1. Introduction

In modern robotics, controlling manipulators has traditionally relied on sophisticated mathematical modeling and Inverse Kinematics (IK). However, in a complex physical world with numerous variables and unpredictable constraints, explicitly coding a perfect set of instructions for every possible condition is nearly impossible. This challenge is particularly evident with the 6-DOF UR5 robotic manipulator, where the combination of geometric complexity and physical reality makes trajectory generation a high-dimensional control problem.

To overcome these limitations, this project utilizes Reinforcement Learning (RL) to enable the UR5 robot to autonomously reach a target pose. Unlike passive data-driven approaches, RL involves a process of active exploration where the agent learns optimal behavioral patterns through trial and error, guided by a reward signal . The primary objective is to develop a robust control policy that navigates the complex causal relationships of the environment to achieve high precision and stability.

This study conducts a comparative analysis of two representative algorithms, each employing a different methodological approach:

- **Proximal Policy Optimization (PPO):** An on-policy algorithm that ensures learning stability by placing a physical limit on the policy update magnitude, preventing the potential for sudden policy instability sometimes encountered in complex joint control.
- **Soft Actor-Critic (SAC):** An off-policy algorithm that maximizes data efficiency by reusing past experiences from a replay buffer and encourages exploration through entropy maximization, allowing for more flexible and fluid movements.

The goal of this project is to improve the robot's accuracy and motion smoothness by improving the reward function and tuning the training parameters. The next sections will explain the theory of these algorithms, how the reward functions were designed, and the results comparing the efficiency of PPO and SAC.

2. Background (Theory Overview)

2.1 Reinforcement Learning Overview

The problem of controlling a UR5 robot arm differs fundamentally from tasks where a fixed correct answer exists, such as grading an exam. In a complex physical world, it is nearly impossible for humans to explicitly code a perfect set of instructions that guides the robot arm to a target under every possible condition. This is where Reinforcement Learning becomes essential. RL is not about passively memorizing correct answers from a dataset; instead, it is a process of active exploration. It is similar to a child learning to walk, experiencing pain when falling, receiving praise when standing, and eventually learning to balance through trial and error. The robot performs random actions in an unknown environment to observe how those actions change the world and whether the outcome is beneficial or detrimental.

In this project, the Reward acts as a compass that guides the UR5 robot through complex physical laws. It receives a positive signal when approaching the target and a negative signal for collisions or wasted time. By accumulating these simple signals, the robot realizes which movements are better and gradually reinforces optimal behavioral patterns. This interaction between the agent and the environment is mathematically formalized as a Markov Decision Process (MDP). An MDP describes the robot's experience through the causal relationships of five key elements.

The first element is the State (S_t), which represents the current reality perceived by the robot. In this project, the robot recognizes its situation using a 19-dimensional information vector, including the angles and velocities of 6 joints, the coordinates of the end-effector, the target position, and collision status. This is the only information the robot uses to make its decisions. Based on this state, the robot chooses an Action, which is simply how the robot moves. Instead of directly controlling motor torque, the robot determines how much to move from its current posture to the next, defined as the Joint Position Delta ($a_t \in R^6$). Once an action is taken, the Transition Probability (P) determines how the environment moves to the next state (S_{t+1}). Here, the MuJoCo physics engine acts as the law of nature, calculating gravity, friction, and inertia to determine the robot's next posture. The outcome is evaluated by the Reward Function (R), which gives an immediate rating of the robot's performance. The system gives points for reaching the target quickly and accurately, while giving penalties for collisions. Finally, the Discount Factor (γ) measures patience, prioritizing long-term gains over immediate rewards ($0 \leq \gamma < 1$). This mechanism helps the robot handle short-term costs to achieve the ultimate goal instead of just taking small points right away.

Ultimately, the goal of reinforcement learning is to find an optimal policy (π) that maximizes the expected value of the Return (G_t). The Return is the sum of not just the current reward but all future rewards, as shown in the following equation:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

Rather than just summing rewards, this equation enables the robot to develop a long-term strategy, allowing it to select the most effective action for the present by accounting for future uncertainties.

While RL is an effective concept for control problems, it faces instability in practical applications. In the process of controlling complex 6-dimensional joints, the robot might mistake a lucky success for a general rule. If the robot drastically alters its policy based on such errors, the previously learned skills may be lost, leading to a sudden drop in performance. This is known as Policy Collapse. For a robot, not forgetting what has been learned while slowly improving is often more important than the speed of learning itself.

Conservative Policy Iteration (CPI) was introduced to address this issue **[1]**. The authors proved a crucial fact using the following Estimated Objective function:

$$L_{CPI}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} A_t \right]$$

The implication of this formula is clear: to prevent learning collapse, the update must be conservative. It mathematically proved the value of caution, showing that a lower bound for performance improvement is guaranteed when the policy does not change drastically. However, CPI has a significant limitation. While it proved the necessity of moving conservatively, it failed to provide a practical solution for how exactly to control the update so it does not cross the safety line. It identified the importance but lacked the means to enforce it.

TRPO (Trust Region Policy Optimization) emerged to solve this practical problem **[2]**. To maintain the conservativeness emphasized by CPI, TRPO introduced a "Trust Region" constraint, forcing the difference between policies (measured by KL Divergence) to stay within a specific threshold (δ).

$$\max_{\theta} \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} A_t \right] \quad \text{subject to} \quad \hat{\mathbb{E}}_t\left[\mathrm{KL}[\pi_{\theta_{old}}(\cdot \mid s_t), \pi_\theta(\cdot \mid s_t)]\right] \le \delta$$

Thanks to this constraint, TRPO successfully implemented the stability pointed out by CPI. However, this approach had a clear limitation in terms of computational complexity. To meet the constraints at every update, the system must process complex second-order information. Even with some optimizations, this iterative calculation remains very slow. This high computational demand makes it difficult to implement in real-time robot control, where the control loop often requires updates within milliseconds

2.2 Proximal Policy Optimization (PPO)

The TRPO algorithm described previously required a mathematically expensive price to ensure stability. To address this issue, the authors of TRPO proposed a efficient alternative methodology called Proximal Policy Optimization (PPO), which achieves similar stability without the need for complex second-order derivatives **[3]**. The core strategy of PPO lies in its name, Proximal. It enforces an upper bound on change using only first-order operations, ensuring that the robot learns new skills without deviating too far from the proven existing policy. To achieve this, PPO utilizes several logical mechanisms.

The first mechanism is the Clipped Surrogate Objective, which acts as a safety brake. PPO works by constraining the probability ratio between the new and old policies, which was a concept first introduced in CPI. Instead of complex constraints, PPO uses a simpler method to keep this ratio within a safe range. Instead of imposing constraints via complex formulas, PPO adopts a method that drastically ignores the excess if the ratio exceeds a defined safety zone. This is expressed in the following equation:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

Here, ϵ represents the allowable range of change and is typically set to 0.2. This formula ensures conservative updates by ignoring performance gains that fall outside this predefined safety zone. Even if the robot encounters a lucky spike in performance that suggests a drastic policy change, the system does not trust that luck and only accepts gains within the safe range. This prevents the policy from being destabilized by noisy data and ensures consistent improvement during the training process.

The second part of the framework involves estimating action values through the Advantage and Critic components. For the robot to determine that it performed well, a comparison target is necessary. It must distinguish whether it simply received high rewards due to luck or if it performed better than average through skill. This is the concept of Advantage $\hat{A}_t$. To achieve this, PPO uses a separate neural network called the Critic V_ϕ. The Critic acts as a judge that objectively evaluates the value of the current state, and the Actor or robot reinforces its actions only when it produces results better than the judge's expectation. This ensures the policy is updated based on actual performance gains, which effectively filters out unnecessary fluctuations during training.

Finally, the Entropy Bonus is included to maintain exploration. The robot must be prevented from becoming overconfident too early and sticking to a single action, which leads to local optima. For this purpose, PPO adds an Entropy Bonus S to the objective function:

$$S[\pi_\theta](s_t) = -\sum \pi_\theta(a \mid s_t) \log \pi_\theta(a \mid s_t)$$

This term provides a reward when the probability distribution is not skewed to one side but reasonably spread out. This encourages the robot to maintain exploration, as the current knowledge may not yet be optimal.

In conclusion, PPO optimizes by combining these three elements into a single formula. It integrates clipping for safe changes, value function error for accurate value judgment, and entropy for continuous exploration:

$$L^{PPO}(\theta) = \hat{\mathbb{E}}_t[L^{CLIP}(\theta) - c_1 L^{VF}(\phi) + c_2 S[\pi_\theta](s_t)]$$

Through this equation, the UR5 robot can avoid the risk of policy divergence while steadily converging towards the target when solving complex trajectory generation problems.

2.3 Soft Actor Critic (SAC)

While PPO offers reliable stability, it has a significant drawback in terms of sample efficiency when applied to robotics. To understand this, the fundamental difference between On-policy and Off-policy data processing must be addressed. On-policy algorithms like PPO only trust data collected by the current policy π_θ. If the robot's behavioral pattern changes even slightly through learning, data collected by past policies becomes invalid for the current policy update. This means that high-cost data is consumed as a single-use resource. In contrast, SAC adopts an off-policy approach, utilizing a Replay Buffer to store and reuse past experiences **[4]**. This is possible because SAC learns the causal relationship between state and reward itself through the value function Q. Whether the movement was generated by a foolish policy or a smart one in the past, the fact that taking a specific action in a specific state produces a certain result remains unchanged. Therefore, SAC can learn efficiently with significantly less data by learning from past experiences.

However, the primary innovation of SAC is not limited to its memory capacity; it fundamentally redefines the learning objective by modifying the value calculation equation. Traditional reinforcement learning follows the Standard Bellman Equation, which solely maximizes the sum of rewards:

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

The core of this equation is the Max operator. The robot looks only at the single action with the highest score in the next state, a process known as Hard Maximization. This approach causes the robot to mistake a locally good path discovered early for the optimal one, leading to premature convergence that excludes other possibilities. To solve this problem, SAC introduces the Soft Bellman Equation. Here, the meaning of Soft in the algorithm's name becomes apparent. Instead of rigidly selecting only the best action, it implies embracing multiple potential actions stochastically:

$$Q(s,a) = r + \gamma \mathbb{E}_{a' \sim \pi}[Q(s',a') - \alpha \log \pi(a' \mid s')]$$

In this formula, the existing Max is replaced by the Expectation $\mathbb{E}$ and an Entropy term $-\alpha \log \pi$. This entropy term represents degrees of freedom. In other words, for the SAC robot, a valuable state is not simply a place with high rewards, but a state where rewards are high and the agent is not stuck but free to choose the next action. Thanks to this formula, the robot prefers a wide and safe path where it can correct its course at any time, rather than a narrow optimal path.

While this Soft Bellman Equation appears theoretically perfect, problems arise in actual implementation due to Q-value overestimation. Since neural networks are not perfect, predictions always contain errors. A problem occurs when attempting to find the best action because if the maximum value is selected from several values containing noise, the system inevitably chooses the action with the largest positive error rather than the action with the best skill. The robot believes this inflated score is its true capability and falls into a overestimation cycle of reinforcing that incorrect behavior. To mitigate this fundamental issue, SAC utilizes a safety mechanism called Clipped Double Q-learning:

$$y = r + \gamma(\min_{j=1,2} Q_{\phi_j}(s', a') - \alpha \log \pi(a' \mid s'))$$

SAC employs two independent judges, Q_1 and Q_2. If both give similar scores, they are trusted. However, if one gives a high score and the other a low score, SAC adopts the lower score as the true value consistently. Thanks to this conservative estimation that high scores potentially containing bubbles are not trusted, the learning process does not become unstable and remains steady.

Ultimately, all these complex processes, soft value calculation, and dual safety mechanisms converge to a single goal which is how to move the robot. The SAC Actor is updated to minimize the following loss function J_π, which increases the value Q evaluated by the Critic while ensuring the action distribution does not become too narrow:

$$J_\pi(\phi) = \mathbb{E}_{s_t \sim D}[\alpha \log \pi_\phi(a_t \mid s_t) - \min_{j=1,2} Q_{\theta_j}(s_t, a_t)]$$

The objective this final formula gives the robot is clear. It instructs the agent to perform actions that the strict two judges rate highly while always maintaining entropy so that the behavior pattern does not become too predictable. In this way, SAC demonstrates the most powerful performance in robot control environments full of uncertainty by effectively integrating data efficiency, flexibility, and stability into a single formula.

3. PPO Implementation

3.1 Reward Function Modification

In reinforcement learning, the Reward Function serves as the primary communication channel indicating what is right or wrong to the autonomous agent. While humans can explain tasks through natural language, the robot understands only numerical feedback. However, the reward system in the provided baseline environment was too coarse to teach the robot sophisticated skills. The reward function was fundamentally redesigned to prevent the UR5 manipulator from overshooting its target. This modification enables high-precision convergence and ensures smooth, stable motion during operation

3.1.1 Modification of Distance Reward (Exponential Shaping)

The first issue addressed was the loss of motivation at the target. The baseline environment provides rewards based on how much the distance decreased compared to the previous step. While effective when the target is far, this approach reveals a fatal weakness when the robot is near the goal. As the distance decreases, the change converges to zero. This is comparable to a student with a 99 score failing to perceive the minute difference required to reach 100, causing growth to stall. Consequently, the robot failed to stop exactly at the target and exhibited oscillating behavior near the goal.

To address this, an Exponential Function term was integrated into the reward structure alongside the existing distance difference. This additional component provides the necessary gradients for precise positioning as the robot nears the target.

$$R_{dist}(d) = 2 \cdot \exp(-4 \cdot d)$$

Here, the weight w_d was set to 2 and the sensitivity coefficient α to 4 to tune the robot's spatial perception. The advantage of the exponential function lies in its Gradient. Unlike linear distance rewards, the reward value rises steeply as it approaches the target. This provides a strong incentive until the very last moment, signaling that a tiny movement here will result in a massive score increase. Through this design, the agent maintains tension and controls its position more precisely as it gets closer to the target.

3.1.2 Action Regularization (Energy Penalty)

The second issue was mechanical stress caused by aggressive optimization. A reinforcement learning agent is primarily driven by cumulative reward maximization. It is interested only in maximizing the total reward, disregarding the motion smoothness. Without braking mechanisms, the robot does not hesitate to apply excessive torque to gain even a fraction of a second or to vibrate joints madly to balance near the target. While this appears as simple graph fluctuations in simulation, it can be catastrophic in the real world. Experience from other hardware projects has shown that servo motors, such as those controlling the steering mechanism of a small-scale drift car, can physically melt down due to high-frequency vibration commands from an untrained agent if this issue is overlooked. Therefore, teaching the robot how to stay calm is not just a design choice, but a matter of hardware survival.

A physical constraint was needed to tell the robot to reach the target but move smoothly with minimal force. A penalty proportional to the magnitude of the action vector was applied.

$$R_{action} = -\lambda_u \parallel a_t \parallel_2^2$$

For stable movement, the penalty weight λ_u needed to be set strongly, around 0.05. However, this raised a serious concern. If a strong penalty of 0.05 is applied to an agent in the early stages of training robot at the beginning of learning, the robot might learn the wrong causal relationship that movement leads to punishment. This creates a risk of adopting a stationary policy, where the agent gives up on moving before it even learns how to move.

3.1.3 Curriculum Learning Strategy for Action Regularization

A balanced trade-off was found between the freedom for exploration and regulations for safety using Curriculum Learning. Just as a child is allowed to play freely before learning manners, the robot was provided with a staged learning environment. Instead of imposing strong constraints from the start, the strategy involves gradually increasing the penalty intensity according to learning progress. The penalty coefficient λ_u was designed to increase linearly over the total training duration.

$$\lambda_u(t) = 0.01 + \frac{t}{T_{total}} \cdot (0.05 - 0.01)$$

In the early stage, the penalty is set low to allow the agent to experience successful outcomes through extensive exploration without excessive constraints. Once the robot has sufficiently learned how to reach the target, the penalty is gradually increased to refine rough movements and optimize the trajectory for smoothness. This design intended for the robot to grow as a fearless explorer in the beginning and converge into a careful technician by the end.

3.2 PPO Hyperparameter Tuning

In the previous section, a reward map was established to guide the robot. However, even an accurate map is insufficient if the algorithm settings used to interpret it are inadequate. Consequently, the default PPO settings were adjusted to account for the geometric complexity and physical reality of the 6-DOF UR5 robot. The learning engine was precisely tuned through several strategic adjustments.

3.2.1 Strategic Balancing of Reward Weights (Orientation Control)

Due to the 20-to-1 ratio between distance and orientation rewards, the agent focused predominantly on positional accuracy at the expense of alignment. This led to incomplete task execution, as the end-effector failed to maintain the specific orientation necessary for grasping. To teach the robot that orientation is as important as position, the orientation control weight was increased tenfold to 1.0. This change aligned the scales of the position and orientation rewards, ensuring the robot recognizes wrist alignment as a mandatory task rather than an optional one.

3.2.2 Maximizing Exploration Efficiency (Entropy Coefficient)

The second tuning involved improving the exploration capabilities of the robot. As discussed previously, the Entropy Bonus is a critical component of the PPO objective function used to induce exploration.

$$L^{PPO}(\theta) = \hat{\mathbb{E}}_t[L^{CLIP}(\theta) - c_1 L^{VF}(\phi) + c_2 S[\pi_\theta](s_t)]$$

An inspection of the baseline code revealed that the entropy coefficient (c_2) was set to 0.0 by default. This meant that the curiosity engine was effectively turned off. Consequently, the robot tended to settle for suboptimal but successful postures found by chance and lacked the incentive to attempting to find more efficient movements. To resolve this, the entropy coefficient was adjusted from 0.0 to 0.01. While this numerical change appears subtle, it revived the entropy term within the objective function and injected necessary randomness into the actions of the robot. This adjustment prevents the agent from settling for early, inefficient successes and encourages it to autonomously discover more natural and efficient paths.

3.2.3 Experimental Adjustment of Clip Range (Golden Ratio Approach)

The final tuning addressed the learning speed of the robot. The Clip Range (ϵ) is a core component of PPO typically set to 0.2, acting as a safety mechanism to prevent drastic policy changes. However, in high-dimensional control problems requiring millions of trials, a constraint of 0.2 can slow down the learning process significantly.

Inspired by the reciprocal of Pi ($1/\pi$), which represents geometric balance, the clipping value was experimentally reset to 0.318. This value targets a zone that is higher than the standard 0.2 but lower than riskier levels like 0.5. By expanding the trust region, the robot is allowed to improve its policy more significantly from a single good experience. This setting created a synergy with the entropy strategy, allowing the robot to explore a wider range of actions while integrating successful paths into its policy more rapidly. As a result, a significant increase in convergence speed was observed during the early stages of training.

3.3 PPO Results

This section analyzes quantitatively and qualitatively how the designed reward functions and tuned parameters progressively evolved the intelligence of the UR5 robot. The results demonstrate not only a numerical increase but clearly illustrate how the injected logic altered the reasoning process of the robot.

3.3.1 Quantitative Analysis: Step-by-Step Performance Enhancement

To ensure data readability, the learning curves were compared by dividing them into two stages which are the structural improvement of the reward function and the internal parameter tuning of the algorithm.

First, the impact of reward function changes on the survival of the agent was analyzed. The Base Model failed to achieve meaningful convergence even after one million trials and showed a success rate of 0.16805. This suggests that rather than learning how to move toward the target, the robot merely moved randomly and was lucky enough to reach near the target. In contrast, adding the exponential distance reward caused the performance to triple to a success rate of 0.57214. This proves that minute reward signals near the target served as an ignition point for learning. Although the energy penalty initially slowed the learning speed due to movement-suppressing constraints, the model reached a higher peak of 0.75321 in the later stages by eliminating unnecessary joint jitter noise. Finally, the curriculum learning strategy showed a success rate of 0.71269. While initial adaptability improved slightly, the changing reward rules confused the agent and resulted in lower final performance compared to the fixed penalty of 0.05.

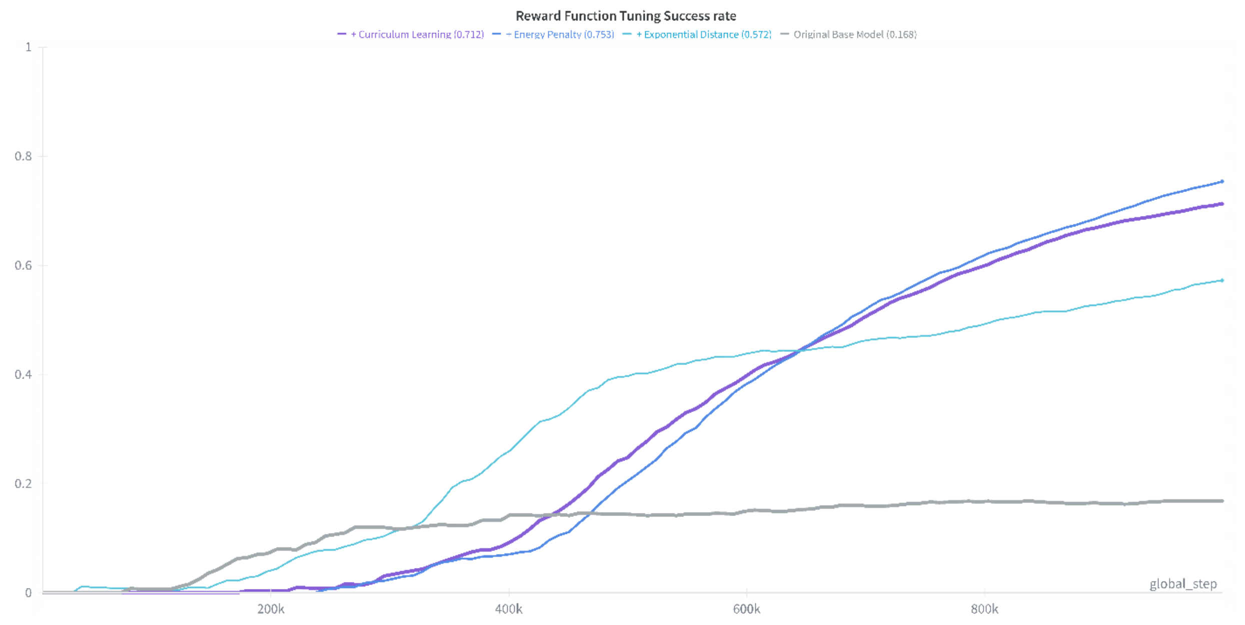


Fig. 1. Comparison of learning success rates based on structural improvements to the reward function.

The results from precise hyperparameter tuning show the intelligence of the algorithm pushed to the limit on top of the optimized reward system. Tuning the orientation weight from 0.1 to 1.0 improved final performance to 0.7896. This change completed the precise control capability even though the learning speed slowed while meeting stricter posture alignment conditions. Reactivating the entropy engine with a coefficient of 0.01 widened the initial exploration range and showed a success rate of 0.75126. Finally, expanding the epsilon clipping range to 0.318 resulted in an unrivaled success rate of 0.90463 in both early and late stages. This serves as decisive evidence of the significant improvement gained in high-dimensional control by appropriately expanding the trust region.

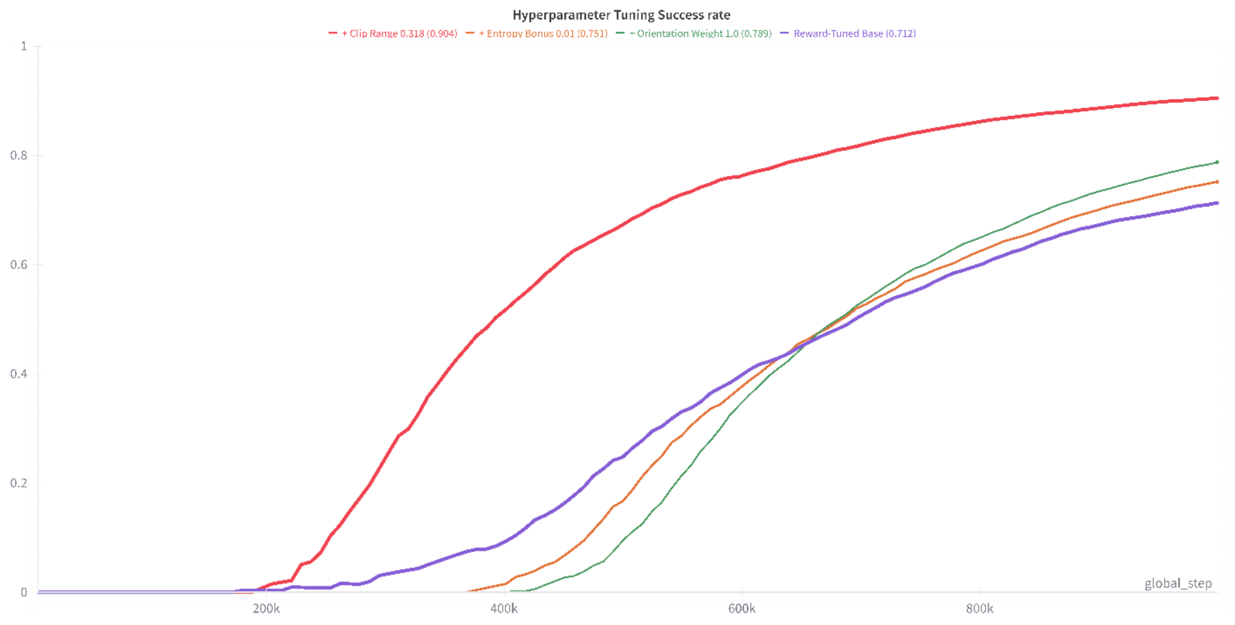
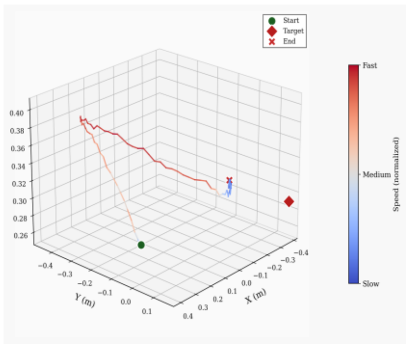


Fig. 2. Comparison of learning performance according to hyperparameter tuning, including orientation weight, entropy, and epsilon clipping.

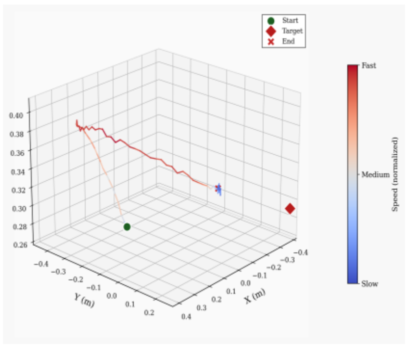
3.3.2 Qualitative Analysis: End-effector Trajectory and Physical Behavior

Just as important as numerical success is the quality of the path drawn by the robot. The end-effector trajectories for five different random seeds were visualized using MATLAB to compare the behaviors between the baseline and the optimized models.

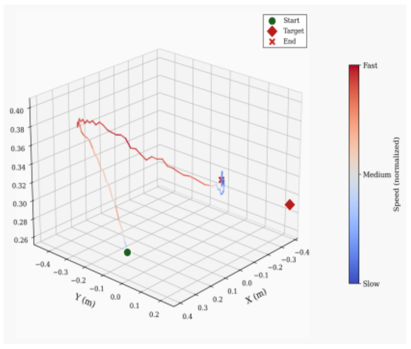
The trajectories of the baseline robot exhibit typical aspects of learning failure where all five seeds failed to reach the target. Notably, the robot could not stop near the target and was observed bouncing off with severe high-frequency jittering. This indicates that the robot failed to learn the concepts of deceleration and convergence entirely.



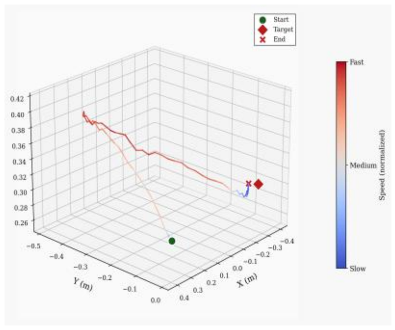
(a) Seed 42



(b) Seed 55



(c) Seed 100


(d) Seed 200

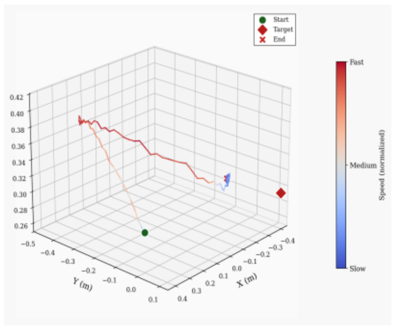
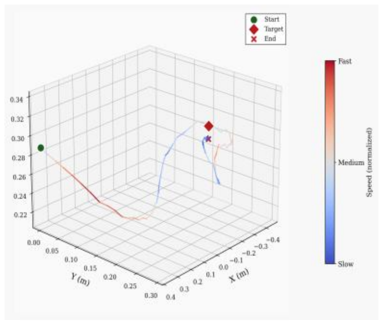
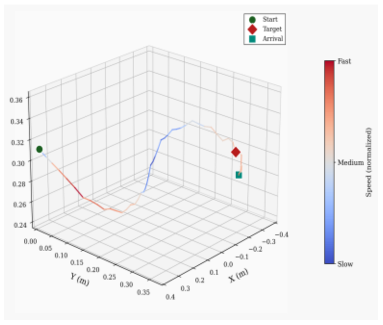
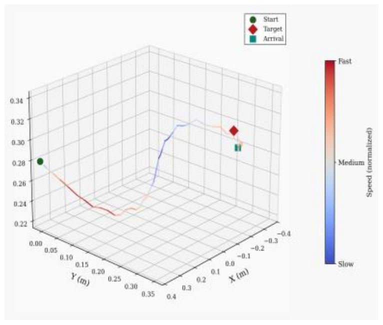

(e) Seed 300

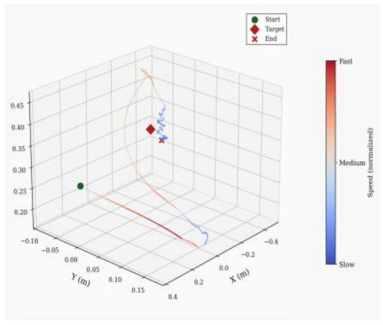
Fig. 3. End-effector trajectories of the baseline agent. Visualization across five random seeds showing typical failure patterns. The robot exhibits severe high-frequency jitter near the target and fails to converge due to the lack of action regularization.

In contrast, the fully tuned model demonstrates smooth behavior similar to a skilled worker. Although perfect shortest paths were not always achieved due to traces of exploration and minor oscillations were occasionally observed near the target, excessive oscillations were eliminated. This visually proves that the combination of energy penalties and hyperparameter tuning secured a level of stability applicable to real hardware.


(a) Seed 42


(b) Seed 55


(c) Seed 100


(d) Seed 200

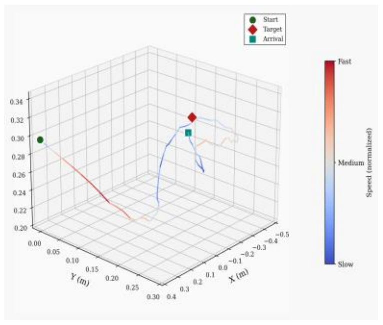

(e) Seed 300

Fig. 4. End-effector trajectories of the optimized agent. Visualization across five random seeds showing successful control behavior. The robot demonstrates smooth deceleration and stable convergence to the target, proving the effectiveness of the proposed tuning.

4. SAC Implementation

4.1 Algorithm Implementation Summary

The SAC implementation for this project is based on the CleanRL library, which provides a transparent codebase and integrated Weights & Biases (wandb) for experiment tracking. Several modifications were required to adapt the original CleanRL code to the UR5 environment and resolve critical technical issues.

First, a fix was implemented for NaN (Not a Number) action outputs. This issue occurred because the gymnasium.wrappers.ClipAction wrapper changes the environment action space bounds to infinity, causing the Actor network action scale calculation to fail. Consequently, the Tanh squashing process produced NaN values. To resolve this, the actual joint bounds were stored before applying the wrapper, and the Actor was modified to reference these finite physical limits.

Second, Q-value divergence was addressed to ensure numerical stability. Due to the scale of the curriculum reward function, cumulative rewards could reach high values over a 1,000-step episode, leading to exponential Q-value growth and gradient explosions. To stabilize training, Q-value clipping was applied to restrict predicted values between -1000 and 1000. Additionally, nan_to_num logic was integrated to convert any invalid numerical results to zero, maintaining the continuity of the learning process.

These modifications successfully resolved MuJoCo simulation instability warnings, such as huge value in QACC. These warnings were a direct result of the NaN action values being passed to the physics engine, which caused the robot joints to transition into physically impossible states. Fixing the action scaling issues inherently stabilized the physical simulation.

The agent neural network follows the standard CleanRL SAC architecture, utilizing an MLP with 256 by 256 hidden layers for both the Actor and Critic networks. The architecture uses ReLU activation functions for internal layers and a Tanh function for the final output to ensure bounded actions. The implementation also retains the Twin-Q structure, which uses two independent Critic networks to mitigate value overestimation bias by selecting the minimum of the two predicted Q-values.

Finally, the GoalAugmentWrapper was integrated to append the 3D target coordinates to the observation data. This ensures that the SAC agent operates within the same observation space as the PPO implementation, allowing for a fair and consistent performance comparison between the two algorithms.

4.2 Hyperparameter Tuning

Various hyperparameter combinations were tested to achieve stable learning for the SAC algorithm. This section details the tuning process, starting from the CleanRL defaults to the final configurations optimized for the UR5 environment.

4.2.1 Hardware Environment and Constraints

Experiments were conducted on a laptop equipped with an Intel Core Ultra 7 155H processor. While an attempt was made to use Intel Extension for PyTorch (IPEX) to utilize the integrated GPU (XPU), the MuJoCo physics simulation runs on the CPU, creating a data transfer overhead that neutralized any speed gains. Consequently, all experiments were performed on the CPU, which imposed fundamental limits on training speed.

For comparison, PPO achieved a Steps Per Second (SPS) of approximately 1,300 using 8 parallel environments, completing 1 million steps in 30 minutes. In contrast, SAC recorded significantly lower SPS due to the computational cost of replay buffer sampling and dual Critic network updates.

4.2.2 Step 1: Speed Optimization for Fair Comparison

Initially, an aggressive speed optimization was attempted to match PPO's 30-minute training time. The configuration included 8 parallel environments (num_envs=8), an update frequency of 256 (update_every=256), and 16 gradient steps per update. This setup improved SPS to approximately 270, allowing 1 million steps to finish in about an hour.

However, this resulted in severe performance degradation. Episodic returns stalled between 15 and 30, and WandB monitoring indicated signs of a local minimum, as the episode lengths remained short and returns showed no improvement over time. The low update frequency meant that the most recent experiences were not integrated quickly enough, and the diverse transitions from multiple environments diluted the learning signals.

4.2.3 Step 2: Reducing Parallelism (Model A)

Following the failure with 8 environments, the number was reduced to 4 while keeping the update frequency high for speed. The configuration was num_envs=4, update_every=256, and gradient_steps=16. This achieved a balance, with episodic returns rising to 50-70 and episode lengths reaching the maximum of 1,000 steps. SPS was approximately 150, allowing 700,000 steps in about 50 minutes.

This model was designated Model A and used for video analysis. Video analysis revealed that while the agent avoided collisions, it stopped moving in front of the obstacle box, which is a classic local minimum behavior.

4.2.4 Step 3: Reward Scaling Trials

To address Q-value divergence, reward scaling was tested. Applying a 0.01x scale stabilized Q-values but weakened the goal-reaching signal so much that the agent failed to learn reaching behavior. A 0.1x scale produced similar poor results.

The NormalizeReward wrapper was also tested, but it caused returns to become negative and training to become unstable, with an SPS of only 76. Ultimately, the original curriculum reward was kept, relying solely on Q-value clipping (-1000 to 1000) to prevent divergence.

4.2.5 Step 4: Entropy Coefficient (Alpha) Tuning

The entropy coefficient (alpha) controls the balance between exploration and exploitation. Initially, alpha=0.2 with autotune=False was used, but excessive exploration continued late into training, preventing policy convergence. Increasing alpha to 0.5 to escape local minima resulted in even more instability and increased collisions.

The final choice was alpha=0.2 with autotune=True. With autotuning, alpha started around 0.2–0.5 and automatically decreased to approximately 0.01 as training progressed, allowing a smooth transition from exploration to exploitation.

4.2.6 Step 5: Optimizing for Stability (Model B)

To escape the local minimum observed in Model A, a more conservative configuration was tested with increased update frequency. The configuration was num_envs=2, update_every=2, and gradient_steps=2, with learning_starts=5000.

This significantly increased the number of gradient updates per environment step, allowing the agent to learn from recent experiences more quickly. SPS dropped to approximately 35, requiring about 3 to 4 hours for 400,000 steps. This model is designated Model B and is used for comparative analysis in Section 4.3.

4.2.7 Time-Performance Trade-off Summary

A clear trade-off exists between training time and performance for SAC in a CPU environment. While PPO can handle 8 parallel environments without quality loss, SAC becomes unstable as the number of environments increases, requiring a lower number of environments and higher update frequencies for reliable results.

Due to these constraints, SAC experiments were limited to 400,000–700,000 steps instead of the 1 million steps used for PPO. Model A (num_envs=4) reached 700,000 steps in approximately 50 minutes, while Model B (num_envs=2) required approximately 3 hours for only 400,000 steps. This stark difference in training efficiency made direct 1-million-step comparisons with PPO impractical within the available time frame. Table 1 summarizes the key hyperparameter differences between the CleanRL baseline and our experimental configurations.

Table 1. Hyperparameter Comparison: CleanRL Default, Model A, and Model B

Parameter	CleanRL Default	Model A	Model B
num_envs	1	4	2
total_timesteps	1,000,000	700,000	400,000
learning_starts	5,000	5,000	5,000
update_every	1	256	2
gradient_steps	1	16	2
policy_lr	3e-4	3e-4	3e-4
q_lr	1e-3	3e-4	3e-4
alpha	0.2	0.2	0.2
autotune	True	True	True
batch_size	256	256	256
gamma	0.99	0.99	0.99
tau	0.005	0.005	0.005
Q-value clipping	None	(-1000, 1000)	(-1000, 1000)
Training Time	-	$\sim$50 min	$\sim$3 hours
SPS	$\sim$50	$\sim$150	$\sim$35

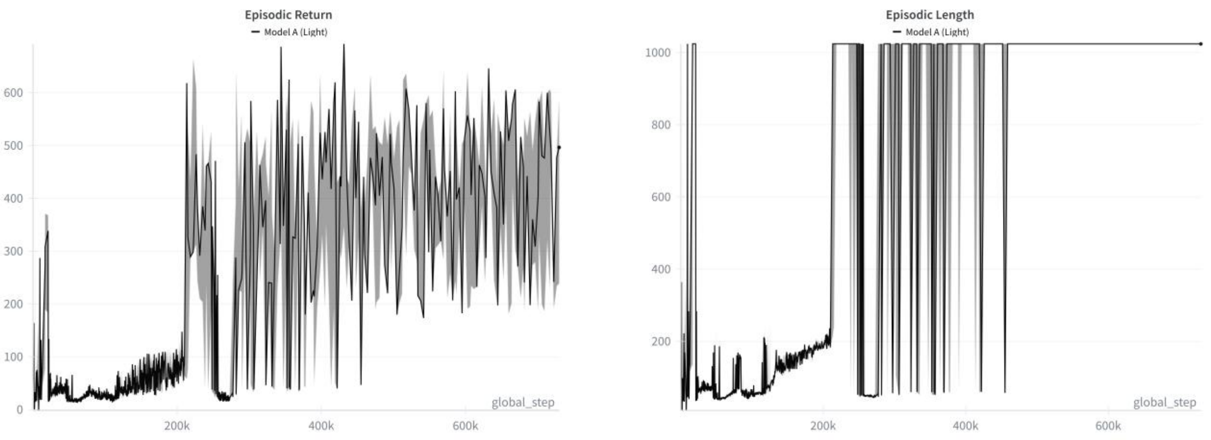
4.3 Experimental Results

This section analyzes the training outcomes of the SAC algorithm and compares them with PPO. Both Model A and Model B configurations are evaluated through learning curves, behavioral analysis, and quantitative and qualitative comparisons with PPO.

4.3.1 Model A (Fast Configuration)

Model A, which used 4 parallel environments and an update frequency of 256, exhibited short episode lengths and episodic returns around 70 during the early stages of training. This indicates frequent collisions or early terminations. However, at a certain point during training, the episode length suddenly jumped to the maximum value of 1,000 steps, and the episodic return rose to approximately 500.

The critical issue was that learning stagnated at this point and converged to this level without further improvement. As shown in Figure 5, both the episodic return and episode length curves clearly demonstrate this convergence behavior, which is characteristic of a local minimum.

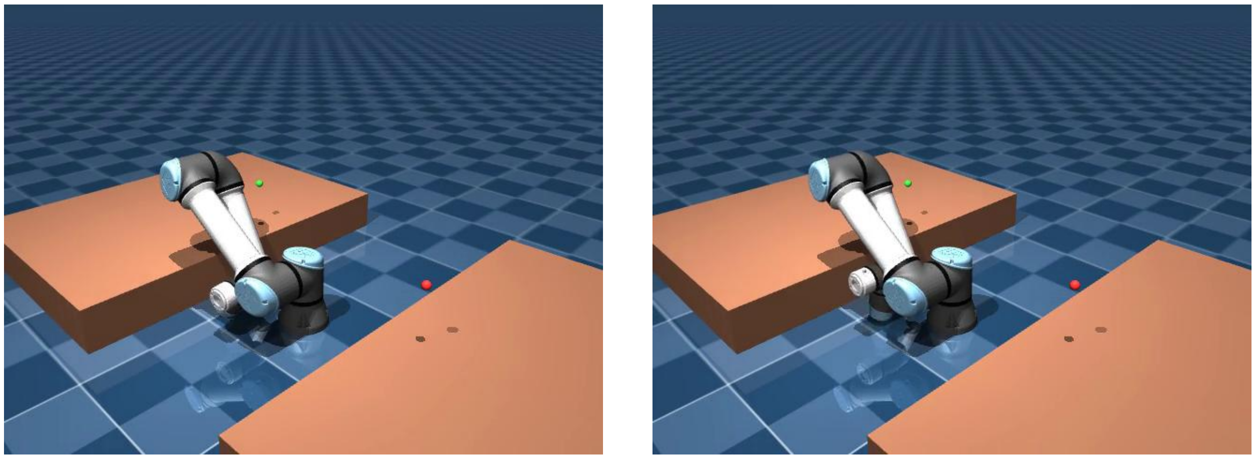


(a) Episodic Return vs. Global Step

(b) Episode Length vs. Global Step

Fig 5. Model A Learning Curves. (a) Episodic Return vs. Global Step, (b) Episode Length vs. Global Step. Both graphs show sudden convergence indicative of local minimum entrapment.

Video analysis confirmed that the agent learned to stop moving completely in front of the obstacle box and simply waited for 1,000 steps without any collision. This is a classic local minimum where the agent discovered that not moving results in no collisions and continuous time based rewards. Despite achieving a high episodic return of approximately 500, Model A recorded a success rate of 0 percent.

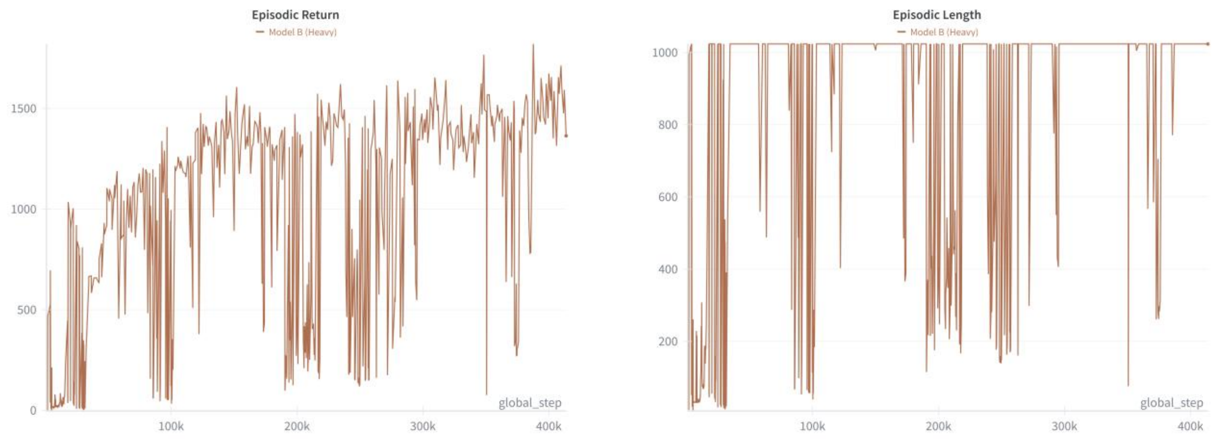


(a) Step 100 (b) Step 200

Fig 6. Model A Behavioral Snapshots. (a) Step 100 and (b) Step 200 of the same episode. The robot arm maintains an identical position in front of the obstacle box, demonstrating the local minimum behavior where the agent stopped moving entirely.

4.3.2 Model B (Heavy Configuration)

Model B, using 2 environments and an update frequency of 2, differed from Model A by achieving the maximum episode length of 1,000 steps from the early stages of training. It rapidly acquired collision avoidance behavior. The episodic return showed high variance but exhibited an overall upward trend, reaching approximately 1,500 by the end of training. This return is roughly three times higher than that of Model A.



(a) Episodic Return vs. Global Step (b) Episode Length vs. Global Step

Fig 7. Model B Learning Curves. (a) Episodic Return vs. Global Step, (b) Episode Length vs. Global Step. The return curve shows an upward trend with high variance, while episode length stabilizes at maximum early in training.

Despite the higher episodic return, Model B also recorded a success rate of 0 percent. While the curriculum reward function successfully guided the agent toward the goal direction, SAC failed to learn the precise control required for final goal reaching.

4.3.3 Comparison with PPO

PPO with the same curriculum reward function achieved approximately 90 percent success rate within just 30 minutes of training. In contrast, both SAC configurations recorded 0 percent, representing a significant performance gap.

Table 2. Performance Comparison between PPO and SAC configurations.

Metric	PPO (Final)	SAC Model A	SAC Model B
Training Time	∼30 min	∼50 min	∼3 hours
Total Steps	1,000,000	700,000	400,000
Final Episodic Return	∼300	∼500	∼1500
Episode Length	70-200	1000 (max)	1000 (max)
Success Rate	90%	0%	0%

The shorter episode length for PPO reflects successful task completion, where the agent reaches the goal quickly and terminates the episode. In contrast, the longer episode lengths of SAC indicate passive behavior, where the agent survives without reaching the goal.

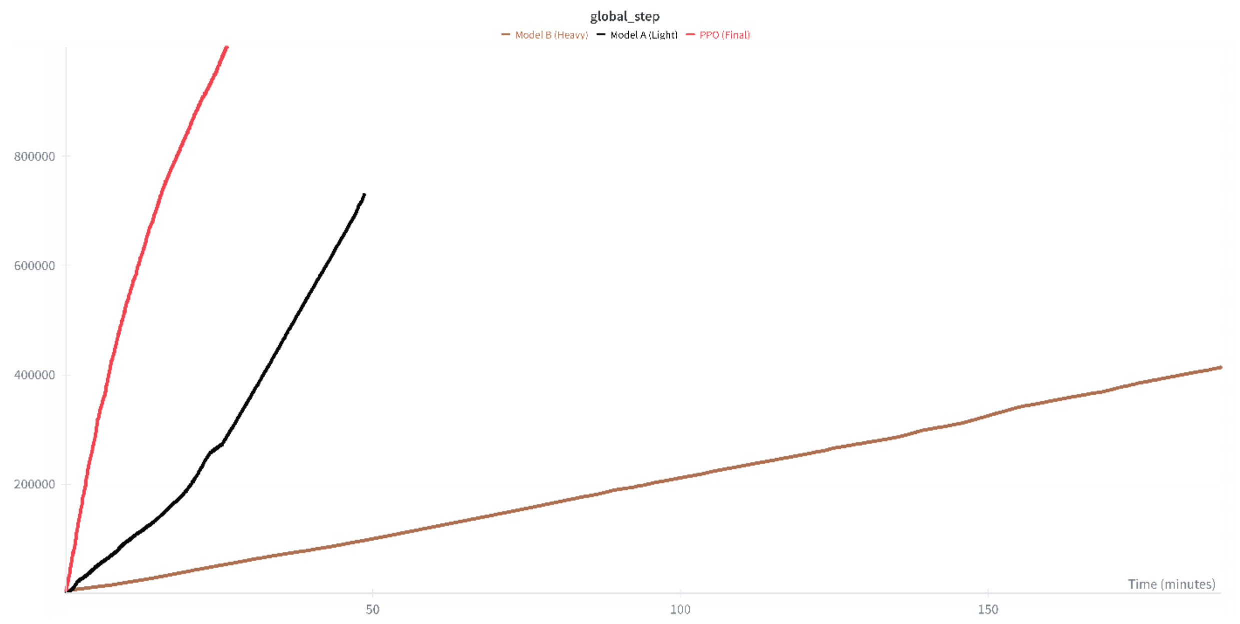


Fig 8. Training Efficiency Comparison. Global Steps vs. Elapsed Time for PPO and SAC. The slope difference illustrates the significant training speed gap between the two algorithms on CPU hardware.

5. Discussion

5.1 Reflections on Reward Functions and Hyperparameters

PPO Clip Epsilon Adjustment. The most impactful hyperparameter modification in this project was increasing the PPO clip epsilon from the default 0.2 to 0.318. This widened the allowed range of policy updates, enabling more aggressive learning and significantly improving the success rate. However, this adjustment relaxes the trust region constraint, which is a core principle of PPO, and may risk training stability. While positive results were obtained in this experiment, further validation is needed to determine whether this setting generalizes to other environments.

Tuning Limitations Due to Time Constraints. Hyperparameter tuning is a critical process in reinforcement learning experiments, but systematically testing all combinations was impractical given the time constraints. SAC experiments, in particular, required several hours per run, making the extensive exploration performed for PPO realistically impossible.

5.2 Practical Challenges of SAC Implementation

The SAC implementation process was far more challenging than anticipated. Theoretically, off-policy algorithms reuse past experiences through a replay buffer, promising higher sample efficiency and faster learning. In practice, however, the computational overhead from dual Critic network updates, replay buffer sampling, and entropy coefficient optimization significantly slowed down environment interaction speed. In the CPU environment, the SPS of SAC was approximately one-sixth that of PPO, requiring over six times longer to train for the same number of steps.

Even using the CleanRL code provided as a reference presented difficulties. Issues such as NaN actions caused by the ClipAction wrapper and Q-value divergence, which are problems that would not have occurred in the original benchmark environments, arose continuously in the UR5 environment. Despite assistance from state of the art LLM tools including Anthropic Claude, OpenAI Codex, Google's Antigravity, complete resolution was not achieved. This demonstrates that RL hyperparameter tuning remains a domain that is difficult to fully automate.

5.3 On-policy vs Off-policy: The Gap Between Theory and Reality

One interesting observation was that despite the entropy based exploration of SAC, it fell into a local minimum while PPO never did. Theoretically, the maximum entropy framework of SAC should guarantee stronger exploration, but this was not the case in practice. A possible hypothesis is that PPO, running with 8 parallel environments, naturally achieved exploration through diverse initial states, whereas SAC, with fewer environments, accumulated similar experiences in the replay buffer and lacked diversity. However, this is speculation, and additional experiments are needed to determine the exact cause.

5.4 Lessons Learned

PPO optimization proceeded as intended and yielded satisfactory results. SAC, on the other hand, presented challenges from the code execution stage, and with each run taking several hours, repeated experimentation itself became a struggle. SAC was chosen as a state of the art off-policy algorithm, but it may not have been the best choice for this specific environment.

Nevertheless, the learning gained from this process was valuable. Implementing and following the evolution of RL algorithms from DQN to DDPG, TD3, and SAC provided deep insights. The impact of each hyperparameter on learning became tangible through hands-on experience. Ultimately, this project reinforced that theoretical understanding alone is insufficient in reinforcement learning. Direct tuning and experiencing failure are essential.

5.5 Future Work

Re-experimentation in GPU Environment. The CPU constraint was the biggest bottleneck in this project. Training SAC for over 1 million steps with GPU acceleration may yield different results.

Automated Hyperparameter Tuning. As AI agent technology advances, building workflows that automatically test hyperparameter combinations and record results will become easier. The productivity gap between researchers who can leverage such tools and those who cannot is expected to widen.

TD3 or DDPG Trials. The entropy term in SAC may not have been suitable for this task. Comparative experiments with TD3 or DDPG, which learn deterministic policies without entropy, would be worthwhile.

In conclusion, this project made it clear that competitiveness in RL research comes not from who is smarter but from who persistently tests many things and logically connects the results. From this project, it became clear that success in reinforcement learning depends less on intelligence and more on the tenacity to keep experimenting and the patience to systematically analyze failures.

6. Reference

[1] S. M. Kakade and J. Langford, “Approximately optimal approximate reinforcement learning,” in *Machine Learning: Proceedings of the Nineteenth International Conference on Machine Learning (ICML 2002)*, C. Sammut and A. G. Hoffmann, Eds., Sydney, Australia, Jul. 8–12, 2002, pp. 267–274.

[2] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz, “Trust Region Policy Optimization,” in *Proceedings of the 32nd International Conference on Machine Learning (ICML 2015)*, *Proceedings of Machine Learning Research*, vol. 37, Lille, France, pp. 1889–1897, 2015.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms,” arXiv:1707.06347, 2017.

[4] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor,” in *Proceedings of the 35th International Conference on Machine Learning (ICML 2018)*, *Proceedings of Machine Learning Research*, vol. 80, pp. 1861–1870, 2018.