POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Integrating Bayesian Optimization, Reinforcement Learning & Language Models in Robotics

Group Project

Mohsen Ghasemi
Aidin Latifi
Shiyuan Liu

Professor:
Loris Roveda

Contents

Bayesian Optimization

1.1 Introduction

Nowadays, robots are required to adapt to (partially) unknown situations, being able to optimize their behaviors through continuous interactions with the environment. In such a way, robots can achieve a high level of autonomy which allows them to face unforeseen situations. This project presents a Bayesian Optimization (BO) framework to auto-tune both dynamic compensation parameters (e.g., equivalent link masses) and joint-level PID gains for trajectory tracking. BO is known to be efficient in terms of number of function evaluations for derivative-free optimization, making it particularly suitable for developing an efficient and easily-applicable approach that can be implemented in real industrial plants.

Two strategies are proposed for parameter tuning. In the first strategy, the equivalent link-mass parameters and the PID gains are jointly optimized according to the trajectory tracking objective using Bayesian optimization. The second strategy is inspired by classical control design strategies for manipulators and consists of two stages. In the first stage, the link masses characterizing the feedback linearization and the feedforward action are tuned with the objective of decoupling the joint-level robot degree-of-freedom (DoFs) by compensating the gravitational and the Coriolis force. In the second stage, PID gains are tuned according to the original trajectory-tracking objective, while keeping link mass parameters fixed to the values optimized in the first stage.

This comparison between the one-stage and two-stage procedures is conducted in terms of tracking performance, providing insights into the most effective approach for robot control parameter optimization.

1.2 Method

This section is devoted to the problem formulation, where the model of the robot dynamics and the considered control architecture are described. Then the details of the Bayesian optimization method for PID parameter tuning will be presented.

1.2.1 Robot Dynamics

The robot manipulator dynamics are modeled by the following second-order nonlinear differential equation:

$$M(q,\theta)\ddot{q} + C(q,\dot{q},\theta)\dot{q} + G(q,\theta) + F(\dot{q}) = \tau, \tag{1.1}$$

where:

- $q \in \mathbb{R}^n$ is the joint position vector,
- $\dot{q}, \ddot{q} \in \mathbb{R}^n$ are the joint velocity and acceleration vectors,
- $M(q,\theta) \in \mathbb{R}^{n\times n}$ is the inertia matrix,
- $C(q,\dot{q},\theta)\dot{q} \in \mathbb{R}^n$ represents the Coriolis and centrifugal forces,
- $G(q,\theta) \in \mathbb{R}^n$ is the gravitational torque vector,
- $F(\dot{q}) \in \mathbb{R}^n$ accounts for joint friction,
- $\tau \in \mathbb{R}^n$ is the control input (joint torques),
- θ is the vector of unknown link mass parameters.

1.2.2 Control Architecture

Figure 1.1 shows the proposed control scheme and tuning strategy, where the parameters of the joint-level PID controller and the equivalent link-mass parameters used by the feedback linearization and feedforward controller are optimized through Bayesian optimization in order to achieve high performance trajectory tracking.The control input τ is composed of three parts:

$$\tau = \tau_{\text{FF}} + \tau_{\text{PID}} + \tau_{\text{FL}}, \tag{1.2}$$

where:

- τ_{FF} is the feedforward term,
- τ_{PID} is the joint-level PID control,
- τ_{FL} is the feedback linearization term.

These are defined as:

$$\tau_{\text{FF}} = M(q,\theta)\ddot{q}_{\text{ref}}, \tag{1.3}$$

$$\tau_{\text{PID}} = K_p e + K_d \dot{e} + K_i \int e\, dt, \tag{1.4}$$

$$\tau_{\text{FL}} = C(q,\dot{q},\theta)\dot{q} + G(q,\theta), \tag{1.5}$$

with:

- q_{ref} the desired joint position,
- $e = q_{\text{ref}} - q$ the position tracking error,
- $K_p, K_d, K_i \in \mathbb{R}^{n\times n}$ are the proportional, derivative, and integral gain matrices.

The feedforward term τ_{FF} aims to track the desired acceleration, the feedback linearization term τ_{FL} compensates for the robot dynamics (Coriolis and gravity), and the PID controller τ_{PID} improves trajectory tracking performance in closed-loop.

This structure allows the controller to be tuned via optimization of both the link-mass parameters θ and the PID gains, enabling effective trajectory tracking performance despite the unknown dynamics.

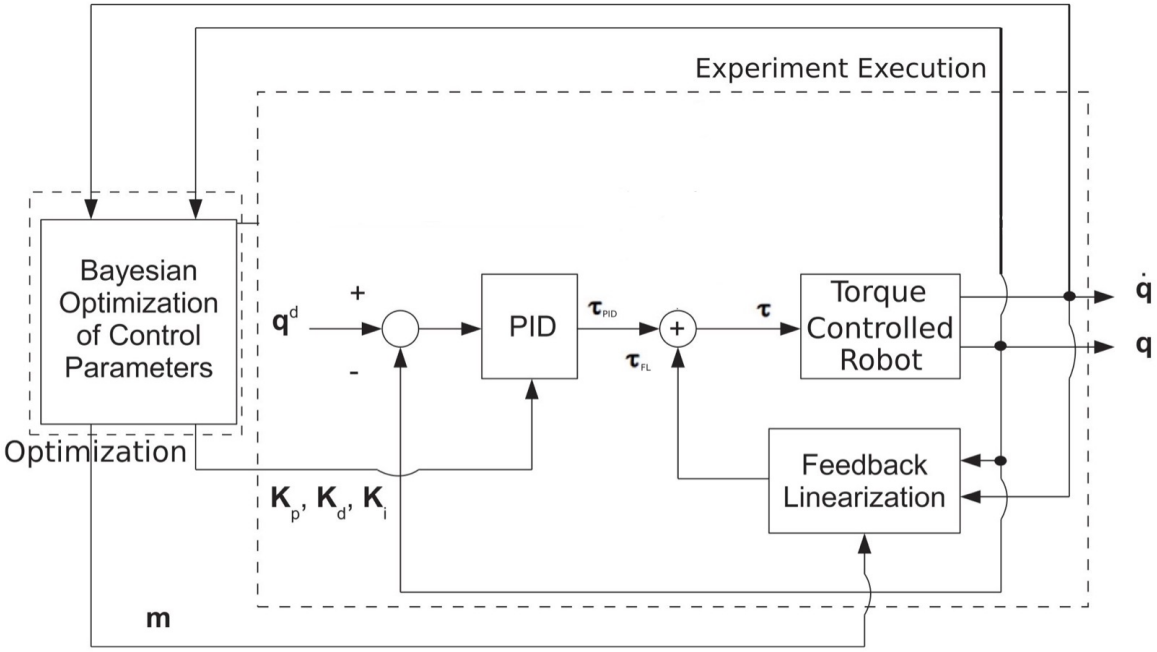


Figure 1.1: Control scheme

1.2.3 One-Stage Optimization Strategy

The one-stage approach simultaneously tunes both the dynamic parameters of the robot (i.e., link masses) and the feedback control gains. Unlike the two-stage method that decouples mass identification from controller tuning, this integrated strategy jointly optimizes all parameters in a Bayesian Optimization process. As such, it explores a higher-dimensional parameter space.

In this formulation, the BO optimizer searches over a joint parameter vector that includes both PID gains $\{K_{p_i}, K_{i_i}, K_{d_i}\}_{i=1}^{7}$ and mass parameters $\{m_1, m_3, m_5, m_7\}$. The cost function, implemented through the `obj_PID_panda` routine, evaluates the closed-loop trajectory tracking performance under the influence of both the estimated model and the PID controller.

The performance metric penalizes large tracking errors and instability events, encouraging configurations that yield smooth, accurate motion. Specifically, the objective function is defined as:

$$J = w_{\text{PID}} \sum_j |\tau_{\text{PID},j}| + w_e \sum_j |\bar{e}_j| + w_{\dot{e}} \sum_j |\bar{\dot{e}}_j| + w_{e_{\max}} \sum_j |e_{\max,j}| + w_{\dot{e}_{\max}} \sum_j |\dot{e}_{\max,j}| + w_L L, \tag{1.6}$$

where $\bar{e}_j$ and $\bar{\dot{e}}_j$ denote the normalized mean position and velocity errors for joint j, $e_{\max,j}$ and $\dot{e}_{\max,j}$ represent the corresponding maximum errors, $\tau_{\text{PID},j}$ is the control input from the PID controller, and L is a penalty term that becomes large in the presence of instability or constraint violations (e.g., exceeding joint error thresholds or producing NaN states).

Although this method may require more function evaluations due to the increased dimensionality, it eliminates the dependency on pre-identified mass parameters and can directly exploit the interaction between dynamics and control gains to achieve improved overall performance.

1.2.4 Two-Stage Optimization Strategy

This strategy decouples the optimization of robot dynamics parameters and feedback control gains into two sequential stages. Each stage targets a different aspect of the control architecture, thereby reducing the dimensionality of the search space at each step and simplifying convergence.

Stage 1: Link-Mass Parameter Identification

The first stage aims to calibrate a simplified dynamic model of the robot by optimizing a subset of effective link-mass parameters: m_1, m_3, m_5, and m_7. These parameters can improve the accuracy of the feedforward torque term $\tau_{\text{ff}} = B(q)\ddot{q}_d$ and the gravity compensation $\tau_{\text{comp}} = g(q)$. We implement this stage using the `obj_M_panda` cost function, which evaluates the tracking performance of an open-loop controller composed solely of feedforward and gravity terms.

The BO process searches for the optimal mass values within predefined bounds using the `bayesopt` function in MATLAB. The cost function penalizes large acceleration errors and early instability. Specifically, the objective is defined as:

$$J = \sum_i \left(\left\| \ddot{e}^{(i)}_{\max} \right\| + 10 \left\| \ddot{e}^{(i)}_{\text{mean}} \right\| \right) + L, \tag{1.7}$$

where $\ddot{e}$ is the acceleration tracking error, and L is a penalty term that grows if the joint error exceeds a threshold or diverges. A sinusoidal reference trajectory is used to excite all joints uniformly.

Stage 2: PID Gain Optimization

Once a sufficiently accurate dynamic model has been obtained, the second stage focuses on tuning the diagonal PID feedback controller, assuming the identified mass parameters are fixed. For each joint $i = 1, \ldots, 7$, a separate BO problem is defined to optimize its PID gains: K_p^i, K_i^i, and K_d^i, as implemented in the `obj_PID_panda` function.

The inner-loop controller combines model-based compensation and joint-wise PID feedback as follows:

$$\tau = \tau_{\text{PID}} + \tau_{\text{comp}}, \quad \text{with} \quad \tau_{\text{PID}} = B(q) \left(K_p e + K_d \dot{e} + K_i \int e \, dt \right), \tag{1.8}$$

The cost function for PID tuning incorporates multiple performance aspects, including normalized position and velocity tracking errors, integrated error, and an instability penalty, which is basically the same as the equation 1.6

The BO for each joint uses the `expected-improvement-plus` acquisition function and is allowed up to 50 evaluations, ensuring efficient convergence even under limited evaluation budgets.

1.3 Results and Analysis

This section presents experimental results on the application of the proposed auto-tuning procedure to the FRANKA Emika 7-DoFs manipulator (Figure 1.2).

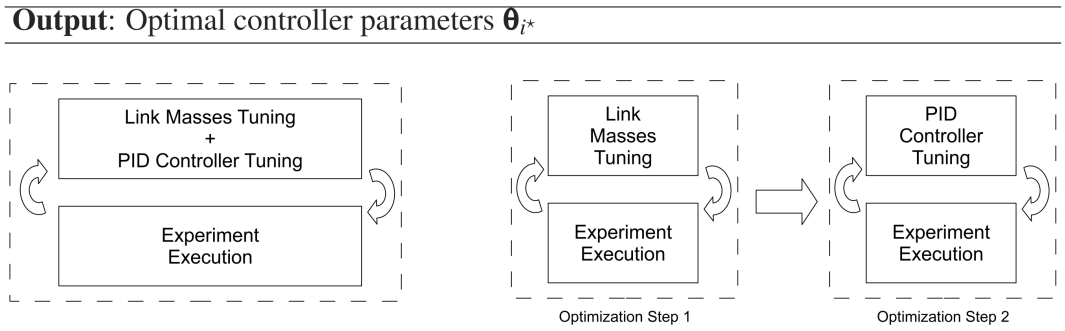


Figure 1.2: Bayesian optimization for robot control parameter design: one-stage tuning (left panel) and two-stage tuning (right panel).

1.3.1 Use case trajectories

The following trajectories are considered to assess the performance of the model:

$$q_r(:, ii) = q_0(ii) + \frac{10\pi}{180}\sin(2\pi \cdot \text{freq} \cdot \text{time}) \tag{1.9}$$

$$\dot{q}_r(:, ii) = 2\pi \cdot \text{freq} \cdot \frac{10\pi}{180}\cos(2\pi \cdot \text{freq} \cdot \text{time}) \tag{1.10}$$

$$\ddot{q}_r(:, ii) = -\left(2\pi \cdot \text{freq}\right)^2 \cdot \frac{10\pi}{180}\sin(2\pi \cdot \text{freq} \cdot \text{time}) \tag{1.11}$$

1.3.2 One stage tuning

The Bayesian optimization is initialized with $n_{in} = 20$ initial experiments with randomly generated parameters, followed by additional 150 iterations. The performance index J is minimized over the trajectory execution time T. Tests are interrupted if the joint position error is larger than 0.1 radians for safty reasons and it will give extra penalty of the position error is larger than 0.02.

The obtained results are summarized in Table 1.1, which shows link-mass parameters, the PID gains.

Parameter	Value	Parameter	Value	Parameter	Value
M1	1.7775	Kp3	9886.2	Kp6	7004.3
M3	3.4381	Ki3	483.35	Ki6	414.52
M5	5.9006	Kd3	129.22	Kd6	105.09
M7	2.4229	Kp4	8626.6	Kp7	9306.4
Kp1	5383.6	Ki4	919.37	Ki7	818.77
Ki1	97.617	Kd4	50.096	Kd7	84.677
Kd1	295.51	Kp5	8224.2		
Kp2	9323	Ki5	426.9		
Ki2	127	Kd5	51.539		
Kd2	78.138				

Table 1.1: Optimized masses and controller parameters

Figure 1.3 shows the cost over optimization iteration. Due to the penalty terms L in equation (1.6), a high performance cost $\mathcal{J}$ is obtained in experiments where the task is interrupted, either because of an unstable behavior of the robot or because the maximum joint position error $\bar{e}$ is exceeded. As expected, this occurs primarily during the first initial iterations, where the control parameters are randomly sampled.

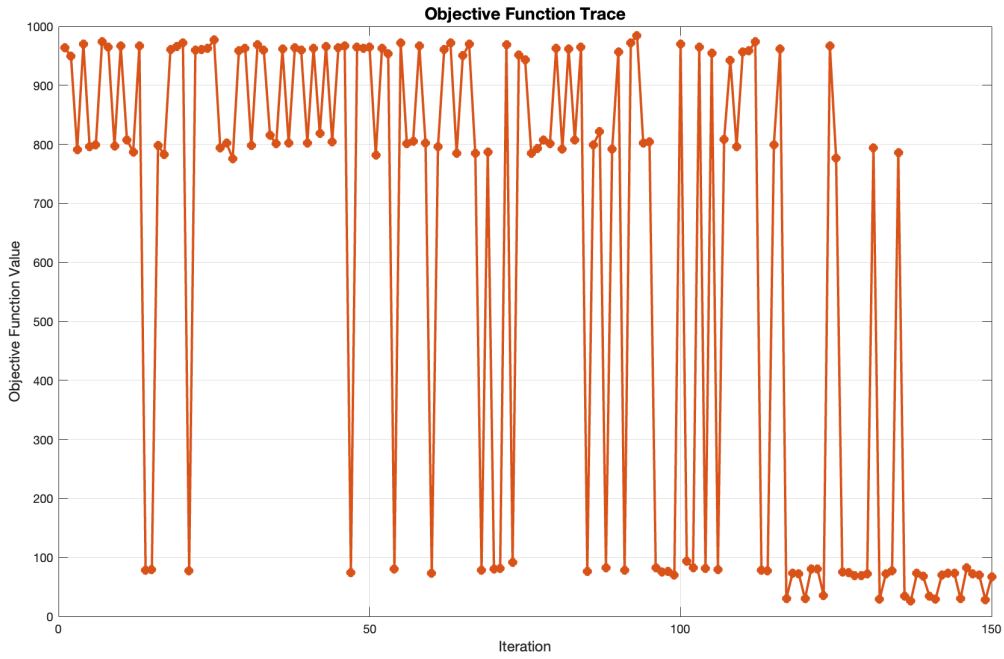


Figure 1.3: Cost evolution over optimization iterations for one stage method

Figure 1.4 to 1.6 shows the simulation results with the optimized parameters. Figure 1.4 presents the reference versus measured joint positions, showing close adherence of the measured values to the desired sinusoidal trajectories for all seven joints. Similarly for the velocities; after a brief initial transient, the measured velocities accurately track their respective reference profiles.

The joint position error, calculated as the difference between reference and measured positions for each joint, is depicted in Figure 1.5, indicating that errors remain bounded in degrees throughout the simulation. Furthermore, Figure 1.6 illustrates the control torques applied to each joint, confirming that they operate within the predefined limits, ensuring physically feasible control actions.

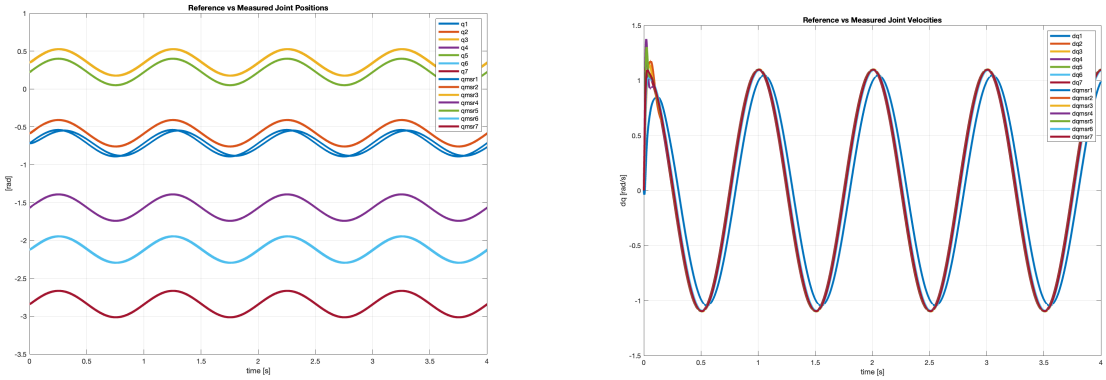


Figure 1.4: Reference tracking performance

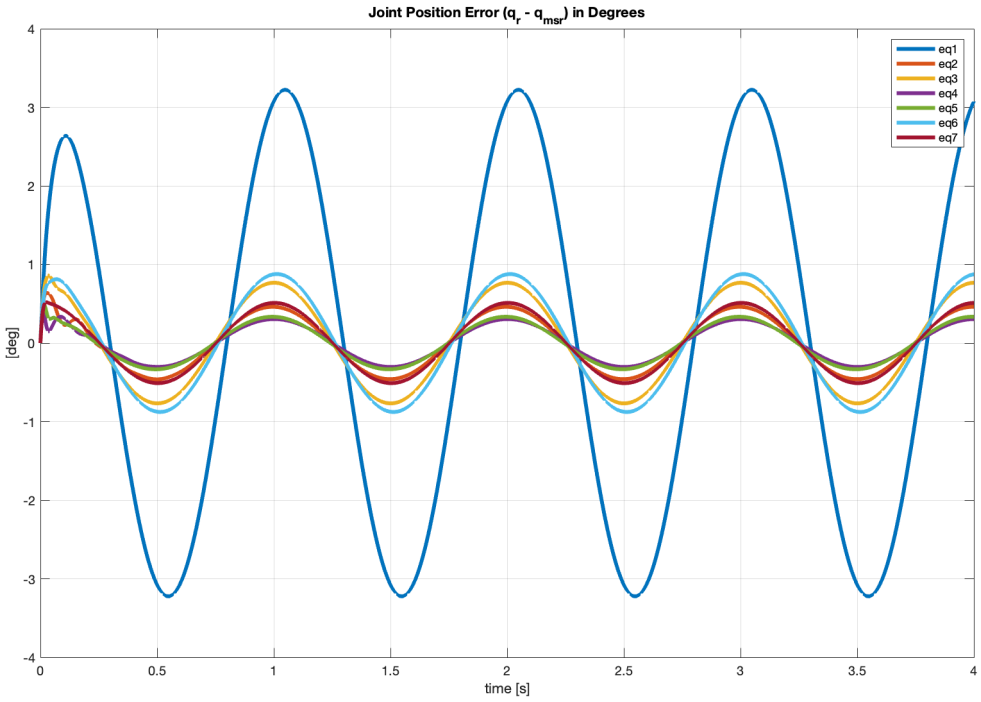


Figure 1.5: Position reference tracking error

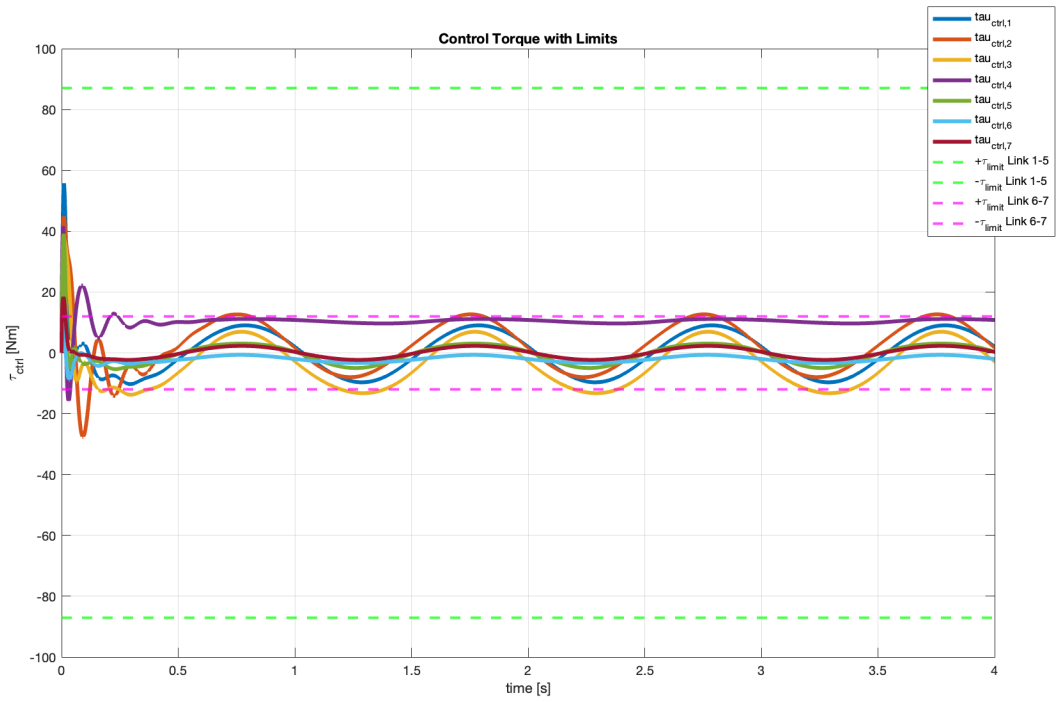


Figure 1.6: Controller torque of each joint

1.3.3 Two stage tuning

As it has been described in the previous section, at the first stage, the equivalent link-mass parameters m characterizing the feedback linearizator and the feedforward action are tuned by minimizing the performance cost. The PID gains are not optimized at this stage. At the second stage, once the link-mass parameters are optimized, the PID controllers are designed. The Bayesian optimization is initialized with $n_{in} = 20$ initial experiments with randomly generated PID parameters, and terminated after additional 50 iterations. The cost trace versus optimization iterations has been shown in Figure 1.7

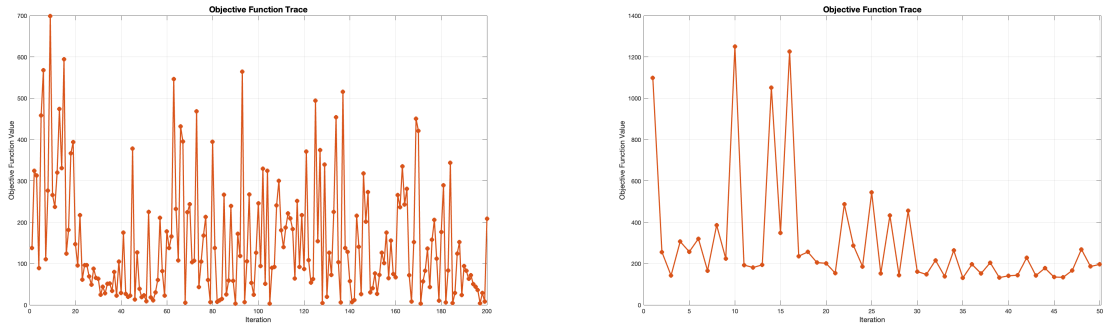


Figure 1.7: Cost evolution over optimization iterations for two stage method, (Left) mass optimization, (right) PID optimization

The obtained results are summarized in Figure 1.8, which shows the evolution of the link-mass parameters and Figure 1.9 the PID gains over the Bayesian optimization iterations are plotted.

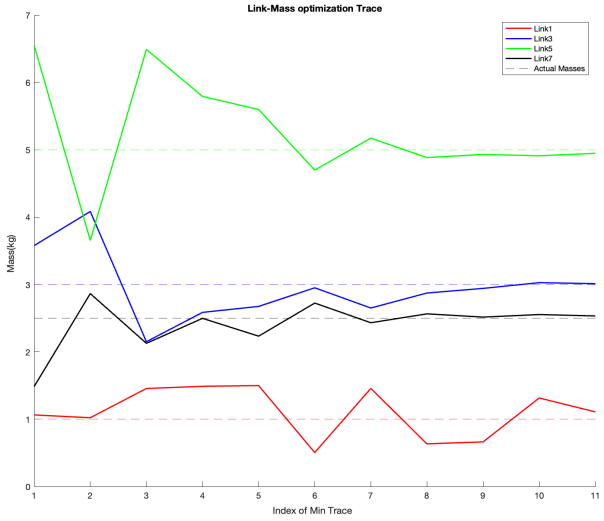


Figure 1.8: Best Link mass trace during optimization

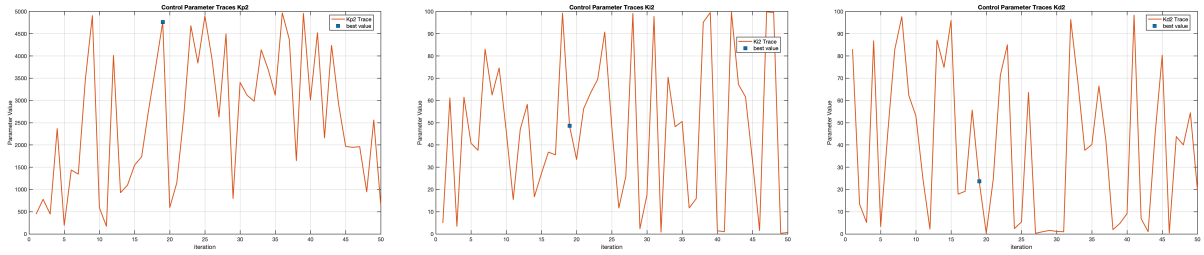


Figure 1.9: An example of parameters (Kp2,Ki2,Kd2) trace over iterations

To further investigate the impact of the cost function design, we conducted two experiments: one using the baseline cost function introduced in Section 1, and another with an additional penalty term on the control effort, specifically targeting the PID torque magnitude.

The comparison between these two experiments highlights the trade-offs introduced by the additional cost component. As shown in Table 1.2 and Figures 1.10 through 1.12, incorporating the PID torque penalty leads to a noticeable reduction in the overall control torques applied by the system. This reduction is beneficial in terms of energy efficiency and hardware longevity, as it discourages unnecessarily aggressive control actions.

However, this benefit comes at a slight cost in positional accuracy. The positional tracking error increased modestly in the experiment with the torque cost, reflecting the controller's more conservative behavior. Despite this, the overall tracking performance remains acceptable. Notably, the added cost also helped reduce oscillations in the early phase of the velocity reference tracking, as seen in Figure 1.12. This indicates improved transient behavior and potentially smoother operation at the beginning of the trajectory execution.

Table 1.2: Comparison of PID Parameters across Two Result Sets

Controller	Without PID cost			With PID cost		
	Kp	Ki	Kd	Kp	Ki	Kd
1	4809.0	10.263	26.695	1495.8	99.490	10.792
2	4761.3	48.595	23.686	4599.6	65.726	59.268
3	4383.3	96.598	17.241	4999.8	45.511	87.207
4	4926.8	99.304	17.412	4999.1	16.622	52.989
5	4999.8	98.853	7.413	4863.3	61.287	33.677
6	4840.7	84.128	16.149	4998.8	85.359	28.250
7	4672.3	44.997	14.295	4992.2	81.077	20.094

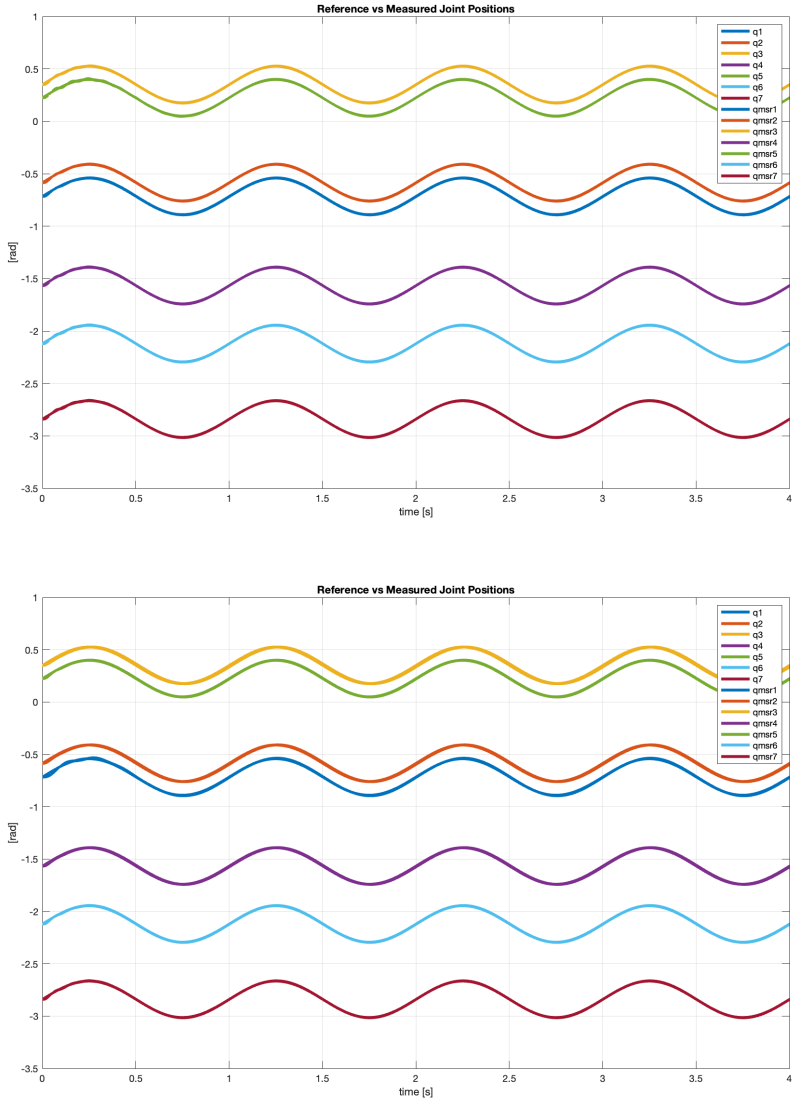


Figure 1.10: The Position reference tracking comparison, (Up) is the results with the cost introduced in the section 1, (Down) is the results with the added torque cost

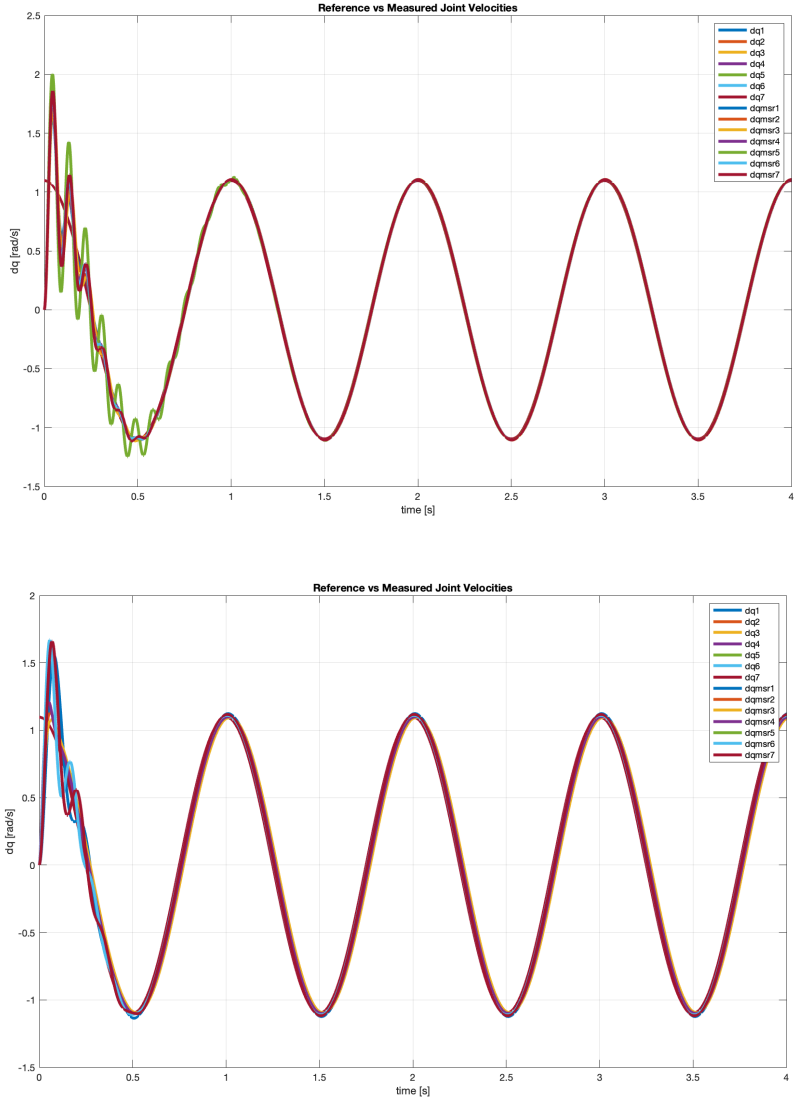


Figure 1.11: The velocity reference tracking comparison, (Up) is the results with the cost introduced in the section 1, (Down) is the results with the added torque cost

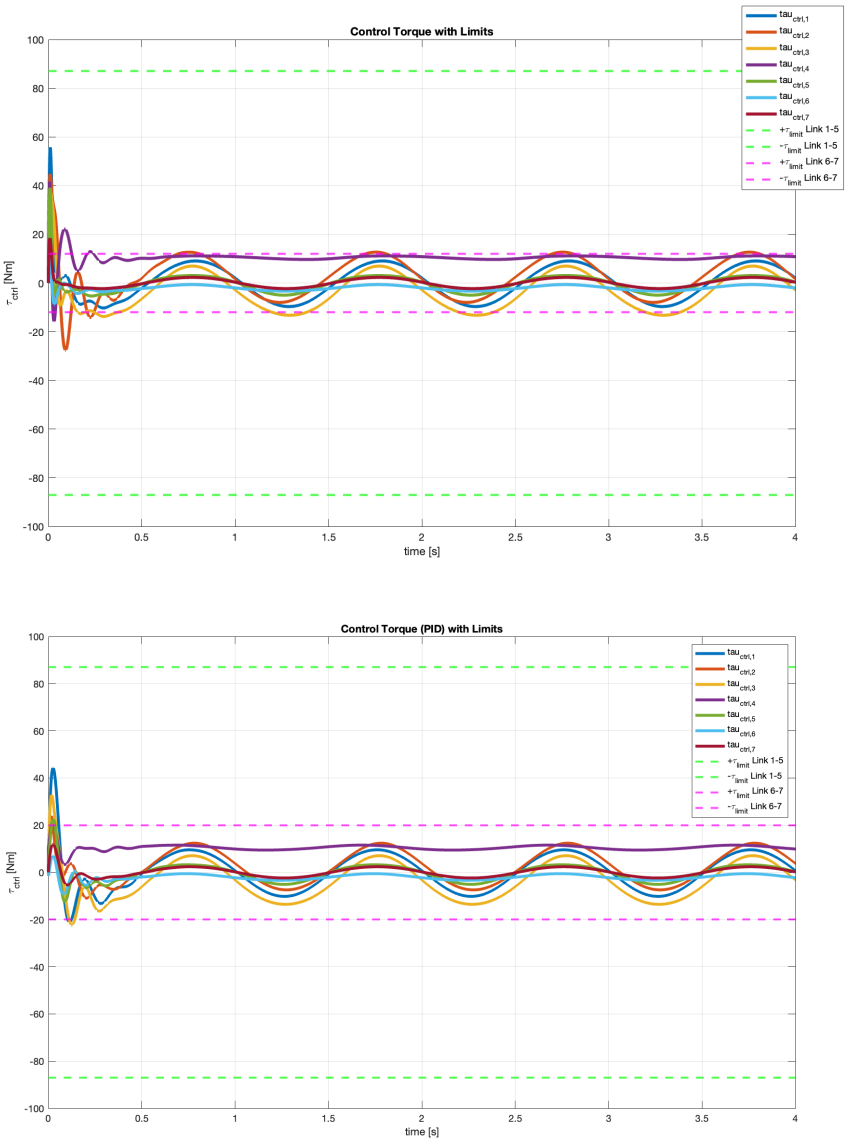


Figure 1.12: The controllers applied torque comparison, (Up) is the torques results with the cost introduced in the section 1, (Down) is the torques results with the added torque cost

1.4 Conclusion

This chapter presented a Bayesian Optimization (BO) framework for the auto-tuning of robot control parameters, encompassing both dynamic compensation parameters like equivalent link masses and joint-level PID gains, to enhance trajectory tracking performance. Two distinct optimization strategies were proposed and evaluated: a one-stage approach that jointly optimizes all parameters, and a two-stage method that sequentially tunes link masses followed by PID gains. The experimental validation on a FRANKA Emika 7-DoFs manipulator indicated that both strategies can achieve effective trajectory tracking. The one-stage strategy tackles a higher-dimensional parameter space directly, while the two-stage strategy simplifies the problem by decoupling parameter identification from controller tuning. Further investigation into the two-stage method

involved comparing a baseline cost function with one that included an additional penalty on PID torque magnitude. This comparison revealed that penalizing control effort led to reduced overall control torques and smoother transient velocity tracking, albeit with a modest increase in positional tracking error, demonstrating a trade-off between energy efficiency and tracking accuracy. Ultimately, the study underscores Bayesian Optimization's efficacy for efficiently tuning complex robot control systems.

Reinforcement learning

2.1 Introduction

This work explores the use of reinforcement learning (RL) for the control of an automated material handling system (AMS) with a specific emphasis on package sorting. The setup is a 5×5 grid of actuated modules that are used to sort packages of varying sizes to their respective exit points on a conveyor belt. Automated package sorting involves a variety of operational complexities in logistics and manufacturing facilities. The system must process packages of varying sizes, ensure a steady throughput flow, and achieve precise sorting results. Conventional control methods tend to utilize pre-established rules that are too rigid to accommodate shifting conditions or to simultaneously optimize for various objectives.

In this work the controller must learn to:

- In one case sort the packages to the middle of the exit
- In another case separate packages into correct exit points based on their sizes.
- Ensure continuous flow in the system with no packet collision
- Tune for sorting accuracy and processing speed

The control issue has a large-dimensional action space (50 dimensions altogether), and the state space of package positions and features is continuous and it is necessary to coordinate various modules to achieve purposeful package movement. The system must also handle dynamic package-to-package interactions, i.e., collision avoidance. We train our controller using a Proximal Policy Optimization (PPO) algorithm, with careful attention to the reward function and observation space to promote the target sorting behavior. Our reward function that we employ includes a number of aspects: remaining in the target lane, maximizing throughput, collision avoidance, and correct exits. This work demonstrates the applicability of recent reinforcement learning methods to real-world industrial control problems that would otherwise involve intricate, human-designed solutions. Following is an account of how we proceeded with this, the result of our experiment, and the conclusions drawn from using reinforcement learning to this material handling issue.

2.2 Method

In this section, we present the overall methodology used to train and evaluate the reinforcement learning agent. We first describe the simulation environment and task setup, followed by the algorithmic approach, including reward structure and network architecture. We then detail the training procedure, including hyperparameters and data collection. Finally, we outline the evaluation strategy used to assess performance and generalization.

2.2.1 Environment Characteristics

The AMS is consists of a 5×5 array of actuated modules in a two-dimensional matrix. Each module is a square with a side length of $d_{AMS} = 0.2$ m, forming a total sorting area of 1.0 m $\times$ 1.0 m. Each module is identified by its coordinates (i, j) where $i \in \{1, 2, 3, 4, 5\}$ indicates the row (increasing in the y-axis direction) and $j \in \{1, 2, 3, 4, 5\}$ indicates the column (increasing in the x-axis direction).

At the end of the AMS grid (beyond row $i = 5$), a conveyor belt moves at a constant velocity of $v_{treadmill} = 0.6$ m/s in the positive y-direction. This conveyor serves as the exit zone where packages complete their sorting journey.

Package Generation and Attributes

Packages are produced at regular intervals at the entry of the system (at $y \approx 0$ m) with the following characteristics:

- Generation Frequency: New packages are created every 0.75 seconds, normally in pairs.
- Package Size: The diameters of the packages (d_{boxes}) are randomly generated within a range having a minimum of 0.05 m and a maximum of 0.4 m. These diameters are of two types: small packages ($d_{boxes} < 0.225$ m) and large packages ($d_{boxes} \geq 0.225$ m).
- Initial Positions: The boxes are produced in the bottom half of the grid with small random y-coordinates ($y_{boxes} \approx 0.001$ m with slight random variations). The x-coordinates are spread out so that there is no overlap between boxes:
  - The first package is located in the interval $x \in [d_{box}/2, 0.5]$ m.
  - The second package is in the range $x \in [0.5, 1.0 - d_{box}/2]$ m.
  - There is a minimum distance of $(d_{box1}/2 + d_{box2}/2 + 0.1)$ m between packages.

System Dynamics

The system evolves in discrete time steps of $dt = 0.01$ seconds according to the following dynamics:

- AMS Module Control: Each module can be independently controlled with two parameters:
  - Rotation angle (θ) in the range $[-\pi/4, \pi/4]$ radians, determining the direction of actuation

  - Linear velocity (v) in the range $[0.5, 2.5]$ m/s, determining the speed of actuation

- Package Movement: When a package is on an AMS module at position (i, j), its velocity components are determined by:

$$v_{x,box} = v_{AMS}(i, j) \cdot \sin(\theta_{AMS}(i, j)) \tag{2.1}$$
$$v_{y,box} = v_{AMS}(i, j) \cdot \cos(\theta_{AMS}(i, j)) \tag{2.2}$$

  The package's position is updated according to:

$$x_{box}(t + dt) = x_{box}(t) + v_{x,box} \cdot dt \tag{2.3}$$
$$y_{box}(t + dt) = y_{box}(t) + v_{y,box} \cdot dt \tag{2.4}$$

- Collision Detection and Resolution: The system continuously monitors for overlapping packages. When a collision is detected (distance between package centers is less than the sum of their radii plus a tolerance of 0.005 m), the positions are adjusted to eliminate overlap. This is implemented through a displacement resolution:
  - For horizontal collisions, the overlapping packages are pushed apart in the x-direction
  - For vertical collisions, the overlapping packages are pushed apart in the y-direction
  - The displacement is distributed equally between the two packages plus a small tolerance

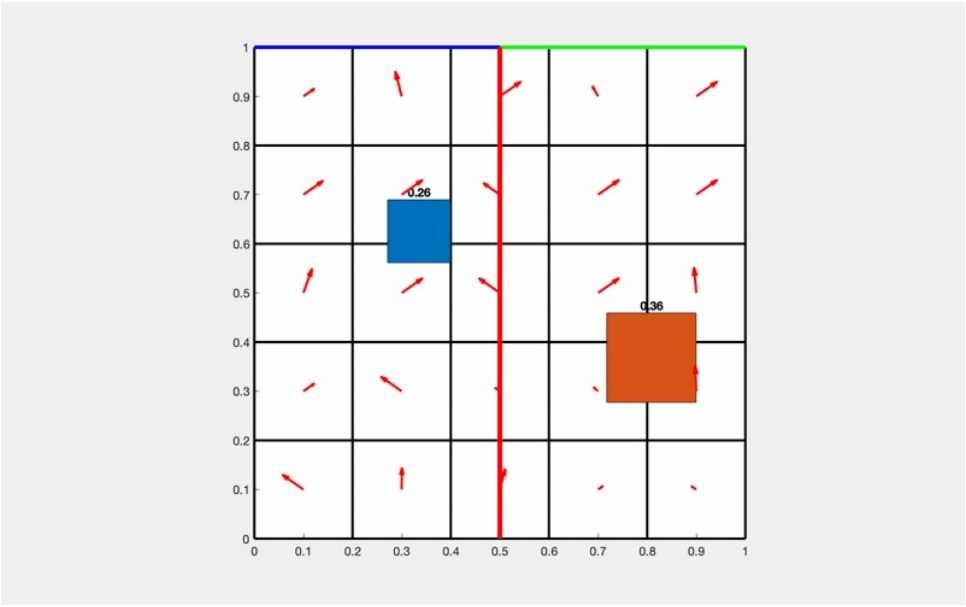


Figure 2.1: The arrows represent the actuations which is basically the velocity and rotation of actuators in each square

Terminal Conditions

An episode ends when all packages (maximum of 10) have exited the system (that is, when $y_{box} > 1.0$ m for all packages). The effectiveness of the sorting is evaluated based on

how accurately packages reached their designated exit locations and how efficiently they moved through the system.

The challenge lies in coordinating the 25 AMS modules to guide packages of varying sizes through the system while satisfying the sorting objectives and operational constraints. This requires a control policy that can process high-dimensional state information and make appropriate adjustments to module rotations and velocities in real-time.

Sorting Objective

In this project we have two separate objectives. In the first case the objective is to sort packages by maintaining a minimum gap of 0.2 m and directing them to exit near the center of the conveyor belt, while maximizing throughput (packages per hour) and ensuring package safety. In the second case, however, the objective of the system is to sort packages according to their size:

- Small packages ($d_{box} < 0.225$ m) should exit the system at $x \approx 0.2$ m
- Large packages ($d_{box} \geq 0.225$ m) should exit the system at $x \approx 0.6$ m

Secondary objectives include:

- Maintaining system throughput (packages should move through the system efficiently)
- Avoiding package congestion (packages should maintain appropriate spacing)
- Minimizing package collisions (packages should follow smooth trajectories)

To approach this goal, a class of **Reinforcement learning** method has been used which in the next section will be explained.

2.2.2 Proximal Policy Optimization

Proximal Policy Optimization (PPO) is an on-policy, policy-gradient reinforcement learning algorithm suitable for environments with either discrete or continuous action spaces. It alternates between interacting with the environment using the current stochastic policy and optimizing a clipped surrogate objective using stochastic gradient descent. The clipped objective improves training stability by restricting the extent of policy updates, thereby simplifying and accelerating learning compared to methods like Trust Region Policy Optimization (TRPO).

PPO agents use two neural network-based function approximators:

- **Actor** ($\pi(a|s;\theta)$): Represents a stochastic policy that outputs either a probability distribution over discrete actions or the parameters (mean and standard deviation) of a Gaussian distribution for continuous actions.
- **Critic** ($V(s;\phi)$): Estimates the expected long-term reward (state-value function), serving as a baseline for advantage estimation.

PPO Objective Function

The PPO algorithm optimizes a composite loss function that combines a clipped surrogate objective for the actor, a mean squared error loss for the critic, and an optional entropy

term to encourage exploration:

$$L(\theta, \phi) = \mathbb{E}\left[L^{\text{actor}}(\theta) + c_1 L^{\text{critic}}(\phi) - c_2 \mathcal{H}(\theta)\right] \tag{2.5}$$

where:

- $L^{\text{actor}}(\theta)$ is the clipped surrogate loss:

$$L^{\text{actor}}(\theta) = -\frac{1}{M}\sum_{i=1}^{M} \min\left(r_i(\theta)\hat{A}_i,\ \text{clip}(r_i(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_i\right)$$

- $r_i(\theta) = \frac{\pi_\theta(a_i|s_i)}{\pi_{\theta_{\text{old}}}(a_i|s_i)}$ is the likelihood ratio between new and old policies.
- $\hat{A}_i$ is the estimated and possibly normalized advantage.
- $L^{\text{critic}}(\phi) = \frac{1}{2M}\sum_{i=1}^{M}\left(G_i - V(s_i;\phi)\right)^2$ is the critic's value loss.
- $\mathcal{H}(\theta)$ is the entropy of the policy, promoting exploration.
- c_1 and c_2 are coefficients controlling the influence of the critic loss and entropy bonus, respectively.

The advantage $\hat{A}_i$ can be estimated using either finite-horizon methods or Generalized Advantage Estimation (GAE), depending on the agent's configuration.

Key Hyperparameters

PPO's performance and stability depend on several important hyperparameters:

- **Experience Horizon (N)**: The number of steps the agent interacts with the environment before each policy update. Set via the `ExperienceHorizon` option.
- **Learning Rate**: Controls how quickly the actor and critic networks update. Affects convergence speed and stability.
- **Clip Factor** (ϵ): Limits the range of the policy update via the clipped objective, helping prevent destabilizing policy shifts. Configured with the `ClipFactor` option.
- **Entropy Loss Weight** (c_2): Determines the strength of the entropy term in the objective function, encouraging stochasticity in policy output. Set with `EntropyLossWeight`.
- **Discount Factor** (γ): Specifies the weight given to future rewards. Typically close to 1.
- **GAE Factor** (λ): Used if Generalized Advantage Estimation is enabled. Balances bias and variance in advantage computation.
- **Number of Epochs (K)**: Number of optimization passes over the collected experiences. Set via `NumEpoch`.
- **Mini-Batch Size (M)**: Number of samples per gradient update step. Controlled by `MiniBatchSize`.

These hyperparameters are configured using an `rlPPOAgentOptions` object and can be tuned to optimize learning dynamics for different environments.

2.2.3 Reward Function Design

The reward function is designed with a combination of discrete and continuous reward terms. A discrete reward can be too sparse and difficult for the agent to spot, therefore the continuous rewards and penalties will help the agent to find its way around. In this project the reward function is composed of four components:

Lane Alignment Reward

Packages are rewarded for aligning with their target lanes. The alignment reward is proportional to both the vertical position and the lateral error from the target lane:

$$R_{\text{alignment}} = \begin{cases} -w_{\text{align}} \cdot y_{\text{pos}} \cdot (|x_{\text{pos}} - x_{\text{target}}| - \Delta_{\text{align}}), & \text{if } |x_{\text{pos}} - x_{\text{target}}| > \Delta_{\text{align}} \\ -w_{\text{align}} \cdot y_{\text{pos}} \cdot (|x_{\text{pos}} - x_{\text{target}}|), & \text{otherwise} \end{cases} \tag{2.6}$$

where:

- w_{align} is the alignment weight.
- y_{pos} is the vertical position of the package, giving higher reward for progress along the lane.
- Δ_{align} m defines the tolerance for acceptable lateral deviation.

Exit Bonus

An additional bonus is awarded when packages approach the exit region while remaining close to their target lane:

$$R_{\text{exit}} = \begin{cases} w_{\text{exit}} \cdot \left(1 - \dfrac{|x_{\text{pos}} - x_{\text{target}}|}{\theta_{\text{align}}}\right), & \text{if } |x_{\text{pos}} - x_{\text{target}}| < \theta_{\text{align}} \\ w_{\text{exit}} \cdot \left(-\dfrac{|x_{\text{pos}} - x_{\text{target}}|}{\theta_{\text{align}}}\right), & \text{otherwise} \end{cases} \tag{2.7}$$

where:

- w_{exit} is the exit bonus weight.
- θ_{align} is the alignment threshold near the exit.

The exit bonus is only activated when the package's vertical position exceeds $y_{\text{pos}} > 0.95 \times (5 \times d_{\text{AMS}})$, ensuring it is close to the end of the sorting system.

Additionally, after all the boxes exited, a +100 discrete reward is added for each correct exit to encourage overall trajectory.

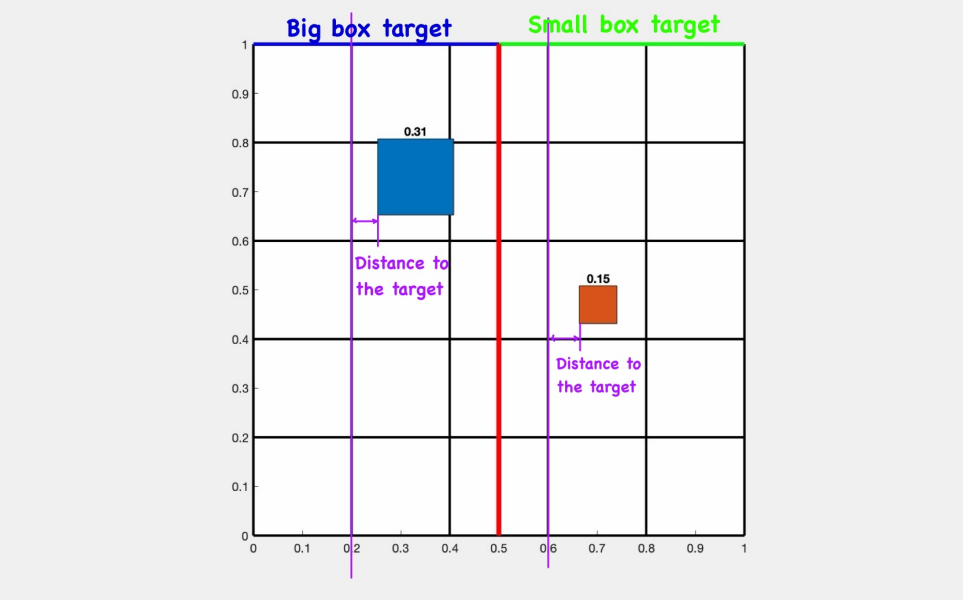


Figure 2.2: The target for boxes bigger than 0.225 and smaller has been indicated.

Gap Penalty

To discourage packages from clustering too tightly, a smooth penalty is applied when the vertical gap between successive packages is small. The penalty is defined as:

$$R_{\text{gap}} = \frac{G_{\text{w}}}{1 + \exp\left(k \cdot \left(|y_{\text{pos},1} - y_{\text{pos},2}| - d_{\text{safe}}\right)\right)} \tag{2.8}$$

where $y_{\text{pos},1}$ and $y_{\text{pos},2}$ are the vertical positions of any two packages, d_{safe} is the desired minimum safe gap (e.g., 0.2 m), G_{w} is the gap penalty weight, and k controls the steepness of the penalty curve. Smaller gaps result in higher penalties, but the function remains bounded and differentiable.

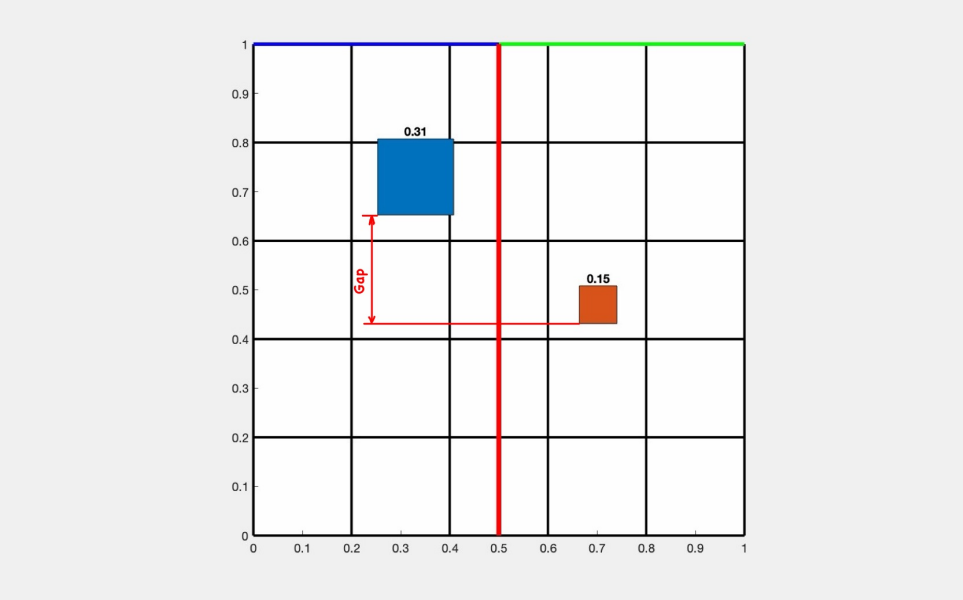


Figure 2.3: Gap between the two box has been calculated based on their lower left distances

Forward Motion Reward

To maintain throughput, the system rewards forward motion based on the average vertical velocity of all active packages:

$$R_{\text{forward}} = w_{\text{velocity}} \cdot \overline{v_y} \tag{2.9}$$

where $\overline{v_y}$ is the mean vertical velocity across active packages.

Total Reward

The total reward at each step is the sum of all components:

$$R_{\text{total}} = R_{\text{alignment}} + R_{\text{exit}} - R_{\text{gap}} + R_{\text{forward}} \tag{2.10}$$

2.2.4 Observation Space Design

In our reinforcement learning framework for the automated material sorting system, we designed an observation space that effectively captures the dynamic state of the environment. The observation space consists of a 60-dimensional feature vector, structured to provide the agent with comprehensive information about the boxes being sorted.

The observation vector $O \in \mathbb{R}^{60}$ is organized into 6 distinct feature categories, with each category containing information for up to 10 boxes:

$$O = [x_1, \ldots, x_{10}, y_1, \ldots, y_{10}, d_1, \ldots, d_{10}, v_{y_1}, \ldots, v_{y_{10}}, v_{x_1}, \ldots, v_{x_{10}}, \Delta x_1, \ldots, \Delta x_{10}] \tag{2.11}$$

where:

- $x_i \in [0, L]$: The x-coordinate of box i (indices 1-10)
- $y_i \in [0, H]$: The y-coordinate of box i (indices 11-20)
- $d_i \in [d_{min}, d_{max}]$: The diameter of box i (indices 21-30)
- v_{y_i}: The vertical velocity component of box i (indices 31-40)
- v_{x_i}: The horizontal velocity component of box i (indices 41-50)
- Δx_i: The distance from box i to its target position (indices 51-60)

For each box in the system, we track:

Position Information The (x_i, y_i) coordinates provide the current position of each box within the sorting system. This allows the agent to understand the spatial distribution of boxes across the conveyors.

Size Characteristics The diameter d_i is a critical feature that determines the target sorting lane for each box. Boxes with $d_i < \theta_{size}$ (where $\theta_{size} = 0.225$ is the size threshold) are directed to one target position ($x = 0.6$), while larger boxes are directed to another ($x = 0.25$).

Velocity Components The (v_{x_i}, v_{y_i}) values indicate the current movement of each box, allowing the agent to understand both the forward progression of boxes through the system and any lateral movement resulting from the actions of active modular sorters (AMS).

Target Alignment The distance to target $\Delta x_i = (x_{target} - x_i)$ provides direct feedback on how well a box is aligned with its intended sorting lane. This feature is particularly valuable as it encodes the primary sorting objective directly in the observation space.

Handling of Inactive Boxes

For efficiency, when boxes exit the system or aren't yet present (when $i > n_{boxes}$ where n_{boxes} is the current number of boxes in the system), their corresponding feature values are set to zero. This approach maintains a consistent observation vector dimension while clearly indicating the absence of certain boxes.

2.2.5 Learning Strategies

Curriculum Learning

To successfully teach the agent to perform intricate sorting activities, curriculum learning is utilized. Curriculum learning is a type of training protocol that borrows from learning effects in humans and animal behavior, beginning with simpler subtasks and gradually moving to more complicated ones as abilities improve. This method encourages quicker convergence and usually leads to improved generalization.

In the context of this material handling system, the curriculum is organized as below:

1. **Centered Sorting**

   First, the agent is trained to send all the packages to one point of exit, in the center. The environment is kept simple on purpose by keeping an equal distance between the packages so that no overlaps or near-collision occurrences take place. This training instructs the agent on the fundamentals of moving in the grid to deliver packages to a common location in order to maintain the system stable and in sync.

2. **Awareness of Space and Categorization** Once the agent has achieved steady competence in centered sorting, the instructional model proceeds to the subsequent level. In this, packages now have to go either left-target or right-target packages. The agent now needs to learn not just the actuation rules but also the ability of scanning package features and making instant decisions to sort each item to the corresponding side.

3. **Refinement**

   As training continues, more challenges in the form of tighter thresholds and higher speed has been introduced to the system. Progressively increasing complexity in this manner enables the agent to improve the learned policies to function in increasingly realistic environments and thus closely replicating actual working conditions.

This structured method enables the agent to learn primitive skills first before tackling the full complexities of the sorting job, thereby enhancing sample efficiency along with the overall performance of the policy.

Hyperparameter Scheduling

In addition to curriculum learning, strategic hyperparameter scheduling is also important to direct the learning process. In particular, the exploration-exploitation trade-off of the agent and policy refinement mechanisms are adaptively tuned throughout training.

Exploration-Exploitation Tradeoff In the initial stages of training, the agent is encouraged to explore a very large variety of actions in order to learn productive behaviors. This is achieved using the application of a relatively high entropy coefficient (0.05) that tends towards randomness in action selection.

Once the agent has begun doing reasonably well at easier tasks, exploration is eliminated (0.001) for favoring learned behavior exploitation.

Policy Refinement via Clip Factor Adjustment In PPO, the clip factor controls how much the policy is allowed to change between updates. Initially, a higher clip factor (0.1) is used to allow the agent to make substantial policy improvements, which is beneficial during the exploratory phases of curriculum learning.

As training progresses and the agent begins handling more complex tasks (e.g., distinguishing and sorting packages to the left and right exits), the clip factor is gradually reduced (0.01). This prevents large, destabilizing updates and encourages fine-tuned, incremental improvements to the policy.

2.3 Results

2.3.1 Stage 1: Centering the Boxes

In the first stage of training, the objective was to align boxes to the center of the conveyor exit at $x_{\text{center}} = 0.4$ m. A simulation was conducted using 100 randomly generated boxes to evaluate performance. The x-coordinate deviation of each box from the target position was measured.

To accelerate training, the box generation rate was reduced from 10 to 2. This reduction shortened episode durations and is consistent with the system's operational behavior, where two boxes appear every 0.75 seconds. The trained model is expected to generalize to higher box densities.

Several training runs were conducted while tuning hyperparameters:

- Clip factor: 0.2 to 0.05
- Entropy weight: 10^{-3} to 10^{-4}
- Actor and critic learning rates: 10^{-3} to 10^{-6}

Figure 2.4 illustrates these results. The error remained negligible for all boxes, indicating successful training of the centering policy.

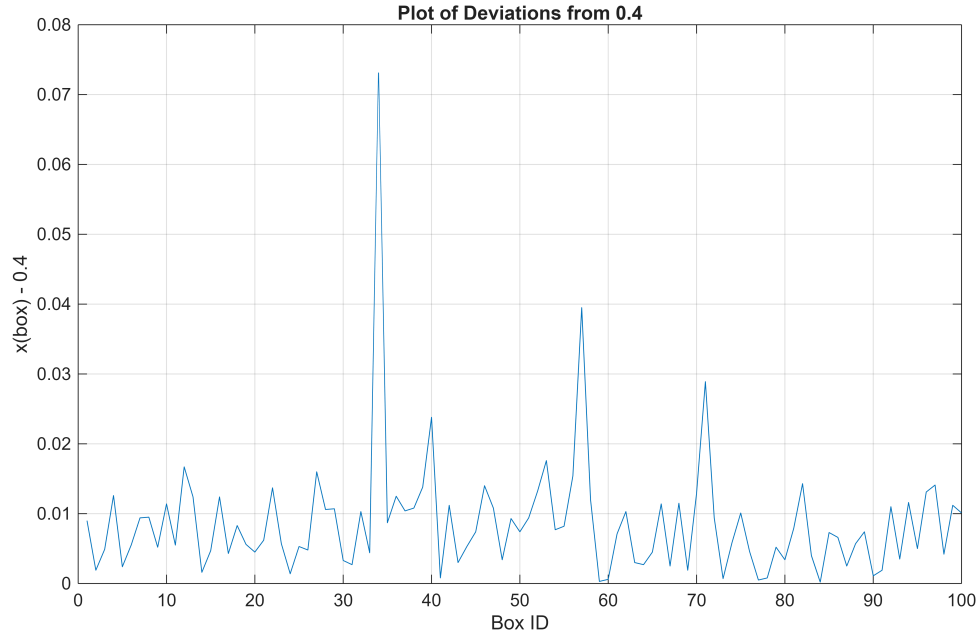


Figure 2.4: Deviation of box x-position from target $x = 0.4$ m at conveyor exit across 100 randomly simulated boxes.

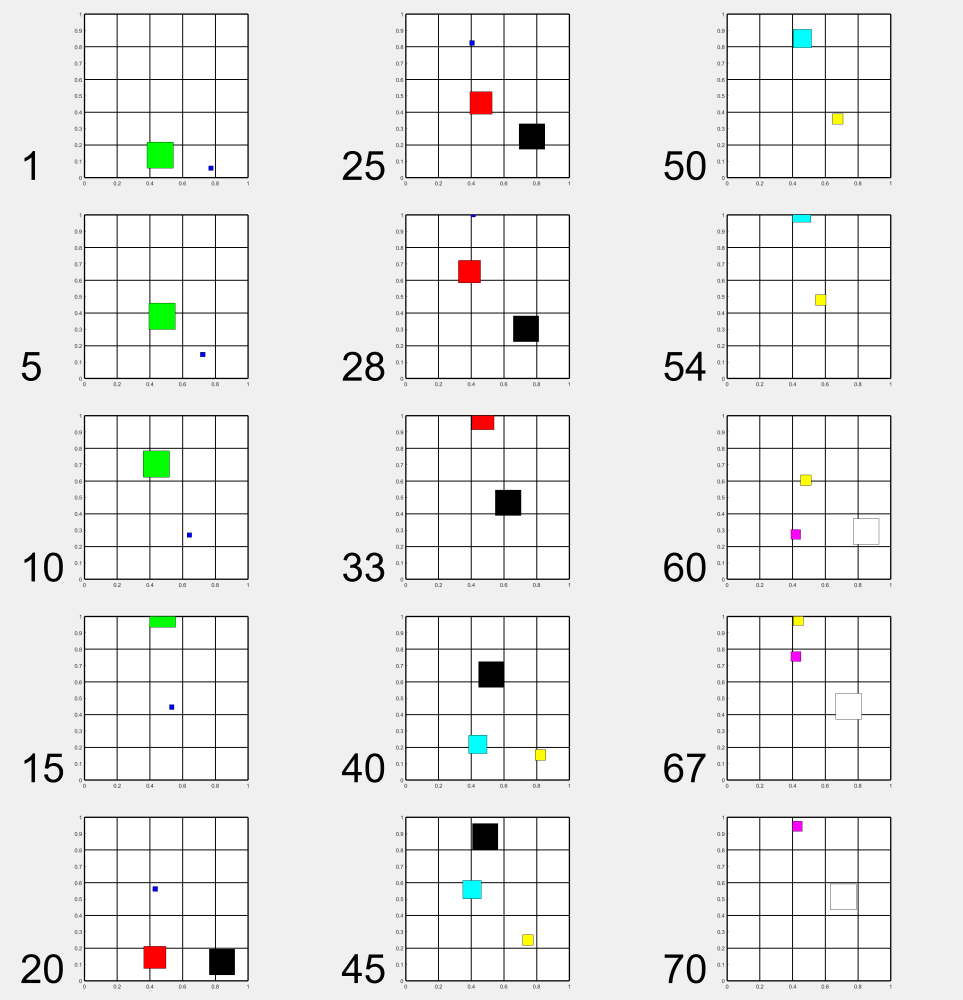


Figure 2.5: Illustration of box handling for a sample of 7 boxes, with exit moments highlighted in the frames on the left.

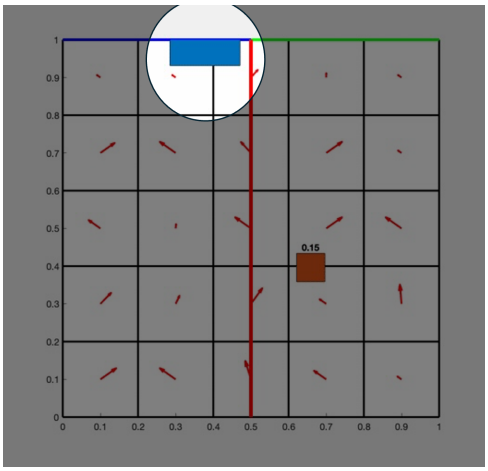
2.3.2 Stage 2: Sorting Based on Box Size

In the second stage, the model was further trained to sort boxes to the left or right side of the conveyor based on their size (Figure 2.6). The policy was fine-tuned using the previously trained centering model as a starting point. In this case the box generation was reduced to 4 since 10 boxes requires a long time training.

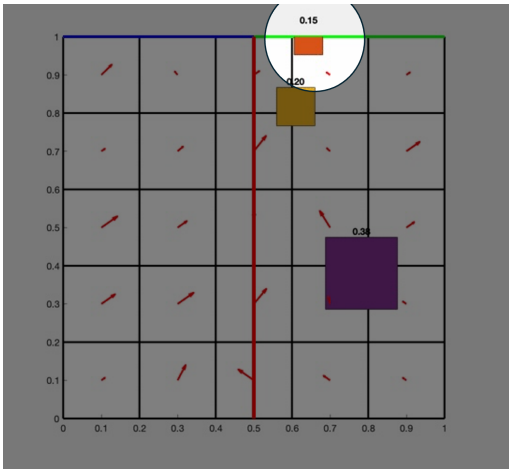
moreover hyperparameters has been changed alog the way as follows:

- Clip factor: 0.1
- Entropy weight: 10^{-2} to 10^{-3}
- Actor and critic learning rates: 10^{-3} to 10^{-5}

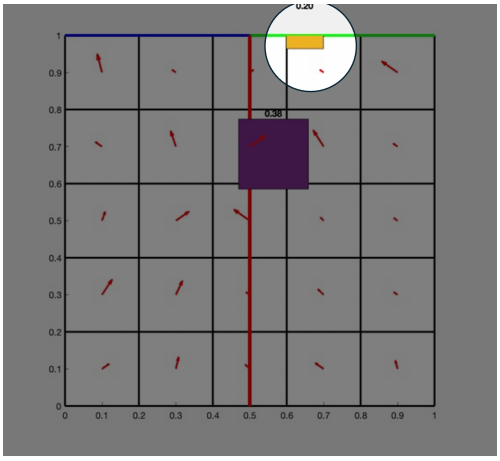
Figure 2.7 shows the final sorting performance, measured as the deviation of each box's x-position from its intended side at the conveyor exit across 100 randomly simulated boxes. The results suggest effective sorting behavior consistent with the desired size-based policy. Some occasional wrong exits can be observed which can be further refined by decreasing learning rate and exploration.



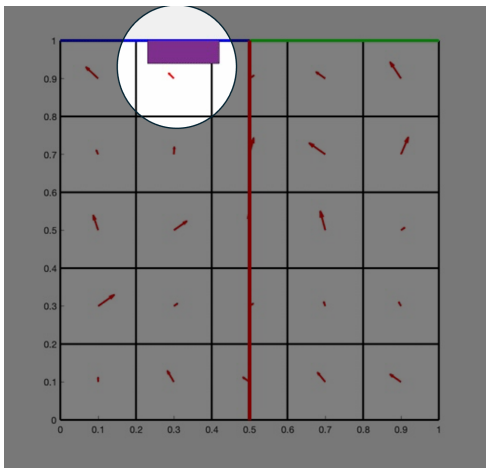
(a) Blue box, size 0.37, exiting left



(b) Orange box, size 0.15, exiting right



(c) Yellow box, size 0.2, exiting right



(d) Purple box, size 0.38, exiting left

Figure 2.6: Four consecutive boxes exiting from their target positions

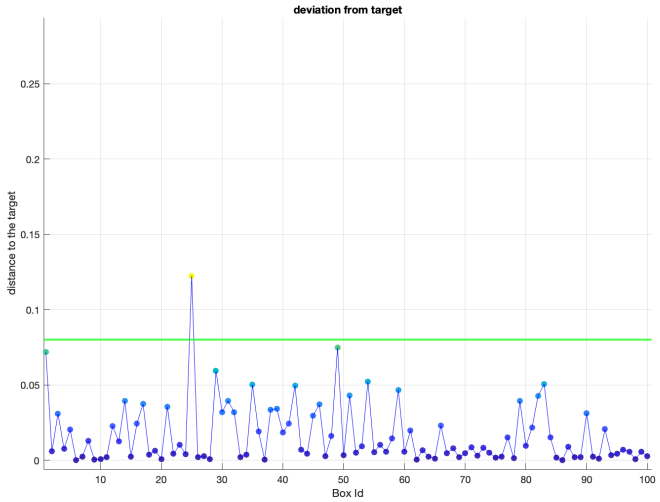


Figure 2.7: Box x-position deviations from target positions (left/right) at conveyor exit for 100 randomly simulated boxes. (Link to the video)

2.4 Conclusion

This chapter detailed the application of reinforcement learning (RL), specifically the Proximal Policy Optimization (PPO) algorithm, to control an automated material handling system (AMS) tasked with package sorting on a 5x5 grid of actuated modules. The primary objectives for the RL agent were to sort packages either to a central exit point or to different exits based on their size, while ensuring continuous flow and avoiding collisions. A curriculum learning strategy was implemented, starting with simpler sorting tasks and progressively increasing complexity, complemented by hyperparameter scheduling to manage the exploration-exploitation balance during training. The reward function was intricately designed to guide the agent, incorporating components for lane alignment, exit proximity, maintaining gaps between packages, and encouraging forward motion. The agent's observations consisted of a 60-dimensional feature vector capturing crucial information about up to 10 packages, including their positions, sizes, velocities, and distances to their respective targets. The results demonstrated the successful training of the PPO agent, achieving negligible error in the initial stage of centering boxes and effectively sorting boxes based on size in the subsequent, more complex stage. This work showcases the viability of modern RL techniques for addressing intricate industrial control challenges characterized by high-dimensional action spaces and dynamic environmental interactions.

Language Models in Robotics

3.1 Introduction

Large language models (LLM) encode a wealth of semantic knowledge about the world that could be extremely valuable for robots executing high-level, temporally extended instructions expressed in natural language. Therefore the question may arises that is it possible to use LLM to make robotic tasks more intelligent? This project presents a method to combining LLM-based decision making with physical robotic actions to perform a task that need a intelligent. Using chess as our domain, we demonstrate how low-level pick-and-place actions can be integrated with language models to enable a robotic arm to play chess following requested strategies, such as specific standard openings.

3.2 Method

The pipeline operates by interpreting human queries, generating candidate actions through an LLM, filtering these actions based on both semantic intent and game legality, and executing validated commands using a robotic arm in a simulated or real chess environment. The following subsections describe the components of this system, including the language model setup, chess-specific constraints, integration architecture, and the scoring and selection mechanisms used to ensure both strategic coherence and physical feasibility.

As illustrated in Figure 3.1, the process begins with user input, which may come from either the black or white player. Each input stream is processed independently through a LLM with their specific queries.

These candidate actions are subsequently passed through two distinct but complementary evaluation modules. On the black side, an "Affordance scoring" module ranks the generated actions based on contextual relevance, feasibility, and task appropriateness. On the white side, a "Command Parsing and Legality Check" module ensures that proposed actions conform to the rules of chess and can be translated into unambiguous board operations.

The filtered and validated commands converge at the "Action output text", which represents the finalized textual representation of the selected action. Special cases such as board resets are also handled at this stage.

Downstream, the action output branches into two key processing modules: "Cliport" and "Mapping". The "Cliport" module grounds the instruction into visual and physical

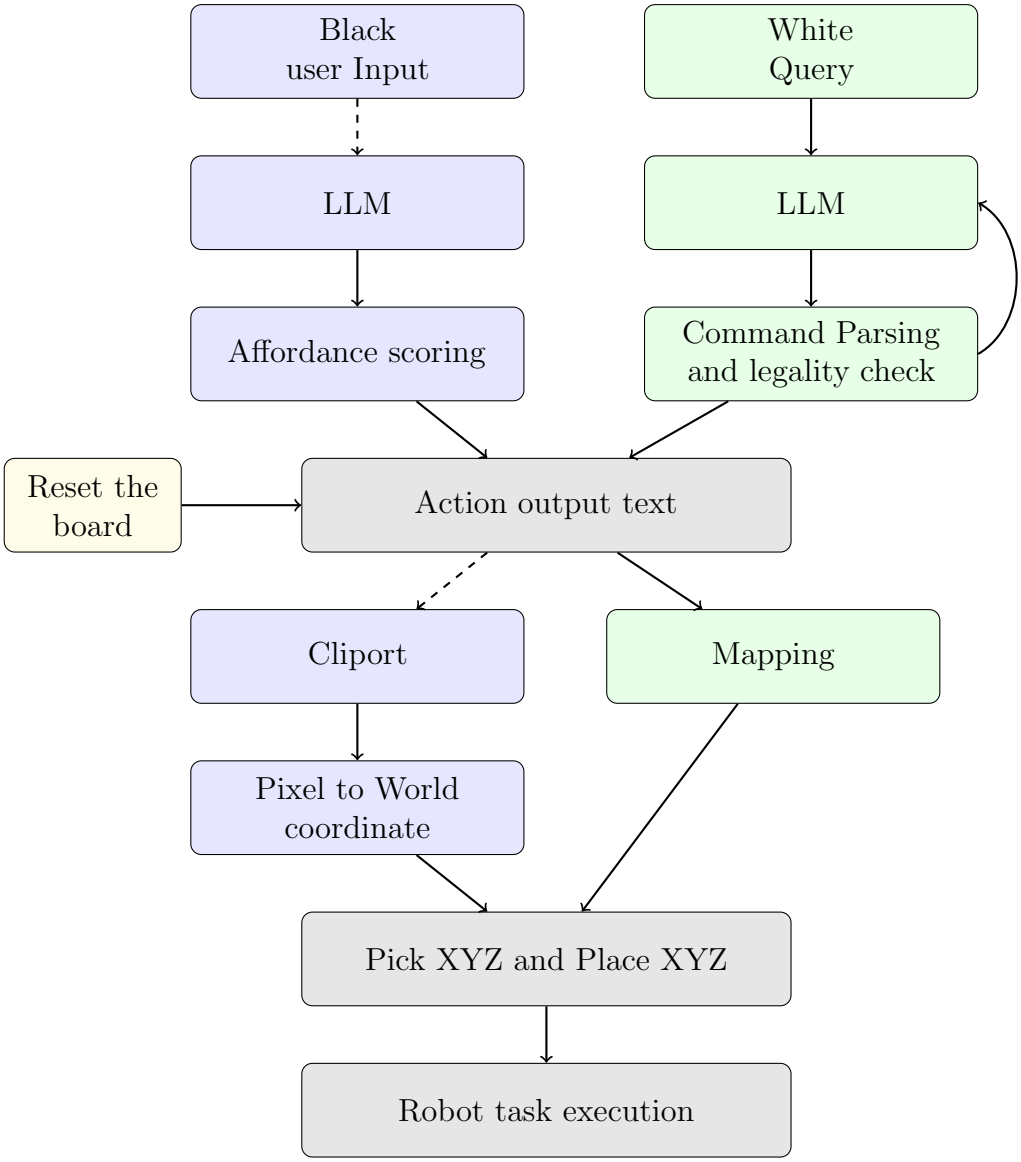


Figure 3.1: System pipeline for language-to-action translation

features, while the "Mapping" module converts the board-level move into robot-reachable 3D coordinates.

The outputs from both modules are fused into low-level motion instructions ("Pick XYZ and Place XYZ"), as shown in the sixth and seventh rows. These instructions are ultimately sent to the robotic controller for execution.

The feedback loop (dashed arrow from the legality check back to the LLM) allows for iterative refinement in case of ambiguous or invalid actions, enhancing the robustness of the system.

3.2.1 LLMs

LLMs estimate the probability distribution of a text token sequence $W = w_0, w_1, w_2, ..., w_n$ by modeling the conditional probability $p(W) = \prod_{j=0}^{n} p(w_j \mid w_{<j})$. This allows the model to predict each token based on previous context. The advancement of self-attention-based

neural architectures, particularly Transformers, has enabled increasingly powerful LLMs such as BERT, T5, GPT-3, and Llama, characterized by their scale (billions of parameters) and strong generalization capabilities across language tasks. In this project pretrained model "gpt-4o-mini" has been used through API provided by OpenAI. The API is invoked using the following command:

API command

```
response = openai.chat.completions.create(
    model=LLM_MODEL,
    messages=[
        {"role": "system", "content": call_law_prompt},
        {"role": "user", "content": query_prompt}
    ], temperature=temp, max_tokens=60)
```

This call constructs a chat-based context for the model, where

- the `system` message defines the role and constraints for the assistant—specifically, it allows to specify the way the model answers questions.
- The `user` message supplies the dynamic content. Equivalent to the queries made by the user in chatbots.
- The `temperature` parameter controls randomness in the output, with higher values allowing more exploration and lower values enforcing determinism—this is adjusted depending on whether prior failures exist.
- The `max_tokens` parameter limits the length of the response, ensuring the output is concise and in the expected format.

3.2.2 Chess

Chess is not merely a game of tactics and rules but also one of long-term strategy and structured decision-making. A critical phase of the game is the opening, which consists of the first 10–15 moves aimed at developing pieces efficiently, controlling the center, ensuring king safety (usually through castling), and preparing for the middlegame. Mastering openings allows players to avoid early weaknesses and steer the game into familiar or advantageous positions. For example, the Queen's Gambit, a classic and highly studied opening, begins with 1.d4 d5 2.c4, where White offers a pawn to gain central control. Variants such as the Queen's Gambit Accepted or Declined lead to very different board dynamics. In this project, we focus on guiding a robotic agent to execute such structured openings by interpreting natural language prompts and integrating language model strategies with physically legal moves on a chessboard. This ensures that gameplay not only follows the rules but also adheres to established strategic conventions from human chess knowledge.

3.2.3 LLM Integration in Chess Context

While LLMs demonstrate impressive generalization and reasoning capabilities, they are not specialized chess engines and can occasionally produce illegal or illogical moves. This occurs because LLMs generate text based on statistical patterns in language rather than

strict rule-based logic. For instance, a model might suggest moving a piece that no longer exists on the board, placing a piece on an invalid square, or proposing a move that violates the rules of chess (e.g., moving through another piece or ignoring check). Furthermore, without explicit constraints, the LLM might overlook the strategic context of the game, resulting in suboptimal or tactically flawed decisions. These limitations necessitate the use of auxiliary systems—such as move legality checks using `python-chess` or curated prompts—to filter or correct model outputs and maintain the validity and integrity of gameplay.

Move Generation System

To guide the LLM to make proper decisions and avoid illegal or illogical moves the `python-chess` library has been used. `python-chess` is a popular Python library for working with chess games. It provides a comprehensive set of tools for representing chess boards, validating moves, generating legal move lists, and handling chess game states in a structured format. The library supports various chess formats, including FEN (Forsyth-Edwards Notation), PGN (Portable Game Notation), and can be used for both standard chess and variant types.

Using python-chess, for each move we construct a carefully engineered prompt including:

- The full board state (in FEN format) and move history
- A structured list of legal moves grouped by piece
- Contextual strategy guidance (e.g., Queen's Gambit for white)
- Feedback on any previous failed move attempts
- Visual description of the board to aid spatial understanding

Response Handling and Validation

The LLM is instructed to respond in a strict format: `"Pick the [piece name] and place it on the [square name]"`, with temperature settings varied based on prior failures—starting higher for creativity and reducing upon retries for stability. LLM outputs undergo validation through:

- Regular expression pattern matching to ensure syntactic correctness
- Verification that piece names and target squares are valid identifiers
- Checking that the move conforms to chess rules in the current board state

Valid commands are forwarded to robot control, while malformed or illegal commands trigger a new prompt with failure feedback. If API errors occur or responses are invalid, the system gracefully falls back to mock mode, ensuring uninterrupted gameplay and robot control. Helper functions format the board into human-readable descriptions and grid-based visual representations, improving the LLM's contextual understanding and decision quality.

3.2.4 Probabilistic Action Selection with Affordance and LLM Scoring

An alternative and complementary approach to direct command generation involves a multi-stage scoring mechanism to select the most appropriate action from a predefined set of possibilities. This method leverages both the LLM's understanding of natural language intent and rigorous physical and rule-based validation.

In human-robot interaction, particularly when guided by Large Language Models (LLMs), it is crucial to ensure that proposed actions are not only linguistically plausible but also physically and contextually valid. The **affordance scoring** module in this system serves this exact purpose. It acts as a critical validation layer, evaluating whether a candidate action (e.g., moving a specific chess piece to a target square) is permissible according to the rules of chess and the current physical state of the game board as represented in the PyBullet simulation. Its primary goal is to filter out illegal or impossible moves, thereby grounding the LLM's suggestions in the reality of the game and the robot's capabilities.

Affordance scoring Pipeline

The `affordance_scoring` function (defined in `redo_llmaffordance_v2_capture.py`) takes a list of potential actions, the current mapping of simulated object names to their IDs, a `chess.Board` object representing the synchronized state of the physical board, and board layout details. For each candidate action string (e.g., `"robot.pick_and_place(white Queen, d5)"`), it performs the following steps:

1. **Action Parsing:** The action string is parsed to identify the specific piece intended for movement (`parsed_pick_piece_name`) and its target destination square (`parsed_target_place_sq`).

2. **Piece Color Determination:** The color of the piece to be moved is inferred from its name (e.g., "white Queen" implies `chess.WHITE`). This is essential for subsequent turn-based legality checks.

3. **Simulation State Verification:**

- The system verifies that the `parsed_pick_piece_name` exists within the current simulation environment (`sim_obj_name_to_id_map`).
- Crucially, it then determines the *actual current square* of this piece in the PyBullet simulation (`detected_current_sq_of_pick_piece`). This is achieved by comparing the piece's simulated XYZ coordinates with the known coordinates of all board squares (`place_targets_coord_map`). This step ensures the evaluation is based on the piece's true physical location, not just an assumed one.

4. **Chess Legality Check using `python-chess`:**

- A temporary copy of the `base_synced_chess_board` (which represents the current physical board state) is created: `board_for_this_option_check = base_synced_chess_board.copy()`.
- **Flexible Turn Handling:** A key aspect of this stage is setting the `turn` attribute of this temporary board to match the color of the piece in the *current candidate action*: `board_for_this_option_check.turn = piece_color_for_this_option`. This allows the system to assess the legality of a move *as if it were that piece's color's turn*,

regardless of the global game turn. This is vital because the LLM might generate options for either color, and each must be evaluated on its own merit.

- The system checks if a piece of the correct color actually exists on the `detected_current_sq_of_pick_piece` on this turn-adjusted board.
- A `chess.Move` object is constructed from the piece's actual current square to the target square. Pawn promotion to a Queen is handled by default if applicable.
- The legality of this move is then determined by checking if `move in board_for_this_option_check.legal_moves`.

5. **Score Assignment:**

- If all checks pass and the move is deemed legal by the `python-chess` library under the temporarily assumed turn, the action is assigned an affordance score of 1.
- If any check fails (e.g., piece not found, move is illegal according to chess rules), the action receives a score of 0.

Inputs and Outputs

- **Inputs:**
  - `options`: A list of string-formatted actions proposed by the LLM.
  - `sim_obj_name_to_id_map`: Dictionary mapping piece names to PyBullet object IDs.
  - `base_synced_chess_board`: A `chess.Board` object, kept in sync with the simulation via `sync_board_with_simulation`.
  - `place_targets_coord_map`: Dictionary mapping chess square names (e.g., "a1") to their [x,y,z] coordinates in the simulation.
  - `board_square_size`: The physical dimension of a chess square.
- **Output:**
  - `affordance_scores`: A dictionary where keys are the input action strings and values are their binary affordance scores (0 or 1).

3.3 LLM-Based Scoring and Action Selection

Beyond physical and rule-based validity, the system must also interpret the user's intent as expressed in natural language. This is achieved through LLM scoring, which is then combined with affordance scores to select the final action.

3.3.1 LLM Scoring of Potential Actions

The system employs a Large Language Model (LLM), specifically GPT-4o-mini via the `gpt3_scoring` function, to assess the alignment of potential actions with the user's natural language query.

1. **Prompting Strategy:** The LLM is provided with the user's query (e.g., "I want to move the white Pawn from d2 to d4") and a comprehensive list of all possible "pick and place" actions, framed as system instructions. The LLM is prompted to indicate the "NEXT STEP IS:".
2. **Log Probability (Logprobs) Extraction:** The core of this scoring method relies on extracting log probabilities. When the LLM generates a response, it also provides the log probability for each token it produces, as well as for its top alternative token choices at each step ('top logprobs'). A temperature of 0 is used for deterministic output.
3. **Option Scoring:** Each predefined candidate action (e.g., `"robot.pick_and_place(white Pawn 4, d4)"`) is tokenized. The system then calculates a score for the option by summing the log probabilities of its constituent tokens, based on how likely the LLM considered that sequence of tokens as a direct continuation of the query. If a token in the option was not highly considered by the LLM for that position, a penalty is applied.
4. **Normalization:** The raw logprob scores, which are typically negative, are normalized to a 0-1 range using the `normalize_scores` function. This facilitates easier interpretation and combination with other scoring mechanisms.

3.3.2 Combined Scoring and Final Action Selection

The final decision for action execution integrates both the LLM's understanding of the user's intent and the physical/rule-based validity of the action:

1. **Score Combination:** The normalized LLM score (0-1 scale) for each potential action is multiplied by its binary affordance score (0 or 1).

   $$\text{Combined Score} = \text{Normalized LLM Score} \times \text{Affordance Score}$$

   This ensures that actions deemed illegal by the affordance check (score 0) receive a combined score of 0, effectively filtering them out, regardless of how highly the LLM might have favored them.
2. **Action Selection:** The action with the highest positive combined score is chosen for execution. If multiple actions share the highest score, one is selected.
3. **Termination Condition:** If the selected task is the predefined `termination_string` (e.g., `"done()"`), the interaction sequence concludes.
4. **Fallback:** In scenarios where no action yields a positive combined score (e.g., all plausible LLM suggestions are illegal, or the LLM misunderstands profoundly), a fallback mechanism can be implemented, such as selecting the highest LLM-scored option that still has a positive raw affordance, or prompting the user for clarification.

3.3.3 Illustrative Example: A Sequence of Moves

To demonstrate the interplay of these scoring mechanisms, consider the following sequence of chess moves.

Move 1: White Pawn to d4

The user issues a query such as: **"I want to move the white pawn from d2 to d4. The robot command should be:"** (Assuming 'white Pawn 4' is on d2 initially). The system processes this, and the scoring is visualized in Figure 3.2 only for first 15 ranks.

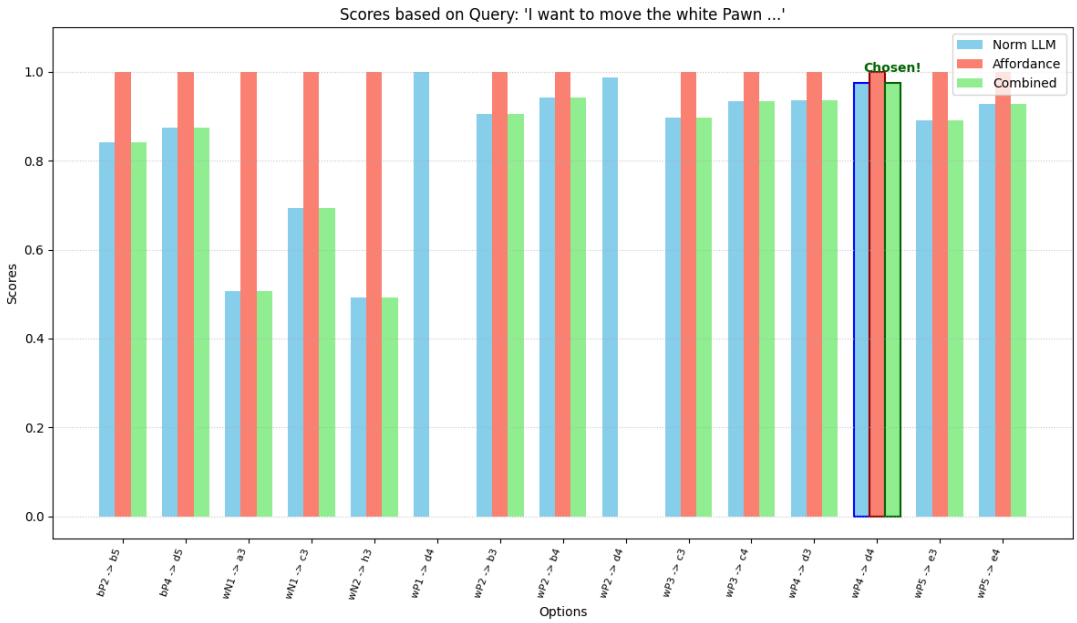


Figure 3.2: Scoring for query "Move white Pawn ... to d4". The 'Norm LLM' bars show the LLM's preference, 'Affordance' bars (1 for legal, 0 for illegal) filter options, and 'Combined' leads to the selection of 'wP4 to d4'.

As seen in Figure 3.2, the LLM correctly identifies moves involving a white pawn to d4 as highly relevant. The affordance scores ensure that only legal moves for white Pawn 4 (or whichever pawn is on d2) are considered. The combined score for wP4 to d4 is highest, and this action is selected.

Move 2: Black Pawn to e5

Following White's move, the user might query: **"Move the black pawn from e7 to e5. Robot command:"** (Assuming 'black Pawn 5' is on e7). The scoring for this step is shown in Figure 3.3.

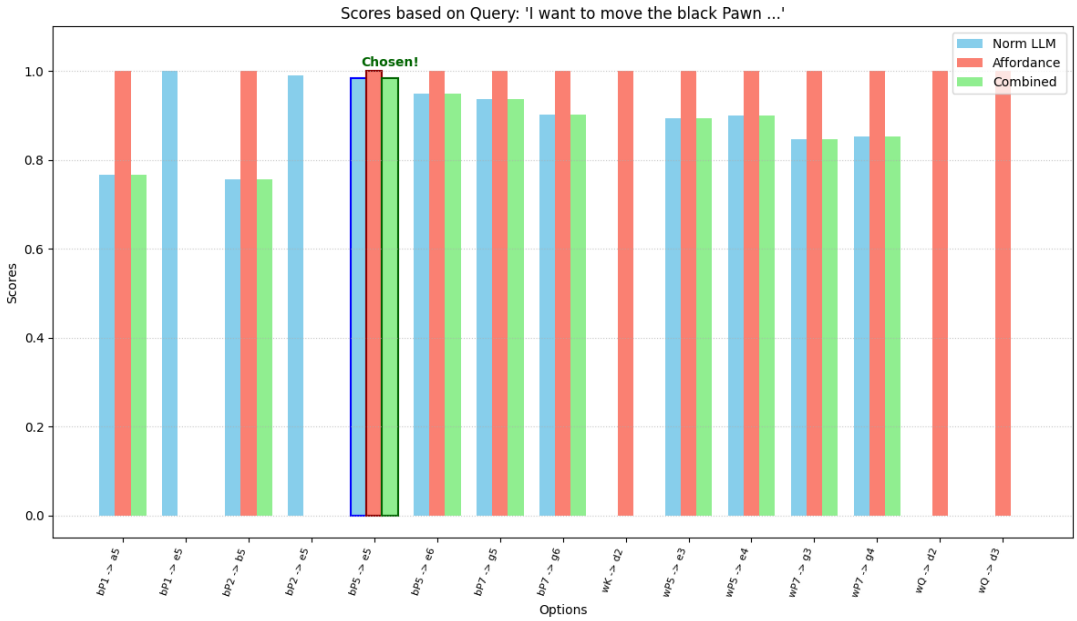


Figure 3.3: Scoring for query "Move black Pawn ... to e5". The system correctly identifies and selects 'bP5 to e5' by combining LLM intent with affordance validation.

Figure 3.3 illustrates a similar process. The LLM preferences align with moving a black pawn to e5. The affordance scoring, now considering Black's turn (due to the flexible turn handling in affordance scoring when evaluating black piece moves), validates legal options. The action `bP5 to e5` is chosen.

Move 3: Capture (e.g., White Pawn d4 takes e5)

Now, assume White intends to capture the black pawn on e5 with the white pawn on d4. The query could be: **"White pawn on d4 captures the piece on e5. Robot command:"**. The scoring is depicted in Figure 3.4.

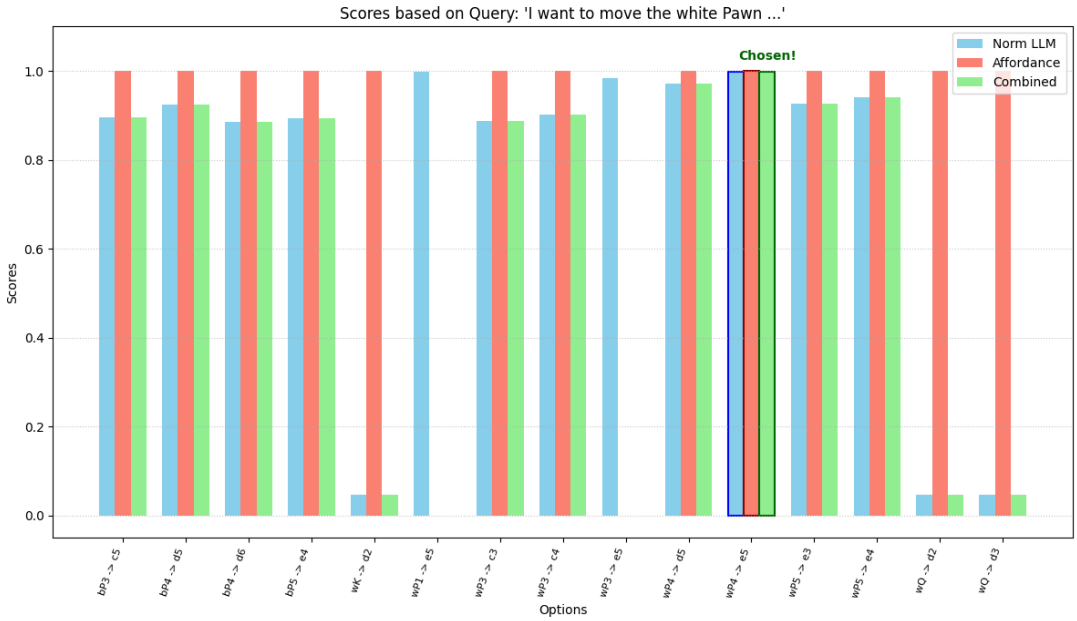


Figure 3.4: Scoring for capture query "White pawn d4 captures e5". Note how the chosen option 'wP4 to e5' reflects the capture. The affordance system would validate this as a legal capture for White.

In Figure 3.4, the LLM is tasked with understanding a capture. The option `wP4 to e5` (representing white Pawn 4 moving from d4 to e5, thereby capturing the piece on e5) receives a high LLM score. The affordance scoring confirms this is a legal capture for White. Consequently, this action is selected by the combined scoring mechanism. This also demonstrates the robot's ability to handle occupied target squares, where the `step` function would first remove the captured black pawn before placing the white pawn.

This sequence demonstrates the system's capacity to interpret varied user instructions, validate them against game rules and physical state, and select appropriate robotic actions.

3.3.4 Impact on System Performance

The affordance scoring mechanism significantly enhances the reliability and intelligence of the robotic chess-playing system.

- **Grounding LLM Outputs:** It prevents the robot from attempting actions that are physically impossible or violate chess rules, even if an LLM suggests them.
- **Error Reduction:** By filtering invalid moves, it reduces the likelihood of the robot entering an error state or performing nonsensical actions.
- **Improved Robustness:** The system becomes more robust to potentially imprecise or creatively (but incorrectly) formulated LLM suggestions.
- **Focused Decision-Making:** The final action selection process (which combines LLM scores with these affordance scores) benefits from a pre-filtered set of valid options, leading to more coherent and game-appropriate robotic behavior.

The "flexible turn handling" is particularly noteworthy, as it allows for a comprehensive evaluation of all LLM-generated options, irrespective of whose turn it is in the overarching

game sequence, making the system adaptable to varied LLM outputs.

3.3.5 LLM and Robotics arm integration

To bridge the gap between natural language (High level action) and robotic execution (Low level action), the structured text command needs to being parsed or in other words, translated. Then the translation process includes:

1. Splits the command at the keyword `"and"` to extract pick and place targets
2. Identifies known objects or locations from pre-registered dictionaries
3. Retrieves 3D coordinates for identified entities
4. Generates an action dictionary with `pick` and `place` keys and associated position vectors

This design restricts vocabulary to valid, environment-specific terms, ensuring actions correspond to tangible entities in the workspace. The final output allows the robot control system to focus on motion planning and execution.

3.3.6 CLIPORT

Another effective approach for command to action translation is CLIPORT (CLIP-Transporter). This model is designed to interpret natural language instructions alongside visual input from a camera to perform robotic manipulation tasks. In the context of this project, the system takes a text command—such as "Pick the black King and place on the c2."—along with a top-down camera image of the chessboard environment to determine appropriate actions.

The core process involves several sequential steps:

1. **Input Processing**: A top-down camera image is captured and resized to 224×224 pixels. The input text command is tokenized and encoded into feature vectors using a pre-trained CLIP (Contrastive Language-Image Pre-training) model.
2. **Affordance Prediction**: The encoded visual and textual features are then passed to Cliport's `TransporterNets` model, which consists of two primary components: a pick network and a place network. The model outputs two heatmaps—one for identifying the best pick location and another for the optimal place location.
3. **Action Execution**:
    - The pixel with the highest activation in the pick heatmap (`pick_map`) is selected as the pick location (`pick_yx`).
    - Similarly, the pixel with the highest activation in the place heatmap (`place_map`) is selected as the place location (`place_yx`).
    - These 2D pixel coordinates are then transformed into 3D world coordinates (`pick_xyz`, `place_xyz`) using the camera's intrinsic and extrinsic parameters.
    - Finally, the robot performs the pick-and-place action based on the calculated 3D coordinates.

To improve spatial understanding, the visual input is augmented with coordinate channels (`coords`), which are concatenated to the image data before being passed to the network. This enhancement provides the model with explicit positional context alongside the visual and textual cues.

Importing Pre-trained Models

The successful operation of Cliport relies heavily on pre-trained models. Two key pre-trained components are utilized:

A pre-trained CLIP model (specifically "ViT-B/32") is loaded to encode textual instructions into meaningful feature vectors. These text features provide the semantic understanding of the command.

The `TransporterNets` architecture, which forms the core of the manipulation logic, can be initialized with pre-trained weights.

Data Generation for Moving Chess Pieces

To train Cliport for the specific task of moving chess pieces, a dataset of demonstrations was generated. This process is encapsulated within the `Generate_data = True` block:

The simulation environment is reset with a configuration of chess pieces. A random selection of pickable pieces (`PICK_TARGETS`) and placeable board locations (`PLACE_TARGETS`) is made for each data generation episode.

A `ScriptedPolicy` is used to generate ground truth actions. For each pickable piece, a prompt is formulated, e.g., `'Pick the {{pick_items[i]}} and place it on the {{place_items[i]}}.'`. The policy then determines the 3D coordinates for the pick and place actions based on this prompt and the current state of the environment.

For each action:

- A top-down camera image (`env.get_camera_image_top()`) is captured and resized to 224×224 pixels.
- The 3D pick and place coordinates from the scripted policy are converted to 2D pixel coordinates (`pick_yx`, `place_yx`) using `env.world_to_pixel(act['pick'], img_height, img_height)` and `env.world_to_pixel(act['place'], img_height, img_height)` respectively.
- Each sample, consisting of the textual prompt, the camera image, and the corresponding pick and place pixel coordinates, is stored.

All generated samples are accumulated and saved into a single pickle file (`dataset_samples/combined_dataset.pkl`) for later use in training. The target dataset size was 8,000 samples.

Training for Chess Piece Manipulation

The `TransporterNets` model is trained using the generated dataset. The `TransporterNets` class defines the neural network, which includes ResNet blocks for feature extraction from images and text embeddings, and attention mechanisms to correlate pick and place actions. Text features are fused mid-way through the ResNet architecture.

During training, batches of data are constructed. Each batch item includes:

- The image (normalized and concatenated with coordinate channels).
- The pre-computed CLIP text features for the command.
- The ground truth pick pixel coordinates (`pick_yx`).
- One-hot encoded maps for pick (`pick_onehot`) and place (`place_onehot`) locations. These maps are the same dimensions as the input image, with a 1 at the target pixel and 0 elsewhere.

To improve robustness, data augmentation is applied by randomly rolling (shifting with wrap-around) the images and their corresponding one-hot encoded label maps. The pick coordinates are adjusted accordingly.

The model is trained using an InfoNCE (Noise Contrastive Estimation) loss, implemented as a softmax cross-entropy loss. This is applied separately to the pick and place predictions. The total loss is the sum of the pick loss and the place loss.

Model Loss Function

```
pick_loss = jnp.mean(optax.softmax_cross_entropy(
    logits=pick_logits,
    labels=batch['pick_onehot']))

place_loss = jnp.mean(optax.softmax_cross_entropy(
    logits=place_logits,
    labels=batch['place_onehot']))

loss = pick_loss + place_loss
```

The Adam optimizer (`optax.adam`) is used to update the model parameters based on the computed gradients. Model checkpoints (parameters and optimizer state) are saved periodically during training, allowing for resumption and model evaluation.

Reason for Limited Chess Piece Set

When working with vision-based robotic systems, the visual distinguishability of objects is crucial. Chess pieces, particularly those of the same color, can present significant visual similarities. For instance, rooks, knights, and bishops, or even multiple pawns, can be challenging for a vision system to differentiate reliably if they have similar shapes, textures, or if lighting conditions are not ideal.

In the provided codebase, the initial configuration of `PICK_TARGETS` includes many standard chess pieces, but only one pice of each king has been kept.

However, a more advanced vision system or more extensive training could potentially handle a greater accuracy. Additionally, in future work, it would be possible to add an indicator to make the similar pieces visually different.

3.4 Results

In this section, the results of LLM usage in robotic arm applications are presented. Firstly, the environment setup, which consists of the objects that need to be manipulated (chess pieces) and the specified targets (chess board), is described. Then, several showcases of the LLM's performance in playing chess follow. Moreover, an experiment with CLIPORT to manipulate chess pieces is shown.

3.4.1 Environment setup

The simulation environment is created using PyBullet, a physics engine that provides precise collision detection and dynamics simulation. The robotic system consists of two main components:

- **UR5e Arm:** The Universal Robots UR5e arm is a 6-degree-of-freedom robotic arm
- **Robotiq 2F85 Gripper:** attached to the end-effector of the UR5e arm

Figure 3.5 shows the environment ready to manipulate. The initial positions of all pieces follow standard chess setup, with white pieces on ranks 1-2 and black pieces on ranks 7-8. Each time a reset function can be called to place each piece in the corresponding place. Two camera views are configured for observation and visual feedback:

Perspective Side-View Camera:

- Position: $(0, -1.05, 0.6)$ meters
- Orientation: $(\frac{\pi}{5} + \frac{\pi}{48}, \pi, \pi)$ radians
- Resolution: 720×720 pixels
- Focal length: 360 pixels
- Z-range: $(0.01, 10.0)$ meters

Orthographic Top-View Camera:

- Position: $(0.05, -0.45, 4.0)$ meters
- Orientation: $(0, \pi, -\frac{\pi}{2})$ radians
- Resolution: 720×720 pixels
- Focal length: 3500 pixels
- Z-range: $(0.01, 1.0)$ meters

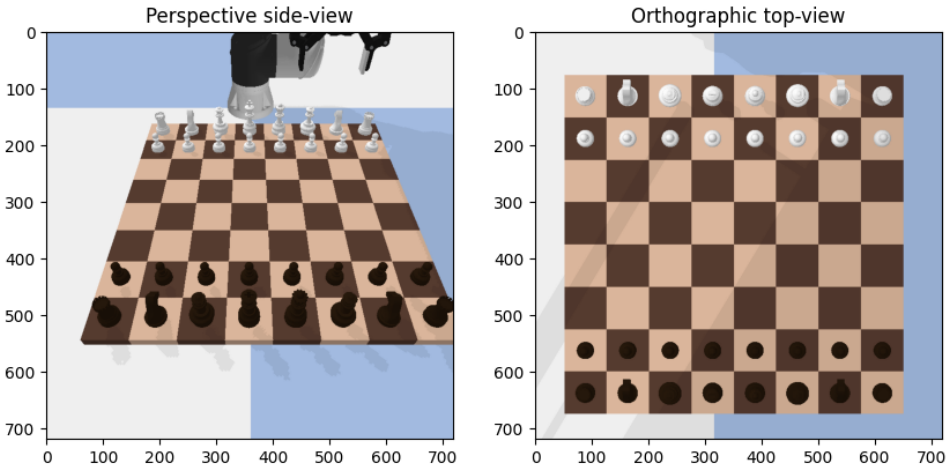


Figure 3.5: Environment configuration

The robotic arm performs pick-and-place operations using the following motion sequence:

1. Hover over the source position at height $z = z_{source} + 0.2$ meters
2. Move down to the pick position with $z = 0.02$ meters
3. Activate gripper to grasp the piece
4. Return to hover height
5. Move to target position at height $z = 0.12$ meters
6. Lower until contact is detected or minimum height is reached
7. Release the gripper
8. Return to hover height
9. Return to home position $(0, -0.5, 0.2)$ meters

Figure 3.6 shows an example of piece manipulation. In this case the input is provided as `'Pick the white King and place it on the e4'`. Then the text is parsed to extract the *object_id* and *place_id* to get the XYZ map. A few additional places has been defined for the capture pieces of each side outside of the board.

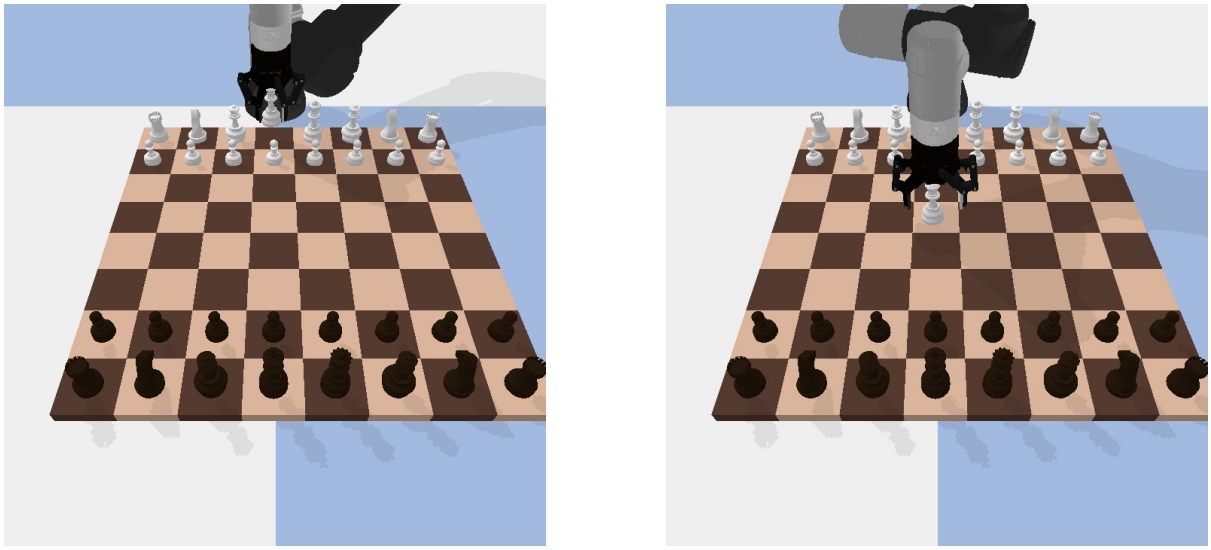


Figure 3.6: Pick and Place action example

3.4.2 LLM

The core of the automated chess gameplay in this project relies on a LLM to determine and suggest moves for one of the players (typically white). The interaction with the LLM is orchestrated to ensure that it proposes moves that are not only strategically sound but also adhere to a specific command format required for robotic execution.

The primary function responsible for interfacing with the LLM is `get_chess_move`. When it's the LLM's turn to play, this function constructs a detailed prompt to guide the model. The system prompt, stored in `call_law_prompt`, instructs the LLM to act as an expert chess engine controlling a pick-and-place robot. It strictly defines the output format: `"Pick the [piece name] and place it on the [square name]"`. This prompt also enumerates the exact, valid piece names the LLM must use, including specific numbering for pieces like pawns, rooks, knights, and bishops (e.g., "white Pawn 3").

The user prompt, `query_prompt`, provides the LLM with comprehensive contextual information about the current game state. This includes:

- The PGN (Portable Game Notation) of the moves played so far, extracted from the `chess_board.move_stack`.
- Feedback on any previously attempted illegal moves by the LLM, including the failed command and the reason for its illegality.
- The current player whose turn it is to move.
- The current positions of all pieces on the board, taken from the `piece_positions` dictionary.
- Strategic guidance, such as a preference for a specific opening like the Italian Game (Giuoco Piano).

System Prompt

```
call_law_prompt = f"""You are an expert chess engine controlling a
pick-and-place robot in a chess game. Your task is to determine
a strong chess move for {player} and output the precise robot command.

OUTPUT FORMAT REQUIREMENTS:
Return ONLY this format with no additional text:
"Pick the [piece name] and place it on the [square name]".
Example: pick the black Bishop 1 and place it on the e8.

STRICT REQUIREMENTS:
1. You MUST select from THESE EXACT piece names only:
   {', '.join(sorted(p for p in AVAILABLE_PIECES if p.startswith(player)))}
2. For pawns, knights, bishops and rooks, ALWAYS include the specific
   number (e.g., "white Pawn 3", not just "white Pawn").

Do not explain your reasoning or add commentary | output ONLY the
required format.
"""
```

User Prompt

```
query_prompt = f"""Move History:
{moves_only}
{feedback_str}
It is {player}'s turn to move.
It's important for White to play the Italian Opening. White should
play the Giuoco Piano, which starts with e4, Ke5, and Bc4.
It's important to provide the response in the required format.
Current pieces are in {piece_positions}
"""
```

The `openai.chat.completions.create` method is then called with the `LLM_MODEL` (specified as "gpt-4o") to generate a move. A temperature parameter is used, set higher for initial attempts and lower for retries after failures, to balance creativity and determinism.

Once the LLM returns a move string, it is processed by `parse_pick_place_command`. This function uses a regular expression (`COMMAND_RE`) to validate if the LLM's output matches the required "Pick the X and place it on the Y" format and if the specified piece and square are valid according to `VALID_PIECES` and `VALID_SQUARES`.

If the command is successfully parsed, the `execute_move` function attempts to apply it. It first checks if the piece belongs to the current player. It then retrieves the piece's current location from `piece_positions`. Crucially, the legality of the move is verified using the `python-chess` library's `is_move_legal` function, which internally converts the move to UCI (Universal Chess Interface) format and checks it against the legal moves on the `chess.Board` object.

If the move is legal:

1. The `piece_positions` dictionary is updated with the new square for the moved

piece.

2. The move is executed on the `chess.Board` object using `chess_board.push_uci()`.
3. The system checks for captures by comparing the target square with opponent piece locations. If a capture occurs, the captured piece is marked as off-board in `piece_positions`.
4. Game-ending conditions like checkmate, stalemate, or insufficient material are also checked using `python-chess` methods (`is_checkmate()`,etc).

If the LLM's proposed move is illegal, `execute_move` returns `False` along with a reason for the failure (e.g., "Not a legal chess move," "Pawns can only move 1-2 squares," etc.). This feedback can then be incorporated into subsequent calls to `get_chess_move` to help the LLM refine its suggestions.

In scenarios where the OpenAI API call fails or if `USE_MOCK_API` is true (e.g., if no API key is found), the system falls back to `generate_mock_chess_move`. This function uses `get_all_legal_moves` (which leverages `python-chess`) to find all legal moves for the current player and selects one, prioritizing captures, to ensure the game can proceed without direct LLM intervention. The opponent's moves (human player) are taken via UCI input and converted into the same command format using the `robot_move` function when it's black's turn. Figure 3.7 to 3.9 show examples of chess game progression focusing on specific opening strategies. Figure 3.7 illustrates the LLM's execution of the Giuoco Piano opening for the white player, as per the system's programming, Figure 3.8 shows the LLM playing famous london system opening ,Figure 3.9 presents a game sequence involving the Nimzowitsch Defense.

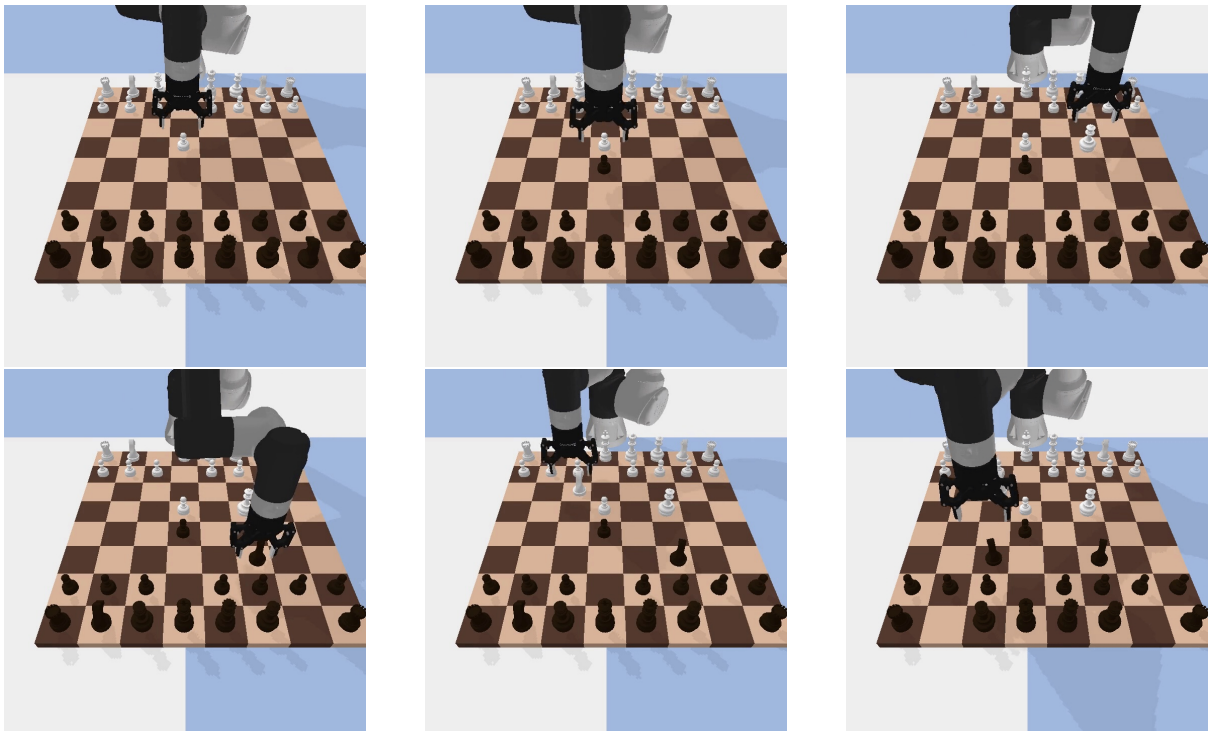


Figure 3.7: Giuoco Piano executed by LLM played as white and the user played as black, Link to the video

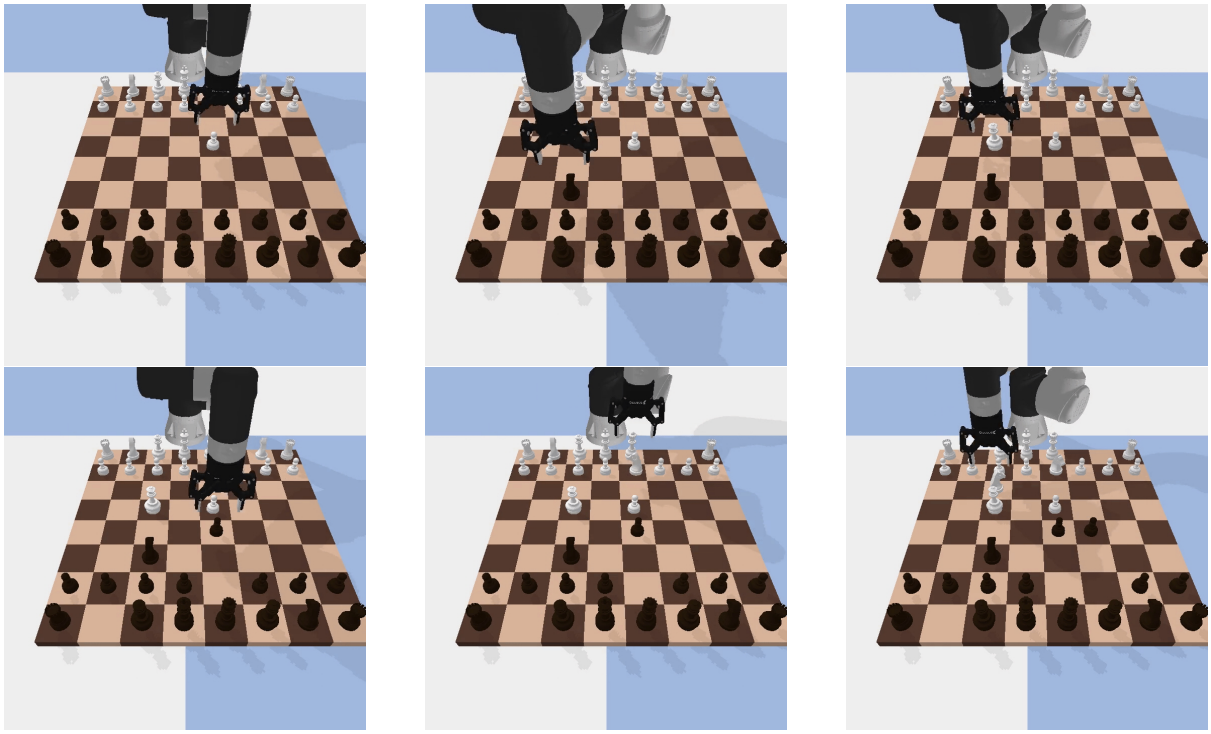


Figure 3.8: London system executed by LLM played as white and the user played as black, Link to the video

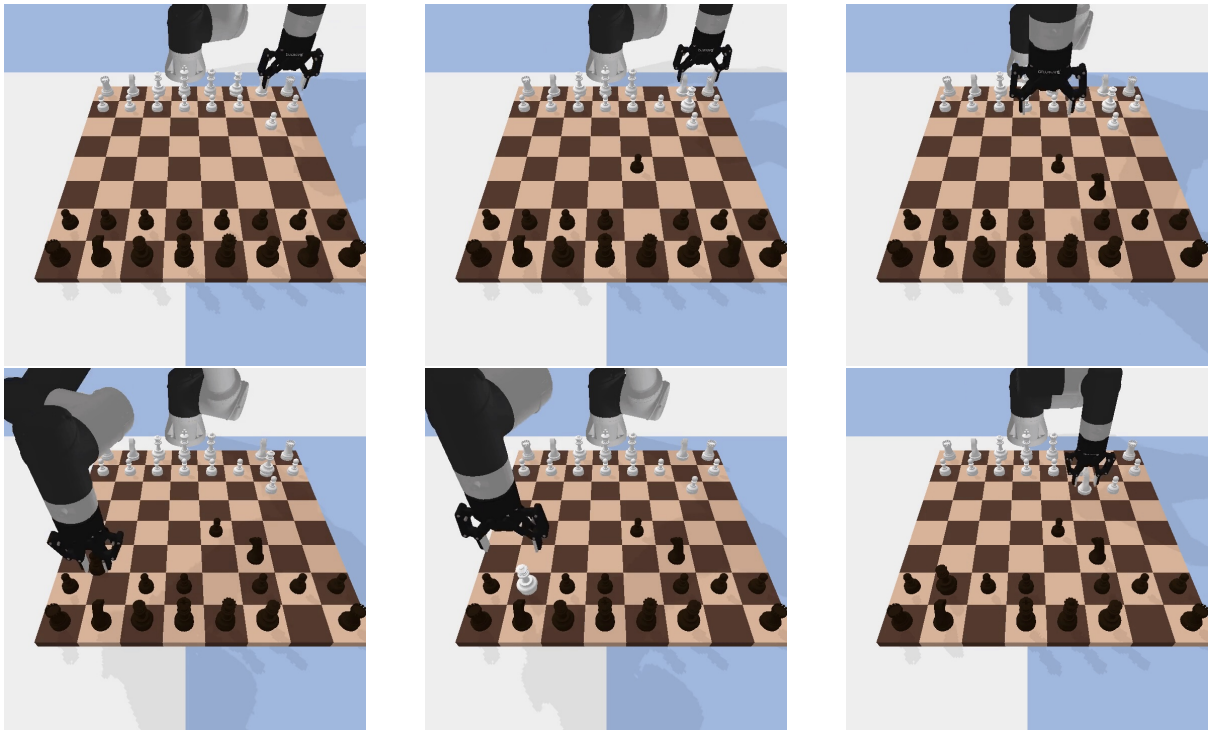


Figure 3.9: Nimzowitsch–Larsen Attack executed by LLM played as white and the user played as black, Link to the video

This visualization of opening strategies highlighted a key aspect of the LLM's performance: while the LLM demonstrated a commendable ability to adhere to the initial guided

sequences for openings like the Giuoco Piano, its performance in navigating the complexities beyond this initial phase was less consistent. It often struggled to maintain a strategically coherent game plan as play advanced into the middlegame and endgame.

A primary reason for this limitation appears to be the complexity of the game and the the LLM model not being trained to play chess.

3.4.3 CLIPORT

CLIPORT is designed to use orthogonal image and text input pairs to execute manipulation tasks. In the context of chess, this approach presents a unique challenge due to the visual similarity between many of the pieces. To mitigate this issue, only one piece from each pair of similar-looking pieces was initially selected. The model was then trained on a limited set of distinguishable pieces, as illustrated in Figure 3.10.

As described in Section 3.3.6, a dataset of 8000 samples was collected to train CLIPORT for a pick-and-place task. The objective was to position each chess piece in its correct location on the board.

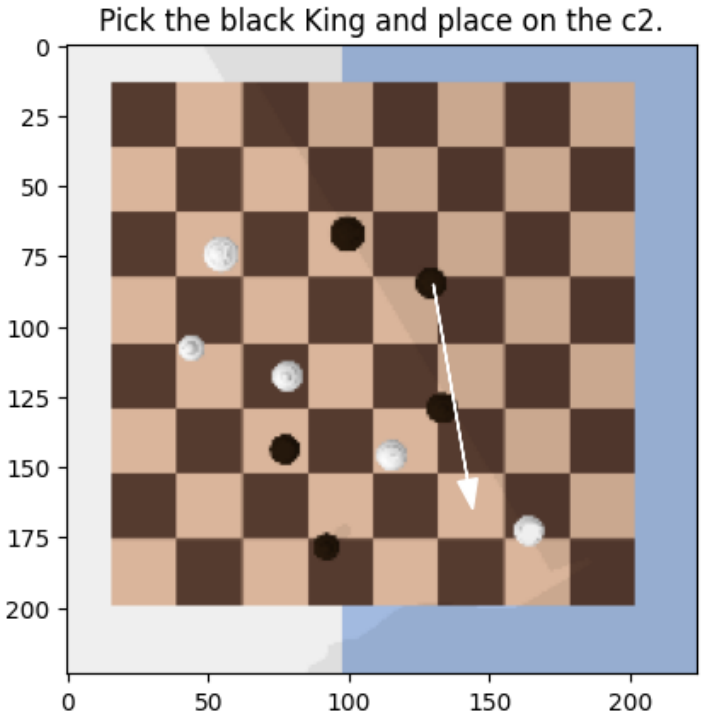


Figure 3.10: Training dataset used for CLIPORT

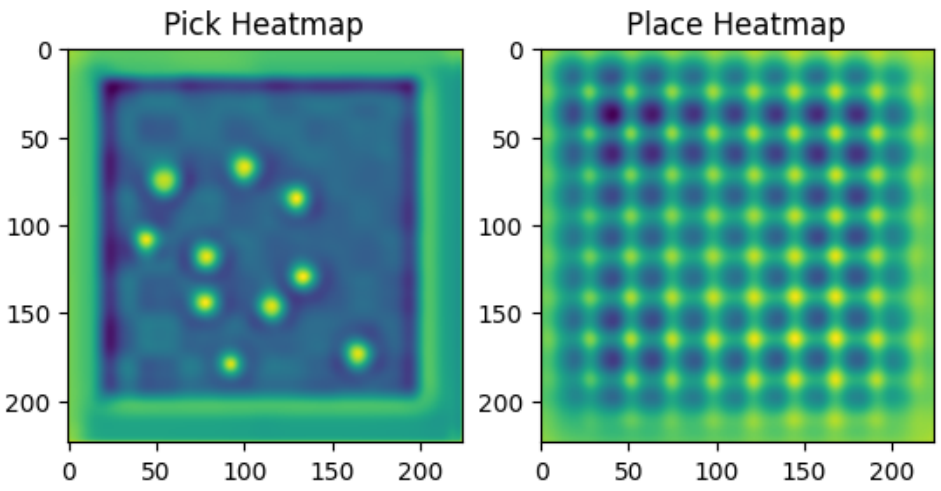


Figure 3.11: Height map used for the pick-and-place task

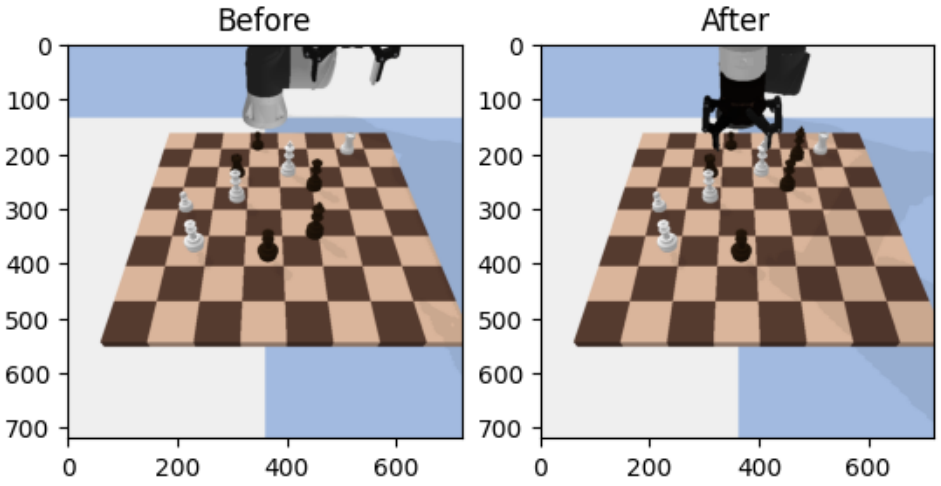


Figure 3.12: Task execution using the trained CLIPORT model

The task was successfully completed, as demonstrated in the following video: https://drive.google.com/file/d/1CZ9BUyGuzbasfpqvZGjSpCL7H5Prx6Kq/view?usp=share_link

3.5 Conclusion

This chapter explored the integration of Large Language Models (LLMs) with robotic systems to enable intelligent task execution, using the game of chess as a practical demonstration domain. The developed system utilized an LLM, specifically "gpt-4o-mini," to interpret natural language queries from a human player and to generate candidate chess moves. These LLM-generated moves were then validated for legality using the python-chess library and subsequently executed by a UR5e robotic arm within a PyBullet simulated environment. The methodology encompassed two main approaches for action selection: direct command generation where the LLM's output was parsed and validated, and a probabilistic action selection mechanism that combined LLM-based scoring of

potential actions with an "affordance scoring" module to ensure both semantic relevance and adherence to game rules and physical constraints. The results indicated that the LLM could successfully follow strategic instructions for specific chess openings, such as the Giuoco Piano, London System, and Nimzowitsch Defense. However, its performance tended to be less consistent as the game progressed into more complex middlegame and endgame scenarios, largely because the LLM is not a specialized chess engine. Additionally, the chapter presented experiments with CLIPORT, a model that uses visual input alongside textual commands, to perform pick-and-place operations for chess pieces. After being trained on a dataset of 8000 samples, CLIPORT successfully executed the task of positioning a limited set of visually distinct chess pieces on the board. Overall, the project demonstrated the promising potential of LLMs to enhance robotic intelligence and facilitate more natural human-robot interaction, while also highlighting current limitations when dealing with highly complex, specialized domains like chess.