6CCS3AIN (Artificial Intelligence)

Practical 06: Pacman meets Bellman

(Version 1.2)

1 Overview

This practical continues your journey towards the coursework. Last week you worked on building a map and having Pacman apply simple decision theory to decide what action to take. The problem with that approach is that it was *myopic* (shortsighted). It tended to pick states that provide immediate gratification without thinking about the long term best thing to do. This week we will make Pacman more thoughtful, and we will do this by equipping Pacman with code that performs value iteration. In other words, we will build a simple MDP-solver and use the policy it calculates to tell Pacman what to do.

2 A new layout

The first thing to do this week is to start working with a new layout. This is designed to make it easy for you to write an MDP-solver which can win games. The layout is the one in Figure 1. To use it, do the following:

1. Download `smallMDPGrid.lay` from KEATS.
2. Place `smallMDPGrid.lay` in the `layouts` folder in your `pacman` folder
3. Run your `MyGreedyAgent` from last week using this layout:

   ```
   python pacman.py --pacman MyGreedyAgent --layout smallMDPGrid
   ```

What you will find is that Pacman does not act in a very purposeful way. It is making decisions based just on the values you assign to the grid squares around it, and these are mainly zero because there is no food in most of the spaces.

The way to improve this behaviour is to compute the utility of the spaces rather than the values that are currently assigned to them. (The values that your `MyGreedyAgent` is using are basically rewards not utilities). We know how to move from rewards to utilities: we apply the Bellman equation in the form of value iteration.

Your job for this practical is to write a `SimpleMDPAgent` that uses value iteration to compute the utility of every state[1], and then uses these utilities to decide what to do. This agent should be able to win games in `smallMDPGrid` very efficiently.

[1] More precisely, the utility of every state under the optimal policy.

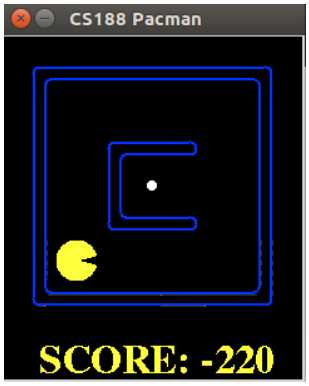

Figure 1: The layout to use with your first MDP solver

3 A Map for Pacman

If you didn't figure out the map building bit of last week's practical, there is code on KEATS which does most of what is necessary. To use it:

1. Download `mapAgents.py` from KEATS.
2. Run the code using:

   ```
   python pacman.py -p MapAgent
   ```

Even if you wrote your own map-handing version of Pacman, you might find this worth looking at. For one thing it provides a reasonably general way to build grid-based representations using just native Python (no libraries necessary)in the form of the class `Grid`. For another, it demonstrates a way of initialising the agent controlling Pacman with game-state information. This is done by using `registerInitialState()`, and if you haven't figured out how to use that yet, it is worth taking a look at `MapAgent` to see how to use it.

Note that `MapAgent` does **not** provide code for computing the greedy action — that bit is too central to the coursework for it to make sense for us to give it to you.

4 Bellman makes an appearance

As you may recall from the lecture and tutorial, the process for value iteration is:

1. Set utilities to an arbitrary value U_0
2. Update utilities according to:

   $$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s')$$

   until the utilities no longer change

The update step:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U_i(s')$$

says that the utility of state s at iteration $i+1$ should be set to the reward for state s plus the discounted maximum expected utility of the legal action with the highest expected utility (where the expected utilities are computed using the utilities from iteration i). If you finished the practical from last week, you will already have code that computes the maximum expected utility of all the available actions for a given state, so implementing the maximum expected utility bit of the update should be easy. If you didn't finish the practical from last week, look at Section 5 for a hint as to how to do this computation.

There will be a few things that are tricky, or at least take a bit of experimenting.

One will be finding a good discount value γ. We know this has to be greater than 0 and less than or equal to 1, but it may take some experimenting to get a value that makes Pacman perform well.

Another will be deciding how to set the reward $R(s)$ for a given state. For the initial layout you can set the reward of the state with food in it to a fixed value[2] (the value that Pacman gets for eating the food might be a good value) and the reward for other states might be set to -1, the "reward" that Pacman gets for each step that the game lasts.

Finally, you may need to adjust the termination condition for value iteration. As we said in the lecture, value iteration is only guaranteed to provide the utilities under the optimal policy if it runs for an infinite number of iterations. However, it is safe to terminate the process after a finite number of iterations when the utilities are not changing any more. (You can figure out how to decide when they aren't changing.) If that is taking too long, you can get an approximation to the right values (which will probably be good enough to win games) if you terminate value iteration when the utilities don't change more than some limit. (You have to experiment to find a good limit.) And if you need the process to run even faster than that, you can always stop value iteration after a certain number of iterations. Again you have no guarantee that the utilities are close to the right values, but experimenting should tell you whether Pacman can still use them effectively.

5 Allow Pacman to make decisions

Once Pacman has a map with utilities in it, they are ready to decide what to do and are able to make rational/optimal decisions. Since movement is (still) non-deterministic, Pacman will, naturally, use the principle of maximum expected utility to decide their next move. (So, yes, this means we use MEU in two places — one in the utility update in value iteration, and then again when picking the best action once value iteration is done.)

As before, one way to do that is as follows:

1. From Pacman's position, identify all the legal moves.
2. For each move, use the map and the motion model to determine:
   (a) All the possible outcomes of the move.
   (b) The probability of the outcomes.
   (c) The utilities of the outcomes (the utilities of the spaces that the move will take Pacman to).
3. From (a), (b) and (c) it is possible to calculate the expected utility of each move.

[2] Once you have more than one food in the layout, you may need to change this if you want to be able to play the game to a win with only one run of value iteration.

4. From the expected utility of each move, it is possible to establish the move with the maximum expected utility.

Of course, this is exactly the same thing that you wrote for `MyGreedyAgent`, but it is using the utilities computed by value iteration rather than the rewards you read from the environment.

This should be able to win games very quickly in `smallMDPGrid`, and is a solid basis for the coursework.

Getting the `SimpleMDPAgent` to a point where it can run value iteration and win games in `smallMDPGrid` is the minimum you should aim to get done this week. You will likely need more than the one hour of scheduled lab time to do this.

6 More

Other things to think about

1. As described, Pacman is using the utility values of each space to decide what to do each time it needs to take an action. An alternative is to have Pacman compute the complete policy. That is we could have Pacman look at every space, decide what is the best thing to do in that space, and record it. Then, each time Pacman needs to choose an action, it just looks up what to do. Implement this.
2. Experiment with different decision criteria. Do you see any difference in performance?
3. How would you go about implementing policy iteration?

7 Version list

- Version 1.2, October 8th 2019 – fixed typos.
- Version 1.1, September 27th 2019 – fixed typos.
- Version 1.0, September 23rd 2019.