RL-Course 2024/25: Final Project Report

Multicultural Agent: Carla López Martínez, Gaurav Niranjan

February 26, 2025

1 Introduction

This project is part of the Reinforcement Learning course, with the main goal of designing and implementing a reinforcement learning (RL) agent that can play a simulated air hockey game. The objective is to implement or modify an RL algorithm that can handle complex environments and gradually adapt to different tasks.
To build a strong foundation, the RL agent is first tested on simpler environments from the Gym library, specifically the Pendulum-v1 and HalfCheetah-v4 environments. These simpler tasks help fine-tune the learning algorithm, evaluate its ability to manage continuous control problems and ensure stability before moving on to the main challenge.
The core focus of this project is the HockeyEnv, a custom-built environment that simulates a two-player air hockey game. The main goal is to train the agent to play against a basic opponent. The environment includes dynamics such as the positions, velocities, and angles of the players, as well as the puck's position and speed. It also tracks how long each player holds the puck. The agent interacts with the environment by choosing actions that control its movement and strategy, aiming to gain control of the puck and score goals against a weak opponent.
The environment updates the game state based on the actions of both players, providing detailed observations that include player positions, velocities, angles, puck movement, and remaining puck possession time. The ultimate goal of this project is to train an RL agent that can consistently defeat a weak opponent in the hockey game environment.
We outline the algorithmic details in Section 2 and explain the experiments done on the different environments in Section 3. Finally, we end the report with a brief discussion in Section 4.

2 Methods

In this section, we detail the implementation of the Soft Actor-Critic (SAC) and Twin Delayed DDPG (TD3) algorithm. We focus on the algorithmic components, including network architectures and objective functions, as well as the techniques and parameters used in the code.

2.1 Soft Actor-Critic (SAC) Algorithm: Gaurav Niranjan

SAC [5] is an off-policy actor-critic method that maximizes both the expected return and the policy entropy. The maximum entropy framework encourages exploration by adding an entropy term to the reward objective. The overall objective for the policy is to maximize:

$$\begin{aligned} V^{\pi}(s) &= \mathbb{E}_{a\sim\pi}\left[Q^{\pi}(s,a)\right] + \alpha\mathcal{H}\left(\pi(\cdot|s)\right) \\ &= \mathbb{E}_{a\sim\pi}\left[Q^{\pi}(s,a) - \alpha\log\pi(a|s)\right]. \end{aligned}$$

where, $Q^{\pi}(s,a)$ represents the expected return from taking action a in state s, and then following policy π afterward. α is the temperature parameter balancing the reward and entropy terms and $\mathcal{H}(\pi(\cdot|s))$ denotes the entropy of the policy at state s [1].

2.1.1 Algorithm Components

1. **Actor Network**: The actor network parameterizes a stochastic policy $\pi_\theta(a|s)$. For continuous action spaces, it outputs the mean $\mu_\theta(s)$ and standard deviation $\sigma_\theta(s)$ of a Gaussian distribution, parameterized by a neural network θ. Actions are sampled via: $a \sim \pi_\theta(a|s) = \mathcal{N}\left(\mu_\theta(s), \sigma_\theta(s)\right)$

2. **Critic Networks**: Two Q-networks, $Q_{\phi_1}(s,a)$ and $Q_{\phi_2}(s,a)$, are used to estimate the expected return. This twin-network setup helps mitigate overestimation bias by taking the minimum value during updates.

2.1.2 Loss Functions and Objective Functions

1. **Critic Loss**: The critic networks are optimized by minimizing the mean squared error (MSE) between the estimated Q-values and a target value. For each critic, the loss is defined as

$$J_Q(\phi_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[\frac{1}{2}\left(Q_{\phi_i}(s,a) - y\right)^2\right],$$

with the target y computed as:

$$y = r + \gamma\,\mathbb{E}_{a'\sim\pi_\theta}\left[\min_{i=1,2} Q_{\phi_i}(s',a') - \alpha\,\log\pi_\theta(a'|s')\right].$$

where r is the current reward and γ is the discount factor.

2. **Actor Loss**: The actor aims to maximize the expected Q-value while promoting exploration through entropy. The loss for the actor is given by:

$$J_\pi(\theta) = \mathbb{E}_{s\sim\mathcal{D}}\left[\mathbb{E}_{a\sim\pi_\theta}\left[\alpha\,\log\pi_\theta(a|s) - \min_{i=1,2} Q_{\phi_i}(s,a)\right]\right].$$

3. **Temperature Parameter Loss**: To automatically adjust the temperature parameter α, we minimize [7] [6]:

$$J(\alpha) = \mathbb{E}_{a\sim\pi_\theta}\left[-\alpha\log\pi_\theta(a|s) - \alpha\bar{\mathcal{H}}\right],$$

where $\bar{\mathcal{H}}$ is the target entropy.

2.1.3 Implementation Details

The code for implementing SAC is inspired from the following Github repositories [9] [11]

Network Architectures

- **Policy Network:** Simple feed-forward network with ReLU activations. Two separate branches compute the mean and standard deviation of the action distribution.
- **Q-Networks:** Takes both state and action as input. Uses a simple feed-forward network with ReLU activations and produces the Q-value estimate. The target Q-network parameters are updated using a soft update mechanism to ensure training stability and reduce overestimation bias.

Replay Buffer A large replay buffer (dequeue) (10^5 transitions) stores agent-environment interactions.

Optimization Adam with learning rate 3×10^{-4}, batch size 256, discount factor (γ) 0.99, Target smoothing coefficient (τ) 0.005.

Exploration Strategy The entropy term naturally encourages exploration. No additional noise is added to actions.

2.2 Twin Delayed DDPG (TD3) Algorithm: Carla López Martínez

Twin delayed DDPG is an off-policy continuous action space reinforcement learning method based on the DDPG algorithm. TD3 implements the following modifications to DDPG in order to reduce instability and overestimation bias by the critic: clipped double Q-learning, delayed update of target and policy networks, and target policy smoothing.
Clipped double Q-learning consists of estimating the Q-value with two different target critic networks, and then selecting the minimum value between the two for the target update. This is done to upper bound the less biased estimator of Q with the biased one in order to reduce overestimation. Only one actor is used in order to make the method less computationally expensive. Therefore, in clipped double Q-learning the target update is [4]:

$$y = r + \gamma \min_{i=1,2} Q_{\theta'_i}(s', \pi_{\phi'}(s'))$$

The delayed update of the target and policy networks is done to reduce instability. It works by updating the target and policy networks after a fixed number of updates of the critic network. The idea behind it is to ensure that the target networks can reduce the error of the value estimates significantly so that when the policy network is updated the policy does not cause unstable behaviors due to poor value estimates. The update rate is defined by the parameter τ [4]:

$$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i, \quad \phi' \leftarrow \tau\phi + (1-\tau)\phi'$$

The final main modification of TD3 is the target policy smoothing, which is used to reduce the variance of the target. The idea is to apply a small amount of random noise to the action selected by the actor following the policy. The random noise is clipped to keep the target close to the original action for an effective regularization [4]:

$$y = r + \gamma Q_{\theta'}(s', \pi_{\phi'}(s') + \epsilon), \quad \epsilon \sim \text{clip}(\mathcal{N}(0,\sigma), -c, c)$$

The objective function that we want to maximize is the expected Q-value from one of the critics:

$$J(\phi) = \mathbb{E}_{s\sim\mathcal{D}}\left[Q_{\theta_1}(s, \pi_\phi(s))\right]$$

To do so we must minimize the critic loss:

$$L(\phi_i) = \mathbb{E}_{(s,a,r,s')\sim\mathcal{D}}\left[(Q_{\phi_i}(s,a) - y)^2\right],$$

The code that implements TD3 as described in this section was written following the paper "Addressing Function Approximation Error in Actor-Critic Methods" [4] and using the scripts from the following GitHub repositories as inspiration [3, 8].

2.2.1 Core Components

The code to implement this algorithm is divided into two scripts: the main script and the TD3 script. The main script initializes the desired environment and allows the user to define different parameters such as the length of training, the learning rate or the strength of the opponent in the hockey environment. It also implements the training loop, which consists of a set of initial time steps where a random policy is used in order to gain experiences, followed by the loop where the policy is updated iteratively during the episode simulations. Every few time steps, the current policy is evaluated by testing it in 10 episodes and calculating the average reward obtained. After each evaluation, the training and testing rewards up until that point are saved, as well as a model checkpoint.
The TD3 script contains classes for the actor and critic networks, as well as a class that implements the replay buffer and another class that implements the training algorithm described in the previous section. Details about each of the classes are explained below.

Actor and Critic Networks The actor and critic neural networks are each formed by three linear layers, in the actor's forward pass the state is given as input and the action is received as output, in the critic networks the state and action are given to the networks and the two Q-values are received as output. The first two layers have a ReLU activation function, and the last layer has a tanh activation.

Replay Buffer A replay buffer of maximum size one million was defined in order to save the experiences of the agent in the environment for each time step so that the training algorithm can sample from them to optimize the policy.

Training Algorithm The training method is implemented in the TD3 class: first the replay buffer is sampled to obtain the current and next states, the current action, the obtained reward and the status of the episode. Then, the next action is selected using the actor network, and the clipped noise is added as part of the target policy smoothing. The target and current Q-values are computed using the critic networks, and then the critic loss is computed using the equation described above.
The critic networks are then optimized using Adam with a learning rate of 0.001. After the delay steps are reached, the networks are updated: the actor loss is computed and the actor network is optimized using the deterministic policy gradient:

$$\nabla_\phi J(\phi) = \mathbb{E}_{s\sim p^\pi}\left[\nabla_a Q^\pi(s,a)\Big|_{a=\pi(s)} \nabla_\phi \pi_\phi(s)\right]$$

The target actor and critic networks are updated using the equation shown in the delayed update section above, with τ = 0.005.

2.2.2 Added Modifications

Once the code for the basic TD3 implementation was written, some modifications were added in order to try and improve the performance of the agent even further.

Adaptive Learning Rate The learning rate was a fixed value set initially as 0.0003. However, it is not optimal to have a fixed learning rate for the whole process, as initially a bigger learning rate may be more efficient for faster learning of the basics and then eventually a smaller learning rate can be used for final convergence. To do this, an exponential learning rate scheduler was set using torch.optim.lr_scheduler.ExponentialLR, which decreases the actor and critic optimizer's learning rate exponentially using as parameter gamma = 0.99. The scheduler changes the learning rate every few time steps, and a method in the TD3 script applies the new learning rate to the optimizers. As a starting learning rate of 0.0003 is too small for the adaptive method, the initial learning rate was set as 0.001.

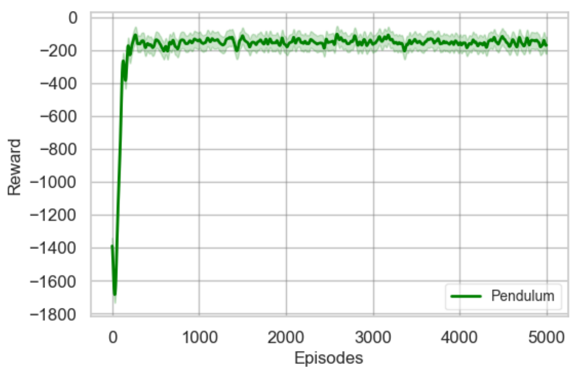
Pink Noise The basic code I wrote used the action selected by the policy directly on the environment without additional exploration noise. This is not optimal, as the algorithm does not explore different actions that could be beneficial. For the exploration noise, pink noise was selected, as the paper "Pink noise is all you need: Colored noise exploration in deep reinforcement learning." [2] shows that it performs better than white noise in TD3. A class was added to the TD3 script that generates this noise for exploration: it sequentially filters white noise using a parameter β, which controls the temporal correlation to ensure that the current pink noise is correlated with past noise. This noise distribution is sampled, scaled to fit a desired standard deviation set by the user, and applied to the action selected by the policy in the main script.

3 Experiments

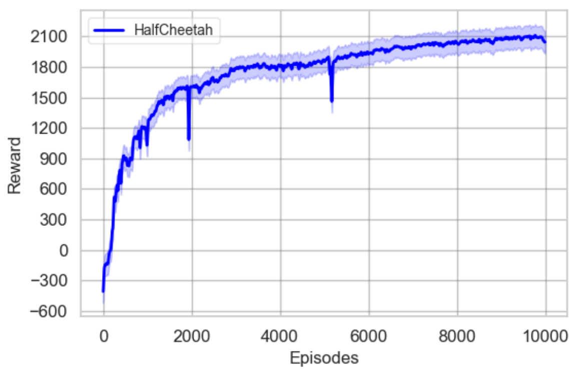
We apply the chosen algorithms to Pendulum-v1, HalfCheetah-v4 and Hockey environments.

3.1 Soft Actor-Critic

After making the necessary changes according to the environment, we analyze the agent's performance on Pendulum and HalfCheetah. For these we set the target entropy to be $-(action\_dim)$ as recommended by the authors of the SAC paper.



(a)



(b)

Figure 1: SAC on (a) Pendulum and (b) HalfCheetah environments.

We then train the agent on the Hockey environment against the weak opponent. The reward signal of this environment is very sparse. At every time step, it gives a signal for win/lose and a small negative value for closeness to the puck. To encourage the agent to touch the puck, one modification we made here is adding some positive reward signal for touching the puck. Since the entropy term in SAC controls exploration, we experiment with different target entropy values to see how they influence the agent's training.

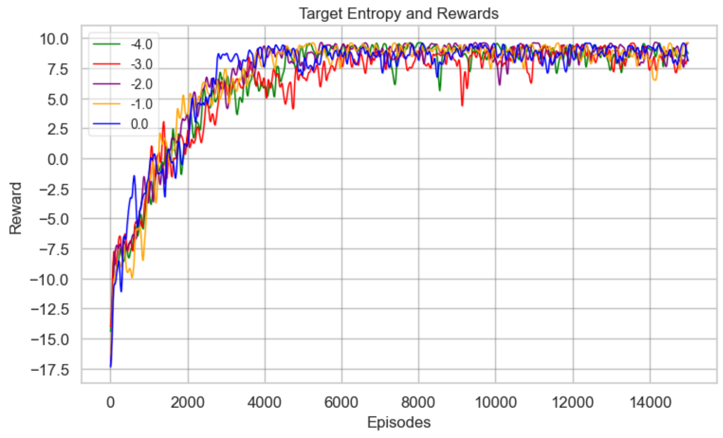


Figure 2: SAC on Hockey environment using different target entropy values

As evident from fig. 2, different values of the target entropy don't seem to have an effect on the agent's training. The agent is able to beat the weak opponent consistently. The automatic temperature update minimizes the difference between the current policy's entropy and the target entropy. When we adjust the target entropy slightly, the temperature parameter α is modified accordingly (increased for lower target entropies and decreased for higher target entropies) to maintain an effective entropy level. As a result, for the range of target entropies we experimented with, the overall balance between exploration and exploitation remains unchanged, leading to similar learning curves over many episodes. This behaviour demonstrates the robustness of the automatic tuning mechanism in SAC. From fig. 3 we can see that the value of $\alpha \log \pi_\theta(a|s)$ remains very low irrespective of the target entropy. So effectively, $\alpha \log \pi_\theta(a|s)$ has a low impact on the critic and policy loss functions.

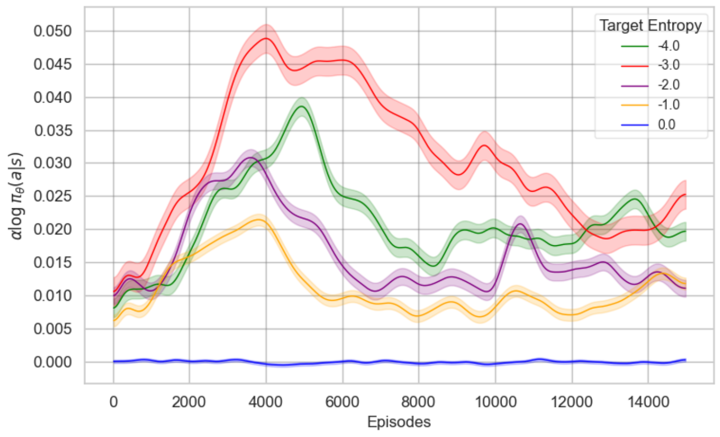
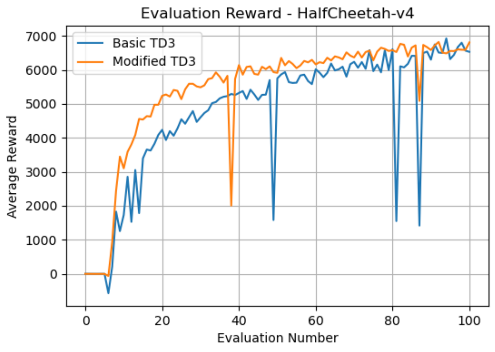


Figure 3: Value of $\alpha \log \pi_\theta(a|s)$ for different target entropies

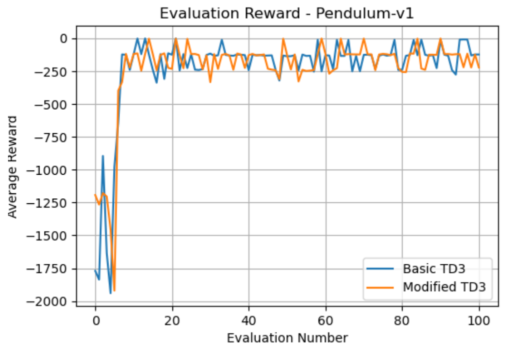
Interestingly, a target entropy of 0 also learns as well as the other agents. We observed that the entropy of the action distribution given by the actor stays very close to 0, meaning it has a very small variance. Even when the variance is extremely small, there is still a tiny amount of randomness when sampling actions. This noise could be enough to occasionally explore alternative actions, even though the policy is effectively near-deterministic. Early in training, before the target entropy adjustment drives the policy to be nearly deterministic, there's typically enough exploration (thanks to the inherent stochasticity of the initial network parameters) to collect a diverse set of experiences. By the time the entropy is very close to 0, the agent has already explored enough of the environment to learn a robust policy.

3.2 Twin Delayed DDPG (TD3)

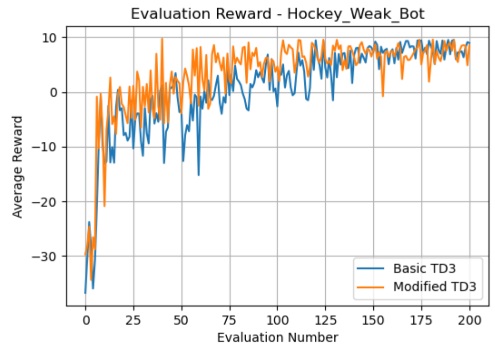
Both the basic and the modified TD3 algorithms were tested in the three environments, including the weak and strong bots for the Hockey environment. Note that the training rewards were not plotted due to space constraints. The x-axis in these figures represents the evaluation number: an evaluation of 10 episodes was done every 5000 time steps, and the average reward of those episodes was calculated and plotted. For the Pendulum and HalfCheetah, the algorithm was run for 500000 time steps, and for the Hockey bots the algorithm was run for one million time steps as it took longer to converge.



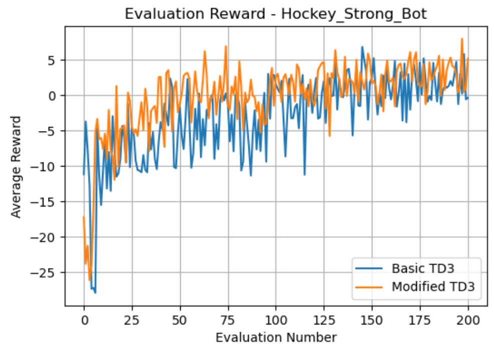
(a) Learning curve for HalfCheetah-v4



(b) Learning curve for Pendulum-v1



(c) Learning curve for Hockey Weak Bot



(d) Learning curve for Hockey Strong Bot

Figure 4: Learning curves for different environments with and without modifications

It can be seen in the HalfCheetah and Hockey environments that the agent performs slightly better with the modifications than with just the basic TD3 code. Although towards the end of the training the performances become similar for both versions, there is a visible performance gap for most of the training process. In Pendulum, both versions perform similarly, which indicates that the added modifications were not useful for this particular environment due to its characteristics. All in all, both versions performed well in the environments, as the rewards obtained were very high in all cases. For the Hockey environment, the agent is capable of beating the weak bot consistently towards the end of the training, as seen by the curves being near the 10 reward mark, which is the maximum reward possible. For the strong bot, the agent struggles more, but towards the end of the training the curves stay in the 0-5 reward range,

which indicates it mostly either draws or beats the strong bot. It can then be concluded that the code was implemented correctly and that the modifications added, while not groundbreaking, were successful in improving the performance of the agent.
Different parameter values were tested to see which ones to select for the final version of the code. For the modified code, an initial learning rate of 0.001, $\tau = 0.005$, $\beta = 0.95$, $\gamma = 0.99$ (both for the discount factor and the learning rate scheduler gamma parameter) and delay = 2 were best. For the non-modified code, a learning rate of 0.0003, $\tau = 0.005$, $\gamma = 0.99$ (discount factor) and delay = 2 were best. Other values that had good results but were not optimal were $\tau = 0.001$ for both versions, $\gamma = 0.995$ and $\gamma = 0.95$ (for the learning rate scheduler gamma parameter) and different learning rates in the range of (0.001, 0.0001) for both versions.

4 Discussion

The algorithms used for this project, SAC and TD3, are both off-policy methods that work well in a continuous action-space context. Nevertheless, there are some important differences in how these methods tackle the training of the agent:

- TD3 uses a deterministic policy gradient approach, and improves upon DDPG using clipped double Q-learning, delayed policy updates and target policy smoothing. These modifications make TD3 stable, efficient and ensure a lower overestimation bias. It has been shown that TD3 can outperform other methods, including SAC, in many environments [4]. However, due to TD3 using a deterministic policy, it may struggle with exploration in more complex environments such as the Hockey one, where stochasticity is beneficial.

- SAC learns a stochastic policy using entropy regularization, which aims to maximize both the reward and the entropy of the policy. This approach encourages the agent to explore during training, and the entropy term also helps the agent avoid getting stuck in local optima early on. This improved exploration strategy makes SAC have a better performance in complex environments that require high exploration, as well as in situations where the actions are high-dimensional. However, it is computationally expensive and may take longer to converge in simpler environments.

For Pendulum, both algorithms perform similarly well (with SAC doing slightly better) and converge at around the same rate. For HalfCheetah, TD3 reaches much higher rewards at a much faster rate: SAC converges to 2100 reward at around 8000 episodes, while TD3 converges to 7000 reward at around 500 episodes (evaluation number 100 corresponds to time step 500000, which for HalfCheetah is episode 500). For the Hockey environment, it is only possible to compare the performance for the weak bot. It can be seen that both algorithms converge to the same reward value (7-10 range) at around the same rate: evaluation numbers 125-200 correspond to time steps 625000-1000000, which for the Hockey environment is around episodes 6000-8000, the same range at which SAC reaches a plateau.
In conclusion, although sometimes one of the methods outperforms the other in certain environments due to their different strength and weaknesses, both algorithms fulfill the requirement of consistently beating the weak bot and obtaining high rewards in all environments, leading to satisfactory project results. The code used for implementing both algorithms can be found at the following GitHub repository: [10].

References

[1] J. Achiam. Spinning Up in Deep Reinforcement Learning. 2018.

[2] O. Eberhard, J. Hollenstein, C. Pinneri, and G. Martius. Pink noise is all you need: Colored noise exploration in deep reinforcement learning. In *The Eleventh International Conference on Learning Representations*, 2023.

[3] S. Fujimoto. Td3 - twin delayed deep deterministic policy gradient. `https://github.com/sfujim/TD3`, 2018. Accessed: 2025-02-20.

[4] S. Fujimoto, H. van Hoof, and D. Meger. Addressing function approximation error in actor-critic methods. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1587–1596, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[5] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, abs/1801.01290, 2018.

[6] T. Haarnoja, A. Zhou, S. Ha, J. Tan, G. Tucker, and S. Levine. Learning to walk via deep reinforcement learning. *CoRR*, abs/1812.11103, 2018.

[7] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine. Soft actor-critic algorithms and applications. *CoRR*, abs/1812.05905, 2018.

[8] X. J. Hao. Td3-pytorch. `https://github.com/XinJingHao/TD3-Pytorch`, 2021. Accessed: 2025-02-20.

[9] S. Huang, R. F. J. Dossa, C. Ye, J. Braga, D. Chakraborty, K. Mehta, and J. G. Araújo. Cleanrl: High-quality single-file implementations of deep reinforcement learning algorithms. *Journal of Machine Learning Research*, 23(274):1–18, 2022.

[10] C. Lopez and G. Niranjan. Reinforced learning project repository. `https://github.com/apoorvagnihotri/reinforced/tree/main/project`, 2023. GitHub repository.

[11] D. Yarats and I. Kostrikov. Soft actor-critic (sac) implementation in pytorch. `https://github.com/denisyarats/pytorch_sac`, 2020.