AI Project: Reinforcement learning for multiagent predators and prey problem
Authors: PB & Marek Drwal

Description of the problem

The aim of this project was to use reinforcement learning to solve the multiagent predators and prey problem. The predators were supposed to learn which moves allowed them to catch the prey. Each predator was supposed to learn individually. However, all predators were supposed to be rewarded when prey was caught by any of the predators.

Basic assumptions

We started solving the problem by imposing several constraints to the environment and the agents.

As for the game environment, we assumed that:

1. The environment should be placed in the two dimensional space in which there are N by M possible states (i.e., places on the map) in which each agent can theoretically be present.
2. In several squares, the obstacle should be placed. Such an obstacle indicates that a given space is inaccessible to the agents.
3. The board is of limited size. Consequently, the agents cannot cross the borders of the game board.

As for the agents, we assumed that:

1. The number of predators should be 3.
2. The number of prey should be 2.
3. The agents should be able to move in one of the eight possible directions:
    a. up,
    b. up-right,
    c. right,
    d. down-right,
    e. down,
    f. down-left,
    g. left,
    h. and up-left.
4. The agents always move in their turn.

Moreover, we concluded that the algorithm, which we should use in this project should be the Q-learning algorithm and the Q-table.

Refined assumptions (after first consultations)

After first consultations, we decided to slightly modify our assumptions. First of all, we decided to change the classical Q-table into Deep Q Network (DQN) as we were informed that the classical Q-table would most likely be too complex for the number of states in our problem and that its computation would be problematic without any modifications such as approximations. Consequently, we estimated the Q values with a neural network with one hidden layer. Additionally, to simplify the problem even more, we decided to reduce the number of prey from 2 to 1 and to reduce the number of possible moves (i.e., actions) to four options: up, right, down, and left.

Chosen technologies

We chose to implement the project in Python. We used Pygame to create the game environment (based on Tiger Deer: https://magent2.farama.org/environments/tiger_deer/) and PyTorch to implement the DQN.

Stages of development

We started with developing the most basic version of the problem. There were two agents, one predator and one prey. To teach the predator to catch the prey, we experimented with various parameterization of the states, reward functions, values of penalty and reward, as well as value of the epsilon hyperparameter.

During this process we eventually ended up with the following assumptions:

1. Each state is described by a 1 x 8 tensor which indicates whether:
    a. the predator can freely move to the adjacent square (there is no obstacle or boarder of the board):
        i. right,
        ii. left,
        iii. up,
        iv. or down.
    b. a prey is located:
        i. right,
        ii. left,
        iii. up,
        iv. or down,

        with respect to the predator.
2. The value in the tensor can take only 0 or 1.
3. The board is fixed to 12x16 cells.
4. When trying to move into an obstacle or outside of the board, the predator receives a penalty equal to -500.
5. When not catching the prey after 10,000 steps of the game, all predators receive a penalty equal to -5,000.
6. When a predator catches the prey it receives a reward equal to 100,000, while the other predators receive a reward of 33,333.
7. When a predator moves to an adjacent field it receives a reward equal to (100,000 / distance between predator and prey) / 100,000 (e.g., when distance = 1, reward = 1; when distance = 2, reward = 0.5).
8. The epsilon should decay over time. During the first round, the agents have 80% chance to choose exploration and 20% to choose exploitation. After each round, the chance for exploration is lower by 0.5% (79.5% in the second round) and higher for exploitation by the same amount (20.5% in the second round).
9. The problem of dependency between the two consecutive actions needs to be addressed. We decided to address it by implementing Prioritised Experience Replay, which saves the experiences in the “long term memory” from which the agent picks a random batch of experiences to learn from.

After successfully teaching one predator to catch the prey, we added obstacles on the map as well as two additional predators. After this addition, each predator was learning individually and the learning was successful as well.

Visualisations

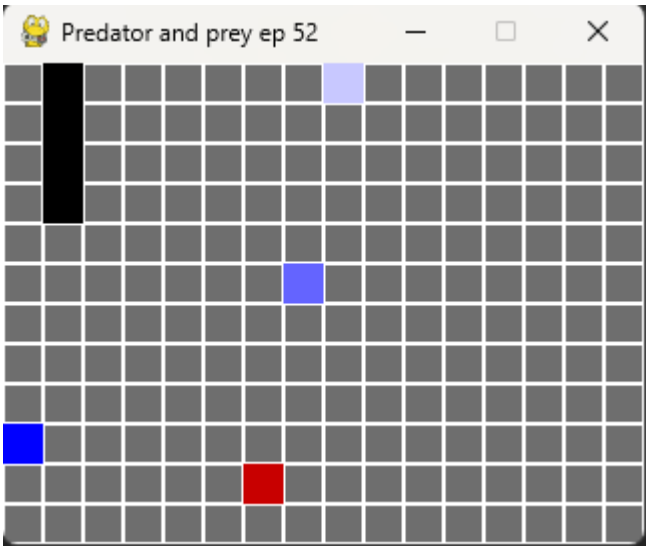

Figure 1. The initial position of the agents.

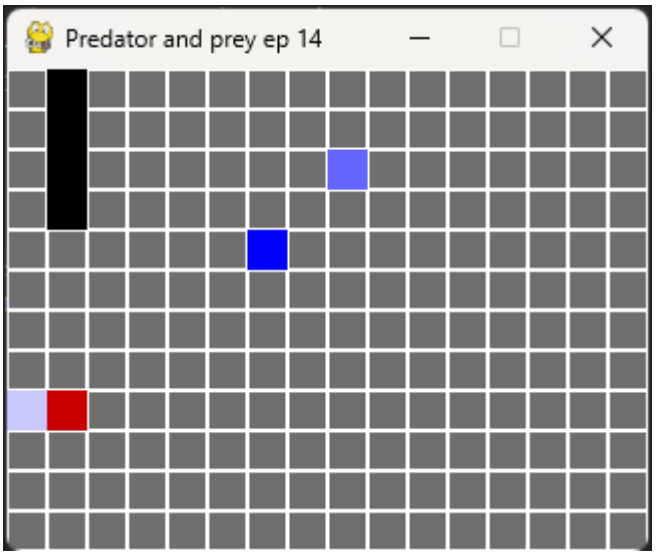

Figure 2. The final position of the agents.

Implementation overview

The source code of the project comprises three files and four classes. *game.py* is responsible for the main game engine. It utilises the Pygame library and runs simulation's logic. Its *Environment* class displays the game, provides calculations for moves and determines the outcome of a step, as well as the reward granted for it. *agent.py* contains class *Agent* and controls the course of the game, sequentially running agents in consecutive episodes. Here the state is observed and training launched. Moreover, agents' actions are determined and executed. *Model.py* consists of two classes. *DQN* implements a three-layer neural network with ReLU as the activation function. Its aim is to model a move of an agent

based on their state. *QTrainer* obtains and evaluates the outcome predicted by the network, given a step or a batch of steps. Ultimately, it optimises the network using Adam as the optimiser and Mean Square Error as the loss function. Overall rigidity is avoided, since all game parameters are easily adjustable via constants.

Limitations

Even though our agents were learning rather well (the visible effects of learning were easily observable between the 50th and the 100th round of the game), we observed that our model had a tendency to overfit after some additional number of rounds (usually between the 150th and 200th round of the game). The overfit was observable as a constant attempt to go outside of the borders of the map. The most likely cause of such behaviour was too fast growth of weights in the artificial neural network. There are several known remedies for this problem. One of the examples is the double DQN. In this approach instead of one network there are two. One network is used to select the action with the highest Q-value, and the other is used to estimate its value.

Sources

1. https://towardsdatascience.com/techniques-to-improve-the-performance-of-a-dqn-agent-29da8a7a0a7e