Faculté des sciences économiques et de gestion
Université de Strasbourg

University year 2023/2024

Reinforcement Learning Project : Stackelberg Competition

Vincent KIRCHEN
Lothaire LEMARQUIS

Table of contents

1. Introduction

The Stackelberg duopoly is a model in game theory and microeconomics that describes a situation where two companies interact in a competitive market. It takes its name from the German mathematician and economist Heinrich von Stackelberg, who developed the model in 1934.

In a Stackelberg duopoly, the two companies are leaders and followers in terms of their strategic decisions. The model assumes that one company makes decisions before the other, giving it a strategic advantage. Thus, there is a leader and a follower.

Here's a brief introduction to the key concepts of the Stackelberg model.
The leader is the company that makes the decision first. He anticipates the follower's reaction and makes his decision accordingly. The follower takes his decision after the leader and makes his decision based on what the leader has chosen.
Companies make decisions based on variables such as production, price or quantity, with the aim of maximizing profit.
The key idea is that the leader, by making his decision first, takes into account an anticipated decision by the follower, which gives him a strategic advantage, but does not necessarily lead to greater profit, as this may depend on the cost function of the two firms, for example.

The decisions of both companies can be modeled mathematically using reaction functions. The key variables are production quantities, fixed prices and production costs.

In this paper, we present our reinforcement learning model based on Stackelberg's duopoly model. We will first present the structure of our model and how agents will learn their best strategies using reinforcement learning methods, then we will present the extension of our model and the implementation of our model.

2. Basic model

Our project is to train two agents to maximize their profit on the Stackelberg duopoly model thanks to reinforcement learning techniques.

In our model, there is a "leader" agent and a "follower" agent. The two agents compete on quantity. The "leader" agent chooses his quantity first and tries to anticipate what the "follower" will choose as quantity, imagining that his goal is to maximize his profit. The follower plays after the leader and knows the quantity he has chosen. The follower therefore plays according to the leader's choice.
We note Q_leader the quantity chosen by the leader and Q_follower the quantity chosen by the follower.

The leader's profit maximization function is:
(99 - (Q leader + Q follower)) * Q leader - (4* (Q leader * Q leader) + 16)

(99 - (Q leader + Q follower)) corresponds to the price and **(4* (Q leader * Q leader) + 16)** corresponds to the cost function.

The follower's profit maximization function is :
(99 - (Q_leader + Q_follower)) * Q_follower - ((Q_follower * Q_follower) + 9).

(99 - (Q leader + Q follower)) corresponds to the price and **((Q_follower * Q_follower) + 9)** corresponds to the cost function.

The leader, by choosing his quantity first, takes more risk, as he doesn't know what the other agent is playing, whereas the follower knows what the other agent has played. This is represented here by a higher cost function ((4 * (Q_leader*Q_leader) + 16) for the leader versus ((Q_follower * Q_follower) + 9) for the follower.

The leader plays first and therefore has only one possible state: 0.
The state in which the follower finds himself, as he plays after the leader and will play according to the quantity he has chosen, corresponds to the quantity chosen by the leader.
To choose the quantity that will maximize his profit, the follower will derive his profit function by the quantity he has chosen, looking for the quantity for which the value of the derived function is equal to 0. Deriving by the quantity chosen by the follower, this gives: Equilibrium Q_follower = **(99 - Q_leader) / 4**

To enable the leader to take into account the action chosen by the follower, we integrate the follower's equilibrium quantity into the leader's profit function, giving :

(99 - (Q_leader + Q_balance follower)) * Q_leader - (4* (Q_leader * Q_leader) + 16), which is equal to :
99 * Q_leader - (Q_leader * Q_leader) - (Equilibrium Q_follower * Q_leader) - 4 * (Q_leader * Q_leader) - 16

We derive this formula by the leader's quantity (Q_leader), looking for the quantity for which the derivative formula is equal to 0 to find the quantity that maximizes his profit. We find equilibrium Q_leader = 7.61.
On intègre cette valeur pour trouver la quantité qui maximise le profit du follower, ce qui donne : (99 - 7,61) / 4 = 22,85.

As the leader agent has an equilibrium quantity of 7.61 , he will be able to play all integers between 0 and 10 in our model.
As the follower has an equilibrium quantity of 22.85, solving by mathematics, he will be able to play all integers between 0 and 30 in our model.

In this case, the follower produces a significantly higher quantity at equilibrium than the leader. This is due to the cost function chosen: for example, when both firms have a cost function ((4 * (Q_leader*Q_leader) + 16), the leader produces slightly more in equilibrium than the follower (9.09 vs. 8.99).

We used Q-Learning, a reinforcement learning technique based on the iterative updating of the Q-function, which measures the quality of an action in a certain state. Q-Learning is used to find an optimal set of actions from an initial state to maximize the expected value of the total reward over all successive stages.

Our Q-Learning model will enable us to build a neural network that will estimate the Q-values for each possible action and allow the agent (leader or follower) to choose the action with the highest Q-value according to the state in which it finds itself.

We use an epsilon-greedy policy :

With probabilty ε, the agent chooses a random action, exploring new actions that are potentially better than those already known. This allows the agent to discover alternative strategies and avoid getting stuck in sub-optimal behaviors.

With probability 1 - ε, the agent chooses the action with the highest Q value from those already known. This allows the agent to exploit current knowledge to make decisions that appear optimal based on past experience.

The total rewards of each episode for the leader and follower are recorded in a replay buffer, and are used repeatedly to improve learning stability and encourage generalization. This also helps prevent the agent from becoming too specialized in specific, recently observed behaviors.
We will now present an extension of the game.

3. Extension

The basic model of our Stackelberg competition consists of a one stage game as explained previously. To add a layer of complexity to this model, an extension can be applied to transform it into a two stage game. As such, the first stage of the game represents an investment phase where both agents can choose an amount they would like to spend to prepare their entrance on the market. The agent making the higher investment will speed up his readiness to enter the market and will be the leader, and the agent making the lesser investment will be considered the follower. This enables a trade-off for the agents between being able to be the first one to choose the quantity provided on the market and thereby potentially take advantage of the supposed gain of the leader, but on the downside observe higher costs, or invest less and thereby have higher chances to be the follower and having to adapt in terms of what the leader played but on the upside having to deal with lower costs. Consequently, the agent's aim is still to achieve the highest possible reward but by making two choices, one in each stage of the game.
The profit function is the same as the one used in the basic model with the exception of the costs not being arbitrary but being influenced by the investment decision in the first stage of the game. This is done in the purpose of having a better understanding of the added layer on the effective results of the model.

4. Implementation

In this segment, we are going to take a closer look at the code and its structure. First of all, let’s break down the different classes and methods implemented.
The first class that is being defined is the ‘class StackelbergEnvironment’ , which sets up the environment in which the agents will interact. Its attributes are:

- state_dim: The dimensionality of the state space
- leader_action_space: The space of all possible leader actions
- follower_action_space: The space of all possible follower actions
- state: Represents the current state of the environment
- time_step: Counts the number of time steps

Once the environment is initiated, we pursue to define three complementing methods. The first method is ‘reset()’ which resets or initializes the environment to an initial state. Furthermore we have the ‘calculate_rewards(leader_action, follower_action)’ method which computes rewards based on leader and follower actions. Finally, the ‘step(leader_action, follower_action)’ updates the environment based on actions and returns the next state, but also the rewards and whether or not the episode is done.

The second class we defined is the ‘class Jointagents’ which sets up the agents’ behaviors. The classes’ attributes are:

- state_dim, leader_action_space, follower_action_space: The dimensionality and action spaces
- gamma: Represents the discount factor.
- epsilon, epsilon_min, epsilon_decay: These are the exploration-exploitation parameters
- learning_rate: Stands for the learning rate for the Q-network.
- model: The neural network.
- replay_buffer: The replay buffer necessary for storing the agents experiences.

There are multiple methods in this class. The first being the ‘build_model()’ method which builds the Q-network model. Then we have the ‘choose_actions(state)’ method for implementing the epsilon-greedy policy in order to choose the agents’ actions. The ‘remember(state, leader_action, follower_action, leader_reward, follower_reward, next_state, done)’ method stores both agents’ experiences in the replay buffer. At last, the ‘replay(batch_size)’ method performs the experience replays to train the Q-network.

Now let’s dive into the simulation process. First of all the environment is initialized with all the appropriate parameters, such as the agents’ range of actions or the number of episodes played for example. Upon completion of this, the training loop orders what happens in every episode. More precisely, for every time step in an episode, the actions are chosen using the epsilon-greedy policy. Then the actions are performed and the next state and rewards are observed. Once this is done, the agents' experiences are stored in the replay buffer. The state is then updated for the next iteration, the agents' experiences replays are performed and train the Q-network. Finally, if the episode is done, the loop is broken.

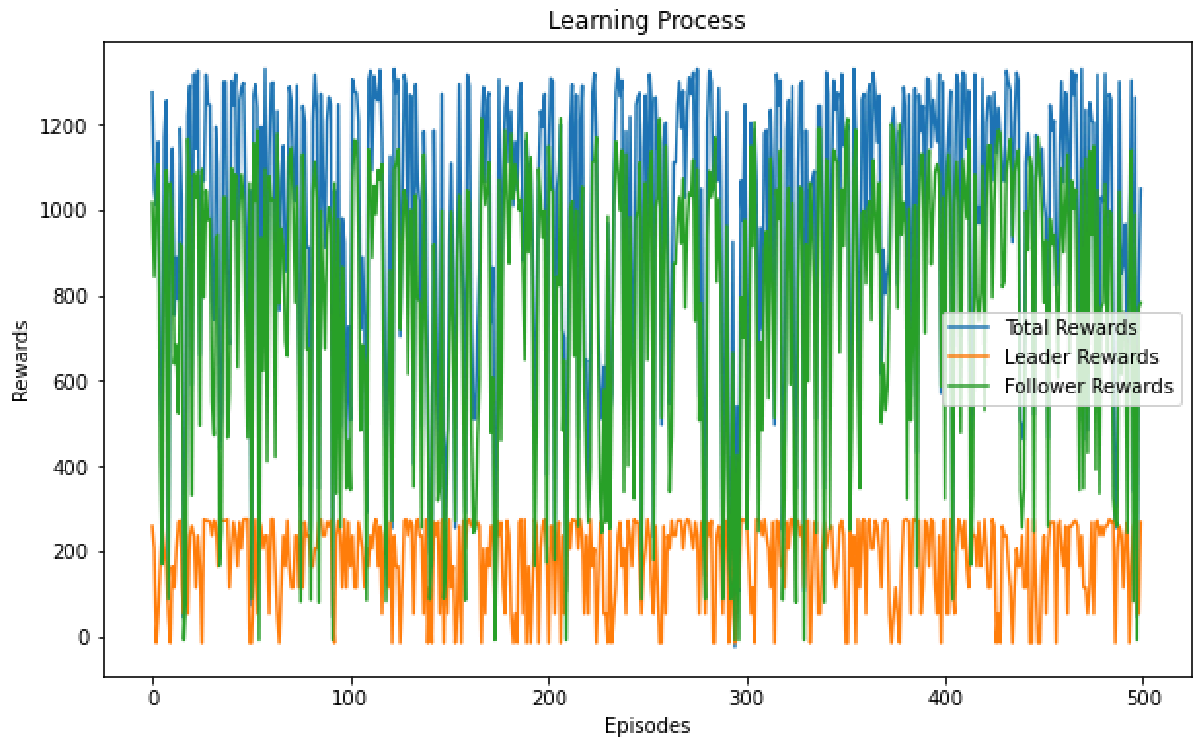
Concerning the Reinforcement Learning Algorithm used, the code implements a Deep Q-Network (DQN). The experience replay is used to store and sample experiences for training stability, and the Q-network is trained using the mean squared error loss and an Adam optimizer.

To go more into detail on the Neural Network Structure, the neural network model applied here is one from Keras library called 'Sequential' which has three dense layers. The input and second hidden layer both have 32 units with ReLu activation. Concerning the output layer, its units are equal to the product of the leader and follower action space sizes.

Finally, taking a closer look at the present hyperparameters, the model uses a discount factor for future rewards 'gamma', as well as parameters for the epsilon-greedy exploration as mentioned above ('epsilon', 'epsionl_min', 'epsilon_decay'). The 'learning_rate' parameter is self-explanatory and represents the learning rate for the Q-network and the 'batch_size' represents the size of the experience replay buffer for training.

5. Analysis

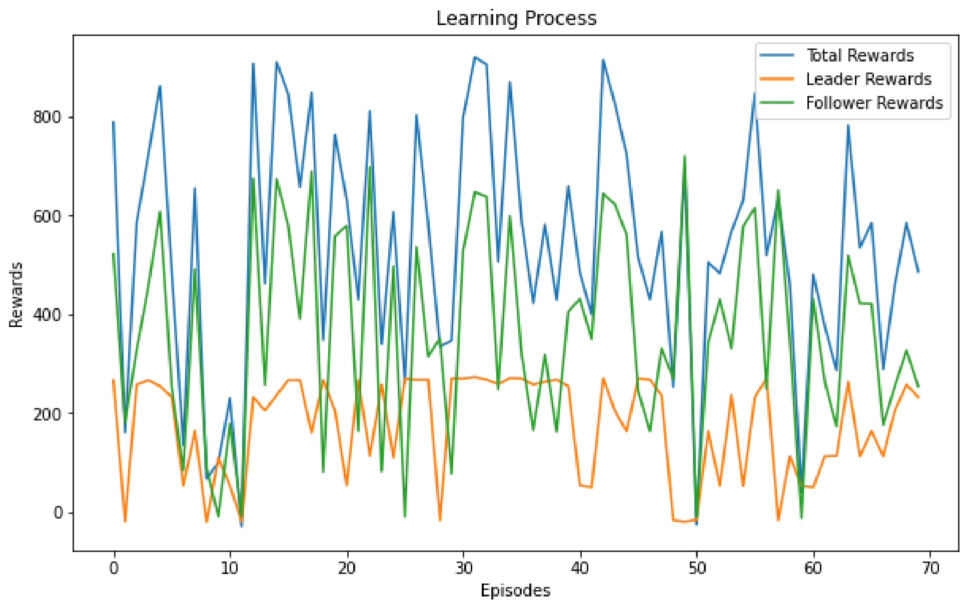
Presented below is a graph depicting the progression of follower, leader, and total rewards—reflecting their respective profits—across 500 episodes in our foundational model.



In our model, the follower consistently outperforms the leader in terms of profits, primarily attributable to a significantly higher cost function for the leader. This diverges from the typical pattern observed in the Stackelberg model, where the leader generally attains higher profits, contingent upon the specific cost function employed. Illustrated in the graph, the leader often achieves optimal profits, whereas the follower's rewards exhibit a less consistent pattern.

Despite running 500 episodes, discernible improvements are challenging to identify. While the leader experiences slightly fewer instances of low profits compared to the initial stages, the issue persists. A more pronounced progression may become apparent with a substantially larger number of episodes, allowing for a clearer depiction of the model's evolution.

In the Graph below, we can observe the same game, but now iterating over two stages and taking into account the market entry investment.



We can observe that the results are similar to the one stage game in terms of the followers gain being superior to the leaders profit. We could conclude that this extension is not enough to accurately represent the competition between the two agents, but we will get more into detail about this in the limits segment.

6. Limitations and Conclusion

There are several improvements or add-ons that could be brought to this model to make it more complex and complete. Firstly, enhancing the environment to take into account market conditions and the dynamic changes that they bring along would be an option to get results closer to reality. Another idea would be to introduce several follower agents with slightly different strategies to observe which one is more successful. Furthermore, with multiple follower agents, there could be room to try and add a notion of corporation between the agents. This could lead them to outperform their results if they were to be the only follower agent or being alongside other follower agents acting only in self-interest.
All in all, applying reinforcement learning techniques to this game can be done in many ways and can be fine-tuned to take in more or less ‘real-world’ information. In our specific application, the results don’t show clear signs of improvement from the agents, which is the main limit to our model and these could be improved by fine tuning the model, applying more relevant hyperparameters or letting the model train for a larger number of episodes.

While the model didn't yield the anticipated results, employing a widely recognized economic model via reinforcement learning revealed the extensive versatility of this algorithm, extending its applications beyond the scenarios we were already familiar with and this experience deepened our appreciation for its capabilities.

Sources

We explored the concept of Stackelberg duopoly by leveraging insights from a third-year university economics course covering Cournot and Stackelberg duopolies. The course provided a relevant profit function that we incorporated into our study. In the implementation phase, we harnessed the assistance of ChatGPT to refine our understanding. This involved adapting the profit functions, fine-tuning parameters for initiating the Q-Learning model, and troubleshooting code errors.