Project Two

Reinforcement Learning

Question 1: Markov Decision Process [15 marks]

The shortest path (SSP) problem is a classical problem in graph theory. The problem is typically represented on a undirected graph $G = (V, E)$, where V is a set of vertices (nodes) and E is a set of edges (arcs) connecting the vertices. Each edge $(u, v) \in E$ has a non-negative weight (cost) $c(u, v)$. The goal is to find a policy that minimizes the total cost of traveling from a given starting node $s \in V$ to a target node $q \in V$.

Provide a Markov decision process (MDP) model of the SSP problem. In your MDP model, clearly define and briefly describe the state space (including the initial and terminal states, action space (including applicable actions in any state), rewards, transition probabilities, and the discount factor.

A Markov Decision Process (MDP) model defining the Shortest Path Problem (SSP) would have a state space, an action space, transition probabilities, rewards, a discount factor, and a goal of determining the optimal policy π^* that maximises the agent's total expected reward (and consequently, the shortest possible path from node ***s*** to node ***q***). To model an SSP we make several assumptions:

- The environment is static,
- The agent has complete observability ($O_t = S_t^a = S_t^e$ that is, the model adheres to the basis of a an MDP),
- Transitions between states are deterministic (that is, selecting ***a*** in ***s*** always moves the agent to $\boldsymbol{s'}$ without randomness),
- Edge weights (costs) ***c(u,v)*** are static and known, and
- Rewards are static and known

The state space $\{\boldsymbol{S}\} = \{\boldsymbol{s_0}, \boldsymbol{s_1}, \dots, \boldsymbol{s_n}, \boldsymbol{q}\}$ consists of all nodes in graph ***V*** for the undirected graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$. Each state $\boldsymbol{s} \in \boldsymbol{S}$ corresponds to a node in the graph. The initial state $\boldsymbol{s_0}$ is the starting node $\boldsymbol{s_1}$ and the target state is terminal node ***q***.

For each state ***s***, the action space $\boldsymbol{A}(\boldsymbol{s})$ includes moving to any adjacent node ***v*** where there exists an edge $(\boldsymbol{s}, \boldsymbol{v})$ in ***E***. Actions are defined by the graph's adjacency list, which reflects the possible transitions from one node to its neighbors.

The transitions are deterministic. If action ***a*** is performed in state ***s*** to move the agent to adjacent node ***v***, then the transition probability $\boldsymbol{P}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s'})$ = 1 for $\boldsymbol{s'} = \boldsymbol{v}$ and 0 for all other states ($\boldsymbol{s'} \neq \boldsymbol{v}$).

The immediate reward $\boldsymbol{R}(\boldsymbol{s}, \boldsymbol{a}, \boldsymbol{s'})$ for transitioning from state ***s*** to state $\boldsymbol{s'}$ following action ***a*** is the non-negative weight of the edge. We can either swap the maximization for minimization, or more simply, we can define the cost following action ***a*** as $-\boldsymbol{c}(\boldsymbol{s}, \boldsymbol{s'})$. Because all costs are positive, the negative sign ensures that maximising the cummulative reward is equivalent to minimising the total path cost.

The discount factor $\boldsymbol{\gamma}$ for future rewards is set to 1 (far-sighted evaluation) because the objective is to minimise the total cost from the start node to the target node without devaluing future path costs.

A deterministic optimal policy always exists for an MDP. Optimal policy π^* will provide the direction to move from any node ***s*** to the target node ***q*** via the shortest path. This policy can be found using dynamic (iterative) programming for value iteration or policy iteration, employing the Bellman Optimality equation:

$$v^*(s) \leftarrow max_{a \in A(s)} \left(R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v^*(s') \right)$$

Once provided a policy π for our MDP, in the case of iterative policy evaluation, we can use an iterative application of the Bellman expectation backup (synchronous) to calculate the updated value for every non-terminal state.

To use dynamic programming to solve the SSP (MDP), we would need to consider two subproblems: prediction, to determine the value function v_π under the current policy, and control, to determine the optimal value function and the optimal policy for MDP $< S, A, P, R, \gamma >$.

The goal would be to iterate on the value function using a greedy policy (for now, defined as a monotonic, deterministic policy that forcibly follows the best action with respect to the updated value function), running episodes (i.e., iterating) until the process inevitably converges on the optimal value function and policy (*see Figure 1, below*). Mathematically, that is – given a policy π,

(1) Evaluate the policy π: $v_\pi = E[R_{t+1} + \gamma R_{t+2} + \cdots | S_t = s]$
(2) Improve the policy by acting greedily on v_π: $\pi' = greedy(v_\pi)$
(3) Iterate until convergence

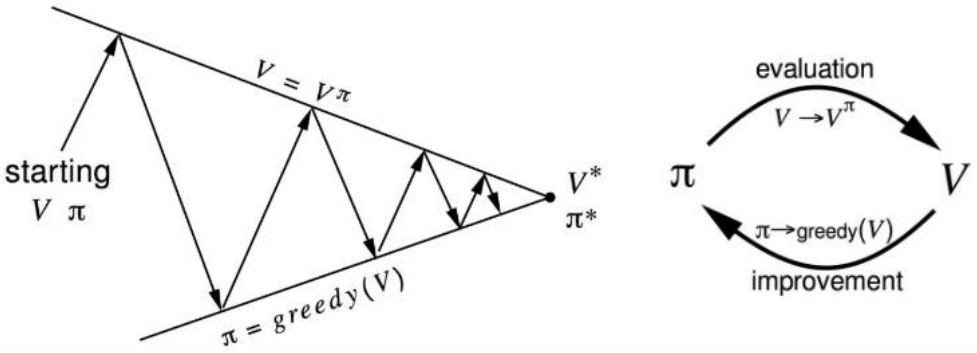


Figure 1: How to Improve a Policy (Chen, 2024)

Question 2: Negative Reward Markov Decision Process [15 marks]

Consider a finite MDP with bounded rewards, where all rewards are negative. That is, $R_t < 0$ always. Let the discount factor $\gamma = 1$. Assume that the MDP is finite horizon (i.e., every state-transition sequence ends in finite steps to a terminal state). Assume further that the initial state S_0 is identical across all state-transition sequences. The MDP also has a *deterministic transition function* (i.e., given any current state S and any action A performed in the current state, the next state can always be uniquely determined). Let $H = (S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{L-1}, A_{L-1}, R_L)$ be a state-transition sequence generated by following a deterministic policy π. Prove that the sequence $V^\pi(S_0), V^\pi(S_1), \ldots, V^\pi(S_{L-1})$ is strictly increasing.

Given:

- A bound MDP with all negative rewards and a finite horizon;
- A deterministic system dictates that R_{t+1} and S_{t+1} is predetermined with the action $\boldsymbol{a} = \boldsymbol{\pi}(\boldsymbol{s})$ for every $\boldsymbol{s}$; that is, $\boldsymbol{a}$ is predictable and fixed whenever the system is in state $\boldsymbol{s}$;
- The state transition sequence generated by following a deterministic policy is: $H = (S_0, A_0, R_1, S_1, A_1, R_2, \dots, S_{L-1}, A_{L-1}, R_L)$;
- The discount factor γ is set to 1.

Goal: Prove that the sequence $V^\pi(S_0), V^\pi(S_1), \dots, V^\pi(S_{L-1})$ is strictly increasing.

1. **First Step:**

   For S_{L-1}, the final step before the terminal state, the value function can be defined as $V^\pi(S_{L-1}) = R\big(S_{L-1}, \pi(S_{L-1})\big)$, since $V^\pi(S_L) = 0$ at the terminal state.
   Further, $R\big(S_{L-1}, \pi(S_{L-1})\big) \leq 0$ under the reward constraints (bounds) of the MDP $(R_t < 0\ always)$.

2. **General Case (S_t):**

   From the Bellman Expectation Equation $v_\pi(s) = E^\pi[R_{t+1} + \gamma V^\pi(S_{t+1}|S_t = s)]$, otherwise expressed as $V^\pi(S_t) = R\big(S_t, \pi(S_t)\big) + \gamma V^\pi(S_{t+1})$, we define:

   - $V^\pi(S_t)$ as the expected value from state S_t following policy π,
   - $R\big(S_t, \pi(S_t)\big)$ as the immediate reward from state S_t following policy π, since deterministic state transitions make action A_t implicit, and
   - $\gamma V^\pi(S_{t+1})$ as $V^\pi(S_{t+1})$, where a discount factor of 1 denotes the absence of time preference

   Because $R\big(S_t, \pi(S_t)\big) \leq 0$ and $V^\pi(S_{t+1})$ is the value of the next state under the deterministic policy, the sequence can be analysed for each backwards step from $L-1$ to 0.

   For each state S_t, $V^\pi(S_t) = R\big(S_t, \pi(S_t)\big) + V^\pi(S_{t+1})$, all rewards $R\big(S_t, \pi(S_t)\big)$ are non-positive (≤ 0) and the progression of the value function from one state to the next involves the accumulation of increasingly negative (or zero) values. To prove this, the value from initial state S_0 can be expressed recursively to S_L:

   - From S_0 to S_1: $V^\pi(S_0) = R\big(S_0, \pi(S_0)\big) + V^\pi(S_1)$,
   - For S_1: $V^\pi(S_1) = R\big(S_1, \pi(S_1)\big) + V^\pi(S_2)$, and so on until we reach S_{L-1} (and S_L which is subsequently 0)
   - Each $V^\pi(S_t)$ can then be substituted back into the previous state's equation, which cascades the sum of the rewards plus S_L (0).
   - The final generalized expression for $V^\pi(S_0)$ is therefore $V^\pi(S_0) = \sum_{t=0}^{L-1} R\big(S_t, \pi(S_t)\big) + V^\pi(S_L)$

   Thus, if "strictly increasing" refers to negative accrual, then by inductive reasoning we can say that the sequence $V^\pi(S_0), V^\pi(S_1), \dots, V^\pi(S_{L-1})$ strictly increases in the negative direction as we approach S_L from S_0.

   However, the meaning of "strictly increasing" remains slightly ambiguous. If "increasing" were referring to increasing the performance of the *policy to in-turn improve the value function,* rather than "increasing" the *values for a single sequence of state transitions*, we would need to use a greedy policy to positively increase (i.e., reduce the "negativeness" of) $V^\pi(S_0)$ over multiple episodes. But for a single sequence within a bound MDP that contains all negative rewards over a finite horizon, $V^\pi(S_0), V^\pi(S_1), \dots, V^\pi(S_{L-1})$ can only *strictly become increasingly negative*.

Question 3: Model Free Reinforcement Learning [20 marks]

Consider an unknown MDP with three states (A, B, C) and two actions (L, R). Suppose a reinforcement learning agent chooses actions according to the *uniform random policy* π in the unknown MDP. Meanwhile, the discount factor $\gamma = 0.9$. The current estimation of the Q-function table associated with policy π is given below.

State	Action	Q
A	L	0.5
A	R	1.0
B	L	1.5
B	R	2.0
C	L	2.5
C	R	3.0

Assume that the agent obtained a new state transition sample $(s_t = A, a_t = L, s_{t+1} = B, r_{t+1} = 10)$ at time $t + 1$. Determine the new Q-function table after performing a single iteration of TD(0) update based on the newly obtained transition sample. Assume that the learning rate $\alpha = 0.1$. In your answer, clearly present the updating rule of TD(0) and explain how this rule is used to update the Q-function table. Additionally, construct a new policy based on the updated Q-function table by using the ϵ-greedy policy improvement strategy, assuming that $\epsilon = 0.1$.

Temporal difference (TD) learning is more common and efficient than Monte Carlo (MC) reinforcement learning because TD methods can learn directly from episodes of experiences. Like MC methods, they are model-free with no knowledge of transition models and reward functions. Unlike MC, they use state transitions to bootstrap their learning. Further, they learn policies from value functions (i.e., Bellman Expectation Equation), as observed in dynamic programming. The goal of a TD method is to learn v_π online from experience under policy π, where 'online' describes as a model capable of simultaneously sampling and learning a ***v***-function. The simplest TD learning algorithm is TD(0).

In TD(0), we update value $v(s_t)$ toward estimated return $R_{t+1} + \gamma v(s_{t+1})$, also known as the TD target (derived from the Bellman Expectation Equation); this target represents our new estimation. For the second step, we define the TD error (δ_t) as $R_{t+1} + \gamma v(s_{t+1}) - v(s_t)$, yielding the complete TD(0) equation: $v(s_t) \leftarrow v(s_t) + \alpha\big(R_{t+1} + \gamma v(s_{t+1}) - v(s_t)\big)$. In this example, we are presented with the transition sample (*Table 1, below*):

Current state s_t	A
Action taken a_t	L
Reward received r_{t+1}	10
Next state s_{t+1}	B
Discount factor γ	0.9
Learning rate α	0.1

Table 1: Question 3, State Transition Sample

Applying the TD(0) principles to a Q-learning update:

$Q(s_t, a_t) \leftarrow Q\big(s_t, a_t\big) + \alpha\big[r_{t+1} + \gamma max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big]$, where

- $Q(s_t, a_t)$ is the current estimate of the Q-value for being in state s_t and taking action a_t,
- α is the learning rate that scales the TD error to moderate the Q-value,
- γ is the discount factor,

- $r_{t+1} + max_a Q(s_{t+1}, a_{t+1})$ remains the TD target, for which:
  - r_{t+1} is the reward received by executing action a_t in state s_t, transitioning to the state s_{t+1} and
  - max_a reflects the selection of maximally achievable Q from s_{t+1} over all possible actions a. *This is what defines Q-learning as an **off-policy learner**; it **learns the value of the optimal policy independently of the agent's actions**.
- $r_{t+1} + max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$ remains the TD error, representing the difference between the estimated Q-value and the actual returned value (plus the best estimated future returns).

Altogether, the update rule adjusts the Q-value $Q(s_t, a_t)$ towards the sum of the reward received for the current action r_{t+1} and the discounted maximum future reward, which is equivalent to incrementally adjusting its estimate towards what it computes as the optimal returns. In this way, the Q-update rule can allow the agent to learn the optimal policy by continuously refining the estimated expected rewards for each action taken in each state.

Applying this to the task at hand, we can update $Q(A, L)$ using this formula:

$Q(s, a) \leftarrow Q(s, a) + \alpha[r_{t+1} + \gamma max_{a'} Q(s', a') - Q(s, a)]$

From the table,
$max_{a'} Q(B, a') = max(1.5, 2.0) = 2.0$

Substituting into the update rule,
$Q(A, L) \leftarrow 0.5 + 0.1(10 + 0.9 * 2 - 0.5)$
$Q(A, L) \leftarrow 0.5 + 0.1(10 + 1.8 - 0.5)$
$Q(A, L) \leftarrow 0.5 + 0.1 * 11.3$
$Q(A, L) \leftarrow 0.5 + 1.13$
$Q(A, L) \leftarrow 1.63$

State	Action	Q
A	L	**1.63**
A	R	1.0
B	L	1.5
B	R	2.0
C	L	2.5
C	R	3.0

Table 2: Question 3, Updated Q-Function Table

Next, we construct a new policy based on the Updated Q-Function table *(Table 2, above)* using the ϵ-greedy policy improvement strategy (assuming $\epsilon = 0.1$). To do this, we select the greedy action (i.e., the action that will yield the highest Q-value) with probability $1 - \epsilon$ (0.9) and any other action with probability ϵ (0.1), resulting in a 90% chance of choosing the greedy action and a 10% chance to choose any other action in the state.

The final representation of the ϵ-greedy policy for each state would be:

Policy π:

- $\boldsymbol{\pi(A) = \{L: 0.9, R: 0.1\}}$
- $\boldsymbol{\pi(B) = \{L: 0.1, R: 0.9\}}$
- $\boldsymbol{\pi(C) = \{L: 0.1, R: 0.9\}}$

State A:
- Best *A*: L (1.63)
- ***A*** **=** $L: 90\%, R: 10\%$ *probability*

State B:
- Best *A*: R (2.0)
- $A = L: 10\%, R: 90\%$ *probability*

State C:
- Best *A*: R (3.0)
- $A = L: 10\%, R: 90\%$ *probability*

Question 4: Value Function Approximation [20 marks]

Consider an MDP $M = (S, A, R, P, \gamma)$. Assume that a state feature function $x(\cdot)$ is given to transform every state $s \in S$ to a feature vector with n dimensions, as shown below.

$$x(s) = \begin{pmatrix} x_1(s) \\ x_2(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

Furthermore, consider an approximated value function below as a weighted linear combination of state features:

$$\hat{V}(s, w) = \sum_{i=1}^{n} x_i(s) w_i$$

Design a stochastic gradient descent algorithm for training w to minimize the loss below:

$$\mathrm{E}_\pi[(v_\pi(s) - \hat{V}(s, w))^2]$$

where policy π refers to any given policy under evaluation. We assume that the true value function of policy π, i.e. $v_\pi(s)$, is known in advance. In your answer, clearly specify how the stochastic gradient with respect to w is derived from the loss. You also need to present the algorithm pseudo-code and discuss in detail how and why the proposed algorithm works. No coding and experimentation of your new algorithm is required for this question.

Design a stochastic gradient descent (SGD) algorithm that trains a weight vector (***w***) to minimize the mean squared error (MSE) between the true value function $v_\pi(s)$ and the approximated value function $\hat{\boldsymbol{V}}(\boldsymbol{s}, \boldsymbol{w})$ for a given policy π, where the model is expressed as a weighted linear combination of state features (commonly used in neural networks).

- Each state ***s*** represented as a feature vector $x(s) = \big(x_1(s), x_2(s), \dots, x_n(s)\big)^T$
- Approximated value function $\hat{\boldsymbol{V}}(\boldsymbol{s}, \boldsymbol{w})$ is linear combination of state features weighted by ***w*** (as above)
- Loss function to minimise is the expected square error between true and approximated value

In using SGD, we are seeking a vector weight w that minimises $L(w)$. To achieve this, we need the gradient of L with respect to w, which requires us to differentiate the loss function (gradient calculation), then we update w (update rule).

Gradient Calculation:

$$\nabla_w L(w) = -2E_\pi \left[\Big(v_\pi(s) - \hat{V}(s, w)\Big)\, x(s)\right]$$

The expectation can be approximated empirically by sampling states according to policy π (Kleijn, 2024).

Update Rule:

The SGD method updates weights directly based on the gradient of the loss L function with respect to the weights w, calculated from the linear relationship between approximated value function $\hat{V}(s,w)$ and the features of the state $x(s)$.

Given a learning rate α:

$$w \leftarrow w - \alpha \nabla_w L(w)$$

This translates to the **practical** update rule:

$$w \leftarrow w + 2\alpha \left(v_\pi(s) - \hat{V}(s,w)\right) x(s)$$

***Remember for later:* Does not include backpropagation, as seen in neural networks.**

> *Unlike NN's, this is a direct calculation and application of gradients to update the weights, without layered backwards flow of errors that seen in backpropagation.*

Pseudocode:

1	Initialize w arbitrarily (e.g., w = 0 or small random values, He or Xavier/Glorot are good)
2	For each episode or until convergence:
3	Sample a state *s* from the environment using policy π
4	Compute the target value $v_\pi(s)$
5	Compute the approximated value $\hat{V}(s,w)$
6	Compute the gradient: grad = 2 * $(v_\pi(s) - \hat{V}(s,w))$ * $x(s)$
7	Update weights: w = w + α * grad

In this way, SGD can be used to iteratively adjust w in the direction that *most* reduces the discrepancy between the predicted values and the true values under policy π. By sampling states according to π, the gradient estimate is aligned with the distribution of states encountered under the actual policy.

Question 5: Understand and Experimentally Study a Reinforcement Learning Algorithm [30 marks]

Q-learning is a model-free reinforcement learning algorithm used to find the optimal action-selection policy for a finite MDP. It aims to learn the value of state-action pairs, known as the optimal Q-values, which indicate the expected return of taking a given action in a given state and following the optimal policy thereafter. To do so, we store all the Q-values in a table that we will update at each time step using the Q-learning updating rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

where:

- $Q(s,a)$ is the current Q-value of taking action a in state s,
- α is the learning rate $(0 < \alpha < 1)$,
- r is the reward received after taking action a in state s,
- γ is the discount factor $(0 < \gamma \leq 1)$,
- s' is the next state after taking action a,
- $\max_{a'} Q(s',a')$ is the maximum Q-value of the next state s' over all possible actions a'.

Once the learning of the optimal Q-values converges, the optimal policy π^* is derived by choosing the action with the highest Q-value in each state:

$$\pi^*(s) = \arg\max_{a} Q(s,a)$$

In this question, you need to study the Q-learning algorithm both theoretically and experimentally. You need to conduct the specific tasks below and report your answers and findings.

1. With the help of an algorithm pseudo-code, describe how the Q-learning algorithm works, assuming that the Q-values being learned are kept in a table known as the Q-table. Your algorithm description should cover several key steps, including initialization (highlight the structure and initialization of the Q-table, the initialization of algorithm parameters etc.), the action selection strategy (i.e., the ϵ-greedy strategy), state-transition sampling, Q-value update, as well as the criteria for terminating the learning process.
2. Understand and describe the FrozenLake-v0 environment as a benchmark reinforcement learning problem. You can find more information about this benchmark problem from here (`https://www.gymlibrary.dev/environments/toy_text/frozen_lake/`). In your description, clearly define the state space, action space, state-transition function, reward function, and the stopping criteria.

3. Implement the Q-learning algorithm to solve the FrozenLake-v0 problem. You can follow the example implementation given online at:

   `https://towardsdatascience.com/q-learning-algorithm-from-explanation-to-implementation-cdbeda2ea187`

   to implement your own version of the Q-learning algorithm. **IMPORTANT**: you must submit your implementation source code as part of this question.

4. Experimentally evaluate the Q-learning algorithm on the FrozenLake-v0 problem and report your findings:

   - Run the Q-learning algorithm on the FrozenLake-v0 environment with the following parameters: learning rate $\alpha = 0.1$, discount factor $\gamma = 1.0$, exploration rate ϵ starts at 1.0 and decays over time, and 10,000 episodes.
   - Draw the learning performance curve of Q-learning with the horizontal axis representing the training episodes and the vertical axis representing the average performance (i.e., the average total reward during testing) of the learned Q-table. You must perform 5 independent runs (each run must use a different random seed) of the Q-learning algorithm on the FrozenLake-v0 environment and report the average learning performance in the performance curve.
   - Summarize the key findings from your experiments. In particular, discuss whether Q-learning is effective at solving the FrozenLake-v0 problem. Discuss how the exploration rate ϵ decays over time during the learning process and the impact of the exploration rate ϵ on the performance of the Q-learning algorithm.

1. Q-Learning Overview:

The theory behind Q-learning has been discussed in question 3, above. For this question, we will focus on the functionality of the Q-learning algorithm and its constituent parts.

We begin with the initialisation step, where Q-values for all state-action pairs are initialised using zeros or small random values to facilitate learning.

The learning rate, alpha ($\boldsymbol{\alpha}$), determines how much new information overrides historical information. A smaller α-value leads to more conservative updates.

The discount factor γ is applied to future rewards. It is used to balance the total value with respect to immediate and future rewards. A higher γ values future rewards more favorably.

The ϵ-greedy action selection strategy is used to encourage (mostly) greedy actions while introducing some randomness for new exploration. The ϵ-greedy action is assigned the value $1 - \epsilon$, where ϵ is the probability assigned to taking a random action in the selected exploration.

Based on the Bellman equation, the Q-values are updated using:

$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$, where

- $Q(s_t, a_t)$ is the current estimate of the Q-value for being in state s_t and taking action a_t,
- α is the learning rate that scales the TD error to moderate the Q-value,
- γ is the discount factor,
- $r_{t+1} + max_a Q(s_{t+1}, a_{t+1})$ is the TD target:
- $r_{t+1} + max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$ is the TD error
- (further details above in the answer to question 3).

As the problem states, once the learning of the optimal Q-values converges, the optimal policy π^* is derived by choosing the action with the highest Q-value in each state:

$$\pi^*(s) = argmax_a Q(s, a)$$

Pseudocode:

1	Initialize Q-table such that *Q(s, a)* is 0 for all *s* in *States*, *a* in *Actions(s)*
2	For each episode (repeat):
3	Initialize s
4	Repeat (for each step of episode):
5	Choose a from s using policy derived from Q-table (e.g., epsilon-greedy)
6	Take action a, observe r, s'
7	Q-table[s, a] <- Q-table[s, a] + alpha * (r + gamma * max(Q-table[s', :]) – Q-table[s, a])
8	s <- s'
9	until s is terminal

2. The Frozen Lake (FrozenLake-v1) Environment:

FrozenLake-v1 is a standard testing environment in OpenAI Gym. The agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. The agent's goal is to reach the goal tile without falling into the water (OpenAI Gym Library, 2022).

- **State Space (S)**: The state is determined by the agent's position in the grid.
- **Action space (A):** There are four discrete actions that an agent can make at each step[Left, Right, Up, Down].
- **Observation space (S_e):** The environment has 16 discrete, fully observable tiles.
- **State Transition Probability (P):** P represents a stochastic transition probability where the *intended* move is chosen by the action, but the *actual* movement may differ because of the "slippery ice" in the environment, leading to stochastic transitions between states.
- **Terminal state (q):** The goal state.
- **Reward (R):** The reward is 1 if the agent reaches the goal and 0 otherwise (hole or frozen).
- **Stopping criteria:** An episode ends when the agent reaches the goal or falls into the water.

This problem is an example of an **MDP**.

3. Initial Run of the Q-learning algorithm: *CODE LINKS IN APPENDIX*

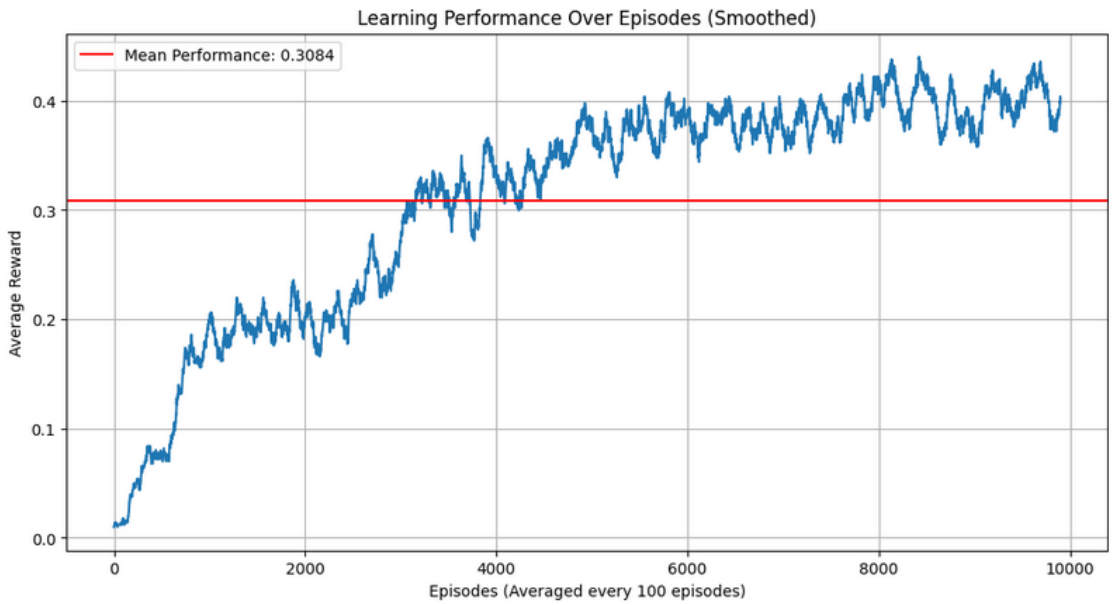
Results of running the Q-learning algorithm on the FrozenLake-v1 environment with given parameters (Amine, 2020).

Mean Reward / Thousand Episodes		
Episode	**Mean Episode R**	**Parameters**
1,000	0.043	$\alpha = 0.1$
2,000	0.014	$\gamma = 1.0$
3,000	0.024	$\epsilon_{start} = 1$
4,000	0.029	$episodes = 10,000$
5,000	0.011	
6,000	0.021	
7,000	0.015	
8,000	0.006	
9,000	0.016	
10,000	0.016	

4. Complete Experiment: *CODE LINKS IN APPENDIX*

4.1 Running the Q-learning algorithm under the specified conditions (see table above).
4.2 Performance curve of the Q-learned table after 5-independent runs:

Independent Run Statistics (N = 5)



Average Learning Performance:	0.3084
Standard Deviation:	0.1907
Median Performance:	0.4000
Minimum Reward Achieved:	0.0000
Maximum Reward Achieved:	0.6000

4.3 Summary and discussion:

There is a significant rise in average reward during the initial learning phase, indicating effective learning and policy improvement. Performance stabilizes with a mean reward of 0.3084, indicating convergence. Despite convergence, the rewards display variability (SD = 0.1907) and fluctuation between a minimum reward of 0.0 and a maximum of 0.6 – likely a consequence of the stochasticity of FrozenLake-v1.

The exploration rate decay is set to allow a gradual shift from exploration to exploitation. As specified by the problem, the exploration rate (ϵ) starts at 1, promoting initial exploration of the state-action space. As ϵ decays, the agent increasingly exploits its accumulated knowledge (i.e., explores less and "acts more greedily"), which correlates with a rise in performance. The exploration rate's exponential decay ensures that the agent does not stop exploring entirely, even in later episodes.

Overall, Q-learning appears to be an effective approach for the FrozenLake-v1 environment, as demonstrated by consistent improvement and eventual stabilisation in the rewards. The algorithm's learning and improvement over time is reflected in its increasing performance (smoothed by moving average) on the curve until it converges, thus supporting its suitability for this type of discrete, stochastic problem.

Appendix

Link to code (GitHub)

Url: https://github.com/Marianette/A2-AIML431

Link to code (Google Collab)

Url: https://colab.research.google.com/drive/1mEyAIGzvC0zm6prEc4ldf2QSU5oUtVUB?usp=sharing

References

Amine, A. (2020). *Q-Learning Algorithm: From Explanation to Implementation*. Medium. https://towardsdatascience.com/q-learning-algorithm-from-explanation-to-implementation-cdbeda2ea187

Chen, A. (2024). *AIML431: Dynamic Programming Lecture Notes*. Victoria University of Wellington. https://ecs.wgtn.ac.nz/foswiki/pub/Courses/AIML431_2024T2/LectureSchedule/dp.pdf

Kleijn, B. (2024). Basic Deep Learning Regressor & Gradient Descent and Context: AIML425 [Lecture 3 & 4 Notes]. *Master of Artificial Intelligence Year One, Victoria University of Wellington*. https://ecs.wgtn.ac.nz/foswiki/pub/Courses/AIML425_2024T2/LectureSchedule/DLintro.pdf

OpenAI Gym Library. (2022). FrozenLake-v0. In *Gym Documentation*. https://www.gymlibrary.dev/environments/toy_text/frozen_lake/#version-history