Crawling Robot

Andrea Roncoli, Bianca Ziliotto

May 2025

Abstract

This work is our submission for Project 11 for the course *Neural Networks and Deep Learning*: "Use Q-learning to train an agent to control a crawling robot that has to learn how to move a 2-link finger to move forward. A positive reward is generated when moving forward, and a negative reward is generated when moving backward or not moving."

1 Physical System

1.1 Simulation environment

All simulations were conducted using MuJoCo (Multi-Joint dynamics with Contact). MuJoCo is a physics engine designed for high-performance simulation of articulated structures with contacts: it enables accurate and efficient modeling of rigid body dynamics, supporting advanced features such as soft constraints, contact dynamics, and detailed sensor feedback. It is well-suited for simulating biomechanics, robotics, and reinforcement learning environments.

The MuJoCo solver was configured with:

- Time step of 2 ms (0.002 s) for numerical integration;
- Projected Gauss-Seidel (PGS) solver with 50 iterations per time step to ensure stable contact resolution.

Gravity is set to standard Earth gravity ($-9.81 m/s^2$) along the Z-axis.

1.2 Agent Physical Model

The agent consists of a two-link articulated mechanism anchored to a free-floating base, as shown in Figure 1. The base and links are connected through hinge joints, allowing for planar motion in a vertical plane. The system is designed to simulate interactions with the environment, including collisions and contact forces.

- **Base**: The main body is a cube with low friction coefficients to enable sliding on the ground;

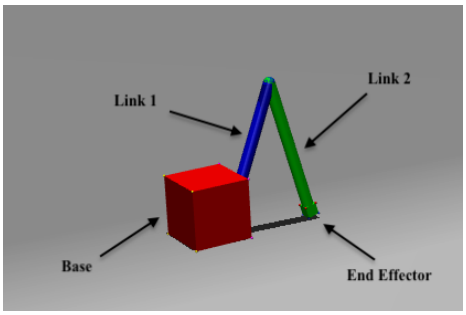

Figure 1: Agent Physical Configuration

- **Link 1**: A capsule-shaped link connected to the center of one of the faces of the base via a hinge joint (`joint1`);
- **Link 2**: A second capsule attached to the distal end of Link 1 via a second hinge joint (`joint2`);
- **End Effector**: A small cube fixed to the distal end of Link 2 with high friction coefficients allowing the system to anchor itself and generate movement.

Each joint is actuated by a torque motor; this is the only way the agent acts on its physical system.

1.2.1 Contact Modeling

The simulation limits interpenetration by modeling the following physical contacts:

- A plane-type ground geometry, which is set as the floor the agent lies on, with proper friction parameters and contact forces allowing a realistic ground interaction;
- Contact pairs between the base and both links, to avoid links penetrating the base.

2 Learning

Reinforcement Learning (RL) provides a natural framework for enabling agents to discover motor skills autonomously through interaction with their environment. In this project, we aim to teach a simple robot to crawl, moving in a single direction. The crawler is treated as an RL agent that learns by trial-and-error in simulation: at every control cycle (step), it receives a multidimensional observation of its state, chooses a discrete torque command for its two hinge

joints, and receives a scalar reward that encourages forward progression while penalizing inefficient or unstable motions.

The entire perception–decision–action loop is captured by a Deep Q-Learning (DQN) algorithm with experience replay and a target network; an optional Double DQN variant alleviates over-estimation bias. This section details the three pillars that make the learning pipeline work in practice: the *observations* fed to the network, the *action space* offered to the agent, and the concrete *Q-Learning* implementation used to train the agent.

2.1 Observations

The observation vector is designed to supply the agent with all proprioceptive information necessary to infer its state, while at the same time excluding the absolute x and y translation of the free-floating base to avoid *state aliasing*, favoring policies that generalize to unseen start positions. Listing 1 sketches the code that constructs the observation vector. The dimension of the vector is given in brackets, and more information about each entry is detailed in the paragraphs below. Ultimately, the observation vector is a vector $s \in \mathcal{S}$, where $|\mathcal{S}| = 40$.

Listing 1: Observation construction (`compute_observations()`).

```
obs = np.concatenate([
    base_z,           # [1]  base height
    base_quat,        # [4]  orientation (w, x, y, z)
    joint_qpos,       # [2]  joint angles q
    base_vel,         # [3]  linear velocity (body frame)
    base_ang_vel,     # [3]  angular velocity (body frame)
    joint_qvel,       # [2]  joint angular rates q'
    sensor_data,      # [23] raw MJCF sensors
    previous_action   # [2]  previous (discretized) action
])
```

Kinematic State. The base *position* is represented by its height z only, while *orientation* is encoded as a unit quaternion (q_w, q_x, q_y, q_z). The six-tuple $(v_x, v_y, v_z, \omega_x, \omega_y, \omega_z)$ reports translational and rotational *velocity* in the base frame. Removing the absolute planar position (x, y) guarantees that a policy learned in a limited corridor length does not over-fit to a particular start location and can be deployed on an arbitrarily long track.

Joint Configuration. Both hinge joints (shoulder and "knee") contribute their current angle q and angular rate $\dot{q}$.

System Sensors. The XML snippet below defines the 19 MuJoCo sensors included in the simulation:

```
<sensor>
    <frameangvel ... objname="link1"/>
    <frameangvel ... objname="link2"/>
    <touch name="floor_touch_1"  site="corner1"/>
    ...
    <touch name="floor_touch_8"  site="corner8"/>
    <touch name="floor_touch_knee" site="knee"/>
    <touch name="floor_touch_1_b"  site="b_corner1"/>
    ...
    <touch name="floor_touch_8_b"  site="b_corner8"/>
</sensor>
```

- Each `frameangvel` sensor outputs a 3-vector of angular velocity expressed in the local link frame; with two links this contributes $2 \times 3 = 6$ scalars.
- The 17 `touch` sensors measure normal contact force magnitude at strategic points on the end effector (`floor_touch_*`), knee (`floor_touch_knee`), and base (`floor_touch_*_b`); each produces a single scalar, adding another 17 components.

Hence, the total sensor data is $n_{\text{sensor\_data}} = 23$ dimensional.

Previous action. To provide the policy with a lightweight memory of recent commands, we append an encoding of the action taken at the preceding control step. For our crawler, this single feature conveys just enough information about phase and short-term torque history to stabilise its cyclic gait, while avoiding the computational overhead of adding recurrence to the network.

2.2 Action Space

The agent controls the crawler through two independent torque motors, actuating the shoulder (`joint1`) and knee (`joint2`) hinges respectively. We discretize each actuator into n equally spaced levels in the interval $[-\tau_{\max}, \tau_{\max}]$ and treat the joint command at time t as the Cartesian product of these per-joint bins.

Formally, the discrete action set is

$$\mathcal{A} = \left\{ (\tau_i^{(1)},\ \tau_j^{(2)}) \;\middle|\; \tau_i^{(k)} = -\tau_{\max} + i\,\Delta\tau,\ i = 0, \ldots, n-1,\ k \in \{1, 2\} \right\}, \quad (1)$$

where $\Delta\tau = 2\tau_{\max}/(n-1)$.

Because both joints share the same discretisation, the cardinality of the joint action space grows quadratically:

$$|\mathcal{A}| = n^2$$

A typical choice in our experiments is $n = 5$, producing 25 distinct torque pairs. This simple discretisation is large enough for the agent to move, and small enough to make the Q-learning process feasible.

2.3 Q-Learning

At every simulation step, the agent observes a state $s_t \in \mathcal{S}$, selects an action $a_t \in \mathcal{A}$, transitions to s_{t+1}, and receives a reward r_t. The goal is to maximise the expected discounted return $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$. We set the discount factor to $\gamma = 0.99$. We approximate the optimal state–action value function $Q^*(s, a)$ with a neural network $Q_\theta(s, a)$ trained to satisfy the Bellman optimality principle:

$$Q^*(s_t, a_t) = \mathbb{E}\big[\, r_t + \gamma \max_{a'} Q^*(s_{t+1}, a') \big]. \tag{2}$$

2.3.1 Network architecture

The Q-network is implemented in `utils/qnet.py` and follows a straightforward multi-layer perceptron (MLP) design:

$$s \xrightarrow{\text{Linear } |\mathcal{S}| \mapsto h_1} \text{LayerNorm} \xrightarrow{\text{ReLU}} \cdots \xrightarrow{\text{Linear } h_L \to |\mathcal{A}|} Q_\theta(s, \cdot).$$

LayerNorm after each fully connected layer stabilises training by normalising the pre-activation distribution across a mini-batch, which is especially useful when concatenating heterogeneous inputs such as contact forces and angles.

Both the *online* network Q_θ and a periodically updated *target* network Q_{θ^-} coexist. Every $k_{\text{target}} = 10^5$ environment steps, the weights of the target network are replaced by the current online parameters.

2.3.2 ε-greedy exploration

Given the output of the network $Q_\theta(s, \cdot)$, action selection employs an ε-greedy strategy:

$$a_t = \begin{cases} \text{Uniform}(\mathcal{A}), & \text{w.p. } \varepsilon_t; \\ \arg\max_a Q_\theta(s_t, a), & \text{w.p. } 1 - \varepsilon_t. \end{cases}$$

The exploration rate decays over time, updating every 100 optimization steps, starting from $\varepsilon_{\max}$ down to $\varepsilon_{\min}$, both hyperparameters of the training routine:

$$\begin{cases} \varepsilon_0 = \varepsilon_{\max}; \\ \varepsilon = \max\big(\varepsilon_{\min},\, \varepsilon \cdot 0.995\big) \end{cases}$$

2.3.3 Reward shaping

The total reward r_t is the sum of two components:

$$r_t = r_{\text{pos}}(t) + \lambda_{\text{upr}}\, r_{\text{upright}}(t),$$

- $r_{\mathbf{pos}}$: a *position reward* based on the signed forward displacement of the base during the last step, $\Delta x = x_{t+1} - x_t$. Positive values encourage progress along the world $+x$ axis.
- $r_{\mathbf{upright}}$: an *upright reward* that encourages the top of the base of the crawler to keep facing upwards, discouraging flips.

2.3.4 Termination Conditions

Episodes last a minimum of 2s and a maximum of 200s of simulated time (10^5 steps at $\Delta t = 2\,\text{ms}$). Termination within this range happens if the average distance elapsed per step falls beneath a given threshold (0.1mm per time step).

2.3.5 Experience replay buffer

Following standard DQN practice, every transition $(s_t, a_t, r_t, s_{t+1}, \text{done})$ is stored in a cyclic buffer of capacity $N_{\text{buffer}} = 5 \times 10^5$. Mini-batches of $B = 32$ samples are drawn uniformly during training. This randomization breaks temporal correlation between observations, improves data efficiency, and allows us to reuse past experience multiple times. To initially fill the buffer, we collect 10^5 transitions with a random policy before the first optimization step (*warm-up period*).

2.3.6 Target calculation: DQN vs. Double DQN

The TD target used in the mean-squared error loss depends on the algorithm variant:

Vanilla DQN.

$$y_t = r_t + \gamma \max_{a'} Q_{\theta^-}(s_{t+1}, a')$$

Double DQN.

$$\begin{cases} a^* = \arg\max_{a'} Q_\theta(s_{t+1}, a') \\ y_t = r_t + \gamma\, Q_{\theta^-}(s_{t+1}, a^*) \end{cases}$$

Decoupling the *selection* and *evaluation* of the greedy action in Double DQN reduces the systematic over-estimation bias observed in the single-network update rule of vanilla DQN.

2.3.7 Optimization loop

Algorithm 1 outlines the overall training procedure implemented in `agent.py`. Gradient updates use the Adam optimiser.

Algorithm 1 Deep Q-Learning with experience replay and target network.

Initialise replay buffer $\mathcal{D}$ and networks Q_θ, Q_{θ^-} with identical weights, step counter $c = 0$
for episode $= 1$ to $E_{\max}$ **do**
  Reset simulation $\rightarrow s_0$, set $t \leftarrow 0$
  while not *done* **do**
    With probability ε_t sample random $a_t \sim \mathcal{U}(\mathcal{A})$
    Otherwise $a_t \leftarrow \arg\max_a Q_\theta(s_t, a)$
    Execute a_t, observe r_t, s_{t+1}, *done*
    Store transition $(s_t, a_t, r_t, s_{t+1}, \textit{done})$ in $\mathcal{D}$
    if $|\mathcal{D}| \geq K_{\text{warmup}}$ **and** $c \bmod k_{\text{update}} == 0$ **then**
      for $i = 1$ to N_{train} **do**
        Sample mini-batch $\mathcal{B}$ from $\mathcal{D}$
        Compute targets y (Double or Vanilla scheme)
        Minimise $L = \frac{1}{B} \sum_{\mathcal{B}} \big(Q_\theta(s, a) - y\big)^2$
      end for
    end if
    if $c \bmod k_{\text{target}} = 0$ **then**
      $\theta^- \leftarrow \theta$ ▷ Target sync
    end if
    $\varepsilon \leftarrow \max(\varepsilon_{\min}, \varepsilon \cdot \lambda_\varepsilon)$
    $s_t \leftarrow s_{t+1}$, $t \leftarrow t + 1$, $c \leftarrow c + 1$
  end while
end for

3 Experiments

In this section, we show the results of three experiments within the previously described framework. After training an agent to move forward (Section 3.1), we train it to crawl backwards, testing whether some of the learned patterns can be transferred from one task to the other to facilitate the learning process (Section 3.2).

3.1 Crawling forward

The first experiment consists in training the agent to move forward, covering the largest possible distance within each episode, starting from random initializations of joint angles. The learning curves showing the episode distance and duration are shown in Figures 2 and 3. After ≈ 30 million simulation steps (roughly 8 hours wall-time on a single RTX 4090) the policy converges to a stable gait that propels the crawler at an average speed of $\approx 0.5\,\mathrm{m\,s^{-1}}$.

Other than qualitative visual evaluation (provided by a video in the GitHub repo), we record the joint trajectories and we give an example of the first seconds of motion in a random episode (Figure 4). As we can see, the learned movement varies periodically with a gait cycle duration of ≈ 0.4s.

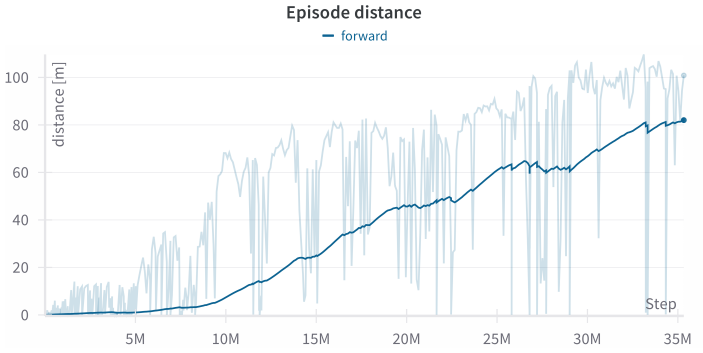

Figure 2: Learning curve (Time-weighted Exponential Moving Average).

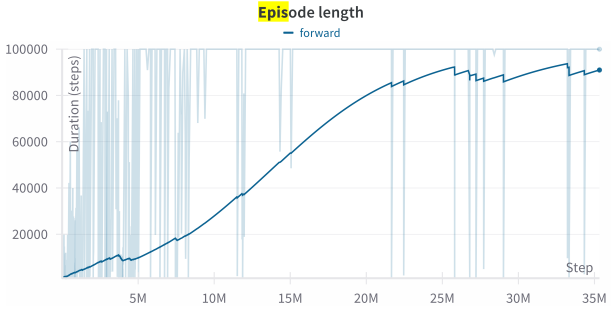

Figure 3: Episode duration (Time-weighted EMA).

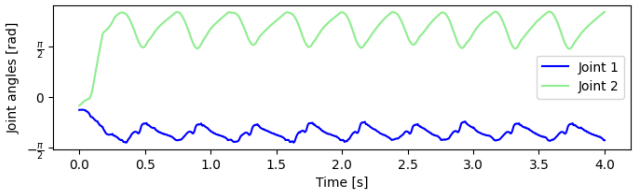

Figure 4: Recorded joint angles in the first 4 seconds of an episode, after random initialization.

3.2 Crawling backward

In this second experiment, we train the agent to crawl backwards in the same settings as in the first experiment, by changing the sign of the distance to compute the position reward and to evaluate the termination conditions.

As we show in Figure 5, the agent is able to learn to move backwards with

approximately the same amount of experience required for the previous task. However, we did not observe the hypothesized skill transferability between forward and backward crawling: the agent seems to learn sligthly faster when trained from scratch. This intuition of low transferability can be qualitatively checked through visualization of the two learned tasks: while in forward crawling the agent starts with single big steps, struggling to find the right grip, in backward crawling it learns early how to move at constant low speed with very little joint movements, and then gradually increases the amplitude of the joint movements and consequently reaches higher speed.

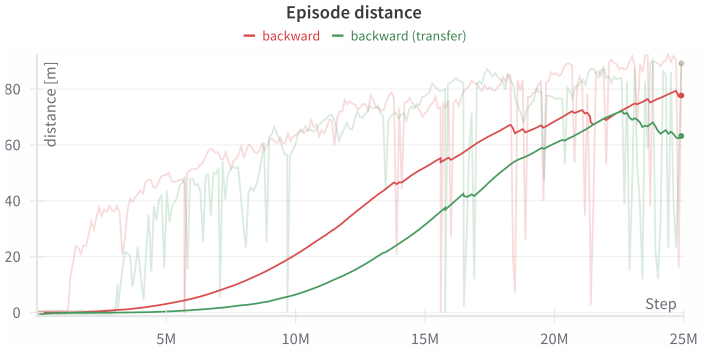

Figure 5: Learning curves (Time-weighted EMA). Red: the agent learns to move backward from scratch. Green: the agent learns to move backward after being pre-trained to move forward.

4 Conclusions

In conclusion, the agent managed to learn to crawl both froward and backward at a sustained average speed, starting from random initializations of joint angles. It is worth noting that the use of Q-learning imposed to discretize the action space to a limited number of configurations; this did not allow to exploit the intrinsic continuity of the physical actions as well as their relation with the continuous state space. We suspect that this limitation might also play a role in the scarce skill transferability that we observed in our last experiment, as learning a correct representation of the environment dynamics would probably result in a higher level of generalization across tasks.