Pacman - Coursework 2

Olivier Roncalez

March 2018

1 Introduction

The aim of this coursework was to use use a reinforcement learning algorithm in order to allow Pacman to learn how to best win the game in a small grid, after having been trained 2000 times.

The Pacman AI projects were developed at UC Berkley[1] by John DeNero (denero@cs.berkeley.edu) and Dan Klein (klein@cs.berkeley.edu). The QLearnAgent template used for this coursework was found in the mlLearningAgents.py file, and was written by Simon Parsons. This code provided the basic skeleton to get started, and was extended by Olivier Roncalez to involve a full working Q-learning algorithm.

2 Q-Learning

As briefly mentioned, the main objective of this coursework was to develop a working Q-learning algorithm which could effectively win the game on the small grid, at least 8 out of 10 times.

In order to develop an effective Q-Learner, several considerations had to be made, including which features to include as part of the state, as well as which exploration parameters to use and set (e.g., function exploration, or epsilon-greedy). Other parameters required adjustment, including the learning rate α, and the the discount factor γ, and the reward function for each state, in order to effectively update the Q-values according to the formula below:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)\right) \tag{1}$$

Before the features comprising the state for the final Q-learner can be described, and because their selection was the result of trial and error (as was the selection of aforementioned parameters), I will first describe the code employed.

[1] http://ai.berkeley.edu.

2.1 Code

The original code left to us by Dr. Simon Parsons consisted of default α, γ, and ϵ values of 0.2, 0.05, and 0.8, respectively, which were all instantiated at the start of the game. And end state was also specified which was used to keep track or the number of learning episodes, and switched off the α, γ, and ϵ parameters. Finally, in between the initialization and final state functions, there was a getAction function, which specified the location of code which was to be run at each clock step. This is where the main Q-learning algorithm was to be written.

2.1.1 Supportive Functions

To begin, I began by defining several additional functions which would help me calculate some of the state parameters I was to include, as well as my reward function and other supportive functions. Two of the functions included code which extracted the coordinates of the walls of the grid, and transformed viable paths (i.e., coordinates which were not walls) into a network graph stored in a dictionary. A Depth-First search algorithm (DFS) was supplemented by borrowing and modifying original code from Edd Mann [1]. This allowed me to calculate grid distances between PacMan, and other objects such as food and ghosts. Two other functions, MoveToInt, and IntToMove, converted PacMan's game state directions into integers, and back to directions again. This was done for indexing purposes, and is otherwise not used to impact anything in the game state itself. A reward function was also defined which used the built-in game-state rewards by subtracting the current game score from the game score in the previous step. A function to define PacMan's legal actions in the form of integers was also defined in the function AvailableActions.

The most important function was the function called key. This function took in some of the many parameters available from the game state, such as PacMan and Ghost positions, the location of food, as well as the network graph aforementioned. This was used to calculate the features which would then be used as the 'state' parameter for Q-learning.

2.1.2 getAction & Q-learning Implementation

In the function getAction, the main algorithm for Q-learning was written. This was the function which was called at each clock step, and thus called other functions and updated the Q-lookup table according to equation (1). To begin, the network map of the grid was built (only once in order to save computational costs). Next, and if loop determined if the first move had been played yet, as Q table updates only update the previous state, as opposed to the current one. If the first move had not been made yet, the reward for that state was instantiated, and the dictionary of state-action pairs within distance of PacMan was created and initialized with value 0. A parameter determining whether the first move

had been played was switched to True, in order to run through the Q-update after the first move (and for all following moves). The rest of the code in this function updated r, s, and a, which refered to the past reward, state, and action. While the reward and state were easy to update, the action was selected based on $\arg\max_{a'} Q(s', a')$, with a' and s' referring to the current state and action.

To choose the action a, a function exploration was used, providing a large reward of 300 if the state had not been visited at least once, or the Q value associated with that state s, if the state and action had previously been seen. This was controlled by an if loop parameter which only activated function exploration in the case that the agent was learning, and switched off when learning was completed (reverting back to using the Q values for a state-action pair, even if it had not been seen before in which case the value was 0). In the case that multiple maximum values (such as when function exploration has provided identical large values to two possible actions which had not been previously seen), a random selection between actions was made.

In the event that the first move had already been played, then the s', and a' values (current state and action) would first be calculated, and the Q-values updated, before updating s, r, and a, and selecting the $\arg\max_{a'} Q(s', a')$ action according to the previous paragraph. A counter of state-action pairs was also incremented prior to the Q-update, registering the fact that the learned had seen the state and action of the previous move. This counter was used to control the function exploration and reduced the learning rate α, based on how many times that state and action had been seen according to $\alpha = \frac{1}{1+n}$ with n being determined by the counter. The Q table was updated according to equation 1.

In the event that the learner died, the Q-value was updated in the 'final' function according to the last known state and action pair it had seen before death. Parameters controlling game initialization were reset in order to prepare the learner for the next game.

2.2 State Feature Selection

While the Q-learning algorithm follows a straight-forward update procedure, the efficiency of learning depended greatly on the state representations used. Using too many states, or states with high variability meant that the learner would be unable to effectively learn the full range of states and action values in 2,000 runs. Using too few state features meant that the information may be inadequate to win the game.

2.2.1 Exploration

A long trial and error process guided my feature selection. To begin, the state was first comprised of the manhattan distance between PacMan and the ghost,

as well as Pacman's positions. This however resulted in a poor state representation, with PacMan ineffectively determining how far the ghost actually was as the distance did not take into account walls, nor did it account for relative positioning. Adding the ghost coordinates to this state representation resulted in a Q-table that was too large to properly learn in the time frame of 2,000 runs.

Given these results, I explored a more abstract state representation which aimed to collapse important state information into smaller numbers which could be more effectively learned. I used the DFS algorithm to calculate the relative move that PacMan should make if he were to reach the food with the smallest distance to his position, following the shortest path to that food. Similarly, I estimated the position that the ghost would make if it were to take the shortest path towards PacMan. Both of these were coded as integers with values between 0-3 (one for each position available). Given that PacMan still did not learn efficiently, often getting stuck in the the center of the maze, I added a feature that would allow him to determine if there were 3 walls around him (i.e., if he was in the centre of the maze where he would often get stuck). While he performed slightly better, he would only converge 4 times out of 5, obtaining a 100% success rate on convergence of Q values, and only 30-70% success rate when values did not converge. Failure to converge may have been the result of a certain degree of luck required to explore certain state-action pairs which was otherwise determined by random chance (according to the exploration function picking between two equally desirable states randomly).

As a countermeasure, I included the amount of food left with the rationale that PacMan may learn to only eat the food in the center of the maze when it was the last food left. However, the abstraction of state space still provided some convergence failures (and subsequent losses in the testing example) on certain training runs. I deemed it important for PacMan to also learn the exact position of the ghost, so that it may supplement that to the ghost's direction. This in turn allowed PacMan to effectively both learn and win 100% of the time.

2.2.2 Final Features

As a result, the final feature was a vector of length 6 with the following information:

- Ghost Coordinates (x, y)
- Ghost Direction (0-3)
- Food Direction for PacMan (0-3)
- Food Remaining (1-2)
- In a Dead-End (Boolean)

Each x and y coordinate of the ghost was a separate number in the vector of state information.

3 Results

3.1 Parameters

The current parameters for the final version of PacMan's Q-Learning were as follows:

- $\alpha = \frac{1}{1+n}$ with n representing the number of times a state-action has been seen.
- $\gamma = 1$
- ϵ = Not used (as function exploration was favored over epsilon-greedy)
- Large Reward = 510 (Reward given to unexplored state-action pairs)
- Ne = 1 (Number of times required to visit a state-action pair before representing it with its true Q-values)

Determining an effective policy with these parameters seemed to occur after about 150-200 training runs (as assessed by the time it takes for PacMan to begin consistently winning games during the learning phase), with PacMan sometimes beginning to win consistently in as early as 50 learning games.

The large majority of parameters were examined, and selected through trial and error. In order to obtain the best parameters, different values of α, γ, the large reward, and the number of times to visit each state, were varied. The solution offering the most consistent results was retained. Results were assessed by running 2 full simulations comprised of learning and 100 test runs for each simulation, and examining the average number of games won for both runs. Interestingly enough, only a few certain specific parameter values significantly changed the results. In general, so long as $\gamma > 3$, and $\alpha < 1$, the agent won 100% of the time with the state features mentioned above. As only one parameter was varied at a time, not all combinations of parameter variation was analyzed. Results are summarized in the table below.

Table 1: Summary of Parameter Modification

Parameters				
α	γ	LR	Ne	GW (at least 100%)
$\frac{1}{1+Nse}$	1	510	1	1
1	1	510	1	.81
$\frac{1}{1+Nse}$	.3	510	1	.83
$\frac{1}{1+Nse}$	.2	510	1	.53
$\frac{1}{1+Nse}$	.1	510	1	.50

Non-exhaustive summary of some results obtained from modifying key parameters. LR = Large reward; GW = Games won; Ne = Number of visits required for Q(s,a) before true utility is used. The first line in the table contain the parameter values used in the final solution.

One parameter which merits some specific consideration was the value of the large reward. This value was initially chosen to be specifically greater than the reward for winning, so that all states would be explored irrespective of any potential winning solution (or paths leading up to the winning solution). That said, varying this parameter did not change the results so long that the values were greater than the cost of a step (-1). Setting values lower than 10 meant that the agent would often favor greedy exploration of states leading up to the consumption of food (+10) or winning the game (+500), even when unexplored state-action pairs were available. Given the highly effective state features selected, the agent would still win the game by finding an effective policy (even if it was not the "optimal" one). In regards to the number of times required to visit each state, the specific values did not matter so long as the agent was able to converge to a policy before learning was over. With the state features selected, the agent was capable of winning with values as large as 50, although they only converged on a policy right before the end of the 2000 learning trials. Because these two values did not affect the results, they were kept at 510 and 1, respectively, in both the analysis table and final solution.

3.2 State Features

While the parameters did not have as much effect on the agent as was expected, the features selected greatly impacted the results. As previously mentioned, these were selected through trial and error. Upon finding a solid set of features which would allow the agent to rapidly win the game, experimentation was conducted in order to determine the effects of removing or adding certain state features.

Certain crucial features (i.e., state information which if removed would result in the agent losing the game more than 20% of the time) included the ghost coordinate positions, and the direction suggested for PacMan to move closer towards the food. Removing the number of food remaining would cause the agent to sometimes lose a few games, but was more importantly required for the agent to find a quicker solution than it would otherwise if this information was removed. By that is meant that the agent would often go around the maze several times avoiding the food in the center before the ghost moved counter-clockwise.

Non-essential information included PacMan's position (which was not included in the final solution), the notification of a dead-end, or the ghost direction. However, the last state feature did not matter ONLY if PacMan's coordinates were included. In the event that they were not, PacMan would lose a significant amount of games (less than 70%). Given that both the dead-end notification and ghost direction allowed PacMan to win games faster than he would if the information was removed, thee were retained in the final solution.

Actual statistics are not included as the aforementioned information depended on if PacMan lost at least one game, or if PacMan won 100% of his games. If PacMan lost at least one game with the removal of state features, it was deemed crucial. If PacMan still won 100% of his games with that removal, it was deemed non-essential. Features were removed one at a time, holding all other features in the final solution constant in order to more specifically determine their effect. The only exception was adding PacMan's coordinates when the ghost direction state information was removed.

4 Conclusion & Future Directions

While PacMan won 100% of the time, examination of his playing style quickly allows us to determine that it is far from what an optimal agent would, or should, do. In other words, rather than go straight for the food while avoiding the ghosts, the agent will sometimes repeat a set of redundant moves such as going North, then South, then North again, despite the presence of food close to it. In fact, the agent seems to be more concerned with actively avoiding the ghost when it is near, eating the food with some degree of chance, rather than actively seeking it.

Despite the small grid and limited opponents, learning the Q-table proved challenging. Finding the specific set of state features which would allow PacMan to win the game 100% of the time proved more difficult than it appeared at first glance, even when considering the agent does not win through an aggressive optimal policy. Some attempts at using Q-learning with function approximation proved to be far more effective, with PacMan only learning two functions: one using the distance to food, and another using the distance to the ghost.

However, I was unable to polish up and refine this approximation before submission. That said, it is clear that unless a highly effective state abstraction is possible (one which minimizes the number of state features that PacMan needs to learn), function approximation is the only feasible solution for larger grids, such as MediumClassic. As an added bonus, an effective function approximation learner would be best suited to generalize to other grids without specific refinement of the functions. This is in stark contrast to my Q-learner which has been tailored to win for this exact grid, and which would most likely fail in different formats.

In conclusion, while PacMan managed to win the game consistently, the solution is not one of an aggressive policy designed to maximize the score, nor is it likely generalizable to other mazes (though testing remains to be seen). Q-learning function approximation would be the appropriate next step from here.

References

[1] Edd Mann. Depth-first search and breadth-first search in python.