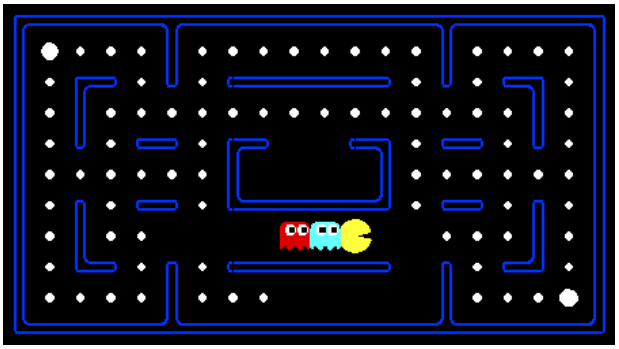
Spring 2024

Introduction to Artificial Intelligence

Homework 3

Apr. 16, 2024



Introduction

For those of you not familiar with Pac-Man, it's a game where pacman (the yellow circle with a mouth) moves around in a maze and tries to eat as many food pellets (the small white dots) as possible, while avoiding the ghosts (the other two agents with eyes). If pacman eats all the food in a maze, it wins. The big white dots are capsules, which give pacman power to eat ghosts in a limited time. You can play this game directly here to get familiar with the game.

In this assignment, you will design agents for the simple version of Pac-Man. The goal of this programming assignment is to learn how to 1) implement **minimax search** and **expectimax search**, 2) implement a basic reinforcement learning model (i.e., **Q-learning)**, and 3) compare the results and performance with the deep reinforcement learning model (i.e., **DQN** provided in this assignment). Please make sure you understand the concept of adversarial search and reinforcement learning before working on HW3.

The code base of Pac-Man was developed by UC Berkeley. (https://inst.eecs.berkeley.edu/~cs188/sp21/project2/)

(https://inst.eecs.berkeley.edu/~cs188/sp21/project6/)

You can only run the code base on a local machine. Google Colab cannot execute it because of not providing GUI. Please install **Python 3.7** on your own machine and be familiar with the code with CLI.

Files

There are 3 folders, which are different ways to implement the pacman. The table below describes files you will edit (marked in blue) and files you need to take a look at.
(You can also check out other files, if you are interested in the details of this game)

Folder	File name	Description
Adversarial_search	multiAgents.py	All of your multi-agent search agents will be resided.
	pacman.py	The main file that runs Pac-Man games. This file also describes a pacman GameState type, which you will use extensively in this assignment.
	game.py	The logic behind how the Pac-Man world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
	util.py	Useful data structures for implementing search algorithms. You don't need to use these for this assignment, but may find other functions defined here to be useful.
Q-learning	valueIterationAgents.py	A value iteration agent for solving known MDPs.
	qlearningAgents.py	Q-learning agents for Gridworld, Crawler and Pacman.
	mdp.py	Defines methods on general MDPs
	learningAgents.py	Define the base classes ValueEstimationAgent and QLearningAgent, which your agents will extend.
	util.py	Utilities, including util.Counter, which is particularly useful for Q-learners.
	gridworld.py	Implementation of Gridworld.
	featureExtractors.py (optional)	Classes for extracting features on (state, action) pairs. Used for the approximate Q-learning agent (in qlearningAgents.py).

DQN	DQN.py	DQN model architecture.
	pacmanDQN_Agent.py	DQN agents (you can modify some parameters.)

Part 1 : Adversarial search

Please move your current path to the Adversarial_search folder first.

Next, play a game of classic Pac-Man by running the following command:

```
python pacman.py
```

and using the arrow keys to move.

Next, run the given ReflexAgent in multiAgents.py:

```
python pacman.py -p ReflexAgent
```

Note that it plays quite poorly even on simple layouts:

```
python pacman.py -p ReflexAgent -l testClassic
```

Other Options:

1. Default ghosts are random. You can also play for fun with slightly smarter directional ghosts using -g DirectionalGhost.
2. You can play multiple games in one command with -n.
3. You can turn off graphics with -q to run games quickly.
4. Use -h to know more options.

Autograding in Part 1:

In this part, TAs will use an autograder to grade your implementation. The autograder has been included in the code base. You can use the following command to test by yourself.

```
python autograder.py
```

The autograder will check your code to determine whether it explores the correct number of game states. This will show what your implementation does on some simulated trees and Pac-Man games. After that, it will show the score you get in Part 1.

Using the autograder to debug is recommended and will help you to find bugs quickly.
To test and debug your code for one particular part, run the following command:

```
python autograder.py -q part1-1
```

To run it without graphics, use the following command:

```
python autograder.py -q part1-1 --no-graphics
```

Please do not change the names of any provided functions or classes within the code for technical correctness, or you will wreak havoc on the autograder.

Requirements in Part 1:

Please modify the codes in multiAgents.py between # Begin your code and # End your code. In addition, do not import other packages.

All agents you will implement should work with any number of ghosts. In particular, your search tree will have multiple min/chance layers (one for each ghost) for every max layer.

Your code should also expand the game tree to arbitrary depth with the provided self.depth. A single level of the search is considered to be one pacman move and all the ghosts' responses, so the depth 2 search will involve pacman and each ghost moving twice.

Your code should score the leaves of your search tree with the provided self.evaluationFunction, which defaults to scoreEvaluationFunction.

All agents you will implement extend MultiAgentSearchAgent, which gives access to self.depth and self.evaluationFunction. Those two variables are populated in response to command line options.

Part 1-1: Minimax Search (10%)

- Write an adversarial search agent in the provided MinimaxAgent class stub in multiAgents.py.
- The actual ghosts operating in the environment may act partially randomly, but the minimax algorithm assumes the worst.

Observations:

- Make sure you understand why pac man rushes to the closest ghost in this case:

```
python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
```

- The minimax values of the initial state in the minimaxClassic layout are 9, 8, 7, -492 for depths 1, 2, 3 and 4 respectively. Note that your minimax agent will often win, despite the dire prediction of depth 4 minimax, whose command is shown below. Our agent wins 50-70% of the time: Be sure to test on a large number of games using the -n and -q flags.

```
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
```

Part 1-2: Expectimax Search (10%)

- Minimax and alpha-beta are great, but they both assume that you are playing against an adversary who makes optimal decisions. As anyone who has ever won tic-tac-toe can tell you, this is not always the case.
- In this part, you will implement the ExpectimaxAgent class in multiAgents.py, which is useful for modeling the probabilistic behavior of agents who may make suboptimal choices.
- Random ghosts are of course not optimal minimax agents, so modeling them with minimax search may not be appropriate. Rather than taking the MIN over all ghost actions, expectimax agent will take the expectation according to your agent's model of how the ghosts act. To simplify your code, please assume you will only be running against an adversary that chooses among its legal actions uniformly at random.

Observations:

- To see how the ExpectimaxAgent behaves in Pac-Man, run the following command:

```
python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3
```

  You should observe a more cavalier approach in close quarters with ghosts.

- In particular, if pacman perceives that he could be trapped but might escape to grab a few more pieces of food, he'll at least try. Investigate the results of the cases:

```
python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10
```

  You should find that your ExpectimaxAgent wins about half the time. Make sure you understand why the behavior here differs from the minimax case.

Part 2 : Q-learning

Please move your current path to the Q-learning folder first.

To get started, run Gridworld in manual control mode, which uses the arrow keys:

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

You will see the two-exit layout. The blue dot is the agent. Note that when you press up, the agent only actually moves north 80% of the time. Such is the life of a Gridworld agent!

You can control many aspects of the simulation. A full list of options is available by running:

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

The default agent moves randomly

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

You should see the random agent bounce around the grid until it happens upon an exit. Not the finest hour for an AI agent.
Note: The Gridworld MDP is such that you first must enter a pre-terminal state (the double boxes shown in the GUI) and then take the special 'exit' action before the episode actually ends (in the true terminal state called `TERMINAL_STATE`, which is not shown in the GUI). If you run an episode manually, your total return may be less than you expected, due to the discount rate (`-d` to change; 0.9 by default).

Look at the console output that accompanies the graphical output (or use `-t` for all text). You will be told about each transition the agent experiences (to turn this off, use `-q`).

As in Pacman, positions are represented by `(x,y)` Cartesian coordinates and any arrays are indexed by `[x][y]`, with `'north'` being the direction of increasing `y`, etc. By default, most transitions will receive a reward of zero, though you can change this with the living reward option (`-r`).

Requirements in Part 2:

Please modify the codes in valueIterationAgents.py and qlearningAgents.py between # Begin your code and # End your code. In addition, do not import other packages.

Autograding in Part 2:

Same as part1, this autograder for you to grade your solutions on your machine. This can be run on all questions with the command:

```
python autograder.py
```

It will show the score you get in Part 2.

Part 2-1: Value Iteration (10%)

- Write a value iteration agent in ValueIterationAgent, which has been partially specified for you in valueIterationAgents.py. Your value iteration agent is an offline planner, not a reinforcement learning agent, and so the relevant training option is the number of iterations of value iteration it should run (option -i) in its initial planning phase. ValueIterationAgent takes an MDP on construction and runs value iteration for the specified number of iterations before the constructor returns.
- Value iteration computes k-step estimates of the optimal values, V_k. In addition to `runValueIteration`, implement the following methods for ValueIterationAgent using V_k.
  - `computeActionFromValues(state)` computes the best action according to the value function given by `self.values`
  - `computeQValueFromValues(state, action)` returns the Q-value of the (state, action) pair given by the value function given by `self.values`
- These quantities are all displayed in the GUI: values are numbers in squares, Q-values are numbers in square quarters, and policies are arrows out from each square.
- important: Use the "batch" version of value iteration where each vector V_k is computed from a fixed vector V_{k-1}, not the "online" version where one single weight vector is updated in place. This means that when a state's value is updated in iteration `k` based on the values of its successor states, the successor state values used in the value update computation should be those from iteration `k-1`
- Note: A policy synthesized from values of depth k (which reflect the next k rewards) will actually reflect the next k+1 rewards (i.e. you return π_{k+1}). Similarly, the Q-values will also reflect one more reward than the values (i.e. you return Q_{k+1}). You may either return the synthesized policy π_{k+1} or the actual policy for the k-th iteration, π_k, which you'll get if you store optimal actions from the most recent round of value iteration updates.
- Hint: Use the `util.Counter` class in util.py, which is a dictionary with a default value of zero. Methods such as `totalCount` should simplify your code. However, be careful with `argMax`: the actual argmax you want may be a key not in the counter!
- Note: Make sure to handle the case when a state has no available actions in an MDP (think about what this means for future rewards).

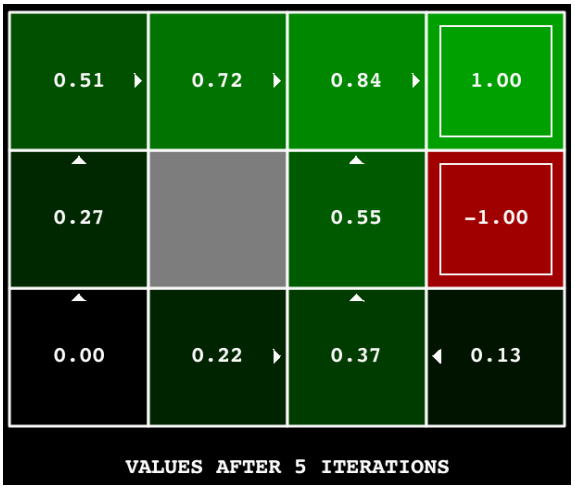
Observations:

- The following command loads your ValueIterationAgent, which will compute a policy and execute it 10 times. Press a key to cycle through values, Q-values, and the simulation. You should find that the value of the start state (`V(start)`, which you can read off of the GUI) and the empirical resulting average reward (printed after the 10 rounds of execution finish) are quite close.

```
python gridworld.py -a value -i 100 -k 10
```

- On the BookGrid, running value iteration for 5 iterations should give you this output:

```
python gridworld.py -a value -i 5
```



Grading:

- Your value iteration agent will be graded on a new grid. We will check your values, Q-values, and policies after fixed numbers of iterations and at convergence (e.g. after 100 iterations).

Part 2-2: Q-learning (15%)

- Note that your value iteration agent does not actually learn from experience. Rather, it ponders its MDP model to arrive at a complete policy before ever interacting with a real environment. When it does interact with the environment, it simply follows the precomputed policy (e.g. it becomes a reflex agent). This distinction may be subtle in a simulated environment like a Gridword, but it's very important in the real world, where the real MDP is not available.
- You will now write a Q-learning agent, which does very little on construction, but instead learns by trial and error from interactions with the environment through its `update(state, action, nextState, reward)` method. A stub of a Q-learner is specified in QLearningAgent in qlearningAgents.py, and you can select it with the

option '`-a q`'. For this question, you must implement the `update`, `computeValueFromQValues`, `getQValue` and `computeActionFromQValues` methods.

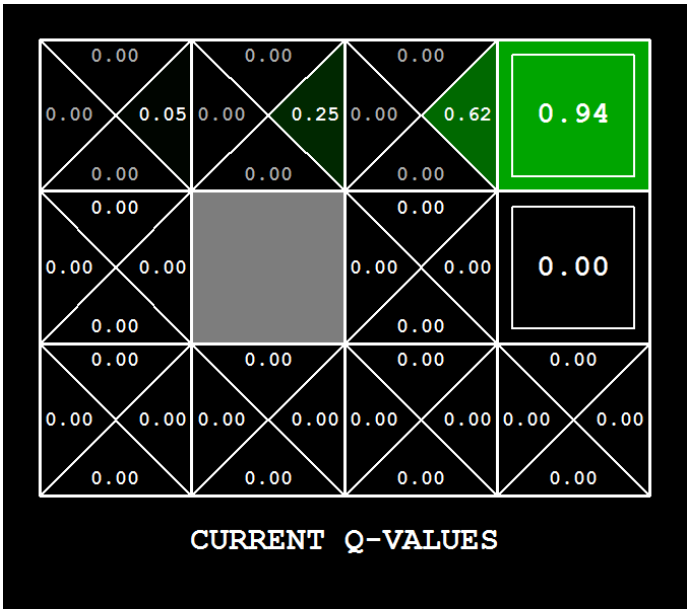
- Note: For `computeActionFromQValues`, you should break ties randomly for better behavior. The `random.choice()` function will help. In a particular state, actions that your agent hasn't seen before still have a Q-value, specifically a Q-value of zero, and if all of the actions that your agent has seen before have a negative Q-value, an unseen action may be optimal.
- Important: Make sure that in your `getValue` and `getPolicy` functions, you only access Q values by calling `getQValue` . This abstraction will be useful for question 9 when you override `getQValue` to use features of state-action pairs rather than state-action pairs directly.

Observations:

- With the Q-learning update in place, you can watch your Q-learner learn under manual control, using the keyboard:

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

- Recall that `-k` will control the number of episodes your agent gets to learn. Watch how the agent learns about the state it was just in, not the one it moves to, and "leaves learning in its wake." Hint: To help with debugging, you can turn off noise by using the `--noise 0.0` parameter (though this obviously makes Q-learning less interesting). If you manually steer Pacman north and then east along the optimal path for four episodes, you should see the following Q-values:



Grading:

- We will run your Q-learning agent on an example of our own and check that it learns the same Q-values and policy as our reference implementation when each is presented with the same set of examples.

Part 2-3: epsilon-greedy action selection (10%)

- Complete your Q-learning agent by implementing epsilon-greedy action selection in `getAction` in qlearningAgents.py, meaning it chooses random actions an epsilon fraction of the time, and follows its current best Q-values otherwise. Note that choosing a random action may result in choosing the best action - that is, you should not choose a random sub-optimal action, but rather any random legal action.
- You can choose an element from a list uniformly at random by calling the `random.choice` function. You can simulate a binary variable with probability p of success by using `util.flipCoin(p)`, which returns `True` with probability `p` and `False` with probability `1-p`.

Observations:

- After implementing the getAction method, observe the following behavior of the agent in gridworld.

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

- Your final Q-values should resemble those of your value iteration agent, especially along well-traveled paths. However, your average returns will be lower than the Q-values predicted because of the random actions and the initial learning phase.
- You can also observe the following simulations for different epsilon values. Does that behavior of the agent match what you expect? (discuss it in your report)

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

Grading:

- Same as part 2-2

Part 2-4: Approximate Q-learning (Bonus) (10%)

- Implement an approximate Q-learning agent that learns weights for features of states, where many states might share the same features. Write your implementation in ApproximateQAgent class in qlearningAgents.py, which is a subclass of PacmanQAgent.
- Note: Approximate Q-learning assumes the existence of 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. We provide feature functions for you in featureExtractors.py. Feature vectors are `util.Counter` (like a dictionary) objects containing the non-zero pairs of features and values; all omitted features have value zero.

- The approximate Q-function takes the following form

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

- where each weight wi is associated with a particular feature $f_i(s,a)$. In your code, you should implement the weight vector as a dictionary mapping features (which the feature extractors will return) to weight values. You will update your weight vectors similarly to how you updated Q-values:

$$\begin{aligned} w_i &\leftarrow w_i + \alpha[correction] f_i(s,a) \\ correction &= (R(s,a) + \gamma V(s')) - Q(s,a) \end{aligned}$$

  (Note that the correction term is the same as in normal Q-learning.)
- Important: ApproximateQAgent is a subclass of QLearningAgent, and it, therefore, shares several methods like getAction. Make sure that your methods in QLearningAgent call `getQValue` instead of accessing Q-values directly, so that when you override `getQValue` in your approximate agent, the new approximate q-values are used to compute actions.

Observations:

- By default, ApproximateQAgent uses the `IdentityExtractor`, which assigns a single feature to every (state, action) pair. With this feature extractor, your approximate Q-learning agent should work identically to PacmanQAgent. You can test this with the following command:

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

- Once you're confident that your approximate learner works correctly with the identity features, run your approximate Q-learning agent with our custom feature extractor, which can learn to win with ease:

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

Grading:

- We will test your ApproximateQAgent, please record the test results for comparison with other methods, and discuss in the report.
- In this part, you can modify your `SimpleExtractor` in featureExtractors.py. if you modified your featureExtractors.py, please submit this file.
- The autograder will run your ApproximateQAgent on the smallClassic layout 100 times. We will assign points to your evaluation function in the following way:
  - If you win at least once without timing out the autograder, you get 1 point. Any agent not satisfying these criteria will receive 0 points.

    - +2 for winning at least 40 times, +3 for winning at least 70 times, +4 for winning all 100 times
    - +2 for an average score of at least 500, +4 for an average score of at least 1000 (including scores on lost games)
    - +1 for no timeout at least 50 times, +2 for no timeout all 100 times.
- The autograder will be run on the same machine with --no-graphics.

Part 3 : DQN (10%)

Please move your current path to the DQN folder first.
This part is provided directly to you, you can just follow the instructions below or you can freely modify the parameters (pacmanDQN_Agents.py) or network architecture (DQN.py), **the score of the Pacman game will be mainly valued in the description of your report.**

Training the DQN model is a time-intensive process; as a result, we have included the pre-trained models, namely **pacman_policy_net.pt** and **pacman_target_net.pt**, within the DQN folder. You have the option to either train your own model and evaluate it or directly utilize the pre-trained model we offer for evaluation.

- To train the DQN network, launch:
  Note: You need to change `model_trained into False` in pacmanDQN_Agents.py (line 26)

  ```
  python pacman.py -p PacmanDQN -n 10000 -x 10000 -l smallClassic
  ```

  After training you will get **pacman_policy_net.pt** and **pacman_target_net.pt.** These two files are the weight files of the DQN model.

- To test the DQN network, launch:
  Note: 1) Instruction will demonstrate the game of subtracting value of -x times from value of -n, so it is recommended that values_n and values_x should not differ too much 2) change `model_trained into True`

  ```
  python pacman.py -p PacmanDQN -n 25 -x 20 -l smallClassic
  ```

  Please record the test results for comparison with other methods, and discuss in the report.

Note: Parameters can be found in the params dictionary in pacmanDQN_Agents.py :

```
Episodes before training starts: train_start
```

Size of replay memory batch size: `batch_size`
Amount of experience tuples in replay memory: `mem_size`
Discount rate (gamma value): `discount`
Learning rate: `lr`
Exploration/Exploitation (ε-greedy):

- Epsilon start value: `eps`
- Epsilon final value: `eps_final`
- Number of steps between start and final epsilon value (linear): `eps_step`

Report (35%)

- You are required to submit a report and it can be written in Chinese or English.
- For parts 1 ~ 3, please take some screenshots of your code and explain how you implement codes in detail. (Please describe every piece of code corresponding to which step in the Minimax Search / Expectimax Search / Q-learning.)
- You can reference the learning resource we provided to answer the following questions. Recommend answering questions as clearly as possible, **the main score in part 3 will be based on these questions**.
    - What is the difference between On-policy and Off-policy
    - Briefly explain value-based, policy-based and Actor-Critic. Also, describe the value function $V^{\pi}(S)$.
    - What is the difference between Monte-Carlo (MC) based approach and Temporal-difference (TD) approach for estimating $V^{\pi}(S)$.
    - Describe State-action value function $Q^{\pi}(s, a)$ and the relationship between $V^{\pi}(S)$ in Q-learning.
    - Describe following tips Target Network, Exploration and Replay Buffer using in Q-learning.
    - Explain what is different between DQN and Q-learning.
- Compare the performance of every method and do some discussions in your report.
- Describe problems you meet and how you solve them.
- Save the report as a **.pdf** file

Discussion

TAs had opened a channel **HW3 討論區** on Microsoft Teams of the course, you can ask questions about the homework in the channel. TAs will answer questions in the channel as soon as possible.

Discussion rules:

1. Do not ask for the answer to the homework.
2. Check if someone has asked the question you have before asking.
3. We encourage you to answer other students' questions, but again, do not give the answer to the homework. Reply to the messages to answer questions.
4. Since we have this discussion channel, do not send emails to ask questions about the homework unless the questions are personal and you do not want to ask publicly.

Submission

1. **The deadline for this homework is 5/6 (Mon.) 23:55:00.**
2. Please submit one zip file that contains the Python code files we need, your weight files after deep reinforcement learning and report with the format hw3_{StudentID}.pdf (e.g., hw3_109123456.pdf).
3. Submit the zip file with the filename of hw3_{StudentID}.zip (e.g., hw3_109123456.zip).
4. Late submission leads to a score of (original score)*0.7, for example, if you submit homework right after the deadline, you will get (original score)*0.7 points.
5. **We only accept one zip file**, wrong format, or naming format cause -10 points to your score (after considering late submission penalty).
6. TA will run your code file. The environment will be **Python 3.10**. **Make sure your code can run in this environment correctly.**
7. **Plagiarism** is not allowed! You will get a huge penalty if we find that.
8. If there is anything you are not sure about submission, ask in the discussion forum.

Files you need to submit	Description
multiAgents.py	The file you edited in the Adversarial search part (part 1)
valueIterationAgents.py	The file you edited in the Q-learning part (part 2)
qlearningAgents.py	The file you edited in the Q-learning part (part 2)
featureExtractors.py (optional)	The file you edited in the Adversarial search part (part 2-4)
pacman_policy_net.pt	DQN model weights (part 3) (you can submit a new model if you train a new one, else submit the model we provided.)
pacman_target_net.pt	
hw3_109123456.pdf	Your report
Bonus_folder	Include GUI, (weight files, if you use DRL method) and python files

Learning Source

You can follow the order of the video links we provide to learn the basics of q-learning and DQN.

- 【機器學習2021】(中文版)： RL(一)、RL(二)、RL(三)
- Deep Reinforcement Learning, 2018: Lecture 3: Q-learning(Basic Idea)