Axelrod-dojo Documentation

The Axelrod project developers

Sep 01, 2018

Contents

This library is a companion library to the Axelrod library: a research tool for the study of the iterated prisoners dilemma. The **Axelrod-dojo** is used to train strategies.

This is done using implementations of:

- Strategy archetypes `Parameters`
- Algorithms

CHAPTER 1

Table of Contents

1.1 Tutorial

In this tutorial we will aim to find the best Finite State Machine against a collection of other strategies from the Axelrod library *[Harper2017]*.

First let us get the collection of opponents against which we aim to train:

```
>>> import axelrod as axl
>>> opponents = [axl.TitForTat(), axl.Alternator(), axl.Defector()]
>>> opponents
[Tit For Tat, Alternator, Defector]
```

We are now going to prepare the training algorithm. First of all, we need to prepare the objective of our strategy. In this case we will aim to maximise `score` in a match with `10` turns over `1` repetition:

```
>>> import axelrod_dojo as dojo
>>> objective = dojo.prepare_objective(name="score", turns=10, repetitions=1)
```

The algorithm we are going to use is a genetic algorithm which requires a population of individuals. Let us set up the inputs:

```
>>> params_class = dojo.FSMParams
>>> params_kwargs = {"num_states": 2}
```

Using this we can now create our Population (with `20` individuals) for a genetic algorithm:

```
>>> axl.seed(1)
>>> population = dojo.Population(params_class=params_class,
...                               params_kwargs=params_kwargs,
...                               size=20,
...                               objective=objective,
...                               output_filename="training_output.csv",
...                               opponents=opponents,
```

(continues on next page)

(continued from previous page)

```
...                                 bottleneck=2,
...                                 mutation_probability=.1)
```

We can now evolve our population:

```
>>> generations = 4
>>> population.run(generations)
Scoring Generation 1
Generation 1 | Best Score: 2.1 0:C:0_C_0_C:0_D_1_C:1_C_1_D:1_D_1_D
Scoring Generation 2
Generation 2 | Best Score: 2.1 0:C:0_C_0_C:0_D_1_C:1_C_1_D:1_D_1_D
Scoring Generation 3
Generation 3 | Best Score: 2.1 0:C:0_C_0_C:0_D_1_C:1_C_1_D:1_D_1_D
Scoring Generation 4
Generation 4 | Best Score: 2.1 0:C:0_C_0_C:0_D_1_C:1_C_1_D:1_D_1_D
```

The `run` command prints out the progress of the algorithm and this is also written to the output file (we passed `output_filename` as an argument earlier). The printing can be turned off to keep logging to a minimum by passing `print_output=False` to the `run`.

The last best score is a finite state machine with representation `0:C:0_C_0_C:0_D_1_D:1_C_1_D:1_D_1_D` which corresponds to a strategy that stays in state `0` as long as the opponent cooperates but otherwise goes to state `1` and defects forever. Indeed, if the strategy is playing `Defector` or `Alternator` then it should just defect, otherwise it should cooperate.

1.2 How to

1.2.1 Use different objective functions

It is currently possible to optimise players for 3 different objectives:

- Score;
- Score difference;
- Probability of fixation in a Moran process.

This is done by passing a different objective `name` to the `prepare_objective` function:

```
>>> import axelrod_dojo as dojo
>>> score_objective = dojo.prepare_objective(name="score", turns=10, repetitions=1)
>>> diff_objective = dojo.prepare_objective(name="score_diff", turns=10,␣
↪repetitions=1)
>>> moran_objective = dojo.prepare_objective(name="moran", turns=10, repetitions=1)
```

1.2.2 Train using the genetic algorithm

WIP: include all details for training with genetic algorithm.

1.2.3 Train using the particle swarm algorithm

WIP: include all details for training with PSO

1.3 Background

Note that there are currently two algorithms implemented:

- Genetic algorithm
- Particle swam optimisation

Note that these two algorithms are not equally suited to each archetype. For example the Genetic algorithm is believed to be better suited to discrete space strategies such as the finite state machines whilst the Particle swarm algorithm would be better suited to a continuous space strategy such as the Gambler.

For more information on these algorithms and their implementations see:

1.3.1 Genetic Algorithm

A genetic algorithm aims to mimic evolutionary processes so as to optimise a particular function on some space of candidate solutions.

The process can be described by assuming that there is a function $f : V \to \mathbb{R}$, where V is some vector space. In the case of the Prisoner's dilemma, the vector space V corresponds to some representation of a particular archetype (which might not actually be a numeric vector space) and the function f corresponds to some measure of performance/fitness of the strategy in question.

In this setting a candidate solution $x \in \mathbb{R}^m$ corresponds to a chromosome with each x_i corresponding to a gene.

The genetic algorithm has three essential parameters:

- The population size: the algorithm makes use of a number of candidate solutions at each stage.
- The bottle neck parameter: at every stage the candidates in the population are ranked according to their fitness, only a certain number are kept (the best performing ones) from one generation to the next. This number is referred to as the bottle neck.
- The mutation probability: from one stage to the next when new individuals are added to the population (more about this process shortly) there is a probability with which each gene randomly mutates.

New individuals are added to the population (so as to ensure that the population size stays constant from one stage to the next) using a process of "crossover". Two high performing individuals are paired and according to some predefined procedure, genes from both these individuals are combined to create a new individual.

For each strategy archetype, this library thus defines a process for mutation as well as for crossover.

Finite state machines

A finite state machine is made up of the following:

- a mapping from a state/action pair to another target state/action pair
- an initial state/action pair.

(See *[Harper2017]* for more details.)

The crossover and mutation are implemented in the following way:

- Crossover: this is done by taking a randomly selected number of target state/actions pairs from one individual and the rest from the other.

- Mutation: given a mutation probability δ each target state/action has a probability δ of being randomly changed to one of the other states or actions. Furthermore the **initial** action has a probability of being swapped of $\delta \times 10^{-1}$ and the **initial** state has a probability of being changed to another random state of $\delta \times 10^{-1} \times N$ (where N is the number of states).

Hidden Markov models

A hidden Markov model is made up of the following:

- a mapping from a state/action pair to a probability of defect or cooperation.
- a cooperation transition matrix, the probability of transitioning to each state, given current state and an opponent cooperation.
- a defection transition matrix, the probability of transitioning to each state, given current state and an opponent defection.
- an initial state/action pair.

(See *[Harper2017]* for more details.)

The crossover and mutation are implemented in the following way:

- Crossover: this is done by taking a randomly selected number of rows from one cooperation transition matrix and the rest from the other to form a target cooperation transition matrix; then a different number of randomly selected rows from one defection transition matrix and the rest from the other; and then a randomly select number of entries from one state/part -> probability mapping and the rest from the other.
- Mutation: given a mutation probability $delta$ each cell of both transition matrices and the state/part -> probability mapping have probability $delta$ of being increased by $varepsilon$, where $varepsilon$ is randomly drawn uniformly from $[-0.25, 0.25]$ (A negative number would decrease.) Then the transition matrices and mapping are adjusted so that no cell is outside $[0, 1]$ and the transition matrices are normalized so that each row adds to 1. Furthermore the **initial** action has a probability of being swapped of $\delta \times 10^{-1}$ and the **initial** state has a probability of being changed to another random state of $\delta \times 10^{-1} \times N$ (where N is the number of states).

Cycler Sequence Calculator

A Cycler Sequence is the sequence of C & D actions that are passed to the cycler player to follow when playing their tournament games.

the sequence is found using genetic feature selection:

- Crossover: By working with another cycler player, we take sections of each player and create a new cycler sequence

from the following formula:

let our two player being crossed be called p1 and p2 respectively. we then find the midpoint of both the sequences and take the first half from p1 and the second half from p2 to combine into the new cycler sequence.

- Mutation: we use a predictor :math:‘delta‘to determine if we are going to mutate a

single element in the current sequence. The element, or gene, we change in the sequence is uniformly selected using the random `package`.

1.4 Reference

This section is the reference guide for the various components of the library.

1.4.1 Bibliography

This is a collection of various bibliographic items referenced in the documentation.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[Harper2017] Marc Harper, Vincent Knight, Martin Jones, Georgios Koutsovoulo, Nikoleta E. Glynatsi and Owen Campbell (2017) Reinforcement Learning Produces Dominant Strategies for the Iterated Prisoner's Dilemma. Arxiv. http://arxiv.org/abs/1707.06307