculum strategy by dividing this data set into two subsets, one consisting of easy puzzles and one consisting of hard puzzles. We first trained using the easy puzzles for 500,000 steps. We then periodically added in harder puzzles for the next 500,000 steps until the entire data set was exhausted.

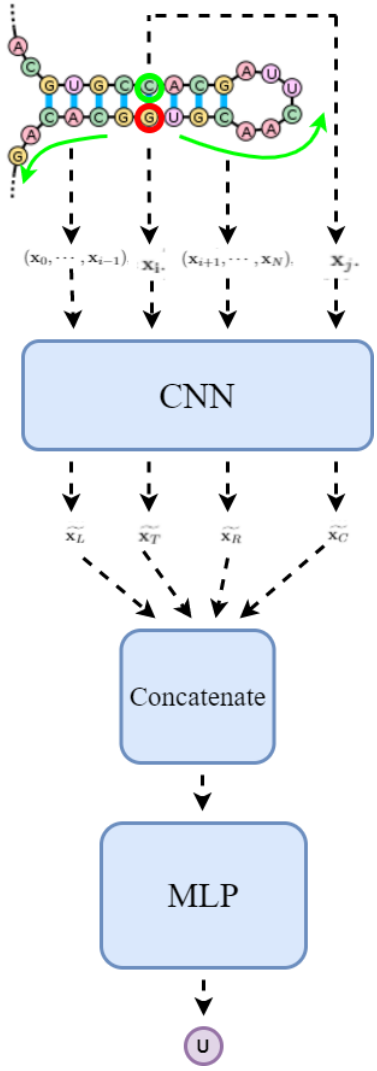

Figure 4: Visual depiction of the architecture of the GNN. In this case, the red outlined base is the target node.

Results

We compared our method with three other methods: RNAInverse (Hofacker et al. 1994), MODENA (Taneda 2011), and NEMO (Portela 2018). RNAInverse is a classical method for the IRFP. It uses an adaptive random walk to minimize

Hyperparameter	Purpose	Value
entropy_weight	Regularizing term	0.1
γ	Discount factor	.7
lr	Learning rate	.1 decaying
ϵ	Exploring factor	.8 decaying
num_workers	Number of agent networks	10
max_rollout	Maximum number of steps	50

Table 1: Table of the various hypereparamenters and their corresponding values used during training.

base pair distance using stochastic mutation. MODENA utilizes two objective functions in the optimization: free energy and base pair distance. These two objectives are minimized using a standard genetic algorithm. NEMO utilizes a simple monte-carlo tree search for the IRFP. This monte-carlo tree search is guided by player heuristics developed from the Eterna game. We compared the performance of our method to these existing methods on the Eterna100 dataset. Since out policy is stochastic, we set a limit on the solve time of our algorithm. Specifically, if our model did not solve the puzzle within 20 minutes, we consider the puzzle to be unsolved.

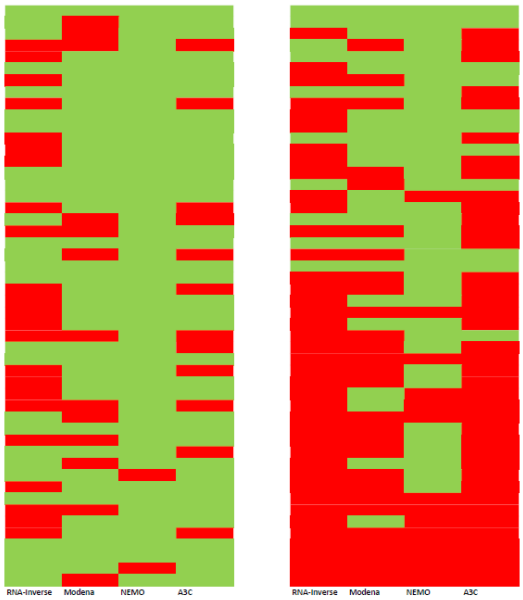

Figure 5: Solve status of each puzzle in the Eterna100 for our method along with RNAInverse, MODENA, and NEMO. Green cells indicate a successful solve and red cells indicate a failed solve. The columns are ordered as follows: InverseRNA, MODENA, NEMO, ours.

Figure 5 and 2 show the solve counts for each method. Although our method does not perform the best out of the other methods, it also does not perform the worst. As expected, our the performance of our method deteriorates as the size of the input puzzles increases.

Method	Number of solves
RNAInverse	35
MODENA	52
NEMO	82
Ours	46

Table 2: Number of puzzles solved by each method.

Discussion and Future Work

There are several possible ways by which our method could be improved. Firstly, it is clear that the performance of our method dwindles on larger puzzles. Although this occurs in part due to the computational constraints associated with solving such a large sequence, another potential reason could be insufficiently large training data. Our method was trained on puzzles with a maximum length of 32, but the maximum length puzzle in the Eterna100 is over 200. Increasing the size of the puzzles in the training set could better prepare the model for inference on larger puzzles.

Another way our method could be improved upon is the calculation of rewards. We treat the solving of a puzzle as an episodic task. Thus, we assign a single reward upon the completion of the puzzle and no rewards for any actions that do not lead to a completion. This leads to sparse rewards and therefore inhibits training. Using more sophisticated rewards, such as rewarding lowering free energy or base pair distance, could lead to a more efficient and powerful model. Our method also takes the entire sequence as the input. This becomes computationally prohibitive for larger sequences. Inputting local sub-structures rather than entire sequences could allow for training and inference to be run on large sequences.Finally, our algorithm does not rely on any human-inspired heuristics. Since humans can solve all 100 puzzles on the Eterna100, it may be worth incorporating human strategies to guide the agent.

Conclusion

In this paper, we address the inverse RNA folding problem using reinforcement learning. We utilized the asynchronous advantage actor critic model along with a graph convolutional neural network in our model's architecture. Our model was trained on 50,000 structures pulled from the Eterna RNA folding game. We evaluated the performance on the Eterna100 benchmark. In a comparison with three other methods, we showed that our method can effectively solve 46 puzzles on the Eterna100 benchmark. Our method is particularly well suited for smaller puzzles, but the performance decays as puzzles get larger. Finally, we outlined several possible ways that our method could be improved.

References

Boyer, R. F. 2002. *Concepts in biochemistry*, volume 139. Brooks/Cole Pacific Grove, CA.

Burke, J. M.; and Berzal-Herranz, A. 1993. In vitro selection and evolution of RNA: applications for catalytic RNA, molecular recognition, and drug discovery. *The FASEB journal* 7(1): 106–112.

Churkin, A.; Retwitzer, M. D.; Reinharz, V.; Ponty, Y.; Waldispühl, J.; and Barash, D. 2018. Design of RNAs: comparing programs for inverse RNA folding. *Briefings in bioinformatics* 19(2): 350–358.

Delebecque, C. J.; Lindner, A. B.; Silver, P. A.; and Aldaye, F. A. 2011. Organization of intracellular reactions with rationally designed RNA assemblies. *Science* 333(6041): 470–474.

Dirks, R. M.; Lin, M.; Winfree, E.; and Pierce, N. A. 2004. Paradigms for computational nucleic acid design. *Nucleic acids research* 32(4): 1392–1403.

Dixon, N.; Duncan, J. N.; Geerlings, T.; Dunstan, M. S.; McCarthy, J. E.; Leys, D.; and Micklefield, J. 2010. Reengineering orthogonally selective riboswitches. *Proceedings of the National Academy of Sciences* 107(7): 2830–2835.

Eastman, P.; Shi, J.; Ramsundar, B.; and Pande, V. S. 2018. Solving the RNA design problem with reinforcement learning. *PLoS computational biology* 14(6): e1006176.

Garcia-Martin, J. A.; Clote, P.; and Dotu, I. 2013. RNAiFOLD: a constraint programming algorithm for RNA inverse folding and molecular design. *Journal of bioinformatics and computational biology* 11(02): 1350001.

Hao, C.; Li, X.; Tian, C.; Jiang, W.; Wang, G.; and Mao, C. 2014. Construction of RNA nanocages by re-engineering the packaging RNA of Phi29 bacteriophage. *Nature communications* 5(1): 1–7.

Hofacker, I. L. 2003. Vienna RNA secondary structure server. *Nucleic acids research* 31(13): 3429–3431.

Hofacker, I. L.; Fontana, W.; Stadler, P. F.; Bonhoeffer, L. S.; Tacker, M.; and Schuster, P. 1994. Fast folding and comparison of RNA secondary structures. *Monatshefte für Chemie/Chemical Monthly* 125(2): 167–188.

Koodli, R. V.; Keep, B.; Coppess, K. R.; Portela, F.; participants, E.; and Das, R. 2019. EternaBrain: Automated RNA design through move sets and strategies from an Internet-scale RNA videogame. *PLoS computational biology* 15(6): e1007059.

Koodli, R. V.; Rudolfs, B.; Wayment-Steele, H. K.; and Das, R. 2021. Redesigning the Eterna100 for the Vienna 2 folding engine. *bioRxiv* .

Lee, J.; Kladwang, W.; Lee, M.; Cantu, D.; Azizyan, M.; Kim, H.; Limpaecher, A.; Gaikwad, S.; Yoon, S.; Treuille, A.; et al. 2014. RNA design rules from a massive open laboratory. *Proceedings of the National Academy of Sciences* 111(6): 2122–2127.

Mathews, D. H.; and Turner, D. H. 2006. Prediction of RNA secondary structure by free energy minimization. *Current opinion in structural biology* 16(3): 270–278.

Mnih, V.; Badia, A. P.; Mirza, M.; Graves, A.; Lillicrap, T.; Harley, T.; Silver, D.; and Kavukcuoglu, K. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, 1928–1937. PMLR.

Portela, F. 2018. An unexpectedly effective Monte Carlo technique for the RNA inverse folding problem. *BioRxiv* 345587.

Sciforce. 2021. Reinforcement learning and asynchronous actor-critic agent (A3C) algorithm, explained. URL https://medium.com/sciforce/reinforcement-learning-and-asynchronous-actor-critic-agent-a3c-algorithm-explained-f0f3146a14ab.

Seetin, M. G.; and Mathews, D. H. 2012. RNA structure prediction: an overview of methods. *Bacterial regulatory RNA* 99–122.

Shijie, C.; Zhijie, T.; Song, C.; and Wenbing, Z. 2006. The statistical mechanics of RNA folding. *Physics* 3: 012.

Taneda, A. 2011. MODENA: a multi-objective RNA inverse folding. *Advances and applications in bioinformatics and chemistry: AABC* 4: 1.

Verbeke, R.; Lentacker, I.; De Smedt, S. C.; and Dewitte, H. 2019. Three decades of messenger RNA vaccine development. *Nano Today* 28: 100766.

Win, M. N.; and Smolke, C. D. 2008. Higher-order cellular information processing with synthetic RNA devices. *Science* 322(5900): 456–460.

Zhao, Q.; Zhao, Z.; Fan, X.; Yuan, Z.; Mao, Q.; and Yao, Y. 2021. Review of machine learning methods for RNA secondary structure prediction. *PLoS computational biology* 17(8): e1009291.

ZHOU, Z. L.; JIE; and Publishers, M. . C. 2020. *INTRODUCTION TO GRAPH NEURAL NETWORKS*, volume 45. MORGAN CLAYPOOL PUBLISH, 1 edition. ISBN 1939-4608.

Proposal CS 4080 Fall 2021

Automated Signal Trading using Ensemble Reinforcement Learning

Daniel Boyle and Jugal Kalita
1420 Austin Bluffs Pkwy
Colorado Springs, CO 80919

Abstract

Deciding when to buy, sell and hold an acquired asset is an extremely complex task. Within financial markets, there are a countless number of possible stimuli that can effect the price of an asset at any given point in time. In this paper, we propose two different Deep Reinforcement Learning (DRL) approaches for automated stock market trading. The first method looks at using a single agent with a multi-policy ensemble and centralized critic, while the second approach utilizes a multi-agent ensemble. Both approaches will be trained on stocks within the US stock exchange, with the goal of maximizing risk-adjusted return. Each approach will then be evaluated against common portfolio allocation strategies.

Introduction

Determining the optimal time to buy and sell an asset within a financial market is a topic that has been constantly sought after for decades. If answered, it could open the door for generating very large profits in a relatively short period of time and ultimately would revolutionize the world of trading. While at first glance it may not seem hard, it is an extremely complicated task, even for the most advanced stockbrokers. Very often, experts are in disagreement on which methods are capable of producing the greatest results.

The complexity of trading stems from the vast number of potential stimuli that can impact a financial market, making it virtually impossible to accurately predict the future price of an asset. For example in the stock market, the price of a company's stock can be affected by numerous business attributes and statistics including but not limited to the companies profits, losses, products sold, research interests, and currently held contracts. On top of these, there are many external factors that can have a large impact on a stock including the overall trend of the entire market and public sentiment created through platforms such as the news and social media posts. Additionally, there are potentially thousands of other attributes that can impact a stock price in some way or another. Due to the large amount of different factors, many traders have tended to focus on analyzing specific attributes of an asset, to help reduce the subset of variables that are considered, ultimately reducing the complexity needed for their strategies. While no subset is completely capable of predicting price movements, they can be used to create strategies that work under certain conditions.

The two largest methods of analysis used during trading are fundamental and technical analysis. Fundamental analysis is focused on the use of detailed data obtained from a company's financial statements as well as an array of other economic factors to predict future earnings (Abarbanell and Bushee 1997). This type of analysis is generally focused on long-term trading. While fundamental analysis tends to dive deep into the public data for a company, technical analysis focuses on studying price movements and chart patterns to help detect trends, reversal patters and other types of technical signals (Edwards, Magee, and Bassetti 2018). Technical analysis tends to more reliable when used for short-term profits through intra-day and swing trading. While it can also be used for long-term trading, often it needs to be paired with other forms of analysis to be the most effective. This is largely due to the potential impact of external factors that can not be foreseen through pure technical analysis.

In the last couple of decades, machine learning has been applied to various areas related to trading within financial markets. Many of the original uses of machine learning within the stock market primarily focus on trying to forecast the price of a stock for a certain number of steps into the future (Bao, Lu, and Zhang 2004), (Niaki and Hoseinzade 2013). Typically, these models are then used as a basis for determining when to buy or sell a stock. Unfortunately, they are only somewhat accurate and their forecasts are only valid for a very short period of time into the future due to unforeseen factors that they do not consider. Additionally, these methods tend to bad at generalization and perform badly when used against stocks that they are not trained on.

More recently, Deep Reinforcement Learning (DRL) has started to become more popular in the realm of machine learning, being used to tackle complex problems. One of the most well known uses of DRL is Google's Alpha Zero Chess Agent that is capable of beating the best chess players in the world (Silver et al. 2018). Outside of the realm of games, researches have started to apply DRL to financial realm including stock trading. Unlike other machine learning approaches related to asset trading such as price forecasting, DRL has the goal of trying to learn automate effective trading strategies.

Currently, more than 80 percent of all trading volume executed within the stock market is a result of automated trading. Therefore, the demand for more accurate and capable automated trading methods is at an all time high. In this paper, we propose the use of two different Ensemble Deep Reinforcement Learning approaches that are used to perform automated short-term signal trading in the US Stock market.

Background

Reinforcement Learning is a subset of machine learning that focuses on teaching an agent or set of agents to map states within an environment to an optimal action, with the goal of maximizing some form of numerical reward (Sutton, Barto, and others 1998). Basic Reinforcement learning problems are traditionally formalized as a Markov Decision Process, or MDP (Howard 1960).

Markov Decision Process

A Markov Decision Process is a typically represented as a 5-tuple model, T = $\{S, A, P, R, \gamma\}$, where:

- S represents the set of all possible states within the environment. Also commonly referred to as the *state space*.
- A is the set of actions that can be taken within an environment. Also known as the *action space*.
- P represents the probability of transitioning from one state to another, given a specific action, i.e. $P(s_t, s_{t+1}) = P(s_{t+1}|a_i, s_t)$ where $s_t \in S$ and $a_i \in A$.
- R represents the reward function, where $R(s_t, s_{t+1})$ represents the reward when transitioning from state s_t to state s_{t+1}.
- γ represents the *discount rate*, which is a value between 0 and 1 that is used to change the emphasis on future rewards.

In Reinforcement learning, the main objective with respect to an MDP, is to generate a policy function $\pi(s_t|a_t)$ that maximizes the expected cumulative reward over N timesteps. The expected cumulative reward can represented as the following:

$$\mathbb{E}_\pi \left[\sum_{t=0}^{N} \gamma^t R_{a_t}(s_t, s_{t+1})] \right]. \quad (1)$$

Learning Approaches

Within RL, there are two different general approaches to maximizing the cumulative reward:

Value-Based Reinforcement Learning.

For value-based learning, the main objective is to learn the policy function π implicitly. One way this is done, is to learn a value function, $V^\pi(s)$, where each state $s \in S$ maps a state s to a numeric value v. This value represents an estimated "goodness" score for a specific state. Another common value-based method, is to learn a quality function $Q_\pi(s, a) = q$. In this situation, each state-action pair maps to a quality score, or q, that measures "how good" an action a is while in state s.

Policy-Based Reinforcement Learning. In policy-based learning, the goal is to directly learn the optimal policy $\pi(s|a)$ that maps each state s to a set of action probabilities.

Additionally, a hybrid approach known as Actor-Critic aims at utilizing the best components of both Value and Policy-based learning. For actor critic methods, the policy function is referred to as the *Actor*, and the value function is referred to as the *Critic*. The the goal of the Actor, or policy function, is to learn the appropriate action to take for all possible states. When the actor performs an action, and the agent receives a reward, the critic uses the reward to criticize the agent on how well it chose an action for the given state. This results in an update to actor's current policy. This process is repeated, until the optimal policy is achieved.

Related Work

Due to the potential profits that can be made from asset trading, there has been a vast amount of research put towards applying machine learning to further improve this process. In 2016, Rajashree Dash and Pradipta Kishore Dash created a trading framework that combined the use of technical analysis and machine learning to generate buy and sell trading signals (Dash and Dash 2016). Their approach starts by analyzing stock data and calculating various common technical indicators on the provided data. These indicators are then used as the input features for a Functional Link Artificial Neural Network (FLANN), that is trained to output a value between 0 and 1. Depending on the current stock trend and the user's current position in a stock, the output is used to determine when to buy, sell or hold a stock. Their methods proved effective, beating out other traditional and machine learning trading stock trading methods with a profit of 47 and 24 percent respectively when tested against the BSE SENSEX and SP500 across a period of 250 days.

Taking this concept one step further, Deng et al.'s proposed an approach for financial signal representation and trading using Deep RL (Deng et al. 2017) to automate the trading process. They take a fuzzy learning approach that utilizes a Convolutional Neural Network (CNN) and Long Short-Term Memory (LTSM) network. To drive the learning process, they utilize the Sharpe Ratio as a reward signal. The approach obtained good results, out performing other current known DRL trading strategies while also increasing its trading efficiency, resulting in a significantly smaller number of performed trades.

Data

There is a large amount of market data that is available in relation to the US stock market and related companies, dating back several decades. This data includes traditional price and volume data, sentiment analysis data, company earning reports and much more. In this paper, we look primarily at historical price data within the previous 5 years. This data can be obtained from numerous different sources including Quandle, FinnHub and Yahoo Finance. Additionally data can be gathered at different intervals with the most popular intervals being 1-minute, 5-minute and 1-day. We will use a wide variety of different stocks obtained at different inter-

vals to determine how well our agent performs in different scenarios.

Methodology

To form the problem of asset trading into a process that reinforcement learning can tackle, we have to outline the environment that the agent(s) will be trading in as well as formulate the problem into a Markov Decision Process that a Reinforcement learning agent can learn.

Environment

For the environment, we propose the use of a simple environment that simulates a financial market using historical data. This gives us the ability to simulate trading within different types of markets including the stock, forex and crypto markets, as well as control the step size of the environment. To ensure a level of simulation fidelity within the environment, closer to an actual financial market, the environment will include a 1 percent transaction fee for all all trades, and will simulate trading time by adding in a random delay between when a trade occurs and the agent obtains the asset.

Problem Formulation

In order to outline the problem of asset trading as a Markov Decision Process, we need to define the notion of state or observation, actions, and rewards to be used within the market environment.

Observation The total number of factors that can impact a stocks price at a given point in time within the the US stock exchange is astronomically high. It includes many different categories of information including price history, transaction history, public sentiment data, company financial statements, overall market trends, political happenings and much more. Unfortunately, it is not possible to capture the entire state of the stock market for an agent to observe. Therefore, we look to the idea of an observation, which is a partial representation of the environment that is visible to the agent at a given point in time within the environment. To keep this simple, we will use a minimal set of items within the observation to avoid overwhelming the agent with too much information, since this can make it difficult for an agent to learn how to map an observation to an appropriate action. For our environment, we use the notion of an observation frame that contains the following information:

- Account Balance
- Available Cash
- Assets held
- OCLHV Values (Open, Close, Low, High, Volume)
- Technical indicators including MA, EMA, RSI, MACD, and ADX.

To help give the agent a better notion of history, each observation contains the previous N number of observation frames, where N will be a tuned parameter.

Actions To avoid over complicating the trading process and entering the realm of margin accounts, we will only allow the agent to perform the following actions:

- Market Buy (N stocks)
- Market Sell (N stocks)
- Hold (Do Nothing)

The market buy option will buy N stock at the current market price, while the market sell action will sell N stock at the current market price. The hold action will do nothing. When buying a stock, the agent will have the ability to use between 0 and 100 percent of its available cash to purchase shares of an asset at a given point in time. Similarly, the agent will be able to sell between 0 and 100 percent of its current available shares.

Rewards There are many different methods that could be used for formulating the notion of rewards within the asset trading process depending on the goal you are trying to reach. Often many papers tend to focus on the use of returns for rewarding the agent. While this has been shown to lead to positive results, we will be more focused on risk-adjusted return, i.e. the Sharpe Ratio. The idea, is that by using a risk-adjusted return, it will help to reduce the overall amount of risk that the agent takes when trading an asset.

Model Approach

For this paper, we propose the use of two different ensemble reinforcement learning approaches for tackling our problem. The first approach consists of a single agent that contains an ensemble of N different policies with a centralized critic. Each policy will be trained alongside one another, and combined using a weighted voting strategy to determine which action is best to take at a given time. One critic function will be used to evaluate the output of the action obtained at each training step. A diagram of this approach can be seen below in Figure 1 below.

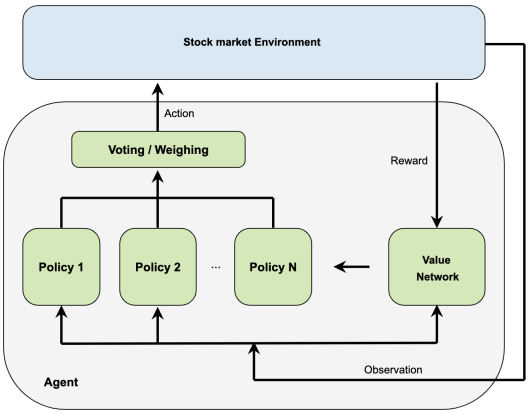

Figure 1: Multi-Policy Ensemble Model

Additionally, we propose a second ensemble approach that N different agents, each that are trained separately on the same set of data. Similarly to the policies above, the best action will be chosen using a weighted voting process. This approach can be seen outlined in Figure 2 below.

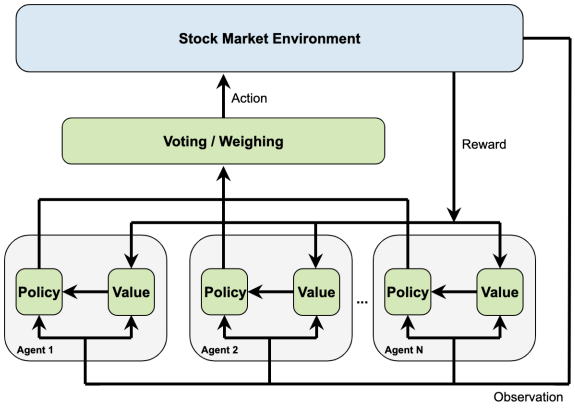

Figure 2: Multi-Agent Ensemble Model

Evaluation

To determine the effectiveness of each approach, we utilize commonly used metrics for evaluating stock trading performance paired with a comparison to baseline trading strategies. These can be seen outlined in the following sections.

Metrics

For asset trading, there are a number of commonly used metrics that are often used to measure the performance of a trading strategy. In this paper, we will use the following metrics for evaluating our agents performance:

- **Profit and Loss (P&L)**: Represents the profit or loss obtained over a period of time.
- **Overall Return**: The percentage of profit or loss obtained over a period of time.
- **Sharpe Ratio**: Risk adjusted return
- **Annualized Return**: The geometric average of the profits gained within an investment period.
- **Volatility**: Annualized measure of variance among within an investment period.

Comparison

On top of evaluating the returns and volatility of each approach, we will also compare their performance against common popular portfolio strategies. This includes a buy and hold strategy against the Dow Jones industrial average index and the min-variance portfolio allocation strategy. The min-variance strategy, is a diversified portfolio strategy that consists of trading a collection of individual, volatile securities that are not correlated with one another, that ultimately result in a low risk strategy with a relatively good rate of return.

Timeline

Below in Table is the proposed timeline for the project leading up to the mid-term milestone.

Date	Task
09/27-09/29	Environment setup
10/04-10/06	Model setup
10/11-10/13	Initial Training / Evaluation
10/18-10/20	Midterm Paper / Presentation

Table 1: Project Timeline

Conclusion

Over the past several decades, automated trading accounts for the majority of all trading volume within the US stock exchange. Many automated trading strategies that exist consist of signficant amount of hard-coded conditional logic and only work in certain scenarios or with certain conditions. In this paper, we propose two different ensemble-based Deep Reinforcement Learning approaches aimed towards generalizing automated stock trading with the goal of maximizing potential risk-adjust returns. Going forward, each method will be evaluated based on their adjusted risk-return and compared with common popular portfolio allocation strategies to determine their effectiveness.

References

Abarbanell, J. S., and Bushee, B. J. 1997. Fundamental analysis, future earnings, and stock prices. *Journal of accounting research* 35(1):1–24.

Bao, Y.; Lu, Y.; and Zhang, J. 2004. Forecasting stock price by svms regression. In Bussler, C., and Fensel, D., eds., *Artificial Intelligence: Methodology, Systems, and Applications*, 295–303. Berlin, Heidelberg: Springer Berlin Heidelberg.

Dash, R., and Dash, P. K. 2016. A hybrid stock trading framework integrating technical analysis with machine learning techniques. *The Journal of Finance and Data Science* 2(1):42–57.

Deng, Y.; Bao, F.; Kong, Y.; Ren, Z.; and Dai, Q. 2017. Deep Direct Reinforcement Learning for Financial Signal Representation and Trading. *IEEE Transactions on Neural Networks and Learning Systems* 28(3):653–664.

Edwards, R. D.; Magee, J.; and Bassetti, W. C. 2018. *Technical analysis of stock trends*. CRC press.

Howard, R. A. 1960. *Dynamic Programming and Markov Processes*. Cambridge, MA: MIT Press.

Niaki, S., and Hoseinzade, S. 2013. Forecasting SP 500 Index using Artificial Neural Networks and Design of Experiments. *Journal of Industrial Engineering International* 9:1–9.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science* 362(6419):1140–1144.

Sutton, R. S.; Barto, A. G.; et al. 1998. *Introduction to Reinforcement Learning*, volume 135. MIT Press Cambridge, MA.

Midterm CS 5080 Fall 2021

Automated Signal Trading using Ensemble Reinforcement Learning

Daniel Boyle and Jugal Kalita
1420 Austin Bluffs Pkwy
Colorado Springs, CO 80919

Abstract

Deciding when to buy, sell and hold an acquired asset is an extremely complex task. Within financial markets, there are a countless number of possible stimuli that can affect the price of an asset at any given point in time. In this paper, we propose using an ensemble-based Deep Reinforcement Learning (DRL) approach for automated stock market trading. The model is trained and tested against the top-10 stock allocations within the Dow Jones Industrial Average (DJIA). Our agent ensemble outperformed the B&H and Min-Variance baseline strategies for all but one stock, obtaining an average Sortino Ratio of 1.620 across the tested stocks. Additionally, our agent outperforms two state-of-the-art DRL trading approaches, obtaining higher Sortino Ratios and returns for three out of five stocks tested within the DJIA.

Introduction

Determining the optimal time to buy and sell an asset within a financial market is a topic that has been constantly sought after for decades. If answered, it could open the door for generating a large amount of wealth in a relatively short period of time and ultimately would revolutionize the world of trading. While at first glance it may not seem hard, it is an extremely complicated task, even for the most advanced stockbrokers. Very often, experts are in disagreement on which methods are capable of producing the greatest results.

The complexity of trading stems from the vast number of potential stimuli that can impact a financial market, making it virtually impossible to accurately predict the future price of an asset. For example, in the stock market, the price of a company's stock can be affected by numerous business attributes and statistics including but not limited to the companies profits, losses, products sold, research interests, and currently held contracts. On top of these, there are many external factors that can have a large impact on a stock including the overall trend of the entire market and public sentiment created through platforms such as the news and social media. Additionally, there are potentially thousands of other attributes that can impact a stock price in some way or another. Due to the large number of factors affecting the price, many traders usually only focus on a subset of specific attributes pertaining to an asset, in order to help reduce the complexity needed for their strategies. While no subset is completely capable of predicting all price movements, they can be used to create strategies that work under certain conditions.

The two largest methods of analysis used during trading are fundamental and technical analysis. Fundamental analysis is focused on the use of detailed data obtained from a company's financial statements as well as an array of other economic factors to predict future earnings (Abarbanell and Bushee 1997). This type of analysis is generally focused on long-term trading. While fundamental analysis tends to dive deep into the public data for a company, technical analysis focuses on studying price movements and chart patterns to help detect trends, reversal patterns and other types of technical signals (Edwards, Magee, and Bassetti 2018). Technical analysis tends be to more reliable when used for short-term profits through intra-day and swing trading. While it can also be used for long-term trading, often it needs to be paired with other forms of analysis to be the most effective. This is largely due to the potential impact of external factors that cannot be foreseen through pure technical analysis.

In the last couple of decades, machine learning has been applied to various areas related to trading within financial markets. Many of the original uses of machine learning within the stock market primarily focus on trying to forecast the price of a stock for a certain number of steps into the future (Bao, Lu, and Zhang 2004), (Niaki and Hoseinzade 2013). Typically, these models are then used as a basis for determining when to buy or sell a stock. Unfortunately, they are only somewhat accurate and their forecasts are only stay valid for a short period of time into the future due to unforeseen factors that they do not consider. Additionally, these methods tend to perform badly when it comes to generalization, performing badly when used against stocks that they are not trained on.

More recently, Deep Reinforcement Learning (DRL) has started to become more popular in the realm of machine learning, being used to tackle complex problems. One of the most well known uses of DRL is Google's Alpha Zero Chess Agent that is capable of beating the best chess players in the world (Silver et al. 2018). Outside of the realm of games, researchers have begun applying DRL to the financial realm for problems such as asset trading. Unlike other machine learning approaches related to asset trading such as

price forecasting, DRL has the goal of trying to learn effective trading strategies.

Currently, more than 80 percent of all trading volume executed within the stock market is a result of automated trading. Therefore, the demand for more accurate and capable automated trading methods is at an all time high. In this paper, we propose the use of an Ensemble-based Deep Reinforcement Learning approach to perform automated signal trading in the US Stock market. We compare our results with several baseline strategies and two current state-of-the-art stock trading DRL approaches for stock trading.

Related Work

Due to the potential profits that can be made from asset trading, there has been a vast amount of research put towards applying machine learning to further improve the outcome. Dash and Dash (2016) created a trading framework that combined the use of technical analysis and machine learning to generate buy and sell trading signals. Their approach starts by analyzing stock data and calculating various common technical indicators on the provided data. These indicators are then used as the input features for a Functional Link Artificial Neural Network (FLANN), that is trained to output a value between 0 and 1. Depending on the current stock trend and the user's current position in a stock, the output is used to determine when to buy, sell or hold a stock. Their methods proved effective, beating out other traditional and machine learning trading stock trading methods with a profit of 47 and 24 percent respectively when tested against the BSE SENSEX and SP500 across a period of 250 days.

Taking this concept one step further, Deng et al. (2017) proposed an approach for financial signal representation and trading using Deep RL to automate the trading process. They take a fuzzy learning approach that utilizes a Convolutional Neural Network (CNN) and Long Short-Term Memory (LTSM) network. To drive the learning process, they utilize the Sharpe Ratio as a reward signal. The approach obtained good results, outperforming other current known DRL trading strategies while also increasing its trading efficiency, resulting in a significantly smaller number of performed trades.

In early 2020, Wu et al. (2020) proposed two DRL models for adaptive stock trading. Their first model is referred to as a Gated Deep Q-Network (GDQN). The GDQN couples a GRU and a deep Q-network for adaptive stock trading using a critic-only format. Their second model, known as Gated Deep Policy Gradient (GDPG), adapts the use of a GRU with the Deep Deterministic Policy Gradient (DDPG) approach. For both of their models, the GRU is used to extract a set of features from the stock data which is then used to represent the state of an asset. Their action space includes buying, selling, shorting, covering and holding. Both models were able to beat the current state-of-the-art strategy known as the Advanced Turtle strategy (Vezeris et al. 2019), with the GDPG approach obtaining the most stable results.

Data

There is a large amount of market data that is available in relation to the US stock market and related companies, dating back several decades. This data includes traditional price and volume data, sentiment analysis data, company earning reports and much more. In this paper, we look primarily at historical price data alongside calculated technical indicators. This data can be obtained from numerous different sources including Quandle, FinnHub and Yahoo Finance at varying different time intervals.

In order to help generalize our agent, we trade across a variety of different stocks. We chose to train and test our agent against the top 10 stocks allocated within the Dow Jones Industrial Average (DJIA). This includes: Apple (AAPL), American Express (AXP), Boeing (BA), Costco (CSCO), General Electric (GE), Home Depot (HD), IBM (IBM), Microsoft (MSFT), Nike (NKE) and Walmart (WMT). For training, all 10 stocks are used within a window of six years between January 1st, 2008 and December 31st, 2015. For testing, the same stocks are used for the subsequent two year window between January 1st, 2016 and December 20th, 2018. These time ranges are selected for two reasons: they provide a wide variety of different types of scenarios and they allow for a simplified one-to-one comparison with Wu et al.'s current state-of-the-art RL approaches that utilize a subset of our stocks within the same time frame.

Methodology

To form the problem of asset trading into a process that reinforcement learning can tackle, we have to outline the environment that the agent(s) will be trading in as well as formulate the problem into a Markov Decision Process that a Reinforcement learning agent can learn.

Environment

For the environment, we propose the use of a simple environment that simulates a financial market using historical data. This gives us the ability to simulate trading within different types of markets including the stock, forex and crypto markets, as well as control the step size of the environment. To ensure a level of simulation fidelity within the environment, closer to an actual financial market, the environment will include a 1 percent transaction fee for all all trades, and will simulate realistic trade executions by adding in a random delay between when a trade occurs and the agent obtains the asset.

Problem Formulation

In order to outline the problem of asset trading as a Markov Decision Process, we need to define the notion of state or observation, actions, and rewards to be used within our stock market environment.

Observation The total number of factors that can impact a stock's price at a given point in time within the the US stock exchange is astronomically high. It includes many different categories of information including price history, transaction history, public sentiment data, company financial statements, overall market trends, political happenings and much

more. Unfortunately, it is not possible to capture the entire state of the stock market for an agent to observe. Therefore, we look to the idea of an observation, which is a partial representation of the environment that is visible to the agent at a given point in time within the environment. For each timestep, the a partial observation is obtained using the following 27 attributes:

- **Account Information**
  - Balance
  - Previous Balance
  - Shares Held
- **Stock Attributes**
  - OCHLV: Open, Close, High, Low, Volume
  - RSI (Relative Strength Index): window=14
  - MACD: macd, signal, diff
  - BBAND (Bollinger Bands): Low, High, MAvg, Perc
  - MFI (Money Flow Index): window=14
  - STC (Schaff Trend Cycle)
- **Stock Signals**
  - MFI: Low Threshold, High Threshold)
  - MACD: Crossover
  - BBAND: Crossover, Low Bandwidth, High Bandwidth
  - Moving Averages: Golden Cross, Death Cross
  - STC: Crossover

To provide the agent with a notion of history, we provide the agent with the current partial observation vector alongside vectors from previous 9 timesteps. This history of vectors is stacked together to create the complete observation provided to the agent at each timestep, which is represented as a 27 by 10 matrix.

Actions To avoid over complicating the trading process and entering the realm of margin accounts, we will only allow the agent to only buy, sell and hold shares. When buying a stock, the agent will have the ability to use between 0 and 100 percent of its available cash to purchase shares of an asset at a given point in time. Similarly, the agent will be able to sell between 0 and 100 percent of its current available shares. This can be represented as a continuous action spaces with a range of [-1, 1].

With this scheme, the values between -1 and 1 are divided up into three sections, one for each possible action. While testing, we found that if the sections were not equal, i.e. we chose too small of a region for any particular action, then the agents would have a harder time learning to perform that action. Therefore, we divide the region into three equal sections ranging from -1 to -0.33, -0.33 to 0.33 and 0.33 to 1.0.

When an agent picks an action value between -0.33 and 0.33, the agent holds and does nothing. There is no discernible difference between any values within this region. If an agent picks, a value v between 0.33 and 1, then the value will be re-scaled between zero and one to represent the percent of its equity that it should use towards purchasing shares of the current stock. For a value $v \exists (0.33, 1]$, the scaled percentage is calculated using the following equation:

$$P(x) = \frac{(1 - v)}{1.0 - 0.33} \tag{1}$$

Similarly, when the agent picks an action value between -1.0 and -0.33, the agent will scale this value between zero and one, and use it to sell a percentage of its held stock. For both the buy and sell actions, if the agent tries to perform an action and does not have enough equity or stock to perform the action, then the action is ignored.

Rewards There are many different methods that could be used for formulating the notion of rewards depending on the goal you are trying to reach. Often many papers tend to focus on the use of returns for rewarding the agent. While this has been shown to lead to positive results, we will be more focused on a form of risk-adjusted return. The idea, is that by using a risk-adjusted return, it will help to reduce the overall amount of risk that the agent takes when trading an asset. For this purpose, we will use a metric known as the Sortino Ratio, a variant of the Sharpe Ratio that is primarily only concerned with the downside risk and volatility.

The Sortino Ratio is calculated using the following formula, where x is the invested amount, r_x is the average rate of return for x, R_f is the risk-free rate of return and D is the square root of target semi-variance of r_x, also known as downside deviation.

$$SR(x) = \frac{(r_x - R_f)}{D(r_x)} \tag{2}$$

Model Approach

For this paper, we propose the use an ensemble based reinforcement learning approach for tackling our problem. The approach can utilize any number of sub-agents, each trained separately on the same set of market assets. For each timestep, each agent produces an action. These actions are then combined to determine which action is best to take for that timestep. A visual representation of the model can be seen outlined in Figure 1 below.

For this paper, we use four different agents within our ensemble. Each agent is outlined in the following sections.

Agent 1 - MLP Based PPO agent

The first two agents both utilize the same actor-critic based approach known as proximal policy optimization (PPO) (Schulman et al. 2017). The idea with the PPO RL approach is to try to ensure that an update to the policy does not deviate too far away from previous policy. This helps to ensure that the agent's performance continues to increase throughout training. For this paper, we use a PPO approach that combines notions from both the PPO-Penalty and PPO-Clip variants. For the this agent, the policy function is represented as a simple Multi-Layer Perceptron (MLP) with two-hidden layers.

Agent 2 - Transformer Based PPO agent

The second agent used, also utilizes the PPO algorithm for training, but contains a transformer-based policy function instead of a simple MLP. Transformers have become increas-

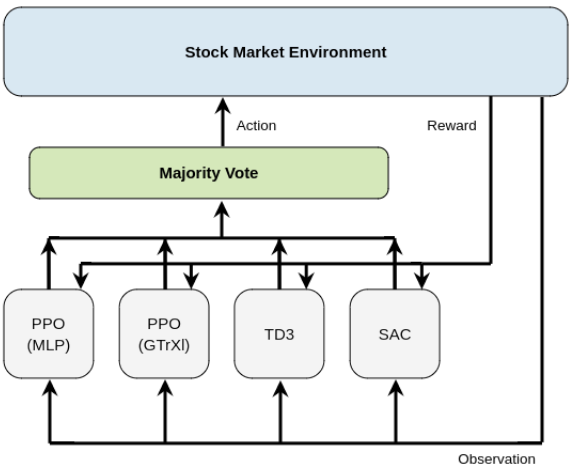

Figure 1: Multi-Agent Ensemble Model

ingly more popular in recent years with their ability to perform exceptionally well in tasks related to time-sequences data. We chose a transformer due to the crucial importance of maintaining knowledge across a long time-horizon within reinforcement learning, as well as for stock trading. We utilize a transformer variant known as the Gated Transformer XL (GTrXL) (Parisotto et al. 2020) which is specifically adapted for use within Reinforcement Learning.

Agent 3 - MLP Based TD3 agent

While the first two agents utilize an on-policy learning approach, the third and forth agents take a different approach by utilizing off-policy learning methods. For the third agent, we use a method known as Twin Delayed Deep Deterministic Policy Gradients (TD3) (Fujimoto, Hoof, and Meger 2018), the newest member to the Deep Deterministic Policy Gradient (DDPG) family. DDPG works by combining the Q-Learning and Policy gradient methods. It concurrently trains a Q-function and a policy function. The Q function is trained through off-policy learning and then further utilized to train the Q-function. The TD3 method extends this notion by adding double Q-learning, delayed updates and policy smoothing. Similar to agent one, this agent uses a MLP for the underlying network and utilizes the same feature abstraction layers between the policy and Q-function networks.

Agent 4 - MLP Based SAC agent

The final agent that we include in the ensemble uses a method known as Soft-Actor Critic (SAC) (Haarnoja et al. 2018). The SAC algorithm is an actor-critic approach that combines off-policy updates with the stochastic formulation of traditional actor-critic approaches. The SAC approach works by maximizing both expected reward and entropy. It has been shown to be extremely effective in the domain of continuous control. Similarly to agents one and three, the SAC agent utilizes a MLP network for the underlying policy and Q-function estimators.

Ensemble Voting

In order to determine the best action to take for a given timestep, the agent ensemble uses a majority voting scheme. The idea is broken down into two parts. The first part, is to determine which action the agent should take. We do this by counting the number of agents that want to buy, sell or hold stock. The action with the most votes is chosen as the resulting action. If voting results in a tie, then the agent ensemble will hold onto its stock and do nothing. Once the action is determined, we determine the quantity of stock we wish to trade by averaging the action values obtained for each agent that chose the winning action. All values suggested by losing agents are ignored.

Evaluation

To determine the effectiveness of each approach, we utilize commonly used metrics for evaluating stock trading performance paired with a comparison to several baseline trading strategies as well as a current state-of-the-art RL approach for stock trading.

Metrics

For asset trading, there are a number of commonly used metrics that are often used to measure the performance of a trading strategy. In this paper, we use the following metrics for evaluating our agent's performance:

- **Cumulative Return**: The percentage of profit or loss obtained over a period of time.
- **Sortino Ratio**: Risk adjusted return, only focused on downside volatility

Baseline Comparison

We compare the the agent ensemble's performance against multiple different baseline strategies. The first baseline strategy that we compare the agent against is the Buy and Hold (B&H) strategy. In this strategy, the maximum amount of an individual stock is purchased on Day-1 and then held for the entire length of the test period.

The second strategy, while closely related, is performing the Buy and Hold strategy specifically against the Dow Jones Industrial Average. This is commonly used to represent a medium risk portfolio allocation strategy.

Furthermore, the final comparison used is the min-variance strategy. Minimum-Variance is a diversified portfolio strategy that consists of trading a weighted collection of individual securities, that ultimately result in a low risk strategy with a relatively good rate of return. To adequately compare with our agent ensemble as well as the other baseline strategies, we use the Min-Variance strategy against the stocks represented within the Dow Jones Industrial index.

State-of-the-Art

In addition to baseline strategies, we compare our results to two state-of-the-art RL approaches used for stock trading: Gated Deep Q-Network (GDQN) and Gated Deterministic Policy Gradient (GDPG) introduced by Wu et al. (2020). For the US stock exchange, these approaches were focused

	Ensemble		B&H		Wu et. al	(GDQN)	Wu et. al	(GDPG)
Symbol	SR	Ret (%)	SR	Ret (%)	SR	Ret (%)	SR	Ret (%)
AAPL	**1.65**	74.31	1.01	56.4	1.02	77.7	1.30	**82.0**
AXP	**1.53**	**60.7**	0.86	46.8	0..39	20.0	0.51	24.3
BA	**2.90**	**181.1**	1.75	144.9	-	-	-	-
CSCO	**2.37**	**105.8**	1.29	77.5	0.31	20.6	0.51	24.3
GE	-2.26	-70.5	-1.86	-73.2	-0.13	-10.8	**-0.22**	**-6.39**
HD	**1.65**	**56.4**	0.89	38.8	-	-	-	-
IBM	**0.35**	**7.37**	-0.01	-6.3	0.07	4.63	0.05	2.55
MSFT	**2.56**	**116.7**	1.49	95.8	-	-	-	-
NKE	**0.91**	**32.7**	0.62	23.6	-	-	-	-
WMT	**1.81**	**64.3**	1.24	62.0	-	-	-	-

Table 1: Sortino Ratio (SR) and Return (Ret) comparing B&H, our Ensemble agent and Wu et. al.'s GDQN / GDPG models

	AAPL	AXP	BA	CSCO	GE	HD	IBM	MSFT	NKE	WMT	Overall
PPO-MLP	1.371	**1.662**	2.120	1.925	**-2.01**	1.539	0.379	2.137	0.749	1.707	1.158
PPO-GTrXL	1.570	1.524	2.798	2.271	-2.340	1.624	0.308	2.442	0.893	1.711	1.280
TD3-MLP	1.592	1.424	**2.990**	2.255	-2.343	1.320	**0.687**	2.178	**1.041**	1.677	1.282
SAC-MLP	1.588	1.472	2.750	2.262	-2.348	1.594	0.326	2.454	0.884	1.725	1.270
Ensemble	**1.651**	1.526	2.902	**2.365**	-2.257	**1.649**	0.353	**2.560**	0.910	**1.809**	**1.347**

Table 2: Sortino Ratios comparison between the individual agents and the collective ensemble agent

on the following stock tickers: APPL, AXP, CSCO, GE, and IBM, each of which are contained within our test suite. Since we utilize the same stocks as well as the same date-ranges for training and testing, it allows for a direct one-to-one comparison against the GDQN and GDPG approaches.

Results

Each of ensemble's base agents are trained against a two year test period, with a fine-tuned number of timesteps for each agent. The number of training timesteps required for each agent ranges from 40000 to 200000, depending on the agent's algorithm and neural network type. After training, each agent is added to an ensemble and tested against each of the top-10 stock allocations within the Dow Jones Industrial Index. To minimize outliers and get a better look into the variability of our approach, we test the agent ensemble against each stock 10 times and obtain the average for the overall return and Sortino Ratio.

In the first experiment, we compare the Sortino Ratio and returns obtained by our agent while holding equal portions of each test stock, and compare them with the DJIA and Min-Variance baseline strategies. This can be seen below in Table 3.

Strategy	SR	Ret (%)
Ensemble	1.619	71.15
Averaged B&H	0.728	46.61
DJIA	0.97	34.48
Min-Variance	0.56	15.44

Table 3: Comparison with Baseline Strategies

Overall, our ensemble performed signficantly better than each of the baseline strategies, obtaining a higher average Sortino ratio and overall return for all but one stock. For this particular set of test data, the returns obtained by min-variance strategy were very small. This suggests that there was a high amount of variance in the stock prices within the DJIA index between 2016 and 2018.

Next, we compare the results of our agent ensemble against the Buy and Hold strategy, and Wu et al's GDQN and GDPG approaches mentioned previously. The overall returns and Sortino Ratios obtained for each strategy against each stock can be seen below in Table 1. Our agent was able to outperform the B&H strategy for both Sortino Ratio and overall returns for all stocks except GE, where our agent incurred a larger loss and lower Sortino Ratio.

When compared against Wu et. al's GDQN and GDPG, our agent-ensemble outperformed each approach with respect to Sortino Ratio for all stocks except GE, where Wu et al's approach excelled due to its ability to short stocks, greatly beating out both ours and the B&H strategy. For overall return, our agent ensemble surpassed the GDQN and QDPG for AXP, CSCO AND IBM, while obtaining lower returns for AAPL and GE. Interesting, for AAPL, we obtain a much higher Sortino Ratio, while obtaining a slightly lower return than both of Wu et al.'s approaches. This suggests that the GDQN and GDPG are using a signficantly riskier trading strategy for this stock.

In addition to comparing our approach with other strategies, we also look at the effectiveness of our ensemble strategy in conjunction with the performance of the agents that it encapsulates. In order to do this, we compare the Sortino Ratios obtained by each agent within the ensemble against the Sortino Ratio obtained by the overall ensemble. These results can be seen in Table 2. Our results show that our ensemble agent obtained the highest Sortino Ratio for five of the 10 test stocks, more than any individual agent was able

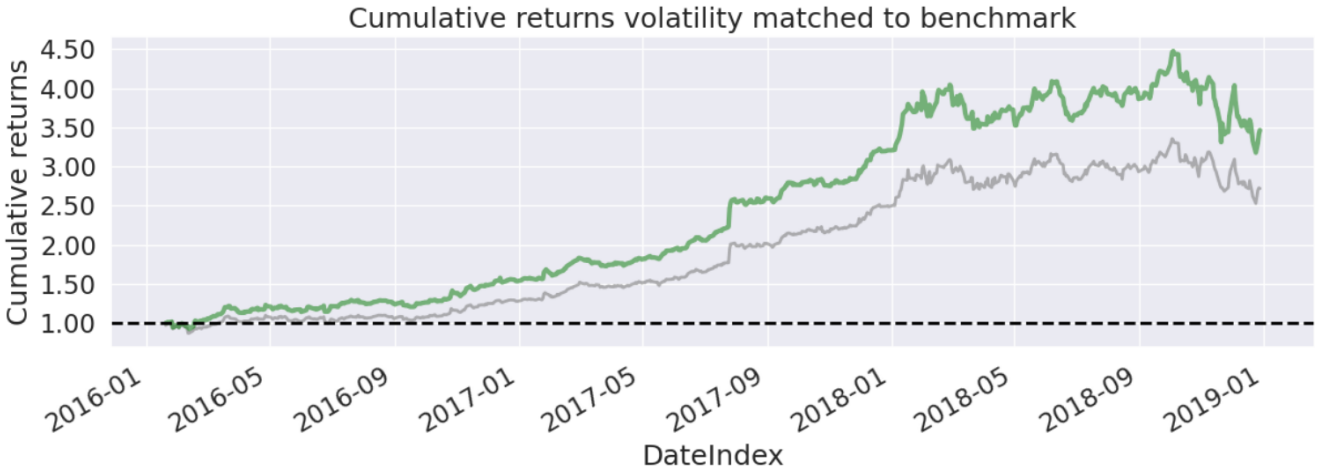

Figure 2: BA Cumulative Returns - Ensemble Agent (Green) vs B&H (Grey)

to obtain. Furthermore, it also obtained the highest average Sortino ratio.

Discussion

The results of our agent-ensemble are very promising, beating out the Buy and Hold strategy for all stocks except GE. The agent was able to greatly surpass both the returns and Sortino Ratios for both the minimum-variance and DJIA index results, two of the most common baseline strategies used to represent low and medium risk trading strategies.

Outside of test strategies, we compared our results with the GDQN and GDPG approaches introduced by Wu et al GDQN and GDPG. Overall, our agent performed very well in comparison, beating out these approaches for four out of five stocks with respect to Sortino Ratio, while only under performing for the GE stock. Looking closer at GE, it easy to see why our approach performed signficantly worse. Between 2016 and 2018, GE was in a strong downward trend, which provided our agent with very little opportunity to gain any profit. Wu et al.'s approaches on the other hand is able to short stocks, which gives them a strong advantage by easily being able to make a profit against decreasing stock prices.

Looking over the results obtained by our agent ensemble, there were numerous interesting observations that we found. The first and most obvious observation, is that the ensemble agent often does not sell off its stock when the price is plummeting. While the agent-ensemble will rarely buy during these scenarios, it will also rarely sell either unless a major indicator such as the Death Cross has been seen. This causes the agent to hold onto the stock during periods of huge loss, reducing the overall potential for larger profits. This causes the agent to perform very poorly against stocks that are in a general downward trend. A huge example of this is with GE, where our agent ensemble fails to adequately gain any traction and it causes the agent to incur major losses. The agents inability to trade against stocks like GE, is most likely due to the fact that all of the stocks used to train the agent are picked from the DJIA index, which mostly primarily consists of well-performing positive trending stocks. Therefore, the data is skewed to favor these types of trends which will cause the agent to have very little experience when training against negative stock trends. To combat this type of behavior in the future, it would be beneficial to include more generally negative trending stocks within the training data in order to help balance out the data.

While the agent tended to incur the large majority of its losses during major downtrends, we found that the majority of our agent's profits were obtained during periods where the stock price was oscillating between minor upswings and downswings in price. Looking closer, we found that our agent often made short-term trades during these periods of time, buying towards the bottom of a downswing, and selling a few days later for a small profit. This phenomenon explains how the agent was able to surpass B&H profits for stocks that had few periods of downward movement. An example where this occurs, is with the Boeing stock (BA). While the stock price grew over 200 percent alone, the agent was still able to achieve a higher return, with the cumulative return of the agent slowly deviating higher over time than that of the B&H strategy. This can be seen in Figure 2 below.

In addition, we found that the agents are extremely sensitive with respect to their hyper-parameters. For example, a very small change to the learning rate or number of timesteps could be the difference between the agent performing very well, and not learning at all. On top of this, each agent in the ensemble has a multitude of algorithm-specific hyper-parameters, most capable of having a major impact on training. When training multiple agents in parallel, this makes it extremely tricky to find optimal parameters for each agent. We believe the performance of our agent ensemble could be improved further by additional fine-tuning of each of the underlying agents.

Future

To further improve the results of our approach, there are a few steps that we could take in future research. First, since there are multiple different types of models that utilize different types of RL algorithms, it was extremely hard to fine

tune each one. It would be worth while performing a more rigorous hyper-parameter search for each model individually, before combining them into a single ensemble agent.

In addition to fine-tuning, another area for consideration would be try different method for combining the sub-agents preferred actions. For this paper, we used a majority voting scheme, which seemed to work well, but there are other methods that could be used that might increase performance further. One method in particular that we would like to research, is the use of a weighted voting scheme that allows for the assignment of weights to each agent in the ensemble based on a post-training evaluation phase. These weights could then be used to put additional emphasis on agents that perform better individually, while not completely discounting the weaker agents. Furthermore, for stocks specifically, it is possible that a weighted scheme could be used at on a per-stock or per-trend level as well. This would help to exploit the agents' strengths and weaknesses more carefully.

Finally, it would be crucial to improve the agents performance against stocks that are in a general downward trend. While are agent's poor performance against these stocks was most likely due to unbalanced stock data, this could potentially be avoided altogether by allowing the agent to short and cover stocks. This would allow the agent to be able to profit from negatively trending stocks, and ultimately make it more versatile towards any type of situation.

Conclusion

Over the past several decades, automated trading accounts for the majority of all trading volume within the US stock exchange. Automated trading strategies usually involve a signficant amount of hard-coded conditional logic that only works in certain scenarios. We introduced an ensemble-based Deep Reinforcement Learning approach aimed towards generalizing automated stock trading with the goal of maximizing potential risk-adjusted return. Our agent ensemble was trained and tested against the 10 highest allocated stocks within the DJIA index. The agent obtained significantly better Sortino Ratios for nine out of the 10 stocks when tested against baseline strategies. Furthermore, our agent was able to out perform two state-of-the-art DRL models across four out of five stocks listed within the DJIA with respect to risk-adjusted return.

References

Abarbanell, J. S., and Bushee, B. J. 1997. Fundamental analysis, future earnings, and stock prices. *Journal of Accounting Research* 35(1):1–24.

Bao, Y.; Lu, Y.; and Zhang, J. 2004. Forecasting stock price by svms regression. In Bussler, C., and Fensel, D., eds., *Artificial Intelligence: Methodology, Systems, and Applications*, 295–303. Berlin, Heidelberg: Springer Berlin Heidelberg.

Dash, R., and Dash, P. K. 2016. A hybrid stock trading framework integrating technical analysis with machine learning techniques. *The Journal of Finance and Data Science* 2(1):42–57.

Deng, Y.; Bao, F.; Kong, Y.; Ren, Z.; and Dai, Q. 2017. Deep Direct Reinforcement Learning for Financial Signal Representation and Trading. *IEEE Transactions on Neural Networks and Learning Systems* 28(3):653–664.

Edwards, R. D.; Magee, J.; and Bassetti, W. C. 2018. *Technical Analysis of Stock Trends*. CRC press.

Fujimoto, S.; Hoof, H.; and Meger, D. 2018. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, 1587–1596. PMLR.

Haarnoja, T.; Zhou, A.; Abbeel, P.; and Levine, S. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, 1861–1870. PMLR.

Niaki, S., and Hoseinzade, S. 2013. Forecasting SP 500 Index using Artificial Neural Networks and Design of Experiments. *Journal of Industrial Engineering International* 9:1–9.

Parisotto, E.; Song, F.; Rae, J.; Pascanu, R.; Gulcehre, C.; Jayakumar, S.; Jaderberg, M.; Kaufman, R. L.; Clark, A.; Noury, S.; et al. 2020. Stabilizing transformers for reinforcement learning. In *International Conference on Machine Learning*, 7487–7498. PMLR.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. *ArXiv* abs/1707.06347.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science* 362(6419):1140–1144.

Vezeris, D.; Karkanis, I.; Kyrgos, T.; and Lv, K. 2019. Adturtle: An Advanced Turtle Trading System.

Wu, X.; Chen, H.; Wang, J.; Troiano, L.; Loia, V.; and Fujita, H. 2020. Adaptive stock trading strategies with deep reinforcement learning methods. *Information Sciences* 538:142 – 158.

Final CS 5080 Fall 2021

Automated Signal Trading using Ensemble Reinforcement Learning

Daniel Boyle and Jugal Kalita
1420 Austin Bluffs Pkwy
Colorado Springs, CO 80919

Abstract

Deciding when to buy, sell and hold an acquired asset is an extremely complex task. Within financial markets, there are a countless number of possible stimuli that can affect the price of an asset at any given point in time. In this paper, we propose using an ensemble-based Deep Reinforcement Learning (DRL) approach for automated stock market trading. The model is trained and tested against the top-10 stock allocations within the Dow Jones Industrial Average (DJIA). Our agent ensemble outperformed the B&H and Min-Variance baseline strategies for all but one stock, obtaining an average Sortino Ratio of 1.620 across the tested stocks. Additionally, our agent outperforms two state-of-the-art DRL trading approaches, obtaining higher Sortino Ratios and returns for three out of five stocks tested within the DJIA.

Introduction

Determining the optimal time to buy and sell an asset within a financial market is a topic that has been constantly sought after for decades. If answered, it could open the door for generating a large amount of wealth in a relatively short period of time and ultimately would revolutionize the world of trading. While at first glance it may not seem hard, it is an extremely complicated task, even for the most advanced stockbrokers. Very often, experts are in disagreement on which methods are capable of producing the greatest results.

The complexity of trading stems from the vast number of potential stimuli that can impact a financial market, making it virtually impossible to accurately predict the future price of an asset. For example, in the stock market, the price of a company's stock can be affected by numerous business attributes and statistics including but not limited to the companies profits, losses, products sold, research interests, and currently held contracts. On top of these, there are many external factors that can have a large impact on a stock including the overall trend of the entire market and public sentiment created through platforms such as the news and social media. Additionally, there are potentially thousands of other attributes that can impact a stock price in some way or another. Due to the large number of factors affecting the price, many traders usually only focus on a subset of specific attributes pertaining to an asset, in order to help reduce the complexity needed for their strategies. While no subset is completely capable of predicting all price movements, they can be used to create strategies that work under certain conditions.

The two largest methods of analysis used during trading are fundamental and technical analysis. Fundamental analysis is focused on the use of detailed data obtained from a company's financial statements as well as an array of other economic factors to predict future earnings (Abarbanell and Bushee 1997). This type of analysis is generally focused on long-term trading. While fundamental analysis tends to dive deep into the public data for a company, technical analysis focuses on studying price movements and chart patterns to help detect trends, reversal patterns and other types of technical signals (Edwards, Magee, and Bassetti 2018). Technical analysis tends be to more reliable when used for short-term profits through intra-day and swing trading. While it can also be used for long-term trading, often it needs to be paired with other forms of analysis to be the most effective. This is largely due to the potential impact of external factors that cannot be foreseen through pure technical analysis.

In the last couple of decades, machine learning has been applied to various areas related to trading within financial markets. Many of the original uses of machine learning within the stock market primarily focus on trying to forecast the price of a stock for a certain number of steps into the future (Bao, Lu, and Zhang 2004), (Niaki and Hoseinzade 2013). Typically, these models are then used as a basis for determining when to buy or sell a stock. Unfortunately, they are only somewhat accurate and their forecasts are only stay valid for a short period of time into the future due to unforeseen factors that they do not consider. Additionally, these methods tend to perform badly when it comes to generalization, performing badly when used against stocks that they are not trained on.

More recently, Deep Reinforcement Learning (DRL) has started to become more popular in the realm of machine learning, being used to tackle complex problems. One of the most well known uses of DRL is Google's Alpha Zero Chess Agent that is capable of beating the best chess players in the world (Silver et al. 2018). Outside of the realm of games, researchers have begun applying DRL to the financial realm for problems such as asset trading. Unlike other machine learning approaches related to asset trading such as

price forecasting, DRL has the goal of trying to learn effective trading strategies.

Currently, more than 80 percent of all trading volume executed within the stock market is a result of automated trading. Therefore, the demand for more accurate and capable automated trading methods is at an all time high. In this paper, we propose the use of an Ensemble-based Deep Reinforcement Learning approach to perform automated signal trading in the US Stock market. We compare our results with several baseline strategies and two current state-of-the-art stock trading DRL approaches for stock trading.

Related Work

Due to the potential profits that can be made from asset trading, there has been a vast amount of research put towards applying machine learning to further improve the outcome. Dash and Dash (2016) created a trading framework that combined the use of technical analysis and machine learning to generate buy and sell trading signals. Their approach starts by analyzing stock data and calculating various common technical indicators on the provided data. These indicators are then used as the input features for a Functional Link Artificial Neural Network (FLANN), that is trained to output a value between 0 and 1. Depending on the current stock trend and the user's current position in a stock, the output is used to determine when to buy, sell or hold a stock. Their methods proved effective, beating out other traditional and machine learning trading stock trading methods with a profit of 47 and 24 percent respectively when tested against the BSE SENSEX and SP500 across a period of 250 days.

Taking this concept one step further, Deng et al. (2017) proposed an approach for financial signal representation and trading using Deep RL to automate the trading process. They take a fuzzy learning approach that utilizes a Convolutional Neural Network (CNN) and Long Short-Term Memory (LTSM) network. To drive the learning process, they utilize the Sharpe Ratio as a reward signal. The approach obtained good results, outperforming other current known DRL trading strategies while also increasing its trading efficiency, resulting in a significantly smaller number of performed trades.

In early 2020, Wu et al. (2020) proposed two DRL models for adaptive stock trading. Their first model is referred to as a Gated Deep Q-Network (GDQN). The GDQN couples a GRU and a deep Q-network for adaptive stock trading using a critic-only format. Their second model, known as Gated Deep Policy Gradient (GDPG), adapts the use of a GRU with the Deep Deterministic Policy Gradient (DDPG) approach. For both of their models, the GRU is used to extract a set of features from the stock data which is then used to represent the state of an asset. Their action space includes buying, selling, shorting, covering and holding. Both models were able to beat the current state-of-the-art strategy known as the Advanced Turtle strategy (Vezeris et al. 2019), with the GDPG approach obtaining the most stable results.

Data

There is a large amount of market data that is available in relation to the US stock market and related companies, dating back several decades. This data includes traditional price and volume data, sentiment analysis data, company earning reports and much more. In this paper, we look primarily at historical price data alongside calculated technical indicators. This data can be obtained from numerous different sources including Quandle, FinnHub and Yahoo Finance at varying different time intervals.

In order to help generalize our agent, we trade across a variety of different stocks. We chose to train and test our agent against the top 10 stocks allocated within the Dow Jones Industrial Average (DJIA). This includes: Apple (AAPL), American Express (AXP), Boeing (BA), Costco (CSCO), General Electric (GE), Home Depot (HD), IBM (IBM), Microsoft (MSFT), Nike (NKE) and Walmart (WMT). For training, all 10 stocks are used within a window of six years between January 1st, 2008 and December 31st, 2015. For testing, the same stocks are used for the subsequent two year window between January 1st, 2016 and December 20th, 2018. These time ranges are selected for two reasons: they provide a wide variety of different types of scenarios and they allow for a simplified one-to-one comparison with Wu et al.'s current state-of-the-art RL approaches that utilize a subset of our stocks within the same time frame.

Methodology

To form the problem of asset trading into a process that reinforcement learning can tackle, we have to outline the environment that the agent(s) will be trading in as well as formulate the problem into a Markov Decision Process that a Reinforcement learning agent can learn.

Environment

For the environment, we propose the use of a simple environment that simulates a financial market using historical data. This gives us the ability to simulate trading within different types of markets including the stock, forex and crypto markets, as well as control the step size of the environment. To ensure a level of simulation fidelity within the environment, closer to an actual financial market, the environment will include a 1 percent transaction fee for all all trades, and will simulate realistic trade executions by adding in a random delay between when a trade occurs and the agent obtains the asset.

Problem Formulation

In order to outline the problem of asset trading as a Markov Decision Process, we need to define the notion of state or observation, actions, and rewards to be used within our stock market environment.

Observation The total number of factors that can impact a stock's price at a given point in time within the the US stock exchange is astronomically high. It includes many different categories of information including price history, transaction history, public sentiment data, company financial statements, overall market trends, political happenings and much

more. Unfortunately, it is not possible to capture the entire state of the stock market for an agent to observe. Therefore, we look to the idea of an observation, which is a partial representation of the environment that is visible to the agent at a given point in time within the environment. For each timestep, the a partial observation is obtained using the following 27 attributes:

- **Account Information**
  - Balance
  - Previous Balance
  - Shares Held
- **Stock Attributes**
  - OCHLV: Open, Close, High, Low, Volume
  - RSI (Relative Strength Index): window=14
  - MACD: macd, signal, diff
  - BBAND (Bollinger Bands): Low, High, MAvg, Perc
  - MFI (Money Flow Index): window=14
  - STC (Schaff Trend Cycle)
- **Stock Signals**
  - MFI: Low Threshold, High Threshold)
  - MACD: Crossover
  - BBAND: Crossover, Low Bandwidth, High Bandwidth
  - Moving Averages: Golden Cross, Death Cross
  - STC: Crossover

To provide the agent with a notion of history, we provide the agent with the current partial observation vector alongside vectors from previous 9 timesteps. This history of vectors is stacked together to create the complete observation provided to the agent at each timestep, which is represented as a 27 by 10 matrix.

Actions To avoid over complicating the trading process and entering the realm of margin accounts, we will only allow the agent to only buy, sell and hold shares. When buying a stock, the agent will have the ability to use between 0 and 100 percent of its available cash to purchase shares of an asset at a given point in time. Similarly, the agent will be able to sell between 0 and 100 percent of its current available shares. This can be represented as a continuous action spaces with a range of [-1, 1].

With this scheme, the values between -1 and 1 are divided up into three sections, one for each possible action. While testing, we found that if the sections were not equal, i.e. we chose too small of a region for any particular action, then the agents would have a harder time learning to perform that action. Therefore, we divide the region into three equal sections ranging from -1 to -0.33, -0.33 to 0.33 and 0.33 to 1.0.

When an agent picks an action value between -0.33 and 0.33, the agent holds and does nothing. There is no discernible difference between any values within this region. If an agent picks, a value v between 0.33 and 1, then the value will be re-scaled between zero and one to represent the percent of its equity that it should use towards purchasing shares of the current stock. For a value $v \exists (0.33, 1]$, the scaled percentage is calculated using the following equation:

$$P(x) = \frac{(1 - v)}{1.0 - 0.33} \tag{1}$$

Similarly, when the agent picks an action value between -1.0 and -0.33, the agent will scale this value between zero and one, and use it to sell a percentage of its held stock. For both the buy and sell actions, if the agent tries to perform an action and does not have enough equity or stock to perform the action, then the action is ignored.

Rewards There are many different methods that could be used for formulating the notion of rewards depending on the goal you are trying to reach. Often many papers tend to focus on the use of returns for rewarding the agent. While this has been shown to lead to positive results, we will be more focused on a form of risk-adjusted return. The idea, is that by using a risk-adjusted return, it will help to reduce the overall amount of risk that the agent takes when trading an asset. For this purpose, we will use a metric known as the Sortino Ratio, a variant of the Sharpe Ratio that is primarily only concerned with the downside risk and volatility.

The Sortino Ratio is calculated using the following formula, where x is the invested amount, r_x is the average rate of return for x, R_f is the risk-free rate of return and D is the square root of target semi-variance of r_x, also known as downside deviation.

$$SR(x) = \frac{(r_x - R_f)}{D(r_x)} \tag{2}$$

Model Approach

For this paper, we propose the use an ensemble based reinforcement learning approach for tackling our problem. The approach can utilize any number of sub-agents, each trained separately on the same set of market assets. For each timestep, each agent produces an action. These actions are then combined to determine which action is best to take for that timestep. A visual representation of the model can be seen outlined in Figure 1 below.

For this paper, we use four different agents within our ensemble. Each agent is outlined in the following sections.

Agent 1 - MLP Based PPO agent

The first two agents both utilize the same actor-critic based approach known as proximal policy optimization (PPO) (Schulman et al. 2017). The idea with the PPO RL approach is to try to ensure that an update to the policy does not deviate too far away from previous policy. This helps to ensure that the agent's performance continues to increase throughout training. For this paper, we use a PPO approach that combines notions from both the PPO-Penalty and PPO-Clip variants. For the this agent, the policy function is represented as a simple Multi-Layer Perceptron (MLP) with two-hidden layers.

Agent 2 - Transformer Based PPO agent

The second agent used, also utilizes the PPO algorithm for training, but contains a transformer-based policy function instead of a simple MLP. Transformers have become increas-

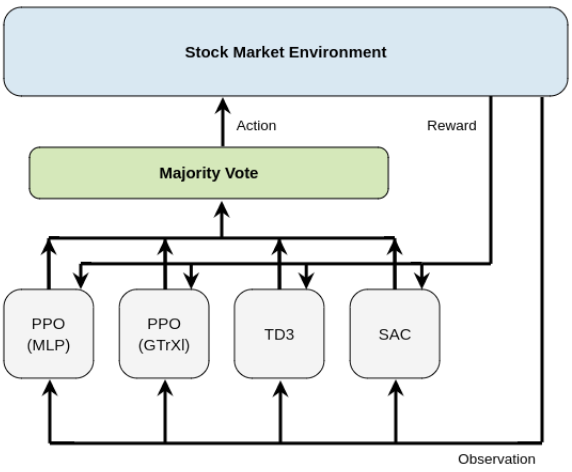

Figure 1: Multi-Agent Ensemble Model

ingly more popular in recent years with their ability to perform exceptionally well in tasks related to time-sequences data. We chose a transformer due to the crucial importance of maintaining knowledge across a long time-horizon within reinforcement learning, as well as for stock trading. We utilize a transformer variant known as the Gated Transformer XL (GTrXL) (Parisotto et al. 2020) which is specifically adapted for use within Reinforcement Learning.

Agent 3 - MLP Based TD3 agent

While the first two agents utilize an on-policy learning approach, the third and forth agents take a different approach by utilizing off-policy learning methods. For the third agent, we use a method known as Twin Delayed Deep Deterministic Policy Gradients (TD3) (Fujimoto, Hoof, and Meger 2018), the newest member to the Deep Deterministic Policy Gradient (DDPG) family. DDPG works by combining the Q-Learning and Policy gradient methods. It concurrently trains a Q-function and a policy function. The Q function is trained through off-policy learning and then further utilized to train the Q-function. The TD3 method extends this notion by adding double Q-learning, delayed updates and policy smoothing. Similar to agent one, this agent uses a MLP for the underlying network and utilizes the same feature abstraction layers between the policy and Q-function networks.

Agent 4 - MLP Based SAC agent

The final agent that we include in the ensemble uses a method known as Soft-Actor Critic (SAC) (Haarnoja et al. 2018). The SAC algorithm is an actor-critic approach that combines off-policy updates with the stochastic formulation of traditional actor-critic approaches. The SAC approach works by maximizing both expected reward and entropy. It has been shown to be extremely effective in the domain of continuous control. Similarly to agents one and three, the SAC agent utilizes a MLP network for the underlying policy and Q-function estimators.

Ensemble Voting

In order to determine the best action to take for a given timestep, the agent ensemble uses a majority voting scheme. The idea is broken down into two parts. The first part, is to determine which action the agent should take. We do this by counting the number of agents that want to buy, sell or hold stock. The action with the most votes is chosen as the resulting action. If voting results in a tie, then the agent ensemble will hold onto its stock and do nothing. Once the action is determined, we determine the quantity of stock we wish to trade by averaging the action values obtained for each agent that chose the winning action. All values suggested by losing agents are ignored.

Evaluation

To determine the effectiveness of each approach, we utilize commonly used metrics for evaluating stock trading performance paired with a comparison to several baseline trading strategies as well as a current state-of-the-art RL approach for stock trading.

Metrics

For asset trading, there are a number of commonly used metrics that are often used to measure the performance of a trading strategy. In this paper, we use the following metrics for evaluating our agent's performance:

- **Cumulative Return**: The percentage of profit or loss obtained over a period of time.
- **Sortino Ratio**: Risk adjusted return, only focused on downside volatility

Baseline Comparison

We compare the the agent ensemble's performance against multiple different baseline strategies. The first baseline strategy that we compare the agent against is the Buy and Hold (B&H) strategy. In this strategy, the maximum amount of an individual stock is purchased on Day-1 and then held for the entire length of the test period.

The second strategy, while closely related, is performing the Buy and Hold strategy specifically against the Dow Jones Industrial Average. This is commonly used to represent a medium risk portfolio allocation strategy.

Furthermore, the final comparison used is the min-variance strategy. Minimum-Variance is a diversified portfolio strategy that consists of trading a weighted collection of individual securities, that ultimately result in a low risk strategy with a relatively good rate of return. To adequately compare with our agent ensemble as well as the other baseline strategies, we use the Min-Variance strategy against the stocks represented within the Dow Jones Industrial index.

State-of-the-Art

In addition to baseline strategies, we compare our results to two state-of-the-art RL approaches used for stock trading: Gated Deep Q-Network (GDQN) and Gated Deterministic Policy Gradient (GDPG) introduced by Wu et al. (2020). For the US stock exchange, these approaches were focused

	Ensemble		B&H		Wu et. al	(GDQN)	Wu et. al	(GDPG)
Symbol	SR	Ret (%)	SR	Ret (%)	SR	Ret (%)	SR	Ret (%)
AAPL	**1.65**	74.31	1.01	56.4	1.02	77.7	1.30	**82.0**
AXP	**1.53**	**60.7**	0.86	46.8	0..39	20.0	0.51	24.3
BA	**2.90**	**181.1**	1.75	144.9	-	-	-	-
CSCO	**2.37**	**105.8**	1.29	77.5	0.31	20.6	0.51	24.3
GE	-2.26	-70.5	-1.86	-73.2	-0.13	-10.8	**-0.22**	**-6.39**
HD	**1.65**	**56.4**	0.89	38.8	-	-	-	-
IBM	**0.35**	**7.37**	-0.01	-6.3	0.07	4.63	0.05	2.55
MSFT	**2.56**	**116.7**	1.49	95.8	-	-	-	-
NKE	**0.91**	**32.7**	0.62	23.6	-	-	-	-
WMT	**1.81**	**64.3**	1.24	62.0	-	-	-	-

Table 1: Sortino Ratio (SR) and Return (Ret) comparing B&H, our Ensemble agent and Wu et. al.'s GDQN / GDPG models

	AAPL	AXP	BA	CSCO	GE	HD	IBM	MSFT	NKE	WMT	Overall
PPO-MLP	1.371	**1.662**	2.120	1.925	**-2.01**	1.539	0.379	2.137	0.749	1.707	1.158
PPO-GTrXL	1.570	1.524	2.798	2.271	-2.340	1.624	0.308	2.442	0.893	1.711	1.280
TD3-MLP	1.592	1.424	**2.990**	2.255	-2.343	1.320	**0.687**	2.178	**1.041**	1.677	1.282
SAC-MLP	1.588	1.472	2.750	2.262	-2.348	1.594	0.326	2.454	0.884	1.725	1.270
Ensemble	**1.651**	1.526	2.902	**2.365**	-2.257	**1.649**	0.353	**2.560**	0.910	**1.809**	**1.347**

Table 2: Sortino Ratios comparison between the individual agents and the collective ensemble agent

on the following stock tickers: APPL, AXP, CSCO, GE, and IBM, each of which are contained within our test suite. Since we utilize the same stocks as well as the same date-ranges for training and testing, it allows for a direct one-to-one comparison against the GDQN and GDPG approaches.

Results

Each of ensemble's base agents are trained against a two year test period, with a fine-tuned number of timesteps for each agent. The number of training timesteps required for each agent ranges from 40000 to 200000, depending on the agent's algorithm and neural network type. After training, each agent is added to an ensemble and tested against each of the top-10 stock allocations within the Dow Jones Industrial Index. To minimize outliers and get a better look into the variability of our approach, we test the agent ensemble against each stock 10 times and obtain the average for the overall return and Sortino Ratio.

In the first experiment, we compare the Sortino Ratio and returns obtained by our agent while holding equal portions of each test stock, and compare them with the DJIA and Min-Variance baseline strategies. This can be seen below in Table 3.

Strategy	SR	Ret (%)
Ensemble	1.619	71.15
Averaged B&H	0.728	46.61
DJIA	0.97	34.48
Min-Variance	0.56	15.44

Table 3: Comparison with Baseline Strategies

Overall, our ensemble performed signficantly better than each of the baseline strategies, obtaining a higher average Sortino ratio and overall return for all but one stock. For this particular set of test data, the returns obtained by min-variance strategy were very small. This suggests that there was a high amount of variance in the stock prices within the DJIA index between 2016 and 2018.

Next, we compare the results of our agent ensemble against the Buy and Hold strategy, and Wu et al's GDQN and GDPG approaches mentioned previously. The overall returns and Sortino Ratios obtained for each strategy against each stock can be seen below in Table 1. Our agent was able to outperform the B&H strategy for both Sortino Ratio and overall returns for all stocks except GE, where our agent incurred a larger loss and lower Sortino Ratio.

When compared against Wu et. al's GDQN and GDPG, our agent-ensemble outperformed each approach with respect to Sortino Ratio for all stocks except GE, where Wu et al's approach excelled due to its ability to short stocks, greatly beating out both ours and the B&H strategy. For overall return, our agent ensemble surpassed the GDQN and QDPG for AXP, CSCO AND IBM, while obtaining lower returns for AAPL and GE. Interesting, for AAPL, we obtain a much higher Sortino Ratio, while obtaining a slightly lower return than both of Wu et al.'s approaches. This suggests that the GDQN and GDPG are using a signficantly riskier trading strategy for this stock.

In addition to comparing our approach with other strategies, we also look at the effectiveness of our ensemble strategy in conjunction with the performance of the agents that it encapsulates. In order to do this, we compare the Sortino Ratios obtained by each agent within the ensemble against the Sortino Ratio obtained by the overall ensemble. These results can be seen in Table 2. Our results show that our ensemble agent obtained the highest Sortino Ratio for five of the 10 test stocks, more than any individual agent was able

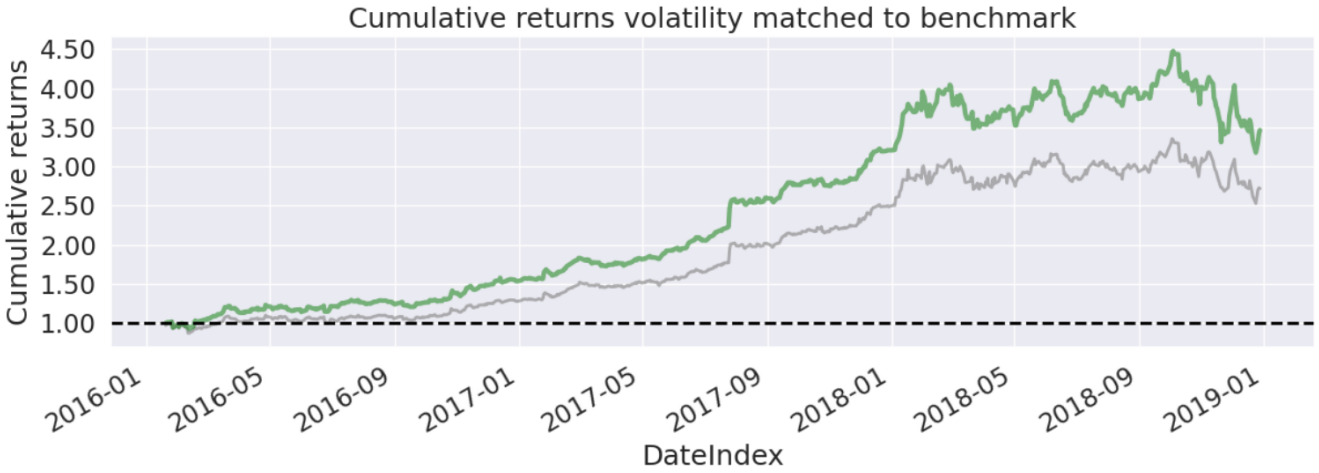

Figure 2: BA Cumulative Returns - Ensemble Agent (Green) vs B&H (Grey)

to obtain. Furthermore, it also obtained the highest average Sortino ratio.

Discussion

The results of our agent-ensemble are very promising, beating out the Buy and Hold strategy for all stocks except GE. The agent was able to greatly surpass both the returns and Sortino Ratios for both the minimum-variance and DJIA index results, two of the most common baseline strategies used to represent low and medium risk trading strategies.

Outside of test strategies, we compared our results with the GDQN and GDPG approaches introduced by Wu et al GDQN and GDPG. Overall, our agent performed very well in comparison, beating out these approaches for four out of five stocks with respect to Sortino Ratio, while only under performing for the GE stock. Looking closer at GE, it easy to see why our approach performed signficantly worse. Between 2016 and 2018, GE was in a strong downward trend, which provided our agent with very little opportunity to gain any profit. Wu et al.'s approaches on the other hand is able to short stocks, which gives them a strong advantage by easily being able to make a profit against decreasing stock prices.

Looking over the results obtained by our agent ensemble, there were numerous interesting observations that we found. The first and most obvious observation, is that the ensemble agent often does not sell off its stock when the price is plummeting. While the agent-ensemble will rarely buy during these scenarios, it will also rarely sell either unless a major indicator such as the Death Cross has been seen. This causes the agent to hold onto the stock during periods of huge loss, reducing the overall potential for larger profits. This causes the agent to perform very poorly against stocks that are in a general downward trend. A huge example of this is with GE, where our agent ensemble fails to adequately gain any traction and it causes the agent to incur major losses. The agents inability to trade against stocks like GE, is most likely due to the fact that all of the stocks used to train the agent are picked from the DJIA index, which mostly primarily consists of well-performing positive trending stocks. Therefore, the data is skewed to favor these types of trends which will cause the agent to have very little experience when training against negative stock trends. To combat this type of behavior in the future, it would be beneficial to include more generally negative trending stocks within the training data in order to help balance out the data.

While the agent tended to incur the large majority of its losses during major downtrends, we found that the majority of our agent's profits were obtained during periods where the stock price was oscillating between minor upswings and downswings in price. Looking closer, we found that our agent often made short-term trades during these periods of time, buying towards the bottom of a downswing, and selling a few days later for a small profit. This phenomenon explains how the agent was able to surpass B&H profits for stocks that had few periods of downward movement. An example where this occurs, is with the Boeing stock (BA). While the stock price grew over 200 percent alone, the agent was still able to achieve a higher return, with the cumulative return of the agent slowly deviating higher over time than that of the B&H strategy. This can be seen in Figure 2 below.

In addition, we found that the agents are extremely sensitive with respect to their hyper-parameters. For example, a very small change to the learning rate or number of timesteps could be the difference between the agent performing very well, and not learning at all. On top of this, each agent in the ensemble has a multitude of algorithm-specific hyper-parameters, most capable of having a major impact on training. When training multiple agents in parallel, this makes it extremely tricky to find optimal parameters for each agent. We believe the performance of our agent ensemble could be improved further by additional fine-tuning of each of the underlying agents.

Future

To further improve the results of our approach, there are a few steps that we could take in future research. First, since there are multiple different types of models that utilize different types of RL algorithms, it was extremely hard to fine

tune each one. It would be worth while performing a more rigorous hyper-parameter search for each model individually, before combining them into a single ensemble agent.

In addition to fine-tuning, another area for consideration would be try different method for combining the sub-agents preferred actions. For this paper, we used a majority voting scheme, which seemed to work well, but there are other methods that could be used that might increase performance further. One method in particular that we would like to research, is the use of a weighted voting scheme that allows for the assignment of weights to each agent in the ensemble based on a post-training evaluation phase. These weights could then be used to put additional emphasis on agents that perform better individually, while not completely discounting the weaker agents. Furthermore, for stocks specifically, it is possible that a weighted scheme could be used at on a per-stock or per-trend level as well. This would help to exploit the agents' strengths and weaknesses more carefully.

Finally, it would be crucial to improve the agents performance against stocks that are in a general downward trend. While are agent's poor performance against these stocks was most likely due to unbalanced stock data, this could potentially be avoided altogether by allowing the agent to short and cover stocks. This would allow the agent to be able to profit from negatively trending stocks, and ultimately make it more versatile towards any type of situation.

Conclusion

Over the past several decades, automated trading accounts for the majority of all trading volume within the US stock exchange. Automated trading strategies usually involve a signficant amount of hard-coded conditional logic that only works in certain scenarios. We introduced an ensemble-based Deep Reinforcement Learning approach aimed towards generalizing automated stock trading with the goal of maximizing potential risk-adjusted return. Our agent ensemble was trained and tested against the 10 highest allocated stocks within the DJIA index. The agent obtained significantly better Sortino Ratios for nine out of the 10 stocks when tested against baseline strategies. Furthermore, our agent was able to out perform two state-of-the-art DRL models across four out of five stocks listed within the DJIA with respect to risk-adjusted return.

References

Abarbanell, J. S., and Bushee, B. J. 1997. Fundamental analysis, future earnings, and stock prices. *Journal of Accounting Research* 35(1):1–24.

Bao, Y.; Lu, Y.; and Zhang, J. 2004. Forecasting stock price by svms regression. In Bussler, C., and Fensel, D., eds., *Artificial Intelligence: Methodology, Systems, and Applications*, 295–303. Berlin, Heidelberg: Springer Berlin Heidelberg.

Dash, R., and Dash, P. K. 2016. A hybrid stock trading framework integrating technical analysis with machine learning techniques. *The Journal of Finance and Data Science* 2(1):42–57.

Deng, Y.; Bao, F.; Kong, Y.; Ren, Z.; and Dai, Q. 2017. Deep Direct Reinforcement Learning for Financial Signal Representation and Trading. *IEEE Transactions on Neural Networks and Learning Systems* 28(3):653–664.

Edwards, R. D.; Magee, J.; and Bassetti, W. C. 2018. *Technical Analysis of Stock Trends*. CRC press.

Fujimoto, S.; Hoof, H.; and Meger, D. 2018. Addressing function approximation error in actor-critic methods. In *International Conference on Machine Learning*, 1587–1596. PMLR.

Haarnoja, T.; Zhou, A.; Abbeel, P.; and Levine, S. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, 1861–1870. PMLR.

Niaki, S., and Hoseinzade, S. 2013. Forecasting SP 500 Index using Artificial Neural Networks and Design of Experiments. *Journal of Industrial Engineering International* 9:1–9.

Parisotto, E.; Song, F.; Rae, J.; Pascanu, R.; Gulcehre, C.; Jayakumar, S.; Jaderberg, M.; Kaufman, R. L.; Clark, A.; Noury, S.; et al. 2020. Stabilizing transformers for reinforcement learning. In *International Conference on Machine Learning*, 7487–7498. PMLR.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal policy optimization algorithms. *ArXiv* abs/1707.06347.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science* 362(6419):1140–1144.

Vezeris, D.; Karkanis, I.; Kyrgos, T.; and Lv, K. 2019. Adturtle: An Advanced Turtle Trading System.

Wu, X.; Chen, H.; Wang, J.; Troiano, L.; Loia, V.; and Fujita, H. 2020. Adaptive stock trading strategies with deep reinforcement learning methods. *Information Sciences* 538:142 – 158.

Proposal CS 5080 Fall 2021

Adaptive Open-Set Clustering for Incremental Learning: A Reinforcement Learning Approach

Justin Leo and Jugal Kalita
Department of Computer Science
University of Colorado at Colorado Springs
jleo@uccs.edu

Abstract

Open-Set classification algorithms are often very effective at distinguishing sets of unrecognizable data apart from a larger mix. However, learning to distinguish multiple groups within the identified data is often difficult. This problem is often noticeable in autonomous continuous learning as degraded open-set classification leads to compounded error within incremental learning algorithms. In this paper, we utilize reinforcement learning to develop a more efficient open-set classification method that has success at identifying unknown data as well as separating their sub-class groups. The method is a form of clustering that has a learning agent monitoring the clustering accuracy and learns the ideal method of data division. The model once initially trained is then applied in an incremental learning task and will also continuously evolve as the agent will continue to monitor performance and adapt based on results.

Introduction

The purpose of any deployed agent using artificial intelligence is to simulate human interaction and comprehension in its environment. While most agents are designed for specific applications; recent research has attempted to push these boundaries and allow an agent to adapt and learn new knowledge based on an evolving environment or an evolving set of required actions (Nilsson, 2014). Most modern forms of artificial intelligence use neural networks and have shown success in multiple tasks such as Computer Vision and Natural Language Processing. However the attributed success is mainly for specific tasks or in a closed-world scenario. An agent deployed in a real world environment would need to be designed such that it's able to interact and learn from unpredictability. This means the system would need to be able to perform in an open-world scenario; in an open-world environment the agent would encounter unknown information and gain knowledge from this information over time (Bendale and Boult, 2015; Bendale and Boult, 2016).

Recent open-world research specifically for neural networks focuses on open-set classification (Leo and Kalita, 2020). Open-set classification is the process where a neural network classifies trained data as well as detects unknown data during the testing phase. This task has many challenges as neural networks need to be able to dynamically modify when new information is presented, as well as be able to distinguish between various classes of unknown data as multiple may be present. One common technique used for this is clustering. While there are numerous clustering algorithms, most typically do not produce high accuracies in open-set classification tasks (Leo and Kalita, 2020; Xu and Tian, 2015). In this paper we focus on developing a clustering method specific to open-set classification using a reinforcement learning approach.

The idea of reinforcement learning was initially conceived in 1954 and has had continuous applications for numerous intelligent systems such as autonomous driving and healthcare applications (Minsky, 1954; Kiran *et al.*, 2021; Coronato *et al.*, 2020). Reinforcement learning is a machine learning technique where an agent attempts to learn to perform actions in an environment based on experience. Rewards are used to help guide the agent to an optimal or final state. Since traditional clustering algorithms perform poorly on open-set classification tasks, a clustering model utilizing reinforcement learning can be specifically designed to perform clustering with greater accuracy.

Background

The main focus of development in this research is the use of reinforcement learning to help with clustering. Reinforcement learning has been shown to have success with numerous problems either using single or multi agent approaches to learn and act on an environment in order to reach a goal (Shoham *et al.*, 2003). This section will discuss a few specific properties and characteristics of reinforcement learning algorithms.

Finite Markov Decision Process

A Markov Decision process is a discrete time decision process controlled by an algorithmic function. A learning agent acts in an uncontrolled environment that is typically not fully known, however the agent itself is fully known. The Markov model is defined by five characteristics: state space, actions, policy, rewards, discount factor. The state space consists of all the possible states the agent can reach and are the basis for the agent's choices. The actions consists of all the possible transition types the agent can perform. The policy is a probability distribution that the agent follows to select

actions based on a function of states. The rewards are the gains or losses an agent receives by reaching certain states. The discount factor is a rate that is used to motivate the agent to converge in a optimal goal state faster. The agent's goal with the Markov process is to reach a certain goal only using these characteristics by maximizing the reward and learning an optimal policy through iterations of learning.

Monte Carlo and Temporal-Difference Methods

The Monte Carlo method is a learning method that requires experiences of a task rather than any knowledge of the environment. The method is used for episodic tasks where the agent performs evaluation at the end of each episode. This method also follows the Markov property where there is no history of each episode. The main advantage of this approach is that an agent can learn an optimal behavior with no background information. This is useful for our approach as the agent will be exposed to data with no prior knowledge of specific class features that allow for easy classification.

The Temporal-Difference method is a learning method similar to the Monte Carlo with a few changes. The main difference is that this method allows for policy changes without the completion of the episode. The main advantage of this feature is that the agent can learn to find an optimal solution faster because typically not as many episodes are required. However, one drawback is that the agent can sometimes not fully explore the state space. For our approach a combination of these methods would be ideal as efficiency and robustness of the agent would be successful.

Related Work

The related work is focused on three main areas: the open-set classification task, incremental learning clustering, and reinforcement learning applications for open-set classification.

Open-set classification for neural networks is the ability to identify unknown class data and assign the data a new label with the purpose of building upon the network's knowledge. Recent work in this area have focused on augmenting the architecture of a neural network to be able to incorporate new knowledge. One such technique called the OpenMax method introduces a new model layer; this layer is used to estimate probability of the input data to be unknown (Bendale and Boult, 2016). The estimation is performed by measuring the distance between an input's activation vector and comparing it with the model vector for a few top classes. The approach proposed by this paper will focus on the idea of input vectors derived by the neural model to derive clusters.

Clustering is the process of combining sets of data to from a common group. There are numerous clustering algorithms each with various advantages, however there has been little research to measure the efficiency of these algorithms in an incremental learning task (Xu and Tian, 2015). One method recently adapted for this task is called δ-open set clustering (Wang *et al.*, 2018). This method is a topological approach to determine sub-cluster groups within sets of data in order to increase classification accuracy. This is greatly important when used for incremental learning as it helps reduces data forgetting. Another method uses Agg-Var clustering to show that different groups of data often have very different cluster spaces; this method solves this issue by creating a set of centroids to represent evolving clusters (Ayub and Wagner, 2020). Since cluster creation is non-deterministic, using a reinforcement learning agent will help produce clusters more fine-tuned for specific tasks.

The open-set classification task is often inadequate when used for incremental learning. The problem is from compounded error through incremental learning iterations. A neural algorithm proposed by Shiraga et al. shows that reinforcement learning can be used to help control "interference" caused by incrementally learning data over time (Shiraga *et al.*, 2002). Using a form of long-term memory, a learning agent is able to help identify data relationships to reduce error. Another approach developed by Bian and Jiang shows that reinforcement learning can be used to develop a continuous-time learning algorithm that can learn without the use of batched data (Bian and Jiang, 2019); this helps reduce error as batched data can incorporate misclassified mixed data. The agent used is also shown to reach convergence even with the presence of data disturbances. Using this idea, a reinforcement learning agent is able to be more resilient to data errors that compound in incremental learning.

Methodology

The methodology for our approach will follow a typical open-set classification approach but is amended to add a reinforcement learning agent. The method described also uses a feature extractor and centroid based approach inspired by recent state-of-the-art research (Ayub and Wagner, 2020). The steps are described below.

- A neural model is trained on a set of known classes. This model is used as a feature extractor.
- The feature vectors are clustered using an untrained RL agent. The agent is given the mean squared error from the previous centroids to the new data sample's feature vector. This creates a group of centroids representing the known data.
- The clustered data is evaluated and the agent is rewarded based on the accuracy of the labels.
- This process is repeated until a high accuracy is achieved.
- The trained agent now is presented with a mix of known data and previously unseen class data. The agent performs clustering and assigns all the data labels.

Based on the approach described, the agent's actions when receiving a new data sample are to either create a new cluster or to assign the sample to an existing cluster. The agent is also capable of creating multiple centroids per class. This is important because classes are not necessarily centered around one local mean; rather they can have different localizatios based on unique features (Ramesh *et al.*, 2018). The agent's rewards being directly correlated to the accuracy of the clustering dictates that the agent would learn to identify and optimize the distance needed between data samples

and clusters in order to take the appropriate action. The mean squared error as shown in Equation 1 is used to calculate the distance difference from a feature vector to all the centroids of a certain class.

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(x_i - c_i)^2 \quad (1)$$

Evaluation Methods

The main area for evaluation is the clustering accuracy as the reinforcement learning agent will be using the clustering results for learning rewards. Also any further applications of the clustering classifications such as a continuous incremental learning algorithm will also be directly dependent on the accuracy. Another metric that will be used in the evaluation is the Incremental Class Accuracy (ICA); this metric is specifically designed for open-set clustering and measures multiple variables (Leo and Kalita, 2020). The ICA metric is shown in Equation 2; this metric is an average of homogeneity $\frac{max(n_{\mathrm{c|k}})}{N_{\mathrm{k}}}$, completeness $\frac{max(n_{\mathrm{c|k}})}{N_{\mathrm{c}}}$, and unknown identification $\frac{(n_{\mathrm{u|k}})}{N_{\mathrm{k}}}$ accuracies .

$$ICA = \left(\frac{max(n_{\mathrm{c|k}})}{N_{\mathrm{k}}} + \frac{max(n_{\mathrm{c|k}})}{N_{\mathrm{c}}} + \frac{(n_{\mathrm{u|k}})}{N_{\mathrm{k}}}\right) * \frac{1}{3} \quad (2)$$

Conclusion

In this paper, we propose a method to perform open-set clustering using a reinforcement learning agent. This will be done by using a training set of data to teach an agent the nuances between different classes. The agent would then perform with greater accuracy than regular clustering algorithms as the agent would learn specific features of specific datasets. The trained agent is then tested against the testing set of data which includes data from additional datasets as well. The agent is then evaluated using traditional accuracy measures as well as the ICA score. The proposed method is also then shown to perform better in clustering needed tasks such as some forms on incremental learning.

Timeline

Date	Task Description
9/25 - 10/6	Setup Environment and Select Data
10/6 - 10/20	Setup Feature Extractor and Create Open-Set Code
10/20 - 11/3	Set up RL agent and Test
11/3 - End	Optimize Results and Make Improvements

References

Ali Ayub and Alan R Wagner. Cognitively-inspired model for incremental learning using a few examples. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 222–223, 2020.

Abhijit Bendale and Terrance Boult. Towards open world recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1893–1902, 2015.

Abhijit Bendale and Terrance E Boult. Towards open set deep networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1563–1572, 2016.

Tao Bian and Zhong-Ping Jiang. Reinforcement learning for linear continuous-time systems: An incremental learning approach. *IEEE/CAA Journal of Automatica Sinica*, 6(2):433–440, 2019.

Antonio Coronato, Muddasar Naeem, Giuseppe De Pietro, and Giovanni Paragliola. Reinforcement learning for intelligent healthcare applications: A survey. *Artificial Intelligence in Medicine*, 109:101964, 2020.

B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

Justin Leo and Jugal Kalita. Moving towards open set incremental learning: Readily discovering new authors. In *Future of Information and Communication Conference*, pages 739–751. Springer, Cham, 2020.

Marvin Lee Minsky. *Theory of neural-analog reinforcement systems and its application to the brain-model problem*. Princeton University, 1954.

Nils J Nilsson. *Principles of artificial intelligence*. Morgan Kaufmann, 2014.

Chundru Raja Ramesh, Komaragiri Raghava Rao, and Gunamani Jena. Fuzzy clustering algorithm efficient implementation using centre of centres. *International Journal of Intelligent Engineering and Systems*, 11(5):1–10, 2018.

Naoto Shiraga, Seiichi Ozawa, and Shigeo Abe. A reinforcement learning algorithm for neural networks with incremental learning ability. In *Proceedings of the 9th International Conference on Neural Information Processing, 2002. ICONIP'02.*, volume 5, pages 2566–2570. IEEE, 2002.

Yoav Shoham, Rob Powers, and Trond Grenager. Multi-agent reinforcement learning: a critical survey. Technical report, Technical report, Stanford University, 2003.

Shuliang Wang, Qi Li, Hanning Yuan, Deren Li, Jing Geng, Chuanfeng Zhao, Yimeng Lei, Chuanlu Liu, and Chengfei Liu. δ-open set clustering—a new topological clustering method. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(6):e1262, 2018.

Dongkuan Xu and Yingjie Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2(2):165–193, 2015.

Midterm CS 5080 Fall 2021

Adaptive Open-Set Clustering for Incremental Learning: A Reinforcement Learning Approach

Justin Leo and Jugal Kalita
Department of Computer Science
University of Colorado at Colorado Springs
jleo@uccs.edu

Abstract

Open-Set classification algorithms are often very effective at distinguishing sets of unrecognizable data apart from a larger mix. However, learning to distinguish multiple groups within the identified data is often difficult. This problem is often noticeable in autonomous continuous learning as degraded open-set classification leads to compounded error within incremental learning algorithms. In this paper, we utilize reinforcement learning to develop a more efficient open-set classification method that has success at identifying unknown data as well as separating their sub-class groups. The method is a form of clustering that has a learning agent monitoring the clustering accuracy and learns the ideal method of data division. The model once initially trained is then applied in an incremental learning task and will also continuously evolve as the agent will continue to monitor performance and adapt based on results.

Introduction

The purpose of any deployed agent using artificial intelligence is to simulate human interaction and comprehension in its environment. While most agents are designed for specific applications; recent research has attempted to push these boundaries and allow an agent to adapt and learn new knowledge based on an evolving environment or an evolving set of required actions (Nilsson, 2014). Most modern forms of artificial intelligence use neural networks and have shown success in multiple tasks such as Computer Vision and Natural Language Processing. However the attributed success is mainly for specific tasks or in a closed-world scenario. An agent deployed in a real world environment would need to be designed such that it's able to interact and learn from unpredictability. This means the system would need to be able to perform in an open-world scenario; in an open-world environment the agent would encounter unknown information and gain knowledge from this information over time (Bendale and Boult, 2015; Bendale and Boult, 2016).

Recent open-world research specifically for neural networks focuses on open-set classification (Leo and Kalita, 2020; Leo and Kalita, 2021). Open-set classification is the process where a neural network classifies trained data as well as detects unknown data during the testing phase. This task has many challenges as neural networks need to be able to dynamically modify when new information is presented, as well as be able to distinguish between various classes of unknown data as multiple may be present. One common technique used for this is clustering. While there are numerous clustering algorithms, most typically do not produce high accuracies in open-set classification tasks (Leo and Kalita, 2020; Xu and Tian, 2015). In this paper we focus on developing a clustering method specific to open-set classification using a reinforcement learning approach.

The idea of reinforcement learning was initially conceived in 1954 and has had continuous applications for numerous intelligent systems such as autonomous driving and healthcare applications (Minsky, 1954; Kiran *et al.*, 2021; Coronato *et al.*, 2020). Reinforcement learning is a machine learning technique where an agent attempts to learn to perform actions in an environment based on experience. Rewards are used to help guide the agent to an optimal or final state. Since traditional clustering algorithms perform poorly on open-set classification tasks, a clustering model utilizing reinforcement learning can be specifically designed to perform clustering with greater accuracy.

Background

The main focus of development in this research is the use of reinforcement learning to help with clustering. Reinforcement learning has been shown to have success with numerous problems either using single or multi agent approaches to learn and act on an environment in order to reach a goal (Shoham *et al.*, 2003). This section will discuss a few specific properties and characteristics of reinforcement learning algorithms.

Finite Markov Decision Process

A Markov Decision process is a discrete time decision process controlled by an algorithmic function. A learning agent acts in an uncontrolled environment that is typically not fully known, however the agent itself is fully known. The Markov model is defined by five characteristics: state space, actions, policy, rewards, discount factor. The state space consists of all the possible states the agent can reach and are the basis for the agent's choices. The actions consists of all the possible transition types the agent can perform. The policy is a probability distribution that the agent follows to select

actions based on a function of states. The rewards are the gains or losses an agent receives by reaching certain states. The discount factor is a rate that is used to motivate the agent to converge in a optimal goal state faster. The agent's goal with the Markov process is to reach a certain goal only using these characteristics by maximizing the reward and learning an optimal policy through iterations of learning.

Monte Carlo and Temporal-Difference Methods

The Monte Carlo method is a learning method that requires experiences of a task rather than any knowledge of the environment. The method is used for episodic tasks where the agent performs evaluation at the end of each episode. This method also follows the Markov property where there is no history of each episode. The main advantage of this approach is that an agent can learn an optimal behavior with no background information. This is useful for our approach as the agent will be exposed to data with no prior knowledge of specific class features that allow for easy classification.

The Temporal-Difference method is a learning method similar to the Monte Carlo with a few changes. The main difference is that this method allows for policy changes without the completion of the episode. The main advantage of this feature is that the agent can learn to find an optimal solution faster because typically not as many episodes are required. However, one drawback is that the agent can sometimes not fully explore the state space. For our approach a combination of these methods would be ideal as efficiency and robustness of the agent would be successful.

Neural Networks

As mentioned before neural networks perform very well in robust tasks and some reinforcement learning algorithms use them as well. Neural networks in reinforcement learning use experienced replay to learn and approximate a state-value function to converge to an optimal policy. One of the original implementations of this is Deep Q-Learning which uses a neural network to implement the Q-Learning algorithm (Mnih *et al.*, 2015). The approach described in this paper also uses a neural network to learn the state space and make optimal decisions.

Related Work

The related work is focused on three main areas: the open-set classification task, incremental learning clustering, and reinforcement learning applications for open-set classification.

Open-set classification for neural networks is the ability to identify unknown class data and assign the data a new label with the purpose of building upon the network's knowledge. Recent work in this area have focused on augmenting the architecture of a neural network to be able to incorporate new knowledge. One such technique called the OpenMax method introduces a new model layer; this layer is used to estimate probability of the input data to be unknown (Bendale and Boult, 2016). The estimation is performed by measuring the distance between an input's activation vector and comparing it with the model vector for a few top classes. The approach proposed by this paper will focus on the idea of input vectors derived by the neural model to derive clusters.

Clustering is the process of combining sets of data to from a common group. There are numerous clustering algorithms each with various advantages, however there has been little research to measure the efficiency of these algorithms in an incremental learning task (Xu and Tian, 2015). One method recently adapted for this task is called δ-open set clustering (Wang *et al.*, 2018). This method is a topological approach to determine sub-cluster groups within sets of data in order to increase classification accuracy. This is greatly important when used for incremental learning as it helps reduces data forgetting. Another method uses Agg-Var clustering to show that different groups of data often have very different cluster spaces; this method solves this issue by creating a set of centroids to represent evolving clusters (Ayub and Wagner, 2020). Since cluster creation is non-deterministic, using a reinforcement learning agent will help produce clusters more fine-tuned for specific tasks.

The open-set classification task is often inadequate when used for incremental learning. The problem is from compounded error through incremental learning iterations. A neural algorithm proposed by Shiraga et al. shows that reinforcement learning can be used to help control "interference" caused by incrementally learning data over time (Shiraga *et al.*, 2002). Using a form of long-term memory, a learning agent is able to help identify data relationships to reduce error. Another approach developed by Bian and Jiang shows that reinforcement learning can be used to develop a continuous-time learning algorithm that can learn without the use of batched data (Bian and Jiang, 2019); this helps reduce error as batched data can incorporate misclassified mixed data. The agent used is also shown to reach convergence even with the presence of data disturbances. Using this idea, a reinforcement learning agent is able to be more resilient to data errors that compound in incremental learning.

Methodology

This section describes the implementation methodology as well as explains implementation choices made for the proposed approach. The methodology will follow a typical open-set classification approach but is amended to add a reinforcement learning agent. The steps are described are also shown in Figure 1.

Feature Extractor

The feature extractor model is used to obtain feature vectors for the data being used. The motivation behind using a feature extractor to obtain feature vectors rather than directly passing the data to the agent is so the characteristics separating the data classes are more distinguished (Perera *et al.*, 2020). Traditional clustering algorithms perform poorly in open-set classification because the full set of data is not pre-trained to the clustering model; this leads to the model suffering from inter-task confusion as the model never jointly learns the specific methods to divide all the classes (Masana *et al.*, 2020). The feature extractor helps alleviate this problem as the clustering algorithm

can now easily focus on the exact distinguishable features. The model being used for this is the ResNet-152 structure pre-trained on the Imagenet dataset (He *et al.*, 2016; Deng *et al.*, 2009); this architecture was chosen as it has been successful as feature extraction in past literature (Ayub and Wagner, 2020). Initially before any training we first divide the dataset to a training set comprised of a set of classes to a specified index C_i and to a testing set comprised of a set of classes to a specified index C_n that includes the training class indexes as well as "unknown" additional classes, Equation 1.

$$\begin{aligned} TrainingClasses &= C_0...C_i \\ TestingClasses &= C_0...C_i...C_n \end{aligned} \tag{1}$$

The pre-trained feature extractor is then fine tuned on the training set. To evaluate the performance of the feature extractor we calculated the Euclidean Distance as shown in Equation 2 from vectors from the same classes and different classes. The results as shown in Table 1 show that the distances from feature vectors in the same class are are much lower than that of other classes.

$$d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{2}$$

	Class 0	Class 1	Class 2	Class 3	Class 4
Class 0	**0.027**	1.394	1.323	1.395	1.388
Class 1	1.411	**0.0008**	1.360	1.413	1.411
Class 2	1.401	1.403	**0.088**	1.402	1.400
Class 3	1.411	1.412	1.358	**0.002**	1.409
Class 4	1.412	1.413	1.314	1.413	**0.002**

Table 1: Euclidean Distance differences between different generated vectors for 5 classes. The values are low for vectors corresponding to different samples from the same class. This shows the feature extractor model is able to generate distinguishable vectors.

Cluster Space

The cluster space is the space representing all the trained classes with respect to each other class. The space initially contains the training data and is later updated by the agent as new data is identified. The cluster space K_T is comprised of various cluster points K_i, each of which represents a number of data samples relatively close. Each class can be represented with more than one cluster as classes are not necessarily centered around one local mean, rather they can have different localizations based on unique features (Ramesh *et al.*, 2018).

During the training phase all the samples pertaining to one class C_i are averaged based on a certain distance threshold such that local average points are obtained, these become the clusters for class C_i. This is repeated for all the training classes and becomes the initial cluster space. The RL agent

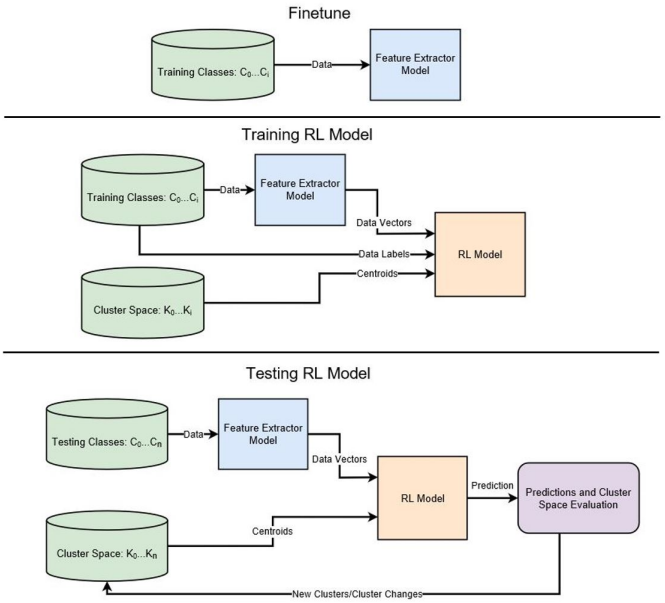

Figure 1: Diagram showing the individual steps of Finetuning the Feature Extractor Model, Training the RL Agent Model, and Testing the RL Agent Model.

trains on this cluster space as this essentially represents the state space. During the testing phase data is presented incrementally to the agent and the agent makes a choice for all the incoming data, the specific clustering steps are explained below (Ayub and Wagner, 2019).

Create New Cluster For a data sample x_i, if the agent chooses to create a new cluster then this sample's vector becomes the new centroid point. The centroid's classification label is calculated based on a voting scheme to create a set of votes V between the group of m closest centroids. If the new centroid is not voted consistently by a large majority then the centroid is assigned a new class label. This is shown in Equation 3.

$$\begin{aligned} V &= \{K_i \mid min(K_i - x_i),\ |V| < m\} \\ K_T \cup x_i &\leftarrow max(V) \end{aligned} \tag{3}$$

Add to Existing Cluster For a data sample x_i, if the agent chooses to add to an existing cluster then the closest centroid point K_i is weighted averaged with the new data sample vector. This is shown in Equation 4 where w_{Ki} is the number of data points covered by the centroid K_i.

$$\begin{aligned} K_i &= \{K_i \mid min(K_i - x_i),\ |K_i| = 1\} \\ K_i^{new} &= \frac{w_{Ki} * (K_i^{old} + x_i)}{w_{Ki} + 1} \end{aligned} \tag{4}$$

The motivation behind comparing input data to existing learned data is due to concept formation and biological learning methods. The hippocampus forms new knowledge by organizing novel information in space defined by

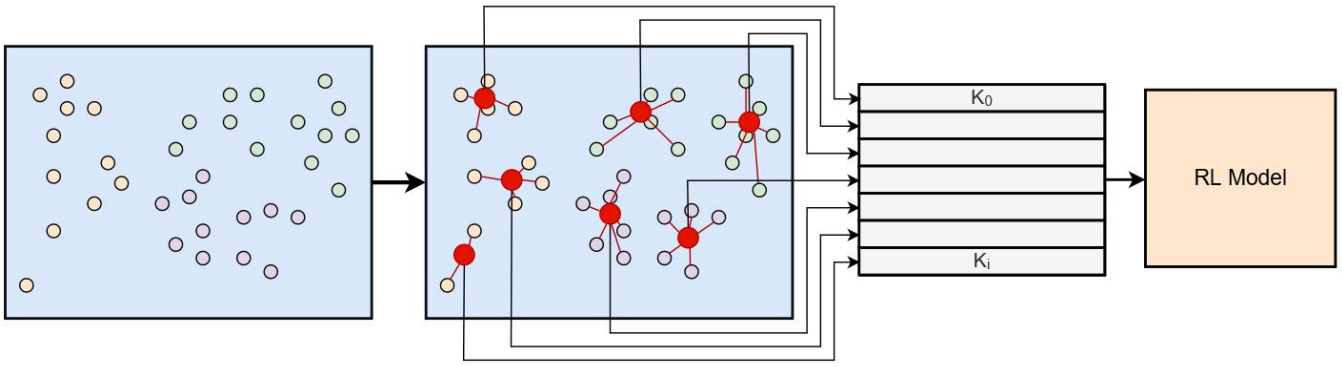

Figure 2: Diagram showing the clusters being created for the set of data and formed to an input for the RL agent. The centroid vectors are appended with their class labels. The input shows a representation of the current state.

existing learnt experiences (Theves *et al.*, 2020; Ayub and Wagner, 2020). After being presented with new information, a *memory-based prediction error* is calculated to represent the overlap between the new information and the previously known information. In the proposed approach, the distance calculation between the data sample x_i and the set of centroid points K_T allows the new information to be shaped based on the model's existing knowledge.

RL Model

The reinforcement learning model is the main component of the approach as it makes the classification decisions. The model will train on the initial cluster space and learn the relationships between the centroid points of the different classes. The motivation behind using a reinforcement learning model to make predictions rather than having a fixed distance calculation is that the relationships between different classes are not always the same; this means any fixed distance parameter would not produce optimal results. This would also affect models tested on mixed datasets as there would be no predictable way to know how the classes would be represented in the cluster space. Using a reinforcement model that constantly updates it's knowledge to find optimal solutions would solve this issue.

The model used for the agent is also a ResNet-152 model. The cluster space is represented by layering the centroid points' vectors and this is directly given to the RL model, this is shown in Figure 2. Since neural models typically expect a fixed input size, the number of centroid points need to be limited. In the event that there are a large number of centroids $|K_T| > Z$, the centroids that represent the least number of data samples will be weighted averaged to the nearest same class centroid. To avoid this problem the initial limit Z used by the constructed RL model will be set to a high value. Once the model is trained, it can take new data and divide the samples into new classes based on their representation in the cluster space. Then once the cluster space is updated the RL agent can train on the space again; this means the agent moves to an incremental learning structure capable of learning new data from the environment.

Evaluation Methods

The main area for evaluation is the clustering accuracy as the reinforcement learning agent will be using the clustering results for learning rewards. Also any further applications of the clustering classifications such as a continuous incremental learning algorithm will also be directly dependent on the accuracy. Another metric that will be used in the evaluation is the Incremental Class Accuracy (ICA); this metric is specifically designed for open-set clustering and measures multiple variables (Leo and Kalita, 2020). The ICA metric is shown in Equation 5; this metric is an average of homogeneity $\frac{max(n_{c|k})}{N_k}$, completeness $\frac{max(n_{c|k})}{N_c}$, and unknown identification $\frac{(n_{u|k})}{N_k}$ accuracies.

$$ICA = \left(\frac{max(n_{c|k})}{N_k} + \frac{max(n_{c|k})}{N_c} + \frac{(n_{u|k})}{N_k} \right) * \frac{1}{3} \quad (5)$$

Conclusion

In this paper, we propose a method to perform open-set clustering using a reinforcement learning agent. This will be done by using a training set of data to teach an agent the nuances between different classes. The agent would then perform with greater accuracy than regular clustering algorithms as the agent would learn specific features of specific datasets. The trained agent is then tested against the testing set of data which includes data from additional datasets as well. The agent is then evaluated using traditional accuracy measures as well as the ICA score. The proposed method is also then shown to perform better in clustering needed tasks such as some forms on incremental learning.

Timeline

Date	Task Description
10/25 - 11/8	Start Initial Experiments Training RL Agent
11/8 - 11/22	Evaluate Performance, Make Changes to Improve
11/22 - End	Obtain Results and Finish Paper

References

Ali Ayub and Alan Wagner. Cbcl: Brain inspired model for rgb-d indoor scene classification. *arXiv preprint arXiv:1911.00155*, 2(3), 2019.

Ali Ayub and Alan R Wagner. Cognitively-inspired model for incremental learning using a few examples. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 222–223, 2020.

Abhijit Bendale and Terrance Boult. Towards open world recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1893–1902, 2015.

Abhijit Bendale and Terrance E Boult. Towards open set deep networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1563–1572, 2016.

Tao Bian and Zhong-Ping Jiang. Reinforcement learning for linear continuous-time systems: An incremental learning approach. *IEEE/CAA Journal of Automatica Sinica*, 6(2):433–440, 2019.

Antonio Coronato, Muddasar Naeem, Giuseppe De Pietro, and Giovanni Paragliola. Reinforcement learning for intelligent healthcare applications: A survey. *Artificial Intelligence in Medicine*, 109:101964, 2020.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

Justin Leo and Jugal Kalita. Moving towards open set incremental learning: Readily discovering new authors. In *Future of Information and Communication Conference*, pages 739–751. Springer, Cham, 2020.

Justin Leo and Jugal Kalita. Incremental deep neural network learning using classification confidence thresholding. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

Marc Masana, Xialei Liu, Bartlomiej Twardowski, Mikel Menta, Andrew D Bagdanov, and Joost van de Weijer. Class-incremental learning: survey and performance evaluation on image classification. *arXiv preprint arXiv:2010.15277*, 2020.

Marvin Lee Minsky. *Theory of neural-analog reinforcement systems and its application to the brain-model problem*. Princeton University, 1954.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Nils J Nilsson. *Principles of artificial intelligence*. Morgan Kaufmann, 2014.

Pramuditha Perera, Vlad I Morariu, Rajiv Jain, Varun Manjunatha, Curtis Wigington, Vicente Ordonez, and Vishal M Patel. Generative-discriminative feature representations for open-set recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11814–11823, 2020.

Chundru Raja Ramesh, Komaragiri Raghava Rao, and Gunamani Jena. Fuzzy clustering algorithm efficient implementation using centre of centres. *International Journal of Intelligent Engineering and Systems*, 11(5):1–10, 2018.

Naoto Shiraga, Seiichi Ozawa, and Shigeo Abe. A reinforcement learning algorithm for neural networks with incremental learning ability. In *Proceedings of the 9th International Conference on Neural Information Processing, 2002. ICONIP'02.*, volume 5, pages 2566–2570. IEEE, 2002.

Yoav Shoham, Rob Powers, and Trond Grenager. Multi-agent reinforcement learning: a critical survey. Technical report, Technical report, Stanford University, 2003.

Stephanie Theves, Guillén Fernández, and Christian F Doeller. The hippocampus maps concept space, not feature space. *Journal of Neuroscience*, 40(38):7318–7325, 2020.

Shuliang Wang, Qi Li, Hanning Yuan, Deren Li, Jing Geng, Chuanfeng Zhao, Yimeng Lei, Chuanlu Liu, and Chengfei Liu. δ-open set clustering—a new topological clustering method. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(6):e1262, 2018.

Dongkuan Xu and Yingjie Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2(2):165–193, 2015.

Final CS 5080 Fall 2021

Adaptive Open-Set Clustering for Incremental Learning: A Reinforcement Learning Approach

Justin Leo and Jugal Kalita
Department of Computer Science
University of Colorado at Colorado Springs
jleo@uccs.edu

Abstract

Open-Set classification algorithms are often very effective at distinguishing sets of unrecognizable data apart from a larger mix. However, learning to distinguish multiple groups within the identified data is often difficult. This problem is often noticeable in autonomous continuous learning as degraded open-set classification leads to compounded error within incremental learning algorithms. In this paper, we utilize reinforcement learning to develop a more efficient open-set classification method that has success at identifying unknown data as well as separating their sub-class groups. The method is a form of clustering that has a learning agent monitoring the clustering accuracy and learns the ideal method of data division. The model once initially trained is then applied in an incremental learning task and will also continuously evolve as the agent will continue to monitor performance and adapt based on results.

Introduction

The purpose of any deployed agent using artificial intelligence is to simulate human interaction and comprehension in its environment. While most agents are designed for specific applications; recent research has attempted to push these boundaries and allow an agent to adapt and learn new knowledge based on an evolving environment or an evolving set of required actions (Nilsson, 2014). Most modern forms of artificial intelligence use neural networks and have shown success in multiple tasks such as Computer Vision and Natural Language Processing. However the attributed success is mainly for specific tasks or in a closed-world scenario. An agent deployed in a real world environment would need to be designed such that it's able to interact and learn from unpredictability. This means the system would need to be able to perform in an open-world scenario; in an open-world environment the agent would encounter unknown information and gain knowledge from this information over time (Bendale and Boult, 2015; Bendale and Boult, 2016).

Recent open-world research specifically for neural networks focuses on open-set classification (Leo and Kalita, 2020; Leo and Kalita, 2021). Open-set classification is the process where a neural network classifies trained data as well as detects unknown data during the testing phase. This task has many challenges as neural networks need to be able to dynamically modify when new information is presented, as well as be able to distinguish between various classes of unknown data as multiple may be present. One common technique used for this is clustering. While there are numerous clustering algorithms, most typically do not produce high accuracies in open-set classification tasks (Leo and Kalita, 2020; Xu and Tian, 2015). In this paper we focus on developing a clustering method specific to open-set classification using a reinforcement learning approach.

The idea of reinforcement learning was initially conceived in 1954 and has had continuous applications for numerous intelligent systems such as autonomous driving and healthcare applications (Minsky, 1954; Kiran *et al.*, 2021; Coronato *et al.*, 2020). Reinforcement learning is a machine learning technique where an agent attempts to learn to perform actions in an environment based on experience. Rewards are used to help guide the agent to an optimal or final state. Since traditional clustering algorithms perform poorly on open-set classification tasks, a clustering model utilizing reinforcement learning can be specifically designed to perform clustering with greater accuracy.

Background

The main focus of development in this research is the use of reinforcement learning to help with clustering. Reinforcement learning has been shown to have success with numerous problems either using single or multi agent approaches to learn and act on an environment in order to reach a goal (Shoham *et al.*, 2003). This section will discuss a few specific properties and characteristics of reinforcement learning algorithms.

Finite Markov Decision Process

A Markov Decision process is a discrete time decision process controlled by an algorithmic function. A learning agent acts in an uncontrolled environment that is typically not fully known, however the agent itself is fully known. The Markov model is defined by five characteristics: state space, actions, policy, rewards, discount factor. The state space consists of all the possible states the agent can reach and are the basis for the agent's choices. The actions consists of all the possible transition types the agent can perform. The policy is a probability distribution that the agent follows to select

actions based on a function of states. The rewards are the gains or losses an agent receives by reaching certain states. The discount factor is a rate that is used to motivate the agent to converge in a optimal goal state faster. The agent's goal with the Markov process is to reach a certain goal only using these characteristics by maximizing the reward and learning an optimal policy through iterations of learning.

Monte Carlo and Temporal-Difference Methods

The Monte Carlo method is a learning method that requires experiences of a task rather than any knowledge of the environment. The method is used for episodic tasks where the agent performs evaluation at the end of each episode. This method also follows the Markov property where there is no history of each episode. The main advantage of this approach is that an agent can learn an optimal behavior with no background information. This is useful for our approach as the agent will be exposed to data with no prior knowledge of specific class features that allow for easy classification.

The Temporal-Difference method is a learning method similar to the Monte Carlo with a few changes. The main difference is that this method allows for policy changes without the completion of the episode. The main advantage of this feature is that the agent can learn to find an optimal solution faster because typically not as many episodes are required. However, one drawback is that the agent can sometimes not fully explore the state space. For our approach a combination of these methods would be ideal as efficiency and robustness of the agent would be successful.

Neural Networks

As mentioned before neural networks perform very well in robust tasks and some reinforcement learning algorithms use them as well. Neural networks in reinforcement learning use experienced replay to learn and approximate a state-value function to converge to an optimal policy. One of the original implementations of this is Deep Q-Learning which uses a neural network to implement the Q-Learning algorithm (Mnih *et al.*, 2015). Neural networks are useful with reinforcement learning as they can learn to approximate value functions for incredibly large state spaces. Manual calculation of large state space tasks becomes infeasible and neural networks provide a resource saving solution. The approach described in this paper uses the DQN approach to learn the state space and make optimal decisions.

Related Work

The related work is focused on three main areas: the open-set classification task, incremental learning clustering, and reinforcement learning applications for open-set classification.

Open-set classification for neural networks is the ability to identify unknown class data and assign the data a new label with the purpose of building upon the network's knowledge. Recent work in this area have focused on augmenting the architecture of a neural network to be able to incorporate new knowledge. One such technique called the OpenMax method introduces a new model layer; this layer is used to estimate probability of the input data to be unknown (Bendale and Boult, 2016). The estimation is performed by measuring the distance between an input's activation vector and comparing it with the model vector for a few top classes. The approach proposed by this paper will focus on the idea of input vectors derived by the neural model to derive clusters.

Clustering is the process of combining sets of data to from a common group. There are numerous clustering algorithms each with various advantages, however there has been little research to measure the efficiency of these algorithms in an incremental learning task (Xu and Tian, 2015). One method recently adapted for this task is called δ-open set clustering (Wang *et al.*, 2018). This method is a topological approach to determine sub-cluster groups within sets of data in order to increase classification accuracy. This is greatly important when used for incremental learning as it helps reduces data forgetting. Another method uses Agg-Var clustering to show that different groups of data often have very different cluster spaces; this method solves this issue by creating a set of centroids to represent evolving clusters (Ayub and Wagner, 2020). Since cluster creation is non-deterministic, using a reinforcement learning agent will help produce clusters more fine-tuned for specific tasks.

The open-set classification task is often inadequate when used for incremental learning. The problem is from compounded error through incremental learning iterations. A neural algorithm proposed by Shiraga et al. shows that reinforcement learning can be used to help control "interference" caused by incrementally learning data over time (Shiraga *et al.*, 2002). Using a form of long-term memory, a learning agent is able to help identify data relationships to reduce error. Another approach developed by Bian and Jiang shows that reinforcement learning can be used to develop a continuous-time learning algorithm that can learn without the use of batched data (Bian and Jiang, 2019); this helps reduce error as batched data can incorporate misclassified mixed data. The agent used is also shown to reach convergence even with the presence of data disturbances. Using this idea, a reinforcement learning agent is able to be more resilient to data errors that compound in incremental learning.

Methodology

This section describes the implementation methodology as well as explains implementation choices made for the proposed approach. The methodology will follow a typical open-set classification approach but is amended to add a reinforcement learning agent. The steps are described are also shown in Figure 1.

Datasets Used

The proposed open-set classification approached is tested with many datasets. These datasets were chosen as they have multiple classes for known class training and unknown class testing.

- **CIFAR-10** (Krizhevsky *et al.*, 2009): This dataset consists of labeled images from ten different classes. The

classes are evenly represented with 60000 32x32 colored images per class.

Feature Extractor

The feature extractor model is used to obtain feature vectors for the data being used. The motivation behind using a feature extractor to obtain feature vectors rather than directly passing the data to the agent is so the characteristics separating the data classes are more distinguished (Perera *et al.*, 2020). Traditional clustering algorithms perform poorly in open-set classification because the full set of data is not pre-trained to the clustering model; this leads to the model suffering from inter-task confusion as the model never jointly learns the specific methods to divide all the classes (Masana *et al.*, 2020). The feature extractor helps alleviate this problem as the clustering algorithm can now easily focus on the exact distinguishable features. The model being used for this is the ResNet-152 structure pre-trained on the Imagenet dataset (He *et al.*, 2016; Deng *et al.*, 2009); this architecture was chosen as it has been successful as feature extraction in past literature (Ayub and Wagner, 2020). Initially before any training we first divide the dataset to a training set comprised of a set of classes to a specified index C_i and to a testing set comprised of a set of classes to a specified index C_n that includes the training class indexes as well as "unknown" additional classes, Equation 1.

$$\begin{aligned} TrainingClasses &= C_0...C_i \\ TestingClasses &= C_0...C_i...C_n \end{aligned} \tag{1}$$

The pre-trained feature extractor is then fine tuned on the training set. To evaluate the performance of the feature extractor we calculated the Euclidean Distance as shown in Equation 2 from vectors from the same classes and different classes. The results as shown in Table 1 show that the distances from feature vectors in the same class are are much lower than that of other classes.

$$d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{2}$$

	Class 0	Class 1	Class 2	Class 3	Class 4
Class 0	**0.027**	1.394	1.323	1.395	1.388
Class 1	1.411	**0.0008**	1.360	1.413	1.411
Class 2	1.401	1.403	**0.088**	1.402	1.400
Class 3	1.411	1.412	1.358	**0.002**	1.409
Class 4	1.412	1.413	1.314	1.413	**0.002**

Table 1: Euclidean Distance differences between different generated vectors for 5 classes. The values are low for vectors corresponding to different samples from the same class. This shows the feature extractor model is able to generate distinguishable vectors.

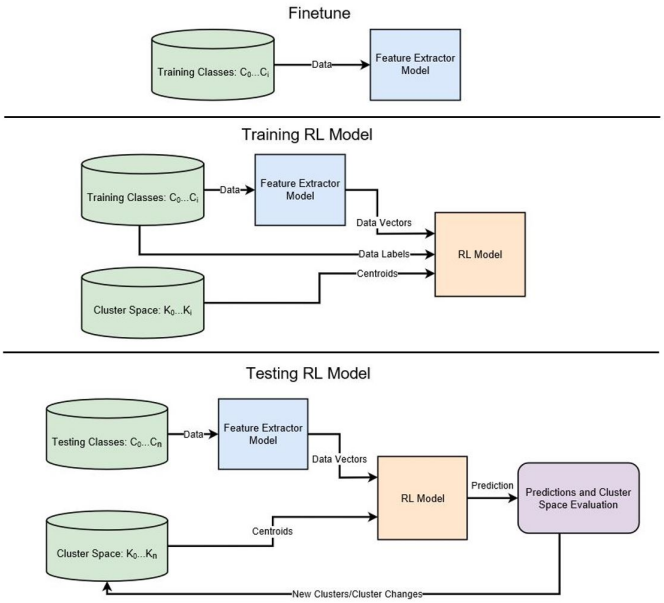

Figure 1: Diagram showing the individual steps of Finetuning the Feature Extractor Model, Training the RL Agent Model, and Testing the RL Agent Model.

Cluster Space

The cluster space is the space representing all the trained classes with respect to each other class. The space initially contains the training data and is later updated by the agent as new data is identified. The cluster space K_T is comprised of various cluster points K_i, each of which represents a number of data samples relatively close. Each class can be represented with more than one cluster as classes are not necessarily centered around one local mean, rather they can have different localizations based on unique features (Ramesh *et al.*, 2018).

During the training phase all the samples pertaining to one class C_i are averaged based on a certain distance threshold such that local average points are obtained, these become the clusters for class C_i. This is repeated for all the training classes and becomes the initial cluster space. The RL agent trains on this cluster space as this essentially represents the state space. During the testing phase data is presented incrementally to the agent and the agent makes a choice for all the incoming data, the specific clustering steps are explained below (Ayub and Wagner, 2019).

Create New Cluster For a data sample x_i, if the agent chooses to create a new cluster then this sample's vector becomes the new centroid point. The centroid's classification label is calculated based on a voting scheme to create a set of votes V between the group of m closest centroids. If the new centroid is not voted consistently by a large majority then the centroid is assigned a new class label. This is shown in Equation 3.

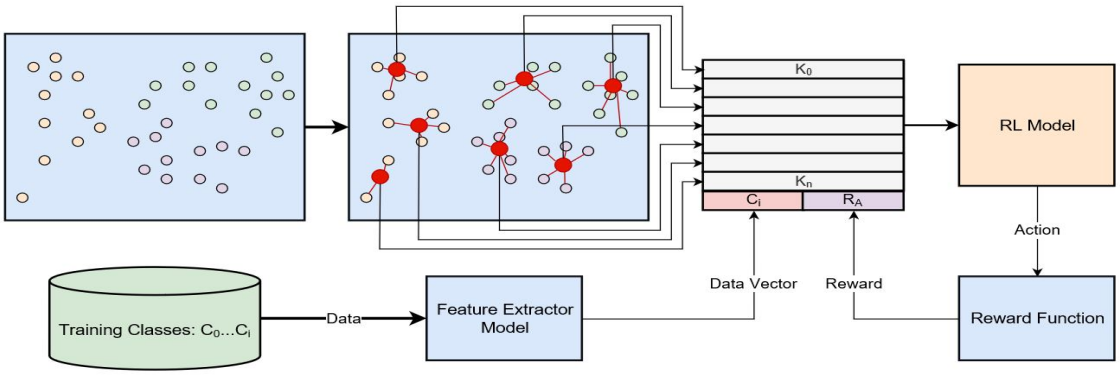

Figure 2: Diagram showing the clusters being created for the set of data and formed to an input for the RL agent. The centroid vectors are appended with their class labels. The input shows a representation of the current state.

$$V = \{K_i \mid min(K_i - x_i),\ |V| < m\}$$
$$K_T \cup x_i \leftarrow max(V) \tag{3}$$

Add to Existing Cluster For a data sample x_i, if the agent chooses to add to an existing cluster then the closest centroid point K_i is weighted averaged with the new data sample vector. This is shown in Equation 4 where w_{Ki} is the number of data points covered by the centroid K_i.

$$K_i = \{K_i \mid min(K_i - x_i),\ |K_i| = 1\}$$
$$K_i^{new} = \frac{w_{Ki} * (K_i^{old} + x_i)}{w_{Ki} + 1} \tag{4}$$

The motivation behind comparing input data to existing learned data is due to concept formation and biological learning methods. The hippocampus forms new knowledge by organizing novel information in space defined by existing learnt experiences (Theves *et al.*, 2020; Ayub and Wagner, 2020). After being presented with new information, a *memory-based prediction error* is calculated to represent the overlap between the new information and the previously known information. In the proposed approach, the distance calculation between the data sample x_i and the set of centroid points K_T allows the new information to be shaped based on the model's existing knowledge.

Reinforcement Learning Model

Motivation The reinforcement learning model is the main component of the approach as it makes the classification decisions. The model will train on the initial cluster space and learn the relationships between the centroid points of the different classes. The motivation behind using a reinforcement learning model to make predictions rather than having a fixed distance calculation is that the relationships between different classes are not always the same; this means any fixed distance parameter would not produce optimal results. This would also affect models tested on mixed datasets as there would be no predictable way to know how the classes would be represented in the cluster space. Using a reinforcement model that constantly updates it's knowledge to find optimal solutions would solve this issue.

Another reason for using a reinforcement learning agent is that it performs clustering for truly unknown data. A lot of traditional clustering approaches need to know the number of classes that are included in the clustering set. However, this information is not known in scenarios where a model is exposed to known data. The use of the RL agent and the multi-centroid based approach allows the model to determine the number of unknown classes as well as the the samples pertaining to each class.

Neural Network RL Algorithm The algorithm used for the RL agent is the Deep Q-Learning (DQN) approach (Mnih *et al.*, 2015). The DQN approach is based off on the Q-Learning algorithm (Watkins and Dayan, 1992) which calculates a value for each state the agent can go through.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma * max(Q(S_{t+1}, a)) - Q(S_t, A_t)] \tag{5}$$

In Equation 5, we can see that the agent would make an action and would update the value Q of the state based on the associated reward. This allows the agent to learn an optimal policy of making decisions based on a given state. The DQN approach uses a neural network to approximate the Q value; this is useful because the state space is constantly different based on dataset length and trained classes.

The architecture used for the agent is a multi-layer convolutional neural network. The cluster space is represented by layering the centroid points' vectors, this is shown in Figure 2. The agent is trained at each step of the episode with the current state S_t, the next training data sample x_i, and the reward based on the agent action R_t. The reward function used is kept simple as there are only two actions the agent can perform, a reward of +100 is used for the correct action and a reward of +0 is used for the incorrect action.

Since neural models typically expect a fixed input size, the number of centroid points need to be limited. In the event that there are a large number of centroids $|K_T| > Z$, the centroids that represent the least number of data samples

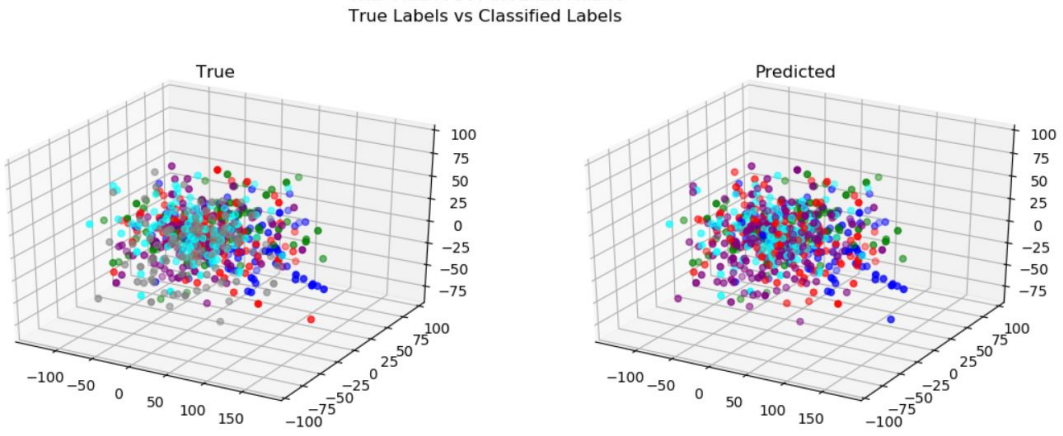

Figure 3: Initial clustering results of the RL agent for the CIFAR-10 dataset. The diagram shows the true classifications and the predicted classifications for training with 5 classes and testing with 6 classes. The average Raw Accuracy is 83.33% and the Incremental Class Accuracy is 81.20%. The plots only show 200 samples per class of the entire dataset, however the accuracy scores were calculatd with the full dataset.

will be weighted averaged to the nearest same class centroid. To avoid this problem the initial limit Z used by the constructed RL model will be set to a high value. Once the model is trained, it can take new data and divide the samples into new classes based on their representation in the cluster space. Then once the cluster space is updated the RL agent can train on the space again; this means the agent moves to an incremental learning structure capable of learning new data from the environment.

Evaluation Methods

The main area for evaluation is the clustering accuracy as the reinforcement learning agent will be using the clustering results for learning rewards. Also any further applications of the clustering classifications such as a continuous incremental learning algorithm will also be directly dependent on the accuracy. Another metric that will be used in the evaluation is the *Incremental Class Accuracy* (ICA); this metric is specifically designed for open-set clustering and measures multiple variables (Leo and Kalita, 2020). The ICA metric is shown in Equation 6; this metric is an average of homogeneity $\frac{max(n_{c|k})}{N_k}$, completeness $\frac{max(n_{c|k})}{N_c}$, and unknown identification $\frac{(n_{u|k})}{N_k}$ accuracies.

$$ICA = \left(\frac{max(n_{c|k})}{N_k} + \frac{max(n_{c|k})}{N_c} + \frac{(n_{u|k})}{N_k} \right) * \frac{1}{3} \quad (6)$$

Experiments and Results

This section explains the experiments performed as well as the results obtained. The results show the performance of the RL agent as well as the efficiency of the clustering technique used for the data.

RL Model Evaluation

For the RL model, first the training dataset had to be aggregated with the centroid space such that the input data included the evolving state space and new samples from the training data. This means as the RL model trains the centroid state space is also evolving. Once the model trained on the training set $C_0...C_i$ for a large number of episodes, the model was then tested with the testing set $C_0...C_i...C_n$ that included both trained classes as well as some variable unknown classes. Based on this experiment we calculate two accuracy measures: the *Raw Accuracy* which measures if the RL agent correctly identified known and unknown classes, and the *Classification Accuracy* which measures the final accuracy of the labeled data with class labels. For the Classification Accuracy we are using the *Incremental Class Accuracy* metric as shown in Evaluation Methods Section. The results are shown in Figure 3. The RL model training accuracy and loss is shown in Figure 4

Conclusion

In this paper, we propose a method to perform open-set clustering using a reinforcement learning agent. This will be done by using a training set of data to teach an agent the nuances between different classes. The agent would then perform with greater accuracy than regular clustering algorithms as the agent would learn specific features of specific datasets. The trained agent is then tested against the testing set of data which includes data from additional datasets as well. The agent is then evaluated using traditional accuracy measures as well as the ICA score. The proposed method is also then shown to perform better in clustering needed tasks such as some forms on incremental learning.

For future work, a lot of additional enhancements as well as experiments can be run. The main problem with the results observed is that the RL agent confuses some specific classes together. To fix this, the training algorithm can be

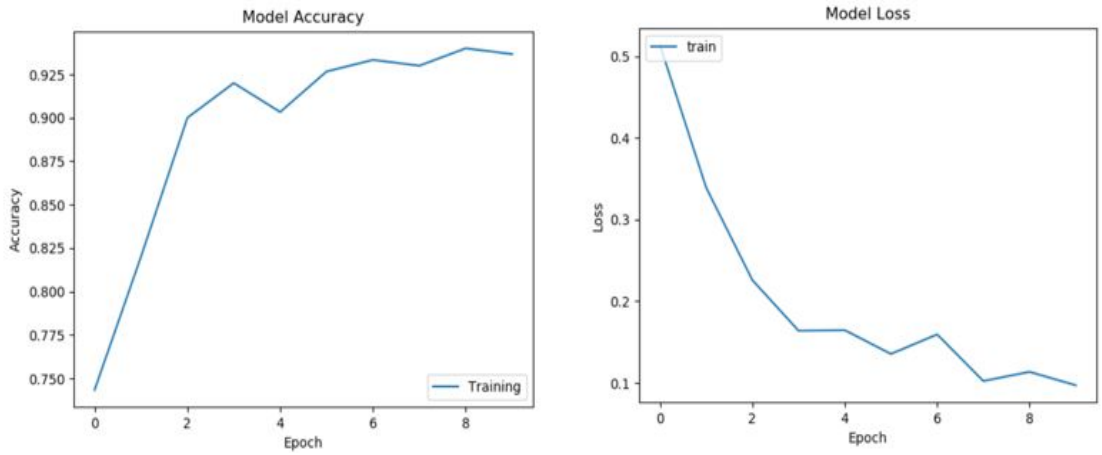

Figure 4: Plots showing the changing accuracy and loss for the RL model training. Each epoch has multiple training steps as the model is updated after each agent action.

modified to include more clear unique data representations with the state space. We can also move the algorithm to test incremental learning such that the RL agent is trained with the new data once it identifies unknown data. This will allow testing the model on multiple iterations of learning to see if the algorithm performs well in continuous evolution.

References

Ali Ayub and Alan Wagner. Cbcl: Brain inspired model for rgb-d indoor scene classification. *arXiv preprint arXiv:1911.00155*, 2(3), 2019.

Ali Ayub and Alan R Wagner. Cognitively-inspired model for incremental learning using a few examples. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 222–223, 2020.

Abhijit Bendale and Terrance Boult. Towards open world recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1893–1902, 2015.

Abhijit Bendale and Terrance E Boult. Towards open set deep networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1563–1572, 2016.

Tao Bian and Zhong-Ping Jiang. Reinforcement learning for linear continuous-time systems: An incremental learning approach. *IEEE/CAA Journal of Automatica Sinica*, 6(2):433–440, 2019.

Antonio Coronato, Muddasar Naeem, Giuseppe De Pietro, and Giovanni Paragliola. Reinforcement learning for intelligent healthcare applications: A survey. *Artificial Intelligence in Medicine*, 109:101964, 2020.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A Al Sallab, Senthil Yogamani, and Patrick Pérez. Deep reinforcement learning for autonomous driving: A survey. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

Justin Leo and Jugal Kalita. Moving towards open set incremental learning: Readily discovering new authors. In *Future of Information and Communication Conference*, pages 739–751. Springer, Cham, 2020.

Justin Leo and Jugal Kalita. Incremental deep neural network learning using classification confidence thresholding. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

Marc Masana, Xialei Liu, Bartlomiej Twardowski, Mikel Menta, Andrew D Bagdanov, and Joost van de Weijer. Class-incremental learning: survey and performance evaluation on image classification. *arXiv preprint arXiv:2010.15277*, 2020.

Marvin Lee Minsky. *Theory of neural-analog reinforcement systems and its application to the brain-model problem*. Princeton University, 1954.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

Nils J Nilsson. *Principles of artificial intelligence*. Morgan Kaufmann, 2014.

Pramuditha Perera, Vlad I Morariu, Rajiv Jain, Varun Manjunatha, Curtis Wigington, Vicente Ordonez, and Vishal M Patel. Generative-discriminative feature representations for

open-set recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11814–11823, 2020.

Chundru Raja Ramesh, Komaragiri Raghava Rao, and Gunamani Jena. Fuzzy clustering algorithm efficient implementation using centre of centres. *International Journal of Intelligent Engineering and Systems*, 11(5):1–10, 2018.

Naoto Shiraga, Seiichi Ozawa, and Shigeo Abe. A reinforcement learning algorithm for neural networks with incremental learning ability. In *Proceedings of the 9th International Conference on Neural Information Processing, 2002. ICONIP'02.*, volume 5, pages 2566–2570. IEEE, 2002.

Yoav Shoham, Rob Powers, and Trond Grenager. Multi-agent reinforcement learning: a critical survey. Technical report, Technical report, Stanford University, 2003.

Stephanie Theves, Guillén Fernández, and Christian F Doeller. The hippocampus maps concept space, not feature space. *Journal of Neuroscience*, 40(38):7318–7325, 2020.

Shuliang Wang, Qi Li, Hanning Yuan, Deren Li, Jing Geng, Chuanfeng Zhao, Yimeng Lei, Chuanlu Liu, and Chengfei Liu. δ-open set clustering—a new topological clustering method. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(6):e1262, 2018.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

Dongkuan Xu and Yingjie Tian. A comprehensive survey of clustering algorithms. *Annals of Data Science*, 2(2):165–193, 2015.