COMP 138 RL: HW3

Harry Li

November 9, 2021

1 Introduction

For this homework, I did not choose a problem from the textbook and instead implemented the n-step Semi Gradient Sarsa algorithm [2] on a version of the video game Pac-Man. I created a line-of-sight state-action encoding and used linear function approximation to train an initial agent. Then as an extra experiment, I implemented a tree search state-action encoding algorithm with improved results. Finally, I discuss the results and possible future work.

You can see the trained agent in action at this website:
https://harryli0088.github.io/reinforcement-learning-pacman/

2 Background and Set Up

2.1 Pac-Man

Pac-Man is a popular video game in which the Pac-Man agent traverses a maze to eat all the biscuits (small white dots) and pills (large white dots). At the same time, Pac-Man must avoid all the dangerous ghosts, but can eat the edible ghosts after eating one of the pills.

I used Dale Harvey's implementation of Pac-Man [1] and modified it to make it easier to add RL training code. In this implementation, the four ghosts move randomly rather than having a specific behavior.

To reduce the number of actions the agent would need to take, I only allowed the agent to to select an action on a whole square (not in between two squares). I also set up the scenarios such that the episode terminates if Pac-Man eats all the pills and biscuits or is eaten by a dangerous ghost. When the episode terminates, the game is completely reset for the next episode.

2.2 Learning Algorithm

I implemented the episodic semi-gradient n-step Sarsa algorithm from [2]

Algorithm 1 Episodic semi-gradient n-step Sarsa for estimating $\hat{q} \approx q_*$ or q_π

Input: a differentiable function $\hat{q} : S \times A \times \mathbb{R}^d \to \mathbb{R}$, possibly π
Initialize value-function weight vector $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = 0$)
Parameters: step size $\alpha > 0$, small $\epsilon > 0$, a positive integer n
All store and access operations (S_t, A_t, and R_t) can take their index mod n
Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  Select and store an action $A_0 \sim \pi(\cdot|S_0)$ or ϵ-greedy wrt $\hat{q}(S_0, ., \mathbf{w})$
  $T \leftarrow \infty$
  For t = 0, 1, 2, . . . :
    If t < T, then:
      Take action At
      Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
      If S_{t+1} is terminal, then:
        $T \leftarrow t + 1$
      else:
        Select and store $A_{t+1} \sim \pi(.|S_{t+1})$ or ϵ - greedy wrt $\hat{q}(S_{t+1}, ., \mathbf{w})$
    $\tau \leftarrow$ t - n + 1 (τ is the time whose estimate is being updated)
    If $\tau \geq 0$:
      $G \leftarrow \sum_{i=\tau+1}^{min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
      If τ + n < T, then $G \leftarrow G + \gamma^n \hat{q}(S_{\tau+n}, A_{\tau+n}, \mathbf{w})$
      $w \leftarrow w + \alpha[G - \hat{q}(S\tau, A\tau, w)]\nabla \hat{q}(S_\tau, A\tau, \mathbf{w})$
  Until $\tau = T - 1$

With this approach, the agent starts with a set of weights that are combined with the state encoding to estimate the value of the state-action pair. I used a linear method to multiple a vector of weights with the vector state-action encoding, and used backpropagation to learn the weights.

I used these parameters:

- ϵ - greedy policy with $\epsilon = 0.1$
- Learning rate $\alpha = 0.001$ (higher values led to extremely unstable training and lower values did not seem to improve training)
- n = 3
- Discount factor $\gamma = 0.95$
- Weights all initialized to 0
- Number of Episodes = 500

3 Line-of-Sight Approach

3.1 Line-of-Sight State Encoding

I create a state-action encoding function that, given the current whole square and a valid available direction action, would look along that direction as far as it could see (until a ghost or wall blocked the view). The state-action is then encoded as a length-4 integer array with the:

- Number of dangerous ghosts
- Number of edible ghosts
- Number of pills
- Number of biscuits

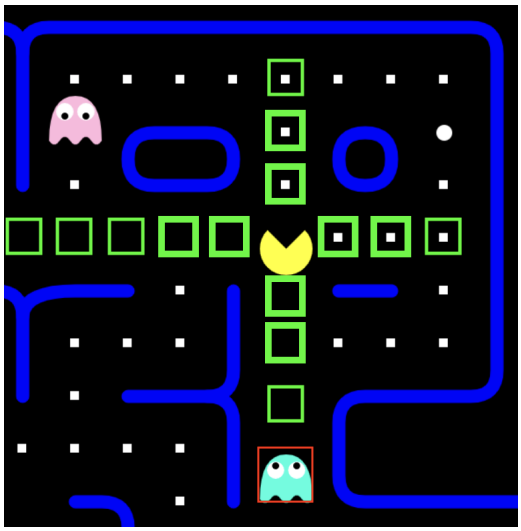

Figure 1: Example Pac-Man screenshot

In Figure 1 for example, Pac-Man can see:

- Up: 3 biscuits
- Right: 3 biscuits
- Down: 1 dangerous ghost
- Left: Nothing

I also incorporated an encoding discounter factor = 0.9 (separate from the learning algorithm discount factor γ) that discounts the values of farther states (ie a biscuit one square away would be encoded as 1, two squares away would be 0.9, 0.81, etc). In this way, items that are closer have more weight.

3.2 Reward

The reward function I implemented was:

- Being eaten by a dangerous ghost (terminal): -100
- Eating a vulnerable ghost: +50 * (total number of ghosts eaten after a pill)
- Eating a pill: +50
- Eating a biscuit: +10

In this arena, if played optimally, the agent can achieve a maximum reward of 3980 = 1980 (from the biscuits and pills) + 4*500 (from eating all 4 ghosts after eating each pill).

3.3 Training

To benchmark the efficacy of the agent over time, after every 10th episode, I ran the agent in a purely-greedy mode ($\epsilon = 0$) on 10 trials and averaged the total reward.

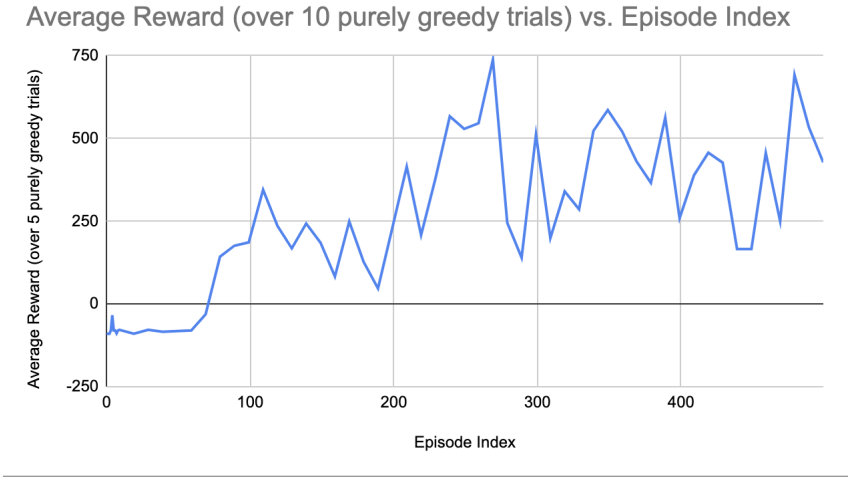

Figure 2: Episode vs Avg Reward for Line of Sight Encoding

In the first 100 episodes, the agent made steady progress, then had increasingly erratic results. Even so, the agent was clearly learning and performing better than it had done with zero weights. Episode 269 had the highest average reward of 738. The weights after that episode were (rounded to 2 decimals):

w = [-10.72, 4, 8.11, 13.57]

These learned weights make sense because the first weight corresponds to negative reward (dangerous ghosts), while the other weights correspond to positive reward (edible ghosts, pills, and biscuits).

3.4 Issues

There were several issues with this implementation of Pac-Man primarily due to the insufficient line-of-sight state encoding, since Pac-Man:

- Can only see along one direction and cannot see ghosts around the corner
- Can get stuck in empty corners that do not have pills, biscuits or ghosts since all actions look equivalent
- Is good at eating nearby pills and biscuits, but does not have the strategic understanding to complete a level

4 Extra Experiment: Tree Search Encoding

In an attempt to solve these issues, I implemented a tree search state-action encoding. Instead of only looking in one direction along a line of sight, the tree search approach, at each block of the arena, constantly looks in every direction for valid (ie no walls) and non-visited blocks. After exploring up to the maximum depth, the algorithm recursively averages the state encodings between branches and sums the discounted states from the leaves back to the origin, conceptually similar to a sum of discounted returns.

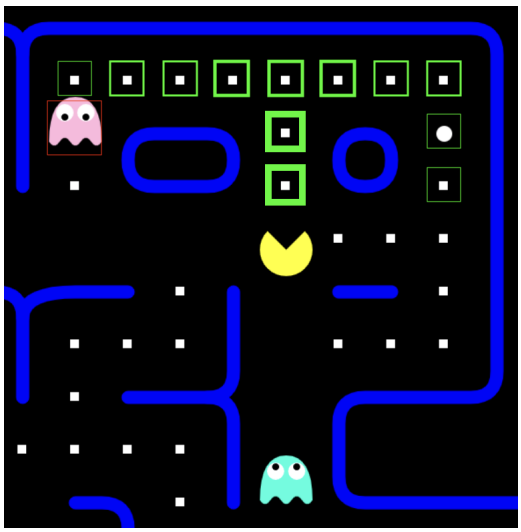

Figure 3: Example depth 8 tree search for the Up action

In the above example, the agent tree searches the Up action with a depth of 8. At the top juncture, the tree search continues looking left and right and then around the respective corners up to 8 spaces.

Parameters:

- Max tree search depth = 10
- Tree search discounter factor (separate from the n-step Sarsa discount) = 0.9

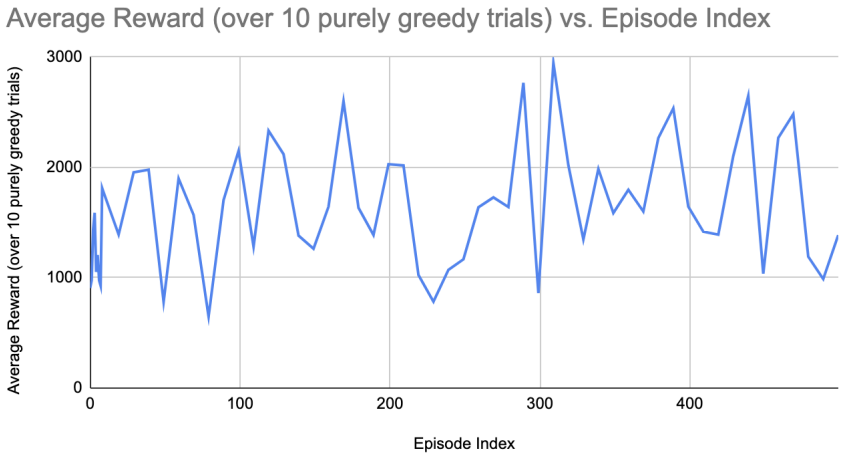

Figure 4: Episode vs Avg Reward for Tree Search Encoding

The plot of the tree search training over 500 episodes is also erratic, but the agent clearly performed better since it can now see around corners to detect nearby biscuits or ghosts. The agent can now more reliably avoid ghosts and complete the level. Episode 309 had the highest average reward of 2947. The weights after that episode were (rounded to 2 decimals):

w = [-26.01, 63.6, 44.92, 18.57]

5 Lingering Issues and Future Work

Despite the improvements from the tree search approach, there are still some lingering issues. One primary issue is that if the available action encodings are fairly symmetrical, the agent does not commit to a direction and oscillates in place, requiring a ghost to come and scare it into one direction. Another is that the linear function approximation doesn't give the agent higher-level strategic understanding (for example, that eating a pill to make the ghosts vulnerable or that there is a multiplier for eating multiple ghosts).

In the future, I could expand on this work by experimenting with:

- Adding a small passive negative reward to prevent stalling
- Adding a state encoding that represents distance to the target ghosts, pills, and biscuits to encourage the agent to move closer towards biscuits

- Using different state encodings, for example allowing the agent to select specific paths in the tree instead of averaging the branches
- Using a non-linear approximation methods like Deep Q Learning so that the agent can learn higher-level strategic understanding
- Continuing training the agent on the partially completed level after it dies to see if it can learn any strategies
- Training and test the agent on non-stationary harder levels (ie faster or coordinated ghosts)
- Changing parameters (ie n, tree search depth, state encoding discount factors, decreasing learning rate α) to see if they improve training

6 Conclusion

For this homework, I implemented a reinforcement learning agent on the Pac-Man video game using linear function approximation and episodic n-step semi-gradient Sarsa for policy control. I found moderate success with a line-of-sight state encoding, but much better success with a depth-10 tree search encoding. I suspect I could improve the agent through even better state encodings or non-linear function approximation via Deep Q Learning.

References

[1] Dale Harvey. daleharvey/pacman: Html5 pacman. `https://github.com/daleharvey/pacman`.

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.