6CCS3ML1 (Machine Learning)

Practical 09: Evolutionary Algorithms

(Version 1.1)

1 Overview

This week the practical looks at evolutionary algorithms. In particular you will get the chance to use a genetic algorithm to develop a control mechanism that will help an agent decide what to do. Naturally Pacman comes into this. The GA will be learning weights that Pacman then uses to decide what to do in response to the same feature set that we used in coursework 1.

2 Getting started

You should:

1. Download:

   `evolutionary.zip`

   from KEATS.

   This is a copy of the usual code from UC Berkeley, but with a few additional files that I set up. So be sure to download it rather than just using a copy of what you had from before.

2. Save that file to your computer.

3. Unzip the archive.

   This will create a folder `evolutionary`

From the command line you operate this just as before. Switch to the folder `evolutionary` and type:

```
python pacman.py -p EvolAgent
```

This then runs the `EvolAgent` controller on Pacman. It currently doesn't do much, but that is because you haven't evolved anything interesting.

To learn a controller using a GA, we need two things. We need an implementation of a GA, and we need a way of evaluating the individuals generated by the GA. You already met the second of these. That is `EvolAgent`. It takes the output of a GA, stored in the file `weights.txt` and uses it to control Pacman. The other piece is the file `run.py`. This implements the GA. The GA starts by generating a random population, evaluates each member by running Pacman games (using `EvolAgent`) and uses the results of the evaluation to rank members of the population. The ranked members of the population are then used to create a new population. Figure 1 shows a typical GA run. This shows the average performance of the best individual in each of 50 generations.

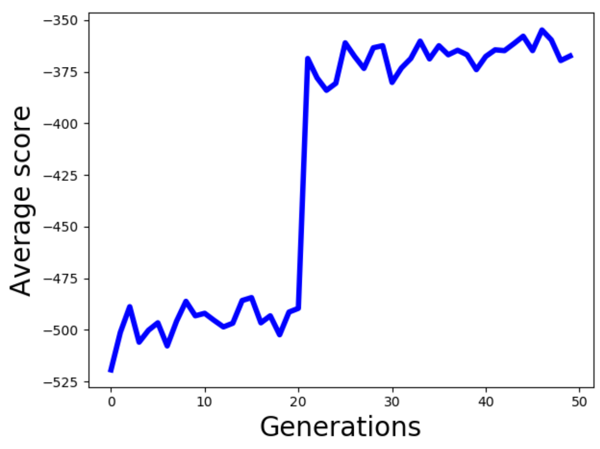

Figure 1: The performance of the genetic algorithm, as measured by the average score of the best individual in each generation.

To demonstrate that this can do something useful, copy `simons-best.txt` over the contents of `weights.txt`. Then run:

```
python pacman.py -p EvolAgent
```

What you should see is Pacman move around a little, and, if you run it a few times, you may even see it react to the ghosts. This behaviour was learnt using the GA.

3 What you need to do

Take a look at `run.py` using your favourite editor.

The first few lines of the file include:

```
populationSize = 50  # How many individuals
generations    = 20  # How many generations to evolve over
mutationRate   = 0.1 # Probability of flipping a bit in a mutated individual
```

As the comments say, these are the main parameters that control the GA. `populationSize` defines the number of individuals in a population, `generations` controls how many generations the GA runs for, and `mutationRate` controls the likelihood that a bit within a string will be flipped from 1 to 0, or 0 to 1[1].

Then we have:

```
exploitPercent = 20 # %age of the population that is exploited
randomPercent  = 10 # #age of the population that is randomly generated
mutatePercent  = 10 # %age of the population that is mutated
crossPercent   = 60 # %age of the population created by crossover
```

[1]Here in the code you will also find `weights.txt`. This specifies the length of the binary string the GA manipulates, and is determined by the controller in `EvolAgent`.

These determine how much of the second and later generations are produced from the previous generation by exploitation, random generation, mutation and through crossover.

You can run the GA using:

```
python run.py
```

at the end of every generation a message will be printed. When the evolution is complete, you will find the best member of the final generation in `best-weights.txt`. (Just copy this to `weights.txt` to try it out).

A first thing to do is to try playing with these parameters and see if you can generate any better individuals than the one in `simons-best.txt`. Since the current code only puts the best individual in the last generation int `best-weights.txt`, you might want to do something more sophisticated, like recording the best individual(s) in every generation and then looking through this set for a good performer.

The second thing to do is to try and write a better GA. The code in `run.py` uses very unsophisticated methods for exploitation, mutation and crossover. You can easily write a better GA by changing the functions that implement these bits. You can find the functions between lines 72 and 190 in `run.py` (helpfully identified by comments). Try some of the methods from Lecture 9.

4 How it works

You can do this practical without understanding anything about how the result of the GA controls Pacman. However, in case you are interested, this is what happens.

`run.py` creates (using the GA) a binary string. This string is written into the file `weights.txt` `run.py` then calls the Pacman game and runs it 5 times (to do a bit to reduce the effect of the randomness in the game). It does this by calling the game as if it had been called from the command line. That is what the function `runOneAgent()` does. (It actually runs a copy of `pacman.py`, called `evolPacman.py` so that I could suppress some of the things that `pacman.py` does that were annoying.) Each game uses `EvolAgent` to play Pacman.

`EvolAgent`, in `evolAgents.py`, reads `weights.txt`, and extracts the binary string generated by the GA. It breaks the string into a series of 4 digit binary numbers, and turns these into their decimal equivalent. These are treated as two sets of weights and two thresholds.

`EvolAgent` pretends that Pacman is a robot controlled by two motors, where each motor is controlled by the output of a perceptron connected to the robot's sensors. For both left and right motors, `EvolAgent` combines the values in the feature vector it gets from its "sensors" (just the features generated by `api.py`) with a set of weights to get a number. This is compared with one of the thresholds. The four combinations of left and right weights being above or below their thresholds are used to decide whether Pacman moves NORTH, SOUTH, EAST or WEST[2]. So Pacman is being controlled by a pair of classifiers, each of which is trained by using a GA that gets feedback by using the current state of the classifier to evaluate that classifier.

The five games generate an average score, and `run.py` attaches this weight to the relevant individual. When a whole generation has been evaluated, the list of individual/score pairs are sorted, and this is then used to create the next generation.

[2]If Pacman was a robot controlled by a pair of perceptrons, the decision made by each one would be whether to turn the relevant motor to run forwards or backwards, and it would have roughly the same effect, either driving Pacman forward or backwards, or causing Pacman to turn clockwise or anti-clockwise.

5 Version list

- Version 1.0, March 1st 2020.
- Version 1.1, February 9th 2021.