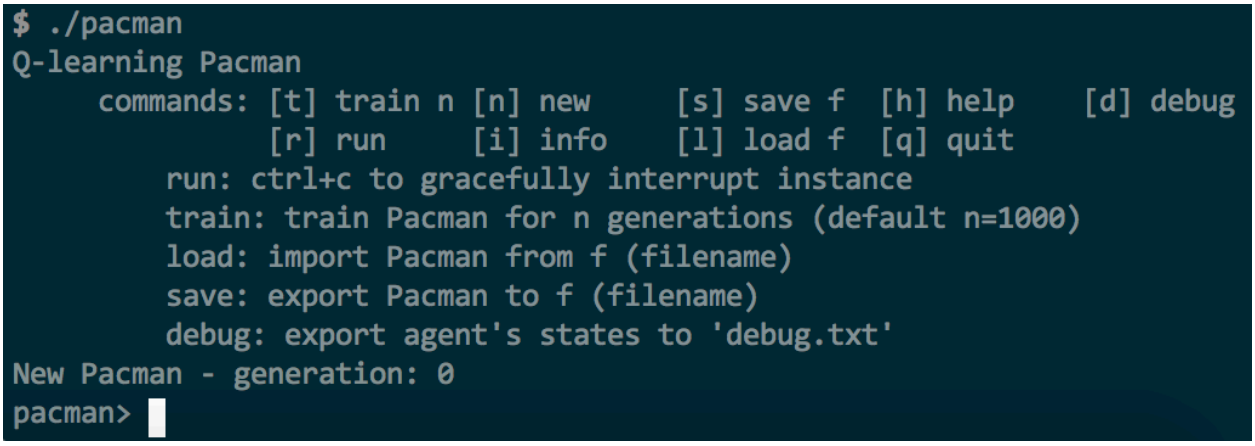
# Reinforcement Learning Techniques: Q-Learning

by Matt Breckon

The pacman application was built on Python 2.7.11 and is ran with:

```
$ ./pacman
```

The provided UI was designed to be straightforward with available commands listed.




## Project

### The Game

From the start, my main goal was to tackle a project that involved some form of reinforcement learning. Stumbling across the assignment idea “Q-learning in pacman” sparked the foundation of this project. The core of the project was to create an extremely simplified version of the classic arcade game PAC-MAN where pacman explores a grid gathering randomly generated pellets with a single ghost pursuing pacman. In this specific version there is no walls within the grid; instead, the grid is surrounded by a wall that results in pacman dying if touched and the game ending. The ghost is also designed to randomly spawn on the edge of the grid in either the column or row pacman occupies; the ghost then follows a straight line towards pacman until it reaches the opposite edge of the grid.

[source: http://modelai.gettysburg.edu/2016/pyconsole/ex5/index.html]

### Goal

The general goal of the project was to incorporate a form of reinforcement learning known as Q-learning. Q-learning is a model-free learning technique that involves an agent

selecting an action based on an action-value function. An agent within an environment is given the current state of the environment and a list of possible actions; the agent selects the most optimal action based on a policy. This policy returns the best possible action based on a list of past experiences the agent has with the given state. By training the agent over several generations, the agent is exposed to an increasing but finite number of states. With these states, actions, and resulting rewards, future generations of the agent appear "intelligent" in selection of actions when navigating their environment.

In the case of pacman, the goal was simple: make him more intelligent within the game's environment. To do this, pacman needed to be able to do two things: avoid death and collect the pellets. More specifically when it comes to avoiding death, pacman needed to elude the ghost and avoid the surrounding walls of the grid.

## Algorithm

At its core, Q-learning is extremely similar to another reinforcement learning technique known as SARSA (State-Action-Reward-State-Action). The idea of both algorithms is to take a given state-action pair and the resulting reward, and then determine a weighting associated with the future state-action pair based on the initial state-action pair. Q-learning assumes that the future action is always optimal while SARSA requires a future action to be explicitly selected for a future reward.

The Q-learning algorithm consists of a Q value associated with a given state-action pair. By default, all state-action pairs are given a fixed value. In the case of this pacman implementation, all state-action pairs default to 0. A hash table is used to store a state-action pair as a key pointing to its associated Q value. As an agent explores, the Q value of a state-action pair is updated within the hash table.

In order to update these weights, the following formula is used:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha * [R + \gamma * maxQ_t(s_{t+1},a) - Q_t(s,a)]$$

To break this formula down to a more digestible format, $Q_t(s,a)$ represents the Q value associated with the initial state-action pair. $\alpha$ represents the learning rate; this determines the rate at which new information overrides old information. $R$ represents the reward associated with the initial state-action pair. $\gamma$ represents the discount factor; this determines the weighting future rewards hold on the Q value.

In general, the algorithm works by taking a “learned value” from a given state-action pair and subtracting its old value. This in turn is multiplied by a “learning rate” and this result is added to the Q value previously associated with the state-action pair. This results in a new weighting future generations of the agent can use to make decisions with.

# Development and Design

## State Decisions

The initial project provided by the previously mentioned source provided basic code to generate and update the grid. This update moved pacman and the ghost; however, pacman’s decisions in what direction to go were chosen at random. To tackle this, my first version of the project consisted of giving pacman a restricted “window” to view the world around him. Due to the nature of the game, the pellets were always randomly generated in different locations. This presented a problem: to store the entire board as a state would require an increasingly large number of visited states to be useful. The likelihood of two boards having pellets in identical locations was extremely unlikely in a 20x20 grid with 10 pellets. Limiting a state down to a window of adjacent tiles to pacman meant a far lesser amount of states were needed to be seen for pacman to gain any benefit out of iterative training sessions.

## Implementation 1.0

Initially, the states were represented as 4 tiles adjacent to pacman. This meant a state consisted of a 9x9 grid with pacman located in the middle. With enough tweaking and enough generations, this resulted in pacman performing exceptionally well at dodging the ghost, avoiding walls, and collecting pellets; however, a few major issues were a result of this design choice.

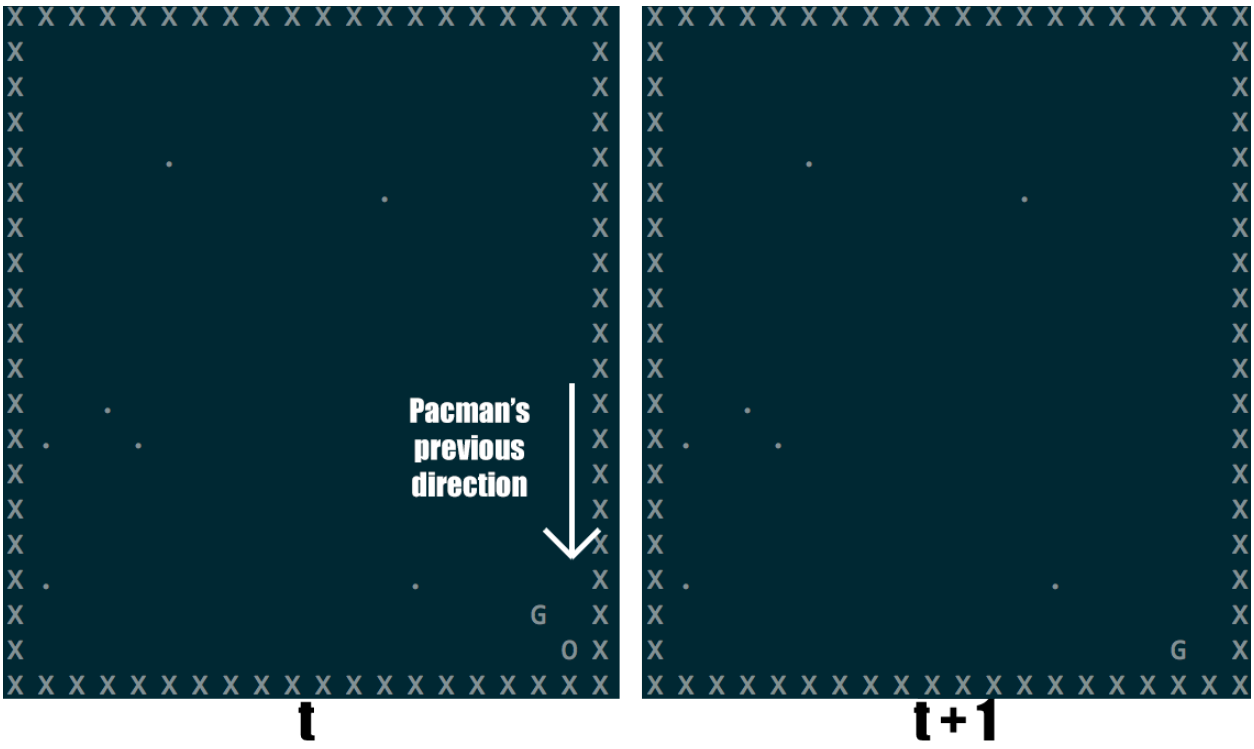
The first problem was that to have pacman appear “intelligent” based on training, **many** generations were required. For a window of 9x9 tiles for pacman to make an optimal decision in action, he had to have experienced an identical 9x9 window in previous iterations. After 10,000 generations, pacman wasn’t perfect at surviving and this resulted in over 150MB in memory usage. After 100,000 generations, pacman was amazing at surviving but the memory usage of 2.5GB was extremely impractical. But even with the 100,000 generations, pacman was still not very smart. This leads into the second problem: he never perused pellets not immediately adjacent to him. This was due to the core of the design relying on a state, action, reward, and a future state and action: immediate moves he can make and potential moves he can make from his immediate move.

## Implementation 2.0

Realizing the size of the window heavily impacted the performance, the next plan was to reduce pacman's view of adjacent tiles to a more optimal size. I decided 2 tiles adjacent to him was perfect because a 5x5 grid meant a vastly smaller amount of states were needed to be visited in training. Ideally, pacman only cares about his immediately adjacent tiles: is it a pellet, ghost, wall, or empty? However, restricting his view to immediately adjacent tiles is naïve as the algorithm is designed to look at future state-action pairs based on a selected action. For example, if a ghost was outside of pacman's immediate adjacent tiles and was 2 tiles away, he may opt to select a tile and the ghost could move into that tile from outside his initial view.

After implementing this design choice, it was clear how quickly and easily pacman could learn to survive. After a mere 1,000 generations, pacman could easily dodge the ghost and nearby walls, and he would also collect adjacent pellets. However, as expected, the problem from the previous implementation still existed: pacman was not designed to seek out pellets across the board. A small but noticeably beneficial change was then added. To avoid pacman "dancing" around, pacman was tweaked to avoid selecting an action if it resulted in him returning to his previous location. This design decision is mostly beneficial as pellets do not move so there can never be a positive reward in returning to a previous location. In doing so, this helped push pacman towards exploring in new directions rather than dance around.

Note: one rare exception I encountered with pacman not returning to previous positions meant it is entirely possible the below situation can arise.



## Implementation 3.0 and Beyond

With a pacman AI smart enough to survive its surrounding environment and gather local pellets, all while being much more optimized than previous implementations, the next goal was to come up with a method to encourage pacman to seek pellets outside of his limited view. To tackle this, a method known as feudal reinforcement learning was examined. The idea behind feudal reinforcement learning is to implement a form of learning, such as Q-learning, at multiple resolutions simultaneously. This method uses a hierarchal system of managers and sub-managers to select actions based on states on multiple levels. For a sub-manager to make a decision, its action must result in a positive or neutral reward from its manager.

[source: http://www.gatsby.ucl.ac.uk/~dayan/papers/dh93.pdf]

In the case of pacman, this could mean a system of two managers: an overall board manager and a local window sub-manager as previously implemented. The sub-manager is designed to handle pacman avoiding the ghost and nearby walls, and is also designed to make pacman collect adjacent pellets. The super-manager would be responsible for guiding pacman to specific regions on the board to explore and gather pellets not within his current view.

Although not implemented in the final design of the project, the general idea was to design a manager to guide pacman. The grid pacman explores is a 20x20 grid. If the grid's resolution is reduced, it can be represented by, for example, a 5x5 grid, where each tile of the 5x5 grid represents a 4x4 section of the original board. A manager could implement a form of Q-learning where a state is the total number of pellets within one of these 4x4 sections. A 4x4 section containing 5 pellets would weigh more in reward compared to a section containing 0 pellets. This hierarchal system would allow a sub-manager to determine immediate moves for pacman but a general manager to encourage the sub-manager to move pacman into regions with more pellets.