Università degli studi di Milano-Bicocca

Decision Models

Final Project

Methodological approaches for Multi-agent RL games

Authors:
Riccardo Cervero - 794126
Federico Moiraghi - 799735
Pranav Kasela - 846965

September 3, 2019

Abstract

The purpose of this project is to create agents able to simulate human behaviors and tendencies when subjected to incentives within different environments. In 4 games, relationships will be established between the agents, fostered by positive or negative signals, and natural factors such as the health and fear level will be recreated. More precisely, in the first experiment three different methods will be used to allow a single Reinforcement Learning system to beat a non-intelligent rival. In the next two, the players will be pushed to face each other in a fight, configured first as a duel and then as a battle - among five of them. Finally, two of the same agents will be encouraged to work together to eliminate a common enemy.
Each game is accompanied by a graphic component that shows the actual progress of the game.

Contents

1 Introduction **2**

2 Game 1: Bilbo and the dragon **2**
2.1 Q-Learning algorithm 4
2.2 Deep Q-Learning approach . . 5
2.3 Genetic Algorithm applied to Deep Q-Learning algorithm . 6

3 Multi-agent Competition **7**
3.1 Game 2: Duel between two agents 8
3.2 Game 3: Battle among five agents 9

4 Multi-agent Cooperation **11**

5 Conclusions **11**

1 Introduction

Reinforcement Learning is the branch of Machine Learning gathering several techniques designed to create a system, an agent, capable of learning the best way to achieve goals within an environment, whose elements - such as, for example, obstacles or objectives - can also vary over time. In other words, this agent simulate the process of human understanding, by exploring the initially unknown environment and consequently receiving rewards or penalization based on the goodness of the choices made.
In this study, we especially set ourselves the goals of:

1. Using different methodologies to perform the same Reinforcement Learning experiment (Game 1);

2. Deepening the dynamics among multiple separate agent, each of which will learn automatically and independently how to win the game by looking and reacting to other agents' choices. In particular, the two dynamics of "competition" and "co-operation" between agents will be examined.

To do so, four "worlds" have been built, with different rules and elements - such as fear or randomly moving obstacles -, within the agents will be trained over several epochs to maximize the reward, beat the rivals or help the allies in the best way possible.
The following sections will present games' and agents' characteristics.

2 Game 1: Bilbo and the dragon

This first experiment has been created with the aim of employing Q-Learning algorithm, Deep Q-Learning and Deep Q-Learning with the integration of Genetic Algorithm on a single system whose objective is beating a non-intelligent rival.
The environment the three agents will be trained within is defined as a class named "World" in the Python programming language, by which it is possible to create the elements they interacts with during the learning process. First of all, it is initialized a Numpy array with dimensione 15x15 as an empty grid, to which various elements are added, so as to compose the world the agent explore:

- Obstacles, whose presence in the grid is denoted by the special character '▮', and which can be situated randomly - if no "entrance" location is specified-, or in a precise position over the epochs;

- A treasure (♕), which can be located in a randomly chosen coordinate or predefined too. It can not be moved, but only caught by the agent;

- A dragon (☠), which is the non-intelligent enemy of the agent. It can be spawn randomly, or, in case the position of treasure was preselected, situated close to it, in order to make the game more difficult. In this experiment, the dragon is not capable to learn any strategy to win the game over the epochs or react to agent's moves, but it merely moves randomly through the grid following the four directions 'up', 'down', 'right', 'left';

- *Bilbo* (☺), the agent to be trained, which is always generated at a random position. Unlike the other components, Bilbo moves around the world trying to maximize the total reward of his game and avoid penalties in best way possible, which is the core operating principle of any Reinforcement Learning methodology.

The creation of these components is carried out by the function *"random_spawn"* at locations where there are no obstacles or other characters, so that the grid remains consistent, while the function *"is_border"* checks whether a cell at a given coordinate is borderline.
Therefore, Bilbo's goal is to reach the cell where the treasure is without getting caught by the dragon, condition that occurs when the two end up in the same cell. The game ends when Bilbo is killed by the dragon in the terms just described, or when he manages to get the treasure, escaping from the enemy among the obstacles scattered around the world. In the meanwhile, another function returns the reward based on every game state:

- If the treasure has been caught by Bilbo, he gets a positive reward of 10;

- If Bilbo has been killed by the dragon, he gets a penalty of -5;

- At each moves, Bilbo receives a penalty of -1. This negative reward serves to incentivize Bilbo to find shorter routes to reach the treasure;

- If Bilbo dumps into an obstacle, so his position remains the same, he gets a penalty of -1, so that he learns to avoid them.

Reward values can be a negative constant or 0, but in order to obtain a faster convergence, it's better to give the agent a positive reward when he gets near the objective.
Another Python class named "Agent" serves to set the main characteristics of Bilbo:

- He can move in the same four direction of the dragon - 'up', 'down', 'right', 'left';

- He is affected by a "fear" factor, in order to simulate an aspect of real human exploring process. This is produced by a function which returns a value based on the distance from the dragon: if Bilbo is far from it, the "fear" value is close to 1; otherwise, as the distance from the dragon decrease, the value increases at most to 2. This measure is calculated as follows:
$$\frac{d}{1+d}$$
with d representing the Euclidean distance between the dragon's position (D) and Bilbo's location (B):
$$d = \sqrt{(D_x - B_x)^2 + (D_y - B_y)^2}$$
This will consist in a divisor of the ϵ parameter used in Reinforcement Learning

algorithms to produce random moves in the grid and allow a better exploration of the world. As one can see from the formulation, this factor is a normalized distance, used to prevent ϵ to be too much affected by it.
In conclusion, in this way, when Bilbo approaches the dragon, his fear level increases and the chance of a random exploration as well.

Given this general aspects of Game 1, three methodologies have been exploited to provide the agent with a learning process, each based on an agent generalization provided by a new Python class.

2.1 Q-Learning algorithm

The first approach consists in Q-Learning algorithm, which is based on the propagation of potential reward from the best possible actions in future states. Following this idea, each action in each state is related to a Q value, updated when new relevant information about the environment, derived from exploration, is made available. The Q-Learning algorithm utilizes the following updating rule:

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma\ max\ Q(s',a') - Q(s,a))$$

with $Q(s,a)$ Q value of current state and $Q(s',a')$ Q value of next state; r the reward obtained by taking action a at the state s; γ discounting factor regulating the impact of future rewards; α learning rate. Hence, the worth of an action in a certain state also depends on the discounted worth produced by the best action in the next state s', and on the extent of the steps to the convergence we selected with parameter *alpha*. In other words, this policy makes agent estimate the goodness of an action in a certain state based on the cascaded, discounted reward from the next states. In this way, at state s, Bilbo won't choose the action providing the best rewards only in the current state, but the action which leads to the best total rewards over all the states, while exploring the world in order to discover the rewards or penalties of all action. As Q-values are updated, they are stored in a Q matrix, so that Bilbo can learn from past experiences. This exploration process, that consists merely in random moves around the "gridworld", is fostered by ϵ parameter, which depends on "fear factor" as aforementioned. More over, ϵ decays by being multiplied to a "decay factor" at each step, up to a minimum of 0.01.
In conclusion, this process can be explained as follows: if a number extracted from a uniform between 0 and 1 is less than ϵ or the Q values of all possible actions in the current state are null, the system performs a random step. Otherwise, it performs the action related to the maximum Q value in that state. The values set for hyperparameters are the following:

- $\alpha = 0.5$, so that Bilbo's "patience level" is perfectly balanced
- $\gamma = 0.8$, making Bilbo a fairly far-sighted agent
- $\epsilon = 0.2$

- *decay factor* $= 0.99$, yielding a slow decay

Then, the limits of epochs and episodes executable are both set to 1000.
During the iterations, the performances of Bilbo are tracked by counting the number of wins, cases where he manages to stay alive and get the treasure, and losses, when he ends up killed by the dragon.

2.2 Deep Q-Learning approach

In the previous methodology, Q-Learning algorithm, the experience acquired by Bilbo during the games was based on explicit tables, - Q matrix and reward matrix - holding the information discovered, so that he could memorize which actions to choose in any given state. This tables could end up to be far too large and unwieldy when the game is composed of a large number of states and potential actions, or the world becomes larger. Similarly, still maintaining a medium or small gridworld, this mechanism leads to a loss of efficiency in the learning process, slowing down the progress of the system - namely the algorithm - which constitutes the agent.
The solution to this problem can be the application of a Deep Reinforcement Learning methodology, which consists in training a neural network to predict Q values for each action in a certain state, instead of having explicit tables.
The input of this neural network, implemented within Q-learning algorithm, is not only the simple set of coordinate describing the location of the agent itself and of the dragon, which is the current state, but a more general from, so that the model will be able to generalize to different scenarios. This reshaped form of current state is produced in two steps, by:

1. computing the distance between Bilbo and the treasure and between Bilbo and the dragon by subtracting respective coordinates;

2. checking whether the position of Bilbo is at the boarder for all four sides - ordered as "up", "down", "right", "left" - and assigning 0 if the given direction is accessible or 1 if it is borderline.

After this transformation, the current state that will form the input of the neural network will appear as an array containing the two distances and the aforementioned binary values. For example, if Bilbo were on the leftmost edge of the gridworld, the input neurons would receive the following elements:

$$\{B_{(x,y)} - T_{(x,y)},\ B_{(x,y)} - T_{(x,y)},\ 0,\ 0,\ 0,\ 1\}$$

The output are the Q values for each action in the given state, which will approach the results produced by the learning update previously utilized. Therefore, using the mean-squared error metrics, the loss or cost function for the neural network will be:

$$(r + \gamma\ max_{a'}\ Q'(s', a') - Q(s, a))^2$$

Starting from this, the neural network is built as a sequential model composed by:

- the input layer, receiving the reshaped current state, namely the information about the current distance from the other entities and with respect to the border;

- a hidden layer composed by 16 neurons activated by a Rectified Linear Unit (ReLU) function;
- the linear activated output layer with one neuron for each possible action in the given state, which hence performs the linear summation of the inputs and the weights, with no additional function applied, providing the estimated Q values.

The model is compiled using the aforementioned mean-squared error loss function and the Adaptive Moment Estimation as optimizer.
As Bilbo explores the world, his experience is described into 5 values: current and next state, game state, reward and action performed. Since, among all the moves performed, the probability that one of them leads to Bilbo's victory - attainment the treasure - or to his defeat - capture by the dragon - is very low, two separate memories are established: one containing the unlikely events just defined, the other storing the rewards obtained in the normal phases of the game. Therefore, when the system has developed sufficient experience in the world, in order to calculate the exact Q-values for each action in the given state, at each epoch the model is trained on a *mini-batch* consisting of the union of 16 elements randomly extracted respectively from both separate memories. While the system performs these steps, it simultaneously updates the Q-values. Finally, this process is exploited within the previously used Q-Learning algorithm, extending the classic updating rule and resulting in a highly efficient Deep Reinforcement Learning methodology.
As before, performances, computed by the number of win and losses, are tracked over the 20000 episodes, which involve at most 150 epochs.
The hyperparameters are set as follows:

- $\gamma = 0.8$, maintaining the same Bilbo's "foresight" as before
- $\epsilon = 0.5$, increasing the fixed component of random exploration probability with respect to the previous section
- $decay\ factor = 0.9998$, further slowing the decay

2.3 Genetic Algorithm applied to Deep Q-Learning algorithm

An issue that needs to be further investigated to achieve accurate estimation performances through a neural network is the tuning of learning parameters involved in the training process. In the previous section, the optimization technique used for updating the weights of the sequential model was the Adaptive Moment Estimation. This one consists in an extension of Stochastic gradient descent method (SGD), considered as standard *de facto* for training artificial neural networks. However, despite its properties, it can

be replaced by other methodologies, including the class of metaheuristics called evolutionary algorithms (EA). The main difference between SGD and EA lies in the fact that the second ones are well suited for multi-criteria optimization, when gradient descent is dedicated to mono-criteria optimization.
A solution part of the EA class is represented by the Genetic Algorithm, which simulates the evolution of an initial population towards the best possible solution, the typical mechanism of the natural selection. During this process, each candidate solution present in the initial population of randomly generated individuals has a set of properties, chromosomes or genotype, which can be mutated and altered. Therefore, in this third subsection, Genetic Algorithm will be presented as an alternative, search-based, optimization method for learning weights within the Deep Reinforcement Learning system.
The operation mode of this third algorithm is completely identical to the previous one, except for the use of GA within the neural network training process. In details, the weights optimization process takes place as follows: the system initializes 100 random sets of weights, Bilbo's learning parameters, and evaluates the fitness of every individual within each generation - actually the value of the objective function in the optimization problem being solved, hence trying to minimize the mean-squared error function for Q-values -, returning the individuals sorted according to their score. From them, it extracts the first 10 - elitism parameter -, which will be used as parents of next generation. Indeed, each new individual will be created with a crossover function which randomly chooses among the genes of two parents, considered as "father" and "mother". After eventually undergoing a random mutation process - the multiplication of genotypes, hence the value of each weight, by a random number between -2 and 2 - , with a probability set equal to 5%, these "children" are in turn used to produce a new generation. When 300 generations have been produced - iterations stopping criterion -, the algorithm obtains a certain solution for the weights values, which are passed to the NN by the function called *set_weights*.

3 Multi-agent Competition

As mentioned above, the second objective is the study of competition dynamics between independent agents. This in-depth analysis is the result of two games characterized by a different complexity from the conceptual and technical point of view. Therefore two new worlds are built, and agents competing within them will be based on a Q-Learning algorithm, without any Deep Reinforcement Learning generalization or alternative optimization methods, so as not to further increase the intricacy distinguishing these experiments.

3.1 Game 2: Duel between two agents

In Game 2, two players learn how to avoid fixed walls scattered in the grid and how to choose the best moment and way to approach and attack the opponent, in a duel where only one winner is allowed. Similarly to the definition of Bilbo's world, another Python class acts as a constructor for the elements of the game: the two rivals, whose locations are randomly initialized - by checking these starting positions to exist and be accessible -, and the obstacles randomly situated as well. Two functions, consequently recalled within the "Agent" class, perform the only two alternative actions allowed to the agents: one for execute a movement towards the opponent, one for carry out an attack. The second one can be carried out at any time of the game, equally to the action of the movement, but it is able to inflict damage only when rival's position is close enough. When one of them is get hit, he suffers a reduction in his so-called "health" parameter, which is initially set at 10 points. In details, the basic dynamics of the game can be summarized into the following rules:

- If a player carries out an attack at a distance that is not null and less than 1 in absolute value, makes sure that the health score of the other agent decreases by one point and this opponent receives a negative reward of -1

- If after 500 epochs, when the game ends, the rival's health score is higher than 0, then the player receives a 10 point penalty.

In this way, the agent is encouraged to approach and attack as soon as the distance becomes short enough, despite the fact that he himself could be injured, since the signal received if the other had residual health at the end of the game is much more negative with respect to the penalty received at each stroke suffered. Therefore, after having explored the world and developed sufficient experience in the various episodes, the two players will tend to face each other, without "fearing" the clash. If, on the other hand, the penalty in case of damage suffered was higher, the two agents could have preferred the escape, that is they would have limited themselves to move away from the opponent and maintain the distance between them beyond the unit in absolute value.
The operation mode is similar to the normal Q-learning algorithm, exception made for the way the two agents learn how to win the game: while Bilbo was limited to acquiring the right way to avoid an element that performed random movements, the dragon, and reach the treasure, in this experiment the duelists will be forced to react to the opponent's moves and choose the right moment to attack. In details, these dynamics are made possible by the function *get_action* within the "Agent" class: at each step, if a number randomly extracted from a uniform variable is less than ϵ hyperparameter, one system performs a random moves among the five possibilities, that are moving in one of the four walkable direction - "up", "down", "right",

"left" - or attack; otherwise, it chooses the action maximizing the Q-value by looking at the Q-matrix entries in correspondence with current state, which is composed of its position and the opponent's coordinates. Then, it gets the two new locations, checks the game state, receives the rewards. Finally, it computes the new Q-value for the next state - with the same updating rule explained in the section 2.1 - and stores it in the Q-matrix. These phases are iterated, as said previously, over 500 epochs.
The game will be repeated 100,000 times. The hyperparameters of both duellists are set as follows:

- $\gamma = 0.9$, making the agent slightly more "concerned" about next moves, hence more reactive to the opponent's strategy
- $\alpha = 0.75$, improving the learning rate, thus the pace at which it acquires new information
- $\epsilon = 0.05$, setting the probability of a random exploration up to 5%.

3.2 Game 3: Battle among five agents

This third experiment is configured as an extension of the previous game, in which 5 players will wander around the world and will be pushed, still by tuning the various penalties, to eliminate each other in a battle all against all. At this point, still generating an environment as a gridworld, the current state, input of each player's epoch, would have 10 dimensions, ie 2 coordinates for each of the 5 players. Therefore, every player's Q-matrix would have an excessive dimensionality, forcing the Q-learning algorithm to calculate the distance and the direction based on each of the 10 coordinates. In order to avoid such a computational weight, a technical improvement has been introduced: the conversion of the initial gridworld into a graph. By replacing the empty cells with a node, based on the positions of the predetermined obstacles, and combining these with edges to delineate the accessible paths, it is possible to exploit a clearly more efficient method of calculating distance: the shortest path. In details, during each epoch, with the purpose of being able to "observe" the moves of the adversaries within the environment, in order to react to them in a reasonable time, which means evaluating the current state and opt for the action that maximizes the q-value of the given state, each system, instead of normally monitoring the proximity of the others through the coordinates, uses the *"shortest_path"* function from the Python *"NetworkX"* package. In this way he gets the fastest route to reach every other character in the game. Then, select the shortest route among these 4 results, which is related to the closest enemy. Finally, to complete the update of the current status, it compares the y-delta associated with the path towards this target to the x-delta, obtaining the direction of the vector. This approach greatly reduces the time required for the computation of a given state, thereby decreasing the total duration of the learning process. Thanks to the new structure, the current status consists of only 2 values for each precise opponent:

- the direction - between the usual 4 - towards which the player must move to reach the rival,
- a binary number equal to 0 if the distance is null, that is when the players are in adjacent positions, and 1 otherwise.

Therefore, when the player has developed sufficient knowledge, his movement will tend to depend on the location of the closest enemy and to coincide with the best route.
As in the previous section, agents will learn to attack enemies to win the game. Also here, the attack action can be performed when the duelists are located on adjacent nodes and entails a decrease in the same health score mentioned above. However, this time, the rules are different because of the more complex dynamics:

- if a player carries out an attack from the shortest distance, he causes a damage that reduces the target's health by one point, and receives a positive reward of + 10. In the event that this attack leads to the annulment of the rival's health, coinciding with his elimination from the game, the agent would receive a reward of + 20.
- The "health" parameter is still set equal to 10. When one player's health becomes null, it means he has been killed, and he receives a penalty of -10.

Dead agents are now moved to an alternative world, outside the main environment within which the normal phases take place. Indeed, their position is set in correspondence with a grave character within a "graveyard" zone of the world, generated by the function *put_p_in_grave*.
Another change with respect to the previous experiment is the presence of the "fear" factor, calculated as follows:

$$\frac{1}{h} * \frac{1+d}{d}$$

with h = health score and d = distance from the enemy. This again becomes a multiplier of the ϵ hyperparameter, but unlike the first time, the probability of exploring the world, namely implementing a random move, is not only influenced by the normalized distance from the nearest enemy, but it is also inversely proportional to the agent's health state. This means that, for the same distance from the enemy, the "less healthy" player will be more likely to perform an action maximizing the Q-value. At the same time, greater proximity translates into greater prudence. This double mechanism tries to simulate the human instinct of self-preservation, increasing in situations of greater danger.
As far as Q-learning algorithm operation mode is concerned, it is identical to the one presented in the previous section, except for the fact of checking each agent's state of life or death during the game, and using distance and direction calculated by the shortest path method instead of the coordinates.
The experiment involves a maximum number of epochs equal to 400, at most 10 episodes and the following hyperparameters:

- $\gamma = 0.9$,
- $\alpha = 0.5$,

- $\epsilon = 0.5$,
- $\epsilon\ decay = 0.995$ up to 0.01.

Performances are still tracked at the end of every game.

4 Multi-agent Cooperation

The last part of the study focuses on the opposite dynamics with respect to competition: cooperation, with the aim of building systems able to learn to help each other - and understand the best way to do it - in achieving a common goal , which, in this case, coincides with the killing of the dragon. A fourth world scattered with obstacles is initialized by creating the same graph structure described above, in order to still efficiently manage the complexity of a collaborative game.

5 Conclusions

Deep Learning per il problema multi-agente ed altre possibili migliorie.