Generalizing PAC-MAN

Can selecting handcrafted features compare to state-of-the-art convolutional feature extraction?

Joost Doornkamp (s2550156) Daniel Marcos (s3734277)
Avinash Pathapati (s3754715) Subilal Vattimunda Purayil (s3587630)

University of Groningen

Abstract

Games have been widely used as a platform to develop and test reinforcement learning. This paper compares two approaches of reinforcement learning applied to a game and their results. An agent for the PAC-MAN game has been developed with two reinforcement learning algorithms; one using a convolutional neural network to represent the game state from which features are learned implicitly and the other using handcrafted features. The paper discusses the implementations of these agents in detail, illustration of output and discussion on results and how this two methods compare in being able to play PAC-MAN in real time.

1 Introduction

Games are an entertainment or educational activity for human which follows certain rules and involves a number of strategies and polices to reach a final goal or maximize a role. The games often involves human factors like decision making, solving conflicts, strategy, skill, competition and fun. Many games involve high challenges and decision making skills which leads artificial intelligence researchers to be interested in this area.

In this paper we try to autonomously play the popular PAC-MAN game in an efficient method. This involves strategy to traverse the maze and eat away all the food without colliding with the enemy ghosts.

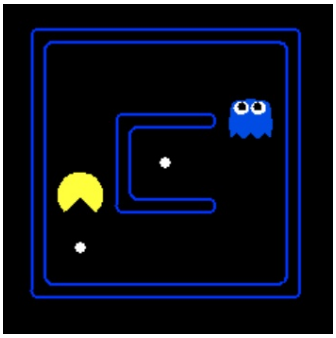
The reinforcement learning algorithms are the goal oriented ones which help to solve complex tasks. The agent is a reinforcement learning software entity that takes decisions on each state and drives to final result, taking the place of the human who is playing the game.

The background where the agent operates and take actions are called the environment. The maze of PAC-MAN with ghost, food and power pills form the environment for this project. The actions are all possible movements that can be taken by an agent in a given environment.

The actions in PAC-MAN are moving left, right, up and down which result in various gaming strategies like eat pills, run away from ghost or chase the ghost. Some implementations of the game also support stopping as an action.

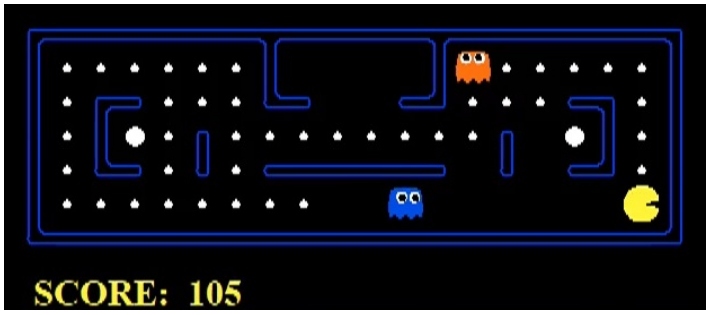
The reward is the score an agent receives from the environment and decides the measure for success or failure over the last actions taken. A state in the game is everything that describes the current scenario the environment is in and the agent selects the action which has the maximum Q value. Q value is a measure of how good the particular action is in reaching the goal from the current state and is computed using Markov decision process. So the decision made in choosing an action is dependent only on the current state the agent is in and is independent of all the other previous states.

This project prioritizes two distinct algorithms in reinforcement learning which are compared on basis of their efficiency to play PAC-MAN. DQN agent use a Convolutional Neural Network (CNN) and feature agent uses a list of handcrafted features to represent the game state respectively. Both agents then use Multi-Layer Perceptron (MLP) to learn an optimal q-value for each action in a given representation and play the game in real-time. The two agents are compared in terms of performance in PAC-MAN, represented by their game score.



(a) The `smallGrid` layout.

(b) The `mediumGrid` layout.



(c) The `smallClassic` layout.

Figure 1: The three different layouts in the Berkeley implementation used.

2 Method

For the game implementation we used the PAC-MAN implementation provided to the public domain by UC Berkeley CS188 Intro to AI Course Materials[1]. This implementation is greatly customizable and developed for the implementation of custom agents. The course material was created for reinforcement learning purposes but any kind of agent could theoretically be developed. Apart from that, we use the some of the code provided in [5] in order to link the neural network with the game environment.

The Berkeley implementation provides a scoring system which can be used for evaluation purposes and different maze layouts that we can develop in. Because of time constraints on development and training, we only used the smaller layouts as shown in Figure 1.

The 'grid' variations of maps are small, only feature one ghost and have a small amount of pills and no capsules. This allows developers to see more directly whether an agent is chasing pills. The `smallClassic` layout has two ghosts and follows the rules of normal PAC-MAN, having a pill or capsule in every empty space, except for the ghosts' starting zone.

By default the game awards 10 points for eating a pill, 200 for eating a ghost and 500 for winning the game. It deducts 500 points for losing the game, and 1 point for each time step.

The implementation of our agents largely follows [3], building on an implementation of the Berkeley code by [5]. We will go further into the image processing methods used for the convolutional agent in section 2.2. For the feature agent, the premise is to replace the image processing part of the pipeline with a selection of handcrafted features and evaluate whether such an agent can perform in similar quality.

Both agents use the Q-learning technique described in their paper, as well as the epsilon greedy strategy in combination with a replay memory. For the latter, parameters have been altered which will be discussed for each agent. In the epsilon greedy strategy, an agent decides on doing something random or to consult its network based on the epsilon (ε) parameter. This parameter starts high, corresponding to a lot of random actions, but decreases over time as the agent learns. Near the end the agent should do (almost) no random actions.

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

2.1 Feature Agent

2.1.1 Feature Selection

[1, 2, 4] describe a wide variety of suggested features. The agent's absolute position, the number of food/capsules eaten/still to be eaten, relative X/Y distances to ghosts, pills and capsules and the last action taken were all considered, implemented and discarded due to a diminutive or negative observed effect on performance.

The features that we have ultimately used are as follows:

- 4 channels indicating the presence of a ghost directly north, east, south or west of the agent;
- 4 channels indicating the presence of a wall directly north, east, south or west of the agent;
- 3x2 channels indicating the path distance and direction of the nearest pill, ghost and capsule;
- 2 channels indicating for each ghost whether or not that ghost can currently be eaten by agent (i.e. PAC-MAN has eaten a power capsule and can now consume said ghost).

This brings us to a total of 16 features.

The first four features logically proved crucial in what was observed to be the agent's most crucial behaviour: staying alive. Since the reward for winning is so large compared to eating any one pill, it proved a fruitful strategy to just avoid the ghosts and incidentally run into the pills. This evasion was so efficient that the only thing that could effectively end the game was either winning or the epsilon value causing the agent to do a random action, almost always leading to its instant demise.

When more features were added and the agent could theoretically see farther than its immediate neighbourhood, and actively seeking out pills became possible, these features still proved useful as they seemingly provide an instant threat response. The agent could be observed switching from more long-term behaviours to the original evasion when an enemy was directly next to it.

The four features regarding nearby walls were not evaluated in-depth. Logically it would allow the agent to realize which actions are impossible, but when investigating action preferences it was seen that the agent still prefers an action which would run it into a wall. However, when this happens, the decision making code catches the mistakes and takes the next best action.

'path distance' and direction features capture the total number of tiles that need to be covered to reach a point via the fastest possible route, considering possible actions, as used in [4]. A breath-first search mazecrawler was implemented to explore the maze from PAC-MAN's position, listing all pills, ghosts and capsules. As such, the distance and direction yielded by it takes into account pathing around the walls.

The breath-first search mazecrawler maps the entire maze for each game state. For large mazes, this may have a serious affect on agent performance, especially when playing in real time. However, since during development it became apparent that the convolutional agent would not have time to train on larger mazes and only small mazes would be used, it was considered satisfactory to leave the mazecrawler as is. If the project is continued, it might prove fruitful to set a maximum distance. Note that while the crawler yields all pills, ghosts and capsules, only the nearest ones of each type are ultimately used in features.

The last two features are simply boolean indicators whether each ghost is currently a threat. Note that while after eating a capsule all ghosts become edible, it is not true that all ghosts are simultaneously edible. If the agent eats a ghost, that ghost is reset and is no longer edible, while any other ghosts that have not been eaten remain edible until their timer runs out or the agent eats them too. Thus there is a need for an indicator for each ghost.

2.1.2 Network

The 16 features are used in a 32x16x4 fully connected neural network, using a sigmoid activation function.

The architecture has a gradually decreasing layer size and an extra hidden layer than may be technically necessary. This allowed the network to more effectively group together the connected features. As described above, the features can be divided into four distinct categories, and the third group can be subdivided into three direction/distance pairs.

2.1.3 Parameters

For each layout an agent was trained for 2000 games and a snapshot of the network was saved every 1000 in-game steps. The replay memory had a size of 100 000 most recent game states and the first 1000 actions were generated completely randomly to fill the memory with enough states.

For the `smallGrid` and `mediumGrid` layouts, the epsilon parameter was gradually decreased from 1 to 0.01 over the course of 10 000 in-game steps (excluding the 1000 steps it takes for training to start).

The reward function was altered to reduce the effect of dying, from -500 to -50. Originally the reward function corresponded directly to the in-game scoring, but with the disproportional penalty for dying that the game adheres to, the agent would be unable to learn anything but survival by getting stuck in simple loops that would never end the game.

The agents were tested from their saved snapshot each time so that learning that occurs during testing did not affect performance on the next test.

2.2 DQN Agent

Rather than using features, in this agent each state of the game is represented by an image from which the features are learned automatically. The image size should not be very large as training the DQN takes more time and at the same time it should not be small as the DQN should be able to identify the features from the pixels and approximate the Q value function. So, for each state, an image is constructed in which each pixel represents the objects of the PAC-MAN Grid. We know this methods works as demonstrated in [3], in which they use a similar implementation. To differentiate the objects in the image, pixels representing different objects are marked with different colours.

After conversion to pixels, the `smallGrid` image size is 7x7, `mediumGrid` is 8x7 and `small-Classic` is 20x7.

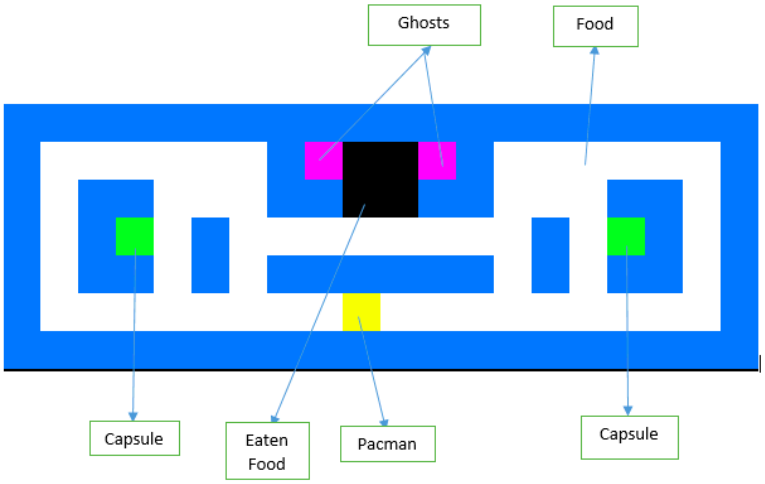


Figure 2: A state of a PAC-MAN game converted to pixel representation.

We tried experimenting with different DQN architectures and the optimal network structure we found is presented in Figure 1. The DQN network consists of 3 convolutional layers. The first convolutional layer has 32 filters of size 3*3, the second convolutional layer has 64 filters of size 3*3 and the third convolutional layer has 256 filters of size 3*3. The output of the last convolutional layer is passed to a fully connected layer of size 256. All of these layers use RELU as the activation function. The final layer is a linear layer which computes the estimated Q value for all the possible actions in a given state.

We experimented with different values for the parameters used in building the model. The optimal values we found for those parameters are shown below, selected on testing performance after training for at least 1.000 games.

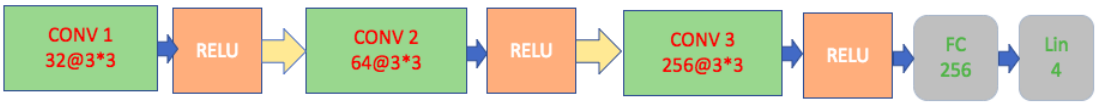


Figure 3: The network architecture for the DQN agent.

	Feature Agent	**DQN Agent**
Replay memory size	100 000 steps	100 000 steps
Batch size	32 states	32 states
Discount rate	0.8	0.9
Learning rate	0.00025	0.00025
Epsilon start	1.0	1.0
Epsilon end	0.1	0.01
Epsilon steps	10 000	10 000

Table 1: Additional parameters used by both agents.

3 Results

In this section, the different results obtained for training both the Feature Agent as well the Deep Q Learning agent are presented by means of the average score and the win rate achieved on each evaluation. So, for every 1000 training steps the model is saved and evaluated on 100 subsequent games. Capturing values this way could provide a good representation of the agent's learning behaviour over time. The results of these test can be observed in Figures 4-6.

On the right of each figure is the win rate; the percentage of testing games that ended in the agents' victory. On the left is the average score achieved by the agents.

In both grid variations, the two graphs seem very similar. This is because in a game with little pills which award a relatively diminutive score the score is mostly determined by winning or not.

4 Conclusion

We've observed that both agents are able to learn the grid variations of the PAC-MAN game. In `smallGrid`, both agents achieved a win rate of almost 100%, though it is notable that the feature agent took much longer to learn. In `mediumGrid`, we see that the feature agent slightly outperform the DQN agent and it achieves almost perfect scores consistently. The most remarkable results, however, appear in the `smallClassic` layout.

This layout is easily the most complex, having many pills, two ghosts and featuring power capsules. It is here that we see the feature agent lose to the DQN by a large margin.

What vexes the feature agent is its short-ranged vision in such a complex scenario. It can effectively

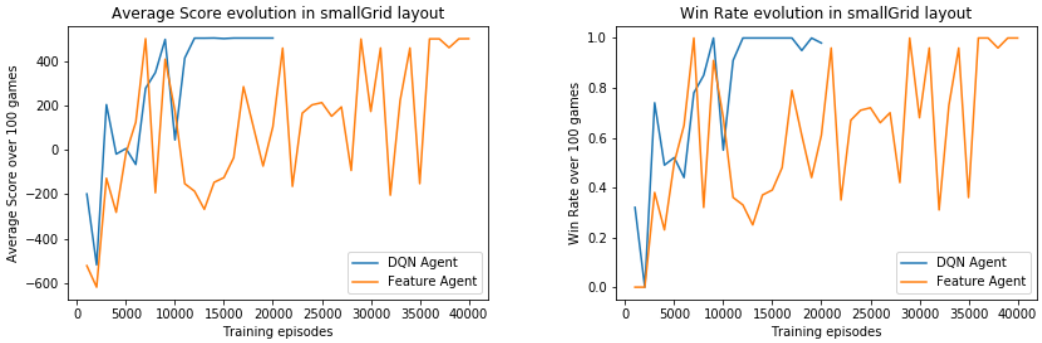


Figure 4: Results for the `smallGrid` layout.

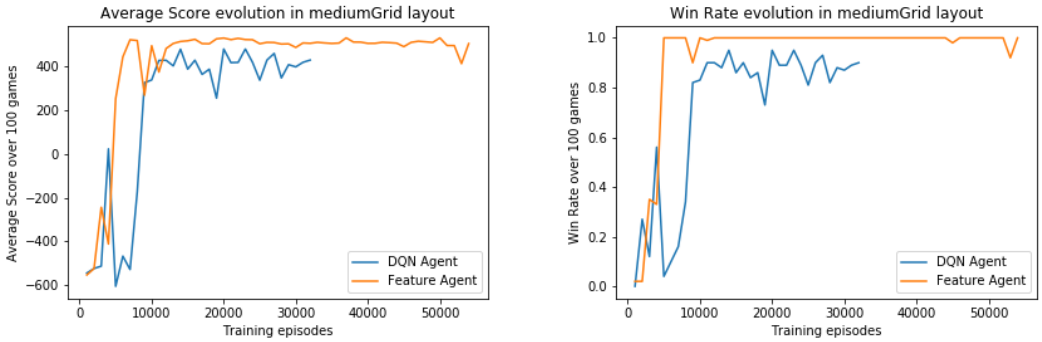


Figure 5: Results for the `mediumGrid` layout.

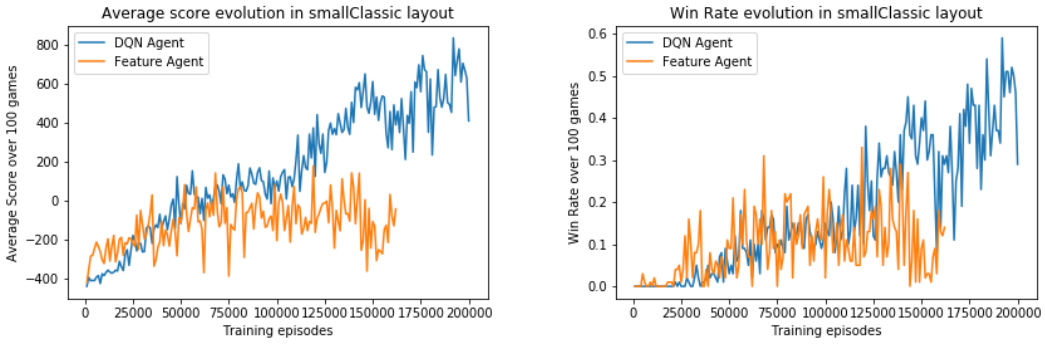


Figure 6: Results for the `smallClassic` layout

avoid ghosts and track down pills using only information about the nearest pills, ghosts and capsules, but once pills get too far away it can get stuck in evasive behaviours. The DQN agent retains a full picture and is more often able to play the game to its end.

A solution would be to give the feature agent a more complete picture, but the issue then becomes how to represent this. It would require configuring the agent to the exact amount of ghosts and possibly pills.

This leads to the most general conclusion: it should always be possible to tweak a feature-based agent to perform near-perfectly in any one scenario. That may often not be desirable, especially when you can see that a convolutional image representation can achieve similar scores but with a general solution for different layout, rather than one fixed to one layout.

We wanted to find out if you could replace a convolutional game state representation with a simple feature vector, and we presume the answer is yes if the problem is simple enough. However, we have shown that it is much simpler for the developer to just use the convolutional representation, at it provides a general solution to many problems.

References

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

[2] Peter Burrow and Simon M Lucas. Evolution versus temporal difference learning for learning to play ms. pac-man. In *Computational Intelligence and Games, 2009. CIG 2009. IEEE Symposium on*, pages 53–60. IEEE, 2009.

[3] Abeynaya Gnanasekaran, Jordi Feliu Faba, and Jing An. Reinforcement learning in pacman.

[4] Simon M Lucas. Evolving a neural network location evaluator to play ms. pac-man. In *CIG*. Citeseer, 2005.

[5] Tycho van der Ouderaa. Deep reinforcement learning in pac-man. 2016.