ARTIFICIAL INTELLIGENCE (E016330)
ALEKSANDRA PIZURICA
GHENT UNIVERSITY
2022/2023 AY

Assistants: Xianlu Li, Nicolas Nadisic,
Ide Van den Borre
Professor: Aleksandra Pizurica

Project Session 2: Reinforcement Learning

1 Introduction

The goal of this project is to understand the theory of reinforcement learning and learn how to apply it to some practical tasks. The project is based on the material from the Berkeley course CS188 in Artificial Intelligence[1]. During your implementation, you will test your agents first on the 4×3 grid world ('Gridworld') that you have seen in the theory classes, then apply them to Pacman.

The code for this project consists of several Python files, which are available on Ufora. Please download the file `reinforcement.zip` in the section "Project 2" on Ufora, and do not reuse the code from previous assignments.

You will only need to modify the file `qlearningAgents.py` to complete all the tasks. Besides, the `learningAgents.py`,`util.py`,`gridworld.py` and `featureExtractors.py` are also quite important for your implementation, please take a look before coding. However, you should not modify them. The other files serve only as supporting files and you should not need to look at them.

This project is done in groups of 2 students (with the exceptions we accepted). Each group will have to submit their code and a small report, see Section 3.

2 Assignment

Markov Decision Processes (MDPs)

To get familiar with the Gridworld environment run:

```
python gridworld.py -m
```

You have to manually select the desired actions using the arrow keys. Note that executing an action leads to the intended outcome only with a probability of 80%. E.g., if you press "right", you will effectively go right only 80% of the time.

To see the list of options, run:

```
python gridworld.py -h
```

[1]https://inst.eecs.berkeley.edu/~cs188/fa22/projects/

Note: In order to exit the Gridworld, you first need to enter a pre-terminal state (the double boxes shown in the GUI) and take the specific *exit* action. The true terminal state is here referred to as `TERMINAL_STATE`. Running an episode manually may result in a total return which is less than you expected because of the implemented discount rate `-d` (default 0.9).

The default agent moves randomly:

```
python gridworld.py -g MazeGrid
```

Notice that the simple random agent wanders around until it accidentally hits an exit. It is certainly not the smartest AI agent. In this project, we would like to improve the decision making process of the agent by implementing Q-Learing, a model-free reinforcement learning algorithm.

As in Pacman, the positions are represented by `(x, y)` Cartesian coordinates, with `'north'` being the direction of increasing `y`. By default, all transitions will receive a reward of zero, except for the exit action.

Question 1 *When does a MDP become a reinforcement learning problem? Explain in a few sentences.*

Task 1: Q-Learning (11 points)

In this task you will have to implement a Q-Learning agent. This agent learns from each experience like the temporal-difference (TD) learning agent that you have seen in the class. Only now the agent learns the Q-Values instead of the values of the states. This is in fact TD learning of the Q-Values. Based on each transition (s, a, s', r), the agent updates the Q-Value with a moving average:

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a')\right] \tag{1}$$

All the Q-Values are initialized as zero. A part of a Q-Learning agent is specified in `QLearningAgent` in `qlearningAgents.py`. For this task, the following methods have to be implemented:

- `update`
- `getQValue`
- `computeValueFromQValues`
- `computeActionFromQValues`

Breaking ties: Select randomly an action in `computeActionFromQValues`, if there are multiple actions that result in the same maximal Q-Value.

Important: Make sure that you only access the Q-Values by calling `getQValue` in your `computeValueFromQValues` and `computeActionFromQValues` methods. This will be useful in the last task where we will overwrite the `getQValue` method in order to use features of state-action pairs.

After implementing Q-Learning, you can observe how the Q-Learning agent learns under manual control in the Gridworld:

```
python gridworld.py -a q -k 5 -m
```

In this case, the agents can learn from a total number of 5 episodes (`-k` option).

To check if your implementation is correct, run:

```
python autograder.py -q t1
```

The autograder will check of your Q-Learning agent learns the same Q-Values and policy as the reference implementation, given the same set of examples.

Question 2 *Compare the expressions for the sample in TD learning and in Q-Learning. What is essentially different and why? (Write the two expressions and explain the essential difference in one or two sentences)*

Question 3 *Why is Q-learning superior to TD learning of values? (Explain in one or two sentences)*

Question 4 *Explain how γ will affect the learning process? And what will happen if γ is set to 0 or 1?*

Task 2: Epsilon Greedy and Q-Learning in Pacman (8 points)

Gridworld: We will now complete the Q-Learning agent by implementing an **epsilon-greedy** action selection in `getAction`. This means that agent will choose random legal actions with a probability of epsilon, and will pick the current best Q-Values otherwise. Note that this random legal action could be the optimal action.

You can use the `util.flipCoin(p)` method to simulate a random binary variable, that will return `True` with a probability of `p` and `False` with a probability of `1-p`.

After this, you should be able to run without any errors:

```
python gridworld.py -a q -k 100
```

We will now simulate the Q-Learning agent using two different epsilon values:

```
python gridworld.py -a q -k 100 --noise 0.0 -e 0.1
```

```
python gridworld.py -a q -k 100 --noise 0.0 -e 0.9
```

Question 5 *Explain why and how the behaviour differs in the two cases with different epsilon values. Does the agent act as you would expect? Explain in about 10 lines.*

You can verify your implemetation again by running the following command:

```
python autograder.py -q t2.1
```

Until now, we kept epsilon fixed while training. We could also define an adaptive epsilon whose value changes over the different episodes:

$$\varepsilon = \varepsilon_{min} + (\varepsilon_{max} - \varepsilon_{min})\varepsilon^{-\beta i} \tag{2}$$

where ε_{min} and ε_{max} denote respectively the minimum and maximum allowed epsilon, β the decay rate for epsilon and i the episode index. Implement this adaptive epsilon in `QLearningAgent`.

Question 6 *Include a plot that depicts how the epsilon values change over the different episodes. You can set ε_{min} and ε_{max} to 0 and 1. Choose an appropriate decay rate for the specific task. Discuss in a few lines, based on the plot, how you expect the agent to behave in the Gridworld over the different episodes.*

Question 7 *Make a plot that shows how the rewards gained by the agent change over the episodes and discuss. You are free to choose the different parameters of the adaptive epsilon but make sure that they make sense.*

Pacman: We will now change to the Pacman world! **If you correctly implemented task 1 and the epsilon greedy action selection, you do not write any code for this part!** You can again use the default epsilon value (and not the adaptive definition).

Pacman will play in two phases: a training phase and testing phase. In the training phase, Pacman will try to learn the values of different positions and actions. The training phase will run quietly without GUI display. After finishing the training phase, Pacman will enter the testing phase. In this phase, `self.epsilon` and `self.alpha` will be set to 0.0, in order to stop the Q-Learning and to allow Pacman to exploit his learned policy. Without any code modifications you should be able to run Q-Learning Pacman for a small grid:

```
python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid
```

The `PacmanQAgent` is already defined in terms of the `QLearningAgent` you wrote. You should observe that Pacman wins at least 80% of the time. Again, you can run the autograder to see the performance on different examples:

```
python autograder.py -q t2.2
```

Task 3: Approximate Q-Learning (11 points)

In the last task, we will implement an approximate Q-Learning agent that you have seen in the theory lecture on active RL. Write your implementation in `ApproximateQAgent` class in `qlearningAgents.py`, which is a subclass of `PacmanQAgent`. The approximate Q-Values are obtained now according to:

$$Q(s, a) = \sum_{i=1}^{n} f_i(s, a) w_i \tag{3}$$

where the weight w_i expresses the importance of the feature $f_i(s, a)$. The approximate Q-Learning agent will try to learn the weights for the particular features that describe pairs of states and actions. Many state-action pairs might share the same feature values. In your code, you should implement the weight vector as a dictionary that maps the features to the corresponding weight values. The weight values are updated as:

$$\begin{aligned} w_i &= w_i + \alpha * diff * f_i(s, a) \\ diff &= (r + \gamma \max_{a'} Q(s', a')) - Q(s, a) \end{aligned} \tag{4}$$

Note: Approximate Q-Learning requires a feature function $f(s, a)$ over state and action pairs, which yields a vector $[f_1(s, a), ..., f_i(s, a), ...f_n(s, a)]$ of feature values. The feature functions are provided in `featureExtractors.py`. Feature vectors are `util.Counter` objects (like a dictionary) containing the non-zero pairs of features and values. Omitted features have value zero.

`ApproximateQAgent` uses by default the `IdentityExtractor`. This assigns a single feature to every `(state,action)` pair. Using this feature extractor, your approximate Q-Learning agent should behave identically to the previously implemented `PacmanQAgent`. You can verify this by running:

```
python pacman.py -p ApproximateQAgent -x 2000 -n 2010 -l smallGrid
```

Important: `ApproximateQAgent` inherits from `QLearningAgent`, and therefore shares several functions like `getAction`. Make sure to always access Q-Values by calling `getQValue` in your implemented methods in `QLearningAgent`. Otherwise, the new approximate Q-Values will not be used when you overwrite `getQValue` in your approximate agent.

Once you feel confident about your approximate Q-Learning agent, you can run the following command with a custom feature extractor:

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor
-x 50 -n 60 -l mediumGrid
```

This agent should learn to win easily, even with only 50 episodes of training. Moreover, much larger layouts should be no problem for this intelligent `ApproximateQAgent`:

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor
-x 50 -n 60 -l mediumClassic
```

If you implemented this agent correctly, the agent should win almost every time. Finally you can run the autograder to check your performance on different examples:

```
python autograder.py -q t3
```

Question 8 *What are the advantages of approximate Q-Learning over the naive Q-Learning?*

Question 9 *Have a look at the `SimpleExtractor` and notice how the features are defined in this case. Propose concretely your own (different) features that would improve the performance of Pacman (hint: Pacman gets bonus points when he eats scared ghosts after eating a power pellet). Explain your choice of features. Give the corresponding expression for the linear value function $Q(s, a)$.*

Bonus Question:

Question 10 *Implement your own features and include your average score of 10 testing games after 50 training episodes on the `mediumClassic` layout. Include a screenshot of the code you added for this question in the report.*

3 Submission

You need to submit 2 files:

- Your final version of the file `qlearningAgents.py` .
- A report in PDF containing the answers to our questions. Please answer the questions separately, indicating the questions numbers.

A few remarks:

- Deadline: **Wednesday 14th December 23:59 CET**
- Submit on the Ufora course page, Ufora-tools → Assignments → Project 2 - Reinforcement Learning
- These files must be submitted as one zip file named after both authors. For example, for the group of Xianlu Li and Ide Van den Borre, name the file `reinforcement_xli_ivandenborre.zip`
- The evaluation of the code is based on its execution. Make sure your code works with the original Pacman framework provided and with the autograder.
- Write clear and concise answers, using the right terminology (as in the theory course).
- Plagiarism will be very heavily penalized. We use automatic tools to check for plagiarism, and they are quite hard to fool. Helping each other is a good thing, but do not share your code or answers.