Naggin' Naagin: A Self Learning Snake

Don Kurian Dennis & Ritobroto Maitra

1 Abstract of the project

Q-Learning is a fundamental approach when dealing with reinforcement learning problems wherein the problem can be modeled as a Markov Decision Process. Using Q-Learning for models with finite decision states, we are able to find the expected utility of each action at an MDP state over multiple training episodes and accordingly find the optimal course of action. This project aims to use an application of a variation of Q-Learning, augmented with deep learning, to play the game Snake. We use approximate Q-Learning where the states are approximated as a linear function of certain specific parameters. We use the power of deep learning to extract these parameters (or features) from higher level sensory data like the pixel information of the game.

1.1 Data Sources

We do not need any additional data for this project. The game agent explores its environment and learns on its own.

1.2 Background

The idea of reinforcement learning [3][2] itself is not new - it is inspired by notions in behavioural psychology, which try to learn from an environment by interacting with it and by defining rewards for these interactions. More formally, it is concerned with how an agent performs actions in an environment to maximize rewards, and minimize regrets. Q-Learning [4] can be used to find an optimal policy of selecting the actions for a given finite Markov decision process.

A recent result from Google DeepMind [1] shows that it is possible to apply and combine the ideas of deep learning and Q-learning; where a model can successfully learn control policies from high-level inputs (such as a game score, and screens and so on). They applied these techniques to playing several games from the Atari 2600 series, and surpassed the best known human experts in half of them.

Our work tries to apply similar techniques to playing the popular retro game Snake [5] in a variety of constraints. Our work has the following phase plan:

- Implement the game in a simple GUI
- Model the game and the states
- Build various search based modeles
- Try to build a Q-Learning-based model
- Extend this to an approximate Q-Learning model
- Extend this to a Deep Q model

2 Work Done

We start with a simple implementation of the game snake, with different configurable mazes to control the difficulty level. The game itself is developed in an modular fashion so that all kinds of controlling or policy agents and other forms of tweaks can be easily implemented by simple class extensions or function overriding.

2.1 Search Based Agents

Our initial approach was to apply various search based agents to the snake game. There were two reasons for this.

Firstly, implementing a MinMax search based agent (and an ExpectiMax search agent) allows us to get an upper bound on the performance that can be attained using simple search algorithms. For the MinMax agent, the further (or deeper) we look into the search tree, the better the score we get in game play, with the trade-off of running time. The deeper we search, the more computationally intensive the search becomes. A depth 1 search works in real time with the run time bottle neck being the game logic or game state update rather than the search itself. A non GUI based game here finishes within seconds. A depth 7 search, on the other hand, gives us a much better total score but takes around 40 minutes on our systems to run. With a reflex agent, we saw an average score of 1645.450 with the average length of the snake being 59.125. With the same settings, running at depth 6 we got an average score of 1857.575 with the snake growing to a length of 47.400 on an average. It is interesting to note that there is a living penalty for each move of the agent currently set to +1. Hence, the agent that makes the minimum number of moves between subsequent food eating moves will have a higher gain in score per unit increase in length. The Reflex agent is able to gain a score of 13.164 per 1 unit increase in its length (1645.450/59.125), while the MinMax agent with depth 6 is able to perform much better returning 31.41 score points per unit increase in length. Hence, not only is the MinMax agent able to return a net total score, it also tries to follow the policy which minimizes the number of moves required.

The second reason for implementing search based agents is to test the features we have handcrafted for approximate Q-learning. The same feature extractor when used with weights engineered manually can be used with a reflex agent. This gives us an idea the impact each factor has on the game policy.

2.2 Overview of the Game State: Q Learning

Currently, we consider the following simple scenario for Q-Learning:

- The snake's state is mapped relative to the position of it's head.
- This divides the graphical space into four quadrants, and the snake moves based on the quadrant in which the food lies. We do this state compression because otherwise we have a huge number of states to keep track of: for every grid point, it may be empty, it may contain a portion of the snake or it may contain the food. That results in 61425 states for each move for even a small 5×5 grid.
- This is a very simple model, and it quickly converges to the policy of moving diagonally across the space towards the food.

This is expected; however we also observed certain anomalies. It turns out that the snake doesn't learn as fast as we expected, and it often thrashes around. Moreover, high reward states are hard to find; and the Q-value updates are extremely slow to propagate. This is even more so in our case since the rewards can be delayed significantly; since the number of moves between the *intent* to move towards the food and *actually* getting to the food can be arbitrary depending on the current configuration of the game board.

2.3 Approximate Q-Learning

2.3.1 Model

The state space in Q-learning grows exponentially with feature dimension, which makes it extremely difficult to implement practically. Instead, approximate Q-learning allows us to represent states, specifically the Q-Values of the states, as a linear function the value as a combination of features and weights. The weights determine the policy and it is learned over a number of game episodes. For each move within a game episode, it essentially performs stochastic gradient descent according to the following

equations, where Q is the Q-value, f_is are the features and the w_is are the weights, while s and a represent the state and action as usual.

$$Q(s,a) = \Sigma_i^n f_i(s,a) w_i$$
$$w_i \leftarrow w_i + \alpha[correction] f_i(s,a)$$
$$correction = (R(s,a) + \gamma V(s')) - Q(s,a)$$

The approximate Q-learning algorithm guarantees convergence assuming infinite state explorations. Clearly, this is not a feasibly exploration strategy and we use the epsilon-greedy algorithm to control the trade-off between exploration and exploitation.

We use this model, along with an epsilon-greedy approach, along with the following handcrafted features:

- Food Vicinity: This is the circular manhattan distance of the food from the snake's head.
- Gradient Factor: This is a measure of the snake colliding with itself in the direction (gradient) of its current movement.
- Collision Factor: This is a general measure of the probability of a snake entering into a collision in a particular state-action pair.

Note that the policy actually converges much before the values themselves do.

2.3.2 Results

We achieved what we can call partial success using these features - while the snake performed quite well, we had to tweak around the weights and feature values a lot to achieve these results. In the end, after a lot of (rather random than structured) attempts, we ended up building what we think is a fairly good configuration. We achieved these benchmark results: after 10 minutes of training over 2000 episodes, we achieved a mean score of 2177.14 with a mean snake length of 71.03, exceeding the previous best of 59.125 mean length which we had achieved using Reflex Agent. All of these results were benchmarked on a 20×20 grid. In fact, the maximum length of the snake in the trial runs we did often reached 100, with the maximum recorded length in our trials being 129. Towards the trailing end of the trial period, the snake consistently reached at least the 90 to 110 range. The scaled score grows significantly once the training period is over, and the weights also converge, as in the graphs below.

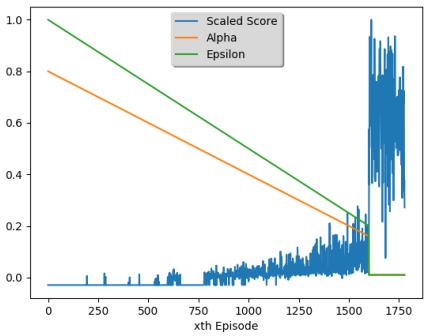

Figure 1: Scaled Scores During and After Training

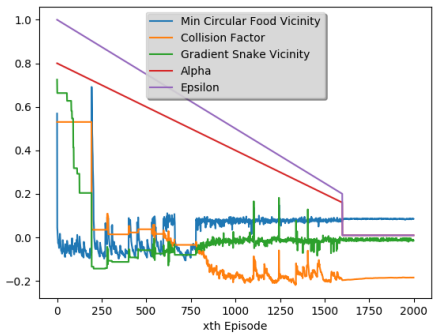

Figure 2: Convergence of the Weights

We can see convergence of weights around the 750 episode mark in the above graph. However, more interesting than what we achieved was what we observed. We delve into some of these in the next section.

2.3.3 Observations

Uncanny Correlations

Initially, when we were working with two features only (excluding the gradient feature), we observed an uncanny correlation in the values of the weights.

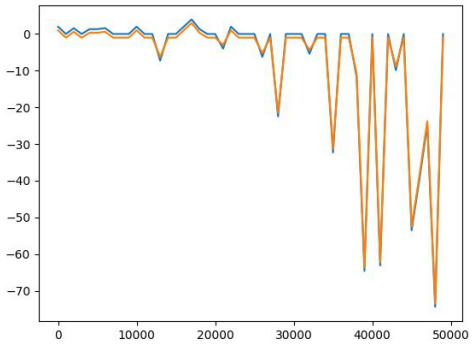

Figure 3: Highly Correlated Values of the Weights

This remains unexplained. Not only are the values correlated, sometimes they are uncannily similar with a difference of exactly 1.000000: the minimum and the maximum for one particular run were exactly 5706.265365 and 5705.265365; and 45000.122433 and 45001.122433 respectively.

Unbounded Growth

On certain initial values with two features, the weights grew unbounded. We tried to solve this using many different approaches, using normalization and tinkering with the features themselves - however, once again, we do not know what the underlying cause of this behaviour is. In fact, a top-level view of the data plotted actually makes it seem that the values converge.

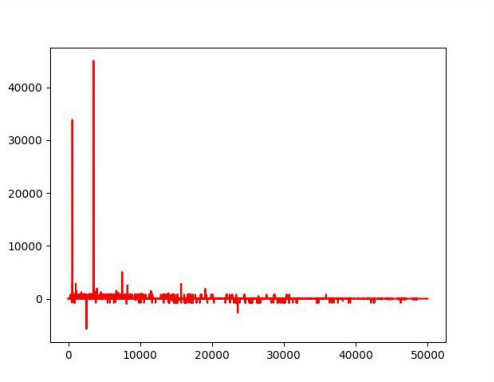

Figure 4: Values which apparently converge...

However, on analyzing this closely, we see that not only do they not converge, they in fact seem to have a ceiling beyond which the values do not grow. The plot in Figure 5 is using the same data set as Figure 4, but here we have only plotted every 1000th episode so as to keep everything in a reasonable scale.

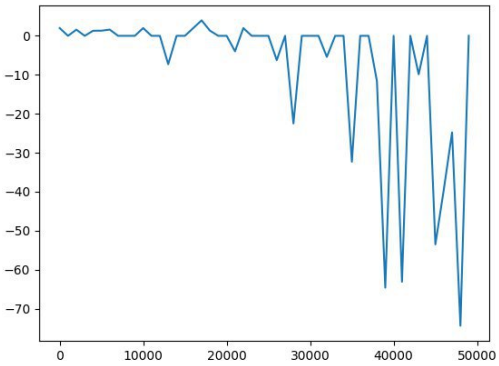

Figure 5: ... but actually don't

Thrashing

In certain episodes of the game, the snake often thrashes about in a loop without doing anything functional (ie either colliding with itself or eating the food). This is also not fully explained, although intuitively (not formally) we understand that the underlying cause has to do with two choices being equally likely in that state as far as the snake has learnt, and cannot move out of it in the same state.

Scaling Issues

We trained our snake on a 20×20 grid, and at optimum levels it covered about 25% of the space before a collision, by growing to a length of about 100 out of a total space 400 grid points. The 30×30 grid being larger, intuitively the snake has far more space to move about before colliding with itself - and it should seem as if the same policy should perform better. Unfortunately, it does not. Instead of achieving a length of about 225, which is 25% of the 900 grid size, it achieves about 120 to 130 - which is a clearly underperforming policy.

Surprisingly enough, the same policy performs far better in a smaller grid. In a 10×10 grid, it grows to about 35, covering about 35% of the grid. In a 5×5 grid with only 25 grid points, it consistently reaches lengths of 20, and almost always 18 - which means that it covers nearly 80% of the board and the episode has to terminate literally because there are often no legal moves left.

Theoretical Proof of Convergence

In the work by Google Deepmind [1], the rewards are near instantaneous. In Atari Breakout, for example, the delay between the reward and the action is 1 - the ball has to move and hit the brick. In

our case, the scenario is vastly different. It is worth considering whether or not this requires a far more theoretical treatment of the matter, investigating the validity of using exponentially decaying rewards with approximate Q-learning itself in the cases of extremely delayed rewards. It might make sense to replace exponential discounting with a linear decay. This line of thought requires further investigation.

3 Future work

As we have elaborated above, our exploration has thrown up several interesting questions. Our first work is to investigate each of these questions, and provide explanations and rationale for each of the behaviours we have documented above. We are hopeful that it will shed some light on the somewhat vague nature of the algorithms, and their interpretations in human-readable terms.

Then, we intend to extend our current work by using Deep Q-Learning approaches. Here, instead of handcrafting features as we have done so far, we extract higher level features from top-level inputs, such as a sensory input (of the pixel values of the grid itself, for example). We expect this to be significantly more difficult than what we have done so far, but it also promises to be a challenging and rewarding journey.

The entire code base of the project is available and will be updated as our work progresses at the following link:
CODE https://github.com/metastableB/Naagin-Naggin

References

[1] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). URL: http://arxiv.org/abs/1312.5602.

[2] Martijn van Otterlo and Marco Wiering. "Reinforcement Learning and Markov Decision Processes". In: *Reinforcement Learning: State-of-the-Art*. Ed. by Marco Wiering and Martijn van Otterlo. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 3–42. ISBN: 978-3-642-27645-3. DOI: 10.1007/978-3-642-27645-3_1. URL: http://dx.doi.org/10.1007/978-3-642-27645-3_1.

[3] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. 1st. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262193981.

[4] Christopher J. C. H. Watkins and Peter Dayan. "Q-learning". In: *Machine Learning* 8.3 (1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698. URL: http://dx.doi.org/10.1007/BF00992698.

[5] Wikipedia. *Snake (video game) — Wikipedia, The Free Encyclopedia*. [Online; accessed 22-March-2017]. 2017. URL: https://en.wikipedia.org/w/index.php?title=Snake_(video_game)&oldid=769462499.