Multi-agent learning programming assignment: the grand table, Nash equilibria, and the replicator dynamic

Jiayuan Hu, Adrián Rodríguez Otero, Yolan van der Horst and Gerard Vreeswijk
Multi-agent Learning Course 2020/2021, Utrecht University

Introduction

The goal of this assignment is to compare a number of multi-agent learning algorithms against each other in a tournament. For this you will create a program that can calculate a so-called *grand table* for different types of matrix games. You will then be asked to analyse the grand table in an evolutionary sense using the replicator dynamic and with regards to Nash equilibria using the Gambit tool. You will also have to submit a report where you present the implementation of the algorithms and discuss your findings.

Sections 1 and 2 explain the requirements and what you will have to submit. Section 3 contain more detailed instructions for creating the grand table, including which algorithms that should be implemented. Sections 4 and 5 explain how to implement the replicator dynamic and how to use Gambit for calculating Nash equilibria respectively.

Contents

1 Requirements

1. Work in groups of 2.
2. The program is written in Python, using the provided Python scripts.
3. The study document is written in LaTeX using the provided template.

Recommendations and practical points.

i. The LaTeX template is meant to create a somewhat uniform organisation for us, the reviewers. The template includes sections, figures, bibliographical references and so on. You are free to add elements to, or remove elements from, it. Sections may also be renamed. The only thing we want to be respected is the global outline.

ii. The scripts create a somewhat logical organisation of the program, which increases understanding for us and you as well. Allowing us to understand and answer your questions quicker. Every script contains information on what you are allowed to change. You may add more scripts if you feel it is needed, but you may not remove any scripts entirely.

iii. Even in this unconventional time, we recommend pair programming. You may use Microsoft Teams group chat to find a partner. You can also send us an email to a.rodriguezotero@uu.nl or j.hu2@uu.nl and we will try to pair you with someone else.

iv. You can download Python as a standalone program, but we recommend Anaconda, which serves as an easy package and environment manager for Python.

v. For programming in Python we recommend an IDE like Spyder (included in Anaconda and similar to RStudio), PyCharm, or Visual Studio Code. These allow for programming in a more object oriented style compared to Jupyter Notebooks.

vi. Working together in LaTeX documents can be done with Overleaf. In the free version you can work on one document with 2 people, which is enough for this assignment. To get the template in Overleaf, select New Project, Upload Project, and upload the study document zip file.

vii. If you decide to use Git, be sure to set your repository to private. We recommend using `https://git.science.uu.nl` as it is hosted by the university.

Some useful links for Python:

- Getting started with Anaconda.
- Spyder tutorial.
- Pycharm quick start guide.
- VS Code quick start guide.
- Python tutorial, it really starts at the basics, so don't be afraid to skip parts.
- Python Classes and Objects tutorial.
- Tutorial for the *matplotlib* package.
- Documentation on the **gambit-enummixed** command line tool.

2 Deliverables

- A Python program, containing at least the provided scripts and complete with explanatory comments, which can do the following:
  - Calculate the grand tables on the required matrix suites with the required strategies.
  - Perform the replicator dynamic on the resulting grand tables, with and without your own strategy. Also includes creating a graph of how the replicator dynamic progresses.
  - Find Nash equilibria in the grand tables, with and without your own strategy, using the `gambit-enum-mixed` command line tool from Gambit.
- A `README` file, which explains how to use the program.
- A `NAMES` file, which contains your names and student numbers.
- A study document, made using the provided LaTeX template.

3 The grand table

An obvious way to compare multi-agent learning algorithms is to let them play against each other and keep track of their scores. This is the idea behind the grand table.

Let's start with a simple example. Say there are 3 strategies[1] A, B and C and a single payoff matrix G. Now a tournament is played where every strategy is played against all strategies, including itself. The game they play is a 2-player game on payoff matrix G, with the goal of getting the highest payoff. After each game is played for N rounds, the grand table might look something like this:

	A	B	C	Mean
A	2	*3*	2.5	**2.5**
B	0	2	1.5	1.17
C	2	0.5	1.5	1.33

Each cell of the table is the average payoff of the row strategy. So, the highest payoff of the table, highlighted in cursive, it is the average payoff of strategy A when played against strategy B. **To keep things simple, A is also the row-player of this game. So row/column position in the grand table also reflects row-/column- player in the game.**

As strategy A has the highest average score, highlighted in bold, it seems to be the best strategy to use on this payoff matrix. However we weren't looking for the best strategy for a single matrix, but the best strategy in general. So we add restarts to the calculation.

Each restart a new payoff matrix is provided and new games are played with it. The scores of these games are then added to the existing grand table, by averaging. So, for a set of k payoff matrices $G = \{G_1, ..., G_k\}$, $k - 1$ restarts are needed. The final grand table will contain the average of the scores from each of the restarts.

A step by step of how to calculate the grand table:

i. Starting variables:

- $S = \{S_1, ..., S_j\}$, A set of j strategies.
- $G = \{G_1, ..., G_k\}$, A set of k payoff matrices, also called a **matrix suite**.
- $k - 1$, the number of restarts.
- N, the number of rounds per restart.

ii. Play a game of N rounds on matrix G_1 for each possible pair of strategies, so $j \times j$ games. Keep the average payoffs of the row-player for each game.

[1] By strategies we mean the multi-agent learning algorithms.

iii. Restart and do the same for for payoff matrix G_2.

iv. Continue restarting for the set amount of restarts.

v. Calculate the average of the average payoff for each game with the same strategies and put them in the grand table.

vi. Add an average score for each row of the grand table, to show which strategy performed best.

So, in total $k \times j \times j$ games will be played with each N rounds.

To calculate a single cell of the grand table, let's say A against B, play a game of N rounds on payoff matrix G_1. Save the average payoff, $p_1^{A^B}$ of strategy A. Restart and play a game of N rounds on G_2, again save the average payoff of A, $p_2^{A^B}$. Continue this for all the necessary restarts. The score of A against B will be the average of the all the saved averages, $p^{A^B} = mean(\{p_1^{A^B}, ..., p_k^{A^B}\})$.

The following subsections describe the calculation of the grand table with relation to the Python scripts. Furthermore, they contain the types of matrix suites and strategies you will need to implement.

3.1 Program overview

For the scripts to work you need Python 3.8 with the Numpy and Matplotlib packages. You also need to install Gambit for the **gambit-enummixed** command line tool, see 5 for more installation instructions. If everything is installed correctly, you should be able to run `Main.py` without any errors.

Below is short description of each script.

1. *Main.py*: The access point of the program. If the Python environment has the right packages and the **gambit-enummixed** executable is present, you should be able to run it without issue.

2. *GrandTable.py*: Template for the **GrandTable** class, which will calculate the grand table on a matrix suite for a given list of strategies, number of restarts and number of rounds per restart.

3. *Game.py*: Template for the **Game** class, which will play 2 strategies against each other on a matrix.

4. *Stratgies.py*: Describes the abstract **Strategy** class of which each strategy should be a subclass. The Aselect strategy is already given.

5. *MatrixSuite.py*: Describes the abstract **MatrixSuite** class of which each matrix suite should be a subclass. The Fixed matrix suite is already given.

6. *ReplicatorDynamic.py*: Template for the **ReplicatorDynamic** class, which will perform the discrete replicator dynamic and make a graph out of it.

7. *Nash.py*: Template for 2 methods which together will find the Nash equilibria of the grand table. The one that calls **gambit-enummixed** from the command line is already implemented.

8. *Utils.py*: Contains some useful methods that are more generally applicable.

There is no strict order you have to follow when writing the program. We do recommend first createing everything for the grand table. The replicator dynamic and Nash equilibria need to have a completed grand table as input, so they can be addressed after that. Here are 2 possible ways in which you could begin.

i) **Quick feedback:** Start by implementing the **Game** class, so 2 players with both the Aselect strategy can play against each other. Move on to the **GrandTable** class and calculate the grand table for a list of Aselect strategies on the Fixed matrix suite. Continue by implementing strategies.

ii) **From the ground up:** Start by implementing the matrix suites and then the strategies. Now you can implement games and try them with any combination of strategies. Continue by implementing the grand table and you have everything you need to calculate all the necessary grand tables.

3.2 Defining matrix suites

The **MatrixSuite** class in `MatrixSuite.py` is the abstract representation of a matrix suite. A matrix suite is the collection of payoff matrices that the strategies will play games on. In total there will be 3 matrix suites which means 3 grand tables, as you will have to create a grand table for each matrix suite. Below are the requirements for each matrix suite.

a) Fixed: A predetermined suite of 10 payoff matrices, which is already implemented for you. The matrices can be found in Appendix A.

b) Random Int: This matrix suite should be able to generate matrices of the following type:

1 A randomly determined number of row actions (R) from the interval (1..5].

2 A randomly determined number of column actions (C) from the interval (1..5].

3 A payoff matrix of size $R * C$ containing tuples of random integer values from the interval (0..3].

c) Random Float: This matrix suite should be able to generate matrices of the following type:

1 A randomly determined number of row actions (R) from the interval (1..5].

2 A randomly determined number of column actions (C) from the interval (1..5].

3 A payoff matrix of size $R * C$ containing tuples of random floating point values from the interval $(0.0, 3.0]$.

Hint: The Fixed matrix suite has a limited number of matrices, so it can only handle 10 restarts. The other matrix suites can generate an arbitrary amount. If you play all games before restarting, you don't have to keep track of an index or save any matrices. You can just call `generate_new_payoff_matrix()` with each restart.

3.3 Defining strategies

The **Strategy** class in `Strategies.py` gives an abstract representation on how to implement a strategy. Each strategy should minimally implement the methods that are prefaced by `@abs.abstractmethod`. In short the `initialize` method should set the starting state of the strategy, the `get_action` method should return the action that the strategy will play at that round and the `update` method should update the internal state of the strategy with the result of that round. As an example we have implemented the Aselect strategy for you. Below is the list of strategies and their input parameters, which will need to be implemented:

- Aselect, this is random play, but may not be named as such because `random` is a reserved word in Python.
- ϵ-Greedy, with $\epsilon = 0.1$.
- Upper confidence bounds (a.k.a. UCB or UCB1).
- Satisficing play, with $\lambda = 0.1$
- Softmax, with $\gamma = 1.0$, $\lambda = 0.1$.
- Fictitious play (in its pure form, so no smoothed FP).
- Bully.
- Proportional regret matching.

Hint: The starting value for geometric averages is not 0, but rather depend on the payoff matrix of the player. It should be $(max + min)/2$ of all the payoffs a player can get.

Hint: If you want to check if you strategy works, you can simulate a single round of play by calling the methods of the strategy. An example of this is included in `Main.py`.

3.3.1 Implement your own strategy

Aside from the mandatory strategies, you also have to come up with your own strategy. It can be anything, as long as it implements the **Strategy** class, so get creative! An explanation of how your strategy works has to be included in the study document, so we can also understand it.

3.4 Playing a single game

The **Game** class in `Game.py` is a single 2-player game. In it 2 strategies should play against each other for N rounds.

Hint: Start by making a method that can play a single round and then create another method that just plays a single round N times.

Hint: You can implement restarts within the Game class, reinitialize the Game class to simulate a restarts or create a new Game instance for each restart.

3.5 Putting everything together

The **GrandTable** class in `GrandTable.py` should calculate the entire grand table for a given set of strategies, matrix suite, number of restarts and number of rounds per restarts. Below is a table with the required parameters for each matrix suite.

matrix suite	Strategies	Restarts	Rounds per restart
Fixed	All	9 (10 matrices)	1000
Random Int	All	19 (20 matrices)	1000
Random Float	All	19 (20 matrices)	1000

As all strategy pairs will need to play a game on the same matrix, it is easier to play all games and then restart, compared to calculating the final score for each pair sequentially.

To get a feel for the correctness of your grand table, we have included the grand table we got on the Fixed matrix suite in appendix B. Your solution can vary due to the random nature of some strategies, so don't worry if it is not exactly the same.

4 The replicator dynamic

The grand table is relatively simple way to compare strategies. A different way is the replicator dynamic, which takes an evolutionary approach.

Let's say there exist a population of strategies, which have a certain proportion. For strategies A, B and C this could be $20\%A$, $30\%B$ and $50\%C$, so $(0.2, 0.3, 0.5)$. This proportion would evolve with regards to strategy strength, with stronger strategies gaining share and weaker strategies losing share. The discrete replicator dynamic can simulate this. In `ReplicatorDynamic.py` there is a method which you will need to complete to perform the discrete replicator dynamic.

Start with the grand table M (without row means) and an initial proportions vector $p = (p_1, ..., p_j)$, $0 \leq p_i \leq 1$ and $\sum_i p_i = 1$. Now calculate the expected score per strategy:

$$s = Mp$$

The proportions can then be updated proportionally to their expected score. This is done with the $\circ$ matrix operator, which is the Hadamard product.

$$q = p \circ s \; [\; (q_1, ..., q_j) = (p_1 * s_1, ..., p_j * s_j)]$$

These new proportions do not yet sum to one, but we can address that with normalisation to the $n-1$-simplex.

$$p = N(q) = \frac{q}{\sum_i q_i}$$

So the complete equation is the following.

$$p_{new} = N(p_{old} \circ (Mp_{old}))$$

It is repeated until the proportions stay stable. This process is the discrete replicator dynamic with $\beta = 0$

Hint: Make methods for the individual steps of the calculation, e.g. matrix multiplication and the Hadamard product.

Hint: You will need a method that checks for changes between p_{new} and p_{old}, so you can decide when it is stable.

You will also need to make a method that graphs how the proportions evolve. The `matplotlib` Python package could help with automating this or you could manually do it by printing and copying the values into Excel.

You will need to perform the replicator dynamic on the 3 grand tables, with and without your own strategy, for uniform and non-uniform starting proportions. This is important because different starting proportions can drastically influence the results. Furthermore, with this setup you can also compare the impact your own strategy has on the results. Below is a table with the required starting proportions for each combination, these proportions are also included as variables in `ReplicatorDynamic.py`.

matrix suite	Include own strategy?	Uniform?	Starting proportions
Fixed	Yes	Yes	1/9 for every strategy
Fixed	Yes	No	[0.12, 0.08, 0.06, 0.15, 0.05, 0.21, 0.06, 0.09, 0.18]
Fixed	No	Yes	1/8 for every strategy
Fixed	No	No	[0.22, 0.19, 0.04, 0.06, 0.13, 0.10, 0.05, 0.21]
Random Int	Yes	Yes	1/9 for every strategy
Random Int	Yes	No	[0.12, 0.08, 0.06, 0.15, 0.05, 0.21, 0.06, 0.09, 0.18]
Random Int	No	Yes	1/8 for every strategy
Random Int	No	No	[0.22, 0.19, 0.04, 0.06, 0.13, 0.10, 0.05, 0.21]
Random Float	Yes	Yes	1/9 for every strategy
Random Float	Yes	No	[0.12, 0.08, 0.06, 0.15, 0.05, 0.21, 0.06, 0.09, 0.18]
Random Float	No	Yes	1/8 for every strategy
Random Float	No	No	[0.22, 0.19, 0.04, 0.06, 0.13, 0.10, 0.05, 0.21]

5 Nash equilibria

The last method of comparing strategies is to find the Nash equilibria of the grand table. The entire grand table, without row means, could be considered a matrix game between 2 players, with actions being the strategies. So Nash equilibria would describe the optimal action profiles of these players.

Gambit is a tool that can calculate Nash equilibria automatically. In particular the command-line tool **gambit-enummixed**. In `Nash.py` most of the work is already done. There are only a few things left to do.

1. Install Gambit 15 and copy the **gambit-enummixed** executable from the installation folder to the folder containing the `Nash.py` script.
2. `Main.py` includes some code to check if Gambit works properly and `Nash.py` includes some examples for running Gambit from the command line, for some checks outside of Python.
3. Complete the `nash_equilibria` method so it provides the correct information to `run_gambit`.
4. Calculate the Nash equilibria for each grand table, with and without your own strategy, so you can compare how it impacts the results.

The `run_gambit` method prints the found equilibria as action profiles. Actions profiles are similar to proportions, so you might be able to compare them in your study document.

NOTE: `run_gambit` has only been tested on Windows systems, so there is a chance it doesn't to work on Mac or Linux. If this is the case, let us know.

A Fixed matrix suite

1. 2x2:
(2, 2) (6, 0)
(0, 6) (4, 4)

2. 2x2:
(9, 2) (2, 8)
(8, 0) (1, 7)

3. 2x2:
(1, 8) (1, 1)
(2, 1) (2, 9)

4. 3x3:
(1, 8) (9, 0) (6, 3)
(9, 0) (0, 9) (0, 9)
(9, 0) (2, 7) (9, 0)

5. 3x3:
(3, 3) (4, 4) (9, 9)
(2, 2) (0, 0) (6, 6)
(1, 1) (5, 5) (8, 8)

6. 3x3:
(9, 1) (0, 2) (10, 1)
(8, 2) (8, 2) (8, 0)
(0, 1) (1, 1) (1, 9)

7. 3x3:
(2, 8) (1, 8) (7, 1)
(7, 0) (1, 7) (8, 2)
(7, 2) (7, 2) (8, 0)

8. 3x3:
(2, 2) (4, 1) (6, 0)
(1, 4) (3, 3) (5, 2)
(0, 6) (2, 5) (4, 4)

9. 4x4:
(5, 9) (0, 10) (9, 6) (3, 2)
(7, 7) (1, 1) (4, 1) (1, 7)
(3, 0) (4, 0) (9, 3) (5, 9)
(2, 1) (2, 7) (0, 10) (0, 9)

10. 4x4:
(10, 0) (4, 6) (5, 5) (8, 2)
(8, 2) (5, 5) (6, 4) (10, 0)
(5, 5) (8, 2) (0, 10) (8, 2)
(1, 9) (4, 6) (7, 3) (6, 4)

B Grand table example

Calculated on Fixed matrix suite, with 9 restarts and 1000 rounds per restart.

```
==================================================================================
       ||Aselect|EGreedy|  UCB  |Satisfi|Softmax|Fictiti| Bully |Proport|| MEAN  |
Aselect|| 4.12  | 3.32  | 3.31  | 3.90  | 3.45  | 3.23  | 2.87  | 3.20  || 3.42  |
EGreedy|| 5.35  | 5.00  | 4.81  | 4.50  | 4.78  | 4.74  | 4.83  | 4.50  || 4.81  |
    UCB|| 5.49  | 4.61  | 4.78  | 4.87  | 4.74  | 4.76  | 4.92  | 4.51  || 4.84  |
Satisfi|| 4.66  | 4.53  | 4.31  | 4.94  | 4.20  | 4.27  | 4.30  | 3.72  || 4.37  |
Softmax|| 5.21  | 4.84  | 4.92  | 5.78  | 4.71  | 4.90  | 5.05  | 4.57  || 5.00  |
Fictiti|| 5.55  | 4.81  | 4.87  | 5.80  | 4.85  | 4.72  | 5.20  | 4.39  || 5.02  |
  Bully|| 5.56  | 4.57  | 4.48  | 5.40  | 4.58  | 4.40  | 4.50  | 4.40  || 4.74  |
Proport|| 5.55  | 5.06  | 5.01  | 5.92  | 4.92  | 5.09  | 5.30  | 4.76  || 5.20  |
==================================================================================
```

End.