Introduction

This week, we will have a look at how AI can be used to create players for video games. We will take as our example using genetic algorithms to evolve a Pac-Man player.

Since you will be using genetic algorithms, if you need an introductory tutorial, you can follow this example before working on the worksheet.

https://towardsdatascience.com/genetic-algorithm-implementation-in-python-5ab67bb124a6

Getting Started

Download the Pac-Man simulator from https://github.com/jspacco/pac3man - this is a Python 3 version of a simulator that was developed to teach AI at the University of California, Berkeley a few years ago. If you want to use this for your project, there are loads of details about it at http://ai.berkeley.edu/project_overview.html

In the downloaded folder, go to the directory called *search*. You will not be using anything outside of this directory for the tasks in this class. Download *runGames.py* and *newAgents.py* from the module Moodle page, and add them into the *search* directory.

Run the following command:

```
python pacman.py --pacman LeftTurnAgent
```

you should see a game window appear, where the Pac-Man is controlled by an agent (defined in *pacmanAgents.py*) that makes it turn left at every opportunity. This is basically how this system works—to add a new behaviour, you define a subclass of the *Agent* class and initialise the Pac-Man (or the Ghosts) with this behaviour. Have a look at the *pacmanAgents.py* class and try to understand broadly what is going on—there is a *getAction* method which is called at each timestep, and which returns the direction that the Pac-Man moves.

Tasks

1. *NewAgent*1 is a new Pac-Man playing agent that works with the existing simulator, it is defined in the *newAgents.py* file. It contains a grid called the *code* that assigns a direction (from *Directions.NORTH, Directions.EAST,* etc.) to each point in the maze. When the Pac-Man reaches that point, it tries to go in that direction, and if that choice is not possible, it goes in a random direction.

Run *runGames.py*. The simulator should appear, and the Pac-Man will be controlled by a code where every square is filled in with *Directions.EAST*. That means that the Pac-Man will go east where it can, otherwise it will go in a random direction. So, it works its way towards the right side of the maze, and then gets stuck there.

2. Let's see if we can use a simple learning method to improve the movement of the Pac-Man. Comment out *runTest()* at the end of *runGames.py*, and uncomment *runGA()*.

This runs a genetic algorithm that evolves the *code* grid. 20 random code grids are created, and agents created that run the code using these grids. The score from running the agent twice using each code in the population is run (this is not really enough to get a good average, but we are going to run these simulations a lot so this makes this tractable in the class time that we have).

Where the code says:

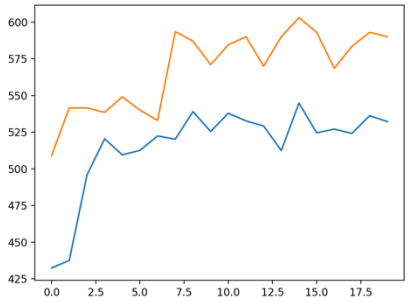
ADD CODE TO PLOT averages AND bests

add code to do that—i.e. plot both the average and best on the same graph. Have a look at this if you need a reminder about how to do this:

https://howtothink.readthedocs.io/en/latest/PvL_H.html

Set *beQuiet* to *True*. This will mean that the simulations will run without graphics, and therefore be much faster.

You should end up with a graph like this, where the orange line is the best performance in each generation, and the blue the average.



Add labels, error-bars etc. as you see fit.

3. Add two additional features where indicated in the code. The first is a *crossover* between two parents, chosen in the same way as the one for mutation (i.e. hold a "tournament" between 7 randomly chosen codes, and choose the best). You will need to create two parents using two tournaments, then create a new code by e.g. randomly choosing at each point from parent one or parent two (the best way to organise this is to define a *crossover* function similar to the existing *mutation* one). Alternatively, cross-over some randomly chosen "blocks" in the grid (this might work better as it preserves areas of the code that work well together.

The second feature is to always keep the highest-scoring code in the current population into the next one, so that we never lose the best one. Add this where indicated—this should only take a line or two of code.

Compare the performance of your code with these changes compared with the earlier version (the changes may be fairly subtle because of the small number of runs—after the class perhaps try some experiments with lots more runs of the simulation).

4. (harder) At present the agent doesn't respond to the ghost. Add some code to *NewAgent1* so that the Pac-Man responds to the ghost in some way (e.g. if it is close to the ghost, then it starts ignoring the *code* and moves away from the ghost). You could incorporate this into the genetic algorithm in some way—e.g. an additional parameter in the code could be the threshold distance at which the agent switches between the behaviours). There is some commented-out code in

5. (harder) Change the movement in *NewAgent1*. Perhaps, rather than having a fixed direction at each position in the *code*, you have a list of weights for choosing each direction. Or, perhaps the Pac-Man continues in its current direction until it cannot move in that direction any more, then looks at the grid value (you would have to add a "previous move" variable into the class, using an __init__ method to set it initially).

Extensions

1. The current representation is rather crude and inflexible—we have to learn a whole grid of directions, and there are lots of similar situations around the grid (e.g. there are many "corners"). Rewrite the representation so that the GA evolves *responses* to being in those

situations. For example, one situation is a “NW corner”, one is a “horizontal corridor”, one is a “ghost ahead to the right”, and each has an entry in a vector which is the thing that is evolved by the GA. Perhaps make these relative to the Pac-Man position rather than absolute (not “horizontal corridor” but “corridor aligned with Pac-Man direction”; not “turn West” but “turn left” etc.). Once you have implemented this, see if you can transfer the learned model to another layout (this is specified by the *layout* variable in *runGames.py*)—there is a list of layouts in the *layout* directory. This could be a very interesting project, with the question being about how well the strategies transfer.

2. Incorporate some strategies about the food and the “capsules”—the special food objects that make the ghosts edible.

3. Experiment with parameters in the genetic algorithm such as population size and number of runs of the simulator—see how much difference these make.

4. Have a look at http://ai.berkeley.edu/project_overview.html and implement some of the reinforcement learning approaches described there.