Teaching Pacman to play

Abhiramon R. (IMT2015002)

May 4, 2019

1 Introduction

Pac-man is an arcade game released in 1980. The game is set in a maze filled with dots called pellets. Users control a character called Pacman whose objective is to eat all the pellets while evading four computer-controlled characters called ghosts which try to eat Pacman. Pacman gains points after eating every pellet and keeps losing them with time. The game ends when all the pellets are consumed or Pacman gets eaten by a ghost. The map also has power-pellets which enable Pacman to eat the ghosts for a short duration earning a lot of points[3]. You can play the game online here: `https://www.google.com/logos/2010/pacman10-i.html`

In this read, we explore how we can teach Pacman to play the game by itself. Primarily we cover Q-learning which is a reinforcement learning algorithm. To understand Q-learning, we first need to see how the Pacman environment can be captured mathematically (as states, actions and rewards). Then we see how Pacman can use this information to decide its actions (using Markov Decision Process). Later we cover how Pacman learns about the environment over time (using Q-learning), improving its decision-making. Finally we explore a more practical way to let Pacman learn (using Q-learning with function approximation). We use the Pacman interface provided at `http://ai.berkeley.edu/reinforcement.html` to implement the algorithm.

2 Formalizing Pacman environment into states, actions and rewards

Actions is the set of all possible moves an agent can take. Agents use actions to interact with the environment [4]. For example, Pacman can take the actions, North, West, East, South, Stop which are the directions in which Pacman can move one step [1].

We formulate state of the game to capture the current situation. The environment assigns a reward based on the state the agent is in. Agents can use actions to change state. In the case of Pacman, state holds the dynamic information such as location of agents, agent speeds, pellets remaining and scared timer of ghosts [1]. Pacman can use all of this information to choose an action.

Rewards are assigned to the agent based on its state. To succeed in the game, an agent should try to maximize the reward it gets. In Pacman an example of rewards can be:

Action	Reward
Eat food	+10
Eat scared ghost	+200
Die to ghost	-500
Finish game	+500

3 Markov Decision process (MDP)

Often agents need to take into account the consequences of their actions in the future. Markov Decision Process (MDP) is a method of decision making that agents can use to serve this purpose. Pacman needs to move towards pellets and away from ghosts. It needs to plan its actions to ultimately finish the game. Hence, future planning is essential for Pacman.

Markov property states that given the current state, the future states are independent of the past. Hence, each state that the game goes through is dependent only on the previous one. We can thus define transition

probabilities which capture how likely the transition is from one state to the next [5].

$$P[S_{t+1}/S_1, S_2, ...S_t] = P[S_{t+1}/S_t]$$

To account for the future while choosing an action, we need to introduce the notions of Discounting, Value and Q-value.
To incentivize an agent to collect its reward earlier as compared to later, rewards are made smaller with each passing time step. This is known as discounting [5].

$$R_t = \gamma^t R_0, \gamma \in [0, 1]$$

Value of a state 's' is defined as the expected reward the agent would get by acting optimally from that state onwards. Whereas, Q-value is defined as the expected reward of taking an action 'a' from a state 's' and acting optimally thereafter. For timestep k, discount value γ, transition function T and reward function R, these values are computed using Bellman equations as given below [1]:

$$V_{k+1}(s) = max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k(s')]$$

$$Q_{k+1}(s, a) = \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma max_{a'} Q_k(s', a')]$$

Given the environment captured as explained above, all the agent now needs to do is pick the action which has the maximum Q-value in the current state.

We have assumed above that the transition probabilities and rewards are known beforehand. This is not the case in most real-life situations, including Pacman. Thus, we need to learn these values through trial runs in the environment, using an algorithm called Q-learning.

4 Q-Learning

We initialize all Q-values to 0 in the beginning. To allow Pacman to learn the environment, we allow it to choose a random action with a certain probability and pick the optimal action otherwise. Once an action is picked,

the following procedure is followed to update the Q-values.

If we observe that action 'a' from state 's' leads to state 's'' and results in a reward of 'r'[1],

$$observedReward = R(s, a, s') + \gamma max_{a'} Q(s', a')$$

We incorporate the observation into our previous Q-value as a running average [1]:

$$Q(s, a) = (1 - \alpha)Q(s, a) + (\alpha)observedReward$$

This way the agent associates higher Q-values with actions from states that lead to better rewards. Pacman learns to avoid ghosts which lead to negative rewards and collect pellets which give positive rewards.

5 Q-learning with function approximation in Pacman

As we have seen earlier, state holds information of agent locations, pellets remaining, etc. Since the number of combinations possible for these values and hence the number of states grows exponentially with the map size, it becomes impractical to use this algorithm for larger maps. However, we can implement a version of Q-learning which uses function approximation for the Q-values. In this method, we learn a model of the environment in terms of weights of certain functions. These functions take the state and action as inputs and are used to compute Q-values. For example, we can use functions such as number of ghosts one step away, presence of pellet, distance to closest food etc. And define Q-value as [1]:

$$Q(s, a) = w_1 f_1(s, a) + w_2 f_2(s, a) + ... + w_n f_n(s, a)$$

In this algorithm, if we observe a sample, (s, a, r, s'), we update the Q-values and weights as follows [1]:

$$difference = [r + \gamma max_{a'} Q(s', a')] - Q(s, a)$$
$$Q(s, a) = Q(s, a) + \alpha[difference]$$
$$w_i = w_i + \alpha[difference] f_i(s, a)$$

After learning, the weights we end up with give us information on how important each of the features are and give us an intuitive understanding of how Pacman is trying to behave. For example, a very negative weight for ghosts one step away indicates that Pacman is trying hard not to get eaten by a ghost.

6 Conclusion

Implementation of Q-learning with function approximation worked very well for Pacman. It was able to get a better average score than me over 15 games. Since we engineered functions for Pacman to rely on, we may be able to further improve its performance by adding/modifying the functions in the future.

References

[1] Intro to AI UC Berkeley,
`http://ai.berkeley.edu/home.html`

[2] Pacman google doodle,
`https://www.google.com/logos/2010/pacman10-i.html`

[3] Pac-Man,
`https://en.wikipedia.org/wiki/Pac-Man`

[4] A beginner's guide to deep reinforcement learning ,
`https://skymind.ai/wiki/deep-reinforcement-learning`

[5] Reinforcement Learning Demystified: Markov Decision Processes,
`https://towardsdatascience.com/reinforcement-learning-demystified-markov-decis`