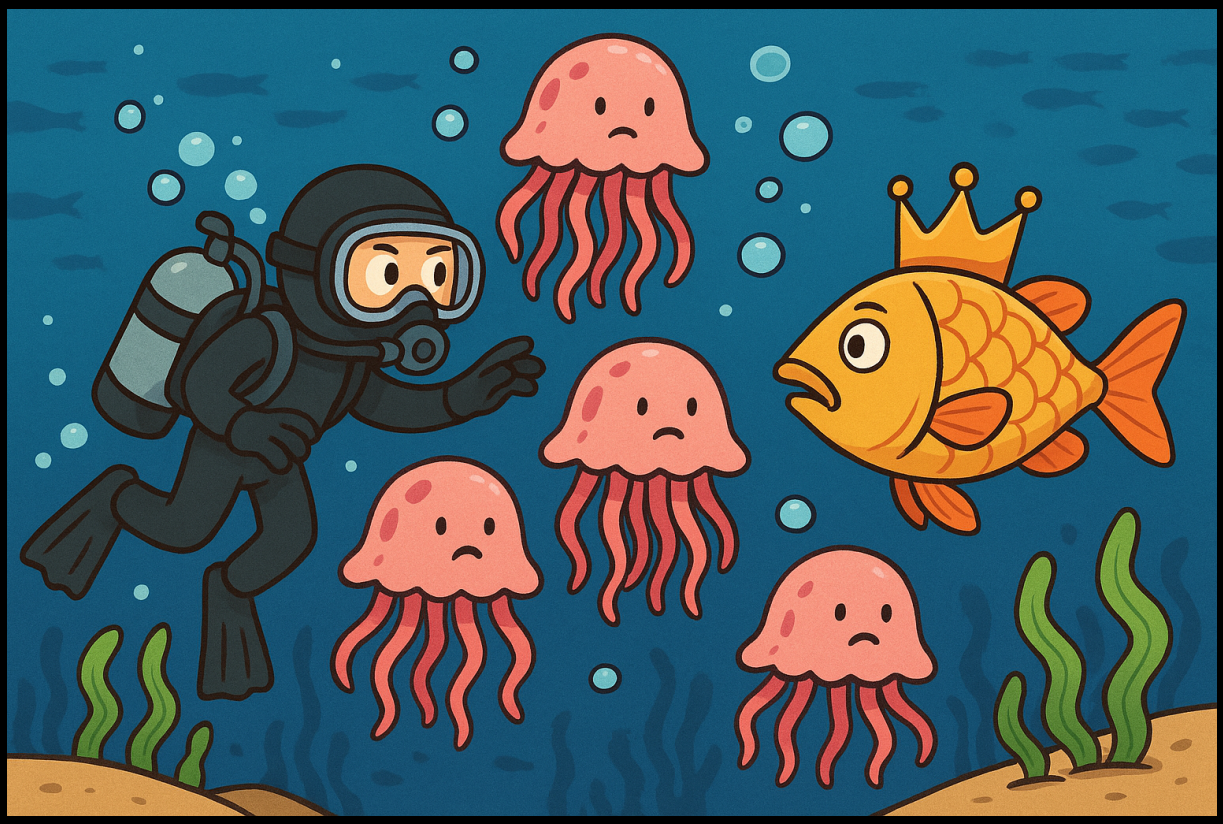

Assignment Reinforcement Learning
The KTH FishingDerbyRL

Ahmet Enis Isik, Francesco Baldassarre
aeisik@kth.se, frabal@kth.se

September 30, 2025

Contents

1 Introduction

In this project a progressively stronger agent for the FishingDerbyRL assignment was implemented by following the exercies. In exercise 1 a random baseline was established and a frequency based policy was produced. In exercise 2 a tabular Q-learning was implemented that initializes a Q-table taking valid actions only, updating with the Bellman equation and introducing an early stopping criterion based o Q-table convergence. In exercise 3, goal directed behavior was demostrated through reward shaping using different parameters implemented in student_3_2_* files; 1st avoids all fish and the 2nd reaches the king by the shortest safe path. In exercise 4, exploration scheduling was introduced (constant vs. linearly annealed epsilon-greedy). The main goal was to find a good trade off to solve the *Exploration vs. Exploitation* dilemma. Finally, in exercise 5 the implementation of a strategy to escape sub-optimal policies was required. By collecting all the best results from the previous exercises, an optimization process has been followed to guarantee that the produced policy is always optimal.

2 Exercise 1:

2.1 Key Design Decisions

In the first phase, the design of the random player controller was shaped to understand the code environment and getting started with the idea of policy selection. In particular, in every episodes we simply generated a random action ("right", "left", "up" or "down") among the available ones from the current state. While doing that, we kept count of the total occurrences of each random action in each state. At the end, the most picked random action across all episodes were selected to be inserted in the final policy returned by our player.

2.2 Code Explanation

2.2.1 Main Components

- **init_states() / init_actions() / allowed_movements():** build state $\leftrightarrow$ index maps, define 4 moves, and compute legal actions per state.
- **player_loop_random():** run episodes, sample random legal actions and update the count matrix n.

```
Initialize n[|States|, |Actions|] <- 0
for episode in 1..E:
    s <- initial_state()
    while not end_episode:
        action <- Random(allowed_moves[s])
        n[s,action] <- n[s,action] + 1

        sender({"action": action_to_str(action), "exploration": True})
        (r, s', end_episode) <- receiver()
        s <- s'

# Deterministic policy from counts
pi[s] = argmax_a n[s,action]  -> return {"policy": policy_to_strs(pi), "exploration":
    False}
```

2.2.2 Logic Flow

1. Receive initial bootstrap
2. For each step choose a uniform random legal action
3. Count actions
4. At the end, produce $\pi(s) = \arg\max_a$ counts.

3 Exercise 2:

3.1 Key Design Decisions

In this phase, we implemented **tabular Q-learning** with a deterministic, greedy action selection over the legal action set and a simple **mean-difference convergence check** between successive Q-tables. Concretely:

- (i) we initialized $Q \in R^{|S|\times|A|}$ with random values in $[0, 1]$ to break ties;
- (ii) at each step we selected $\arg\max_a Q(s, a)$ restricted to allowed actions;
- (iii) we applied the standard **Bellman update** with fixed α and γ;
- (iv) at the end of each episode we computed diff $= |\text{mean}(Q - Q_{\text{old}})|$ as a **stopping proxy**. No ϵ-greedy was used here yet

3.2 Code Explanation

3.2.1 Main Components

- **allowed_movements():** precomputes the legal action set per state.
- **q_learning():** runs episodes, selects greedy actions over legal moves, updates Q with Bellman rule, tracks convergence via mean absolute difference.
- **get_policy():** extracts deterministic policy $\pi(s) = \arg\max_a Q(s, a)$.

3.2.2 Pseudocode

```
def q_learning():

    # Random init in [0,1] of the Q-table
    Q = Uniform(0,1, shape=(ns, na))

    while not converged:

        while not end_episode:

            legal = allowed_moves[s_current]

            # Select action with the highest Q values (greedy)
            action = argmax_over_legal(Q[s_current, :], legal)

            receive_rewards

            # Bellman update
            Q[s_current, action] = Q[s_current, action] \
                + lr * (R + discount * max_a Q[s_next, a] - Q[s_current, action])


        # Convergence proxy: mean absolute diff between
        diff = abs(mean_over_entries(Q - Q_old))

    return Q
```

3.2.3 Logic Flow

1. Initialize Q randomly; precompute allowed actions per state.
2. For each step: select greedy action over legal moves; apply TD update.
3. After each episode: compute mean absolute Q-difference; stop when small.
4. Return Q and derive π via arg max.

4 Exercise 3:

4.1 Key Design Decisions

Here we explored **hyperparameter settings and reward shaping** via two separate configuration files, keeping the Q-learning core unchanged. The intent was to study how α, γ, ϵ schedule, and per-event rewards affect convergence speed and policy optimality:

- **student_3_2_1.py:** very harsh penalties (such as strong negatives for jellyfish and a small step cost), conservative $\alpha = 0.15$, highly farsighted $\gamma = 0.99$, ϵ annealed to 0.05.
- **student_3_2_2.py:** generous positive reward for the goal, milder penalties, more aggressive $\alpha = 0.5$ to speed learning, and **shorter-horizon** $\gamma = 0.85$ to favor shorter paths.

We did *not* change the algorithmic logic here, only the **reward vector** and **(α, γ, ϵ) settings** to quantify their behavioral impact.

4.2 Code Explanation

4.2.1 Main Components

- **student_3_2_1.py** and **student_3_2_2.py**: define `rewards`, `alpha`, `gamma`, `epsilon_initial/final`, `annealing_timesteps`, `threshold`.
- **player_3.py**: same Q-learning loop as Exercise 2, now parameterized by the selected config.

4.2.2 Pseudocode

```
# student_3_2_1.py
rewards = [-1000.0, -100.0, -100.0, ..., -0.10]
alpha   = 0.15
gamma   = 0.99
epsilon_initial = 1.0
epsilon_final   = 0.05
annealing_timesteps = 10000
threshold = 1e-6

#student_3_2_2.py
rewards = [1000.0, -50.0, -50.0, ..., -2.0]
alpha   = 0.5
gamma   = 0.85
epsilon_initial = 1.0
epsilon_final   = 0.01
annealing_timesteps = 8000
threshold = 1e-6
```

4.2.3 Logic Flow

1. Load the configs; plug α, γ, ϵ and reward vector into the existing Q-learning loop.
2. Train until the mean-diff threshold is met (as in Exercise 2).
3. Compare stability/speed and path optimality across the two settings.

5 Exercise 4:

5.1 Key Design Decisions

We introduced **exploration** with an ϵ-greedy policy over **legal actions only**, implemented in two variants:

1. **Constant** ϵ (Exercise 4.1): fixed exploration probability.
2. **Linear annealing** ϵ_t (Exercise 4.2): a **ScheduleLinear** from $\epsilon_{\text{initial}}$ to ϵ_{final} over a prescribed number of steps.

This is the first place where we blend exploration with the Q-learning loop; the rest of the algorithm remains as in Exercise 2.

5.2 Code Explanation

5.2.1 Main Components

- **epsilon_greedy(...)**: returns an action given Q, state index, legal actions, and exploration schedule (constant or linear).
- **ScheduleLinear**: maps `current_total_steps` to ϵ_t in $[\epsilon_{\text{final}}, \epsilon_{\text{initial}}]$.

5.2.2 Pseudocode

```
def epsilon_greedy(...):

    if eps_type == "constant":

        if rand() < epsilon:
            # Exploration
            return UniformChoice(legal_actions)

        else:
            # Exploitation
            return argmax_over_legal(Q[state, :], legal_actions)


    elif eps_type == "linear":
        sched = ScheduleLinear()
        epsilon_t = sched.value(current_total_steps)

        if rand() < epsilon_t:
            # Exploration
            return UniformChoice(legal_actions)

        else:
            # Exploitation
            return argmax_over_legal(Q[state, :], legal_actions)
```

5.2.3 Logic Flow

1. At each step, compute ϵ (constant or linearly annealed).
2. With prob. ϵ pick a uniform legal action; otherwise pick the legal $\arg\max_a Q(s, a)$.
3. Apply the usual Q-update afterward.

6 Exercise 5:

6.1 Key Design Decisions

This was the most delicate part, focused on **escaping sub-optimal policies** and making training **robust** under unknown scenarios:

- **Early stopping via reward moving average**: we tracked a `rewards_window` (moving average over a fixed window) and **stopped early** when the windowed average stabilized within a tolerance. This complements the convergence method from Ex. 2 and prevents wasting episodes when returns have stabilized.
- **Always return the best policy**: independently of how/when we stop, we stored and returned the **best policy so far** (which is the episode that brought the highest rewards R_{total}) to avoid finishing with a suboptimal policy.
- **Annealed learning rate** α_t: by analogy with ϵ annealing, we introduced a **linear schedule** α_t from α_{initial} to α_{final} over the same annealing horizon, improving late-stage stability.
- **Hyperparameter tuning**: we tuned ϵ-schedule, $(\alpha_{\text{initial}}, \alpha_{\text{final}})$, γ, the reward window size and convergence tolerance, and a hard cap on episodes/steps to meet the assignment's constraints.

6.2 Code Explanation

6.2.1 Main Components

- **epsilon_greedy(..., eps_type="linear")**: same as Ex. 4 but used consistently here to escape local optima.
- **anneal_alpha(...)**: linear annealing for the learning rate.
- **q_learning():** integrates: (i) reward moving-average stopping; (ii) best-so-far Q/policy tracking; (iii) α_t and ϵ_t schedules; (iv) episode/step budget constraint.

6.2.2 Pseudocode (core snippets)

```
function anneal_alpha(a0,a1,step,T):
    return a1 if step>=T else a0 + (a1-a0)*(step/T)

function q_learning():
    init Q(ns,na); Q_old = Q.copy()
    rewards_window = Deque(maxlen=W);

    while episode <= MAX_EP and |mean(Q-Q_old)| > tol and not window_converged:
        s = init_state(); current_reward = 0; end = False

        while episode not end

            action = epsilon_greedy(linear)
            lr = anneal_alpha(linear)

            receive_rewrd
            update_Q

        rewards_window.append(current_reward)

        if len(rewards_window) == max_window_lwngth

            if (highest_reward_in_window -window_avg) <= AVG_TOL:
                window_converged = True

            if current_reward == highst_reward_in_window:
                store_best_policy

    return best_Q
```

6.2.3 Logic Flow

1. Use **linear** ϵ_t to encourage early exploration and late exploitation.
2. Use **linear** α_t to reduce step variance as learning progresses.
3. After each episode, update a **reward moving average** and stop when stable.
4. Keep a **best-so-far** (Q, π) and return it, regardless of stopping reason.
5. Respect the **episode/step budget** prescribed by the assignment.

7 Individual Reflection

7.1 Ahmet Enis Isik

My personal main contribution was implementing the tabular Q-learning backbone (Exercise 2), running a structured hyperparameter and reward-shaping study (Exercise 3). For the learner, I set up the legal-action masking, the Bellman update and a mean-difference convergence check to make the training loop both stable and time-bounded. On top of that, I designed two contrasting configurations for reward shaping and $(\alpha, \gamma, \epsilon)$: the first to strongly discourage all collisions and wandering, the second to push the agent toward the King via the shortest safe path. This let us observe clear behavioral changes without altering the algorithm itself.

From this work, I learned how sensitive tabular Q-learning is to the interaction between reward scales, learning rate and discount factor. Small changes in γ can flip the agent from conservative long-horizon hedging to assertive short-horizon goal seeking. I also saw how essential it is to tie exploration to *legal* actions only and to use a simple, robust convergence proxy, otherwise you burn episode budget without improving the policy. The hyperparameter sweeps gave me practical intuition on stability vs. speed: aggressive α accelerates early gains but needs either annealing or a careful ceiling to avoid oscillations.

What I found challenging was stabilizing learning under sparse/peaky rewards and avoiding silent failures. Early on, forgetting to mask illegal actions polluted arg max with NaNs and caused misleading updates. Another challenge was picking a convergence threshold that actually reflects policy improvement: too tight and you never stop, too loose

and you freeze a still-noisy table. Balancing these constraints under a wall-clock budget was the core engineering lesson of this part.

7.2 Francesco Baldassarre

I think this project helped me with a better understanding of the subject precisely because it forced me to connect simple ideas to a messier setting. In Exercise 1, building a random baseline and extracting a frequency policy looked trivial at first, yet it was the right way to familiarize myself with the notion of state, the legal action set and the mechanics of the environment messages. Seeing a deterministic policy emerge from pure counts made the later improvements feel grounded rather than magical.
In Exercise 4, I focused on the scheduling strategy: constant ϵ versus a linearly annealed schedule, always restricted to legal moves. The most difficult part for me was choosing an ϵ schedule that does not either "freeze" the agent too early or keep it oscillate forever. I tied the schedule to a global step counter instead of episodes so that decay reflects actual experience. That small detail made the behavior much more predictable. Integrating the policy with the Q-learning loop from the earlier exercise without leaking features from later tasks, also sharpened my sense for clean interfaces between components.
My learning experience is mainly about the practical side of the exploration-exploitation dilemma. A good schedule feels like a "contract": you promise the gent wide curiosity up front and then you gradually demand decisiveness. I also appreciated how reproducibility depends on seemingly minor choices (seeds, action masking, tie-breaking).

8 Conclusion

In this project we implemented a progression from a purely random baseline to a tabular Q-learning agent capable of producing reliable policies for the FishingDerbyRL environment. Starting with a frequency-based policy (Exercise 1), we then introduced a legal-action–aware Q-table with Bellman updates and a mean-difference convergence proxy (Exercise 2). We studied how reward shaping and $(\alpha, \gamma, \epsilon)$ choices steer behavior without modifying the learning rule itself (Exercise 3), integrated principled exploration via constant and linearly annealed ϵ tied to a global step counter (Exercise 4), and finally made the training loop robust with a moving-average early stop, best-policy tracking, and an annealed learning rate (Exercise 5).
The key challenge throughout was balancing *exploration vs. exploitation* under sparse and sometimes peaky rewards while keeping learning both stable and sample-efficient. We addressed this by:

- (i) masking illegal actions to ensure well-defined $\arg\max$ and updates
- (ii) using a simple, robust convergence signal on Q
- (iii) decaying ϵ with actual interaction steps rather than episodes
- (iv) annealing α to reduce late-stage variance
- (v) enforcing an early-stopping criterion on windowed returns so episode budget is not wasted once performance plateaus.

Overall, the project strengthened our understanding of practical reinforcement learning: how reward scales, discounting, and exploration schedules interact; why seemingly minor details (legal-action masking, tie-breaking, seeding) drive reproducibility; and how simple mechanisms such as Bellman updates, scheduled exploration, and disciplined stopping rules, compose into a competent agent. The incremental design across Exercises 1–5 linked course theory directly to implementation, yielding policies that are both interpretable (via shaped rewards) and robust (via convergence checks and best-so-far safeguards).