1 Introduction

Reinforcement learning (RL) is a machine learning paradigm in which an agent learns the best actions to play through repeated experience. RL is at the heart of many cutting-edge AI game-playing techniques. For example, TDGammon and AlphaZero both used RL techniques to outperform human players (as well as other top computer algorithms) at backgammon, chess, go, and shogi, all of which are highly complicated games which computer scientists struggled to conquer before the advent of RL.

The purpose of this lab is to provide a brief introduction to reinforcement learning. RL algorithms abound; you will be implementing one of the most straightforward, called Q-learning. You will simulate this algorithm in two environments: the first is a repeated two-player, two-action game called Chicken, and the second is a market game. The goal of the first half of this lab is to stress the importance of representation, while in the second part, you will observe that Q-learners can learn collusive, rather than Nash equilibrium, behavior.

2 Setup

You can find the stencil code for Lab 3 here. Once everything this lab is set up correctly, you should have a project with files for nine Python classes:

- `basic_chicken_agent.py`
- `lastmove_chicken_ql.py`
- `lookback_chicken_ql.py`
- `my_agent.py`*
- `q_learning.py`*
- `i_fixed_policy.py`
- `uniform_policy.py`
- `chicken_env.py`
- `game.py`

The star annotations indicate which files you will edit during this lab. The others are purely support code and/or already-implemented opponent bots for the simulations.

Please be sure to read the `README.md`. Here is an abridged version of the setup guide described in there:

1. Clone the repository with the command
   `git clone https://github.com/brown-agt/lab-stencils.git`
2. Create a python virtual environment and activate it. Be sure to use `python 3.10` or higher.
3. Run `pip install -r requirements.txt`

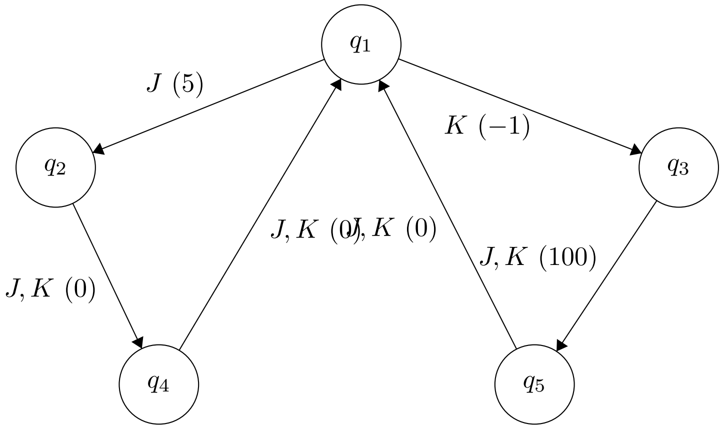
3 Markov Decision Processes

RL algorithms are designed to operate in environments called **Markov Decision Processes** (MDPs), which are characterized by:

1. S: the set of states
2. A: the set of actions
3. $T(s, a, s')$: a transition function describing the probability of going from state s to s' via action a
4. $R(s, a, s')$: the reward for taking action a at state s and ending up at state s'
5. an initial state, or a probability distribution over initial states

In general, the transition function could depend on *all* past states and actions. When it instead depends only on a trajectory of length $k \leq t$, i.e., when $T(s_0, a_0, \ldots, s_{t-1}, a_{t-1}, s_t) = T(s_{t-k}, a_{t-k}, \ldots, s_{t-1}, a_{t-1}, s_t)$, for all times t, it is called k-Markov. In particular, the transition function $T(s_0, a_0, \ldots, s_{t-1}, a_{t-1}, s_t) = T(s_{t-1}, a_{t-1}, s_t)$ is 1-Markov. Note also that the transition function of an MDP is usually also assumed to be stationary, which in the 1-Markov case means $T(s_t, a_t, s) = T(s_{t+r}, a_{t+r}, s)$, whenever $s_t = s_{t+r}$ and $a_t = a_{t+r}$, for all states $s_t, s_{t+r}, s \in S$, actions a_t, a_{t+r}, and times $r \geq t$.

Example: Here is an example of a Markov Decision Process:



1. $S = \{q_1, q_2, q_3, q_4, q_5\}$, with initial state q_1
2. $A = \{J, K\}$
3. T: All transitions are deterministic, and are represented in the diagram. (E.g., if you start at state q_1 and take action J, you will end up at state q_2.)
4. R: The rewards corresponding to each transition are depicted in parentheses, next to the relevant action. (E.g., $R(q_1, J, q_2) = 5$)

A **policy** in an MDP is a function $\pi : S \to A$ from states to actions. An **optimal policy** is one that selects an optimal action at all states. Try to figure out the optimal policy in this sample MDP, and then read on.

At state q_1, action J earns you an immediate reward of 5, while action K earns you an immediate reward of -1. However, after taking your first action and transitioning to the next state (either q_2 or q_3), you will earn 100 if you had chosen K, compared to 0, if you had chosen J. Therefore, the optimal choice at q_1 is K.

4 Part I: Q-Learning

In Part I of this lab, you will implement Q-learning.

4.1 The Algorithm

In this section, we introduce you to **Q-learning**, a classic reinforcement learning algorithm.

The goal of RL is to learn an optimal policy in an MDP from simulated experience, without prior knowledge of either the transitions or the rewards. In their most basic form, these algorithms learn a Q-table of dimension $|S| \times |A|$, which stores the long-term expected return of choosing action a in state s, for all $s \in S$ and $a \in A$.

The Q-table can be initialized arbitrarily.[1] Then each time the agent transitions from state s to s' via action

[1] Initializations closer to the expected long-term rewards lead to faster convergence.

a and earns reward r, the Q-table is updated.

Q-learning updates the Q-value at the current state-action pair (s, a) using this rule:

$$Q(s,a) = \alpha(r + \gamma \max_{a'} Q(s', a')) + (1 - \alpha)Q(s,a) \ .$$

Here, $\alpha \in [0, 1]$ is a **learning rate**, which controls how fast new information is incorporated into Q-values; and $\gamma \in [0, 1)$ is **discount factor**, which describes how much the agent cares about its present vs. its future rewards—a value close to 1 means it values future rewards nearly as much as present rewards; a value close to 0 (which is unusual) means it values future rewards much much less than present rewards.

Question: Starting from an initial Q-table of all zeroes, apply the Q-learning update rule to our sample MDP a few times to see if/how it learns the optimal policy. Assume an initial state of q_1, a learning rate of 0.5 and a discount factor of 0.9.

An RL algorithm can learn online or offline. To learn **offline** means to learn while simulating play, without actually earning any rewards. If an agent is learning offline, there are often distinct training and testing phases. During the former, the agent learns in simulated play, without accruing any rewards; during the latter, it "really" plays, meaning it does accrue rewards. If an agent is learning offline, it is free to explore its environment to its heart's content (i.e., to play **off-policy**), without risk: i.e., without foregoing any rewards, since its actions are simulated anyway.

To learn **online** means to learn while "really" playing: i.e., to learn and accrue rewards simultaneously. If an agent is learning online, it probably prefers to play **on-policy**, meaning according to its latest and greatest policy, so that it does not forego too many rewards. The trade-off between playing according to the latest and the greatest vs. favoring exploring as-yet-unexplored parts of the environment in the hopes of discovering an even better policy is called the **exploration–exploitation** trade-off.

To play well, given its current knowledge—to **exploit**—the agent can choose an action with the highest expected long-term rewards, as recorded in the current Q table. To continue learning—to **explore**—the agent can occasionally chooses an action at random, with **exploration rate** $\epsilon \in [0, 1]$.

The pseudocode in Table 1 outlines Q-learning algorithm in detail.

```
Q_LEARNING(MDP, $\gamma$)
  Inputs       discount factor $\gamma$, exploration policy
  Output       action-value function $Q^*$
  Initialize   $Q = 0$, $\alpha$ according to schedule

repeat
    1. initialize $s, a$
    2. while $s$ is non-terminal do
        (a) take action $a$
        (b) observe reward $r$, next state $s'$
        (c) $Q(s,a) = Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s,a)]$
        (d) choose action $a'$ according to the exploration policy
        (e) $s = s'$, $a = a'$
        (f) decay $\alpha$ according to schedule
forever
```

Table 1: Q-Learning: Off-policy reinforcement learning.

4.2 An Implementation

To implement Q-learning, you should navigate to `q_learning.py`. This class contains some important instance variables:

- `self.q` is your Q-table. It has dimensions `[self.num_possible_states][self.num_possible_actions]`.
- `self.num_possible_states` is the size of the state space.
- `self.num_possible_actions` is the size of the action space.
- `self.learning_rate` is the learning rate hyperparameter (α).
- `self.discount_factor` is the discount factor hyperparameter (γ).
- `self.exploration_rate` is the exploration rate hyperparameter (ϵ).
- `self.training_mode` is a boolean indicating whether your agent is in "training mode" or not. In training mode, your agent will play a exploration-exploitation strategy, but in testing or "competition" mode, your agent will play a learned policy based on its Q-table.
- `self.training_policy` is a fixed policy that your agent will play with probability `self.exploration_rate` when choosing the next action during exploration-exploitation. It is an object implementing the `IFixedPolicy` interface, which has one method, `get_move()`, that produces an action, given a state. By default, your agents will play uniformly randomly, but you are free to alter this policy, if you can envision a better one for training.
- `self.s` is your current state, representing the variable s in the Q-learning algorithm.
- `self.a` is your current action, representing the variable a in the Q-learning algorithm.

Task: Implement the following methods in `q_learning.py`:

- `update_rule()`
- `choose_next_move()`

(Look for `TODO` in the code.)

When this task is complete, you will have implemented a Q-learning agent. But, the success or failure of a reinforcement learning agent is determined by more than the algorithm alone; it depends greatly on the state-space representation. If you represent a repeated game as an MDP in a smart way, you'll see great results, but otherwise Q-learning won't be effective, no matter how many rounds you train for. In what follows, you will explore a few state-space representations so you can observe this phenomenon yourself.

5 Part II: State-Space Representations

One of the key challenges in reinforcement learning is the design of an effective state-space representation. Take the game of chess, for example. There are too many valid configurations of pieces on the board to represent each one as its own state.[2] Consequently, we usually resort to a feature representation, which summarizes the information about a state that is most relevant to decision making. In chess, features might include the total value of black pieces, the total value of white pieces, etc.

Similarly, in order to implement RL agents that play repeated normal-form games, the first step is to morph the game into an MDP by choosing a state-space representation. For example, your agent can choose as the state its last action, its opponents' last action, both agents' last actions, its own past two actions and its opponents' past four actions, etc. This choice can be difficult, because the history in a repeated game is in general unbounded, while MDPs with finite state (and action) spaces are usually much easier to solve.

In Part II of this lab, you will experiment with several different state-space representations for Q-learning agents that play a simple repeated game called Chicken.

[2] Claude Shannon famously estimated that there are at least 10^{120} valid board configurations in chess.

5.1 The Game of Chicken

Chicken, like the Prisoners' Dilemma, is a symmetric two-player, two-action, non-zero-sum game.

The premise is that two daredevil stuntmen are trying to impress a casting director in order to be chosen for *Fast and Furious 12*. The two stuntmen are driving in opposite directions on the road and are about to collide, head-on. Each has the option to **Swerve** or **Continue** going straight. If both players continue, they will crash, and receive a massive negative payoff in the form of injuries. If they both swerve, neither is rewarded with the part, as the casting director is left unimpressed. But if one swerves and one continues, the swerving player loses face, while the player who continued is rewarded handsomely.

Chicken is defined by the following payoff matrix:

	S	**C**
S	0, 0	-1, 1
C	1, -1	-5, -5

The only way to win is to continue while the other player swerves. But are you willing to take the risk?

5.2 State-Space Representations of Chicken

A Q-learning agent is incomplete without a state-space representation. Indeed one more method is needed in your implementation, namely `determine_state()`, in which you will define your state-space representation. This method can use all the information available to the agent about the past rounds of the game. to determine the MDP state the game is currently in.

We have implemented two sample state-space representations for you. Your next task is to create two sample Q-learning Chicken agents by combining your Q-learning code and our state-space representations. This exercise will serve to simultaneously verify the correctness of your implementation, and to illustrate the importance of a robust state-space representation. After training our sample agents and observing their performance, you will design your own agent for Chicken, meaning your own state-space representation for this game, and then you will train it and test its performance. How well can your agent do?

You will be training agents that use our sample state-space representations against a *very* simple agent located in `basic_chicken_agent.py`. It simply plays actions $[1, 0, 0]$ in sequence, repeatedly. Can Q-learning pick up on this pattern and play a perfect response? As you have probably surmised by now, the answer depends on the state-space representation!

5.2.1 `lastmove_chicken_ql.py`: An Insufficient State Space

Navigate to `lastmove_chicken_ql.py`. Here, you will find a basic Chicken-playing Q-learning agent, with a state space of size 2, with each state corresponding only to the opponent's last action. If the opponent played 0, the state is 0, while if the opponent played 1, the state is 1.

Task: Run this agent. Doing so will launch a 20,000-round training phase against the aforementioned `BasicChickenAgent`. Then, having learned its policy, it will turn off "training mode" and play an additional 300 "real" rounds against the opponent. Finally, it will print out the results: your agent's average reward per round, and a list of your agent's per-round rewards. If you implemented Q-learning correctly, the agent should have achieved an average reward of about 0, alternating between rewards of 0, 1, and -1.

Although the agent plays imperfectly, it does learn to avoid the `(CONTINUE, CONTINUE)` outcome—a reward of -5.0 should not occur. Thus, it avoids a large negative reward. So it is doing *something* right, but it wasn't quite able to learn the entirety of the other agent's strategy. How can we improve its performance?

Note: If your implementation works correctly, you will notice that your agent performs poorly during the training phase, but plays better during the testing phase, once it begins playing its learned policy. This is because Q-learning is training off-policy. You should bear this in mind later, when we contrast off-policy Q-learning with the on-policy SARSA algorithm.

5.2.2 `lookback_chicken_ql.py`: A Sufficient State Space

Navigate to `lookback_chicken_ql.py`. You will notice a very similar setup in this agent, the only salient difference being `determine_state()`. States are still represented as integers, but each state now represents a combination of the opponent's past *two* actions. Is this enough information to predict the agent's next action, and generate a best response?

Task: Run this agent. It will do the same thing as the last agent, printing out the results at the end. You will be able to see a difference here; your agent should have learned to play perfectly, averaging about 1/3 reward per round and alternating between rewards of 1, -1, and 1 most of the time.

As you can see, even a small difference in the state space can lead to a large difference in performance between our two sample agents, all of which is highly dependent on the behavior of the opponent.

Question: Why was the second state-space representation sufficient to produce a best response to the opponent's strategy, whereas the first one was not? In other words, why was it sufficient for your agent to know this opponent's last 2 actions in order to play perfectly, while knowing ust 1 action was not enough?

5.2.3 Implementing your Own State-Space Representation

Navigate to `my_agent.py`.

Task: Your next task is to devise your own state-space representation of the game of Chicken as an MDP, in order to maximize your reward against a *mystery* agent. Your opponent's strategy this time will be more intricate than the strategies faced by the sample agents, so you will have to design a more robust representation, in order to account for its possible behaviors.

Your representation can be basic and low-level, incorporating only the previous (k) state(s) and action(s) of your opponent. However, this representation explodes exponentially, so is impractical. You might do better to summarize the history of the game to determine your state, including your own or your opponent's past action, your past states, and your past rewards, into a feature vector representation of the state.

N.B. One of the reasons for the massive success of **deep** RL is its ability to automate feature extraction.

Important: After you choose a state-space representation, be sure to edit the `NUM_POSSIBLE_STATES` constant to reflect the number of states in your representation. The `determine_state()` method should then return a value in `[0, NUM_POSSIBLE_STATES)`.

Run your agent. It will do the same thing as the previous simulations, though it will face the more complicated "mystery" opponent rather than the one with the basic alternating strategy.

How did you do? *If you don't do well, do not despair!* It can be very difficult to achieve positive reward against an unknown opponent in Chicken, so achieving even a small negative reward is itself impressive. You should aim for at least about -0.05, on average across all 300 rounds.

Try out a few different ideas, but feel free to move on when you feel your agent is performing as well as it can, in light of the time constraints imposed by the lab.

5.2.4 Discussion: Strategy Considerations

Please read this section, and discuss its implications with your partner.

Since the success of your strategy hinges on your state-space/MDP representation of a repeated normal-form game, the key to creating a successful agent will be a good choice of what information to incorporate into your states. For example, what your opponent played during the last round may be more relevant than what they played 10 rounds ago. But by including too many of their past actions, your agent might pick up on a spurious pattern in their behavior, whereas by including too few actions, you agent might *fail* to pick up on an actual pattern (see `LastMoveChicken`, for example). Additionally, including more actions causes the state space to grow, which requires longer training times.

There are plenty of further strategic considerations. When multiple agents are learning simultaneously in a repeated game, their behavior is inherently correlated. So an important additional consideration is whether your opponent's actions seem to depend on *your* previous actions, or whether they seem to be acting independently. Once again, the more robust your state-space representation, meaning the more relevant past behaviors—both yours and your opponent's—it can represent, the better your performance should be.

6 Class Competition

Having now implemented more advanced learning strategies, we will once again play a total of 100 rounds of the game of Chicken—50 rounds as player 1 and 50 rounds as player 2. However, this time, instead of competing against TA bots, your bots will face off against those of other students.

You are free to use any strategy you like in this competition, whether inspired by reinforcement learning or otherwise. However, to keep the competition engaging, your strategy should *not* be constant, as that would obviate any need for strategic play.

To further encourage adaptive approaches, we will also be introducing four unique, exploitable strategies into the competition!

7 Part III: A Market Game

In this third and final part of the lab, you will again experiment with Q-learning agents, but this time by simulating a market game.

The market game involves n (discrete) sellers, each selling a unique product, and a continuum of buyers. Seller i's product is characterized by a quality factor a_i. A further outside option (i.e., a good on offer outside this market) is characterized by a_0. Each seller chooses a price p_i from a set of discrete prices, and the demand q_i for seller i's product depends on *all* the sellers' prices via the following logit demand equation:[3]

$$q_i = \frac{\exp\left(\frac{a_i - p_i}{\mu}\right)}{\sum_{j=1}^{n} \exp\left(\frac{a_j - p_j}{\mu}\right) + \exp\left(\frac{a_0}{\mu}\right)} \tag{1}$$

[3] Here, μ is a horizontal differentiation parameter, meant to capture product characteristics other than quality.

Each seller's goal is to set its prices so as to maximize its profits π_i, in light of its competition, where $\pi_i = q_i(p_i - c)$, for some cost $c > 0$.[4]

With this market game in mind, the simulation proceeds as follows: at each time step,

1. Each seller i posts a price p_i from a set of m discrete prices.
2. Each seller then earns a profit based on its deterministic demand q_i (Equation 1).
3. The sellers all observe one another's prices, but there is no direct communication among them.

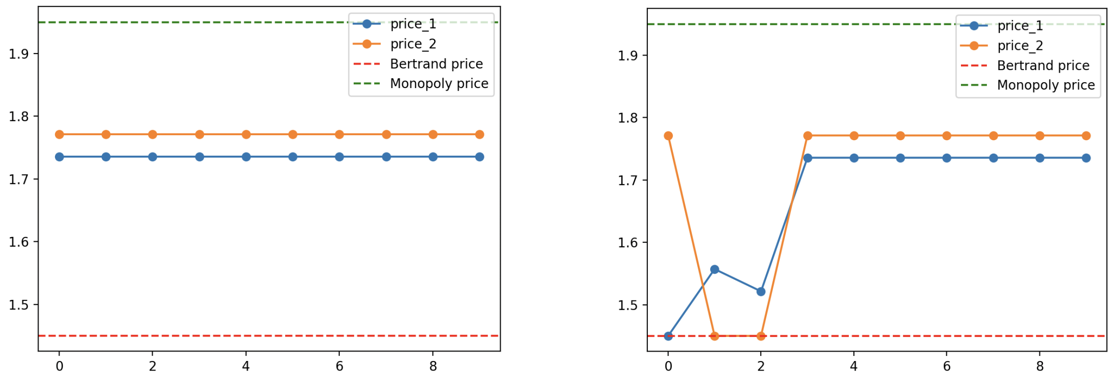
There are two prices of note in this market game, the Bertrand equilibrium price,[5] and the monopoly price. The **Bertrand equilibrium price** is the cost c, as all sellers compete to drive the price down as low as possible. In contrast, the **monopoly price** is the price that would be charged in a monopoly, when only one seller controls the entire market.

One key question is whether reinforcement learning agents, such as Q-learners, learn equilibrium behavior in this and other market games. You will investigate this question presently assuming exactly two sellers.

Task 1. In the file `collusion.py`, implement the `determine_state` function. Experiment with various choices for the state-space representation. You can try no state at all, just your agent's past price, just the opponent's past price, or both of your past prices. (We do not recommend you go much bigger than this here in lab, as it may take a while for the agents' behavior to converge, if it even converges at all!)

When you run your code, it should output plots like those depicted in Figure 1, which demonstrate collusion. Figure 1 shows the outcome of 1 million iterations of two Q-learners using the default parameters.

In Figure 1a, the prices are above the Bertrand equilibrium level, while Figure 1b demonstrates another characteristic of collusion: If the first seller deviates from the cartel price to attract more demand, the second seller follows suit, leading to a price war, before the two resume their collusive behavior.



(a) Sellers' prices.

(b) Price war after seller 1's deviation.

Figure 1: Outcome of two Q-learners after 1 million iterations.

Task 2. Your next task is to investigate the robustness of the collusive behavior you just observed by experimenting with various parameter settings. Try adjusting `self.learning_rate`, `self.discount_factor`, the exploration rate `self.beta` (which decays exponentially transitioning the policy from off- to on-policy), `self.max_steps` (the number of time steps), and the parameters of the market game itself (e.g., μ and c).

Feel free to play around with the parameter settings, namely the learning rate `alpha`, the exploration rate `beta`, which decays exponentially transitioning the policy from off- to on-policy, `self.max.steps`, the number of iterations, `mu`, etc.

[4] For simplicity, we assume all sellers incur the same cost.

[5] A model of competition between two sellers competing on price is named for François Bertrand.