6CCS3ML1 (Machine Learning)

Coursework 2

(Version 1.3)

1 Overview

For this coursework, you will have to implement a reinforcement learning algorithm. Your code will again be controlling Pacman, in the classic game, and the reinforcement learning algorithm will be helping Pacman choose how to move. Your reinforcement learning algorithm should be able, once it has done its learning, to be able to play a pretty good game, and its ability to play a good game will be part of what we assess.

No previous experience with Pacman (either in general or with the specific UC Berkeley AI implementation that we use) is required, though you should have used it for Coursework 1 and Practical 8.

This coursework is worth 10% of the marks for the module.

Note: Failure to follow submission instructions will result in a deduction of 10% of the marks you earn for this coursework.

2 Getting started

2.1 Start with Pacman

The Pacman code that we will be using for the 6CCS3ML1 coursework was developed at UC Berkeley for their AI course. The person who developed this code then kindly made it available to everyone. The homepage for the Berkeley AI Pacman projects is here:

`http://ai.berkeley.edu/`

Note that we will **not** be doing any of their projects. Note also that the code only supports Python 2.7, so that is what we will use[1].

You should:

1. Download:

   pacman-cw2.zip

   from KEATS.
2. Save that file to your account at KCL (or to your own computer).
3. Unzip the archive.

   This will create a folder `pacman-cw2`

[1] If you insist on using Python 3, you are on your own in terms of support, and if the code you submit does not work (which is likely) you will lose marks.

4. From the command line (you will need to use the command line in order to use the various options), switch to the folder `pacman-cw2`.

5. From KEATS, download the file:

   `sampleAgents.py`

   and put it in the folder `pacman-cw2`

6. Now type:

   `python pacman.py -p RandomAgent -n 10 -l smallGrid`

   and watch it run.

This command illustrates a few aspects of the Pacman code that you will need to understand:

- `-n 10` runs the game 10 times.
- `-l smallGrid` runs the very reduced game you see in Figure 1.

  This is not a very interesting game for a human to play, but it is moderately challenging for a reinforcement learning program to learn to play.
- `-p RandomAgent` tells the `pacman.py` code to let Pacman be controlled by an object that is an instance of a class called `RandomAgent`.

  The program then searches through all files with a name that **ends** in:

  `Agents.py`

  looking for this class. If the class isn't in an appropriately named file, you will get the error:

```
Traceback (most recent call last):
  File "pacman.py", line 679, in <module>
    args = readCommand( sys.argv[1:] ) # Get game components based on input
  File "pacman.py", line 541, in readCommand
    pacmanType = loadAgent(options.pacman, noKeyboard)
  File "pacman.py", line 608, in loadAgent
    raise Exception('The agent ' + pacman + ' is not specified in any *Agents.py.')
```

  In this case the class is found in `sampleAgents.py`

`RandomAgent`, as its name suggests, just picks actions at random. When there is no ghost, it will win a game eventually by eating all the food[2], but when a ghost is present, `RandomAgent` dies pretty quickly.

2.2 Towards an RL Pacman

Now, your job is to write code to learn how to play this small version of Pacman. To get you started, we have provided a skeleton of code that will do this. You can download this code in the file:

`mlLearningAgents.py`

[2] If you want to see this happen, run `python pacman.py -p RandomAgent -k 0 -l smallGrid`. The flag `-k` sets the number of ghosts.

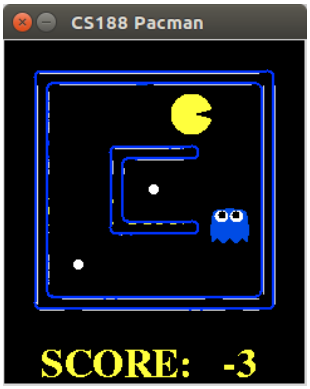

Figure 1: The `smallGrid` version of Pacman

from KEATS. This file contains a class `QLearnAgent`, and that class includes the following methods. (You should look at the code in order to fully understand this description.)

`init()` This is the constructor for `QLearnAgent`. It is called by the game when the game starts up (because the game starts up the learner).

The version of `init()` in `QLearnAgent` allows you to pass parameters from the command line. Some of these you know from the lectures:

- `alpha`, the learning rate
- `gamma`, the discount rate
- `epsilon`, the exploration rate

and you will use them in the implementation of your reinforcement learning algorithm. The other:

- `numTraining`

allows you to run some games as training episodes and some as real games.

All the constructor does is take these values from the command line, if you pass them, and write them into sensibly named variables. If you don't pass values from the command line, they take the default values you see in the code. These values work perfectly well, but if you want to play with different values, then you can do that like this:

```
python pacman.py -p QLearnAgent -l smallGrid -a numTraining=2 -a alpha=0.2
```

Note that you need to have no space between parameter and value `alpha=0.2`, and you need a separate `-a` for each one.

`getAction()` This function is called by the game every time that it wants Pacman to make a move (which is every step of the game).

This is the core of code that controls Pacman. It has access to information about the position of Pacman, the position of ghosts, the location of food, and the score. The code shows how to access all of these, and if you run the code (as above) all these values are printed out each time `getAction()` is called.

The only bit that maybe needs some explanation is the food. Food is represented by a grid of letters, one for each square in the game grid. If the letter is `F`, there is no food in that square. If the letter is `T`, then there is food in that square. (Walls are not represented).

The main job of `getAction()` is to decide what move Pacman should make, so it has to return an action. The current code shows how to do that but just makes the choice randomly.

`final()` This function is called by the game when a Pacman has been killed by a ghost, or when Pacman eats the last food (and so wins).

Right now all this function does is to keep track of the number of episodes (more on what they are below) and sets `epsilon` and `alpha` to zero when a certain number have happened (more on that in a minute also).

3 What you have to do (and what you aren't allowed to do)

3.1 Write some code

Now you should fill out `QLearnAgent` with code that performs reinforcement learning.

Because of the way that Pacman works, your learner needs to include two separate pieces:

1. Something that learns, adjusting utilities based on how well the learner plays the game.
2. Something that chooses how to act. This decision can be based on the utilities, but also has to make sure that the learner does enough exploring.

You need to do that because using Pacman forces you to do online learning. The only way that Pacman learns is by actually playing in a game. Initially they will play badly, but over time they should get better and better. However, they will learn slowly. Indeed, they will need to play around 2000 games to get to be any good.

See Section 4 for a description of what to submit. Note that we will evaluate your code by running the command:

```
python pacman.py -p QLearnAgent -x 2000 -n 2010 -l smallGrid
```

so we will train the learner for 2000 episodes and then run it for 10 non-training episodes (see below). We will use the score that your learner gets in those 10 episodes as a way to decide how good your solution is. To get good marks, your `QLearnAgent` needs to win around 8 of the games. That is very achievable.

3.2 Things to know

Here are some things to think about when writing your learning code:

- Your code must contain a recognisable reinforcement learning algorithm. Code that does not contain such an algorithm will lose marks.
- What learning reinforcement algorithm you implement is up to you. It should be one of the methods we covered in Lectures 7 or 8, but you are free to pick between them. I used Q-learning[3], but other methods will work just as well. (Indeed, I know that adaptive dynamic programming can work really well here, learning to win in less than 100 episodes.)

[3] Hence `QLearnAgent`, since that is a stripped down version of my code.

The marks you get for your code will be partly determined by the sophistication of the reinforcement learner that you implement. If you just use a simple bandit learner, that will not get as much credit as a Q-learner or an implementation of adaptive dynamic programming. (And if you use the bandit learner that is in the solution posted for Practical 8 without acknowledgement, that is plagiarism.)

- The code has some support for learning. First, you can run several games one after another by using the `-n` parameter. This:

  ```
  python pacman.py -p QLearnAgent -l smallGrid -n 5
  ```

  will run the same game 5 times. Each of these runs is one of the *episodes* that were mentioned in the text about `final()`. The really useful bit is that the same instance of `QLearnAgent` gets run all five times, so that it can learn over all five episodes.

  Even more helpful, you can do some of the runs without the graphical interface, so that they run quickly. If you run:

  ```
  python pacman.py -p QLearnAgent -l smallGrid -x 5 -n 10
  ```

  it will do the first 5 runs without the interface. The remainder of the runs will happen with the interface. During the training runs, the system does not record the scores, so the average score (and results) printed at the end do not include the results of the training episodes.

  The command-line switch `-numTraining` is just a synonym for `-x`, so the value you specify with `-x` is the value that ends up in the variable `numTraining` in the code. This is why `final()` counts the number of episodes and compares it to `numTraining` — that is how the agent knows when training is over and it is showtime. We set `epsilon` and `alpha` to zero at that point because (as you know from the lecture) these are parameters that control learning. An ϵ-greedy learner chooses not to do the best action that it knows with a probability ϵ, so if we set ϵ to zero the learner will always do what it thinks is best (and it won't get killed because it does something random)[4]. Similarly, if you set α to zero in a Q-learning/SARSA/temporal difference learner, then the update doesn't change the Q-values/utilities.

- Though, as the lecture says, in theory you need to adjust the learning rate over time, I did not find that was necessary.[5]

- There is no way to access the reward for winning (or the cost of losing) outside `final()`. (`getAction()` is not called once the episode is over.) You will need those rewards to learn how to avoid the ghost and how to win the game.[6]

3.3 Limitations

There are some limitations on what you can submit.

1. Your code should be in Python 2.7.

   Code written in a language other than Python will not be marked.

   Code written in Python 3.X is unlikely to run with the clean copy of `pacman-cw2` that we will test it against. If it doesn't run, you will lose marks.

[4] It might still make a bad move, but it won't know it is a bad move.

[5] It is quite possible that the utilities did not converge as a result, or that they did not converge on the optimal values, but the learner could play a good game, and that is what matters.

[6] Without those rewards, my learner will choose to run towards the ghost to end the game quickly and minimise its losses — very rational when you don't get the big positive reward for winning.

The reason for this is that we do not have the resources to deal with code written in multiple languages, and to ensure that we can run code written in Python 3.X.

2. Code using libraries that are not in the standard Python 2.7 distribution *may* not run when we test it. If you choose to use such libraries and your code does not run when we test it, you will lose marks.

   The reason for this is that we do not have the resources to deal with setting up arbitrarily complex environments (with the possibility of libraries with arcane interactions) for every submission.

3. You are not allowed to modify any of the files in `pacman-cw2.zip`.

   In fact, the only thing you can do as part of the coursework is to write code in `mlLearningAgents.py`.

   The idea is that everyone solves the same problem — you can't change the problem by modifying the base code that runs the Pacman environment.

4. You are not allowed to copy, without credit, code that you might get from other students or find lying around on the Internet. (This includes the use of code that was distributed as part of the module — if you use code from files other than `mlLearningAgents.py` without attribution, we will consider that to be plagiarism.) We will be checking.

   This is the usual plagiarism statement. When you submit work to be marked, you should only seek to get credit for work you have done yourself.

   When the work you are submitting is code, you can use code that other people wrote, but you have to say clearly that the other person wrote it — you do that by putting in a comment that says who wrote it. That way we can adjust your mark to take account of the work that you didn't do.

5. You may be tempted to use code from the files `qlearningAgents.py` or `learningAgents.py` that are part of the Berkeley AI Pacman project. These files provide a big part of the solution to the problem of using reinforcement learning to play Pacman, so if you use them, you are avoiding doing an important part of the work of the coursework.

   If you use them and acknowledge it, we will deduct marks because you have avoided doing some of the work we want you to do. If you use them and do not acknowledge it, that is plagiarism, and you will have to answer to the relevant Misconduct committee.

6. Following on the last point, the code you submit must be in a class called `QLearnAgent`, and as in `mlLearningAgents.py`, this class must be a direct subclass of `Agent`:

   ```
   class QLearnAgent(Agent):
   ```

   This, again, is intended to make sure that everyone solves the same problem, starting from the basic Pacman `Agent`.

4 What you have to hand in

Your submission should consist of a single ZIP file. (KEATS will be configured to only accept a single file.) This ZIP file must include a single Python file (your code).

The ZIP file must be named:

```
cw2-<lastname>-<firstname>.zip
```

so my ZIP file would be named `cw2-cocarascu-oana.zip`. Submissions that are not in zip format will lose marks.

Remember that we are going to evaluate your code by running your code using variations on

```
python pacman.py -p QLearnAgent -x 2000 -n 2010 -l smallGrid
```

and we will do this in a vanilla copy of the `pacman-cw2` folder, so the base class for your agent must be called `QLearnAgent`.

To streamline the marking of the coursework, you must put all your code in one file, and this file must be called `mlLearningAgents.py`. If your file is named otherwise, you will lose marks.

Do not just include the whole `pacman-cw2` folder. You should only include the one file that includes the code you have written.

Submissions that do not follow these instructions will lose marks.

5 How your work will be marked

There will be six components of the mark for your work, four concerned with functionality and two with form.

1. Functionality

   We will test your code by running your `.py` file against a clean copy of `pacman-cw2`.

   (a) As discussed above, for full marks for functionality, your code is required to run when we invoke the command:

   ```
   python pacman.py -p QLearnAgent -x 2000 -n 2010 -l smallGrid
   ```

   If your code does not run, you will lose marks.

   (b) We will also look at your code for evidence of the use of reinforcement learning. Code that does not use RL will lose marks.

   (c) When we run your code using:

   ```
   python pacman.py -p QLearnAgent -x 2000 -n 2010 -l smallGrid
   ```

   it is required to win 8 of 10 games. (If it fails to do this on the first run, we will run it again and average the number of wins.) If your code wins less than 8 games, on average, you will get a mark that reflects how many games your code wins — more wins equals more marks.

   Note that, since we have a lot of coursework to mark, we will limit how long your code has to demonstrate that it can win 8 games out of 10. We will terminate the run after five minutes. If your code has won less than 8 games within these five minutes, you will get a mark that reflects how many games your code has won — more wins equals more marks.

   (d) Finally, to test generalisability, we will run your code on another grid. Code that fails to run on another grid will lose marks. We will not test the performance of your code, that is how many games it wins, we will only test if it runs. (There is no point in asking us what grid we will test your code on because we won't tell you. For one thing, the point is that the code should not be making assumptions about which grid it is running on

and so neither should you. For another, we know that if you know which grid, some of you will write code that exploits particular features of the grid to improve performance, which is exactly what makes for bad solutions.)

2. Form

   (a) There are no particular requirements on the way that your code is structured, but it should follow standard good practice in software development and will be marked accordingly.

   (b) All good code is well documented, and your work will be partly assessed by the comments you provide in your code. If we cannot understand from the comments what your code does, then you will lose marks.

A copy of the marksheet, which shows the distribution of marks across the different elements of the coursework, will be available from KEATS.

Version list

- Version 1.0, March 17th 2019
- Version 1.1, March 27th 2019.
- Version 1.2, March 13th 2020.
- Version 1.3, February 22nd 2021.