CSCI 5512: Playing Snake with Reinforcement Learning

David Ma
maxxx818@umn.edu

May 14, 2020

Abstract

Game playing has always been a major part of Artificial Intelligence research, and creating game-playing agents for increasingly complex games has historically driven a significant body of innovation in the AI field. In this project, I seek to build an agent to play the classic video game Snake. While simple compared to games like Go, Chess, and Starcraft, the game is complex enough to allow for an interesting application of reinforcement learning. I first build a representation of the game in Python and then seek to apply deep reinforcement learning to the model with a Deep Q Network (DQN), comparing the learned policy with a random model and human player. To improve the model, I experiment with various hyperparameters and reward functions, trying to make the results as close to optimal gameplay as possible. Finally, I discuss the results of the testing and further improvements to the algorithms.

1 Introduction

Since games typically have defined actions and states based on their rules, their limited nature makes them easy targets for programming simulation. Although solving these games do not inherently have utility to human society, the research that goes into solving these games can often be used to develop new algorithms or techniques for machine learning and artificial intelligence. One of the most recent examples was when Deepmind built an AI to play Starcraft 2 at a master level, an incredibly complex competitive game. As noted in [12], the ability of Deepmind to play Starcraft 2 suggests that machine learning algorithms could be very useful in tackling real-world problems.

Unfortunately, as I have neither the computing resources nor the capabilities of Google, I had to choose a substantially smaller problem. In this project, I seek to apply game playing algorithms to play the classic arcade game, Snake.

For those unfamiliar, Snake is a game where the user controls a 'snake,' a sequence of blocks. They guide the sequence around the screen with the arrow keys, trying to avoid crashing into walls or the body of the snake itself, all the while collecting apples to grow its body.

The goal was to create an agent that could fulfill the two objectives of the game - collect apples and stay alive - using the four actions available to it - up, down, left, and right. Because the game is single-player, the game-playing agent does not have to worry about any other agents opposing its objectives. Almost all states contain perfect information, since the agent can see its own position and the position of the apple. However, the apple will randomly spawn in a new location after being eaten, adding an element of randomness.

While hard-coding an agent to play Snake optimally wouldn't be too difficult due to the simplicity of the game, the goal of this project was to create a model-free agent; that is to say, an agent that can learn the optimal game-playing policy by trial and error, seeking to maximize its own reward without actually understanding the structure of the game itself. The hope is that through reinforcement learning, the agent will somewhat learn the structure of the game and its various states. This is a simple application of reinforcement learning, but it was an illuminating implementation of a state-of-the-art RL baseline. The difficulty of model-free problems is largely in how the agent learns all the features that define the game, without any external guidance.

To approach the game, I first created a simple implementation of the game in Python, because it was difficult to use an existing implementation for this purpose. Once given this implementation, I created an agent to manipulate the environment, before using Deep Q Learning to train for an optimal policy. Given a finished model, I looked for ways that I could improve the policy and where the training might be lacking through the various hyperparameters. I evaluated the policies subjectively, based on how I thought the agent was playing, as well as objectively, based on the in-game statistics.

In the following sections, I first discuss the general field of game-playing and then the more specific field of reinforcement learning. I explain my approach, the simulation set-up, and then the results of the project. Finally, I finish with ideas for future improvements and my overall conclusions.

2 Related Work

2.1 An Overview of Games and AI

As an overview of games and artificial intelligence, [15] writes that games offer an interesting challenge for the AI field because they are both **hard** and **interesting**. Because games can be incredibly complex to solve even to humans, the ability for artificial intelligence to play complex games well can be used as a benchmark to see how capable it is of making difficult decisions. As AI has advanced, it has surpassed humans in increasingly difficult games: Checkers, Chess, and more recently Go.

At the most general level, game playing agents can be divided into two different categories: general game playing agents and agents created to play specific games. Initially, most agents were built to play very specific games - as written in [15], the most famous example of this was probably when DeepBlue beat Gary Kasparov in chess. However, as noted by [1], something like DeepBlue is specified to do one task, and one task only. Therefore, it can be more illuminating to work with general game playing agents, which are agents which are not tailored to a specific game. Rather, they can take in declarative descriptions of different games and play each of them effectively without human intervention. Reinforcement learning borrows from such ideas. With model-free learning, the agent is placed into an uncertain environment and learns the environment, as opposed to having any predetermined knowledge.

2.2 Perfect and Imperfect Information Games

I will also briefly discuss two broad categories of games: perfect and imperfect information games.

Perfect information games are just as they sound. [5] notes that in perfect information games such as Chess and Checkers, both players have complete access to the entire state of a game. This makes it theoretically possible for each player to calculate the absolutely optimal solution to each state.

By comparison, as written in [5], in imperfect information games each player does not have full access to all the information available, therefore introducing elements of probability and guessing. The most famous example of an imperfect information game analyzed in the field of AI would be Poker, since no players can see their opponents' hole cards at any given time. As a result, there is no perfect action to take for any state and algorithms that might solve perfect information games might not be entirely applicable to imperfect information games.

Snake is largely a perfect information game, as there is a possible optimal solution in every scenario. However, the question is whether the agent can learn this optimal solution.

2.3 Alternative Algorithms for Game-playing

Since reinforcement learning is not the only way to approach solving games with an agent, I will briefly touch on other algorithms that have been used to solve games. Notably, simple games such as Tic Tac Toe can be solved with tree-based search algorithms such as MiniMax and its offshoot Alpha Beta Pruning, as described in [10]. These algorithms are not based on an agent so much as they're based on heuristics, seeking to calculate the optimal move at any given state instead of following a predetermined policy. Similarly, for more complex games, sampling algorithms such as Monte Carlo Tree Search use rollout methods to find optimal steps.

2.4 Reinforcement Learning for Game-playing

As noted in [4], the strongest allure of reinforcement learning is in its promise to train agents using only reward and punishment, as opposed to specifying how the task is actually to be achieved.

Although the fundamental concept of reinforcement learning is simple, there are a number of ways that it has been implemented and explored. One of the most common implementations is in Q-learning [14], which has an agent converge to an optimal policy under some environment. Specifically, Q-learning works by having an agent try many actions in many states, and then updates its Q-values based on both its immediate reward and its expected future reward. By doing so, a Q-learning agent can calculate its total expected reward, which it seeks to maximize.

However, pure Q-learning has a significant number of limitations. Specifically, it becomes more and more difficult to represent states and calculate optimal policies for every single state as the state space gets large and even continuous. For example, it can be very difficult to choose an optimal policy based on an image input. However, with the advent of stronger machine learning and neural networks, Q-learning has been combined with deep learning to create Deep Q Networks (DQN). One of the best examples of this is in in [6], where the authors trained a convolutional neural network with Deep Learning to play seven Atari games to a significant degree of success. The network took in raw pixel input and outputted a value function estimating future rewards. Similar to pure Q-learning, the output estimate can become a solid indication of future reward after training.

In recent years, the combination of neural networks and reinforcement learning has shown a significant degree of success, as shown by programs such as AlphaZero [9]. Specifically, state of the art algorithms such as AlphaZero replace the handcrafted features of the past with deep

neural networks, general reinforcement learning algorithms, and a tree-search approach. The neural network represents a board state with learned features and uses it to output a movelist, with each move associated with a certain utility. Innovatively, it learns all of these features uniquely from self-play, through nothing but trial and error.

3 Approach

3.1 Motivations and Initial Ideas

For this project, I mostly just wanted to try to implement a DQN, as a chance to apply the concepts that we covered in class. At their core, most of the state of the art baseline RL methods combine deep learning and reinforcement learning algorithms, so I thought this would be a useful project to experiment with these concepts.

Originally, I had wanted to do something more visual, such as playing the Google Chrome Dinosaur game using Selenium. The problem I ran into was that anything relying on an actual visual representation would need to be run on my own computer, which made training the network incredibly slow. Because I need to use Google Colab for its GPU capabilities for training a network, I ended up having to implement a non-visual implementation of a game, as opposed to being able to use screenshots of some other game.

I was strongly inspired by a Medium article by Pratyush Khare titled 'Reinforcement Learning — Chrome Dino Game (Using Deep Q-Learning),' which inspired much of the structure for my work. But since I worked on a different game and a different implementation, I ended up branching out quite a lot.

3.2 Game Environment

Before implementing any algorithms, I needed an environment to test those algorithms on; that is to say, I needed a representation of the game Snake. I looked online for a number of implementations, but most of the online ones were done in PyGame, and I needed a non-visual implementation. At the end of the day, I modified a simple online tutorial to create the environment that I needed.

The game state is represented on a 20 by 20 board. The Game class tracks the position of the 'snake' in a list of tuples, where each tuple is the (x,y) position of one of the blocks of the snake. Other than that, the Game class also tracks the position of an apple with a (x,y) position. The class has methods for turning the snake in the four cardinal directions, as well as for checking for collisions against its own body, the apple, or the edges of the grid. Upon colliding with an apple, the snake grows in length by one unit. Finally, the class has a number of helper functions for getting the status of the game, the positions, and any relevant variables. The core of the Game class is the update() function, which does all of the collision checking and advances the game from one state to the next.

Lacking an actual visual representation of the game, the following figure is a reasonably close depiction of what the game would look like if it was visualized.

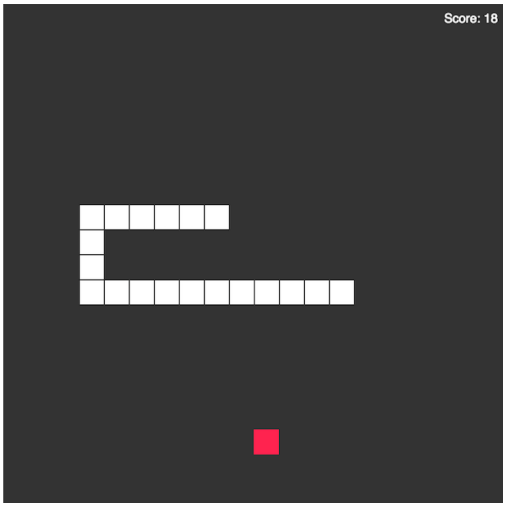

Figure 1: Snake Game

3.3 DQN Algorithm

Obviously, the primary algorithm that I used was Deep Q Network, or DQN for short. Pseudocode of the algorithm is shown below, courtesy of DeepMind.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 2: Before Processing

The implementation of my code was almost identical to the pseudocode shown above. The one main difference was that I only used a single loop of iterations, instead of breaking it down into episodes. Other than that, I did pretty much the exact same thing. I used Mean Squared Error for loss and Adam for optimization, all included with Pytorch; I made those two choices mostly based on sample code.

At its core, Deep Q-learning is pretty similar to normal Q-learning. It's notable that when y_j is

being set in the above pseudocode, it effectively follows the Bellman equation, summing the current reward and the discounted future reward. The most fundamental difference is that the y_j is used for gradient descent instead.

The advantage of Deep Q Learning is that it can estimate increasingly complex input states, even being able to represent continuous states. As mentioned in [3], DQN allows one to approximate states with a neural network instead of maintaining a table of Q values. Thus, for Q, the action-value function, I initialized a convolutional neural network with PyTorch. The details of the network will be described in the next section.

There were a number of parameters available for tuning, as listed below:

- ϵ: The exploration rate. That is to say, the probability that the model will choose an action other than the one calculated to be optimal. This is important because one of the biggest factors in reinforcement learning is managing the tradeoff between exploration and exploitation. If ϵ is too low, it's easy for the network to converge to a suboptimal solution.

- T: The number of iterations. This is pretty simple, but T should be large enough that the policy converges to the optimal policy.

- γ: The discount factor. γ is used in the value calculation equation and it determines how heavily future states are valued. The higher the γ, the higher future states are valued. A low γ, on the other hand, means we focus more on the current state.

- N: The replay memory capacity. This is relevant to Experience Replay, a concept associated with DQN.

3.4 Experience Replay

Experience Replay is a method meant to help solve some of the problems associated with DQN. Instead of performing a network update at every step, it stores the values associated with each step in memory and samples from them randomly. This means that each state in memory is not associated with its immediate preceding and succeeding states, and also that those stored memories can be used multiple times. All in all, it's an innovation that can be greatly useful in RL, as written in [8].

3.5 Neural Network

For Q, the action-value function, I used a convolutional neural network built on Pytorch. The structure of my network wasn't as deep as the one in [6], since I didn't think I had nearly as many features to extract. Because of the nature of how my game was represented, the input to the neural network was a 20 by 20 tensor, where most of the tensor values were just 0; if I had been taking in raw pixels as input, it would have been somewhere along the lines of 84 by 84.

Thus, I used a network with two convolutional layers and two fully connected linear layers; the code to initialize it is copied below.

```
self.conv1 = nn.Conv2d(4, 32, 8, 2)
self.relu1 = nn.ReLU(inplace=True)
self.conv2 = nn.Conv2d(32, 64, 4, 2)
self.relu2 = nn.ReLU(inplace=True)
```

```
self.fc4 = nn.Linear(256, 512)
self.relu4 = nn.ReLU(inplace=True)
self.fc5 = nn.Linear(512, self.number_of_actions)
```

Listing 1: NN Code

3.6 Training

I attempted training a number of times using Google Colab, leveraging their inbuilt GPU capabilities. Relevant hyperparameters include:

- ϵ: The exploration rate. I started with .1 but tried values all the way up to .5. Ultimately, I settled on $\epsilon = .2$
- I set the final ϵ to be .0001, which means ϵ would decay to .0001 over T iterations.
- T: The number of iterations. More iterations is usually better, but also takes more time. I started off with 200,000 iterations, but tried a run on 500,000 iterations as well.
- γ: The discount factor. I tried anywhere from .9 to .99, but ended up settling on $\gamma = .95$
- N: The replay memory capacity. I settled on a memory capacity of 10,000.
- I set batch size to be 32

On average, it took around 6 seconds for every 1000 iterations, which meant that 500,000 iterations took around 50 minutes. One problem that I faced when training the DQN was that Google Colab only allows a limited amount of time per notebook, which means that I wouldn't be able to train it for any lengthy period of time. This wasn't a huge deal, since the policy converged anyways; however, if I was looking to improve the network with methods that took more time in the future, I would probably need to invest into a GPU.

4 Results

I found a degree of success with the DQN training, but it definitely wasn't optimal. Figure 3 on the next page is a graph of the max Q-values over the iterations:

While they did fluctuate up and down a fair amount, and I'm not sure what the massive spike around 200,000 was, it can be seen that the values steadily increased and then converged to around 40. This is a promising sign, and means that the policy is slowly improving.

The other main metric by which the learned policy can be evaluated is how well it actually played the game of Snake, which is where it didn't do very well. I compared the policy against a random agent as well as against myself playing the game. Results are shown in table 1 below.

A random agent obviously did very poorly, since it had no concept of avoiding walls and had no concept of looking for the apple either. Most of the time, it would just die without managing to pick up the apple a single time. The learned policy did a little better, being able to get an apple or two. However, it didn't even come close to playing as well as a mediocre human (me) could.

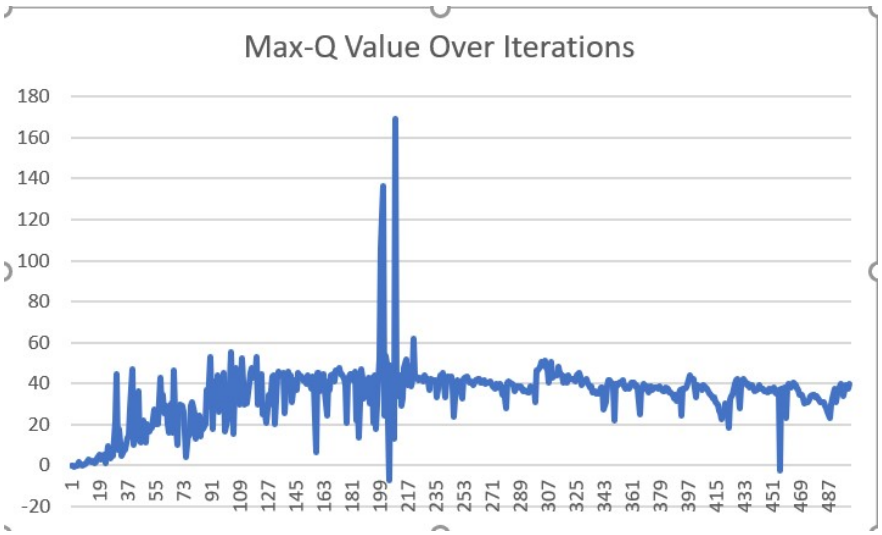

Figure 3: Q Values

	Average Score
Random Agent	1.05
Me	12
Learned Policy	2.6

Table 1: Table Comparing Policy Learning to Other Agents

4.1 Discussion

The biggest problem with the agent was that it would get stuck in loops, seeking to stay alive instead of actually seek out the apple. The optimal policy can pretty consistently seek out the first fruit and consume it, but after a new fruit spawns, it doesn't seem to prioritize the new fruit very highly. I think this might be a problem with the way that the game is represented, and I'm not sure whether or not the neural network is placing enough weight on the fruit. This is a substantial obstacle towards making the agent play well, because while staying alive is the most important objective in Snake, consuming fruit is almost equally important.

I tried to approach this problem by tuning the reward function, increasing the rewards of the fruit while penalizing the agent for idling without accomplishing anything, but it didn't do too much to help the issue.

Overall, though, it's clear that the agent is at least doing some work in learning the optimal snake-playing policy, both through the game scores and in its increasing Q values. While it's not close to perfect, this is a strong indication that its model-free assessment is succeeding to some degree.

5 Future Work

There are a number of improvements that can be made to the approach, both on a theoretic level and in response to its current issues.

For now, the biggest challenge to tackle is in having the policy place a higher weight onto

actually consuming the fruit. If I had more time, I would look into two things: the Neural Network representation and structure, as well as the reward function. I could also look into tuning the hyperparameters a little bit more, but I don't think that the problem is there.

On a higher level, there is a whole family of techniques available in DQN that could merit implementation. For example, we have Double Q Learning [11], prioritized replay [8], dueling networks [13], multistep learning [7], distributional RL, and noisy nets. Altogether, I would want to look into the Rainbow Algorithm [2], which effectively combines a spectrum of modern DQN techniques.

6 Conclusion

In this paper, I described an implementation of the classic game Snake in Python, and also attempted to train an agent to play it using a Deep Q Network. I implemented DQN using a convolutional Neural Network in Pytorch, training the CNN to estimate the value of various state-action pairs in Snake. The approach was able to result in a moderate degree of success, showing improvement in its policy but not being able to outperform any significant baselines.

Overall, this project probably did nothing to contribute to the general AI knowledge base, since it only uses existing methods and cannot outperform state of the art baselines. However, I did find it quite useful as a learning experience and I think it taught me a lot about more modern approaches to agent AI and reinforcement learning.

In the future, I would want to apply more of the state of the art DQN techniques to the model while looking for even more places to innovate. Although the project wasn't perfectly successful, I think it shows a decent amount of promise for the potentials of Reinforcement Learning.

References

[1] M. Genesereth, N. Love, and B. Pell. General game playing: Overview of the aaai competition. *AI magazine*, 26(2):62–62, 2005.

[2] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver. Rainbow: Combining improvements in deep reinforcement learning. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[3] T. Hester, M. Vecerik, O. Pietquin, M. Lanctot, T. Schaul, B. Piot, D. Horgan, J. Quan, A. Sendonaris, I. Osband, et al. Deep q-learning from demonstrations. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[4] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[5] D. Koller and A. Pfeffer. Generating and solving imperfect information games. In *IJCAI*, pages 1185–1193. Citeseer, 1995.

[6] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[7] J. Peng and R. J. Williams. Incremental multi-step q-learning. In *Machine Learning Proceedings 1994*, pages 226–232. Elsevier, 1994.

[8] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

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

[10] G. Synnaeve. Bayesian programming and learning for multi-player video games: application to rts ai. *Doctor of Philosophy, Grenoble University*, 2012.

[11] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[12] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

[13] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

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

[15] G. N. Yannakakis and J. Togelius. *Artificial intelligence and games*, volume 2. Springer, 2018.